Fork is All You Need in Heterogeneous Systems

Zixuan Wang
zxwang@ucsd.edu
University of California, San Diego

Jishen Zhao
jzhao@ucsd.edu
University of California, San Diego

ABSTRACT

We present a unified programming model for heterogeneous computing systems. Such systems integrate multiple computing accelerators and memory units to deliver higher performance than CPU-centric systems. Although heterogeneous systems have been adopted by modern workloads such as machine learning, programming remains a critical limiting factor. Conventional heterogeneous programming techniques either impose heavy modifications to the code base or require rewriting the program in a different language. Such programming complexity stems from the lack of a unified abstraction layer for computing and data exchange, which forces each programming model to define its abstractions. However, with the emerging cache-coherent interconnections such as Compute Express Link, we see an opportunity to standardize such architecture heterogeneity and provide a unified programming model. We present CodeFlow, a language runtime system for heterogeneous computing. CodeFlow abstracts architecture computation in programming language runtime and utilizes CXL as a unified data exchange protocol. Workloads written in high-level languages such as C++ and Rust can be compiled to CodeFlow, which schedules different parts of the workload to suitable accelerators without requiring the developer to implement code or call APIs for specific accelerators. CodeFlow reduces programmers' effort in utilizing heterogeneous systems and improves workload performance.

1 INTRODUCTION

Modern workloads heavily rely on heterogeneous systems. Such systems integrate multiple types of specialized computing and memory devices toward high-performance computing. GPU and high-bandwidth memory (HBM) support high-performance machine learning systems, SmartNICs accelerate network processing in cloud computing, and processing-in-memory (PIM) improves computing efficiency in memory-intensive workloads. With the advancement of device interconnection technologies such as PCIe, NVLink, and Compute Express Link (CXL), more types of accelerators and memory will be integrated into heterogeneous systems to support irregular performance requirements of workloads such as large language models, graph processing, and serverless.

Software support remains one critical limitation of heterogeneous systems' scalability. Historically, accelerators have established unique protocols to exchange data and manage tasks, relying on specialized operating system drivers and programming libraries. Thus, to use heterogeneous systems, applications must employ specialized code for accelerators while ensuring compatibility between them, and the operating system must manage different drivers. Such programming complexity drives the development of programming supports for heterogeneous computing. However, they either introduce unconventional programming models [10, 9, 6] requiring rewriting applications, or targeting domain specific use cases such as machine learning [3, 11, 8] making them hard to generalize.

```c
#include <device_library>
int main() {
    int r[N], s[N];    // Results and inputs
    for (int i = 0; i < TOTAL_DEVICES; i++) {
        device_func(&(r[i]), &(s[i]));
    }
}
void device_func(int *result, int *input){
    device_copy(input, device_memory); // Explicit memory copy through PCIe
    device_calculate();              // Call external library and compiler
    device_wait_result();            // Device-specific synchronization
    device_copy(device_memory, result);
}
```

(a) Heterogeneous code example.

```c
int main() {
    std::thread t[N]; // N threads
    int r[N], s[N];    // Results and inputs
    for (int i = 0; i < N; i++){
        t[i] = std::thread(thread_func, std::ref(r[i]), std::ref(s[i]));
        t[i].join();  // Native synchronization support
    }
}
void thread_func(int &result, const int &input) {
    result = calculate(input); // C++ native code, transparent memory sharing
}
```

(b) Multithreading code example.

Figure 1: Heterogeneous programming is generally more complex than multi-threading programming due to the need of using external libraries, system drivers, and compiler toolchains.

Programming heterogeneous systems is a special type of parallel programming where CPUs send data and code to accelerators, signal them to start tasks, and wait for results. In this process, CPUs and accelerators work in parallel and communicate through PCIe or similar interconnection buses. Such parallel computing in heterogeneous systems (Figure 1a) is conceptually similar to multi-CPU systems (Figure 1b), where multiple CPU cores run different program threads and communicate through core-to-core buses. However, multi-CPU programming has a much more intuitive programming model where the parallel tasks are defined as native functions and spawned by `fork` or `clone` system calls which are natively supported by commonly used languages such as C, C++, Rust. Such multi-threading models are enabled by (1) the unified CPU architecture that enables the code to be implemented in a unified language, and (2) the cache-coherent connections between CPUs that enables different threads to share memory transparently without relying on explicit calls to libraries. Historically, these two architectural properties are not available in heterogeneous systems, because different accelerators implement their unique architectures, and they are typically interconnected through non-cache-coherent peripheral buses such as PCIe. With the recent advancement of Compute Express Link (CXL) as a general cache-coherent interconnection, we see an opportunity to generalize heterogeneous computing.

