Striving for SDR Performance Portability in the Era of Heterogeneous SoCs

Jeffrey S. Vetter
Seyong Lee
Mehmet Belviranli
Jungwon Kim
Richard Glassbrook
Abdel-Kareem Moadi
Seth Hitefield

FOSDEM
Brussels
2 Feb 2020
Highlights

• Architectural specialization
• Performance portability of applications and software
• DSSoC ORNL project investigating on performance portability of SDR
  – Understand applications and target architectures
  – Use open programming models (e.g., OpenMP, OpenACC, OpenCL)
  – Develop intelligent runtime systems
• Goal: scale applications from Qualcomm Snapdragon to DoE Summit Supercomputer with minimal programmer effort
Sixth Wave of Computing

http://www.kurzweilai.net/exponential-growth-of-computing
Predictions for Transition Period

Optimize Software and Expose New Hierarchical Parallelism

- Redesign software to boost performance on upcoming architectures
- Exploit new levels of parallelism and efficient data movement

Architectural Specialization and Integration

- Use CMOS more effectively for specific workloads
- Integrate components to boost performance and eliminate inefficiencies
- Workload specific memory+storage system design

Emerging Technologies

- Investigate new computational paradigms
  - Quantum
  - Neuromorphic
  - Advanced Digital
  - Emerging Memory Devices
Complex architectures yield...

System: MPI, Legion, HPX, Charm++, etc

Node: OpenMP, Pthreads, U-threads, etc

Cores: OpenACC, CUDA, OpenCL, OpenMP4, ...

- Low overhead
- Resource contention
- Locality
- SIMD
- NUMA, HBM

Memory use, coalescing
Data orchestration
Fine grained parallelism
Hardware features

Complex Programming Models
During this Sixth Wave transition, **Complexity** is our major challenge!

<table>
<thead>
<tr>
<th>Design: How do we design future systems so that they are better than current systems on mission applications?</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Entirely possible that the new system will be slower than the old system!</td>
</tr>
<tr>
<td>• Expect ‘disaster’ procurements</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programmability: How do we design applications with some level of performance portability?</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Software lasts much longer than transient hardware platforms</td>
</tr>
<tr>
<td>• Adapt or die</td>
</tr>
</tbody>
</table>
DARPA Domain-Specific System on a Chip (DSSoC) Program
Getting the best out of specialization when we need programmability

Three Optimization Areas
1. Design time
2. Run time
3. Compile time

Addressed via five program areas
1. Intelligent scheduling
2. Domain representations
3. Software
4. Medium access control (MAC)
5. Hardware integration

Looking at how Hardware/Software co-design is an enabler for efficient use of processing power
DSSoC ORNL Project Overview

Aspen

- Ontologies
  - Structured
  - Composable
  - Interoperable

Parameterizable Performance Models

- Concurrency
- Work
- Communication
- Capacities
- Synchronization

Machine Model

- Compute
- Memory
- Interconnects

PFU API

Applications

- Streaming (e.g., SW Radio)
- Sensing (e.g., SAR, vision)
- Deep learning (e.g., CNN)
- Analytics (e.g., graphs)
- Robotics (e.g., sense and react)
- Science and Engineering (e.g., CFD)

Ontologies include Motifs, Parallel Execution Patterns, Locality, Affinity, Sync, etc.

Programming Systems

- Compiler
- DSL
- JIT
- Libraries
- etc

Performance Models combine Ontologies and Machine Models to produce both static and dynamic cost estimates.

Runtime and Operating Systems

- Task scheduling
- Memory Mgt
- IO
- Synchronization

Machine Models include architecture configuration information, microbenchmarks, and API for PFU

DSSoC Hardware

- CPU
- GPU
- FPGA
- Accelerator
- DSP
- Deep Memory
- Neuromorphic
Development Lifecycle

- **Applications**: Create scalable application Aspen models manually, with static or dynamic analysis, or using historical information.

- **Ontologies**: Ontologies based on Aspen models using statistical and machine learning techniques.

- **Programming systems**: Programming systems built to support ontologies. Query Aspen models and PFU for automatic code generation, optimization, etc.

- **Runtime and Scheduling**: Intelligent runtime scheduling uses models and PFU to inform dynamic decisions. Dynamic resource discovery and monitoring.

- **DSSoC Chip**: DSSoC design quantitatively derived from application Aspen models. Early design space exploration with Aspen.

- **Performance Functional API**: As feature of DSSoC, PFU API provides dynamic performance response of deployed DSSoC to intelligent runtime and programming system.

