Oak Ridge National Laboratory

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

U.S. DEPARTMENT OF ENERGY

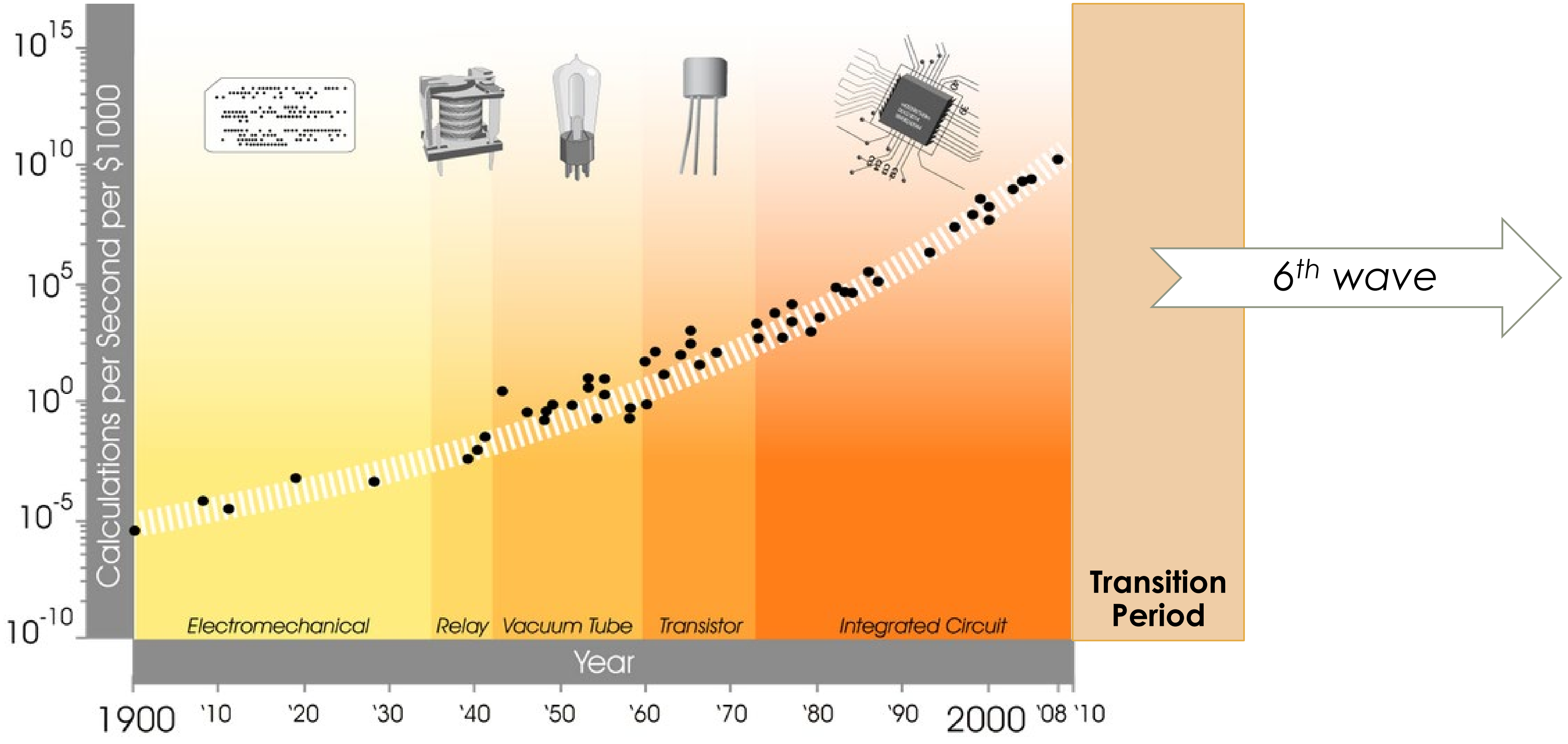
Future Technologies Group

http://ft.ornl.gov vetter@computer.org

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

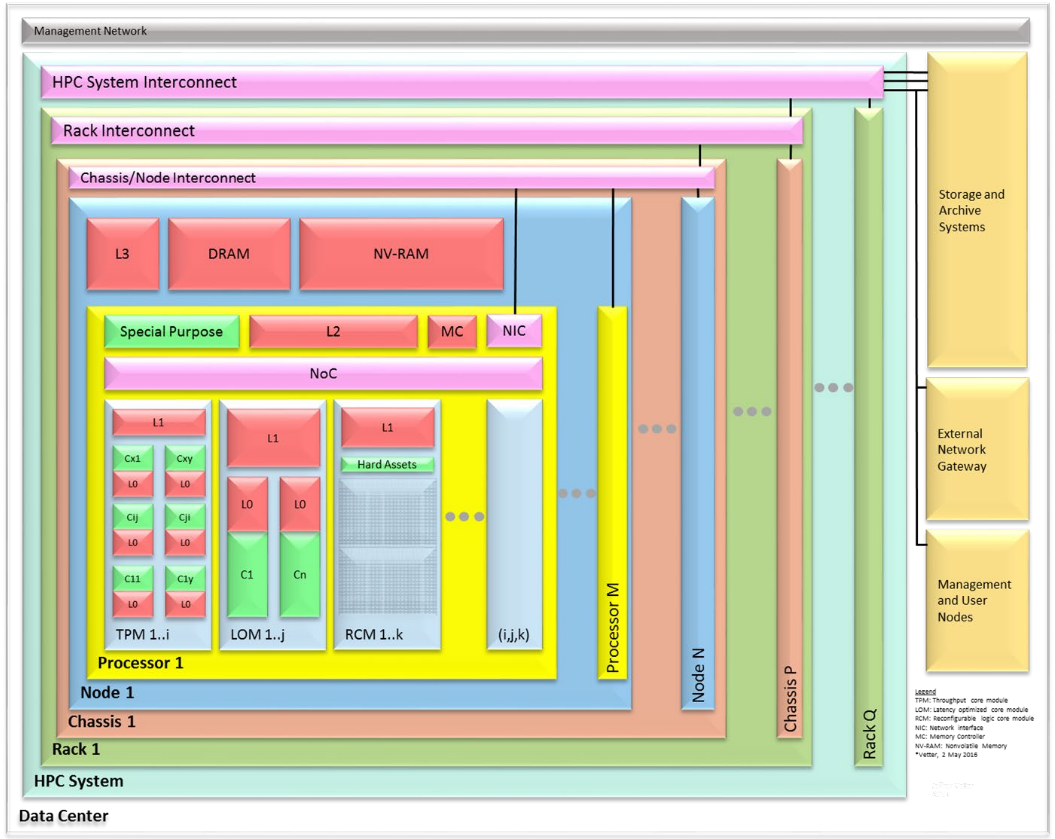
- Use CMOS more effectively for specific workloads
- Integrate components to boost performance and eliminate inefficiencies
- Workload specific memory+storage system design

Emerging Technologies

- Investigate new computational paradigms
  - Quantum
  - Neuromorphic
  - Advanced Digital
  - Emerging Memory Devices

OAK RIDGE National Laboratory

Complex architectures yield…



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

- Low overhead
- Resource contention
- Locality

Node: OpenMP, Pthreads, U-threads, etc

- SIMD
- NUMA, HBM

Cores: OpenACC, CUDA, OpenCL, OpenMP4, …

- Memory use, coalescing
- Data orchestration
- Fine grained parallelism
- Hardware features

Complex Programming Models

During this Sixth Wave transition, Complexity is our major challenge!

Design: How do we design future systems so that they are better than current systems on mission applications?

- Entirely possible that the new system will be slower than the old system!
- Expect 'disaster' procurements

Programmability: How do we design applications with some level of performance portability?

- Software lasts much longer than transient hardware platforms
- Adapt or die

OAK RIDGE
National Laboratory

DARPA Domain-Specific System on a Chip (DSSoC) Program
Getting the best out of specialization when we need programmability

DARPA ERI DSSoC Program: Dr. Tom Rondeau

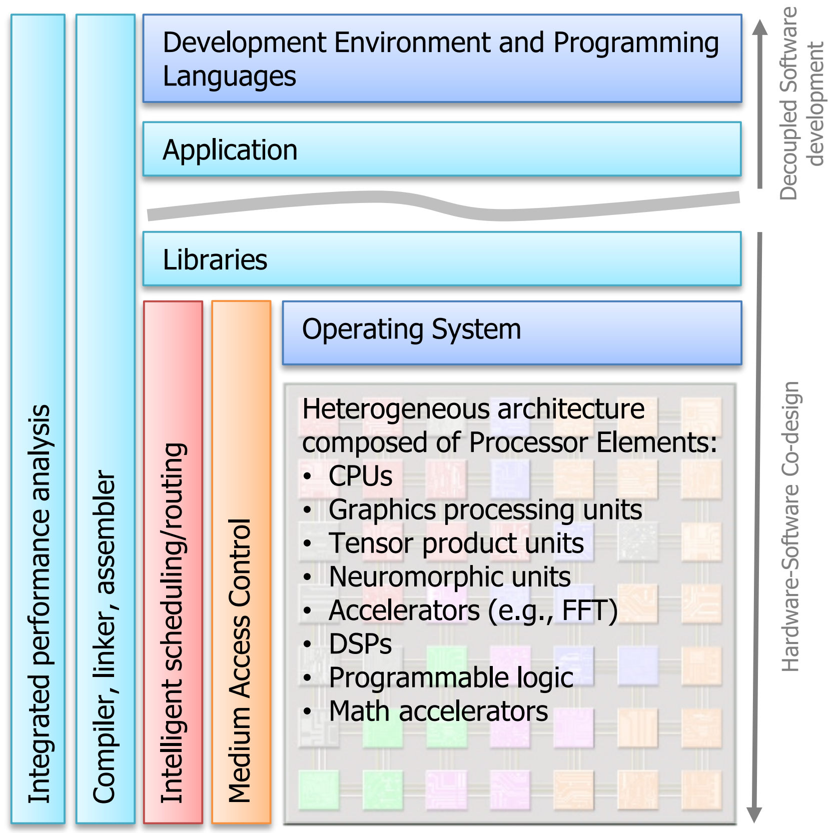
Three Optimization Areas

1. Design time
2. Run time
3. Compile time

Addressed via five program areas

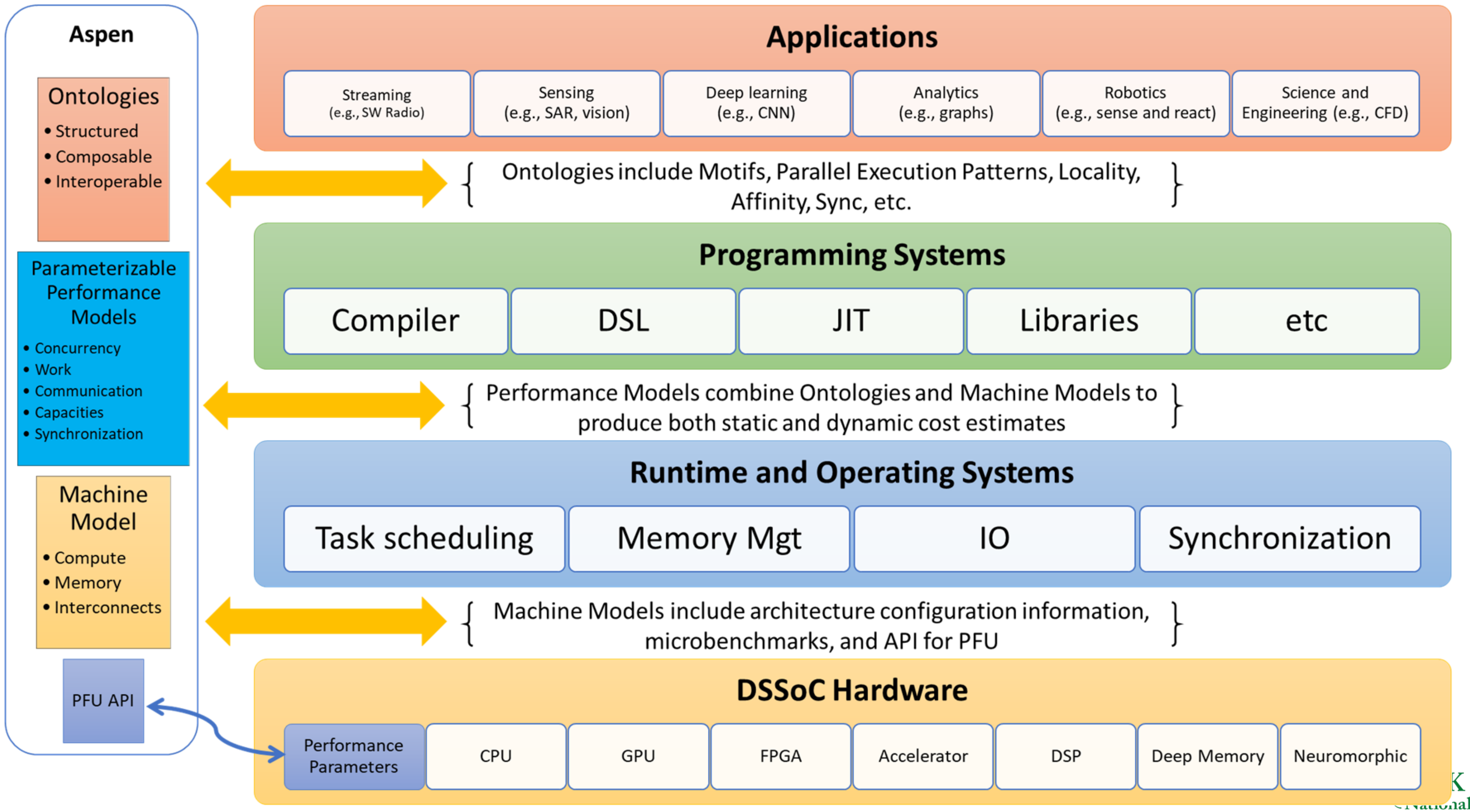
1. Intelligent scheduling
2. Domain representations
3. Software
4. Medium access control (MAC)
5. Hardware integration

DSSoC's Full-Stack Integration

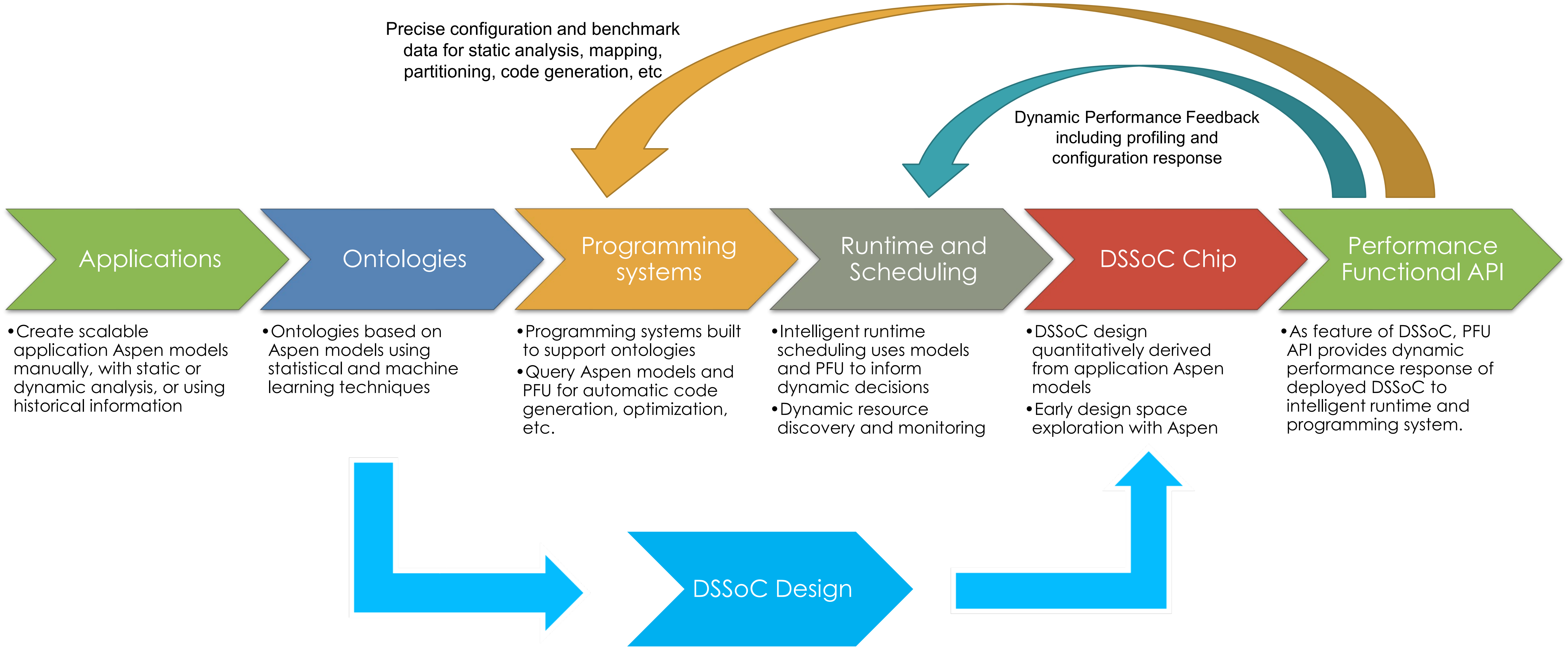

Development Environment and Programming Languages
Application
Libraries
Operating System
Heterogeneous architecture composed of Processor Elements:
• CPUs
• Graphics processing units
• Tensor product units
• Neuromorphic units
• Accelerators (e.g., FFT)
• DSPs
• Programmable logic
• Math accelerators
Integrated performance analysis
Compiler, linker, assembler
Intelligent scheduling/routing
Medium Access Control
Decoupled Software development
Hardware-Software Co-design

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
Performance Parameters
CPU
GPU
FPGA
Accelerator
DSP
Deep Memory
Neuromorphic

Development Lifecycle


Precise configuration and benchmark data for static analysis, mapping, partitioning, code generation, etc
Dynamic Performance Feedback including profiling and configuration response
Applications
Ontologies
Programming systems
Runtime and Scheduling
DSSoC Chip
Performance Functional API
DSSoC Design

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.

