GSOHC: Global Synchronization Optimization in Heterogeneous Computing

Soumik Kumar Basu
Department of Computer Science and Engineering, IIT Hyderabad, India

Jyothi Vedurada
Department of Computer Science and Engineering, IIT Hyderabad, India

Abstract

The use of heterogeneous systems has become widespread and popular in the past decade with more than one type of processor, such as CPUs, GPUs (Graphics Processing Units), and FPGAs (Field Programmable Gate Arrays) etc. A wide range of applications use both CPU and GPU to leverage the benefits of their unique features and strengths. Therefore, collaborative computation between CPU and GPU is essential to achieve high program performance. However, poorly placed global synchronization barriers and synchronous memory transfers are the main bottlenecks to enhanced program performance, preventing CPU and GPU computations from overlapping.

Based on this observation, we propose a new optimization technique called *hetero-sync motion* that can relocate such barrier instructions to new locations, resulting in improved performance in CPU-GPU heterogeneous programs. Further, we propose GSOHC, a compiler analysis and optimization framework that automatically finds opportunities for hetero-sync motion in the input program and then performs code transformation to apply the optimization. Our static analysis is a context-sensitive, flow-sensitive inter-procedural data-flow analysis with three phases to identify the optimization opportunities precisely. We have implemented GSOHC using LLVM/Clang infrastructure. On A4000, P100 and A100 GPUs, our optimization achieves speedups of up to 1.8x, 1.9x and 1.9x over the baseline, respectively.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Frameworks; Software and its engineering → Automated static analysis

Keywords and phrases Static Analysis, Synchronization, CPU-GPU, Heterogeneous Computing, Parallelization

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.2

Supplementary Material *Software (ECOOP 2025 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.11.2.17`

Funding *Soumik Kumar Basu*: Funded by SERB SRG.

Acknowledgements We thank Anilava Kundu for his insightful comments during implementation. Further, we thank the anonymous reviewers of ECOOP for their constructive feedback.

1 Introduction

Heterogeneous computing is a programming paradigm that uses more than one type of processor, such as a CPU, in conjunction with accelerators, such as GPUs (Graphics Processing Units) and FPGAs (Field Programmable Gate Arrays). In a wide range of applications, CPUs and GPUs are both used, and both have strengths and limitations, which are essential for high-performance computing. Over the decades, several compiler optimizations [45, 6, 38] have been proposed to improve the performance of CPU programs. In the same vein, numerous recent research works [12, 33, 51, 1, 2] have developed new analyses and optimizations to enhance only GPU (device) side performance in the CPU-GPU heterogeneous applications. Nevertheless, CPU- and GPU-only optimization strategies are incapable of realizing the full

```
 1 void foo(int *x, int size2)
 2 {
 3   ...
 4   if(k < y_points)
 5     x[k] = x[0] + y_points;
 6   ...
 7 }
 8 void main(){
 9   ...
10   kernel1<<<grid1, block1>>>(d_u,...);/*Device-side computations*/
11   cudaMemcpy(u, d_u, size1, cudaMemcpyDeviceToHost); /*Sync*/
12   ...
13   new_data = processData(data); /*Host-side computations*/
14   ...
15   cudaMemcpy(d_u, new_data, data_size, cudaMemcpyHostToDevice);
16   kernel2<<<grid2, block2>>>(d_u,...);/*Device-side computations*/
17   cudaMemcpy(v, d_u, size2, cudaMemcpyDeviceToHost);
18 
19   cudaDeviceSynchronize();/*Sync*/
20   ...
21   /*Host-side computations*/
22   foo(v, size2);
23   ...
24   for(int i=0; i<size2; i++)
25     v[i] = v[i] + x_points;  ...
26 }
```

Figure 1 Sample CUDA code snippet showing blocking calls, host- and device-side computations.

potential of heterogeneous computing. For example, a GPU can remain idle during memory copy to CPU, and specialized global optimizations [23, 4, 27] are needed to improve overall performance by efficiently managing host-to-device data transfers.

In a similar vein, heterogeneous systems suffer from another important global performance problem: the CPU being idle after launching the kernel and allocating the task to the GPU, resulting in the underutilization of processors. The importance of this problem has been revealed through several manual optimization efforts [7, 46, 55, 47] of several domain-specific heterogeneous applications (efficiently distributing the workload between CPU and GPU). Further, in a performance study by White III et al. [53], manual optimization of CPU-GPU computation overlap yielded significantly better performance than most other optimization methods, such as communication-computation overlap etc. A poorly placed synchronization statement or blocking call is most often the reason for the CPU idleness in a heterogeneous system since it prevents CPU and GPU computations from overlapping.

Optimization Opportunities and Associated Challenges. Consider a sample example in Figure 1. There are two device kernel launches from the CPU function `main`, one at Line 10 and the other at Line 16. A device kernel is a function that runs asynchronously with CPU code on multiple cores of a GPU (in SIMT [20] fashion), and we refer to its computations as *device-side computations*. Programmers must explicitly add synchronization calls (e.g. `cudaDeviceSynchronize` call at Line 19) in their code to ensure that CPU-side computations halt until the device-side computations are complete. Further, memory transfers (e.g. `cudaMemcpy` calls at Lines 11, 15, and 17) also act as synchronization/blocking calls in the CUDA programming model. We can see that the host-side computation at Line 13 is not dependent on the device-side kernel computation at Line 10, as it does not use variables `u`, `d_u` and `size1`. Nevertheless, the host-side computation still awaits for completion of the device kernel due to the poorly placed synchronous data-transfer call at Line 11, resulting in suboptimal performance. As a result, as shown in Figure 2 (above the horizontal dotted lines), the program takes $t_2 - t_0$ time to execute the program statements from Line 10 to

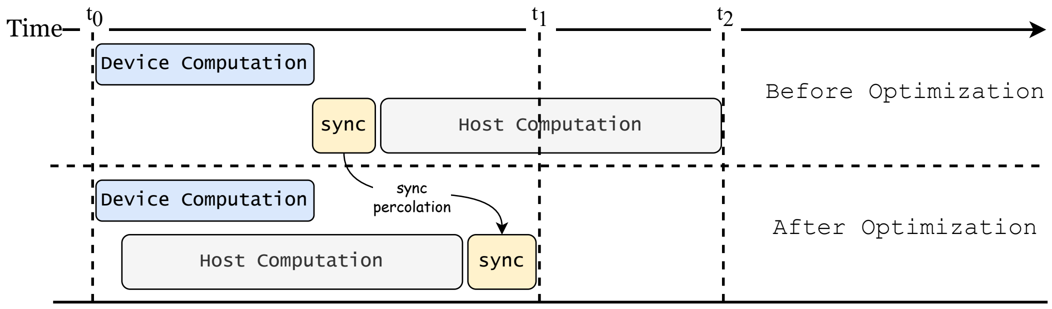


Figure 2 Execution of host- and device-side computations before and after hetero-sync motion.

Line 13. Alternatively, if the barrier is relocated after the host-side computation (say, from Line 11 to Line 14), the CPU will not wait for the kernel to complete, resulting in parallel execution of host- and device-side computations. Figure 2 (below the horizontal dotted lines) indicates that the time will then reduce from $t_2 - t_0$ to $t_1 - t_0$. This single synchronization statement relocation has opened a new avenue for optimizing program execution time. We call this optimization *hetero-sync motion*.

Manually identifying the target location for hetero-sync motion on a given synchronization statement is challenging. Firstly, for the synchronization statement at Line 11, it is required that the relocation target cannot be a successor of Line 15, which uses the variable `d_u`. Further, alias relationships between variables should be taken into account when performing this verification. Secondly, for the synchronization statement at Line 17, simply checking for data dependency may result in incorrectly moving it to Line 5, which is not guaranteed to get executed in all program paths, unlike the source location (this may also result in redundant or spurious data transfers). Thirdly, for the synchronization statement at Line 19, although it has no argument variables, to find its target location, we need to consider the farthest successor in the execution path for performance without violating previous synchronization dependencies. Fourthly, a kernel call (Line 16) can be associated with several synchronization calls (Line 17 and Line 19), and the challenge is to relocate all these synchronizations to the most suitable target location. The code snippet in Figure 3a from the `xlqc` benchmark program, which computes Coulomb and exchange matrices for quantum chemistry simulations, exemplifies such challenges in real-world applications. Here, `memcpy` operations at Lines 8 and 13, though dependent on Line 26, cannot be directly moved because doing so would make them conditional on the for loop at Line 19 resulting in correctness issues or multiple executions that degrade performance. Therefore, we need a dedicated analysis that addresses these challenges precisely and efficiently.

Furthermore, real-world programs exhibit the complexity of finding optimal target locations for synchronization statements across functions. Consider the code snippet in Figure 3b, from `CLINK` [11] (Compact LSTM INference Kernel), which performs efficient sequence prediction by minimizing memory and computation overhead in Long Short-Term Memory (LSTM) models. The synchronous `memcpy` call at Line 8 (along with `cudaDeviceSynchronize` call at Line 6) can be moved outside the `lstm_n5` function and placed after Line 19, thereby overlapping the kernel computation (at Line 5) with `init` (at line 18). Identifying such optimization opportunities requires a thorough analysis of call sites, precise mapping of function arguments, and context- and flow-sensitive analysis, posing significant challenges even for experienced developers. Moreover, the subsequent transformation requires adjusting function definitions, callsite arguments, and relocating instructions across functions, as seen

```
 1 void main(){
 2  ...
 3  while(...){
 4   if (use_dp)
 5    cuda_mat_J_PI_dp<<<...>>>(...);
 6   else
 7    cuda_mat_J_PI<<<...>>>(...);
 8   cudaMemcpy(h_mat_J_PI, dev_mat_J_PI,
        size_N_BYTES, cudaMemcpyDeviceToHost);
 9   if (use_dp)
10    cuda_mat_K_PI_dp<<<...>>>(...);
11   else
12    cuda_mat_K_PI<<<...>>>(...);
13   cudaMemcpy(h_mat_K_PI, dev_mat_K_PI,
        size_N_BYTES, cudaMemcpyDeviceToHost);
14    for (int a = 0; a < p_basis->num; ++a) {
15      for (int b = 0; b <= a; ++b) {
16         ...
17      }
18    }
19    for (int i = 0; i < n_pbf; ++i)
20    { ...
21      for (int j = 0; j < n_pbf; ++j)
22      {
23        int b = h_pbf_to_cbf[j];
24        if (a < b) { continue; }
25        ...
26        ▶h_mat_J[ab] += h_mat_J_PI[ij];
27        h_mat_K[ab] += h_mat_K_PI[ij];
28      }
29    } ...
30   } ...
31 }
```

(a) Code snippet from `xlqc` [58].

```
 1 long lstm_n5(..., float* y)
 2 { ...
 3  float *d_y;
 4  ...
 5  lstm_inference<<<...>>>(...,
        d_outB, d_y);
 6  cudaDeviceSynchronize();
 7  ...
 8  cudaMemcpy(y, d_y, ...,
        cudaMemcpyDeviceToHost);
 9  ...
10 }
11
12 int main(int argc, char* argv[])
13 { ...
14  for (int n = 0; n < repeat; n++)
15  { ...
16    lstm_n5(sample_input, inW,
          intW, intB, outW, &outB,
          infer1_out);
17    ...
18    init(work_path, input_filename
          , weight2_filename,
          sample_input, in, intW,
          intB, outW, &outB);
19    lstm_n5(sample_input, inW,
          intW, intB, outW, &outB,
          infer2_out);
20    ▶...
21    free(infer1_out);
22    free(infer2_out);
23   ...}
24  ...}
```

(b) Code snippet from `clink` [11].

