18-643 Lecture 9: Program-to-HW-Acceleration HLS

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today: understand Intel's interpretation of OpenCL for FPGAs
- Notices
  - Handout #5: lab 2, due Monday, 10/10
  - Project status report due each Friday
- Readings (see lecture schedule online)
  - for concrete reference: *Intel SDK for OpenCL: Programming Guide and Best Practices Guide*
  - *Reconfigurable Computing Architecture*, Tessier, et al., 2015
  - See also SYCL/DPC++

Khronos' OpenCL

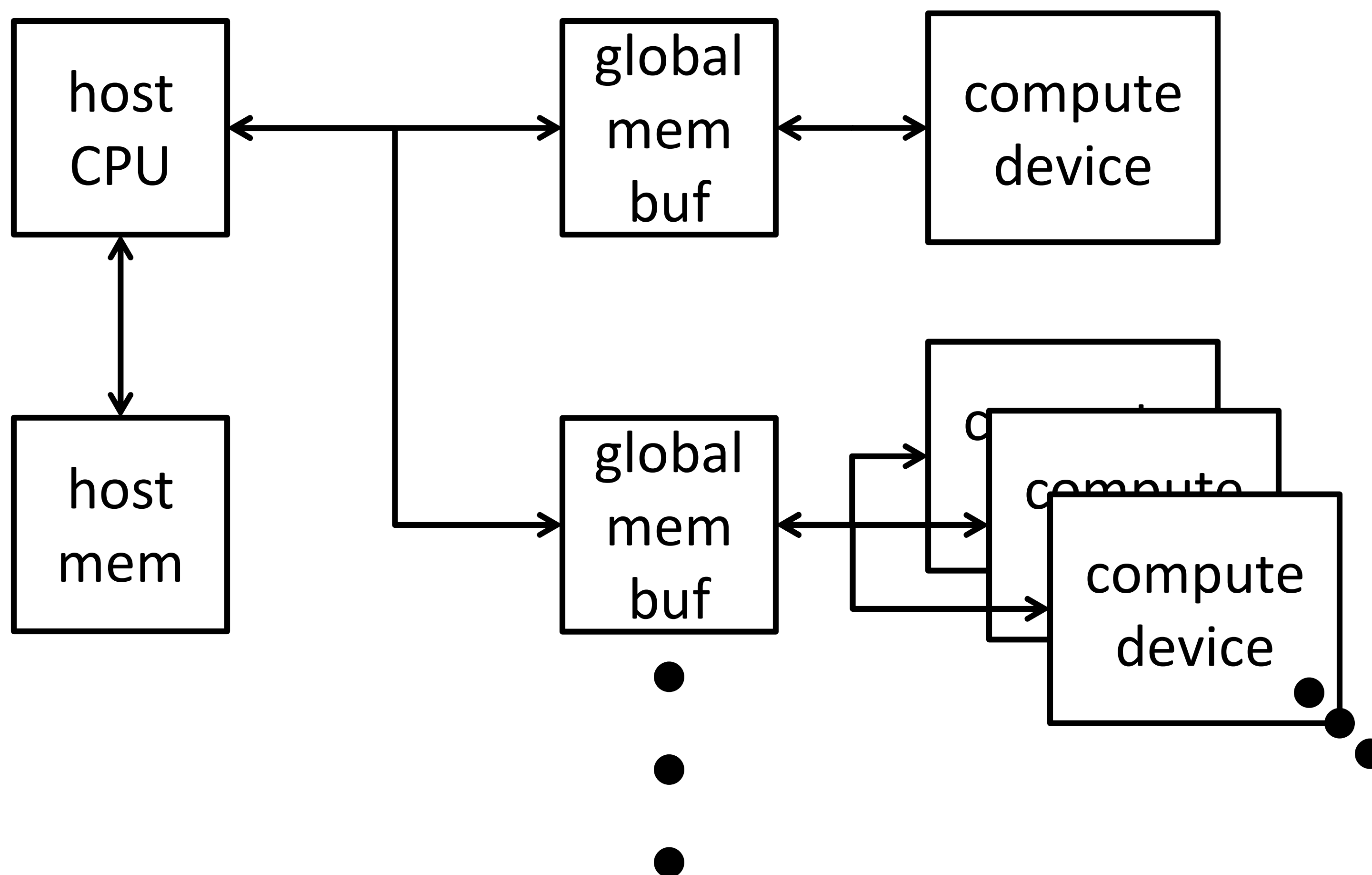
Two Parts to OpenCL

1. Platform model
   - **host** (processor & memory)
   - 1 or more accelerator **devices**

     + device-side mem hierarchy: **global**/**local**/**private**
   - APIs for **host-thread** to interact with devices
     - launch compute **kernels** to devices
     - prepare (load/unload) device memory
2. Kernel programming language
   - perfect triply-nested loops
   - no loop-carried dependence

OpenCL terms introduced in **bold**

OpenCL Platform Model



What are these "compute devices"???

Basic Host Program Example

```
main ( ) {
    . . . get device handle and queue . . .
    . . . allocate memory buf objects . . .
    . . . get kernel object. . .
    while ( ) {
        . . . initialize memory buf data . . .
        . . . bind buf objects to kernel arguments . . .
        . . . add kernel and buf objects to device queue . . .
        . . . wait for kernel to finish . . .
        . . . retrieve memory buf object for result . . .
    }
```

What are these "kernels"???

What are these kernels?

Specifically talking about <u>OpenCL C</u>

Conceptually . . .

```
for (int i=0; i < R0; i++)
  for (int j=0; j < R1; j++)
    for (int k=0; k < R2; k++) {
