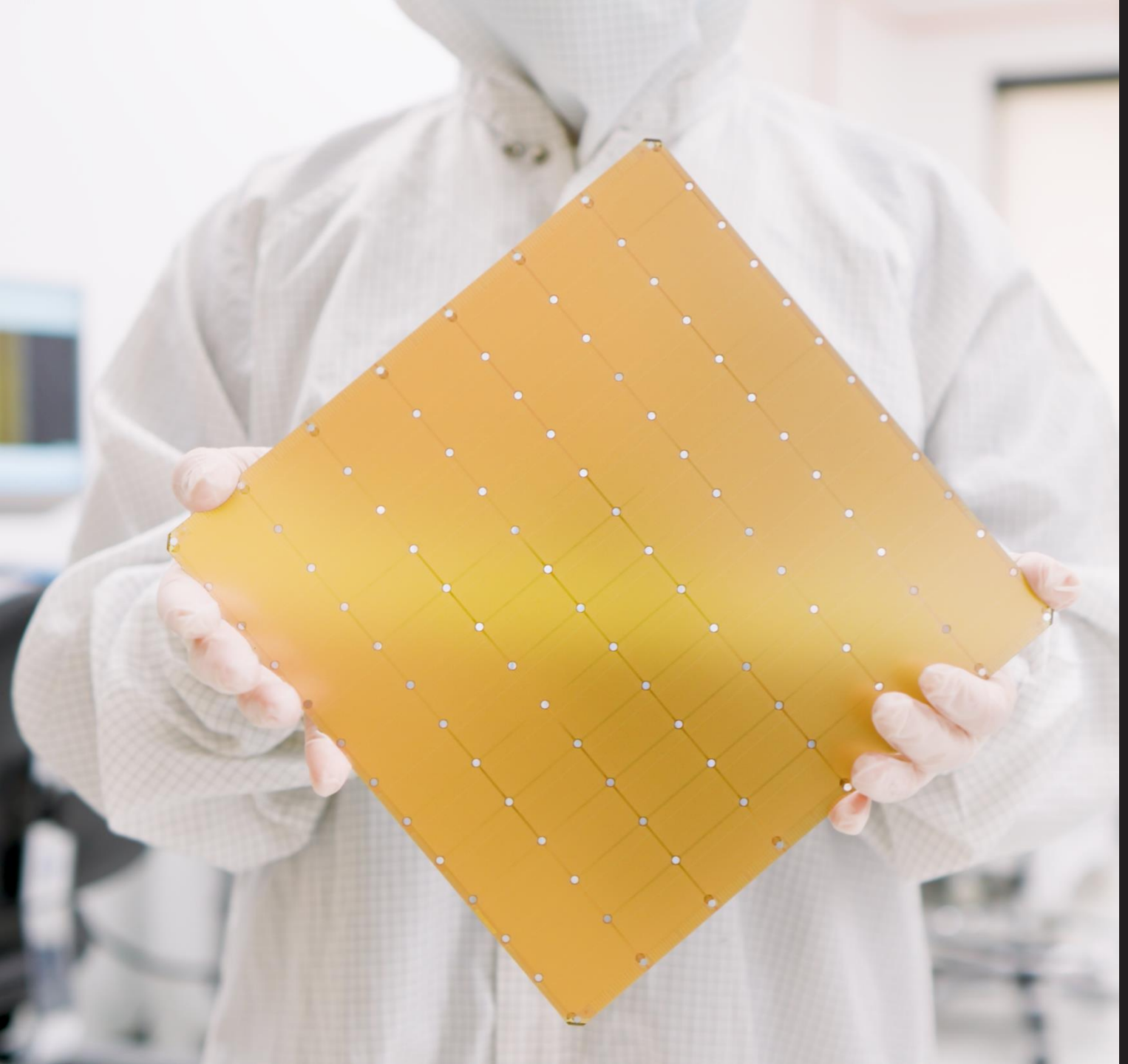
Cerebras HPC Research and SDK Overview



Cerebras Wafer-Scale Engine

The fastest AI chip on earth again

4 trillion transistors

46,225 mm2 silicon

900,000 cores optimized for sparse linear algebra

5nm TSMC process

125 Petaflops of AI compute

44 Gigabytes of on-chip memory

24 PByte/s memory bandwidth

245 Pbit/s fabric bandwidth

CS-3 vs. GPU

Orders of Magnitude Performance Advantage

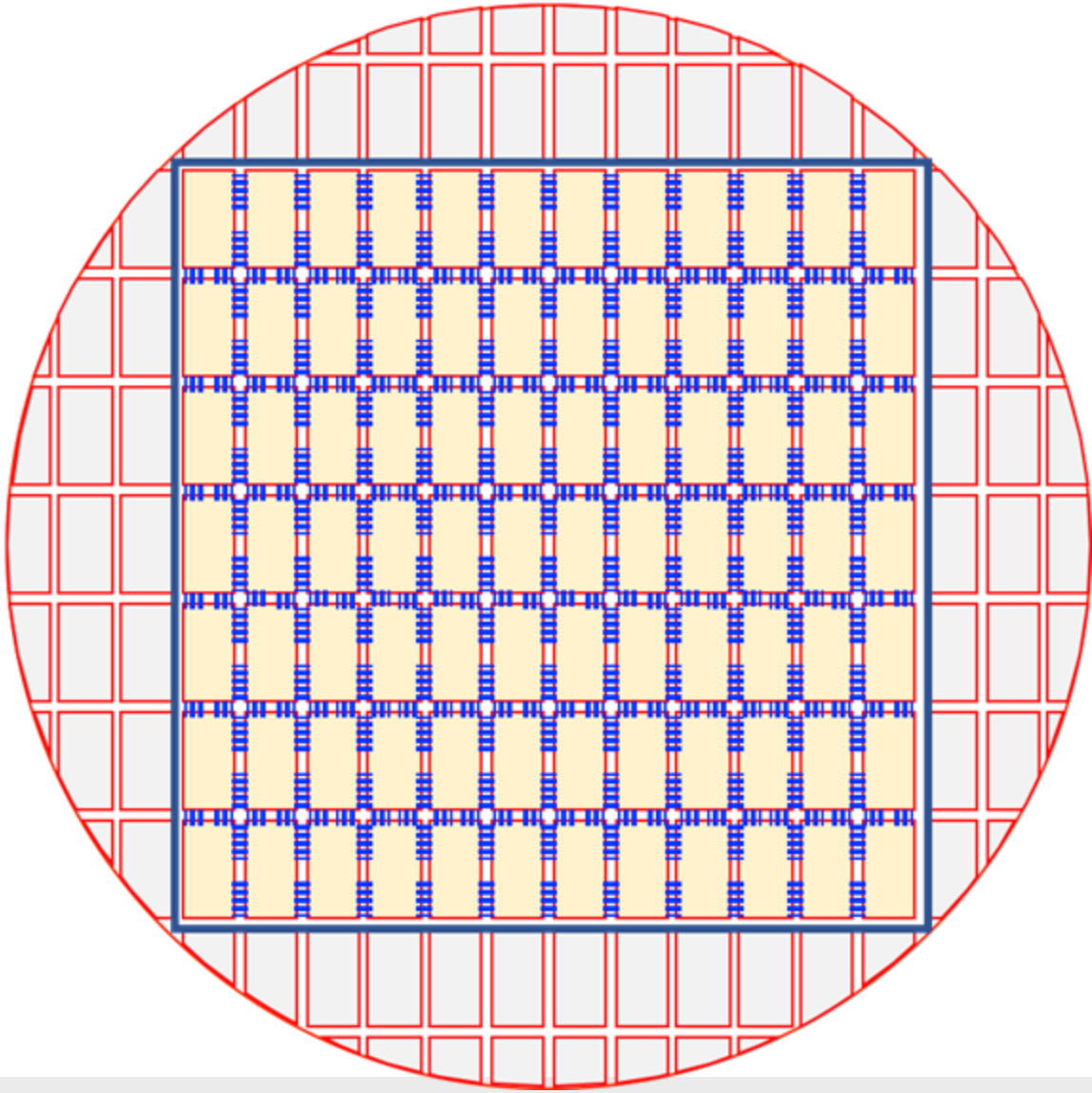
	Cerebras CS-3	Nvidia H100	Cerebras Advantage
Chip size	46,225 mm^2	814 mm^2	57x
Cores	900,000	16,896 FP32 + 528 Tensor	52x
On-chip memory	44 Gigabytes	0.05 Gigabytes	880x
Memory bandwidth	25 Petabytes/sec	0.003 Petabytes/sec	7,000X
Fabric bandwidth	245 Petabits/sec	0.0576 Petabits/sec	3,715X

Does your application scale poorly across nodes?

Examples: *FFT-based solvers, particle simulators, non-linear problems with iterative solvers*

The Cerebras solution:

- The WSE-3 has a fabric that is **high bandwidth** and **low-latency**, allowing for excellent parallel efficiency for non-linear and highly communicative codes
- The CS-3 system has **900k cores** and can fit problems on an individual chip that take tens to hundreds of traditional small compute nodes.
  - Each core is individually programmable



Is your application constrained by data access?

Examples: *Stencil based PDE solvers, linear algebra solvers, signal processing, sparse tensor math, big data analysis*

The Cerebras solution:

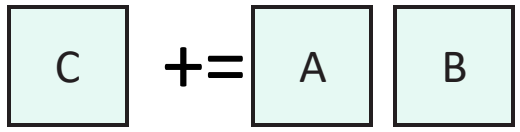
- The CS-3 system has **44 GB of SRAM** uniformly distributed across the wafer that is **1 cycle** away from the processing element
  - Speeds up memory access by orders of magnitude
- The CS-3 system is capable of **1.2 Tb/s bandwidth** onto the chip
  - Stream data onto the chip as required

Full Performance on All BLAS Levels

Matrix-Matrix

GEMM

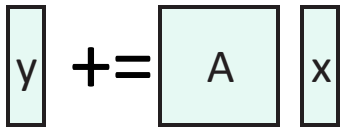
$C \leftarrow \alpha AB + \beta C$



Matrix-Vector

GEMV

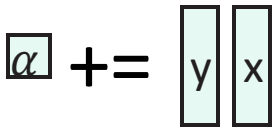
$y \leftarrow \alpha Ax + \beta y$



Vector-Vector

DOT

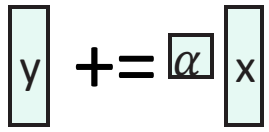
$\alpha \leftarrow (x, y)$



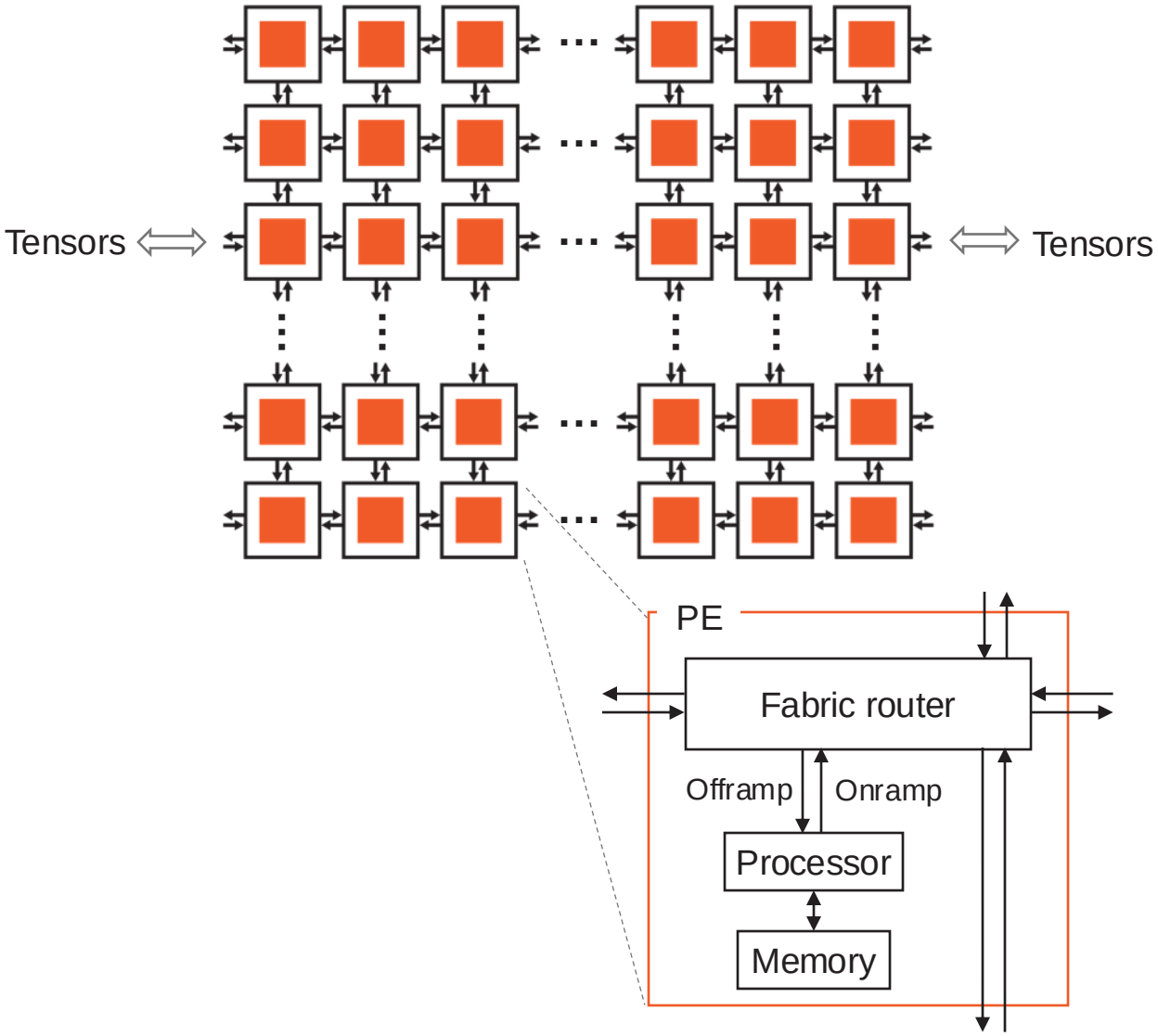
Vector-Scalar

AXPY

$y \leftarrow \alpha x + y$



WSE-3 Architecture Basics



The WSE appears as a logical 2D array of individually programmable Processing Elements

Flexible compute

- 900,000 general purpose CPUs
- 16- and 32-bit native FP and integer data types
- **Dataflow programming**: Tasks are activated or triggered by the arrival of data packets

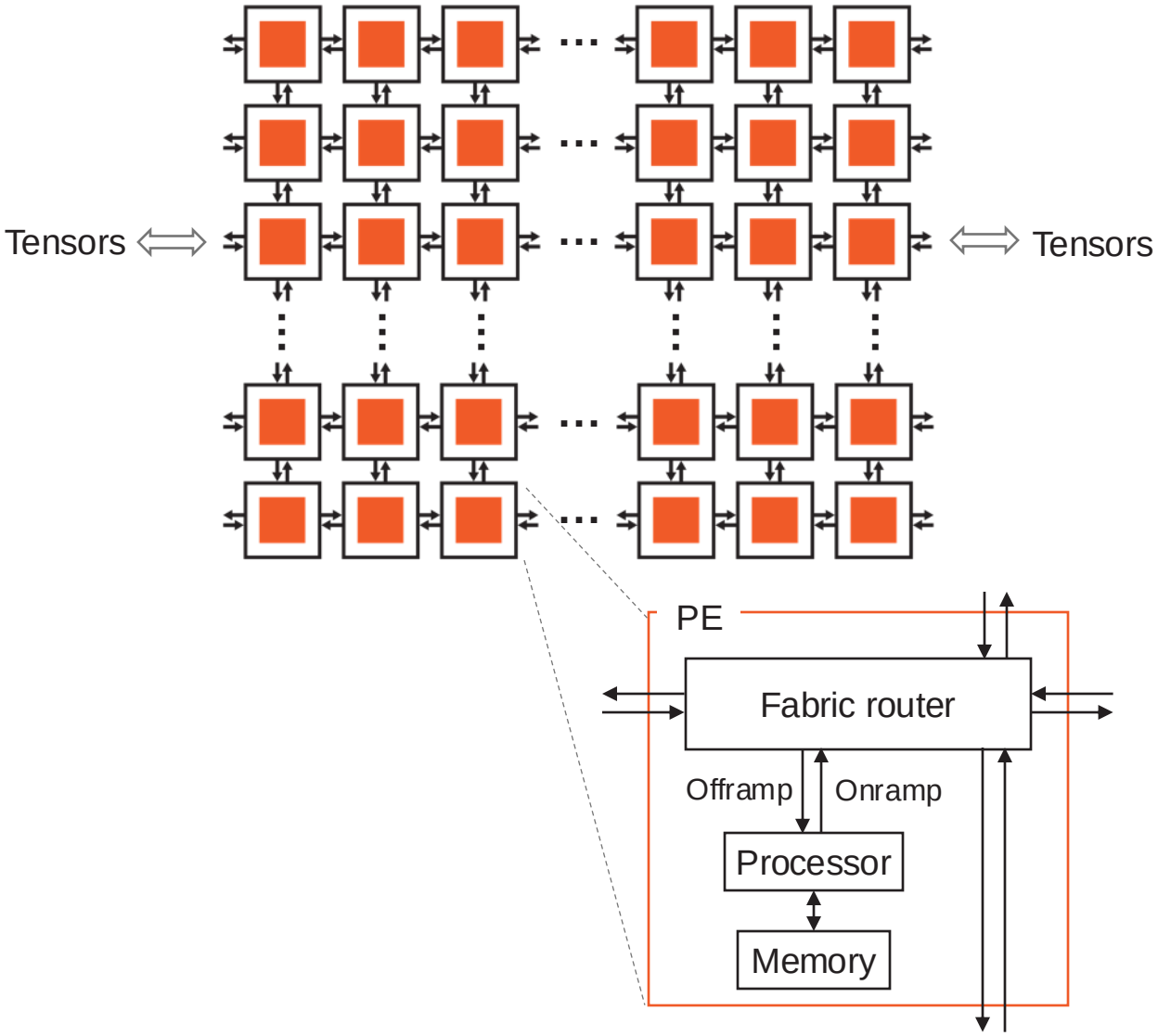
Flexible communication

- Programmable router
- Static or dynamic routes (**colors**)
- Data packets (**wavelets**) passed between PEs
- 1 cycle for PE-to-PE communication

Fast memory

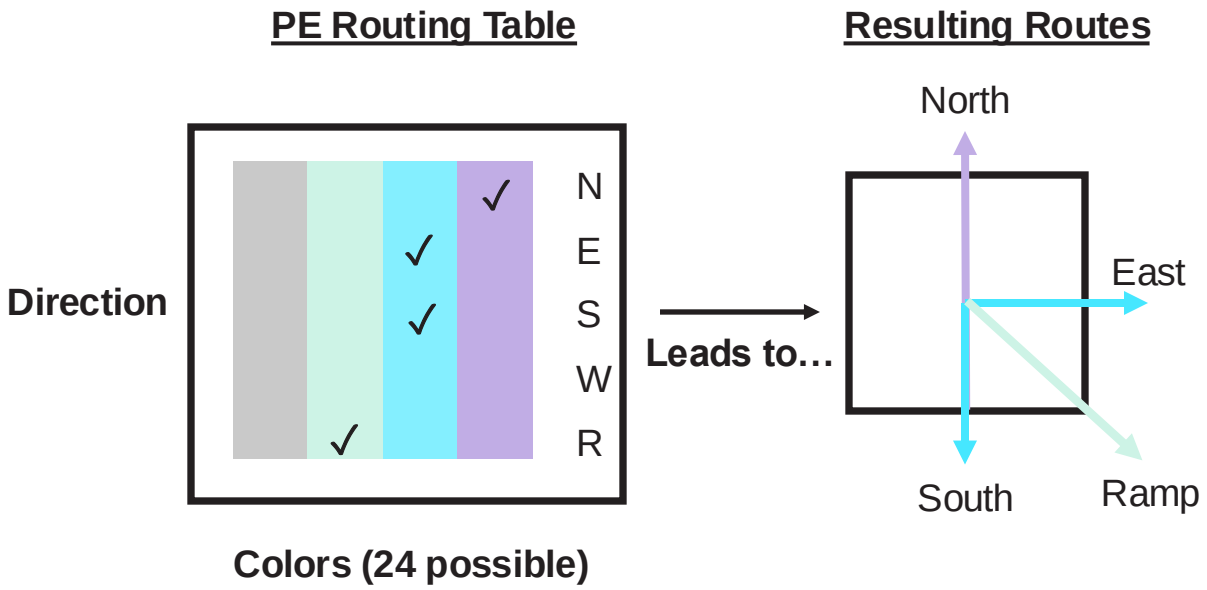
- 44GB on-chip SRAM
- Data and instructions
- 1 cycle read/write

Flexible Compute



- **Dataflow Execution Model**
  - Tasks may be triggered by **wavelets** or activated
  - Each color activates a distinct task
- Independent programs specified for regions of PEs
  - Programs specify computation for the processor and communication via **colors**
  - Parametrized programs allow execution of different control flow on different PEs
- Asynchronous operations performed by launching *microthreads*
- Control flow is straightforward to reason about
  - Tasks are non-preemptive
  - Instruction to activate another task enable state-machine behavior

Flexible Communication

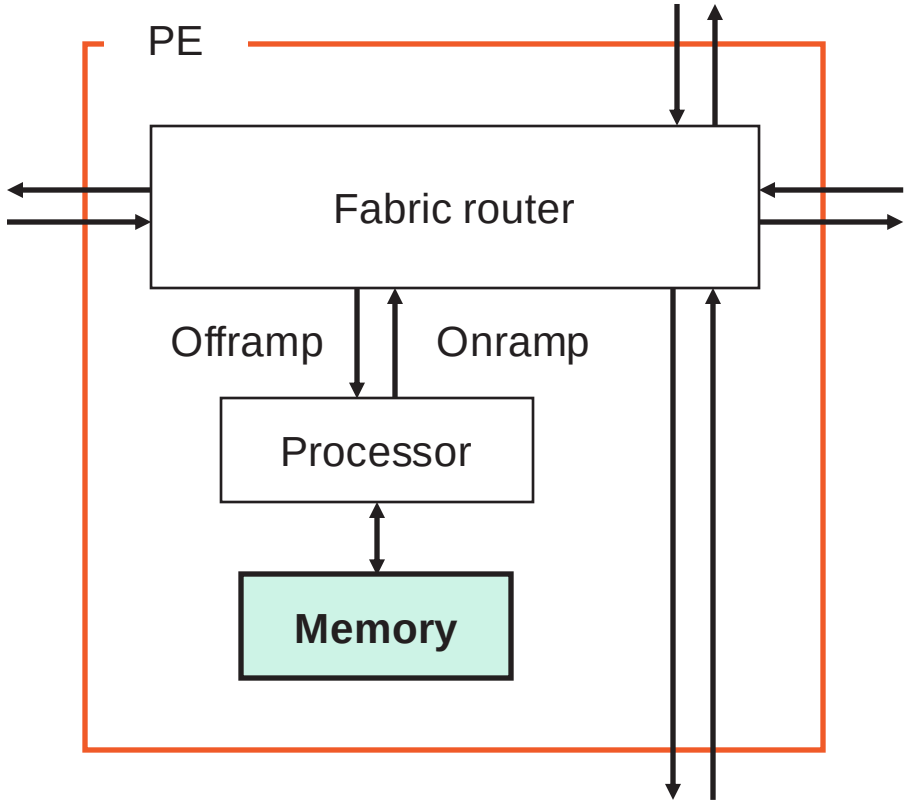


Router-to-router communication: **1 cycle**

Router-to-processor communication: **5 cycles**

- PEs communicate to adjacent PEs and their processor through their **routers**
- The **router** is a 24-entry table on each PE associating colors with directions
  - Table entries mapped to PE memory
  - Up to 24 routes (i.e. **colors**) may be specified at compile-time for each PE
- Complex communication patterns
  - Dynamic updating of routes at runtime
  - Multiple routing table entries per color enable *multicast*: broadcasting data in multiple directions at once each cycle
- Input/ output queues in each PE alleviate back pressure at routers during runtime
- Programmer feeds tensors into the fabric from outside world, specified in host program

Fast Memory



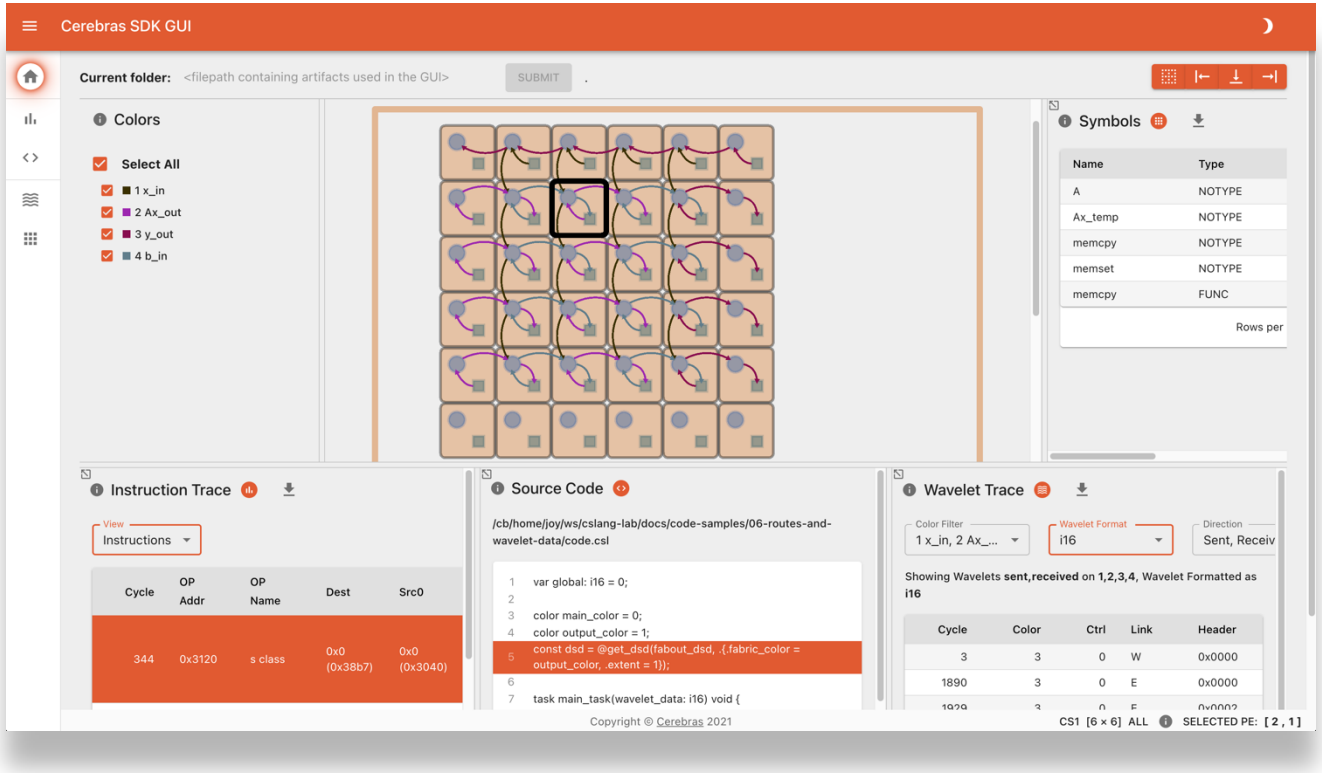
PE local memory read-write: **1 cycle**

- 44GB of on-chip SRAM
  - Uniformly distributed on wafer
  - 48kB per PE
- Programmer can read/write memory for regions of PEs at once from host
- Local PE memory is not directly addressable by other PEs, but is directly addressable by host program
- SIMD possible for vector instructions

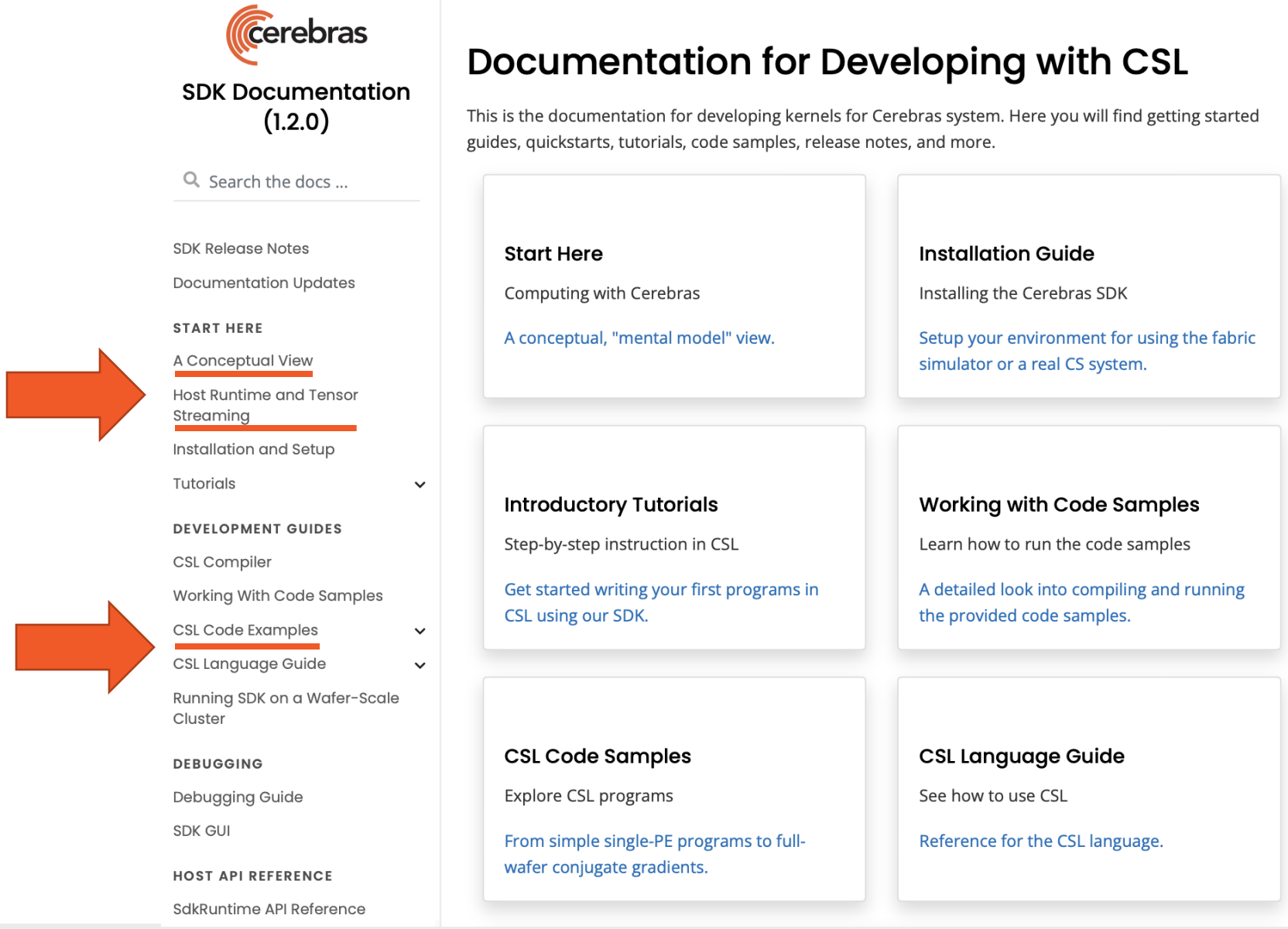
Cerebras SDK

A general-purpose parallel-computing platform and API allowing software developers to write custom programs (“kernels”) for Cerebras systems.

