IRIS: A Portable Runtime System Exploiting Multiple Heterogeneous Programming Systems

Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S. Vetter
Oak Ridge National Laboratory
Oak Ridge, TN, USA
{kimj, lees2, johnstonbe, vetter}@ornl.gov

***Abstract*—Across embedded, mobile, enterprise, and high performance computing systems, computer architectures are becoming more heterogeneous and complex. This complexity is causing a crisis in programming systems and performance portability. Several programming systems are working to address these challenges, but the increasing architectural diversity is forcing software stacks and applications to be specialized for each architecture. As we show, all of these approaches critically depend on their runtime system for discovery, execution, scheduling, and data orchestration. To address this challenge, we believe that a more agile and proactive runtime system is essential to increase performance portability and improve user productivity. To this end, we have designed and implemented IRIS: a portable runtime system exploiting multiple heterogeneous programming systems. IRIS can discover available resources, manage multiple diverse programming systems (e.g., CUDA, Hexagon, HIP, Level Zero, OpenCL, OpenMP) simultaneously in the same execution, respect data dependencies, orchestrate data movement proactively, and provide for user-configurable scheduling. Our evaluation on three architectures, ranging from Qualcomm Snapdragon to a Summit supercomputer node, shows that IRIS improves portability across a wide range of diverse heterogeneous architectures with negligible overhead.**

***Index Terms*—heterogeneous architectures, runtime systems, compilers**

I. Introduction

Experts expect the trend toward heterogeneous computing architectures and domain-specific computing to continue into the foreseeable future [1]–[3]. Years ago, this trend materialized in the mobile and embedded market, and it is now entering the enterprise, machine learning, high performance computing (HPC), and cloud computing markets. Contemporary architectures, such as NVIDIA Xavier, Qualcomm Snapdragon, and Xilinx Zynq, offer glimpses of future architectures in which several processors are systems-on-a-chip (SOCs) that contain multiple devices for accelerating applications, including a CPU, GPU, AI accelerator, and a vision or camera unit, among others. In HPC, 7 of the top 10 systems on the TOP500 list are heterogeneous, and this trend is expected to continue into the foreseeable future.

A fundamental challenge for this architectural diversity is that few, if any, programming systems span contemporary architectures while also providing a reasonable level of performance portability [4]. A plethora of programming models and system, such as CUDA [5], HIP [6], oneAPI [7], OpenCL [8], OpenACC [9], OpenMP [10], SYCL [11], and others, do exist, but their implementation and performance portability is inconsistently realized across architectures and implementations. Yet, a common feature in most of these heterogeneous programming models is that they must rely on a runtime system (RTS) for discovering available resources, managing multiple devices (e.g., CPU and GPU), resolving data dependencies, orchestrating data movement either explicitly or implicitly, and generating efficient work schedules on the available resources.

We expect that, as the community moves toward increasingly diverse heterogeneous systems [3], it is more likely that the RTS must bear more of the responsibility of application execution. Because many of the constraints and dependencies will not be known until execution time (e.g., size of a matrix, location of data in devices, resource availability), the RTS will be keenly positioned to dynamically balance these goals at execution time. Moreover, the scheduling challenge for heterogeneous architectures is considerably more difficult than for homogeneous architectures. To strive for performance portability, the heterogeneous RTS must map individual application kernels to heterogeneous devices efficiently, balancing concerns of several competing criteria: performance, data movement costs, and current load balance, among others. Moreover, these specialized cores often have idiosyncratic data movement semantics. Practically, the penalty for a poor decision when scheduling on a heterogeneous architecture can be a significant slowdown by an order of magnitude, which is much higher than a poor scheduling decision on homogeneous, shared-memory nodes.

Simply put, we propose new capabilities for heterogeneous RTSs to support performance portability, including dynamic resource discovery, online adaptive scheduling, proactive data movement, and support for simultaneous execution of multiple native programming models.

A. Contributions

To address these challenges, we have designed, implemented, and evaluated IRIS, a new heterogeneous RTS. In this paper, we contribute the following.

(1) We survey existing programming systems and identify the major challenges for performance portability of RTSs for diverse heterogeneous architectures.
(2) We describe IRIS, which provides several new key capabilities on heterogeneous architectures ranging from the SoCs to supercomputers, as outlined in Section II.
(3) We empirically evaluate IRIS on three different systems equipped with heterogeneous hardware (Qualcomm Snapdragon SoC, AMD CPU with AMD GPU, and POWER9 CPU with NVIDIA GPU), using microbenchmarks, kernels, and a proxy application to demonstrate IRIS's portability, productivity, and performance.