Precise configuration and benchmark data for static analysis, mapping, partitioning, code generation, etc.

Dynamic Performance Feedback including profiling and configuration response.
### Summit Node Overview

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Performance</td>
<td>300 TF</td>
</tr>
<tr>
<td>Number of Nodes</td>
<td>4,608</td>
</tr>
<tr>
<td>Node performance</td>
<td>42 TF</td>
</tr>
<tr>
<td>Memory per Node</td>
<td>512 GB DDR4 + 96 GB HBM2</td>
</tr>
<tr>
<td>NV memory per Node</td>
<td>1690 GB</td>
</tr>
<tr>
<td>Total System Memory</td>
<td>&gt; 10 PB DDR4 + HBM2 + Non-volatile</td>
</tr>
<tr>
<td>Processors</td>
<td>2 IBM POWER9® 9,216 CPUs, 6 NVIDIA Volta™ 27,648 GPUs</td>
</tr>
<tr>
<td>File System</td>
<td>250 PB, 2.5 TB/s, GPFS™</td>
</tr>
<tr>
<td>Power Consumption</td>
<td>13 MW</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Mellanox EDR 100G InfiniBand</td>
</tr>
<tr>
<td>Operating System</td>
<td>Red Hat Enterprise Linux (RHEL) version 7.4</td>
</tr>
</tbody>
</table>

**Diagram:**

- **TF:** 42 TF (6x7 TF)
- **HBM:** 96 GB (6x16 GB)
- **DRAM:** 512 GB (2x16x16 GB)
- **NET:** 25 GB/s (2x12.5 GB)
- **MMs/s:** 83

HBM & DRAM speeds are aggregate (Read+Write). All other speeds (X-Bus, NVLink, PCIe, I/O) are bi-directional.
Intel Stratix 10 FPGA

Experimental Computing Lab (ExCL) managed by the ORNL Future Technologies Group

• Intel Stratix 10 FPGA and four banks of DDR4 external memory
  – Board configuration: Nallatech 520 Network Acceleration Card
• Up to 10 TFLOPS of peak single precision performance
• 25MBytes of L1 cache @ up to 94 TBytes/s peak bandwidth
• 2X Core performance gains over Arria® 10
• Quartus and OpenCL software (Intel SDK v18.1) for using FPGA
• Provide researcher access to advanced FPGA/SOC environment

For more information or to apply for an account, visit https://excl.ornl.gov/
NVIDIA Jetson AGX Xavier SoC
Experimental Computing Lab (ExCL) managed by the ORNL Future Technologies Group

- NVIDIA Jetson AGX Xavier:
- High-performance system on a chip for autonomous machines
- Heterogeneous SoC contains:
  - Eight-core 64-bit ARMv8.2 CPU cluster (Carmel)
  - 1.4 CUDA TFLOPS (FP32) GPU with additional inference optimizations (Volta)
  - 11.4 DL TOPS (INT8) Deep learning accelerator (NVDLA)
  - 1.7 CV TOPS (INT8) 7-slot VLIW dual-processor Vision accelerator (PVA)
  - A set of multimedia accelerators (stereo, LDC, optical flow)
- Provides researchers access to advanced high-performance SOC environment

For more information or to apply for an account, visit https://excl.ornl.gov/
Qualcomm 855 SoC (SM8510P) Snapdragon™
Experimental Computing Lab (ExCL) managed by the ORNL Future Technologies Group

**Adreno 640**
- Vulkan, OpenCL, OpenGL ES 3.1
- Apps: HDR10+, HEVC, Dolby, etc
- Enables 8k-360° VR video playback
- 20% faster compared to Adreno 630

**Hexagon 690**
- Quad threaded Scalar Core
- DSP + 4 Hexagon Vector Xccelerators
- New Tensor Xccelerator for AI
- Apps: AI, Voice Assistance, AV codecs

**Kyro 485**
- Snapdragon X24 LTE (855 built-in) modem LTE Category 20
- Snapdragon X50 5G (external) modem (for 5G devices)
- Qualcomm Wi-Fi 6-ready mobile platform: (802.11ax-ready, 802.11ac Wave 2, 802.11ay, 802.11ad)
- Qualcomm 60 GHz Wi-Fi mobile platform: (802.11ay, 802.11ad)
- Bluetooth Version: 5.0
- Bluetooth Speed: 2 Mbps
- High accuracy location with dual-frequency GNSS.

