CuPBoP: Making CUDA a Portable Language

RUOBING HAN, Georgia Institute of Technology, Atlanta, United States
JUN CHEN, Georgia Institute of Technology, Atlanta, United States
BHANU GARG, Georgia Institute of Technology, Atlanta, United States
XULE ZHOU, Georgia Institute of Technology, Atlanta, United States
JOHN LU, Georgia Institute of Technology, Atlanta, United States
JEFFREY YOUNG, Georgia Institute of Technology, Atlanta, United States
JAEWOONG SIM, Seoul National University, Gwanak-gu, Korea (the Republic of)
HYESOON KIM, Georgia Institute of Technology, Atlanta, United States

CUDA is designed specifically for NVIDIA GPUs and is not compatible with non-NVIDIA devices. Enabling CUDA execution on alternative backends could greatly benefit the hardware community by fostering a more diverse software ecosystem.

To address the need for portability, our objective is to develop a framework that meets key requirements, such as extensive coverage, comprehensive end-to-end support, superior performance, and hardware scalability. Existing solutions that translate CUDA source code into other high-level languages, however, fall short of these goals.

In contrast to these source-to-source approaches, we present a novel framework, CuPBoP, which treats CUDA as a portable language in its own right. Compared to two commercial source-to-source solutions, CuPBoP offers a broader coverage and superior performance for the CUDA-to-CPU migration. Additionally, we evaluate the performance of CuPBoP against manually optimized CPU programs, highlighting the differences between CPU programs derived from CUDA and those that are manually optimized.

Furthermore, we demonstrate the hardware scalability of CuPBoP by showcasing its successful migration of CUDA to AMD GPUs.

To promote further research in this field, we have released CuPBoP as an open-source resource.

CCS Concepts: • **Computer systems organization → Parallel architectures;**

Additional Key Words and Phrases: GPU, code migration, compiler transformations

Jun Chen, Bhanu Garg, Xule Zhou, and John Lu are authors performed this work while affiliated with the Georgia Institute of Technology.

This research was supported in part by research infrastructure and services provided by the Rogues Gallery testbed [40], hosted by the Center for Research into Novel Computing Hierarchies (CRNCH) at Georgia Tech. The Rogues Gallery testbed is primarily supported by the National Science Foundation (NSF) under NSF Award Number #2016701. This work was also supported in part by the Ministry of Science and ICT under the ITRC support program (IITP-2023-RS-2022-00156295).

Authors' Contact Information: Ruobing Han, Georgia Institute of Technology, Atlanta, Georgia, United States; e-mail: hanruobing@gatech.edu; Jun Chen, Georgia Institute of Technology, Atlanta, Georgia, United States; e-mail: jchen706@gatech.edu; Bhanu Garg, Georgia Institute of Technology, Atlanta, Georgia, United States; e-mail: bgarg@gatech.edu; Xule Zhou, Georgia Institute of Technology, Atlanta, Georgia, United States; e-mail: marcuszhou@gatech.edu; John Lu, Georgia Institute of Technology, Atlanta, Georgia, United States; e-mail: jlu393@gatech.edu; Jeffrey Young, Georgia Institute of Technology, Atlanta, Georgia, United States; e-mail: jyoung9@gatech.edu; Jaewoong Sim, Seoul National University, Gwanak-gu, Seoul, Korea (the Republic of); e-mail: jaewoong@snu.ac.kr; Hyesoon Kim, Georgia Institute of Technology, Atlanta, Georgia, United States; e-mail: hyesoon@cc.gatech.edu.

ACM 1084-4309/2024/06-ART60
https://doi.org/10.1145/3659949

ACM Reference Format:
Ruobing Han, Jun Chen, Bhanu Garg, Xule Zhou, John Lu, Jeffrey Young, Jaewoong Sim, and Hyesoon Kim. 2024. CuPBoP: Making CUDA a Portable Language. *ACM Trans. Des. Autom. Electron. Syst.* 29, 4, Article 60 (June 2024), 25 pages. https://doi.org/10.1145/3659949

1 INTRODUCTION

CUDA provides developers with a flexible and powerful toolkit to harness the processing power of GPUs to accelerate compute-heavy applications, such as physics simulations [29, 34, 50] and Deep Learning models [41, 45]. The CUDA programming model offers a high-level interface for expressing data-parallel computations, allowing programmers to focus on the algorithmic and computational aspects of their applications. In addition, CUDA features several powerful libraries, including cuBLAS [13], cuFFT [15], and cuDNN [14], which provide users with optimized implementations of fundamental operations in linear algebra, FFT, and Deep Learning. This enables programmers to write efficient and scalable code with minimal effort.

However, it should be noted that CUDA is a proprietary language that can **only** be executed on NVIDIA GPUs. Due to the high cost and occasional shortages in supply chains, high-end or server-class NVIDIA GPUs may not be affordable for every researcher and developer. Furthermore, some of the most powerful supercomputers do not contain NVIDIA GPUs. For example, Fugaku contains only Fujitsu A64FX CPUs [31], and Frontier uses both AMD CPUs and AMD GPUs [62]. These supercomputers, although providing a huge amount of computational resources, cannot execute CUDA applications.

1.1 Motivation

It is worthwhile to summarize the benefits of making CUDA portable to other devices. First, as an increasing number of researchers use CUDA to implement GPU programs, enabling the portability of these programs to other devices can enhance their software ecosystem and promote better hardware design for non-NVIDIA vendors. Additionally, it can diversify the application ecosystem for supercomputers that do not have NVIDIA GPUs. Specifically, since CPUs are among the most ubiquitous and affordable hardware in data centers and consumer platforms, executing CUDA on CPUs can provide additional benefits. First, it allows single-kernel-multiple-device execution [48] in CPU-GPU heterogeneous systems, which can reduce runtime [47] and/or lower energy consumption [63]. Second, developers can leverage well-developed CPU debug toolkits for CUDA programs.

Taking into account all of the benefits mentioned above, we can summarize several goals that should be considered when designing frameworks for CUDA migrations:

Software Coverage: The solutions should handle the most common CUDA applications, specifically supporting widely used CUDA APIs in popular applications.

End-to-end: The solutions should seamlessly support off-the-shelf CUDA programs without requiring any manual pre/post-processing of the CUDA programs or the translated programs. The solutions should migrate both CUDA kernel programs and host programs.

Performance: The solutions should effectively utilize the computational resources of the target devices. For CPU backends, the transformed programs should utilize SIMD instructions and multiple cores. Although some solutions [21, 30, 58] aim to use CPU toolkits to debug CUDA programs, real CUDA applications often contain heavy workloads that cannot be executed on CPUs within a reasonable timeframe without efficiently utilizing the CPU computation resources.

Compatibility: The CUDA migration process involves various toolkits, such as compilation and runtime components. The solutions should strive to be compatible with the existing

compilation/runtime software as much as possible and should avoid any modifications to off-the-shelf compilers or NVIDIA SDKs.

Hardware Scalability: The solutions should be scalable to support different devices, and the migration process for different devices should follow the same workflow to avoid redundant work.

We acknowledge that the design of solutions for CUDA migration may be constrained by commercial limitations. However, in this article, we will **NOT** consider these limitations.

1.2 Limitations of Existing Solutions

The most popular approach for CUDA migrations is source-to-source translation [5, 7, 21, 52, 60, 61]. For instance, AMD proposes HIPIFY [7], a compiler-level toolkit that translates CUDA programs to HIP programs, which can be run on AMD GPUs. Intel also develops DPCT [5] to translate CUDA to SYCL, enabling execution on Intel CPUs/GPUs. We refer to these solutions as "src2src" solutions. These solutions are typically lightweight, as they rely solely on source code transformations. Moreover, since the input and output are human-readable high-level source code, they are easy to maintain. However, the effectiveness of these solutions heavily relies on the similarity between CUDA and the target languages. As discussed in Section 3.1, the differences between CUDA and target languages can hinder src2src solutions from achieving high software coverage without manual post-processing. Additionally, as CUDA is still rapidly evolving, the target languages may not have counterparts for certain CUDA features, posing challenges in achieving high software coverage. Another challenge arises from hardware scalability. To make CUDA portable to various hardware, the translators need to translate CUDA to portable languages such as SYCL [38] and OpenCL [54]. However, since portable languages encompass different hardware architectures, their development often lags behind the evolution of CUDA languages. This presents challenges in supporting new CUDA features, as updating the portable languages requires significant effort due to the involvement of multiple hardware architectures.

1.3 Insight and Contribution

In contrast to existing approaches that translate CUDA to portable languages, this article presents a novel approach of making CUDA itself a portable language. This involves implementing both compiler and runtime components. The compiler components are responsible for parsing CUDA source code and generating executable programs for non-NVIDIA devices. The runtime components handle CUDA-specific functionality, such as kernel launches and memory management, by implementing them for non-NVIDIA devices and integrating them into libraries.

By eliminating the need for third-party languages, our approach simplifies the achievement of the proposed goals. We can focus directly on CUDA without being concerned about the differences between CUDA and other languages. Supporting new CUDA features only requires updating the compiler and/or runtime components, without the involvement of updating portable languages or CUDA translators.

This article contributes the following technical contributions:

— Introduce CuPBoP, a framework that makes CUDA a portable language.
— Discuss the co-design of compilation and runtime in the implementation of CuPBoP.
— Compare the software coverage and performance of CuPBoP with two commercial CUDA migration solutions.
— Compare CuPBoP with manually optimized CPU programs and identify the gaps between CPU programs translated from CUDA and manually optimized CPU programs.
— Demonstrate the hardware scalability of CuPBoP by describing its support for AMD GPUs as a proof-of-concept [25].

— Release the codebase as open-source projects.[1][2]

To provide readers with a comprehensive understanding of the compiler and runtime components in CuPBoP, this article primarily focuses on the CPU backend. Supporting GPU-to-CPU migration poses greater challenges compared to GPU-to-GPU migration, such as CUDA-to-HIP. However, we also include the results of supporting CUDA on AMD GPUs as a proof of concept to showcase the hardware scalability of CuPBoP.

