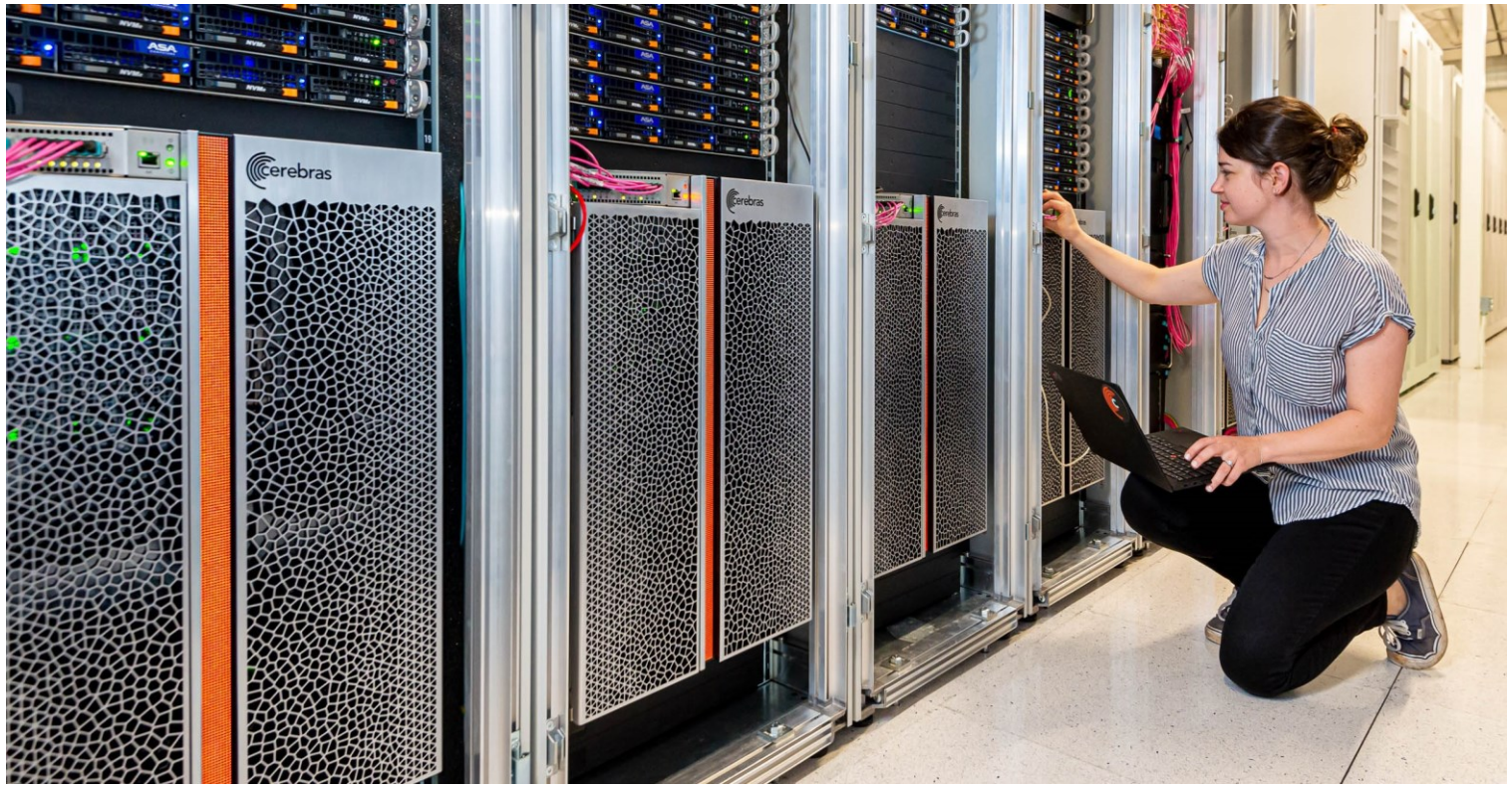
Hands-on 1: GEMV on a Single PE



EXCALIBUR 10

epcc

cerebras

Objective and steps

- Perform Matrix-Vector Multiplication on a Single PE

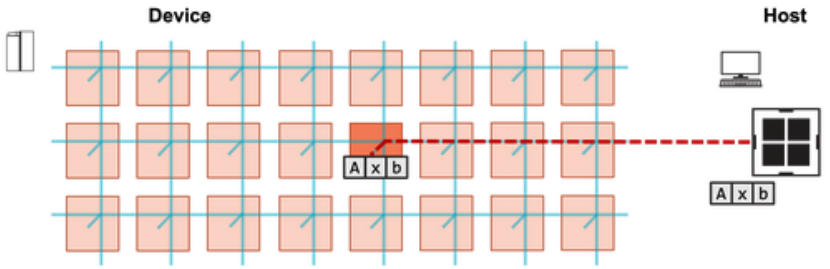
- Matrix A of size (MxN – M rows, N columns)
- Vector x of size N
- Vectors b and y of size M