We present CodeFlow, a unified programming model for heterogeneous computing. CodeFlow leverages WebAssembly System Interface (WASI) [4] to build a language runtime system, which serves as an intermediate layer between higher-level language and lower-level heterogeneous architectures. This runtime system allows programs
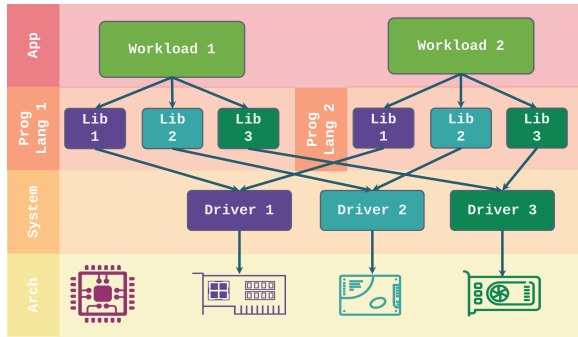
Figure 2: Heterogeneous system stack with conventional programming techniques, where workloads have to use multiple libraries and compilers to utilize underlying heterogeneous architectures. Such programming complexity limits the system's scalability of adding new architectures.

to be implemented in a single language and compiled to a single runtime representation, without the need to explicitly implement code for different architectures and compile with multiple toolchains. At low level, CodeFlow leverages CXL to enable coherent memory sharing among heterogeneous accelerators, thus minimizing explicit library calls for cross-devices data movements.

We demonstrate that CodeFlow can significantly simplify heterogeneous programming. The heterogeneous code can now be written as plain multi-threading code, which leverages the language's native supports for standard multi-threading mechanisms, such as synchronization and memory sharing. The runtime schedules different threads to run on different accelerators, handles memory sharing through CXL, and just-in-time compiles code for accelerator architectures. CodeFlow also allows conventional multi-threading programs–designed for multi-CPU systems only–to leverage heterogeneous systems by re-compiling with CodeFlow toolchain.

2 BACKGROUND AND MOTIVATION

Heterogeneous computing is similar to multi-threading in many ways. We observe a few fundamental limitations in existing heterogeneous computing that hold back the system scalability and point out potential solutions with the recent advancements in programming and architecture techniques.

2.1 Multi-Threading Programming

Multi-threading is a programming approach that allows a process to be divided into multiple threads, with each thread capable of executing concurrently. This model enables applications to perform multiple operations simultaneously, leveraging the computational power of modern multi-CPU systems more effectively. In such systems, cache coherence plays a critical role at architecture level, ensuring the consistency of data cross the caches of a multi-core or multi-processor systems. When multiple threads running on different CPU cores access shared data, cache coherence mechanisms ensure that all modifications are propagated across all caches. This guarantees that any thread accessing a shared data will see the most

recent update, irrespective of which core the updating thread runs on.

Cache coherence fundamentally simplifies the multi-threading programming model while ensuring correctness and improving performance. Cache coherence allows developers to write multithreaded applications without the cumbersome need to manually synchronize data between threads or processors. This abstraction reduces the complexity of coding for concurrency and helps applications scale efficiently by adding more CPU cores.

2.2 Heterogeneous Systems

Heterogeneous systems integrate diverse processors and accelerators, such as CPUs, GPUs, FPGAs, and DSPs, to leverage their specialized computational capabilities for enhanced performance and efficiency across various applications. This integration, while beneficial for addressing the irregular demands of modern computing tasks such as machine learning and graph processing, introduces significant programming challenges. As shown in Figure 2, developers are imposed to utilize multiple programming models and languages, alongside the use of specialized libraries designated to each architecture. Although many libraries and languages emerges to serve as an intermediate layer between developers and heterogeneous systems, aiming to provide easy-to-use programming interfaces, they either introduce new programming models [10, 9] or target scoped use cases, requiring application rewriting and raising the complexity of adopting new architectures.

