A Unified Portable and Programmable Framework for Task-Based Execution and Dynamic Resource Management on Heterogeneous Systems

Serhan Gener[1], Sahil Hassan[1], Liangliang Chang[2],
Chaitali Chakrabarti[2], Tsung-Wei Huang[3], Umit Ogras[3], Ali Akoglu[1]
[1]University of Arizona, Tucson, AZ, USA
[2]Arizona State University, Phoenix, AZ, USA
[3]University of Wisconsin at Madison, Madison, WI, USA
{gener,sahilhassan,akoglu}@arizona.edu, {tsung-wei.huang,uogras}@wisc.edu, {lchang21,chaitali}@asu.edu

ABSTRACT

Heterogeneous computing systems are essential for addressing the diverse computational needs of modern applications. However, they present a fundamental trade-off between easy programmability and performance. This paper addresses this trade-off by enabling performance and energy efficiency optimization while facilitating easy programming without delving into hardware details. It introduces CEDR-Taskflow, a comprehensive framework that automatically parallelizes user applications and dynamically schedules its tasks to heterogeneous platforms, enabling efficient resource utilization and ease of programming. Emulation-based studies on the Xilinx ZCU102 and NVIDIA Jetson AGX Xavier SoC platforms demonstrate that this integrated framework improves application execution time by up to 1.47x compared to state-of-the-art, while maintaining hardware-agnostic application development. Furthermore, this integration approach enables features such as streaming-enabled execution and schedule caching that reduce the time spent on task scheduling by up to 29.6x and results in up to 6.1x lower execution time.

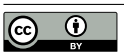
CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**.

KEYWORDS

Auto parallelization, dynamic scheduling, heterogeneous runtime

ACM Reference Format:
Serhan Gener, Sahil Hassan, Liangliang Chang, Chaitali Chakrabarti, Tsung-Wei Huang, Umit Ogras, and Ali Akoglu. 2025. A Unified Portable and Programmable Framework for Task-Based Execution and Dynamic Resource Management on Heterogeneous Systems. In *The 4th International Workshop on Extreme Heterogenei (ExHET '25), March 1–5, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3720555.3721988



ExHET '25, March 1–5, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1536-5/2025/03.
https://doi.org/10.1145/3720555.3721988

1 INTRODUCTION

Heterogeneous systems are becoming increasingly widespread and used across multiple scales, from High-Performance Computing (HPC) environments to System-on-Chip (SoC) architectures. Heterogeneity enables matching the computing needs of emerging and growing application domains with a rich set of accelerators based on architectures such as Graphical Processing Unit (GPU), Data Processing Unit (DPU), Tensor Processing Unit (TPU), and Field-Programmable Gate Array (FPGA). As heterogeneity increases, system designers face ongoing challenges in optimizing performance while ensuring programming productivity, which has motivated the development of several runtime frameworks [5, 9, 17, 20, 21, 23, 24, 28].

A prominent state-of-the-art system belonging to this class is the Compiler-integrated Extensible DSSoC Runtime [20] (CEDR), which supports productive application development with a hardware-agnostic, API-based programming model. CEDR optimizes application execution on heterogeneous platforms by parallelizing and dynamically scheduling API calls across heterogeneous processors. Although this approach maintains high programmability with API optimization, it overlooks parallelization opportunities for non-API application segments. This limitation is illustrated in Fig. 1 (top box), where a sample application modeled as Directed Acyclic Graph (DAG) with parallel Fast Fourier Transform (FFT) task nodes followed by parallel vector Multiplication (MULT) task nodes. In the programming model, the user invokes the FFT task with an API call given that the system has an FFT accelerator, and the FFT API can be deployed at runtime on either the accelerator or the CPU, depending on the availability of the resources. On the other hand, the MULT task has no accelerator support. Therefore, it belongs to the non-API region of the user code and can only be deployed on the CPU.

While CEDR parallelizes and schedules API-based FFT tasks across CPUs and accelerators, it serially executes the non-API MULT tasks on the CPU, *leaving significant performance gains untapped.* Addressing this bottleneck requires a programming model that provides the runtime with a comprehensive application view without overburdening programmers. Task Graph Computing System (TGCS) [1, 2, 14, 15] aims to address this issue by decomposing the applications into parallel tasks and structuring them as a task-dependency graph, that can be scaled across many processors.

Among various TGCSs, Taskflow [15] is a task-parallel programming model that allows users to define task dependencies and parallelize applications while enabling high programming productivity. Adopted by several academic and industrial entities [10–12, 18], Taskflow improves performance through static compile-time task parallelization. As illustrated in Fig. 1 (middle box), while Taskflow parallelizes both the API (FFT) and non-API (MULT) regions of the application, it requires users to bind each task to a specific processing element in the system at design time. However, computing systems in real-world scenarios receive evolving workloads from multiple users sharing the resources. In such cases, with the presence of other applications on the system, the static scheduling is prone to causing resource oversubscription or contention by confining parallel tasks to specific resource types, thereby leading to suboptimal execution.

