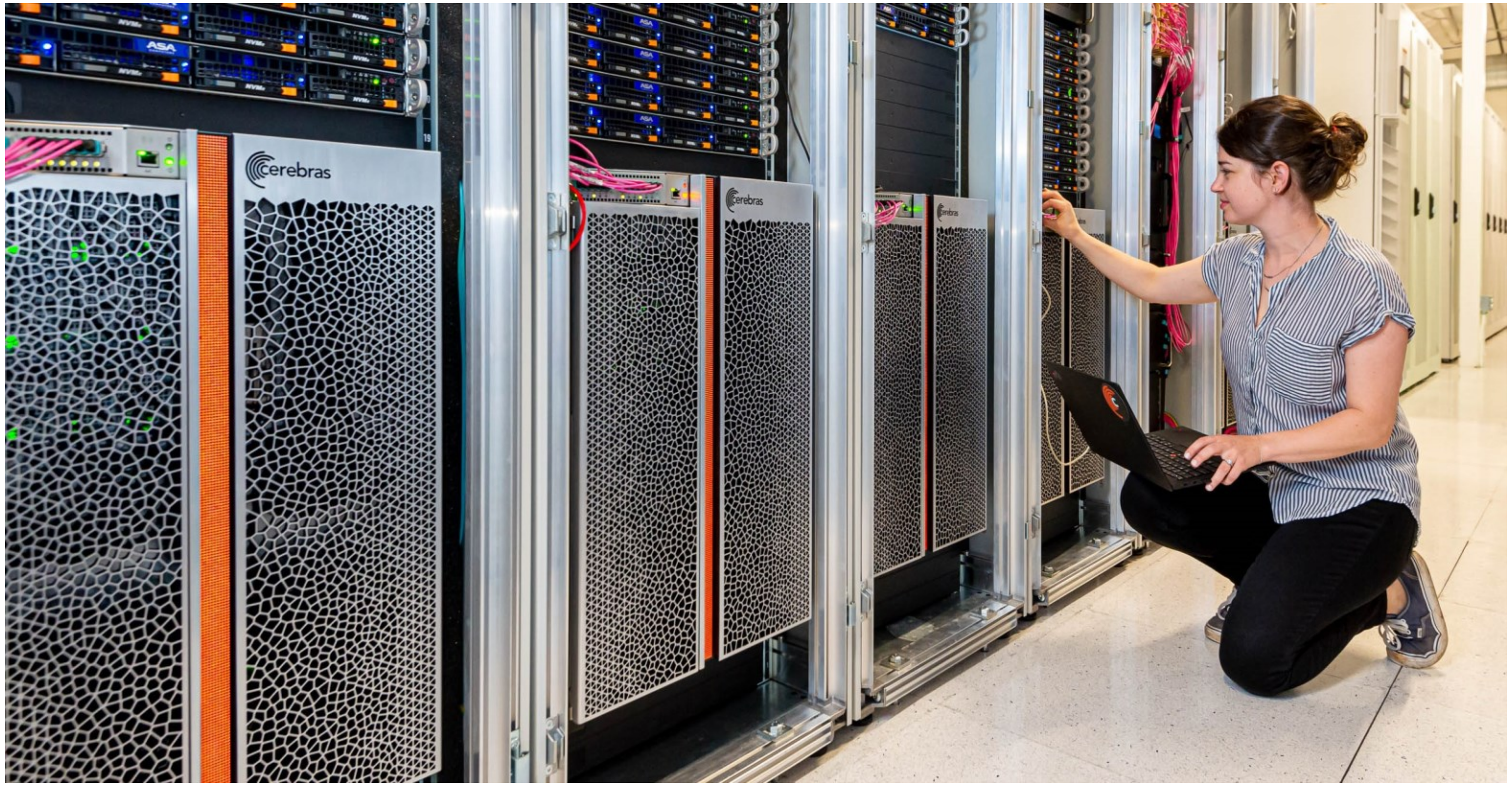
Cerebras SDK walk-through part one



ExCALIBUR 10

epcc

cerebras

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

Let’s get to programming the machine….

