INTEL CCI: CORE CACHE INTERFACE

Michael Adler
2017/09
Outline

- System overview
- Core Cache Interface (CCI-P) abstraction
- Application-specific memory hierarchies (Memory Properties Factory – MPF)
- Clocking
- Simulation (ASE)
- GitHub open source repositories
OPAE: Open Programmable Acceleration Engine
- FPGA discovery and I/O configuration
AFU: Accelerated Function Unit (User IP)
- Also knows as “Green Bitstream”
FIU: FPGA Interface Unit (Intel-provided)
- “Blue Bitstream”
FPGA connects to system I/O controllers via one or more physical channels
CCI exposes physical channels as a single, multiplexed interface
Broadwell Platform

Arria10 GX1150

14 Cores / 2.4 GHz
Broadwell Platform

Arria10 GX1150

14 Cores / 2.4 GHz
CPU Coherence Domains

- The cache hierarchy is coherent
- CPU registers are outside the coherence domain
  - Explicit load/store by CPU to/from registers
  - No shoot-down of register state from cache activity
FPGA QPI Coherence

- Caches are coherent
- BRAM & LUTs are not
  - They are like CPU registers
  - Load/store to/from them explicitly
  - No shoot-down from cache
FPGA PCIe Coherence

- The CPU's PCIe controller is coherent!
- Load/store requests reaching the PCIe controller from the FPGA are coherent with system memory and CPU caches
- BRAM & LUT behavior is unchanged: updates are explicit
- No shoot-down from any coherence engine
System Coherence

- All 3 FPGA channels serviced coherently
- QPI channel handles polling loops locally
CORE CACHE INTERFACE

CCI-P
We will focus on CCI and AFUs

CCI: FIU / AFU Interface

FPGA

AFU

FIU

Physical Channel

Physical Channel
Motivation

- Accelerator writers don’t want to spend 6 months getting PCIe to work
- A memory bus expert will do a better job
- Platform interfaces should be consistent across a family of devices

- These principles have been established on CPUs for over 40 years!
  - libc’s API keeps “Hello World!” working despite major kernel changes:
    - New instruction sets and calling conventions
    - Virtual memory
    - New file systems
    - ...
CCI is an Abstract Interface

- The slides that follow rarely mention specific physical channels
- The interface is largely independent of:
  - The number of physical channels
  - The bus protocols of physical channels
  - Whether physical channels have FPGA-side caches
CCl Components

- Discovery & control
  - Identify accelerators
  - Identify features within an accelerator
  - AFU command and control
  - AFU statistics & state flags

- Host memory access
  - Read & write
  - Consistency

FPGA-side Address Space

Host-side Address Space
DISCOVERY AND CONTROL

MMIO & CSRs
Discovery & Control: MMIO

- **AFU-side 256KB address space**
- **Software requests reads and writes to MMIO address space**
- **Perhaps better to call it control/status registers (CSRs)**
  - MMIO comes from a software-centric view
  - Semantics are like CSRs: doesn’t actually have to be a memory!
    - MMIO space may be implemented sparsely in AFU
    - Support whatever is needed to control a given AFU
    - MMIO write values don’t have to be reflected in MMIO read responses
Discovery: Device Feature List

- Linked list in MMIO space
- Each object in the list begins with a “Device Feature Header” (DFH):
  - Pointer to next DFH
  - Feature UUID
- AFU DFH & UUID are mandatory at address 0
- Software uses the AFU UUID to identify accelerator instances
- Software may use feature UUIDs to guide interaction
- See CCI-P documentation for full format
Conventions:
- CSR address range wrapped in device feature lists
- Associate CSR with feature UUID
- Read & write semantics of a given CSR are application-defined

Mostly unenforced – just convention. The only true requirements right now are:
- Correctly formed feature list
- Valid AFU UUID

Future hierarchical designs may depend on well-wrapped CSR ranges
Typical Discovery & Control Protocols

- OPAE library searches device feature list for an AFU UUID
- User software or libraries walk feature list to detect capabilities
- User software calls fpgaWriteMMIO64() to set state and start AFU
- Execute
- User software calls fpgaReadMMIO64() to read statistics or other AFU state
HOST MEMORY ACCESS

CCI-P Memory Reads & Writes
CCI: Split Request & Response Channels

- All request wires flowing toward the FIU are wrapped in struct t_if_ccip_Tx
- All response wires from the FIU are wrapped in struct t_if_ccip_Rx

