Workload Scheduling on Heterogeneous Devices

Harsh Khetawat*, Frank Mueller*

*Department of Computer Science, North Carolina State University
Email: hkhetaw@ncsu.edu, fmuelle@ncsu.edu

***Abstract*—Hardware accelerators have become the backbone of many cloud and HPC workloads, but workloads tend to statically choose accelerators leaving devices unused while others are oversubscribed. We propose a holistic framework that allows a computational kernel to span across multiple devices on a node, as well as multiple applications being scheduled on the same node. Our work sharing and co-scheduling framework allows kernels to be migrated between devices, expand to span more devices, or contract to fewer devices. The scheduler can make these decisions dynamically based on a pluggable scheduling algorithm in order to optimize for different objectives, e.g., job throughput, job priorities or some hybrid. Experiments on a CPU+GPU+FPGA platform indicate speedups of 2.26X over different applications and up to 1.25X for co-scheduled workloads over baselines. Besides performance, a major contribution of our work lies in ease of programmability with a single code base compiled and runtime controlled across three vastly different execution devices.**

***Index Terms*—Work sharing, co-scheduling, OpenCL, heterogeneous, FPGA, GPU**

I. Introduction

Motivation: Over the last decade significant changes have impacted HPC hardware. The introduction of accelerators, such as GPUs, FPGAs and DSPs, have added considerable computational capabilities to HPC systems, albeit at the cost of programming complexity. Furthermore, recent HPC systems such as Summit [1] and Sierra [2] (both IBM Power9 + NVIDIA GV100) as well as Frontier [3] (AMD X86_64 + AMD MI230X) and Perlmutter [4] (AMD X86_64 + NVIDIA A100) are adopting fatter nodes with multiple CPUs and GPUs in favor of many more thinner nodes to achieve the desired peak performance. This trend is expected to accelerate further [5] with more and diverse accelerators becoming a mainstay not only in HPC systems (e.g., Sandia's NextSilicon FPGA cluster) but also in the cloud, where AWS and Azure already feature FPGAs besides GPUs [6], [7]. While current applications need to be updated to leverage diverse computational hardware, they could instead greatly benefit from automatic resource management with a common code base, as we show here.

Limitation of state-of-art approaches: Current production applications are designed to target specific hardware. Applications target GPUs using frameworks such as CUDA [8] or ROCm, CPUs with OpenMP, and FPGAs with OpenCL (HLS) or VHDL. OpenCL provides a common interface for developing applications for a diverse set of computational hardware but applications running on a specific device leave other computational hardware on the node idle. While there has been some prior work on sharing work amongst heterogeneous computing devices, such studies either require significant changes to the application or do not support a wide variety of devices.

Scheduling multiple applications on a single node has raised much interest in HPC. Co-scheduling applications allows HPC systems to use the available HPC resources more effectively. Current approaches either target specific devices for co-scheduling while reducing interference (e.g., by mitigating shared resources when multiple applications use the same CPU) or avoiding interference by having co-scheduled applications utilize disjoint accelerating devices (e.g., one using GPU and another using FPGAs) on a heterogeneous HPC node.

Contributions: We aim to leverage both work sharing and co-scheduling in a common framework to more holistically use the heterogeneous nature of current and future HPC systems. We split each OpenCL computational kernel into a "bag of tasks" exploiting data parallelism, where each task (or slice) is potentially scheduled on a different device. This enables our work-sharing framework to achieve higher performance than running on any single device. Furthermore, we use this framework to seamlessly migrate, expand, or contract our application between devices. This capability creates a co-scheduling framework to enable multiple applications to run on a single node.

- We create a work-sharing framework to schedule OpenCL kernels on multiple devices without requiring the application developer to make significant changes in the application.
- We provide the capability to seamlessly migrate kernels from one accelerator to another, and to expand/contract kernels to use more/fewer accelerators.
- We augment the work-sharing framework with co-scheduling capabilities by providing a framework with pluggable scheduling algorithms and the ability to optimize for job throughput, job priority, or a hybrid of multiple objectives.
- We evaluate our work-sharing and co-scheduling framework under different scheduling algorithms showing performance benefit by combining accelerators for work sharing as opposed to selecting a single device. Also, workload characteristics such as inter-arrival times and the application mix are shown to impact the suitability of different co-scheduling algorithms.
- We ease programmability across three heterogeneous platforms beyond any prior capability we know of, where a kernel is coded once, compiled automatically to multiple backends, yet executes in a coordinated fashion under a common scheduler that supports migration as resource availability changes.

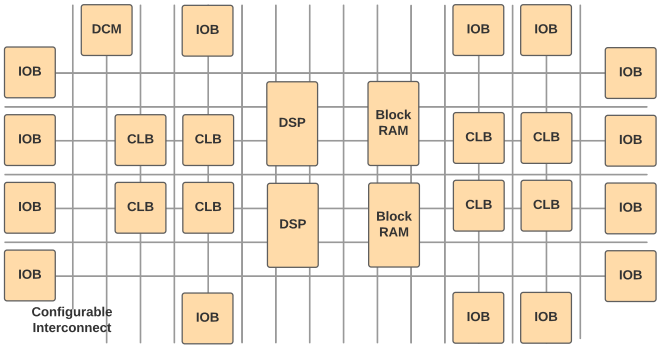


Fig. 1. Block diagram of an FPGA with its components

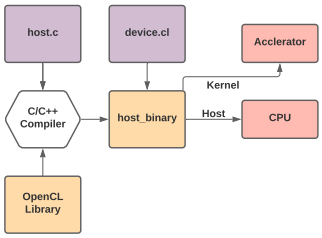


Fig. 2. OpenCL app flow

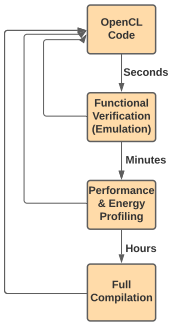


Fig. 3. HLS dev. workflow

Experimental methodology and artifact availability: We evaluate our work sharing and co-scheduling framework using a single node on a mid-sized HPC cluster comprising of one CPU, one GPU and one FPGA. We use four applications to evaluate our work-sharing framework with different combinations of devices. We also create four workloads comprised of different sizes of the applications and evaluate our co-scheduling approach under different scheduling algorithms. We will make our framework and its code base available as open source to the community opening up novel research and community contributions based on our work.

II. Background

This section provides a brief overview of accelerators used in modern HPC systems — GPUs and FPGAs. It also provides a fundamental background on OpenCL and High Level Synthesis (HLS).

GPUs in HPC: General Purpose Graphics Processing Units (GPGPUs) have become increasingly popular and have been driving scientific computation with significant improvements in performance and power efficiency [9], [10]. The development of frameworks and languages such as NVIDIA's Compute Unified Device Architecture (CUDA [8]), the Open Compute Language (OpenCL) [11] and, more recently, Intel's OneAPI [12] have contributed toward making GPUs first class computation devices in modern HPC systems with recently commissioned systems such as Summit [1] and Sierra [2] relying primarily on GPUs for their peak floating point operations per second (Flop/s).