Application development on heterogeneous platforms often faces a fundamental trade-off between programmability and performance, as illustrated in Fig. 1. Dynamic runtime systems like CEDR enhance programmability and runtime performance at the expense of missing out on fine-grained parallelization opportunities due to simpler programming models. On the other hand, task-based frameworks, such as Taskflow, provide a comprehensive programming model to fully expose parallelization opportunities in the application, while lacking dynamic resource management under varying workload scenarios. An ideal end-to-end system should combine the complementary advantages of these two types of framework to fully leverage the performance potential of applications through holistic application representation, task-level parallelization, and dynamic resource management for heterogeneous systems at runtime, all while maintaining ease of programming. However, this is a challenging task, as it requires a deep understanding of both CEDR and Taskflow frameworks, followed by a careful redesign of the programming model. Specifically, this effort requires system modification to implement communication protocols that ensure alignment of Taskflow's task-dependency graph with CEDR's dynamic execution model to enable seamless co-existence between the two frameworks without adding excessive processing overhead.

To address this challenge, in this work, we present our design approach on integrating Taskflow's holistic application view with CEDR's dynamic scheduling capabilities. Figure 1 (bottom box) illustrates this synergy, where Taskflow identifies parallelization opportunities for FFT and MULT tasks and provides this information to CEDR. Subsequently, CEDR dynamically schedules FFT tasks across CPUs and accelerators, in addition to now executing MULT tasks on CPUs in parallel. By combining the programmability and dynamic scheduling capability of API-based CEDR with the comprehensive task-based execution flow approach of Taskflow, the proposed framework results in more effective system resource utilization and, in turn, improves the execution time of a given application when deployed on a heterogeneous SoC without sacrificing from hardware-agnostic application development and deployment. In particular, through our integration approach, Taskflow, which has been widely adopted by the community, now gains the ability to perform dynamic scheduling on heterogeneous systems through CEDR. Additionally, this integration enables stream-based execution for applications with streaming data processing, as well as cached scheduling decisions that reduce overhead when scheduling

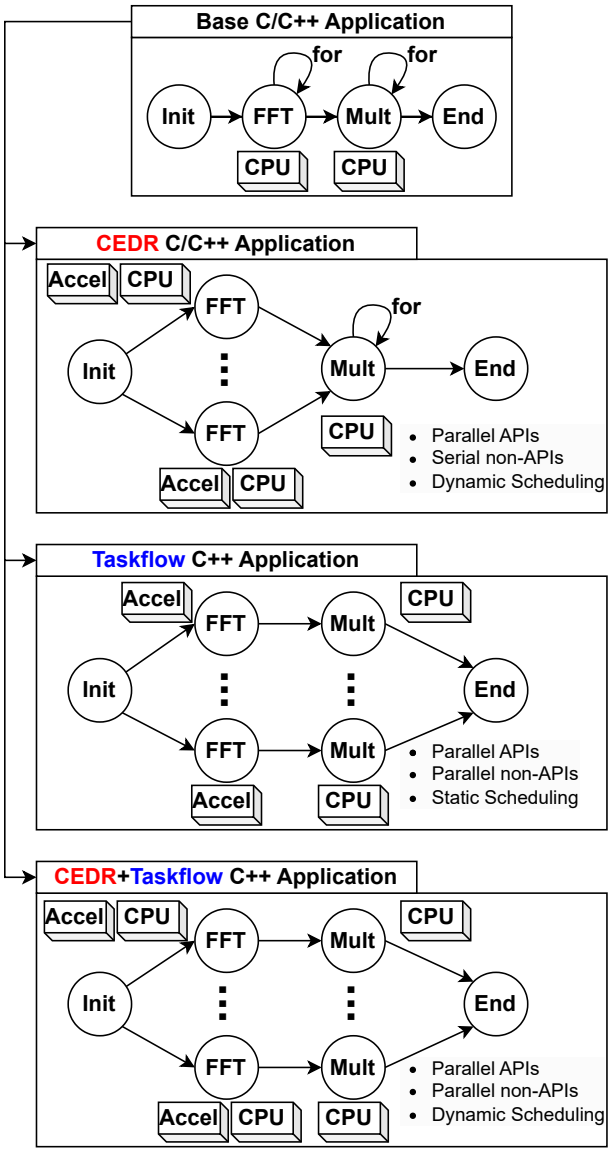


Figure 1: CEDR and Taskflow integration flow for a base C/C++ application.

tasks for repetitive application instances. Our approach enables finer control over data dependencies for streaming-based execution, and it is portable across platforms, as we demonstrate with experiments carried out over both FPGA and GPU-based SoC platforms. The contributions of this work are as follows:

- A generalizable methodology for building communication protocols that is applicable for integrating runtime systems and task-based programming frameworks.
- A runtime integrated task-level programming framework that is portable on any given commercial off-the-shelf SoC platform with a heterogeneous set of compute resources.
- A robust framework that allows hardware-agnostic application development and deployment with parallelism from

task-to-application levels on heterogeneous systems, while balancing ease of programmability, dynamic resource management, and performance.
- Demonstrate ability to resolve limitations of task-level programming framework by replacing static scheduling and single application at a time-based execution with dynamic resource management across multiple application instances executing on heterogeneous systems.