**Kyro 485 (8-ARM Prime+BigLittle Cores)**
- Snapdragon X64 modem

**Spectra 360 ISP**
- New dedicated Image Signal Processor (ISP)
- Dual 14-bit CV-ISPs; 48MP @ 30fps single camera
- Hardware CV for object detection, tracking, streo depth process
- 6DoF XR Body tracking, H265, 4K60 HDR video capture, etc.

**Connectivity (5G)**
- Snapdragon X24 LTE (855 built-in) modem LTE Category 20
- Snapdragon X50 5G (external) modem (for 5G devices)
- Qualcomm Wi-Fi 6-ready mobile platform: (802.11ax-ready, 802.11ac Wave 2, 802.11ay, 802.11ad)
- Qualcomm 60 GHz Wi-Fi mobile platform: (802.11ay, 802.11ad)
- Bluetooth Version: 5.0
- Bluetooth Speed: 2 Mbps
- High accuracy location with dual-frequency GNSS.

**Qualcomm Development Board connected to (mcmurdo) HPZ820**
- Connected Qualcomm board to HPZ820 through USB
- Development Environment: Android SDK/NDK
- Login to mcmurdo machine
  $ ssh –Y mcmurdo
- Setup Android platform tools and development environment
  $ source /home/nqx/setup_android.source
- Run Hello-world on ARM cores
  $ git clone https://code.ornl.gov/nqx/helloworld-android
  $ make compile push run
- Run OpenCL example on GPU
  $ git clone https://code.ornl.gov/nqx/opencl-img-processing
  $ Run Sobel edge detection
  $ make compile push run fetch
- Login to Qualcomm development board shell
  $ adb shell
  $ cd /data/local/tmp

For more information or to apply for an account, visit https://excl.ornl.gov/
End-to-End System: Gnu Radio for Wifi on two NVIDIA Xavier SoCs

- Signal processing: An open-source implementation of IEEE-802.11 WIFI a/b/g with GR OOT modules.
- Input / Output file support via Socket PDU (UDP server) blocks
- Image/Video transcoding with OpenCL/OpenCV
• **GR-Tools**
  • **First tools are released**
    • Block-level Ontologies [ontologyAnalysis]
      • Following properties are extracted from a batch of block definition files: Descriptions and IDs, source and sink ports (whether input/output is scalar, vector or multi-port), allowed data types, and additional algorithm-specific parameters
    • Flowgraph Characterization [workflowAnalysis]
      • Characterization of GR workloads at the flowgraph level.
      • Scripts automatically run for for 30 seconds and reports a breakdown of high-level library module calls
    • Design-space Exploration [designSpaceCL]
      • Script to run 13 blocks included in gr-enabled
        - Both on a GPU and on a single CPU core
        - By using input sizes varying between 24 and 227 elements.
  • Two prototype tools have been added recently
    • cgran-scraper
    • GRC-analyzer

https://github.com/cosmic-sdr
Applications Profiling

- **Preliminary SDR Application Profiling:**
  - Created fully automated GRC profiling toolkit
  - Ran each of the 89 flowgraph for 30 seconds
  - Profiled with performance counters
  - Major overheads:
    - Python glue code (libpython), O/S threading & profiling (kernel.kallsyms, libpthread), libc, ld, Qt
  - Runtime overhead:
    - Will require significant consideration when run on SoC
    - Cannot be executed in parallel
    - Hardware assisted scheduling is essential

### Library Percentage

<table>
<thead>
<tr>
<th>Library</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>[kernel.kallsyms]</td>
<td>27.8547</td>
</tr>
<tr>
<td>libpython</td>
<td>18.6281</td>
</tr>
<tr>
<td>libgnuradio</td>
<td>11.7548</td>
</tr>
<tr>
<td>libc</td>
<td>7.7503</td>
</tr>
<tr>
<td>ld</td>
<td>3.8839</td>
</tr>
<tr>
<td>libvolk</td>
<td>3.7963</td>
</tr>
<tr>
<td>libperl</td>
<td>3.7837</td>
</tr>
<tr>
<td>[unknown]</td>
<td>3.6465</td>
</tr>
<tr>
<td>libQt5</td>
<td>2.9866</td>
</tr>
<tr>
<td>libpthread</td>
<td>2.1449</td>
</tr>
</tbody>
</table>

### libgnuradio CPU-time Breakdown

- libgnuradio-analog: 28%
- libgnuradio-blocks: 4%
- libgnuradio-channels: 13%
- libgnuradio-digital: 6%
- libgnuradio-dtv: 3%
- libgnuradio-fec: 10%
- libgnuradio-fft: 8%
- libgnuradio-femt: 0%
Block proximity analysis

- Creates a graph:
  - **Nodes**: Unique block types
  - **Edges**: Blocks used in the same GRC file.
  - Every co-occurrence increases edge weight by 1.