// CCI-P Tx Port
output t_if_ccip_Tx pck_af2cp_sTx,

// CCI-P Rx Port
input t_if_ccip_Rx pck_cp2af_sRx
Request & Response Structures Contain Channels

- Reads are always in c0
  - Read request in Tx c0
  - Read response (with data) in Rx c0

- Writes are always in c1
  - Write request (with data) in Tx c1
  - Write response ACK in Rx c1

- Host-generated MMIO is asymmetric
  - Read & write requests arrive on Rx c0
    (Sharing data wires with read response)
  - AFU responds on Tx c2

- Rx AlmFull signals govern Tx requests

typedef struct packed {
  t_if_ccip_c0_Tx  c0; // Read
  t_if_ccip_c1_Tx  c1; // Write
  t_if_ccip_c2_Tx  c2; // MMIO Rsp
} t_if_ccip_Tx;

typedef struct packed {
  t_if_ccip_c0_Rx  c0; // Read
  t_if_ccip_c1_Rx  c1; // Write

  logic c0TxAlmFull;
  logic c1TxAlmFull;
} t_if_ccip_Rx;
Read Request (c0 Tx)

- c0 Tx valid bit indicates active request this cycle

- Request type:
  - eREQ_RDLINE_I
    - Read a line and don’t insert in FIU cache
    - Streaming reads
  - eREQ_RDLINE_S
    - Read a line and maybe insert in FIU cache
    - FIU will decide
  - Both options behave the same on physical channels without FPGA-side caches

```c
typedef struct packed {
    t_ccip_vc vc_sel;
    t_ccip_clLen cl_len;
    t_ccip_c0_req req_type;
    t_ccip_c1Addr address;
    t_ccip_mdata mdata;
} t_ccip_c0_ReqMemHdr;

logic valid;
```
The FIU expects physical addresses
- Buffer physical addresses are returned by OPAE's fpgaGetIOAddress()
- We will talk about transforming CCI to allow an AFU to use virtual addresses later (see MPF VTP)

Addresses are cache-aligned (64 bytes)
- Drop the low 6 bits of a byte-level address
- Responses always return a full cache line

typedef struct packed {
    t_ccip_vc vc_sel;
    t_ccip_c1Len c1_len;
    t_ccip_c0_req req_type;
    t_ccip_c1Addr address;
    t_ccip_mdata mdata;
} t_ccip_c0_ReqMemHdr;

logic valid;
Multi-line requests improve throughput
- Some physical channels (e.g. PCIe) are more efficient with payloads larger than one line

1, 2 or 4 lines

Address must be aligned to the size
- Low bit 0 for 2 line requests
- Low two bits 0 for 4 line requests

Read response arrives as multiple beats
- Each beat is a single line
- Lines may arrive out of order! (More on this later)

typedef struct packed {
    t_ccip_vc vc_sel;
    t_ccip_clLen cl_len;
    t_ccip_c0_req req_type;
    t_ccip_clAddr address;
    t_ccip_mdata mdata;
} t_ccip_c0_ReqMemHdr;

logic valid;
vc_sel: AFU control over physical channel

- Four options on Broadwell:
  - Any of the 3 physical channels (QPI, PCIe0/1)
  - “Any” (eVC_VA): FIU picks least busy channel
- Usually best to pick eVC_VA
  - Balancing channels is difficult!
  - Works on any physical platform
  - Only exception: use eVC_VL0 (QPI) for polling a cache line to avoid off-FPGA traffic
  - See MPF VC Map section for more details

```c
typedef struct packed {
  t_ccip_vc vc_sel;
  t_ccip_clLen cl_len;
  t_ccip_c0_req req_type;
  t_ccip_clAddr address;
  t_ccip_mdata mdata;
} t_ccip_c0_ReqMemHdr;

logic valid;
```
mdata: Sorting Read Responses

- Read responses are unordered! (More about this later in MPF ROB)
- mdata from requests is returned along with read responses
  - Send a unique mdata for each outstanding read request
  - Use mdata response to match response and request
  - Multi-line requests share mdata

```c
typedef struct packed {
    t_ccip_vc      vc_sel;
    t_ccip_clLen   cl_len;
    t_ccip_c0_req  req_type;
    t_ccip_clAddr  address;
    t_ccip_mdata   mdata;
} t_ccip_c0_ReqMemHdr;

logic valid;
```
Write Requests Nearly Identical to Reads (c1 Tx)