2 BACKGROUND

A runtime system with an integrated scheduler is needed to manage the task-to-processing element (PE) mapping decisions at runtime for dynamically arriving workload scenarios across a diverse set of PEs. Several runtime systems have been proposed [3, 5, 7, 9, 13, 17, 20, 21, 23, 24, 28] for dynamic resource management to maximize performance while exploiting the heterogeneity at PE-level based on the state of the system resources and continuously changing workload scenarios. The features and ease of usability of runtime systems play a crucial role in their broader adoption. Programming models used by runtime systems can be broadly classified into two categories. The first is an application programming interface (API) based approach [17, 20], which focuses on ease of programmability, where developers call predefined functions to execute tasks. This method simplifies the application development but often limits control over task dependencies and execution order. The second category involves decomposing an application into independent tasks represented as DAGs [9, 21] and allows exposing full transparency into the task dependencies and the dataflow. While this programming model provides the runtime system with a global view of the entire application and enables greater flexibility and optimization potential, it requires developers to define task dependencies and manage execution flows manually. This explicit control flow management places a significant burden on the programmer and limits the system's ability to execute applications with dynamic control-flow.

Taskflow [15] is an open-source parallel programming model that addresses these issues by automatically constructing a task dependency graph and expressing task-based parallelism within the application. Unlike traditional task-based approaches [1, 2, 4, 6, 8, 16], the lightweight runtime of the Taskflow enables execution of these tasks on the designated resources by scaling parallel execution across numerous processors. While Taskflow enables exploiting parallelism in the application, the runtime relies on static task-to-PE mapping decisions made by the application developer. The lack of dynamic runtime scheduling and resource management limits its ability to optimize performance under varying workload conditions on heterogeneous platforms. This is a potential performance concern in scenarios where a common tasks across multiple user applications is mapped onto one type of PE, which inevitably results with oversubscription of that resource type while others may be available to execute the same task. In this study, we integrate Taskflow with a runtime system such that, tasks that are suitable for execution on multiple PE types (e.g., accelerator and CPU types) are scheduled dynamically.

While there are multiple approaches that utilize API-based and DAG-based programming models for runtime systems, in this work, we integrate Taskflow with the open-source Compiler-integrated, Extensible, DSSoC Runtime (CEDR) [20, 21]. CEDR is one of the unique runtime systems originally introduced with DAG-based [21] and later API-based [20] programming model for the users. Through its hardware-agnostic API-based programming model, CEDR allows users to seamlessly develop, compile, and deploy applications on off-the-shelf heterogeneous computing platforms. Importantly, this framework is portable across a wide range of Linux-based systems, ensuring that effort to migrate across systems is minimal. Additionally, CEDR has extended support and compatibility across programming models (GNURadio [19], PyTorch [27]) and architectures (RISC-V [26], FPGA, GPU, and ARM-based SoCs [21, 22]), making it a highly versatile runtime for heterogeneous computing. In the following section, we present our integration approach.

Table 1: Overview of key components and APIs

Position	Name	Type	Description
Application	*CEDR_∗*	API	CEDR APIs for task execution
	for_each_index	API	Taskflow API for parallel loop
	emplace	API	Taskflow API for standalone single tasks
Taskflow	*tf::Taskflow*	Class	Taskflow semantic for DAG
	tf::Executor	Class	Taskflow semantic for executing the DAG
CEDR	*CEDR_DAG_EXTRACT*	New API	CEDR API for extracting the DAG from tf::Taskflow
	CEDR_RUN_DAG	New API	CEDR API for executing the DAG from tf::Taskflow
	cedr_task_config	*struct* Variable	Optional CEDR variable storing configuration information for CEDR APIs

3 CEDR-TASKFLOW INTEGRATION

Since both CEDR and Taskflow rely on header-based includes for an application to access their API calls, integration of the two, from an application's perspective, starts with the inclusion of both headers to the application file. However, since the integration approach does not end with just using the existing APIs, additional steps are required to enable CEDR and Taskflow to communicate with each other, such as transferring the Taskflow-generated DAG to the CEDR for runtime processing. Table 1 provides an overview of relevant components and APIs from CEDR and Taskflow that are leveraged or newly introduced during the integration of these frameworks. Utilizing these components, Fig. 1 illustrates the integration flow of CEDR and Taskflow into a base C/C++ application, highlighting the main benefits of the integration approach and the combination of both frameworks into a single application. Applications integrated with CEDR and Taskflow are available online.

3.1 Application Preparation

Starting with a base application that does not utilize either CEDR or Taskflow, Listing 1 illustrates a simple C/C++ program executing 512 128-point FFTs, referred to as *FFT for loop* or API region, followed by 512 128-pt vector multiplications, called *Multiplication for*

```
1
2
3  ...
4  int start=0,end=512,size=128;
5  bool forward=true;
6  complex input=allocate(512);
7  complex output=allocate(512);
8  // FFT for loop
9
10
11
12 for (int i=start; i<end; i++){
13   FFT(input[i],
14     output[i],
15     size,
16     forward);
17 }
18 // Multiplication for loop
19
20
21
22 for (int i=start; i<end; i++){
23   for (int j=0; j<size; j++){
24     output[i][j] =
           output[i][j] * 2;
25   }
26 }
27 ...
28 deallocate(input);
29 deallocate(output);
```

Listing 1: Base C/C++ Code

```
1  #include <libcedr.h>
2
3  ...
4  int start=0,end=512,size=128;
5  bool forward=true;
6  complex input=allocate(512);
7  complex output=allocate(512);
8  // FFT for loop
9
10
11
12 for (int i=start; i<end; i++){
13   CEDR_FFT(input[i],
14     output[i],
15     size,
16     forward);
17 }
18 // Multiplication for loop
19
20
21
22 for (int i=start; i<end; i++){
23   for (int j=0; j<size; j++){
24     output[i][j] =
           output[i][j] * 2;
25   }
26 }
27 ...
28 deallocate(input);
29 deallocate(output);
```

