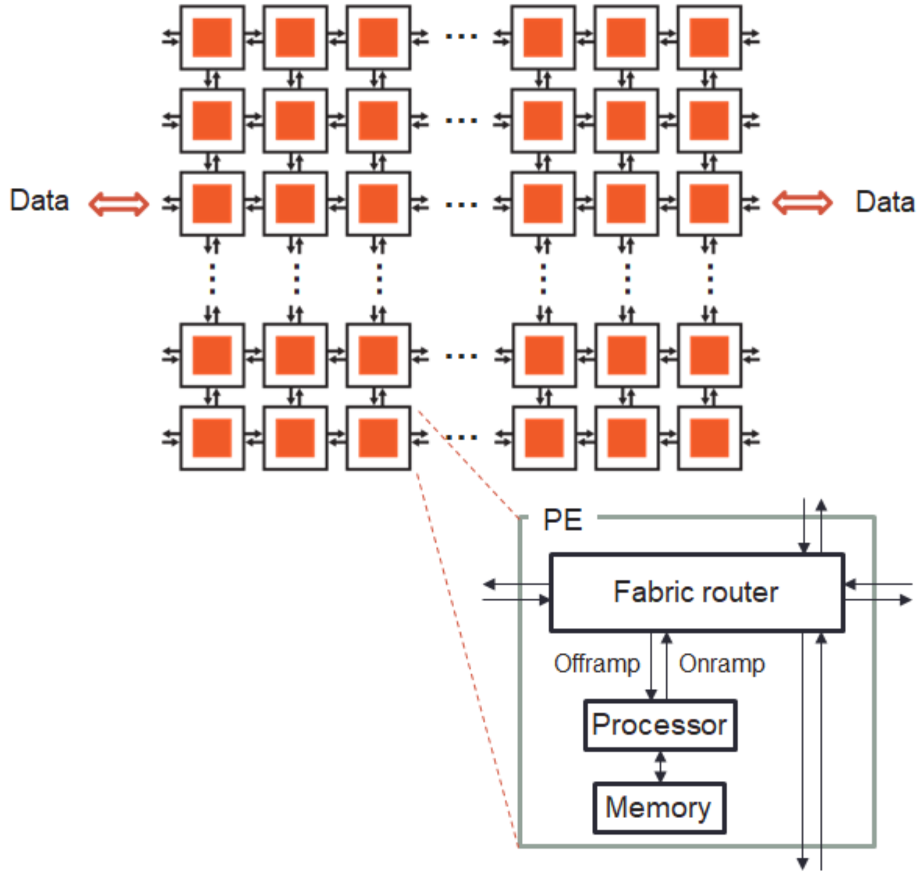