      << local variable declarations >>
      << arbitrary C-code with access to
         global memory >>
    }
```

- Loop body must be data-parallel
  - local variables limited to scope
  - disallow loop-carried dependencies through global memory

==> statements from different iterations can interleave in any order (using disambiguated local variables)

Concretely . . .

- Only specify loop body as a **kernel function**

```
__kernel foo(<< pointers to global mem buf>>) {
  int i=get_global_id(2), j=get…(1), k=get…(0);
  << local variable declarations >>
  << arbitrary C-code with access to
     global memory >>
}
```

- Triply-nested loops hardwired as **NDRange**
  - specified as 3 integer constants, i.e., the loop bounds (R_0, R_1, R_2)
  - 1 execution of kernel function is a **work-item**

    work-item has **private memory** for local var's
  - 1 kernel execution is $R_0 \times R_1 \times R_2$ work-items

Example: N-by-N MMM

```
__kernel mmm(__global float *A, … *B, … *C) {
  int i=get_global_id(1);
  int j=get_global_id(0);

  for(int k=0; k<N; k++)
    C[i*N+j]=C[i*N+j]+A[i*N+k]*B[k*N+j];
}
```

- NDRange=(N, N, 1)
  - kernel function executed by NxNx1 work-items
  - each work-item sees a different combination of dimension-0 and dimension-1 global id's
  - no assumption about work-items' relative progress

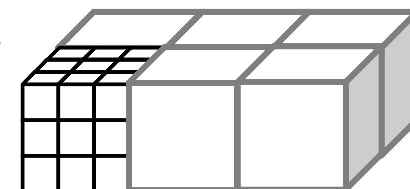
(For Your Reference: N-by-N MMM in C)

```
float A[N][N], B[N][N], C[N][N];

for(int i=0; i<N; i++)
    for(int j=0; j<N; j++) {
        for(int k=0; k<N; k++)
            C[i][j]=C[i][j]+A[i][k]*B[k][j]
    }
```

- Note:
  - Loop body of the inner-most loop is not data-parallel---dependency through `C[i][j]`
  - Loop body of the second inner-most loop is

