18-643 Lecture 11: Intel OpenCL for FPGA

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today: understand Intel’s interpretation of OpenCL on FPGA
• Notices
  – Handout #4: lab 2, due noon, 10/7
  – 3 weeks to project proposal
• Readings
  – skim Ch 10, Reconfigurable Computing
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**
What are these “compute devices”???
Basic Host Program Example

```c
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 OpenCL C
Conceptually . . .

```
for (int i=0; i < R_0; i++)
    for (int j=0; j < R_1; j++)
        for (int k=0; k < R_2; 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

```c
__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**

```c
__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++)
}
```

- **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)

```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(dimen)`: id within group
  – `get_group_id(dimen)`: id of group

• Work-group signifies “locality” btwn work-items
  – execute together by a **processing element**
  – can share per-group **local memory**
  – can synchronize by `barrier()`

Why do we need this?
OpenCL Kernels on GPGPUs

- Work-items 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 can execute on upto 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*
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
To get 8,873 GFLOPS . . .

- # work-items ≥ 128 x StrmProc pipeline depth x 20
- Computation entirely of Fused-Multiply-Add
  Interleaved warps so no worries about RAW stalls
- No if-then-else (branch divergence)

**BTW:**
- ld’s and st’s take up inst. issue BW of-the-top
- 320 GB/sec DRAM BW ⇒ AI > 108 SP FP / float
  - only certain access pattern can sustain peak BW
  - SIMD ld’s and st’s in a warp has to go to the same memory line (memory coalescing)
Intel OpenCL for FPGA
OpenCL FPGA Platform Model

Compute devices synthesized from kernel functions
Example: N-by-N MM “Add”

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

- NDRRange=(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 = \((0,1),...,(0,N-1),(1,0)...(1,N-1)\ldots(N-1,0)...(N-1,N-1)\)

Fully unrolled loop in kernel fxn also okay
What about MMM?

```c
__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++)
}
```

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

```c
__kernel mmm(__global float *A, ... *B, ... *C) {
    for(int i=0; i<N; i++)
        for(int j=0; j<N; j++)
            for(int 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
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

Also supports OpenCL 2.0 Pipes
Parting Thoughts

• OpenCL = platform model/API + SIMD language
  – kernel language forces regular and explicit parallelism
  – SIMD parallelism manifest differently on GPUs vs FPGAs

• FPGA OpenCL = platform model/API + regular 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

GPUs great at SIMD; FPGAs good for more than SIMD