Language	CSL: Cerebras Software Language
	Host APIs with Python
Libraries	Optimized primitives
Tools	Simulator
	Debugger
	Visualization



Documentation: sdk.cerebras.net



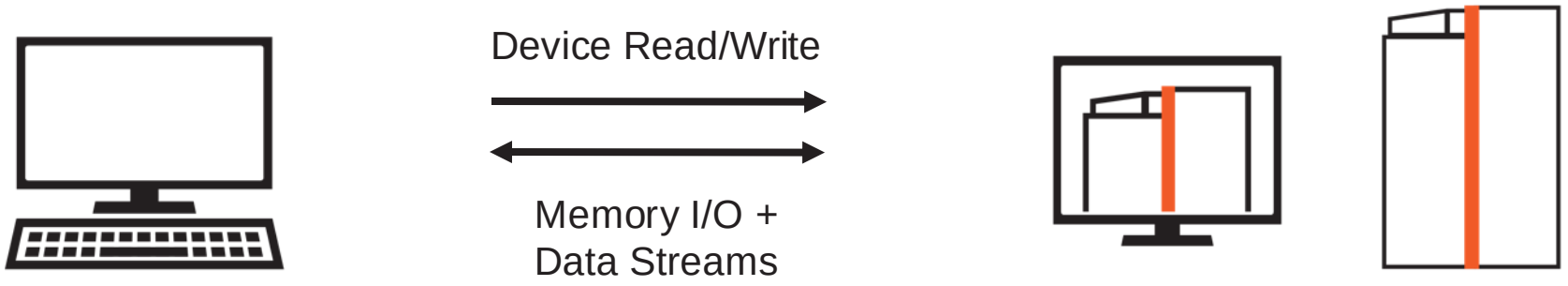
From a Programmer's Perspective

Host CPU(s): Python

- Loads program onto simulator or CS-3 system
- Streams in/out data from one or more workers
- Reads/writes device memory

Device: CSL

- Target software simulator or CS-3
- CSL programs run on groups of cores on the WSE, specified by programmer
- Executes dataflow programs



CSL: Language Basics

- Types
- Functions
- Control structures
- Structs/Unions/Enums
- Comptime

Straight from C
(via Zig)

- Builtins
- Module system
- Params
- Tasks
- Data Structure Descriptors
- Layout specification

CSL specific

Used for writing device kernel code

Familiar to C/C++/HPC programmers

Familiar Features

Types

- Syntax similar to other modern languages – Go, Swift, Scala, Rust
- Float (`f16`, `f32`), signed (`i16`, `i32`), unsigned (`u16`, `u32`), boolean (`bool`)

```
var x : i16;
const y = 42;
var arr : [16, 4]f32;
var ptr : *i16;
```

Functions

- Zig-style syntax
- Pass by value or reference and inlining automatically handled

```
fn factorial(x : i32) i32 {
    if (x <= 2) return x;
    return x * factorial(x - 1);
}
```

Control Structures

- Traditional control flow: **if**, **for**, **while**, with zig and C style syntax

```
if (x < 10) {
    y += 5;
} else {
    y += 10;
}
```

conditionals

```
var x: u16 = 100;
while(x > 99) {
    ...
}
```

while loop

```
var idx: u16 = 0;
while (idx < 5) : (idx += 1) {
  ...
}
```

while loop with iterator

```
const xs = [10]i16 { 0, 1, 2, 4 };
for (xs) |x,idx| {
    ...
}
```

range **for** loop
(also provides C-style **for**)

Quality of Life Features

Comptime

- From Zig, block of code where all evaluation occurs at compile time
- Useful for frontloading computation to avoid runtime overhead

```
comptime {
  const f23 = factorial(23);
  ...
}
```

Params

- Like `#define`, but strongly typed
- Have to be “bound” completely during compilation

```
param M : i16;
param N : i16;
param is_left_edge : bool;
```

Modules

- Any CSL source code file is a “Module,” importable into other modules
- Imported modules acts as an *instance* of a unique struct type
- Multiple imports of the same module allowed

m1.csl

```
var x = 0;
fn incr() void {
    x = x + 1;
}
```

p1.csl

```
const v1 = @import_module("m1.csl");
const v2 = @import_module("m1.csl");

v1.incr();
v2.incr(); v2.incr();

// v1.x == 1; v2.x == 2;
```

Performance Features

Builtins

- Similar to function calls with @ in front of function name
- Language extensions without special syntax
- Used for invoking special compiler functionality

```
// Initialize a tensor of four rows
// and five columns with all zeros.
  var matrix = @zeros([4,5]f16);
```

Tasks

- Core building blocks of CSL
- Special functions used to implement dataflow programs
- Triggered by incoming wavelets on a specific color

```
color recvColor;
var globalValue: u16 = 0;

task recvTask(data: u16) void {
  globalValue = data;
}

comptime {
  @bind_task(recvTask, recvColor);
  @set_local_color_config(recvColor,
    .{ .rx = .{ WEST }, .tx = .{ RAMP } });
}
```

Performance Features

Data Structure Descriptors (DSDs)

- Provide a mechanism to consider an array, and an access pattern, as a complete unit
- Operations using DSDs run for multiple cycles to complete an instruction on all data referenced by the DSD
- Performance *and* ease of use: lifts level of program to talking about whole structures, while lowering cost of computing indexing into hardware

```
const dstDsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{5} -> dst[i] });
const src0Dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{5} -> src0[i] });
const src1Dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{5} -> src1[i] });

const fabDsd = @get_dsd(fabout_dsd, .{.fabric_color = output_color, .extent = 1});

task main_task() void {
  @faddh(dstDsd, src0Dsd, src1Dsd);
  @fmovh(fabDsd, dstDsd);
}
```

DSDs are a ***unifying concept*** that provides for complex memory reads and writes and fabric reads and writes

SDK Example Programs Available

Repository: github.com/Cerebras/csl-examples

- Introductory Tutorials
- GEMV
- GEMM
- Cholesky Decomposition
- 1D and 2D FFT
- 7-Point Stencil SpMV
- Power Method
- Conjugate Gradient
- Preconditioned Conjugate Gradient
- Finite Difference Stencil Computations
- Mandelbrot Set Generator
- Shift-Add Multiplication
- Hypersparse SpMV
- Histogram Computation

SDK Access and Next Steps

Get local access to the SDK simulator!

- Email developer@cerebras.net for access

Join the Cerebras Developer Community

- Forums at discourse.cerebras.net

View our public SDK examples GitHub repository

- See github.com/Cerebras/csl-examples

Questions? developer@cerebras.net

discourse.cerebras.net

cerebras.net/developers/sdk-request

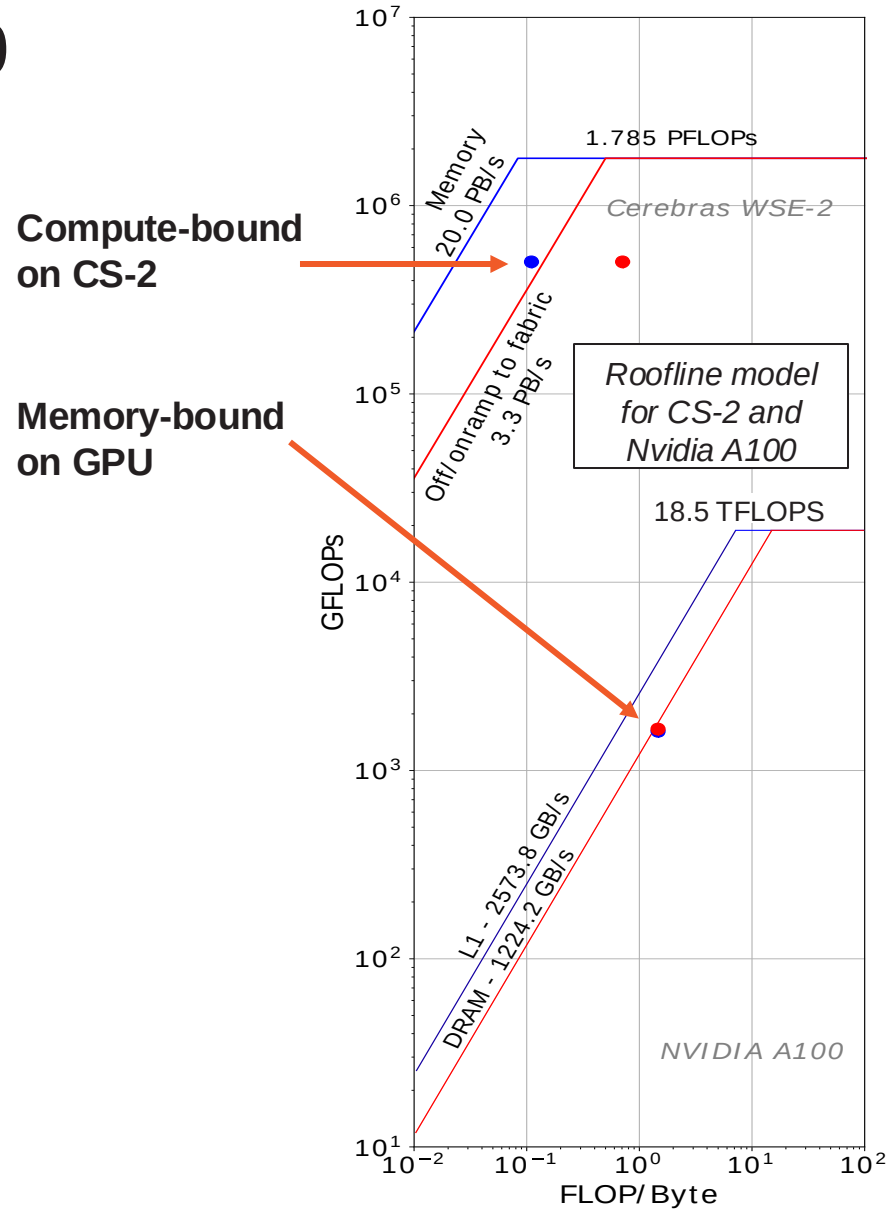
TotalEnergies achieves 228x speedup vs. A100 on seismic imaging algorithm

Common computational approaches to solving seismic imaging problems, such as stencil methods, are typically memory-bound.

Additionally, strong scaling is typically limited by fabric bandwidth between compute nodes.

Total has addressed these challenges with Cerebras:

- Implemented 25-point stencil for the 3D wave equation with source perturbation, achieved **228x speedup over A100**. Presented at SC22.
- Implemented finite volume flux computation for single phase flow, achieved **204x speedup over A100**. Presented at SC23.
- Additionally developed proprietary RTM (Reverse Time Migration) code for internal use.



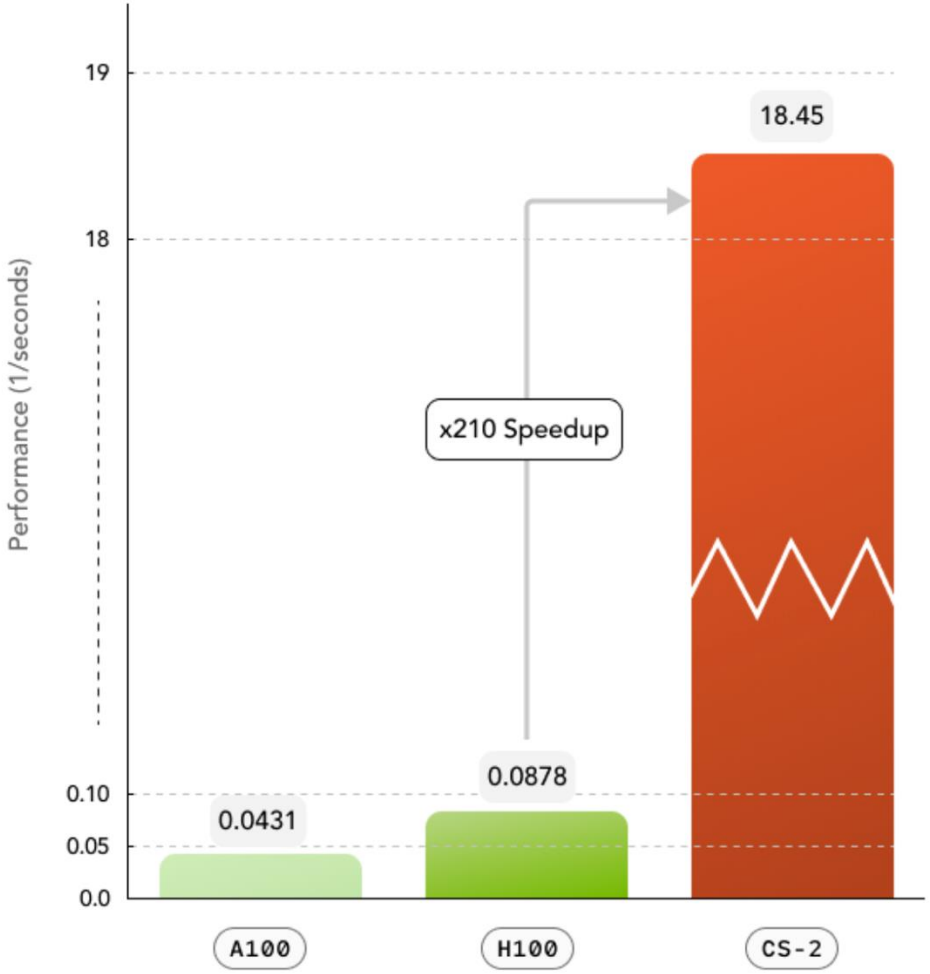
Papers: https://arxiv.org/abs/2204.03775 and https://arxiv.org/abs/2304.11274

TotalEnergies achieves >200x ***H100*** on finite volume simulation

TotalEnergies keeps innovating with Cerebras in 2024

- Matrix-free finite volume solvers are an essential tool for Total's work on geological carbon capture and storage
- Solver implementation for CS-2 achieved **210x speedup over H100** GPU

- Total has also developed StencilPy, a Python framework for stencil computations on the wafer-scale engine
- StencilPy 25-pt stencil for seismic acoustic wave propagation achieved **95x speedup over H100** GPU
- Both papers to appear at SC24



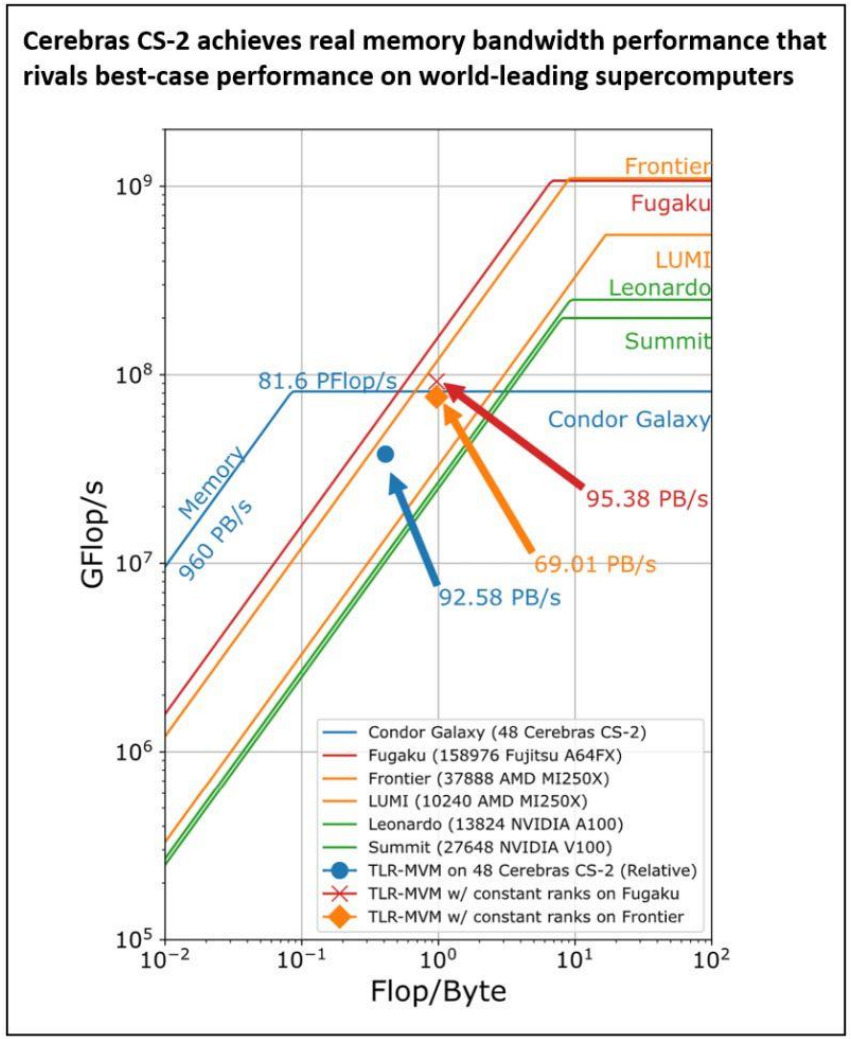
Papers: https://arxiv.org/pdf/2408.03452 and https://arxiv.org/pdf/2309.04671
Blogs: Matrix-free finite volume and StencilPy

Cerebras and KAUST break records on seismic processing

- Researchers redesigned a Tile Low-Rank Matrix-Vector Multiplication (TLR-MVM) algorithm for Cerebras CS-2, taking advantage of the ultra high memory bandwidth
- Provided researchers with CG-1 AI supercomputer to run this simulation
- Achieved sustained memory bandwidth of **92.58 PB/s** across 48 CS-2 systems – higher than Frontier (#1 TOP500), comparable to Fugaku (#4 TOP500)

2023 Gordon Bell Prize finalist

Paper: https://dl.acm.org/doi/10.1145/3581784.3627042

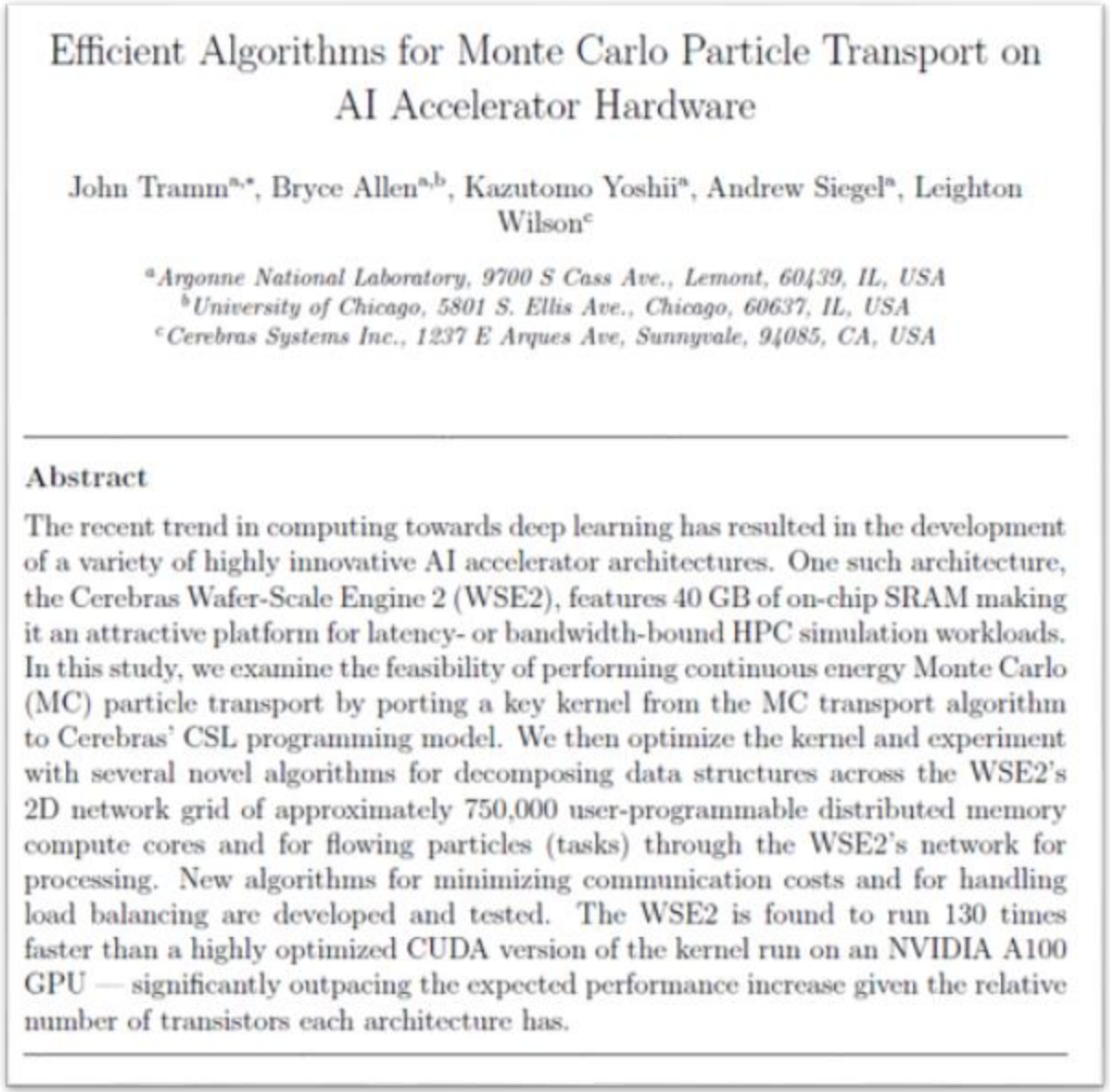


Argonne National Labs Uses CS-2 to Accelerate Monte Carto Particle Transport by **180x** Over A100

*"The WSE is found to run **130 times faster** than a highly optimized CUDA version of the kernel run on an NVIDIA A100 GPU – significantly outpacing the expected performance increase given the relative number of transistors each architecture has"*

New PHYSOR publication demonstrates **180x** over A100.

Paper: https://arxiv.org/abs/2311.01739



Efficient Algorithms for Monte Carlo Particle Transport on AI Accelerator Hardware

John Tramm[a,*], Bryce Allen[a,b], Kazutomo Yoshii[a], Andrew Siegel[a], Leighton Wilson[c]

[a] *Argonne National Laboratory, 9700 S Cass Ave., Lemont, 60439, IL, USA*
[b] *University of Chicago, 5801 S. Ellis Ave., Chicago, 60637, IL, USA*
[c] *Cerebras Systems Inc., 1237 E Arques Ave, Sunnyvale, 94085, CA, USA*

Abstract

The recent trend in computing towards deep learning has resulted in the development of a variety of highly innovative AI accelerator architectures. One such architecture, the Cerebras Wafer-Scale Engine 2 (WSE2), features 40 GB of on-chip SRAM making it an attractive platform for latency- or bandwidth-bound HPC simulation workloads. In this study, we examine the feasibility of performing continuous energy Monte Carlo (MC) particle transport by porting a key kernel from the MC transport algorithm to Cerebras' CSL programming model. We then optimize the kernel and experiment with several novel algorithms for decomposing data structures across the WSE2's 2D network grid of approximately 750,000 user-programmable distributed memory compute cores and for flowing particles (tasks) through the WSE2's network for processing. New algorithms for minimizing communication costs and for handling load balancing are developed and tested. The WSE2 is found to run 130 times faster than a highly optimized CUDA version of the kernel run on an NVIDIA A100 GPU — significantly outpacing the expected performance increase given the relative number of transistors each architecture has.

CS-2 Accelerates molecular dynamics for metallic alloys
179x faster than Frontier

"Measured performance and power efficiency of WSE, GPU, and CPU systems on 800,000-atom simulations. WSE used FP32 precision while GPU and CPU used FP64 precision. (a) A ***single WSE wafer results in 179x and 55x speedup compared to Frontier*** *and* ***CPU*** *based simulations; (b) WSE provides one to two orders of magnitude improvement in power efficiency over both CPU and GPU systems; (c) Relative power efficiency and speedup of WSE compared to CPU and GPU systems."*

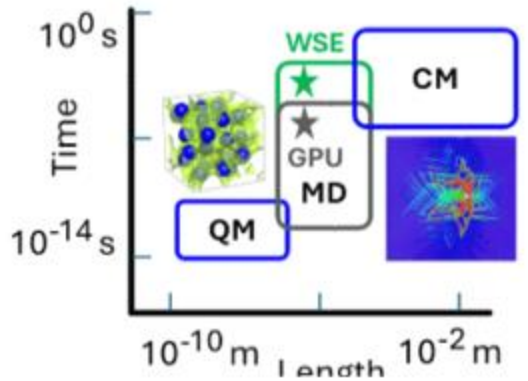
2024 Gordon Bell Prize finalist

Fast Molecular Dynamics on a Wafer-Scale System

Kylee Santos*, Stan Moore†, Tomas Oppelstrup‡, Amirali Sharifian*, Ilya Sharapov*, Aidan Thompson†,
Delyan Z Kalchev*, Danny Perez§, Scott Pakin§, Edgar A. Leon‡, James H Laros III†,
Michael James*, and Sivasankaran Rajamanickam†
*Cerebras Systems, Sunnyvale, CA
†Sandia National Laboratories, Albuquerque, NM
‡Lawrence Livermore National Laboratory, Livermore, CA
§Los Alamos National Laboratory, Los Alamos, NM

***Abstract*—Molecular dynamics (MD) simulations have transformed our understanding of atomic systems, driving breakthroughs in material science, computational chemistry and several other fields like biophysics and drug design. Using the Cerebras Wafer-Scale Engine, we demonstrate an improvement in MD iteration rate that enables a transformative capability for long-time simulations. This unlocks currently inaccessible timescales of slow microstructure transformation processes that are critical for understanding material behavior and function.**

Our dataflow algorithm runs an Embedded Atom Method (EAM) simulation at rates over 270,000 timesteps per second for problems with up to 800k atoms. This corresponds to a nearly 180-fold speedup versus the Frontier GPU-based Exascale platform. It simultaneously achieves an over 30-fold improvement in energy efficiency. This demonstrated performance is unprecedented for general-purpose processing cores. With further parallelization of the algorithm, we project performance in excess of one million timesteps per second for 200,000 atoms. This projected perfor-



Example: GEMV

See: gemv-08-routes-3
gemv-09-streaming

Example: GEMV

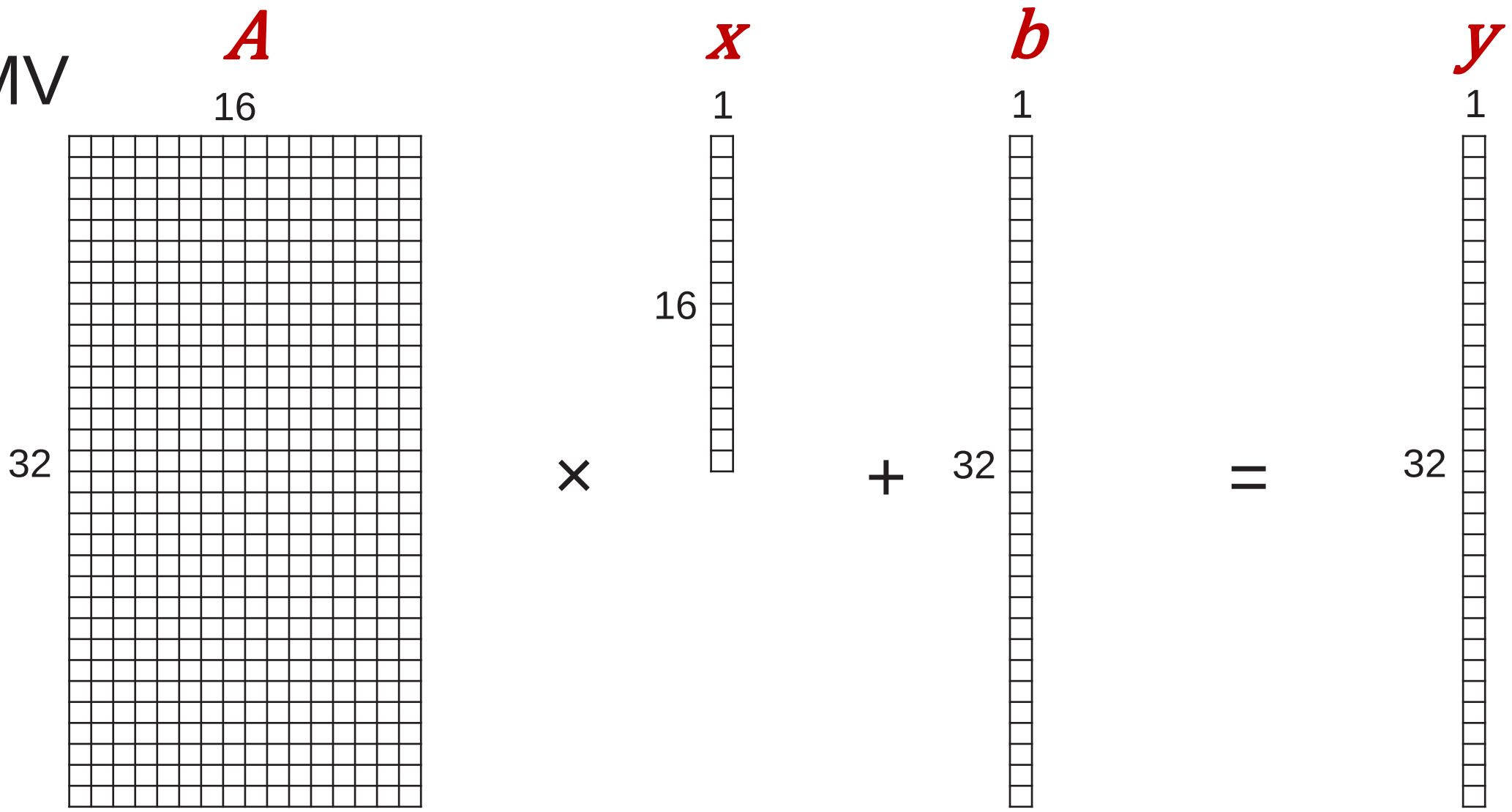
Run a program with non-trivial communication that performs general matrix-vector multiplication

Goals:

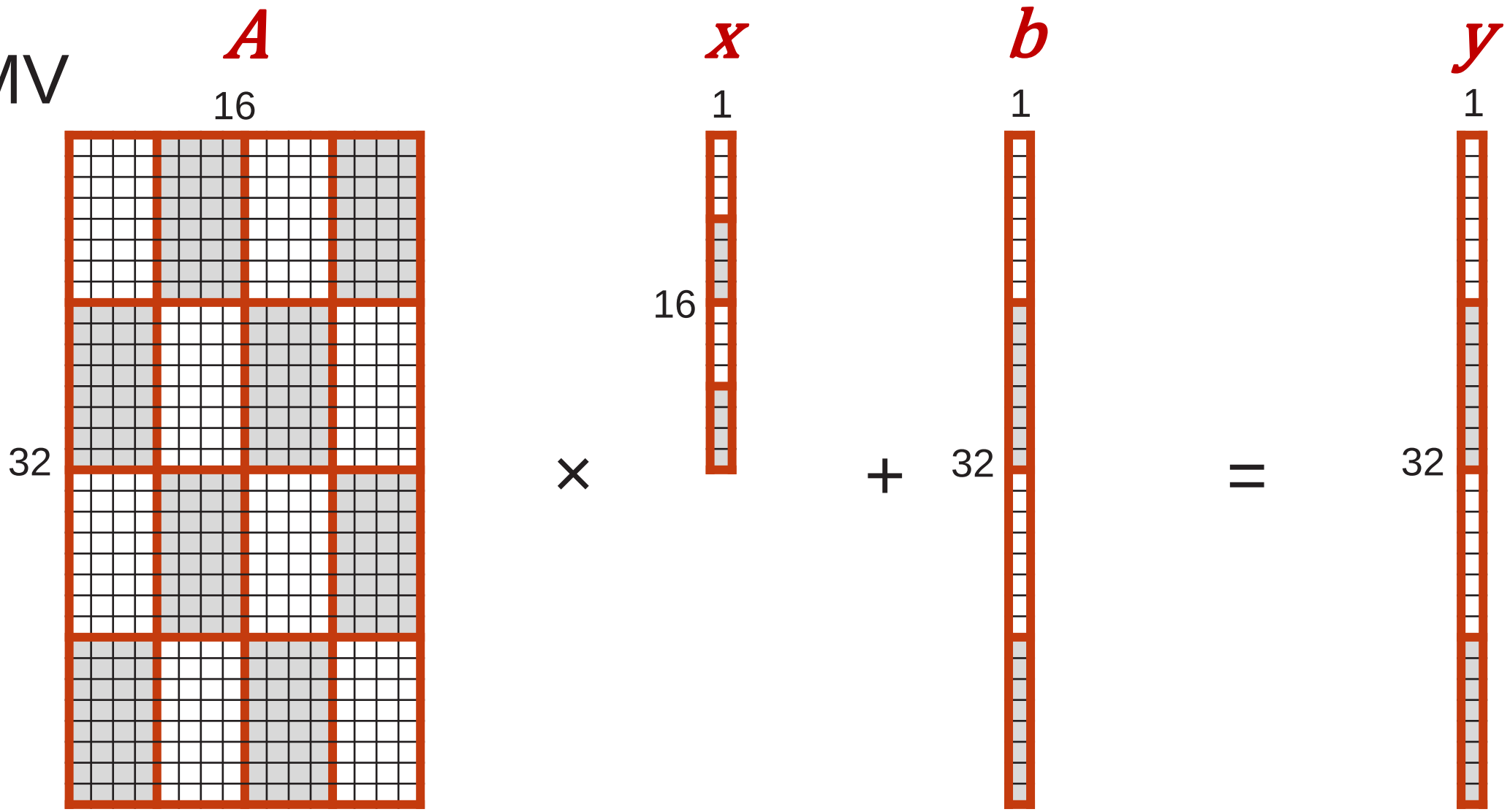
- Highlight **(1)** layout, **(2)** PE programming, and **(3)** runtime components of a non-trivial program
- Write a CSL program with multiple tasks and communication between PEs
- Use memory, fabric input, and fabric output DSDs
- Use the checkerboard pattern for communicating data across wafer
- Use the `memcpy` framework to move data on and off wafer

Location of example:
https://sdk.cerebras.net/csl/code-examples/tutorial-gemv-09-streaming

GEMV

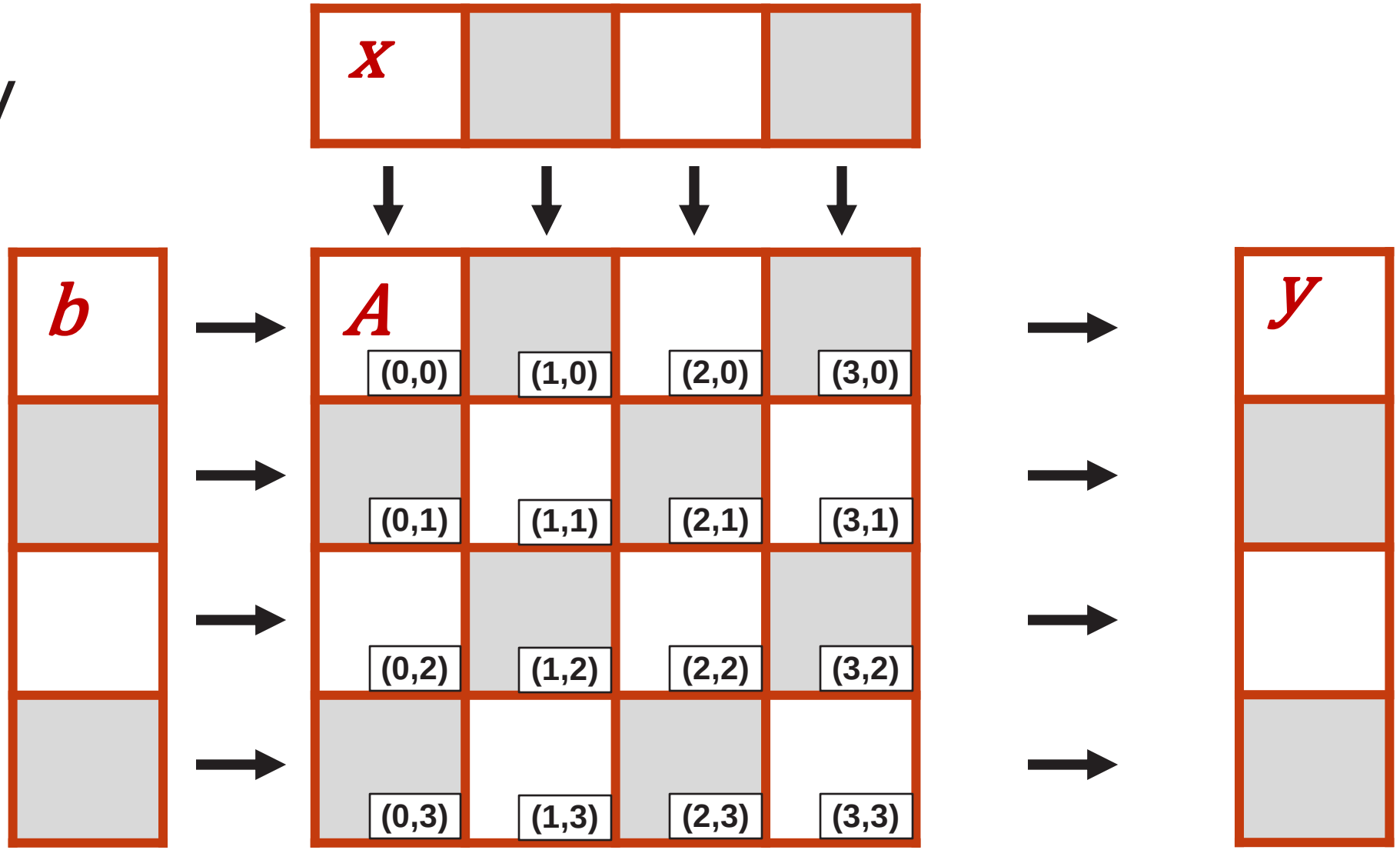


GEMV



Problem is distributed onto 4 x 4 grid of PEs.

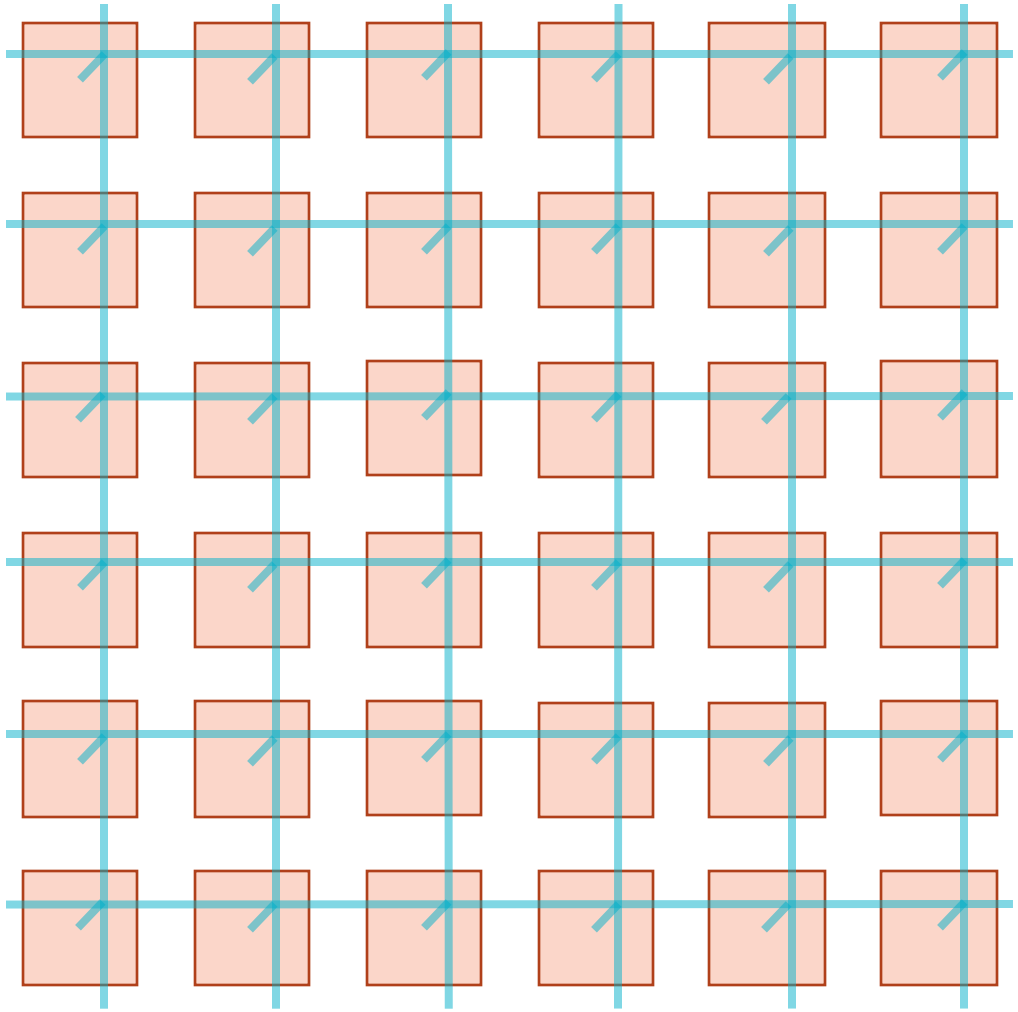
GEMV



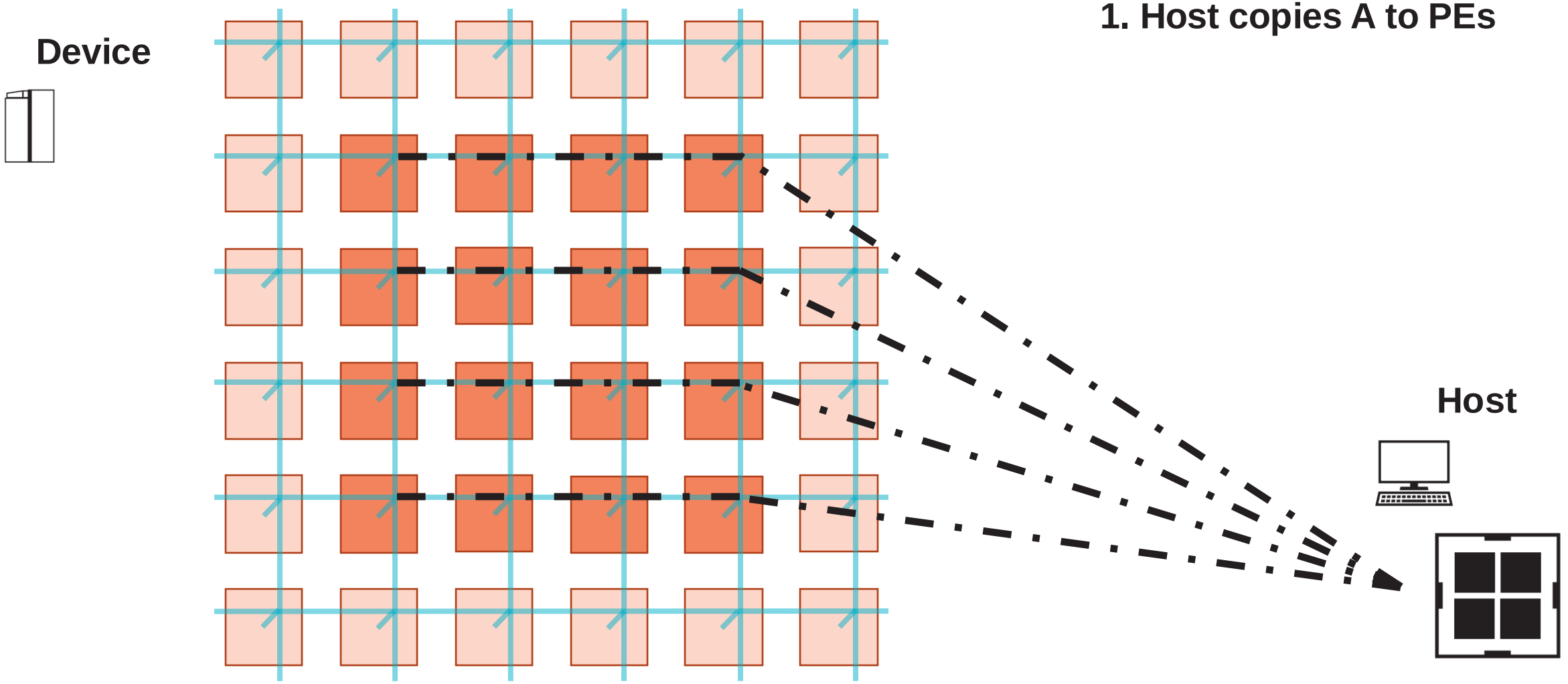
- Host streams in b from the West, x from the North; streams out y to the East
- A is copied to the wafer

GEMV Problem Steps

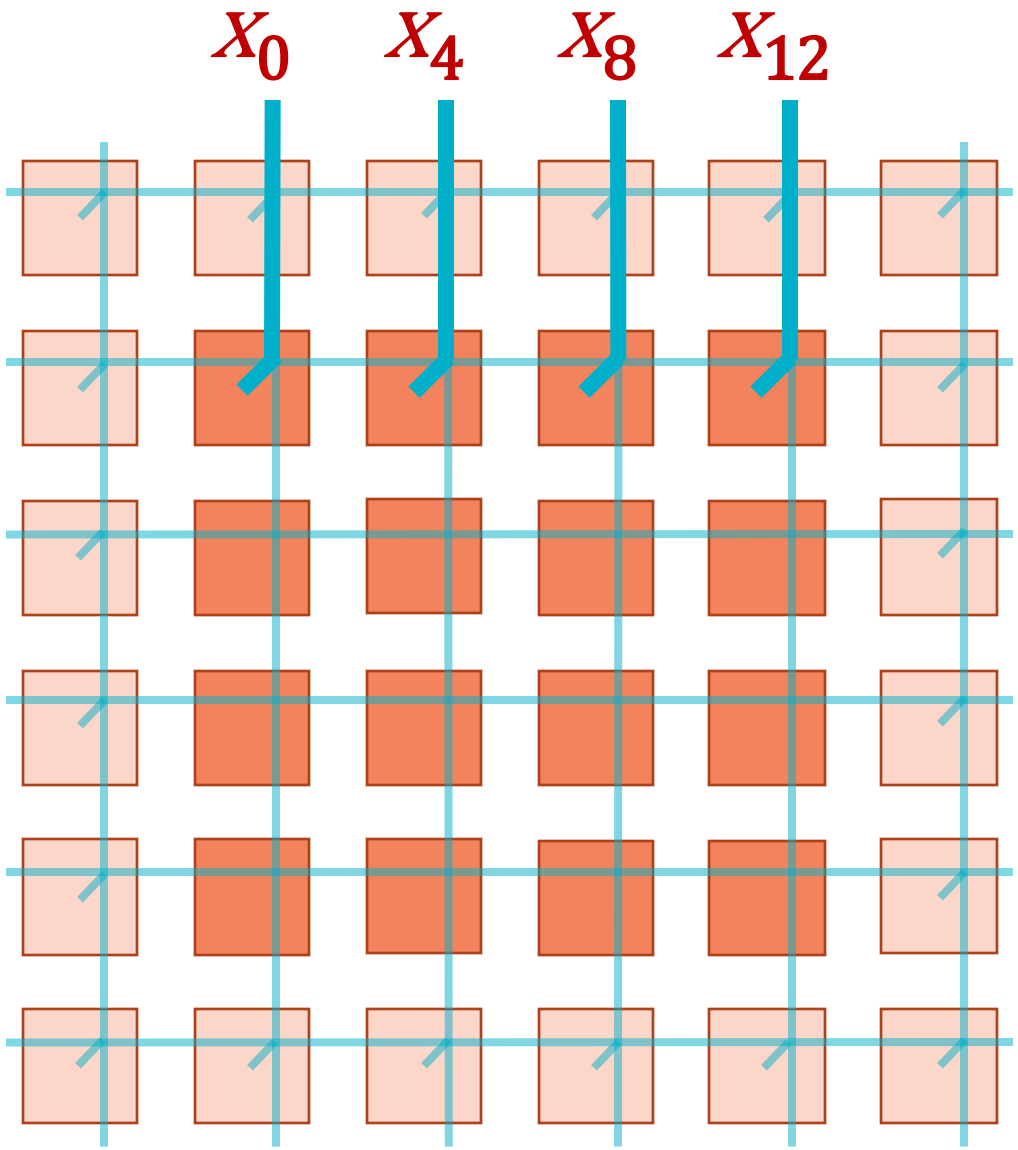
Device



Host



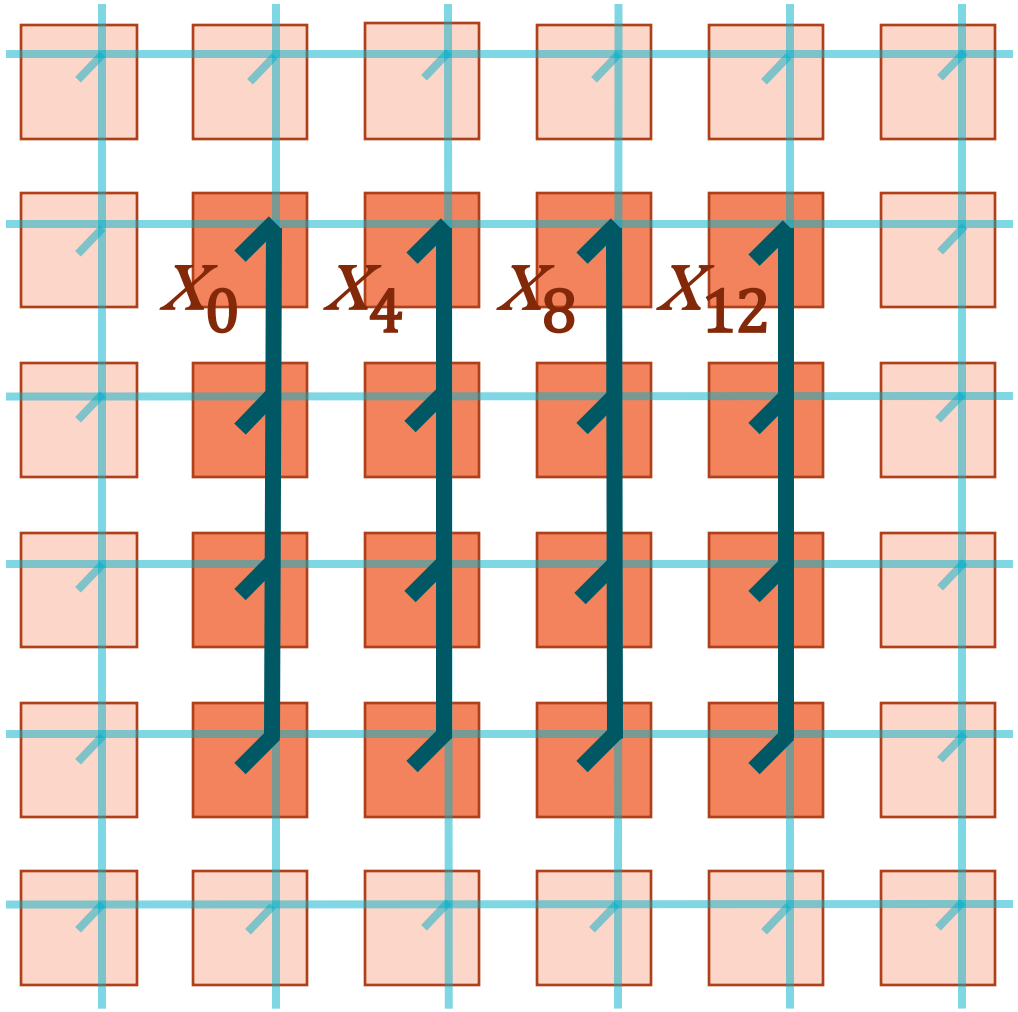
Device



2. Host streams in 0th, 4th, 8th, 12th elements of X into PEs (0,0), (1,0), (2,0), (3,0), respectively

Host

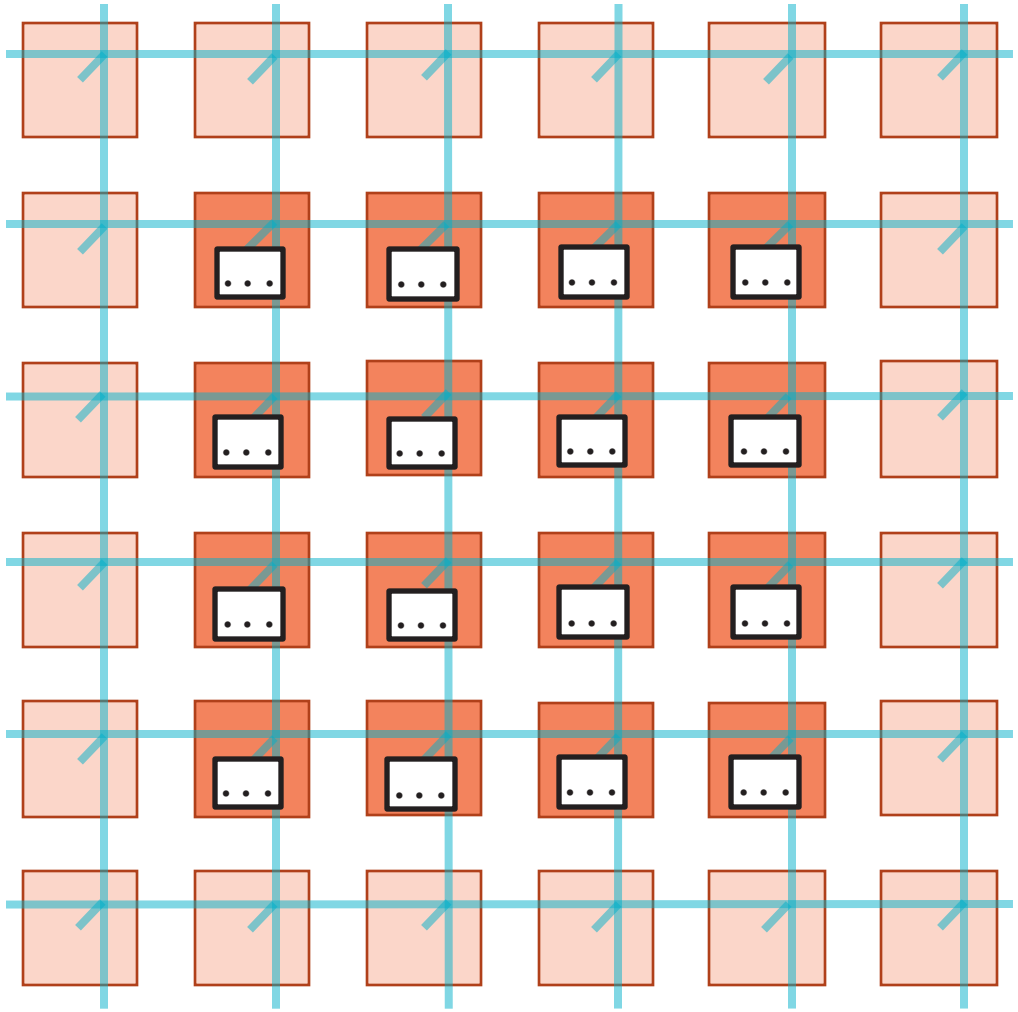
Device



3. Activated task forward elements to PEs (0,*i*), (1,*i*), (2,*i*), (3,*i*)

Host

Device



4. Device computes for $j = 0, 4, 8, 12$:

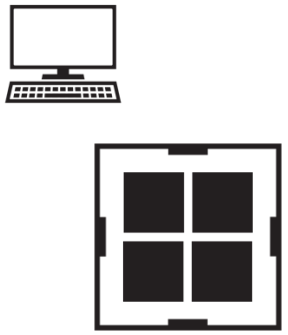
$\text{tmp}_i = A_{i,j}x_j,\ i = 0{:}7$ **on 0th row PEs**

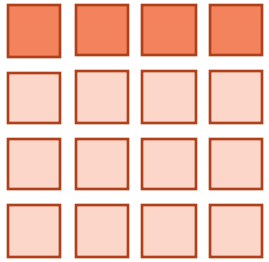
$\text{tmp}_i = A_{i,j}x_j,\ i = 8{:}15$ **on 1st row PEs**

$\text{tmp}_i = A_{i,j}x_j,\ i = 16{:}23$ **on 2nd row PEs**

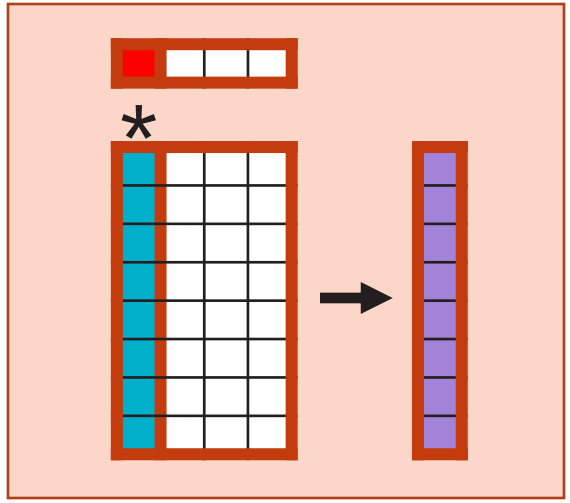
$\text{tmp}_i = A_{i,j}x_j,\ i = 24{:}31$ **on 3rd row PEs**

Host



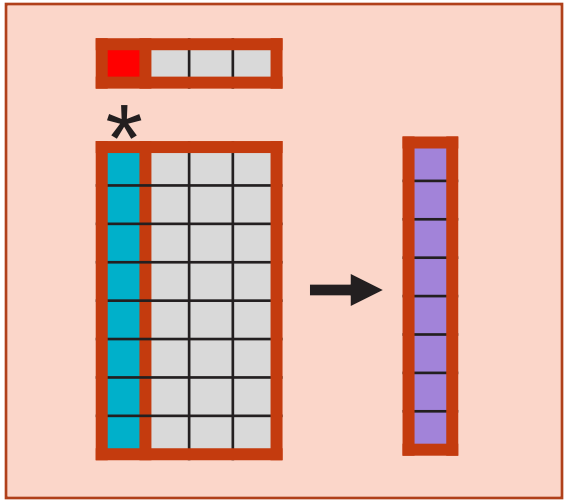


$\text{tmp}_i = A_{i,0}x_0,$
$i = 0:7$



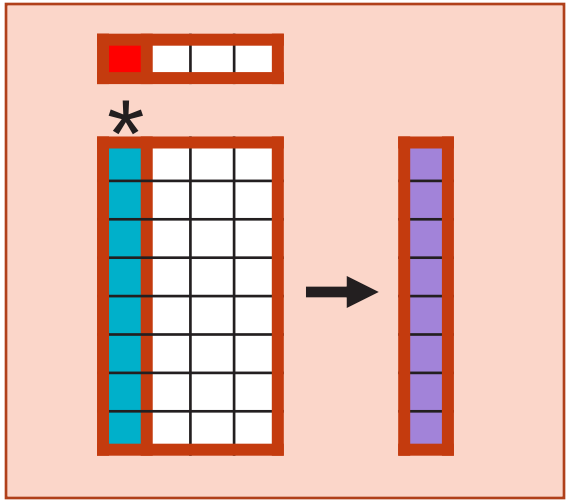
PE (0,0)

$\text{tmp}_i = A_{i,4}x_4,$
$i = 0:7$



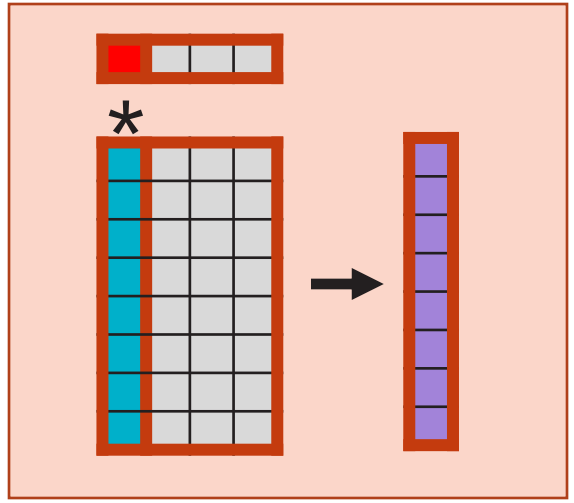
PE (1,0)

$\text{tmp}_i = A_{i,8}x_8,$
$i = 0:7$

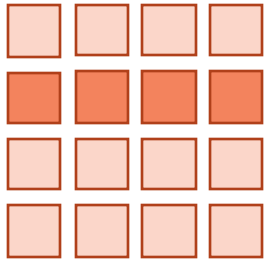


PE (2,0)