Such programming complexity in heterogeneous systems is mainly caused by (1) the absence of cache coherence and (2) the diversity of architectures. The lack of a unified cache coherence mechanism leads to data inconsistencies, requiring explicit data management and synchronization efforts. Moreover, the need to navigate and optimize for different execution models and memory systems adds to the development burden. These two challenges underscore the need for unified programming models and toolchains that can simplify data management and exploit the full potential of heterogeneous systems. In following sections, we describe two key techniques that enable us to address the cache coherence and architecture heterogeneity challenges respectively.

2.3 Compute Express Link

Compute Express Link (CXL) [2] is an open industry-standard interconnection protocol, designed to facilitate high-speed communications between processors, accelerators (such as GPUs and SmartNICs), and memory devices in computing systems. A unique feature of CXL is its capability at hardware level to enable coherent memory access across heterogeneous architectures. This coherence ensures that various processors and accelerators can transparently share memory resources, as though they were part of a unified system. By leveraging CXL, we can simplify the memory access operations in heterogeneous programming models.

2.4 WebAssembly System Interface

WebAssembly is a language runtime system initially designed to accelerate web application. It is a hardware-agnostic binary instruction format designed as a portable compilation target for high-level
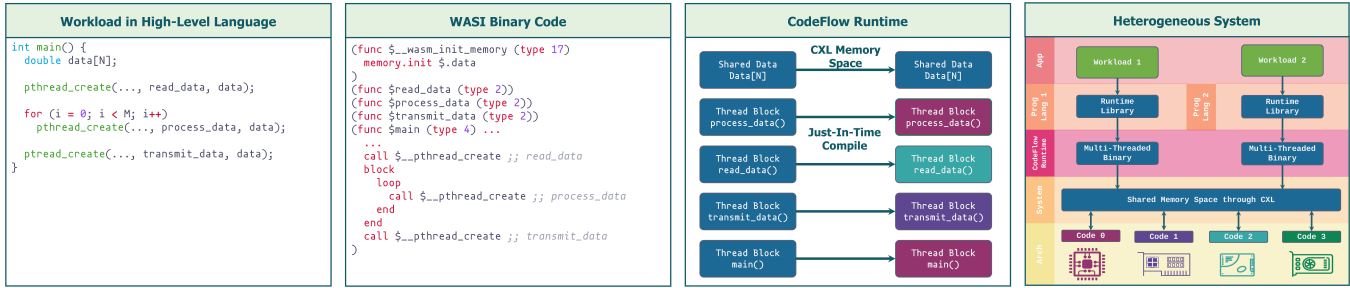
Figure 3: Heterogeneous system programming flow with CodeFlow.

languages like C, C++, and Rust. It enables code to run with near-native performance across various platforms, initially within web browsers to improve the performance of web applications. Recently, WebAssembly System Interface (WASI) has been proposed to provide WebAssembly applications with a standardized system interface, allowing them to interact with the operating system's underlying capabilities, such as file systems, network sockets, and other system resources. With WASI, high-level code can now compile to WebAssembly, run in WASI environment, access system resources through standardized interface, and execute on any suported architecture, all without rewriting or recompiling the code. This portability makes WASI a perfect candidate for us to design an abstraction layer between high-level code and low-level heterogeneous architectures.

3 DESIGN

This section introduces the design of CodeFlow framework, from compilation, runtime management, to CXL-based shared memory and heterogeneous system mapping.

3.1 Compile High-Level Code

As shown in Figure 3, the workload code is written in high-level language as a multithreading program. The workload is divided as a few sub-tasks each running as a standalone thread, sharing the access to the shared memory. In this example, each thread is responsible for a specific job, from reading the data from the file, to processing the data in parallel, and finally transmitting the data. All threads are implemented in native language (C in this example) without using any hardware-specific library or compiler. The program invokes the C standard libraries provided by WASI [5], which implement native library functionalities while inherently interact with the WebAssembly runtime system instead of the bare metal operating system. The workload can call third-party libraries for specific tasks such as network I/O, and all libraries are compiled to WASI binary format with the workload.