FPGAs in HPC: Field Programmable Gate Array (FPGA) devices as HPC accelerators have gained traction in recent times [13], [14]. FPGAs consist of configurable logic blocks (CLBs), digital signal processing (DSP) blocks, I/O blocks, a block RAM, a digital clock manager (DCM) module and a programmable interconnect to connect these blocks. Figure 1 shows the block diagram of an FPGA and its components. The CLBs consist of look up tables (LUTs), which are used to program the application logic into the FPGA hardware. FPGAs are often connected to the system as PCIe devices and while current generation FPGAs might lag behind other accelerators, they have been shown to be more energy efficient for certain applications [15], [16]. Furthermore, high level synthesis (HLS) has made the process of programming for FPGAs much more accessible leading to an increase in popularity.

OpenCL: Unlike CUDA, a proprietary framework for NVIDIA's GPUs, OpenCL provides an open standard for programming parallel applications for a wide range of accelerators, such as multi-core CPUs, GPUs, FPGAs, DSPs, Tensor cores, etc. It provides a standard interface for task-based and single instruction multiple data (SIMD) parallelism that the application developer can leverage based on the target architecture and application. Like CUDA, the application is divided into the host code and the device (or kernel) code. The host code is responsible for initializing the target device(s), allocating memory on the device, coordinating the movement of data between the host DRAM and the device memory, and enqueuing the kernel on the device. The kernel code describes the computation that is to take place on the accelerator. Figure 2 shows the development flow for an OpenCL application.

The host code is compiled using a C/C++ compiler and linked to the OpenCL library to generate the host binary. The kernel code (shown in the figure as device.cl) can either be pre-compiled for a specific accelerator, or the host code can compile the device code at run-time for any target accelerator (with some exceptions to be discussed later). The host binary is then executed on the CPU, it initializes the target accelerator, co-ordinates data movement, compiles the device code and finally enqueues the kernel on the device before it is run on the accelerator.

High Level Synthesis: Traditionally FPGAs have been dependent on Register Transfer Language (RTL) code to describe the logic that needs to be programmed into the accelerator. High level synthesis (HLS) allows application developers to write the accelerator logic in a high-level language such as OpenCL and compile that to its equivalent RTL code that can be programmed into the FPGA. This allows for abstraction of the core application logic from the underlying hardware description. HLS tools enable compilation of high-level code to RTL and provide tools to ensure correctness. Figure 3 shows the development workflow for FPGAs under HLS. While a full compile for the FPGA can take several hours, application developers can ensure correctness using emulation (on the CPU, without FPGA execution) and make changes to the code if necessary. Furthermore, profiling tools can be used to ensure the energy and performance requirements are

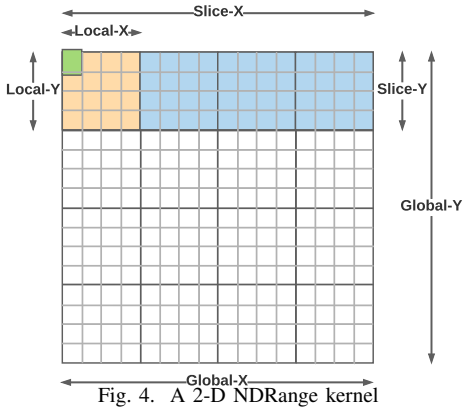


Fig. 4. A 2-D NDRange kernel

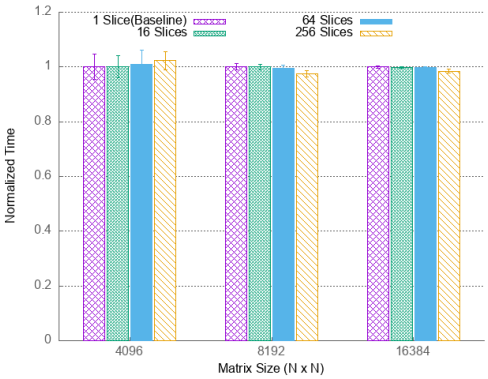


Fig. 5. GEMM slicing overhead + stddev

met without having to embark on a full scale compilation. Finally, once correctness is ensured and performance and energy constraints have been met, the application developers can run the application on the physical hardware.

III. Design and Implementation

This section describes the design and implementation of our work-sharing and co-scheduling framework by detailing our technique of slicing of OpenCL kernels that enables work-sharing and supports co-scheduling within our framework as well as kernel migration between slice invocations.

Kernel Slicing: OpenCL kernels can either be programmed as a single task or an NDRange kernel. In a single-task kernel, SIMD parallelism is not leveraged and there is only one thread of execution. This is not suitable for accelerators such as GPUs or even multi-core CPUs, but it lends well to FPGAs, which utilize pipelined parallelism. An NDRange kernel describes the computation as work items and utilizes SIMD parallelism. In this work, we exclusively study NDRange kernels, which can be programmed to be suitable for all three types of accelerators, CPUs, GPUs and FPGAs.

In an NDRange kernel, the computation is divided into work items using data parallelism, where each work item has its own logical thread of execution. The work items are grouped into a local work group with the ability for threads to synchronize only within the local work group. The overall computation is referred to by the global work group. Along with the local and global work groups, we add the notion of a *slice*, which is a multiple of the local work group size. Figure 4 shows an NDRange kernel for a two-dimensional (2D) computation with a single work item shown in green. Since synchronization is possible only within a local work group, each slice can be computed independently. While it is possible to use the local work group as a slice, in practice, larger slices composed of multiple local work groups allow better utilization of the compute units available on an accelerator. Slices can be constructed using multiples of local work groups in any dimension or even multiple dimensions. Our framework is configured to use slices constructed by extending the local work group in the first dimension.

We leverage this slicing mechanism to transform a single kernel into a series of kernels (equal to the number of slices) with each slice being computed one after the other. Each of the accelerators acquires the next available slice atomically. Figure 5 shows the overhead of slicing with GEMM with different number of slices. For a small input size (4096x4096), we observe that a large number of slices (256) degrades performance by about 5.6% but for larger input sizes the performance is very close to that of a single kernel execution.

Note: For the slicing mechanism to produce correct output, we changed the kernel to take an extra parameter that specifies the current slice. This enables the kernel to write the result of the computation to the correct location in the output buffer(s). This can be avoided by using the clEnqueueNDRangeKernel function (used to enqueue a kernel on an accelerator), which accepts a parameter that specifies an offset for the global work id. However, one of the versions of OpenCL on our system has not implemented this functionality, which forced us to resemble it by code refactoring [17].

Work and Data Sharing: With the ability to organize a single OpenCL kernel into several slices, abstracted in a "bag of tasks", each of which can be scheduled as independent kernel executions, we create a fine-grained work sharing framework. Figure 6 shows the combined design for our work sharing and co-scheduling framework. Memory for the input data structures as well as the output data structures are allocated on each of the accelerators participating in the work sharing for the job. Input data structures are copied to each of the accelerators (if necessary), i.e., each selected target device (e.g., CPU+GPU) receives a complete copy of all input data that the kernel operates on. We later discuss potential optimizations for data partitioning. Then independent threads for each accelerator atomically pick slices from the bag to execute as a kernel. Once all the slices have completed, output buffers are copied from the accelerators. Finally, each of the accelerators' clean-up routine is executed to release the memory and device(s).

