Berkeley Architecture Research

Gemmini AuRORA Integration

Seah Kim
UC Berkeley
seah@berkeley.edu

View AuRORA SoC Configs

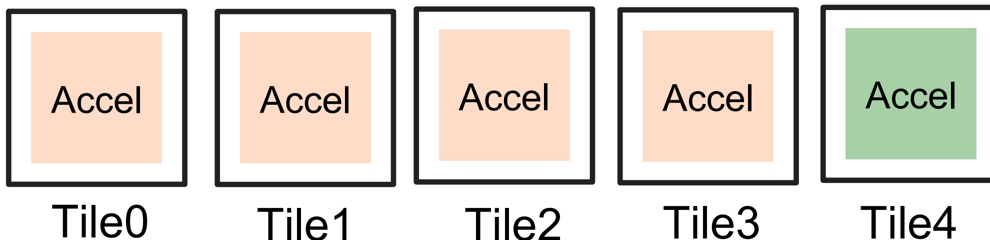
```
cd $MCYDIR/generators/chipyard/src/main/scala

vim configs/RoCCAcceleratorConfigs.scala
```

```scala
// ReRoCC Integration
class TutorialGemminiReRoCCConfig extends Config(
  new rerocc.WithReRoCC ++
  new gemmini.LeanGemminiConfig ++                                   // rerocc tile3 is gemmini
  new gemmini.LeanGemminiConfig ++                                   // rerocc tile2 is gemmini
  new gemmini.LeanGemminiConfig ++                                   // rerocc tile1 is gemmini
  new gemmini.LeanGemminiConfig ++                                   // rerocc tile0 is gemmini
  new freechips.rocketchip.rocket.WithNHugeCores(2) ++
  new chipyard.config.WithSystemBusWidth(128) ++
  new freechips.rocketchip.subsystem.WithoutTLMonitors ++
  new chipyard.config.AbstractConfig)
```

View AuRORA SoC Configs

```
cd $MCYDIR/generators/chipyard/src/main/scala

vim configs/RoCCAcceleratorConfigs.scala
```

```scala
class ReRoCCTestConfig extends Config(
  new rerocc.WithReRoCC ++
  new chipyard.config.WithCharacterCountRoCC ++        // rerocc tile4 is charcnt
  new chipyard.config.WithAccumulatorRoCC ++           // rerocc tile3 is accum
  new chipyard.config.WithAccumulatorRoCC ++           // rerocc tile2 is accum
  new chipyard.config.WithAccumulatorRoCC ++           // rerocc tile1 is accum
  new chipyard.config.WithAccumulatorRoCC ++           // rerocc tile0 is accum
  new freechips.rocketchip.rocket.WithNHugeCores(1) ++
  new chipyard.config.AbstractConfig)
```



AuRORA Architectural Extensions

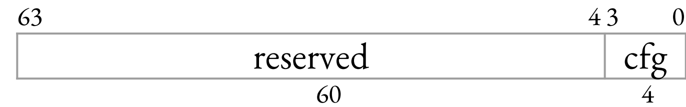
Platform-level:

Assumes a global physical ID-space of up to 256 remotely-attached RoCC accelerators
Platform should encode the physical ID-space

ISA-level:

`rr_acquire, rr_release`

`rr_set_opc`



CSRs: `rrcfg0 - rrcfgX`

- State indicates which system-wide physical accelerators are locked to this thread
- Writes acquire/release shared accelerators
- <100 bits of additional state

CSRs: `rropc0 - rropc3`

- Set which accelerator (cfg) should receive instructions of this opcode
- Enables virtualization of the opcode/accelerator space
- 16 bits of additional state

View ReRoCC Header

```
cd $MCYDIR/generators/aurora
```

```
vim aurora-sw/rerocc.h
```

```c
static bool rr_acquire(uint32_t cfgId, uint64_t accelId) {
  uint32_t csrid = CSR_RRCFG0 + cfgId;
  uint64_t w = RR_CFG_ACQ_MASK | (accelId & RR_CFG_MGR_MASK);
  write_rr_csr(csrid, w);
  return (read_rr_csr(csrid) & RR_CFG_ACQ_MASK) != 0;
}

static void rr_release(uint32_t cfgId) {
  uint32_t csrid = CSR_RRCFG0 + cfgId;
  uint64_t w = 0;
  write_rr_csr(csrid, w);
}

static void rr_set_opc(uint8_t opc, uint32_t cfgId) {
  write_rr_csr(CSR_RROPC0 + opc, cfgId);
}

static void rr_fence(uint32_t cfgId) {
  write_rr_csr(CSR_RRBAR, cfgId);
  asm volatile("fence");
}
```