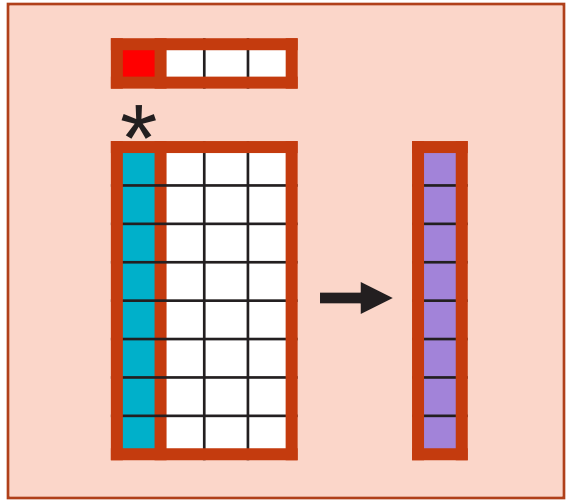
$\text{tmp}_i = A_{i,12}x_{12},$
$i = 0:7$



PE (3,0)

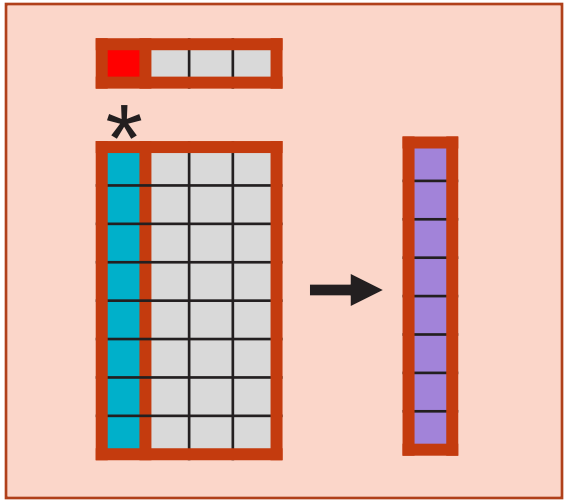


$\text{tmp}_i = A_{i,0}x_0,$
$i = 8:15$



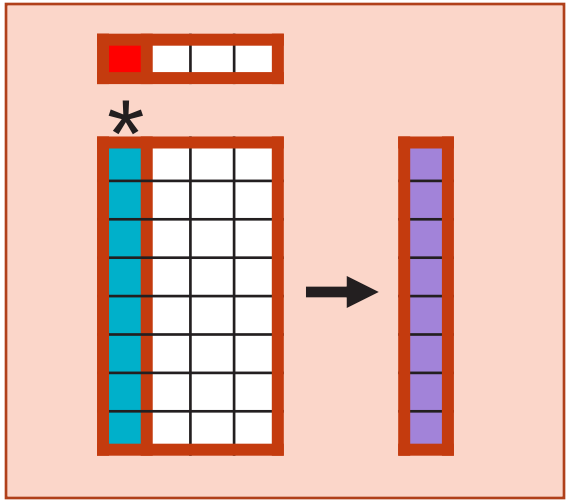
PE (0,1)

$\text{tmp}_i = A_{i,4}x_4,$
$i = 8:15$



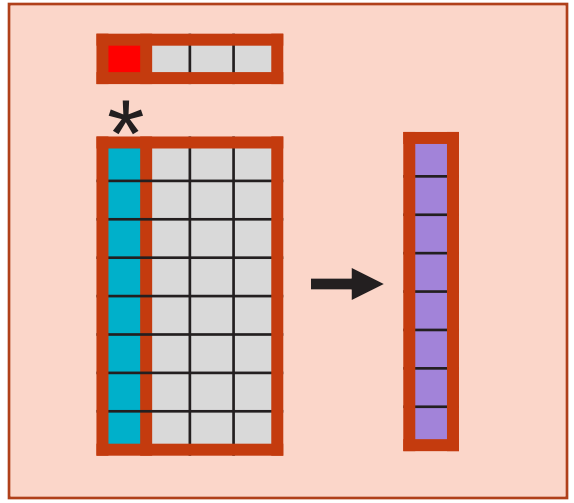
PE (1,1)

$\text{tmp}_i = A_{i,8}x_8,$
$i = 8:15$

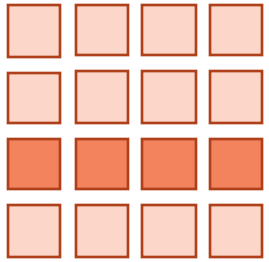


PE (2,1)

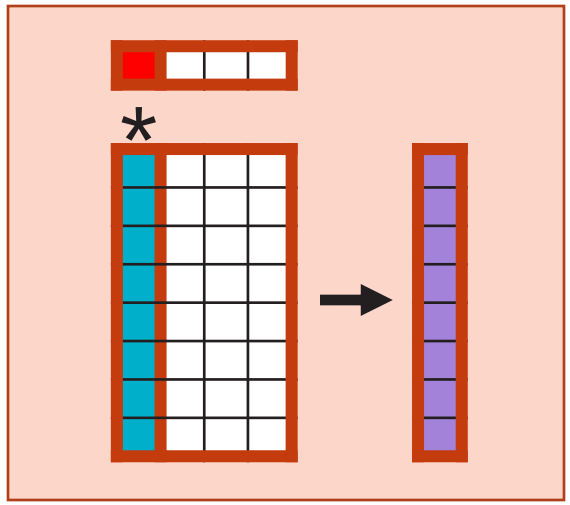
$\text{tmp}_i = A_{i,12}x_{12},$
$i = 8:15$



PE (3,1)

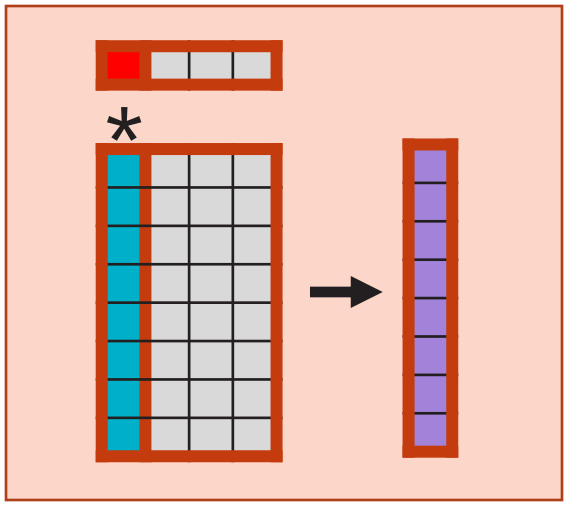


$\text{tmp}_i = A_{i,0}x_0,$
$i = 16:23$



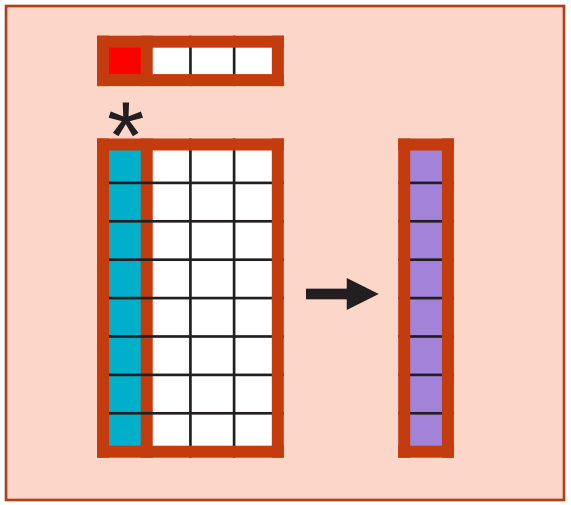
PE (0,2)

$\text{tmp}_i = A_{i,4}x_4,$
$i = 16:23$



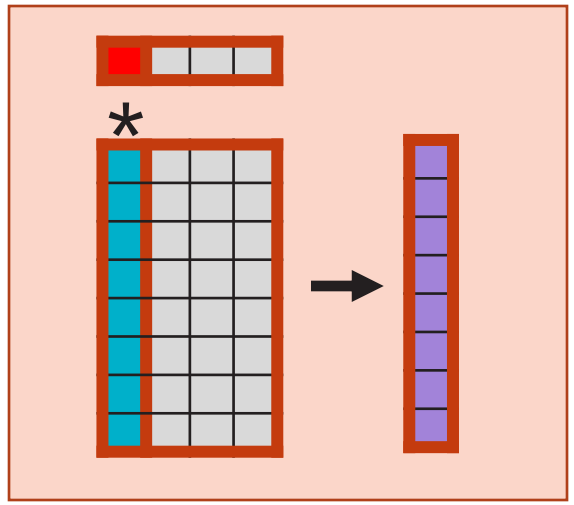
PE (1,2)

$\text{tmp}_i = A_{i,8}x_8,$
$i = 16:23$

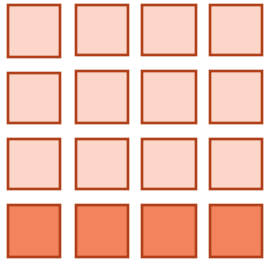


PE (2,2)

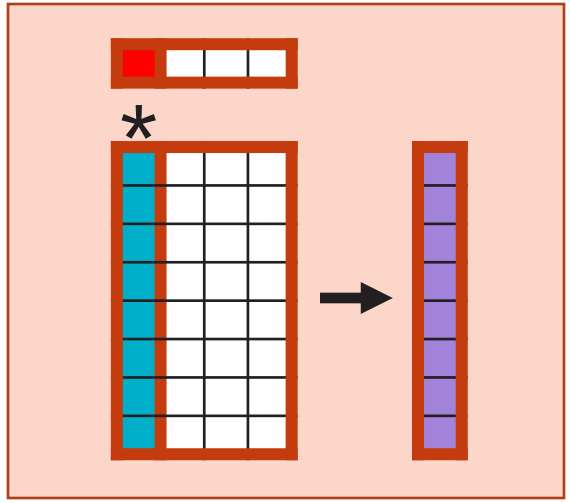
$\text{tmp}_i = A_{i,12}x_{12},$
$i = 16:23$



PE (3,2)

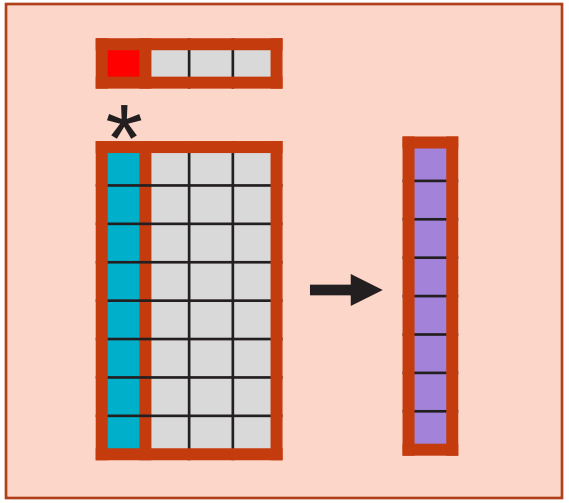


$\text{tmp}_i = A_{i,0}x_0,$
$i = 24:31$



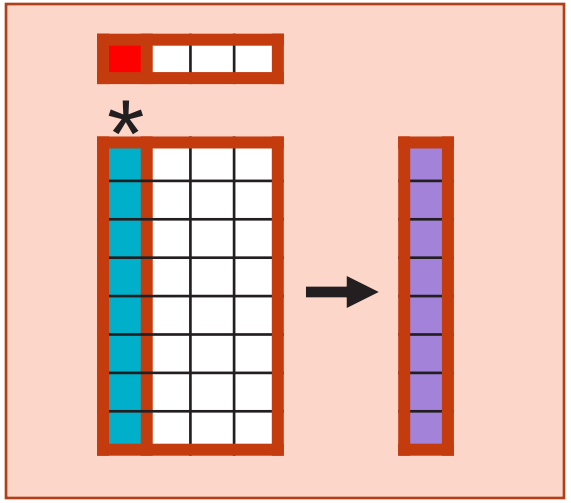
PE (0,3)

$\text{tmp}_i = A_{i,4}x_4,$
$i = 24:31$



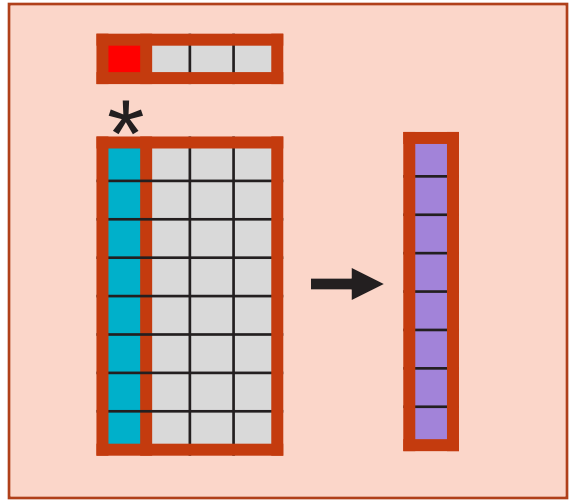
PE (1,3)

$\text{tmp}_i = A_{i,8}x_8,$
$i = 24:31$



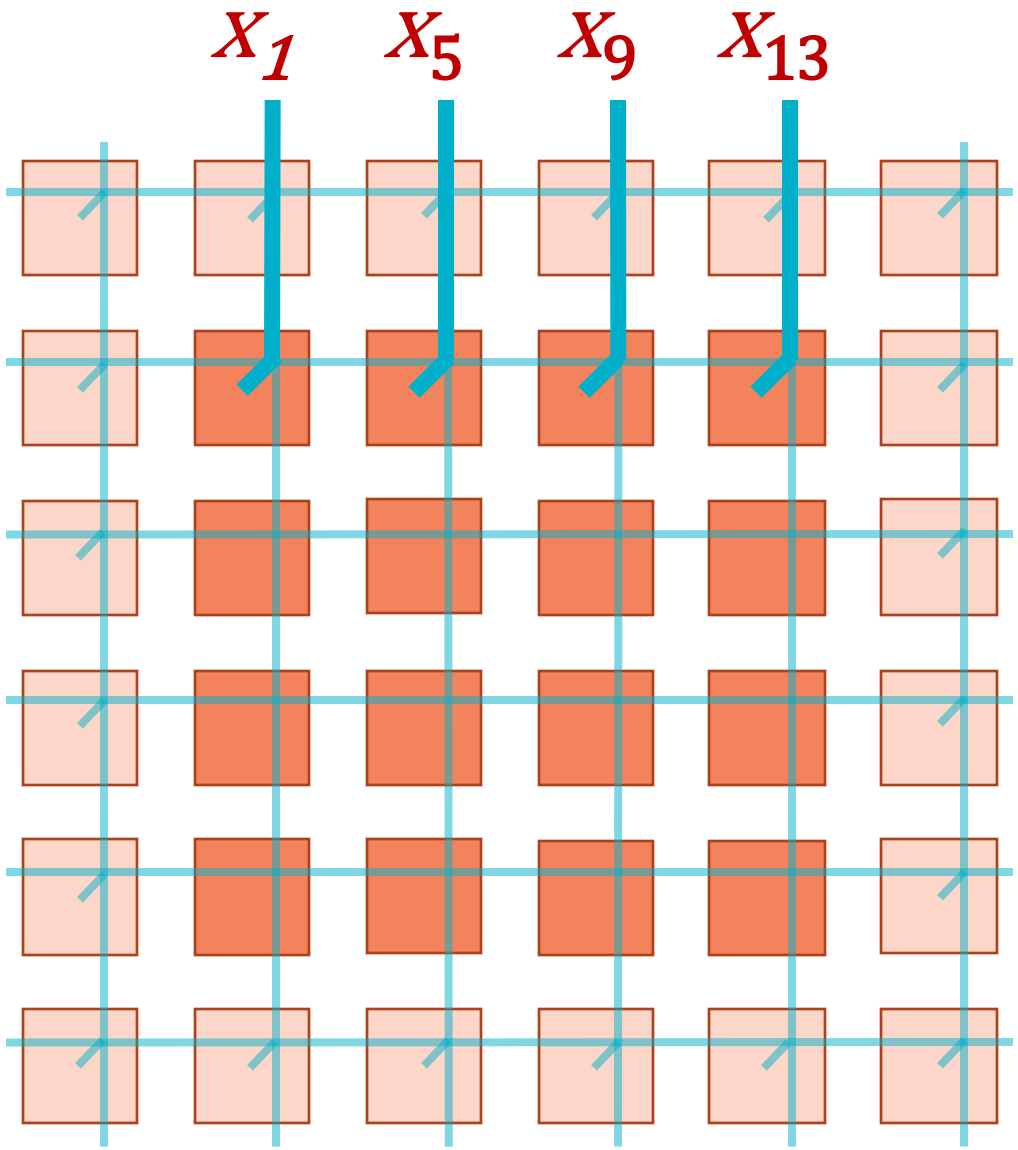
PE (2,3)

$\text{tmp}_i = A_{i,12}x_{12},$
$i = 24:31$



PE (3,3)

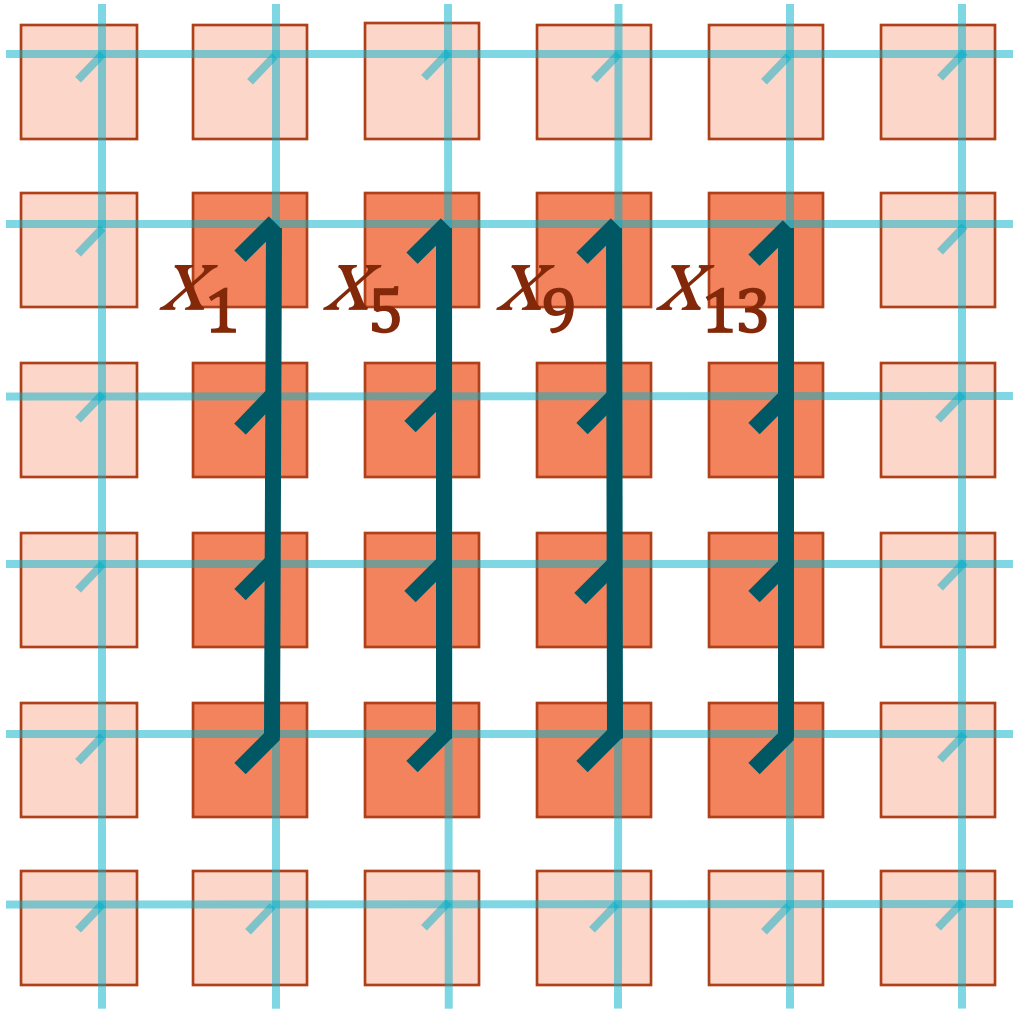
Device



5. Host streams in 1st, 5th, 9th, 13th elements of X into PEs (0,0), (1,0), (2,0), (3,0), respectively

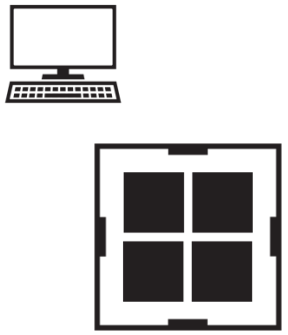
Host

Device

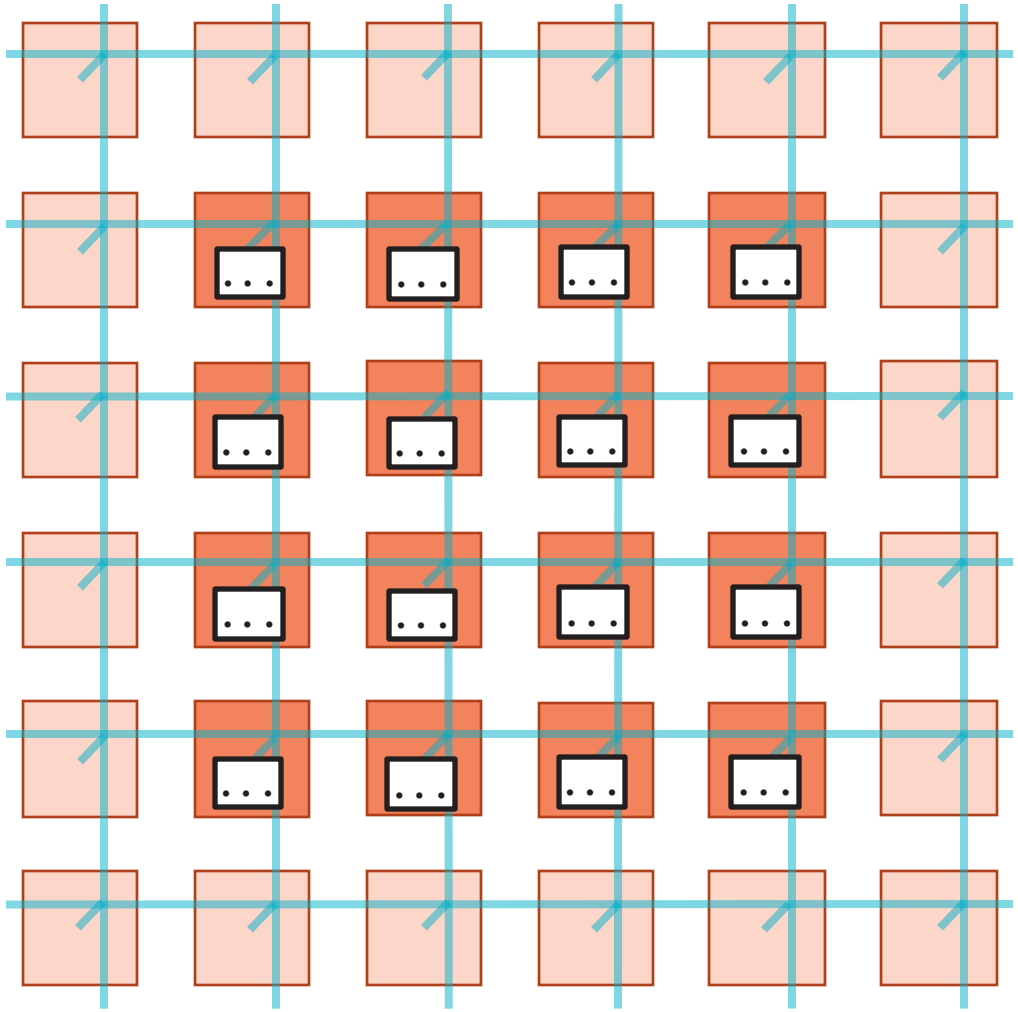


6. Activated task forward elements to PEs (0,*i*), (1,*i*), (2,*i*), (3,*i*)

Host



Device



7. Device computes for $j = 1, 5, 9, 13$:

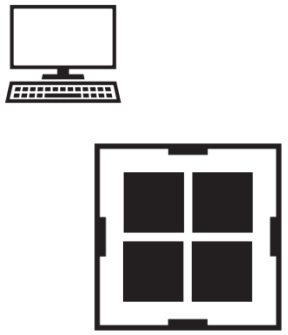
$\text{tmp}_i = A_{i,j}x_j,\ i = 0{:}7$ **on 0th row PEs**

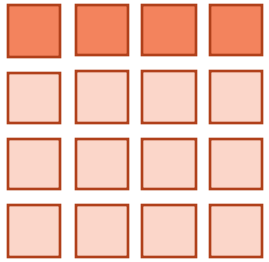
$\text{tmp}_i = A_{i,j}x_j,\ i = 8{:}15$ **on 1st row PEs**

$\text{tmp}_i = A_{i,j}x_j,\ i = 16{:}23$ **on 2nd row PEs**

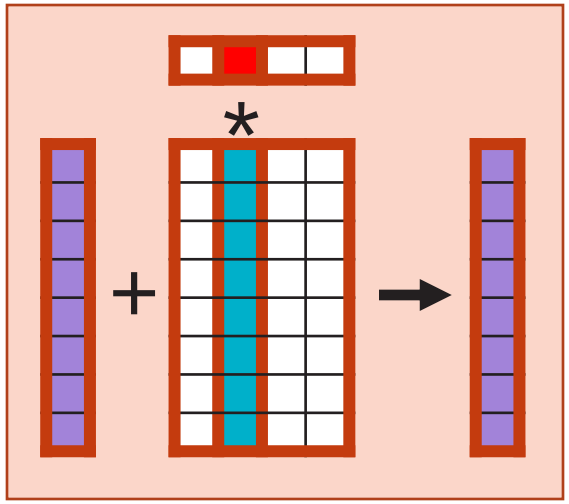
$\text{tmp}_i = A_{i,j}x_j,\ i = 24{:}31$ **on 3rd row PEs**

Host



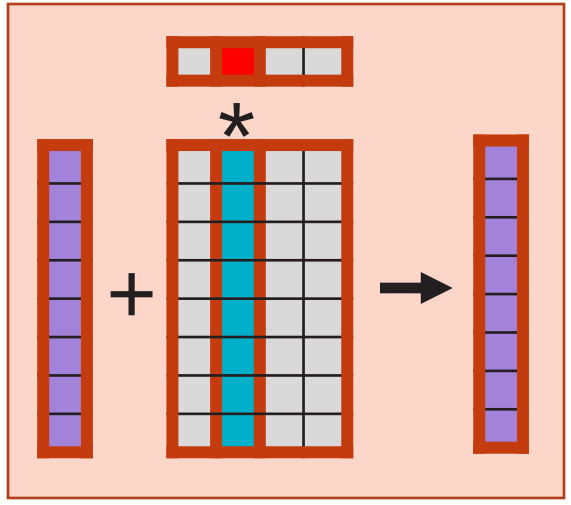


$\text{tmp}_i \mathrel{+}= A_{i,1}x_1,$
$i = 0:7$



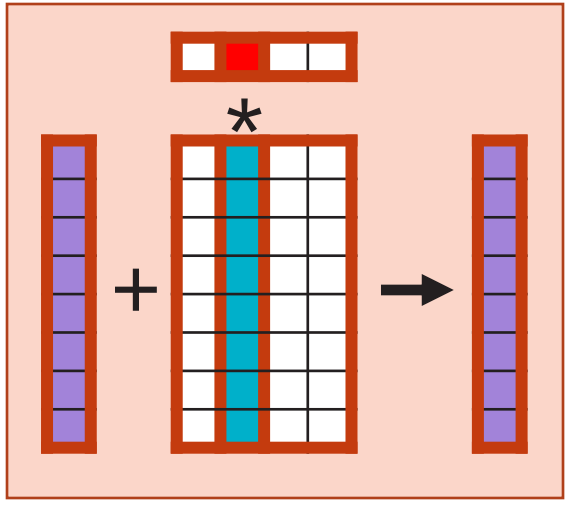
PE (0,0)

$\text{tmp}_i \mathrel{+}= A_{i,5}x_5,$
$i = 0:7$



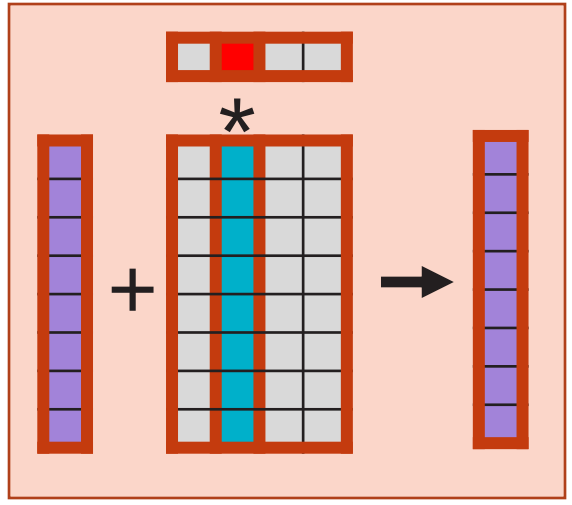
PE (1,0)

$\text{tmp}_i \mathrel{+}= A_{i,9}x_9,$
$i = 0:7$

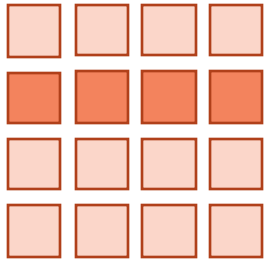


PE (2,0)

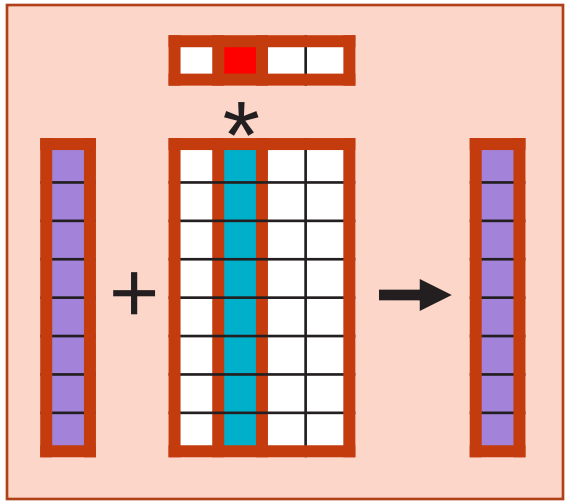
$\text{tmp}_i \mathrel{+}= A_{i,13}x_{13},$
$i = 0:7$



PE (3,0)