Listing 2: C/C++ Code Using CEDR

```
1  #include <libcedr.h>
2  #include <taskflow.hpp>
3  ...
4  int start=0,end=512,size=128;
5  bool forward=true;
6  complex input=allocate(512);
7  complex output=allocate(512);
8  // FFT for loop
9  task0=taskflow.for_each_index(
10   ref(start), ref(end), 1,
11   [input, &output,
12   size, forward])(int i){
13   CEDR_FFT(input[i],
14     output[i],
15     size,
16     forward);
17 }
18 // Multiplication for loop
19 task1=taskflow.for_each_index(
20   ref(start), ref(end), 1,
21   [&output,
22   size])(int i){
23   for (int j=0; j<size; j++){
24     output[i][j] =
           output[i][j] * 2;
25   }
26 }
27 ...
28 deallocate(input);
29 deallocate(output);
```

Listing 3: C++ Code Using Taskflow

loop or non-API region. To integrate CEDR, as shown in Listing 2, the `libcedr.h` header is added on line 1 of the base application, followed by a modification on line 13 where the *FFT* function is replaced with CEDR's hardware-agnostic wrapper function for FFT, *CEDR_FFT*, API call in the C/C++ source code. The next step involves combining Taskflow, demonstrated in Listing 3. Here, the `taskflow.hpp` C++ header is included on line 2, and the `for` loop is replaced by Taskflow's *for_each_index* construct, which parallelizes the inner loop based on the `start`, `end`, and increment count (set to 1 in this case) on line 10. This modification uses `i` as the loop index, defined on line 12. Furthermore, a lambda function captures the necessary variables, along with those requiring modification (e.g., `output`) captured by reference, as shown in lines 11 and 12. The *FFT for loop* section of the code (lines 8 to 17) is modified using both CEDR and Taskflow APIs. In contrast, the *Multiplication for loop*, a non-API section (lines 22-26 in Listing 1), is modified only when Taskflow is added to parallelize this part of the application. The modification applied to the loop on line 12 is similarly applied to the loop on line 22, enabling Taskflow to parallelize the non-API region. In addition to the above-mentioned scenario, any standalone task that is not a `for` loop-based parallelization candidate can be created using the *emplace* construct instead of *for_each_index*. Along with the creation of tasks, Taskflow requires an initialization of a class *tf::Executor*, which manages the execution of the generated task flow graph (presented by class *tf::Taskflow*). A task flow graph is constructed by combining tasks, such as *task0* and *task1* on lines 9 and 19 of Listing 3, based on their successor and predecessor relationships. Once the graph is ready, it is passed to the *tf::Executor* that executes it starting from the head node. While the Taskflow library offers many additional features and APIs, the ones described here represent the core functionality utilized in this work.

In this setup, since the CEDR's APIs are wrapped within Taskflow APIs when the *tf::Executor* starts the execution of the flow graph, if a task node using a CEDR API is executed, the API call is sent to CEDR for scheduling and execution on the appropriate PE based on the scheduling results. This dynamic runtime approach eliminates the need for users to assign tasks statically to specific PEs during graph creation. With this integration, applications continue to leverage the benefits of CEDR, effectively utilizing the resources of a heterogeneous system. During the execution of the task flow graph, Taskflow's *tf::Executor* manages the assignment of threads to physical CPU cores, while CEDR handles workload management across PEs, including CPUs, FPGA-based accelerators, and GPUs, for tasks involving CEDR API calls.

3.2 CEDR and Taskflow Communication

Now that we have an approach for integrating both CEDR and Taskflow APIs into an application, the next step is enabling CEDR to utilize the DAG created by Taskflow. To implement this process, we introduce *CEDR_DAG_EXTRACT* API call in CEDR that allows Taskflow to communicate the generated DAG to CEDR. This API accepts two input arguments: (1) the Taskflow generated graph (*tf::Taskflow*), and (2) a map containing configuration information on the graph tasks. This optional configuration (*cedr_task_config*) allows users to incorporate any pre-scheduling decision for an API if desired. CEDR can use this information to execute task nodes containing CEDR API calls. Being called, the *CEDR_DAG_EXTRACT* API converts the task flow into a DAG representation native to DAG-based CEDR [21], and performs initial scheduling for CEDR API tasks in the DAG. If a single application runs on the system, users can directly utilize this scheduling by passing the *cedr_task_config* argument to the CEDR APIs. Otherwise, scheduling will proceed as it usually does within the system; each CEDR API is scheduled when the running application thread invokes the API call.

Once the DAG is extracted using *CEDR_DAG_EXTRACT*, it can be executed with *tf::Executor* as typically done with Taskflow from the application side. However, in scenarios with multiple concurrent applications running on the system, each application will create its own *tf::Executor* instance that will have a local view of CPU threads present on the system belonging to its parent application, assuming it is the exclusive manager in the system. As independent executors lack a global view of CPU workloads on the system, this can reduce the execution efficiency of the overall system. Therefore, despite multiple executors being functionally supported, using a single centralized executor with a global view of the system is more effective in managing CPU workloads. To address this, we introduced a new API, *CEDR_RUN_DAG*, which accepts the task flow graph (*tf::Taskflow*) and a repetition count indicating the number of times the graph should be executed. This API allows CEDR to take control of multiple concurrent task flow graphs and execute them using a single *tf::Executor* instance managed centrally within the runtime system rather than requiring each application to maintain its own executor. Additionally, the use of repetition count enables the graph to be executed continuously in a streaming manner without repeating the redundant and expensive graph initialization process. By centralizing the *tf::Executor*, it can make more informed CPU workload distribution, improving efficiency across multiple applications and workloads.