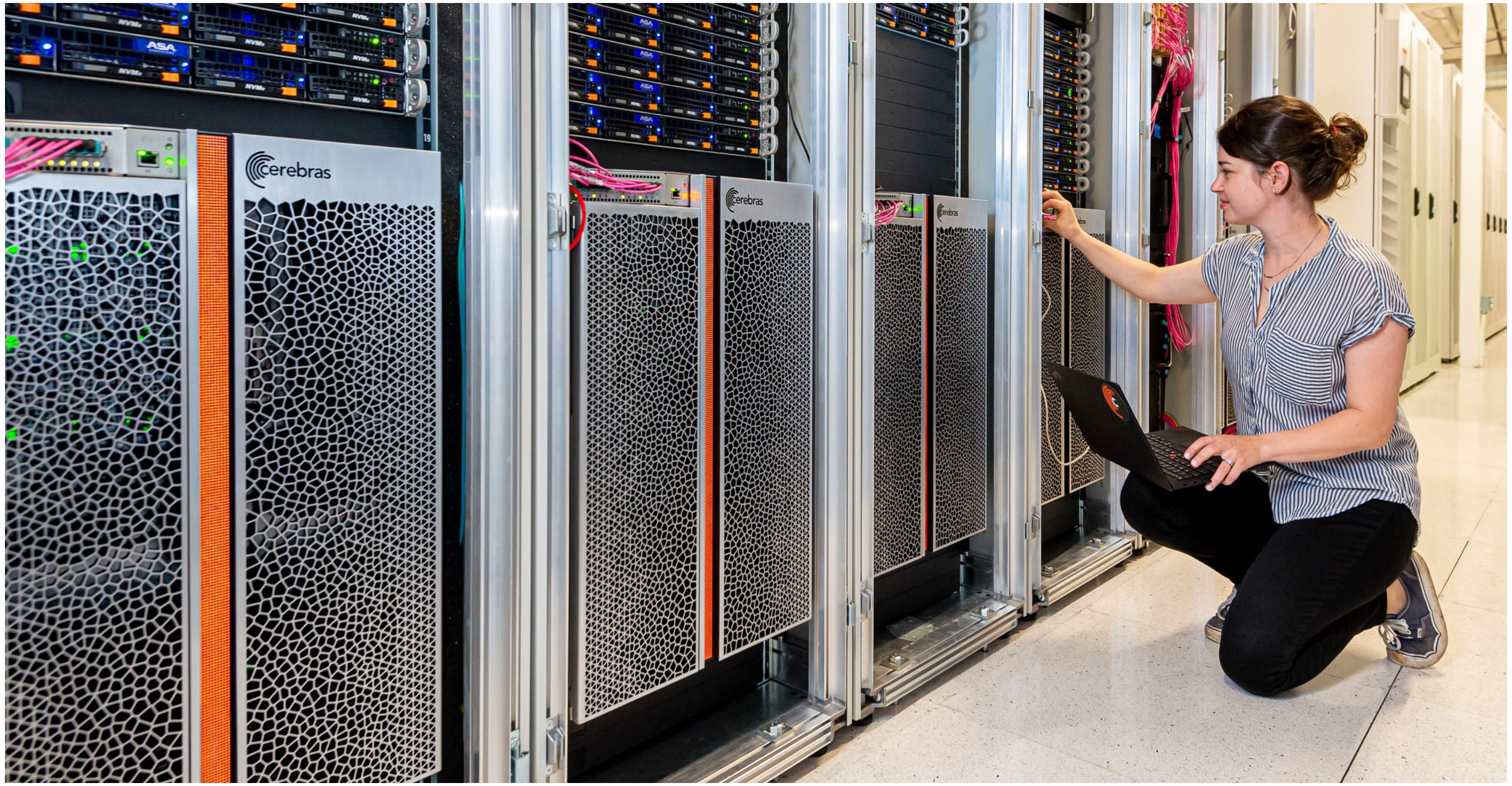
Cerebras SDK walk-through part two