II. Motivation

Many RTSs have been created over years, and much of the related work focuses on functional portability and performance efficiency on CPU and possibly GPU architectures; however, in our review, these existing RTSs lack one or more of the critical capabilities for supporting portable yet diverse heterogeneous architectures. This section highlights these critical capabilities.

- **Dynamic Discovery and Portability**: Heterogeneous functionality will vary considerably across processors, and thus the RTS must dynamically catalog and manage these devices. At initialization, the RTS must discover and enumerate the heterogeneous capabilities on a node, including their preferred programming model and data transfer protocols. This makes the application portable across diverse heterogeneous architectures.
- **Online Adaptive Scheduling with Introspection**: Individual application kernels will perform differently across heterogeneous cores, so the RTS will need to adaptively schedule the mapping of these kernels to specific heterogeneous cores using a *policy*. The application developers need to select the specific policy or write their own custom policies for their target applications and systems.
- **Orchestrate Data Movement Proactively**: Data movement across heterogeneous system memory can dominate performance, so the RTS must initiate, monitor, and optimize data movement across memory. Ideally, the RTS could manage data movement without explicit user intervention or application commands.
- **Support Simultaneous Execution of Multiple Programming Models**: Existing programming models are fragmented across architectures, and, in most cases, specific programming models are more mature and optimized for their respective architectures. As processors become heterogeneous, so will the best programming models for each device on the processor. In this case, the RTS must blend these different programming models as seamlessly as possible.
- **Provide for Online Code Generation (Just-in-Time Compilation)**: Given the diversity of the heterogeneous cores available, the RTS may need to compile and optimize kernels for execution on the target heterogeneous cores. Furthermore, this capability can facilitate autotuning and optimization owing to the additional knowledge gained at execution time.

The survey of existing work (Section V) shows that no single system captures all of these concerns for contemporary heterogeneous architectures, though several RTSs provide a subset of these capabilities. IRIS is, to the authors' best knowledge, the first RTS to provide these capabilities across a wide range of diverse, contemporary heterogeneous architectures.

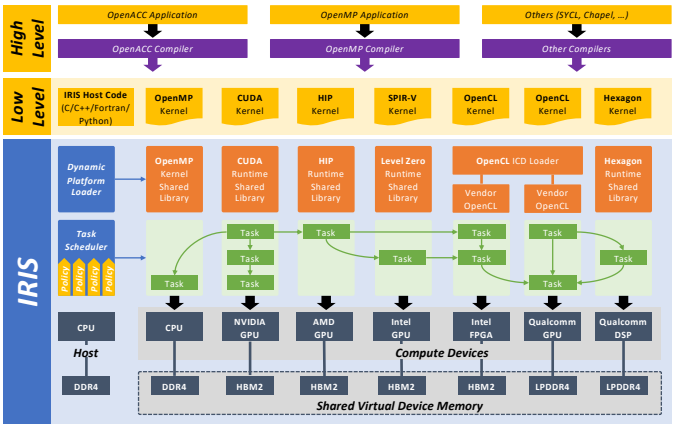

Fig. 1: The IRIS architecture.

III. IRIS Overview

IRIS is a new prototype RTS designed to incorporate the capabilities outlined in Section II. This section describes the core abstractions and prototype implementation of IRIS. Fig. 1 shows IRIS's major abstractions and implementation components.

A. IRIS Abstractions

IRIS presents a set of abstractions to the user: platform model, memory model, programming model, and execution model. These abstractions define their functionalities.

1) Platform Model: IRIS's platform model consists of a host connected to one or more compute devices. The host and all compute devices reside in a single-node system. The host node can be a multisocket, multicore CPU configuration. A compute device can be a GPU, Xeon Phi, field-programmable gate array (FPGA), DSP, or even a CPU. A compute device, except for the CPU, communicates with the host and other compute devices via a peripheral interconnect, such as PCI Express or NVLink.

2) Memory Model: IRIS's memory model describes the contents and behavior of the memory exposed by the IRIS platform. The memory in IRIS is divided into two parts: host memory and device memory. The host memory is directly available to the host. The device memory is directly available to kernels executing on its attached compute device. The physical device memory can be completely separate from the host memory for discrete compute devices (e.g., NVIDIA and AMD GPUs, Intel Xeon Phi KNCs, and FPGAs). On the other

hand, for integrated compute devices (e.g., CPUs, Intel Xeon Phi KNLs), the device and the host share the same physical memory and virtual address space.

To enable flexible task scheduling across multiple compute devices and make programming easier, the IRIS memory model presents the shared virtual device memory for all compute device memories with a relaxed memory consistency model, as shown in Fig. 1. All compute devices can share memory objects in the shared virtual device memory, and they can see the same content in the memory objects using the IRIS synchronization primitives.