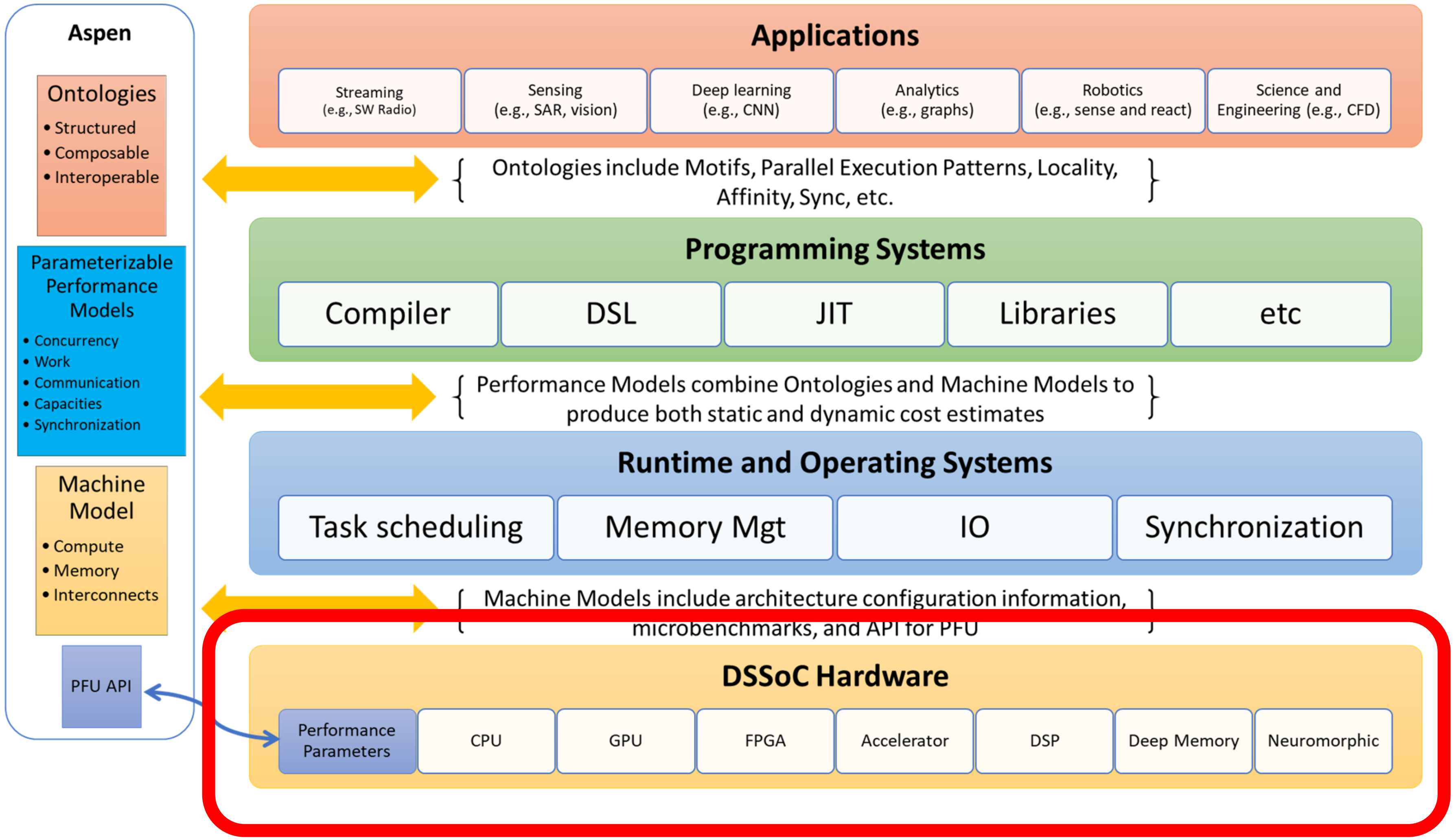
Runtime and Scheduling
- Intelligent runtime scheduling uses models and PFU to inform dynamic decisions
- Dynamic resource discovery and monitoring

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.

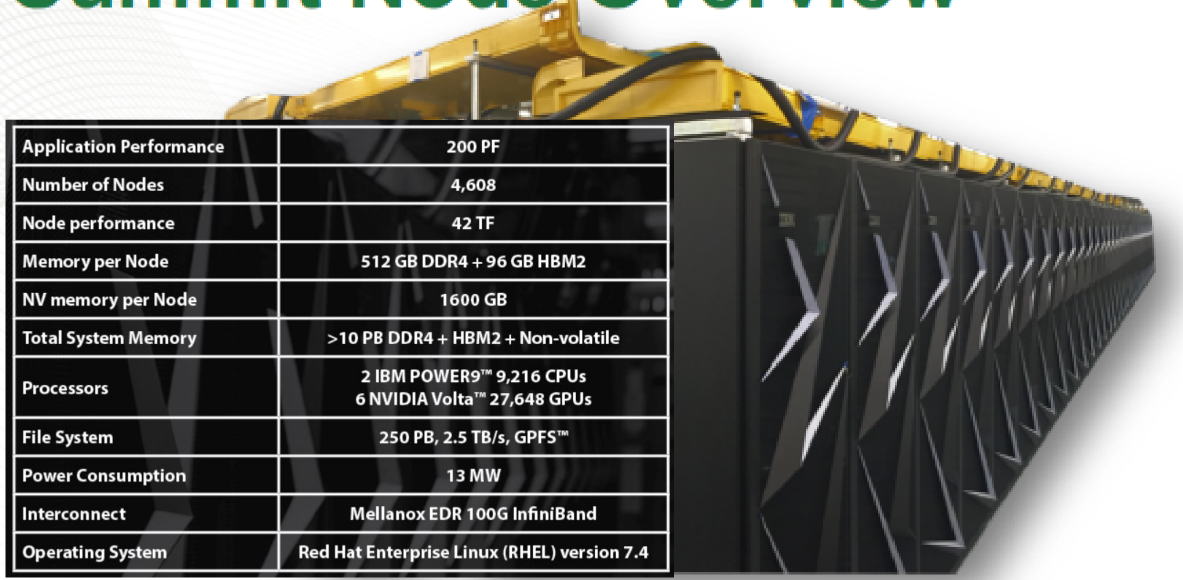
DSSoC Design
- DSSoC design quantitatively derived from application Aspen models
- Early design space exploration with Aspen

Architectures

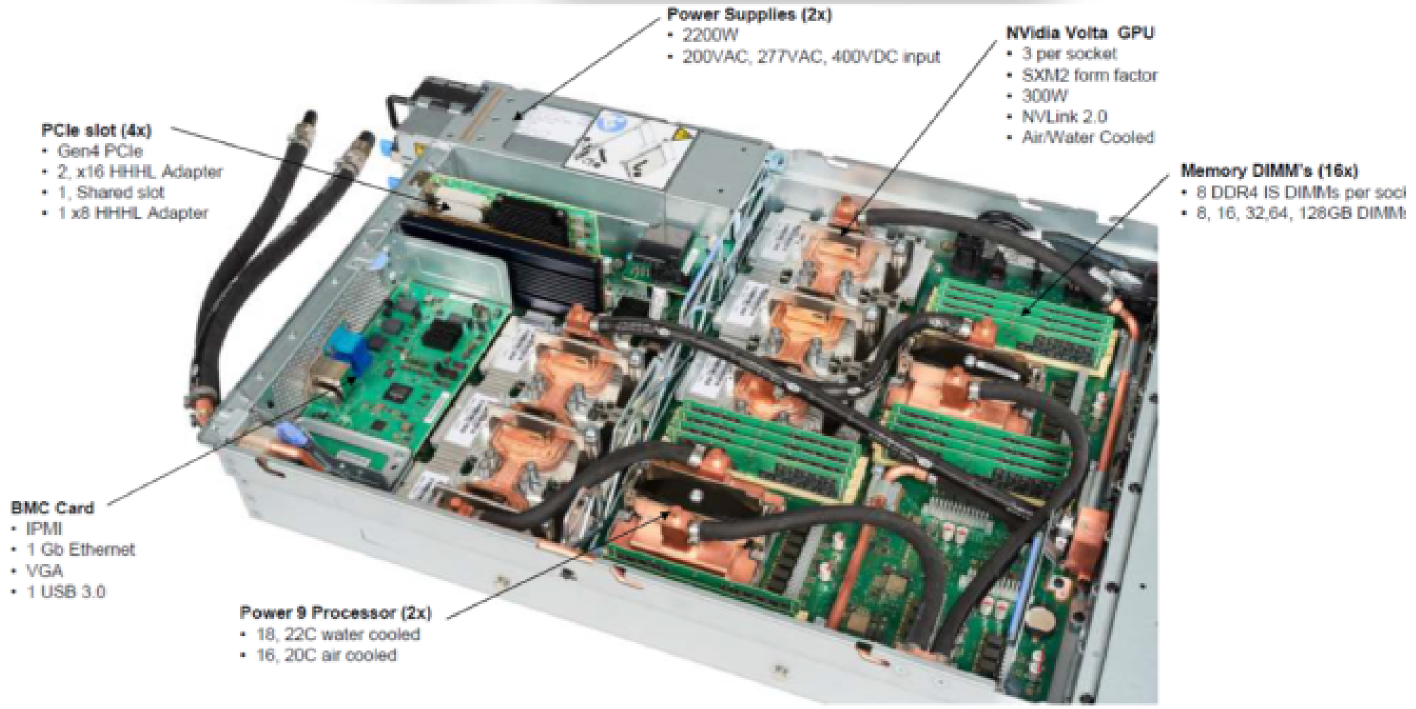

Aspen
Ontologies
• Structured
• Composable
• Interoperable
Parameterizable Performance Models
• Concurrency
• Work
• Communication
• Capacities
• Synchronization
Machine Model
• Compute
• Memory
• Interconnects
PFU API
Applications
Streaming (e.g., SW Radio)
Sensing (e.g., SAR, vision)
Deep learning (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

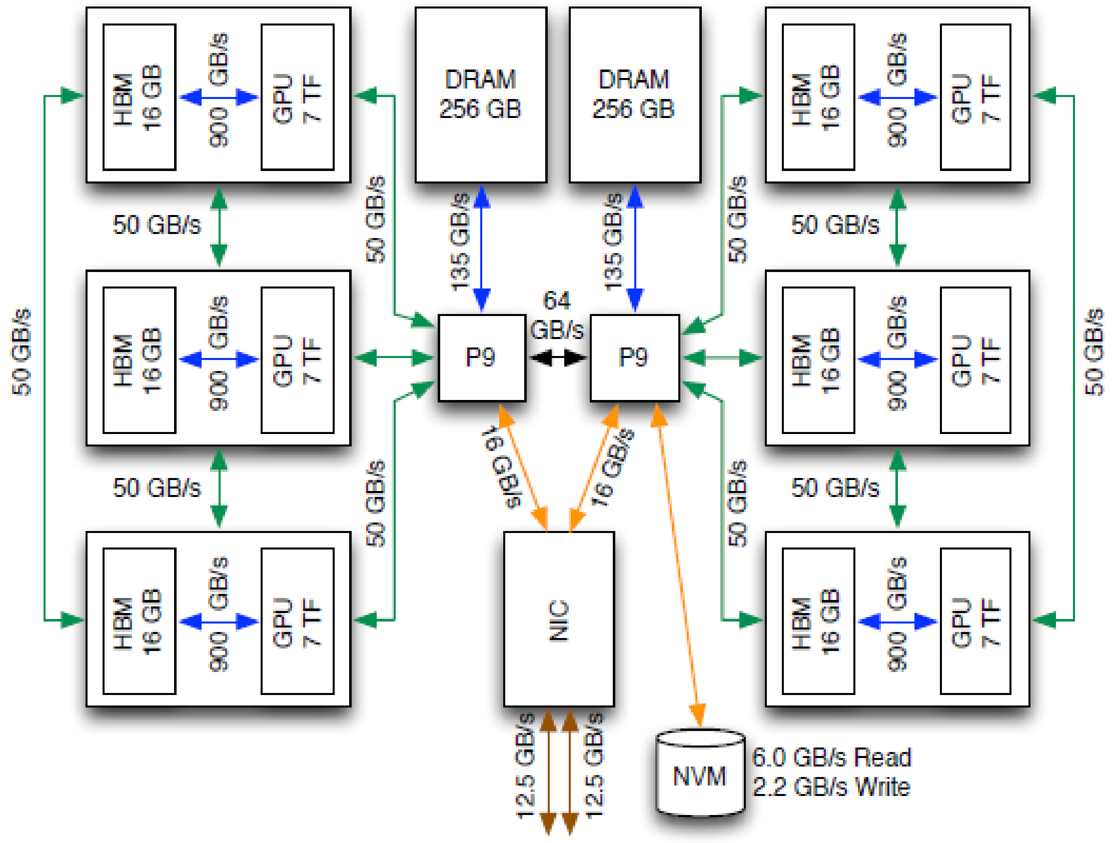
OAK RIDGE National Laboratory

Summit Node Overview



Application Performance	200 PF
Number of Nodes	4,608
Node performance	42 TF
Memory per Node	512 GB DDR4 + 96 GB HBM2
NV memory per Node	1600 GB
Total System Memory	>10 PB DDR4 + HBM2 + Non-volatile
Processors	2 IBM POWER9™ 9,216 CPUs 6 NVIDIA Volta™ 27,648 GPUs
File System	250 PB, 2.5 TB/s, GPFS™
Power Consumption	13 MW
Interconnect	Mellanox EDR 100G InfiniBand
Operating System	Red Hat Enterprise Linux (RHEL) version 7.4





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

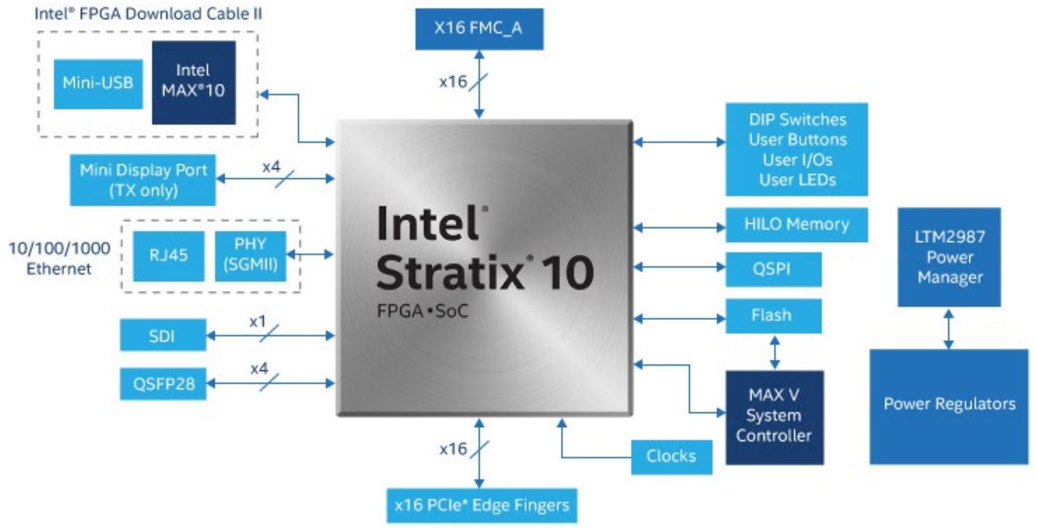
⟷	HBM/DRAM Bus (aggregate B/W)
⟷	NVLINK
⟷	X-Bus (SMP)

HBM & DRAM speeds are aggregate (Read+Write).
All other speeds (X-Bus, NVLink, PCIe, IB) 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

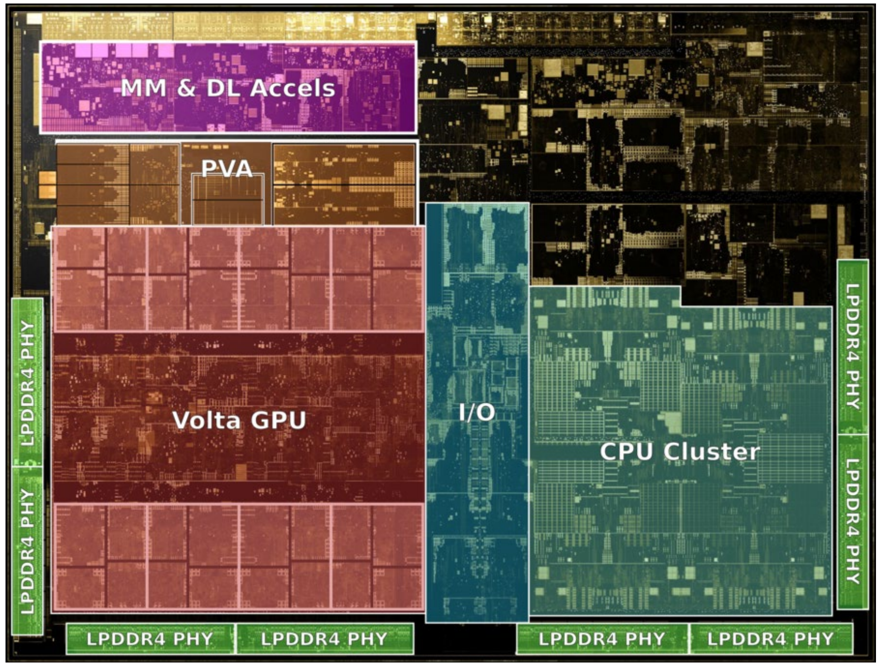


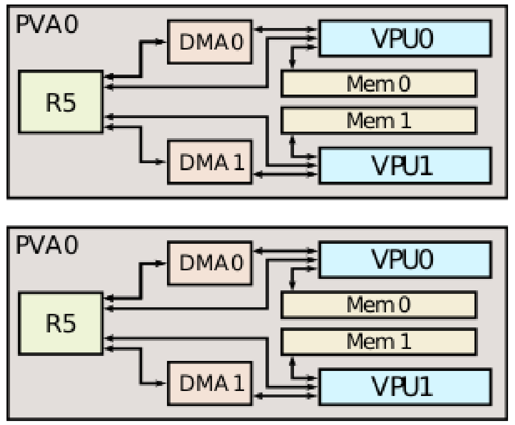
For more information or to apply for an account, visit https://excl.ornl.gov/

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





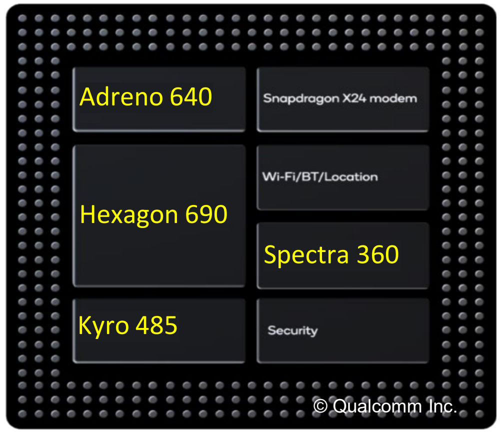
For more information or to apply for an account, visit https://excl.ornl.gov/