- This example was run
  - **With** --mode proximityGraph
  - **On** randomly selected sub-set of GRC files
Programming Systems

Aspen
- Ontologies
  - Structured
  - Composable
  - Interoperable
- Parameterizable Performance Models
  - Concurrency
  - Work
  - Communication
  - Capacities
  - Synchronization
- Machine Model
  - Compute
  - Memory
  - Interconnects
PFU API

Applications
- Streaming (e.g., SW Radio)
- Sensing (e.g., SAR, vision)
- Deep learning (e.g., CNN)
- Analytics (e.g., graphs)
- Robotics (e.g., sense and react)
- Science and Engineering (e.g., CFD)

Ontologies include Motifs, Parallel Execution Patterns, Locality, Affinity, Sync, etc.

Programming Systems
- Compiler
- DSL
- JIT
- Libraries
- etc

Performance Models combine Ontologies and Machine Models to produce both static and dynamic cost estimates

Runtime and Operating Systems
- Task scheduling
- Memory Mgt
- IO
- Synchronization

Machine Models include architecture configuration information, microbenchmarks, and API for PFU

DSSoC Hardware
- Performance Parameters
  - CPU
  - GPU
  - FPGA
  - Accelerator
  - DSP
  - Deep Memory
  - Neuromorphic
Programming Solution for DSSoC

- **Main input programming models**: OpenMP, OpenACC
- **Main input/main output programming models**: MPI
- **Optional input programming models**: Verilog, CASH
- **Main output/main output programming models**: HIP, CUDA, OpenCL

**Compilers**

- OpenARC
- LLVM

**Used as both input and output programming model to the compiler**: OpenARC
**Used as input programming model to the compiler**: LLVM, CUDA, OpenCL
**Used as main input/main output programming model**: MPI
**Used as main input programming model**: OpenMP, OpenACC
**Main input programming model**: IRIS, CASH
New OpenACC GR Block Mapping Strategy for Heterogeneous Architectures

IRIS offers a common API for diverse heterogeneous devices and also allows intermixing of multiple programming models (mix CUDA, OpenMP, OpenCL, etc.).
OpenACC GR Block Code Structure

//Constructor
accLog_impl::accLog_impl(...
  int contextType, int deviceId, int copy_in, int copy_out)
  : gr::sync_block("accLog", ...), ...
  GRACCBase(contextType, deviceId) {
    accLog_init(deviceType, deviceId, threadID);
    ...
  }

//Reference CPU implementation
int accLog_impl::testCPU(...)
  ...
  for (int i = 0; i < n; i++) {
    out[i] = n_val * log10(in1[i]) + k_val;
  }
  ...

//OpenACC implementation
int accLog_impl::testOpenACC(...)
  ...
  if (acc_init_done == 0) {
    gracc_poopin(...); //Create and copy input data to device memory.
    gracc_pcreate(...); //Create device buffer for output data.
    acc_init_done = 1;
  } else if (gracc_copy_in == 1) {
    gracc_update_device(...); //Copy input data to device memory.
    accLog_kernel(...); //Execute an OpenACC kernel.
  if (gracc_copy_out == 1) {
    gracc_update_self(...); //Copy output data to host memory.
  }
  ...
  }

int accLog_impl::work(...)
  ...
  if (contextType == ACC_TYPE_CPU) {
    retVal = testCPU(...); //Execute reference CPU version.
  } else {
    retVal = testOpenACC(...); //Execute OpenACC version.
  }
  ...

Constructor
- OpenACC GR block class inherits GRACCBase class as a base class.
- GRACCBase constructor assigns a unique thread ID per OpenACC GR block instantiation, which is internally used for thread safety.
- OpenACC backend runtime is also initialized.

Reference CPU Implementation
- Contains the same code as that in the original GR block, which may have already been vectorized using Volk library.

OpenACC Implementation
- Contains the OpenACC version of the reference CPU implementation.
- Performs the following tasks:
  - Copy input data to device memory.
  - Execute the OpenACC kernel.
  - Copy output data back to host memory.
- OpenARC will translate the OpenACC kernel to multiple different output programming models (e.g., CUDA, OpenCL, OpenMP, HIP, etc.)