Figure 3 Code snippet demonstrating opportunities in hetero-sync motion. The yellow box highlights synchronization statements, while the blue triangle (▶) marks the destination location.

in this example, where the `memcpy`'s arguments at Line 3 must be moved to the `main` (caller) function and passed as arguments to the `lstm_n5` (source) function. Further, the `memcpy` statement in `lstm_n5` function is invoked from two contexts (Lines 16 and 19), leading to the percolation of synchronization statements from both contexts to the target location. To address these complexities and reduce programming overhead, we propose an automated framework, GSOHC, to identify and move synchronization statements to optimal target locations, streamlining the code transformation process.

Existing Approaches and the Need for a New Solution. Hetero-Sync Motion optimization is analogous to compiler optimizations [42, 19, 16, 41, 17, 43] that relocate inefficient synchronization in the context of alternative programming paradigms such as MPI (Message Passing Interface) and CPU TLP (Thread-Level Parallelism). However, these approaches are inadequate to address the problem of hetero-sync motion due to the semantic differences in synchronization operations: (1) APIs such as `cudaMemcpy` in CPU-GPU heterogeneous programs perform both synchronization and data-transfer operations, unlike `wait` calls in TLP or MPI paradigms that only handle synchronization, (2) by default, synchronization statements in CUDA execute in the default stream, and data transfer begins only after all previous CUDA calls are completed, whereas MPI `wait` calls wait for the completion of a specific communication call based on the corresponding `request_ID` (similar to TLP). Using TLP approaches [42, 19, 43] for hetero-sync motion may percolate *sync* calls into divergent branch paths (e.g., moving Line 17 to Line 5 and Line 25 in Figure 1) resulting in redundant execution of data transfers that may adversely affect the correctness and performance of a CPU-GPU program. On the other hand, MPI approaches [16, 41, 17] require a one-to-one pairing of `wait` calls with non-blocking communication calls, and handling each pair separately through data dependence checks; whereas, in CPU-GPU heterogeneous systems, the challenge lies in consolidating multiple synchronization calls surrounding a single kernel call into one target location. Further, previous works [23, 27, 4, 26, 30] have focused on CPU-GPU global optimization problems through techniques such as multi-kernel computation overlap and data transfer–kernel computation overlap using hybrid runtime and compile-time approaches. However, they do not address the synchronization relocation challenges posed by hetero-sync motion. Therefore, new static data-flow analysis approaches are needed to address synchronization bottlenecks in CPU-GPU heterogeneous programs by analyzing the flow of multiple synchronization calls and their associated data.

GSOHC. In Figure 2, below the horizontal dotted line, we can see how GSOHC pushes the synchronization statement after the host-side computation, facilitating overlap between host-side and device-side computations, thereby enhancing overall program performance. GSOHC does not optimize the host-side nor device-side computations, but it improves performance by only pushing the barrier to a successor program point in the execution path. Our static analysis is a context-sensitive, flow-sensitive inter-procedural data-flow analysis with three phases to identify the optimization opportunities precisely: (1) identifies the poorly placed synchronization statements in the code and maps them to target location sets (not unique since dependencies can be found in multiple program points in different execution paths), (2) updates the mapped location sets based on the control flow and contextual information, and (3) unifies the location sets so that each synchronization statement in a given context is mapped to a unique location.

We implemented our framework using LLVM/Clang infrastructure [36, 31]. GSOHC takes the input program in LLVM/IR format and produces the optimized IR with overlapping host-side computation and device-side computation, resulting in performance improvement. The key contributions of this work are as follows:

- A novel idea of hetero-sync motion optimization that relocates synchronization statements from a source location to a target location to improve performance in CPU-GPU heterogeneous programs.
- A novel compiler framework with three stages, including an interprocedural, context- and flow-sensitive data-flow analysis pass to identify optimization opportunities and determine optimal target locations for synchronization statements.
- A transformation pass that automatically relocates barrier statements to an optimal location so that host-side and device-side computations can overlap. Implementation of both passes in the LLVM infrastructure.
- An extensive evaluation of our approach and comparison with the base code. On A4000, P100, and A100 GPUs, our optimization achieves up to 1.8x, 1.9x, and 1.9x speedups over baseline, respectively.

2 Formalization and Overview

In this section, we discuss some preliminaries that are required to explain our static analysis in Section 3, and then we formally define our hetero-sync motion optimization.

We use the programming language in Figure 4 for our formalization. While we use this language to simplify our presentation, our implementation (see Section 5 for details) works on LLVM IR. It is easy to correlate every analysis step discussed in this paper with LLVM

Program $\mathcal{P}$	=	$s;\ m^{+}$
Method m	=	$\mathcal{H}(\vec{v})\{s_h\} \mid \mathcal{D}(\vec{v})\{s_d\}$
Host statement s_h	=	$\textbf{call } \mathcal{H}(\vec{e}) \mid sync \mid s \mid s_d \mid \textbf{return } v$
Device statement s_d	=	$\textbf{call } \mathcal{D}(\vec{e}) \mid s$
Synchronization call $sync$	=	$\textbf{call } global\_sync\_api(\vec{e})$
Expression e	=	$v \mid c \mid e_1 \oplus e_2 \mid \neg e \mid e_1 \vee e_2 \mid e_1 \wedge e_2 \mid e_1 \otimes e_2$
Statement s	=	$v := e \mid \textbf{if}(e)\ s \textbf{ else } s \mid \textbf{while}(e) \textbf{ do } s \mid s; s \mid \textbf{skip}$

$v \in$ **Variable** $\quad c \in$ **Constant** $\quad \vec{v} = (v_1, v_2, \ldots v_n)$

$\oplus \in \{+, -, \times, \div\} \quad \otimes \in \{<, >, \leq, \geq, =, \neq\} \quad \vec{e} = (e_1, e_2, \ldots e_n)$

Figure 4 Syntax of the input program.

IR instructions. Our implementation supports full LLVM IR, including pointer operations, though we do not show them in Figure 4. Furthermore, we consider heterogeneous computing systems with a CPU and a single GPU setup. Although our ideas are generic, we use the CUDA [8] programming model whenever we discuss concrete semantics.

The program $\mathcal{P}$ is a heterogeneous program consisting of statements and methods that can run on a CPU or a GPU. $\mathcal{D}(\vec{v})$ represents a method whose body of statements gets executed on a GPU (device) [1]. $\mathcal{H}(\vec{v})$ represents a method executed on a CPU (we do not specify the return type for simplicity). The statement *sync* denotes function calls that serve as global barriers to synchronize CPU and GPU. In CUDA programming model, $global\_sync\_api(\vec{e})$ can represent `cudaMemcpy` or `cudaDeviceSynchronize` calls. In addition, variable accesses v, expressions e, assignment statements ($v := e$), branch and loop instructions, etc., are all standard programming constructs. We now formally define some terminology that is required to introduce hetero-sync motion optimization.

▶ **Definition 1** (Device-side Computation). *We say an operation d is* device-side computation *if it resides within the body of a method $\mathcal{D}$ executed on a device (using multiple threads in SIMT fashion).*

▶ **Definition 2** (Host-side Computation). *We say an operation h is* host-side computation *if it resides within the body of a method $\mathcal{H}$ executed on a host (using one or more threads in SIMD or MIMD or SISD fashion).*

▶ **Definition 3** (Synchronization Call, *sync*). *In a heterogeneous program $\mathcal{P}$, a program point or statement s_i within a host-side method $\mathcal{H}$ is called a* synchronization call *(or sync) if it invokes dedicated APIs that cause $\mathcal{H}$ to wait at s_i until all device-side methods $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n$ launched prior to s_i complete execution.*

For example, `cudaMemcpy` call (Line 17, Figure 1) and `cudaDeviceSynchronize` call (Line 19, Figure 1) are synchronization calls.

The goal of GSOHC is to identify large regions of code with host-side computations that are independent of a *sync* and result in overlap with device-side computations, maximizing performance through concurrent execution.

▶ **Definition 4** (Data Dependence). *Given two program statements s_i and s_j in a heterogeneous program $\mathcal{P}$, we say s_j is* data dependent *on s_i or vice versa iff*

$$(rd(s_i) \cap wr(s_j)) \cup (wr(s_i) \cap wr(s_j)) \cup (wr(s_i) \cap rd(s_j)) \neq \phi$$

where $rd(s_i)$ gives the set of variables read by statement s_i and $wr(s_i)$ gives the set of variables written by statement s_i.

[1] We use “GPU” and “device” interchangeably and, similarly, “host” and “CPU” in this paper.

For example, in Figure 1, Line 17 is data dependent on Line 5, since Line 5 reads and writes to the same data array that is written by Line 17.

▶ **Definition 5** (Statement Region, Ω). *In a program $\mathcal{P}$ with Inter-procedural Control Flow Graph $ICFG(\mathcal{P})$, a* statement region *$\Omega(p_i, p_j)$ is the sequence of program statements between two program points p_i and p_j (excluding p_i and p_j), where the basic block of p_j is a successor of the basic block of p_i in $ICFG(\mathcal{P})$.*

▶ **Definition 6** (Safe Region, $\mathcal{R}$). *A statement region $\Omega(p_i, p_j)$ is a* safe region, *$\mathcal{R}(p_i, p_j)$, if no statement in $\Omega(p_i, p_j)$ is data-dependent on the statement at p_i.*

▶ **Definition 7** (Refined Safe Region, $\mathcal{K}$). *A* refined safe region, *$\mathcal{K}(p_i, p_j)$ is derived from a safe region $\mathcal{R}(p_i, p_k)$ (represented as $\mathcal{R}(p_i, p_k) \vdash \mathcal{K}(p_i, p_j)$) iff:*

$$(p_i \textit{ dominates } p_j) \wedge (p_j \textit{ post-dominates } p_i) \wedge (p_j \textit{ dominates } p_k)$$

While $\mathcal{R}$ considers only data dependency, $\mathcal{K}$ considers both data dependency (from $\mathcal{R}$) and control dependency (via dominance relations). A statement p_i *dominates* p_j if all paths from the entry node to p_j pass through p_i, and p_j *post-dominates* p_i if all paths from p_i to the exit node pass through p_j.