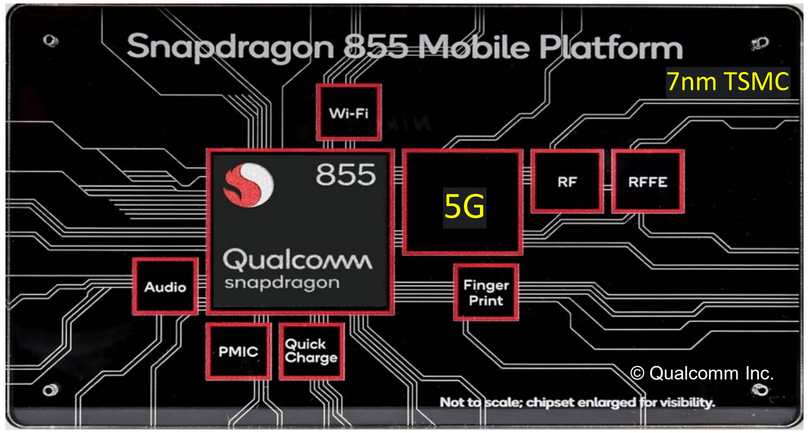
Mar 2019

https://excl.ornl.gov/

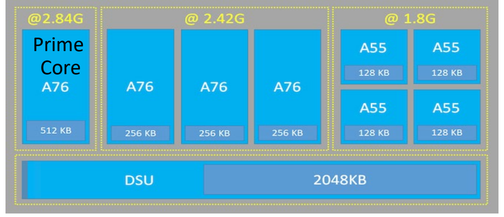
Qualcomm 855 SoC (SM8510P) Snapdragon™

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





Kyro 485 (8-ARM Prime+BigLittle Cores)



Hexagon 690 (DSP + AI)

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

Adreno 640

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

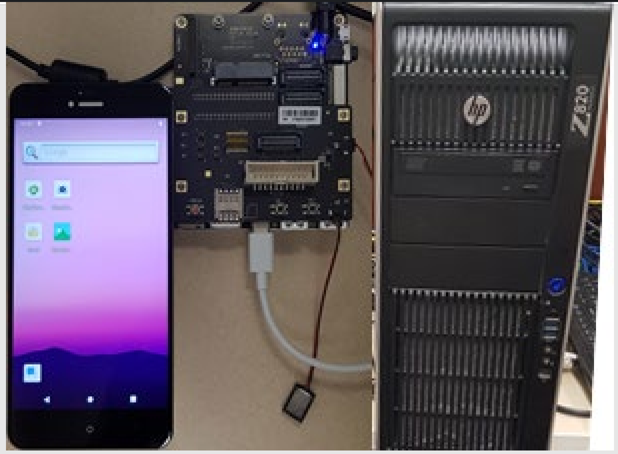
Connectivity (5G)

- 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.

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.

Qualcomm Development Board connected to (mcmurdo) HPZ820



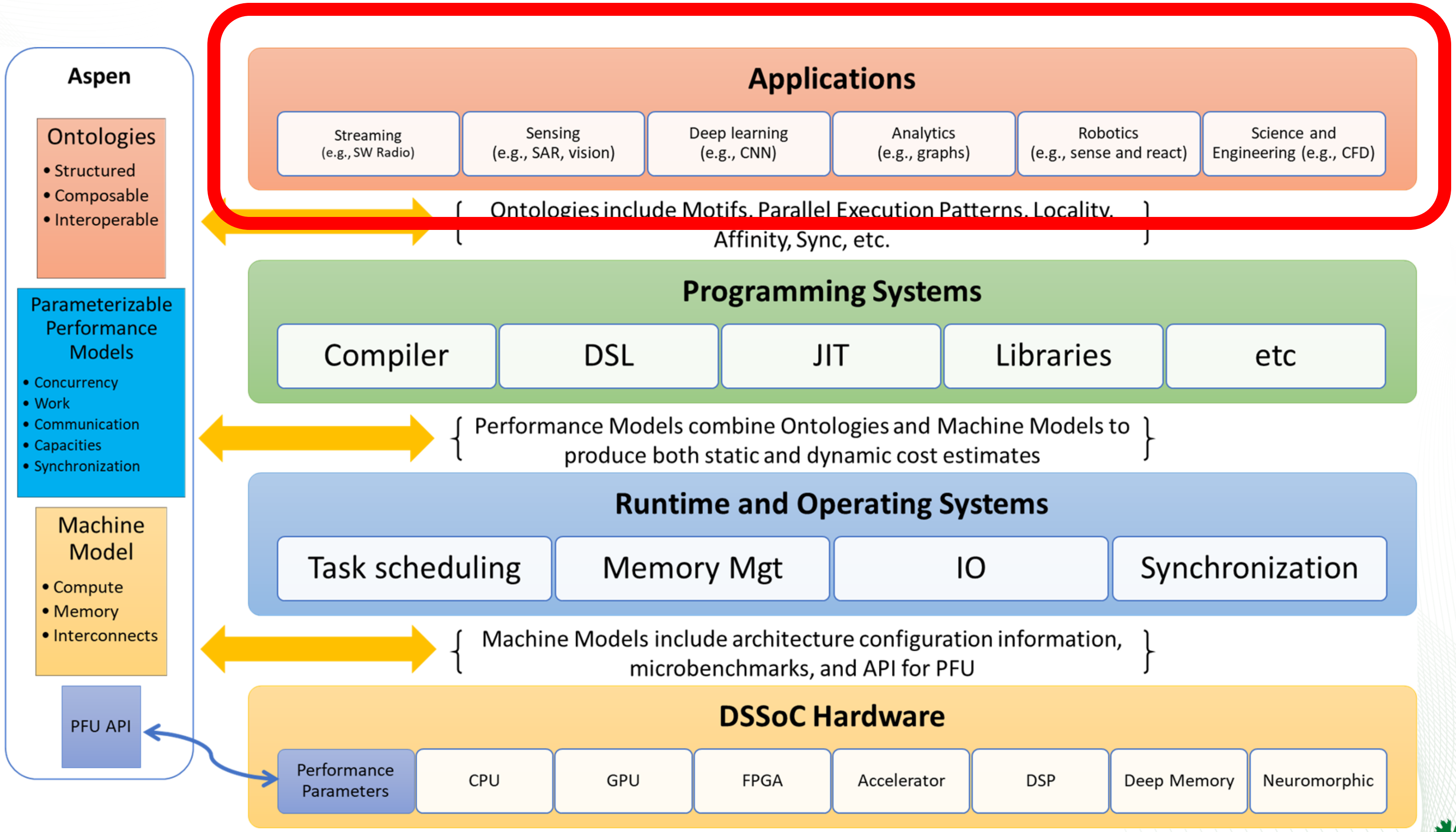
- Connected Qualcomm board to HPZ820 through USB
- Development Environment: Android SDK/NDK
- Login to mcmurdo machine
  ```
  $ ssh –Y mcmurdo
  ```
- Setup Android platform tools and development environment
  ```
  $ source /home/nqx/setup_android.source
  ```
- Run Hello-world on ARM cores
  ```
  $ 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/

Oak Ridge National Laboratory

Created by Narasinga Rao Miniskar, Steve Moulton

Applications

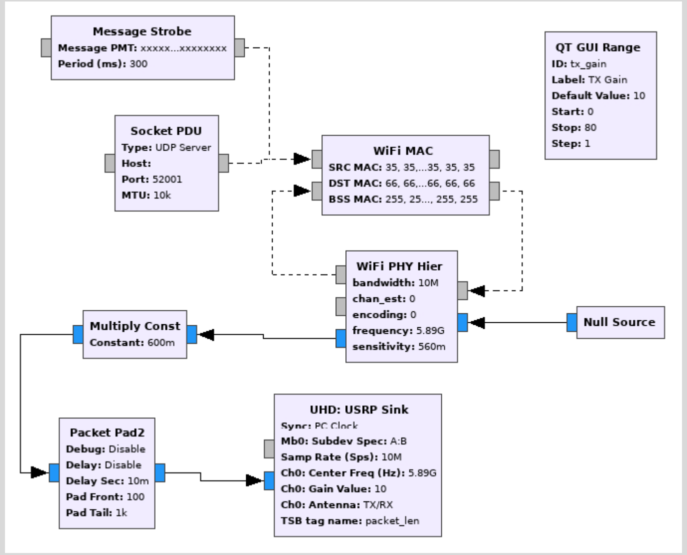

Aspen
Ontologies
• Structured
• Composable
• Interoperable
Parameterizable Performance Models
• Concurrency
• Work
• Communication
• Capacities
• Synchronization
Machine Model
• Compute
• Memory
• Interconnects
PFU API
Applications
Streaming (e.g., SW Radio)
Sensing (e.g., SAR, vision)
Deep learning (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

End-to-End System: Gnu Radio for Wifi on two NVIDIA Xavier SoCs

Xavier SoC #1

Video/Image Files

UDP



Message Strobe
Message PMT: xxxxx...xxxxxxxx
Period (ms): 300

Socket PDU
Type: UDP Server
Host:
Port: 52001
MTU: 10k

QT GUI Range
ID: tx_gain
Label: TX Gain
Default Value: 10
Start: 0
Stop: 80
Step: 1

WiFi MAC
SRC MAC: 35, 35,...35, 35, 35
DST MAC: 66, 66,...66, 66, 66
BSS MAC: 255, 25..., 255, 255

WiFi PHY Hier
bandwidth: 10M
chan_est: 0
encoding: 0
frequency: 5.89G
sensitivity: 560m

Multiply Const
Constant: 600m

Null Source

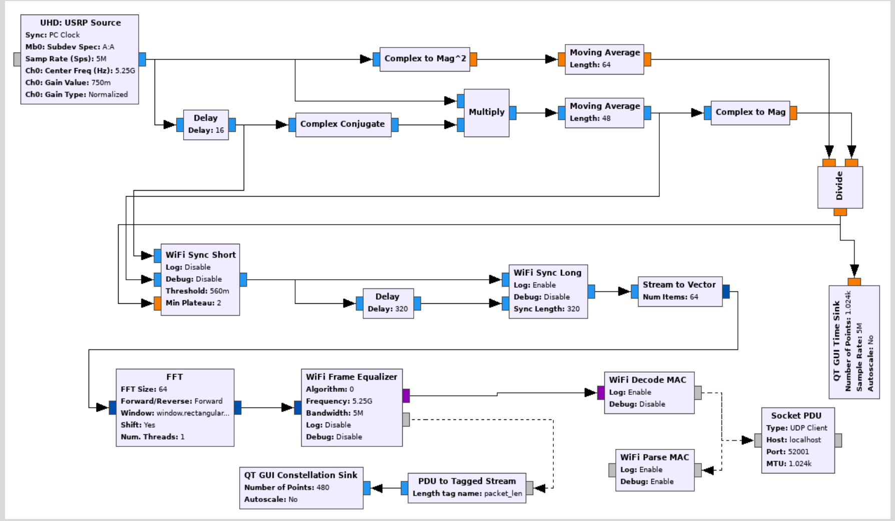
Packet Pad2
Debug: Disable
Delay: Disable
Delay Sec: 10m
Pad Front: 100
Pad Tail: 1k

UHD: USRP Sink
Sync: PC Clock
Mb0: Subdev Spec: A:B
Samp Rate (Sps): 10M
Ch0: Center Freq (Hz): 5.89G
Ch0: Gain Value: 10
Ch0: Antenna: TX/RX
TSB tag name: packet_len

GR IEEE-802.11 Transmit (TX)

Antenna

Xavier SoC #2



UHD: USRP Source
Sync: PC Clock
Mb0: Subdev Spec: A:A
Samp Rate (Sps): 5M
Ch0: Center Freq (Hz): 5.25G
Ch0: Gain Value: 750m
Ch0: Gain Type: Normalized

Complex to Mag^2

Moving Average
Length: 64

Delay
Delay: 16

Complex Conjugate

Multiply

Moving Average
Length: 48

Complex to Mag

Divide

WiFi Sync Short
Log: Disable
Debug: Disable
Threshold: 560m
Min Plateau: 2

Delay
Delay: 320

WiFi Sync Long
Log: Enable
Debug: Disable
Sync Length: 320

Stream to Vector
Num Items: 64

QT GUI Time Sink
Number of Points: 1.024k
Sample Rate: 5M
Autoscale: No

FFT
FFT Size: 64
Forward/Reverse: Forward
Window: window.rectangular...
Shift: Yes
Num. Threads: 1

WiFi Frame Equalizer
Algorithm: 0
Frequency: 5.25G
Bandwidth: 5M
Log: Disable
Debug: Disable

WiFi Decode MAC
Log: Enable
Debug: Disable

Socket PDU
Type: UDP Client
Host: localhost
Port: 52001
MTU: 1.024k

WiFi Parse MAC
Log: Enable
Debug: Enable

QT GUI Constellation Sink
Number of Points: 480
Autoscale: No

PDU to Tagged Stream
Length tag name: packet_len

IEEE-802.11 Receive (RX)

UDP

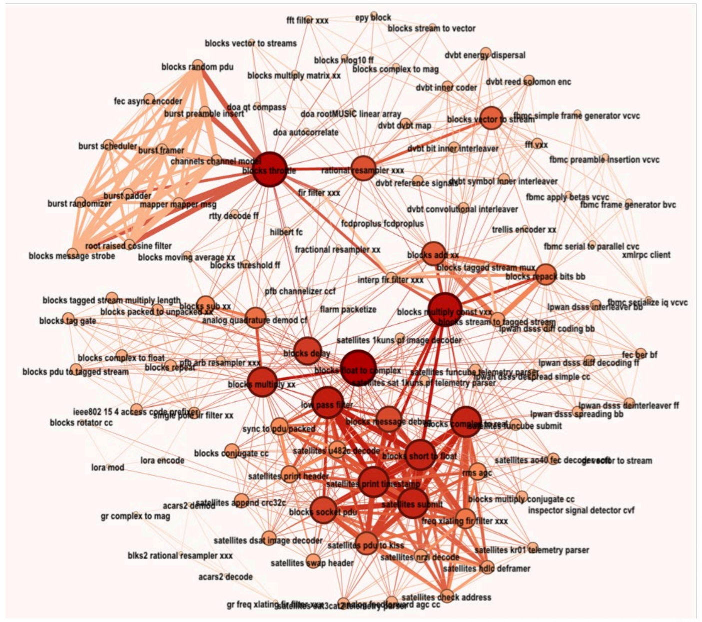
OpenCV

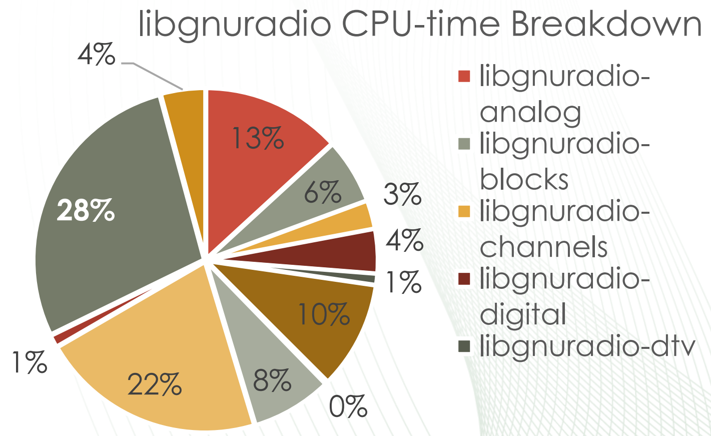
OpenCL

- 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

- 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-clenabled
        - 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





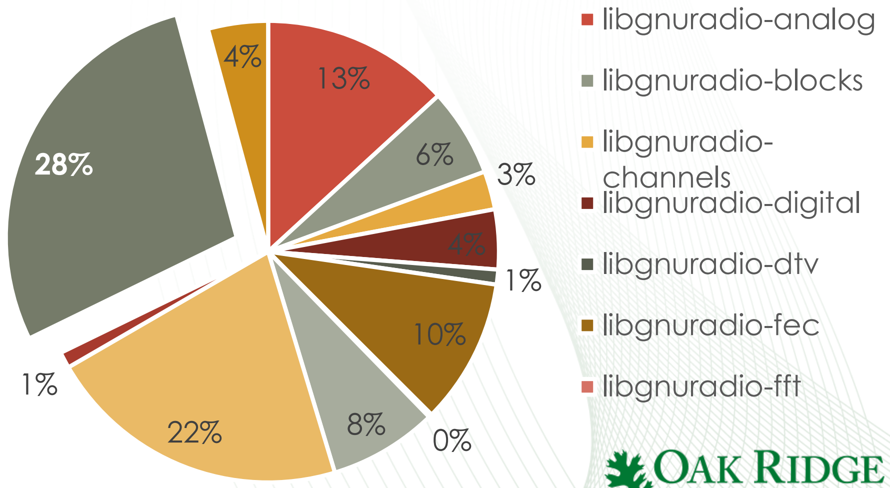
https://github.com/cosmic-sdr

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.kallsysms, 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
[kernel.kallsyms]	27.8547
libpython	18.6281
libgnuradio	**11.7548**
libc	7.7503
ld	3.8839
libvolk	**3.7963**
libperl	3.7837
[unknown]	3.6465
libQt5	2.9866
libpthread	2.1449

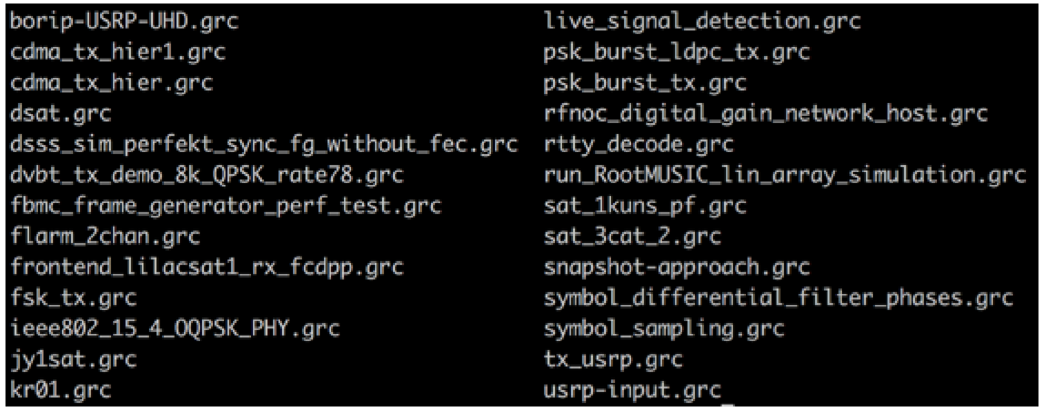
libgnuradio CPU-time Breakdown



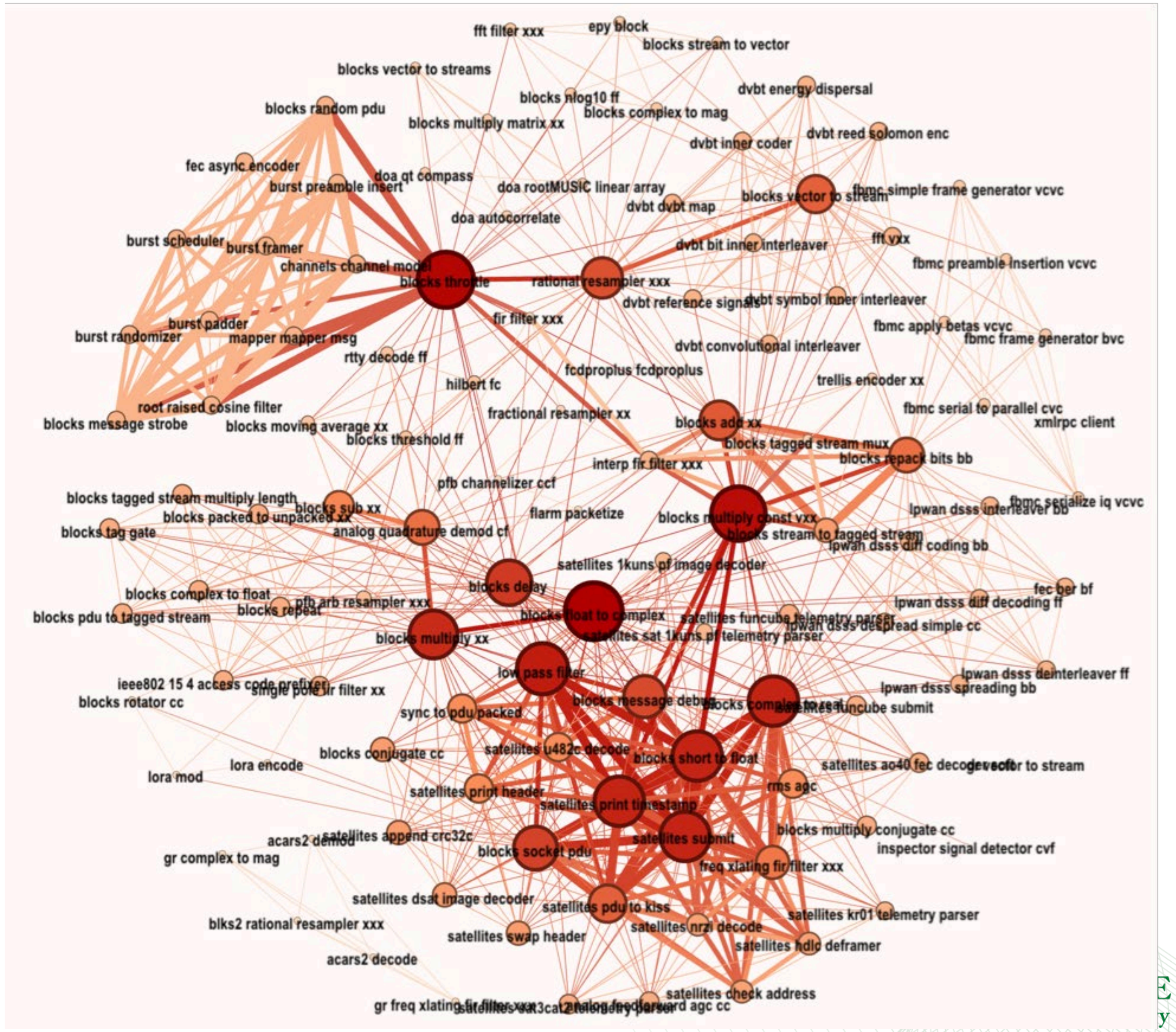
GRC statistics: Block Proximity Analysis

Block proximity analysis

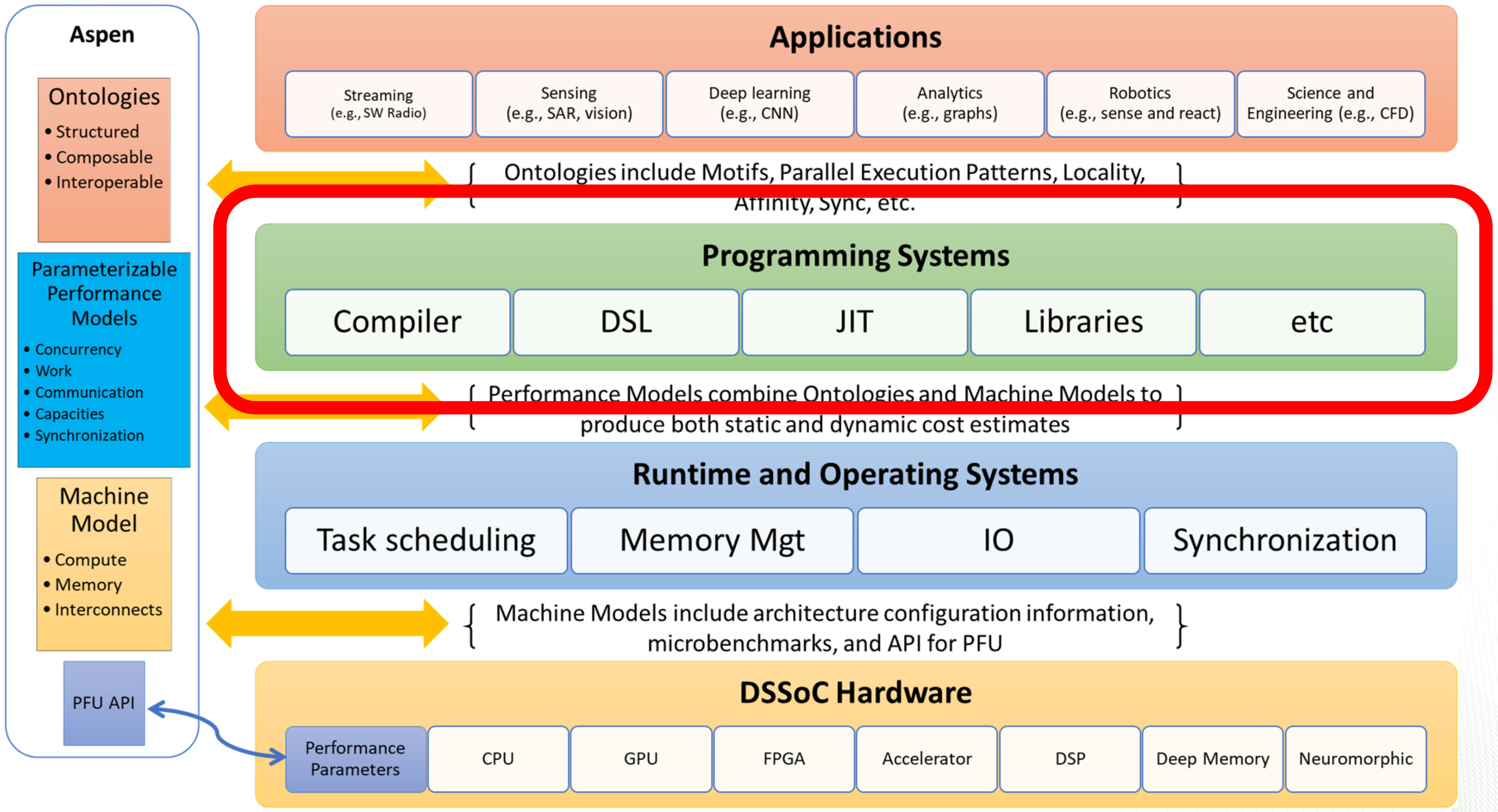
- Creates a graph:
  - Nodes: Unique block types
  - Edges: Blocks used in the same GRC file.
  - Every co-occurrence increases edge weight by 1.
- This example was run
  - With `--mode proximityGraph`
  - On randomly selected sub-set of GRC files



