IRIS Reimagined: Advancements in Intelligent Runtime System for Task-Based Programming

Narasinga Rao Miniskar[1 [0000−0001−8259−8891]], Seyong Lee[1 [0000−0001−8872−4932]], Johnston Beau[1 [0000−0001−5426−1415]], Aaron Young[1 [0000−0002−5448−4667]], Mohammad Alaul Haque Monil[1 [0000−0003−3419−4037]], Pedro Valero-Lara[1 [0000−0002−1479−4310]], and Jeffrey S. Vetter[1 [0000−0002−2449−6720]]

Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
(miniskarnr, lees2, johnstonbe, youngar, monilm, valerolarap, vetter)@ornl.gov

Abstract. Task-based programming models are gaining traction in scientific computing. IRIS is a portable runtime system that exploits multiple heterogeneous programming systems and can discover available resources and manage multiple diverse programming systems (e.g., CUDA, Hexagon, HIP, Level Zero, OpenCL, and OpenMP) simultaneously. It accounts for the constraints of task dependencies and provides customizable scheduling policies to map those tasks to heterogeneous devices. In this paper, we present new capabilities added to IRIS to improve its portability for heterogeneous programming, build-friendliness, and performance efficiency. The new additions include vendor-specific kernel support, a runtime system with a foreign function interface to eliminate writing wrapper or boilerplate code for heterogeneous kernels, an easy-to-use and configurable CMake-based build environment, automatic and efficient data transfers and orchestration, and the Hunter and DAGGER toolchains to evaluate IRIS's task scheduling algorithms.

Keywords: Heterogeneous Computing · Runtime System · IRIS · DMEM · CUDA · HIP · Task based programming

1 Introduction

The current trend in computer architectures is a transition toward extreme heterogeneity, with a focus on domain-specific computing. Initially observed in mobile and embedded markets, this trend is expanding into high-performance computing (HPC), machine learning, enterprise, and cloud computing[8]. Contemporary architectures such as NVIDIA Xavier and Qualcomm Snapdragon are notable examples that indicate the movement toward systems-on-chip (SOCs) containing multiple heterogeneous processors for diverse applications. In HPC, most of the top systems are heterogeneous [17], and this trend is expected to persist. However, the challenge lies in the lack of programming systems that span these architectures while ensuring performance portability. Various programming models exist—from directive-based high-level programming (OpenMP [16], OpenACC [15], etc.) and C++ template metaprogramming (Kokkos [5], RAJA [4], SYCL [7], etc.) to low-level device-specific programming (CUDA [14], HIP [1], etc.). However, these models' implementation and portability are inconsistent. One common feature among these heterogeneous programming models is the reliance on runtime systems (RTS), which necessitates efficient management of resources and data dependencies, as well as dynamic balancing of goals during execution.

To address the challenges posed by increasingly diverse heterogeneous systems, IRIS [8] was developed to provide enhanced capabilities in heterogeneous RTSs. IRIS includes features such as online adaptive scheduling, dynamic resource discovery, proactive data movement, support for simultaneous execution of multiple heterogeneous devices, and an interface for online code generation. These advancements aim to achieve performance portability by dynamically adapting to system constraints and dependencies that are often unknown until execution time, offering a solution to the complex scheduling challenges inherent in heterogeneous architectures. This paper presents work that builds on the original IRIS framework by adding new, more advanced capabilities to IRIS to make it more portable, easier to program and build, and more performant. The summary of enhancements and contributions to IRIS is given below.

- Vendor-specific kernels: IRIS was extended to allow vendor-specific kernel calls. These are written with host wrapper kernel functions to enable vendor-specific libraries (e.g., cuBLAS) to be callable from IRIS instead of having to write the kernels in their native device languages (e.g., CUDA).
- Foreign function interface: this contribution extends IRIS to use the foreign function interface (FFI) library [6] to call the device-specific kernels with parameters. This means that users can avoid writing the explicit boilerplate code or wrapper code (written in C++) for each IRIS kernel and each of its parameters.
- Distributed data memory management: IRIS was extended with a new heterogeneous and distributed memory handling mechanism (DMEM) to automate data movement across devices based on the requirements—but without the need for the programmer to explicitly specify the host-to-device (H2D)

and device-to-host (D2H) data transfer APIs. DMEM reduces the overall data transfers by 4× and achieves performance gains of 5× when compared to the performance of manually handled memory management through H2D and D2H APIs.
- A heterogeneous build environment was added for IRIS-based applications to ease the development for heterogeneous systems.
- Support for Hunter: a new framework developed for exploration of scheduling algorithms. It also offers a runtime simulator for heterogeneous platforms.
- Support for DAGGER: another framework developed to create the artificial task graphs for evaluating the task scheduling policies of IRIS. A brief example of its usage is included in this paper to highlight the device-scaling performance portability of IRIS.

