Unleashing CPU Potential for Executing GPU Programs through Compiler/Runtime Optimizations

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

Jisheng Zhao
Georgia Institute of Technology
Atlanta, USA
jisheng.zhao@cc.gatech.edu

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

***Abstract*—Although modern CPUs deliver high performance, they are often under-utilized in CPU-GPU heterogeneous systems. Enabling CPUs to execute GPU programs facilitates workload sharing between CPUs and GPUs, which can increase CPU device utilization and overall benefit heterogeneous systems.**

The flat collapsing transformation represents a state-of-the-art solution for GPU-to-CPU migration and is widely accepted in both academic and commercial projects. However, in this paper, we identify that CPU programs transformed by flat collapsing transformation are not compatible with standard CPU compiler optimizations and runtime environments, which leads to suboptimal performance. Based on the observation, we propose four compiler/runtime optimizations. These optimizations complement flat collapsing transformation and help generate high-performance programs. Our evaluations demonstrate an average performance improvement of 20.84% over the state-of-the-art framework on an x86 CPU and 16.10% on an ARM CPU.

I. Introduction

For the stereotype of CPU-GPU heterogeneous systems, GPUs are the predominant devices for executing massively computing applications, while CPUs are primarily required to provide more accessible options to users. However, it is often overlooked that CPUs also provide a substantial amount of computational resources. For example, the latest Intel Gold 6423N [6] and AMD EPYC 9654P [5] CPUs feature 28 and 96 CPU cores, respectively, along with AVX support, providing up to 6.5 and 10.9 TFLOPS/sec of computational power. Similarly, ARM-based CPUs like Fujitsu's A64FX deliver a peak performance of 5.4 TFLOPS/sec [2]. These CPUs achieve peak data processing performance at a magnitude similar to that of NVIDIA A100 GPUs, which offer 19.5 TFLOPS [4]. This implies that, with full utilization of the performance capabilities, CPUs might achieve performance comparable to GPUs.

Although CPUs provide a substantial amount of computational resources, they are consistently underutilized in heterogeneous systems. It is not surprising to find users waiting for hours to have GPU devices scheduled while many CPUs remain idle. This observation raises an important question: "Can CPUs be effectively utilized to run GPU applications?"

GPU programs typically consist of a large number of lightweight parallel tasks, whereas CPUs are designed to handle a relatively small number of heavy workloads. Running GPU programs on CPUs requires transforming programs designed for the bulk-synchronous parallel model into traditional programming models, which do not support a high number of active threads. MCUDA [49] proposes a compiler transformation that wraps the workload of an entire GPU block and executes it using a single CPU thread. This transformation, named '`flat collapsing transformation`' in [23], enhances performance compared to directly mapping one GPU thread to one CPU thread by reducing the overhead of thread context switching and better utilizing the vector units in CPU cores. An example of flat collapsing transformation is demonstrated in Figure 1. The flat collapsing transformation can be implemented using either a source-to-source translator [9], [49] or low-level Intermediate Representation (IR) transformation [12], [21], [23], [40]. We use a CUDA-to-OpenMP example to demonstrate how flat collapsing transformation aims to utilize CPU computational resources for executing GPU programs.

CUDA program

```
1  __global__ void tiled_reverse(int *d,
                                 int n) {
2    __shared__ int s[BLOCK_SIZE];

3    int idx = blockIdx.x * BLOCK_SIZE +
                            threadIdx.x;
4    s[threadIdx.x] = d[idx];

5    __syncthreads();

6    d[idx] = s[(n - idx - 1) % BLOCK_SIZE];
   }
7. void main() {
8.   tiled_reverse<<<GRID_SIZE,
                     BLOCK_SIZE>>>(d, n);
   }
```

Transformed CPU program

```
1  void tiled_reverse(int *d,
                      int n, int bid) {
2    int s[BLOCK_SIZE];
3    int idx[BLOCK_SIZE];
   #pragma omp simd
     for (int tid=0; tid<BLOCK_SIZE: tid++) {
4.     idx[tid] = bid*BLOCK_SIZE+tid;
5.     s[tid] = d[idx[tid]];
     }
   #pragma omp simd
     for (int tid=0; tid<BLOCK_SIZE: tid++)
6.     d[idx[tid]] = s[(n-idx[tid]-1)%BLOCK_SIZE];
   }
7. void main() {
   #pragma omp parallel for
     for (int bid=0; bid<GRID_SIZE; bid++)
8.     tiled_reverse(d, n, bid);
   }
```

Fig. 1: An example of a CUDA program and the corresponding CPU program transformed by flat collapsing transformation.

The workflow for existing GPU-to-CPU solutions [3], [21], [23], [27], [49] is illustrated by the solid black line in Figure 2. Initially, GPU programs are transformed into CPU programs using flat collapsing transformation. These generated CPU programs are then subjected to standard compiler optimizations. Finally, runtime libraries that implement GPU APIs (e.g., cudaMalloc, cudaMemcpy, kernel launch) with equivalent CPU functions are linked to produce executable CPU files.

However, in this paper, we highlight that this pipeline does not effectively utilize the computational potential of CPUs for two main reasons. First, the CPU programs are transformed from GPU programs, which are optimized for GPU

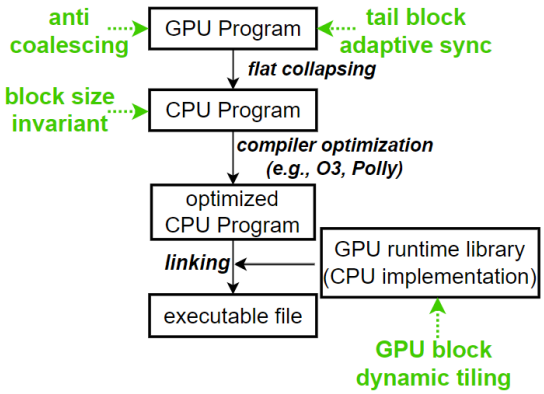


Fig. 2: The existing GPU-to-CPU migration process is depicted in black text. We propose four optimizations, highlighted in green text, which can be integrated into this workflow.

architectures. Some GPU optimizations do not benefit CPU architectures. Second, the existing pipeline treats transformed CPU programs as ordinary CPU programs and optimizes them with standard compiler optimizations. However, CPU programs transformed by flat collapsing transformation are usually more complex than manually written CPU programs. Therefore, the existing compiler optimizations are too general to effectively optimize these programs.

Based on these observations, we propose four compiler/runtime optimizations designed to enhance the performance of transformed CPU programs. These optimizations, which are compatible with flat collapsing transformation and highlighted in green in Figure 2, specifically address key inefficiencies. The anti-coalescing transformation and tail block adaptive synchronization are compiler transformations applied to GPU programs, while block size invariant analysis is a transformation applied to CPU programs. GPU-block dynamic tiling is a runtime optimization that targets the kernel launch function.

This paper offers several noteworthy contributions:

- Identification of unresolved efficiency challenges in existing GPU-to-CPU migration solutions.
- Introduction of three compiler optimizations designed to enhance the performance of generated CPU programs.
- Introduction of a runtime optimization that dynamically adjusts the workload distribution to optimally align with CPU computational resources.
- Integration of the proposed optimizations into a state-of-the-art GPU-to-CPU solution, resulting in significant performance improvements on both x86 (20.84%) and ARM CPUs (16.10%).

In this paper, we focus on migrating CUDA applications to CPUs. However, the proposed optimizations are not limited to any specific programming languages. Consequently, the technical contributions are also applicable to other parallel programming models (e.g., HIP [1], OpenCL [41]) and libraries (e.g., OCCA [38], Kokkos [51]).

II. Background

A. Flat Collapsing Transformation

GPUs are designed to handle the efficient execution of a large number of lightweight tasks, while CPUs are optimized for a smaller quantity of heavy tasks. To reconcile the disparity in parallelism and workload granularity, MCUDA [49] introduces a compiler transformation referred to as the flat collapsing transformation. The transformation aims to migrate GPU programs to CPUs in an efficient manner. Essentially, the flat collapsing transformation wraps the workload in a CUDA block into a CPU function to be executed by a CPU thread. Figure 1 showcases a CUDA application and the CPU program transformed by the flat collapsing transformation.

There are several important points to note in this example. Firstly, all CUDA blocks are transformed into iterations within a loop in the main function of the transformed CPU program, with length equals to the grid size, allowing parallelization across multiple CPU threads. Secondly, the CUDA threads are also transformed into iterations within loops, with lengths equivalent to the block size. These loops can be vectorized using SIMD instructions. We call these loops **flat loops** in this paper. Thirdly, the CUDA shared memory variable s is mapped to CPU thread-local memory. Fourthly, a barrier is present in the CUDA program (line 5th), causing the flat collapsing transformation to generate two sequential flat loops that wrap the CUDA code before (line 3-4th) and after the barrier (line 6th). Lastly, the local variable idx is extended to arrays with lengths equal to the CUDA block size, as they are referenced in both flat loops. This transformation is critical for maintaining the correctness of flat collapsing transformation, which is called 'selective replication' in MCUDA [49]. The insight is that, after migrating to a CPU program, where a CPU thread needs to execute a GPU block workload, the CPU thread should create arrays to store local variables of each GPU thread in the block.