3) Programming Model: The IRIS programming model is defined in terms of two distinct units: a host code that executes on the host and kernels that execute on compute devices. The IRIS RTS is a user-level library linked into the host code. To submit kernels to compute devices, the host code creates tasks. A task contains zero or more commands. There are three types of commands: (1) host-to-device memory copy command, (2) device-to-host memory copy command, and (3) kernel launch command. A task can have a hierarchical structure; that is, a task can contain multiple child tasks called *subtasks*. All last-level commands in a task hierarchy are executed in a single compute device in a first in, first out execution order. A task can have a dependency on other tasks. When a task depends on other tasks, it cannot start until the prerequisite tasks complete.

The host code must be portable across diverse programming systems. Thus, IRIS introduces a unified application programming interface (API) that abstracts away the differences of programming models from diverse programming systems and offers the C, C++, Fortran, and Python APIs. Kernels usually contain the most compute-intensive and time-consuming parts of an application, and the overall performance of the application depends on the kernels. Therefore, the kernels must be written and optimized for the target compute device. The IRIS RTS runs the target-specific kernels for the target compute device.

Writing multiple kernels for the same code on multiple programming systems hurts application portability and programmer productivity. Writing a single, low-level kernel across multiple programming systems is also not a good strategy because it exposes the low-level details of underlying hardware architecture to the programmer, which results in poor performance portability. To address this issue, IRIS provides a compiler that translates a source code written in a high-level and target-independent programming language to multiple, target-optimized kernels. Our prototype uses OpenACC or OpenMP with device offload as an input high-level programming model.

4) Execution Model: Fig. 2 shows an overview of the IRIS execution model. The application host code submits a task to the application task queue. This task submission includes information about the task, such as a hint, a target device parameter, the synchronization mode (blocking or non-blocking), and a policy selector that indicates on which resource the task should be executed. The application task queue is an out-of-order queue scheduled by the IRIS task scheduler. The task scheduler honors the dependencies among tasks, and

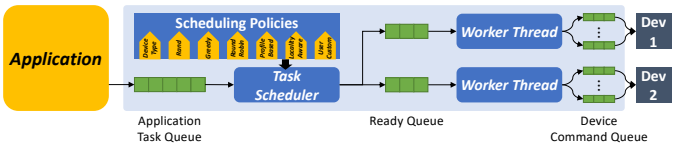

Fig. 2: The IRIS execution model.

synchronization is enforced by the host code.

When the task scheduler finds an executable task, in which all prerequisite tasks are complete, it dequeues the task from the application task queue and selects the target compute device using the device selection hint provided by the application. Once this selection is made, the task scheduler enqueues the task into the ready queue attached to the target compute device. A ready queue is an in-order queue managed by a worker thread. Each compute device has a worker thread. A worker thread dequeues a task from its attached ready queue and submits all commands in the task to the device command queues on its attached compute device. A device can have one or more device command queues (e.g., CUDA stream, HIP stream, OpenCL command queue) to enable simultaneous execution of multiple kernels. When the compute device completes the task, the worker thread notifies the task scheduler that the task is complete. At this point, the task scheduler resolves the dependencies among the tasks in the application task queue and searches for newly available executable tasks. To reduce CPU load, the runtime puts the task scheduler and work threads in a *sleep* mode while the devices are computing and no work is required by the task scheduler or work threads. The task scheduler wakes up only for two conditions: (1) when the host code submits a task or (2) when a worker thread notifies it of a task completion. A worker thread wakes up only when the scheduler enqueues a task into its attached ready queue.

B. IRIS Compiler and Code Generation

Kernels executed by IRIS can be hand written, compiler generated, or translated from a domain-specific language, among other methods. The IRIS compiler is a source-to-source translator built on top of OpenARC [12]. The compiler translates the compute-intensive codes augmented by the OpenMP/OpenACC directive in the input source codes into device kernel codes written in CUDA, Hexagon, HIP, OpenCL, OpenMP, and the IRIS polyhedral model. It also generates the host code using the IRIS runtime API, which orchestrates the execution of the kernels on devices from the input source code. This paper mainly focuses on the RTS, and the details of the code generation are beyond the scope of this writing.

C. Shared Virtual Device Memory

To achieve easy programming and flexible task scheduling with effective data orchestration, IRIS provides programmers with weakly consistent, shared virtual memory across multiple, disjoint device memories. In the IRIS execution and memory models, different compute device can access the same memory objects. If the compute devices only read a memory object,

they can have their own local copy of the memory object in their device memories. Otherwise, if at least one device writes the memory object, the updated contents in the memory object must be shared with other compute devices at synchronization points (e.g., task dependency edges; task, graph, device, and platform synchronization function calls).