In the remaining sections of the article, we present a high-level overview of CuPBoP in Section 2. We discuss the insights and critical design decisions underlying the framework in Section 3. Section 4 delves into several technical challenges encountered during the implementation of CuPBoP. The evaluation of CuPBoP and its comparison with other CUDA migration solutions are presented in Section 5. In Section 6, we introduce the AMD backend of CuPBoP as a proof-of-concept to demonstrate its hardware scalability. We explore related works in Section 7, and finally, we conclude the article in Section 8.

2 CUPBOP

CuPBoP (**Cu**da for **P**arallelized and **Bro**ad-range **P**rocessors) is a framework designed to address the limitations of src2src solutions by making CUDA a portable language. It enables the execution of CUDA programs on non-NVIDIA devices.

2.1 Workflow

Figure 1 illustrates the workflow of CuPBoP for generating CPU/AMD GPU executables from CUDA source code. The CUDA code consists of both host and kernel programs, where the host program executes on the CPU device, and the kernel program executes on NVIDIA GPUs. The workflow can be divided into three stages.

In the first stage, CuPBoP utilizes Clang [46] to compile the CUDA source code, producing two separate **intermediate representations (IRs)**: LLVM IRs for the host program and NVVM IRs [55] for the kernel program.

In the second stage, CuPBoP applies various compilation transformations on the LLVM/NVVM IRs. These transformations are crucial for adapting the CUDA code to the target CPU or AMD GPU architectures. To avoid modifying the off-the-shelf compiler, we do not integrate these transformations into the LLVM codebase. Instead, we utilize existing LLVM APIs to build two IR-to-IR translators. These translators accept the LLVM/NVVM IRs compiled from the first stage, apply IR-to-IR transformations on them, and output transformed LLVM IRs.

In the final stage, the transformed LLVM/NVVM IRs are compiled to object files and linked with the CuPBoP CPU/AMD GPU runtime libraries, resulting in the generation of CPU/AMD GPU executable files.

Although supporting different backends requires various compilation transformations and runtime libraries, all backends follow the same workflow.

2.2 Supported/Unsupported CUDA Features

CuPBoP supports most commonly used CUDA APIs, ensuring compatibility with memory management, kernel launches, static/dynamic shared memory, global synchronization, CUDA streams, CUDA events, constant memory, and common math functions (e.g., ceil, floor, sqrt). However, there are certain CUDA features that are not supported in CuPBoP for various reasons:

[1]CPU backend: https://github.com/cupbop/CuPBoP
[2]AMD GPU backend [25]: https://github.com/gthparch/CuPBoP-AMD

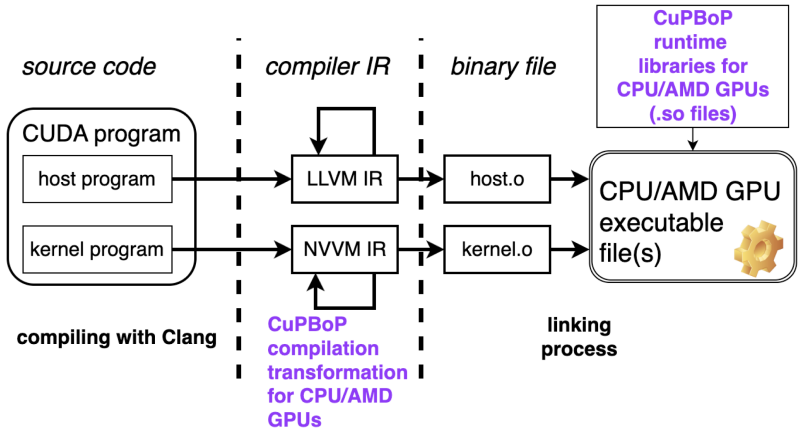


Fig. 1. The workflow of CuPBoP for generating CPU/AMD GPU executable files from CUDA source code. The purple elements in the diagram represent the compilation transformations and runtime libraries included in CuPBoP.

Texture memory: While it is theoretically possible to support texture memory on CPUs, executing texture memory operations with high performance on CPUs is challenging, as these operations are closely tied to NVIDIA GPU hardware. Therefore, supporting texture memory on CPUs is not a high priority for CuPBoP. However, CuPBoP does support texture memory for AMD GPUs.

PTX assembly: PTX assembly is specific to NVIDIA and may not be portable to other architectures due to differences in register and memory fences. Since the primary goal of CuPBoP is to achieve performance portability [59] with CUDA, it does not function as a CUDA emulator; therefore, PTX assembly is not supported.

CUDA libraries: Certain CUDA libraries, such as cuBLAS and cuDNN, play a critical role in HPC applications. However, migrating these libraries is highly dependent on the target hardware. For example, CuPBoP can utilize Intel oneMKL[16] and AOCL [12] for Intel CPUs and AMD CPUs migration, respectively. Supporting these libraries requires significant effort, and it is an area that CuPBoP plans to explore in future updates.

It is important to note that, to the best of our knowledge, none of the existing CUDA-to-CPU solutions [5, 7, 28, 53, 58, 65] support texture memory or CUDA libraries. Support for PTX assembly is exclusive to projects [18, 32] that utilize CPUs as CUDA emulators without targeting high performance.

3 INSIGHTS

In this section, we discuss three important technical decisions for migrating CUDA to CPUs. The first challenge is how to parse and analyze CUDA source code to achieve high coverage without requiring manual pre- or post-processing. The second challenge is how to efficiently execute **Single-Program-Multiple-Data (SPMD)** applications on CPUs. The third decision is about how to launch CUDA kernels efficiently by utilizing the multiple CPU cores.

3.1 Front-end

The first design decision is how to parse and analyze CUDA source code to achieve high coverage without requiring manual pre- or post-processing. The src2src solutions typically treat CUDA source code as text files and convert CUDA programs into other high-level languages such as HIP [7], OpenCL [37, 52, 60], and SYCL [5]. These translators rely on AST analysis or regular

Table 1. The Differences between APIs in CUDA and the Target Languages

CUDA	OpenCL	SYCL
✗	platform	selectors
✗	context	✗
✗	device	selectors
stream	command queue	queue
✗	program	✗
function	kernel	single_task
kernel launch	enqueueNDRangeKernel	submit
__shfl_up_sync	✗	shift_group_left*

*shift_group_left does not support operations on a subset of sub_group. Thus, it cannot be used to replace __shfl_up_sync directly.

expressions, which are lightweight. Moreover, the input and output of these translators are high-level languages, which makes the translation process human-readable and amenable to manual pre- or post-processing.

However, src2src solutions face challenges due to differences in APIs across languages. When translating CUDA to another language, such as OpenCL or SYCL, translators must handle the discrepancies in API functions and data structures. Table 1 provides an example of some APIs used in CUDA, OpenCL, and SYCL, highlighting the differences. These differences can require additional code generation or manual modifications to ensure correctness and compatibility. For example, when translating CUDA into OpenCL, translators have to generate platform, context, device, and program variables that do not exist in CUDA. Additionally, some CUDA features (e.g., warp shuffle functions) do not have direct counterparts in the target language, necessitating updates to both the target language and the translators to support these features.

Instead of using AST analysis toolkits, CuPBoP uses the Clang compiler to parse CUDA source code and generate IR files, which is the standard approach for compiling CUDA programs for NVIDIA GPUs. The resulting IR files are transformed, compiled to object binary files, and linked with CuPBoP's runtime libraries. Unlike src2src solutions, CuPBoP does not rely on other programming languages. In our evaluation section, we show that CuPBoP achieves higher coverage than two commercial src2src solutions. The difference between CuPBoP and src2src solutions is shown in Figure 2.

3.2 Kernel Launch

One of the most crucial CUDA APIs is the kernel launch: The host thread invokes a kernel with a specified grid size and block size. The kernel is then executed by $grid\_size * block\_size$ GPU threads. To support GPU-to-CPU migration, the most straightforward solution is to use a CPU thread to replace a GPU thread. Specifically, the transformed CPU programs fork the same amount of threads as GPU programs. While this mechanism is simple to implement, it typically results in lower performance. In most cases, CUDA programs contain more than thousands of threads, whereas a modern commercial CPU only has hundreds of cores. Thus, launching thousands of threads will cause frequent context switching, significantly impairing CPU performance. This mechanism, although with low efficiency, is frequently used for debugging toolkits [2, 58], as it can generate CPU executable file without significantly changing CUDA programs.

To support the execution of SPMD programs on CPUs with high performance, two solutions are commonly used: the fiber solution and the loop solution (Figure 3). Both solutions collapse the workload within a GPU block into a single function, which can be executed by a CPU thread. Thus, for a CUDA kernel launch with *grid_size* and *block_size*, only *grid_size* CPU threads are

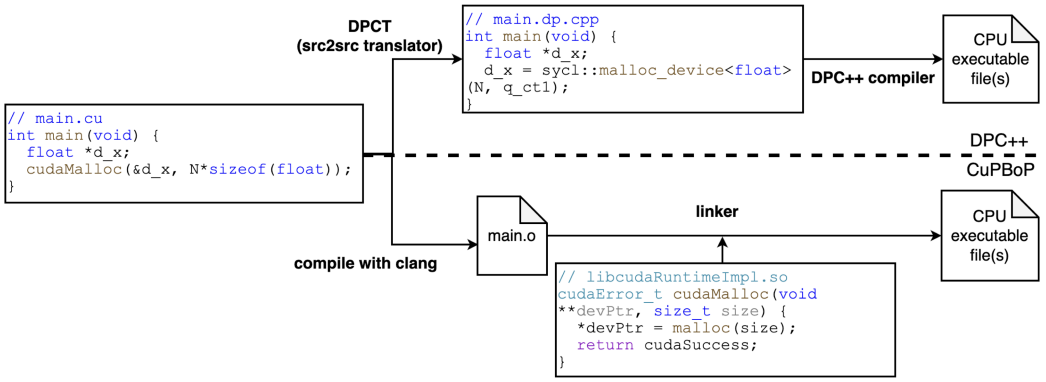


Fig. 2. Different mechanisms are employed for parsing CUDA source code. Src2src solutions translate CUDA code to other high-level languages. In contrast, CuPBoP relies on an off-the-shelf compiler to compile the CUDA source code and generate IRs. These IRs are then subjected to various transformations (not shown in the diagram) and linked with the CuPBoP runtime libraries.

needed after the transformations. Although this is a software solution, it also represents a hardware mapping where a GPU **Streaming Multiprocessor (SM)** is mapped to a CPU core. Given that modern GPUs and CPUs have a comparable number of SMs and cores (e.g., the NVIDIA A100 GPU with 108 SMs and the AMD EPYC 9654P CPU with 96 cores), this mapping is rational.