In general, flat collapsing transformation is a compiler transformation that groups multiple parallel fine-grained workloads into a single coarse-grained task. It reduces the number of parallel tasks by increasing the workload of each task, thereby bridging the gap between the number of concurrent threads required by software developers and the concurrent cores supported by hardware. Similar transformations are utilized to improve the runtime of GPU programs [8] or to support performance portability of SPMD languages on devices with less parallelism [23], [27], [31]. Thus, we believe the optimizations proposed in this paper can also be extended for GPU-to-GPU transformations and projects aimed at SPMD performance portability.

B. Challenges for Existing Solutions

Although flat collapsing transformation is expected to generate programs suitable for CPUs, we find that the generated CPU programs consistently exhibit suboptimal performance. The low performance stems from two main reasons. First, the original GPU programs are designed for utilizing GPU

computation resource, which may not perform optimally on CPU architectures. Second, flat collapsing transformation is not compatible with existing compiler optimizations. Specifically, the CPU programs generated by flat collapsing transformation are too complex to be effectively analyzed and transformed by standard compiler optimizations. Consequently, these programs cannot be optimized significantly.

We illustrate a CUDA program and the corresponding CPU program generated by flat collapsing transformation in Figure 1. Although the original CUDA program is simple, the transformed CPU program is more complex: The CPU program includes indirect memory accesses (lines 5-6th) introduced by flat collapsing transformation. These indirect memory accesses bring challenges for compiler optimization, as the compiler cannot clearly determine memory access dependencies. An attempt to optimize this code section using LLVM Polly [19], a popular polyhedral model compiler, results in the warning message "The array subscript of 's' is not affine," preventing Polly from optimizing this section. Additionally, flat collapsing transformation generates a significant number of loops where the length is equal to the CUDA block size. Since the CUDA block size is typically a runtime variable, these loops are dynamic[1] and challenging to optimize because the compiler has limited knowledge about their properties.

III. Problem Statements

We provide detailed discussions about the limitations of existing GPU-to-CPU solutions in this section and introduce four corresponding optimizations in Section IV.

A. Divergent Memory Access Preferences

GPU and CPU architectures have different memory access preferences. CUDA optimizes memory bandwidth utilization by grouping interleaved global memory access among threads into a single request, a technique known as memory coalescing. In contrast, CPUs favor sequential memory access patterns to achieve high spatial locality. Consequently, directly transforming CUDA's interleaved access pattern into CPU code could result in poor cache utilization on the CPU. An example that derived from Hetero-Mark [50] is shown in Listing 1. The CUDA program contains an interleaved access pattern (line 7th) to apply global memory coalescing. However, the transformed CPU program has poor spatial locality (line 20th).

We profile this program on both an NVIDIA GTX TITAN X GPU and Intel CPUs using three GPU-to-CPU migration solutions [21], [26], [49]. The comparison of Last-Level-Cache (LLC) performance, as shown in Table I, reveals that the CPU programs transformed from CUDA do not utilize the CPU cache as efficiently as the original programs on NVIDIA GPUs.

[1]Dynamic loops are loops with unknown number of iterations at compile time.

[2]The GPU hit rate is not split into load and store, as the profiling toolkit does not distinguish between load and store hit rates.

Metrics	NV-GPU	GPU-to-CPU solutions		
		DPC++	MCUDA	CuPBoP
# of LLC ld	4,578,831	9,086,165	11,034,617	8,899,486
# of LLC st	4,179,010	6,989,671	414,866	637,932
LLC ld Hit	90.83%[2]	72.2%	63.4%	72.6%
LLC st Hit	90.83%[2]	76.1%	76.5%	83.7%

TABLE I: The profiling result for the CUDA program and CPU programs transformed by three CUDA-to-CPU solutions.

B. Flat Loop Optimization

All transformed CPU programs are **loop-intensive** because flat collapsing transformation introduces flat loops to wrap the original CUDA code. Therefore, optimizing these flat loops is crucial. Each flat loop has a uniform length equivalent to the CUDA block size, which is typically a runtime variable. Consequently, all flat loops are classified as dynamic loops.

There are two common strategies for loop optimization: loop vectorization and reduction of the loop body. However, both approaches face significant challenges. Dynamic flat loops present greater complexity for vectorization compared to static loops. Similarly, optimizing the loop body is challenging due to the great amount of instructions dependent on runtime variables, which hinders effective analysis and optimization. Moreover, flat collapsing transformation often introduces indirect memory access, which poses challenges for memory access dependency analysis and impedes the application of compiler optimizations.

It is important to note that even for CUDA programs that use a constant value as the block size, it is still challenging for flat collapsing transformation to generate static loops. This limitation arises because flat collapsing transformation is a module-level transformation, which cannot utilize cross-module information. A CUDA program contains two modules: host and kernel, executed on the CPU and GPU respectively. The block size value is specified in the host module. The kernel module gets the block size by reading from NVIDIA GPU intrinsic registers at runtime. Thus, it is challenging for flat collapsing transformation to retrieve the static block size across the module boundary since it is a module-level transformation.

C. Uneven Workload Configuration

The third challenge arises from the mapping strategy employed by flat collapsing transformation, which assigns a GPU block to a CPU thread. From the hardware perspective, this approach maps a GPU streaming multiprocessor (SM) to a CPU core. Workloads optimized for a GPU SM may not perform equally well on a CPU core. For instance, an SM in the NVIDIA A100 GPU is equipped with 192KB of L1 cache and shared memory, and it provides a computational capacity of 180 GFLOPs. In contrast, a core in the Intel 6424N CPU comes with a 2MB L2 cache and has a computational capacity of 230 GFLOPs. Consequently, direct mapping of a CUDA block to a CPU thread can lead to suboptimal utilization of the CPU's computational resources.

IV. Optimizations

To address the issues outlined in Section III, we propose four compiler/runtime optimizations: In response to the disparities in memory access patterns, we introduce anti-coalescing transformation, a novel GPU-to-GPU transformation that eliminates the negative effects of memory coalescing. We propose block size invariant analysis and tail block adaptive synchronization to tackle the challenges associated with flat loops. Furthermore, to effectively cater to the specific requirements of CPU architectures regarding workload configuration, we propose GPU-block dynamic tiling, which redistributes workloads to ensure optimal utilization of CPU resources.

These optimizations can be integrated into the existing GPU-to-CPU workflow (highlighted in green in Figure 2). The anti-coalescing transformation and tail block adaptive synchronization are GPU-to-GPU transformations, while block size invariant analysis optimizes the transformed CPU programs. GPU-block dynamic tiling is a runtime optimization that targets the kernel launch function within the runtime library.

```
// original CUDA program
__global__ void cuda_Histogram(uint32_t *pixels,
    uint32_t num_pixels) {
  uint32_t priv_hist[256];
  uint32_t gsize = blockDim.x * gridDim.x;
  uint32_t index = blockDim.x * blockIdx.x+threadIdx.x;
  while (index < num_pixels) {
    uint32_t color = pixels[index]; // global memory
     coalescing
    priv_hist[color]++;
    index += gsize;
  }
  ...
}
// CPU program transformed by the flat collapsing
void cpu_Histogram(uint32_t *pixels, uint32_t
    num_pixels) {
  for (int tid = 0; tid < BLOCK_SIZE; tid++) { // flat
     loop
    uint32_t priv_hist[256];
    uint32_t gsize = BLOCK_SIZE * GRID_SIZE;
    uint32_t index = BLOCK_SIZE * BLOCK_ID + tid;
    while (index < num_pixels) { // original loop
      uint32_t color = pixels[index]; // poor locality
      priv_hist[color]++;
      index += gsize;
    }
    ...
  }
}
```

Listing 1: The Histogram benchmark in Hetero-Mark.

A. Anti-Coalescing Transformation

1) Insight: As noted in Section III, one of the most common CUDA memory access patterns, global memory coalescing, can lead to poor performance in transformed CPU programs. This is because transformed CPU programs exhibit low spatial locality. An example of global memory access coalescing is shown in Listing 1, where the transformed CPU program has a large stride ($gsize$) between each memory access of array $pixels$, resulting in the low spatial locality. To address the slowdown caused by the global memory coalescing, we propose the anti-coalescing transformation, a transformation applied **before** the flat collapsing transformation (i.e. operates at CUDA program). This policy makes the anti-coalescing transformation effectively circumvents the complications arising from flat collapsing transformation.

The CPU program transformed by flat collapsing transformation (the bottom part of Listing 1) contains nested loops. The outer loop is generated by the flat collapsing transformation (refer to $flat\_loop$), and the inner loop is the loop that contains the global memory coalescing in the original CUDA program (refer to $original\_loop$). The high locality can be achieved by exchanging the order of outer/inner loops, as the memory access index ($index$) is linearly increased with the $flat\_loop$'s induction variable (tid). However, since there is a data dependency (tid->index) between these loops, the loop reordering cannot be applied.