```
borip-USRP-UHD.grc                          live_signal_detection.grc
cdma_tx_hier1.grc                           psk_burst_ldpc_tx.grc
cdma_tx_hier.grc                            psk_burst_tx.grc
dsat.grc                                    rfnoc_digital_gain_network_host.grc
dsss_sim_perfekt_sync_fg_without_fec.grc    rtty_decode.grc
dvbt_tx_demo_8k_QPSK_rate78.grc             run_RootMUSIC_lin_array_simulation.grc
fbmc_frame_generator_perf_test.grc          sat_1kuns_pf.grc
flarm_2chan.grc                             sat_3cat_2.grc
frontend_lilacsat1_rx_fcdpp.grc             snapshot-approach.grc
fsk_tx.grc                                  symbol_differential_filter_phases.grc
ieee802_15_4_OQPSK_PHY.grc                  symbol_sampling.grc
jy1sat.grc                                  tx_usrp.grc
kr01.grc                                    usrp-input.grc
```



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

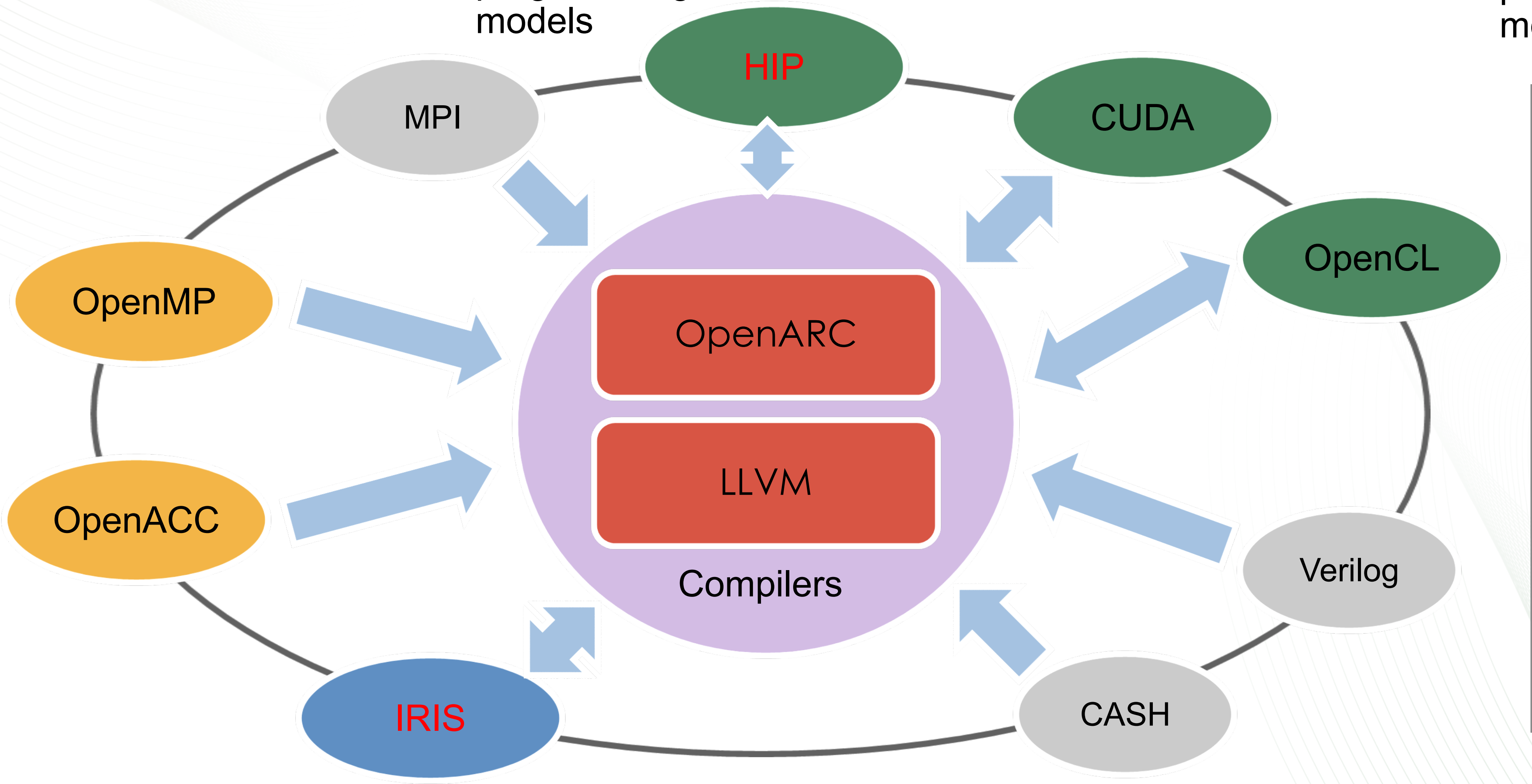
Programming Solution for DSSoC

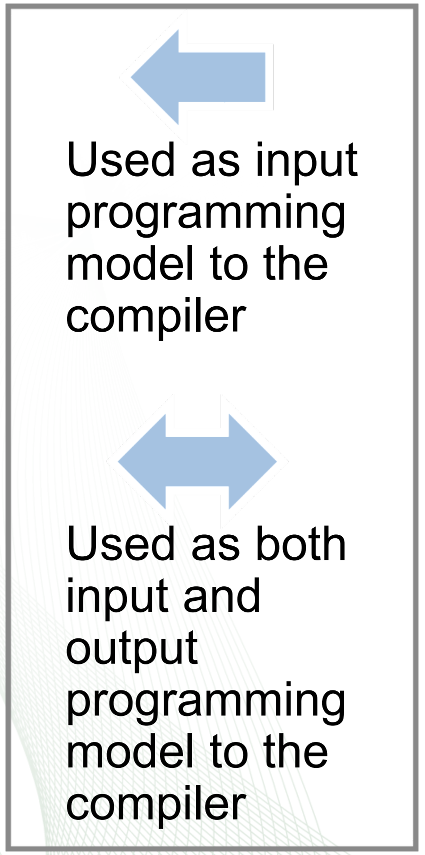
🟧 Main input programming models

🟦 Main input/ main output programming models

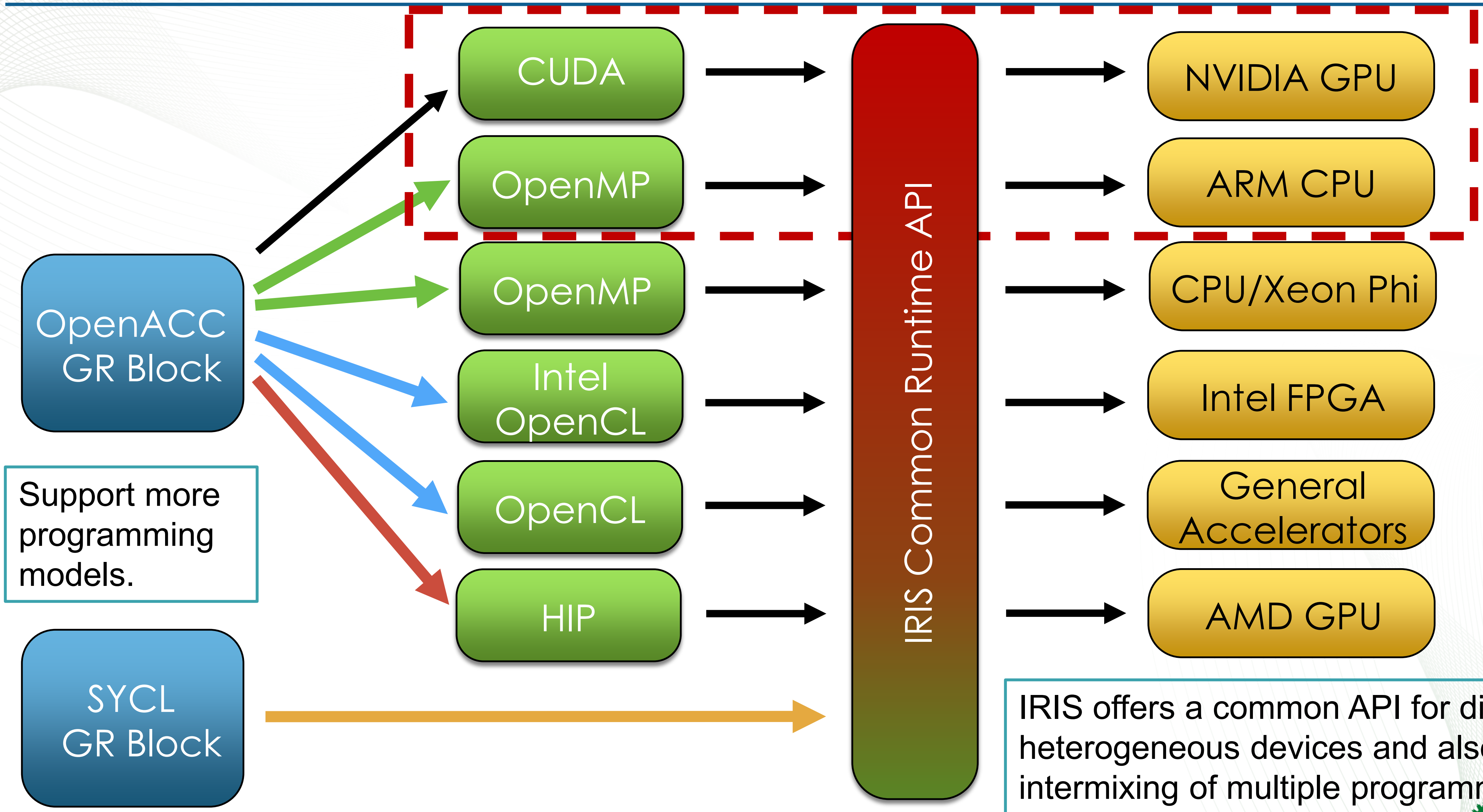
⬜ Optional input programming models

🟩 Main output/ optional input programming models




Used as input programming model to the compiler
Used as both input and output programming model to the compiler

New OpenACC GR Block Mapping Strategy for Heterogeneous Architectures



Support more programming models.

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<noi;i++) {
        out[i] = n_val * log10(in1[i]) + k_val;
    }
    ...
}

//OpenACC implementation
int accLog_impl::testOpenACC(...) {
    ...
    if( acc_init_done == 0 ) {
        gracc_pcopyin(...); //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 == ACCTYPE_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

