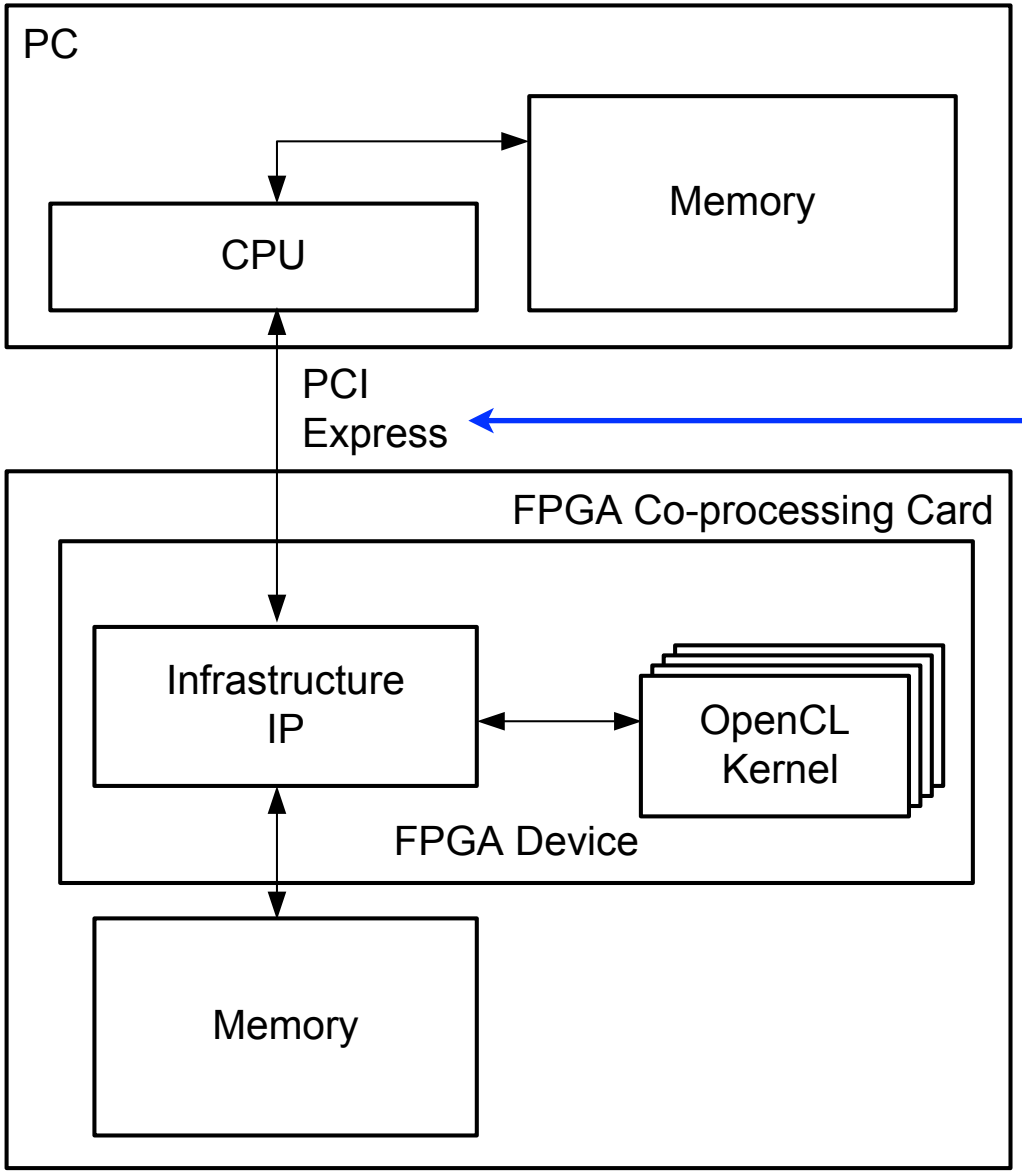
Experience with FPGA HDK AMI and F1:

(all statements are subject to large systematic uncertainties)