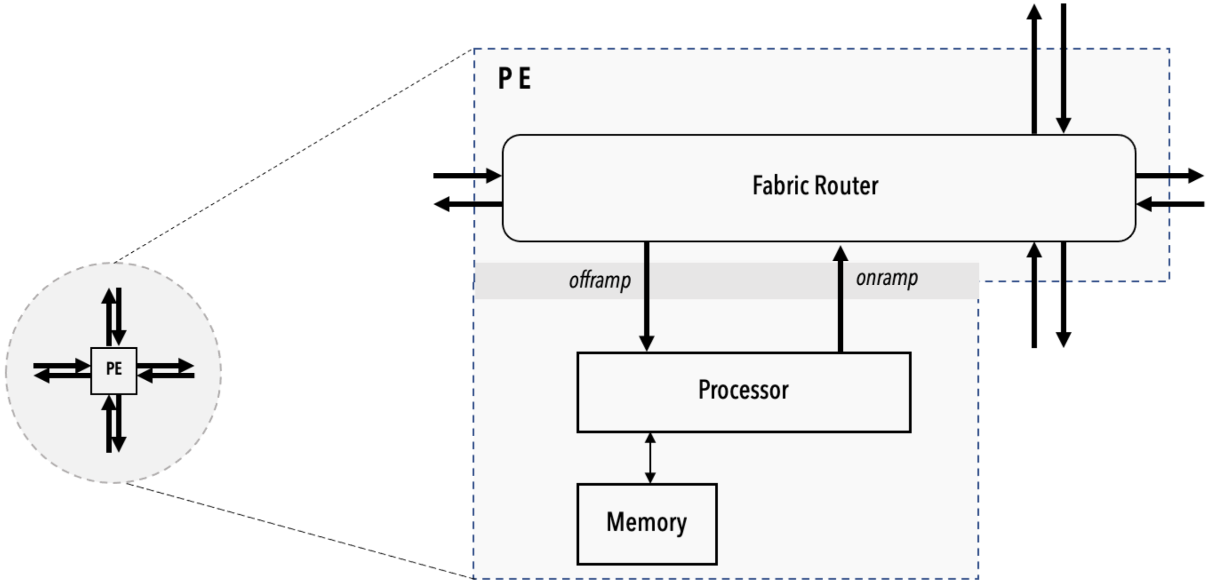
ExCALIBUR

epcc

cerebras

Until this point…

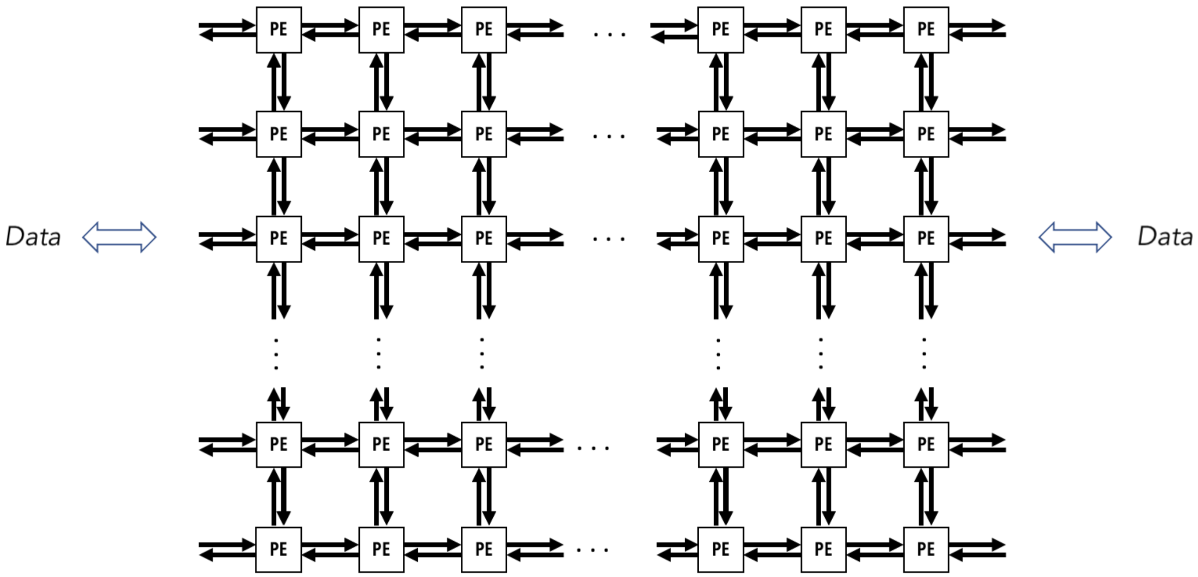
- We have focussed on running over a single Processing Element
  - You should have been able to undertake the first hands on exercise fairly easily using the concepts that we discussed



- However, we have over 850,000 PEs so it's a bit wasteful only running on one of these!
  - In this second walk through we are going to explore running on multiple PEs and communicating between them