We attempt to use LLVM optimization and the Polly compiler [19] to optimize the `cpu_Histogram` function. The LLVM loop-interchange optimization reports a dependency between the flat loop and the original loop and cannot transform the nested loops. Polly detects indirect memory access when accessing the `priv_hist` array and leaves the code section unchanged. This evaluation demonstrates the challenges existing compiler optimizations face when analyzing programs generated by flat collapsing transformation.

```
// incorrect CUDA program
__global__ void transformed_cuda_Histogram(uint32_t *
    pixels, uint32_t num_pixels) {
  uint32_t priv_hist[256];
  uint32_t gsize = blockDim.x * gridDim.x;
  uint32_t index = blockDim.x * blockIdx.x+threadIdx.x;
  while (index < num_pixels) {
    uint32_t color = pixels[index];
    priv_hist[color]++;
    index += gsize;
    __syncthreads(); // incorrect barrier usage, as not
     all threads can access it
  }
}
// correct CUDA program, generated by the anti-
    coalescing transformation
__global__ void transformed_cuda_Histogram(uint32_t *
    pixels, uint32_t num_pixels) {
    uint32_t priv_hist[256];
    uint32_t gsize = blockDim.x * gridDim.x;
    uint32_t index = blockDim.x * blockIdx.x+threadIdx.
     x;
    __shared__ bool has_activated_thread;
    bool activated = true;
    do {
        has_activated_thread = false;
        __syncthreads(); // correct barrier usage, all
     threads can access it
        activated &= (index < num_pixels);
        has_activated_thread |= activated;
        if (activated) {
          uint32_t color = pixels[index];
          priv_hist[color]++;
          index += gsize;
        }
    } while (has_activated_thread);
}
```

Listing 2: The incorrect and correct solutions for inserting a barrier into the loop of the CUDA program in Listing 1.

Instead of trying to reordering the generated loops, the anti-coalescing transformation applies transformation on the CUDA program to **guide** the flat collapsing transformation directly generate $flat\ loop$ **inside** $original\ loop$.

The anti-coalescing transformation utilizes an internal property of the flat collapsing transformation. Specifically, as

discussed in [23], [27], ***when there is a barrier inside the original loop, the flat collapsing transformation will then generate the flat loop inside the original loop***. Thus, we could insert a barrier in the *original loop* to make the *flat loop* the inner loop in the transformed CPU program. However, directly inserting a barrier inside the original loop may generate incorrect CUDA programs. For example, at the top of Listing 2, the inserted barriers may not be reached by all threads. In CUDA, it results in undefined behavior. Thus, the anti-coalescing transformation has to apply extra transformations to transform the *original loop* to a do-while loop with auxiliary instructions to maintain correctness. The CUDA program transformed by the anti-coalescing transformation is shown at the bottom of Listing 2. The transformed CPU program is shown in Listing 3: with a barrier inside the transformed *original loop*, the flat collapsing transformation generates the *flat loop* (line 13th) inside the *original loop* (line 11th). In this order, the array index $index[tid]$ (line 17th) is linearly increased with the induction variable (tid) of the inner loop, which achieves high spatial locality.

2) Implementation: The anti-coalescing transformation is a pattern matching and rewriting method: it first scans the CUDA program to identify code sections that contain global memory coalescing patterns. Then, it applies compiler transformations to these sections.

```
1  bool has_activated_thread;
2  void cpu_Histogram(uint32_t *pixels, uint32_t
       num_pixels) {
3    uint32_t priv_hist[BLOCK_SIZE][256];
4    uint32_t gsize = BLOCK_SIZE * GRID_SIZE;
5    uint32_t index[BLOCK_SIZE];
6    for (int tid = 0; tid < BLOCK_SIZE; tid++)
7      index[tid] = BLOCK_SIZE * BLOCK_ID + tid;
8    bool activated[BLOCK_SIZE];
9    for (int tid = 0; tid < BLOCK_SIZE; tid++)
10     activated[tid] = true;
11   do {  // original loop
12     has_activated_thread = false;
13     for (int tid = 0; tid < BLOCK_SIZE; tid++) { //
       flat loop
14       activated[tid] &= (index[tid] < num_pixels);
15       has_activated_thread |= activated[tid];
16       if (activated[tid]) {
17         uint32_t color = pixels[index[tid]]; // high
       locality
18         priv_hist[tid][color]++;
19         index[tid] += gsize;
20       }
21     }
22   } while (has_activated_thread);
23 }
```

Listing 3: The CPU program generated from the bottom of Listing 2, which achieves high spatial locality.

Detecting global memory coalescing: Although there are multiple ways to implement global memory coalescing, after analyzing CUDA programs in real applications, we have identified a common code pattern that satisfies three conditions:

1) global memory coalescing is always implemented in loop constructs;
2) the stride value in that loop has a linear relationship with the GPU block dimension value;
3) there is a global memory access inside the loop, and the index used to access has a linear relationship with the loop induction variable and the thread index.

Our compiler analysis takes the loops in the CUDA function to check if they satisfy the 2nd and 3rd conditions listed above. For example, the code in Figure 3 contains a global memory coalescing pattern and satisfies all three conditions. It includes a for-loop construct (condition 1), and its stride ($gsize$) has a linear relationship with the block dimension (condition 2). Additionally, there is a global memory access inside the loop, and the $index$ has a linear relationship with the loop induction variable and thread index (condition 3). Therefore, the anti-coalescing transformation identifies this code section as a global memory coalescing code section.

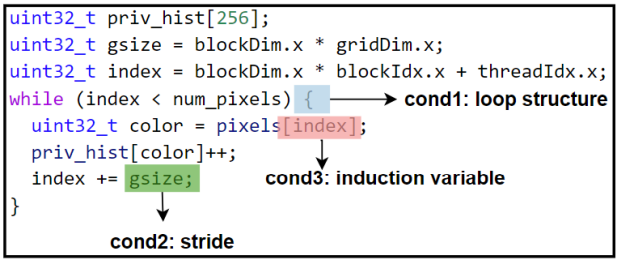


Fig. 3: The code sections that satisfy all conditions are identified as global memory coalescing code sections.

Applying the transformation: In the second phase, the anti-coalescing transformation applies code transformation to those detected code sections, i.e. loop nest. The actual transformation is to visit the loop nest in a top-down manner and rewrite each level of the loop by substituting loop-iterator related operations and loop body into a new loop template. As shown in Figure 4, the anti-coalescing transformation extracts loop-body (red), loop-cond (blue), loop-initialization (green), and loop stride (orange) information from the original loop structure, and fills them into a template to get the new loop. The template provides auxiliary variables and related operations that record whether a CUDA thread is still activated ($activated$) and whether there are any activated threads in the CUDA block ($has\_activated\_thread$). The former is required for each thread, and thus, is defined as a CUDA local variable, while the latter is shared within a block and defined as a CUDA shared variable. To guide the flat collapsing transformation generates the *flat loop* as the inner loop, a barrier operation is applied before executing the original loop body.

3) Proof: In this section, we provide a theoretical proof of the correctness of anti-coalescing transformation. To guarantee correctness, the transformed program must preserve identical semantics in terms of data access. Considering CUDA's SPMD model, the correctness of data access is assured by ensuring the accuracy of the workload distribution. Specifically, anti-coalescing transformation applies transformations only to loops that contain global memory coalescing. Therefore, the most critical aspect is to ensure that the loop bodies are executed the same number of times and that the induction variables maintain original values across all iterations.

For the original loop (left of Figure 4), each thread may

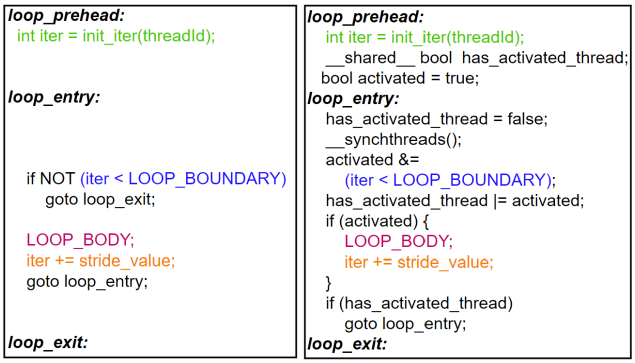


Fig. 4: The anti-coalescing transformation rewrites the original loop by extracting its iterator information and loop body, filling them into a template to construct the new loop.

have a different loop length. For thread t, the loop length is

$$original\_length = \left\lceil \frac{LOOP\_BOUNDARY - init\_iter(t)}{stride\_value} \right\rceil \tag{1}$$

In the transformed loop (right of Figure 4), let $iter_i^t$ denote the value of the $iter$ variable for thread t in iteration i. Then, the following relationship can be established:

$$iter_0^t = init\_iter(t) \tag{2}$$

$$iter_i^t = iter_{i-1}^t + stride\_value \tag{3}$$

In the transformed loop, the original loop body resides within the generated do-while loops and is wrapped by an if-statement with the conditional variable $activated$. The do-while loop is governed by the shared variable $has\_activated\_thread$, indicating that all threads possess the same loop length. Let $has\_activated\_thread_i$ represent the value of the shared variable $has\_activated\_thread$ in the i th iteration, and $activated_i^t$ denote the value of $activated$ in the i th iteration for thread t. The following relationships are then established:

$$has\_activated\_thread_i = \bigvee_{t=0}^{n-1} activated_i^t \tag{4}$$

$$activated_i^t = \bigwedge_{p=0}^{i} (iter_p^t < \text{LOOP\_BOUNDARY}) \tag{5}$$

Combing Eq 4, Eq 5 and Eq 3, we get Eq 6.

$$activated_i^t = \begin{cases} true, & \text{if } i < \left\lceil \frac{\text{LOOP\_BOUNDARY} - \text{init\_iter(t)}}{\text{stride\_value}} \right\rceil \\ false, & \text{otherwise} \end{cases} \tag{6}$$

We understand that for each iteration i, the loop body is executed only when both $has\_activated\_thread_i$ and $activated_i^t$ are true. From Eq 4, it is evident that when $activated_i^t$ is true, $has\_activated\_thread_i$ is invariably true as well. Additionally, according to Eq 6, the $activated$ variable remains true for the first $\left\lceil \frac{\text{LOOP\_BOUNDARY} - \text{init\_iter(t)}}{\text{stride\_value}} \right\rceil$ iterations. Consequently, the loop body in the transformed loop is executed the same number of times as in the original loops (Eq 1).