Main Entry Function
- Main entry function executed whenever GR scheduler invokes the OpenACC GR block.
- The GR block argument, contextType decides which to execute between the reference CPU version and OpenACC version.
- OpenACC backend runtime may choose CPU as an offloading target (e.g., offloading OpenMP3 kernel to CPU).
Example Translation of GR accLog Module

Input OpenACC code

163163
151x503

Output host code

Output CUDA kernel code
Port an Example SDR Workflow to Xavier

OpenACC-enabled workflow using `gr-openacc` blocks

Reference CPU workflow using original `gr-blocks`
• In the basic memory management scheme, each invocation of an OpenACC GR block performs the following three tasks:
  1) Copy input data to device memory.
  2) Run a kernel on device.
  3) Copy output data back to host memory.
In the optimized memory management scheme, some blocks can bypass unnecessary memory transfers between host and device and directly communicate each other using device memory if both producer and consumer blocks are running on the same device.

Notice that device buffer needs extra padding to handle the overwriting feature in the host circular buffer.
Sample Output of the Example SDR Workflow

**CPU Workflow**

- Amplitude
- Time (ms)

**OpenACC Workflow**

- Amplitude
- Time (ms)
• CPU versions of OpenACC blocks are algorithmically equivalent to those in the original GR blocks.

Some OpenACC blocks (B, D) use a simple register caching optimization, which causes them to perform better than the original GR blocks.
• OpenACC blocks are automatically translated to OpenMP3 versions and run on Xavier CPU.

Some of original GR blocks (A, C) were already vectorized with Volk library.

Some of original GR blocks (B, C) performed better than OpenACC blocks (B, C).
OpenACC blocks are automatically translated to CUDA versions and run on Xavier GPU.
Each invocation of an OpenACC block executes three tasks: 1) copy input data to device memory, 2) run a kernel on device, and 3) copy output data back to host memory.

Due to extra memory transfer overheads, most OpenACC blocks perform worse than original GR blocks, except for the OpenACC block D1 and D2.
OpenACC blocks are automatically translated to CUDA versions and run on Xavier GPU. Optimized OpenACC blocks bypass memory transfers between host and device and directly communicate each other using device memory if both producer and consumer blocks are running on the same device.

Most of the OpenACC blocks perform better than original GR blocks, except for the block A; the original GR block A is vectorized with Volk library, which performs better than the OpenACC block A.
This example offloads more OpenACC blocks to Xavier GPU than the previous example.

OpenACC-enabled workflow using gr-openacc blocks

Reference CPU workflow using original gr-blocks
Profiling Results When Opt. OpenACC Blocks Offloaded to GPU

- OpenACC blocks are automatically translated to CUDA versions and run on Xavier GPU.
- Optimized OpenACC blocks bypass memory transfers between host and device and directly communicate each other using device memory if both producer and consumer blocks are running on the same device.

This example shows similar performance behaviors as the previous example.
Updated the programming system to use our new heterogeneous runtime system, called IRIS, as the common backend runtime.

- IRIS allows intermixing of multiple different output programming models (e.g., OpenMP3, OpenMP4, OpenACC, CUDA, HIP, etc.) and runs them on heterogeneous devices concurrently.

Developed a host-device memory transfer optimization scheme, which allows OpenACC GR blocks to bypass memory transfers between host and device and directly communicate each other if both producer and consumer blocks are running on the same device.

Performed preliminary evaluation of the new programming system by creating synthetic SDR workflow using the OpenACC GR blocks.

Next Steps

- Port more complex GR blocks to OpenACC and evaluate more complex SDR workflow.
- Continue to improve and fix bugs in the programming system.
Runtime systems for intelligent scheduling

Applications

- Streaming (e.g., SW Radio)
- Sensing (e.g., SAR, vision)
- Deep learning (e.g., CNN)
- Analytics (e.g., graphs)
- Robotics (e.g., sense and react)
- Science and Engineering (e.g., CFD)

Ontologies include Motifs, Parallel Execution Patterns, Locality, Affinity, Sync, etc.

Programming Systems

- Compiler
- DSL
- JIT
- Libraries
- etc

Performance Models combine Ontologies and Machine Models to produce both static and dynamic cost estimates.