Reminder

- Very many individual Processing Elements (PEs)
  - These run independent of each other (e.g. their own program counter)
- PEs are connected by 2D rectangular mesh across the chip
  - 32-bit messages (called wavelets) can be communicated with neighbours in a single cycle



- Each physical channel has 24 virtual communication channels known as ***colors*** that can be used for passing wavelets
- Each wavelet has associated with it a 5-bit identifier which defines which channel it is communicated on
  - Determines the wavelet's routing through the fabric and its consumption
  - This is a bit like a tag in MPI point-to-point communications, and similarly many messages on one color does not block messages with a different color using the same physical link

Multiple Processing Elements (PEs)

- Scaling up to multiple PEs for embarrassingly parallel codes is simple
  - This will be our initial focus in the second part of the walk through
  - Change into the *wt5-multiple-PEs* directory

- The number of PEs must be known at compile time, this is an example of a constant that is useful to provide as a parameter at compile time (and hence easy to modify)
  - We can do this using parameters (actually we have already been using parameters to specify the size of the data array!)

```
[vistor01@sdf-cs1 wt4-memoryDSDs]$ cd ../wt5-multiple-PEs
[vistor01@sdf-cs1 wt5-multiple-PEs]$ cslc layout.csl --fabric-dims=9,3 --fabric-offsets=4,1 --params=N:3,width:2 --
memcpy --channels=1 -o out
```

N and *width* are compile time parameters that we provide as part of the compilation command

Multiple Processing Elements (PEs)

The modifier *param* in the CSL code denotes that the value will be provided as a parameter

Imports the memcpy module onto the cores, now across *width by 1* PEs (previously it was across 1x1)

Set the rectangle of PEs in use to be *width by 1* (previously it was 1x1)

Use a for loop to assign the code to each PE and set appropriate parameters. We need to do this explicit assign for each PE (as in some cases different PEs have different code)

```
// N: array size
// width: Number of PE columns
param N: i16;
param width: i16;

// Color used by memcpy for RPC mechanism
const LAUNCH: color = @get_color(8);

// Import memcpy layout module for (width=2) x 1 grid of PEs
// This module defines parameters passed to program on the 2 PEs
const memcpy = @import_module("<memcpy/get_params>", .{
  .width = width,
  .height = 1,
  .LAUNCH = LAUNCH
});

layout {
  // set rectangle for 2x1 PEs
  @set_rectangle(width, 1);

  for (@range(i16, width)) |x| {
    @set_tile_code(x, 0, "pe_program.csl", .{
      .memcpy_params = memcpy.get_params(x),
      .N = N
    });
    }

  // Export device symbol for array "x", "y"
  @export_name("x", [*]f32, true);
  @export_name("y", [*]f32, true);

  // Export host-callable device function
  @export_name("compute", fn()void);
}
```

- The change here is in the layout.csl file
  - Although we have also tweaked *pe_program.csl* to use the *N* parameter for the array size, instead of this being hard coded to 3 in previous examples
- Using the *commands.sh* script, build and run this example
- Now rebuild to run over a different number of PEs
  - Hint – look in the *commands.sh* file to find the *width* parameter

Running on multiple PEs

```
[vistor01@sdf-cs1 wt5-multiple-PEs]$ ./commands.sh
INFO: Using SIF: /home/y26/shared/cs_sdk-1.0.0/cbcore_sdk-202311111408-10-4a54bce5.sif
compile successful
INFO: Using SIF: /home/y26/shared/cs_sdk-1.0.0/cbcore_sdk-202311111408-10-4a54bce5.sif
Reading file out/out.json
Reading file out/bin/out_rpc.json
Reading file out/west/out.json
Reading file out/east/out.json
fab w,h = 9,3
Kernel x,y w,h = 4,1 2,1
memcpy x,y w,h = 1,1 7,1
SUCCESS!
```

- Ensure you are in the w5-multiple-PEs directory and run the *commands.sh* script
- Now rebuild to run on four PEs and then rerun
  - Hint – you can edit the *width* parameter passed in via the *commands.sh* script
  - You must also increase the first fabric dimension by three