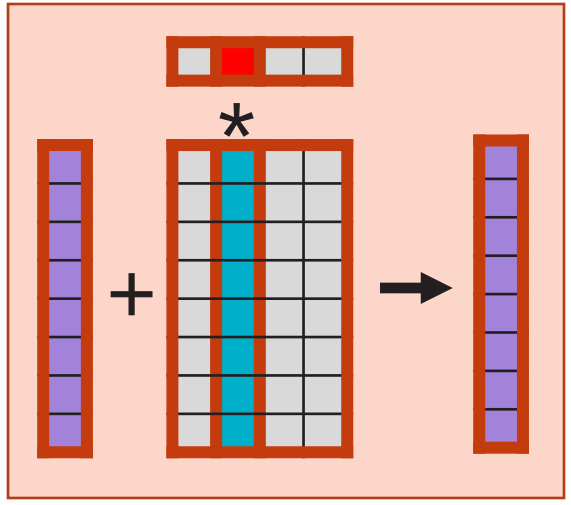


$\text{tmp}_i \mathrel{+}= A_{i,1}x_1,$
$i = 8:15$



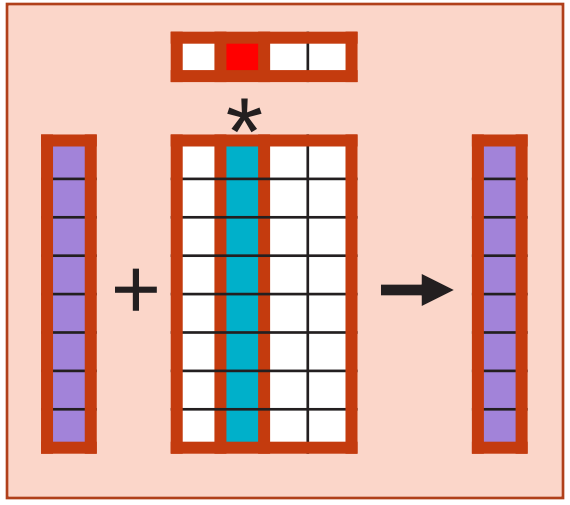
PE (0,1)

$\text{tmp}_i \mathrel{+}= A_{i,5}x_5,$
$i = 8:15$



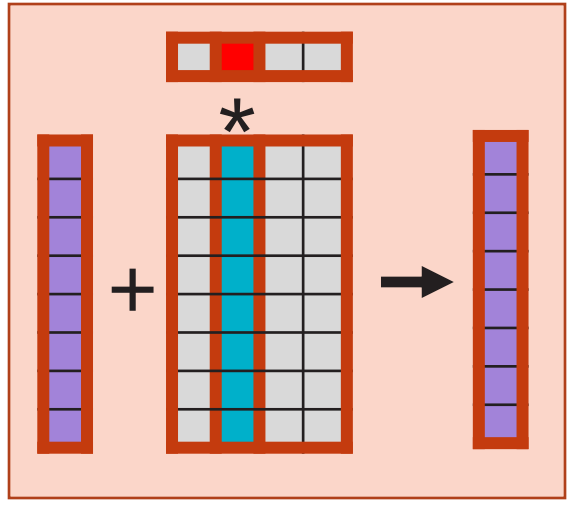
PE (1,1)

$\text{tmp}_i \mathrel{+}= A_{i,9}x_9,$
$i = 8:15$

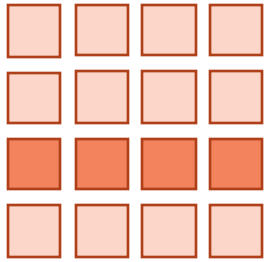


PE (2,1)

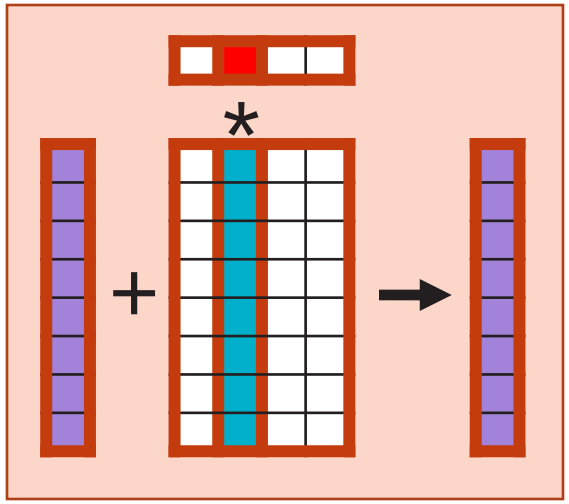
$\text{tmp}_i \mathrel{+}= A_{i,13}x_{13},$
$i = 8:15$



PE (3,1)

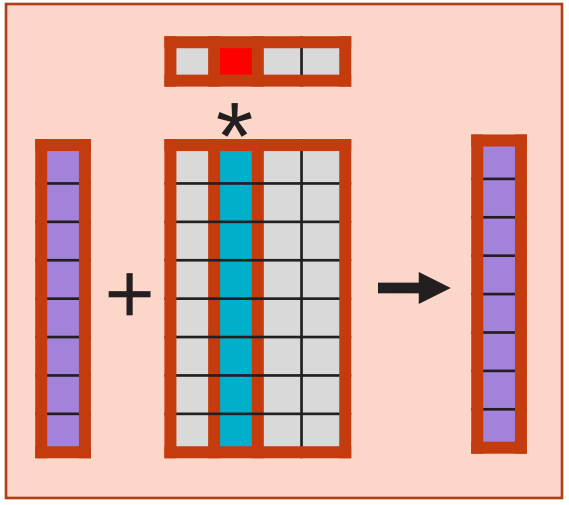


$\text{tmp}_i \mathrel{+}= A_{i,1}x_1,$
$i = 16:23$



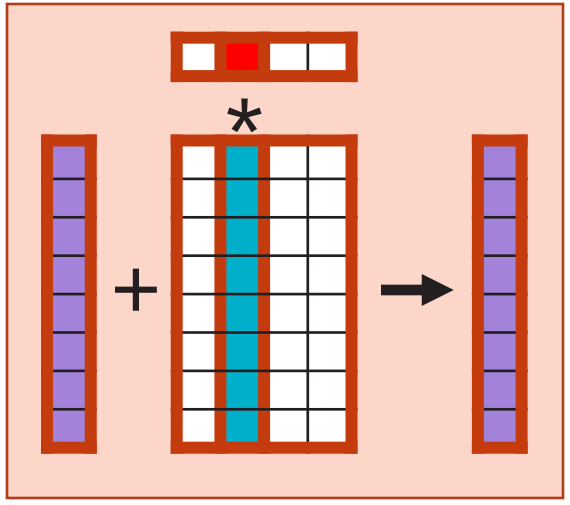
PE (0,2)

$\text{tmp}_i \mathrel{+}= A_{i,5}x_5,$
$i = 16:23$



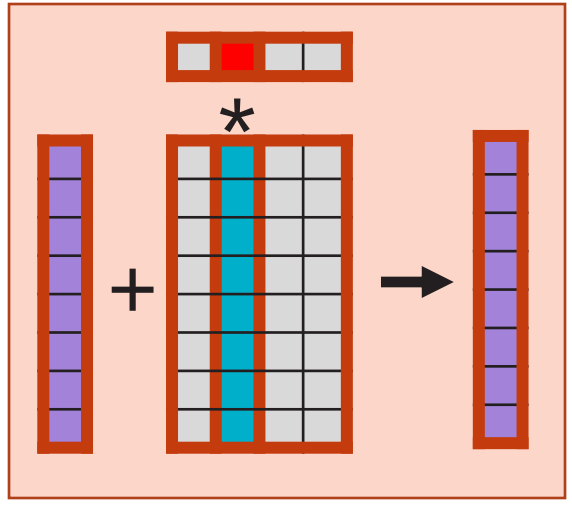
PE (1,2)

$\text{tmp}_i \mathrel{+}= A_{i,9}x_9,$
$i = 16:23$

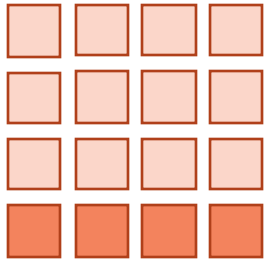


PE (2,2)

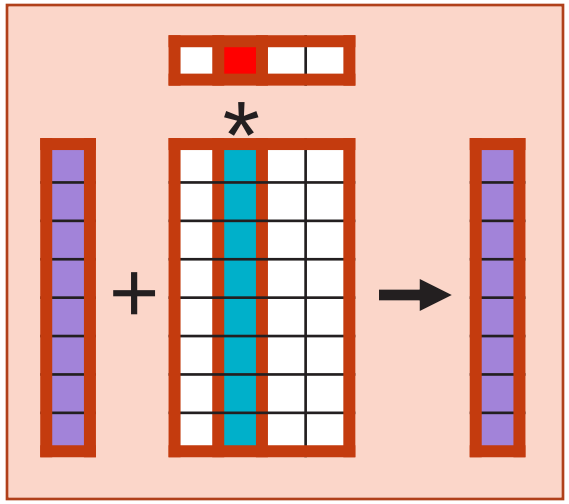
$\text{tmp}_i \mathrel{+}= A_{i,13}x_{13},$
$i = 16:23$



PE (3,2)

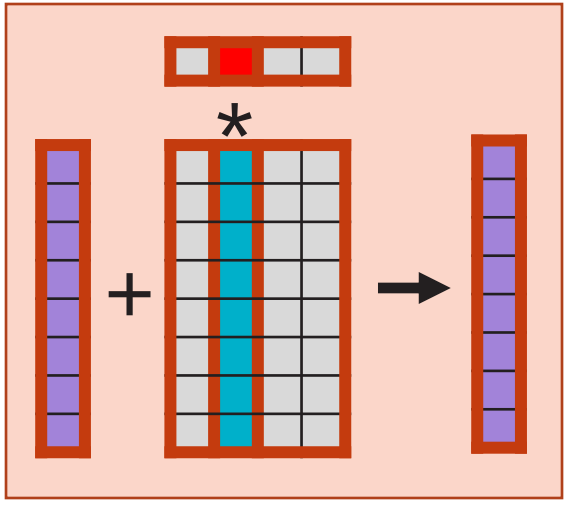


$\text{tmp}_i \mathrel{+}= A_{i,1}x_1,$
$i = 24:31$



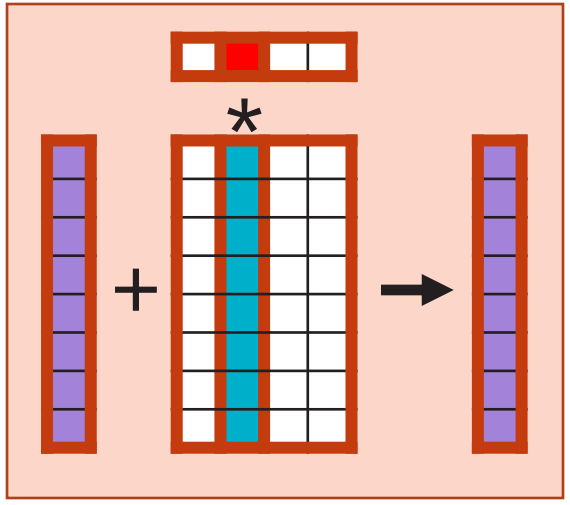
PE (0,3)

$\text{tmp}_i \mathrel{+}= A_{i,5}x_5,$
$i = 24:31$



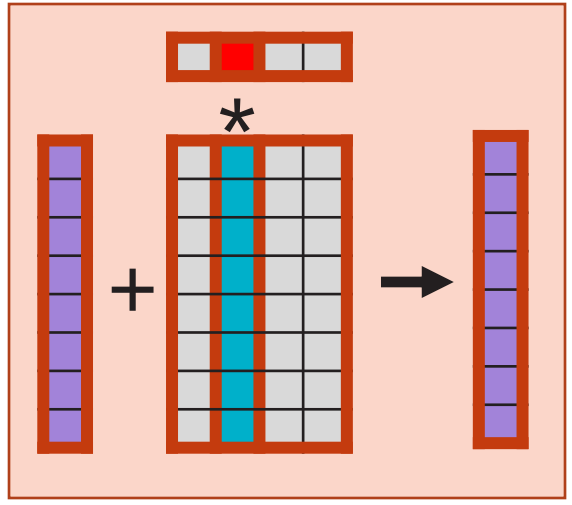
PE (1,3)

$\text{tmp}_i \mathrel{+}= A_{i,9}x_9,$
$i = 24:31$



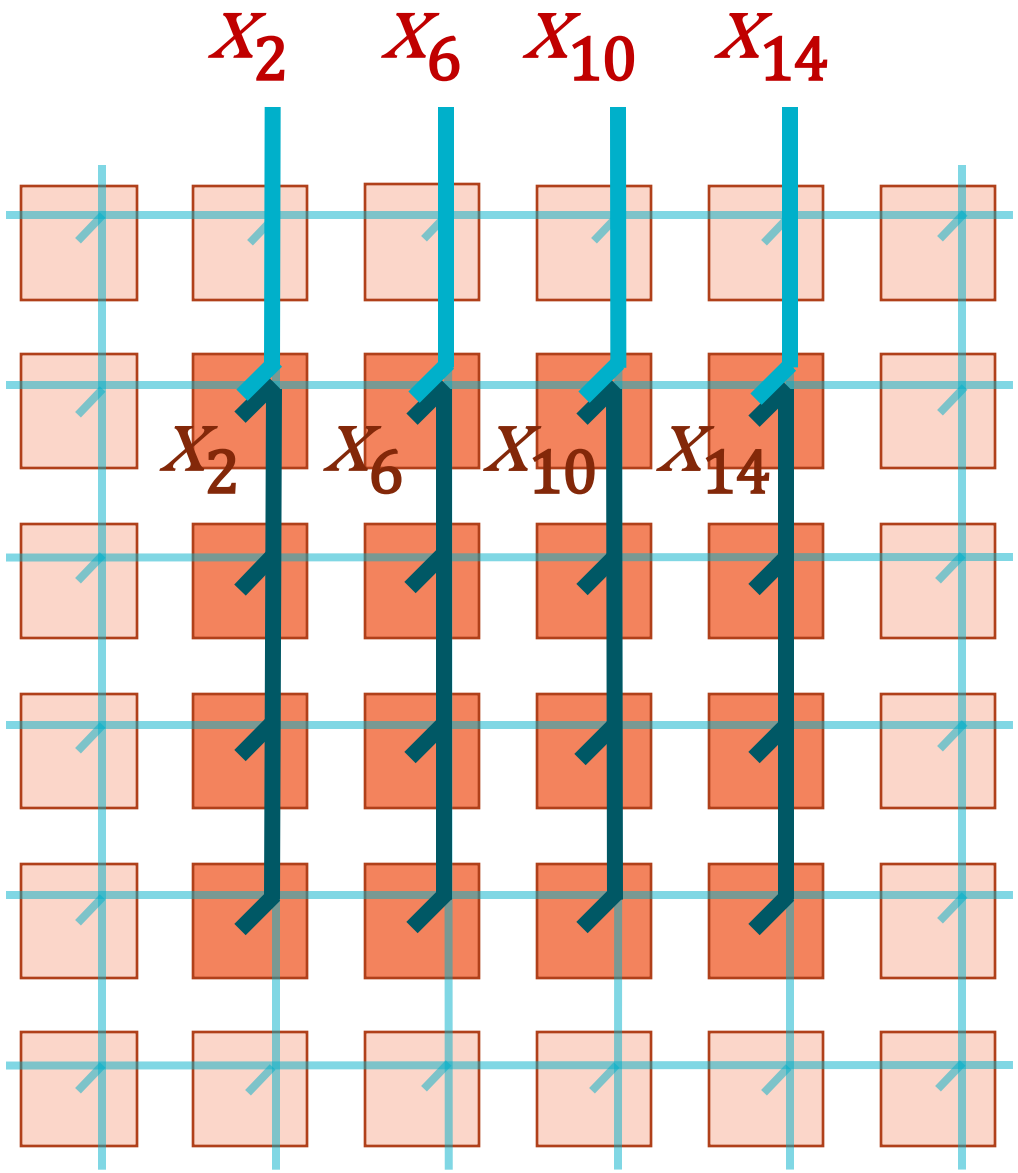
PE (2,3)

$\text{tmp}_i \mathrel{+}= A_{i,13}x_{13},$
$i = 24:31$



PE (3,3)

Device

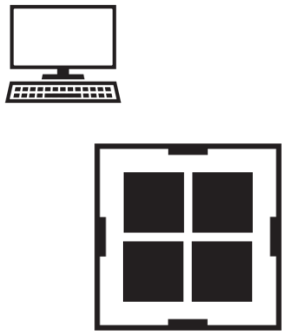


8. Host streams in X_2, X_6, X_{10}, X_{14}

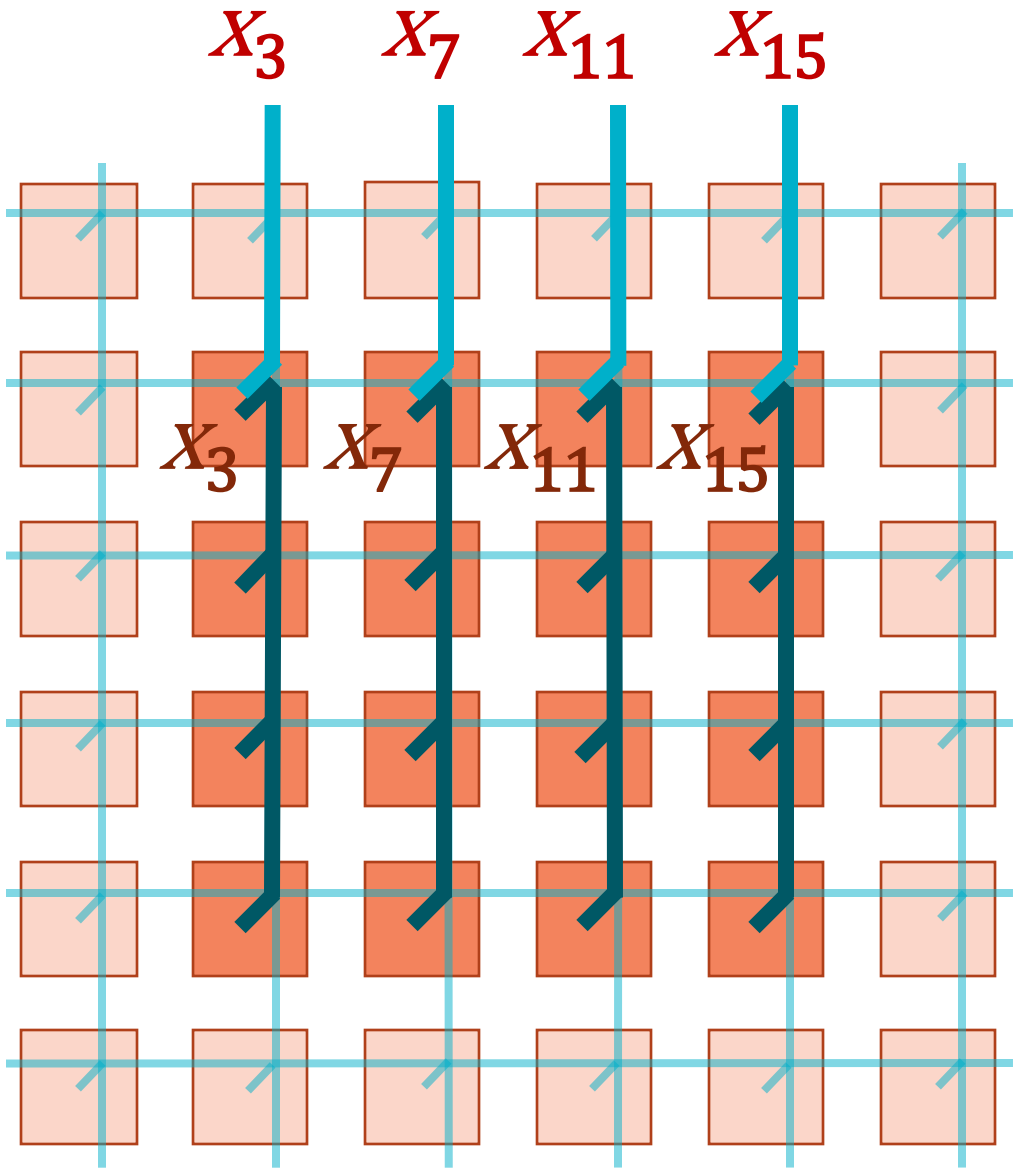
9. Task forwards elements South

10. Device computes contributions to tmp

Host

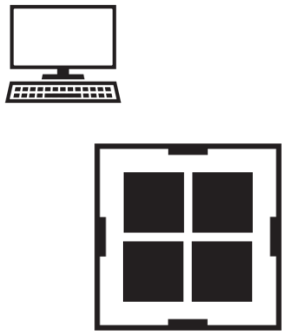


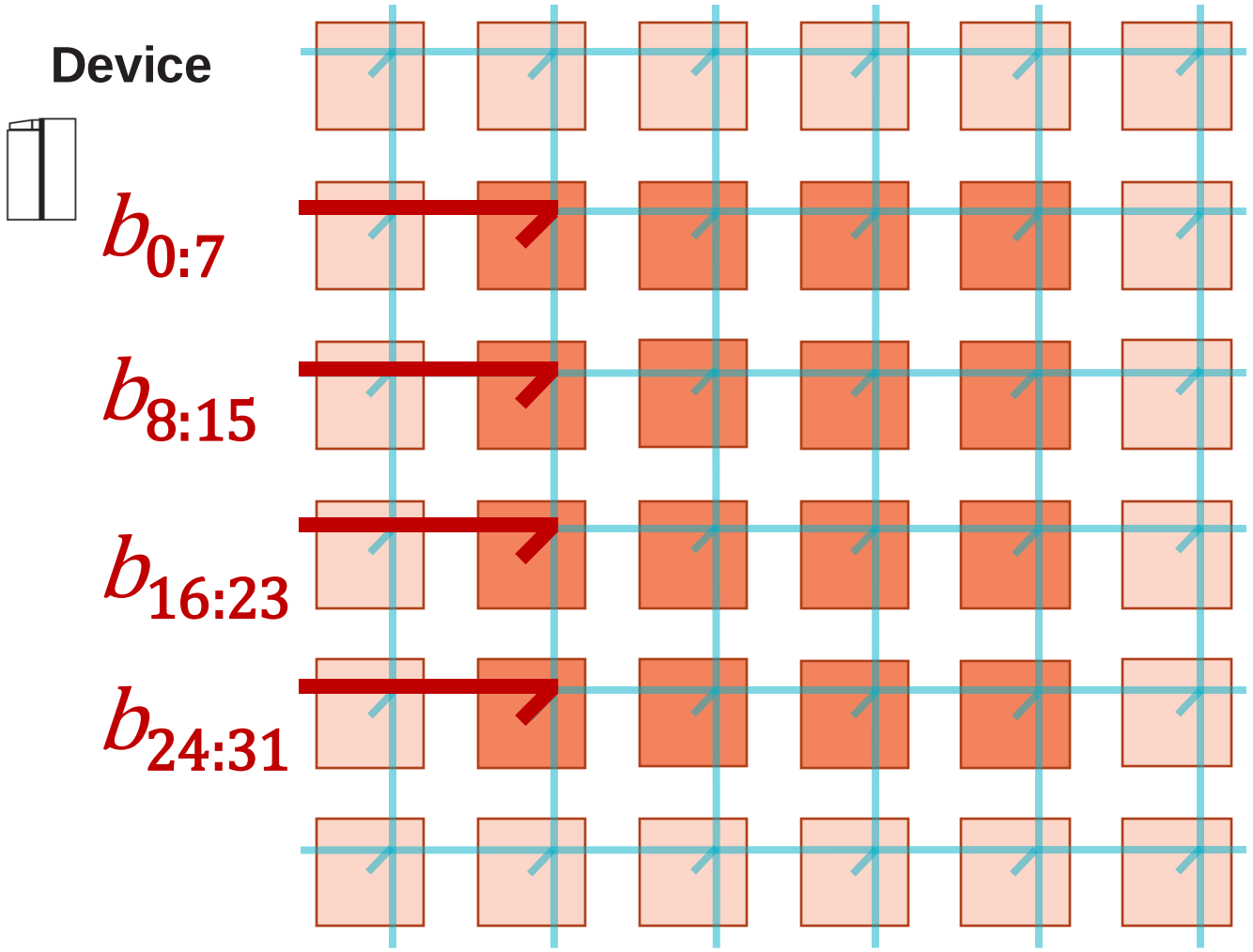
Device



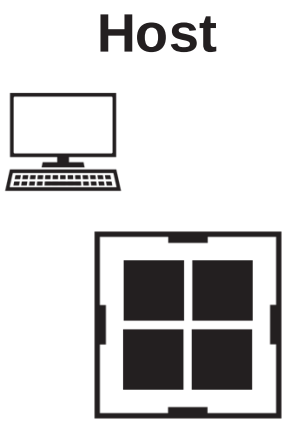
11. Host streams in X_3, X_7, X_{11}, X_{15}
12. Task forwards elements South
13. Device computes contributions to tmp

Host

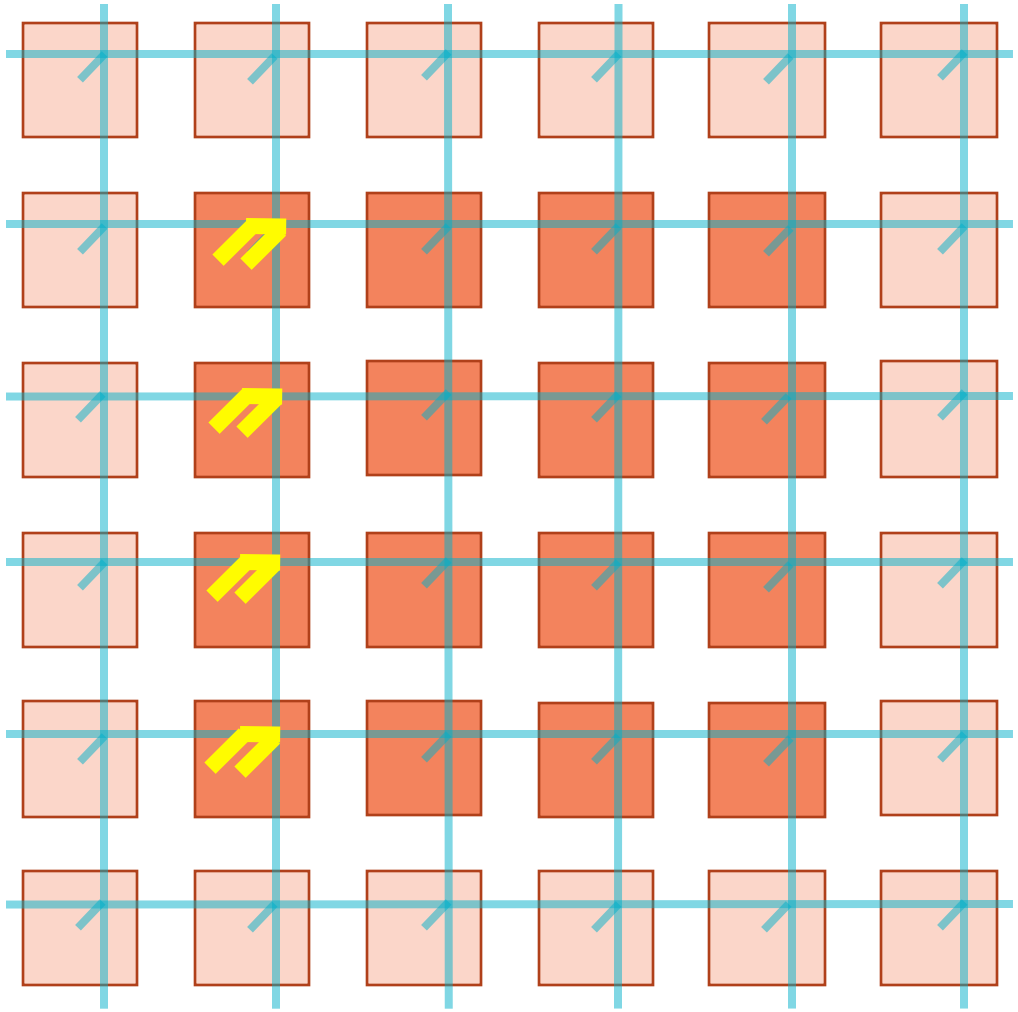




14. Host streams in elements of b in equal chunks to PEs (0,0), (0,1), (0,2), (0,3), respectively



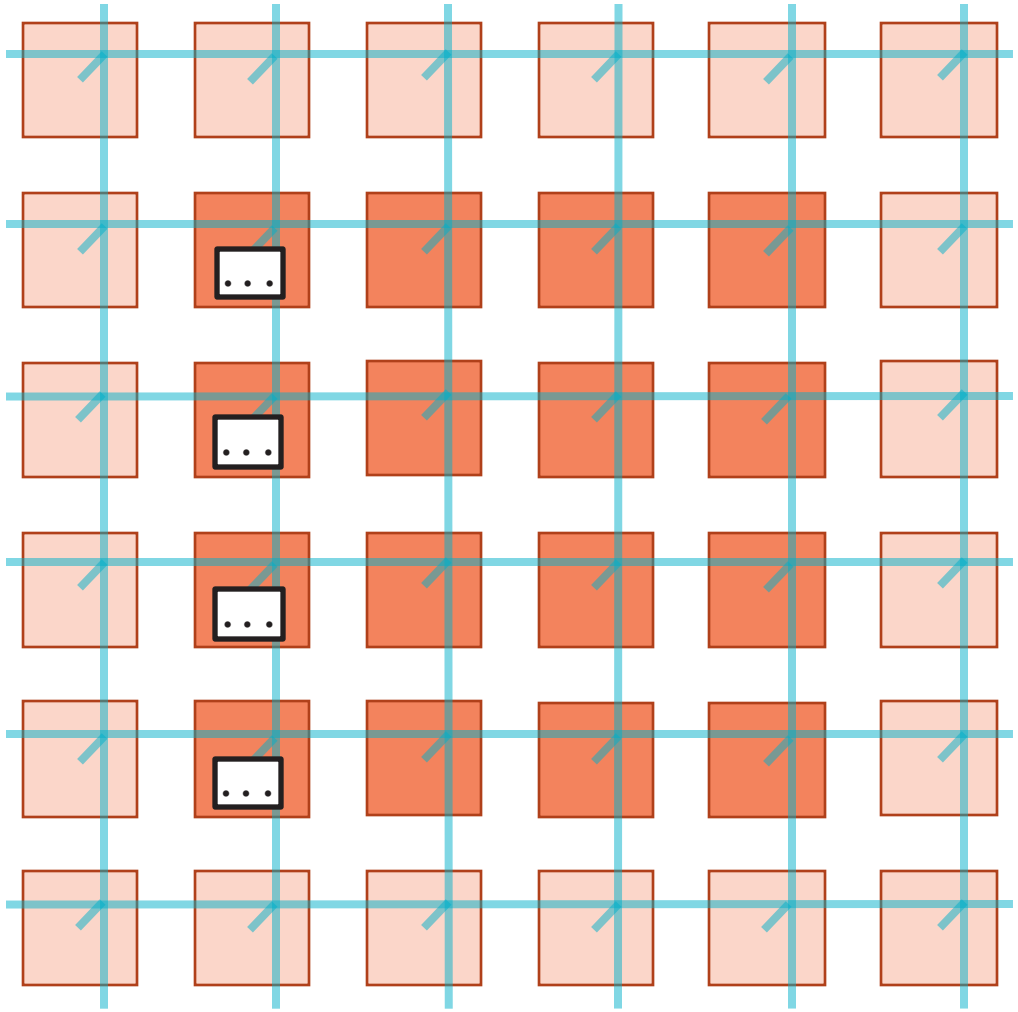
Device



15. Send values along RAMP → RAMP routing to be received from fabric in compute task

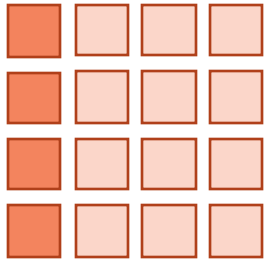
Host

Device

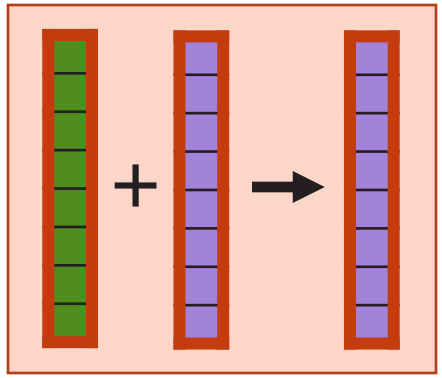


16. When Ax chunk and b values are available, PEs along left edge calculate contribution of b to tmp

Host

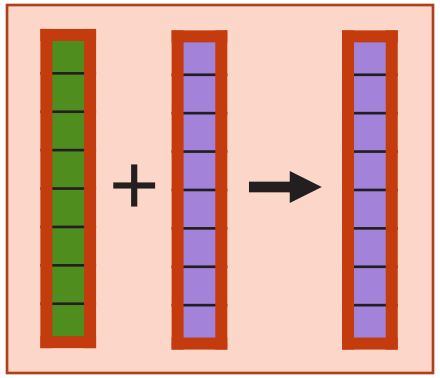


$\text{tmp}_i \mathrel{+}= b_i,$
$i = 0 : 7$



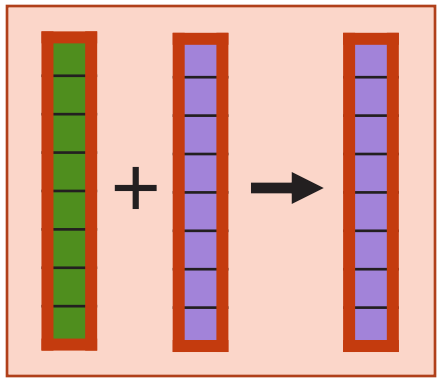
PE (0,0)