The fiber solution, which is used in HIP-CPU [6], replaces GPU threads with lightweight CPU fibers. [3] This solution provides runtime libraries that implement CUDA kernel launch by launching CPU fibers. For a CUDA launch with *grid_size* blocks and *block_size* threads for each block, the fiber solution launches *grid_size* CPU threads, and each CPU thread contains *block_size* fibers. A CPU thread executes a fiber at a time and switches to execute the next fiber when it encounters a "*yield*" function. Thus, these fibers form an implicit loop.

The loop solution is a compilation transformation that is proposed in MCUDA [65]. The loop solution generates loops to wrap the original CUDA source code. The loop length (a.k.a. trip count) is *block_size*. Each CUDA thread is mapped to an iteration in the transformed CPU program. The loop solution has less overhead than the fiber solution, since the iteration switching overhead is much lower than the fiber switching overhead. Additionally, the loop solution generates parallelized loops, allowing transformed CPU programs to potentially utilize SIMD instructions [36, 44]. Therefore, CuPBoP chooses the loop solution. In Section 5.3.1, we compare the performance of CuPBoP (loop solution) and HIP-CPU (fiber solution).

Although this article does **NOT** present fiber solutions or loop solutions as our contributions, it is the first to provide a performance comparison of these two different mechanisms. The article aims to contribute by shedding light on these mechanisms, highlighting their differences, and evaluating their performance. By doing so, it offers valuable insight into the efficacy of using fiber solutions and loop solutions for specific use cases.

3.3 Runtime

To fully utilize CPU computational resources, CuPBoP implements a runtime system that supports multiple thread execution. During initialization, CuPBoP launches a thread pool consisting of a number of threads equal to the number of CPU cores. A host thread is responsible for memory management and kernel launch, while communication between the host thread and pool threads

[3] Fiber is a unit of execution that is lighter than the thread [51].

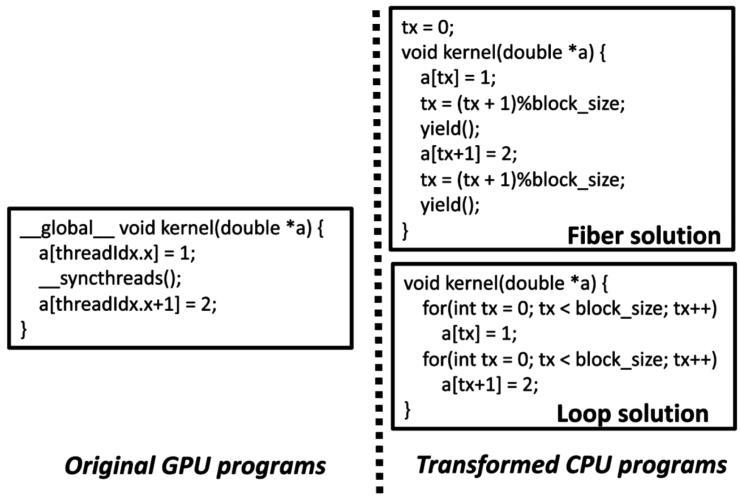


Fig. 3. Two solutions for executing SPMD programs on CPUs with high performance. CuPBoP implements the loop solution. While this article does **NOT** present fiber solutions or loop solutions as contributions, it is the first to provide a performance comparison of these two different mechanisms.

is achieved using the producer-consumer model: When a CUDA kernel launch is executed, the host thread creates multiple instances of kernel variable (Listing 3) and pushes them to the queue. The pool threads then check the queue and fetch any available tasks. The kernel launch process is illustrated in Figure 4.

Since both the push and fetch operations must be atomic, accessing the task queue frequently can lead to performance degradation. For instance, when a CUDA kernel launch has 64 GPU blocks, the host thread pushes 64 kernel variable instances to the queue, and the pool threads need to fetch the queue 64 times. To minimize the number of queue accesses, CuPBoP provides a hyperparameter called *block_per_fetch* that specifies how many block instances are executed for each fetch. In Figure 4, block_per_fetch is set to 16, so each kernel variable instance corresponds to 16 blocks. This way, the host thread and pool threads only need to push and fetch the queue 4 times, respectively.

The block_per_fetch parameter can significantly impact performance. If block_per_fetch is too small, then there will be many atomic push and fetch operations that harm performance. However, if block_per_fetch is too large, then the number of kernel variable instances may be fewer than the number of pool threads, causing some threads to remain idle. To give users control over this trade-off, CuPBoP allows the block_per_fetch value to be set as a hyperparameter. By default, CuPBoP sets block_per_fetch to match the number of CPU cores.

It should be noted that while OpenMP offers similar functionality to schedule parallel tasks on CPUs, the decision to implement a separate schedule system in CuPBoP is driven by the specific requirements and goals of the framework, prioritizing lightweight design and scalability to different backend devices. Another reason for not using OpenMP is its interface, which is overly general and makes it difficult to support certain CUDA runtime functions. For example, implementing CUDA streams is difficult within the OpenMP framework.

4 IMPLEMENTATION

In this section, we summarize several challenges and corresponding solutions for the CuPBoP implementation. We use the program in Listing 1 as an example. The CUDA program comprises

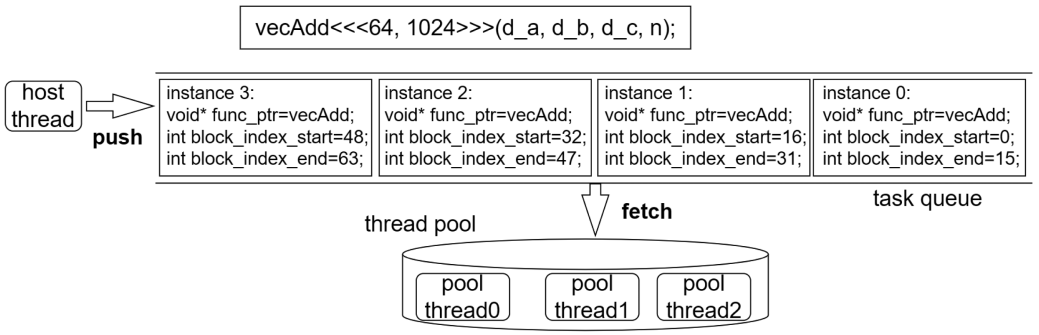


Fig. 4. The process of kernel launch. CuPBoP maintains a producer-consumer model to implement the kernel launch. For a CUDA kernel involves 64 GPU blocks, and with *block_per_fetch* set to 16, the host thread initializes four kernel variable instances and pushes them into the task queue. These instances will be fetched and executed by threads in the pool.

two functions: a host function and a kernel function. The kernel function declares an external variable named `s` that resides in the GPU-shared memory. The size of this variable is determined at runtime and is equal to `n * sizeof(int)`.

There are several challenges to migrating the program. First, the CUDA program uses shared memory features. There is no corresponding memory for CPUs. CuPBoP has to implement memory mapping to replace these memories. Second, the kernel uses a CUDA intrinsic function to get the block index. CuPBoP needs to support the CUDA intrinsic functions correctly on CPUs. Third, the CUDA kernel launch is a variadic function, which can accept arbitrary numbers/types of arguments. CuPBoP has to support all the use cases. Last but not least, the CUDA program uses dynamic shared memory, where the size of *s* is known at the runtime when invoking kernel launch. Thus, CuPBoP has to rely on compilation-runtime co-design to support this feature.

```
1 __global__ void kernel(int *d, int offset) {
2   extern __shared__ int s[];
3   int t = threadIdx.x;
4   s[t] = d[t]+offset+blockIdx.x;
5 }
6 void host() {
7   kernel<<<1, n, n * sizeof(int)>>>(d_d, n);
8 }
```

Listing 1. A CUDA program with dynamic shared memory.

The transformed program is shown in Listing 2. In real cases, the transformed program is in binary format, but we have presented it in C++ for easy understanding.

We now go through several compilation transformations that are applied by CuPBoP.

4.1 Memory Mapping

NVIDIA GPUs have an explicit hierarchical memory system, including private memory for single threads, shared memory within a block, and constant/global memories that are shared between blocks. These memories do not have direct mappings on CPU architectures.

To support these memories on CPUs, CuPBoP implements a compilation transformation to replace all CUDA global and constant memory with CPU heap memory, since CPU heap memory is accessible to all CPU threads. Additionally, CuPBoP replaces all CUDA shared memory with CPU thread local storage. The insight is, as described in Section 3.2, a CUDA block is mapped to a CPU

thread. Therefore, GPU-shared memories, which are shared within a block but different among blocks, should be mapped to CPU thread local storage.

As shown in Listing 2, the shared memory pointer is mapped to thread local storage (line 3). The assignment of the pointer is done at runtime, which is described in Section 4.4.

4.2 CUDA Intrinsic Functions

Certain CUDA intrinsic functions are essential in CUDA applications. For example, `blockIdx.x` can return the block index for a block. NVIDIA GPUs use special registers [11] to support these intrinsic functions. For example, the `ctaid` register stores the unique CUDA block ID.

However, these registers do not have equivalents in CPU architectures. Therefore, CuPBoP applies a transformation on the kernel code to insert declarations of auxiliary variables that are used to replace these registers. In Listing2, CuPBoP inserts external declarations of blockIdx_x and block_size on lines 1–2. The variable blockIdx_x is used to replace the CUDA intrinsic function `blockIdx.x` to represent the block index. The assignment of these variables is done at runtime.

```
1  extern __thread size_t blockIdx_x;
2  extern __thread size_t block_size;
3  extern __thread int* dynamic_shared_p;
4  void cpu_kernel(int** p) {
5    // prologue for unpacking in a kernel side
6    int * d = *((int **) p[0]); // dereference the first element of the packed pointer to
       get the first argument
7    int offset = *((int *) p[1]);  // dereference the second element of the packed pointer
       to get the second argument
8    // end of prologue
9    for(size_t threadIdx = 0; threadIdx < block_size; threadIdx++) {
10     int t = threadIdx;
11     dynamic_shared_p[t] = d[t]+offset+blockIdx_x;
12   }
13 }
14 void host() {
15   // prologue for packing
16   int** p = malloc (2 * sizeof(int *)); // allocate an array of pointers. Since the
       original kernel has 2 arguments, the array size should also be 2.
17   int** p0 = new int *; // allocate a pointer that refers to the first argument
18   *p0 = d_d; // store the value of the first argument
19   p[0] = (int *)p0; // store the first argument into the first element of the array
20   int* p1 = new int; // allocate a pointer that refers to the second argument
21   *p1 = n; // store the value of the second argument
22   p[1] = (int *)p1; // store the second argument into the second element of the array
23   // end of prologue
24   // the CUDA kernel launch will be replaced later
25   cpu_kernel<<<1, n, n * sizeof(int)>>>(p);
26 }
```

Listing 2. The CPU program translated by CuPBoP from Listing 1.