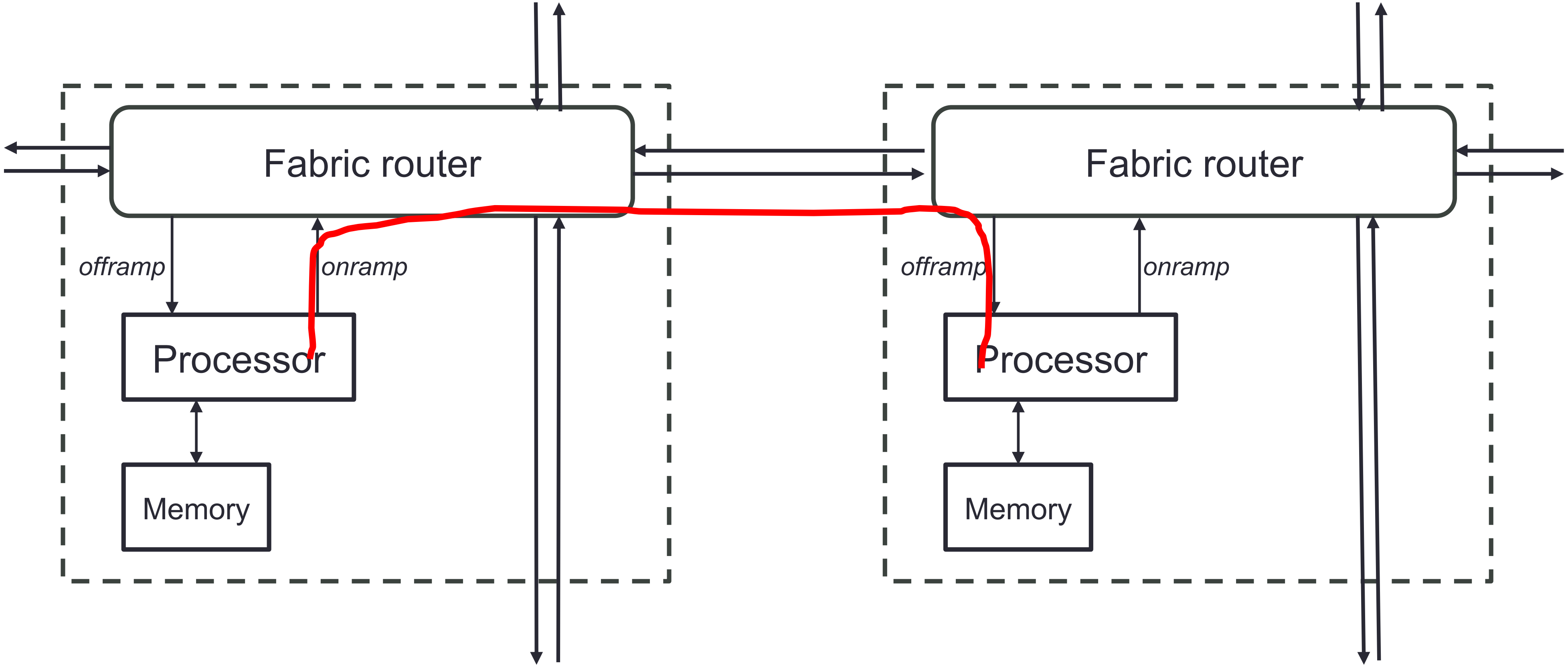
Solution: Running on multiple PEs

- Using the compile and run commands separately (rather than driving via the commands.sh file) to show the difference here

```
[vistor01@sdf-cs1 wt5-multiple-PEs]$ cslc layout.csl --fabric-dims=11,3 --fabric-offsets=4,1 --params=N:3,width:4 --
memcpy --channels=1 -o out
INFO: Using SIF: /home/y26/shared/cs_sdk-1.0.0/cbcore_sdk-202311111408-10-4a54bce5.sif
compile successful
[vistor01@sdf-cs1 wt5-multiple-PEs]$ cs_python run.py --name out
INFO: Using SIF: /home/y26/shared/cs_sdk-1.0.0/cbcore_sdk-202311111408-10-4a54bce5.sif
Reading file out/out.json
Reading file out/bin/out_rpc.json
Reading file out/west/out.json
Reading file out/east/out.json
fab w,h = 9,3
Kernel x,y w,h = 4,1 4,1
memcpy x,y w,h = 1,1 9,1
SUCCESS!
```

- There are some small differences in the output
  - The kernel width and height have changed from 2,1 to 4,1
  - The memcpy width and height have changed from 7,1 to 9,1

Communicating between PEs



- We will have two PEs, left and right, with the left PE sending data to the right PE
  - The left PE's sending route will be onramp and EAST, the right PE will use the receive route of WEST and offramp
  - Remember, there are 24 virtual communication channels (colors) each capable of communicating a 32-bit wavelet per cycle (router to router)

Communicating between PEs: layout.csl

```
// Colors
const send_color: color = @get_color(0); // Color used to send/recv data
// Task IDs
const exit_task_id: local_task_id = @get_local_task_id(9); // Task ID used by local task
```

- In *layout.csl* we have introduced a color to send/receive data between the PEs, and a task_id used to trigger the exit task
  - These are passed as parameters to the pe_program.csl code that runs on each PE

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

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

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

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

We are setting up each PE separately now, setting the *pe_id* parameter explicitly

The router receives a wavelet from the ramp and then sends EAST

We set up the communication route for each PE