3.3 Broader Applicability

While integrating CEDR and Taskflow demonstrates a specific implementation, the methodology outlined in this work has broader implications for designing and utilizing runtime systems in heterogeneous computing environments. This integration's principles are not inherently limited to CEDR or Taskflow; they can be generalized and adapted to other runtime systems and task-based programming frameworks, provided the developer has sufficient understanding of their integration requirements and underlying architectures. For instance, the communication mechanism for transferring task flow graphs or DAGs between frameworks could be adapted for other runtime systems. Similarly, an API-focused runtime like IRIS [17] could integrate with a DAG-oriented library to provide greater flexibility and optimization opportunities for complex applications. While this work focuses on CEDR and Taskflow, the principles and mechanisms presented here lay a foundation for enhancing resource management and interoperability across various runtime systems.

4 EXPERIMENTAL SETUP

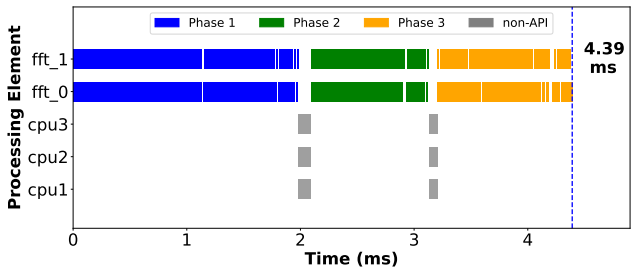
In this work, we utilize the Xilinx Zynq Ultrascale+ development board (ZCU102) [29] to emulate heterogeneous architectures. The ZCU102 hosts four ARM-based hard CPU cores. We implemented five HW accelerators using its programmable logic fabric: (1) two Fast Fourier Transform (FFT) accelerators, supporting up to 2048-point FFTs, (2) two general matrix multiplication (GEMM) accelerators, and (3) one point-wise vector operation (ZIP) accelerator. The FFT accelerator is generated using the Xilinx FFT IP, while GEMM and ZIP accelerators are implemented using High-Level Synthesis (HLS). The CPUs operate at 1.2 GHz, while all accelerators run at 300 MHz. To demonstrate the portability of our approach, we also performed experiments on the NVIDIA Jetson Xavier AGX platform (Jetson) [25]. This platform consists of a 512-core Volta GPU running at 1.3 GHz and is capable of accelerating FFT, GEMM, and ZIP functions, in addition to having an eight-core ARM-based CPU running at 2.3 GHz.

We generate workloads based on benchmark applications from the signal processing domain that include Radar Correlator (RC), Temporal Mitigation (TM), WiFi-TX, Pulse Doppler (PD), and Synthetic Aperture Radar (SAR). These applications provide diverse workloads for evaluating CEDR and Taskflow integration. RC processes radar pulses to measure distances using three 256-point FFTs at a rate of 1,000 samples per second. In RC, the main execution path of the control flow graph consists of two FFTs, followed by spectral correlation, which is then fed into an inverse FFT (IFFT). TM focuses on successive interference cancellation of low-energy radar signals mixed with high-energy communication data through GEMM and ZIP-based operations. WiFi-TX implements a WiFi transmission chain that involves a 128-point IFFT per packet for 10 packets. PD consists of three phases of 256, 128, and 128 parallel 128-point FFTs, used to estimate the distance and velocity of objects by analyzing frequency shifts in radar pulses. SAR is centered on 3D landscape reconstruction, employing 1,537 FFTs and 768 ZIP operations, with substantial parallelism across two primary processing stages.

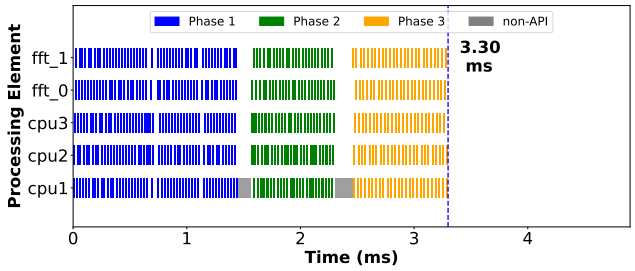
5 EXPERIMENTAL EVALUATIONS

5.1 Performance Analysis

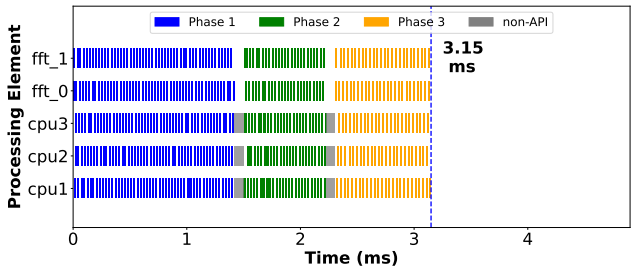
Figure 2 shows the makespan for a single instance of the PD over a system composed of 3 CPU cores and 2 FFT accelerators emulated on the ZCU102 based on the compilation and deployment using only Taskflow (a), only CEDR (b) and finally CEDR and Taskflow integrated setup (c). The makespan of the Gantt charts represents the time from the start of the first API call to the end of the last API call, excluding any memory initialization, cleanup, allocation, or deallocation phases of the application. Taskflow parallelizes all FFT APIs across the FFT accelerators and all non-API regions in the application across the CPU cores, as shown in Fig. 2 (a). However, even though CPU cores are available to support FFT



(a) Taskflow application



(b) CEDR application



(c) CEDR-Taskflow application

Figure 2: Single instance of PD running on ZCU102 with (a) CEDR-only implementation, (b) Taskflow-only implementation, and (c) both CEDR and Taskflow-based implementation.