To manage memory consistency across multiple devices, IRIS maintains an owner list for each memory object. Each entry in the list contains a linear memory region and its owner device, which has the latest local copy of the memory region on its device memory. When a worker thread executes a host-to-device memory copy command, it removes all entries of the target memory object from the owner list, and it adds a new entry with the memory region and relevant attached device to the list. When the worker thread executes a kernel launch command, it checks whether its attached device has the latest copy of the accessed memory objects in the kernel. That is, it references the owner lists in the memory object. If the attached device does not have the latest copy of the memory, it copies the accessed memory region from one of the owners to the attached device. If the memory region in the kernel is read only, the worker thread adds a new entry in the owner list. Otherwise, the region is written in the kernel, and the worker thread clears the list and adds a new entry. When the worker thread executes a device-to-host memory copy command, it copies the latest version of its device memory to the host. If the device does not have the latest copy, the worker thread copies the device memory from an owner in the same manner it would for the kernel launch commands.

D. Configurable Device Selection Policies

The host code submits a kernel to a compute device by submitting a task that contains the kernel launch command to the application task queue in the IRIS RTS. In the OpenMP specification, a device clause in a target directive specifies the non-negative device number of the target device that executes the task. IRIS extends the device clause to the device selection policy. IRIS presents seven built-in device selection policies: (1) device number, (2) device type, (3) random, (4) greedy, (5) round robin, (6) profile-based, and (7) locality-aware, as shown in Fig. 2. Users can also create their own custom policies and plug them into IRIS.

E. Dynamic Platform Loader

The dynamic platform loader is the core mechanism that enables diverse programming systems from different vendor devices to coexist on a single programming environment. When an IRIS application calls the IRIS initialization function, the platform loader loads all available shared runtime libraries in the underlying system and places them on the private IRIS linkchains. The platform loader surveys all available compute devices, loads their native kernels, and prepares them to run the IRIS tasks by calling the native runtime API functions from the shared libraries in the linkchains. An IRIS application executable is linked with the IRIS library only. It does not have any direct dependencies with the underlying native programming systems. This makes the application portable across diverse, heterogeneous architectures at the source code and executable levels.

TABLE I: Evaluation systems.

System	Snapdragon 855	AMD	Summit
CPUs	Qualcomm Kryo 485	2 × AMD EPYC 7702	2 × IBM Power9
Cores/CPU	8	64	22
Host Mem	6 GB LPDDR4 (shared)	512 GB DDR4	512 GB DDR4
Devices	Qualcomm Adreno 640 GPU Qualcomm Hexagon 690 DSP	2 × AMD MI60 GPU	6 × NVIDIA V100 GPU
Device Mems	6 GB LPDDR4 (shared)	2 × 32 GB HBM2	6 × 16 GB HBM2
Runtime(s)	Qualcomm OpenCL 2.0 Qualcomm Hexagon SDK 3.5.2	AMD HIP 4.1	NVIDIA CUDA 10.1
Compiler	Android NDK r20	GCC 9.1	IBM XL C/C++ 16.1
OS	Android 9	CentOS 7.8	RHEL 7.6

F. Task Partitioning

In IRIS, multiple compute devices can cooperate to execute a task using the task partitioning mechanism. To achieve this, a programmer can manually partition a task containing a large index space into multiple subtasks and offload them to multiple compute devices using their memory access information. The task partitioning can also be done automatically with the IRIS runtime and polyhedral compiler. If the kernel access memory objects in affine forms, the runtime partitions the loop index using the memory access regions information from the polyhedral kernel code.

IV. Evaluation

As described in Section II, we designed IRIS to provide several new capabilities to achieve a new level of portability across a diverse range of heterogeneous architectures. Our evaluation focuses first on the basic properties of the RTS, and then on two kernels and a proxy application using three heterogeneous systems. Table I summarizes the systems targeted for evaluation.

A. Microbenchmarks

We measured the kernel launch overhead of IRIS on three evaluation systems (Fig. 3). First, for a baseline, we ran empty kernels and measured the kernel launch overhead using the underlying native programming platforms without IRIS (i.e., OpenCL for GPU and Hexagon for DSP in Snapdragon, HIP for AMD, and CUDA for Summit). Then we measured the kernel launch overhead with IRIS. As shown in Fig. 3, IRIS introduces additional runtime overhead as a trade-off for portability and flexible scheduling. The overhead includes task creation, command creation, task scheduling, communication and synchronization across the host code, task scheduler, and worker threads. For both cases (with and without IRIS), the average overhead per kernel launch decreased as the kernel launch count increased, except for native Hexagon on

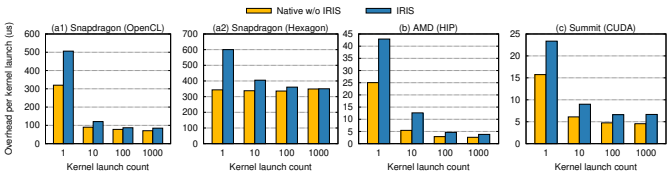

Fig. 3: Kernel launch overhead.