4.3 Variadic Function

As discussed in Section 3.3, when a host thread executes a kernel launch function, it pushes instances of kernel variable to the task queue. However, CUDA programs contain kernel functions with varying numbers of parameters and types, making it challenging to create a universal interface for all kernel functions.

To address this issue, CuPBoP applies compilation transformations on both host programs and kernel programs. These transformations insert auxiliary instructions to pack/unpack the kernel parameters. Specifically, CuPBoP inserts instructions to pack all parameters into a pointer that points to the packed parameters (Listing 2, lines 16–22). Thus, the arguments of any kernel launch would be the same. CuPBoP also inserts instructions to unpack the packed argument (p) into the original arguments (d and $offset$) (Listing 2, lines 6–7) in the kernel programs.

4.4 Kernel Launch

The kernel launch implementation involves compilation and runtime co-design. For example, to support dynamic shared memory features, CuPBoP has to replace dynamic shared memory variables by CPU thread local storage during compilation and assign valid addresses to these pointers at runtime.

```
1 typedef struct kernel {
2   void *(*func_ptr)(void *);
3   void **args;
4   // auxiliary variables
5   int block_index_start;
6   int block_index_end;
7   int block_size;
8   size_t dynamic_shared_mem_size;
9 };
```

Listing 3. The structure of kernel variable.

As described in Section 3.3, to implement CUDA kernel launch, the host thread constructs kernel variable instances and pushes them into the task queue. The definition of the kernel variable structure is shown in Listing 3. The kernel variable stores the function point of the transformed kernel programs (*cpu_kernel* in Listing 2) in func_ptr. The packed argument is stored in args. The kernel variable also stores runtime configuration, such as the block size and the size of dynamic shared memory.

The pool threads fetch kernel variable instances from the queue. When a thread successfully fetches a kernel variable instance, it sets the auxiliary variables for kernel execution and then executes the kernel function as a normal CPU function. The code shown in Listing 4 is part of the function run in the pool threads. After successfully fetching a kernel variable instance *ker*, the pool thread uses the kernel variable instance to set auxiliary variables (blockIdx_x, block_size, and dynamic_shared_p) required in the kernel programs and then executes the kernel program as a normal function call (line 6). The pool thread allocates the memory for the dynamic shared memory variables in line 4.

```
1 // after fetching a kernel variable instance ker
2 __thread size_t block_size = ker.block_size;
3 __thread size_t blockIdx_x;
4 __thread int* dynamic_shared_p = (int *)malloc(ker.dynamic_shared_mem_size);
5 for (blockIdx_x=ker.block_index_start;blockIdx_x<=ker.block_index_end;blockIdx_x++){
6     ker.func_ptr(ker.args);
7 }
```

Listing 4. Set auxiliary variables and execute kernels.

5 EVALUATION

5.1 Setting Up

The evaluations are executed on two platforms that include Intel CPUs, AMD CPUs, and NVIDIA GPUs. The platforms' specifications are summarized in Table 3. For certain evaluations, we compare the results of CuPBoP with those of DPC++[5] and HIP-CPU [6]. As CuPBoP is primarily designed to achieve high performance on CPUs, we do not evaluate frameworks [21, 30, 58] that are designed solely for debugging purposes rather than aiming for high performance.

Two benchmarks, Rodinia [24] and Hetero-Mark [66], along with an application called Cloverleaf [4], are utilized for evaluation purposes. The Rodinia benchmark is used to assess the software coverage, as it consists of applications with various C/C++ syntax, code structure, and build processes. However, for performance evaluation, the Hetero-Mark benchmark is preferred for three

Table 2. The Versions of Software Used for Evaluation

Software	Version/Commit
CUDA	12
DPC++ [5]	2024.0.2
HIPIFY [7]	rocm-5.7.0
HIP-CPU [6]	56f559
Hetero-Mark [66]	b8ea32
Rodinia [24]	9c10d3
Cloverleaf [4]	03c780

Table 3. Hardware Environment Configurations

Server Name	CPU/GPU	CPU cores/GPU SMs	Peak FLOP/s	Memory (GB)	L2 cache/shared memory
Server-Intel	Intel Gold6226R (×2)	32	5.93 T*	376	16 MB
Server-AMD-A30	AMD EPYC 7502 (×2)	64	5.12 T*	264	16 MB
	NVIDIA A30 GPU	56	10.3 T	24	128 KB

*estimated value.

reasons. First, all frameworks achieve high coverage on the Hetero-Mark benchmark, providing a larger set of data points for evaluation. Second, the Hetero-Mark applications offer knobs to adjust input size and computation workload, allowing for sufficiently large execution times to mitigate measurement errors. Last, the Hetero-Mark benchmark provides well-defined profiling code sections that facilitate the understanding of kernel execution time. Additionally, profiling results from Cloverleaf, a mini-app that solves the compressible Euler equations, are included. The developers of Cloverleaf have implemented various manually optimized versions, including an 18-kernel CUDA implementation, with host programs written in both C++ and Fortran. As neither HIPIFY nor DPCT successfully translate Cloverleaf to CPUs, we only measure the execution time of CuPBoP and compare it with Cloverleaf's OpenMP implementation. The software information for the evaluation is listed in Table 2.

5.2 Software Coverage

We evaluate the software coverage on the Rodinia benchmark [24] on the Sever-Intel platform. We show the overall software coverage evaluation result in Figure 5. We also list the evaluation results for each application in Table 4. There are 23 applications in Rodinia-CUDA. Four of them (hybridsort, kmeans, mummergpu, and leukocyte) utilize CUDA texture memory, which is not supported by any framework. Compared to HIP-CPU (60.9%) and DPC++ (65.2%), CuPBoP achieves the highest coverage (73.9%) in the evaluation.

CuPBoP raises runtime error/incorrect results in two applications: The Huffman application involves instructions that left shift a 4 bytes integrated variable for 32 bits. This instruction shows different behaviors on NVIDIA GPUs and Intel CPUs. CuPBoP raises a segfault in the Heartwall application. This is due to the compiler overoptimizing some variables, which are freed before references. We find that manually preprocessing the CUDA source code enables CuPBoP to generate correct results. Specifically, for the heartwall application, inserting an extra `__syncthreads()` in the source code allows the compiler to correctly apply the analysis on the transformed CPU program. For Huffman, changing the instruction from "`tmp = dw << 32-bit`" to "`tmp = (bit != 0) ? (dw << (32 - bit)) : 0;`" fixes the incorrect result.

HIP-CPU does not contain any compilation transformations. It only contains header files that implement HIP built-in functions for CPUs. Thus, it does not raise runtime errors. As HIP-CPU does not have compilation transformation, it cannot support dynamic shared memory, which involves

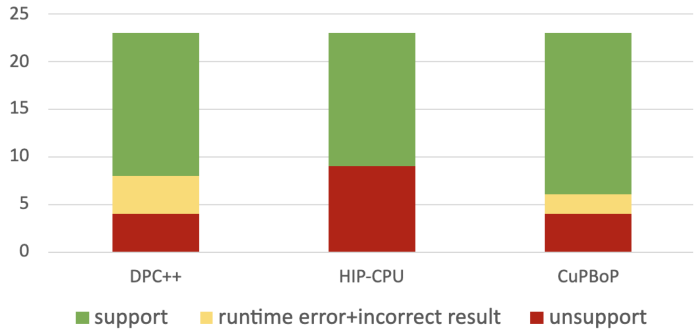


Fig. 5. The coverage on the Rodinia benchmark. The detail information is shown in Table 4.

Table 4. We Evaluate All 23 Applications in Rodinia-Benchmark on Different CUDA-to-CPU Solutions and Verify the Correctness of Transformed CPU Programs

Program	DPC++	HIP-CPU	CuPBoP
b+tree	✓	no support	✓
backprop	✓	no support	✓
bfs	incorrect result	✓	✓
gaussian	✓	✓	✓
hotspot	✓	✓	✓
hotspot3D	✓	✓	✓
lud	✓	✓	✓
myocyte	incorrect result	✓	✓
nn	✓	✓	✓
nw	✓	✓	✓
particlefilter	✓	✓	✓
pathfinder	✓	✓	✓
srad	✓	✓	✓
streamcluster	✓	✓	✓
cfd	✓	✓	✓
lavaMD	✓	✓	✓
heartwall	incorrect result	no support	runtime error
huffman	✓	no support	incorrect result
dwt2d	incorrect result	no support	✓
hybridsort	no support	no support	no support
kmeans	no support	no support	no support
mummergpu	no support	no support	no support
leukocyte	no support	no support	no support

The CuPBoP achieves the highest coverage.

compiler-runtime co-design. Additionally, it relies on C++17. Some of its code is incompatible with legacy C syntax, which is used in b+tree and backprop applications. Thus, it has more unsupported applications than CuPBoP.

DPC++ converts CUDA to SYCL and utilizes the Intel oneAPI compiler to generate executable CPU programs. DPC++ produces incorrect outputs for bfs and myocyte, with many values being `NAN`. We assume that these issues primarily arise from the implementation of the SYCL on CPUs. This assumption is supported by the fact that for the b+tree and Huffman examples, the same

transformed SYCL source code produces incorrect outputs with DPC++ 2021.4.0, yet generates correct outputs with DPC++ 2024.0.2. Thus, this suggests that the CUDA-to-SYCL translator is likely functioning correctly, and the errors stem from the SYCL environment. It is worth noting that our evaluated DPC++ coverage (65.2%) is different from the one (87%) reported in Reference [22]. This is due to two reasons. First, the authors in Reference [22] focus on exploring the upper bound of the coverage. Thus, they involve manual post-processing on transformed SYCL programs to generate the correct programs. Second, the runtime evaluations are done on different platforms. Our coverage is measured by executing the transformed SYCL programs on Intel 6226R CPUs and comparing the output with the original CUDA programs executed on NVIDIA A30 GPUs. As the authors in Reference [22] do not report the details of the correctness evaluation, we assume they verify the correctness of SYCL programs by executing them on Intel GPUs/NVIDIA GPUs.

