AMD + Xilinx Converged Runtime Technology Demonstration

CTO Team

SuperComputing 2020
Current: Two Stacks

User App Calls Two Stacks

HPC Apps

ML Frameworks

HIP Libraries

FFT
Sparse
MIGraph (TensorRT)
Collectives (RCCL)
MIOpen/ DNN
BLAS

HIP C++ (Host & Device Code)

ROCm Runtime

CPU
GPU

Vitis Suite
(Host & Kernel)

XRT API +
XLNX Device Code

XRT

CPU
FPGA

© Copyright 2020 Xilinx
Preview: ROCm Runtime Integration

HIP Libraries
- FFT
- Sparse
- MIGraph (TensorRT)
- Collectives (RCCL)
- MIOpen/DNN
- BLAS

HIP C++ (Host & Device Code) Interop XLNX Device Code

Converged ROCm Runtime
- CPU
- GPU
- FPGA
  - Scalar Threaded
  - High IPC Compute
  - Vector Threaded
  - Throughput Compute
  - Fine Grain Spatial
  - Dataflow Compute

ROCm
- ROCm Runtime Integration
  - Unified Resource Discovery
  - Unified Dispatch & Sync
  - Shared Virtual Memory
- XLNX Device Languages
  - HLS C, AIE C
In Development: Integrated SW Stack

HIP Libraries
- FFT
- Sparse
- MIGraph (TensorRT)
- Collectives (RCCL)
- MIOpen/ DNN
- BLAS

HIP C++ (Host & Device Code)

Converged ROCm Runtime

ROCm
- ROCm Runtime Integration
  - Unified Resource Discovery
  - Unified Dispatch & Sync
  - Shared Virtual Memory
- Unified Device Languages
  - Thread, Pipeline, Spatial
- ROCm Library Integration
  - Harmonized Library APIs
  - Harmonized Framework Plugins

HPC Apps
- HPC Apps

ML Frameworks
- ML Frameworks

CPU
- CPU
  - Scalar Threaded
  - High IPC Compute

GPU
- GPU
  - Vector Threaded
  - Throughput Compute

FPGA
- FPGA
  - Fine Grain Spatial
  - Dataflow Compute
Unified ROCm Runtime for GPUs and FPGAs

- **Unified Device and Topology Discovery**
  Runtime resource reservation
  Scale-out

- **Unified Dispatch and Synchronization**
  Common user-mode queuing architectural queuing model
  Peer-to-peer events

- **Shared Virtual Memory**
  Common virtual address space
  Visibility of FPGA buffers from GPU and CPU
Demonstration of Capabilities
Demo 1: Device Discovery

ROCm Runtime discovers CPUs, GPUs and FPGAs in consistent way.

```c
static hsa_status_t show_name(hsa_agent_t agent, void *data) {
    char name[64] = { 0 };
    hsa_status_t status = hsa_agent_get_info(agent, HSA_AGENT_INFO_NAME, name);
    printf("Found agent %s\n", name);
    return status;
}

int main(int argc, char ** argv) {
    hsa_status_t err = hsa_init();
    printf("Initialized\n");
    // Display the agents found in the system
    hsa_agent_t agent;
    err = hsa_iterate_agents(show_name, &agent);
}
```

- CPU
- GPU
- FPGA

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Initialized
Found agent AMD Ryzen 9 3900X 12-Core Processor
Found agent gfx803
Found agent Xilinx U280
Demo 1: Device Discovery

ROCm Runtime discovers CPUs, GPUs and FPGAs in consistent way

Common method to discover and reserve CPU, GPU and FPGA resources

```c
36 static hsa_status_t show_name(hsa_agent_t agent, void *data) {
37     char name[64] = { 0 };
38     hsa_status_t status = hsa_agent_get_info(agent, HSA_AGENT_INFO_NAME, name);
39     printf("Found agent %s\n", name);
40     return status;
41 }
42
43 int main(int argc, char ** argv) {
49     // Display the agents found in the system
50     hsa_agent_t agent;
51     err = hsa_iterate_agents(show_name, &agent);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Initialized
Found agent AMD Ryzen 9 3900X 12-Core Processor
Found agent gfx803
Found agent Xilinx U280
Demo 2: Kernel Dispatch and Synchronization
Usermode Queues and Events