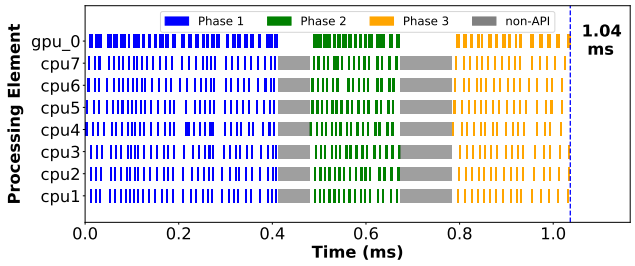


Figure 3: Single instance of PD running on Jetson with both CEDR and Taskflow-based implementation.

execution, they remain idle since APIs are statically assigned to the FFT accelerator. The contention on the FFT accelerators due to the FFT-intensive PD application combined with under-utilization of the CPU cores results in performance loss. Figure 2 (b) shows API to PE assignments with CEDR alone. While CEDR dynamically schedules tasks across all available resources, the non-API regions

Table 2: Execution time comparison when applications are deployed as a single instance through CEDR only, Taskflow only, and CEDR-Taskflow Integrated setup on ZCU102.

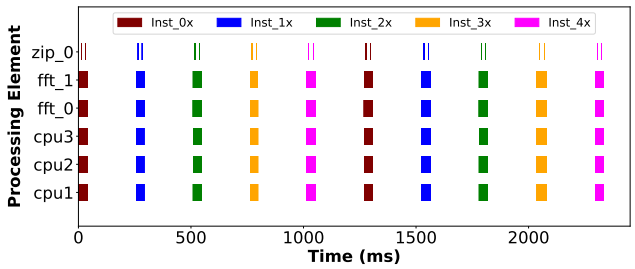
App Name	Taskflow only (ns)	CEDR only (ns)	CEDR and Taskflow (ns)
RC	120,972	120,612	120,162
TM	2,597,770	2,575,658	1,762,166
WiFi-TX	714,621	712,721	651,685
PD	5,144,564	3,868,427	3,790,988
SAR	37,351,722	38,111,968	28,980,316

are serialized onto a single CPU core. While CEDR does not support parallelization of the non-API regions, with its dynamic scheduling ability, FFT tasks are distributed across the accelerators and CPU cores and, in turn, reduces the makespan from 4.39 ms to 3.30 ms. Finally, Fig. 2 (c) demonstrates the benefit of integrating Taskflow with CEDR, where CEDR dynamically schedules APIs across PEs, and Taskflow parallelizes non-API regions. Overall, the makespan of the PD reduces from 4.39 ms to 3.15 ms.

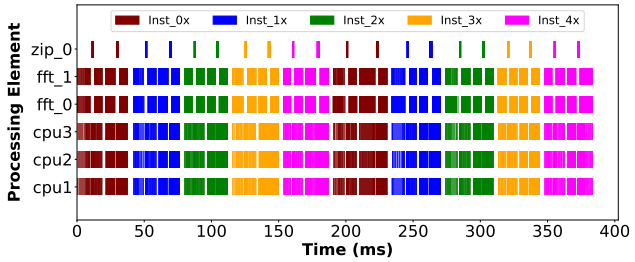
Table 2 shows the execution time for a single instance of each benchmark application when deployed on the ZCU102 platform based on Taskflow only, CEDR only, CEDR-Taskflow integrated implementations, using Earliest Finish Time (EFT) as CEDR's scheduler. These results include full application execution, from start to finish, including memory allocation, initialization, and deallocation. The combined implementation demonstrates performance improvements for all applications. For instance, the TM application shows a notable speedup of 1.47x compared to Taskflow-only and 1.46x compared to CEDR-only. In contrast, the RC application, with limited parallelization opportunities on API and non-API regions, shows only a minimal improvement, indicating that the benefit of the combined approach is negligible for such tasks. The PD application, heavy in the API region, benefits from a 1.36x improvement over Taskflow-only and a 1.02x improvement over CEDR-only. The WiFi-TX application achieves a 1.09x speedup with the combined approach compared to both Taskflow-only and CEDR-only versions. The SAR application, which features a mix of long API and non-API regions, benefits from the combination of CEDR and Taskflow with improvements of 1.28x compared to Taskflow-only and 1.31x compared to CEDR-only. These results highlight that the effectiveness of combining CEDR and Taskflow depends on the application characteristics, particularly the balance between API and non-API regions. Overall, the combined implementation consistently outperforms the standalone versions, with a speedup of up to 1.47x.

