# Scaling GPU-to-CPU Migration for Efficient Distributed Execution on CPU Clusters

Ruobing Han
Georgia Institute of Technology
Atlanta, USA
rhan38@gatech.edu

Hyesoon Kim
Georgia Institute of Technology
Atlanta, USA
hyesoon@cc.gatech.edu

## Abstract

The growing demand for GPU resources has led to widespread shortages in data centers, prompting the exploration of CPUs as an alternative for executing GPU programs. While prior research supports executing GPU programs on single CPUs, these approaches struggle to achieve competitive performance due to the computational capacity gap between GPUs and CPUs.

To further improve performance, we introduce CuCC, a framework that scales GPU-to-CPU migration to CPU clusters and utilizes distributed CPU nodes to execute GPU programs. Compared to single-CPU execution, CPU cluster execution requires cross-node communication to maintain data consistency. We present the CuCC execution workflow and communication optimizations, which aim to reduce network overhead. Evaluations demonstrate that CuCC achieves high scalability on large-scale CPU clusters and delivers runtimes approaching those of GPUs. In terms of cluster-wide throughput, CuCC enables CPUs to achieve an average of 2.59× higher throughput than GPUs.

***CCS Concepts:*** • **Computing methodologies → Distributed computing methodologies**.

***Keywords:*** compiler optimization, GPU-to-CPU migration, CPU cluster

**ACM Reference Format:**
Ruobing Han and Hyesoon Kim. 2026. Scaling GPU-to-CPU Migration for Efficient Distributed Execution on CPU Clusters. In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '26), January 31 – February 4, 2026, Sydney, NSW, Australia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3774934.3786435

## 1 Introduction

Numerous GPU applications are released weekly in fields such as artificial intelligence [24, 45] and high-performance computing [5, 14]. The growing demand for GPU resources, coupled with supply chain shortages, has significantly constrained their availability [9, 20, 39].

Data center maintainers frequently observe an imbalance in usage between CPUs and GPUs. We measure the utilization of four CPU partitions and four GPU partitions in the Georgia Tech PACE cluster. By monitoring the Slurm scheduling system, we record the waiting time of all jobs submitted between March 2nd and 8th, 2025 in Figure 1. The waiting time represents the duration jobs wait for resources to become available for execution. Figure 1 shows that CPU partitions experience significantly shorter waiting times than GPU partitions. This indicates that while users wait for GPU resources, a substantial number of CPUs remain idle.

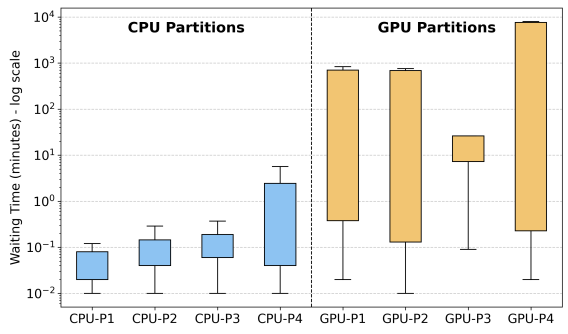


**Figure 1.** Waiting times for CPU and GPU partitions.

The imbalance in CPU/GPU usage motivates researchers to explore using CPUs to alleviate the GPU shortage. Researchers [7, 16, 21, 23, 32, 38, 42] propose compiler and runtime solutions. With these optimizations, GPU programs can be executed on **single** CPUs with high performance.

However, a gap still exists between GPU and CPU runtimes due to differences in computational capacity. CPUs are designed to support a broad range of general applications and cannot match the computational power of GPUs, which are optimized for high-throughput workloads. For instance, in 2020, NVIDIA released the A100 GPU, which delivers 19.5 TFLOP/s for single-precision floating-point computation. In contrast, one of the most advanced CPUs released a year later, AMD EPYC 7713, achieves only 4.096 TFLOP/s. As GPUs continue to integrate more computational units, the performance gap between CPUs and GPUs is widening.

ACM ISBN 979-8-4007-2310-0/2026/01
https://doi.org/10.1145/3774934.3786435

Given that CPUs are typically more accessible in data centers, this paper explores a new direction: executing GPU programs on CPU clusters. By leveraging multiple CPU nodes, this approach aggregates greater computational resources, bringing the overall capacity closer to that of a single GPU and enabling GPU programs to migrate to CPUs without significant performance loss.

Compared to a single CPU, CPU cluster migration is significantly more challenging, as CPU clusters and GPUs have different memory models. The GPU programming model follows a shared memory model, where all threads can access a global memory space, and data consistency is implicitly maintained by hardware. In contrast, CPU clusters use a distributed memory model, where each node has its own memory space. Thus, to support the migration of GPU programs (shared memory model) to CPU clusters (distributed memory model), auxiliary cross-node communication operations are required to ensure data consistency.

The distributed shared memory (DSM) problem, which aims to provide a shared memory model on a distributed system, is a classical challenge for which researchers have proposed numerous solutions [4, 11, 26, 33, 46]. However, existing DSM solutions are designed for traditional CPU programs, which are typically Multiple Program Multiple Data (MPMD) and contain relatively few memory accesses with irregular patterns. Consequently, peer-to-peer communication is often used to provide flexibility. When these DSM solutions are applied to programs migrated from GPUs, which contain a vast number of memory accesses, they introduce significant communication overhead that degrades overall performance. A detailed analysis is provided in Section 3.1.

In this paper, we propose CuCC (CUDA on CPU Clusters), a new solution **tailored** for migrating GPU programs to CPU clusters. GPU programs follow a Single Program Multiple Data (SPMD) model, where all threads execute the same programs. This results in memory access patterns that are highly regular. Our solution exploits this regularity by coalescing multiple memory accesses into a single, larger operation and uses collective communication primitives to achieve high bandwidth to lower network overhead.

We implement CuCC as an end-to-end framework that translates CUDA programs into CPU cluster executables. We demonstrate that CuCC is 12.81× faster than existing single-CPU solutions and achieves runtimes approaching those of GPUs. The contributions are summarized as follows:

- Propose a solution for executing GPU programs on CPU clusters with low communication overhead.
- Introduce an auxiliary compiler analysis for GPU-to-CPU-cluster migration.
- Implement an end-to-end framework for migrating GPU programs to CPU clusters.
- Evaluate the proposed solution on CPU clusters and compare its performance against GPUs.

Although this paper focuses on migrating CUDA applications to CPU clusters, the proposed approach is general and not restricted to any specific GPU programming language.

## 2 Background

### 2.1 Programming Model

**2.1.1 GPU.** The programming model is structured with two levels of parallelism. The GPU block represents coarse-grained parallelism, while each GPU block consists of a fixed number of GPU threads that provide fine-grained parallelism. Importantly, there are no dependencies among execution units at either level, enabling highly parallel execution.

All GPU threads access the same memory space, and modifications made by one thread are visible to all other threads. [1]

**2.1.2 CPU Cluster.** A CPU cluster consists of multiple CPU nodes (Figure 2). All nodes are connected through networks such as an InfiniBand. In contrast to the GPU model, where all threads access the same memory space, CPU clusters follow a distributed memory model, with each node maintaining its own memory space. As a result, cross-node network communication is required to ensure memory consistency among the distributed nodes.

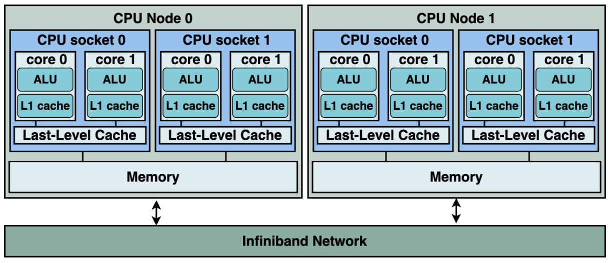


**Figure 2.** CPU Cluster Structure.

### 2.2 Executing GPU Programs on Single CPUs

GPUs are designed to execute a large number of lightweight tasks, while CPUs are optimized for a smaller number of heavier tasks. To address this disparity, researchers [21, 22, 32, 38, 42] apply compiler transformations to wrap the workload within a CUDA block into a CPU function, which is then executed by a CPU thread.

