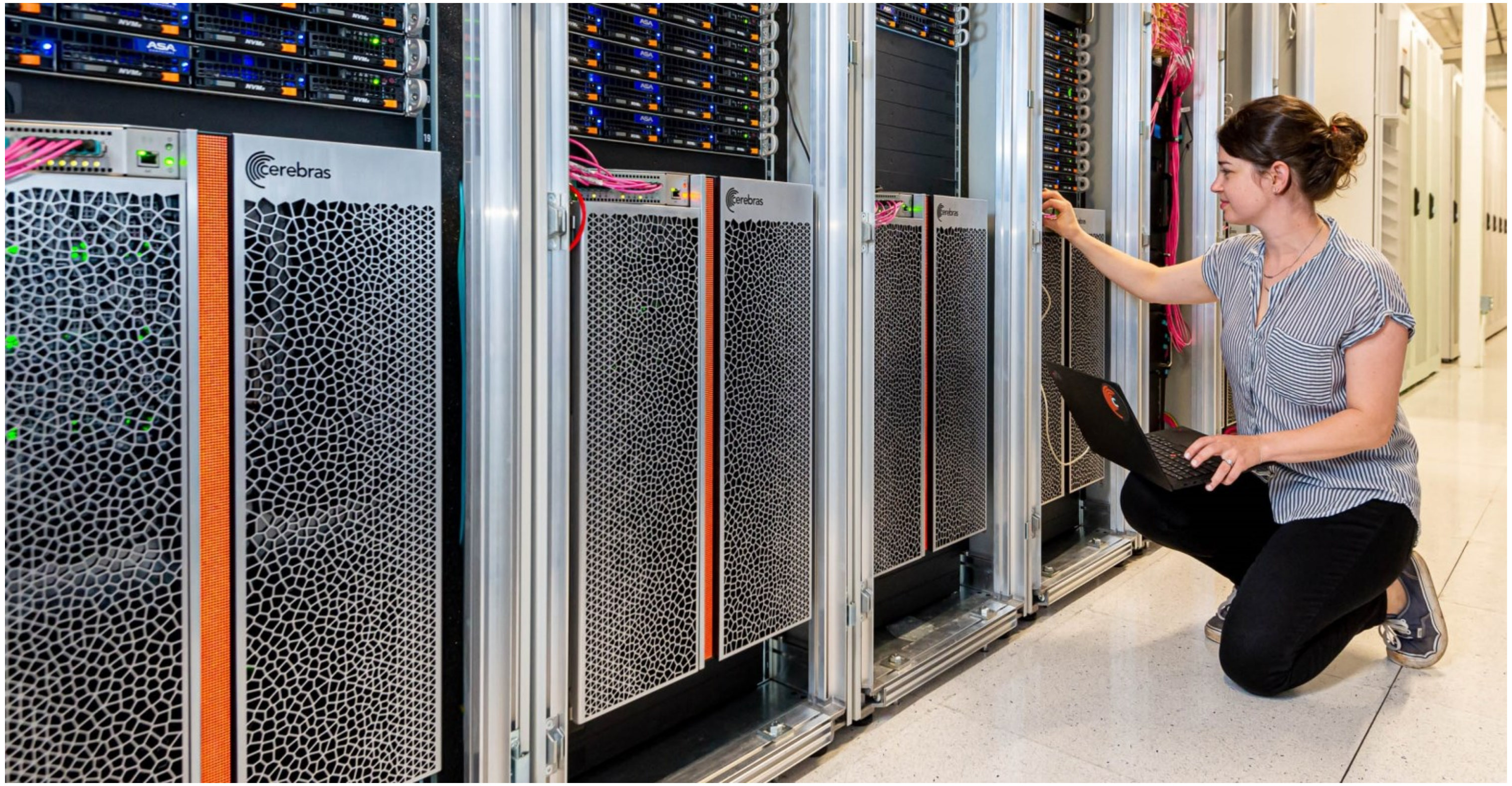
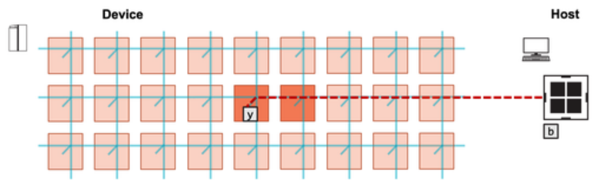
Hands-on 2: GEMV on a Multiple PEs



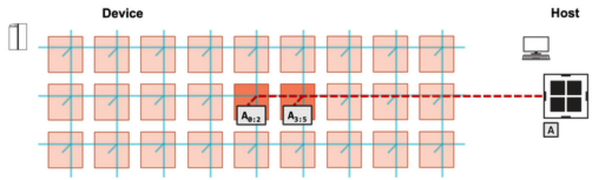
Objective

- Perform Matrix-Vector Multiplication on two adjacent PEs
- Matrix A of size (MxN with M rows, N columns)
  - N columns will be split across the two PEs
- Vector x of size N
  - x will be split across two PEs
- Vectors b and y of size M
- y = b + A@x

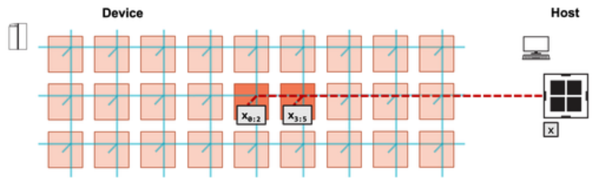
1. Host copies b into y array of left PE.