5.2 Portability

The CEDR-Taskflow integration is portable across SoC platforms. Figure 3 shows the execution of the PD application on the Jetson platform *without any modifications to the application code* that was used in the generation of the Gantt chart shown in Fig. 2 (c). This seamless portability is enabled by Taskflow's adaptive parallelism, which automatically leverages the available CPUs on the Jetson platform. At the same time, CEDR continues to manage dynamic



(a) SAR processing 10 inputs as application instances



(b) SAR processing 10 inputs in a streaming manner

Figure 4: Ten SAR instances running back to back by (a) direct injections from CEDR, (b) repeating the task flow graph. Labels Nx show instances N and N+5.

resource management for APIs, demonstrating its robustness across different hardware configurations. Since the Jetson platform has more cores than ZCU102, the non-API regions of the PD scale automatically to utilize all the CPU cores. Furthermore, whenever a CPU core is available, the FFT tasks are distributed across the CPU cores. As a result, the execution time on the Jetson platform reduces to 1.04 ms compared to the 3.15 ms observed on the ZCU 102 platform. This result underscores the flexibility of our integration approach, enabling efficient task scheduling and resource utilization across heterogeneous platforms without additional developer effort.

5.3 Features Enabled by CEDR-Taskflow Integration

5.3.1 Streaming Input Processing. Figure 4 (a) shows the execution of 10 SAR instances by repeatedly running SAR one after another using the CEDR-Taskflow integrated setup. Each color coded region corresponds to a single instance of the SAR when executed on a system with 3 CPU cores, 2 FFT accelerators, and 1 ZIP accelerator emulated on the ZCU 102 board. The white space between each SAR instance corresponds to the time spent on repeated allocation, deallocation, and initialization activities. By leveraging the repeat functionality of the Taskflow integrated CEDR, we observe that the gap between each SAR instance closes significantly because the application can process different inputs without spending time on setting up the context, as shown in Fig. 4 (b). This, in turn, reduces the makespan of the workload from 2,334.54 ms to 383.42 ms.

5.3.2 Cached Scheduling. The streaming-enabled execution paves the way for reusing the scheduling decisions made for one instance of the user application for the subsequent instances by simply caching the task-to-PE mapping decisions. We examine the time spent on scheduling decisions throughout the execution of each benchmark application based on streaming-based execution without and with schedule caching mechanism. In this experiment, each application is repeated 1,000 times using the EFT scheduler, and the results are summarized in Table 3. The API Count column indicates the number of APIs in each application, the Stream column presents the total scheduling time in the μs scale, and the Cached column shows the total time in μs scale for extracting the DAG and scheduling the APIs. With cached scheduling, we observe significantly reduced scheduling overhead across all applications. As shown in Table 3, API intensive applications benefit more from this approach, indicating that using cached decisions for applications with a high API count and leveraging a streaming approach for multiple inputs provides better performance improvements.

Table 3: Time spent on scheduling for each application when repeated 1,000 times on the ZCU102 platform

App Name	API Count	Stream (μs)	Cached (μs)	Improvement
RC	3	2,376	283	8.37x
TM	5	3,759	643	5.84x
WiFi-TX	10	7,662	723	10.59x
PD	512	291,790	10,769	27.09x
SAR	2,305	1,405,034	47,475	29.60x

6 CONCLUSION

The integration of CEDR and Taskflow presents a portable framework for addressing the dual challenges of performance optimization and ease of programmability in heterogeneous systems. The proposed system enables hardware-agnostic application development and achieves higher resource utilization by leveraging Taskflow's ability to define task dependencies and CEDR's dynamic scheduling capabilities. Hence, it consistently decreases the execution time without additional complexity for application developers. Experimental evaluations highlight its applicability across diverse platforms and workloads, showcasing efficient resource management and reduced execution latency. Future work will explore automated DAG generation to ease further development efforts and automatic pipelined execution for applications, merging DAGs of multiple running applications to gather a holistic view of all the DAGs running on the system. This study sets a foundational approach for advancing runtime frameworks in heterogeneous computing environments.

ACKNOWLEDGMENTS

This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7860. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those

of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL and DARPA or the U.S. Government.

We appreciate the continuous and generous support from the AMD University Program, including the donation of FPGA prototyping board used in this work.

Dr. Akoglu and Dr. Ogras have disclosed an outside interest in DASH Tech IC to the University of Arizona and University of Wisconsin, respectively. Conflicts of interest resulting from this interest are being managed by the respective universities in accordance with their policies.

REFERENCES

[1] 2021. Intel oneTBB. https://github.com/oneapi-src/oneTBB. [Online; accessed November 18, 2024].

[2] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2017. Fastflow: High-Level and Efficient Streaming on Multicore. *Programming multi-core and many-core computing systems* (2017), 261–280. https://doi.org/10.1002/9781119332015.ch13

[3] Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla. 2012. A compiler and runtime for heterogeneous computing. In *DAC Design Automation Conference 2012*. 271–276. https://doi.org/10.1145/2228360.2228411

[4] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2012.71

[5] Cristiana Bolchini, Stefano Cherubin, Gianluca C. Durelli, Simone Libutti, Antonio Miele, and Marco D. Santambrogio. 2018. A runtime controller for openCL applications on heterogeneous system architectures. *SIGBED Rev.* 15, 1 (mar 2018), 29–35. https://doi.org/10.1145/3199610.3199614

[6] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J. Dongarra. 2013. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45. https://doi.org/10.1109/MCSE.2013.98

[7] Jani Boutellier, Jiahao Wu, Heikki Huttunen, and Shuvra S. Bhattacharyya. 2018. PRUNE: Dynamic and Decidable Dataflow for Signal Processing on Heterogeneous Platforms. *IEEE Transactions on Signal Processing* 66, 3 (2018), 654–665. https://doi.org/10.1109/TSP.2017.2773424

[8] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[9] Georgios Christodoulis, François Broquedis, Olivier Muller, Manuel Selva, and Frédéric Desprez. 2018. An FPGA target for the StarPU heterogeneous runtime system. In *2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. 1–8. https://doi.org/10.1109/ReCoSoC.2018.8449373

[10] Guannan Guo, Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2020. An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints. In *ACM/IEEE Design Automation Conference (DAC)*.