This fine-grained scheduling approach ensures that the job is held up for a short time waiting (given by the slice granularity) for a single accelerator to complete execution for the last remaining slice. As seen in Figure 5, such fine-grained execution does not add significant overhead as only a small overhead is imposed by work sharing. Multiple copies of each data structure need to be allocated for each of the

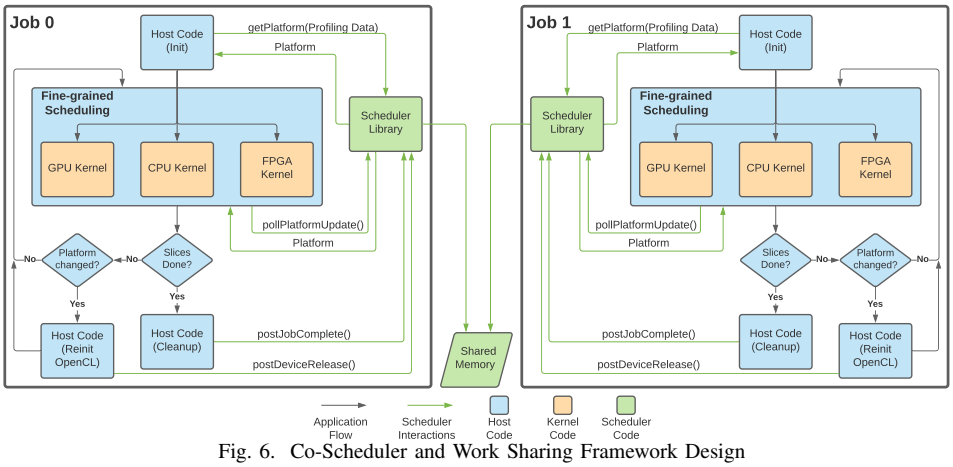


Fig. 6. Co-Scheduler and Work Sharing Framework Design

accelerators and threads, and each accelerator needs to be spawned. Furthermore, the data structures need to be copied to and from the device memory for each accelerator that participates in the work sharing adding additional overhead, just as for GPUs. Finally, for a slow device executing a particular kernel, the last slice allocated to this device might only complete execution significantly later than when the other accelerators have completed execution. This can be mitigated by increasing the number of slices (and reducing their respective size) for the kernel, which may, conversely, hamper performance on the faster devices.

To maximize performance, the above mentioned trade-off needs to be carefully assessed for each application. In this work, we show a common OpenCL kernel being executed on a CPU, a GPU and an FPGA. In general, the work scheduling framework can be used to combine multiple programming paradigms such as CUDA for GPUs or OpenMP for CPUs. Further, our system comprises of a CPU, a GPU and an FPGA, but the work sharing framework can be used for multi-GPU systems (e.g., Petascale systems such as Summit and Sierra) or even systems where GPUs with differing capabilities are present on the same node or within a cluster.

Co-Scheduling: The details of our co-scheduling framework are given in Figure 6, which shows the design of our framework with the scheduler and its interactions. Each job links to the scheduler library during compilation, and each instance of the scheduler library uses a shared block of memory to store all scheduling information. Scheduling decisions are taken whenever a new job arrives, a job switches devices, or a job completes execution. All accesses to the scheduler's critical data structure is protected using a semaphore to ensure that multiple instances of the scheduler read coherent data. While a single job can be assigned multiple accelerators, an accelerator can only be assigned to a single job.

The interactions between jobs and scheduler are as follows: 1) Before initialization of the application, the application requests an available platform using **getPlatformId**. It sends the scheduler profiled information about the application's relative performance on different combinations of devices. The scheduler either returns an available platform to the application (CPU, GPU, FPGA, CPU/GPU, etc.) or waits for a platform to become available.

2) Once the **getPlatformId** function returns indicating the assigned platform, the job starts execution as described in the previous subsection. After completing each slice, it polls the scheduler to check if a different platform has been assigned using **pollPlatformUpdate**.

3) If there is no change in the platform, the job continues with the next slice, otherwise the job waits for all of the currently executing slices to complete. Once all current slices are complete, the application executes a clean-up of the held devices before finally informing the scheduler using **postDeviceRelease**. The application can then initialize the newly assigned devices and continue execution from the next available slice.

4) Once all the slices are done, the job informs the scheduler using **postJobComplete** that it may terminate.

Switching accelerator platforms incurs a significant overhead on the application since partial output buffers need to be copied out from the current devices, the current devices need to be released, the newly assigned devices need to be initialized and input data needs to be copied to them. It is possible to mitigate some of this overhead. For example, if the scheduler assigns the GPU+FPGA to an application currently using the GPU, we do not need to release the GPU. But we run into an issue with the FPGA: If the FPGA library is loaded within an application, any use of the OpenCL library in that application would cause any other application using the FPGA to freeze due to an internal runtime lock in the vendor library (beyond our control). Hence, we load the FPGA library *conditionally* if and only if the platform returned by the scheduler includes the FPGA using **dlsym**.

While this solves our initial issue, it creates another one. An application using the GPU cannot switch to the FPGA since loading the FPGA library using **dlsym** will not properly link it to the OpenCL library, which was loaded when execution started on the GPU. Fixing this issue requires us to unload OpenCL and reload both OpenCL and the FPGA library for the application to be able to use the FPGA. Therefore, we have to release all devices and reinitialize them whenever a change in platform occurs. Notice that a fix for this issue with the FPGA library could significantly decrease platform

switching overhead, but this is beyond the scope of this work due to partially proprietary software stacks that we do not have sources of.

We implement and evaluate *four scheduling algorithms* for co-scheduling:

Baseline: This scheduling algorithm runs applications only on best performing device. While applications can be co-scheduled, they will wait for their best device to become available before being scheduled and dispatched. Priorities are assigned to the jobs based on their arrival times. A lower priority job can be scheduled before a higher priority job only if both are scheduled on different devices.

Greedy + Up Migration: This algorithm starts application execution on any device that is available. Once faster device becomes available, jobs are switched to run on a new device. Priorities are still assigned based on arrival time, but in this algorithm lower priority jobs cannot run before a higher priority job since any job can run on any device. Furthermore, preference for switching to a newly released device is given to the higher priority jobs.

Elastic: This algorithm leverages our work sharing framework in conjunction with co-scheduling. Priorities to jobs are still assigned by arrival time, but this algorithm can schedule higher priority jobs on multiple devices using work sharing. Like the previous algorithm, jobs can start on any device. As other devices become available, jobs expand to share work amongst multiple devices.

Elastic — Device Limiting: Finally, we implement Elastic-DL by modifying the Elastic algorithm. Elastic-DL is similar to Elastic with one key difference. Here, we limit the maximum number of devices that a single job can be allocated to. In our experiments, we set the limit to the number of devices given by a node.