The rest of the paper is organized as follows: Section 2 provides a description of the IRIS runtime framework. Section 3 discusses the state-of-the-art runtime systems and a comparison with IRIS. The extensions of IRIS and new capabilities are described in Section 4. Section 5 provides the results for each of IRIS's new capabilities. Finally, we conclude the paper and describe future work in Section 6.

2 Background: IRIS

IRIS is a task-based programming model for extremely heterogeneous devices in a system [8,11,10]. It enables application developers to write applications for diverse heterogeneous programming platforms with native device-specific programming models, including CUDA, HIP, Level Zero, OpenCL, and OpenMP. IRIS orchestrates multiple programming platforms and consolidates them into a single execution/programming environment by providing portable tasks and shared virtual device memory. IRIS provides shared virtual device memory across multiple disparate physical device memories to achieve application portability and flexible task scheduling with effective data orchestration [12]. It does so by automatically transferring data across multiple devices to maintain memory consistency across tasks. This memory-model abstraction allows developers to prioritize the application's primary feature set and overall function rather than worrying about coding for device-specific memory spread across multiple heterogeneous devices (unscalable and less portable).

A *Task* is a scheduling unit in IRIS and is mapped to a device by the customizable scheduling policy available in IRIS to run on a single device of heterogeneous system. A task comprises zero or more commands; these can be either data transfer commands or kernel execution commands on an accelerator. The data transfer commands can be H2D, D2H, or device to device (D2D). The DMEM methodology added to IRIS automatically derives these data transfer commands for each task based on the data required for the task, data availability on other devices, and data dependency on other tasks. IRIS provides APIs to indicate the dependency of a task on other tasks.

3 Related Work

Task-based dynamic runtimes are positioned as part of the solution to some of the most important challenges in HPC today [2,11], such as programming productivity, performance portability, and extreme heterogeneity. However, novel developments are needed in software abstractions to increase application portability using fully dynamic runtime systems to schedule and control highly varied resources. Existing standard approaches rely mostly on static mapping and scheduling, such that the programmer must decide which device to use for each task, such as OpenMP tasking [18] or others [9]. We can find two dynamic runtime systems with good support for the aforementioned challenges :StarPU [3] and IRIS [8]. Although StarPU provides a relatively easy-to-use interface and has support for heterogeneous architectures, IRIS goes further by supporting a simpler interface for a greater variety of devices and programming models. Moreover, IRIS eliminates the need for a wrapper for the codes to be computed in the tasks and provides highly optimized memory management that can scale up performance on a high number of disparate accelerators [13].

4 IRIS Re-imagined

This paper presents work done to extend the IRIS framework with new capabilities to further increase its performance, portability, and programming efficiency.

4.1 Vendor-specific kernels

HPC vendor libraries (such as math libraries) face important challenges caused by the explosion of in-node heterogeneity. Due to this increasing heterogeneity, manual computation orchestration across runtimes and devices is becoming intractable. Historically, vendors and open-source BLAS libraries have mostly focused on a single architecture. However, recently, some math libraries from the open-source community have been focusing on supporting heterogeneity in their software stack (e.g., Chameleon). However, the need to support heterogeneity across a diversity of architectures is still a challenge. The IRIS runtime system was developed to provide the orchestration: it provides functionalities such as task offloading, whereby, the computation uses vendor-optimized kernels according to the device to which the task is issued. Using this feature, IRIS facilitates newer abstractions to solve larger math problems by ensuring that that most appropriate processors for a given task to harness scalability and performance. Several vendor library abstractions are built on top of IRIS using these capabilities, such as MatRIS [13], which encapsulates previous efforts such as IRIS-BLAS [11], and LARIS [12]. Because of this feature support in IRIS, MatRIS is fully heterogeneous and portable across heterogeneous systems.