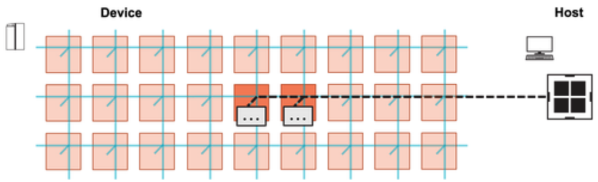
2. Host copies left N/2 columns of A to left PE, right N/2 columns to right PE.



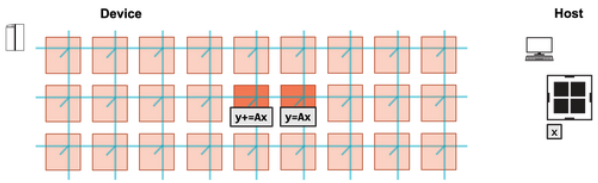
3. Host copies first N/2 elements of x to left PE, last N/2 elements to right PE.



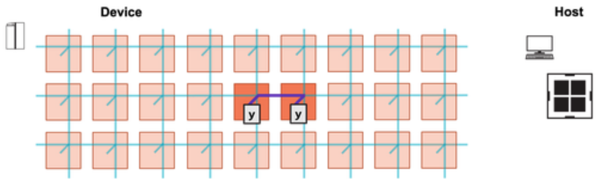
4. Host launches function to compute GEMV.



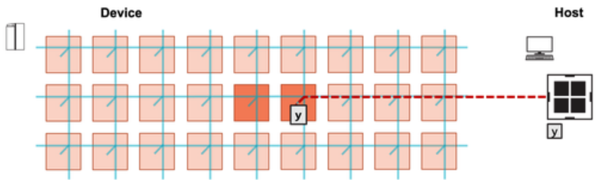
5. Each PE increments local y by local portion of matrix-vector product Ax.



6. Left PE sends local y to right PE, and right PE increments y by received values.



6. Right PE now contains final result y. Host copies back y from right PE.



What you need to do here

- In *layout.csl*:
- TO DO 1: set tile code for Left and Right PEs
- TO DO 2: set color config for Left and Right PEs

- In *pe_program.csl*:
- TO DO 1: Define send_right()
- TO DO 2: Define recv_left()
- TO DO 3: Define compute() function

Hints

- Example of setting tile code for multiple PE from walk-through 5
- Example of setting color configuration and communication functions from walk-through 6

Wash-up for hands on exercise two:

- Placing the program on each individual PE by tiling:
  - As with the walk-through, you can see how we explicitly set the *pe_id* parameter depending upon whether it's the left or right neighbour

```
// Left PE (0, 0)
@set_tile_code(0, 0, "pe_program.csl", .{
  .memcpy_params = memcpy.get_params(0),
  .M = M,
  .N_per_PE = N / 2,
  .pe_id = 0,
  .send_color = send_color,
  .exit_task_id = exit_task_id
});
```

```
// Right PE (1, 0)
@set_tile_code(1, 0, "pe_program.csl", .{
  .memcpy_params = memcpy.get_params(1),
  .M = M,
  .N_per_PE = N / 2,
  .pe_id = 1,
  .send_color = send_color,
  .exit_task_id = exit_task_id
});
```

Wash-up for hands on exercise two:

- Setting the color configuration:
  - Similarly to the walk-through example, the left PE's router will receive the wavelet from the ramp and then send it east.
  - The right PE will receive a wavelet from the west and then send it down the ramp to it's processor
  - The *send_color* variable defines which, of 24, virtual channels the wavelet will travel on

```
// Left PE sends its result to the right
@set_color_config(0, 0, send_color, .{.routes = .{ .rx = .{RAMP}, .tx = .{EAST} }});
```

```
// Right PE receives result of left PE
@set_color_config(1, 0, send_color, .{.routes = .{ .rx = .{WEST}, .tx = .{RAMP} }});
```

Wash-up for hands on exercise two:

- The *send_right* and *recv_left* functions will send and receive data respectively

- With the *recv_left* function also undertaking the required operation on the data when it arrives

```
fn send_right() void {
  const out_dsd = @get_dsd(fabout_dsd, .{
                    .fabric_color = send_color, .extent = M,
                    .output_queue = @get_output_queue(1)
                  });
  // After fmovs is done, activate exit_task to unblock cmd_stream
  @fmovs(out_dsd, y_dsd, .{ .async = true, .activate = exit_task_id });
}
```

```
fn recv_left() void {
  const in_dsd = @get_dsd(fabin_dsd, .{
                   .fabric_color = send_color, .extent = M,
                   .input_queue = @get_input_queue(1)
                 });
  // After fadds is done, activate exit_task to unblock cmd stream
  @fadds(y_dsd, y_dsd, in_dsd, .{ .async = true, .activate = exit_task_id });
}
```

Wash-up for hands on exercise two:

The compute function calls into the *gemv* function on both PEs, and then branches depending upon the *pe_id* parameter (the rank of the PE).

```
// Call gemv function and send/ receive partial result y
fn compute() void {
  gemv();
  if (pe_id == 0) {
    send_right();
  } else {
    recv_left();
  }
}
```