Runtime and Operating Systems

- Task scheduling
- Memory Mgt
- IO
- Synchronization

Machine Models include architecture configuration information, microbenchmarks, and API for PFU

DSSoC Hardware

- Performance Parameters
- CPU
- GPU
- FPGA
- Accelerator
- DSP
- Deep Memory
- Neuromorphic

Aspen

- Ontologies: Structured, Composable, Interoperable
- Parameterizable Performance Models: Concurrency, Work, Communication, Capacities, Synchronization
- Machine Model: Compute, Memory, Interconnects
- PFU API
IRIS: An Intelligent Runtime System for Extremely Heterogeneous Architectures

- Provide programmers a unified programming environment to write portable code across heterogeneous architectures (and preferred programming systems)

- Orchestrate diverse programming systems (OpenCL, CUDA, HIP, OpenMP for CPU) in a single application
  - OpenCL
    - NVIDIA GPU, AMD GPU, ARM GPU, Qualcomm GPU, Intel CPU, Intel Xeon Phi, Intel FPGA, Xilinx FPGA
  - CUDA
    - NVIDIA GPU
  - HIP
    - AMD GPU
  - OpenMP for CPU
    - Intel CPU, AMD CPU, PowerPC CPU, ARM CPU, Qualcomm CPU

https://github.com/swiftcurrent2018
The IRIS Architecture

- **Platform Model**
  - A single-node system equipped with host CPUs and multiple compute devices (GPUs, FPGAs, Xeon Phis, and multicore CPUs)

- **Memory Model**
  - Host memory + shared device memory
  - All compute devices share the device memory

- **Execution Model**
  - DAG-style task parallel execution across all available compute devices

- **Programming Model**
  - High-level OpenACC, OpenMP4, SYCL* (*planned)
  - Low-level C/Fortran/Python IRIS host-side runtime API + OpenCL/CUDA/HIP/OpenMP kernels (w/o compiler support)
## Supported Architectures and Programming Systems by IRIS

<table>
<thead>
<tr>
<th>ExCL* Systems</th>
<th>Oswald</th>
<th>Summit-node</th>
<th>Radeon</th>
<th>Xavier</th>
<th>Snapdragon</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon</td>
<td>IBM Power9</td>
<td>Intel Xeon</td>
<td>ARMv8</td>
<td>Qualcomm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Kryo</td>
</tr>
<tr>
<td>Programming</td>
<td>Intel OpenMP</td>
<td>IBM XL OpenMP</td>
<td>Intel OpenMP</td>
<td>GNU GOMP</td>
<td>Android</td>
</tr>
<tr>
<td>Systems</td>
<td>Intel OpenCL</td>
<td></td>
<td>Intel OpenCL</td>
<td></td>
<td>NDK OpenMP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA P100</td>
<td>NVIDIA V100</td>
<td>AMD Radeon VII</td>
<td>NVIDIA Volta</td>
<td>Qualcomm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adreno 640</td>
</tr>
<tr>
<td>Programming</td>
<td>NVIDIA CUDA</td>
<td>NVIDIA CUDA</td>
<td>AMD HIP</td>
<td>NVIDIA CUDA</td>
<td></td>
</tr>
<tr>
<td>Systems</td>
<td>NVIDIA OpenCL</td>
<td></td>
<td>AMD OpenCL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPGA</td>
<td>Intel/Altera</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stratix 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programming</td>
<td>Intel OpenCL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Systems</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* ORNL Experimental Computing Laboratory (ExCL) [https://excl.ornl.gov/](https://excl.ornl.gov/)
IRIS Booting on Various Platforms
Task Scheduling in IRIS

- A task
  - A scheduling unit
  - Contains multiple in-order commands
    - Kernel launch command
    - Memory copy command (device-to-host, host-to-device)
  - May have DAG-style dependencies with other tasks
  - Enqueued to the application task queue with a device selection policy
    - Available device selection policies
      - Specific Device (compute device #)
      - Device Type (CPU, GPU, FPGA, XeonPhi)
      - Profile-based
      - Locality-aware
      - Ontology-base
      - Performance models (Aspen)
      - Any, All, Random, 3rd-party users’ custom policies

- The task scheduler dispatches the tasks in the application task queue to available compute devices
  - Select the optimal target compute device according to task’s device selection policy
SAXPY Example on Xavier

- Computation
  - $S[] = A \times X[] + Y[]$

- Two tasks
  - $S[] = A \times X[]$ on NVIDIA GPU (CUDA)
  - $S[] += Y[]$ on ARM CPU (OpenMP)
    - $S[]$ is shared between two tasks
    - Read-after-write (RAW), true dependency