An example is shown above. For the GPU program (Listing 1), 5 * 256 GPU threads are invoked during execution. Since thread operation overhead on CPUs is significantly higher than on GPUs, CPUs cannot efficiently support the same number of threads. With the proposed compiler optimization (Listing 2), the entire workload of a GPU block is

[1]We only discuss GPU global memory for CPU cluster migration. The GPU shared memory and local memory do not require cross-node communication to maintain consistency, as all threads within a GPU block are scheduled to the same CPU node for execution.

mapped to a single function (line 1), reducing the requirement to just 5 CPU threads (line 10).

```
1 #define N 1200
2 __global__ void vec_copy(char *src, char *dest) {
3   int id = blockDim.x * blockIdx.x + threadIdx.x;
4   if(id < N)
5     dest[id] = src[id];
6 }
7 int main() {
8   vec_copy<<<ceil(N/256), 256>>>(src, dest);
9 }
```

**Listing 1.** Original GPU program.

```
1  void vec_copy(char *src, char *dest, int block_id) {
2  #pragma omp simd
3    for(int thread_id=0;thread_id<256;thread_id++){
4      int id = 256 * block_id + thread_id;
5      if (id < N)
6          dest[id] = src[id];
7    }
8  }
9  int main() {
10 #pragma omp parallel for
11   for(int block_id=0;block_id<ceil(N/256);block_id++)
12     vec_copy(src, dest, block_id);
13 }
```

**Listing 2.** Transformed single-node CPU program.

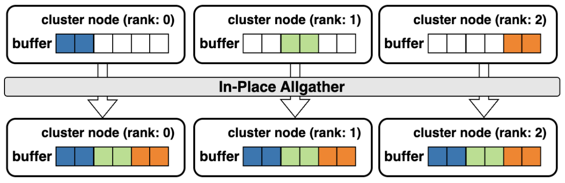
The transformed CPU program utilizes CPU resources for high performance. In Listing 2, a for-loop is introduced (line 3), where each iteration represents a GPU thread. Since there are no dependencies among these iterations, the for-loop is well-suited for optimization using CPU SIMD instructions. Similarly, the host program contains a parallelized for-loop (line 11), where each iteration corresponds to a GPU block. This for-loop can be executed by multiple CPU threads to maximize performance.

For single-CPU migration, GPU global memory is mapped to CPU heap memory, which is accessible to all CPU threads, with consistency maintained by the OS and hardware. However, when extending to a CPU cluster, CPU threads are distributed across multiple nodes, and no unified memory space exists among all CPU threads. Thus, auxiliary communication operations are required to maintain consistency.

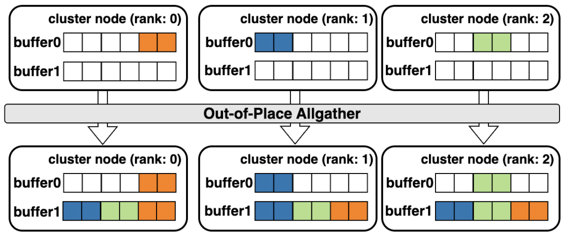
### 2.3 Allgather Communication

Our solution utilizes Allgather communication to maintain data consistency across CPU nodes. Allgather collects data from each node, concatenates them sequentially, and returns the concatenated data to all nodes (Figure 3).

From a data placement perspective, Allgather can be categorized into two types: `in-place` and `out-of-place`. In in-place Allgather (Figure 3a), the input and output share the same buffers, so the local data in the input buffer does not need to be moved to another location. In contrast, out-of-place Allgather (Figure 3b) uses separate buffers for input and output. Processing the output buffer requires not only communication between nodes but also local memory movement from the input buffer to the output buffer. Additionally, out-of-place Allgather requires two buffers, resulting in double memory usage compared to in-place Allgather.



**(a)** In-place Allgather.



**(b)** Out-of-place Allgather.

**Figure 3.** Allgather communication.

In addition to data placement, we observe that data distribution also affects communication overhead. Specifically, a `balanced` Allgather, where all distributed nodes have the same data size, is typically faster than an `imbalanced` Allgather. For example, in a 2-node cluster with a total data size of $N$ GB, a balanced Allgather—where each node holds $\frac{N}{2}$ GB—usually outperforms an imbalanced Allgather, where one node has $\frac{N}{4}$ GB and the other has $\frac{3N}{4}$ GB.

Based on our network evaluation, we observe that balanced-in-place Allgather consistently achieves the highest performance. Therefore, CuCC utilizes balanced-in-place Allgather communication to maintain data consistency.

## 3 Problem Statement and Solution

### 3.1 Challenges of Existing Solutions

It is challenging to migrate the GPU shared memory model to the distributed memory space of CPU nodes. A possible solution is to first migrate a GPU program to a single-CPU program. Then, scale this single-CPU program to a CPU cluster using DSM frameworks by replacing local memory accesses with distributed memory accesses. An example of migrating the program in Listing 1 with a popular DSM solution, PGAS, is shown in Listing 3.

Although state-of-the-art PGAS solutions [4, 11] integrate network optimizations like GASNet-EX [8] and RDMA, they perform poorly for GPU program migration due to heavy communication overhead. For example, Listing 3 introduces 1200 remote memory accesses (line 7), where each access is only 1 byte. This large number of fragmented communications limits overall performance. We evaluate the performance on a 32-node cluster (Figure 4); most GPU programs do not achieve high scalability, and some even slow down when scaled to distributed nodes, as the communication overhead significantly exceeds the performance gains.

```
void vec_copy(char* src, pgas::global_ptr<char> dest,
              int block_id){
  for(int tid=0;tid<256;tid++) {
    int id = 256 * block_id + tid;
    if(id < N)
      // Async one-side remote memory access
      pgas::remote_put(dest + id, src[id]);
  }
}
int main() {
  // Cluster-level global variable
  pgas::global_ptr<char> global_dest(N);
  // Distributed Execution
  int c_rank = pgas::rank_me();
  int c_size = pgas::rank_n();
  int local_size = ceil(N/256)/c_size;
  for(int bid=local_size * c_rank;
      bid<local_size * (c_rank+1); bid++)
    vec_copy(src, global_dest, bid);
}
```

**Listing 3.** The PGAS migration for GPU program in Listing 1.

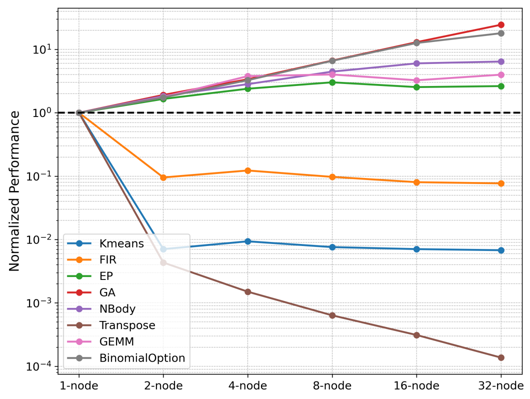


**Figure 4.** Performance of CPU cluster migration using PGAS.

## 3.2 Insight of CuCC

As summarized in Section 3.1, GPU programs contain a large number of threads, and each thread typically issues a small number of memory accesses. When migrated with existing solutions, this massive number of fine-grained memory accesses turns into a large volume of fragmented network communications, which introduces significant overhead.

We propose a new solution, CuCC, that executes with low network overhead. The insight is that, since GPU programs follow the SPMD model, all threads execute the same programs, differing only by their thread index. This leads many GPU programs to contain memory access patterns where sequential threads access consecutive memory locations. Therefore, instead of issuing a separate network communication for each GPU thread's memory access, CuCC coalesces all memory accesses within a GPU block and services them with a single **coarse-grained** network operation. Additionally, as all GPU blocks execute the same program, their memory accesses are highly symmetric. This symmetry allows CuCC to use **collective** communication primitives.

## 4 CuCC Execution Workflow

To execute a GPU kernel on CPU cluster, CuCC follows a three-phase workflow. In Section 6, we prove that the workflow is theoretically capable of supporting the execution of all types of GPU programs on CPU clusters.

1. **Partial Block Execution**: In this phase, each CPU node executes a distinct set of GPU blocks in parallel. These GPU blocks write to non-overlapping memory locations, with all blocks writing the same amount of data. Importantly, some GPU blocks may not be executed during this phase. Specifically, if a block does not meet the requirements introduced in Section 6, its execution is deferred to the third phase. These deferred blocks are referred to as `callback blocks`.
2. **Balanced-In-Place Allgather**: After the first phase, each CPU node holds a unique memory copy. To ensure consistency across the cluster, balanced-in-place Allgather is applied to synchronize memory spaces.
3. **Callback Block Execution**: After Allgather, all CPU nodes have an identical memory copy. The `callback blocks`, which were not executed in the first phase, are then executed independently by all CPU nodes. Since all CPU nodes execute the same set of blocks, the memory space remains identical across the cluster.