- data passed in c1 Tx with header
- sop: Start of packet
  - Set on first line in every group
  - Clear on later lines in multi-line groups

```c
typedef struct packed {
   t_ccip_vc vc_sel;
   logic sop;
   t_ccip_clLen cl_len;
   t_ccip_c1_req req_type;
   t_ccip_clAddr address;
   t_ccip_mdata mdata;
} t_ccip_c0_ReqMemHdr;

t_ccip_clData data;
logic valid;
```
Pipeline for Performance

- Maximum throughput requires many requests in flight
- Number in flight depends on:
  - Physical channel properties of a specific platform
  - Number of lines in AFU-generated requests (cl_len field)
- Broadwell with 400 MHz AFU:
  - Peak read performance with about 400 lines in flight
  - Peak write performance with about 150 lines in flight
Response Channels

- c0 read response channel returns read data
  - One line at a time

- c1 write response channel returns write ACKs
  - Indicates write has been committed by the FIU to an ordered physical channel
  - Does not indicate write has reached host memory
Memory is Almost Completely Unordered!

- Only guarantee: responses arrive after requests are generated 😛
- Otherwise unordered, even to same address:
  - Reads and writes unordered
  - Two unacknowledged writes to same address and channel may swap order
  - Two writes to same address and different physical channels have no defined order
    - Even if one has already returned its ACK!
    - This includes eVC_VA
    - WrFence on c1 Tx forces channel synchronization, but is slow
  - See MPF VC Map for a solution
ARE WE CRAZY?
FPGA Design Philosophy

- Standard platform code should:
  - Provide base services and semantics needed by all applications
  - Consume as little FPGA area as possible

- FPGAs overcome a huge frequency disadvantage relative to CPUs through application-specific, spatial solutions

- Application-specific memory semantics are part of this solution
  - Instantiate exactly the memory semantics required
  - Avoid wasting FPGA resources on unnecessary memory management
CCI – Lowest Common Denominator

- Simple semantics are most efficient
  - E.g. PCIe channel has no mechanism for ACK on reaching system memory
  - PCIe is ordered though, so CCI ACK on channel entry guarantees intra-channel order
  - Sorting read responses costs area and latency

- These minimal requirements satisfy major classes of algorithms, e.g.:
  - Double buffered kernels that read from and write to different buffers
  - Streaming kernels that read from one memory-mapped FIFO and write to another
Higher Level Memory Services and Semantics

- Some applications need one or more of:
  - Virtually addressed memory for large, contiguous buffers or sharing pointers with software
  - Ordered read responses
  - Write/read memory consistency guarantees
  - ...
- Applications may not need all of these attributes
- Can we construct a custom memory interface specific to each AFU?
MPF: MEMORY PROPERTIES FACTORY

Basic Building Blocks (BBB)
MPF – Memory Properties Factory

- Provides a common collection of memory semantic extensions to CCI
- Applications instantiate only the semantics they require
- Each MPF block is implemented as a CCI to CCI shim
  - Consume CCI requests
  - Implement some feature (e.g. translate virtual addresses to physical)
  - Produce transformed CCI requests
- Application-specific memory hierarchies are formed by composing MPF shims
Base Core Cache Interface (CCI)

- Low-level CCI
  - Physical addresses
  - No ordering
CCI With MPF

- Low-level CCI
  - Physical addresses
  - No ordering

- AFU may instantiate MPF as a CCI to CCI bridge, maintaining the same interface but adding new semantics
MPF Composable Shims

All MPF shims may be enabled or disabled independently and run at full speed:

- VTP: Virtual to physical address translation
- ROB: Reorder buffer to sort read responses and return them in request order
- VC Map: Map requests to system memory channels explicitly
- WRO: Intra-line write/read ordering
- PWRITE: Partial (masked) write emulation using read-modify-write

Note: Some shims depend on other shims, e.g:
- WRO on VC Map
- PWRITE on WRO
CCI Memory Spaces

- OPAE allocates shared memory in process virtual space
- Physical memory pinned to I/O space
- FIU performs no address translation
  - AFU requests host physical addresses
  - FIU emits host physical addresses
CCI Memory Spaces