- Step one – using the visitor account assigned to you, login to our CS-2 host machine

```
ssh vistor01@sdf-cs1.epcc.ed.ac.uk
```

- We are now logged into the host machine that connects to the CS-2
- Step two – cd into the *cs2-sdk-training/practicals/walk-through* which we will be working in for this part of the tutorial

```
[vistor01@sdf-cs1 ~]$ cd cs2-sdk-training/practicals/walk-through
[vistor01@sdf-cs1 walk-through]$ ls
wt1-getting-started  wt2-basic-syntax  wt3-memcpy  wt4-memoryDSDs
wt5-multiple-PEs  wt6-routes-fabricDSDs  wt7-optimisation  wt8-debugging
```

Let one of the tutors know if you are struggling with this

Running my first CSL program

- Change into the *wt1-getting-started* directory

- Compile the code using the *cslc* command

```
[vistor01@sdf-cs1 walk-through]$ cd wt1-getting-started
[vistor01@sdf-cs1 wt1-getting-started]$ cslc layout.csl --fabric-dims=8,3 --fabric-
offsets=4,1 --memcpy --channels=1 -o out
INFO: Using SIF: /home/y26/shared/cs_sdk-1.0.0/cbcore_sdk-202311111408-10-4a54bce5.sif
compile successful
[nebcs1@sdf-cs1 wt1-getting-started]$ ls out/
bin  east  out.json  west
[nebcs1@sdf-cs1 wt1-getting-started]$ ls out/bin/
out_0_0.elf  out_rpc.json
```

Running my first CSL program

- Run (via the simulator)

```
[vistor01@sdf-cs1 walk-through]$ cd wt1-getting-started
[vistor01@sdf-cs1 wt1-getting-started]$ 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 = 8,3
Kernel x,y w,h = 4,1 1,1
memcpy x,y w,h = 1,1 6,1
SUCCESS!
```

Congratulations! You have run your first CSL program (although it doesn't do very much yet!)

What are these arguments to the CSL compiler?

```
[vistor01@sdf-cs1 wt1-getting-started]$ cslc layout.csl --fabric-dims=8,3 --fabric-offsets=4,1 --memcpy --channels=1 -o out
```

- *--fabric-dims=8,3* defines the size of the simulated fabric, which is 8 x 3
- *--fabric-offsets=4,1* defines where the program is placed on the fabric.
- --memcpy this flag is required to enable memcpy within the host program (we discussed this in the architecture slides)
- *--channels=1* determines the number of ethernet links that can be used to transfer data to/from the CS-2 (maximum of 12)
- *-o out* is the directory where the executables will be saved

For convenience…..

- To avoid you typing the compile and run command each time, we provide these in the *compile.sh* script that is in each walk through directory

```
[vistor01@sdf-cs1 wt1-getting-started]$ ./compile.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 = 8,3
Kernel x,y w,h = 4,1 1,1
memcpy x,y w,h = 1,1 6,1
SUCCESS!
```

What's this layout.csl?

- Defines the layout of the program on the CS-2

```
[vistor01@sdf-cs1 wt1-getting-started]$ cslc layout.csl --fabric-dims=8,3 --fabric-
offsets=4,1 --memcpy --channels=1 -o out
```

```
const LAUNCH: color = @get_color(8);

const memcpy = @import_module("<memcpy/get_params>", .{
  .width = 1,
  .height = 1,
  .LAUNCH = LAUNCH
});

layout {

  @set_rectangle(1, 1);


  @set_tile_code(0, 0, "pe_program.csl", .{ .memcpy_params = memcpy.get_params(0) });


  @export_name("init_and_compute", fn()void);
}
```

Colour (virtual channel) used for Remote Procedure Call (RPC) mechanism

Imports the memcpy module (WSE side of memcpy support) needed to launch kernel

We are using just one PE (columns=1, rows=1)

The lone PE should execute "pe_program.csl" and we pass the memcpy parameters as a parameter

Expose this function "init_and_compute" to the host

In the layout.csl file you will see comments to explain these lines (omitted here for space)