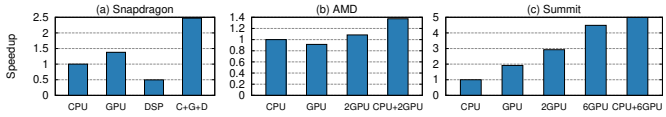

Fig. 4: SAXPY.

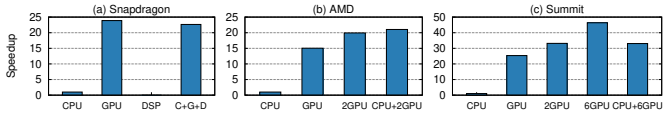

Fig. 5: SGEMM.

Snapdragon (Fig. 3[a2]). This is because the kernel launches in OpenCL, HIP, CUDA, and IRIS are asynchronous with the application, but kernel launch in Hexagon is a synchronous operation. As shown in Fig. 3, the additional runtime overhead from IRIS is negligible.

B. Kernels

We used two kernels for the evaluation, SAXPY and SGEMM. SAXPY is a memory-intensive kernel with few computations performed per memory operation. On the other hand, SGEMM is a compute-intensive kernel. Because the Qualcomm OpenCL for Adreno GPU does not support double-precision OpenCL kernels, we used only single-precision operations to show the performance variation of the same kernels across different architectures.

Fig. 4 shows the speedup over a CPU (i.e., all CPU cores in a system) of SAXPY on three different evaluation systems. The array size was set to 64 M for Snapdragon and 256 M for AMD GPUs and Summit (NVIDIA) GPUs. We ran OpenMP SAXPY kernels on all CPU devices in these systems. OpenCL and Hexagon kernels were used for the Snapdragon GPU and DSP, the HIP kernel was used for the AMD GPUs, and the CUDA kernel was used for the Summit GPUs.

For Snapdragon, the GPU achieved a 1.4× speedup, and the DSP achieved a 0.5× speedup over the CPU-only implementation (Fig. 4[a]). All CPUs, GPUs, and DSPs share the same physical device memory on Snapdragon. Therefore, memory copy overheads are the same. The performance differences come from the kernel execution time. We also ran SAXPY using CPUs, GPUs, and DSPs simultaneously. We distributed the workload among them based on their throughputs using a task partitioning technique (Section III-F), and it achieves a nearly 2.5× speedup.

The AMD GPUs achieved a 0.96× speedup over the CPU-only implementation. Unlike Snapdragon, CPUs and GPUs in the AMD machine have their own discrete device memories. Therefore, the memory copy overhead in CPU-GPU is higher than CPU-CPU. Furthermore, the OpenMP CPU kernel in SAXPY can exploit a total of 128 CPU cores in the machine. This high memory copy overhead on the GPU, coupled with the good kernel execution performance on the CPU, resulted in poor performance on the GPU compared to the CPU. We also ran SAXPY using 2 GPUs and all devices (CPU + 2 GPUs). They achieved a speedup of 1.1× to 1.4× over the CPU.

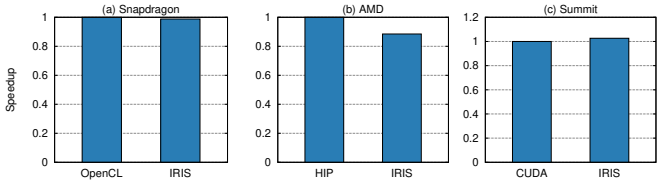

Fig. 6: LULESH.

As shown in Fig. 4(c), the GPUs achieved a 2× speedup over the CPU kernel on Summit. Summit's NVLink2 interconnect improves CPU-GPU communication time, which results in high performance for the GPU implementation. We vary the number of GPU devices up to 6, and it shows good scalability—up to a 4.5× speedup. When SAXPY uses all devices in the system, it achieves a 5× speedup.