$\text{tmp}_i \mathrel{+}= b_i,$
$i = 8 : 15$



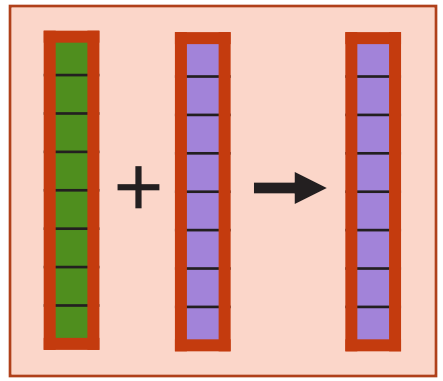
PE (0,1)

$\text{tmp}_i \mathrel{+}= b_i,$
$i = 16 : 23$



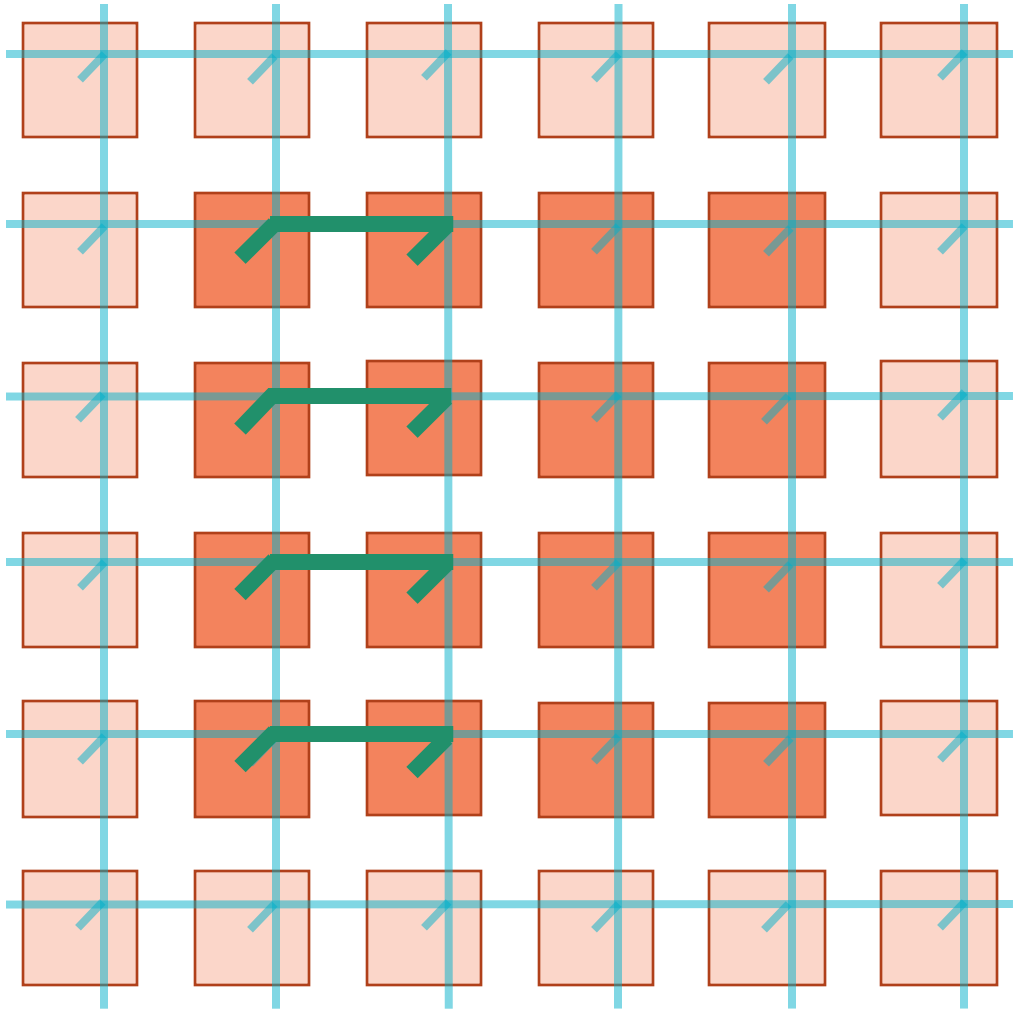
PE (0,2)

$\text{tmp}_i \mathrel{+}= b_i,$
$i = 24 : 31$



PE (0,3)

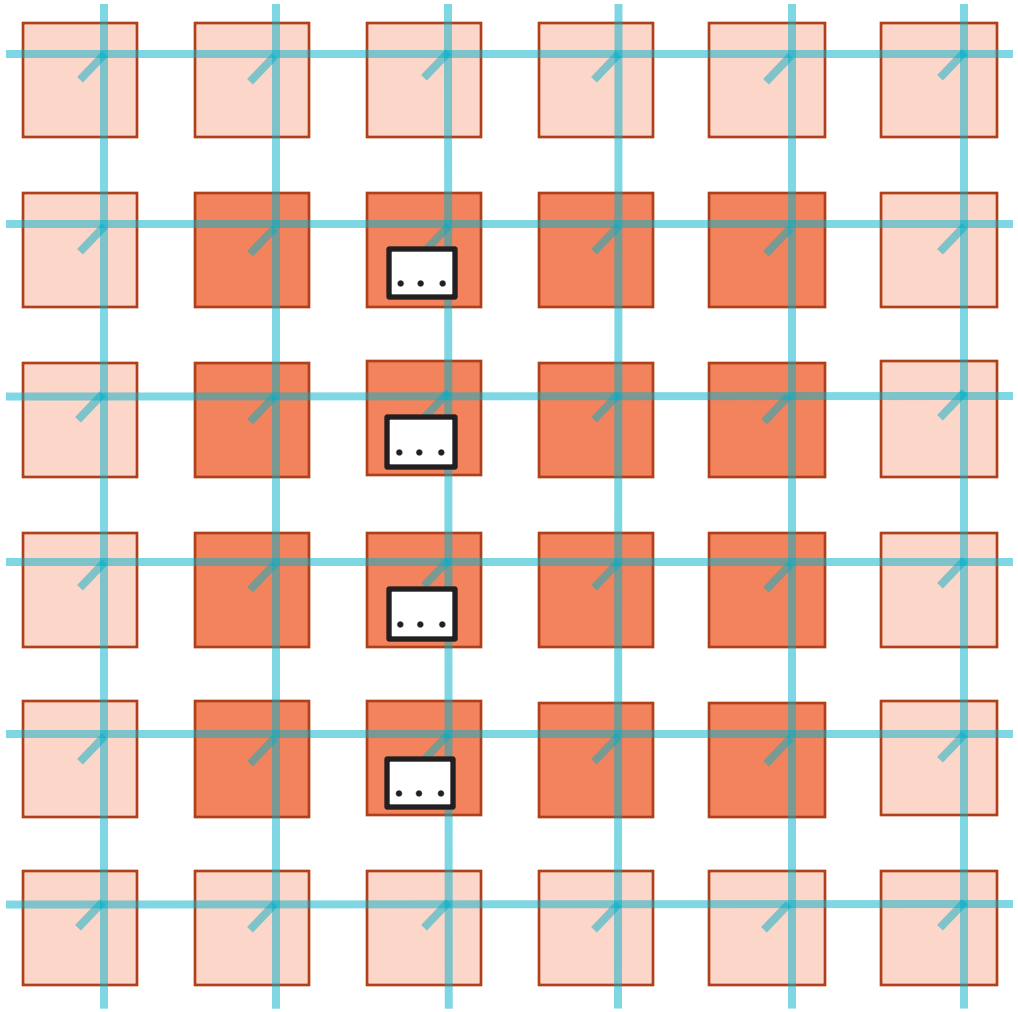
Device



17. PEs along left edge send their running sum tmp to the East

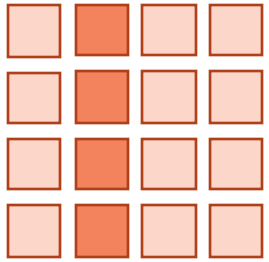
Host

Device



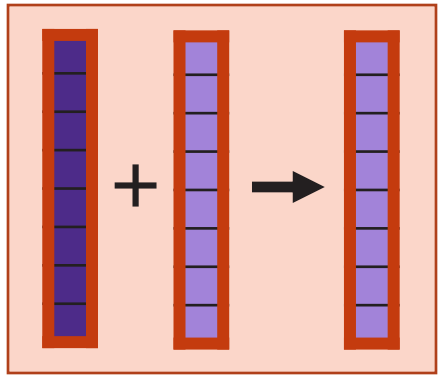
18. PEs in column 1 accumulate contribution to tmp sent by PEs from column 0

Host



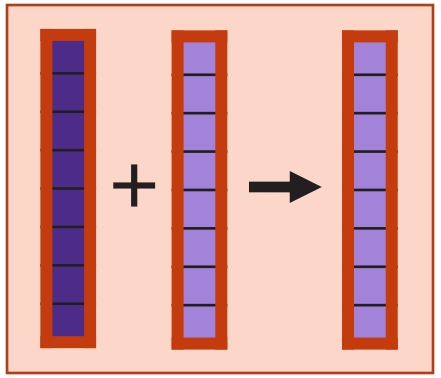
Let u_i be values sent in from West.

$\text{tmp}_i \mathrel{+}= ui,$
$i = 0 : 7$



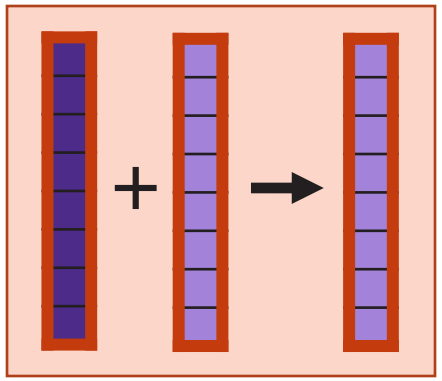
PE (1,0)

$\text{tmp}_i \mathrel{+}= ui,$
$i = 8 : 15$



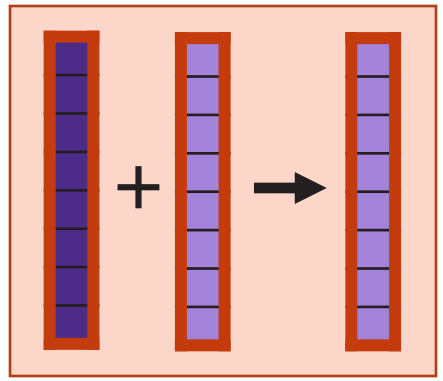
PE (1,1)

$\text{tmp}_i \mathrel{+}= ui,$
$i = 16 : 23$



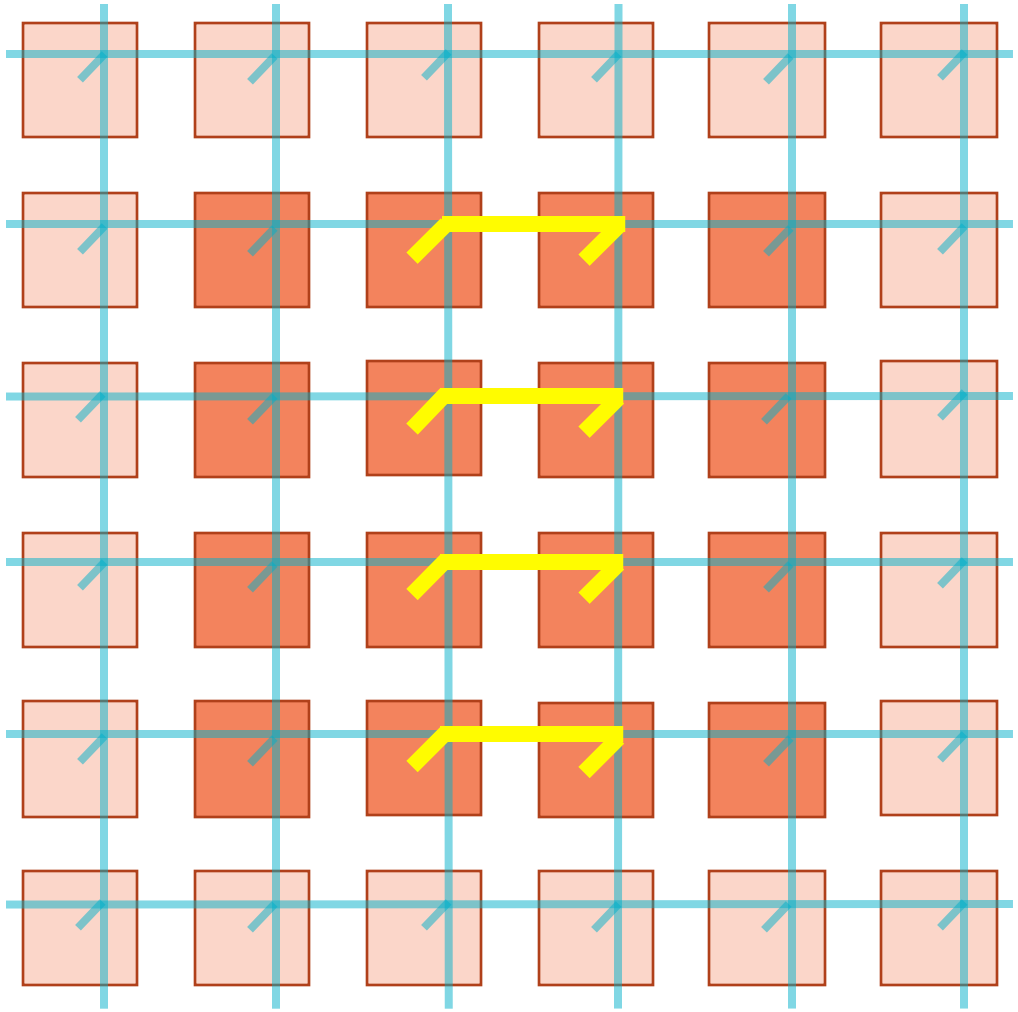
PE (1,2)

$\text{tmp}_i \mathrel{+}= ui,$
$i = 24 : 31$



PE (1,3)

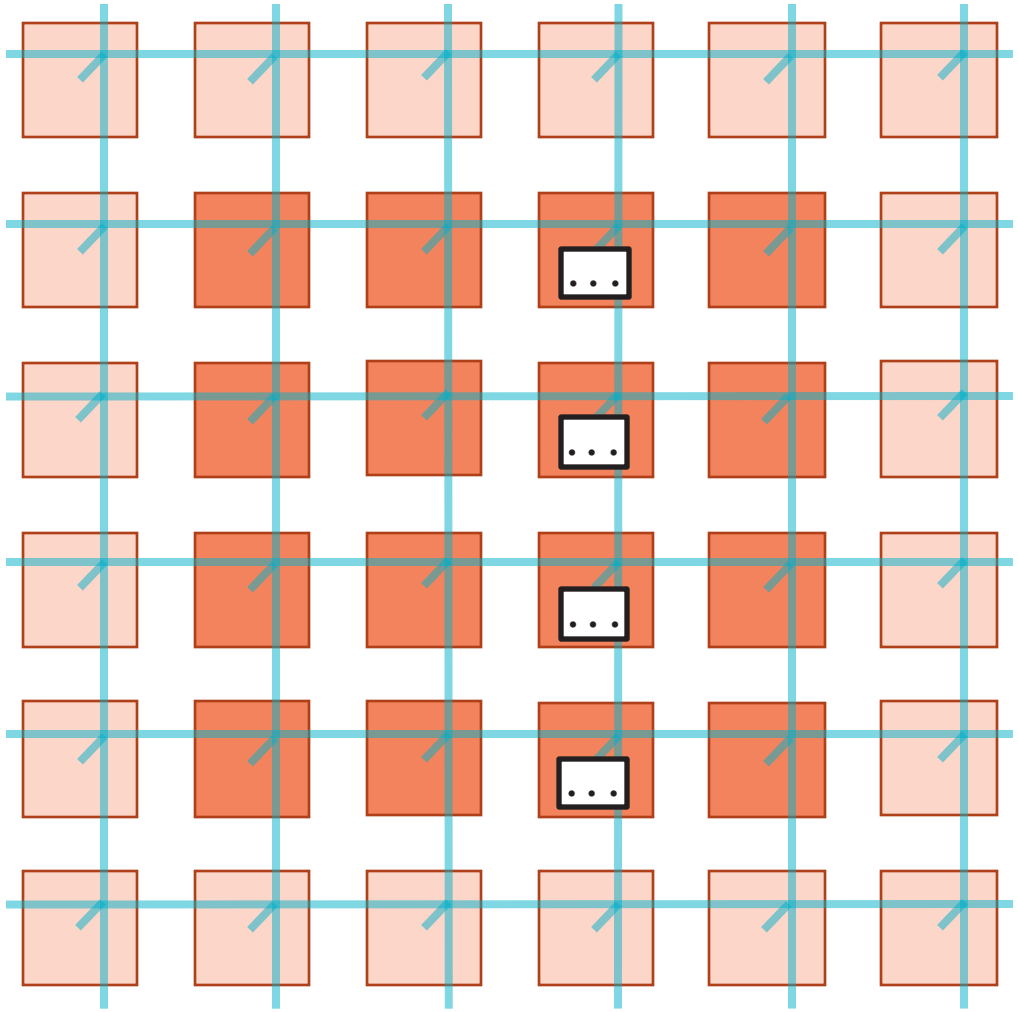
Device



19. PEs in column 1 send their running sum tmp to the East

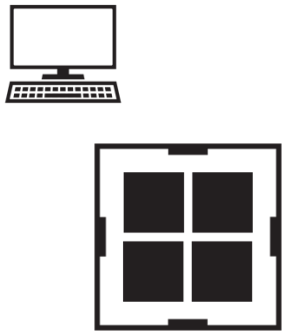
Host

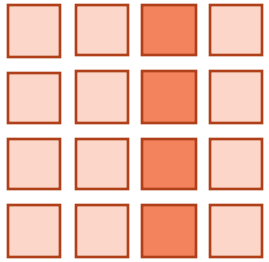
Device



20. PEs in column 2 accumulate contribution to tmp sent by PEs from column 1

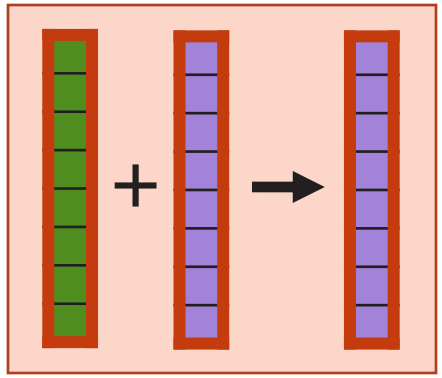
Host





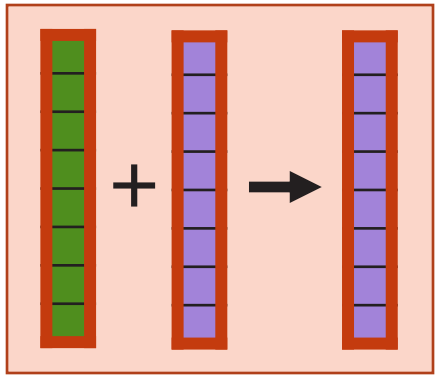
Let u_i be values sent in from West.

$\text{tmp}_i \mathrel{+}= ui,$
$i = 0:7$



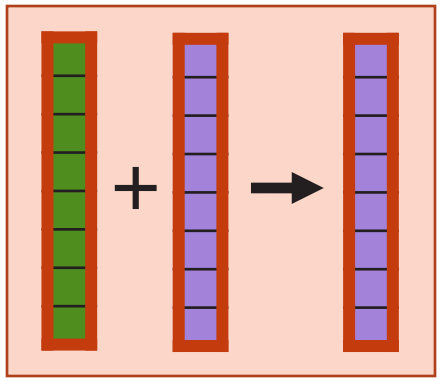
PE (2,0)

$\text{tmp}_i \mathrel{+}= ui,$
$i = 8:15$



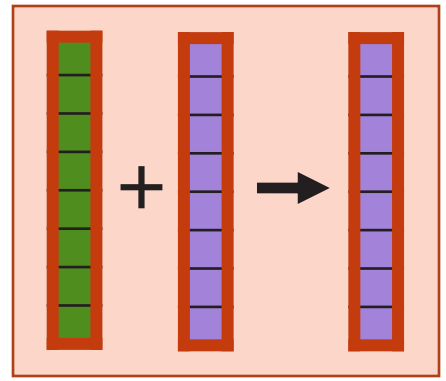
PE (2,1)

$\text{tmp}_i \mathrel{+}= ui,$
$i = 16:23$



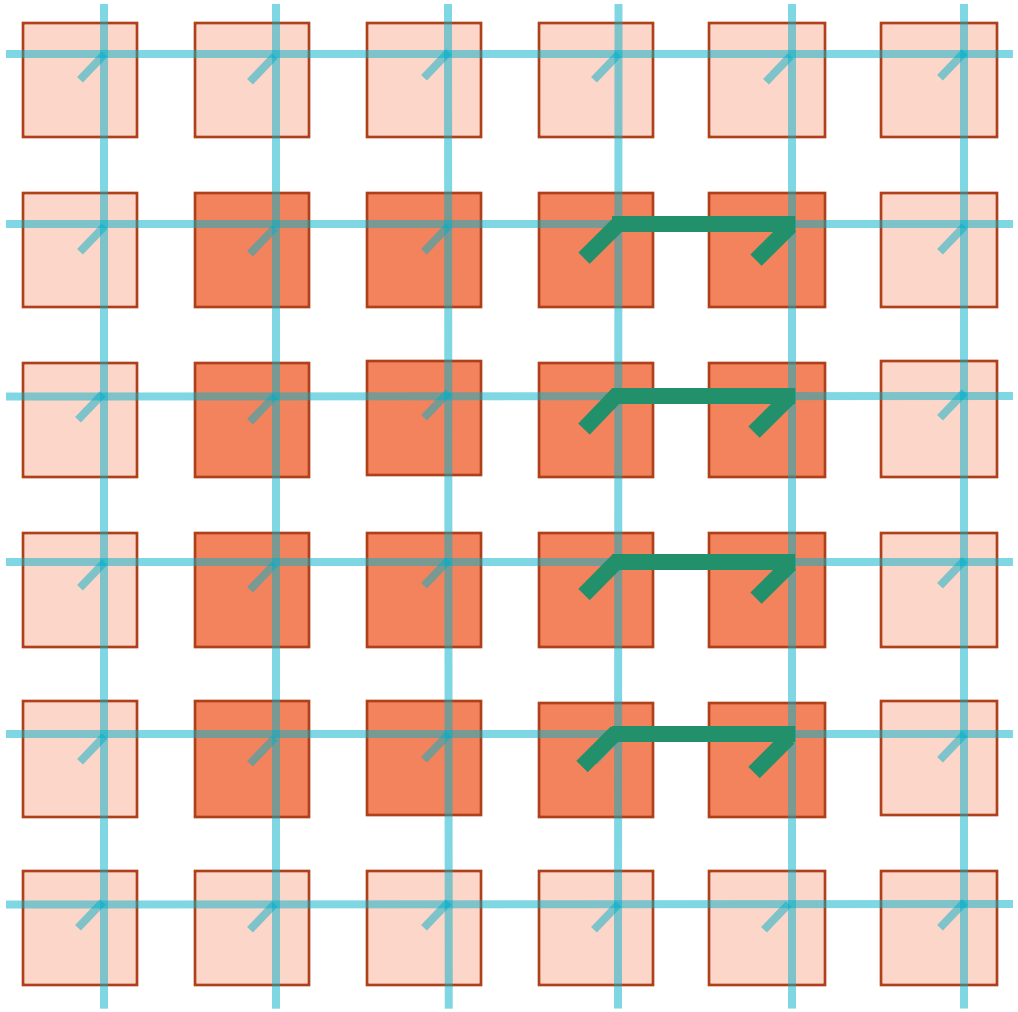
PE (2,2)

$\text{tmp}_i \mathrel{+}= ui,$
$i = 24:31$



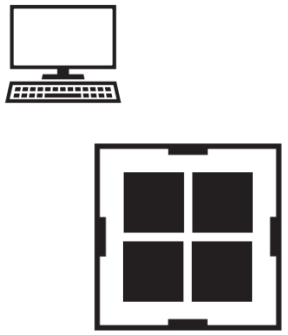
PE (2,3)

Device

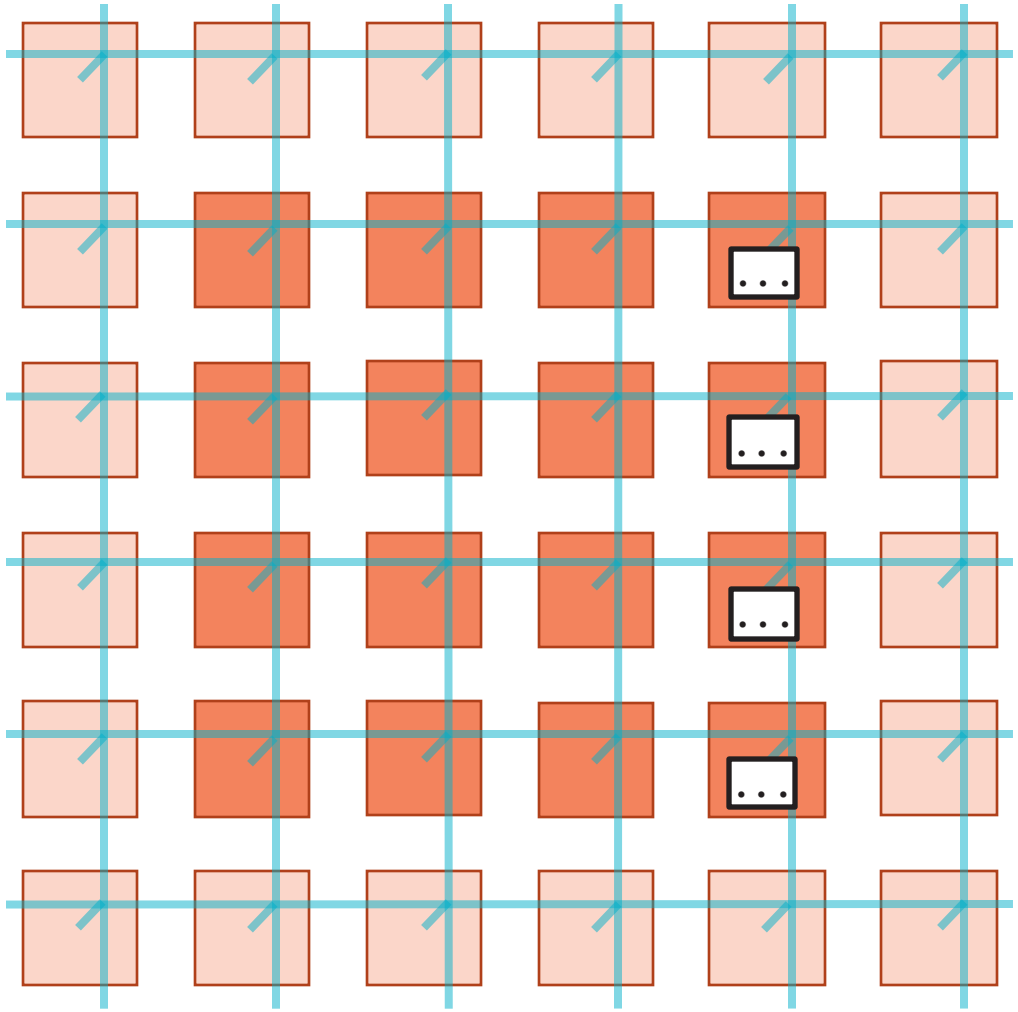


21. PEs in column 2 send their running sum tmp to the East

Host

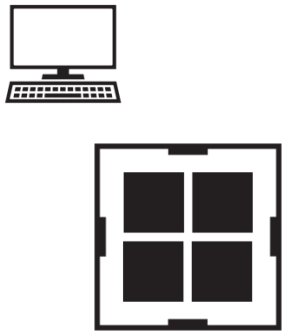


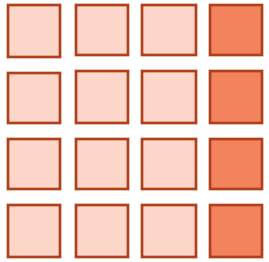
Device



22. PEs in column 3 accumulate contribution to tmp sent by PEs from column 2

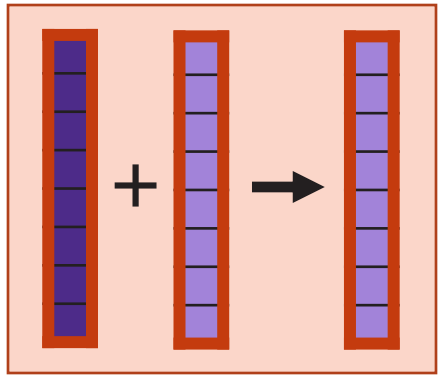
Host





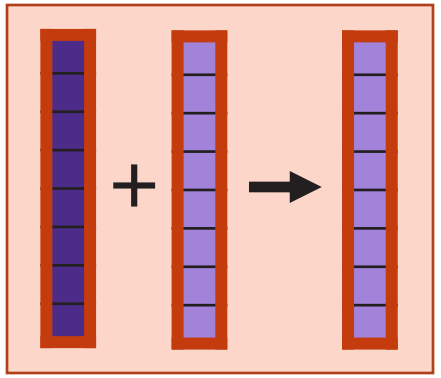
Let u_i be values sent in from West.
y_i is in same memory location as tmp_i.