We also measure the coverage on the Hetero-Mark benchmark. The applications in Hetero-Mark are relatively simple and do not involve complex C/C++ syntax and CUDA features. As a result, all three frameworks support 8 out of the 10 benchmarks. The BST and KNN applications rely on CUDA system-wide atomics features that are not supported by any frameworks.

We also attempt to migrate CloverLeaf-CUDA to CPUs. The CUDA implementation consists of 18 CUDA kernels, while the host programs are written in both C++ and Fortran. CuPBoP successfully translates the CUDA implementation to a CPU program. However, both DPC++ and HIPIFY fail to generate correct CPU programs. They struggle to handle the syntactic sugars that are widely used in the codebase, such as the example shown in Listing 5. This poses a challenge for src2src solutions, as they aim to produce human-readable code as output. To avoid generating complex code, they do not expand macros during translation, which makes it difficult to support syntactic sugars that involve intricate macro usages. However, as CuPBoP implements transformations on the LLVM IR level, it does not need to handle these syntactic sugars.

```
1 #define CUDALAUNCH(funcname, ...)  \
2     funcname<<<num_blocks, BLOCK_SZ>>>(x_min, x_max, y_min, y_max, __VA_ARGS__);
```

Listing 5. A syntactic sugar that cannot be translated by DPC++ or HIPIFY.

5.3 Performance

5.3.1 CUDA-to-CPU Frameworks. In this section, we measure the runtime of the transformed CPU application for the Rodinia and Hetero-Mark benchmarks. To ensure accurate measurements, we adjust the tuning knobs provided by the benchmarks to increase the execution time and minimize measurement discrepancies. We execute and measure each application seven times and report the median execution time. Although there are available SYCL/HIP programs for the same applications [42], to ensure a fair comparison, we use DPCT/HIPIFY to generate SYCL/HIP programs from CUDA programs instead of using off-the-shelf SYCL/HIP programs.

We assess the end-to-end runtime of three CUDA-to-CPU solutions on x86 CPUs (Serverl-Intel), and the results are presented in Table 5. Among the three solutions, CuPBoP demonstrates the highest performance in 17 of the 24 applications. On average, CuPBoP achieves a speedup of approximately 1.31× compared to DPC++, and around 1.66× compared to HIP-CPU.

Some applications exhibit significant performance variances among the frameworks. We analyze the performance of these applications in detail:

BS and FIR: The BS application invokes around 20,000 CUDA kernel launches, and each kernel launch only invokes 32 GPU blocks. Additionally, the CUDA kernel contains only simple floating point calculations. Thus, the workload is too lightweight for CPU cores, and most of the runtime is spent on kernel launch. CuPBoP uses a lock-free queue [9] to implement kernel launch and provides knobs to set the *block_per_fetch* parameter, which can further decrease the overhead

Table 5. The Runtime (Second) on Rodinia and Hetero-Mark Benchmarks

Benchmark	Application	DPC++	HIP-CPU	CuPBoP
Rodinia	backprop	2.42	N/A	**1.96**
	b+tree	2.67	N/A	**2.24**
	bfs	N/A	1.27	**1.14**
	Gaussian	**0.85**	8.49	1.67
	hotspot	1.24	1.27	**1.07**
	hotspot3D	1.32	1.73	**1.27**
	lud	1.33	**0.95**	1.16
	myocyte	N/A	0.40	**0.15**
	nn	1.23	1.35	**1.21**
	nw	**1.32**	1.77	1.59
	particlefilter	0.85	0.84	**0.83**
	pathfinder	2.57	2.42	**2.36**
	srad	3.93	8.31	**2.89**
	streamcluster	**17.97**	21.09	18.44
	lavaMD	0.25	0.18	**0.15**
	CFD	6.86	**5.38**	5.94
Hetero-Mark	AES	56.41	55.87	**50.54**
	BS	1.70	2.00	**0.64**
	EP	**2.81**	27.02	3.65
	FIR	2.92	3.44	**0.63**
	GA	1.04	1.43	**0.33**
	Hist	2.21	1.90	**1.89**
	Kmeans	**1.63**	3.50	3.29
	PR	4.11	4.35	**3.84**

We bold the shortest runtime for each application. For all 24 applications, CuPBoP achieves the highest performance in 17 of them.

for kernel launch. As a result, CuPBoP achieves the highest performance. The same reason also applies to the FIR application.

EP and KMeans: The CUDA kernel in EP contains nested loops, which can be optimized with vector instructions. DPC++, developed by Intel, optimizes these loops with the highest performance on Intel CPUs. As HIP-CPU applies the fiber solution, which does not group CUDA threads into loops, it cannot benefit from SIMD instructions. While CuPBoP depends on the off-the-shelf LLVM compiler to apply auto-vectorization, it does not achieve the same level of vectorization as the Intel compiler. We attempt to manually set some flags during LLVM optimizations (e.g., allowing floating-point calculation reordering) and get programs with higher performance. This issue is a traditional compiler optimization problem that extends beyond the scope of CUDA migration. Similarly, KMeans contains loops that can only be well-optimized by DPC++, which relies on the Intel compiler.

CFD: CFD implements the redundant flux calculation that is used in the fields of electrical engineering. The CUDA program is computation-intensive and contains a significant number of floating point calculations. Therefore, compiler optimization is critical for performance. Both DPC++ and CuPBoP rely on the loop solution, which applies compiler optimizations to transformed CPU programs that are relatively complex and challenging to optimize effectively. In contrast, HIP-CPU is implemented using the fiber solution, which directly applies compiler optimizations to GPU kernels. Since GPU kernels are much simpler than transformed CPU programs,

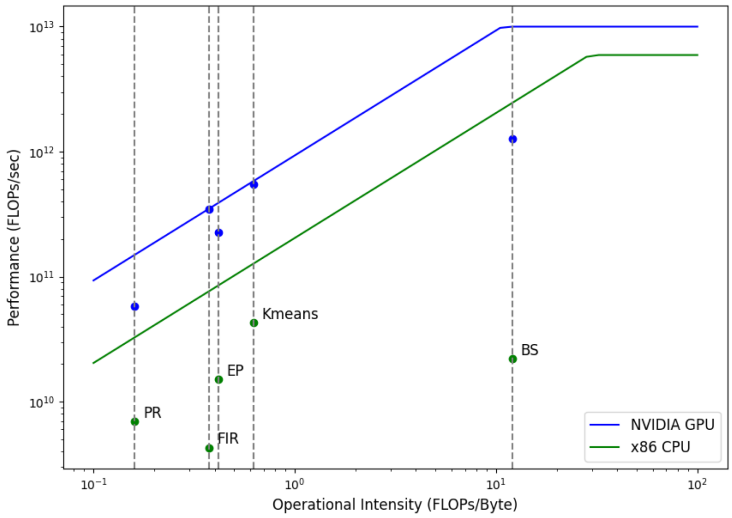


Fig. 6. The Roofline model for the CPU and GPU in Table 3.

compilers can optimize them more effectively. As a result, HIP-CPU achieves the highest performance. To assess the importance of compiler optimizations on original GPU kernels, we intentionally skip compilation optimization on CUDA programs and apply O3 optimization only to transformed CPU programs. Compared with applying O3 optimization to both the GPU kernels and transformed CPU programs (5.94 sec), applying optimizations only to transformed CPU programs increases the runtime to 10.53 sec for CuPBoP.

5.3.2 Manual Optimized CPU Programs. Although CuPBoP can achieve higher performance compared to other CUDA-to-CPU solutions, it remains an open question whether it can execute CUDA on CPUs with performance comparable to executing CUDA on GPUs. As described in Table 3, modern CPUs and GPUs possess similar magnitudes of computational resources, leading to the expectation that CUDA programs could achieve comparable performance on both platforms. However, our evaluation of five applications from Hetero-Mark, which involve floating point computations and are easy to estimating FLOPs and memory access, reveals significant insights. The results, illustrated with the Roofline model [67] in Figure 6, show that while the CUDA programs are well-optimized for NVIDIA GPUs—achieving high performance close to the upper bound—the transformed CPU programs fall far short of the CPU's peak performance. Consequently, we conclude that high-performance CUDA programs may not necessarily be transformed into high-performance CPU programs.

To investigate the performance gap between CPU programs transformed from CUDA and manually optimized CPU programs, we use the Cloverleaf application as a case study. This application offers both well-optimized CUDA (for GPU) and OpenMP (for CPU) implementations.

We profile the runtime on Sever-AMD-A30 and show the result in Figure 7. We have two observations. First, the ratio of execution time in CuPBoP and CUDA is different. For the CUDA program, the Halo Exchange takes around 50% total execution time. While in CuPBoP, the Momentum advection becomes the hot spot, which takes around 33% execution time.

Another observation is the runtime distribution between CuPBoP and OpenMP are similar, but the runtime in CuPBoP is longer than OpenMP. We study the functions for the Momentum Advection and show the simplified code in Listing 6. The CUDA program and the OpenMP program have different data mappings: They use different functions (*THARR2D* and *FTNREF2D*) to

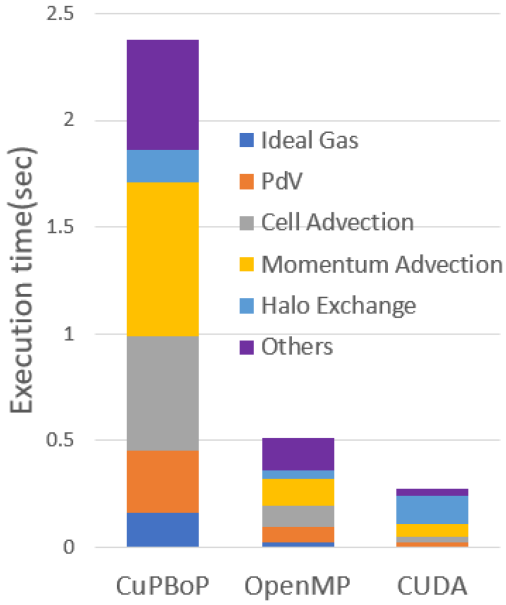


Fig. 7. The execution time of CloverLeaf application. (CuPBoP: 2.374 sec, OpenMP: 0.5162 sec, NVIDIA-A30: 0.2785 sec).