IV. Experimental Framework

We next describe system, applications and workloads used to evaluate work sharing and co-scheduling.

Applications: We use four applications for the evaluation of our work sharing framework, which together comprise workloads to evaluate the co-scheduling algorithms.

Mandelbrot (double precision): Calculation of Mandelbrot is an important application particularly in the field of encryption and security. Several methods have been proposed to speed up parallel computation of them using HPC [18], [19]. We extend the implementation of the Mandelbrot set provided with the Intel FPGA library to support work sharing. We use the application to generate 67 frames of the set with a certain number of colors and vary the size of each frame.

GEMM (single precision): General Matrix Multiplication (GEMM) is a critical kernel for HPC systems with a wide variety of applications requiring GEMM for computation. From AI workloads to physics simulations, all rely on GEMM. For GEMM, we modify the implementation provided with the OpenCL library to support our framework and evaluate its performance for a range of different square matrix sizes using the mean of 10 runs.

SpMV (single precision): Sparse Matrix-Vector Multiplication (SpMV) is another linear algebra kernel extensively used in scientific and engineering applications and processing of large data sets. It multiplies a sparse matrix with a dense vector to produce a vector. We implement an SpMV application using the compressed sparse row (CSR) format as a sparse matrix representation. We use a constant number of non-zero elements distributed randomly across the rows for different matrix sizes during evaluations. Each configuration is executed for 10 iterations for evaluation.

XSBench (double precision): XSBench is mini-app representing the computation for a Monte Carlo neutron transport algorithm [20]. We modify an OpenCL implementation of the application provided by Argonne National Laboratory (ANL) to support work sharing and co-scheduling. Our evaluation uses the event-based simulation with 340k gridpoints and a nuclide grid search. We vary the number of look-ups to generate different application sizes and we utilize the mean of 10 iterations for each configuration.

Workloads: We have designed and implemented a workload generator to create randomized workloads using the above applications with different input sizes, where inputs are pre-distributed over all devices. Besides varying inputs, it takes the maximum inter-arrival time between jobs as a parameter. We created a total of 15 jobs shown in Table I.

TABLE I
Applications and Input Sizes

Kernel/Prec. (Size)	S	M	L	XL
M: Mandelbrot double-prec. (Frame Size NxN)	2048	4096	8192	16384
G: GEMM single-prec. (Matrix Size NxN)	4096	8192	16384	22400
S: SpMV single-prec. (Matrix Size 2^Nx2^N)	28	29	30	31
X: XSBench double-prec. (Lookups)	16M	32M	64M	-

Based on these jobs, we create three randomly generated workloads consisting of all four applications with one instance of each job by varying the maximum inter-arrival time (IAT) parameter (see Table II). We also create another workload with just two of the four applications, namely GEMM and Mandelbrot. Workloads 1, 2, 3, and 4 have inter-arrival times of 2.3s, 10.5s, 23.9s and 2.2s, respectively.

TABLE II
Workload Details wrt. Apps from Tab. I

Workload	Applications	Jobs	Avg. IAT
Workload 1	M,G,S,X	15	2.3s
Workload 2	M,G,S,X	15	10.5s
Workload 3	M,G,S,X	15	23.9s
Workload 4	M,G	12	2.2s

These applications and their parameterization provide a diverse mix of different workload characteristics. Each of these workloads are evaluated for the co-scheduling algorithms described in the previous section. GEMM and Mandelbrot kernels are part of the Intel FPGA SDK and highly optimized

for both FPGA and GPU. SpMV is optimized for the CPU and XSBench for GPUs. SpMV has the most potential for sharing and GEMM has some but Mandelbrot may not benefit much from sharing.

System Details: We run both our work sharing and co-scheduling experiments on a single node on a mid-tier HPC cluster. The system consists of an Intel multi-core CPU, an NVIDIA GPU and an Intel Altera FPGA.

The CPU is a 16 core Intel Broadwell Xeon E5-2620 v4 running at 2.10GHz with 20MB of L3 cache capable of up to 32 threads (with Hyper-Threading), 64GB of DDR4 memory and PCI-E 3.0. The GPU on the system is an NVIDIA RTX 2060 with 6GB of device memory and a memory bandwidth of 336 GB/s (theoretical peak of 6.451 TFlops single and 201.6 GFlops double precision). The FPGA is an Altera Arria 10 GX DE5a-Net-DDR4 with 8GB of device memory with PHY of up to 2666 Mbps. It is capable of 1.366 TFlops of single precision and consists of 1.5 million logic elements and DSP blocks.

On the software side, the system is running CentOS with the 4.10.13 version of a backpatched Linux kernel. Each of the three devices utilizes a different version of OpenCL: The CPU uses OpenCL 2.1 provided by Intel, the GPU exploits OpenCL 1.2 provided by NVIDIA, and the FPGA relies on OpenCL 1.0 also provided by Intel. The initialization cost of accelerators is small in all cases compared to compute times.

V. Results

This section present the results of our work sharing and co-scheduling framework for the applications and their workloads discussed in the previous section. The objective of the results is to highlight the capabilities of our work-sharing framework given a set of kernels. These kernels may be optimized to various degrees, i.e., our work is subject to the code optimization capabilities of the compilers and their OpenCL runtime systems. Instead of seeking peak performance for a device, we aim to show that our framework can flexibly adapt to *relative* performance differences between devices, irrespective of their relation to *absolute* peak performance of a given kernel (which is driven by compiler optimizations beyond the scope of this paper). The GEMM and Mandelbrot kernels are part of Intel's FPGA framework and are therefore optimized for the FPGA. The XSBench kernel is provided as open-source by Argonne National Laboratory, it is optimized for the GPU. SpMV is a CPU-optimized kernel. First, we assess results for work sharing, followed by a discussion on how we use those results to achieve efficient co-scheduling of these applications.

A. Work Sharing

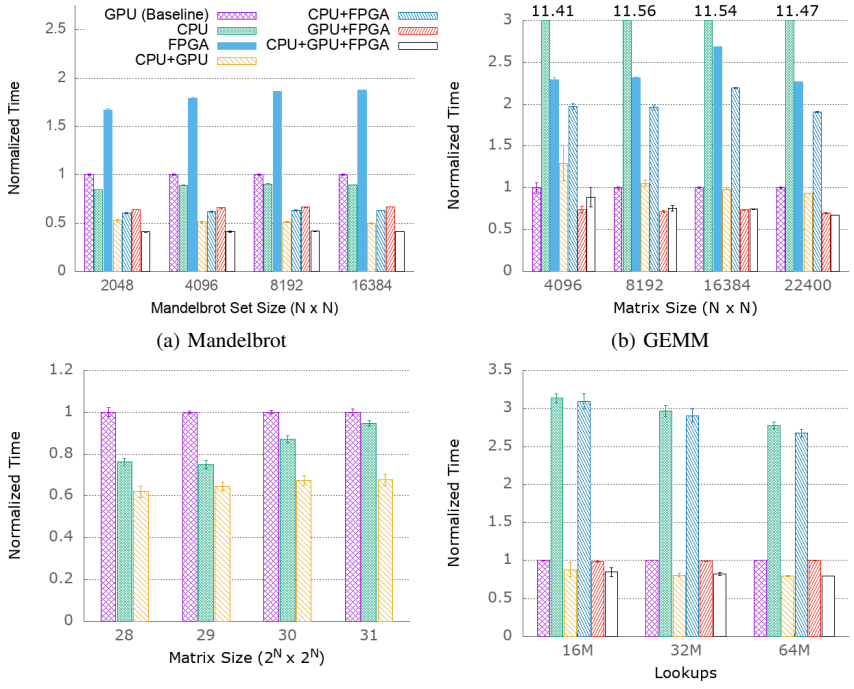
Figure 7 depicts performance in time (normalized to the GPU baseline, lower is better) per application (y-axis) for various input sizes (x-axis) with work being shared amongst combinations of devices indicated by bars in the legend. For XSBench and SpMV, we omit certain FPGA combinations from the results. This is due to the poor performance of the FPGA for these applications. These results comprise the time spent in computation including copying to and from the device but neither include the input initialization nor the OpenCL initialization times.