B. Flat Loop Optimization

All transformed CPU programs are loop intensive. As illustrated in Figure 1, flat collapsing transformation typically generates a significant number of flat loops. Each iteration within these loops represents a CUDA thread. These loops have a desirable characteristic — they are parallelizable. Since GPU programs generally lack dependencies among threads, the corresponding transformed flat loops should similarly exhibit no inter-iteration dependencies. Therefore, optimizing these flat loops is crucial for achieving high performance. While loop optimization is a well-explored area for traditional CPU programs, our evaluation reveals that modern compilers' loop optimization strategies are often too generic and conservative for these specific flat loops. In response, we propose two compiler transformations that convert the programs into formats that enable more compiler optimizations.

1) Block Size Invariant Analysis: All flat loops have the GPU block size as their length. The block size is a variable initialized in the host program and passed to the kernel program during kernel launch. Since the host and kernel programs are **distinct** modules that do not share information during compilation, flat collapsing transformation cannot access the block size from the host program while transforming the kernel program. Therefore, flat collapsing transformation makes GPU block size a runtime variable. Consequently, all flat loops are dynamic loops, which are difficult to optimize.

Another critical observation is that, in real-world GPU applications, block sizes are often determinable through static analysis. Developers commonly choose constant block sizes for two main reasons. First, it facilitates efficient utilization of shared memory and synchronization within a CUDA SM. Second, GPUs traditionally impose a limit on the maximum number of threads per block to align with hardware constraints.

Based on these observations, we propose block size invariant analysis, which analyzes the **host program** and utilizes the results to optimize the **kernel program**, thereby converting dynamic loops to static loops. The block size invariant analysis represents an inter-module optimization, which is challenging to implement within modern compiler infrastructures.

The block size invariant analysis analyzes the original GPU program and applies transformations to the CPU program. Specifically, it analyzes kernel launch instructions in CUDA host programs to determine block sizes for a CUDA kernel f_k. It then generates a wrapper function k_w that contains multiple versions of f_k's body, each tailored to different block sizes. The block size invariant analysis also replaces the original call sites of f_k with k_w. The process is illustrated in Figure 5.

The block size invariant analysis initially applies a compiler analysis that constructs pointer alias information and a call graph. It then performs inter-procedural constant propagation, which helps identify constant block sizes from the host programs and passes these to the kernel programs. The block

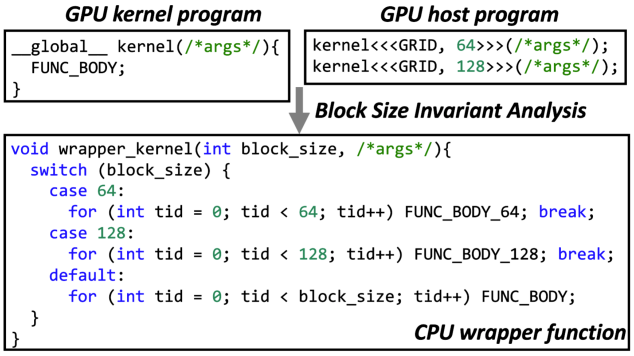


Fig. 5: Wrapper function generation.

size invariant analysis subsequently synthesizes a wrapper function composed of several versions of the kernel function. This is achieved by substituting the runtime block size variable with constants detected during the analysis phase. Ultimately, the original call sites are modified by replacing the kernel function with this wrapper function.

The block size invariant analysis always maintains correctness, even in cases where the block size cannot be analyzed at compile time. As demonstrated in Figure 5, the generated switch instruction contains a default branch for cases where the block size cannot be predicted. In such cases, the block size will be a runtime variable, and the original program generated by flat collapsing transformation will be executed.

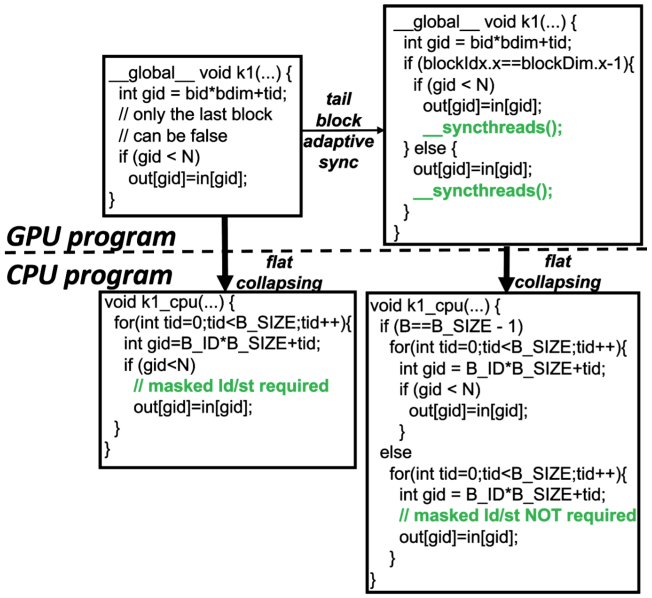


Fig. 6: An example of tail block adaptive synchronization.

2) Tail Block Adaptive Synchronization: GPU programs exhibit parallelism at two levels: grid and block. Consequently, tail blocks are commonly required to manage edge cases, as illustrated at the top-left of Figure 6. When the block size is not a divisor of N, certain threads in the last block should not activate, necessitating an if-statement branch to filter out these inactive threads. However, this conditional branch is also applied to other blocks. In the transformed CPU program, shown at the bottom-left, masked load/store operations are required when vectorizing these flat loops. For CPUs that have no or limited support for masked instructions, this will hinder the utilization of SIMD instructions.

Based on the observation, we propose tail block adaptive synchronization. Similar to block size invariant analysis, it is a cross-module analysis. It analyzes the grid/block size information from the host program and uses this information to detect condition statements in the kernel program that only diverge at the tail block. For example, in the top-left CUDA program in Figure 6, the condition $if(gid < N)$ will be true for all blocks except the tail block. Thus, the if statement can be eliminated in these blocks.

The tail block adaptive synchronization is applied to the CUDA program, resulting in the modified CUDA program (top-right) that retains the if-statement only in the tail block, while eliminating it from all other blocks. Consequently, the transformed CPU programs, depicted at the bottom-right, have a reduced requirement for masked load/store operations. For those architectures that do not support masked load/store instructions, the tail block adaptive synchronization brings higher performance.

It is important to note that existing compiler optimizations cannot achieve the same transformation. For compiler optimizations designed for GPU execution, this kind of transformation (from Figure 6 top-left to top-right) will not achieve any speedup on GPU execution since it introduces extra synchronization and does not decrease the total number of if-statements for each thread. Similarly, CPU optimizations (e.g., loop peeling, loop unswitching) cannot translate the CPU program from bottom-left to bottom-right in Figure 6 either. This is because the branch condition depends not only on the loop induction variable (tid), but also on the block size and grid size, which are runtime variables passed from the host module. By analyzing host programs, tail block adaptive synchronization can apply code transformations that cannot be done by existing compiler optimizations.

C. GPU-Block Dynamic Tiling

1) Insight: As introduced in Section III-C, directly mapping CUDA blocks to CPU threads often yields suboptimal results. This is because, from the hardware perspective, GPU SMs and CPU cores have different capacities.

To mitigate the mismatch, we propose GPU-block dynamic tiling, enabling runtime redistribution of the CUDA workload across CPU cores to accommodate architectural differences and improve computational efficiency. Instead of mapping a single GPU block to a CPU thread, the GPU-block dynamic tiling maps *N* GPU blocks to a CPU thread. The optimal *N* depends on both the characteristics of the CUDA kernels and the capacity of the CPU. The GPU-block dynamic tiling initially tiles GPU blocks such that the total number of tiled blocks matches the number of CPU cores, with the tiled block size equal to $\frac{grid\_size \times block\_size}{number\_of\_CPU\_cores}$. This approach ensures full

CPU utilization and avoids unnecessary context switches as the number of thread is same as the number of CPU cores.

Furthermore, we find that for some CUDA programs, particularly those with lightweight kernels, further tiling the GPU blocks can enhance performance. This seemingly counterintuitive phenomenon arises from the fact that additional tiling, although it does not utilize all CPU cores, reduces the overhead associated with CPU synchronization. The execution of lightweight kernels primarily suffers high CPU synchronization overhead; hence any reduction in this aspect can improve overall performance. Additionally, further tiling of GPU blocks enables more effective use of CPU computation resources, such as increasing cache locality by aggregating more CUDA blocks be executed by the same CPU cores.

Based on the observation, we propose GPU-block dynamic tiling to dynamically adjust the tile size. This optimization is motivated by the observation that for most CUDA programs, the same kernel is launched multiple times with similar workloads, e.g., the same kernels are launched iteratively to process different batches in batch processing programs. GPU-block dynamic tiling is a hill-climbing-like approach: it iteratively increases the tile size and measures the achieved speedup for kernel execution until no further performance improvement is observed. Therefore, this dynamic approach provides a more efficient mapping of CUDA blocks onto CPU cores, particularly for lightweight kernels. Compared with a normal kernel launch, the tuning process only includes extra simple calculations and time measurements, resulting in negligible overhead. In our evaluation, we include the tuning overhead in the kernel execution time to demonstrate the overall runtime improvement.