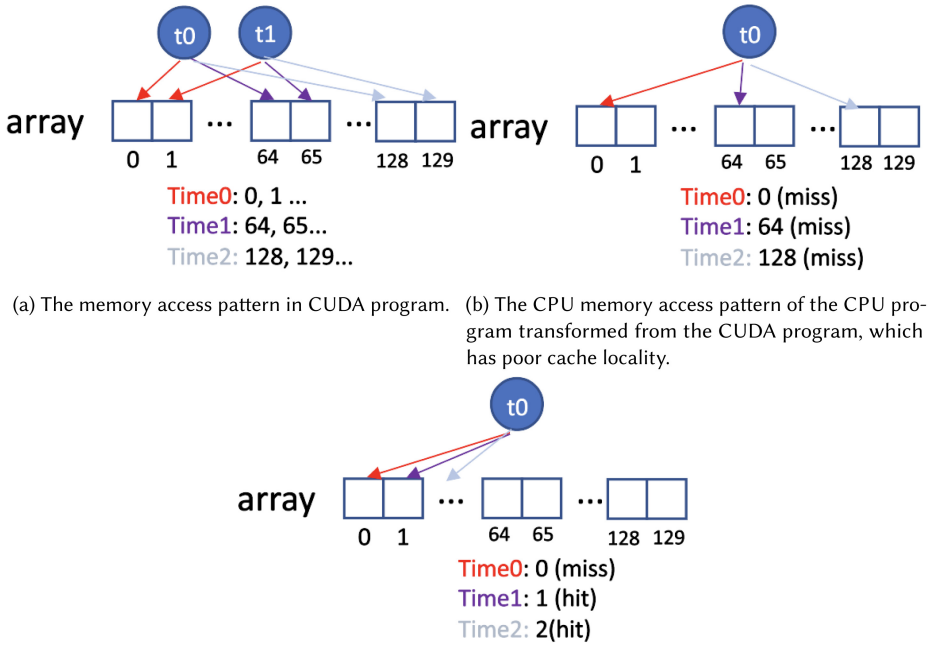
maintain a 2D array. Additionally, OpenMP uses explicit pragma to guide the vectorization. As CuPBoP translates programs from CUDA, the translated CPU program has complicated array access and cannot use the SIMD instructions well.

```
// CUDA
node_flux[THARR2D(0, 0, 1)] = mass_flux_x[THARR2D(0, -1, 1)];
// OpenMP
#pragma omp for private(j)
for (k=y_min;k<=y_max+1;k++) {
    #pragma ivdep
    for (j=x_min-2;j<=x_max+2;j++) {
        node_flux[FTNREF2D(j,k,x_max+5,x_min-2,y_min-2)]=mass_flux_x[FTNREF2D(j,k-1,x_max
     +5,x_min-2,y_min-2)];
    }
}
```

Listing 6. The simplified code of the Momentum Advection function in CUDA and OpenMP.

```
// original CUDA program
uint32_t index = threadIdx.x;
while (index < num_pixels) {
  uint32_t color = pixels[index];
  priv_hist[color]++;
  index += blockDim.x;
}
// transformed CPU program
for(size threadId=0;threadId<BLOCK_SIZE;threadId++) {
  uint32_t index = threadId;
  while (index < num_pixels) {
    uint32_t color = pixels[index]; // poor locality
    priv_hist[color]++;
    index += BLOCK_SIZE;
  }
}
```

Listing 7. The original CUDA program and transformed CPU program for Hist benchmark.



(a) The memory access pattern in CUDA program.

(b) The CPU memory access pattern of the CPU program transformed from the CUDA program, which has poor cache locality.

(c) The CPU memory access pattern can gain high cache locality after the reordering.

Fig. 8. The ideal CUDA memory access pattern may lead to poor CPU locality after the GPU-to-CPU transformation. We could apply memory reordering to gain higher cache locality to speed up the transformed CPU programs.

In addition to the vectorization issue, we have identified another critical memory access issue. We present the original CUDA program and the transformed CPU program for the Hetero-Mark Hist benchmark in Listing 7. We also visualize the memory access patterns for easy understanding in Figure 8. Figure 8(a) shows that each GPU thread accesses memory addresses with a large stride. This pattern enables coalesced memory accesses within a GPU warp (i.e., all memory requests within a warp are sequential). However, this feature will cause a low cache hit rate when the kernel is transformed into a CPU program (Figure 8(b)). By reordering the memory access sequence (Figure 8(c)), the CPU program can achieve a much higher hit rate. A similar transformation can also be used to speed up other examples like the GA. With manual memory access transformation, the LLC hit rate can be significantly improved, as shown in Table 6. Although memory reordering may resemble loop reordering, which involves interchanging the outer and inner loops in Listing 7 (outer loop: line 9, inner loop: line 11), the existing loop interchange transformations cannot handle this case. This limitation arises because the inner loop depends on the induction variable (`threadId`) of the outer loop.

From the evaluation, we demonstrate that CuPBoP exceeds two existing CUDA-to-CPU solutions in terms of coverage and performance. However, there still exists a gap between CPU programs transformed from CUDA and manually optimized CPU programs. We identify two aspects (vectorization and memory access pattern) that hinder the current framework from achieving performance portability and propose a potential solution (memory reordering) to enable CUDA performance portability on CPUs. We do **NOT** apply the manual optimizations on CuPBoP when comparing with other CUDA-to-CPU solutions.

Table 6. For Some CUDA Programs, the Transformed CPU Programs Have Poor Locality

	reordering?	LLC-loads (1e9)	LLC-load misses (1e9)	LLC-stores (1e9)	LLC-store misses (1e9)
HIST	yes	359	165	152	38
	no	37,290	26,656	2,999	62
GA	yes	133	13	80	11
	no	492	148	636	446

Applying the memory access reordering can significantly speed up these CPU programs by gaining high cache locality.

Table 7. The Software Dependency

projects	dependency
DPC++	Intel oneAPI
HIP-CPU	TBB, LLVM, pthreads, C++17
MCUDA	Java, Intel ICC, Cetus
CuPBoP	LLVM, pthreads

CuPBoP only relies on open-source projects.

5.4 Dependency

We present the software dependencies for different projects in Table 7. DPC++ relies on Intel development toolkits, which include the src2src translator (DPCT) and the compiler (DPC++) for compiling SYCL programs to CPU executable files. For CPU backend, Intel oneAPI can only be executed on CPUs that are compatible with Intel 64 architecture [10].

HIP-CPU utilizes Clang to translate CUDA source code to HIP source code. The translated HIP source code can be compiled with official GCC/Clang by setting the include path to HIP-CPU's provided directory. HIP-CPU runtime is based on TBB [8] and pthreads.

MCUDA [65] relies on Cetus [27], a source-to-source compiler, to convert CUDA programs to C programs. As stated in the MCUDA documentation, the translated C programs can only be compiled by the Intel ICC compiler to generate CPU executable files. The Java environment is required to execute MCUDA, as Cetus is written in Java.

Regarding CuPBoP, the compilation part is based on LLVM, and the runtime relies on pthreads. Users can utilize CuPBoP by downloading the pre-built LLVM binary and building CuPBoP from the source code.

5.5 Debug

The CPU executable files generated by CuPBoP can be analyzed by CPU debug toolkits. For example, users can use *valgrind* to find out-of-bound memory access (Listing 8).

However, compared to NVIDIA computer sanitizer [17], there are some limitations of CuPBoP. First, not all CUDA bugs are maintained in the transformed CPU programs. For example, the GPU thread race condition will not exist in generated CPU programs, as all threads within a block will be executed sequentially. Additionally, the output in *valgrind* does not contain the block index and thread index information, so researchers cannot know which thread triggers the bugs.

To get well-organized debug information, we have to also maintain *DWARF* information during compilation transformations. Further work is needed to fully address this topic.

```
// mem_error.cu
__device__ void out_of_bounds_function(void)
{
    *(int*) 0x87654320 = 42;
}
// analyze the generated CPU programs with valgrind
valgrind  --tool=memcheck  ./mem_error
// valgrind output
==4105214== Invalid write of size 4
==4105214==    at 0x4010E9: out_of_bounds_function()
```

Listing 8. CuPBoP can be used with valgrind to detect CUDA memory bugs.

6 CUPBOP FOR AMD GPU

We describe the CUDA to AMD GPU support in CuPBoP as a proof the hardware scalability. The workflow for supporting CUDA on AMD GPUs is similar to CPUs: CuPBoP utilizes Clang to compile CUDA source code into NVVM/LLVM IRs. Then, several compilation transformations are applied to each of them. For the kernel programs, CuPBoP applies transformations to implement local variable and parameter address space conversions, block and thread indexing replacements, constant and shared memory attribute modifications, texture memory transformations, and vector, atomic, and math function substitutions. After all these transformations, the kernel programs are compiled into fat binaries. After that, CuPBoP applies transformations on host programs to replace the embedded CUDA fat binary with the newly generated fat binaries.

The transformed programs are then compiled into object files and linked with CuPBoP-AMD runtime libraries. The CuPBoP-AMD runtime libraries implement the same CUDA built-in functions as CuPBoP-CPU, but the implementation is based on HIP instead of C/C++.

CuPBoP-AMD supports all the CUDA features that are supported on CuPBoP-CPU. Additionally, CuPBoP-AMD supports texture memory and cooperative groups.

6.1 Comparison with HIPIFY

In this section, we compare CuPBoP-AMD with HIPIFY [7], a state-of-the-art source-to-source translation tool developed by AMD that focuses on refactoring existing CUDA source code to use one-to-one mapped HIP APIs.

As HIP is specifically designed to be similar to CUDA, in many cases, HIPIFY can successfully translate CUDA to HIP. Compared with CuPBoP, HIPIFY is lightweight. However, there are some cases where CuPBoP is preferred.