The PE rank that this applies so

The color to use

The route

The router receives a wavelet from the WEST and then sends down the ramp

Communicating between PEs: pe_program.csl

```
fn compute() void {
  if (pe_id == 0) {
    const a: f32 = 2.0;
    @fmuls(x_dsd, a, x_dsd);
    send_right();
  } else {
    y[0] = 1.0;
    y[1] = 1.0;
    y[2] = 1.0;
    recv_left();
  }
}
```

- In *pe_program.csl* we branch based upon the *pe_id* value which has been provided as a parameter by *layout.csl*
  - The compute function on the left PE first computes the multiplication a*x, and calls send_right to transfer the result to the left. On the right PE, it initializes y, then receives the results from the left

```
fn send_right() void {
  const out_dsd = @get_dsd(fabout_dsd, .{.fabric_color = send_color, .extent = N, .output_queue = @get_output_queue(1)});
  @fmovs(out_dsd, x_dsd, .{ .async = true, .activate = exit_task_id });
}
```

Define a fabout DSD to send wavelets to the fabric along the color *send_color*

Extent is *N* as we intend to send *N* elements

Copies the N elements accessed by x_dsd into out_dsd

The operation is asynchronous, with the *exit_task_id* activated once it completes

Communicating between PEs: pe_program.csl

```
fn recv_left() void{
  const in_dsd = @get_dsd(fabin_dsd, .{fabric_color = send_color, .extent = N, .input_queue = @get_input_queue(1)});
  @fadds(y_dsd, y_dsd, in_dsd, .{ .async = true, .activate = exit_task_id});
}
```

Define a fabin DSD to receive wavelets from the fabric along the color *send_color*

Extent is *N* as we intend to receive *N* elements

Undertake an add operation on the *y_dsd* DSD using the values received in *in_dsd*

The operation is asynchronous, with the *exit_task_id* activated once it completes

- Thus, after this operation *y_dsd* contains the AXPY result
  - Advice is to always make communication operations asynchronous for performance

```
task exit_task() void {
  sys_mod.unblock_cmd_stream();
}

comptime {
  @bind_local_task(exit_task, exit_task_id);
}
```

This is the task executed when communication operations complete, which unblocks the memcpy stream. This must be executed on both PEs for completion to complete from the host perspective.