Mandelbrot: For Mandelbrot (Figure 7a), each of the devices has somewhat comparable performance. This results in efficient work sharing between any combination of devices. While the CPU is the best performing of the three devices, sharing work between any combination of devices yields better performance. For instance, combining GPU and FPGA (slower individual devices) yields a speed-up of 1.308 over running only on the CPU (fastest individual devices). Similarly, running on all three devices simultaneously results in a speed-up of 2.26 over running on the CPU alone.

GEMM: In the case of GEMM (Figure 7b), only the GPU and FPGA show comparable performance. The CPU performs quite poorly for this particular kernel. The GPU performs on average 2.522 better than the FPGA for GEMM, but sharing work between the GPU and FPGA results in a speed-up of 1.386 over running exclusively on the GPU. Adding the CPU into the mix results in performance either being worse that running on the GPU alone (for 4096x4096), or worse than running on the GPU and FPGA (for 8192x8192), or only marginal improvements (for 16384x16384 and 22400x22400) with a speed-up of less than 1.01 over GPU and FPGA. This is because the added overhead of scheduling slices on the CPU is not amortized by adding the compute capability of the CPU. Another reason for this is the long tail created by the last slice allocated to the CPU. While the GPU and FPGA have completed their last slices, the CPU is still working on completing the last slice causing the long tail. The comparatively better performance of the FPGA can be attributed to the fact that the application uses the DSPs on-board the FPGA to improve GEMM performance.

SpMV: Work sharing results for SpMV are shown in Figure 7c. Due to the low memory bandwidth on the FPGA, the performance of SpMV is over an order of magnitude worse on the FPGA compared to CPU and GPU, and has therefore been omitted. The CPU performs best for SpMV with an average speed-up of 1.358 over the GPU. This is because SpMV has an irregular memory access pattern that benefits the CPU more than the GPU as the latter favors regular memory access patterns. For SpMV, as seen before for other applications, sharing work between CPU and GPU results in better performance than using any single device with an average speed-up of 1.236 over the best performing device (CPU). While there has been work to improve the performance of SpMV on GPUs [21], this requires significant GPU-specific changes to the kernel and data representations. These mechanisms could be incorporated with our work-sharing framework but this is beyond the scope of the paper. The CPU performance of SpMV can also stand to gain from a better optimized kernel to improve its utilization of the system cache.

XSBench: Like SpMV, XSBench (shown in Figure 7d) also experiences poor performance on the FPGA, but unlike SpMV, the GPU performs better for this application with an average



(a) Mandelbrot (b) GEMM

(c) SpMV (d) XSBench

Fig. 7. Normalized performance over various input sizes per device mix of each application (with standard deviation).

speed-up of 2.519 over the CPU. Sharing work between the CPU and GPU results in the most significant performance improvement with a speed-up of 1.3x over the GPU. Adding the FPGA along with the CPU and GPU results in either a slight increase (for 16M) or a slight decrease (for 32M) in performance. This makes FPGA an unsuitable candidate for work sharing for XSBench.

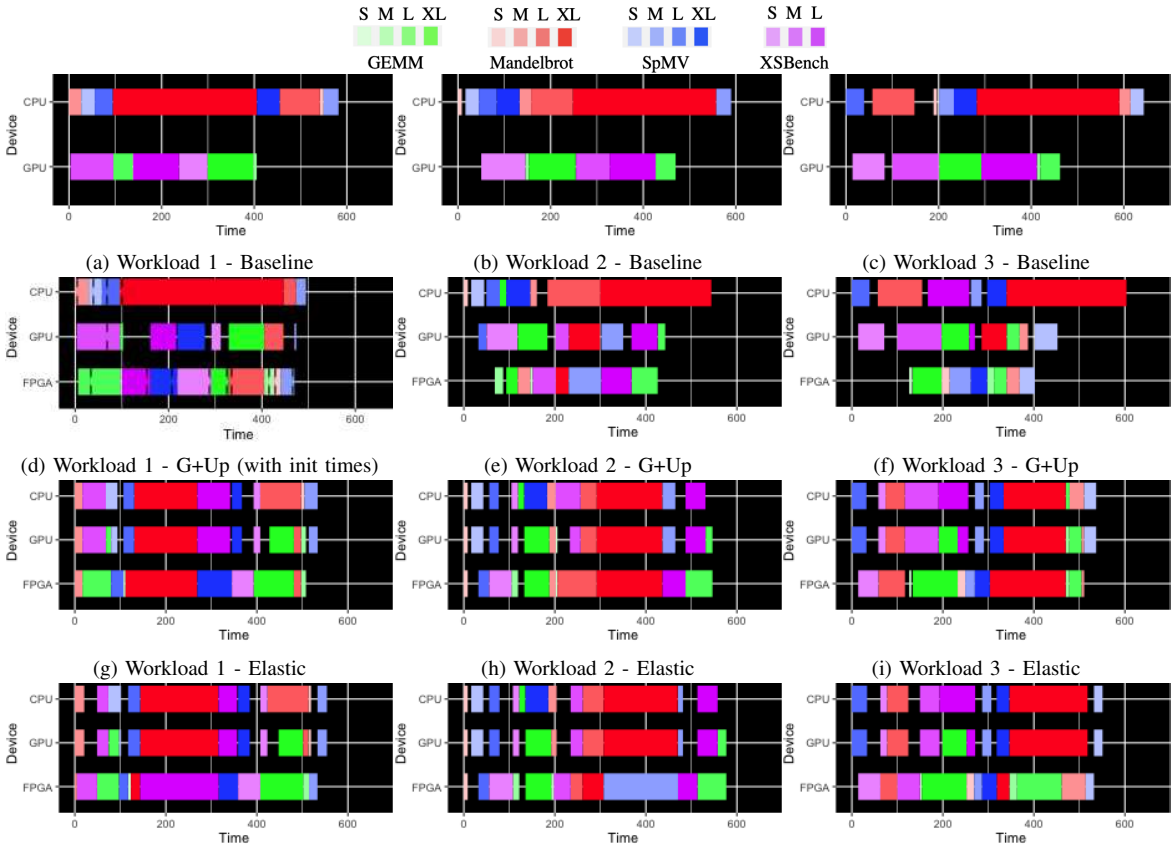
Discussion: Improvement in device technologies (e.g., higher memory bandwidths on the FPGA, unified memory between accelerators, and adding more compute units to devices) can affect the relative performance of each of the accelerators for the applications studied. The work sharing framework would be able to leverage these enhancements to further improve overall application performance. Furthermore, the framework can be used on systems with heterogeneous accelerators, e.g., multiple GPUs with different capabilities. With the current framework, the output is copied out of the device when all slices are complete. A more fine-grained approach, where the output of each slice would be copied to the other devices while the kernel is still executing, could improve performance even more, particularly for applications that rely on multiple kernel executions such as stencil computations.