2) Implementation: The GPU-block dynamic tiling is integrated into the kernel launch function as a runtime optimization. Upon kernel launch, the optimization first determines if the kernel has been executed previously. If not, the tiled block size is set to $\frac{grid\_size \times block\_size}{number\_of\_CPU\_cores}$, which matches the number of CPU cores. Otherwise, if the kernel has been executed before, the method doubles the last tiled block size to further grouping the blocks into CPU cores. The execution time with the new tile size is then measured and compared with the runtime of previous tile size. If the current tile size exhibits superior efficiency, the optimal solution is updated accordingly. Conversely, the tile size is set back to the previous one and remains invariant for subsequent kernel launches.

An example of the GPU-block dynamic tiling process is demonstrated in Figure 7. For a GPU kernel with 128 blocks, when executed on a 4-core CPU, at the first launch, GPU-block dynamic tiling assigns blocks to four threads to fully utilize all CPU cores. At the second launch, GPU-block dynamic tiling tries to double the tile size. Thus, only two threads are required, each executing 64 blocks. After execution, GPU-block dynamic tiling detects a speedup compared to the first iteration. Therefore, it tries to further increase the tile size in the third kernel launch but detects a slowdown. Consequently, GPU-block dynamic tiling rolls back to the best-evaluated tile size for all remaining kernel launches.

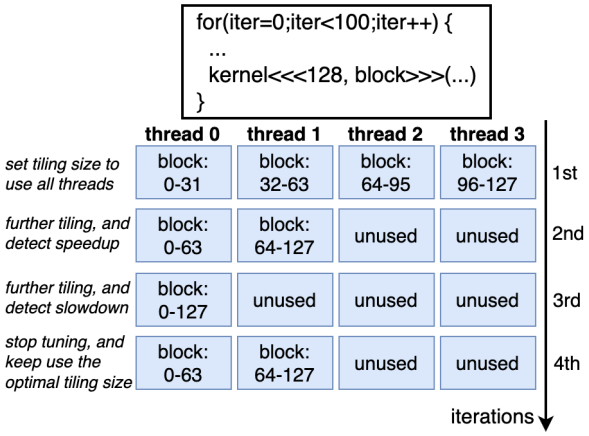


Fig. 7: The process of GPU-block dynamic tiling.

D. Scope of Proposed Optimizations

We summarize the scope of GPU programs that benefit from each proposed optimization: anti-coalescing transformation is used to optimize GPU kernels that contain global memory coalescing code sections; block size invariant analysis is suitable for GPU programs whose block sizes are static values in host programs; tail block adaptive synchronization can be used to optimize GPU kernels that contain branch instructions which diverge only for the last block; GPU-block dynamic tiling is a runtime optimization and requires that kernels are launched multiple times with similar workloads.

V. Evaluation

To illustrate the compatibility of the proposed optimizations with flat collapsing transformation, we integrate these optimizations into CuPBoP [21], [22] (commit: fd5681), a GPU-to-CPU framework that utilizes flat collapsing transformation. To execute CUDA source code on CPUs, the source code is first compiled to LLVM IR. Then, the proposed compiler optimizations and flat collapsing transformation are applied as LLVM-IR transformations. The transformed IR is finally compiled to CPU executable files.

Additionally, two other GPU-to-CPU solution, DPC++ [26] (version 2024.0.2) and MCUDA [49] (version 1.0.1), are used as the baselines.

Two platforms are utilized for evaluation: one is equipped with two Intel Gold 6226R CPUs (x86), and the other features a single ARM A64FX CPU (AArch64). To avoid randomness in runtime evaluation, we run each evaluation 7 times and report the median as the final result.

We use benchmarks from different areas: Parboil [48], Rodinia [10], and Hetero-mark [50] offer CUDA implementations of conventional HPC applications. Polybench [18] comprises linear algebra operations implemented in various languages. We do not evaluate all applications in these benchmarks, since some applications are not supported by DPC++, CuPBoP and MCUDA: MCUDA does not support C++ syntax, dynamic shared memory, or using integer types for grid/block sizes. Therefore, for almost all benchmarks, we must apply manual

preprocessing and post-processing to the source code, resulting in a significant workload. Similarly, DPC++ and CuPBoP do not support texture memory and atomic instructions. In general, these GPU-to-CPU solutions are error-prone; therefore, we only evaluate CUDA benchmarks that can be successfully migrated.

For all evaluations, we apply compiler O3 pipeline to the transformed CPU programs.

We use CuPBoP as the baseline, and demonstrate the runtime improvement achieved by applying all proposed optimizations in Figure 8. While all optimizations are applied **simultaneously**, it is observed that different applications experience varying degrees of speed-up from each optimization. Consequently, applications are categorized based on the optimization that delivers the most significant impact on runtime. For the x86 (Figure 8a), out of all 16 applications, 4 demonstrate a notable speed-up due to anti-coalescing transformation, 9 due to block size invariant analysis, and 3 owing to GPU-block dynamic tiling. On average, the proposed optimizations contribute to a 20.84% speed-up. Analyzing each optimization individually, anti-coalescing transformation contributes to a 24.46% speed-up for applications that benefit from it, while block size invariant analysis and GPU-block dynamic tiling result in speed-ups of 14.18% and 53.46%, respectively. Since x86 supports masked ld/st, applying tail block adaptive synchronization does not yield significant improvement.

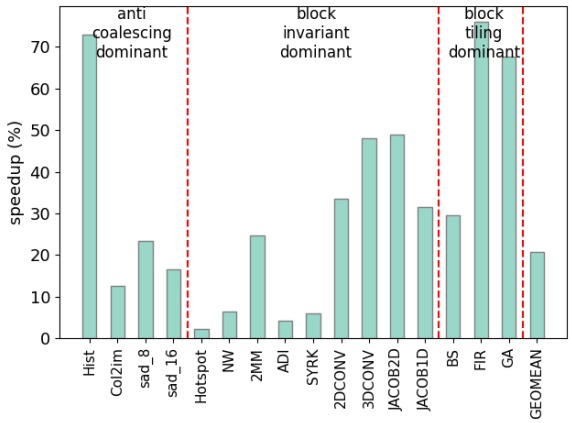
A similar trend is observed for ARM CPUs, as depicted in Figure 8b. Since ARM CPUs do not support masked load/store, four applications benefit from tail block adaptive synchronization, which can be optimized using SIMD instructions to achieve a speed-up of 25.23%. The cumulative effect of all optimizations results in a 16.10% speed-up.

Comparing the runtime with other GPU-to-CPU solutions (Figure 9), our solution achieves the highest performance across all benchmarks, indicating that the observed speed-up from the proposed optimizations is not due to potential implementation drawbacks in CuPBoP. Our solution is 47.39%, 51.46%, 20.84% faster than MCUDA, DPC++, and CuPBoP on average.

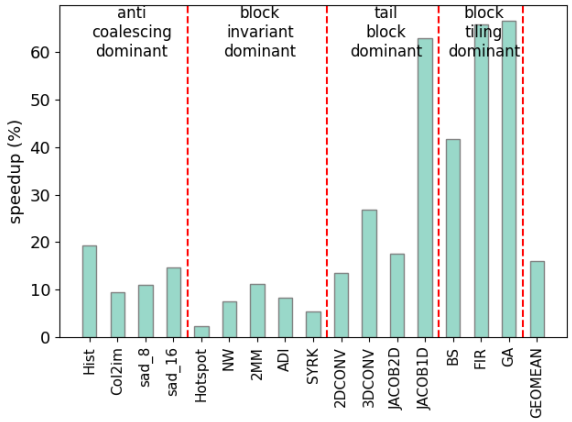
In the following sections, we evaluate the effect of each optimization individually. The anti-coalescing transformation, block size invariant analysis, and GPU-block dynamic tiling optimizations are evaluated using the Intel CPU, which offers mature profiling toolkits. The tail block adaptive synchronization is assessed on the ARM CPU which does not support masked load/store operations.

A. Anti-Coalescing Transformation

We evaluate the effectiveness of anti-coalescing transformation on five CUDA kernels that leverage global memory coalescing. The performance of the transformed CPU kernels, both with and without anti-coalescing transformation, is compared in Figure 10. The results demonstrate that anti-coalescing transformation significantly reduces the kernel execution time for all kernels. This reduction is attributable to the improved cache locality, as evidenced by the decrease in the number of LLC misses (middle of Figure 10). It is important to recognize that the anti-coalescing transformation introduces extra branch



(a) Intel x86 CPU evaluation.



(b) ARM AArch64 CPU evaluation.

Fig. 8: The speed-up achieved by the proposed optimizations. Although all four optimizations are applied **concurrently**, programs exhibit varied improvements from each. Consequently, programs are categorized based on the optimization that yields the most significant speed-up.

instructions in the transformed code. To assess this overhead, we present the results at the bottom of Figure 10. Despite the extra branch instructions, the overhead is negligible compared to the achieved runtime improvement.

In addition to enhanced cache utilization, we also observe that the CPU program optimized by anti-coalescing transformation provides more opportunities for leveraging SIMD instructions, as illustrated in Table II. This is due to the fact that the optimized programs' inner loops access memory addresses sequentially, a pattern that compilers are more likely to optimize through vectorization transformations. Nevertheless, the most significant speed-up is attributed to improved cache utilization. When evaluating with loop-vectorization settings turned off, all five applications still achieve the same speed-up.