We use GPU kernel in Listing 1 to illustrate the workflow. The program consists of five GPU blocks: blocks 0-3 each write 256 elements, while block 4 writes only 176 elements to *dest*. Figure 5 illustrates the workflow for executing the GPU program on a two-node CPU cluster. Before kernel execution, both CPU nodes have identical copies of the memory space.

During the partial block execution phase, GPU blocks are split between CPU nodes for execution. To ensure that the results can later be synchronized using balanced-in-place Allgather, CuCC assigns blocks 0 and 1 to Node 0 and blocks 2 and 3 to Node 1. Each CPU node then writes 512 elements to the *dest* array locally. GPU block 4 is designated as a callback block and is not executed during this phase.

After the first phase, each CPU node holds a different copy of the memory space. To synchronize the memory spaces, CuCC invokes balanced-in-place Allgather across the CPU cluster. After this communication, both CPU nodes have identical memory spaces, equivalent to executing GPU blocks 0 through 3 independently on each node.

Finally, in the callback block execution phase, each CPU node executes the callback block (GPU block 4) independently. From a sequential perspective, the CuCC execution workflow is equivalent to executing all callback blocks after the other blocks. Since the GPU programming model does not enforce a specific GPU block execution order, the results will be consistent with those produced by the GPUs.

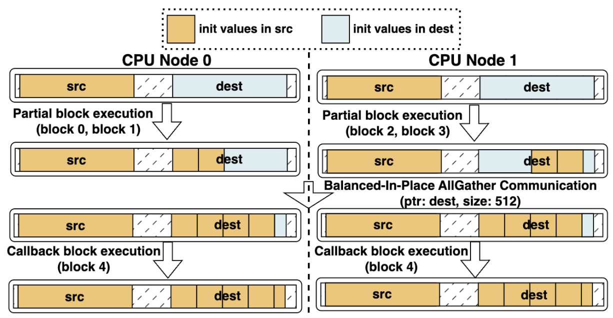

init values in src
init values in dest
CPU Node 0
CPU Node 1
src
dest
Partial block execution (block 0, block 1)
Partial block execution (block 2, block 3)
Balanced-In-Place AllGather Communication (ptr: dest, size: 512)
Callback block execution (block 4)
Callback block execution (block 4)

**Figure 5.** CuCC execution workflow for Listing 1.

## 5 Implementation

CuCC consists of both compilation and runtime components. CuCC first compiles the GPU source code to LLVM IR, then applies compiler analysis and transformations to generate CPU object files. These object files are subsequently linked with the CuCC runtime library to produce CPU cluster executables. The generated CPU cluster program follows the workflow introduced in Section 4.

Figure 6 illustrates the process of migrating the GPU program in Listing 1. All analysis and transformations are applied at the LLVM IR level. The source code shown in Figure 6 is provided for illustrative purposes only.

The GPU program is first analyzed by a compiler analysis, *Allgather distributable analysis* (Section 6), to collect metadata used for generating CPU executable. Metadata values are based on symbolic analysis. Thus, for programs with runtime-dependent values (e.g., dynamic GPU block size), CuCC can still perform the migration.

With the metadata, CuCC applies a **template-based** approach to generate CPU host module. The template consists of three code sections corresponding to three phases in the workflow. The first code section (Figure 6 lines 1–4 in CPU host module) represents the Partial Block Execution phase. To generate this section, CuCC analyzes the GPU host module to determine the grid size ($ceil(N/256)$) and retrieves the tail-divergent information from the metadata. Using these values, CuCC generates the instruction (line 1) to calculate the number of GPU blocks to be executed by each CPU node. This variable (`p_size`) is also used to generate the Callback Block Execution section (line 6–8). The second code section (line 5) represents the Balanced-In-Place Allgather phase. To generate this section, CuCC retrieves metadata information ($mem\_ptr$, $unit\_size$) to determine the variables and their sizes that need to be communicated.

In CuCC, all GPU threads within a GPU block are always executed by the same CPU node. Thus, for CPU kernel module generation, which corresponds to the execution of a single GPU block, CuCC utilizes the same compiler transformations as existing GPU-to-single-CPU solutions. CuCC is developed by extending the codebase of CuPBoP [21], a GPU-to-single-CPU project reported to achieve high performance on single CPUs. Therefore, the single-node performance is equivalent to that of CuPBoP, which is used as the baseline.

The transformed modules are linked with the CuCC runtime library, which provides implementations of cluster operations. In our evaluation, CuCC utilizes MPI primitives to implement these functions.

## 6 Allgather Distributable Analysis

To generate CPU programs that follow the CuCC three-phase workflow, the most important decision is how to assign GPU blocks to each CPU node. To make balanced-in-place Allgather feasible for maintaining consistency, CuCC must distribute GPU blocks across CPU nodes in the partial block execution phase such that each node performs the same amount of memory writes (*balanced*) and the order of memory writes aligns with the cluster ranks of the CPU nodes (*in-place*). We propose a compiler analysis, *Allgather distributable analysis*, to analyze GPU programs and distribute workloads in alignment with the requirements.

### 6.1 Terminology

**Definition: Write Interval**

The *Write Interval* of a GPU thread represents the range of global memory addresses written by that thread. For a GPU block, the *Write Interval* can be expressed as:

$$\text{write\_interval}(block) = \bigcup_{t \in block} \text{write\_interval}(t),$$

where $t$ denotes a thread within $block$.

For a set of GPU blocks $\mathbf{B} = \{\text{block}_1, \text{block}_2, \ldots, \text{block}_K\}$, the *Write Interval* is the union of the Write Intervals of all blocks in the set:

$$\text{write\_interval}(\mathbf{B}) = \bigcup_{block \in \mathbf{B}} \text{write\_interval}(block).$$

As an example, in the GPU kernel shown in Listing 1, the Write Intervals for block 0 through 4 are:

$$[\text{dest}, \text{dest} + 256), [\text{dest} + 256, \text{dest} + 512), .., [\text{dest} + 1024, \text{dest} + 1200).$$

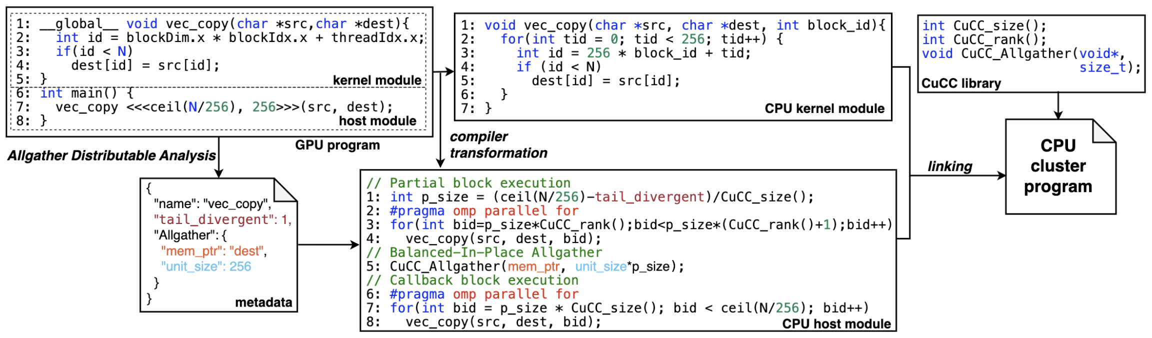
CuCC considers only the write interval for GPU global memory access, as GPU local and shared memory accesses do not require cross-node communication.