Nhan

SDAccel



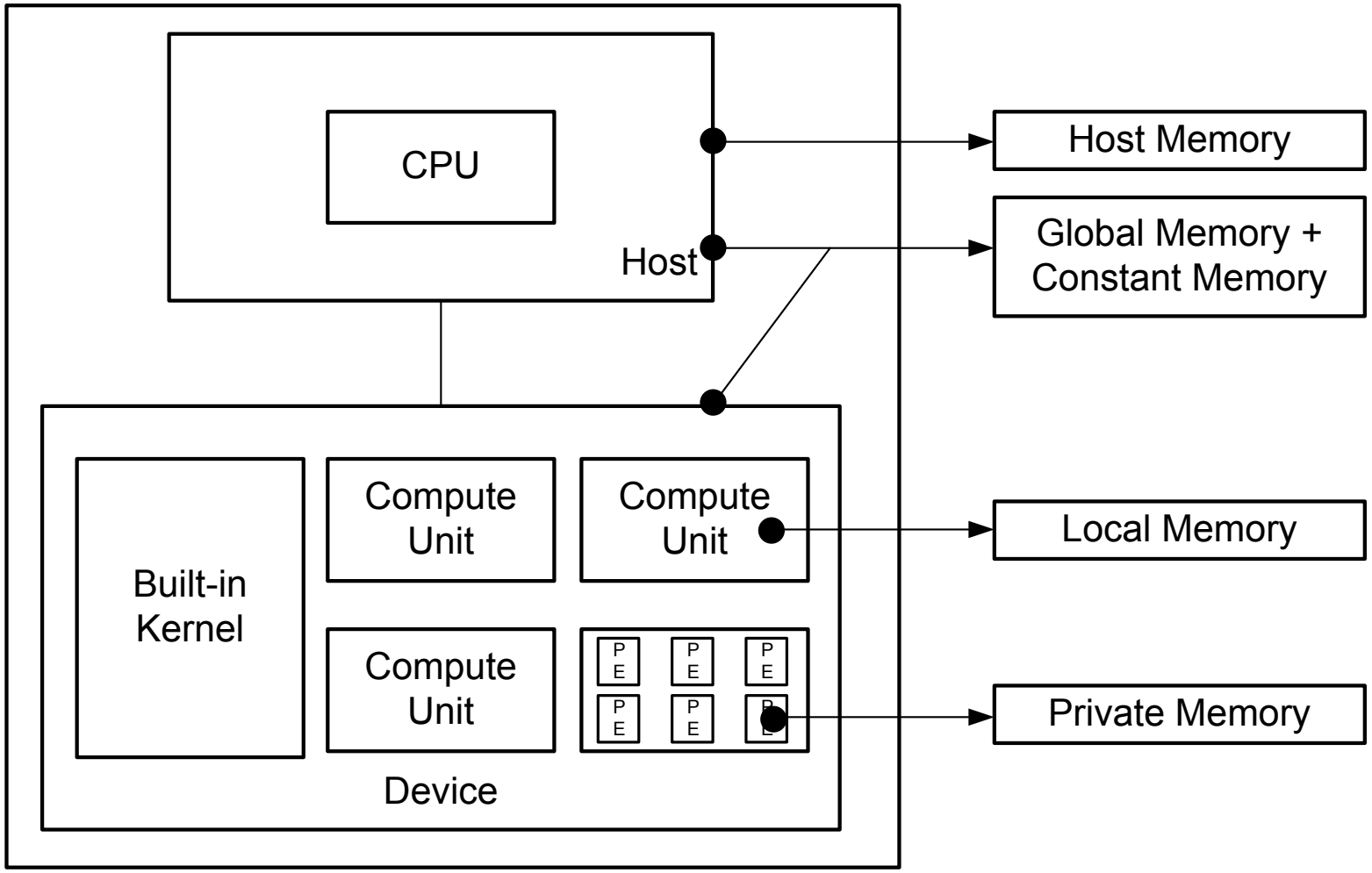
Write “host” code runs on CPU

communicates through PCIe, must be streaming (AXI)

Write “kernel” code runs on FPGA

SCAccel converts the kernel code into a form that is acceptable to the kernel compiler which is based on **Vivado HLS**

SDACCEL MEMORY MODEL



Workflow on AWS

Write the host code and kernel code on a decently powered CPU
(I'm using t2.2xlarge)

Then make the "kernel" file, upload it to some place for the f1 instance to read it and run from an f1

Setting up, see the slack post pinned to #f1-business for recipes for running:

https://github.com/Xilinx/SDAccel_Examples

WORKFLOW ON AWS

Write the host code and kernel code on a decently powered CPU

(I'm using t2.2xlarge)

Example project:

Compile the code: `make check TARGETS=hw_emu DEVICES=$AWS_PLATFORM all`

under the hood its using xocc (xilinx enabled open CL compiler?)