- OPAE allocates shared memory in process virtual space
- Physical memory pinned to I/O space
- FIU performs no address translation
  - AFU requests host physical addresses
  - FIU emits host physical addresses
- Virtual addresses in AFU require translation
  - MPF VTP (Virtual to Physical) acts as a TLB
  - Accepts process virtual addresses from AFU
  - Translates to host physical addresses
VTP: Virtual to Physical

- Resembles a traditional TLB
  - Separate translation tables for 4KB and 2MB pages
  - Level 1: 512 entry TLB, direct mapped, one per request channel
  - Level 2: 512 entry four-way set-associative TLB, shared across all channels
  - Hardware, caching page table walker

- Size and associativity of each table is configurable

- No prefetcher: we have not found a need
  - L2 Misses are rare with 2MB pages
  - MPF's caching page table walker generates only one memory read per 16MB of memory with stride one streams and 2MB pages
VTP Micro-Architecture

- L1 hits flow around L1 misses
- Separate direct-mapped L1 cache for reads and writes, 2MB and 4KB pages
- Shared set-associative L2 cache for reads and writes, separate 2MB and 4KB
- Page table walker is not pipelined but does cache recent page table lines
- Default sizes are optimized for M20k minimum depth and bandwidth
  - Programs with some page locality should require no VTP tuning
  - Programs with completely random access patterns to very large footprints may benefit from larger caches. See hw/rtl/cci_mpf_config.vh.
VTP Software Component

- VTP maintains its own page table, shared with the VTP FPGA shim
- Applications allocate & deallocate memory with VTP software service:
  - mpfVtpBufferAllocate()
  - mpfVtpBufferFree()
  - The VTP page table is updated as a side effect of allocation
  - Allocation & deallocation may occur at any time during a run
  - See test/test-mpf/base/sw/opae_svc_wrapper.cpp
ROB: Reorder Buffer

- CCI returns read responses unordered
  - AFU tags read requests with a unique number
  - FIU returns the tag with the response
- ROB sorts read responses
  - Eliminates need for AFU tagging
  - CCI reads behave more like FPGA-attached DRAM
- ROB is sized to enable maximum bandwidth
- ROB adds latency, especially when physical channels have different latencies
VC Map: Channel Sync

- Channels multiplexed as a single bus in CCI-P
  - Addressable independently using vc_sel field in request header
  - Ganged together by blue bitstream as a single high-bandwidth logical channel with eVC_VA tag

- CCI-P channels have deliberate races:
  - No guarantee that reads to one channel will return the result of a write to a different channel, even when the write has already returned an ACK!
  - Consistent with design philosophy: base platform supports only universal requirements. Races are no problem when streaming or double buffering.
  - Write fence to eVC_VA channel synchronizes all channels but is too slow for frequent use.
VC Map: Address-Based Host Channel Mapping

- VC Map avoids inter-channel races:
  - AFU passes requests to MPF using eVC_VA, the same mechanism for CCI mapping
  - VC Map selects explicit physical channels before routing requests to CCI
  - Channel mapping is a function of a request’s address: a given address is always mapped to the same channel

- AFUs that enable WRO almost always required VC Map
VC Map Optimizations

- Channel mapping is surprisingly complicated
  - Optimal throughput is achieved only when requests are balanced across physical channels
  - Optimal request rate balance varies with the types and sizes of requests

- VC Map dynamically responds to AFU-generated traffic, picking the optimal request rates to each physical channel

- VC Map may choose to rebalance traffic as request patterns vary. It must:
  - Stop all AFU traffic by asserting Almost Full
  - Wait for all current traffic to retire
  - Emit a write fence
WRO: Write/Read Ordering

- CCI provides no intra- or inter-line order guarantees
  - Even conflicting writes are unordered
- CCI leaves synchronization to the AFU: track write ACKs or use fences
- Fences are slow!
  - No problem for kernels that:
    - Maintain discrete read and write spaces
    - Write each address only once
    - Emit fences infrequently
  - Avoid fences when they would be required frequently
WRO: Write/Read Ordering

- WRO guarantees that requests within a line complete in order
  - Writes to the same line retire in order
  - Reads always return the most recently written value
  - Reads have priority when arriving in the same cycle as a conflicting write
  - Still no guarantees about inter-line consistency!