**Definition: Allgather Distributable**

Let a set of blocks $\mathbf{B} = \{\text{block}_1, \text{block}_2, \ldots, \text{block}_K\}$ represent the GPU blocks of a GPU kernel. The GPU kernel is considered *Allgather distributable* for an $N$-node cluster if there exist a subset $\mathbf{C} \subseteq \mathbf{B}$, and the set difference $\mathbf{B} - \mathbf{C}$ can be partitioned into $N$ disjoint subsets $S = \{S_i\}$, where $i = 1, \ldots, N$, such that the following conditions are satisfied:

1. **Equal Length of Write Intervals**: All subsets $S_i$ have write intervals of equal length:

$$\text{len}(\text{write\_interval}(S_i)) = \text{len}(\text{write\_interval}(S_j)), \quad \forall S_i, S_j \in S.$$



```
__global__ void vec_copy(char *src,char *dest){
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if(id < N)
    dest[id] = src[id];
}
int main() {
  vec_copy <<<ceil(N/256), 256>>>(src, dest);
}
```

```
void vec_copy(char *src, char *dest, int block_id){
  for(int tid = 0; tid < 256; tid++) {
    int id = 256 * block_id + tid;
    if (id < N)
      dest[id] = src[id];
  }
}
```

```
int CuCC_size();
int CuCC_rank();
void CuCC_Allgather(void*,
                    size_t);
```

```
{
  "name": "vec_copy",
  "tail_divergent": 1,
  "Allgather": {
    "mem_ptr": "dest",
    "unit_size": 256
  }
}
```

```
// Partial block execution
int p_size = (ceil(N/256)-tail_divergent)/CuCC_size();
#pragma omp parallel for
for(int bid=p_size*CuCC_rank();bid<p_size*(CuCC_rank()+1);bid++)
  vec_copy(src, dest, bid);
// Balanced-In-Place Allgather
CuCC_Allgather(mem_ptr, unit_size*p_size);
// Callback block execution
#pragma omp parallel for
for(int bid = p_size * CuCC_size(); bid < ceil(N/256); bid++)
  vec_copy(src, dest, bid);
```

**Figure 6.** GPU-to-CPU-cluster migration process in CuCC.

2. **Disjoint Write Intervals**: The write intervals of any two subsets are disjoint:

$$\text{write\_interval}(S_i) \cap \text{write\_interval}(S_j) = \emptyset, \quad \forall S_i \neq S_j \in S.$$

3. **No Gaps between Write Intervals**: The union of all write intervals in the subsets must cover the entire write interval of the set difference:

$$\text{len}(\text{write\_interval}(\mathbf{B} - \mathbf{C})) = \sum_{S_i \in S} \text{len}(\text{write\_interval}(S_i)).$$

Allgather distributable GPU kernels can be executed on CPU clusters using the three-phase workflow. In the partial block execution phase, each CPU node executes the workload of a subset in $S$. Then, a balanced-in-place Allgather operation is invoked to synchronize the memory space across all CPU nodes, with a length equal to len(write_interval($S_i$)). Finally, in the callback block execution phase, the GPU blocks in **C** are executed independently on each CPU node.

Theoretically, all GPU kernels are Allgather distributable. Even GPU kernels that contain irregular memory access can still satisfy the definition of Allgather distributable by setting $C$ equal to $B$. We refer to these kernels as *trivial Allgather distributable*. When three-phase workflow executes these trivial kernels, the first and second phases do not perform any work, and all GPU blocks are executed in the third phase, similar to GPU-to-single-CPU execution. With the generalization of Allgather distributable, CuCC is theoretically capable of supporting all types of GPU programs.

In this paper, we primarily discuss non-trivial kernels that benefit from distributed execution. Unless explicitly stated otherwise, all references to Allgather distributable refer to this non-trivial subset. In Section 7.1, we analyze real-world GPU kernels in AI/HPC applications and demonstrate that a large number of them are non-trivial Allgather distributable and can be accelerated by cluster execution.

## 6.2 Compiler Analysis Implementation

Given the abstract definition, implementing a compiler analysis to detect the Allgather distributable property poses challenges. To address this, we break down the Allgather distributable criteria into a series of conditions suitable for static analysis. During compiler analysis, CuCC identifies all write instructions that target GPU global memory. For each write instruction, the following conditions are checked.

1. When treating the GPU block index and block size as constants, the index of the write position is an affine function of the thread index, with an invariant coefficient and intercept.
2. The write instruction is not enclosed within conditional statements with thread-variant conditions, unless the conditional statements are *tail divergent*.
3. When treating the thread index and block size as constants, the index of the write position is an affine function of the block index with a positive coefficient.

The first condition ensures that each block writes the same number of bytes. In most real GPU applications, each thread writes to a memory location determined by its global index (i.e., $blockIdx.x \times blockDim.x + threadIdx.x$). When the memory write index is an affine function of the thread index, each thread writes the same amount of data. Since the GPU programming model enforces that all blocks contain the same number of threads, this guarantees that all blocks write the same number of bytes to memory.

The second condition extends the first condition to cases where write instructions are enclosed within conditional statements (e.g., if-else statements). If the conditional statement is thread-variant, it cannot be guaranteed that each block has the same number of threads executing the write instruction; thus, blocks may write different amounts of bytes, which is not suitable for three-phase workflow.

We observe that the second condition frequently excludes GPU applications that satisfy all other conditions. For example, the GPU program in Figure 6 contains a global memory write, where the write index is an affine function of both the thread index and block index. Nevertheless, this write instruction is enclosed within an if-statement with a thread-variant condition (`id<N`), causing it to fail the second condition.

To enable more GPU kernels to be migrated to CPU clusters, we relax the second condition and introduce a concept called **tail divergence**. The key insight is that a specific if-statement pattern is widely present in GPU programs. When the output data size is not a multiple of the block size, GPU programs include if-statements to filter out out-of-bound memory accesses. These if-statements evaluate to `True` for all blocks except the last block. We refer to such if-statements as **tail divergent**, as they only diverge at the tail block. The GPU kernel in Figure 6 is tail divergent, as the if-statement (line 3) can evaluate to false only in the last block. For GPU kernels with tail-divergent write instructions, the last block can be designated as a callback block, while the remaining blocks write an equal number of bytes.

The first and second conditions ensure that all GPU blocks (except the tail block) write the same amount of data, a requirement for achieving *balanced*. The third condition, on the other hand, ensures that the write locations increase linearly with the GPU block index, which is necessary for achieving $in-place$. This enables CuCC to partition the GPU blocks evenly in ascending order and assign each partition to the CPU node corresponding to its cluster rank.

Kernels that satisfy all three conditions are classified as Allgather distributable, and the compiler records the corresponding information in the metadata. As shown in Figure 6, the metadata includes tail divergence (*tail_divergent*), the memory variables that require communication (*mem_ptr*), and the number of bytes each block writes (*unit_size*). This metadata is then used to generate CPU cluster executable.

The conditions form a sufficient but not necessary condition for the actual Allgather distributable criteria. As a result, the analysis may produce false negatives. In CuCC, GPU kernels mistakenly identified as not Allgather distributable are executed independently by all CPU nodes. This ensures that false-negative cases still maintain correctness. Despite the potential for false negatives, our evaluation demonstrates that these conditions accurately identify Allgather distributable kernels in real-world benchmarks.

## 7 Evaluation

We use two CPU clusters: a *Thread-Focused* cluster that features CPUs with high thread-level parallelism, and a *SIMD-Focused* cluster that is equipped with CPUs supporting wide SIMD instructions. It is important to note that while these names highlight specific architectural strengths, both CPUs support SIMD instructions and multi-core execution; we enable both optimizations on both clusters. Both clusters are connected via a 100 Gb/s InfiniBand network with RDMA support. Detailed specifications are provided in Table 1.

**Table 1.** Cluster Specifications.

| Name | Nodes | Single Node Config. | Year | Cores/ SMs | FLOPs (Tera) | Network |
|---|---|---|---|---|---|---|
| **SIMD-Focused** | 32 | 2 × Intel 6226 | 2019 | 24 | 4.15 | 100 Gbps IB |
| **Thread-Focused** | 4 | 2 × AMD 7713 | 2021 | 128 | 8.19 | 100 Gbps IB |
| **A100 GPU** | 1 | NVIDIA A100 | 2020 | 108 | 19.5 | N/A |
| **V100 GPU** | 1 | NVIDIA V100 | 2017 | 80 | 15.7 | N/A |

For performance evaluation, to reduce noise, we filter out GPU programs with kernel execution times less than 100 ms on an NVIDIA A100 GPU. Each experiment is executed seven times, and the median is reported as the final result.

### 7.1 Coverage Evaluation

We analyze the generalization of Allgather distributable in real-world GPU kernels. We analyze the kernels in two popular AI models: BERT [15], for Natural Language Processing, and Vision Transformer (ViT) [18], for Computer Vision. Since many GPU programs in AI applications are generated by Deep Learning Compilers, we compile the PyTorch implementations of these models with Triton [44] to generate GPU programs (NVVM IR) and analyze them. We also analyze GPU programs implemented manually in CUDA from Hetero-Mark GPU benchmarks [43] for HPC applications.