targets = sw_emu | hw_emu | hw

sw_emu ~ csim

hw_emu ~ csim + csynth

hw ~ make SDAccel firmware kernel (like bit file but for SDAccel platform)

KERNEL CODE (OPENCL)

memory declarations in openCL, I decided not to mess with this

“__global”
“__local”

Things that look like HLS pragmas

__attribute__((xcl_pipeline_loop))

```
kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void mmult( __global int* in1,  //Read-only input matrix1
            __global int* in2,  //Read-only input matrix2
            __global int* out,  //Output matrix
            int dim             //One dimension of the matrix
          )
{
    //Local memory to store input matrices
    //Local memory is implemented as BRAM memory blocks
    //Complete partition done on dim 2 for in1, on dim 1 for in2 and on dim 2 for out
    __local int local_in1[MAX_SIZE][MAX_SIZE] __attribute__((xcl_array_partition(complete, 2)));
    __local int local_in2[MAX_SIZE][MAX_SIZE] __attribute__((xcl_array_partition(complete, 1)));
    __local int local_out[MAX_SIZE][MAX_SIZE] __attribute__((xcl_array_partition(complete, 2)));

    //Burst reads on input matrices from DDR memory
    //Burst read for matrix local_in1 and local_in2
    read_in1: for(int iter = 0, i = 0, j = 0; iter < dim * dim; iter++, j++){
        if(j == dim){ j = 0; i++; }
        local_in1[i][j] = in1[iter];
    }
    read_in2: for(int iter = 0, i = 0, j = 0; iter < dim * dim; iter++, j++){
        if(j == dim){ j = 0; i++; }
        local_in2[i][j] = in2[iter];
    }

    //Based on the functionality the number of iterations
    //to be executed for "loop_3" must be "dim" size.
    //But for the pipeline to happen in the "loop_2" the
    //"loop_3" must be unrolled, to unroll the size cannot be dynamic.
    //It gives better throughput with usage of additional resources.
    loop_1: for(int i = 0; i < dim; i++){
        __attribute__((xcl_pipeline_loop))
        loop_2: for(int j = 0; j < dim; j++){
            local_out[i][j] = 0;
            __attribute__((opencl_unroll_hint))
            loop_3: for(int k = 0; k < MAX_SIZE; k++){
                local_out[i][j] += local_in1[i][k] * local_in2[k][ j];
            }
        }
    }

    //Burst write from local_out to DDR memory
    write_out: for(int iter = 0, i = 0, j = 0; iter < dim * dim; iter++, j++){
        if(j == dim){ j = 0; i++; }
        out[iter] = local_out[i][j];
    }
}
```

KERNEL CODE (HLS)

Turns out there are actually some HLS examples in the Xilinix SDAccel repo

e.g.

https://github.com/Xilinx/SDAccel_Examples/tree/master/getting_started/kernel_to_gmem/burst_rw_**c**

All the examples with *_c are HLS examples

KERNEL CODE (HLS)

now instead, you define the ports to the global memory using HLS pragmas

```
35  //Includes
36  #include <stdio.h>
37  #include <string.h>
38
39  //define internal buffer max size
40  #define BURSTBUFFERSIZE 256
41
42  extern "C" {
43  void vadd(int *a, int size, int inc_value){
44      // Map pointer a to AXI4-master interface for global memory access
45      #pragma HLS INTERFACE m_axi port=a  offset=slave bundle=gmem
46      // We also need to map a and return to a bundled axilite slave interface
47      #pragma HLS INTERFACE s_axilite port=a bundle=control
48      #pragma HLS INTERFACE s_axilite port=size bundle=control
49      #pragma HLS INTERFACE s_axilite port=inc_value bundle=control
50      #pragma HLS INTERFACE s_axilite port=return bundle=control
51
52      int burstbuffer[BURSTBUFFERSIZE];
53
54      //Per iteration of this loop perform BURSTBUFFERSIZE vector addition
55      for(int i=0; i < size;  i+=BURSTBUFFERSIZE)
56      {
57      #pragma HLS LOOP_TRIPCOUNT min=1 max=64
58          int chunk_size = BURSTBUFFERSIZE;
59          //boundary checks
60          if ((i + BURSTBUFFERSIZE) > size)
61              chunk_size = size - i;
62
63          //memcpy creates a burst access to memory
64          //multiple calls of memcpy cannot be pipelined and will be scheduled sequentially
65          //memcpy requires a local buffer to store the results of the memory transaction
66          memcpy(burstbuffer,&a[i],chunk_size * sizeof (int));
67
68          //calculate and write results to global memory, the sequential write in a for loop
```