Driving from the host

```
import argparse

from cerebras.sdk.runtime.sdkruntimepybind import SdkRuntime

# Read arguments
parser = argparse.ArgumentParser()
parser.add_argument('--name', help="the test compile output dir")
parser.add_argument('--cmaddr', help="IP:port for CS system")
args = parser.parse_args()


# Construct a runner using SdkRuntime
runner = SdkRuntime(args.name, cmaddr=args.cmaddr)


# Load and run the program
runner.load()
runner.run()


# Launch the init_and_compute function on device
runner.launch('init_and_compute', nonblock=False)


# Stop the program
runner.stop()
```

We create an instance of the Cerebras SdkRuntime (the host side library that interacts with the CS-2).

Load our compiled program onto the WSE

Start running the program on the WSE. This won't do anything yet, it activates and is ready for a Remote Procedure Call (RPC) from the host.

Launch the "init_and_compute" function on the WSE and wait until it has completed

Stop the program on the WSE and clean up

Running on the actual CS-2

- Much of our development will be run via the simulator, but we then want to run on the actual machine for production runs
- Two changes are required:

1. The fabric-dims setting in the compile command must be replaced with the fabric dimension of the actual CS-2, 757 x 996.

```
[vistor01@sdf-cs1 wt1-getting-started]$ cslc layout.csl --fabric-dims=757x996 --fabric-offsets=4,1 --memcpy --channels=1 -o out
```

2. The IP address of the CS-2 system needs to be passed to the host program runtime SDK, the new run command becomes

```
[vistor01@sdf-cs1 wt1-getting-started]$ cs_python run.py --name out --cmaddr $CS_IP_ADDR:9000
```

Exploring (and fixing!) the code

- Change into the *wt2-basic-syntax* directory

```
[vistor01@sdf-cs1 wt1-getting-started]$ cd ../wt2-basic-syntax
```

- We are going to use this as a basis for a (very simple) code update to fix the code

Exploring the *pe_program.csl* code

```
// Struct containing parameters for memcpy layout
param memcpy_params: comptime_struct;
const sys_mod = @import_module("<memcpy/memcpy>", memcpy_params);

// Constants defining dimensions of our data
const N: i16 = 3;

// 48 kB of local PE memory contains x and y
var x: [N] f32;
var y: [N] f32;

fn sum(x_ptr : *[N] i16, y_ptr : *[N], value : i16) void
{
  …
}

fn initialize() void {
  …
}

fn init_and_compute() void {
  …
}

comptime {
  // Export function so it is host-callable by RPC mechanism
  @export_symbol(init_and_compute);

  // Create RPC server using color LAUNCH
  @rpc(@get_data_task_id(sys_mod.LAUNCH));
}
```

Infrastructure we need for the memcpy library support

Defines a constant integer of 16 bits

Declares arrays x and y to be of size N and type single-precision floating point. These are allocated in the PE's local memory

We will look at these functions in detail in a minute

Sets up the program by exporting the "init_and_compute" function

Create an RPC server (so it can be called from the host) using the colour defined in layout.csl

Exploring the *pe_program.csl* code

```
fn initialize() void {
    for (@range(i16, N)) |idx| {
      x[idx] = 1.0;
      y[idx] = @as(f32, idx);
  }
}


fn init_and_compute() void {
  initialize();
  sum(&x, &y, 2);

  // After this function finishes, memcpy's cmd_stream must
  // be unblocked on all PEs for further memcpy commands
  // to execute
  sys_mod.unblock_cmd_stream();
}
```

This *initialise* function accepts no arguments and returns no value

For loop, looping up to *N* with the *idx* variable containing the loop value at each iteration

Accesses the global variables *x* and *y*, setting *x* to be the value 1.0 and *y* to be the current loop value (cast to a 32-bit floating point)

Entry point called by the host

Calls the initialise function followed by *sum* , with *x and y* global variables as arguments and the scalar value *2*

After the work completes need to do some cleanup on the command stream to ensure can execute further commands

Exploring the *pe_program.csl* code

Defines *sum* to be a function with a void return type. It accepts three arguments