Fig. 5 shows the speedup over a CPU device using the SGEMM kernel on three different evaluation systems. We used 2048×2048 matrices for Snapdragon and 6144×6144 matrices for AMD and Summit. Unlike SAXPY, SGEMM is a very compute-intensive kernel, and the kernel execution time dominates its total execution time. For all three systems, GPUs show much higher performance than the CPU because of the massively parallel kernel execution on the GPU and high strided memory access overhead in the CPU and DSP. On Snapdragon and Summit, SGEMM shows performance degradation when using all devices, owing to its additional synchronization overhead. However, the single CPU + 2 GPUs AMD configuration shows performance improvements compared to only 2 GPUs, owing to the relatively high performance of the AMD CPU.

C. Proxy Application

Our final evaluation used a proxy application called Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) [13]; this code is far more complex than any individual kernel and implements a simplified version of an unstructured Lagrangian explicit shock hydrodynamics application. LULESH is a significant proxy application composed of 8,000 lines of code with straightforward problem size scaling and architecture scalability. We automatically generated the IRIS version of LULESH from the OpenACC version using the IRIS compiler (see high-level programming in Fig. 1). Our implementation of LULESH spawns 1.6 million kernel tasks with a problem size of 100^3 per domain, which is a large number of tasks for any task-based system. For comparison, we also created native GPU versions for each platform (e.g., OpenCL for Snapdragon, HIP for AMD, and CUDA for Summit), which uses the device-specific backend runtime directly instead of using IRIS but has nearly identical execution patterns as the IRIS version.

Fig. 6 shows the performance of LULESH's IRIS version running on multiple platforms without any change to the source code; its performance is normalized to the native GPU versions of each tested platform. The results indicate that IRIS incurs acceptable overhead on all tested systems

when compared to the native GPU versions. Interestingly, the performance of IRIS is slightly better than the native CUDA version on Summit GPUs, which is due to the task merging optimization implemented in the IRIS compiler and runtime. The task merging optimization merges multiple kernel launch calls and memory transfer calls into a single task. This reduces the number of IRIS task submissions and reduces the runtime task launch and scheduling overhead. The task merging optimization is applied on all tested platforms, but the results show that the benefit of the optimization outweighs the IRIS runtime overhead on Summit only.

V. Related Work

In Table II, we compare related work along twelve criteria: (1) support for **CPU**s, (2) support for **GPU**s, (3) support for **FPGA**s, (4) support for **DSP**s, (5) can manage **multiple devices** simultaneously (Section III-A1), (6) can manage multiple devices from **multiple vendors** simultaneously (Section III-A1), (7) can manage **multiple programming systems** on multiple devices simultaneously (Section III-A3), (8) can manage **shared** device memory among multiple devices and programming systems (Sections III-C and III-A2), (9) can perform workload **partitioning** (Section III-F), (10) can provide **tasking** with dependencies (Section III-A3), (11) can facilitate configurable **scheduling policies** (Section III-D), and (12) can support execution of **distributed** memory nodes.

Because a comprehensive review of this material is not possible due to space constraints, we highlight the research that is most closely related to our work. Many successful task-based RTSs had design goals (e.g., supporting distributed memory) or implementation targets (e.g., only GPUs) that were different from ours. For example, as shown in Table II, most systems support a CPU and possibly a GPU—often only NVIDIA GPUs—but few of them support FPGAs or various components of SoCs. Even fewer of these RTSs support simultaneous execution across multiple heterogeneous devices. Most of these RTSs do not support simultaneous execution of multiple programming systems on these heterogeneous devices. A review of Table II shows that OmpSs [14] and StarPU [15] are the two RTSs most closely related to IRIS.

Unlike IRIS, which provides a unified host code API, with StarPU, a programmer must write different versions of host code to launch the CUDA and/or OpenCL kernels using CUDA and/or OpenCL runtime APIs. In OmpSs and StarPU, the programmers must explicitly specify the available programming systems (CUDA, OpenCL) in the target system when their runtime libraries are built. On the other hand, the IRIS RTS automatically finds the available programming systems in the target system, dynamically loads their respective shared libraries, and indirectly calls their API functions. Also, these other runtimes do not support automatic workload partitioning.