- Low-level Python IRIS host code + CUDA/OpenMP kernels
  - saxpy.py
  - kernel.cu
  - kernel.openmp.h
SAXPY: Python host code & CUDA kernel code

<table>
<thead>
<tr>
<th>saxpy.py (1/2)</th>
<th>saxpy.py (2/2)</th>
<th>kernel.cu (CUDA)</th>
</tr>
</thead>
</table>
| #!/usr/bin/env python | #!/usr/bin/env python | extern "C" __global__ void saxpy0(float* S, float A, float* X) {
| import iris | kernel0 = iris.kernel("saxpy0") | int id = blockIdx.x * blockDim.x + threadIdx.x;
| import numpy as np | kernel0.setmem(0, mem_s, iris.iris_w) | S[id] = A * X[id];
| import sys | kernel0.setint(1, A) | }
| iris.init() | kernel0.setmem(2, mem_x, iris.iris_r) | extern "C" __global__ void saxpy1(float* S, float* Y) {
| SIZE = 1024 | off = [0] | int id = blockIdx.x * blockDim.x + threadIdx.x;
| A = 10.0 | ndr = [SIZE] | S[id] += Y[id];
| x = np.arange(SIZE, dtype=np.float32) | task0 = iris.task() | }
| y = np.arange(SIZE, dtype=np.float32) | task0.h2d_full(mem_x, x) | extern "C" __global__ void saxpy0(float* S, float A, float* X) {
| s = np.arange(SIZE, dtype=np.float32) | task0.kernel(kernel0, 1, off, ndr) | int id = blockIdx.x * blockDim.x + threadIdx.x;
| print 'X', x | task0.submit(iris.iris_gpu) | S[id] = A * X[id];
| print 'Y', y | | }
| mem_x = iris.mem(x.nbytes) | kernel1 = iris.kernel("saxpy1") | extern "C" __global__ void saxpy1(float* S, float* Y) {
| mem_y = iris.mem(y.nbytes) | kernel1.setmem(0, mem_s, iris.iris_rw) | int id = blockIdx.x * blockDim.x + threadIdx.x;
| mem_s = iris.mem(s.nbytes) | kernel1.setmem(1, mem_y, iris.iris_r) | S[id] += Y[id];
| | task1 = iris.task() | }
| | task1.h2d_full(mem_y, y) | |
| | task1.kernel(kernel1, 1, off, ndr) | |
| | task1.d2h_full(mem_s, s) | |
| | task1.submit(iris.iris_cpu) | |
| | print 'S =', A, '* X + Y', s | |
| | iris.finalize() | |
SAXPY: Python host code & OpenMP kernel code

**saxpy.py (1/2)**

```python
#!/usr/bin/env python

import iris
import numpy as np
import sys
iris.init()

SIZE = 1024
A = 10.0

x = np.arange(SIZE, dtype=np.float32)
y = np.arange(SIZE, dtype=np.float32)
s = np.arange(SIZE, dtype=np.float32)
print 'X', x
print 'Y', y

mem_x = iris.mem(x.nbytes)
mem_y = iris.mem(y.nbytes)
mem_s = iris.mem(s.nbytes)

**saxpy.py (2/2)**

```python
kernel0 = iris.kernel("saxpy0")
kernel0.setmem(0, mem_s, iris.iris_w)
kernel0.setint(1, A)
kernel0.setmem(2, mem_x, iris.iris_r)
off = [ 0 ]
dr = [ SIZE ]
task0 = iris.task()
task0.h2d_full(mem_x, x)
task0.kernel(kernel0, 1, off, dr)
task0.submit(iris.iris_gpu)

kernel1 = iris.kernel("saxpy1")
kernel1.setmem(0, mem_s, iris.iris_rw)
kernel1.setmem(1, mem_y, iris.iris_r)

```

**kernel.openmp.h (OpenMP)**