Binds this function as a task with the specific task id

Communicating between PEs

```
[vistor01@sdf-cs1 wt5-multiple-PEs]$ cd ../wt6-routes-fabricDSDs
[vistor01@sdf-cs1 wt6-routes-fabricDSDs]$ ./commands.sh
INFO: Using SIF: /home/y26/shared/cs_sdk-1.0.0/cbcore_sdk-202311111408-10-4a54bce5.sif
pe_program.csl:37:59: error: use of undeclared identifier
  const out_dsd = @get_dsd(fabout_dsd, .{ .fabric_color = TODO, .extent = TODO, .output_queue =
@get_output_queue(1)});
                                                          ^

layout.csl:29:3: error: semantic error in module imported here
  @set_tile_code(0, 0, "pe_program.csl", .{
  ^
```

- The code isn't quite finished, look in pe_program.csl at lines 37 and 43

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

fn recv_left() void{
  const in_dsd = @get_dsd(fabin_dsd, .{ .fabric_color = TODO, .extent = TODO, .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});
}
```

- Based on the previous two slides, can you complete the four TODOs (two in the send_right and two in the recv_left) functions?

Communicating between PEs

- Once we have completed the TODOs in the code…

```
[vistor01@sdf-cs1 wt6-routes-fabricDSDs]$ ./commands.sh
INFO: Using SIF: /home/y26/shared/cs_sdk-1.0.0/cbcore_sdk-202311111408-10-4a54bce5.sif
compile successful
INFO: Using SIF: /home/y26/shared/cs_sdk-1.0.0/cbcore_sdk-202311111408-10-4a54bce5.sif
Reading file out/out.json
Reading file out/bin/out_rpc.json
Reading file out/west/out.json
Reading file out/east/out.json
fab w,h = 11,3
Kernel x,y w,h = 4,1 2,1
memcpy x,y w,h = 1,1 7,1
SUCCESS!
```

- Then it all works, this illustrates point to point communications on the CS-2
  - The next question is how we could do collectives – these can of course be built out of P2P calls, but that would require a lot of work from the programmer…

Collective communications library

- We can perform collective communications directly using fabric DSDs and setting up the routes like we did with point-to-point communications
  - But this is time consuming and error prone

- Cerebras provides the *collectives_2d* library with the following functionality

```
fn init() void
fn broadcast(root: u16, buf: [*]u32, count: u16, callback: local_task_id) void
fn scatter(root: u16, send_buf: [*]u32, recv_buf: [*]u32, count: u16,
           callback: local_task_id) void
fn gather(root: u16, send_buf: [*]u32, recv_buf: [*]u32, count: u16,
          callback: local_task_id) void
fn reduce_fadds(root: u16, send_buf: [*]f32, recv_buf: [*]f32, count: u16,
                callback: local_task_id) void
```

- Communications are non-blocking, and activate a task when they complete

Collective communications library: layout.csl

```
// Colors
const c2d_x_color_0: color = @get_color(0);
const c2d_x_color_1: color = @get_color(1);
const c2d_y_color_0: color = @get_color(4);
const c2d_y_color_1: color = @get_color(5);

// Task IDs
const c2d_x_entrypt_0: local_task_id = @get_local_task_id(10);
const c2d_x_entrypt_1: local_task_id = @get_local_task_id(11);
const c2d_y_entrypt_0: local_task_id = @get_local_task_id(12);
const c2d_y_entrypt_1: local_task_id = @get_local_task_id(13);

const c2d = @import_module("<collectives_2d/params>");

layout {
  @set_rectangle(Pw, Ph);