Work-Group

- Partition NDRange of $R_0 \times R_1 \times R_2$ work-items into 3D **work-groups** of $G_0 \times G_1 \times G_2$ work-items
  - $G_{0/1/2}$ must divide $R_{0/1/2}$ evenly
  - `get_local_id(dim)`: id within group
  - `get_group_id(dim)`: id of group



NDRange=(9, 3, 6)
Group Size=(3, 3, 3)
Group Range=(3,1,2)

- Work-group signifies "locality" btw work-items
  - execute together by a **processing element**
  - can share per-group **local memory**
  - can synchronize by `barrier()`

Why do we have this?

OpenCL Kernels on GPGPUs

- Work-item is a ***CUDA thread***
- Work-group executes as a ***thread block***---broken down into 32-work-item SIMD ***Warps***
- Work-groups from same and different kernels are interleaved on a ***Streaming Processor***

  128-SIMD lanes, 1 INT+1 FMA per lane, 1.73GHz
- 1 kernel could fully consume all 20 StrmProc's (as 1 compute device), peak 8,873 GFLOPS
- Global=GDDR; local=***shared memory*** 96KB SRAM; private=***register file*** 256KB SRAM

*Nvidia terms in **italic-bold**; numbers for GTX 1080*

To fully utilize the 8,873 GFLOPS . . .

- # work-items ≥ 128 x StrmProc pipeline depth x 20
- Computation entirely of Fused-Multiply-Add insts

  Interleaved warps so no RAW stalls within a thread
- No if-then-else (branch divergence)

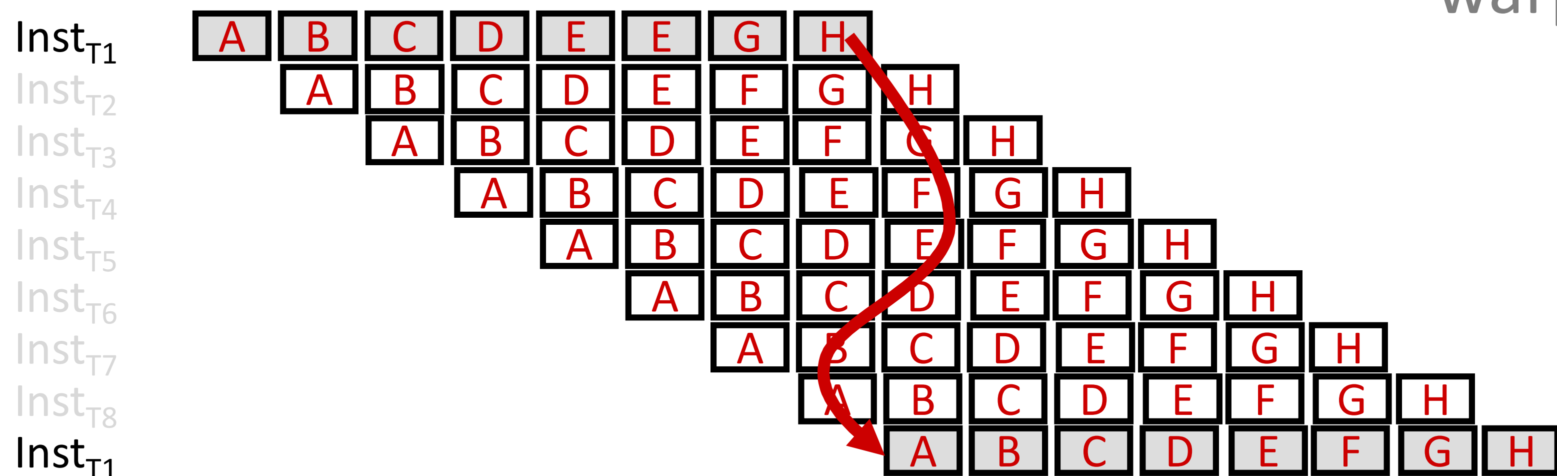
By the way:

- 320 GB/sec DRAM BW $\Rightarrow$ AI > 108 SP FP / float
- ld's and st's take up inst. issue BW off-the-top
- only certain access pattern can sustain peak BW
  - SIMD ld's and st's in a warp must go to the same memory line (memory coalescing)

Aside: Barrel Processor [HEP, Smith]

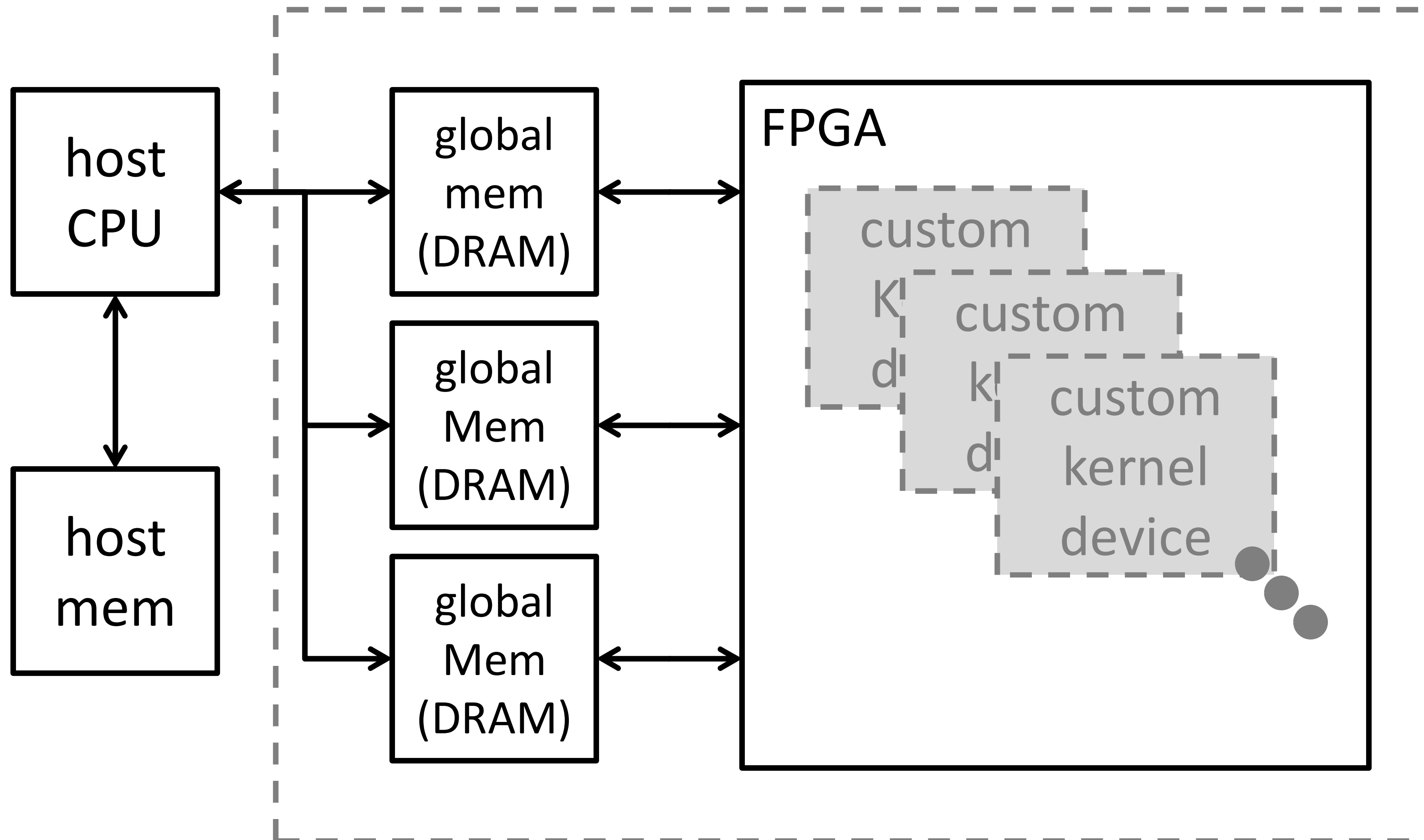
- Each cycle, select a "ready" thread from scheduling pool
  - only one instruction per thread in flight at once
  - on a long latency stall, remove the thread from scheduling
- Simpler and faster pipeline implementation since
  - no data dependence, hence, no stall or forwarding
  - no penalty in making pipeline deeper
- Cost: need to hold multiple context state