B. Block Size Invariant Analysis

The block size invariant analysis enables the generation of flat loops with constant lengths and loop bodies with constant block sizes, which are more amenable to optimization

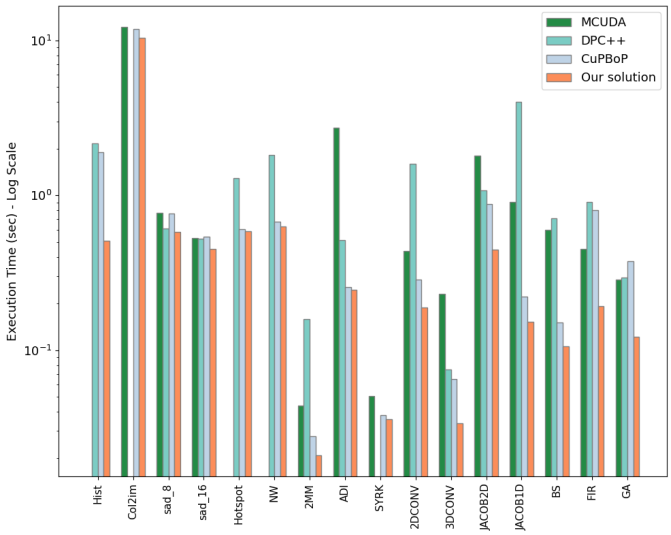


Fig. 9: The runtime result comparision with three CUDA-to-CPU solutions: DPC++(commercial), MCUDA (academic), and CuPBoP (academic).

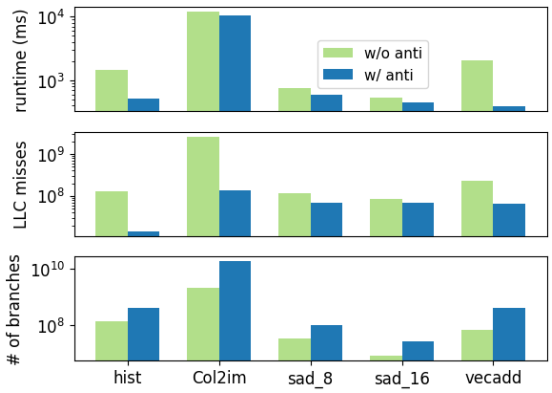


Fig. 10: The runtime, number of LLC misses and executed branch instructions on transformed CPU programs.

App.	Anti	AVX Instructions		
		128	256	512
Hist	w/	190,063,921	0	93,068,672
	w/o	190,067,557	0	13,483,904
Col2im	w/	22,884,123,056	11,475,615,858	23,085,449,918
	w/o	0	0	0
Sad_8	w/	33,554,561	4,194,320	22,020,182
	w/o	0	0	0
Sad_16	w/	5	16	1,048,620
	w/o	1	0	3
Vecadd	w/	213,961,856	58,722,304	71,574,784
	w/o	0	0	0

TABLE II: Number of executed AVX instructions with and without anti-coalescing transformation.

by standard compiler techniques. The block size invariant analysis achieves a significant speed-up (over 30%) for 2DCONV, 3DCONV, JACOB1, and JACOB2D applications, as it enables loop vectorization to generate programs that utilize SIMD instructions effectively. All these applications employ CUDA blocks with two dimensions. Since flat collapsing transformation generates a single for-loop to simulate all threads, it incurs additional instructions such as $threadIdx.y = tid\%block\_size.y$ to calculate the thread index in all dimensions. These instructions are associated with DIV and REM operations. For the LLVM compiler, its cost function is affected by these instructions, leading to an estimated overhead for vectorization exceeds that of the scalar version, thereby preventing the application of loop vectorization to these programs. With block size invariant analysis, the block size variables are replaced with constants. Notably, as most block size values are powers of two, these expensive DIV/REM operators can be replaced by SHIFT and AND operators. Consequently, the compiler opts to apply vectorization on the transformed programs, thereby enhancing the utilization of SIMD instructions. The utilization of AVX instructions is documented in Table III.

App.	invariant	AVX Instructions		
		128	256	512
2DCONV	w/	23,134,208	0	78,249,984
	w/o	0	0	0
3DCONV	w/	28,285,440	0	102,737,920
	w/o	0	0	0
JACOB1D	w/	41,120,000	30,720,000	100,800,000
	w/o	0	0	0
JACOB2D	w/	5,200,000	0	26,880,000
	w/o	0	0	0

TABLE III: The number of executed AVX instructions with-/without the block size invariant analysis.

In addition to loop vectorization, the constant block size further facilitates other compiler optimizations, including constant propagation and constant folding. As a result, applications not optimized by loop vectorization still attain improvements. Owing to the ubiquity of flat loops and instructions that utilize block size variables as operands in the transformed CPU programs, almost all programs benefit from block size invariant analysis to varying extents.

C. Tail Block Adaptive Synchronization

The tail block adaptive synchronization is specifically designed for CPUs lacking support for masked load/store instructions. Therefore, its effect is evaluated on the ARM CPU. Four applications — 2DCONV, 3DCONV, JACOB1D, and JACOB2D — are transformed using tail block adaptive synchronization, and each experiences a significant speed-up of 13.63%, 26.81%, 62.92%, and 17.60% respectively.

This speed-up is attributed to the utilization of SIMD instructions, as detailed in Table IV. Without tail block adaptive synchronization, all load/store instructions reside within the

if-statements, necessitating masked load/store for vectorization. However, with tail block adaptive synchronization, most load/store instructions, except those executed by the last blocks, are moved out of the conditional statement, thereby allowing optimization using general vectorization instructions provided by ARM CPUs. This substantial amount of SIMD computation results in significant speed-up,

App.	Tail	simd_inst	ase_spec	ase_sve_int
2DCONV	w/	37 483 230	33 541 230	31 398 389
	w/o	4 222 977	2172	1724
3DCONV	w/	55 305 575	48 205 721	34 415 726
	w/o	7 087 320	2438	1744
JACOB1D	w/	92 277 566	91 866 604	85 218 510
	w/o	24 408	22 046	1693
JACOB2D	w/	124 677 629	123 423 652	98 905 416
	w/o	565 787	1964	1690

TABLE IV: The number of executed SIMD instructions with/without the tail block adaptive synchronization.

D. GPU-Block Dynamic Tiling

The GPU-block dynamic tiling is tailored for GPU programs that iteratively launch the same kernels. We evaluate the tiling processes of four CUDA benchmarks that iteratively launch same kernels at least 10 times, with results presented in Figure 11. In the initial phase, the tiled block size is set to match the number of CPU cores. The tiling size is gradually increased during execution, until further increases do not yield any additional speed-up.

We compared the execution time with the workload configurations in CuPBoP, which equally distribute CUDA blocks to all CPU cores. Three programs (BS, FIR, and GA) benefit from the GPU-block dynamic tiling, achieving speed-ups of 29.67%, 75.98%, and 67.78%, respectively. For the Hist applications, optimal CPU performance is already achieved by evenly distributing blocks across all CPU cores

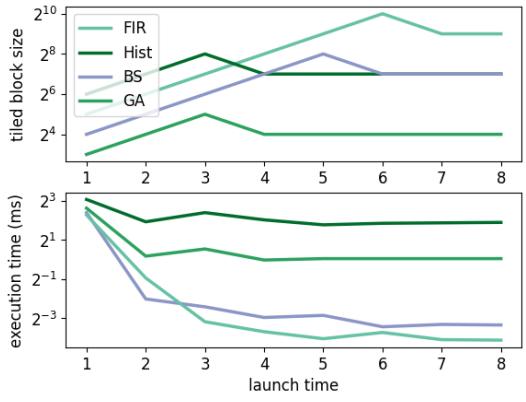


Fig. 11: The process of the GPU-block dynamic tiling.

The FIR benchmark experiences the most significant speed-up. Initially, each CUDA block contains only 32 threads, each has a lightweight workload of just 32 FLOPs and minimal memory access. As a result, the transformed CPU program could not fully utilize the available CPU resources. In its default configuration, each kernel launch involved 128 CUDA blocks, translating to 128 CPU threads in the transformed CPU program. This setup led to a substantial synchronization overhead relative to the kernel workload. With the application of GPU-block dynamic tiling, each CPU core is tasked with executing 16 CUDA blocks, reducing the number of required CPU threads to eight and thus diminishing the kernel launch and synchronization overhead. Such an adjustment enables the FIR application to substantially benefit from the optimization.

VI. Related work

For more than a decade, researchers have sought to achieve high performance for GPU programs on CPUs. To bridge the parallelism gap between these two architectures, the researchers in MCUDA [49] propose a compiler transformation, flat collapsing transformation, that generates loops to wrap CUDA kernels. This transformation enables the use of fewer CPU threads, with each thread handling a heavier workload that is better suited for CPU architectures. The transformation has been widely used for GPU-to-CPU migration. For example, POCL [27] and SYCL [24] use it to support CPU backends for OpenCL and SYCL languages.

This transformation is widely used [15], [21], [29], [47] with different implementations: Moses et al. [40] implement the transformation on MLIR [39]; Ocelot [12] and CuPBoP [21] implement it as LLVM IR transformations; and Cumulus [9] implements it as a CUDA-to-C++ transformation. We implement the proposed optimizations as LLVM IR transformations to make them compatible with CuPBoP.