Syntactic sugar: HIPIFY is based on AST analysis or regular expressions, which cannot successfully translate some syntactic sugars that involves complicated code usage. For example, the code shown in Listing 5 contains a syntactic sugar that invokes a CUDA kernel launch, which cannot be translated by HIPIFY. As CuPBoP is based on LLVM IR, all syntactic sugars are compiled into IR instructions and have no effect on the migration process.

Complicated file structures: Many heterogeneous applications prefer to split the kernel programs, storing them in multiple ".cuh" (CUDA) header files and including them from a central ".cu" source file without explicitly specifying them all to the compiler. HIPIFY requires either manual modifications to each source and header file or transformations on all source and header files. However, CuPBoP utilizes Clang to compile the source code. In most cases, during the compilation process, these header files will be extended into a single kernel program (NVVM IR), making CuPBoP more convenient for projects with complicated file structures.

We evaluated CuPBoP and HIPIFY on the Rodinia benchmark to measure coverage and performance. Out of the 23 applications, HIPIFY was able to support 18 of them. Three applications

(hybridsort, kmeans, leukocyte) raised segfaults after translation. These applications use CUDA texture memory. While CUDA's *cudaBindTexture* function could accept a null pointer as an argument, this behavior is illegal in HIP's corresponding function *hipBindTexture*. Thus, the translated HIP code raises segfault errors. On the contrary, CuPBoP correctly supports the translation by generating an auxiliary dummy variable and passing its address as the argument to the *hipBindTexture*. Thus, CuPBoP can support these three functions. In total, CuPBoP supports 21 applications. Two applications (Huffman, srad) are not transformed correctly in either CuPBoP or HIPIFY. We assume this is due to internal differences (e.g., floating-point calculation, memory management) between NVIDIA GPUs and AMD GPUs.

Both HIPIFY and CuPBoP use the same HIP functions on the transformed AMD GPU programs. We evaluated the performance of the translated programs on AMD MI210 GPUs and found that the execution times are mostly within the margin of error. On average, the difference in runtime between the two approaches is less than 1%.

7 RELATED WORKS

7.1 CUDA-on-CPU Solutions

All projects that execute CUDA on CPUs can be categorized into two aspects: debugging and performance.

Debugging: For debugging purposes, these projects focus on integrating with off-the-shelf CPU debug toolkits to obtain detailed information from CUDA programs. Horus [32] and GPUGPU-SIM [18] are PTX simulators that parse the PTX binaries and execute them on CPUs. Cumulus [21] applies source code transformations to translate CUDA to C++. The transformed C++ programs aim for debugging purposes, as they do not utilize multiple-thread execution. VGPU [58] directly uses CPU threads to replace GPU threads. These projects always avoid or decrease the transformations applied to CUDA programs to make it easy for debugging.

Performance: In contrast to debugging projects, some projects focus on executing CUDA on CPUs with high performance by utilizing CPU computational resources. MCUDA [65] proposes compilation transformations to wrap the workload within a CUDA block into a single CPU function. This solution is called the loop solution in Section 3.2. Swan [37] and POCL [39] implement the same algorithm to support executing OpenCL on CPUs. The authors in Reference [53] implement the transformation at the MLIR level, which allows the transformation to be co-executed with other compiler optimizations. Ocelot [28] implements the transformation at the PTX level to avoid recompilation of CUDA source code. COX [36] extends the loop solution to support CUDA warp-level functions. Guo et al. [35] add additional static analysis to correctly translate the implicit warp-level lock steps that widely exist in CUDA programs. The authors in References [44, 64] propose optimizations to better utilize SIMD instructions on the translated programs.

Instead of directly executing CUDA on CPUs, some solutions rely on portable languages. The AMD team proposes HIPIFY [7] to translate CUDA to HIP. To date, two versions of HIPIFY have been released: $hipify - perl$ is a PERL script based on regular expressions that solely performs text replacements from CUDA APIs to HIP APIs; $hipify - clang$ is a Clang-based translator that performs source-to-source refactoring based on the abstract syntax tree analysis. Similarly, Intel proposes DPCT [5] to translate CUDA to SYCL [38], which can be executed on Intel CPUs and Intel GPUs. OpenCL [52, 60, 61] is also a popular target for CUDA transformation. The effectiveness of these source-to-source solutions is highly dependent on the similarity between CUDA and the target languages. The authors in Reference [23] summarize several challenges when migrating CUDA to SYCL. ChipStar [1] is a framework that supports the execution of HIP/CUDA programs on platforms that support SPIR-V. The key insight of ChipStar is that it implements HIP/CUDA

Table 8. Summary of Projects for Executing CUDA on CPUs

Project(s)	Translate from	Translate host programs?	active	Runtime implementation	output
MCUDA [65]	CUDA source	no	no	OpenMP	C/C++ code
Cumulus [21]		yes	yes	no multi-thread	C/C++ code
Swan [37]		no	no	OpenCL	OpenCL code
CU2CL [52] and [60, 61]		yes	yes	OpenCL	OpenCL code
HIPIFY [7]		yes	yes	HIP framework	HIP code
DPCT [5]		yes	yes	Intel oneAPI	DPC++ code
Ocelot [28]	PTX Assembly	yes	no	Hydrazine threading library	Executable files
Horus [32]		simulator			
GPGPU-Sim [18]		simulator			
COX [36]	NVVM IR	no	yes	not provided	LLVM IR
CuPBoP	NVVM/LLVM IR	yes	yes	portable run-time system	Executable files

functions using OpenCL instructions. Since both the input (HIP/CUDA) and output (OpenCL) are SPMD models, this project does not involve SPMD-to-MPMD transformation. HIPCL [20] is another project that implements HIP APIs with OpenCL functions, which can potentially be used to support CUDA on CPUs. Similarly, HIPLZ [69] employs a similar solution to support HIP on Intel GPUs by using Intel Level Zero [3].

We summarize the features of several related works and show in Table 8. We also provide a summary of the two related works that are used for comparison in the evaluation section.

DPC++: DPC++ is based on OpenCL. POCL [39] is the only open-source OpenCL implementation that offers CPU backend support. For the given OpenCL kernels, POCL applies the loop solutions. Similar to CuPBoP, POCL maintains a thread pool and task queue to facilitate kernel launching and execution. Additionally, POCL supports **just-in-time (JIT)** compilation. Instead of keeping the block and grid sizes as runtime variables in the transformed kernels, POCL replaces these variables with actual values during kernel launch. While this approach may introduce higher JIT compilation latency, it enables the transformed kernels to be easily optimized by compilers.

HIP-CPU: HIP-CPU does not involve compiler-level transformations; instead, it provides libraries for HIP kernel functions and HIP runtime functions. It implements the fiber solution, which results in a higher overhead for context switching, compared with the loop solution. Additionally, as HIP-CPU does not apply compilation-level transformations, its performance may be slower. For instance, HIP-CPU conservatively synchronizes all threads before memory movement between the host and devices, regardless of whether these threads read or write memory that has a race condition.

7.2 CUDA-on-FPGA Solutions

FPGA is another popular target for CUDA migration. FCUDA [56] supports executing CUDA programs on FPGAs by providing a source-to-source translator to convert CUDA to parallel C programs that can be compiled using AutoPilot [68]. ML-GPS [57] automatically extracts multilevel granularity parallelism from CUDA programs and maps it to FPGA accelerators. FlexGrip [19] is a GPU architecture optimized for FPGA implementation, capable of directly executing compiled CUDA binary programs. FlexGrip is configurable, allowing developers to easily adjust the number of cores to strike a balance between performance and area cost.

In addition to directly executing CUDA on FPGAs, there are projects that support other portable languages on FPGAs. For example, References [26, 43, 49] propose execute OpenCL programs on FPGA. Reference [33] supports SYCL for Xilinx FPGA. Taking into account the existing source-to-source translators from CUDA to OpenCL and SYCL, these solutions can potentially be used to support CUDA on FPGAs.

8 CONCLUSION

Most existing approaches for running CUDA on non-NVIDIA devices rely on src2src solutions that translate CUDA code to other programming languages. However, these src2src solutions have certain limitations. Building upon these observations, we propose a framework called CuPBoP. Instead of translating CUDA to portable languages, CuPBoP enables CUDA to become a portable language itself. In our evaluation, we compare CuPBoP with two commercial src2src solutions (DPC++ and HIP-CPU) for the CUDA-to-CPU migration and demonstrate that CuPBoP achieves the highest coverage. Additionally, CuPBoP achieves the highest performance in 17 of the 24 applications. Furthermore, we compare CuPBoP with manually optimized CPU programs and analyze the result. Additionally, we showcase the hardware scalability of CuPBoP by supporting CUDA on AMD GPUs.

We believe that CuPBoP holds great promise for HPC developers and provides a unique compilation-runtime co-design approach to explore the possibility of executing CUDA code on non-NVIDIA devices.

ACKNOWLEDGMENTS

Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

We sincerely appreciate AMD, BAH, and NSF CCRI 2016701 for providing the evaluation platforms.