The coverage is demonstrated in Figure 7. All 21 kernels in the ViT and BERT models are Allgather distributable. The high coverage is due to the kernels being lowered from the Triton language. Compared to low-level GPU programming languages like CUDA and OpenCL, Triton provides a more abstract programming interface. For example, Triton does not support inter-block barriers, which encourages the generation of GPU programs with regular memory access patterns that do not have data races between blocks, making it favorable to execute these blocks in distributed nodes.

On the other hand, the Hetero-Mark benchmark contains manually written CUDA kernels, each with a unique code structure. In this benchmark, 8 of the 13 GPU kernels are Allgather distributable. Of the remaining five, four have memory access patterns that overlap the written interval, which makes it difficult to maintain data consistency in a distributed system, and one contains indirect memory access, making it impossible to analyze statically. To support these kernels, peer-to-peer communication is needed, which introduces high network overhead and may outweigh the performance gain from CPU cluster execution.

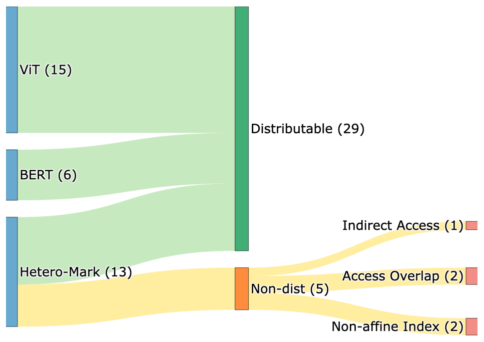


**Figure 7.** Coverage Evaluation for Allgather Distributable.

## 7.2 Performance Evaluation

Eight GPU programs, previously used in other GPU migration projects [6, 10, 17, 21, 22, 27, 28], are used to evaluate the performance. We do not directly reuse the kernels in Section 7.1, as we find that their execution times are too short to reflect meaningful speedup. Specifically, as GPU kernels generated by Triton are hard-coded with bound checking, making it hard to scale the data size to increase the workload to make the runtime evaluation stable.

We execute CuCC on clusters of varying sizes and present the results in Figure 8. The scalability evaluations follow the principle of **strong scalability**, where the problem size remains fixed across all cluster configurations.

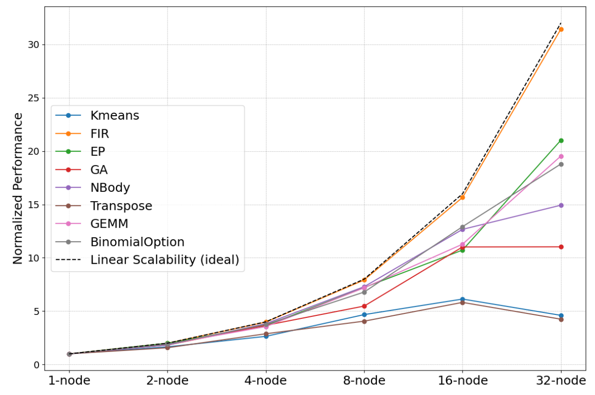
For SIMD-Focused cluster, most kernels demonstrate high scalability on 2-node and 4-node clusters. As the cluster size increases, Kmeans and Transpose fail to achieve further performance gains and even experience slower execution times. Similar behavior is observed on Thread-Focused cluster.

The Matrix Transpose kernel consists of lightweight operations primarily involving memory movement. As the cluster size increases, the memory access overhead on each node decreases. However, the overall communication volume remains constant since the matrix size does not change. Consequently, as the per-node execution time decreases, the communication overhead becomes increasingly significant, limiting scalability. In contrast, for other kernels, the communication overhead is negligible compared to the total execution time. This overhead is illustrated in Figure 9.

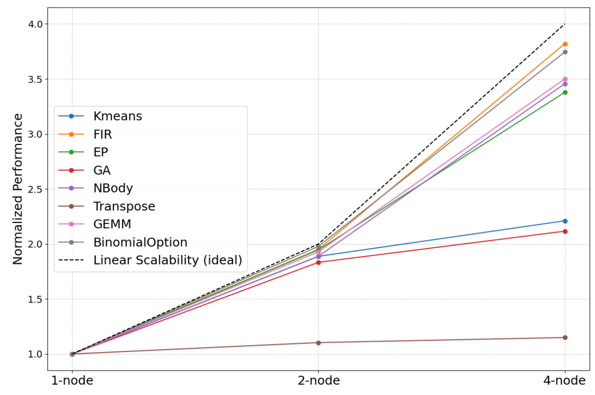
For the Kmeans, the GPU program consists of 313 GPU blocks. When executed on a 16-node cluster, each CPU node is assigned 19 GPU blocks. Each GPU block is executed by a CPU thread, resulting in a CPU thread count close to the number of available CPU cores (24 cores). However, when scaling up to a 32-node cluster, each CPU node is assigned fewer GPU blocks, which cannot efficiently utilize the CPU cores, thereby limiting further scalability.

Another reason for the Kmeans slowdown is the overhead caused by the callback blocks. When executed on a 16-node cluster, each CPU node processes 19 GPU blocks ($\lfloor \frac{313}{16} \rfloor$) during the partial block execution phase, while 9 GPU blocks ($313 - 16 \times 19$) are designated as callback blocks to be executed after the Allgather operation. Each CPU node executes 28 GPU blocks in total. However, when the cluster scales up to 32 nodes, each CPU node processes only 9 GPU blocks during the partial block execution phase and executes 25 callback blocks. Thus, each node executes 34 GPU blocks in total, which leads to a overall execution time slowdown.

On the other hand, the FIR (Finite Impulse Response) achieves near-linear scalability, even on a 32-node cluster. FIR involves heavy computation, including a for-loop that traverses the input sequence to accumulate results. The computed results are scalars, making FIR computation-intensive with minimal memory access overhead. Consequently, the communication overhead is much lower than the computation, making it well-suited for scaling to large clusters.



**(a)** SIMD-Focused cluster.



**(b)** Thread-Focused cluster.

**Figure 8.** CuCC scalability evaluation results.

Thread-Focused cluster also achieves speedup compared to single-node, however, the scalability is lower than SIMD-Focused cluster. For instance, the Transpose kernel achieves a 2.88× speedup on the 4-node SIMD-Focused cluster, but only a 1.14× speedup on 4-node Thread-Focused cluster.

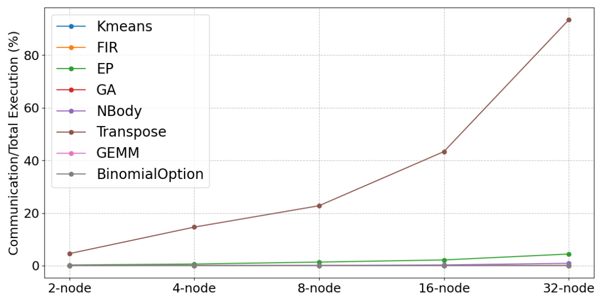


**Figure 9.** Network overhead in SIMD-Focused cluster.

The difference in scalability can be attributed to two factors. First, a single Thread-Focused node contains 128 CPU cores, whereas a single SIMD-Focused node has only 24 CPU cores. Consequently, for GPU kernels with $N$ blocks, the Thread-Focused cluster cannot achieve further speedup beyond $\frac{N}{128}$ nodes, as adding more nodes would result in idle CPU cores. In contrast, the upper bound of scalability for the SIMD-Focused cluster is much higher, at $\frac{N}{24}$ nodes. Second, as further discussed in Section 8.2, a single Thread-Focused node usually achieves significantly higher performance than a single SIMD-Focused node. The expected speedup for a $K$-node cluster can be estimated using Amdahl's law:

$$speedup = \frac{single\ node\ execution}{communication + \frac{single\ node\ execution}{K}}$$

Since both SIMD-Focused and Thread-Focused clusters have the same network bandwidth, their communication overheads are similar. However, the single node execution time on the SIMD-Focused nodes is greater than that on the Thread-Focused nodes. Consequently, for the same $K$, the SIMD-Focused cluster demonstrates better scalability.

### 7.3 Comparison with PGAS Solution

PGAS is another approach for GPU-to-CPU-cluster migration. To migrate a GPU program, the corresponding memory variables are replaced with PGAS global variables, and their associated read/write operations are substituted with remote memory access (Section 3.1). PGAS is designed for general programs, which uses fine-grained remote access for flexibility. For the example in Listing 3, the PGAS solution performs 1200 cluster-level communication operations (line 7). In contrast, CuCC utilizes coarse-grained collective communication. The CPU program generated by CuCC contains only a single collective communication operation (Figure 6, CPU host module, line 5).