```
void accLog_init(acc_device_t deviceType, ...) {
    ...
    acc_init(deviceType);
}

void accLog_kernel(...) {
#pragma acc kernels loop gang worker present(in, out)
    for( int i=0; i<noutput_items; i++ ) {
        out[i] = n_val * log10(in[i]) + k_val;
    }
}
```

Input OpenACC code

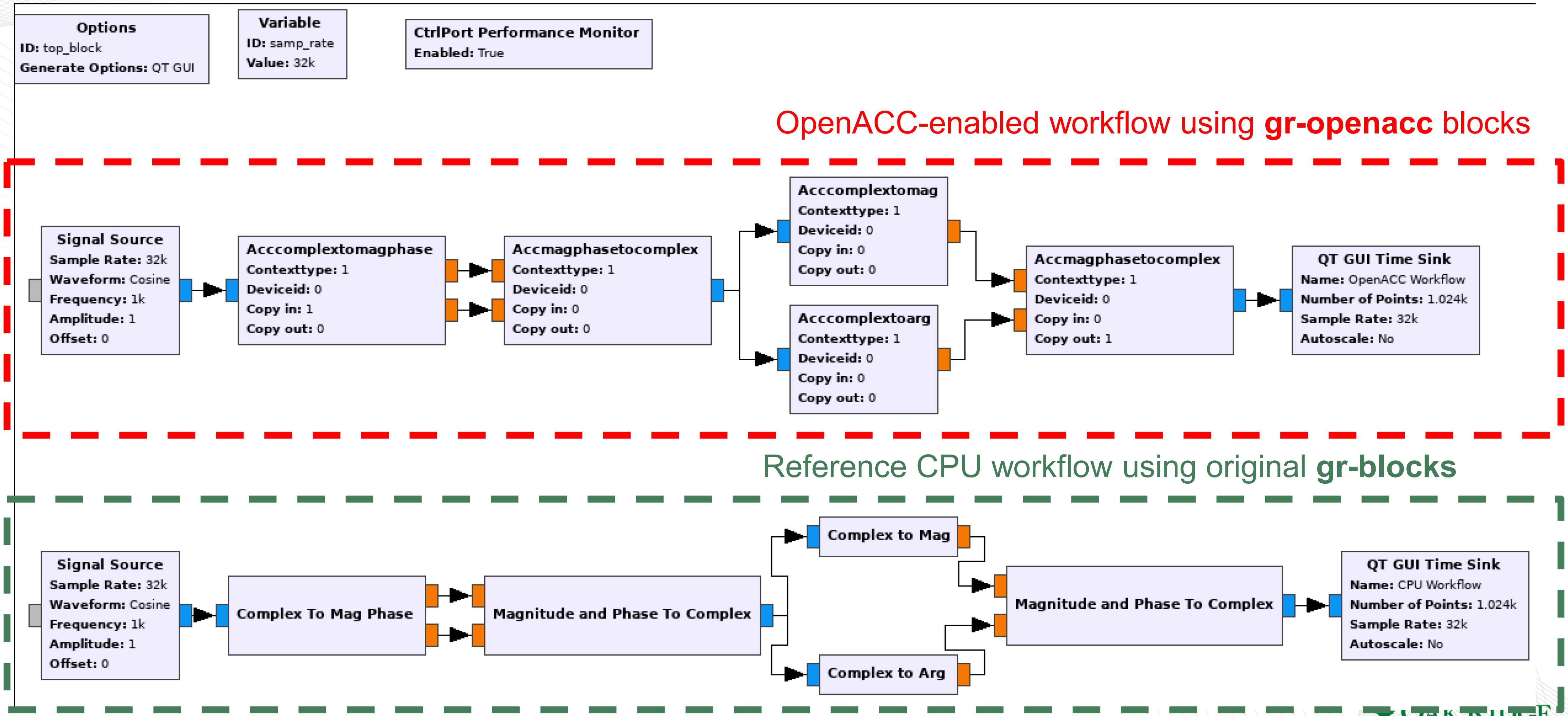
```

void accLog_init(acc_device_t deviceType, int devId, int threadID)
{

//////////////////////////////////
// CUDA Device Initialization //
//////////////////////////////////

fprintf(stderr, "Thread %d initializes the accLog block!\n", threadID);
std::string kernel_str[1];
kernel_str[0]="accLog_kernel_kernel0";
acc_init(deviceType, 1, kernel_str, "kernel", threadID);
return ;
}