```c
int main(int argc, char **argv) {
    hsa_status_t status = hsa_init();
    hsa_agent_t fpga_agent[1];
    status = hsa_iterate_agents(get_fpga_agent, fpga_agent);
    hsa_queue_t *queue = malloc(sizeof(hsa_queue_t));
    // create a user mode queue on FPGA
    status = hsa_queue_create(fpga_agent[0], 256,
                              HSA_QUEUE_TYPE_SINGLE, 0, 0, 0, 0,
                              &queue);
    hsa_signal_t completion_signal;
    status = hsa_signal_create(1 /* Initial Value */ , 1, fpga_agent, &completion_signal);
    uint64_t index = hsa_queue_load_write_index_relaxed(queue);
    const uint32_t mask = queue->size - 1;
    hsa_agent_dispatch_packet_t *packet =
        &(((hsa_agent_dispatch_packet_t*)queue->base_address)[index & mask]);
    // set parameters in packet
    packet->completion_signal = completion_signal;
    hsa_queue_store_write_index_relaxed(queue, index + 1);
    hsa_signal_wait_ssc acquire(completion_signal,
                                 HSA_SIGNAL_CONDITION_EQ, 0, 2000,
                                 HSA_WAIT_STATE_ACTIVE);
    printf("Success\n");
}
```
Demo 2: Kernel Dispatch and Synchronization
Usermode Queues and Events

```c
#include <hsa.h>

int main(int argc, char ** argv) {
    hsa_status_t status = hsa_init();
    hsa_agent_t fpga_agent[1];
    status = hsa_iterate_agents(0, fpga_agent);
    hsa_queue_t * queue = malloc(sizeof(hsa_queue_t));
    // create a user mode queue on FPGA
    status = hsa_queue_create(fpga_agent, 256,
                              type, nullptr, nullptr,
                              private_segment_size, group_segment_size, &queue);
    hsa_signal_t completion_signal;
    status = hsa_signal_create(1, 0, nullptr, &completion_signal);
    hsa_agent_dispatch_packet_t packet;
    packet.completion_signal = completion_signal;
    // dispatch packet
    hsa_signal_wait_scacquire(completion_signal, HSA_SIGNAL_CONDITION_EQ, 0, 2000,
                              HSA_WAIT_STATE_ACTIVE);
    printf("Success\n");
}
```

Common method to dispatch and synchronize work on GPUs and FPGAs
Demo 3: Kernel Queues and Events using Shared Virtual Memory

```c
void *input_address, *output_address;
hsa_agent_iterate_regions(fpga_agent[0], get_karg_region, &karg_region);
// create a user mode queue on FPGA
status = hsa_queue_create(fpga_agent[0],
256,
HSA_QUEUE_TYPE_SINGLE,
0, 0, 0,
&queue);

status = hsa_signal_create(1 /* Initial Value */ ,1, fpga_agent, &completion_signal);

/* Allocate FPGA-visible input buffer on the host */
hsa_memory_allocate(karg_region, 512, &input_address);
/* Allocate FPGA-visible output buffer on the host */
hsa_memory_allocate(karg_region, 512, &output_address);

uint64_t index = hsa_queue_load_write_index_relaxed(queue);
const uint32_t mask = queue->size - 1;
hsa_agent_dispatch_packet_t * packet =
\[
((hsa_agent_dispatch_packet_t *) (queue->base_address)) [index & mask] ;
\]

// set parameters in packet
packet->completion_signal = completion_signal;
packet->arg[0] = (uint64_t)input_address;
packet->arg[1] = (uint64_t)output_address;
packet->header = HSA_PACKET_TYPE_AGENT_DISPATCH << HSA_PACKET_HEADER_TYPE;
hsa_queue_store_write_index_relaxed(queue, index+1);

hsa_signal_wait_scacquire(completion_signal,
HSA_SIGNAL_CONDITION_EQ, 0,
2000,
HSA_WAIT_STATE_ACTIVE);
printf("Success\n");
```
Demo 3: Kernel Queues and Events using Shared Virtual Memory

Shared Virtual Memory: access buffers from CPU, GPU, and FPGA
Demo 4: Interop with GPU