WASI defines a hardware-agnostic assembly binary representation by extending the WebAssembly binary representation with capabilities to access operating system resources such as filesystem. The high level workload is compiled once as WASI binary, where each thread is represented by a WASI code block, as shown in Figure 3. Such WASI binary is then executed by CodeFlow runtime system which schedules threads in the heterogeneous system.

3.2 Runtime System

CodeFlow runtime system executes WASI multi-threading binaries in the CXL-interconnected heterogeneous system. On loading a WASI binary, CodeFlow analyzes each thread's code and detects a processor or accelerator to execute the thread. E.g., a file access thread will be assigned to CPU or an in-storage processor, a parallel data processing thread will be scheduled to use GPU, and network traffic is scheduled to use SmartNICs. Developers can annotate the workload source code to assist CodeFlow's device detection process. Once devices are enumerated, CodeFlow performs just-in-time (JIT) compilation that compiles the WASI binary to the device's code and executes it. This code generation is built-on-top-of conventional heterogeneous computing libraries, including device libraries such as CUDA, and frameworks such as PyTorch [3] and SYCL [10]. The workload's shared memory is placed in the CXL memory space shared by processors and accelerators to provide coherent access across threads. To minimize the JIT overhead, the developer can choose to perform ahead-of-time (AOT) compilation that compiles the WASI binary to device code ahead-of-time, thus reducing the JIT time consumption at runtime.

3.3 CXL Interconnection

The heterogeneous system is interconnected through CXL, where processors and accelerators share a coherent memory address space, allowing all connected devices to coherently access such memory space. Following CXL's specification, an accelerator with device memory is defined as a CXL Type 2 device, and a pure memory device is defined as a CXL Type 3 device. The heterogeneous system could be composed by a combination of Type 2 and 3 devices and their on-device memory has divergent performance characteristics. CodeFlow places shared data in CXL memory space for coherent accesses from different devices. To maximize the workload performance, CodeFlow plans the data placement in device memory (e.g., GPU memory) as part of its profiling-guided optimizations at runtime, and such data can be migrated between devices to maximize temporal access performance [7].

4 EVALUATION

We build an experimental system with CXL-capable CPUs and FPGAs. As shown in Table 1, we use a two-socket Intel CPU machine. We configured the BIOS to enable CXL connections between CPU and devices on PCIe bus. We implemented CXL devices in Intel

Table 1: System for Evaluation

CPU	Intel SapphireRapids 4416+ 2 Sockets, 20 cores per socket
Memory	256 GiB DDR5
CXL	Intel Agilex-7 FPGA CXL hard IP type 2 & type 3 64 GiB DDR4 SODIMM
OS	Ubuntu 22.04, Linux 6.7

Table 2: CXL Performance

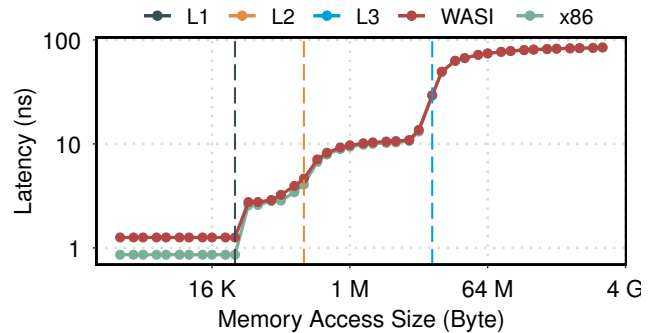| Component	Performance	
	Latency (ns)	Bandwidth (GiB/s)
Local DDR5	108.2	105.0
Remote DDR5	171.5	59.1
Local CXL	371.2	17.4
Remote CXL	538.0	9.0

Agilex-7 FPGAs by invoking Intel's CXL hard IP on the FPGA, connecting these IPs with on-board memory, and presenting the FPGA as a CXL device to the system. We modified the Linux kernel's CXL driver to recognize our CXL devices due to the incompatibilities between our CXL 1.1/2.0 IPs and the Linux kernel's implementation. We implemented CodeFlow on top of Wasmtime [1] to be compatible with WASI SDK 21 with thread extension, and evaluated the system's performance.