▶ **Definition 8** (Unified Safe Region, $\mathcal{U}$). *A* unified safe region, *$\mathcal{U}(p_i, p_j)$ corresponds to a unique refined safe region, $\mathcal{K}(p_i, p_m)$, from the set of refined safe regions for p_i $\{\mathcal{K}_1(p_i, p_1), \mathcal{K}_2(p_i, p_2), \mathcal{K}_3(p_i, p_3), \ldots, \mathcal{K}_n(p_i, p_n)\}$ iff:*

$$\exists m \forall s,\ m, s \in \{1, \ldots, n\} \mid (p_m \textit{ dominates } p_s)$$

While the refined safe region $\mathcal{K}$ addresses control and data dependencies, multiple regions for p_i may result in redundant executions of a *sync* call along a path. The unified region $\mathcal{U}$ uniquely updates and unifies these regions for p_i.

▶ **Definition 9** (Hetero-Sync Motion). *Given a program point p_{src} with a sync call in a host-side method $\mathcal{H}$, if $\mathcal{U}(p_{src}, p_{dst})$ is a unified safe region, relocating the sync from p_{src} to p_{dst} to enable concurrent execution of a host-side statement $s_h \in \mathcal{U}(p_{src}, p_{dst})$ and a device-side computation statement s_d is called as* Hetero-Sync Motion.

Since *sync* calls block host-side and device-side computations from overlapping, our optimization *Hetero-Sync Motion* aims to move the *sync* call from program point p_{src} to p_{dst} to enable concurrent execution and improve performance.

▶ **Example.** In Figure 1, $\Omega(\text{Line 17}, \text{Line 5})$ is an example of a statement region, which is also a safe region i.e., $\mathcal{R}(\text{Line 17}, \text{Line 5})$. Since Line 5 does not post-dominate Line 17, it cannot form a refined safe region. Hence, Line 17 and Line 3 form a refined safe region, i.e., $\mathcal{K}(\text{Line 17}, \text{Line 3})$. Similarly, another refined safe region is $\mathcal{K}(\text{Line 17}, \text{Line 23})$. The unified safe region becomes $\mathcal{U}(\text{Line 17}, \text{Line 3})$, indicating that hetero-sync motion optimization percolates the *sync* from source Line 17 to target Line 3.

Overview. Figure 5 depicts the pipeline of our GSOHC. Given a program $\mathcal{P}$ as input, GSOHC performs hetero-sync motion optimization (see Definition 9) on $\mathcal{P}$ and produces an optimized program $\mathcal{P}_{opt}$ automatically. GSOHC consists of two major components: a top-down inter-procedural context- and flow-sensitive data-flow analysis pass to identify the opportunities for hetero-sync motion optimization and a transformation pass to optimize the input code using the analysis results. Further, our static analysis framework follows

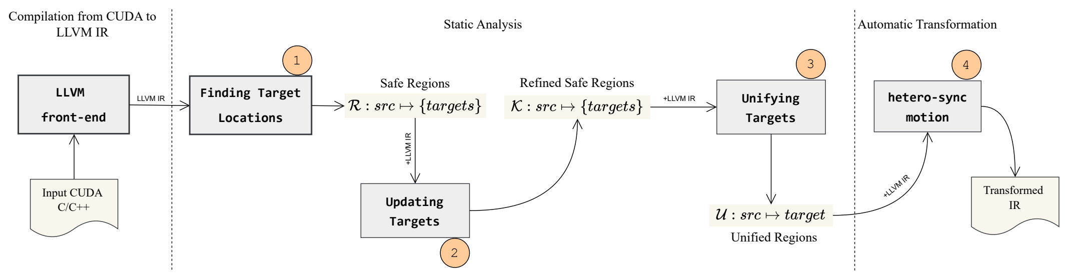


Figure 5 Complete pipeline of GSOHC.

a three-stage approach. The first stage identifies target location sets for poorly placed synchronization statements, represented as safe regions (see Definition 6). The second stage updates these safe regions to maintain the control-flow dependency between the source and target locations to form refined safe regions (see Definition 7). The third stage unifies each target location set into a single, unified safe region (see Definition 8), ensuring each *sync* statement is mapped to a unique location. Finally, the transformation pass relocates *sync* statements from source to target locations, to enhance the performance of the input programs. We will discuss the different stages of the analysis pass in Section 3 and then discuss the transformation pass in Section 4.

3 Static Detection of Hetero-Sync Motion Opportunities

Our static analysis for identifying opportunities of hetero-sync motion optimization takes a program $\mathcal{P}$ as input and outputs a map $\mathcal{M}$, which maps a *sync* statement to a set of target statement(s) in $\mathcal{P}$. The *sync* statement representing a global barrier call (see Definition 3) in $\mathcal{P}$ is considered as a source location, and the corresponding target location (singleton set at the end) is identified by our static analysis based on the concept of unified safe region (Definition 8). We will discuss the three stages of our static analysis in the following sections: finding target location sets (Section 3.1), updating target location sets (Section 3.2), and unifying target location sets (Section 3.3).

3.1 Finding Target Location Sets

The key idea is to perform a top-down context- and flow-sensitive inter-procedural data-flow analysis to identify the hetero-sync motion optimization opportunities in the form of safe regions (see Definition 6). Specifically, our analysis reports for each *sync* statement at p_{src} [2] a set of target locations, which is a collection of p_{dst} program points corresponding to safe regions identified in multiple execution paths.

Figure 6 shows the transfer function rules of our static analysis for statements in the language described in Figure 4. The first rule shows that if a program statement s is a synchronization call (see Definition 3), the analysis collects the variables that are read and written by s into sets $\mathbf{R}$ and $\mathbf{W}$, respectively, and then inserts s into the set Σ. In this paper, we refer to the sets $\mathbf{R}$ and $\mathbf{W}$ as the *read* set and the *write* set, respectively, representing the variables that are read from and written to by the *sync* statements. Likewise, we refer to Σ

[2] We use the words "source statement" and *sync* interchangeably, and, similarly, the words target and destination in this paper.

[SYNCHRONIZATION CALL]

$$\frac{s : \textbf{call}\ \textit{global\_sync\_api}(\vec{e}) \quad e_i \in \vec{e}}{\mathbf{R}^s_{out} := \mathbf{R}^s_{in} \cup \mathtt{rd}(e_i) \quad \mathbf{W}^s_{out} := \mathbf{W}^s_{in} \cup \mathtt{wr}(e_i) \quad \Sigma^s_{out} := \Sigma^s_{in} \cup \{s\} \quad \mu^s_{out}[s] := \{s\}}$$

[HOST FUNCTION CALL]

$$\frac{s : \textbf{call}\ \mathcal{H}(\vec{e}) \quad s_1 : \mathtt{entryPoint}(\mathcal{H})}{\mathbf{R}^{s_1}_{in} := \mathbf{R}^s_{in} \quad \mathbf{W}^{s_1}_{in} := \mathbf{W}^s_{in} \quad \Sigma^{s_1}_{in} := \Sigma^s_{in} \quad \mu^{s_1}_{in} := \mu^s_{in} \quad \Gamma := \Gamma \cdot s}$$

[ASSIGN]

$$\frac{s : v := e \quad var \in \mathtt{rd}(e) \quad w \in \mathbf{W}^s_{in} \quad r \in \mathbf{R}^s_{in} \quad (var \odot w \ \vee \ v \odot w \ \vee \ v \odot r)}{\mathbf{R}^s_{out} := \emptyset \quad \mathbf{W}^s_{out} := \emptyset \quad \Sigma^s_{out} := \emptyset \quad \forall src \in \Sigma^s_{in}\ \mu^s_{out}[src] := \{s\}}$$

[DEVICE FUNCTION CALL]

$$\frac{\begin{array}{c} s : \textbf{call}\ \mathcal{D}(\vec{e}) \quad e_i \in \vec{e} \quad v_r \in \mathtt{rd}(e_i)\ v_w \in \mathtt{wr}(e_i) \quad w \in \mathbf{W}^s_{in} \quad r \in \mathbf{R}^s_{in} \\ (v_r \odot w \ \vee \ v_w \odot w \ \vee \ v_w \odot r) \end{array}}{\mathbf{R}^s_{out} := \emptyset \quad \mathbf{W}^s_{out} := \emptyset \quad \Sigma^s_{out} := \emptyset \quad \forall src \in \Sigma^s_{in}\ \mu^s_{out}[src] := \{s\}}$$

Figure 6 Transfer function rules for our flow- and context-sensitive static analysis. Here, $\mathtt{rd}(e)$ and $\mathtt{wr}(e)$ denote the sets of variables read and written by the expression e, respectively. Further, the target map μ stores each *sync* statement with its corresponding set of target locations.

as the *synchronization* set, which stores the *sync* statements. In the rule, $\mathbf{R}^s$ represent the flow-sensitive instance of the data-flow fact $\mathbf{R}$ at program point s, while sets $\mathbf{R}^s_{in}$ and $\mathbf{R}^s_{out}$ represent the data-flow facts flowing to and from statement s (similarly for $\mathbf{W}$, Σ and μ). Note that for the simplicity of the design of our static analysis, we have assumed that each basic block in an ICFG contains a single program statement.

As the analysis aims to identify an optimization opportunity in the form of safe region $\mathcal{R}(p_{src}, p_{dst})$, the first rule involves identifying the start of a safe region at the program point s [3] (i.e., s represents p_{src} here). This rule also initializes the target locations (end of safe regions) of s by adding itself to the set $\mu^s[s]$, which other rules can update later. We refer to μ as *target* map, which stores the mapping of a *sync* statement to the set of target locations where it can be percolated.

▶ **Example.** In Figure 7, `cudaMemcpy` call at Line 19 reads from `arr_gpu` and writes to `arr_output_gpu`. Hence, the read set and the write set are {`arr_gpu`} and {`arr_output_gpu`}, respectively. Further, the synchronization set includes Line 19, and the target map μ maps Line 19 to itself, i.e. $\mu[\text{Line } 19] = \{\text{Line } 19\}$.

The second rule handles host function calls. As the analysis is inter-procedural, it analyzes the callee host function at each call site. It first propagates the data-flow facts from the call site to the entry point of the callee and appends the call site to the *call-string* Γ, which stores the function calling context (i.e., caller-callee sequence as a string) and then analyzes the callee. In our analysis, we restrict the length of the call-string to 4.

[3] Whenever we refer to a program statement, we refer to it in a given calling context Γ, i.e. $s \equiv (\Gamma, s)$.

```
1  void foo(...,arr_output_gpu,...)
2  {
3      /*entry point of function foo*/
4      int *x = malloc(n * sizeof(int));
5      ...
6      if(pred){
7          for(i=0;i<n;i++){
8              x[i] = arr_output_gpu[i];
9          }
10     }
11     ...
12     return;
13 }
14 void main(){
15     ...
16     /* Device-side computation */
17     cuda_kernel<<< grid,block >>>(..., arr_gpu);
18     /* barrier */
19     cudaMemcpy(arr_output_gpu, arr_gpu, size, cudaMemcpyDeviceToHost);
20     ...
21     foo(...,arr_output_gpu,...);
22     ...
23 }
```

Figure 7 Sample code with hetero-sync motion optimization opportunity.