We migrate the GPU benchmark using UPC++ [4], one of the most popular PGAS implementations, and execute it on SIMD-focused cluster. We calculate the relative runtime of PGAS and CuCC and present the results in Figure 10. Compared to PGAS, CuCC achieves higher performance across all benchmarks and scales. Moreover, the speedup becomes increasingly significant as the cluster size grows. After filtering out the Transpose benchmark as an outlier, CuCC achieves an average speedup of 4.09× over the PGAS solution on 2-node cluster and 12.81× on 32-node cluster.

CuCC and PGAS exhibit the most significant runtime difference in Transpose benchmark. In Transpose, data movement constitutes the majority of the workload. This data movement involves GPU global memory, which, in PGAS solution, is mapped to remote memory access. The original GPU program assigns a single thread to handle each matrix element. Therefore, for an $N \times N$ matrix, the PGAS program results in $N^2$ communications, introducing substantial overhead. In contrast, CuCC coalesces all memory accesses and requires only a single Allgather communication.

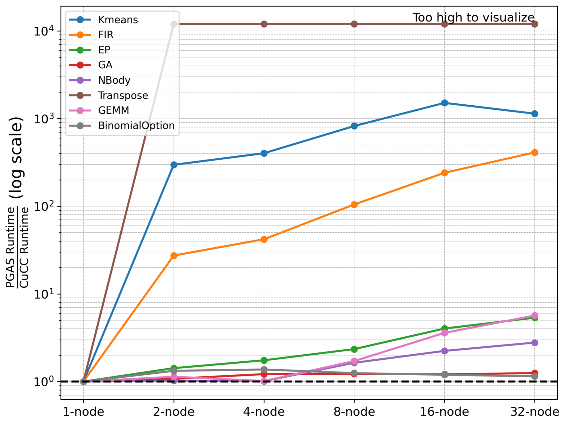


**Figure 10.** CuCC and PGAS solution runtime comparison.

CuCC and PGAS achieve similar performance on the GA and BinomialOption benchmarks. In GA (Gene Alignment), remote memory access occurs only when specific target gene sequences are found in the query gene sequence, which happens infrequently for the given dataset. In BinomialOption benchmark, only the first thread in each GPU block writes to global memory, resulting in minimal communication overhead. As a result, CuCC and PGAS exhibit similar runtimes.

### 7.4 CPU Cluster vs GPU

We compare the performance of CPU clusters and GPUs. Two GPUs, the NVIDIA A100 and V100, **released in the same era as the evaluated CPUs**, are used for comparison.

**7.4.1 Runtime Analysis.** We measure the execution time of GPUs running the original GPU programs and the execution time of CPUs running the migrated CPU programs. The results are presented in Figure 11. For CPU runtimes, we report the best result achieved across various cluster sizes.

On average (geometric mean), the SIMD-Focused cluster has a runtime that is 2.55× slower than the NVIDIA V100 GPU and 4.14× slower than the NVIDIA A100 GPU. In comparison, the Thread-Focused cluster achieves a runtime that is 1.57× slower than the V100 and 2.54× slower than the A100. We provide a detailed analysis of representative applications:

**Transpose:** Both CPU platforms achieve lower execution times than the V100 and A100 GPUs. The Matrix Transpose is memory-intensive, involving frequent transfers between global memory and shared memory. CPUs benefit from large last-level caches (SIMD-Focused CPU: 19.25 MB, Thread-Focused CPU: 256 MB), which are comparable in size to those on GPUs (V100: 6 MB, A100: 40 MB). Additionally, these memory transfers can be efficiently optimized using SIMD instructions, enabling CPUs to deliver performance that is close to or even better than GPUs.

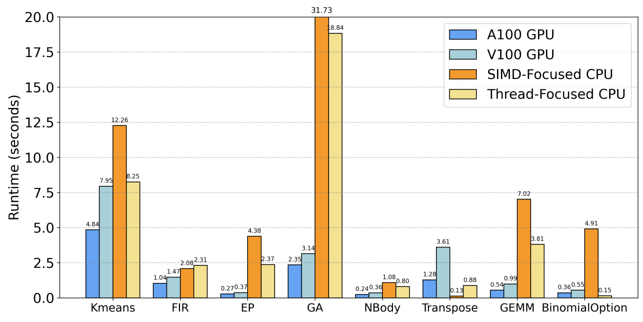


**Figure 11.** Runtime Comparison between CPUs and GPUs.

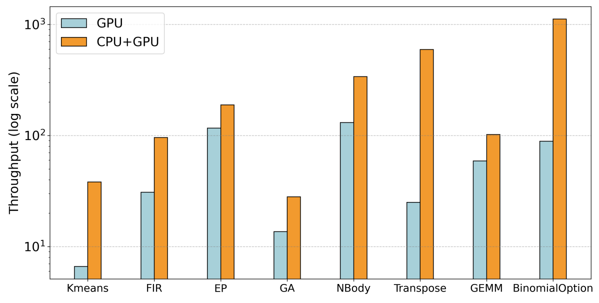


**Figure 12.** Throughput provided by GPUs and GPUs+CPUs.

**BinomialOption:** The program features a two-level nested for-loop and is highly compute-intensive, resulting in significant workloads for each GPU thread. Each GPU block performs an internal reduction, with only the first GPU thread writing a scalar value to global memory. This memory access pattern is ideal for CPU cluster migration, as it incurs low communication overhead. Consequently, the Thread-Focused CPU cluster achieves the highest performance with 4-node execution. With a computational capacity of up to 32 TFLOPs, the 4-node Thread-Focused CPU cluster outperforms both the A100 (19.5 TFLOPs) and V100 (14 TFLOPs).

The SIMD-Focused CPU cluster achieves its highest performance with 32 nodes, further demonstrating that BinomialOption is well-suited for CPU clusters. However, the SIMD-Focused cluster does not achieve the same level of performance as the Thread-Focused cluster. This is because the nested for-loop in the GPU program has loop dependencies that cannot be parallelized with SIMD.

**EP and GA:** On both benchmarks, GPUs outperform CPU clusters by a factor of 5×–10×. These programs contain relatively few GPU blocks (EP: 512, GA: 256), which cannot fully utilize thread-level parallelism in large-scale CPU clusters. Additionally, the kernel code includes for-loops that cannot be optimized with SIMD instructions. As a result, these programs are unable to effectively leverage either the thread-level or data-level parallelism available in CPUs.

#### 7.4.2 Throughput Analysis.

In data centers, CPUs, designed for general applications, are typically much more accessible than GPUs. For example, the TACC Lonestar6 cluster [2] has 560 CPU nodes but only 16 GPU nodes. Similarly, the Frontera cluster [1] contains 8,368 CPU nodes and only 90 GPU nodes. This vast difference in quantity enables CPUs to provide significantly more aggregate compute capacity, making them well-suited to complement GPU resources.

We estimate the cluster-wide throughput for the TACC Lonestar6 cluster (Figure 12). The throughput of batch processing for each GPU program is measured over a one-second time frame. The cluster contains a significant number of Thread-Focused CPUs, providing substantial aggregated computational capacity. On average, instead of executing programs on GPU nodes alone, utilizing these additional CPU nodes improves throughput by 3.59×.

Based on this cluster-wide throughput analysis, we conclude that GPU-to-CPU migration can unlock the potential of idle CPU resources to alleviate GPU shortages. Specifically, for batch-processing applications that are less sensitive to latency, executing on CPU clusters can achieve higher throughput compared to GPU execution.

## 8 Discussion

### 8.1 Target GPU Applications

Our solution proposes integrating multiple CPU nodes to execute GPU programs with minimal network overhead. Thus, the benefit a GPU application receives depends highly on two factors.

**Parallelism:** Our solution applies GPU block-level parallelism, in which GPU blocks are distributed to CPU nodes for parallel execution. Therefore, it is critical to have a sufficient number of GPU blocks to distribute. For example, to achieve linear scalability for a CPU cluster with $C$ nodes, where each node has $T$ CPU cores, we need at least $C \times T$ GPU blocks to fully utilize the CPU resources.

**Local Execution Overhead:** If the local execution (e.g., computation, local memory access) is heavy, migrating to CPU-cluster execution significantly decreases the execution time, bringing an end-to-end speedup. In Sec. 7.2, we found that all GPU applications achieve a speedup from CPU-cluster migration compared to the single-CPU solution. This is because the input GPU programs are originally designed for single-GPU execution. As a single GPU typically has higher capacity than a single CPU, the workload designed for a single GPU is inherently heavy for a single CPU execution, which allows it to benefit from distributing the workload to CPU clusters.