Another aspect to note here is that when either the GPU or the FPGA is used along side the CPU, some of the CPU time is spent on scheduling work on the accelerators, data movement between the primary DRAM and the accelerator's DRAM.

B. Co-Scheduling

Next, we present results for co-scheduling the workloads for the previous section. These results present an end-to-end look at the work-sharing and co-scheduling mechanisms. Each of the scheduling algorithms leverages different aspects of work-sharing except the baseline, which does not use any. G+Up utilizes the ability to migrate kernels, Elastic builds on that to split work and expand to more accelerators and Elastic-DL leverages the ability to contract to fewer accelerators. Figure 8 shows the timeline per and the devices they are allocated for each of the randomly generated workloads and the scheduling algorithms. Figure 10 shows the timeline for the *dual application / 12 job* workload, where colors indicate the application while shades indicate input sizes of a given application per job. Figure 9 depicts the turn-around-time for each job per workload and scheduling algorithm. Finally, Figure 11 shows the time taken for the overall workload per scheduling algorithm. While the previous section presented results for the computation time, the scheduling results are for overall job time (in seconds).

Workload 1: Figures 8a,d,g,j and 9a depict results for Workload 1. For this workload with low inter-arrival times (2.2s on average), we see that Greedy+Up (G+Up) outperforms all other scheduling algorithms with a speed-up of 1.225 over the baseline. While both Elastic and Elastic-DL outperform the baseline (with a speed-up of 1.091 and 1.051x, respectively), they still show inferior performance when compared to G+Up. In terms of turn-around-time, we see that Elastic and Elastic-DL favor the longest and most elastic jobs to the detriment of most other jobs. We see idle time on certain devices, such as the GPU in Figure 8d in the time range of 100-150 seconds. This occurs because the scheduler has allocated the GPU to XSBench after it begins execution on the FPGA. Even though the scheduler assigns the GPU shortly after to XSBench, the job cannot switch to the newly assigned device until it either completes initialization or



(a) Workload 1 - Baseline (b) Workload 2 - Baseline (c) Workload 3 - Baseline

(d) Workload 1 - G+Up (with init times) (e) Workload 2 - G+Up (f) Workload 3 - G+Up

(g) Workload 1 - Elastic (h) Workload 2 - Elastic (i) Workload 3 - Elastic

(j) Workload 1 - Elastic-DL (k) Workload 2 - Elastic-DL (l) Workload 3 - Elastic-DL

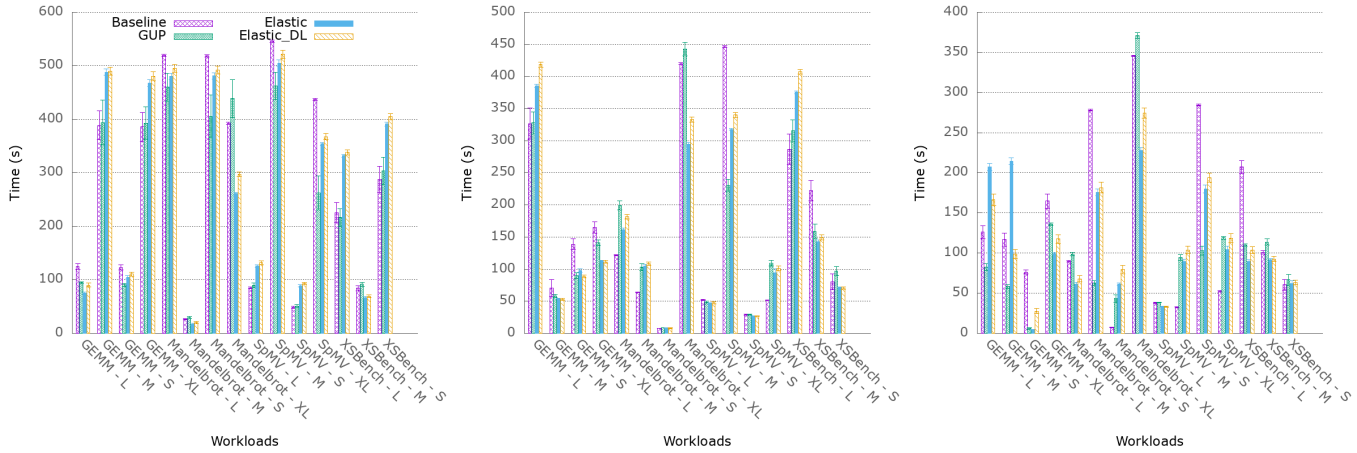
Fig. 8. Timeline for the jobs on each accelerator in the randomly generated workloads for each of the scheduling algorithms

execution of the current slice. In general, if the best resource is not available, execution on an alternate device still benefits overall performance in practice, even when considering data migration costs.

We show the timeline with initialization times for Figure 8d. The dashes in the figure show when application initialization is complete, after which the application starts its kernel execution. If application initialization has begun with a certain device allocation, it will hold the device until initialization has completed, even if another device has been assigned to the application. This is particularly visible for XSBench (large input), which starts on the FPGA as the GPU is still busy with GEMM (large), but after initialization and a single kernel run on the FPGA, XSBench is moved to the now available GPU. An earlier switch to the GPU was not possible since any single kernel cannot be preempted but rather needs to first complete. And the FPGA had already been committed to the first XSBench kernel in this case. Notice that the delay in migration is dominated by initialization cost, which is high for XSBench (large), the single FPGA kernel run contributes only insignificantly to the migration delay. For other figures, dashed lines are omitted to improve legibility, but their timeline still includes application initialization time, including migration delays, albeit none of them as significant as in Figure 8d.