TABLE II: Summary of related work. (CPU = {Intel, AMD, IBM, ARM}; GPU = {NVIDIA, AMD, Intel, ARM, Qualcomm}; FPGA = {Intel, Xilinx, o=other}; DSP = {Qualcomm})

Name	CPU	GPU	FPGA	DSP	Multiple Devices	Multiple Vendors	Multiple Prog Sys	Shared	Partitioning	Tasking	Scheduling Policy	Distributed
IRIS	✓✓✓✓	✓✓✓✓✓	✓✓	✓	✓	✓	✓	✓	✓	✓	✓	
OmpSs [14]	✓✓✓✓	✓✓ ✓✓	✓✓		✓	✓	✓	✓		✓		
StarPU [15]	✓✓✓✓	✓✓ ✓✓	✓✓		✓	✓	✓	✓		✓	✓	
AMGE [16]		✓			✓			✓	✓			
Argobots [17]	✓✓✓✓					✓				✓	✓	
COSP [18]		✓			✓			✓	✓			
CUDA [5]		✓			✓			✓				
Charm++ [19]	✓✓✓✓	✓			✓			✓		✓		✓
Cilk [20]	✓✓✓✓				✓					✓		
Dandelion [21]	✓	✓	✓		✓	✓		✓		✓		✓
GeMTC [22]	✓	✓								✓		
HIP [6]		✓✓			✓							
HPVM [23]	✓✓✓✓	✓✓ ✓✓								✓	✓	
HPX [24]	✓	✓			✓	✓		✓				
Legion [25]	✓	✓			✓	✓		✓	✓	✓		✓
MCL [26]	✓✓✓✓	✓✓ ✓✓	✓✓		✓	✓				✓	✓	
Maestro [27]		✓✓			✓	✓		✓				
MultiGPU [28]		✓			✓			✓	✓			
OCR [29]	✓✓				✓					✓		✓
OpenACC [9]	✓✓✓✓	✓✓			✓							
OpenCL [30]	✓	✓✓✓✓✓	✓✓		✓	✓		✓		✓		
OpenMP [31]	✓✓✓✓	✓✓✓			✓					✓		
PALMOS [32]	✓	✓✓ ✓✓	✓✓		✓	✓		✓		✓		✓
PaRSEC [33]	✓✓	✓			✓	✓		✓		✓		✓
PTask [34]	✓	✓✓ ✓✓	✓✓		✓	✓	✓	✓		✓		
SnuCL [35]	✓✓ ✓	✓✓ ✓✓	✓✓		✓	✓		✓		✓		✓
SYCL [11]	✓✓✓✓	✓✓✓✓✓	✓✓		✓	✓		✓		✓	✓	
STAPL-RTS [36]	✓✓✓									✓		✓
TBB [37]	✓✓✓✓									✓		
TVM [38]	✓✓ ✓	✓	o						✓	✓	✓	
Uintah [39]	✓✓✓✓	✓								✓		✓
VAST [40]		✓			✓			✓	✓			
VirtCL [41]		✓			✓			✓			✓	

VI. Conclusions

Nearly all contemporary programming models for heterogeneous systems depend on rich RTSs to manage execution and orchestrate data movement. Moving forward into the era of heterogeneity, we believe that RTSs must provide even more capability to ensure some level of performance portability across diverse architectures. To address these challenges, we have designed, implemented, and evaluated a prototype system named IRIS: a portable RTS for exploiting multiple heterogeneous programming systems. IRIS can discover available functionality, manage multiple diverse programming systems simultaneously in the same application, represent data dependencies, proactively orchestrate data movement, and allow configurable policies for scheduling work on multiple heterogeneous devices. IRIS also offers a shared device memory with relaxed consistency among different heterogeneous devices, thereby allowing dynamic management of data movement. We evaluated IRIS on multiple architectures, ranging from Qualcomm Snapdragon to the Summit supercomputer, to show that IRIS provides performance portability across diverse architectures and allows the application to effectively exploit all available devices within a node. We believe IRIS's capabilities will be crucial in the era of heterogeneous architectures.

ACKNOWLEDGMENT

This research used resources of the Experimental Computing Laboratory and the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory, which are supported by the US Department of Energy's Office of Science of under contract no. DE-AC05-00OR22725.

This research was supported by (1) the Defense Advanced Research Projects Agency's Microsystems Technology Office, Domain-Specific System-on-Chip Program and (2) the US Department of Defense, Brisbane: Productive Programming Systems in the Era of Extremely Heterogeneous and Ephemeral Computer Architectures.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[2] M. J. Schulte, M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt, and G. Rodgers, "Achieving exascale capabilities through heterogeneous computing," *Micro, IEEE*, vol. 35, no. 4, pp. 26–36, 2015.

[3] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. V. Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. P. Jr., T. Peterka, M. Strout, and J. Wilke, "Extreme heterogeneity 2018 - productive computational science in the era of extreme heterogeneity: Report for DOE ASCR workshop on extreme heterogeneity," USDOE Office of Science (SC) (United States), Tech. Rep., 2018.

[4] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, vol. 92, pp. 947–958, 2019.

[5] J. Nickolls and I. Buck, "NVIDIA CUDA software and GPU parallel computing architecture," in *Microprocessor Forum*, 2007.