### 8.2 Target CPU Architectures

The SIMD-Focused CPU (Intel Gold 6226) provides wide SIMD instructions (AVX-512), while the Thread-Focused

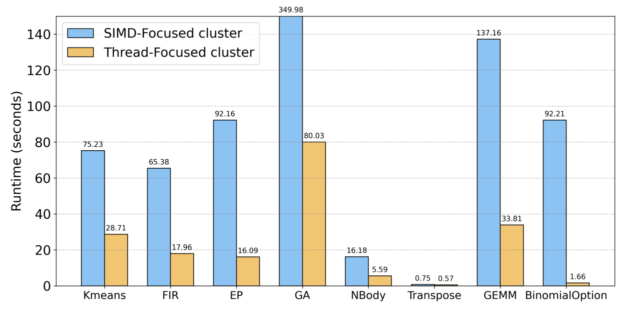
CPU (AMD EPYC 7713) offers high thread-level parallelism with 64 cores per socket. To ensure a fair comparison between them, in this section, we limit execution on the Thread-Focused cluster nodes to 64 CPU cores. This results in comparable theoretical computational capacities: 4.147 TFLOPs for the SIMD-Focused cluster and 4.096 TFLOPs for the Thread-Focused cluster.

We compare the runtime performance of the two clusters. As shown in Figure 13, the Thread-Focused cluster demonstrates significantly higher performance for migrated GPU programs. Based on the geometric mean, the Thread-Focused cluster is 4.61×, 4.66×, and 4.32× faster than the SIMD-Focused cluster for 1, 2, and 4 nodes, respectively.

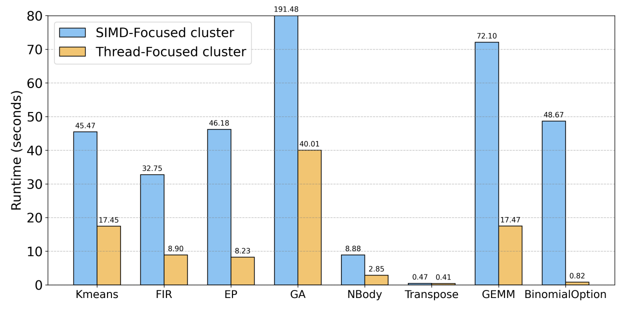
For a single-node, the largest performance difference is observed in the BinomialOption kernel, where Thread-Focused CPUs are 55× faster than SIMD-Focused CPUs. The BinomialOption contains 1024 GPU blocks, each with a non-parallel for-loop to calculate accumulated values. This structure is highly suited to thread-level parallelism, as the large number of blocks can be executed in parallel. Conversely, since the kernel includes a non-parallel for-loop as its inner loop, it is challenging to apply SIMD optimization. As a result, the SIMD-Focused CPU, which is optimized for data-parallel operations, achieves lower performance compared to the Thread-Focused CPU, which excels in thread-parallel workloads.

The SIMD-Focused and Thread-Focused clusters show the closest performance on the Transpose kernel. In single-node execution, the Thread-Focused CPU is 1.3× faster than the SIMD-Focused CPU. The Transpose consists of parallelized for-loops, with the loop body primarily containing memory movement operations, which are highly amenable to SIMD optimization. To further analyze performance, we measured execution time with SIMD optimization disabled. Compared to execution with SIMD optimization enabled, the Thread-Focused CPU showed no performance degradation, while the SIMD-Focused CPU experienced a slowdown of 61.66%.

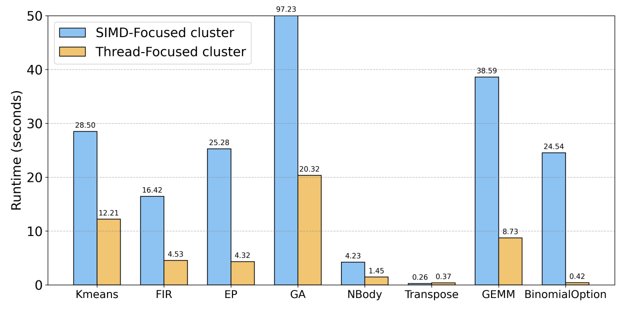
Based on the evaluation, we conclude that for executing migrated GPU programs, CPUs with a higher core count are more likely to achieve high performance, and thread-level parallelism is more effective than data-level parallelism.



**(a)** Single-Node Cluster.



**(b)** 2-Node Cluster.



**(c)** 4-Node Cluster.

**Figure 13.** Runtime in SIMD/Thread-Focused clusters.

## 8.3 Future Directions for GPU-to-CPU Migration

Based on the evaluation result, we propose several suggestions for future GPU-to-CPU migration solutions.

**Workload Redistribution:** Our evaluation reveals that GPU programs with few GPU blocks cannot scale effectively to large CPU clusters. For instance, for a 4-node CPU cluster with 64 cores per node, at least 256 GPU blocks are required to fully utilize thread-level parallelism in CPUs.

In addition to parallelism, adjustable block sizes could also help redistribute workloads to align with hardware capabilities. After GPU-to-CPU migration, each GPU block maps to a CPU thread, effectively replacing a GPU Streaming Multiprocessor (SM) with a CPU core. Since GPU SMs and CPU cores have different computational capacities, workloads optimized for an SM may not be well-suited for a CPU core.

However, in practical GPU programs, developers often hard-code block sizes to control resource utilization within a GPU SM, such as shared memory and registers. These hard-coded values create challenges when attempting to modify the number of blocks through compiler transformations.

Therefore, we suggest developing a more flexible GPU programming framework that enables the adjustment of GPU block workloads through compiler transformations. Such a framework would not only facilitate GPU-to-CPU migration but also benefit other portable programming solutions.

**SIMD Optimization:** CPUs leverage both data-parallelism and thread-parallelism to achieve high performance. However, our evaluation reveals that utilizing data-parallelism in transformed CPU programs is more challenging.

In the transformed CPU programs, a GPU thread is replaced by an iteration of a for-loop. To achieve high performance, each GPU thread should ideally be executed by a lane of an SIMD instruction. Although these iterations are independent, applying SIMD optimization to such for-loops remains challenging. This is because these parallel for-loops are usually the outermost loops in transformed CPU programs, whereas SIMD instructions are most suitable for parallelizing inner loops. Additionally, Han et al. [23] suggest that transformed CPU programs often involve complex control flow and data flow, making them difficult for static analysis.

In our evaluation, we observed that Thread-Focused CPUs typically achieve higher performance than SIMD-Focused CPUs, even when both have the same peak theoretical performance. This observation highlights the need for developing SIMD optimizations tailored for CPU programs transformed from GPU programs.

### 8.4 Cost and Energy Aspects

Our solution proposes a way to utilize idle CPU resources to alleviate the GPU shortage. It is important to note that idle CPUs have non-negligible energy consumption [30, 36]. Consequently, cloud providers often offer spot services at discounted rates [25, 31], encouraging users to leverage these idle resources. Based on these observations, we believe our solution, which offers a new way to utilize idle CPUs, provides an attractive option for saving energy and reducing costs in data centers.

## 9 Related Work

### 9.1 GPU-to-CPU Migration

To address the disparity in parallelism between GPU programs and CPU architectures, researchers [42] propose compiler transformations to group lightweight GPU workloads. This approach significantly reduces the number of threads required for the transformed CPU programs and is widely adopted in projects supporting SPMD programs on CPUs [7, 16, 35, 37, 40, 41, 47]. Han et al. [23] highlight that transformed CPU programs are often incompatible with existing optimizations and propose novel compiler and runtime optimizations. Moses et al. [32] propose to optimize the migration with a polyhedral model.

All existing GPU-to-CPU projects focus on single CPUs, whereas our solution extends migration to CPU clusters.

### 9.2 Single-Device to Multi-Device Migration

Researchers propose solutions to migrate programs written for a single device to execute on multiple devices. OmpSS [19] and StarPU [3] are frameworks that offload workloads to distributed nodes. These solutions focus on **inter-kernel** parallelism, where a single task is executed exclusively by one device. In contrast, our project focuses on **intra-kernel** parallelism, where multiple distributed nodes collaborate to execute a single task (i.e., GPU kernel).