[11] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Path-based Timing Analysis. In *IEEE/ACM Design Automation Conference (DAC)*.

[12] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-accelerated static timing analysis. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Article 147, 9 pages.

[13] Chenying Hsieh, Ardalan Amiri Sani, and Nikil Dutt. 2019. SURF: Self-aware Unified Runtime Framework for Parallel Programs on Heterogeneous Mobile Architectures. In *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*. 136–141. https://doi.org/10.1109/VLSI-SoC.2019.8920374

[14] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[15] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Transactions on Parallel and Distributed Systems* 33, 6 (2022), 1303–1320. https://doi.org/10.1109/TPDS.2021.3104255

[16] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. 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* (Eugene, OR, USA) *(PGAS '14)*. Association for Computing Machinery, New York, NY, USA, Article 6, 11 pages. https://doi.org/10.1145/2676870.2676883

[17] Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S. Vetter. 2024. IRIS: A Performance-Portable Framework for Cross-Platform Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems* 35, 10 (2024), 1796–1809. https://doi.org/10.1109/TPDS.2024.3429010

[18] Wan Luan Lee, Dian-Lun Lin, Tsung-Wei Huang, Shui Jiang, Tsung-Yi Ho, Yibo Lin, and Bei Yu. 2024. G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner. In *ACM/IEEE Design Automation Conference (DAC)*.

[19] Joshua Mack, Serhan Gener, Ali Akoglu, Jacob Holtom, Alex Chiriyath, Chaitali Chakrabarti, Daniel Bliss, Anish Krishnakumar, Alper Goksoy, and Umit Ogras. 2022. GNU Radio and CEDR: Runtime Scheduling to Heterogeneous Accelerators. In *Proceedings of the GNU Radio Conference*, Vol. 7.

[20] Joshua Mack, Serhan Gener, Sahil Hassan, H. Umut Suluhan, and Ali Akoglu. 2023. CEDR-API: Productive, Performant Programming of Domain-Specific Embedded Systems. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 16–25. https://doi.org/10.1109/IPDPSW59300.2023.00016

[21] Joshua Mack, Sahil Hassan, Nirmal Kumbhare, Miguel Castro Gonzalez, and Ali Akoglu. 2023. CEDR: A Compiler-integrated, Extensible DSSoC Runtime. *ACM Trans. Embed. Comput. Syst.* 22, 2, Article 36 (jan 2023), 34 pages. https://doi.org/10.1145/3529257

[22] Joshua Mack, Nirmal Kumbhare, Anish NK, Umit Y. Ogras, and Ali Akoglu. 2020. User-Space Emulation Framework for Domain-Specific SoC Design. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 44–53. https://doi.org/10.1109/IPDPSW50202.2020.00016

[23] Kasra Moazzemi, Biswadip Maity, Saehanseul Yi, Amir M. Rahmani, and Nikil Dutt. 2019. HESSLE-FREE: Heterogeneous Systems Leveraging Fuzzy Control for Runtime Resource Management. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 74 (oct 2019), 19 pages. https://doi.org/10.1145/3358203

[24] Raúl Nozal, Jose Luis Bosque, and Ramon Beivide. 2020. EngineCL: Usability and Performance in Heterogeneous Computing. *Future Generation Computer Systems* 107 (2020), 522–537. https://doi.org/10.1016/j.future.2020.02.016

[25] Nvidia AGX [n. d.]. *Jetson AGX Xavier Evaluation Board*. Retrieved September 06, 2024 from https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/

[26] H. Umut Suluhan, Serhan Gener, Alexander Fusco, Joshua Mack, Ismet Dagli, Mehmet Belviranli, Cagatay Edemen, and Ali Akoglu. 2024. A Runtime Manager Integrated Emulation Environment for Heterogeneous SoC Design with RISC-V Cores. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 23–30. https://doi.org/10.1109/IPDPSW63119.2024.00013

[27] H. Umut Suluhan, Serhan Gener, Alexander Fusco, H. Fatih Ugurdag, and Ali Akoglu. 2023. PyTorch and CEDR: Enabling Deployment of Machine Learning Models on Heterogeneous Computing Systems. In *2023 20th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA)*. 1–8. https://doi.org/10.1109/AICCSA59173.2023.10479315

[28] Xubin Tan, Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé, and Mateo Valero. 2019. A Hardware Runtime for Task-Based Programming Models. *IEEE Transactions on Parallel and Distributed Systems* 30, 9 (2019), 1932–1946. https://doi.org/10.1109/TPDS.2019.2907493

[29] Xilinx ZCU102 [n. d.]. *ZCU102 Evaluation Board*. Retrieved September 06, 2024 from https://docs.amd.com/v/u/en-US/ug1182-zcu102-eval-bd