HOST CODE (OPENCL/HLS)

This is the same for openCL or HLS

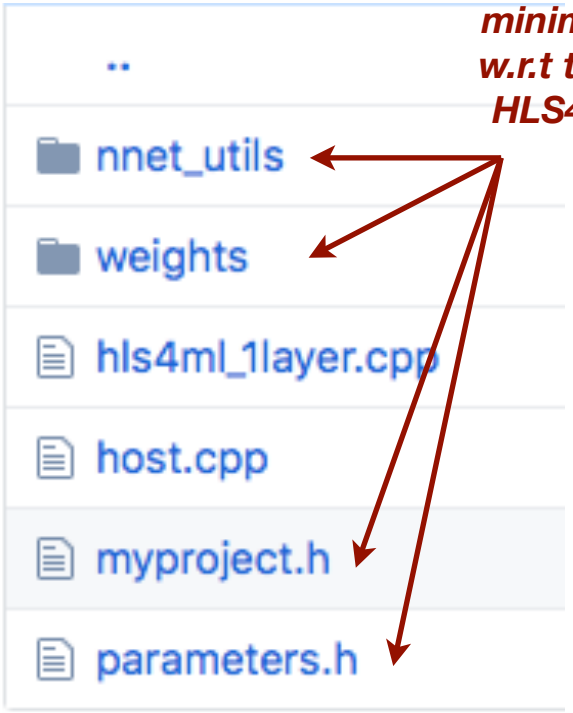
Have to be careful with defining memory buffers

```
52  //OPENCL HOST CODE AREA START
53      std::vector<cl::Device> devices = xcl::get_xil_devices();
54      cl::Device device = devices[0];
55
56      cl::Context context(device);
57      cl::CommandQueue q(context, device, CL_QUEUE_PROFILING_ENABLE);
58      std::string device_name = device.getInfo<CL_DEVICE_NAME>();
59
60      std::string binaryFile = xcl::find_binary_file(device_name,"vadd");
61      cl::Program::Binaries bins = xcl::import_binary_file(binaryFile);
62      devices.resize(1);
63      cl::Program program(context, devices, bins);
64      cl::Kernel kernel(program,"vadd");
65
66      //Allocate Buffer in Global Memory
67      std::vector<cl::Memory> bufferVec;
68      cl::Buffer buffer_rw(context, CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR,
69              vector_size_bytes, source_inout.data());
70      bufferVec.push_back(buffer_rw);
71
72      //Copy input data to device global memory
73      q.enqueueMigrateMemObjects(bufferVec,0/* 0 means from host*/);
74
75      auto krnl_add = cl::KernelFunctor<cl::Buffer&, int, int>(kernel);
76
77      //Launch the Kernel
78      krnl_add(cl::EnqueueArgs(q,cl::NDRange(1,1,1), cl::NDRange(1,1,1)),
79              buffer_rw, size, inc_value);
80
81      //Copy Result from Device Global Memory to Host Local Memory
82      q.enqueueMigrateMemObjects(bufferVec,CL_MIGRATE_MEM_OBJECT_HOST);
83      q.finish();
```

SDAccel + HLS4ML

a first working example that combines with HLS4ML

https://github.com/nhanvtran/SDAccel_Examples/tree/first-try/getting_started/host/hls4ml_1layer_hls



minimal changes w.r.t the standard HLS4ML project here

entry point to HLS4ML top function

```
92  extern "C" void hls4ml_1layer(int* a, int* c)
93  //                int n_elements_ptr)
94  {
95
96      #pragma HLS INTERFACE m_axi port=c offset=slave bundle=gmem
97      #pragma HLS INTERFACE m_axi port=a offset=slave bundle=gmem
98      #pragma HLS INTERFACE s_axilite port=c bundle=control
99      #pragma HLS INTERFACE s_axilite port=a bundle=control
100     #pragma HLS INTERFACE s_axilite port=return bundle=control
101
102     input_t arrayA[N_INPUTS];
103     result_t arrayC[N_OUTPUTS];
104
105     #pragma HLS array_partition variable=arrayA complete
106     #pragma HLS array_partition variable=arrayC complete
107
108     for (int i = 0 ; i < N_INPUTS ; i += BUFFER_SIZE)
109     {
110         int size = BUFFER_SIZE;
111         if (i + size > N_INPUTS) size = N_INPUTS - i;
112         readA: for (int j = 0 ; j < size ; j++) arrayA[j] = (input_t) a[i+j];
113     }
114
115     // addVectors(arrayA,arrayC);
116     myproject_hw(arrayA,arrayC);
117
118     for (int i = 0 ; i < N_OUTPUTS ; i += BUFFER_SIZE){
119         int size = BUFFER_SIZE;
120         for (int j = 0 ; j < size ; j++) c[i+j] = (int) arrayC[j];
121     }
122
123     return;
124 }
```