"warp"



Intel OpenCL for FPGA

OpenCL FPGA Platform Model



Compute devices synthesized from kernel functions

Example: N-by-N MM"Add"

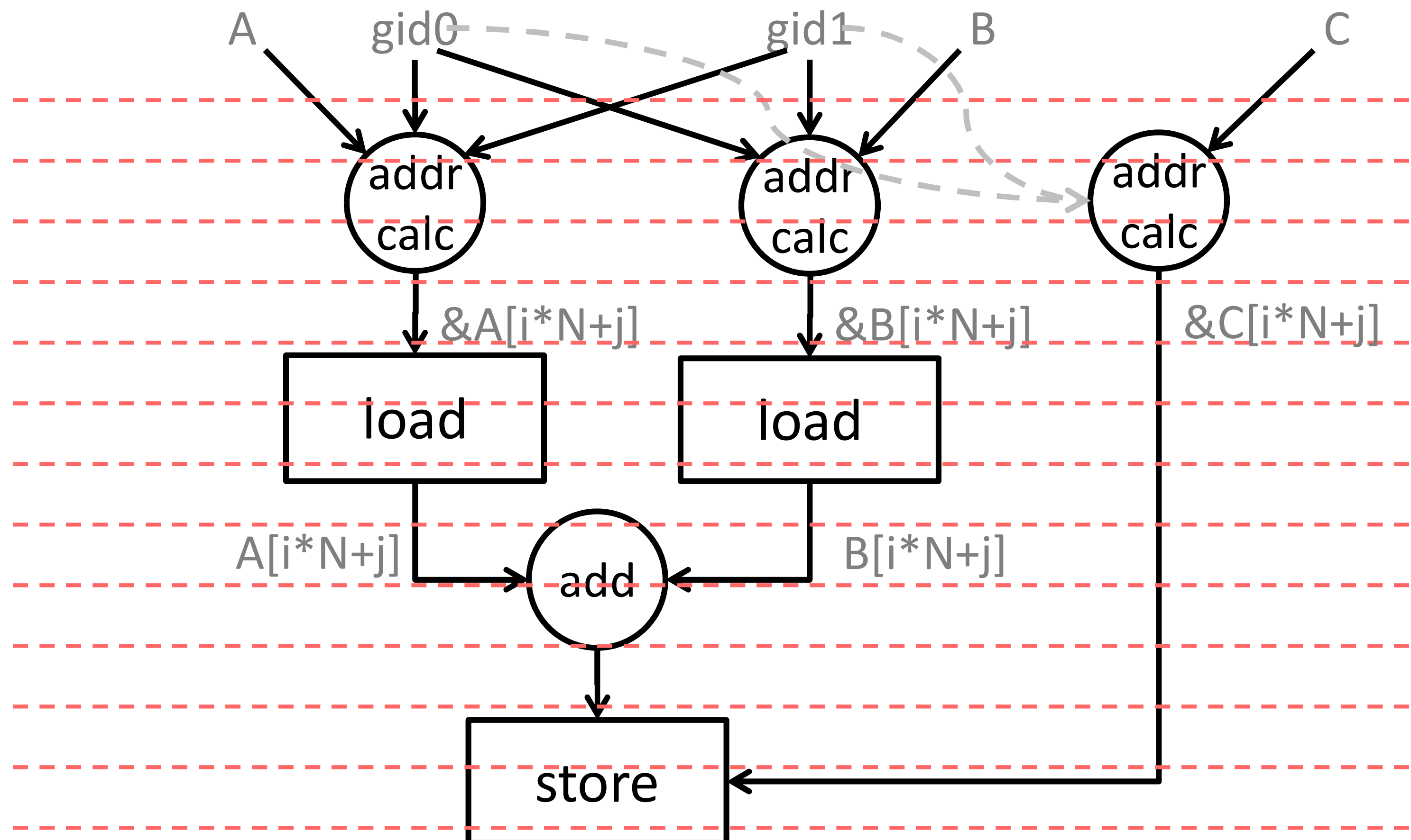
```
__kernel mma(__global float *A, … *B, … *C) {
  int i=get_global_id(1);
  int j=get_global_id(0);

  C[i*N+j]=A[i*N+j]+B[i*N+j];
}
```