void accLog_kernel(int noutput_items, float n_val, float k_val, const float * in, float * out, int threadID)
{
float * gpu__in;
float * gpu__out;
HI_set_context(threadID);
if (HI_get_device_address(in, ((void * *)( & gpu__in)), DEFAULT_QUEUE, threadID)!=HI_success)
{
printf("[ERROR] GPU memory for the host variable, in, does not exist. \n");
printf("Enclosing annotation: \n#pragma acc  kernels loop gang(((int)ceil((((float)noutput_items)/64.0F)))) worker(64) copyin(k_val, n_val, noutput_items) present(in[0:noutput_items], out[0:noutput_items]) private(i) \n");
exit(1);
}
if (HI_get_device_address(out, ((void * *)( & gpu__out)), DEFAULT_QUEUE, threadID)!=HI_success)
{
printf("[ERROR] GPU memory for the host variable, out, does not exist. \n");
printf("Enclosing annotation: \n#pragma acc  kernels loop gang(((int)ceil((((float)noutput_items)/64.0F)))) worker(64) copyin(k_val, n_val, noutput_items) present(in[0:noutput_items], out[0:noutput_items]) private(i) \n");
exit(1);
}
size_t dimGrid_accLog_kernel_kernel0[3];
dimGrid_accLog_kernel_kernel0[0]=((int)ceil((((float)noutput_items)/64.0F)));
dimGrid_accLog_kernel_kernel0[1]=1;
dimGrid_accLog_kernel_kernel0[2]=1;
size_t dimBlock_accLog_kernel_kernel0[3];
dimBlock_accLog_kernel_kernel0[0]=64;
dimBlock_accLog_kernel_kernel0[1]=1;
dimBlock_accLog_kernel_kernel0[2]=1;
gpuNumBlocks=((int)ceil((((float)noutput_items)/64.0F)));
gpuNumThreads=64;
totalGpuNumThreads=(((int)ceil((((float)noutput_items)/64.0F)))*64);
HI_register_kernel_numargs("accLog_kernel_kernel0",5,threadID);
HI_register_kernel_arg("accLog_kernel_kernel0",0,sizeof(void*),( & gpu__in),1,threadID);
HI_register_kernel_arg("accLog_kernel_kernel0",1,sizeof(void*),( & gpu__out),1,threadID);
HI_register_kernel_arg("accLog_kernel_kernel0",2,sizeof(void*),( & k_val),0,threadID);
HI_register_kernel_arg("accLog_kernel_kernel0",3,sizeof(void*),( & n_val),0,threadID);
HI_register_kernel_arg("accLog_kernel_kernel0",4,sizeof(void*),( & noutput_items),0,threadID);
HI_kernel_call("accLog_kernel_kernel0",dimGrid_accLog_kernel_kernel0,dimBlock_accLog_kernel_kernel0,DEFAULT_QUEUE,0,NULL,threadID);
HI_synchronize(1,threadID);
gpuNumBlocks=((int)ceil((((float)noutput_items)/64.0F)));
return ;
}
```

Output host code

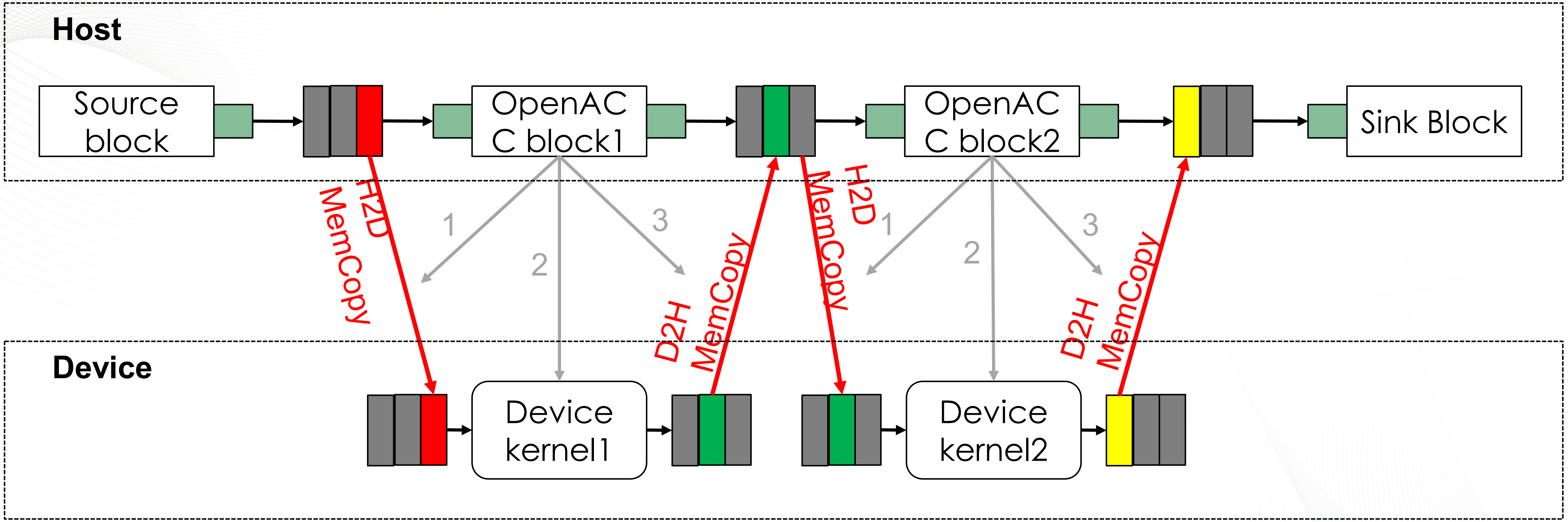
```
...
extern "C" __global__ void accLog_kernel_kernel0(float * in,
float * out, float k_val, float n_val, int noutput_items)
{
    int lwpriv__i;
    lwpriv__i=(threadIdx.x+(blockIdx.x*64));
    if (lwpriv__i<noutput_items)
    {
        out[lwpriv__i]=((n_val*log10(in[lwpriv__i]))+k_val);
    }
}

```

Output CUDA kernel code

Port an Example SDR Workflow to Xavier

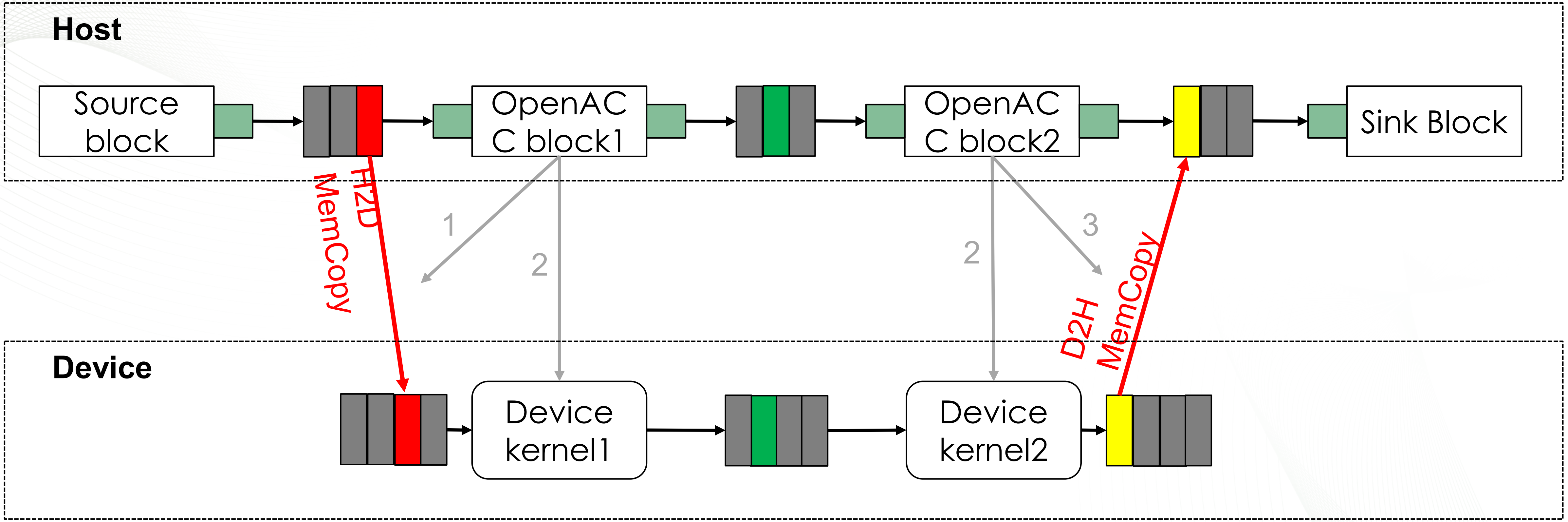


Basic Memory Management for OpenACC-Enabled GR Workflow



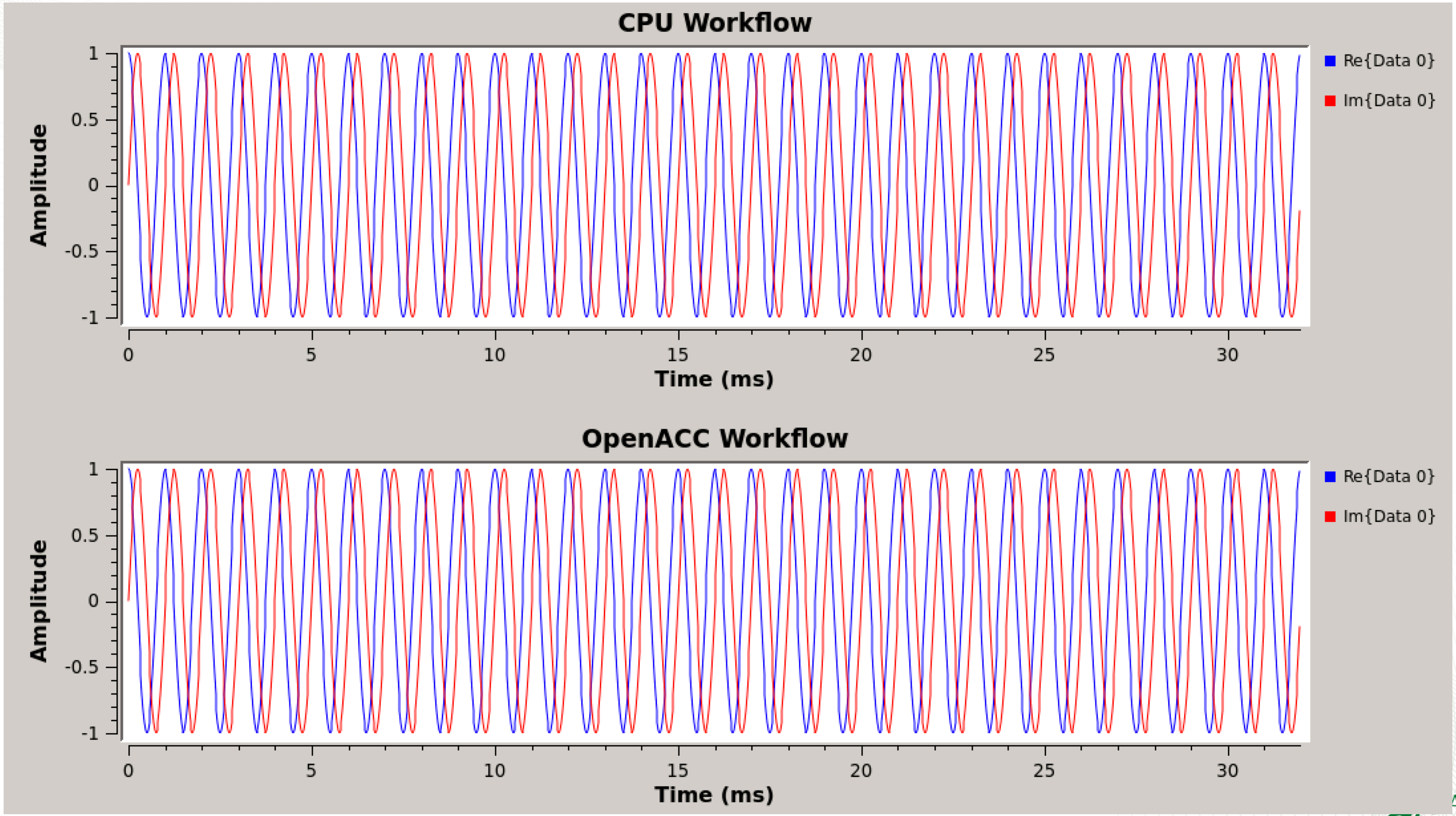
- In the basic memory management scheme, each invocation of an OpenACC GR block performs the following three tasks:
  1) Copy input data to device memory.
  2) Run a kernel on device.
  3) Copy output data back to host memory.

Optimized Memory Management for OpenACC-Enabled GR Workflow



- 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.

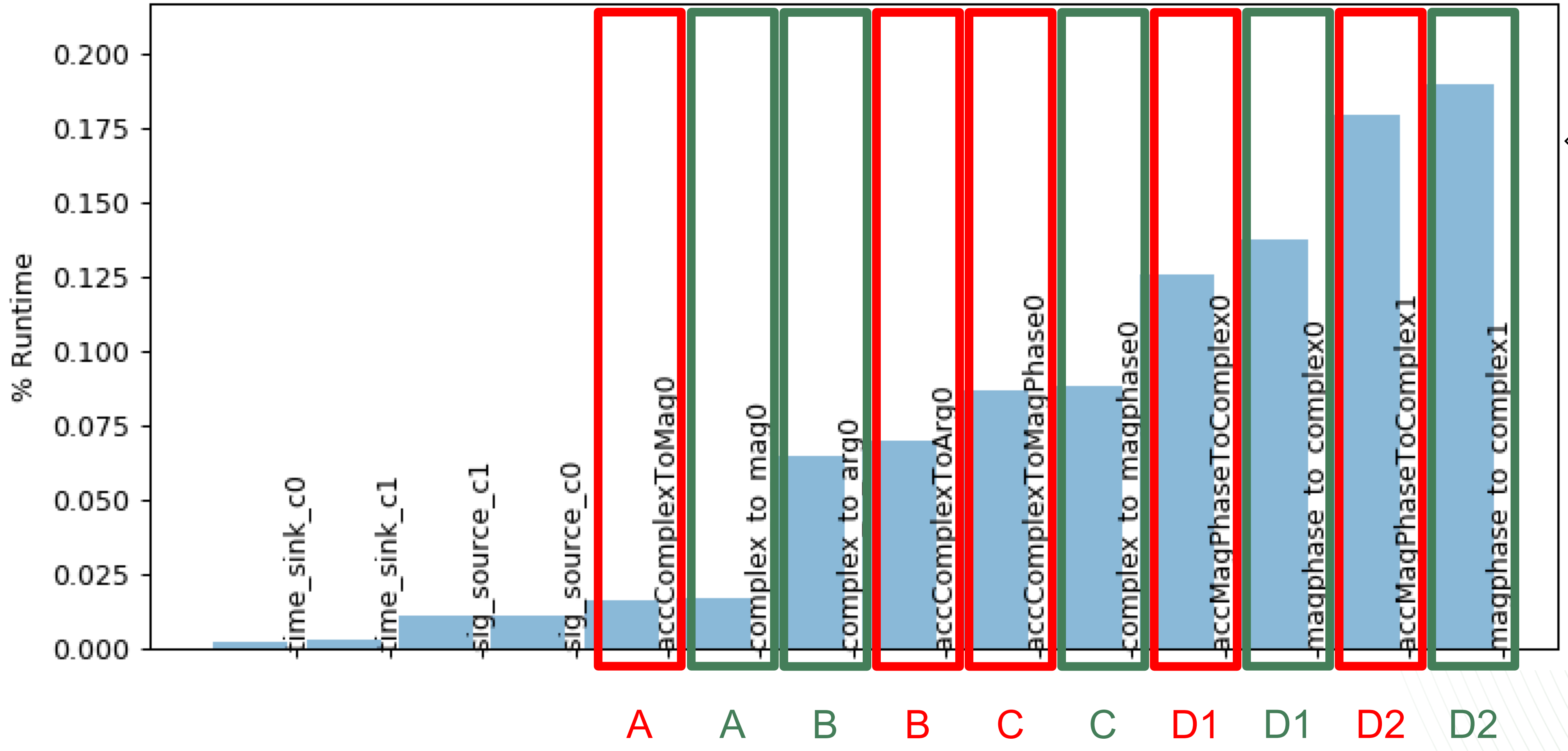
Sample Output of the Example SDR Workflow



SDR Workflow Profiling Using a Built-in GR Performance Monitoring Tool

- CPU versions of OpenACC blocks are algorithmically equivalent to those in the original GR blocks.

🟥 OpenACC Blocks on Xavier CPU 🟩 Original GR Blocks on Xavier CPU



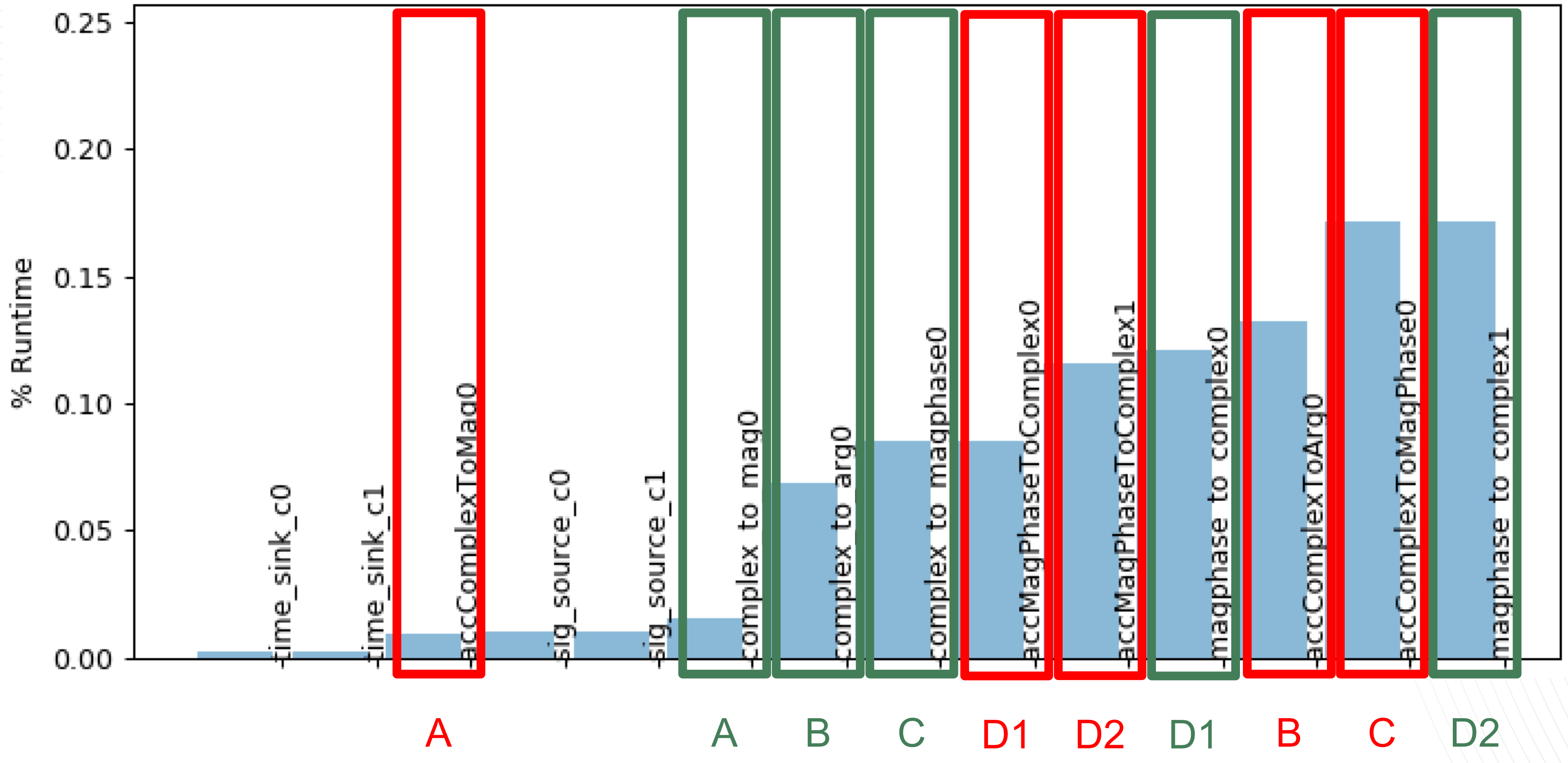
Some OpenACC blocks (B, D) use a simple register caching optimization, which causes them to perform better than the original GR blocks.

SDR Workflow Profiling Results When OpenACC Blocks Offloaded to CPU

- OpenACC blocks are automatically translated to OpenMP3 versions and run on Xavier CPU.

🟥 OpenACC Blocks on Xavier CPU via OpenMP

🟩 Original GR Blocks on Xavier CPU



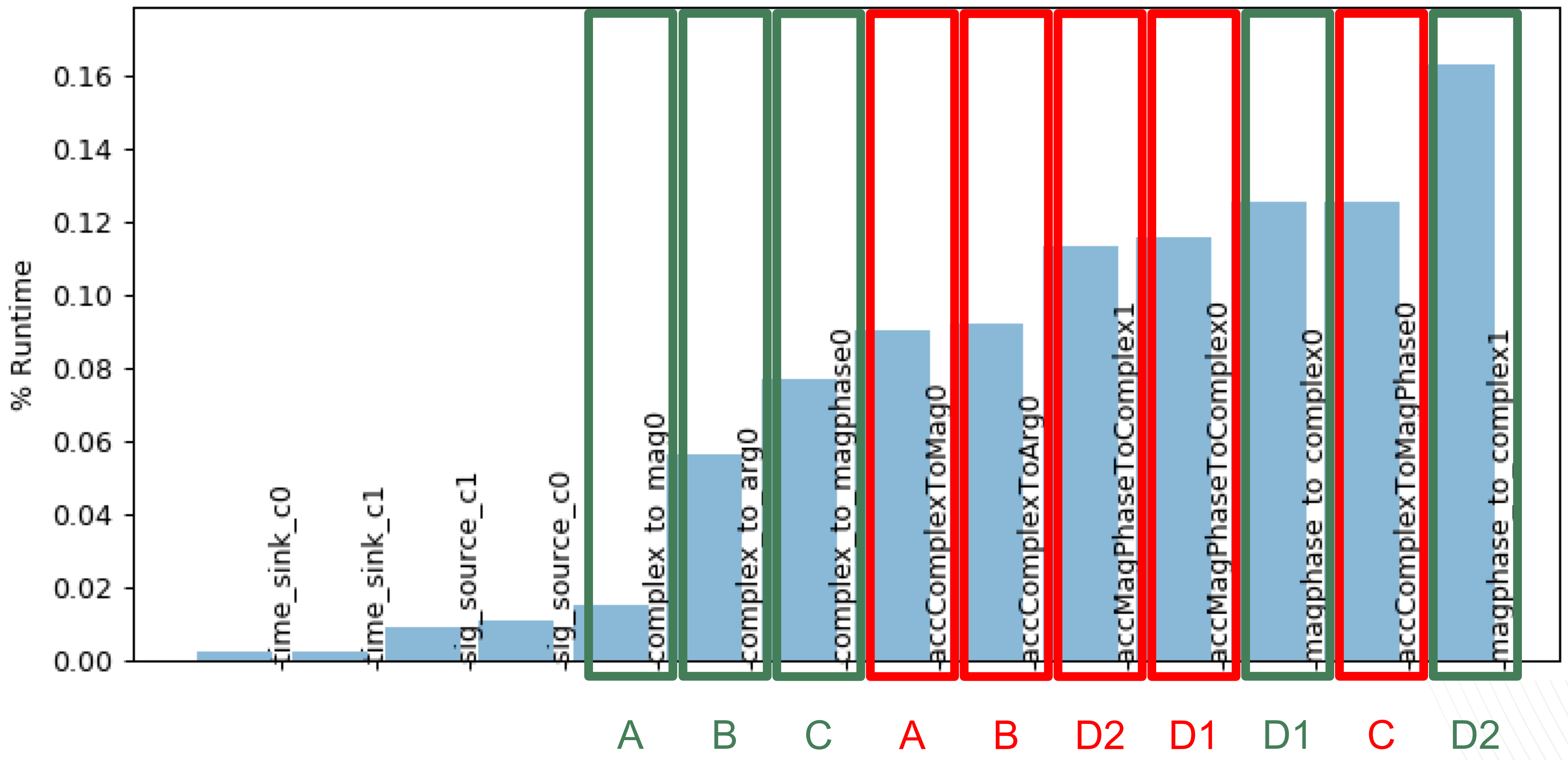
Some of original GR blocks (A, C) were already vectorized with Volk library.

Some of original GR blocks (B, C) performed better than OpenACC blocks (B, C).

SDR Workflow Profiling Results When OpenACC Blocks Offloaded to GPU

- 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

🟥 OpenACC Blocks on Xavier GPU 🟩 Original GR Blocks on Xavier CPU

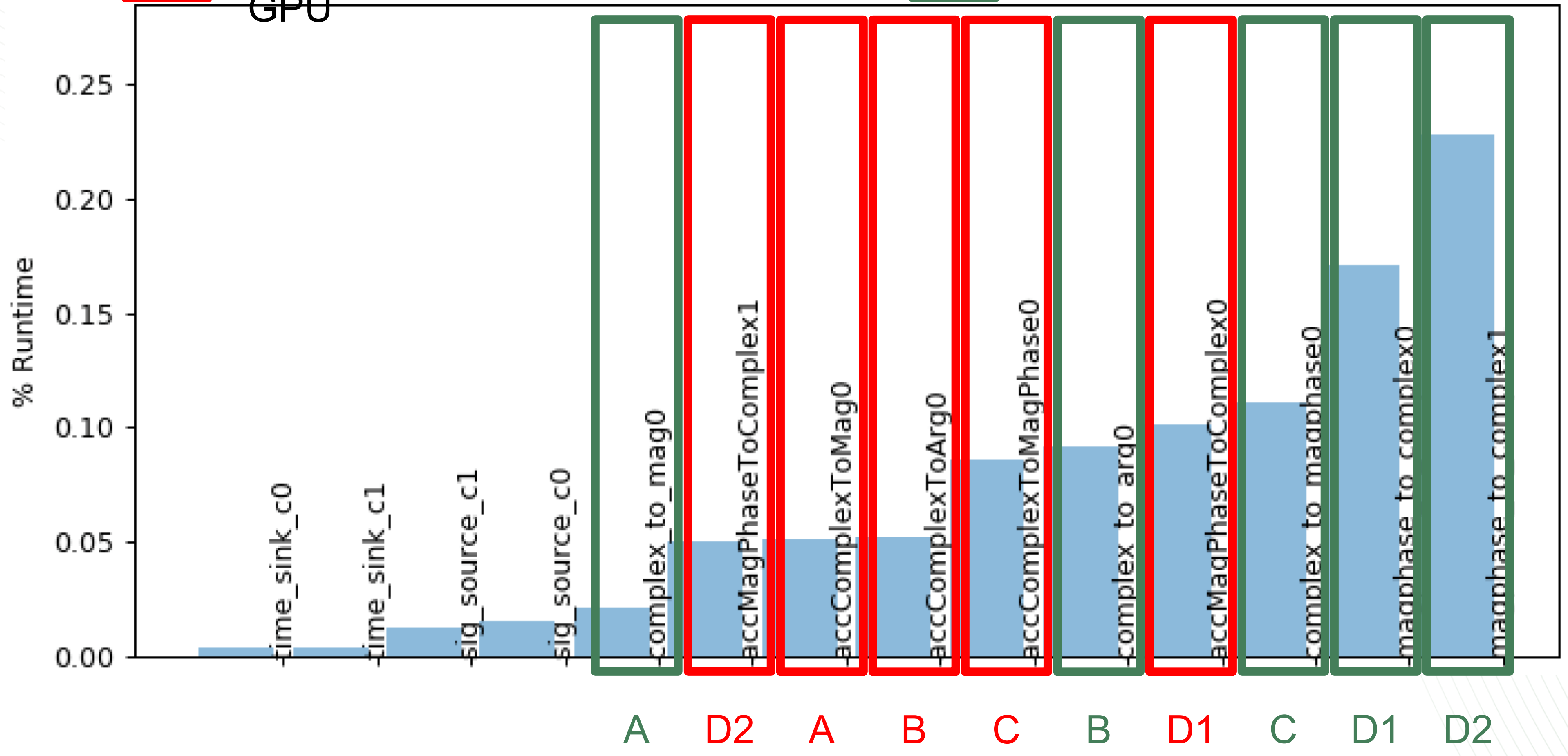


Due to extra memory transfer overheads, most OpenACC blocks perform worse than original GR blocks, except for the OpenACC block D1 and D2.

SDR Workflow 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.

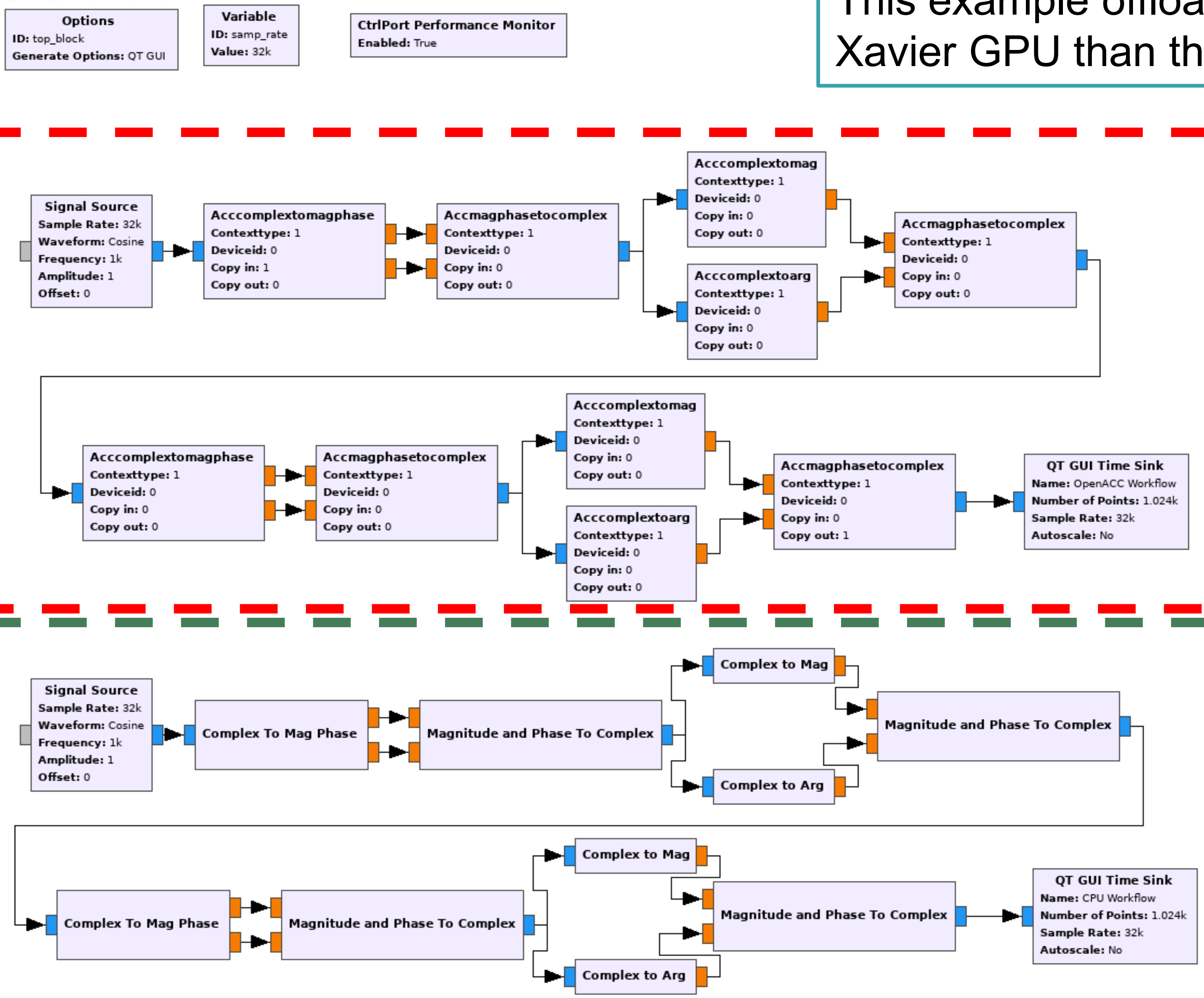
🟥 Opt. OpenACC Blocks on Xavier GPU 🟩 Original GR Blocks on Xavier CPU



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.

More Complex SDR Workflow Example

This example offloads more OpenACC blocks to Xavier GPU than the previous example.



Options
ID: top_block
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 32k

CtrlPort Performance Monitor
Enabled: True

OpenACC-enabled workflow using **gr-openacc** blocks

Signal Source
Sample Rate: 32k
Waveform: Cosine
Frequency: 1k
Amplitude: 1
Offset: 0

Acccomplextomagphase
Contexttype: 1
Deviceid: 0
Copy in: 1
Copy out: 0

Accmagphasetocomplex
Contexttype: 1
Deviceid: 0
Copy in: 0
Copy out: 0

Acccomplextomag
Contexttype: 1
Deviceid: 0
Copy in: 0
Copy out: 0

Acccomplextoarg
Contexttype: 1
Deviceid: 0
Copy in: 0
Copy out: 0

Accmagphasetocomplex
Contexttype: 1
Deviceid: 0
Copy in: 0
Copy out: 0

Acccomplextomagphase
Contexttype: 1
Deviceid: 0
Copy in: 0
Copy out: 0

Accmagphasetocomplex
Contexttype: 1
Deviceid: 0
Copy in: 0
Copy out: 0

Acccomplextomag
Contexttype: 1
Deviceid: 0
Copy in: 0
Copy out: 0

Acccomplextoarg
Contexttype: 1
Deviceid: 0
Copy in: 0
Copy out: 0

Accmagphasetocomplex
Contexttype: 1
Deviceid: 0
Copy in: 0
Copy out: 1

QT GUI Time Sink
Name: OpenACC Workflow
Number of Points: 1.024k
Sample Rate: 32k
Autoscale: No

Reference CPU workflow using original **gr-blocks**

Signal Source
Sample Rate: 32k
Waveform: Cosine
Frequency: 1k
Amplitude: 1
Offset: 0

Complex To Mag Phase

Magnitude and Phase To Complex

Complex to Mag

Complex to Arg

Magnitude and Phase To Complex

Complex To Mag Phase

Magnitude and Phase To Complex

Complex to Mag

Complex to Arg

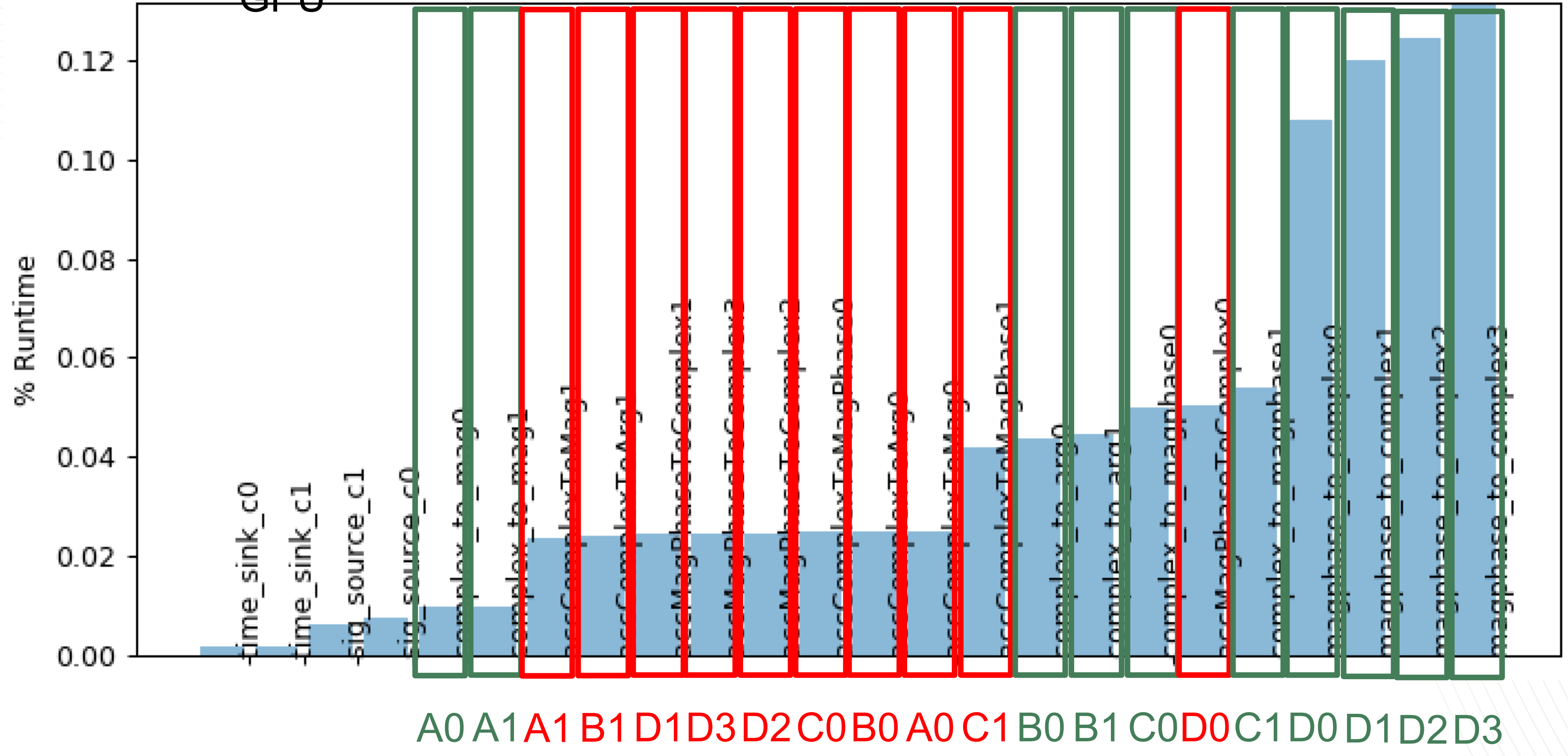
Magnitude and Phase To Complex

QT GUI Time Sink
Name: CPU Workflow
Number of Points: 1.024k
Sample Rate: 32k
Autoscale: No

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.

🟥 Opt. OpenACC Blocks on Xavier GPU 🟩 Original GR Blocks on Xavier CPU

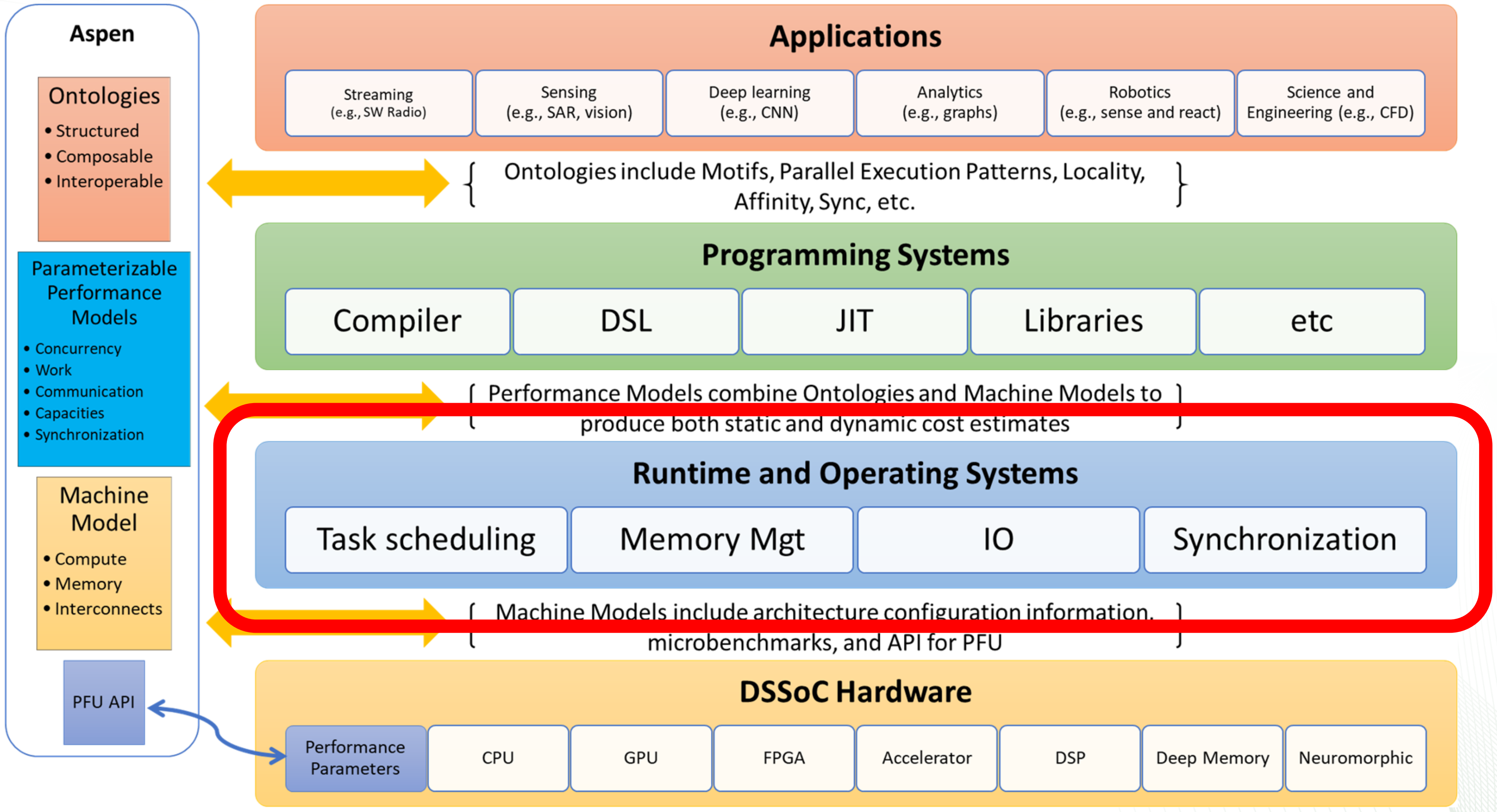


This example shows similar performance behaviors as the previous example.

Programming Systems Update Summary and Next Steps

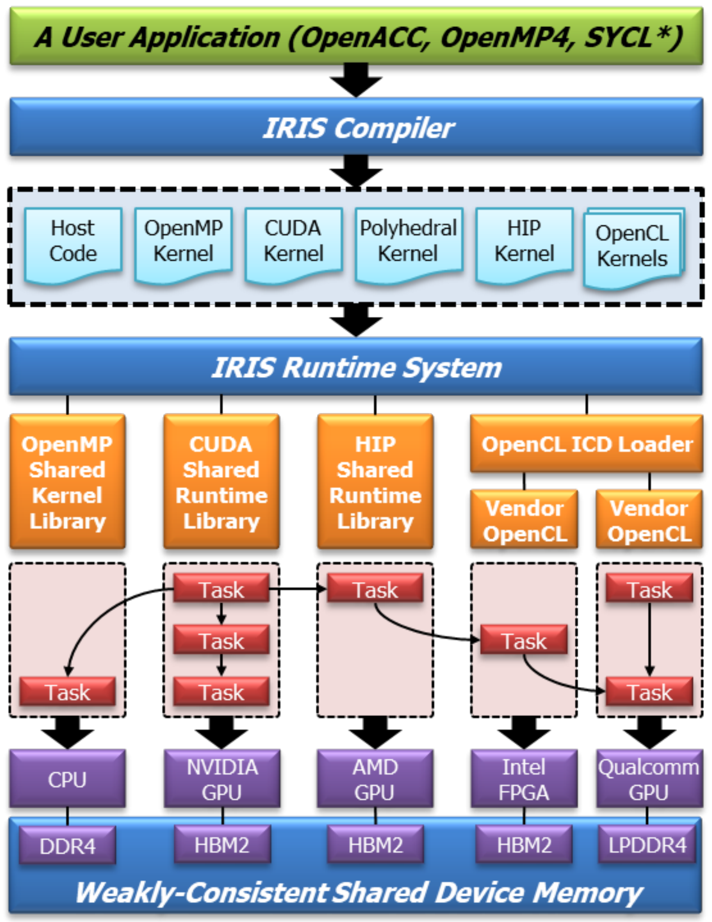
- Updated the programming system to use our new heterogeneous runtime system, called IRIS, as the common backend runtime.
    - IRIS allows intermixing of multiple different output programming models (e.g., OpenMP3, OpenMP4, OpenACC, CUDA, HIP, etc.) and runs them on heterogeneous devices concurrently.
- 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


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

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

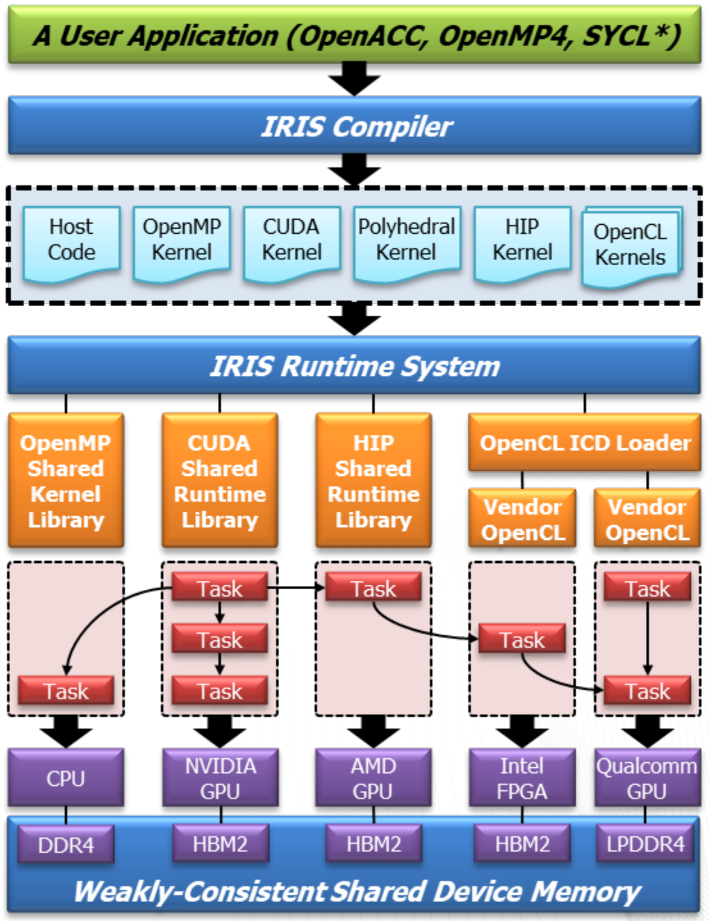


https://github.com/swiftcurrent2018

Oak Ridge National Laboratory

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)

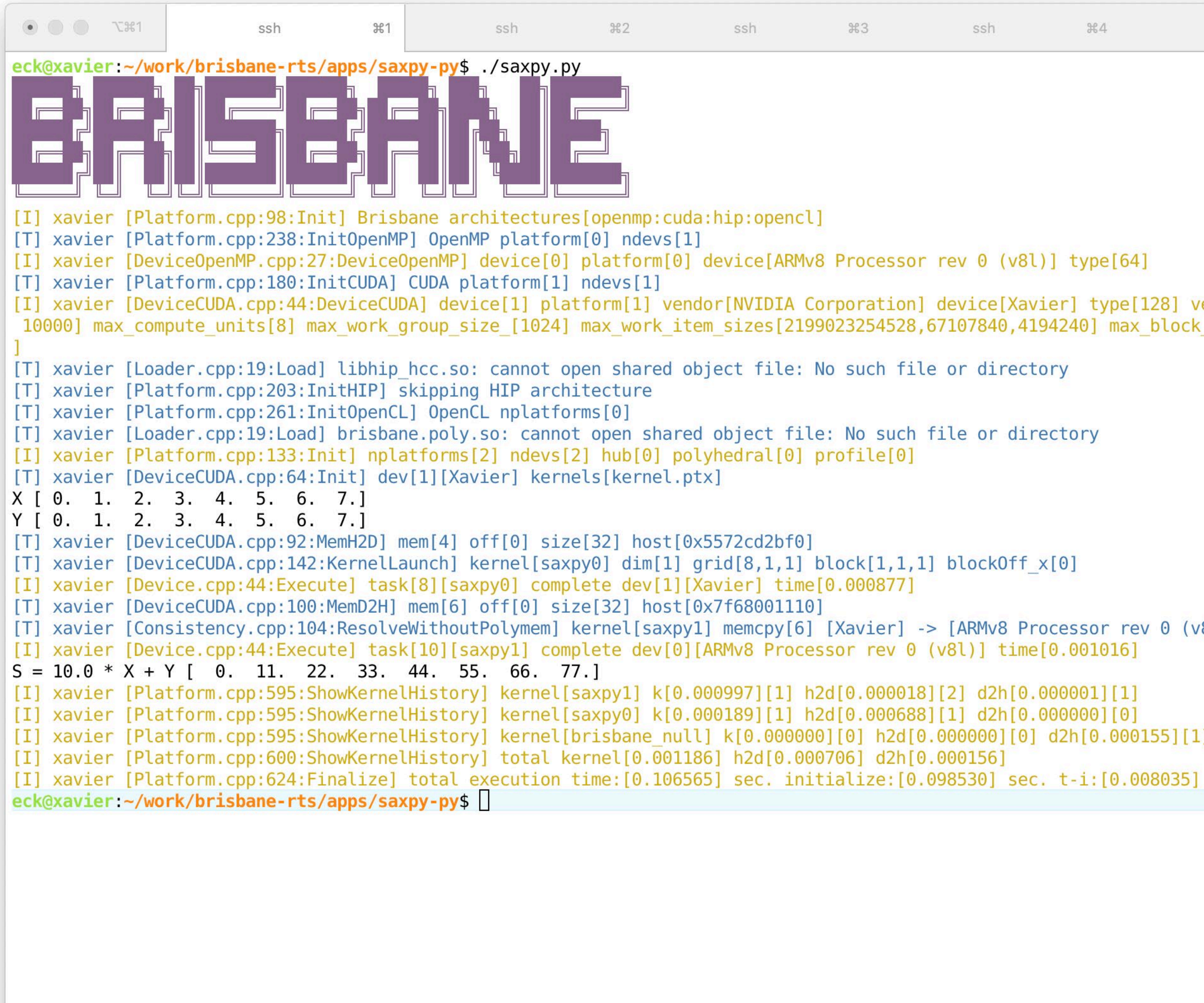


Supported Architectures and Programming Systems by IRIS

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

* ORNL Experimental Computing Laboratory (ExCL) https://excl.ornl.gov/

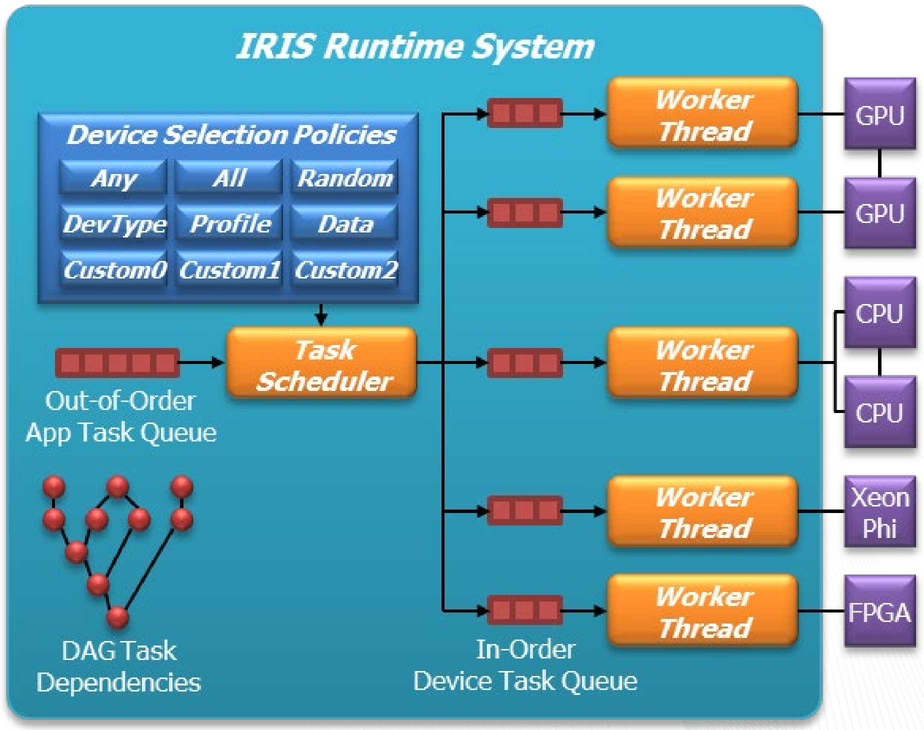
IRIS Booting on Various Platforms