```c
#include <iris/iris_openmp.h>

static void saxpy0(float* S, float A, float* X, IRIS_OPENMP_KERNEL_ARGS) {
    int id;
    #pragma omp parallel for shared(S, A, X)
    private(id)
    IRIS_OPENMP_KERNEL_BEGIN
    S[id] = A * X[id];
    IRIS_OPENMP_KERNEL_END
}

static void saxpy1(float* S, float* Y, IRIS_OPENMP_KERNEL_ARGS) {
    int id;
    #pragma omp parallel for shared(S, Y)
    private(id)
    IRIS_OPENMP_KERNEL_BEGIN
    S[id] += Y[id];
    IRIS_OPENMP_KERNEL_END
}
```

print 'S =', A, '* X + Y', s
iris.finalize()
Memory Consistency Management

saxpy.py (1/2)

#!/usr/bin/env python
import iris
import numpy as np
import sys
iris.init()
SIZE = 1024
A = 10.0
x = np.arange(SIZE, dtype=np.float32)
y = np.arange(SIZE, dtype=np.float32)
s = np.arange(SIZE, dtype=np.float32)
print 'X', x
print 'Y', y
mem_x = iris.mem(x.nbytes)
mem_y = iris.mem(y.nbytes)
mem_s = iris.mem(s.nbytes)

saxpy.py (2/2)

kernel0 = iris.kernel("saxpy0")
kernel0.setmem(0, mem_s, iris.iris_w)
kernel0.setint(1, A)
kernel0.setmem(2, mem_x, iris.iris_r)
off = [0]
ndr = [SIZE]
task0 = iris.task()
task0.h2d_full(mem_x, x)
task0.kernel(kernel0, 1, off, ndr)
task0.submit(iris.iris_gpu)

kernel1 = iris.kernel("saxpy1")
kernel1.setmem(0, mem_s, iris.iris_rw)
kernel1.setmem(1, mem_y, iris.iris_r)
task1 = iris.task()
task1.h2d_full(mem_y, y)
task1.kernel(kernel1, 1, off, ndr)
task1.d2h_full(mem_s, s)
task1.submit(iris.iris_cpu)

print 'S =', A, '* X + Y', s
iris.finalize()
Locality-aware Device Selection Policy

saxpy.py (1/2)

```python
#!/usr/bin/env python
import iris
import numpy as np
import sys
iris.init()
SIZE = 1024
A = 10.0
x = np.arange(SIZE, dtype=np.float32)
y = np.arange(SIZE, dtype=np.float32)
s = np.arange(SIZE, dtype=np.float32)
print 'X', x
print 'Y', y
mem_x = iris.mem(x.nbytes)
mem_y = iris.mem(y.nbytes)
mem_s = iris.mem(s.nbytes)
```

saxpy.py (2/2)

```python
kernel0 = iris.kernel("saxpy0")
kernel0.setmem(0, mem_s, iris.iris_w)
kernel0.setint(1, A)
kernel0.setmem(2, mem_x, iris.iris_r)
off = [ 0 ]
ndr = [ SIZE ]
task0 = iris.task()
task0.h2d_full(mem_x, x)
task0.kernel(kernel0, 1, off, ndr)
task0.submit(iris.iris_gpu)

kernel1 = iris.kernel("saxpy1")
kernel1.setmem(0, mem_s, iris.iris_rw)
kernel1.setmem(1, mem_y, iris.iris_r)
task1 = iris.task()
task1.h2d_full(mem_y, y)
task1.kernel(kernel1, 1, off, ndr)
task1.d2h_full(mem_s, s)
task1.submit(iris.iris_data)
print 'S =', A, "* X + Y", s
iris.finalize()
```

iris_data selects the device that requires minimum data transfer to execute the task
IRIS: Task Scheduling Overhead – Running One Million (Empty) Tasks

```
#!/usr/bin/env python
import iris
iris.init()
NTASKS = 1000000

t0 = iris.timer_now()
for i in range(NTASKS):
    task = iris.task()
    task.submit(iris.iris_random, False)
iris.synchronize()

t1 = iris.timer_now()
print 'Time:', t1 - t0
iris.finalize()
```

user@xavier:~/work$ ./ntasks.py
Time: 11.46s

<table>
<thead>
<tr>
<th>Throughput</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>87,268 tasks/sec</td>
<td>11.4 μs/task</td>
</tr>
</tbody>
</table>

asynchronous task submission
concurrent tasks execution on multiple devices

CPU or GPU randomly
Closing

Summary

• Architectural specialization

• Performance portability of applications and software

• DSSoC ORNL project investigating on performance portability of SDR
  – Understand applications and target architectures
  – Use open programming models: OpenACC, OpenCL, OpenMP
  – Developing intelligent runtime systems: IRIS

• Goal: scale applications from Qualcomm Snapdragon to DoE Summit Supercomputer with minimal programmer effort

• Work continues...

Acknowledgements

• Thanks to staff and students for the work!

• Thanks to DARPA, DOE for funding our work!

• This research was developed, in part, with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. This document is approved for public release: distribution unlimited.