Some researchers expand the flat collapsing transformation: COX [23] proposes the hierarchical collapsing to support CUDA warp-level functions, and [20] proposes additional compiler analysis to detect implicit barriers in CUDA programs to ensure the correctness of the flat collapsing transformation.

The flat collapsing transformation focuses solely on barrier instructions. To ensure program correctness, it generates separate loops to wrap instructions before and after these barriers. For instructions that are not barriers, flat collapsing transformation simply wraps them into loops. Therefore, there is potential for further optimization, as discussed in this paper.

OCCA [38] introduced a library-based approach that utilizes APIs to abstract backend and kernel languages, making it compatible with various parallel programming models such as OpenMP and OpenCL. The library employs Just-In-Time compilation to compile kernel functions and offers minor extensions to C to facilitate programming for each backend. Similarly, Kokkos [51] is designed to provide an easy-to-program solution for various types of devices, such as CPU, GPU, and FPGA, through its C++ library. It offers an abstraction layer that separates user algorithms from execution details, enabling performance optimization across different architectures. The optimizations introduced in this paper are also applicable to these two library-based approaches.

There are several projects [14], [35]–[37], [44], [45], [53] that address the differences in high-performance programs on CPUs and GPUs. Majeti et al. [35] propose a source-to-source translation that converts data structures (e.g., AOS to SOA) based on the target backend to optimize CPU and GPU preferences. In [53], tunable factors like caching, tiling, and prefetching are manually tuned for OpenCL programs to improve performance on both CPUs and GPUs. On the other hand, Falch et al. [14] present an auto-tuning approach based on machine learning. These factors also influence energy efficiency, as explored in [45]. These methods require developers to provide explicit tuning knobs in programs to achieve performance portability across different hardware. In contrast, our approach can be applied to general GPU programs without the need for explicit tuning knobs.

VII. Discussions

A. Why Existing Compiler Optimizations Cannot Help?

In Section V, for the baseline, we apply LLVM's O3 optimization on all CPU programs. Despite this, it does not achieve the same performance as our proposed optimizations.

We also attempt to use Polly [19], a popular Polyhedral model compiler, to optimize the CPU programs generated by flat collapsing transformation. However, we find that it cannot analyze most of the programs. To apply Polyhedral model optimization effectively, the loop sections must be static control parts (SCoPs) [16], [17], where loop bounds and memory dependencies are amenable to static analysis. Unfortunately, the CPU programs generated by flat collapsing transformation typically contain indirect memory accesses, which pose significant challenges for memory dependency analysis and prevent the polyhedral compiler from functioning effectively. Specifically, flat collapsing transformation expands local variables into arrays, transforming scalars into array elements. Consequently, a CUDA program that accesses memory using a scalar index might be transformed into one involving indirect memory access. For example, flat collapsing transformation converts idx to $idx[tid]$ in Figure 1. When we attempt to optimize this CPU program using Polly, it reports the warning message "The array subscript of 's' is not affine," leaving the code section unchanged.

Similarly, the complexities introduced by flat collapsing transformation also prevent other general compiler optimizations. Thus, in this paper, we summarize the common patterns in CPU programs generated by flat collapsing transformation and propose four optimizations accordingly, which can be used complementarily with standard compiler optimizations.

B. Do These Optimizations Nullify CUDA Optimizations?

The tail block adaptive synchronization, block size invariant analysis, and GPU-block dynamic tiling can be applied alongside general CUDA programs and are not related to any CUDA-specific optimizations.

It is true that anti-coalescing transformation undoes what programmers have intentionally implemented. Specifically, anti-coalescing transformation reverses the global memory coalescing optimization, which intentionally makes threads access interleaving memory addresses. However, this reversal is necessary to provide portability for **real-world** GPU programs. All GPU-to-CPU solutions aim to migrate unchanged GPU programs, and most off-the-shelf GPU programs are optimized specifically for GPU architectures. Thus, reversing GPU optimizations is necessary to achieve performance portability.

C. Why Run GPU Programs on CPUs?

There are multiple benefits to executing GPU programs on CPUs. First, it allows workload sharing between GPUs and CPUs, which can address the imbalance of utilization in CPU-GPU clusters, thereby reducing runtime [30], [32] and lowering energy consumption [46]. Second, the CPU offers a more mature development environment, allowing developers to use CPU toolkits to debug GPU programs [7], [9], [13], [43]. Lastly, it provides a platform for education and research for those who do not have access to GPUs and wish to explore proof-of-concept GPU programming.

We acknowledge that supporting the translation of CPU programs to GPUs is also important. Some researchers focus on executing OpenMP programs written for CPUs on GPUs [11], [25], [33], [34], [42]. PPCG [52] is a source-to-source translator based on polyhedral analysis, which translates sequential CPU programs to GPU programs.

VIII. Conclusion

The flat collapsing transformation is widely used for GPU-to-CPU migration solutions. Nevertheless, there is untapped potential to achieve superior performance since the transformed CPU programs do not fully utilize CPU computational resources. We have identified three unaddressed challenges in current solutions. Accordingly, we introduce four compiler/runtime optimizations to facilitate the generation of more performance-efficient CPU programs. We integrate these optimizations into an off-the-shelf GPU-to-CPU solution and demonstrate the achieved speedup through experiments. We evaluate the speed-ups of CPU programs on both x86 and ARM architectures. On average, the optimized applications yield speed-ups of 20.84% on x86 CPUs and 16.10% on ARM CPUs.

These optimizations pave the way for further advancements in the performance portability of GPU programs to CPUs, thereby making them more efficient and effective in heterogeneous computing environments.

Acknowledgements

This research was partially supported by NSF PPOSS 2119523, Intel, AMD and BAH. We also want to acknowledge the research infrastructure and services provided by the Rogues Gallery testbed [28], 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. 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 NSF.

APPENDIX

A. Abstract

The artifact integrates the proposed compiler optimizations into CuPBoP [22] to provide a solution for executing GPU programs on CPUs. The artifact also provides three examples containing GPU programs that can be transformed by each of the proposed optimizations.

The LLVM IR-to-IR transformations in the artifact are the optimizations (anti-coalescing transformation, block size invariant analysis, and tail block adaptive synchronization) proposed in the paper.

B. Artifact check-list (meta-information)

- **Compilation: CUDA, LLVM and GCC.**
- **Run-time environment: Linux (Ubuntu 20.04).**
- **Hardware: x86 CPU.**
- **Output: CPU executable file.**
- **How much disk space required (approximately)?: 5 GB.**
- **How much time is needed to prepare workflow (approximately)?: 20 minutes.**
- **How much time is needed to complete experiments (approximately)?: 5 minutes.**
- **Publicly available?: Yes.**
- **Code licenses (if publicly available)?: Apache-2.0 license.**
- **Archived (provide DOI)?: 10.5281/zenodo.13250264.**

C. Description

1) How to access: The artifact can be downloaded from https://zenodo.org/doi/10.5281/zenodo.13250263. It is also available on GitHub at https://github.com/drcut/CuPBoP-MICRO-AE/tree/v1.0.0.

2) Hardware dependencies: x86 CPU.

3) Software dependencies:

- Ubuntu 20.04;
- GCC-8;
- LLVM-14;
- CUDA-10.1 toolkit;
- moodycamel::ConcurrentQueue (C++ library);

D. Installation

Download artifact: We recommend downloading the artifact from GitHub, which can automatically download the third-party libraries required for execution.

```
git clone --recursive --branch v1.0.0 --depth 1 https
    ://github.com/drcut/CuPBoP-MICRO-AE.git
```

(Optional) Download LLVM: The artifact requires LLVM-14. For users who do not have it installed, we recommend downloading the pre-built LLVM.

```
wget https://github.com/llvm/llvm-project/releases/
    download/llvmorg-14.0.0/clang+llvm-14.0.0-x86_64-
    linux-gnu-ubuntu-18.04.tar.xz
tar -xvf clang+llvm-14.0.0-x86_64-linux-gnu-ubuntu
    -18.04.tar.xz
export PATH=$PWD/clang+llvm-14.0.0-x86_64-linux-gnu-
    ubuntu-18.04/bin:$PATH
```

Build artifact:

```
cd CuPBoP-MICRO-AE
export CuPBoP_PATH=$PWD
mkdir build && cd build
# $CUDA_PATH should be set
CC=gcc CXX=g++ cmake ..
make
export CuPBoP_BUILD_PATH=$PWD
export PATH=$CuPBoP_BUILD_PATH/compilation:$PATH
export LD_LIBRARY_PATH=$CuPBoP_BUILD_PATH/runtime:
    $CuPBoP_BUILD_PATH/runtime/threadPool:
    $LD_LIBRARY_PATH
```

E. Experiment workflow

All evaluations follow the same workflow. First, the LLVM IR-to-IR translator applies the proposed optimizations. Then, the transformed IR is used to generate CPU executable files. Finally, the executable files are executed to verify correctness.

We provide the script `run.sh` for each evaluation, which integrates the above workflow. Users can directly execute `run.sh` to run the evaluation.