4.2 Foreign Function Interface (FFI)

IRIS requires boilerplate (wrapper) code for each kernel implementation for an OpenMP device. Adding support for vendor-specific kernel calls [11] high-

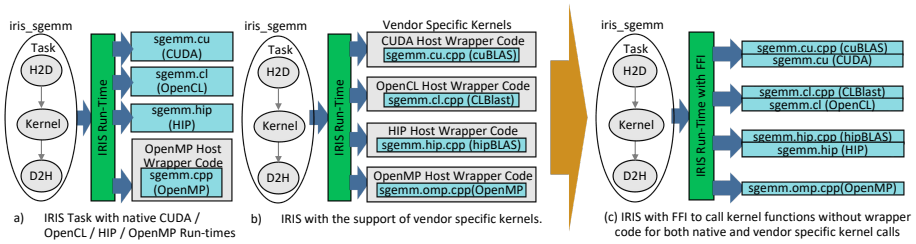

Fig. 1: IRIS with FFI

lighted that for each call, IRIS also needs boilerplate code—often multiple versions for each type of supported device (such as CUDA, HIP, and OpenMP). Figure 1(a,b) shows the boilerplate code requirement for IRIS-native kernels and vendor-specific kernels. The boilerplate code contains an implementation of method APIs to set the positional kernel arguments as well as a method to call the actual kernel with positional parameters. The positional parameters are stored in a temporary run-time memory chunk. It is tedious to write boilerplate code for each kernel. A simple SAXPY kernel for OpenMP device requires ∼100 lines of boilerplate code to be written, and this effort is required for all vendor-specific kernels and for all devices. IRIS was extended to use *libFFI* so that programmers need not write boilerplate code for IRIS kernels, as shown in Figure 1(c).

4.3 Distributed Data Memory Management (DMEM)

Orchestrating memory objects and their copies across heterogeneous devices during execution plays a vital role in heterogeneous systems. Data movement between heterogeneous devices often becomes a bottleneck in achieving strong performance because the data transfer overhead nullifies the impact of greater parallelism. Moreover, accelerators such as GPUs in contemporary HPC nodes have faster connections with higher bandwidth to transfer data. Such a connection requires one D2D transfer as opposed to two transfers (i.e., D2H and H2D). To orchestrate memory object copies in different devices and efficiently utilize the data transfer among them, IRIS introduces DMEM, which provides transparent and efficient data communication (such as invoking faster D2D data transfer when possible) and managing the set of copies of memory in a heterogeneous system.

As a logical memory handler, DMEM stores the addresses of an application's data objects in host and device memory. By keeping a *dirty flag*, DMEM keeps track of which copy of a particular memory object is the most recent. As the execution progresses, the DMEM logic controller continues updating the flags when a change occurs in a memory object. Because IRIS dependencies in a directed acyclic graph (DAG) prevent race conditions, only one device can write to a particular memory object. As seen in Figure 2, the *A00* tile is a shared IRIS–DMEM memory object for all tasks (*X1, X2, ..., XN tiled matrix multiplication*

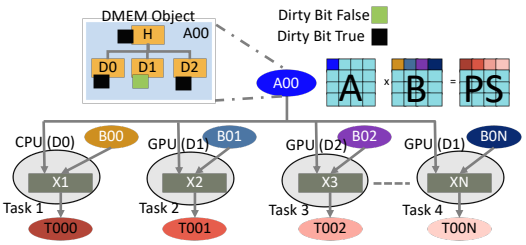

Fig. 2: IRIS DMEM data object example for tiled matrix multiplication. A and B are input tiled matrices, and PS denotes the partial sum tiled matrix

kernels). Programmers do not need to use H2D and D2H command APIs for DMEM memory objects because the DMEM controller handles data transfers at runtime, based on the data requirements of the task kernels.

The DMEM memory handler acts like a write-back cache to prevent unnecessary data movement to host memory. The data transfer to output host memory object is required in two scenarios: (1) when the application accesses it after the execution of tasks or task graphs and (2) when the device lacks valid data that is available in another device but a direct D2D data transfer is not possible. DMEM cannot avoid the second scenario, but it can postpone the first scenario's D2H data transfer until the application is really needed. The application programmer musr call an IRIS API with an explicit DMEM flush-out command and submit it as a part of the last task or as a standalone task. For more technical details about the DMEM methodology and data transfer priorities, refer to the HPEC conference paper [10].