RoCC Programming Model

Code:

Directly invoke accelerator software stack

```
libaccel_execute_task(data);
```

AuRORA + ReRoCC Programming Model

To use a ReRoCC accelerator, only need to wrap the accelerator software with minimal additional code

Code:

```
do {
  csr_write(CSR_RRCFG0, RRCFG_ACQ | 0x0);
} while (csr_read(CSR_RRCFG0) & !RRCFG_ACQ);

csr_write(CSR_OPC0, 0x0);


libaccel_execute_task(data);



csr_write(CSR_CFG0, 0x0);
```

AuRORA + ReRoCC Programming Model

Procedure:

1. Software should attempt to acquire an accelerator from the system

Code:

```
do {
  write_csr(CSR_RRCFG0, RRCFG_ACQ | 0x0);
} while (read_csr(CSR_RRCFG0) & !RRCFG_ACQ);
```

- Accelerator acquisition can happen in user mode
- Overhead is just interconnect latency to query the accelerator for availability
  - O(10s) of cycles

- Threads which fail to acquire can `sleep()` to deschedule themselves

- Platforms can implement multiple homogeneous accelerators
  - Ex: 4 instances of accelX, 2 instances of accelY, etc.
- User threads can query to request any, or multiple available accelerators

AuRORA + ReRoCC Programming Model

Procedure:

1. Software should attempt to acquire an accelerator from the system
2. Software should map a local opcode to the accelerator

Code:

```
do {
  write_csr(CSR_RRCFG0, RRCFG_ACQ | 0x0);
} while (read_csr(CSR_RRCFG0) & !RRCFG_ACQ);
```

```
write_csr(CSR_RROPC0, 0x0);
```

- Fast, low-latency, just writing some CSR on the core
- Supporting more acquired accelerators than opcodes lets us get around limited available opcode space

AuRORA + ReRoCC Programming Model

Procedure:

1. Software should attempt to acquire an accelerator from the system
2. Software should map a local opcode to the accelerator
3. After acquisition, accelerator appears to be architecturally part of the host thread

Code:

```
do {
  csr_write(CSR_RRCFG0, RRCFG_ACQ | 0x0);
} while (csr_read(CSR_RRCFG0) & !RRCFG_ACQ);
```

```
csr_write(CSR_RROPC0, 0x0);
```

```
libaccel_execute_task(data);
```

- Unmodified accelerator kernels/libraries can be executed
- Architectural illusion of a unified system is maintained for accelerator software

AuRORA + ReRoCC Programming Model

Procedure:

1. Software should attempt to acquire an accelerator from the system

Code:

```
do {
  csr_write(CSR_RRCFG0, RRCFG_ACQ | 0x0);
} while (csr_read(CSR_RRCFG0) & !RRCFG_ACQ);
```

2. Software should map a local opcode to the accelerator

```
csr_write(CSR_RROPC0, CFG0);
```

3. After acquisition, accelerator appears to be architecturally part of the host thread

```
libaccel_execute_task(data);
```

4. Software should release the accelerator to the system after completion

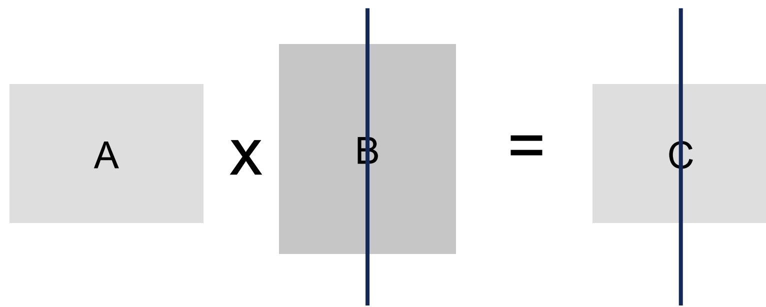
```
csr_write(CSR_RRCFG0, 0x0);
```

Pre-Run Test

```
cd $MCYDIR/generators/aurora

# build workload
./build.sh

cd ../../sims/verilator
# run test
make CONFIG=TutorialGemminiReRoCCConfig run-binary-hex
BINARY=../../generators/aurora/build/bareMetalC/tiled_matmul_ws_perf
-baremetal
```