- y = b + A@x

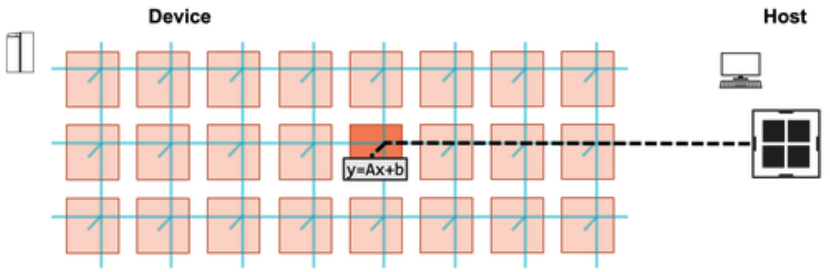
Problem Steps

Visually, this program consists of the following steps:

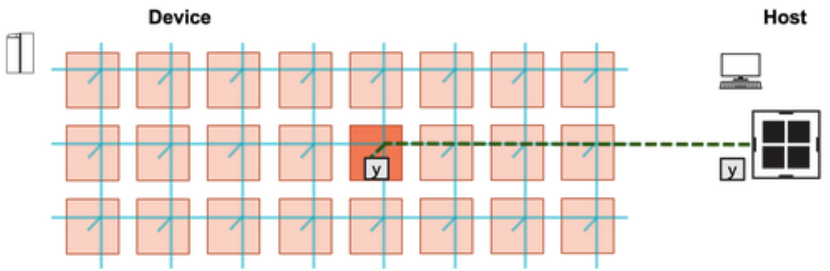
1. Host copies A, x, b to device.



2. Host launches function to compute y.



3. Host copies result y from device.



What you need to do here

- In *pe_program.csl*:
- TO DO 2: Construct gemv() function
  - Option 1 (simpler) – use a for-loop
  - Option 2 (more advanced) – use memory DSDs

- In *layout.csl*:
- TO DO 1: Define parameters for matrix dimensions

- In *run.py*:
- TO DO 1: Copy A, x, b to device via memcpy
- TO DO 2: Copy y back from device via memcpy

Hints

- Example of param from walk-through 2
- Example of memcpy from walk-through 3
- Example of memory DSD from walkthrough 4

Wash-up for hands on exercise one:

- Key points:
- Parameters: (in *pe_program.csl* and *layout.csl*):

```
// Matrix dimensions
param M: i16;
param N: i16;
```

Wash-up for hands on exercise one:

- Memcpy in the host code *run.py*

```
# Copy A, x, b to device
runner.memcpy_h2d(A_symbol, A, 0, 0, 1, 1, M*N, streaming=False,
  order=MemcpyOrder.ROW_MAJOR, data_type=MemcpyDataType.MEMCPY_32BIT, nonblock=False)
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(b_symbol, b, 0, 0, 1, 1, M, streaming=False,
  order=MemcpyOrder.ROW_MAJOR, data_type=MemcpyDataType.MEMCPY_32BIT, nonblock=False)
```

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

Wash-up for hands on exercise one: gemv() solution A

- This is the approach that will be more familiar to people
  - We use two *for* loops, looping over the column and row

```
// Compute gemv
fn gemv() void {
    for (@range(i16, M)) |i| {
        var tmp: f32 = 0.0;
        for (@range(i16, N)) |j| {
            tmp += A[i*N + j] * x[j];
        }
        y[i] = tmp + b[i];
    }
}
```

Wash-up for hands on exercise one : gemv() solution B

- However, we can potentially obtain more performance by using DSDs, as this provides the compiler with our intentions, and it is then free to implement this as it sees best

- The plan
  - Have a single *for* loop over the columns, in this we issue *@fmacs*
    - The *@fmacs* operation performs multiply-add in single precision
  - DSDs: *b_dsd* and *y_dsd* are straightforward as these access all elements
  - The *A_dsd* accesses each column (*A* is stored in row major format)

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

Wash-up for hands on exercise one : gemv() solution B

- The single for loop iterates over columns, performs the *@fmacs* operation and then increments the *A_dsd* offset
  - This increment is done so we can access the next column in the next iteration
- *@fadds* is used for the final addition of *b_dsd*

```
// Compute gemv
fn gemv() void {
  // Loop over all columns of A
  for (@range(i16, N)) |i| {
    // Calculate contribution to A*x from ith column of A, ith elem of x
    @fmacs(y_dsd, y_dsd, A_dsd, x[i]);
    // Move A_dsd to next column of A
    A_dsd = @increment_dsd_offset(A_dsd, 1, f32);
  }
  // Add b to A*x
  @fadds(y_dsd, y_dsd, b_dsd);
}
```