4.4 Heterogeneous Build Environment for IRIS Applications

The CMake build utility was added to IRIS to build applications, along with heterogeneous kernels. This utility enables the developers to configure the CMake variables with appropriate sources, and it builds the necessary libraries for each heterogeneous device. It also supports application sources that developers can use to build application libraries and executables with the included IRIS library links for IRIS APIs.

4.5 Hunter

The Hunter Framework is designed to enable the exploration of scheduling algorithms for large-scale heterogeneous architectures. Hunter was developed and leveraged for this work to model heterogeneous platforms and evaluate the performance of various scheduling algorithms on these platforms. Hunter is tightly coupled with the IRIS runtime API, which allows task graphs within IRIS to be exported to Hunter. Likewise, schedules within Hunter can be run using the IRIS runtime.

Hunter is a Python library with multiple components designed to work together. Hunter uses an object-oriented programming style; a parent class defines

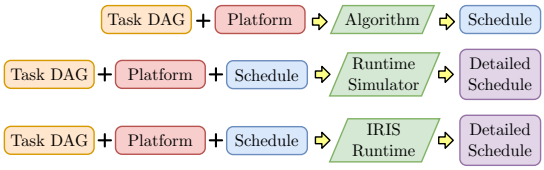

Fig. 3: Block diagram highlighting the use flows for the Hunter Framework.

the API and multiple derived classes, which can each have their own implementation of the API. The main input components of Hunter are a task DAG to define the application task graph and a platform to define the model of the computer system. Scheduling algorithms implemented in Hunter takes a task DAG and platform as inputs to provide scheduling decisions (i.e., schedule). The task DAG and platform, along with a schedule, can then be passed to the Hunter runtime simulator or the IRIS runtime for evaluation. The output of the evaluation includes scheduling with additional execution details, more accurate simulated execution times, and the actual execution times from running on the hardware using IRIS. These evaluation flows are highlighted in Figure 3.

4.6 DAGGER

The Directed Acyclic Graph Generator to Evaluate Runtimes (DAGGER) is a tool developed to synthesize payloads, generating task DAGs of arbitrary complexity. It was initially built to test interesting corner cases easily missed when building a runtime system, and so was used to verify the scheduling decisions when developing IRIS. This functionality was extended for evaluations of different desired DAG characteristics—allowing us to examine and evaluate scheduling policies, and the general performance of IRIS, without having to write an exhaustive corner cases (on the order of hundreds of diverse applications).

DAGGER is composed of a generator and a runner program. The generator accepts a range of parameters used to statistically determine the structure and shape of the DAG. Additionally, the user provides the kernel names (with appropriate kernel arguments) with the probabilities to which each generated task will be assigned. The resultant DAG is recorded as an IRIS-readable JSON file. The DAGGER runner accepts the same arguments and loads the JSON file by handing it directly to IRIS, but it serves, importantly, as a proxy application by allocating the correct number of memory objects and other kernel arguments. It also uses IRIS's internal profiling information but controls where it is logged—for recording results.

Users adjust variables that allow varying the width and depth, and the number of tasks, whereas the shape of the DAG and distribution of tasks are determined by adjusting the cumulative distribution functions' mean and standard deviation variables. Finally, complexity in the generated DAG can be set by increasing the number of skips, which can lead to interesting interactions among tasks by increasing the potential interactions of tasks between levels. Additionally, it allows each task to be assigned kernel names that are statistically selected

by providing the associated probability, the dimensionality of the kernels, and the memory buffers associated with each kernel task. DAGGER allows the synthetic generation of DAGs with interesting shapes and interactions.

New features recently added to DAGGER include the ability to specify the number of concurrent memory buffers allowed for each kernel name and the sharing of memory objects used between tasks; both allow the user to indicate the potential concurrency by mitigating data dependencies between tasks. Finally, both the runner and generator can generate explicit D2H and H2D memory transfers (to get the memory into IRIS when submitting the task graph) or using IRIS's DMEM, which replaces the final tasks in the graph from `D2H` with the required *DMEM_FLUSH_OUT_CMD*.

5 Results

The effectiveness of IRIS's new capabilities was evaluated using a tiled matrix multiplication benchmark running on a truly heterogeneous system—comprising four NVIDIA A100 and four AMD MI100 GPUs. The tiled matrix multiplication algorithm is implemented using vendor-specific kernels support in IRIS and is part of the MatRIS framework [13,11,12]. We then explored the scaling of IRIS's scheduling policies by running an identical DAG on multiple systems, each with different combinations of GPUs from multiple vendors.