View ReRoCC Header & Test

```
cd $MCYDIR/generators/aurora

vim aurora-sw/aurora.h
vim aurora-sw/gemmini_aurora.h
vim bareMetalC/tiled_matmul_ws_perf.c
```



Division of matmul across two accelerators

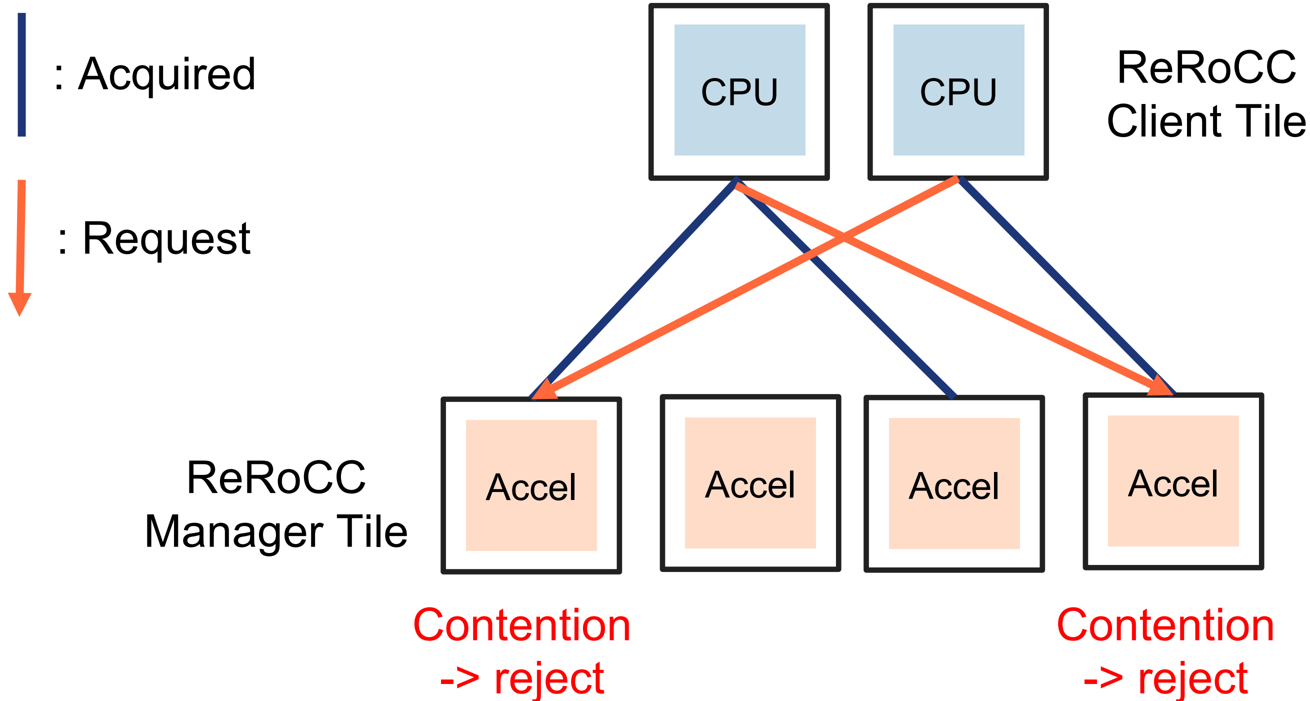
```
[UART] UART0 is here (stdin/stdout).
Acquired 2 accelerators
Starting gemmini matmul
I: 256, J: 256, K: 256
NO_BIAS: 1, REPEATING_BIAS: 1
A_TRANSPOSE: 0, B_TRANSPOSE: 0
Cycles taken: 37948
Total macs: 16777216
Ideal cycles: 32768
Utilization: 86%
```

Run Contention Test

```
cd $MCYDIR/generators/aurora

vim bareMetalC/mt_contention.c


# build multi workload
./build_multi.sh
```