- NDRange=(N, N, 1)
- Note in this example:
  - data-parallel kernel function
  - no loop in kernel function

Fully-pipelined Kernel Datapath

NDRange = (N,N,1)

(gid0,gid1) stream =(0,1),…(0,N-1),(1,0)…(1,N-1)……(N-1,0)…(N-1,N-1)



fully unrolled loop in kernel fxn also okay

What about MMM?

```
__kernel mmm(__global float *A, … *B, … *C) {
  int i=get_global_id(1);
  int j=get_global_id(0);

  for(int k=0; k<N; k++)
    C[i*N+j]=C[i*N+j]+A[i*N+k]*B[k*N+j];
}
```

- NDRange=(N, N, 1)
- Can’t easily pipeline work-items like before
- PE can unroll and pipelines the k iterations
  - dependency on C[i*N+j]
  - kernel function scope limits the tricks we can play

Try barrel pipeline like in GPU?

Single Work-Item Kernel:
clEnqueueTask( )

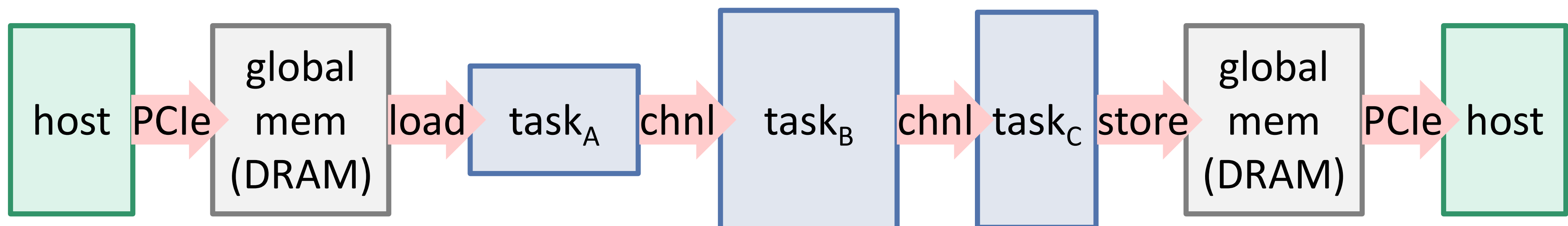
```
__kernel mmm(__global float *A, … *B, … *C) {
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
      for(in k=0; k<N; k++)
        C[i*N+j]=C[i*N+j]+A[i*N+k]*B[k*N+j]
```

- NDRange=(1, 1, 1) **never do this on GPU!!**
- Arbitrary control flow (loops, if's) and dependencies
- Becomes just "regular" C-to-HW synthesis
  - pipeline and parallelize loops
  - schedule for initiation-interval, resource, etc.