▶ **Example.** In Figure 7, the function `main` invokes function `foo` at Line 21. Hence, the data-flow facts at the call site are propagated to the entry point of function `foo` (Line 4). The corresponding read set, write set, synchronization set, and target map for Line 4 are: $\mathbf{R}^s_{in} = \{$`arr_gpu`$\}$, $\mathbf{W}^s_{in} = \{$`arr_output_gpu`$\}$, $\Sigma^s_{in} = \{\text{Line 19}\}$, and $\mu^s_{in} = (\text{Line 19}, \{\text{Line 19}\})$. The analysis also appends the call site (Line 21) to the call-string Γ, i.e. $\Gamma := \Gamma\ .\ \text{Line 21}$.

The next rule addresses the assignment instruction $v := e$ involving RAW (Read-After-Write), WAR (Write-After-Read), and WAW (Writer-After-Write) dependencies. The analysis checks for any variable var read in expression e (i.e., $var \in \texttt{rd}(e)$) that aliases with variable w in the write set ($\mathbf{W}^s_{in}$) of the propagated *sync* statements at the program point s, thereby detecting a RAW dependency, represented as $var \odot w$. Here, $x \odot y$ indicates that x and y may refer to the same memory location (i.e. potential aliases). Further, for the variable v written in the assignment statement s, the analysis checks for WAR and WAW dependencies by examining the alias relationship between v and the variables in the set $\mathbf{R}^s_{in} \cup \mathbf{W}^s_{in}$. If a dependency exists, it creates a mapping from the propagated *sync* statements in μ^s to s, marking s as the target. This indicates the end of a safe region $\mathcal{R}$ for a *sync* statement and, thus, an optimization opportunity for hetero-sync motion optimization has been found. Note that the analysis terminates the safe regions for all the *sync* calls that reach s because after the code transformation, even if a single *sync* relocates to s, it becomes a blocking call for the host, and hence it would not be beneficial to extend the safe region of other propagated synchronization calls. Therefore, the analysis reinitializes the data structures $\mathbf{R}^s$, $\mathbf{W}^s$, and Σ^s to identify new optimization opportunities in the unanalyzed code.

▶ **Example.** In Figure 7, the assignment statement at Line 8 reads from array `arr_output_gpu`, which was previously written at Line 19. Due to this RAW data dependence, the analysis updates the target set for Line 19 in map μ to $\{\text{Line 8}\}$, i.e., $\mu[\text{Line 19}] := \{\text{Line 8}\}$, followed by the re-initialization of other data-flow facts ($\mathbf{R}$, $\mathbf{W}$, and Σ).

The fourth rule addresses device function calls. Since *sync* statements cannot be percolated into device functions, there is no need to analyze the code in these functions. This rule checks the data dependency between the function arguments passed at the calling context and the data-flow facts propagated to that program point. The effect of this rule is similar to that of an assignment statement (discussed above).

For all other program statements, the analysis directly copies the data-flow facts for propagation, i.e., $\mathbf{R}^s_{out} := \mathbf{R}^s_{in}$ (Similarly for $\mathbf{W}$, Σ, and μ). Upon completing the analysis of a function f, the framework consolidates the data-flow facts from each **return** statement s_r of f and propagates them to each successor *succ* of the corresponding call site in the caller function (found in the call-string Γ). That is, $\mathbf{R}^{succ}_{in} := \mathbf{R}^{succ}_{in} \cup \mathbf{R}^{s_r}_{out}$ (Similarly for $\mathbf{W}$, Σ, and μ). The call site is then removed from the call-string Γ.

▶ **Example.** The data flow facts corresponding to the `return` statement (Line 12, Fig. 7) in function `foo` are copied to the successor (Line 22) of the call site (of function `foo`).

Our analysis handles branches and loops by performing join operations on data-flow facts at control-flow merge points. The confluence operator of the analysis is a union, meaning the data-flow facts at the entry of a basic block (*in* sets) n in a CFG are the union of the *out* sets of all predecessors of n.

3.2 Updating Target Location Sets

After the analysis in the previous step terminates, GSOHC retrieves the target location set for each *sync* statement (from map μ^s at the main function's exit statement s) and provides this information in map $\mathcal{M}$ as input to the current step. The target location sets computed in the previous step respect data dependencies but do not account for control-flow dependencies. For example, consider the *sync* call at Line 19 in Figure 7. The analysis in the previous step identifies the data dependency between the write to `arr_output_gpu` in `cudaMemcpy` call at Line 19 and the read of the same variable at Line 8 (in function `foo`, called at Line 21). Hence, it terminates the safe region at Line 8 and maps this as a target location for the *sync* call at Line 19. However, relocating `cudaMemcpy` synchronization call from Line 19 to Line 8 would make its execution conditional on the value of `pred`, unlike in the original program where it executes on all paths. Thus, this step validates the target mappings $\mathcal{M}$ reported by the previous step and updates invalid target locations with valid ones by checking control-flow dependency through Algorithm 1.

For each *sync* statement *src* (source) and its corresponding target location *dst* (destination), Algorithm 1 validates if *src* dominates *dst* and *dst* post-dominates *src*, establishing a *dom-pdom* relationship. If validation fails, the algorithm iteratively identifies the farthest successor s_t from *src* that satisfies the dom-pdom relationship, updating s_t as the target location and thus establishing a refined safe region (see Definition 7) from *src* to s_t. We propose a context-sensitive, demand-driven algorithm for determining inter-procedural dom-pdom relationships between source and destination instructions. Our approach utilizes the transitivity property of dominance (if s_a dominates s_b and s_b dominates s_c, then s_a dominates s_c) and extends LLVM's intra-procedural dom-pdom tree analysis [34] to handle inter-procedural relationships. Further, unlike previous work [18] that exhaustively computes dominance relationships for all node pairs in the Inter-procedural Control Flow Graph (ICFG), our targeted analysis efficiently focuses only on the specific dom-pdom relationships required for our framework.

To present our algorithm, we extend the source and destination call-strings, Γ_{src} and Γ_{dst}, by appending the source instruction and destination instruction, respectively. We refer to this augmented data structure as *contextual path* $\mathbb{K}$. For example, the source and destination contextual paths are $\mathbb{K}_{src} = \langle sc_1,\ sc_2,\ \ldots,\ sc_n\rangle$, and $\mathbb{K}_{dst} = \langle dt_1,\ dt_2,\ \ldots,\ dt_m\rangle$, where each sc_i or dt_i is a function call, and sc_n and dt_m are the *src* and *dst* instructions, respectively.

Algorithm 1 Updation of Target Locations.

```
   Input: source $src$, target $dst$, call-strings $\Gamma_{src}$, $\Gamma_{dst}$
   Output: updated target statement $s_t$
1  Procedure updateTarget
2      $\mathbb{K}_{src},\ \mathbb{K}_{dst} \leftarrow \Gamma_{src}.src,\ \Gamma_{dst}.dst$
3      $n,\ m \leftarrow |\mathbb{K}_{src}|,\ |\mathbb{K}_{dst}|$
4      $com_i = \max_{1\le i\le \min(n,m)}\{\texttt{enclosingFn}(sc_i) = \texttt{enclosingFn}(dt_i)\}$
5      $s_{extd} \leftarrow \texttt{exitDominance}(\mathbb{K}_{src},\ com_i,\ n)$
6      if $s_{extd} \neq \mathbb{K}_{src}[com_i]$ then
7          $s_r \leftarrow \texttt{exitNode}(\texttt{enclosingFn}(s_{extd}))$
8          $s_t \leftarrow \texttt{validateRelocation}(s_{extd},\ s_r)$
9          return $s_t$
10     if $\mathbb{K}_{src}[com_i]\ dom\ \mathbb{K}_{dst}[com_i]\ \wedge \mathbb{K}_{dst}[com_i]\ pdom\ \mathbb{K}_{src}[com_i] = False$ then
11         $s^s_{com} \leftarrow \mathbb{K}_{src}[com_i]$
12         $s^d_{com} \leftarrow \mathbb{K}_{dst}[com_i]$
13         $s_t \leftarrow \texttt{validateRelocation}(s^s_{com},\ s^d_{com})$
14         return $s_t$
15     $s_{entd} \leftarrow \texttt{entryDominance}(\mathbb{K}_{dst},\ com_i,\ m)$
16     $s_e \leftarrow \texttt{entryNode}(\texttt{enclosingFn}(s_{entd}))$
17     if $s_{entd}\ pdom\ s_e = False$ then
18         $s_t \leftarrow \texttt{validateRelocation}(s_e,\ s_{entd})$ return $s_t$
19     return $dst$
```

To determine inter-procedural dom-pdom relationships, we require a reference point within the contextual paths of both the source and destination instructions. We call this reference point the Nearest Common Caller (NCC) function, which is the function where the backward paths from *src* and *dst* in their respective contextual paths first meet. We refer to the index in a contextual path corresponding to the NCC function as the NCC index. The NCC index com_i is the maximum index where the functions in the contextual paths $\mathbb{K}_{src}$ and $\mathbb{K}_{dst}$ are same (formally defined at Line 4, Algorithm 1). Using the NCC function as a reference point, the algorithm effectively models the source and destination as if they were inlined within this common context. For instance, in Figure 8a, function `f₁` is the nearest common function to both *src* and *dst*, thereby establishing it as the NCC function. Similarly, in Figures 8b, 8c, and 8d, functions `foo`, `foo`, and `main` are the respective NCC functions.

Algorithm 1 transitively establishes dom-pdom relationships across the entry and exit points of the functions involved in the paths from *src* to *dst* that pass via the NCC function, ensuring that *src* instruction can be safely propagated across function boundaries. This task can be divided into three phases, each addressing dominance at critical points in the paths: (1) Exit Dominance (Lines 5 - 9), (2) Common Dominance (Lines 10 - 14), and (3) Entry Dominance (Lines 15 - 19). In the ICFGs shown in Figure 8, the paths for Exit Dominance, Common Dominance, and Entry Dominance are highlighted in green, red, and blue, respectively. Notice that, together, these paths form a path from *src* to *dst* through the NCC function.