[6] AMD, "HIP: C++ heterogeneous-compute interface for portability," 2020.

[7] *oneAPI Programming Model*, https://www.oneapi.com/.

[8] K. Group, "OpenCL: The open standard for parallel programming of heterogeneous systems," 2019.

[9] OpenACC, "OpenACC: Directives for accelerators," 2015.

[10] *OpenMP Application Programming Interface. Version 5.0*, https://www.openmp.org/specifications/.

[11] K. Group, "SYCL: C++ single-source heterogeneous programming for openCL," 2019.

[12] S. Lee and J. S. Vetter, "Openarc: Open accelerator research compiler for directive-based, efficient heterogeneous computing," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14, 2014, pp. 115–120.

[13] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[14] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta, "Productive programming of gpu clusters with ompss," in *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '12, 2012, pp. 557–568.

[15] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par'09, 2009, pp. 863–874.

[16] J. Cabezas, L. Vilanova, I. Gelado, T. B. Jablin, N. Navarro, and W.-m. W. Hwu, "Automatic parallelization of kernels in shared-memory multi-gpu nodes," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15, 2015, pp. 3–13.

[17] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, A. Castello, D. Genet, T. Herault, P. Jindal *et al.*, "Argobots: A lightweight threading/tasking framework," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2016.

[18] A. Sabne, P. Sakdhnagool, and R. Eigenmann, "Scaling large-data computations on multi-gpu accelerators," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13, 2013, pp. 443–454.

[19] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93, 1993, pp. 91–108.

[20] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95, 1995, pp. 207–216.

[21] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: A compiler and runtime for heterogeneous systems," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, 2013, pp. 49–68.

[22] S. J. Krieder, J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu, "Design and evaluation of the gemtc framework for gpu-enabled many-task computing," in *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 153–164. [Online]. Available: https://doi.org/10.1145/2600212.2600228

[23] M. Kotsifakou, P. Srivastava, M. D. Sinclair, R. Komuravelli, V. Adve, and S. Adve, "HPVM: heterogeneous parallel virtual machine," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Vienna, Austria: ACM, 2018, pp. 68–80.

[24] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14, 2014, pp. 6:1–6:11.

[25] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, pp. 66:1–66:11.

[26] R. Gioiosa, B. O. Mutlu, S. Lee, J. S. Vetter, G. Picierro, and M. Cesati, "The minos computing library: Efficient parallel programming for extremely heterogeneous systems," in *Proceedings of the 13th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, ser. GPGPU '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–10. [Online]. Available: https://doi.org/10.1145/3366428.3380770

[27] K. Spafford, J. Meredith, and J. Vetter, "Maestro: Data orchestration and tuning for opencl devices," in *Proceedings of the 16th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par'10, 2010, pp. 275–286.

[28] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a single compute device image in opencl for multiple gpus," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11, 2011, pp. 277–288.

[29] T. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, and M. Lee, "The open community runtime: A runtime system for extreme scale computing," in *2016 IEEE High Performance Extreme Computing Conference*, ser. HPEC '16, 2016, pp. 1–7.

[30] K. Group, "Open compute language (openCL)," 2008.

[31] OpenMP, "OpenMP reference," 1999.

[32] C. Margiolas and M. F. O'Boyle, "Palmos: A transparent, multi-tasking acceleration layer for parallel heterogeneous systems," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15, 2015, pp. 307–318.

[33] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science and Engg.*, vol. 15, no. 6, pp. 36–45, Nov. 2013.

[34] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "Ptask: Operating system abstractions to manage gpus as compute devices," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, 2011, pp. 233–248.

[35] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "Snucl: An opencl framework for heterogeneous cpu/gpu clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12, 2012, pp. 341–352.

[36] I. Papadopoulos, N. Thomas, A. Fidel, N. Amato, and L. Rauchwerger, "Stapl-rts: An application driven runtime system," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. Newport Beach, California, USA: Association for Computing Machinery, 2015, p. 425–434.

[37] J. Reinders, *Intel Threading Building Blocks*, 1st ed. O'Reilly & Associates, Inc., 2007.

[38] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, and L. Ceze, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[39] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins, "Investigating applications portability with the uintah DAG-based runtime system on petascale supercomputers," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*, pp. 1–12, 2013.

[40] J. Lee, M. Samadi, and S. Mahlke, "Vast: The illusion of a large memory space for gpus," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14, 2014, pp. 443–454.

[41] Y.-P. You, H.-J. Wu, Y.-N. Tsai, and Y.-T. Chao, "Virtcl: A framework for opencl device abstraction and management," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015, 2015, pp. 161–172.