- Write/read hazard detection is implemented as a collection of filters
  - CAMs would be too expensive to support the number of requests in flight
  - Filters are sized to balance FPGA area against the rate of false conflicts
  - Multiple reads to the same location are permitted
  - Filter sizes are configurable
PWRITE: Partial Write Emulation

- CCI currently provides no mechanism for masked writes

- PWRITE emulates masked writes:
  - Reading the requested line
  - Updating the masked bytes
  - Writing the merged data
  - MPF extends the write request header with byte mask bits

- PWRITE does not lock the line. It is not an atomic operation!
  - Conflicting CPU stores in the middle of a PWRITE sequence may be lost
  - WRO and VC Map may be used to guarantee order within the FPGA
BUILDING WITH MPF
Instantiating MPF in an AFU

- See tutorial example `hello_world_mpf`
- MPF uses SystemVerilog interfaces to represent CCI-P wires
  - 1:1 mapping from CCI-P structs to `cci_mpf_if` buses
  - MPF shims have multiple CCI-P buses: one toward AFU, one toward FIU
  - Interfaces simplify arguments to MPF shims
  - MPF module `ccip_wires_to_mpf()` converts CCI-P wires to a `cci_mpf_if`
Instantiate MPF –

`hw/rtl/cci_mpf.sv` has extensive comments

ci_mpf_if afu(.clk(pClk));

cci_mpf

#(
  .SORT_READ_RESPONES(1),
  .PRESERVE_WRITE_MDATA(0),
  .ENABLE_VTP(1),
  .ENABLE_VC_MAP(0),
  .ENABLE_DYNAMIC_VC_MAPPING(1),
  .ENFORCE_WR_ORDER(0),
  .ENABLE_PARTIAL_WRITES(0),
  .DFH_MMIO_BASE_ADDR(MPF_DFH_MMIO_ADDR)
)

mpf

(  .clk(pClk),
  .fiu,
  .afu
);
MPF Internal Configuration Options

- Some applications, especially those with large memory footprints, may benefit from resizing MPF structures

- `hw/rtl/cc_i_mpf_config.vh` defines and documents many options
  - Configure VTP TLB size and associativity
  - Configure WRO hashing parameters

- These options may be set without modifying source, e.g.
  ```
  set_global_assignment -name VERILOG_MACRO "VTP_N_C0_TLB_2MB_SETS=1024"
  ```
CLOCKING
Multiple Incoming Clocks

- CCI Tx/Rx run at pClk
- CCI asynchronous clock crossing shim provided as a BBB (see tutorial)
- Quartus does not allow clocks to be defined in a partial reconfiguration region
  - uClk_usr is a substitute: configurable frequency
  - Must adjust place & route constraints for a particular uClk_usr frequency

```vhdl
input logic pClk, // 400MHz - (Primary)
input logic pClkDiv2, // 200MHz - (Aligned with pClk)
input logic pClkDiv4, // 100MHz - (Aligned with pClkDiv2)
input logic uClk_usr, // User configurable
input logic uClk_usrDiv2, // Half speed (aligned with uClk_usr)

input logic pck_cp2af_softReset, // CCI-P ACTIVE HIGH Soft Reset
```
AFU Simulation Environment
Abstract CCI Simulation with ASE

- Emulates hardware-independent CCI
  - Applications link against an ASE version of OPAE library
    - OPAE interface unchanged
    - Programs should work with FPGA and ASE without modification
    - One exception: MMIO
      - ASE can't detect CPU's direct store/load to/from MMIO
      - Must use access functions (e.g. fpgaWriteMMIO64)
  - Relatively fast – emulates the API, not a particular platform

- ASE aims to help validate programs
  - Variable response latency
  - Out of order responses
ASE Examples

- Tutorial samples in BBB release demonstrate ASE configuration and use
- ASE configuration
  - ASE provides a `generate_ase_environment.py` script that attempts to auto-configure an environment – ok for basic configurations
  - Tutorials use an explicit, hierarchical configuration
    - Works with RTL libraries like BBBS
    - Wrapper around `generate_ase_environment.py`
    - Easy to map to new projects
OPEN SOURCE
GitHub: OPAE (https://github.com/OPAE/)

- User space libraries & ASE: opae-sdk
- Basic building blocks (MPF, clock crossing, etc.): intel-fpga-bbb
- Tutorial & examples in Basic Building Blocks: samples/tutorial