```c
uint64_t fpga_qidx = hsa_queue_load_write_index_relaxed(fpga_queue);
const uint32_t fpga_mask = fpga_queue->size - 1;
hsa_agent_t * fpga_packet = &((hsa_agent_dispatch_packet_t*)(fpga_queue->base_address))[fpga_qidx & fpga_mask];

// set parameters in packet
fpga_packet->completion_signal = fpga_completion_signal;
fpga_packet->arg[0] = (uint64_t)input_address;
fpga_packet->arg[1] = (uint64_t)output_address;
fpga_packet->header = HSA_PACKET_TYPE_AGENT_DISPATCH << HSA_PACKET_HEADER_TYPE;
/* Dispatch Kernel to FPGA */
hsa_queue_store_write_index_relaxed(fpga_queue, +fpga_qidx);

uint64_t gpu_qidx = hsa_queue_load_write_index_relaxed(gpu_queue);
const uint32_t gpu_mask = gpu_queue->size - 1;
hsa_barrier_and_packet_t * barrier_packet = &((hsa_barrier_and_packet_t*)(gpu_queue->base_address))[gpu_qidx & gpu_mask];

memset(barrier_packet, 0, sizeof(hsa_barrier_and_packet_t));
barrier_packet->dep_signal[0] = fpga_completion_signal;
barrier_packet->header = HSA_PACKET_TYPE_BARRIER_AND << HSA_PACKET_HEADER_TYPE;
/* Write Barrier packet into GPU queue */
hsa_queue_store_write_index_relaxed(gpu_queue, +gpu_qidx);
hsa_kernel_dispatch_packet_t * gpu_packet = &((hsa_kernel_dispatch_packet_t*)(gpu_queue->base_address))[gpu_qidx & gpu_mask];
/* Sets up kernel dispatch and makes packet visible */
populate_gpu_kernel_packet(gpu_packet, &gpu_completion_signal);
/* Write GPU kernel packet into GPU queue, will only start when FPGA task is complete */
hsa_queue_store_write_index_relaxed(gpu_queue, +gpu_qidx);
hsa_signal_wait_scacquire(gpu_completion_signal,
HSA_SIGNAL_CONDITION_EQ, 0, 2000,
HSA_WAIT_STATE_ACTIVE);
printf("Success\n");
```
Demo 4: Interop with GPU

Peer-to-peer data sharing and synchronization between GPU and FPGA

```c
uint64_t fpga_qidx = hsa_queue_load_write_index_relaxed(fpga_queue);
const uint32_t fpga_mask = fpga_queue->size - 1;
hsa_agent_dispatch_packet_t * fpga_packet =
  &((hsa_agent_dispatch_packet_t*)(fpga_queue->base_address))[fpga_qidx & fpga_mask];

// set parameters in packet
fpga_packet->completion_signal = fpga_completion_signal;
fpga_packet->arg[0] = (uint64_t)input_address;
fpga_packet->arg[1] = (uint64_t)output_address;
fpga_packet->header = HSA_PACKET_TYPE_AGENT_DISPATCH << HSA_PACKET_HEADER_TYPE;
/* Dispatch Kernel to FPGA */

memset(barrier_packet, 0, sizeof(hsa_barrier_and_packet_t));
barrier_packet->dep_signal[0] = fpga_completion_signal;
barrier_packet->header = HSA_PACKET_TYPE_BARRIER_AND << HSA_PACKET_HEADER_TYPE;
/* Write Barrier packet into GPU queue */
hsa_queue_store_write_index_relaxed(gpu_queue, +gpu_qidx);
hsa_kernel_dispatch_packet_t * gpu_packet =
  &((hsa_kernel_dispatch_packet_t*)(gpu_queue->base_address))[gpu_qidx & gpu_mask];
/* Sets up kernel dispatch and makes packet visible */
populate_gpu_kernel_packet(gpu_packet, &gpu_completion_signal);
/* Write GPU kernel packet into GPU queue, will only start when FPGA task is complete */
hsa_queue_store_write_index_relaxed(gpu_queue, +gpu_qidx);
hsa_signal_wait_scaquire(gpu_completion_signal,
    HSA_SIGNAL_CONDITION_EQ, 0, 2000,
    HSA_WAIT_STATE_ACTIVE);
printf("Success\n");
```
Xilinx Mission

Building the Adaptable, Intelligent World