Only want OpenCL's platform model and API; "work-group" & "work-item" not too meaningful

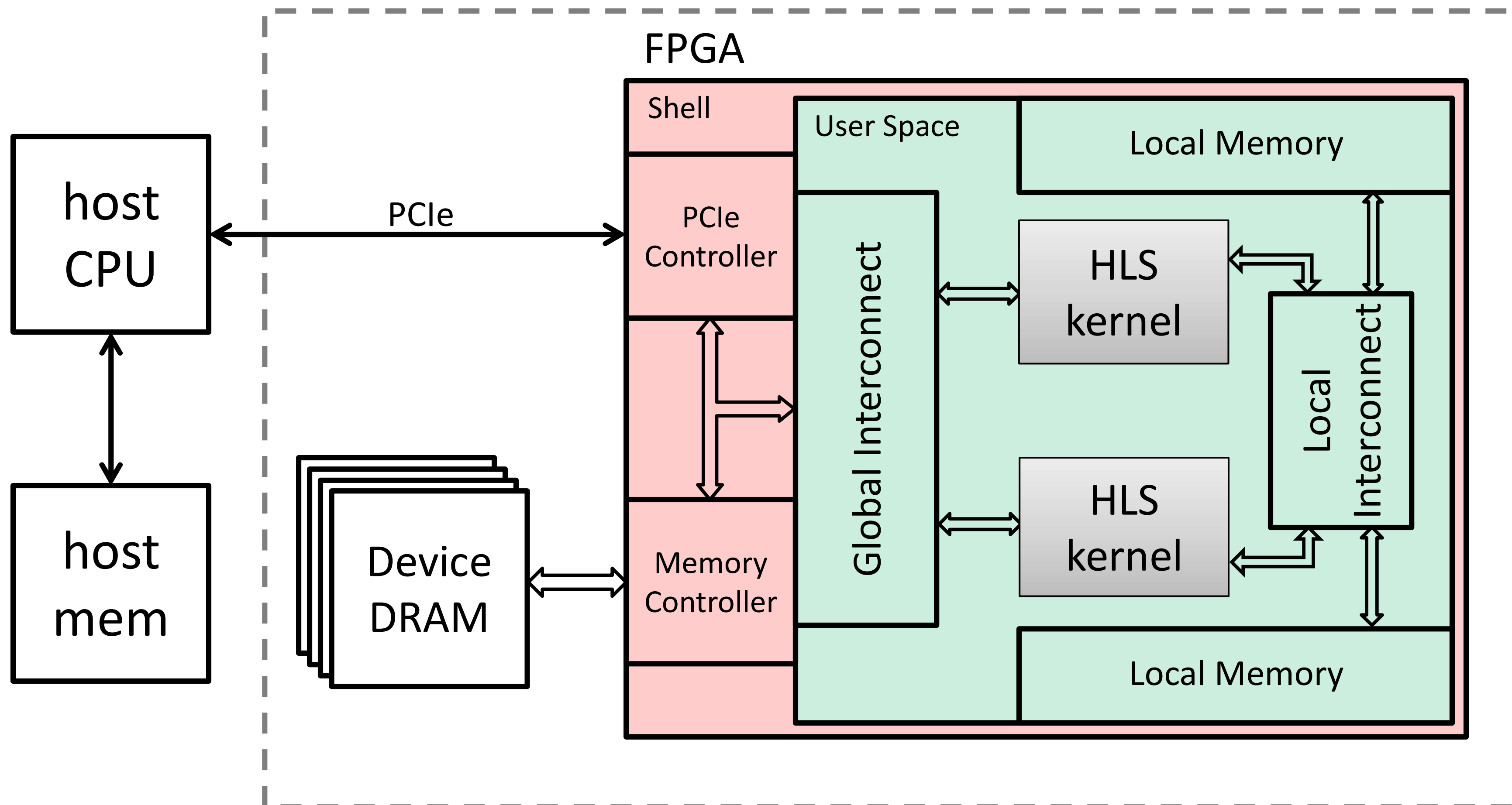
Use of Kernel-Kernel Channels

- GPU multi-kernel OpenCL program
  - kernel computes from global-mem to global-mem
  - next kernel continues from last kernel's output buf
  - producer and consumer kernels fully serialized
- For streaming processing on FPGA, connect kernels with streaming channels to bypass DRAM
  - concurrent producer and consumer kernels
  - reduce DRAM and PCIe bandwidth requirements