The *value* argument is a scalar of type i16 (16-bit integer). CSL uses a colon to provide type information

```
fn sum(x_ptr : *[N] i16, y_ptr : *[N], value : i16) void
{
    for (@range(i16, N)) |idx| {
     // TO DO: EDIT following line to multiply value to Right Hand Side
     y_ptr.*[idx] += x_ptr.*[idx];
  }
}
```

x_ptr and *y_ptr* are defined as pointers of over memory of size *N*. Like C, the asterisk, *, represents a pointer

This is a for loop, which loops over some array and at each iteration the variable *idx* contains the loop's current value

- This is very similar to a loop in Python but just with a different syntax.
- The in-built range function is used here to provide an array containing 0 to *N*-1 with each element of type i16. Again this is similar to Python's range function

The asterisk is used to dereference a pointer, so here we are accessing the *idx* element of *y_ptr*

Fixing the *pe_program.csl* code

- For the algorithm to be correct we need to also multiply by the *value* scalar on the RHS
  - Then recompile and run

```
fn sum(x_ptr : *[N] i16, y_ptr : *[N], value : i16) void
{
    for (@range(i16, N)) |idx| {
      // TO DO: EDIT following line to multiply value to Right Hand Side
      y_ptr.*[idx] += x_ptr.*[idx];
  }
}
```

This will look something like *value * x_ptr.*[idx];*

```
[vistor01@sdf-cs1 wt2-basic-syntax]$ ./compile.sh
```

Exposing data in layout.csl

- What we have done so far isn't terribly useful as we are not transferring any data to or from the host (so at the very least we can't even view the results!)
  - Very similar to our previous layout.csl, with two additional lines that export symbols *x* and *y*

```
const LAUNCH: color = @get_color(8);


const memcpy = @import_module("<memcpy/get_params>", .{
  .width = 1,
  .height = 1,
  .LAUNCH = LAUNCH
});

layout {

  @set_rectangle(1, 1);


  @set_tile_code(0, 0, "pe_program.csl", .{ .memcpy_params = memcpy.get_params(0) });

  @export_name("x", [*]f32, true);
  @export_name("y", [*]f32, true);

  @export_name("init_and_compute", fn()void);
}
```

These two new lines of code export symbols *x* and *y* so that we can view them from the host

Binding variables to exported symbols *x* and *y*

- In *program.csl* we then bind program variables to the symbols *x* and *y* that have been exposed to the host
  - Again very similar to the previous code, just with two new lines added here

```
comptime {
  // Export symbol pointing to x, y so it is host-read/writeable
  @export_symbol(x_ptr, "x");
  @export_symbol(y_ptr, "y");

  // Export function so it is host-callable by RPC mechanism
  @export_symbol(compute);

  // Create RPC server using color LAUNCH
  @rpc(@get_data_task_id(sys_mod.LAUNCH));
}
```

These two new lines of code bind our global variables *x_ptr* and *y_ptr* to *x* and *y* respectively that have been exposed to the host in the layout.csl file

Interfacing from the host

- In *run.py* we then allocate data using Numpy, and can then copy input data it and/or copy results out

```
# Construct a runner using SdkRuntime
runner = SdkRuntime(args.name, cmaddr=args.cmaddr)

# Get symbol for copying x, y onto and off device
x_symbol = runner.get_id('x')
y_symbol = runner.get_id('y')

......

y = np.full(shape=N, fill_value=1.0, dtype=np.float32)
x = np.full(shape=N, fill_value=1.0, dtype=np.float32)
runner.memcpy_h2d(x_symbol, x, 0, 0, 1, 1, N, streaming=False,
  order=MemcpyOrder.ROW_MAJOR, data_type=MemcpyDataType.MEMCPY_32BIT, nonblock=False)
runner.memcpy_h2d(y_symbol, y, 0, 0, 1, 1, N, streaming=False,
  order=MemcpyOrder.ROW_MAJOR, data_type=MemcpyDataType.MEMCPY_32BIT, nonblock=False)


# Launch the compute function on device
runner.launch('compute', nonblock=False)

# Copy y back from device
y_result = np.zeros([N], dtype=np.float32)
runner.memcpy_d2h(y_result, y_symbol, 0, 0, 1, 1, N, streaming=False,
  order=MemcpyOrder.ROW_MAJOR, data_type=MemcpyDataType.MEMCPY_32BIT, nonblock=False)
```