Exit Dominance. First, we verify if *src* ($\mathbb{K}_{src}[n]$) dominates the exit node of its enclosing function f_{src}, ensuring that the refined safe region for *src* can be extended to the call site invoking f_{src} in the caller function. The `exitDominance` function (Line 1, Algorithm 2)

Algorithm 2 `exitDominance` and `entryDominance`.

```
Function exitDominance($\mathbb{K}_s$, $index$, $n$)
    for $i \leftarrow n$ to $index + 1$ by $1$ do
        $s_r \leftarrow$ exitNode(enclosingFn($\mathbb{K}_s[i]$))
        if $\mathbb{K}_s[i]$ $dom$ $s_r = False$ then
            return $\mathbb{K}_s[i]$
    return $\mathbb{K}_s[index]$
Function entryDominance($\mathbb{K}_d$, $index$, $m$)
    for $i \leftarrow index + 1$ to $m$ by $1$ do
        $s_e \leftarrow$ entryNode(enclosingFn($\mathbb{K}_d[i]$))
        if $\mathbb{K}_d[i]$ $pdom$ $s_e = False$ then
            return $\mathbb{K}_d[i]$
    return $\mathbb{K}_d[n]$
```

iterates backwards from the last statement in $\mathbb{K}_{src}$, checking if each statement in the contextual path dominates the exit node of its enclosing function. If all these dominances hold, `exitDominance` returns the instruction at the NCC index in $\mathbb{K}_{src}$, establishing a refined safe region for *src* up to $\mathbb{K}_{src}[com_i]$. For example, in Figure 8c, the source function `bar` is invoked by `fun`, which is further called by the destination function `foo`, making `foo` the NCC function. The `exitDominance` function validates the dominance relationship between the source node `S` and the exit node `T` of `bar`, and then between node `Q` and node `U`. If all dominances hold, `exitDominance` returns the instruction at the NCC index (node `O`), indicating that the refined safe region can be established from node `S` to node `O`. (Similarly, in Figures 8a and 8b if nodes `D` and `L` dominate nodes `E` and node `M`, respectively, safe regions are established up to NCC indices `B` and `J`.) If a statement at index i fails to dominate the exit node of its enclosing function f, `validateRelocation` (Algorithm 3) finds a valid target location within f by iterating from $\mathbb{K}_{src}[i]$ to the farthest successor that satisfies the dom-pdom relationship. In Figure 8c, if node `Q` does not dominate node `U`, `validateRelocation` finds a valid target location along the path from `Q` to `U`. Note that if *src* is within the NCC function, i.e., $\mathbb{K}_{src}[com_i] = src$ as shown in Figure 8d, the `exitDominance` function has no effect, as it directly returns $\mathbb{K}_{src}[com_i]$, the instruction at the NCC index, without iterating through the loop. The algorithm proceeds to the Common Dominance phase only when a refined safe region for *src* is established up to the instruction at the NCC index ($\mathbb{K}_{src}[com_i]$).

Common Dominance. This phase verifies the dom-pdom relationship between the instructions $\mathbb{K}_{src}[com_i]$ and $\mathbb{K}_{dst}[com_i]$ at the NCC index to confirm whether the refined safe region for *src* can extend from $\mathbb{K}_{src}[com_i]$ to $\mathbb{K}_{dst}[com_i]$. If validation fails, the algorithm invokes the `validateRelocation` function to find a valid target location on the path from $\mathbb{K}_{src}[com_i]$ to $\mathbb{K}_{dst}[com_i]$ within the NCC function. If successful, the algorithm proceeds to the Entry Dominance phase. For example, in Figure 8a, if nodes `B` and `F` satisfy the dom-pdom relationship, the refined safe region for *src* can be extended into function `f`$_3$. Similarly, in Figure 8d, if nodes `W` and `X` satisfy the dom-pdom relationship, the refined safe region extends into `foo` through the Entry Dominance phase.

Algorithm 3 Validation of Target Location Set.

```
1  Function validateRelocation(stmt $s_n$, stmt $s_i$)
2      $worklist \leftarrow \{s_n\}$; $s_c \leftarrow s_i$
3      repeat
4          $b \leftarrow$ first($worklist$)
5          $worklist \leftarrow worklist - \{b\}$
6          if $b = s_i$ then
7              break
8          if $s_n$ dom $b \wedge b$ pdom $s_n$ then
9              $s_c \leftarrow b$
10         foreach $succ \in$ successor($b$) do
11             if $succ$ is not visited then
12                 $worklist \leftarrow worklist \cup \{succ\}$
13     until $worklist = \emptyset$;
14     return $s_c$
```

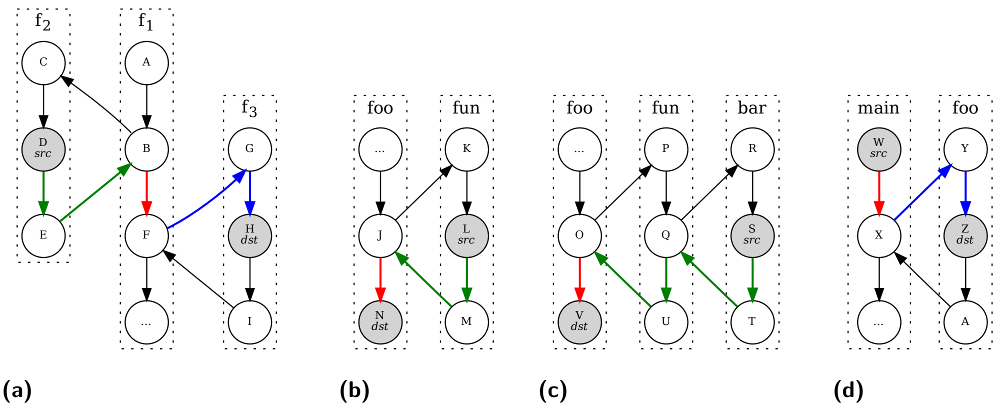


Figure 8 Examples for Exit Dominance, Common Dominance and Entry Dominance phase.

Entry Dominance. This phase verifies if an instruction s_d inside a callee function f_d (of the NCC function) post-dominates the entry node of f_d such that the refined safe region for *src* can be extended up to s_d. The `entryDominance` function (see Line 7, Algorithm 2) iterates through the destination contextual path $\mathbb{K}_{dst}$ to validate if each statement in the contextual path post-dominates the entry node of its enclosing function. If all these post-dominances hold, the algorithm establishes a refined safe region from *src* up to *dst*. For example, in Figure 8a, the refined safe region is extended till node `H` if `H` post-dominates the entry node `G`. Similarly, in Figure 8d, the refined safe region for *src* is established till node `Z` if `Z` post-dominates `Y`. If any statement at index i fails to post-dominate the entry node N of its enclosing function f, `validateRelocation` finds a valid target location within f by iterating from N to the farthest successor in f that satisfies the dom-pdom relationship. Note that, if *dst* resides within the NCC function, i.e., $\mathbb{K}_{dst}[com_i] = dst$, as shown in Figure 8b, `entryDominance` has no effect, as it simply returns $\mathbb{K}_{dst}[com_i]$, without iterating through the destination contextual path. Finally, `entryDominance` and Algorithm 1 terminate by returning the instruction up to which the refined safe region is established for the given *src*.

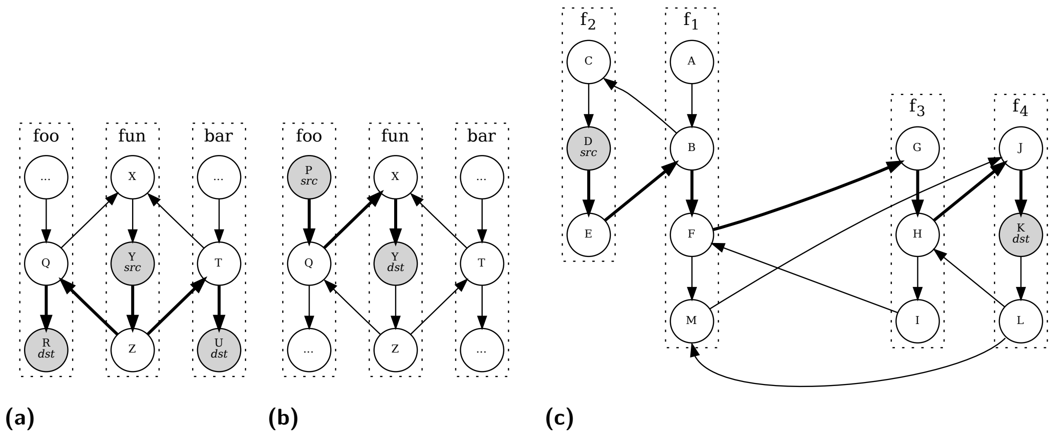


Figure 9 Locations of *src* and *dst* in inter-procedural hetero-sync motion, with paths from *src* to *dst* highlighted in bold.

▶ **Example.** In Figure 7, source *src* is Line 19 and target *dst* is Line 8, and the contextual paths are $\mathbb{K}_{src} = \langle \text{Line } 19 \rangle$ and $\mathbb{K}_{dst} = \langle \text{Line } 21, \text{ Line } 8 \rangle$. Since, `enclosingFn`$(\mathbb{K}_{src}[1]) =$ `enclosingFn`$(\mathbb{K}_{dst}[1]) =$ `main`, the function `main` becomes the NCC function. Here, the Exit Dominance phase has no effect as *src* belongs to the NCC function (Line 19). In the Common Dominance phase, Line 19 and Line 21 satisfy the dom-pdom relationship, confirming that refined safe region can be extended into `foo`. In the Entry Dominance phase, *dst* (Line 8), however, does not post-dominate the entry node of `foo`. Hence, `validateRelocation` identifies Line 5 as the farthest successor from the entry node, which satisfies the dom-pdom relationship, establishing the refined safe region as $\mathcal{K}(\text{Line } 19, \text{ Line } 5)$ and setting Line 5 as the new destination.

Updating Target Locations Across Contexts. After identifying a *dst* for an *src* in a specific context by means of a refined safe region, we verify its safety across different contexts. For a *src* in a method m, the *dst* location can fall into one of the three cases (see Figure 9): (1) *dst* inside a caller of m (2) *dst* inside a callee of m, and (3) *dst* inside a function f where m and f are called by NCC function.

The first case is safe because each invocation of the source function in different contexts maps *src* to a specific destination(s), such that the effect of percolation remains valid across all contexts when percolated to the respective caller functions. The example in Figure 9a demonstrates this with source function (`fun`) being invoked from two contexts (node `Q` in `foo` and node `T` in `bar`). The source instruction at context `Q` has a destination at node `R`, while at context `T`, the destination is `U`. Thus, percolating *src* to the caller functions ensures its execution along both paths: `Q` $\to$ `X` $\to$ `Y` $\to$ `Z` $\to$ `Q` $\to$ `R`, and `T` $\to$ `X` $\to$ `Y` $\to$ `Z` $\to$ `T` $\to$ `U` (as in the original program before optimization).

In the second case, relocating *src* to a callee can be unsafe in some cases. For example, in Figure 9b, `fun` is called from two different contexts in functions `foo` and `bar`. Percolating *src* (the *sync* statement) from `P` (in `foo`) to `Y` (in `fun`) enables computation in node `X` to run in parallel with kernels launched above `P` along the path `P` $\to$ `Q` $\to$ `X` $\to$ `Y` $\to$ `Z` $\to$ `Q` but introduces unintended memory transfers along path `T` $\to$ `X` $\to$ `Y` $\to$ `Z` $\to$ `T`, causing correctness issues. Hence, we percolate *sync* from *src* to the farthest node, whose enclosing function is invoked from a single callsite. In Figure 9b node `Q` is the farthest location from *src* (node `P`), with the enclosing function (`foo`) invoked from a single call site.