$y_i = \mathrm{tmp}_i + ui,$
$i = 0 : 7$



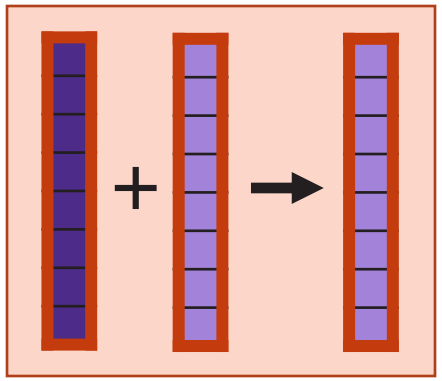
PE (3,0)

$y_i = \mathrm{tmp}_i + ui,$
$i = 8 : 15$



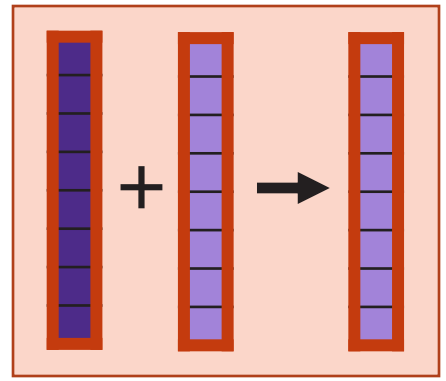
PE (3,1)

$y_i = \mathrm{tmp}_i + ui,$
$i = 16 : 23$



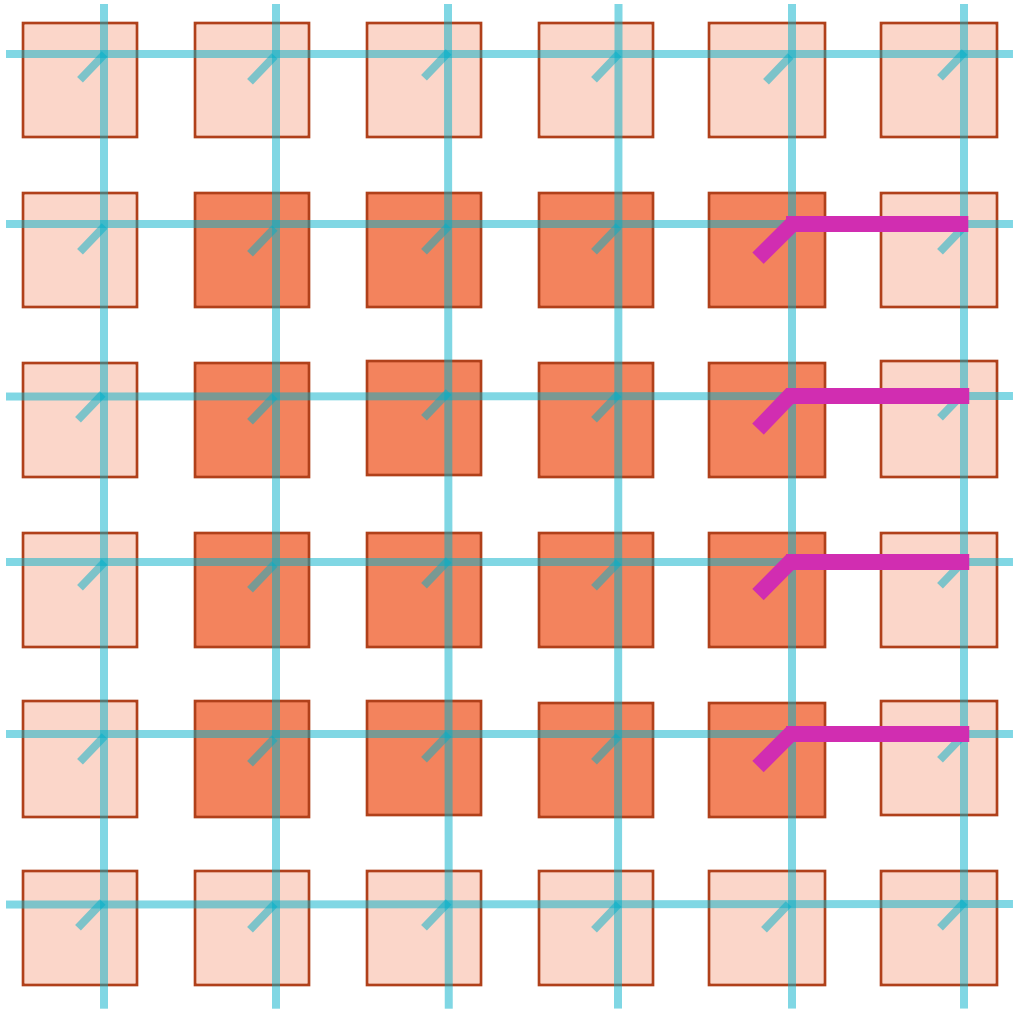
PE (3,2)

$y_i = \mathrm{tmp}_i + ui,$
$i = 24 : 31$



PE (3,3)

Device

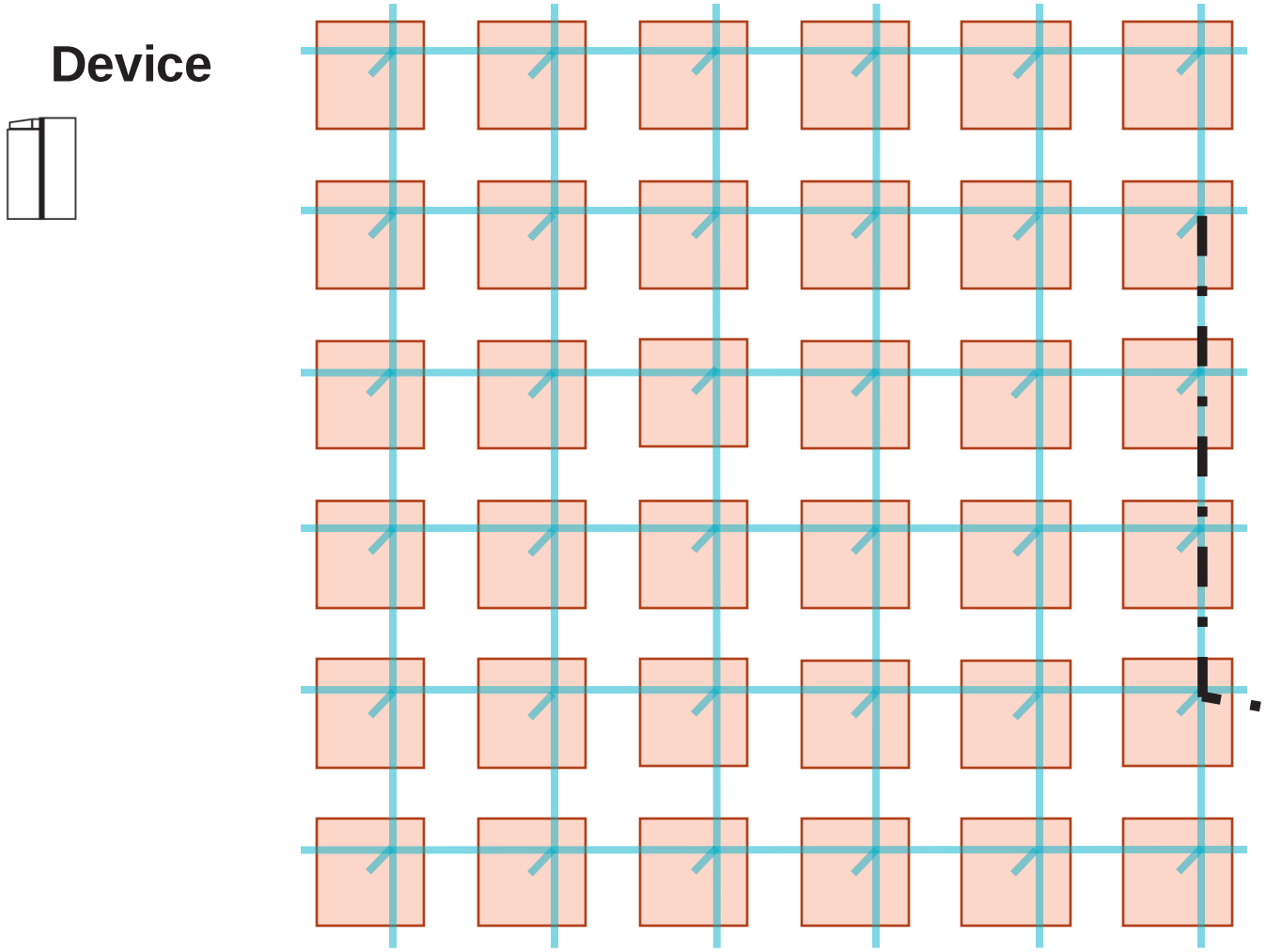


23. PEs along right edge send final sum y to the East

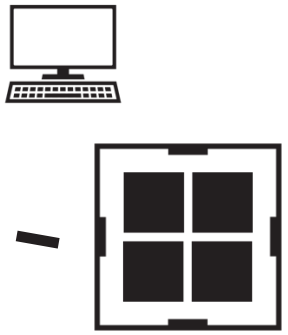
Host

24. Host receives y from fabric

Device

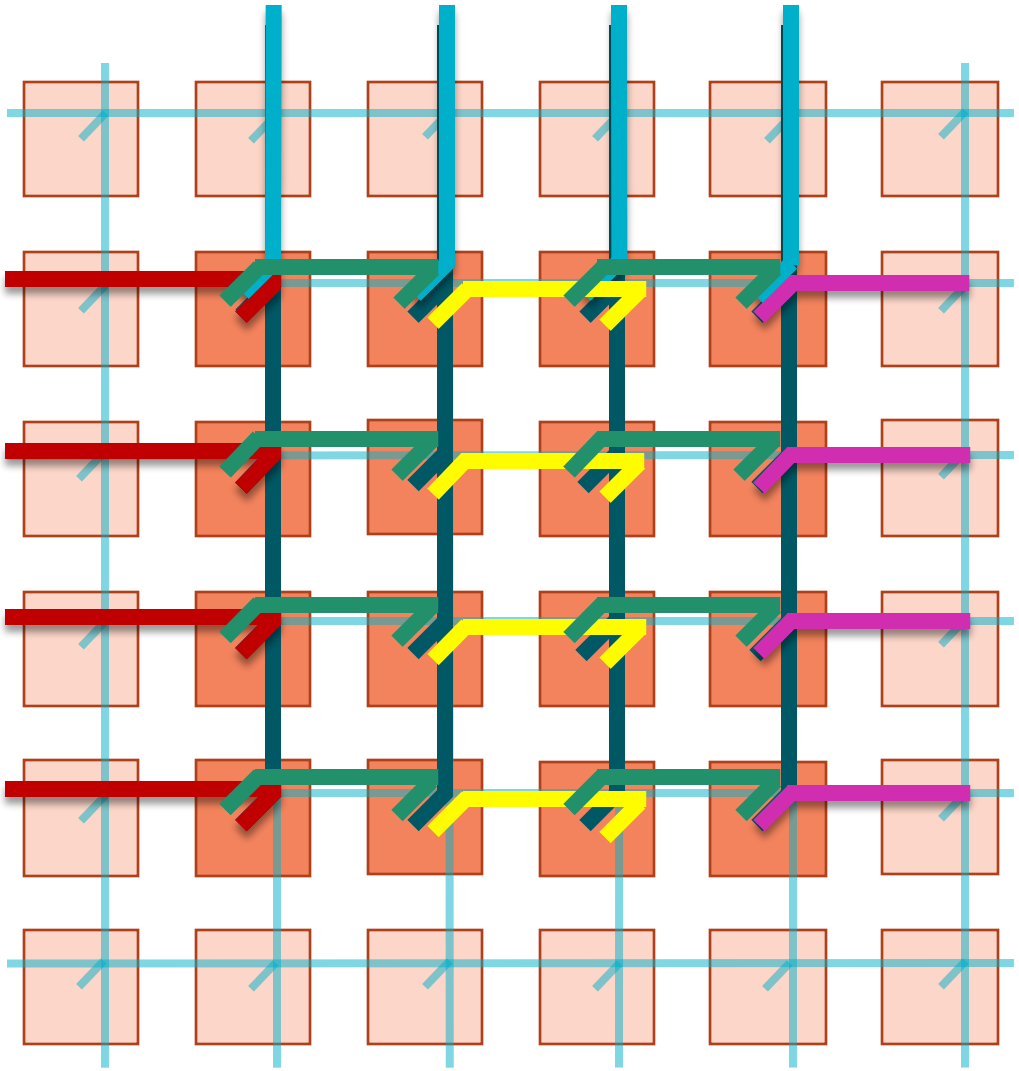


Host



GEMV Color Routes

Device

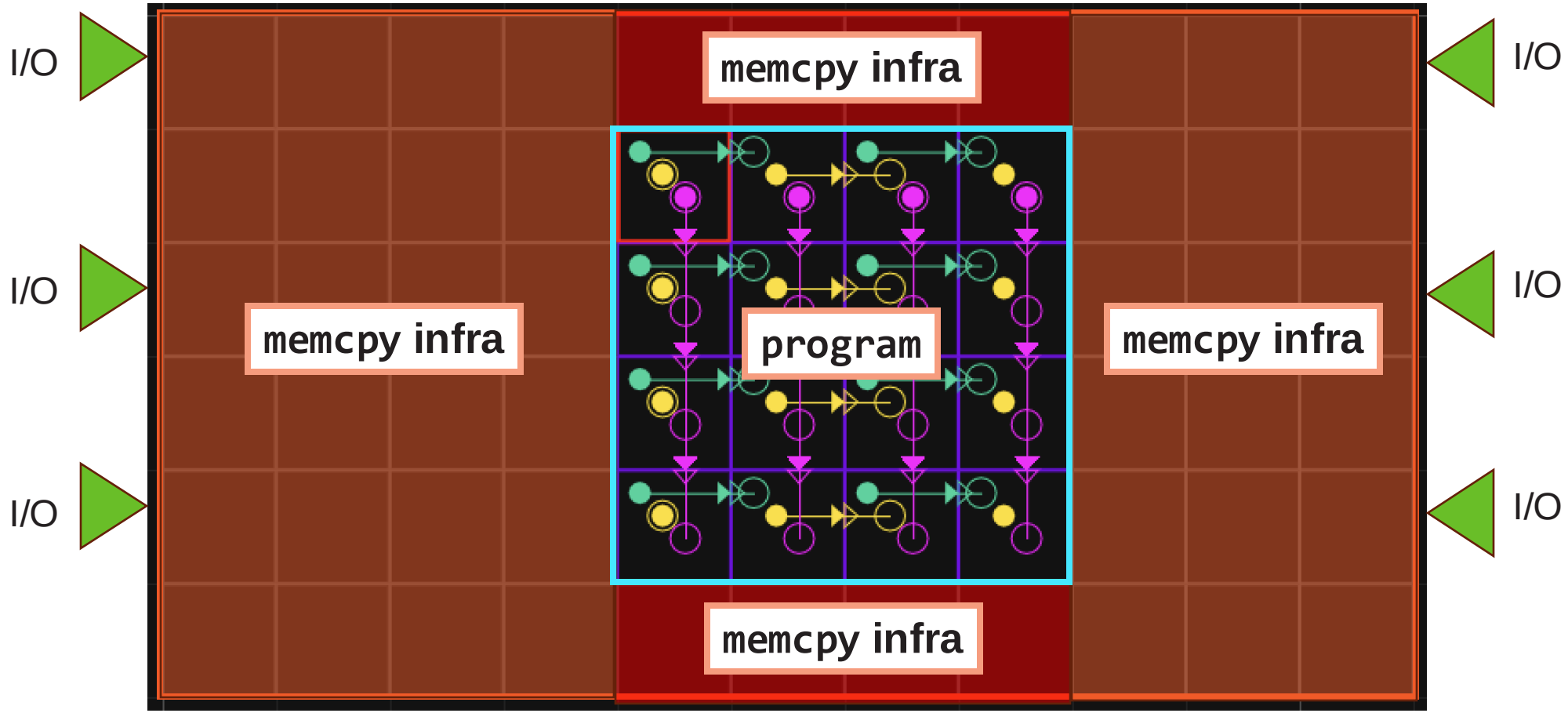


Host

GEMV Color Routes

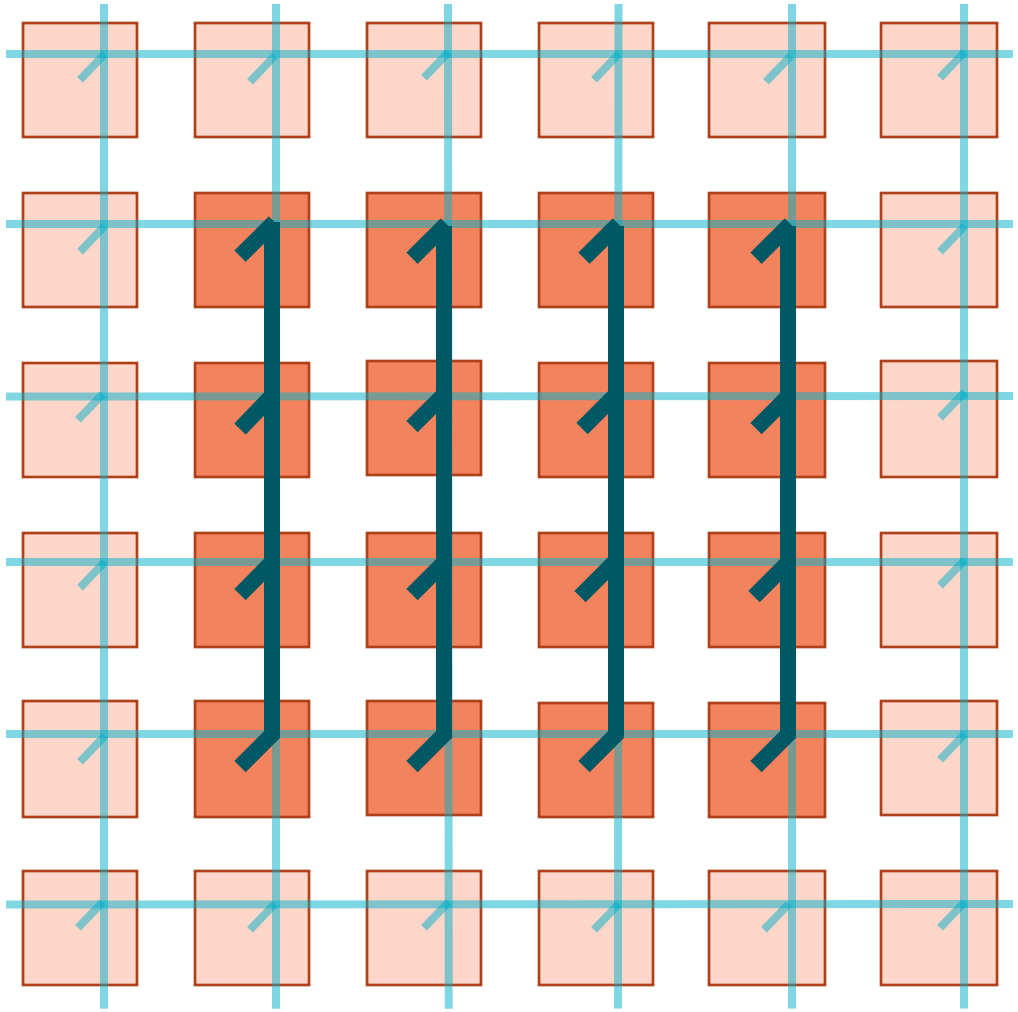
Circle: `RAMP`
Outline: `rx`
Solid: `tx`

GEMV Color Routes



GEMV Color Routes

Device

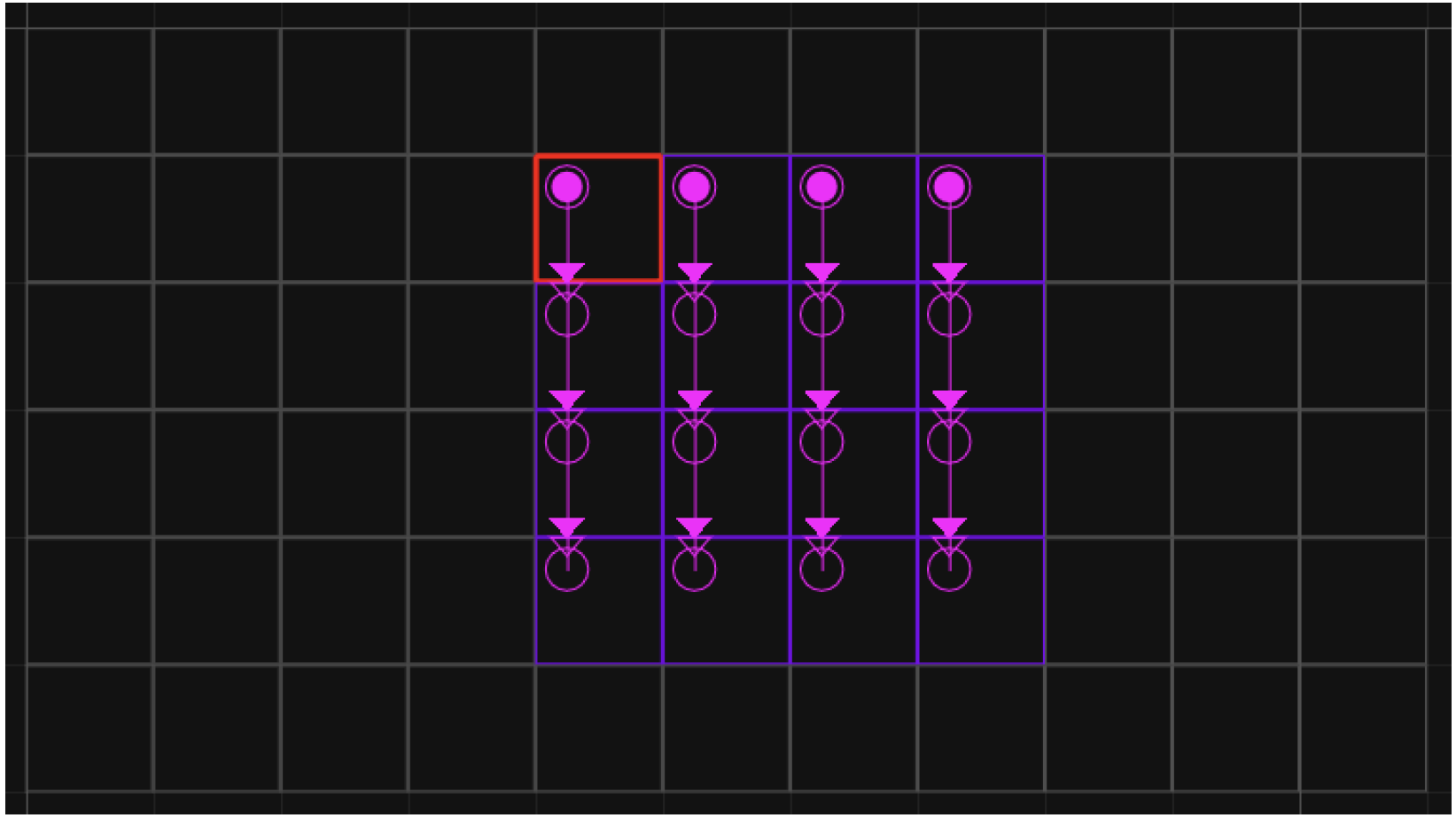


Color 5:

- **On row 0, receive from Ramp**
- **On rows 1, 2, 3, receive from North**
- **On rows 0, 1, 2, transmit down Ramp and South**
- **On row 3, transmit down Ramp**

Host

GEMV Color Routes



Circle: `RAMP`
Outline: `rx`
Solid: `tx`

GEMV Color Routes–CSL Layout

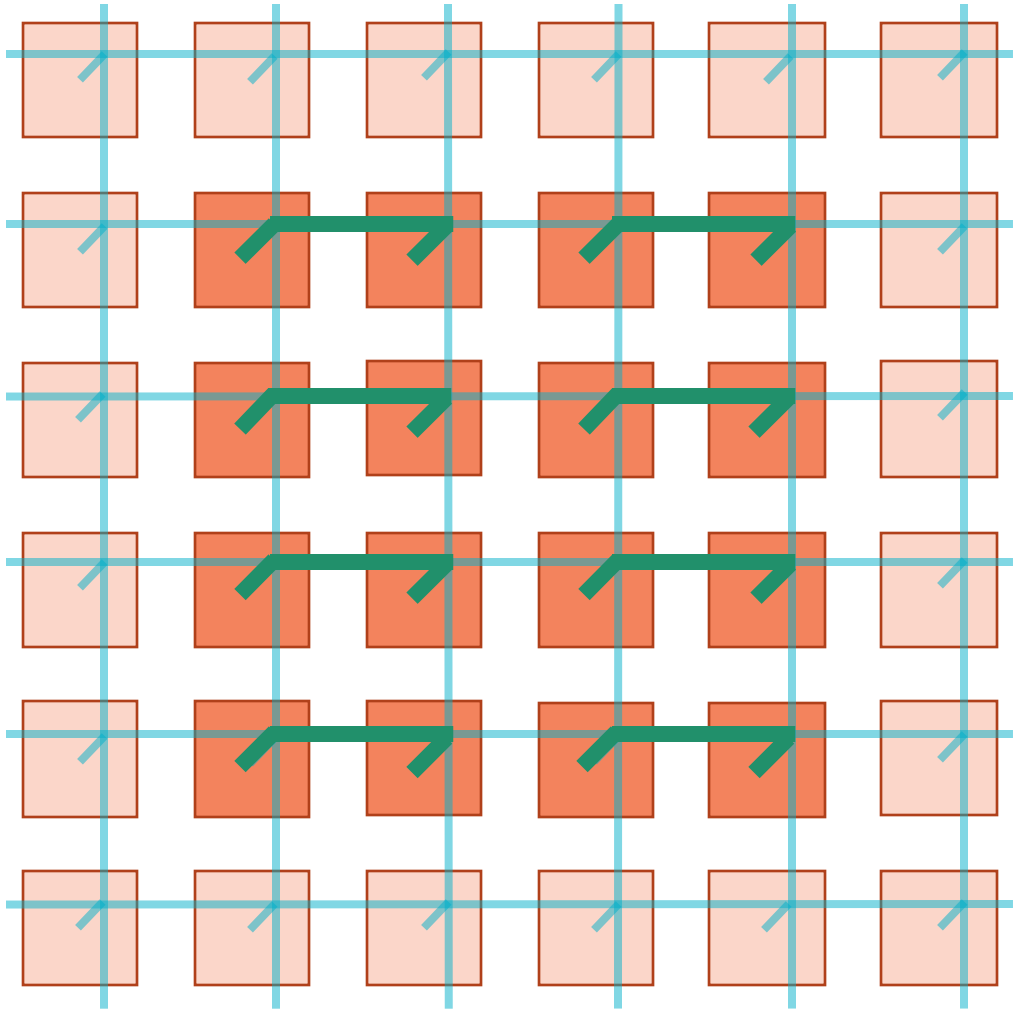
```
// Set routes for color blue, used for communicating x
for (@range(i16, kernel_x_dim)) |i| {    // Loop over all four columns
  for (@range(i16, kernel_y_dim)) |j| {  // Loop over all four rows

    // Handle first row
    if (j == 0) {
      @set_color_config(i, j, blue, .{.rx=.{RAMP}, .tx=.{RAMP, SOUTH}});

    // Handle last row
    } else if (j == kernel_y_dim-1) {
      @set_color_config(i, j, blue, .{.rx=.{NORTH}, .tx=.{RAMP}});

    } else {
      @set_color_config(i, j, blue, .{.rx=.{NORTH}, .tx=.{RAMP, SOUTH}});
    }
  }
}
```

GEMV Color Routes

- Note that we use **one color** for sending values of ***x*** down fabric
- However, we **can’t** do this for sending accumulated values across fabric
- Why?
  - PEs are also sending values **to** ramp, not just receiving **from** ramp
  - We must be able to differentiate values being sent up and down ramp
  - We need at least two colors, in a **checkerboard** pattern

GEMV Color Routes

Device



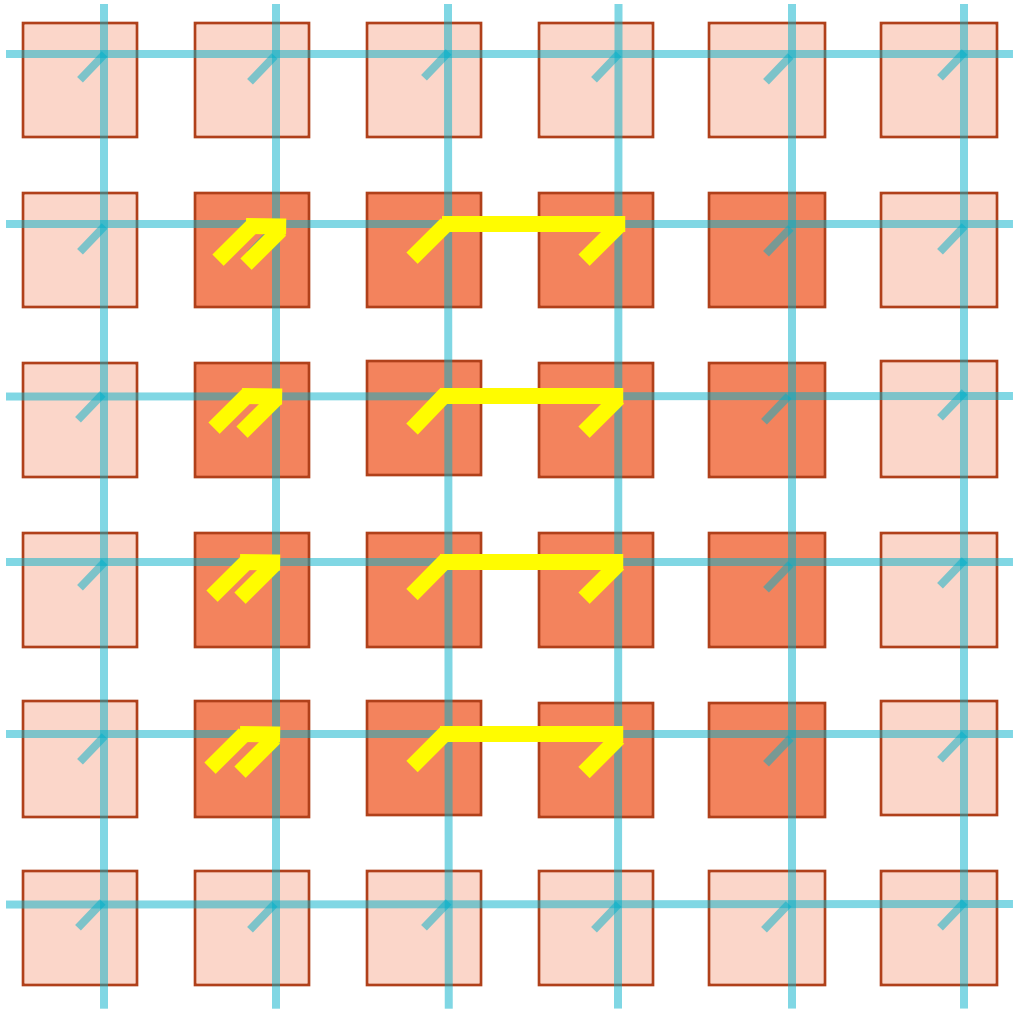
Color 3:

- **On column 0 and 2, receive from Ramp and transmit East**
- **On column 1 and 3, receive from East and transmit down Ramp**

Host

GEMV Color Routes

Device



Color 4:

- **On column 2, receive from West and transmit down Ramp**
- **On column 1, receive from Ramp and transmit East**

Host

GEMV Color Routes

Circle: `RAMP`
Outline: `rx`
Solid: `tx`

GEMV Color Routes–Simplified CSL Layout

- Each PE has an "**out color**" and an "**in color**"

	Column 0	Column 1	Column 2	Column 3
in color	*red** → yellow	green	yellow	green
out color	green	yellow	green	*magenta**

- "**out color**" is the routing color associated with transmitting accumulated sum
  - For the last column, this is the final result y
- "**in color**" is the routing color associated with receiving accumulated sum
  - For the first column, this is the input vector b
- ***italicized*** colors are `memcpy` colors

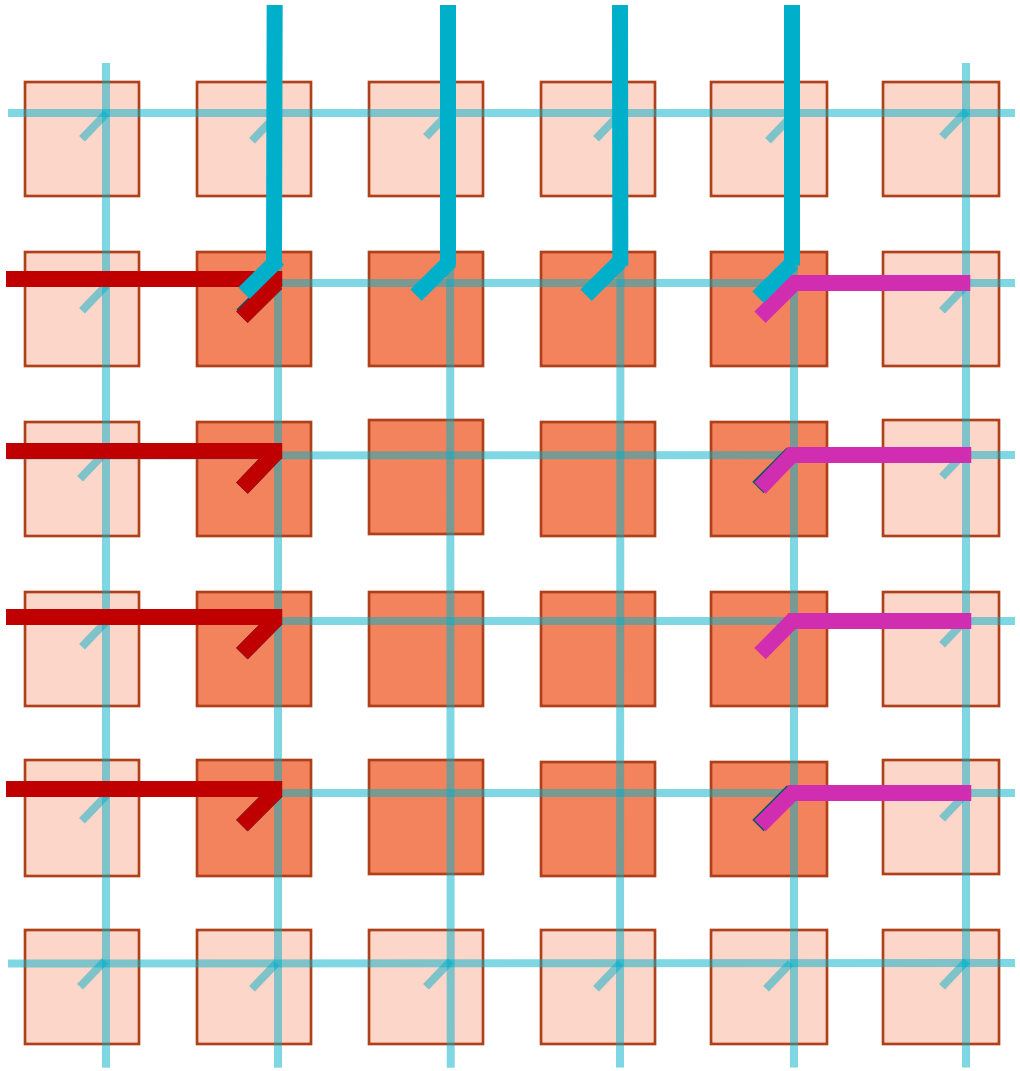
GEMV Color Routes–CSL Layout

```
for (@range(i16, kernel_x_dim)) |i| {    // Loop over all four columns
  for (@range(i16, kernel_y_dim)) |j| {  // Loop over all four rows

    if (i == 0) { // Handle first column
      @set_color_config(i, j, yellow, .{.rx=.{RAMP}, .tx=.{RAMP}});
      @set_color_config(i, j, green,  .{.rx=.{RAMP}, .tx=.{EAST}});

    } else if (i % 2) { // Handle even columns
      @set_color_config(i, j, yellow, .{.rx=.{WEST}, .tx=.{RAMP}});
      @set_color_config(i, j, green,  .{.rx=.{RAMP}, .tx=.{EAST}});

    } else { // Handle odd columns
      @set_color_config(i, j, yellow, .{.rx=.{RAMP}, .tx=.{EAST}});
      @set_color_config(i, j, green,  .{.rx=.{WEST}, .tx=.{RAMP}});
    }
  }
}
```

*yellow is unused on last column

GEMV Color Routes

Device



Memcpy colors 0, 1, 2:

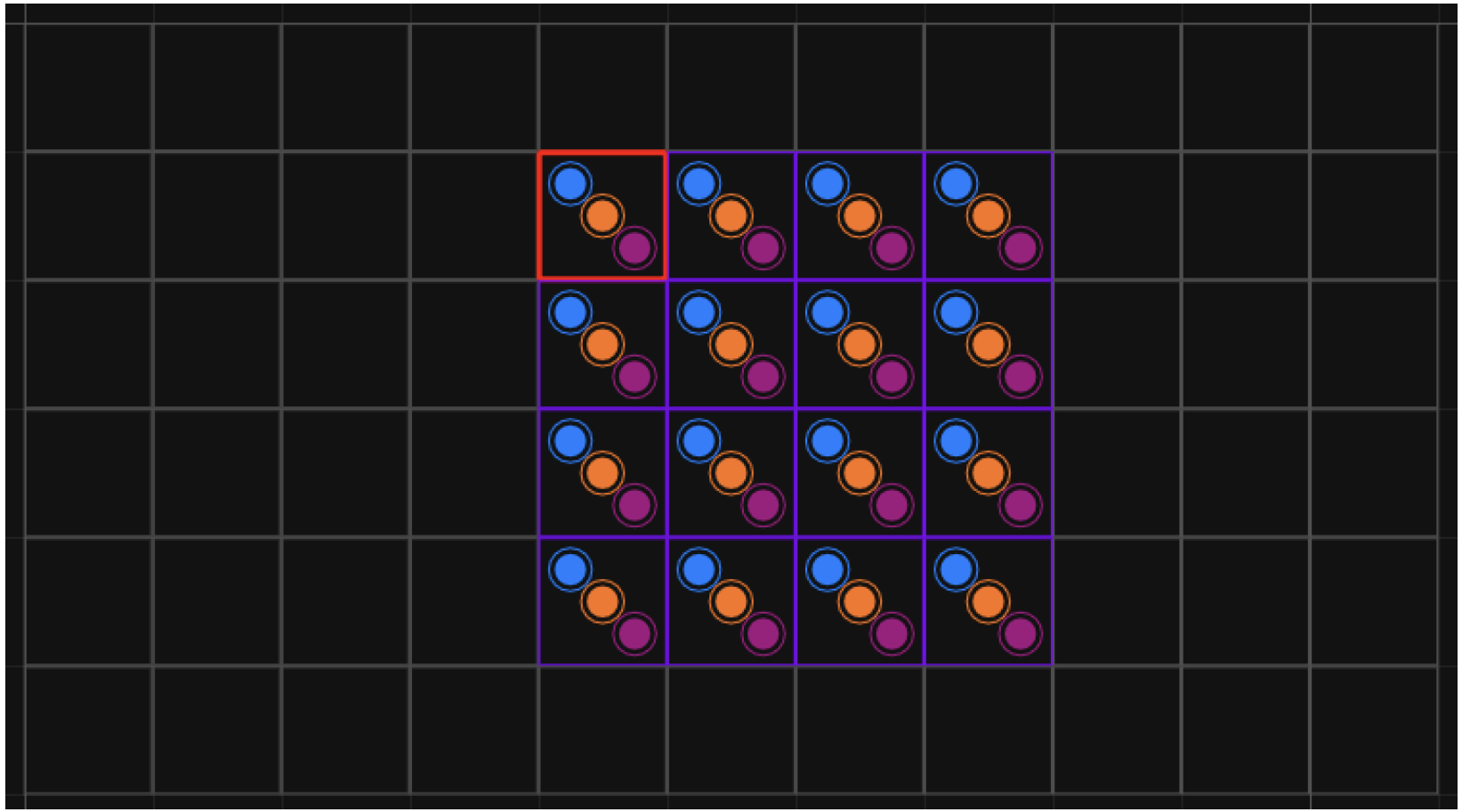
- **Used to stream data into and out of our program area**
- **Memcpy handles the routing**

Memcpy module uses 21, 22, 23

Host

Memcpy User Color Routes

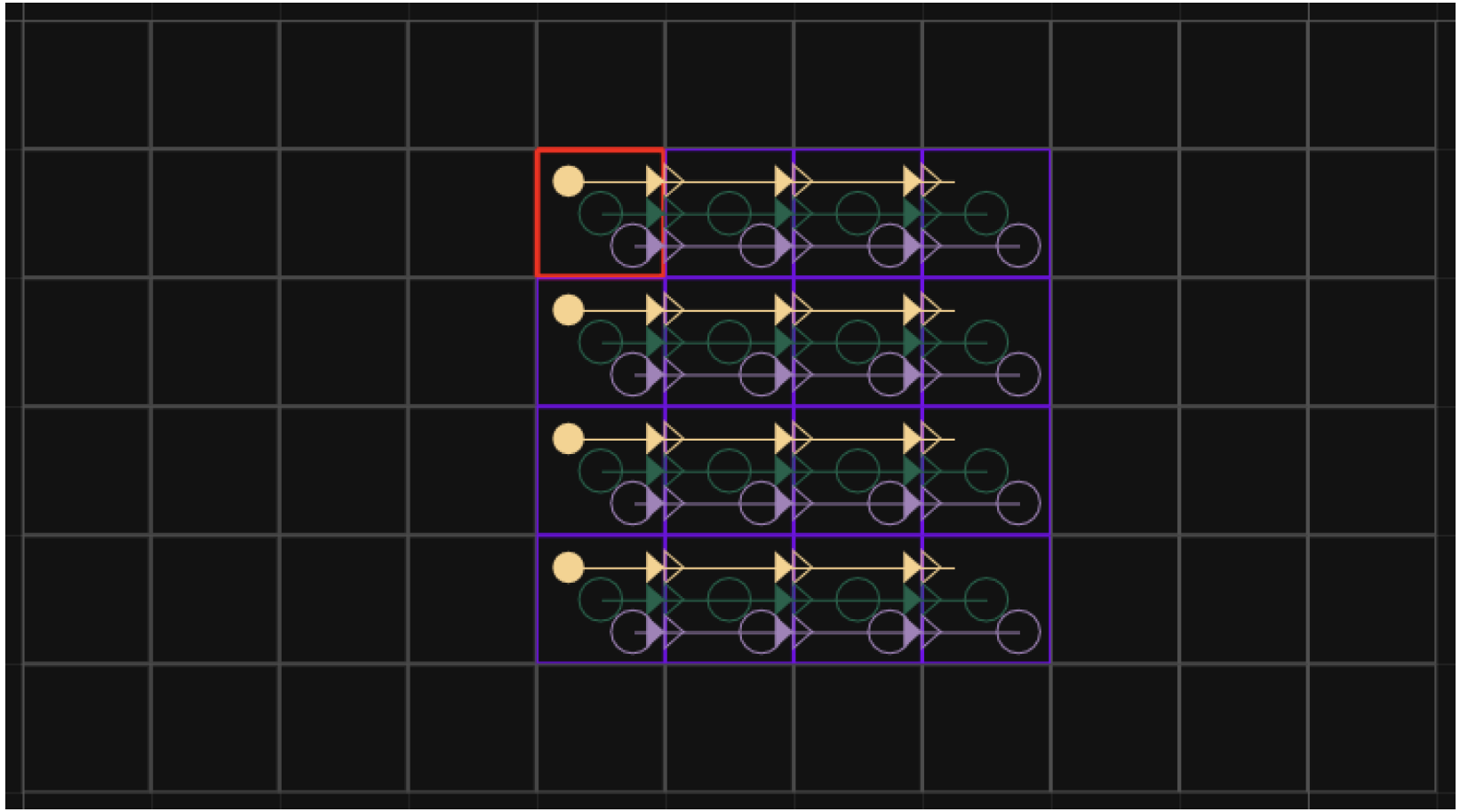
Color 0
Color 1
Color 2



Circle: `RAMP`
Outline: `rx`
Solid: `tx`

Memcpy Module Routes

Color 21
Color 22
Color 23



Circle: `RAMP`
Outline: `rx`
Solid: `tx`

Routes shown are only in program rectangle.

GEMV: Writing the CSL

We create a file named `pe_program.csl` that defines our program

What do we need to do in our device code?

1. Define color parameters
2. Define global variables for each PE
3. Define DSDs for our PE
4. Define task that activates for multiplying elements of A by x
5. Define task that accumulates sum
6. Define initial state of our task and colors, bind tasks to a color
7. Define layout of program and set parameters
8. Define routes for each color

What resources are available?

Routable Colors

- 24 colors available
- Labeled by IDs 0–23
- `memcpy` uses 21–23

Input Queues

- 8 IQs available
- Associated w/ color
- `memcpy` uses IQ 0, 1

Output Queues

- 8 OQs available
- Associated w/ color
- `memcpy` uses OQ 0

Tasks

- 8 data tasks (IDs 0–7)
  - Bound to IQs
- 23 local tasks (IDs 8–30)

Microthreads

- 12 microthreads
- Used for ops w/ fabric
- `memcpy` uses UT 0

Data Structure Registers

- 48 DSRs
- CSL handles allocation
- `memcpy` uses 2

A Word on Tasks

- In this example, we have **data tasks** and **local tasks**
- Whenever a task finishes, a **task picker** chooses a task from eligible tasks in the task table
- A task is eligible if it **activated** and **unblocked**
- **Data tasks** are activated by the receipt of wavelets from the fabric
- **Local tasks** are activated by explicit command from the programmer

```
var result: f32 = 0.0;
var sum: f32 = 0.0;

task foo_task(wavelet_data: f32) {
    result = wavelet_data;
    @activate(foo_task_id);
}

task bar_task() {
    sum += result;
}
```

Activated by receipt of wavelet in queue to which `foo_task` is bound

Activated by `@activate` call in `foo_task`

A Word on Tasks

```
var result: f32 = 0.0;
var sum: f32 = 0.0;

task main_task(wavelet_data: f32) {
  result = wavelet_data;
  @activate(foo_task_id);
  @activate(bar_task_id);
}

task foo_task() {
  sum += result;
  @unblock(bar_task_id);
}

task bar_task() {
  sum *= 2.0;
}

comptime { @block(bar_task_id); }
```

Activate both `foo_task` and `bar_task`, but only `foo_task` is eligible, as `bar_task` is blocked

Unblock `bar_task` when `foo_task` runs, so `bar_task` runs next

Block `bar_task` at compile time

A Word on Queues and Microthreads

- IQs and OQs serve as “buffers” for sending and receiving wavelets to and from fabric
- In addition to data tasks, IQs are also used when an operation has an input fabric operand
- OQs are used when an operation has an output fabric operand
- Operations with fabric operands can execute on asynchronous microthreads
- By default, microthread ID is taken from ID of queue used in operation
- IQs and OQs are bound to colors (but can be re-bound to different color at runtime)

```
task foo_task() void {
    const in_dsd = @get_dsd(fabin_dsd, .{
      .fabric_color = recv, .extent = 2, .input_queue = recv_iq });

    const out_dsd = @get_dsd(fabout_dsd, .{
      .fabric_color = send, .extent = 2, .output_queue = send_oq });

    @fadds(out_dsd, in_dsd, 5.0, .{ .async = true, .activate = bar_task });
}
```

Colors

Color ID	Name	Purpose
0	MEMCPYH2D_DATA_1_ID	Receive x from host
1	MEMCPYH2D_DATA_2_ID	Receive b from host
2	MEMCPYD2H_DATA_1_ID	Send y to host
3	**Even columns:** send_east_color **Odd columns:** recv_west_color	**Even columns:** Transmit psum **Odd columns:** Receive psum
4	**Even columns:** recv_west_color **Odd columns:** send_east_color	**Even columns:** Receive psum **Odd columns:** Transmit psum
5	x_color	**Top row:** Transmit x elems **All rows:** Receive x elems
…		
…		

Tasks

Task ID	Name	IQ ID	IQ Name	Color ID	Color Name	Purpose
0	***(reserved)***					
1	***(reserved)***					
2	`memcpy_recv_x`	2	`h2d_x_iq`	0	`MEMCPYH2D_DATA_1_ID`	On top row, receive x from fabric and forward along `x_color`
3	`memcpy_recv_b`	3	`h2d_b_iq`	1	`MEMCPYH2D_DATA_2_ID`	On left column, receive b from fabric and forward along `send_east_color`
4	`recv_x`	4	`recv_x_iq`	2	`x_color`	Receive x and compute Ax chunk. Activate `reduce` task when all x elems processed
…						
…						
10	reduce	--	--	--	--	Receive running sum from fabin DSD on `recv_west_color`, reduce with local chunk, send out on `send_east_color`. Last column sends to host.

Input Queues

IQ ID	Name	Color ID	Color Name	Purpose
0	***(reserved)***			
1	***(reserved)***			
2	`h2d_x_iq`	0	`MEMCPYH2D_DATA_1_ID`	Bound to `memcpy_recv_x` task
3	`h2d_b_iq`	1	`MEMCPYH2D_DATA_2_ID`	Bound to `memcpy_recv_b` task
4	`recv_x_iq`	5	`x_color`	Bound to `recv_x` task
5	`recv_w_iq`	**Even columns:** 4 **Odd columns:** 3	`recv_west_color`	Receives running sum in fabric input DSD
6	`..`			
7	`..`			

Output Queues

IQ ID	Name	Color ID	Color Name	Purpose
0	***(reserved)***			
1	***…***			
2	`x_oq`	5	`x_color`	On top row, send x to fabric
3	`recv_w_oq`	**Even columns:** 4 **Odd columns:** 3	`recv_west_color`	On left column, send b to fabric
4	`d2h_oq`	2	`MEMCPYD2H_DATA_1_ID`	Send final result y to host
5	`send_e_oq`	**Even columns:** 3 **Odd columns: 4**	`send_east_color`	Send out running sum to the East
6	…			
7	…			

recv_x task

```
// 48 kB of global memory contain A, x, b, y
var A: [M_per_PE*N_per_PE]f32; // A is stored column major
var y: [M_per_PE]f32;

// DSDs for accessing A, x, y
// A_dsd accesses column of A
var A_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> A[i] });
var y_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> y[i] });

// Use to keep track of # of invocations of recv_x task
// when num_recv_x == N_per_PE, we are done receiving x elements
var num_recv_x: i16 = 0;

task recv_x(x_val: f32) void {
    @fmacs(y_dsd, y_dsd, A_dsd, x_val);
    A_dsd = @increment_dsd_offset(A_dsd, M_per_PE, f32);
    num_recv_x += 1;
    if (num_recv_x == N_per_PE) @activate(reduce_task_id);
}
```

recv_x task

```
// 48 kB of global memory contain A, x, b, y
var A: [M_per_PE*N_per_PE]f32; // A is stored column major
var y: [M_per_PE]f32;
```

Global arrays in PE's memory

```
// DSDs for accessing A, x, y
// A_dsd accesses column of A
var A_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> A[i] });
var y_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> y[i] });

// Use to keep track of # of invocations of recv_x task
// when num_recv_x == N_per_PE, we are done receiving x elements
var num_recv_x: i16 = 0;

task recv_x(x_val: f32) void {
    @fmacs(y_dsd, y_dsd, A_dsd, x_val);
    A_dsd = @increment_dsd_offset(A_dsd, M_per_PE, f32);
    num_recv_x += 1;
    if (num_recv_x == N_per_PE) @activate(reduce_task_id);
}
```

recv_x task

M_per_PE is rows per PE (8)

N_per_PE is cols per PE (4)

```
// 48 kB of global memory contain A, x, b, y
var A: [M_per_PE*N_per_PE]f32; // A is stored column major
var y: [M_per_PE]f32;

// DSDs for accessing A, x, y
// A_dsd accesses column of A
var A_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> A[i] });
var y_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> y[i] });

// Use to keep track of # of invocations of recv_x task
// when num_recv_x == N_per_PE, we are done receiving x elements
var num_recv_x: i16 = 0;

task recv_x(x_val: f32) void {
    @fmacs(y_dsd, y_dsd, A_dsd, x_val);
    A_dsd = @increment_dsd_offset(A_dsd, M_per_PE, f32);
    num_recv_x += 1;
    if (num_recv_x == N_per_PE) @activate(reduce_task_id);
}
```

recv_x task

```
// 48 kB of global memory contain A, x, b, y
var A: [M_per_PE*N_per_PE]f32; // A is stored column major
var y: [M_per_PE]f32;

// DSDs for accessing A, x, y
// A_dsd accesses column of A
var A_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> A[i] });
var y_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> y[i] });

// Use to keep track of # of invocations of recv_x task
// when num_recv_x == N_per_PE, we are done receiving x elements
var num_recv_x: i16 = 0;

task recv_x(x_val: f32) void {
    @fmacs(y_dsd, y_dsd, A_dsd, x_val);
    A_dsd = @increment_dsd_offset(A_dsd, M_per_PE, f32);
    num_recv_x += 1;
    if (num_recv_x == N_per_PE) @activate(reduce_task_id);
}
```

accesses first M_per_PE elems of A (1st column)

recv_x task

```
// 48 kB of global memory contain A, x, b, y
var A: [M_per_PE*N_per_PE]f32; // A is stored column major
var y: [M_per_PE]f32;

// DSDs for accessing A, x, y
// A_dsd accesses column of A
var A_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> A[i] });
var y_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> y[i] });

// Use to keep track of # of invocations of recv_x task
// when num_recv_x == N_per_PE, we are done receiving x elements
var num_recv_x: i16 = 0;

task recv_x(x_val: f32) void {
    @fmacs(y_dsd, y_dsd, A_dsd, x_val);
    A_dsd = @increment_dsd_offset(A_dsd, M_per_PE, f32);
    num_recv_x += 1;
    if (num_recv_x == N_per_PE) @activate(reduce_task_id);
}
```

accesses `M_per_PE` elems of `y`, used to store `tmp` (running sum of `A*x` chunk)

recv_x task

```
// 48 kB of global memory contain A, x, b, y
var A: [M_per_PE*N_per_PE]f32; // A is stored column major
var y: [M_per_PE]f32;

// DSDs for accessing A, x, y
// A_dsd accesses column of A
var A_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> A[i] });
var y_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> y[i] });

// Use to keep track of # of invocations of recv_x task
// when num_recv_x == N_per_PE, we are done receiving x elements
var num_recv_x: i16 = 0;

task recv_x(x_val: f32) void {
    @fmacs(y_dsd, y_dsd, A_dsd, x_val);
    A_dsd = @increment_dsd_offset(A_dsd, M_per_PE, f32);
    num_recv_x += 1;
    if (num_recv_x == N_per_PE) @activate(reduce_task_id);
}
```

Global var to keep track of # x elems processed

recv_x task

```
// 48 kB of global memory contain A, x, b, y
var A: [M_per_PE*N_per_PE]f32; // A is stored column major
var y: [M_per_PE]f32;

// DSDs for accessing A, x, y
// A_dsd accesses column of A
var A_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> A[i] });
var y_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> y[i] });

// Use to keep track of # of invocations of recv_x task
// when num_recv_x == N_per_PE, we are done receiving x elements
var num_recv_x: i16 = 0;
```

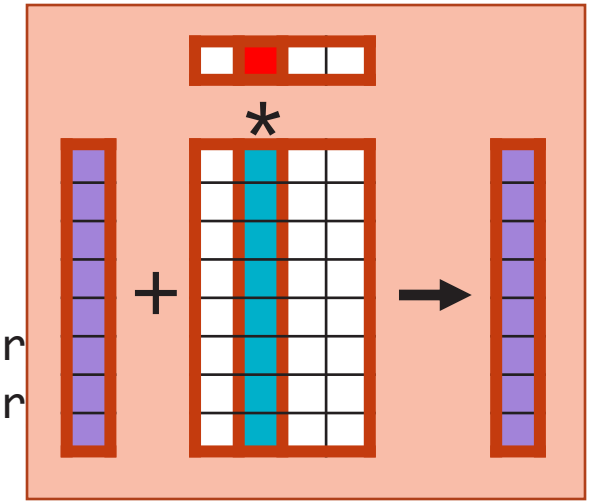
Task calculates x elem contribution to A*x

```
task recv_x(x_val: f32) void {
    @fmacs(y_dsd, y_dsd, A_dsd, x_val);
    A_dsd = @increment_dsd_offset(A_dsd, M_per_PE, f32);
    num_recv_x += 1;
    if (num_recv_x == N_per_PE) @activate(reduce_task_id);
}
```

recv_x task

```
// 48 kB of global memory contain A, x, b, y
var A: [M_per_PE*N_per_PE]f32; // A is stored column major
var y: [M_per_PE]f32;

// DSDs for accessing A, x, y
// A_dsd accesses column of A
var A_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per
var y_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per

// Use to keep track of # of invocations of recv_x task
// when num_recv_x == N_per_PE, we are done receiving x elements
var num_recv_x: i16 = 0;

task recv_x(x_val: f32) void {
    @fmacs(y_dsd, y_dsd, A_dsd, x_val);
    A_dsd = @increment_dsd_offset(A_dsd, M_per_PE, f32);
    num_recv_x += 1;
    if (num_recv_x == N_per_PE) @activate(reduce_task_id);
}
```

Computes $A_j * x_j$, accumulates to y



recv_x task

```
// 48 kB of global memory contain A, x, b, y
var A: [M_per_PE*N_per_PE]f32; // A is stored column major
var y: [M_per_PE]f32;

// DSDs for accessing A, x, y
// A_dsd accesses column of A
var A_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> A[i] });
var y_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> y[i] });

// Use to keep track of # of invocations of recv_x task
// when num_recv_x == N_per_PE, we are done receiving x elements
var num_recv_x: i16 = 0;

task recv_x(x_val: f32) void {
    @fmacs(y_dsd, y_dsd, A_dsd, x_val);
    A_dsd = @increment_dsd_offset(A_dsd, M_per_PE, f32);
    num_recv_x += 1;
    if (num_recv_x == N_per_PE) @activate(reduce_task_id);
}
```

Advance A_dsd to next col of A

recv_x task

```
// 48 kB of global memory contain A, x, b, y
var A: [M_per_PE*N_per_PE]f32; // A is stored column major
var y: [M_per_PE]f32;

// DSDs for accessing A, x, y
// A_dsd accesses column of A
var A_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> A[i] });
var y_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M_per_PE} -> y[i] });

// Use to keep track of # of invocations of recv_x task
// when num_recv_x == N_per_PE, we are done receiving x elements
var num_recv_x: i16 = 0;

task recv_x(x_val: f32) void {
    @fmacs(y_dsd, y_dsd, A_dsd, x_val);
    A_dsd = @increment_dsd_offset(A_dsd, M_per_PE, f32);
    num_recv_x += 1;
    if (num_recv_x == N_per_PE) @activate(reduce_task_id);
}
```

After receiving last x elem, begin reduction

GEMV: Host Code

```
runner.load(); runner.run() # Load and run program

# Copy chunks of A into all PEs. Each chunk stored column major
runner.memcpy_h2d(A_symbol, A_array, 0, 0, kernel_x_dim, kernel_y_dim, M_per_PE*N_per_PE,
streaming=False, ...)

# Stream x into PEs (0, 0) to (kernel_x_dim-1, 0)
runner.memcpy_h2d(MEMCPYH2D_DATA_1, x, 0, 0, kernel_x_dim, 1, N_per_PE, streaming=True, ...)

# Stream b into PEs (0, 0) to (0, kernel_y_dim-1) runner.memcpy_h2d(MEMCPYH2D_DATA_2, b, 0,
0, 1, kernel_y_dim, M_per_PE, streaming=True, ...)

# Stream y back from PEs (kernel_x_dim-1, 0) to (kernel_x_dim-1, kernel_y_dim-1)
y_result = np.zeros([M], dtype=np.float32)
runner.memcpy_d2h(y_result, MEMCPYD2H_DATA_1, kernel_x_dim-1, 0, 1, kernel_y_dim, M_per_PE,
streaming=True, ...)

runner.stop() # Stop program
```

GEMV: Compiling and Running

Simulator:

```
cslc --arch=wse3 layout.csl --fabric-dims=11,6 --fabric-offsets=4,1 ... -o out
cs_python run.py --name out
```

Machine*:

```
cslc --arch=wse3 layout.csl --fabric-dims=762,1172 --fabric-offsets=4,1 ... -o out
cs_python run.py --name out --cmaddr <CS_IP_ADDR>
```

*Additional wrappers provided to compile and run on Wafer-Scale Cluster appliance

Thank you