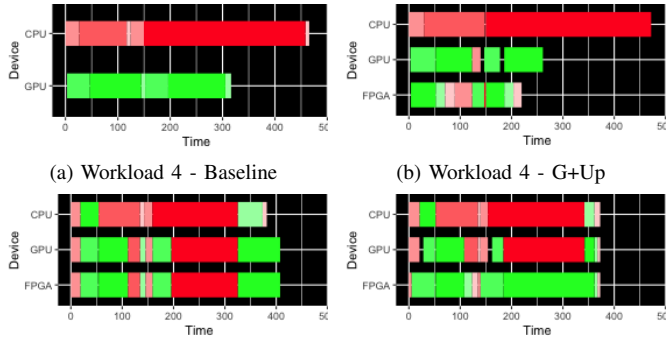
Workload 2: Figures 8b,e,h,k and 9b depict results for Workload 2 with an average inter-arrival time of 10.5s. We see that both G+Up and Elastic perform comparably with a speed-up of 1.085 and 1.08x, respectively, over the baseline. Elastic-DL experiences almost similar performance to baseline with a speed-up of 1.023 due to the overhead of switching between devices. We see a similar pattern with the turn-around-time as Workload 1, where Elastic and Elastic-DL give preference to the longest and most elastic jobs while showing an increase in the turn-around-time for most other jobs. The idle times on the GPU and FPGA at the beginning of the workload are due to there not being a job in the scheduler queue. Finally, we still see the impact of idle devices due to applications waiting for initialization and slice completion before switching devices.

Workload 3: Figures 8c,f,i,l and 9c depict results for Workload 3, which has an inter-arrival time of 23.9s. We observe that both Elastic and Elastic-DL convincingly outperform G+Up migration. Elastic and Elastic-DL have speed-ups of 1.198 and 1.167x, respectively, while G+Up has a smaller speed-up of just 1.066 over the baseline. Because of the comparatively longer inter-arrival times, G+Up migration experiences significant idle times on the FPGA. Elastic and Elastic-DL, in contrast, leverage work-sharing resulting in higher utilization of the available accelerators on the system. We see improvements in turn-around-time for more jobs compared to Workloads 1 and 2 but it is still the longer and more elastic jobs that experience a noticeable decrease in turn-around-times. Furthermore, jobs that do suffer from a negative



(a) Workload 1 (b) Workload 2 (c) Workload 3

Fig. 9. Turn around time for the jobs in the workloads for each scheduling algorithm (with standard deviation).



(a) Workload 4 - Baseline (b) Workload 4 - G+Up

(c) Workload 4 - Elastic (d) Workload 4 - Elastic-DL

Fig. 10. Timeline for jobs over accelerators in the two-application workload per scheduling algorithm, same color coding as in Fig. 8

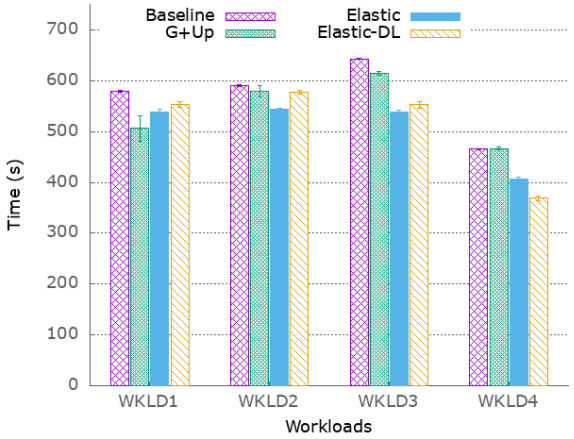


Fig. 11. Overall time for workloads per scheduling algorithms (w/ std. dev.)

impact on their turn-around-times do so to a much lesser extent compared to the previous workloads.

Workload 4: Finally, we run a workload consisting of two applications, GEMM and Mandelbrot, and a total of 12 jobs for each of our scheduling algorithms. Figure 10 shows the results for Workload 4 (average inter-arrival time of 2.25). We observe that Elastic-DL performs the best with a speed-up of 1.249 over the baseline while Elastic shows a speed-up of 1.142 over the baseline. G+Up, in contrast, shows a slight slowdown of 0.986 due to the overhead of device switching. This workload shows that applications with fine-grained slices as well as small application initialization times cause minimal idle times on devices, and therefore achieve superior performance with work-sharing even for low inter-arrival times.

Discussion: As for work-sharing, co-scheduling opens several optimization opportunities.

1) While our scheduling algorithms prioritize arrival time of the jobs when allocating accelerators to jobs, other scheduling algorithms can be developed that schedule jobs based on relative performances of applications on different accelerators. For instance, a job that arrives later can be scheduled earlier if the available accelerator is better suited for it. This can be achieved using static profiling and accelerator-specific priorities.

2) Coordination between work-sharing, data distribution and co-scheduling can further enhance performance. Instead of devices holding all kernel input data, a scheduled prefetch action could selectively populate only the input data of a slice on a device, just ahead of kernel execution but as late as possible, which would also facilitate the handling of data exceeding the accelerator's memory capacity. If the scheduler determines that the overhead of switching devices is more than running the job on the current device, it can let the application continue with the existing configuration. Furthermore, the scheduler can be made aware of the initialization time of the job. This would resolve a significant amount of the idle time that we see on devices in Figure 8.

3) Instead of offline profiling, the scheduler can estimate relative performance of jobs on different accelerators in order to make scheduling decisions. This would eliminate the need to collect profiling data for each application as well as enable applications to run on arbitrary systems.

VI. Related Work

Prior work has explored methods to efficiently use heterogeneous devices on HPC systems. Scogland et al. [22] use OpenMP-like directives to schedule computational load

across CPUs and GPUs. Aji et al. [23] schedule task-parallel workloads on devices by mapping OpenCL queues to devices at run-time to achieve ideal performance. Spafford et al. [24], Guzman et al. [25] and Pandit et al. [26] provide frameworks to orchestrate data and task decomposition for multi-device cooperative execution. Ahmed [27] implements OpenCL support for a Xilinx FPGA and demonstrates pipelined execution across FPGA, GPU and CPU with an application without kernel sharing across devices. Kim et al. [28] co-schedule over CPU/GPU devices by translating OpenCL to CUDA for GPUs but do not handle FPGAs. Al-Zoubi et al. [29] propose a predictive approach for coarse-grained OpenCL scheduling to consider both power consumption and execution time, yet no experiments are conducted for this conceptual proposal. Rodriguez et al. [30] derive a model for near-optimal chunk prediction for CPU+FPGA execution of a single kernel. Our approach goes further with its dynamic migration and elasticity in response to workload and system environment changes, readily supported by our data-oriented bag-of-task scheduling for CPU, GPU and FPGA devices.

Several techniques have been explored in prior work to make maximum use of available resources by reducing idle time. Weidendorfer et al. [31] characterize applications for their suitability for co-scheduling. Frachtenberg et al. [32] use jobs as fillers to reduce fragmentation and achieve reduced idle time. Zacarias et al. [33] create a resource manager that uses machine learning to predict the cost of co-scheduling and a scheduler that reduces performance degradation. Further, Xiong et al. [34] propose Tangram, which oversubscribes nodes resulting in CPU sharing. They use prior knowledge to determine if co-scheduling will result in overall performance improvement. Dauwe et al. [35] create a model to predict an application's execution time and energy usage when co-scheduled with other applications. They demonstrate that their model can significantly improve scheduling performance.

In contrast to these works, to the best of our knowledge, ours is the first that holistically combines work sharing and elasticity of kernels with the ability to expand, contract and migrate kernels on devices combined with co-scheduling and enabled by pluggable scheduling algorithms to coordinate the execution of multiple kernels on the same node.