Algorithm 4 Updating Targets based on Context.

```
Input: $\mathbb{K}_{dst}$, $com_i$
Output: $s_t$
1 Procedure updateTargets
2     $n \leftarrow |\mathbb{K}_{dst}|$
3     foreach $j \leftarrow com_i$ to $n-1$ by $1$ do
4         $s_{call} \leftarrow \mathbb{K}_{dst}[j]$
5         $f \leftarrow$ getCallee($s_{call}$)
6         $s_t \leftarrow s_{call}$
7         if numCallSites($f$) $> 1$ then
8             break
```

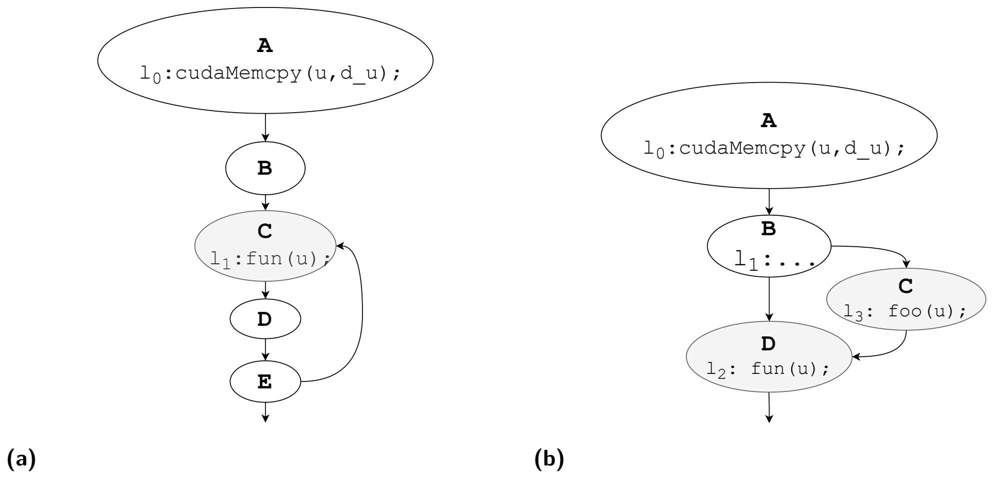


Figure 10 Example of two CFGs showing the necessity of loop-based updation and unification.

In the third scenario, where *src* and *dst* reside in functions that lack a direct caller-callee relationship, direct percolation from *src* to *dst* can be unsafe in some cases. For instance, in Figure 9c where function $\texttt{f}_1$ invokes functions $\texttt{f}_2$, $\texttt{f}_3$, and $\texttt{f}_4$ at nodes `B`, `F`, and `M`, respectively, and $\texttt{f}_3$ further invokes $\texttt{f}_4$ at node `H`. The *sync* at `D` in $\texttt{f}_2$ cannot be directly percolated to `K` in $\texttt{f}_4$ due to invocations of $\texttt{f}_4$ from different callsites. Like in the previous case, we percolate *sync* to the farthest node from *src* (node `D`), which is `H` in Figure 9c, where the enclosing function ($\texttt{f}_3$) is called from a single callsite.

Algorithm 4 updates target statements using contextual information by iterating over the destination contextual path from the NCC index com_i and checking whether the callee function at each index is invoked from a single callsite. If a callee at a callsite s_{call} is invoked from multiple callsites, the algorithm updates the target to s_{call} (Line 6), preventing the percolation of *sync* statements to such callee functions. For example, in Figure 9c, with NCC function $\texttt{f}_1$ and destination contextual path $\langle$`F`, `H`, `K`$\rangle$, the algorithm updates the destination to node `H` as the function $\texttt{f}_3$, called at callsite `F`, is invoked from a single callsite, whereas function $\texttt{f}_4$, called at `H`, is invoked from two distinct callsites (nodes `H` and `M`).

Updating Target Locations based on Loops. We now discuss another kind of updating process with an example. Suppose, in Figure 10a, the analysis identifies l_1 in basic block `C` as the optimal target location for the synchronization statement `cudaMemcpy` at statement l_0

```
void foo(float *arr_output_gpu)
{
    float *arr_gpu, int size;
    ...
    cuda_kernel<<< grid,block >>>(..., arr_gpu); /* Device-side computation */
    cudaMemcpy(arr_output_gpu, arr_gpu, size, cudaMemcpyDeviceToHost); /* barrier */
}
void bar(float *arr_output_gpu)
{
    ...
    for(int i=0; i<n; i++)
        sum+=arr_output_gpu[i];
}
void main(){

    float *arr_out_gpu, *cpu_arr;
    foo(arr_out_gpu);
    cpu_func(cpu_arr);
    bar(arr_out_gpu);
    ...
}
```

Figure 11 Sample code showing the challenges in automated transformation.

in basic block A (due to data dependency on u). While A *dominates* C, and C *post-dominates* A, relocating *sync* from l_0 to l_1 may incur performance overhead as C is a loop-header. A loop-header dominates its loop and is the target of a back edge forming the loop. The statement l_0, which gets executed once when it resides in A, would get executed in multiple iterations after optimization if placed in C. Thus, we mark l_0 to move to B since it is the farthest successor from the source (A) with the same loop scope as A.

At the end of this phase, the *sync* instructions and their updated target locations are stored in the map $\mathcal{M}$. This map is then passed to the next phase for further processing.

3.3 Unifying Target Location Sets

After the previous two steps, $\mathcal{M}$ may still map a single *src* (*sync* statement) to multiple destination locations. For example, in Figure 10b, the *sync* statement l_0 in basic block A has dependent statements l_3 in C and l_2 in D, both using variable u. While the first step identifies a set of target locations $\{l_2, l_3\}$, the second step refines it to $\{l_2, l_1\}$, replacing l_3 with l_1 in basic block B because l_1 post-dominates l_0, unlike l_3. However, since both l_1 and l_2 are on the same execution path, relocating *sync* to both l_1 and l_2 can cause redundant memory transfers, degrading performance. We address this issue in the current phase of our framework by unifying all the target locations that are found and updated by the previous two steps into a single target location. Here, we again use the inter-procedural dominance relationship to solve the problem. Among all the target location statements given by the previous step, GSOHC automatically selects the statement that dominates all the other statements in them. In Figure 10b, since basic block B dominates basic block D, the final set of target locations will contain only statement l_1. That is, GSOHC outputs a unique target location for a given *sync* statement in the form of a unified safe region (see Definition 8) for performing hetero-sync motion.

4 Code Transformation for Hetero-Sync Motion

The transformation pass takes the map $\mathcal{M}$ and a source program in IR format as inputs and outputs an optimized IR (see Figure 5). The map $\mathcal{M}$ associates a unique target location for a *sync* statement in a given context. When the target location is within the same function as *sync*, the *sync* can be moved directly before the target statement. However, if the source and target locations are in different functions, direct percolation of the *sync* is infeasible due to the unavailability of argument variables in the target scope. For example, in Figure 11, the *sync* statement at Line 6 in function `foo` blocks potential overlap between kernel computation (Line 5) and host-side function (Line 18). According to our analysis step, the correct location for *sync* to achieve maximum computation overlap is Line 10. Since arguments for the *sync* call are defined locally in `foo` (Line 3), relocation requires transforming the IR to preserve these definitions across functions.

To address percolations across functions, GSOHC transforms the LLVM IR based on three different cases. Firstly, where the source and target functions lack a direct callee-caller relationship, *sync* argument variable definitions are first relocated to the NCC function (see Section 3), added to formal and actual argument lists of the source and target functions and then, the *sync* statement is percolated to the target instruction. In Figure 11, variables `arr_gpu` and `size`, defined at Line 3, are shifted from `foo` to the NCC function `main`, appended to parameter lists, appended to the actual arguments at call sites (Lines 17 and 19) and the *sync* is subsequently moved to the target location. Secondly, if the target is in the caller of the source function, the caller (target function) becomes the NCC function. Therefore, unlike the first case, the *sync* arguments do not need to be added to the formal and actual argument lists of the target function before percolating the *sync*. Thirdly, if the *sync* is in the caller of the target function, the caller (source function) becomes the NCC function. Similar to the second case, here, the argument variables of *sync* do not need to be added to the source function's formal and actual argument lists. Figure 7 demonstrates an example where hetero-sync motion requires transformation detailed in the third case, where the argument variables of *sync* (`arr_gpu` and `size`) at Line 19 are added as formal and actual arguments of the target function `foo`, prior to percolating the *sync* to Line 5.

5 Evaluation

This section investigates the following research questions to evaluate our hetero-sync motion optimization and GSOHC's capability in identifying and performing this optimization.

RQ1. Can GSOHC precisely identify opportunities for hetero-sync motion?
RQ2. How efficient are GSOHC's analysis and transformation passes?
RQ3. What is the impact of hetero-sync motion on the execution time of input benchmarks?

Implementation. GSOHC consists of two primary components: (1) inter-procedural flow- and context-sensitive static analysis to identify hetero-sync motion opportunities, (2) automated code transformation to perform the optimization. We have implemented our analysis and transformation passes using LLVM/Clang-14.x [36, 31] infrastructure. An open-source GPGPU compiler, `gpucc` [54] embedded with LLVM/Clang-14.x, converts CUDA programs into LLVM IR. Our custom analysis pass works on the SSA (Static Single Assignment) representation obtained using `-mem2reg` flag (which promotes memory references into register references, resulting in a "pruned" SSA). In addition, we use LLVM's intra-procedural Dominator and Post Dominator Tree [34], and Loop Info wrapper [35] passes.

Experimental Setup. We run our experiments on NVIDIA RTX A4000 (Ampere), NVIDIA Tesla P100 (Pascal) and NVIDIA A100 (Ampere) GPUs. Table 1 details the system specifications. This multi-generational GPU selection validates the robustness of our framework, enables analysis of performance variations due to communication and computation differences, and provides insights into its scalability across architectures.

Table 1 System configuration.

GPU	NVIDIA RTX A4000 (16 GB)	NVIDIA P100 (16 GB)	NVIDIA A100 (84 GB)
CPU	64 × AMD EPYC CPU @ 2.20GHz	20 × Intel(R) Xeon(R) Silver CPU @ 2.20GHz	64 × Intel(R) Xeon(R) Gold CPU @ 3.50 GHz
OS	Ubuntu 22.04 LTS	Debian GNU/Linux 11	Ubuntu 22.04 LTS
CPU RAM	128 GB	189 GB	84 GB

Benchmarks. To rigorously evaluate GSOHC with respect to diverse real-world scenarios, we use benchmark programs from benchmark suites: PolyBench [21], Rodinia [10], CUDA Sample SDK [44], and HeCBench [25]. Although (many) host-side computations in these benchmarks verify the results produced by GPU processors, they also exhibit complex control flows, encompassing nested loops, branches and aliasing behaviours, and offer rich inter-procedural opportunities replicating real-world challenges. For example, in the `entropy` program [57], a host-side function computes entropy for each matrix element using a 5×5 window with three nested loops, performing local histogram computation and probability evaluation using a logarithmic operation. Section 6 provides an in-depth analysis of CPU computations and various optimization opportunities in real-world programs. PolyBench, Rodinia and CUDA Sample SDK are widely-used GPU benchmark suites used in recent research works [2, 1, 26], while HeCBench is an extensive collection of heterogeneous computing benchmarks written in CUDA [9], HIP [5], SYCL [29], and OpenMP [14] across domains like physics, linear algebra, and machine learning (sourced from other standard suites such as Rodinia [10], PolyBench [21], N-body simulation [48], Hetero Mark [52], CUDA Sample SDK [44], etc.).