Different programming mindset from GPU

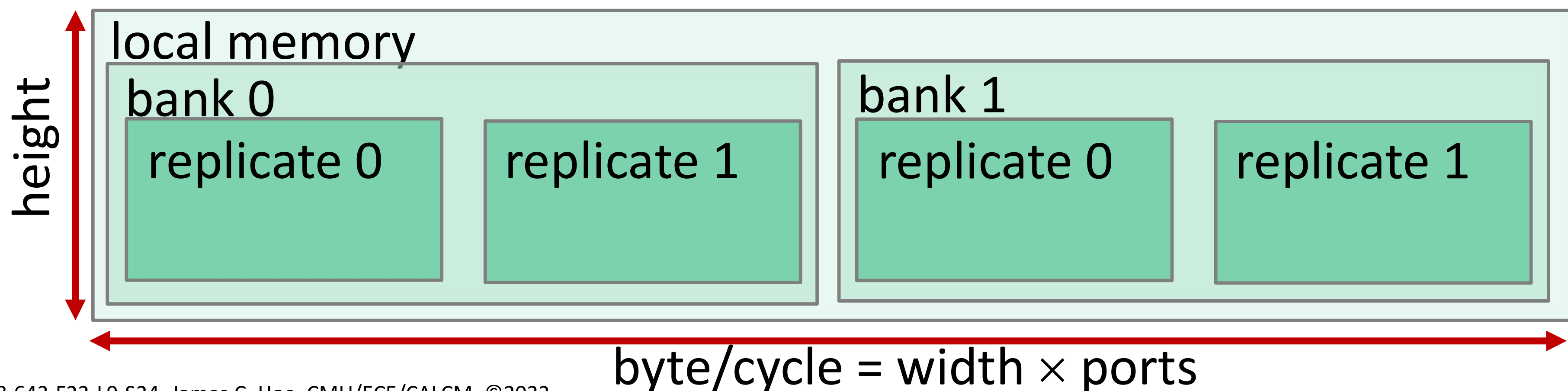
Program-to-HW, not Function-to-IP



Object of design/optimization is the entire FPGA system

Performance-Geared Synthesis

- Automatic pipelining and unrolling
  - no pipeline directive, but disable_loop_pipelining so you still have control
  - unroll factor controls extent of unrolling
- Local Memory (BRAM) Optimizations
  - if you don’t say anything, AOC does it’s best
  - banks, replicates, private copies and interconnect



Global Memory (DRAM) Optimizations

- Load-Store Units
  - automatically chosen based on access pattern

	Sequential	**Random Access**
Area Efficient	Prefetching	Pipelined
Area Expensive	Burst-Coalesced	Burst-Coalesced Cached (loads only)

- Constant cache memory
- DDR banking automatically handled

Parting Thoughts

- OpenCL = platform model/API + SIMD language
  - kernel language forces regular and explicit parallelism
  - SIMD parallelism different on GPUs vs FPGAs

  GPUs great at SIMD; FPGAs good for more than SIMD
- **FPGA OpenCL = platform model/API + "smart" memory system + "regular" kernel HLS**
  - single-work-item kernel unlocks parallelism style
  - kernel-kernel channels alleviate DRAM and PCIe bottleneck for streaming use cases
  - develop/debug/analysis tools integral to appeal
- Same tool for HPC and SDSoC, but used differently