REPORTING

Because it's built all on HLS, you get the usual report files

+ Latency (clock cycles):
    * Summary:

Latency		Interval		Pipeline
min	max	min	max	Type
837	837	838	838	none

== Utilization Estimates

* Summary:

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	122	-
FIFO	-	-	-	-	-
Instance	5	32	2080	2492	-
Memory	0	-	36	9	-
Multiplexer	-	-	-	218	-
Register	-	-	249	-	-
Total	5	32	2365	2841	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	~0	~0	~0	~0	0

REPORTING

You also get this fancy HTML file that I don't know how to parse yet

OpenCL API Calls

API Name	Number Of Calls	Total Time (ms)	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
clReleaseProgram	1	9328.34	9328.34	9328.34	9328.34
clFinish	1	671.56	671.56	671.56	671.56
clCreateProgramWithBinary	1	156.974	156.974	156.974	156.974
clEnqueueTask	1	0.940704	0.940704	0.940704	0.940704
clEnqueueMigrateMemObjects	2	0.869673	0.417139	0.434836	0.452534
clCreateBuffer	2	0.787005	0.357958	0.393502	0.429047
clCreateCommandQueue	1	0.018148	0.018148	0.018148	0.018148
clCreateKernel	1	0.011873	0.011873	0.011873	0.011873
clGetExtensionFunctionAddress	1	0.010683	0.010683	0.010683	0.010683
clReleaseMemObject	4	0.00809	0.000436	0.0020225	0.00523
clSetKernelArg	2	0.00738	0.00102	0.00369	0.00636
clCreateContext	1	0.006002	0.006002	0.006002	0.006002
clGetPlatformInfo	4	0.005348	0.000527	0.001337	0.002761
clGetDeviceIDs	2	0.004628	0.000443	0.002314	0.004185
clReleaseCommandQueue	1	0.003941	0.003941	0.003941	0.003941
clRetainMemObject	2	0.003597	0.000948	0.0017985	0.002649
clReleaseKernel	1	0.003498	0.003498	0.003498	0.003498
clRetainDevice	2	0.002399	0.000637	0.0011995	0.001762
clReleaseContext	1	0.002395	0.002395	0.002395	0.002395
clGetDeviceInfo	2	0.002116	0.00073	0.001058	0.001386
clReleaseDevice	2	0.001735	0.000319	0.0008675	0.001416

Kernel Execution (includes estimated device times)

Kernel	Number Of Enqueues	Total Time (ms)	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
hls4ml_1layer	1	0.006656	0.006656	0.006656	0.006656

Compute Unit Utilization (includes estimated device times)

Device	Compute Unit	Kernel	Global Work Size	Local Work Size	Number Of Calls	Total Time (ms)	Minimun
xilinx:aws-vu9p-f1:4ddr-xpr-2pr:4.0-0	hls4ml_1layer_1	hls4ml_1layer	1:1:1	1:1:1	1	0.006648	0.006648

Data Transfer: Host and Global Memory

Context:Number of Devices	Transfer Type	Number Of Transfers	Transfer Rate (MB/s)	Average Bandwidth Utilization (%)	Average Size (KB)	Total
context0:1	READ	1	N/A	N/A	0.004	N/A
context0:1	WRITE	1	N/A	N/A	0.04	N/A

What's next?

Actually run the full chain — have to create the kernel, upload to S3 disk and then read and perform inference on the actual F1 instance

Understanding IO (Phil ++)

- There are lots of schemes (and examples) for how to control the IO in the SDAccel examples repo. Need to understand how to efficiently read the data into the FPGA — stream, burst, etc…

Dataflow

- Given an IO scheme, how do we control the data flow through the chip? All streaming/ serial? Try a pipelined setup (once data on/off-loaded)?

Build an extension of HLS4ML which makes an HLS-based SDAccel project instead of a bare HLS project?

Benchmark a more beefy network implementation against a normal CPU and GPU?

What else am I missing?