Run Contention Test

TutorialGemminiReRoCCConfig



Run Contention Test

```
cd $MCYDIR/sims/verilator

# run test
make CONFIG=TutorialGemminiReRoCCConfig run-binary-hex
BINARY=../../generators/aurora/build/bareMetalC/mt_contention-
baremetal
```

req: requested
use: acquired

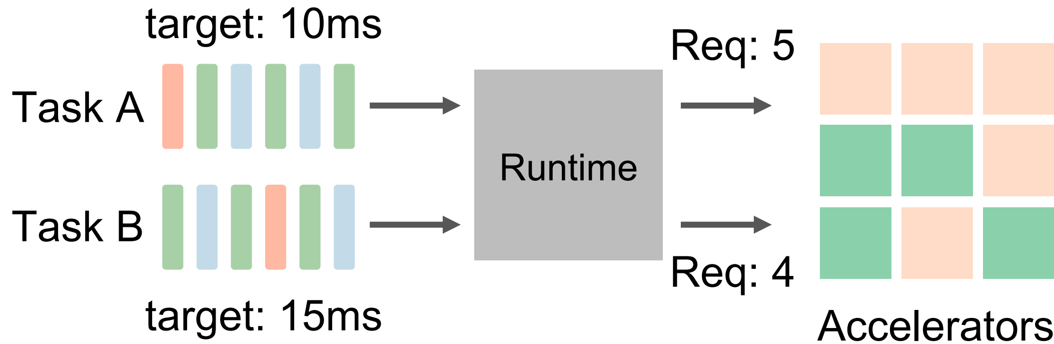
```
[UART] UART0 is here (stdin/stdout).
Thread 0/2 starting
Thread 1/2 starting
round 0 thread 0 array req: 1, 1, 2, 1, 2, 3, 2, 1, 3, 3, 3, 2, 2, 2, 2,
round 0 thread 0 array use: 1, 1, 2, 1, 2, 2, 2, 1, 1, 3, 2, 1, 2, 2, 1,
round 0 thread 1 array req: 2, 2, 3, 2, 3, 3, 3, 2, 3, 1, 3, 3, 3, 3, 3,
round 0 thread 1 array use: 2, 2, 2, 2, 2, 3, 1, 2, 3, 1, 2, 3, 3, 3, 3,
round 1 thread 0 array req: 3, 1, 3, 2, 3, 3, 3, 3, 3, 2, 1, 1, 1, 1, 1,
round 1 thread 0 array use: 1, 1, 2, 1, 3, 2, 3, 2, 2, 2, 1, 1, 1, 1, 1,
round 1 thread 1 array req: 3, 2, 3, 3, 1, 3, 1, 3, 3, 3, 2, 2, 2, 2, 2,
round 1 thread 1 array use: 3, 2, 3, 1, 1, 2, 1, 1, 2, 2, 2, 2, 2, 2, 2,
```

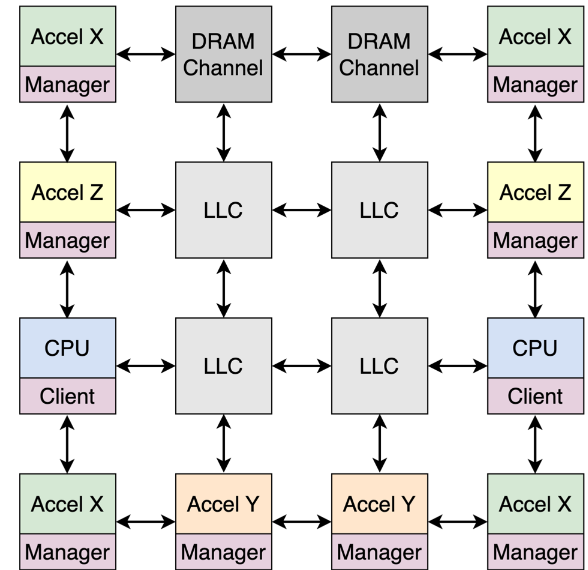
Summary

We showed in this simple example how AuRORA's virtual interface handles contention

It can be extended to complex heterogeneous many-accelerator SoC

It can be extended to complex multi-tenant case with its ability to handle contention by adding runtime to partition resources





Example heterogenous SoC integrated with AuRORA