```
eck@xavier:~/work/brisbane-rts/apps/saxpy-py$ ./saxpy.py
[I] xavier [Platform.cpp:98:Init] Brisbane architectures[openmp:cuda:hip:opencl]
[T] xavier [Platform.cpp:238:InitOpenMP] OpenMP platform[0] ndevs[1]
[I] xavier [DeviceOpenMP.cpp:27:DeviceOpenMP] device[0] platform[0] device[ARMv8 Processor rev 0 (v8l)] type[64]
[T] xavier [Platform.cpp:180:InitCUDA] CUDA platform[1] ndevs[1]
[I] xavier [DeviceCUDA.cpp:44:DeviceCUDA] device[1] platform[1] vendor[NVIDIA Corporation] device[Xavier] type[128] ve
 10000] max_compute_units[8] max_work_group_size_[1024] max_work_item_sizes[2199023254528,67107840,4194240] max_block_
]
[T] xavier [Loader.cpp:19:Load] libhip_hcc.so: cannot open shared object file: No such file or directory
[T] xavier [Platform.cpp:203:InitHIP] skipping HIP architecture
[T] xavier [Platform.cpp:261:InitOpenCL] OpenCL nplatforms[0]
[T] xavier [Loader.cpp:19:Load] brisbane.poly.so: cannot open shared object file: No such file or directory
[I] xavier [Platform.cpp:133:Init] nplatforms[2] ndevs[2] hub[0] polyhedral[0] profile[0]
[T] xavier [DeviceCUDA.cpp:64:Init] dev[1][Xavier] kernels[kernel.ptx]
X [ 0.  1.  2.  3.  4.  5.  6.  7.]
Y [ 0.  1.  2.  3.  4.  5.  6.  7.]
[T] xavier [DeviceCUDA.cpp:92:MemH2D] mem[4] off[0] size[32] host[0x5572cd2bf0]
[T] xavier [DeviceCUDA.cpp:142:KernelLaunch] kernel[saxpy0] dim[1] grid[8,1,1] block[1,1,1] blockOff_x[0]
[I] xavier [Device.cpp:44:Execute] task[8][saxpy0] complete dev[1][Xavier] time[0.000877]
[T] xavier [DeviceCUDA.cpp:100:MemD2H] mem[6] off[0] size[32] host[0x7f68001110]
[T] xavier [Consistency.cpp:104:ResolveWithoutPolymem] kernel[saxpy1] memcpy[6] [Xavier] -> [ARMv8 Processor rev 0 (v8
[I] xavier [Device.cpp:44:Execute] task[10][saxpy1] complete dev[0][ARMv8 Processor rev 0 (v8l)] time[0.001016]
S = 10.0 * X + Y [  0.  11.  22.  33.  44.  55.  66.  77.]
[I] xavier [Platform.cpp:595:ShowKernelHistory] kernel[saxpy1] k[0.000997][1] h2d[0.000018][2] d2h[0.000001][1]
[I] xavier [Platform.cpp:595:ShowKernelHistory] kernel[saxpy0] k[0.000189][1] h2d[0.000688][1] d2h[0.000000][0]
[I] xavier [Platform.cpp:595:ShowKernelHistory] kernel[brisbane_null] k[0.000000][0] h2d[0.000000][0] d2h[0.000155][1
[I] xavier [Platform.cpp:600:ShowKernelHistory] total kernel[0.001186] h2d[0.000706] d2h[0.000156]
[I] xavier [Platform.cpp:624:Finalize] total execution time:[0.106565] sec. initialize:[0.098530] sec. t-i:[0.008035]
eck@xavier:~/work/brisbane-rts/apps/saxpy-py$ 
```

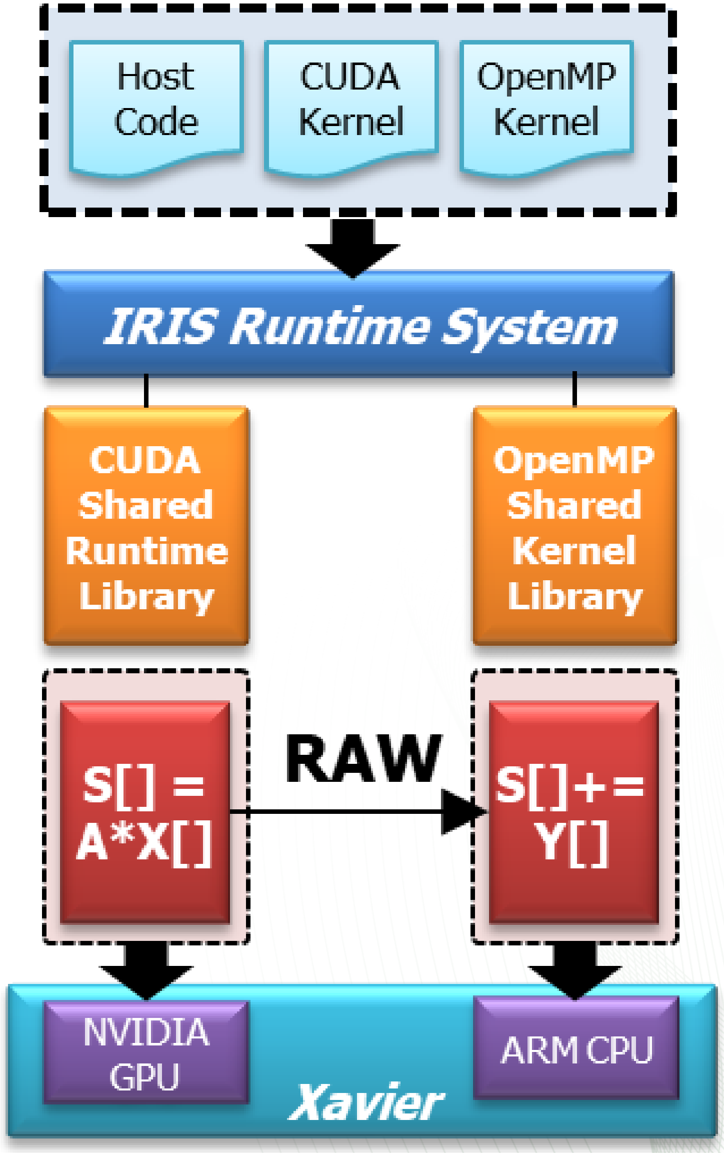
Task Scheduling in IRIS

- 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 * X[] + Y[]
- Two tasks
  - S[] = A * 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

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()
```