F. Evaluation and expected results

There are three independent evaluations for the three proposed compiler optimizations, respectively.

```
cd $CuPBoP_PATH/MICRO_AE_example/anti-coalescing-
    transformation
bash run.sh

cd $CuPBoP_PATH/MICRO_AE_example/block-size-invariant-
    analysis
bash run.sh

cd $CuPBoP_PATH/MICRO_AE_example/tail-block-adaptive-
    synchronization
bash run.sh
```

Anti-Coalescing: The input CUDA program contains a global memory coalescing access pattern. The translator is expected to detect the code section and apply the anti-coalescing transformation. The translator is expected to output:

```
Find global memory coalescing
Loop at depth 1 containing: %37<header><exiting>,%41<
    latch>
```

Block Size Invariant Analysis: The input CUDA program contains three different kernel launches, with block sizes of 16, 32, and 42. The translator is expected to collect all block sizes. Thus, it is expected to output:

```
possible block sizes:
x: 16 y: 1 z: 1
x: 32 y: 1 z: 1
x: 42 y: 1 z: 1
```

Tail Block Adaptive Synchronization: The input CUDA kernel contains an if condition that will only diverge at the last block (with block index equal to grid size - 1). Thus, the translator should apply tail block adaptive synchronization to the kernel. The translator should output:

```
Condition is always true (without tail block) with
    threshold 512 and global index ranges: [0, 496)
The function can be optimized by tail block adaptive
    sync
```

G. Notes

The workflow of executing CUDA source code on CPUs first requires compiling CUDA source code to LLVM IR, which necessitates the CUDA toolkit. In the artifact, we choose to directly provide the generated LLVM IR and focus on the evaluation of the IR-to-IR transformation. However, we also provide the CUDA source code (i.e., hist.cu and vecadd.cu). Although this source code is not directly used for artifact evaluation, it provides a more comprehensive understanding of the LLVM IRs used for evaluation.

REFERENCES

[1] "Hip," 2016. [Online]. Available: https://github.com/ROCm-Developer-Tools/HIP

[2] "A64fx specifications," 2018. [Online]. Available: https://www.fujitsu.com/global/about/resources/news/press-releases/2018/0822-02.html

[3] "Hip-cpu," 2020. [Online]. Available: https://github.com/ROCm-Developer-Tools/HIP-CPU

[4] "Nvidia a100 specifications," 2021. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9361255

[5] "Amd epyc™ 9654," 2023. [Online]. Available: https://www.amd.com/en/products/cpu/amd-epyc-9654

[6] "Intel xeon gold 6423," 2023. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/236591/intel-xeon-gold-6423n-processor-52-5m-cache-2-00-ghz/specifications.html

[7] T. M. Aamodt, W. W. Fung, I. Singh, A. El-Shafiey, J. Kwa, T. Hetherington, A. Gubran, A. Boktor, T. Rogers, and A. B. et al., "Gpgpu-sim 3.x manual," 2012.

[8] P. Barua, J. Shirako, and V. Sarkar, "Cost-driven thread coarsening for gpu kernels," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 1–14.

[9] V. Blomkvist Karlsson, "Cumulus-translating cuda to sequential c++: Simplifying the process of debugging cuda programs," 2021.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[11] A. Chikin, T. Lloyd, J. N. Amaral, E. Tiotto, and M. Usman, "Memory-access-aware safety and profitability analysis for transformation of accelerator-bound openmp loops," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 3, pp. 1–26, 2019.

[12] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "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, pp. 353–364.

[13] A. S. Elhelw and S. Pai, "Horus: A modular gpu emulator framework," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 104–106.

[14] T. L. Falch and A. C. Elster, "Machine learning based auto-tuning for enhanced opencl performance portability," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015, pp. 1231–1240.

[15] J. Fang, P. Zhang, T. Tang, C. Huang, and C. Yang, "Implementing and evaluating opencl on an armv8 multi-core cpu," in *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*. IEEE, 2017, pp. 860–867.

[16] P. Feautrier, "Parametric integer programming," *RAIRO-Operations Research*, vol. 22, no. 3, pp. 243–268, 1988.

[17] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *International Journal of Parallel Programming*, vol. 34, pp. 261–317, 2006.

[18] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 innovative parallel computing (InPar)*. Ieee, 2012, pp. 1–10.

[19] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-polyhedral optimization in llvm," in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, vol. 2011, 2011, p. 1.

[20] Z. Guo, E. Z. Zhang, and X. Shen, "Correctly treating synchronizations in compiling fine-grained spmd-threaded programs for cpu," in *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2011, pp. 310–319.

[21] R. Han, J. Chen, B. Garg, J. Young, J. Sim, and H. Kim, "Cupbop: A framework to make cuda portable," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023, pp. 444–446.

[22] R. Han, J. Chen, B. Garg, X. Zhou, J. Lu, J. Young, J. Sim, and H. Kim, "Cupbop: Making cuda a portable language," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 4, pp. 1–25, 2024.

[23] R. Han, J. Lee, J. Sim, and H. Kim, "Cox: Exposing cuda warp-level functions to cpus," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2022.

[24] L. Howes and M. Rovatsou, "Sycl integrates opencl devices with modern c++," *Khronos Group*, 2015.

[25] J. Huber, M. Cornelius, G. Georgakoudis, S. Tian, J. M. M. Diaz, K. Dinel, B. Chapman, and J. Doerfert, "Efficient execution of openmp on gpus," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022, pp. 41–52.

[26] Intel, "Dpct," 2020. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compatibility-tool.html

[27] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable opencl implementation," *International Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, 2015.

[28] A. Jezghani, J. Young, W. Powell, R. Rahaman, and J. E. Coulter, "Future computing with the rogues gallery," in *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2023, pp. 262–269.

[29] R. Karrenberg and S. Hack, "Improving performance of opencl on cpus," in *International Conference on Compiler Construction*. Springer, 2012, pp. 1–20.

[30] C. Lee, W. W. Ro, and J.-L. Gaudiot, "Boosting cuda applications with cpu–gpu hybrid computing," *International Journal of Parallel Programming*, vol. 42, no. 2, pp. 384–404, 2014.

[31] J. Lee, J. Kim, S. Seo, S. Kim, J. Park, H. Kim, T. T. Dao, Y. Cho, S. J. Seo, S. H. Lee *et al.*, "An opencl framework for heterogeneous multicores with local memory," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010, pp. 193–204.

[32] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Skmd: Single kernel on multiple devices for transparent cpu-gpu collaboration," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, pp. 1–27, 2015.

[33] S. Lee and R. Eigenmann, "Openmpc: Extended openmp programming and tuning for gpus," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[34] S. Lee, S.-J. Min, and R. Eigenmann, "Openmp to gpgpu: a compiler framework for automatic translation and optimization," *ACM Sigplan Notices*, vol. 44, no. 4, pp. 101–110, 2009.

[35] D. Majeti, R. Barik, J. Zhao, M. Grossman, and V. Sarkar, "Compiler-driven data layout transformation for heterogeneous platforms," in *Euro-Par 2013: Parallel Processing Workshops: HeteroPar*. Springer, 2014, pp. 188–197.

[36] D. Majeti, K. S. Meel, R. Barik, and V. Sarkar, "Automatic data layout generation and kernel mapping for cpu + gpu architectures," in *Proceedings of the 25th International Conference on Compiler Construction*, 2016, pp. 240–250.

[37] D. Majeti and V. Sarkar, "Heterogeneous habanero-c (h2c): a portable programming model for heterogeneous processors," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015, pp. 708–717.

[38] D. S. Medina, A. St-Cyr, and T. Warburton, "Occa: A unified approach to multi-threading languages," *arXiv preprint arXiv:1403.0968*, 2014.

[39] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist: Raising c to polyhedral mlir," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021, pp. 45–59.

[40] W. S. Moses, I. R. Ivanov, J. Domke, T. Endo, J. Doerfert, and O. Zinenko, "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*, 2023, pp. 119–134.

[41] A. Munshi, "The opencl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009, pp. 1–314.
[42] G. Ozen and M. Wolfe, "Performant portable openmp," in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, 2022, pp. 156–168.
[43] A. Patel, S. Tian, J. Doerfert, and B. Chapman, "A virtual gpu as developer-friendly openmp offload target," in *50th International Conference on Parallel Processing Workshop*, 2021, pp. 1–7.
[44] A. Qasem, A. M. Aji, and M. L. Chu, "Investigating data layout transformations in chapel," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 915–924.
[45] A. Qasem and S. Teich, "Evaluating the impact of data layout and placement on the energy efficiency of heterogeneous applications," in *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*. IEEE, 2017, pp. 1–8.
[46] E. Stafford, B. Pérez, J. L. Bosque, R. Beivide, and M. Valero, "To distribute or not to distribute: the question of load balancing for performance or energy," in *Euro-Par 2017*. Springer, 2017, pp. 710–722.
[47] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W.-m. W. Hwu, "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*, 2010, pp. 111–119.
[48] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, p. 27, 2012.
[49] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu, "Mcuda: An efficient implementation of cuda kernels for multi-core cpus," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2008, pp. 16–30.
[50] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. McCardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for cpu-gpu collaborative computing," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016, pp. 1–10.
[51] C. R. Trott *et al.*, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, 2022.
[52] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–23, 2013.
[53] Y. Zhang, M. Sinclair, and A. A. Chien, "Improving performance portability in opencl programs," in *Supercomputing: 28th International Supercomputing Conference, ISC 2013, Leipzig, Germany, June 16-20, 2013. Proceedings 28*. Springer, 2013, pp. 136–150.