5.1 FFI

Table 1: Matrix multiplication performance with and without FFI. Each experiment is run for 10 times and present the median values.

Matrix Size	Tile Count	Tasks	FFI (GFLOPS)	Boilerplate code(GFLOPS)
4096	2x2	8	5.1	4.2
8192	2x2	8	11.3	9.8
16384	2x2	8	20.8	20.8
16384	4x4	64	19.9	19.2
16384	8x8	512	18.6	17.9
16384	16x16	4096	0.76	0.78

The effectiveness of FFI was measured using a tiled matrix multiplication benchmark. We varied the matrix and tile size to scale over the number of tasks and measured the performance of the FFI version versus the traditional IRIS-based boilerplate code. It has been observed that FFI-based kernel calls have no additional overhead compared to that of boilerplate code. Moreover, in some cases, the performance is slightly better than explicit boilerplate code.

5.2 DMEM

The capabilities of distributed memory objects (DMEM) in IRIS are demonstrated in Table 2. For this experiment, we ran MatRIS [13] LU factorization on a CADES cloud node with four NVIDIA A100 and four AMD MI100 GPUs. We considered a $\approx 32K \times 32K$ matrix with 16×16 tiling, which created around 1500 tasks scheduled to eight GPUs (both NVIDIA and AMD). The first data row of Table 2 shows H2D and D2H data transfer considering each task-initiated data transfer from the host, and, after computing, returned the updated data to the

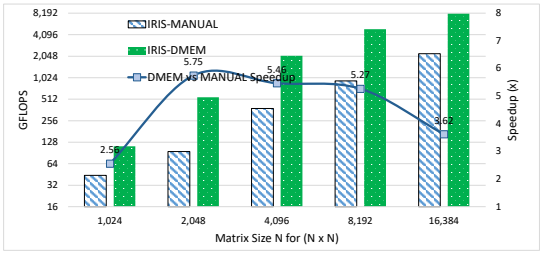

Fig. 4: Performance improvement by DMEM for a tiled matrix multiplication benchmark. Platform: four NVIDIA A100 CUDA GPUs and four AMD MI100 GPUs. The y-axis is on a log scale. Each experiment was run 10 times, and median values are presented.

host: it shows around 5000 data transfers in total. However, the use of DMEM (second row of the table) significantly reduced both H2D and D2H transfers because DMEM can find the last location of the data and initiate the appropriate transfer when required. Also, DMEM uses D2D transfers when possible, which provides an additional performance boost. When data needs to be transferred from an AMD GPU to an NVIDIA GPU, DMEM orchestrates the corresponding D2H and H2D transfer. Because of these strategies employed by DMEM, IRIS yields an order of magnitude lower number of total data transfers, which enables superior performance. We also observed nearly 5× gains on matrix multiplication using our intelligent DMEM memory handling technology, as shown in Figure 4. The gains are due to a reduced number of data transfers and using optimal data transfer APIs.

Table 2: Count of data transfers and their type for DMEM and without DMEM on the 8× GPUs (NVIDIA and AMD). Benchmark: LU factorization on 32,678 × 32,768 dense matrix

System	**CADES**			
Transfer	H2D	D2H	D2D	total
Without DMEM	4,219	1,497	0	5,716
With DMEM	567	395	426	1,388

5.3 DAGGER

In this experiment, an identical DAGGER-generated DAG was run on several systems to highlight the scaling of IRIS's dynamic scheduling policies. Each system is different, featuring unique combinations of the number of GPUs, from both NVIDIA and AMD, in different generations, as shown in Table 3. For brevity, we excluded OpenMP and OpenCL runtimes from the evaluation. IRIS provides the abstract task view given by the DAG; it can automatically link and resolve the appropriate kernel to the underlying backend/runtime on the system—so no code changes are needed. Additionally, IRIS honors data locality, internally tracking ownership/modification of memory buffers of devices; thus, we can change the scheduling policy to affect performance, and it will add the required memory movement to ensure correctness.