  var Px: u16 = 0;
  while (Px < Pw) : (Px += 1) {
    var Py: u16 = 0;
    while (Py < Ph) : (Py += 1) {
      const params = c2d.get_params(Px, Py, .{
        .x_colors      = .{ c2d_x_color_0,   c2d_x_color_1 },
        .x_entrypoints = .{ c2d_x_entrypt_0, c2d_x_entrypt_1 },
        .y_colors      = .{ c2d_y_color_0,   c2d_y_color_1 },
        .y_entrypoints = .{ c2d_y_entrypt_0, c2d_y_entrypt_1 },
      });
      ...
    }
  }
  ...
}
```

- The collective communications library contains the *get_params* helper to set up the context required for collective communications
- This involves specifying colors and task ids for communicating in dimensions X and Y
- If you look in this file then you will see there are other aspects too, but these should already be familiar

Collective communications library: pe_program.csl

- In the *pe_program.csl* we then initialise the communications library and broadcast data to other PEs (the first argument is the root rank)

```
var broadcast_data = @zeros([Nx]u32);
var broadcast_recv = @zeros([Nx]u32);

var ptr_broadcast_data: [*]u32 = &broadcast_data;
var ptr_broadcast_recv: [*]u32 = &broadcast_recv;

task task_x() void {
   mpi_x.init();
   var send_buf = @ptrcast([*]u32, &broadcast_data);
   var recv_buf = @ptrcast([*]u32, &broadcast_recv);
   if (mpi_x.pe_id == 0) {
      mpi_x.broadcast(0, send_buf, Nx, task_x_id);
   } else {
      mpi_x.broadcast(0, recv_buf, Nx, task_x_id);
   }
}

fn f_run_x() void {
   @activate(task_x_id);
}
```

Allocate data of size *Nx* for both the send and recv arrays that are passed to the call

Get pointers of these arrays which can be provided to the communications library

Initialise the communications library

Cast pointers so they are type compatible with communications library function signature

pe_id member retrieves the ID of the current PE

Issue the *broadcast* of *Nx* elements of data with 0 as the root

Called from the host and activates the *task_x* task with our collectives communication in it

Collective communications library

- Our walkthrough example also illustrates a reduction, scatter and gather

```
[vistor01@sdf-cs1 wt6-routes-fabricDSDs]$ cd ../wt7-collective-communications
[vistor01@sdf-cs1 wt7-collective-communications]$ ./commands.sh
INFO: Using SIF: /home/y26/shared/cs_sdk-1.0.0/cbcore_sdk-202311111408-10-4a54bce5.sif
compile successful
INFO: Using SIF: /home/y26/shared/cs_sdk-1.0.0/cbcore_sdk-202311111408-10-4a54bce5.sif
Pw = width of the core = 15
Ph = height of the core = 15
chunk_size = 3
Nx = 45, Ny = 45
Reading file out/out.json
Reading file out/bin/out_rpc.json
Reading file out/west/out.json
Reading file out/east/out.json
fab w,h = 22,17
Kernel x,y w,h = 4,1 15,15
memcpy x,y w,h = 1,1 20,15
step 1: copy mode H2D(broadcast_data) to 1st column PEs
step 2: copy mode H2D(scatter_data) to 1st row PEs
step 3: call f_run_x to test broadcast and reduction
step 4: call f_run_y to test scatter and gather
step 5: copy mode D2H(broadcast_recv)
step 6: copy mode D2H(faddh_result) from 1st column PEs
step 7: copy mode D2H(gather_recv) from 1st row PEs
SUCCESS
```

- Have a look at the different collective calls in *pe_program.csl*

Conclusions

- The *layout.csl* file drives the number of PEs and placement of code onto each of them
- It's common to provide the PE id as a parameter (e.g. the loop index variable)

- The WSE supports both point-to-point and collective communications
  - We use colors to specify which virtual channel our 32-bit wavelets will be communicated on between PEs
    - With in-built intrinsic functions to send and receive the data
  - The collective communications library provides a convenient way in which we can leverage common collective communication calls between the cores
    - Detailed documentation is available at https://sdk.cerebras.net/csl/language/libraries#collectives-2d