Retrieve references to the *x* and *y* symbols on the device

Using numpy to allocate (and initialise) input data on the host

Copy both fields on the host to the CS-2 device

Run the *compute* function on the device and wait for completion

Allocate result data on the host and copy back data held referenced by the *y* symbol

Let’s see if this works….

```
[vistor01@sdf-cs1 wt2-basic-syntax]$ cd ../wt3-memcpy
[vistor01@sdf-cs1 wt3-memcpy]$ ./compile.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 = 8,3
Kernel x,y w,h = 4,1 1,1
memcpy x,y w,h = 1,1 6,1
Traceback (most recent call last):
  File "run.py", line 66, in <module>
    np.testing.assert_allclose(y_result, expected_y, atol=0.01, rtol=0)
AssertionError:
  Not equal to tolerance rtol=0, atol=0.01

Mismatched elements: 3 / 3 (100%)
Max absolute difference: 1.
Max relative difference: 0.33333334
x: array([2., 2., 2.], dtype=float32)
y: array([3., 3., 3.], dtype=float32)
```

- The assertion at the end of *run.py* fails, can you change the value in *program.csl* to fix this?
  - Hint: Look at the value in *program.csl* passed to the *sum* function. Ask one of the tutors if you get stuck here

Memory Data Structure Descriptors (DSDs)

- Provides a mechanism for efficiently performing operations on arrays of data (known as tensors)
  - By describing operation at this more abstract level, the CSL compiler has more information upon which it can then determine the best way to drive the actual code

- There are two steps to using DSDs:
  1. Define the DSD(s)
  2. Use builtin operations to operate upon your defined DSDs

Handle to the newly created DSD

Tell CSL that we work with a 1D array

i is the induction variable

N is the loop bound (i.e. loop all the way to N)

Array access expression evaluated at each iteration

```
var x_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{N} -> x[i] });
```

```
var A_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{M} -> A[i*N] });
```

This DSD is slightly more complicated, accessing the *i*N* element at each iteration, i.e. it loops until *M*, but at each iteration the loop access is strided by *N*

Memory Data Structure Descriptors (DSDs)

```
// DSDs for accessing x, y
// .tensor_access field defines the access pattern of these DSD
// |i| specifies the induction variable
// {N} specifies the loop bound
var x_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{N} -> x[i] });
var y_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{N} -> y[i] });
```

Creates DSDs for both *x* and *y*

- We use the *fmacs* builtin that executes a multiply accumulate
  - The line of code will be *@fmacs(y_dsd, y_dsd, x_dsd, a);*
  - cd into wt4-memoryDSDs and replace the loop in the *sum* function of *program.csl* with this DSD operation
  - The answer is on the next slide!

Memory Data Structure Descriptors (DSDs)

```
// DSDs for accessing x, y
// .tensor_access field defines the access pattern of these DSD
// |i| specifies the induction variable
// {N} specifies the loop bound
var x_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{N} -> x[i] });
var y_dsd = @get_dsd(mem1d_dsd, .{ .tensor_access = |i|{N} -> y[i] });

fn compute() void {
  const a: f32 = 2.0;

  // @fmacs is a builtin for multiply-add that operates on DSDs
  @fmacs(y_dsd, y_dsd, x_dsd, a);

  // After this function finishes, memcpy's cmd_stream must
  // be unblocked on all PEs for further memcpy commands
  // to execute
  sys_mod.unblock_cmd_stream();
}
```

We have replaced the loop with the fused multiple add DSD intrinsic

Conclusions

- We have explored the key concepts required for getting started in writing code for the Cerebras CS-2
- This is enough for our first hands-on activity
  - Which we will introduce you to in a moment!
- Our focus so far has been on a single Processing Element (PE)
  - For now we will limit our focus to this
  - In the second part of the walk through (after the break) we will explore running on multiple PEs and communicating data between them