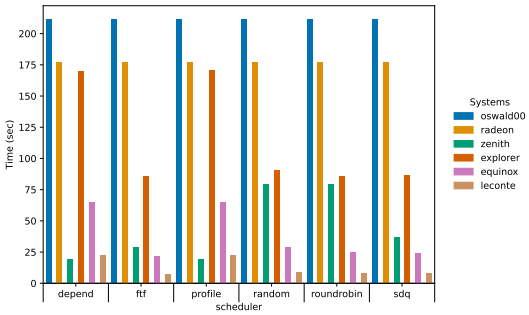

Fig. 5: Execution times of DAG on systems with varying dynamic scheduling policies.

The generated task DAG used for this experiment contains 240 `bigk` kernel tasks with 6 concurrent duplicates of memory objects at each level; the instance of each memory object used by each task is stochastic, resulting in a large workload of complex dependencies. This synthetic workload is largely compute-bound, featuring a double-nested `for loop` to compute a sum. The width of the DAG is 6 at each level and is thus 40 levels deep. Each kernel invocation was run at a size of 1024 (i.e., the largest possible number of work items on AMD GPU-based systems).

Table 3: The GPU configuration of systems used in the evaluation.

System	Vendor	Generation	Card Name	# of GPUs	Runtimes
Oswald	NVIDIA	Pascal	P100	1	CUDA
Radeon	AMD	Vega II	Vega 20	1	HIP
Zenith	NVIDIA	Ampere	GA102	1	CUDA
	AMD	Navi II	Navi 21	1	HIP
Explorer	AMD	Vega II	MI60	2	HIP
Equinox	NVIDIA	Volta	V100	4	CUDA
Leconte	NVIDIA	Volta	V100	6	CUDA

The results are shown in Figure 5. Typically, we see systems with fewer GPUs benefit less from scheduling policy selection, whereas systems with more devices benefit from dynamic policy selection. A good example of this effect can be seen by focusing on the 2× speedup achieved by the *Explorer* system when using **ftf**, **random**, **roundrobin**, or **sdq** over the **depend** or **profile** policies. By experimental design, we assigned all memory to one device at the start of the graph submission, and both *depend* and *profile* policies aim to avoid unnecessary memory movement, yielding a sequential baseline for our comparison. In contrast, the speedups gained by the First-to-Finish **ftf** (only assigns the next task to an idle device, effectively work-stealing), Shortest-Device-Queue **sdq** (assigns each task to the device with the fewest tasks in its queue), **roundrobin**, and **random** ignore the cost of memory transfers and thus can fully saturate the devices—provided the DAG has enough parallelism, which our experiment ensures. Systems with more devices further highlight this trend, achieving a 3× speedup on *Equinox* and *Leconte*: there are memory movement costs that prevent this workload from reaching the theoretical limit of 4 and 6×, respectively.

The only system that was observed to go against this trend is *Zenith*, which, unfortunately, is the only truly heterogeneous machine in this study. The penalty here is in moving memory between two different runtimes: (D2D) memory movement in this case defaults to moving memory through the host (as separate H2D and subsequent D2H calls), which both have their own synchronization points. Performance is worsened here when we consider the hardware. Each stage of this communication must occur over the PCI-E interconnect, whereas this cost is notapplicable for single runtime systems that have vendor-specific interconnects (NVLink and Infinity Fabric) for fast D2D memory transfers. This results in more reckless policies (in terms of ignoring memory locality) such as **roundrobin** and **random** suffering by taking 3–4× longer than the memory-aware scheduling decisions (**depend** or **profile**), whereas **ftf** and **sdq** performance falls in between because these policies do not consider memory locality but do rely on feedback from the device queues.

The older generation and single-card systems (*Oswald* and *Radeon*) were not affected by the choice of scheduling policy since they do not have enough hardware to exploit the concurrency in DAG. Here, the discrepancy in absolute performance is attributed to hardware differences in the P100 and Vega II GPUs.

6 Conclusion

This paper presents the new capabilities added to IRIS to achieve better performance efficiency, higher portability, improved programmability, and a more convenient build environment. The IRIS DMEM memory handler has achieved 2.5× to 5.7× improvement in performance when compared to manually introduced data transfer calls. Features such as FFI in IRIS allow programmers to avoid writing boilerplate code for each kernel, which is usually 70 to 100 extra lines of code per kernel. Moreover, the addition of the DAGGER and Hunter frameworks enhanced IRIS capabilities for verification and scalability by providing an unbounded range of task and device experiments. This paper shows the scaling performance of IRIS using dynamic scheduling policies on a fixed DAGGER workload. In the future, we will present the performance of static scheduling policies—leaning heavily on the Hunter framework.