From these benchmark suites, we identified 56 benchmark programs (listed in Table 2) that contain opportunities for our hetero-sync motion optimization, excluding those without kernel or `global_synchronization` calls or with fewer device-side or host-side computations. Note that our framework can also analyze programs that have no hetero-sync motion opportunities and correctly report their absence, ensuring no false positives.

5.1 RQ1: Hetero-Sync Motion in Practice

We found 361 optimization opportunities from 56 benchmark programs for performing hetero-sync motion.

The results of our static analysis are shown in Table 2. The second column lists the number of lines in the LLVM IR file corresponding to each benchmark program. In the third column, O_f represents the number of manually found optimization opportunities. Our static analysis successfully identified all the optimization opportunities in the benchmarks, resulting in zero false negatives. For example, in the `xlqc` [58] program (from HecBench), the code computes Coulomb and exchange matrices for quantum chemistry simulations by first calculating their primitive forms on the GPU and transferring the results to host memory. On the CPU, two 2D contracted matrices are initialized, and GPU-computed primitive matrices are aggregated into these contracted matrices, which are then stored in GNU Scientific Library (GSL) data structures. GSOHC identifies an opportunity for hetero-sync motion here, where the performance can be enhanced by overlapping GPU computations with CPU initialization of the contracted matrices.

We have observed that many of our benchmark programs (around 48%) have synchronization barriers spanned across different functions. Using an intra-procedural analysis would have caused to miss many opportunities in this scenario. Moreover, optimization opportunities that span across different functions are much harder to identify manually (see Section 1). To address this, we designed an inter-procedural, flow- and context-sensitive static analysis framework that enables practical and effective hetero-sync motion.

Table 2 Benchmark statistics and the results of GSOHC's static analysis.

suite	Benchmark	LOC	O_f	T_{an}	suite	Benchmark	LOC	O_f	T_{an}
PolyBench	correlation	1215	6	24	HeCBench	clink	1147	6	12
	covariance	885	5	20		channelsum	9202	6	57
	2mm	983	11	26		chemv	698	6	7
	3mm	1208	9	34		chi2	1054	5	5
	atax	702	6	20		conv Sep	241	3	5
	bicg	775	8	47		dp	317	6	3
	doitgen	863	5	19		entropy	871	8	9
	gemm	653	5	18		extrema	4635	40	63
	gemver	1127	11	39		floyd-warshall	571	5	8
	gesummv	621	2	10		keogh	637	9	9
	mvt	732	8	27		langevin	666	7	10
	syr2k	668	5	18		lif	695	5	8
	syrk	597	4	14		lombscargle	938	7	6
	gramschmidt	986	5	18		maxpool3D	464	3	6
	lu	654	3	11		mrc	923	11	11
	adi	1630	7	43		mt	971	2	6
	2DConvolution	636	4	14		nw	1033	3	8
	3DConvolution	916	4	16		pool	933	3	9
	fdtd-2d	1015	6	22		rainflow	1028	3	6
	jacobi-1D	612	5	26		rtm8	1300	3	9
	jacobi-2D	763	5	29		s8n	1717	7	7
	adam	1214	7	8		scel	677	5	8
	affine	1419	5	6		shuffle	1144	22	291
	atan2	1993	11	15		snake	1679	2	11
	bilateral	1434	9	14		softmax	469	3	5
	bitonic-sort	604	3	5		stddev	509	3	4
	burger	1523	4	27		tissue	864	2	11
	car	1084	8	8		xlqc	17848	9	253

Further, we thoroughly tested our analysis and transformation phases. For each benchmark program, we recorded the source and corresponding target location sets at each phase of our static analysis and manually verified them. We ran each program five times before and after transformation with identical inputs, confirming identical outputs. We also manually inspected the transformed IRs to ensure correctness of the transformation pass.

5.2 RQ2: Efficiency of GSOHC

This section presents the time taken by our analysis and transformation passes for the benchmark programs. In Table 2, the last column (T_{an}) shows the analysis time in milliseconds. Our analysis time ranges from a few milliseconds to 291 milliseconds (measured on the CPU node associated with the RTX A4000 GPU, see Table 1). Further, our transformation pass takes only a few microseconds, requiring three steps: (1) adding variables to function argument lists and callsite modifications, (2) declaring variables in the common caller, and (3) percolating *sync* to the target location.

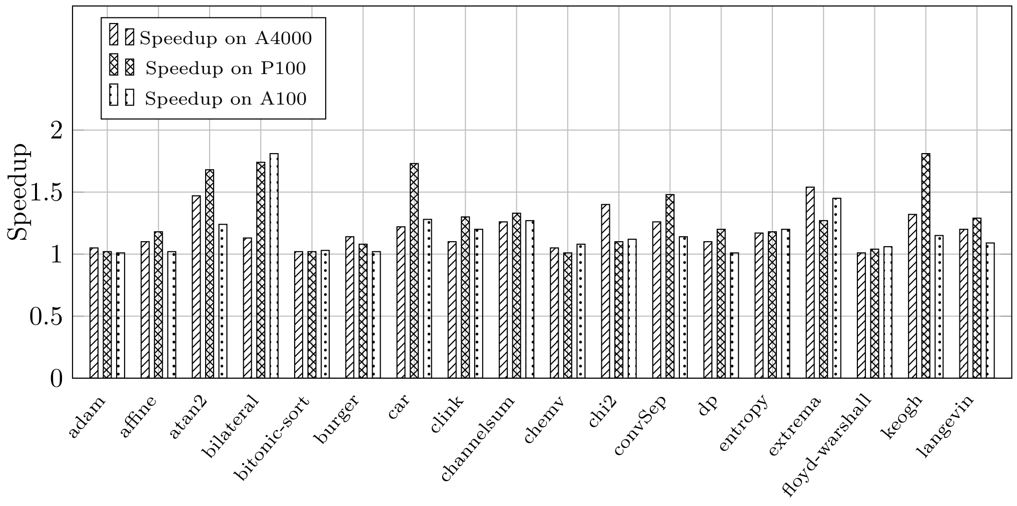
Although the benchmark programs have similar lines of code, the analysis time for the `shuffle` benchmark is significantly higher due to its greater number of hetero-sync motion opportunities and more complex control flow (with more loops, branches, and function calls). On the other hand, though benchmark `extrema` has 40 optimization opportunities, the control flow in it is much simpler (1D loops, fewer function calls and fewer branches with no nesting), resulting in a shorter analysis time. The data flow analysis may require multiple iterations over the loop body, with new flow information in each iteration, taking longer to converge to a fixed point. Therefore, the analysis time is more influenced by the number and location of opportunities rather than program size.

5.3 RQ3: Effectiveness of Hetero-Sync Motion

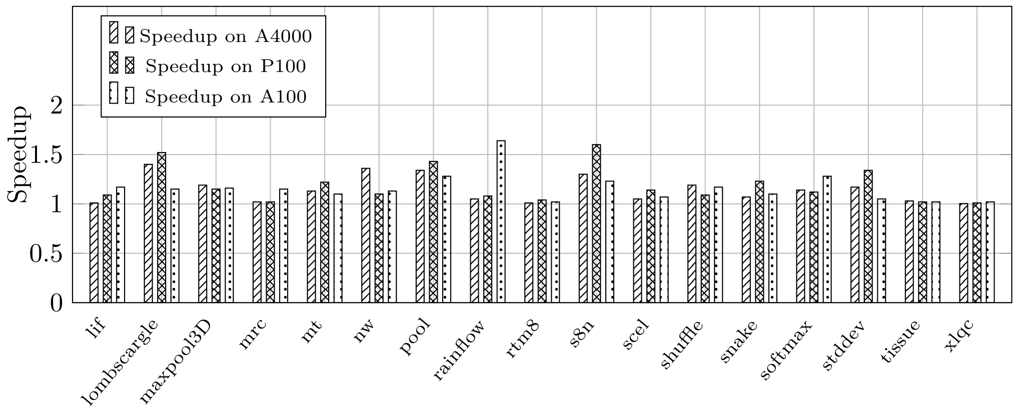
Figure 12a, 12b and 12c show the performance improvements achieved by hetero-sync motion on different benchmarks run on RTX A4000, P100 and A100 machines. We executed each benchmark program five times before and after optimization and then reported their arithmetic mean execution times. Our hetero-sync motion optimization achieves substantial performance gains on all machines, with geomean speedups of approximately 1.15x, 1.17x, and 1.13x with standard deviations of 0.19, 0.20 and 0.15 (and up to 1.8x, 1.9x and 1.9x speedups on A4000, P100 and A100, respectively) and a geometric mean performance improvement of around **11**%, reaching up to **34**%.

In many cases, the reduction in total execution time is also significant. Specifically, 36 benchmarks demonstrated performance improvements exceeding 500 milliseconds on the P100 machine, with 29 of these benchmarks showing improvements of over 1 second (11 benchmarks having improvements over 10 seconds). Given that the total running times of these programs are only a few seconds, hetero-sync motion optimization is notably impactful. The benchmarks with higher execution times also have higher improvements in execution time. As detailed in Figure 12b, the most pronounced improvement was observed in the `snake` benchmark from the HeCbench suite, where the execution time was reduced from 368.5 seconds to 230.6 seconds, yielding a 1.6x speedup and an overall reduction of 137.9 seconds. Further, the transformed code results in the best performance if T_{host}, the execution time of identified host-side computations, and T_{dev}, the execution time of identified device-side computations, are significant and are comparable to one another such that they can hide each other's latency. Thus, benchmarks such as `extrema`, `bilateral`, and `atan2` with significant overlaps show approximately 1.5x to 1.7x speedup with the optimization.