REFERENCES

[1] Michal Babej and Pekka Jääskeläinen. 2020. chipStar. Retrieved from https://github.com/CHIP-SPV/chipStar
[2] NVIDIA. 2008. Compiling and Executing CUDA Programs in Emulation Mode. Retrieved from https://developer.nvidia.com/cuda-toolkit
[3] Intel. 2019. Introduction to the oneAPI Level Zero Interface. Retrieved from https://www.intel.com/content/www/us/en/developer/articles/technical/using-oneapi-level-zero-interface.html
[4] 2020. CloverLeaf. Retrieved from https://github.com/UK-MAC/CloverLeaf
[5] 2020. DPCT. Retrieved from https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compatibility-tool.html
[6] 2020. HIP-CPU. Retrieved from https://github.com/ROCm-Developer-Tools/HIP-CPU
[7] 2020. HIPIFY. Retrieved from https://github.com/ROCm-Developer-Tools/HIPIFY
[8] 2021. IntelTBB. Retrieved from https://github.com/oneapi-src/oneTBB
[9] 2021. Moodycamel::ConcurrentQueue. Retrieved from https://github.com/cameron314/concurrentqueue
[10] 2022. Intel oneAPI Requirements. Retrieved from https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.html#gs.2vokod
[11] 2022. NVIDIA Document. Retrieved from https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#special-registers
[12] 2023. AOCL-BLIS. Retrieved from https://www.amd.com/en/developer/aocl/blis.html
[13] 2023. cuBLAS. Retrieved from https://developer.nvidia.com/cublas
[14] 2023. cuDNN. Retrieved from https://developer.nvidia.com/cudnn
[15] 2023. cuFFT. Retrieved from https://developer.nvidia.com/cufft
[16] 2023. Intel MKL. Retrieved from https://www.intel.com/content/www/us/en/docs/oneapi/programming-guide/2023-0/intel-oneapi-math-kernel-library-onemkl.html
[17] 2023. NVIDIA Compute Sanitizer. Retrieved from https://developer.nvidia.com/nvidia-compute-sanitizer
[18] Tor Aamodt, Wilson W. L. Fung, Ali Bakhoda, George Yuan, Ivan Sham, Henry Wong, Henry Tran, Andrew Turner, Aaron Ariel, Inderpret Singh, Tim Rogers, Jimmy Kwa, Andrew Boktor, and Ayub Gubran Tayler Hetherington. 2012. Gpgpu-Sim 3.x Manual.
[19] Kevin Andryc, Murtaza Merchant, and Russell Tessier. 2013. FlexGrip: A soft GPGPU for FPGAs. In *International Conference on Field-Programmable Technology (FPT'13)*. IEEE, 230–237.
[20] Michal Babej and Pekka Jääskeläinen. 2020. HIPCL: Tool for porting CUDA applications to advanced OpenCL platforms through HIP. In *Proceedings of the International Workshop on OpenCL*. 1–3.

[21] Vera Blomkvist Karlsson. 2021. Cumulus-translating CUDA to Sequential C++: Simplifying the Process of Debugging CUDA Programs.
[22] Germán Castaño, Youssef Faqir-Rhazoui, Carlos García, and Manuel Prieto-Matías. 2022. Evaluation of Intel's DPC++ compatibility tool in heterogeneous computing. *J. Parallel Distrib. Comput.* 165 (2022), 120–129.
[23] Germán Castaño Roldán. 2021. Intel-oneAPI for heterogeneous computing. (2021).
[24] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*. IEEE, 44–54.
[25] Jun Chen, Xule Zhou, and Hyesoon Kim. 2023. CuPBoP-AMD: Extending CUDA to AMD platforms. In *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. 1093–1104.
[26] Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P. Singh. 2012. From OpenCL to high-performance hardware on FPGAs. In *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL'12)*. IEEE, 531–534.
[27] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. 2009. Cetus: A source-to-source compiler infrastructure for multicores. *Computer* 42, 12 (2009), 36–42.
[28] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. IEEE, 353–364.
[29] Christian Dick, Joachim Georgii, and Rüdiger Westermann. 2011. A real-time multigrid finite hexahedra method for elasticity simulation using CUDA. *Simul. Model. Pract. Theor.* 19, 2 (2011), 801–816.
[30] J. Doerfert, M. Jasper, J. Huber, K. Abdelaal, G. Georgakoudis, T. Scogland, and K. Parasyris. 2022. *Breaking the Vendor Lock-performance Portable Programming Through OpenMP as Target Independent Runtime Layer*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).
[31] Jack Dongarra. 2020. *Report on the Fujitsu Fugaku System*. Technical Report. University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICLUT-20-06.
[32] Amr S. Elhelw and Sreepathi Pai. 2020. Horus: A modular GPU emulator framework. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'20)*. IEEE, 104–106.
[33] Andrew Gozillon, Ronan Keryell, Lin-Ya Yu, Gauthier Harnisch, and Paul Keir. 2020. triSYCL for Xilinx FPGA. In *Proceedings of the International Conference on High Performance Computing and Simulation. IEEE*.
[34] Simon Green. 2010. Particle simulation using CUDA. *NVIDIA Whitepaper* 6 (2010), 121–128.
[35] Ziyu Guo, Eddy Zheng Zhang, and Xipeng Shen. 2011. Correctly treating synchronizations in compiling fine-grained SPMD-threaded programs for CPU. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 310–319.
[36] Ruobing Han, Jaewon Lee, Jaewoong Sim, and Hyesoon Kim. 2022. COX: Exposing CUDA warp-level functions to CPUs. *ACM Trans. Archit. Code Optim.* 19, 4 (2022).
[37] Matt J. Harvey and Gianni De Fabritiis. 2011. Swan: A tool for porting CUDA programs to OpenCL. *Comput. Phys. Commun.* 182, 4 (2011), 1093–1099.
[38] Lee Howes and Maria Rovatsou. 2015. SYCL integrates OpenCL devices with modern C++. *Khronos Group* (2015).
[39] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. 2015. PoCL: A performance-portable OpenCL implementation. *Int. J. Parallel Program.* 43, 5 (2015), 752–785.
[40] Aaron Jezghani, Jeffrey Young, Will Powell, Ronald Rahaman, and J. Eric Coulter. 2023. Future computing with the Rogues Gallery. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'23)*. IEEE, 262–269.
[41] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. 2018. Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes. *arXiv preprint arXiv:1807.11205* (2018).
[42] Zheming Jin and Jeffrey S. Vetter. 2023. A benchmark suite for improving performance portability of the SYCL programming model. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'23)*. IEEE, 325–327.
[43] Gangwon Jo, Heehoon Kim, Jeesoo Lee, and Jaejin Lee. 2020. SOFF: An OpenCL high-level synthesis framework for FPGAs. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*. IEEE, 295–308.
[44] Ralf Karrenberg and Sebastian Hack. 2012. Improving performance of OpenCL on CPUs. In *Proceedings of the International Conference on Compiler Construction*. Springer, 1–20.
[45] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (2017), 84–90.

[46] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. IEEE, 75–86.
[47] Changmin Lee, Won Woo Ro, and Jean-Luc Gaudiot. 2014. Boosting CUDA applications with CPU–GPU hybrid computing. *Int. J. Parallel Program.* 42, 2 (2014), 384–404.
[48] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2015. SKMD: Single kernel on multiple devices for transparent CPU-GPU collaboration. *ACM Trans. Comput. Syst.* 33, 3 (2015), 1–27.
[49] Mingjie Lin, Ilia Lebedev, and John Wawrzynek. 2010. OpenRCL: Low-power high-performance computing with reconfigurable devices. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. IEEE, 458–463.
[50] Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Müller-Wittig. 2008. Accelerating molecular dynamics simulations using Graphics Processing Units with CUDA. *Comput. Phys. Commun.* 179, 9 (2008), 634–641.
[51] Robert Love. 2013. *Linux System Programming, 2nd Edition*. O'Reilly Media.
[52] Gabriel Martinez, Mark Gardner, and Wu-chun Feng. 2011. CU2CL: A CUDA-to-opencl translator for multi-and many-core architectures. In *Proceedings of the IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE, 300–307.
[53] William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. 2023. High-performance GPU-to-CPU transpilation and optimization via high-level parallel constructs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 119–134.
[54] Aaftab Munshi. 2009. The OpenCL specification. In *Proceedings of the IEEE Hot Chips 21 Symposium (HCS'09)*. IEEE, 1–314.
[55] NVIDIA. 2012. NVVM. Retrieved from https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html#abstract
[56] Alexandros Papakonstantinou, Karthik Gururaj, John A. Stratton, Deming Chen, Jason Cong, and Wen-Mei W. Hwu. 2009. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *Proceedings of the IEEE 7th Symposium on Application Specific Processors*. IEEE, 35–42.
[57] Alexandros Papakonstantinou, Yun Liang, John A. Stratton, Karthik Gururaj, Deming Chen, Wen-Mei W. Hwu, and Jason Cong. 2011. Multilevel granularity parallelism synthesis on FPGAs. In *Proceedings of the IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 178–185.
[58] Atmn Patel, Shilei Tian, Johannes Doerfert, and Barbara Chapman. 2021. A virtual GPU as developer-friendly OpenMP offload target. In *Proceedings of the 50th International Conference on Parallel Processing Workshop*. 1–7.
[59] S. John Pennycook and Jason D. Sewall. 2021. Revisiting a metric for performance portability. In *Proceedings of the International Workshop on Performance, Portability and Productivity in HPC (P3HPC'21)*. IEEE, 1–9.
[60] Hugh Perkins. 2017. CUDA-on-CL: A compiler and runtime for running NVIDIA® CUDA™ C++ 11 applications on OpenCL™ 1.2 Devices. In *Proceedings of the 5th International Workshop on OpenCL*. 1–4.
[61] Paul Sathre, Mark Gardner, and Wu-chun Feng. 2019. On the portability of CPU-accelerated applications via automated source-to-source translation. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. 1–8.
[62] David Schneider. 2022. The exascale era is upon us: The frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second. *IEEE Spect.* 59, 1 (2022), 34–35.
[63] Esteban Stafford, Borja Pérez, Jose Luis Bosque, Ramón Beivide, and Mateo Valero. 2017. To distribute or not to distribute: The question of load balancing for performance or energy. In *Proceedings of the 23rd International Conference on Parallel and Distributed Computing: Parallel Processing (EURO-PAR'17)*. Springer, 710–722.
[64] John A. Stratton, Hee-Seok Kim, Thoman B. Jablin, and Wen-Mei W. Hwu. 2013. Performance portability in accelerated parallel kernels. *Cent. Reliab. High-perf. Comput.* (2013).
[65] John A. Stratton, Sam S. Stone, and Wen-Mei W. Hwu. 2008. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*. Springer, 16–30.
[66] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-Mark, a benchmark suite for CPU-GPU collaborative computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'16)*. IEEE, 1–10.
[67] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.
[68] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. 2008. AutoPilot: A platform-based ESL synthesis system. *High-level Synth.: From Algor. Digit. Circ.* (2008), 99–112.
[69] Jisheng Zhao, Colleen Bertoni, Jeffrey Young, Kevin Harms, Vivek Sarkar, and Brice Videau. 2023. HIPLZ: Enabling performance portability for exascale systems. *Concurr. Computat.: Pract. Exper.* 35, 25 (2023), e7866.

Received 17 September 2023; revised 25 March 2024; accepted 6 April 2024