kernel.cu (CUDA)

```
extern "C" __global__ void saxpy0(float*
S, float A, float* X) {
  int id = blockIdx.x * blockDim.x +
threadIdx.x;
  S[id] = A * X[id];
}

extern "C" __global__ void saxpy1(float*
S, float* Y) {
  int id = blockIdx.x * blockDim.x +
threadIdx.x;
  S[id] += Y[id];
}
```

SAXPY: Python host code & OpenMP kernel code

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()
```

kernel.openmp.h (OpenMP)

```
#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
}
```

Memory Consistency Management

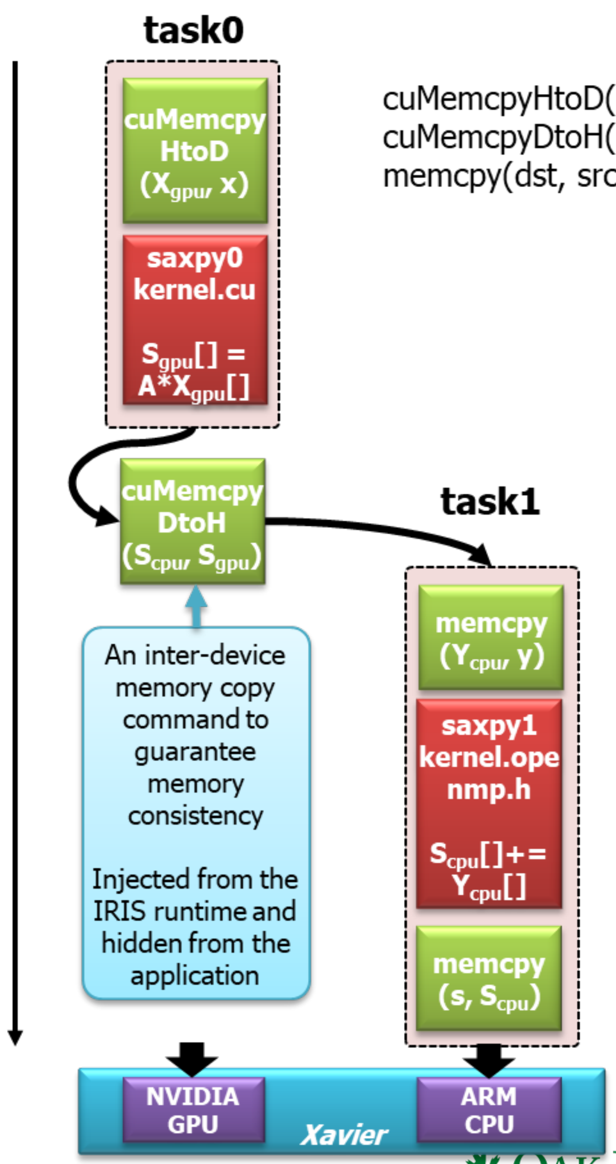
saxpy.py (1/2)

```
#!/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()
```

mem_s is shared between GPU and CPU



task0
cuMemcpy HtoD (X_{gpu}, x)
saxpy0 kernel.cu
$S_{gpu}[] = A*X_{gpu}[]$
cuMemcpy DtoH (S_{cpu}, S_{gpu})
An inter-device memory copy command to guarantee memory consistency
Injected from the IRIS runtime and hidden from the application
task1
memcpy (Y_{cpu}, y)
saxpy1 kernel.openmp.h
$S_{cpu}[] += Y_{cpu}[]$
memcpy (s, S_{cpu})
NVIDIA GPU
Xavier
ARM CPU

cuMemcpyHtoD(dst, src)
cuMemcpyDtoH(dst, src)
memcpy(dst, src)

OAK RIDGE National Laboratory

Locality-aware Device Selection Policy

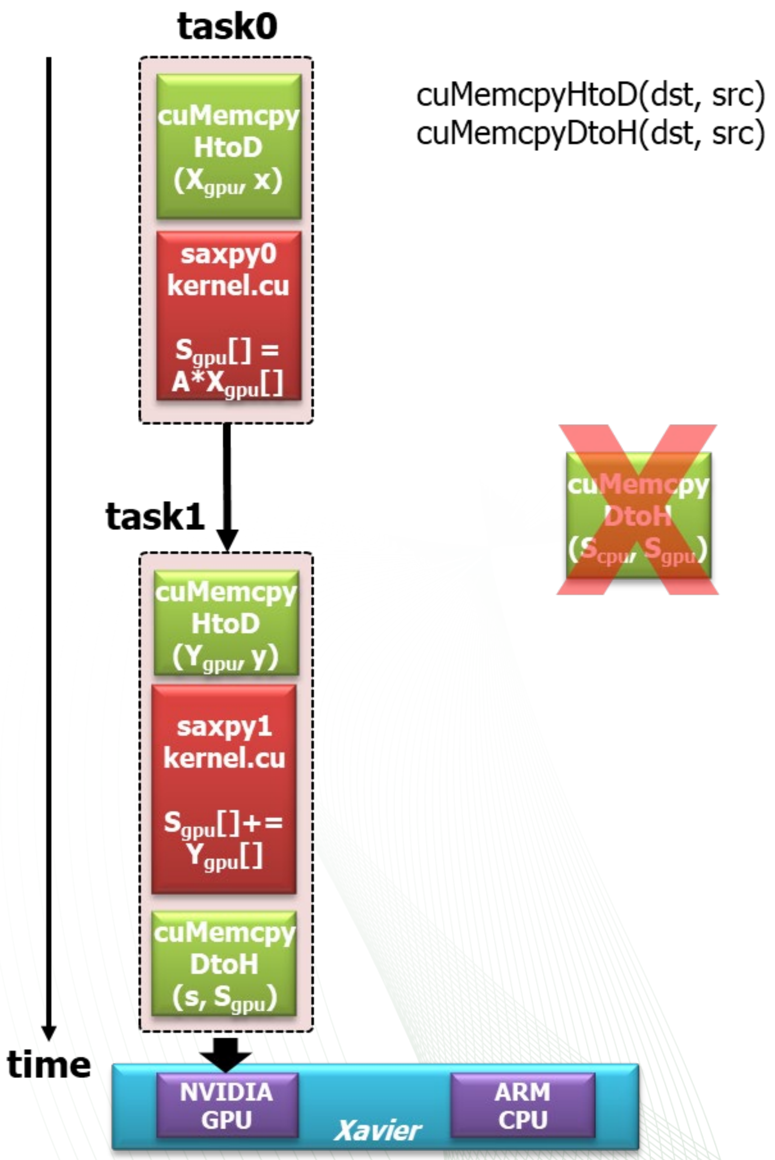
saxpy.py (1/2)

```
#!/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_data)

print 'S =', A, '* X + Y', s

iris.finalize()
```

iris_data selects the device that requires minimum data transfer to execute the task



task0
cuMemcpy HtoD (X_{gpu}, x)
saxpy0 kernel.cu
$S_{gpu}[] = A*X_{gpu}[]$
task1
cuMemcpy HtoD (Y_{gpu}, y)
saxpy1 kernel.cu
$S_{gpu}[] += Y_{gpu}[]$
cuMemcpy DtoH (s, S_{gpu})
~~cuMemcpy DtoH (S_{cpu}, S_{gpu})~~
time
NVIDIA GPU
Xavier
ARM CPU

cuMemcpyHtoD(dst, src)
cuMemcpyDtoH(dst, src)

IRIS: Task Scheduling Overhead – Running One Million (Empty) Tasks

ntasks.py

```
#!/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()
```

CPU or GPU randomly

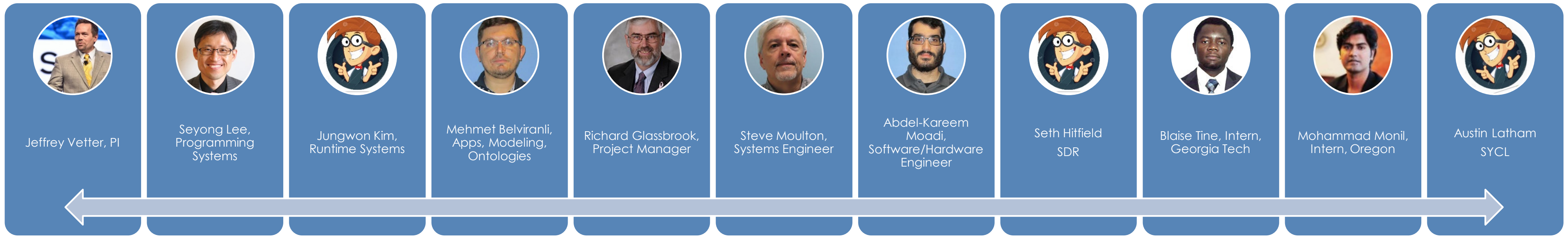
asynchronous task submission

concurrent tasks execution on multiple devices

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

Throughput	Latency
87,268 tasks/sec	11.4 µs/task

Closing


Jeffrey Vetter, PI
Seyong Lee, Programming Systems
Jungwon Kim, Runtime Systems
Mehmet Belviranli, Apps, Modeling, Ontologies
Richard Glassbrook, Project Manager
Steve Moulton, Systems Engineer
Abdel-Kareem Moadi, Software/Hardware Engineer
Seth Hitfield SDR
Blaise Tine, Intern, Georgia Tech
Mohammad Monil, Intern, Oregon
Austin Latham SYCL

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.

OAK RIDGE National Laboratory