Other intra-kernel parallelism solutions either rely on hardware-supported shared memory to maintain data consistency [12, 13] or use peer-to-peer communication to synchronize CPU and GPU memory on the same node [29, 34]. Our work is the first to migrate a single GPU program to a CPU cluster, an environment with no hardware-supported shared memory and where peer-to-peer communication is too expensive for high performance.

### 9.3 Partitioned Global Address Space

PGAS is a parallel programming model that maintains a global memory space across distributed nodes. The global memory is partitioned among nodes, and PGAS provides primitives that allow each node to access memory located on other nodes. PGAS is a widely used model with many implementations (e.g., UPC++ [4], SHMEM [11]).

These solutions are designed for general programs, utilizing flexible but costly communication operations. However, for GPU-to-CPU-cluster migration, communication overhead is a significant concern due to the high volume of communication. Our work analyzes common patterns in GPU programs and proposes the use of coarse-grained collective communication to reduce network overhead.

## 10 Conclusion

We introduce CuCC, a framework for migrating GPU programs to CPU clusters. CuCC incorporates compiler analysis and transformations to generate CPU cluster programs with low communication overhead. Our evaluation demonstrates that GPU programs can be efficiently executed on CPU clusters, achieving execution times within the same order of magnitude as GPUs. Furthermore, since modern data centers typically contain far more CPU nodes than GPUs, cluster-wide throughput analysis shows that CPUs achieve 2.59× higher throughput compared to GPU execution.

As the industry moves toward higher-bandwidth networks such as 400 Gbps and 800 Gbps, the performance of clustered CPUs will continue to improve. Hence, we expect that running GPU programs on clustered CPUs will become even more compelling.

## Acknowledgments

We thank the anonymous reviewers for their insightful feedback. This work was supported in part by NSF grants CCF-2316176 and CCF-2452082. We also thank Georgia Tech's CRNCH (Center for Research into Novel Compute Hierarchies) Rogues Gallery.

## References

[1] 2025. TACC Frontera. https://docs.tacc.utexas.edu/hpc/frontera/

[2] 2025. TACC Lonestar6. https://tacc.utexas.edu/systems/lonestar6/

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In *European Conference on Parallel Processing*. Springer, 863–874.

[4] John Bachan, Scott B Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H Hargrove, and Hadia Ahmed. 2019. UPC++: A high-performance communication framework for asynchronous computation. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 963–973.

[5] Peter J Baddoo, Benjamin Herrmann, Beverley J McKeon, J Nathan Kutz, and Steven L Brunton. 2023. Physics-informed dynamic mode decomposition. *Proceedings of the Royal Society A* 479, 2271 (2023), 20220576.

[6] Tal Ben-Nun, Ely Levy, Amnon Barak, and Eri Rubin. 2015. Memory access patterns: The missing piece of the multi-GPU puzzle. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[7] Vera Blomkvist Karlsson. 2021. Cumulus-translating CUDA to sequential C++: Simplifying the process of debugging CUDA programs.

[8] Dan Bonachea and Paul H Hargrove. 2018. GASNet-EX: A high-performance, portable communication library for exascale. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 138–158.

[9] Javier Borrachero Prieto. 2022. Confronting technology's greatest crisis: Global chip shortage. Design of an innovative new supply chain. (2022).

[10] Javier Cabezas, Lluís Vilanova, Isaac Gelado, Thomas B Jablin, Nacho Navarro, and Wen-mei W Hwu. 2015. Automatic parallelization of kernels in shared-memory multi-gpu nodes. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 3–13.

[11] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. 1–3.

[12] Younghyun Cho, Florian Negele, Seohong Park, Bernhard Egger, and Thomas R Gross. 2018. On-the-fly workload partitioning for integrated CPU/GPU architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 1–13.

[13] Younghyun Cho, Jiyeon Park, Florian Negele, Changyeon Jo, Thomas R Gross, and Bernhard Egger. 2022. Dopia: online parallelism management for integrated CPU/GPU architectures. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 32–45.

[14] Salvatore Cuomo, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi, and Francesco Piccialli. 2022. Scientific machine learning through physics–informed neural networks: Where we are and what's next. *Journal of Scientific Computing* 92, 3 (2022), 88.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 4171–4186.

[16] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 353–364.

[17] Tahir Diop, Steven Gurfinkel, Jason Anderson, and Natalie Enright Jerger. 2013. DistCL: A framework for the distributed execution of OpenCL kernels. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 556–566.

[18] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[19] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters* 21, 02 (2011), 173–193.

[20] Nándor Hajdu. 2021. COVID-19 pandemic and managing supply chain risks: NVIDIA's graphics card shortage case analysis. (2021).

[21] 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 Transactions on Design Automation of Electronic Systems* 29, 4 (2024), 1–25.

[22] Ruobing Han, Jaewon Lee, Jaewoong Sim, and Hyesoon Kim. 2022. COX: Exposing CUDA Warp-Level Functions to CPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* (2022).

[23] Ruobing Han, Jisheng Zhao, and Hyesoon Kim. 2024. Unleashing CPU Potential for Executing GPU Programs through Compiler/Runtime Optimizations. In *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture.*

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[25] Syed M Iqbal, Haley Li, Shane Bergsma, Ivan Beschastnikh, and Alan J Hu. 2022. Cospot: a cooperative vm allocation framework for increased revenue from spot instances. In *Proceedings of the 13th Symposium on Cloud Computing*. 540–556.

[26] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. 2015. Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. 3–14.

[27] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. 2011. Achieving a single compute device image in OpenCL for multiple GPUs. *ACM Sigplan Notices* 46, 8 (2011), 277–288.

[28] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2013. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 245–255.

[29] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2015. Skmd: Single kernel on multiple devices for transparent cpu-gpu collaboration. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 1–27.

[30] Jie Li, George Michelogiannakis, Brandon Cook, Dulanya Cooray, and Yong Chen. 2023. Analyzing resource utilization in an hpc system: A case study of nersc's perlmutter. In *International Conference on High Performance Computing*. Springer, 297–316.

[31] Liduo Lin, Li Pan, and Shijun Liu. 2022. Methods for improving the availability of spot instances: A survey. *Computers in Industry* 141 (2022), 103718.

[32] 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.

[33] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. {Latency-Tolerant} software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 291–305.

[34] Prasanna Pandit and R Govindarajan. 2014. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 273–283.

[35] Atmn Patel, Shilei Tian, Johannes Doerfert, and Barbara Chapman. 2021. A virtual GPU as developer-friendly OpenMP offload target. In *50th International Conference on Parallel Processing Workshop*. 1–7.

[36] Arman Shehabi, Alex Hubbard, Alex Newkirk, Nuoa Lei, Md Abu Bakkar Siddik, Billie Holecek, Jonathan Koomey, Eric Masanet, Dale Sartor, et al. 2024. 2024 United States Data Center Energy Usage Report. (2024).

[37] Jun Shirako, Jisheng M Zhao, V Krishna Nandivada, and Vivek N Sarkar. 2009. Chunking parallel loops in the presence of synchronization. In *Proceedings of the 23rd international conference on Supercomputing*. 181–192.

[38] Jan Solanti, Michal Babej, Julius Ikkala, Vinod Kumar Malamal Vadakital, and Pekka Jääskeläinen. 2021. PoCL-R: A scalable low latency distributed OpenCL runtime. In *International Conference on Embedded Computer Systems*. Springer, 78–94.

[39] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. 2024. ML Training with Cloud GPU Shortages: Is Cross-Region the Answer?. In *Proceedings of the 4th Workshop on Machine Learning and Systems*. 107–116.

[40] John A Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei W Hwu. 2010. Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 111–119.

[41] John A Stratton, Hee-Seok Kim, Thoman B Jablin, and Wen-Mei W Hwu. 2013. Performance portability in accelerated parallel kernels. *Center for Reliable and High-Performance Computing* (2013).

[42] John A Stratton, Sam S Stone, and Wen-Mei W Hwu. 2008. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In *Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31-August 2, 2008, Revised Selected Papers 21*. Springer, 16–30.

[43] 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 *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–10.

[44] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.

[45] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).

[46] Jin Zhang, Xiangyao Yu, Zhengwei Qi, and Haibing Guan. 2022. Falcon: A Timestamp-based Protocol to Maximize the Cache Efficiency in the Distributed Shared Memory. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 974–984.

[47] Yao Zhang, Mark Sinclair, and Andrew A Chien. 2013. Improving performance portability in OpenCL programs. In *International Supercomputing Conference*. Springer, 136–150.

Received 2025-08-31; accepted 2025-11-10