References

1. AMD: HIP: C++ heterogeneous-compute interface for portability (2020)
2. Ang, J., Chien, A.A., Hammond, S.D., Hoisie, A., Karlin, I., Pakin, S., Shalf, J., Vetter, J.: Reimagining Codesign for Advanced Scientific Computing: Report for the ASCR Workshop on Reimaging Codesign (2022), `https://www.osti.gov/biblio/1822199`, [Online; accessed 6-July-2022]
3. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.: Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In: Sips, H.J., Epema, D.H.J., Lin, H. (eds.) Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5704, pp. 863–874. Springer (2009)

4. Beckingsale, D.A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A.J., Pearce, O., Robinson, P., Ryujin, B.S., Scogland, T.R.: Raja: Portable performance for large-scale scientific applications. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC). pp. 71–81 (2019)
5. Edwards, H.C., Sunderland, D.: Kokkos array performance-portable manycore programming model. In: Guo, M., Huang, Z. (eds.) Proceedings of the 2012 PPOPP International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM 2012, New Orleans, LA, USA, February 26, 2012. pp. 1–10. ACM (2012). https://doi.org/10.1145/2141702.2141703
6. Group, F.W.: libffi: A Portable Foreign Function Interface Library. GNU Project (Year of the latest version), https://sourceware.org/libffi/
7. Group, K.: SYCL: C++ single-source heterogeneous programming for openCL (2019)
8. Kim, J., Lee, S., Johnston, B., Vetter, J.S.: IRIS: A portable runtime system exploiting multiple heterogeneous programming systems. In: 2021 IEEE High Performance Extreme Computing Conference, HPEC 2021, Waltham, MA, USA, September 20-24, 2021. pp. 1–8. IEEE (2021)
9. Korakitis, O., Gonzalo, S.G.D., Guidotti, N., Barreto, J.P., Monteiro, J.C., Peña, A.J.: Towards ompss-2 and openacc interoperation. In: Lee, J., Agrawal, K., Spear, M.F. (eds.) PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022
10. Miniskar, N.R., Haque Monil, M.A., Valero-Lara, P., Liu, F.Y., Vetter, J.S.: Irisdmem: Efficient memory management for heterogeneous computing. In: 2023 IEEE High Performance Extreme Computing Conference (HPEC). pp. 1–7 (2023)
11. Miniskar, N.R., Mohammad, A.H.M., Pedro, V.L., Liu, F., Vetter, J.S.: Iris-blas: Towards a performance portable and heterogeneous blas library. In: 29th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2022, Bengaluru, India, December 18-21, 2022. pp. 1–10. IEEE (2022)
12. Monil, M.A.H., Miniskar, N.R., Liu, F., Vetter, J.S., Valero-Lara, P.: LaRIS: Targeting Portability and Productivity for LaPACK Codes on Extreme Heterogeneous Systems using IRIS. In: IEEE/ACM Redefining Scalability for Diversely Heterogeneous Architectures Workshop, Dallas, TX, USA, November 13–18, 2022. IEEE
13. Monil, M.A.H., Miniskar, N.R., Teranishi, K., Vetter, J.S., Valero-Lara, P.: Matris: Multi-level math library abstraction for heterogeneity and performance portability using iris runtime. In: Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. pp. 1081–1092 (2023)
14. Nickolls, J., Buck, I.: NVIDIA CUDA software and GPU parallel computing architecture. In: Microprocessor Forum (2007)
15. OpenACC: OpenACC Application Programming Interface (2024), https://www.openacc.org, [Online; accessed 24-January-2024]
16. OpenMP: OpenMP Application Programming Interface (2024), https://www.openmp.org/specifications/, [Online; accessed 24-January-2024]
17. TOP500.org: November 2023 TOP500 (2023), https://www.top500.org/lists/top500/2023/11/, [Online; accessed 24-January-2024]
18. Valero-Lara, P., Kim, J., Hernandez, O., Vetter, J.S.: Openmp target task: Tasking and target offloading on heterogeneous systems. In: Euro-Par 2021: Parallel Processing Workshops - International Workshops, Lisbon, Portugal, August 30-31, 2021. Lecture Notes in Computer Science, vol. 13098. Springer (2021)