VII. Conclusion

This work contributes novel methods to more effectively use the computational resources available in a current and future HPC node. We create a framework to share single kernels across several different accelerators, to migrate a kernel from one device to another, to expand the kernel to occupy more devices, or to contract the kernel to occupy fewer devices, all done dynamically. Our framework leverages these capabilities to enable sharing of a single node amongst multiple, concurrently running applications.

Evaluations with four applications composed into workloads with different input sizes show speedups of up to 2.26X when work is shared amongst all the available accelerators compared to our baseline. Furthermore, co-scheduling can achieve speedups of up to 1.25X with an Elastic-DL algorithm over baseline co-scheduling. Beyond performance, our work contributes an unprecedented ease of programmability, where a single code base readily compiles and executes under fine-grained runtime controlled scheduling with migration across three heterogeneous execution platforms.

Acknowledgments

This work was supported in part by NSF awards CISE-2316201, CISE-2217020, PHY-1818914 and a subcontract from LLNL and by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE).

References

[1] ORNL, "Summit," https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/, Oak Ridge National Laboratory, 2023.

[2] LLNL, "Sierra," https://computing.llnl.gov/computers/sierra, Lawrence Livermore National Laboratory, 2023.

[3] ORNL, "Frontier," https://www.olcf.ornl.gov/frontier/, Oak Ridge National Laboratory, 2023.

[4] NERSC, "Perlmutter," https://www.nersc.gov/systems/perlmutter/, NERSC, 2023.

[5] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman *et al.*, "Extreme heterogeneity 2018-productive computational science in the era of extreme heterogeneity: Report for doe ascr workshop on extreme heterogeneity," 2019.

[6] Amazon, "Amazon ec2 f1 instances," https://aws.amazon.com/ec2/instance-types/f1/, Amazon AWS.

[7] Microsoft, "Deploy ml models to field-programmable gate arrays (fpgas) with azure machine learning," https://learn.microsoft.com/en-us/azure/machine-learning/v1/how-to-deploy-fpga-web-service, Microsoft Azure.

[8] C. Nvidia, "Compute unified device architecture programming guide," *Nvidia*, 2007.

[9] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[10] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W. Hwu, "Gpu clusters for high-performance computing," in *2009 IEEE International Conference on Cluster Computing and Workshops*, 2009, pp. 1–8.

[11] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[12] Intel, "Intel oneapi," https://software.intel.com/en-us/oneapi, Intel, 2023.

[13] R. Dimond, S. Racaniere, and O. Pell, "Accelerating large-scale hpc applications using fpgas," in *2011 IEEE 20th Symposium on Computer Arithmetic*. IEEE, 2011, pp. 191–192.

[14] W. Vanderbauwhede and K. Benkrid, *High-performance computing using FPGAs*. Springer, 2013, vol. 3.

[15] B. Betkaoui, D. B. Thomas, and W. Luk, "Comparing performance and energy efficiency of fpgas and gpus for high productivity computing," in *2010 International Conference on Field-Programmable Technology*. IEEE, 2010, pp. 94–101.

[16] T. Nguyen, C. MacLean, M. Siracusa, D. Doerfler, N. J. Wright, and S. Williams, "Fpga-based hpc accelerators: An evaluation on performance and energy efficiency," *Concurrency and Computation: Practice and Experience*, p. e6570, 2021.

[17] Khronos, "Opencl api," https://khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/, Khronos, 2023.

[18] X. W. Duan, W. C. Shen, and J. Guo, "The mpi and openmp implementation of parallel algorithm for generating mandelbrot set," in *Applied Mechanics and Materials*, vol. 571. Trans Tech Publ, 2014, pp. 26–29.

[19] B. M. S. V. Gamage and V. M. Baskaran, "Efficient generation of mandelbrot set using message passing interface," *arXiv preprint arXiv:2007.00745*, 2020.

[20] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis," *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[21] P. Jiang, C. Hong, and G. Agrawal, "A novel data transformation and execution strategy for accelerating sparse matrix multiplication on gpus," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 376–388.

[22] T. R. Scogland, W.-c. Feng, B. Rountree, and B. R. de Supinski, "Coretsar: Adaptive worksharing for heterogeneous systems," in *International Supercomputing Conference*. Springer, 2014, pp. 172–186.

[23] A. M. Aji, A. J. Peña, P. Balaji, and W.-c. Feng, "Multicl: Enabling automatic scheduling for task-parallel workloads in opencl," *Parallel Computing*, vol. 58, pp. 37–55, 2016.

[24] K. Spafford, J. Meredith, and J. Vetter, "Maestro: data orchestration and tuning for opencl devices," in *European Conference on Parallel Processing*. Springer, 2010, pp. 275–286.

[25] M. A. D. Guzmán, R. Nozal, R. G. Tejero, M. Villarroya-Gaudó, D. S. Gracia, and J. L. Bosque, "Cooperative cpu, gpu, and fpga heterogeneous execution with enginecl," *The Journal of Supercomputing*, vol. 75, no. 3, pp. 1732–1746, 2019.

[26] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014, pp. 273–283.

[27] T. Ahmed, "Opencl framework for a cpu, gpu, and fpga platform," Master's thesis, Master's thesis, 2011.

[28] 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. New York, NY, USA: Association for Computing Machinery, 2012, p. 341–352. [Online]. Available: https://doi.org/10.1145/2304576.2304623

[29] A. Al-Zoubi, K. Tatas, and C. Kyriacou, "Towards dynamic multi-task schedulling of opencl programs on emerging cpu-gpu-fpga heterogeneous platforms: A fuzzy logic approach," in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2018, pp. 247–250.

[30] A. Rodríguez, A. Navarro, K. Nikov, J. Nunez-Yanez, R. Gran, D. Suárez Gracia, and R. Asenjo, "Lightweight asynchronous scheduling in heterogeneous reconfigurable systems," *Journal of Systems Architecture*, vol. 124, 2022.

[31] J. Weidendorfer and J. Breitbart, "Detailed characterization of hpc applications for co-scheduling," in *Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications*, 2016, p. 19.

[32] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez, "Flexible coscheduling: Mitigating load imbalance and improving utilization of heterogeneous resources," in *Proceedings International Parallel and Distributed Processing Symposium*. IEEE, 2003, pp. 10–pp.

[33] F. V. Zacarias, V. Petrucci, R. Nishtala, P. Carpenter, and D. Mossé, "Intelligent colocation of hpc workloads," *Journal of Parallel and Distributed Computing*, vol. 151, pp. 125–137, 2021.

[34] Q. Xiong, E. Ates, M. C. Herbordt, and A. K. Coskun, "Tangram: Colocating hpc applications with oversubscription," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[35] D. Dauwe, E. Jonardi, R. D. Friese, S. Pasricha, A. A. Maciejewski, D. A. Bader, and H. J. Siegel, "Hpc node performance and energy modeling with the co-location of applications," *The Journal of Supercomputing*, vol. 72, no. 12, pp. 4771–4809, 2016.