4.1 CXL Performance

We built a memory performance microbenchmark suite based on prior research on heterogeneous memory systems [13, 12], and use it to profile the CXL device performance. This benchmark suite uses strided memory access with pointer-chasing pattern to measure memory access latency. Table 2 shows our evaluation of latency and bandwidth of CXL compared to CPU system memory. The *Local* represents performance between a local NUMA CPU and a local NUMA/CXL memory and *Remote* represents performance between remote NUMA nodes. We find that local CXL latency is much lower than remote CXL latency, while being higher than CPU system memory. The CXL memory bandwidth is lower than remote CPU memory bandwidth. This is because our Intel FPGA has DDR4 on-board memory which is slower than DDR5, and FPGA CXL controller adds up latencies due to FPGA frequency limited to 475MHz. We expect production CXL memory to have higher performance.

4.2 WASI Performance

We adapt our microbenchmark suite to run in WebAssembly and profile the CodeFlow runtime system performance. Figure 4 shows our evaluation of memory access latency given different memory access sizes. We use a pointer chasing memory access pattern to minimize the system noises (from cache prefetchers and compiler optimizations) in latency measurements. We observe that CodeFlow

Figure 4: CodeFlow runtime performance compared to running native x86 code, using pointer-chasing read.

has the same latency compared to x86 except a slightly higher latency when memory region is with CPU L1 cache size. This shows that CodeFlow introduces very minimal memory footprint overhead when accessing memory.

5 FUTURE DIRECTIONS

We propose a new programming model in heterogeneous systems allowing workload to be written in multi-threading programming model which is widely used in existing software. To support this programming model, we present our initial effort on implementing a language runtime system, CodeFlow, based on the WebAssembly System Interface. CodeFlow abstracts the underlying architecture and generates code at runtime, relying on just-in-time compilations in WASI. Looking forward, we envision this programming model is beneficial for future workloads to easily adopt heterogeneous systems. We are working on extending CodeFlow's compiler and code generation system to support more types of accelerators. At architecture level, we are implementing CXL accelerators on CXL FPGA to support high-level workloads' requirements, and converting conventional workloads to utilize heterogeneous systems through CodeFlow.

6 CONCLUSION

In this paper, we present CodeFlow, a unified programming model for heterogeneous programming. CodeFlow employs the emerging CXL interconnection to provide coherent data exchange between heterogeneous devices, and leverages WASI's portable binary format as an intermediate representation between high-level workload and the underlying heterogeneous architecture. With CodeFlow, application developers can implement workloads as multithreading programs using native language supports, compile it once, and let CodeFlow to execute these threads on heterogeneous architectures. This programming model decouples the complexity in heterogeneous programming, allowing programmers to focus on designing and optimizing high-level logics, leaving device-specific implementations to system developers, and leveraging a multithreading programming model as a unified intermediate layer. We envision that with CodeFlow, future workloads can be ported to various heterogeneous systems without rewriting or recompiling code, thus incentivizing the adoption of heterogeneous computing.

REFERENCES

[1] Bytecode Alliance. 2024. Wasmtime: a fast and secure runtime for webassembly. https://github.com/bytecodealliance/wasmtime.

[2] CXL Consortium. 2024. Compute express link. https://computeexpresslink.org/.

[3] The Linux Foundation. 2024. Pytorch. https://pytorch.org/.

[4] WebAssembly Community Group. 2024. The webassembly system interface. https://wasi.dev/.

[5] WebAssembly Community Group. 2024. Wasi libc implementation for webassembly. https://github.com/WebAssembly/wasi-libc.

[6] Intel. 2024. oneAPI: a new era of heterogeneous computing. https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html.

[7] Hasan Al Maruf et al. 2023. Tpp: transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (ASPLOS 2023).

[8] OpenAI. 2024. Introducing triton: open-source gpu programming for neural networks. https://openai.com/research/triton.

[9] The Khronos Group. 2024. OpenCL: open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/.

[10] The Khronos Group. 2024. SYCL overview. https://www.khronos.org/sycl/.

[11] TornadoVM. 2024. Java acceleration with TornadoVM. https://www.tornadovm.org/.

[12] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[13] Zixuan Wang, Mohammadkazem Taram, Daniel Moghimi, Steven Swanson, Dean Tullsen, and Jishen Zhao. 2023. NVLeak: Off-Chip Side-Channel attacks via Non-Volatile memory systems. In *32nd USENIX Security Symposium (USENIX Security 23)*.