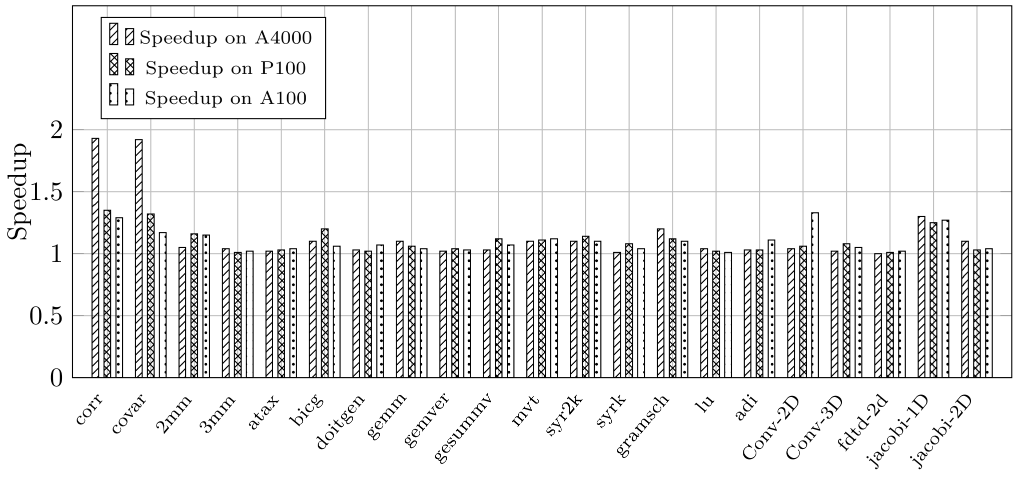
On the other hand, if T_{host} highly dominates T_{dev} or vice versa, the optimized code is constrained by the dominated time and does not result in significant performance gains. Thus the performance gains of less than 1.06x in specific benchmarks can be attributed to an imbalanced load distribution between CPU and GPU computations. For example, in `3mm`, `gemm` (from Polybench), and `bitonic-sort`, `lif` (from HeCBench), the CPU-side computations take significantly longer time than device-side computations. Thus, the optimized code takes almost as long as the CPU-side computation, resulting in smaller speedups. Further, we describe an interesting case in `floyd-warshall` benchmark program where it does not exhibit performance improvement post-optimization. In this case, the synchronization statement is a `cudaMemcpy` call present outside of a loop. However, additional `cudaMemcpy` calls within the loop, occurring before each kernel invocation, cause the kernel to block at each iteration. Since these synchronization calls are not loop-invariants, they cannot be moved outside the loop. Furthermore, relocating the *sync* statement from outside the loop body to a program point beyond the CPU computation may result in minimal overlap between host and device computation (particularly in the final iteration of the loop), yielding negligible speedup.



(a) Speedup achieved by hetero-sync motion on HeCBench.



(b) Speedup achieved by hetero-sync motion on HeCBench (contd.).



(c) Speedup achieved by hetero-sync motion on PolyBench.

Figure 12 Speedup achieved by hetero-sync motion on various benchmark programs.

```
 1 void backward_pass(...){
 2   ...
 3   Matrix_Multiplication_GPU<<<...>>>
          (..., device_first, ...);
 4   cudaMemcpy(first->gradient,
          device_first, ..., DtoH);
 5
 6   /* Compute the gradient for the
          second matrix */
 7   for (int row = 0; row<first->nRows
          ; ++row) {
 8    for (int column = 0; column<first
          ->nColumns; ++column) {
 9     temp[column * first->nRows + row
          ]=first->value[row * first
          ->nColumns + column];}
10   }
11
12   cudaMemcpy(device_first, temp,
          ..., HtoD);
13   cudaMemcpy(device_second, second->
          gradient, ..., HtoD);
14
15   Matrix_Multiplication_GPU<<<...>>>
          (..., device_second, ...);
16
17   cudaMemcpy(..., device_second,
          ..., DtoH);
18  ...
19 }
```

(a) Code snippet from `GraphFlow` [22].

```
 1 void divideWork(...){
 2   if(value < size) {
 3     for(int i = 0; i < value; ++i) {...}}
 4   else {
 5     int portion = value / size;
 6     for(int i = 0; i < size; ++i) {
 7       h_head[i] = i * portion;
 8       h_tail[i] = (i + 1) * portion;}
 9       h_tail[size - 1] = value;}
10   cudaMemcpy(d_head, h_head, ..., HtoD);
11   cudaMemcpy(d_tail, h_tail, ..., HtoD);
12 }
13
14 void main(){...
15  while(!(*h_complete)) {
16   divideWork(h_head, h_tail, d_head, d_tail,
          NBLOCKS, V);
17   assignColorsKernel<<<...>>> (d_head, ...);
18   cudaDeviceSynchronize();
19   *h_complete = true;
20   cudaMemcpy(d_complete, ..., HtoD);
21   divideWork(...);
22   detectConflictsKernel<<<...>>> (...,
          d_complete, ...);
23   cudaDeviceSynchronize();
24   cudaMemcpy(..., d_complete, ..., DtoH);
25   divideWork(...);
26   forbidColorsKernel<<<...>>> (d_head, ...);
27   cudaDeviceSynchronize();}
28  ...}
```

(b) Code snippet from `GCOL` [13].

Figure 13 Hetero-Sync Motion optimization opportunitites in real-world codes. For brevity, `cudaMemcpyDeviceToHost` is abbreviated as `DtoH` and `cudaMemcpyHostToDevice` as `HtoD`.

6 Case Study

Building upon the evaluation in Section 5 that demonstrates GSOHC's effectiveness, this section explores the broader applicability of hetero-sync motion in real-world open-source applications. We have surveyed several top-rated CUDA/C++ projects on GitHub and highlighted relevant opportunities in Figure 13. We present these case studies separately from the broader benchmark evaluation to provide a focused analysis of opportunities identified in specific real-world applications and to highlight practical insights that may otherwise be obscured in larger-scale evaluations. Note that certain real-world opportunities are embedded within large frameworks (rather than stand-alone programs) and benchmark suites that lack the necessary testing environment, such as readily available datasets. As a result, even though our static analysis pass is capable of effectively identifying these opportunities, we could not include them in our empirical evaluation. We first present intra-procedural opportunities, then inter-procedural opportunities.

Intra-procedural Opportunities. `GraphFlow` [22] (see Figure 13a), a CUDA/C++ deep learning framework, facilitates symbolic and automatic differentiation, supports dynamic computation graphs, and provides GPU-accelerated tensor and matrix operations, along with implementations of various state-of-the-art graph neural networks. An opportunity for hetero-sync motion optimization exists here when a code segment (in `GraphFlow`) uses GPU-based matrix multiplication for gradient computation. This code initiates two kernel launches. The first (Line 3) computes the gradient for the first matrix and synchronously transfers (Line 4) the result back to the host, which is followed by a CPU computation (Line 9) transposing the first matrix into a temporary data structure, which is then transferred to the device memory along with the second matrix. The second kernel launch (Line 15) then performs matrix multiplication, updating the gradient for the second matrix, which is subsequently copied back to the host. In this case, performance can be further improved by overlapping the first kernel's computation with the CPU transposition of the first matrix. However, this optimization is currently restricted by the synchronous host-device data transfer immediately after the first kernel launch in the original implementation.

Inter-procedural Opportunities. We now discuss the `GCOL` and `GCON` programs from ScoR [13] benchmark suite. Both are CUDA/C++ applications focused on GPU-accelerated graph colouring and graph connectivity algorithms, respectively. Because these two applications share similar programming patterns, we will only discuss `GCOL` (see Figure 13b) in detail. The program operates in three phases within a loop that continues until all colour assignments are conflict-free. In the first phase, a GPU kernel (Line 17) is launched to assign initial colours to the vertices, while in the second phase, the code launches another GPU kernel (Line 22) to verify whether no adjacent vertices share the same colour. Lastly, a third GPU kernel (Line 26) is invoked to forbid specific colours from being assigned to prevent conflicts. Between consecutive phases, a host-side function is called (at Line 21) to partition the graph data for the next GPU kernel computation. However, `cudaDeviceSynchronize` calls (Lines 18 and 23) and synchronous data transfer operations (Lines 20 and 24) after each GPU kernel prevent overlap between the CPU's data partitioning and GPU computations, limiting potential performance improvements. This points to a hetero-sync motion optimization opportunity, especially since the CPU computation for workload division is encapsulated in a separate host function (distinct from that of the source function having *sync* call), indicating the need for inter-procedural optimization.

We further discuss real-world opportunities found in a GPU-based recursive filtering application called `gpufilter` [3] (not shown). An opportunity for hetero-sync motion optimization exists in a code segment that invokes three different host-side functions to perform different Gaussian filtering on an input image. Each function first computes the coefficients and weight vectors, followed by the creation of transformation matrices, which include generating zero and identity matrices, transposed convolution matrices, and performing matrix-vector multiplications. These transformation matrices are transferred to the device memory, followed by multiple GPU kernel launches to apply the filtering operations. Finally, the filtered image is transferred back to the host memory. The synchronous data transfer from device to host limits the overlap between the GPU filtering process and the CPU computation of transformation matrices for subsequent filtering, indicating an area for hetero-sync motion optimization.

7 Related Work

Global Optimizations in Heterogeneous Computing. GSOHC relocates synchronization calls on the host side to maximize the overlap between host and device computations. In this context, we refer to global optimizations as those approaches that target host-side code but improve interactions between CPU and GPU in heterogeneous systems, leading to overall better performance. CGCM [23] improves program performance by managing CPU-GPU communication through compiler optimizations and runtime libraries, breaking cyclic communication patterns by transferring data to the GPU early and retrieving only when necessary. Kessler et al. [27] proposed a static-analysis technique to reduce the number of data transfer messages between CPU and GPU by analyzing dependence graphs over kernel calls, reordering operand arrays in memory, and merging memory allocations of adjacent operands. Ashcraft et al. [4] developed a compiler technique for automatically scheduling data transfers from CPU to GPU and vice-versa based on data used within the kernel to reduce the number of bytes transferred, leading to performance improvement.

Apart from the above-mentioned static analysis-based approaches, architectural-based or run-time-based approaches also have been proposed to address global performance issues in heterogeneous systems. HUM [26] has provided a run time technique that leverages CUDA Unified Memory to overlap `H2Dmemcpy` time with host or kernel computation, resulting in an overall program speedup. Kim et al. [30] have proposed a software-hardware-based approach to improve the performance of workloads that have multiple dependent GPU kernels by automatically overlapping the execution of such kernels and exploiting implicit pipeline parallelism. As there have been many similar attempts to improve the performance of heterogeneous programs, we cite a survey paper [37] that summarizes the efforts of numerous research papers. Unlike GSOHC, which enhances program performance through hetero-sync motion increasing CPU-GPU compute overlap, these approaches achieve performance through other dimensions such as multi-kernel computation overlap, data transfer overlap, and communication optimizations.

Synchronization Optimization. In the field of compilers, there is a large body of work [41, 19, 39, 42, 16, 15, 17, 50, 49, 43, 56, 24, 40] on reducing synchronization overheads in the context of various programming paradigms, including Thread Level Parallelism (TLP) programs, Task Parallel programs, and MPI (Message Passing Interface) programs.

Hetero-Sync Motion is analogous to the synchronization optimization proposed by Nicolau et al. [42, 43] for efficient placement of `post` and `wait` methods in a `non-DOALL` loop increasing the degree of Thread Level Parallelism (TLP) in CPU programs. Because they focus on the percolation of synchronization calls with only blocking properties, they fail to consider the control-flow dependency of inserted `wait` calls resulting in duplicate `wait` calls (and code explosion). This can lead to correctness and performance issues in heterogeneous CPU-GPU systems because synchronization calls such as `cudaMemcpy` have both blocking and data transfer properties. In contrast to this approach, our proposed framework efficiently manages such scenarios with a dedicated static analysis.

Our work on hetero-sync motion optimization can be compared to works by Danalis et al., Nguyen et al. and Das et al. [16, 41, 17], that proposed code motion techniques to improve computation-communication overlap in the MPI programming model. However, these approaches require matching between communication and blocking calls, can percolate only a single blocking or non-blocking call at a time, and are limited in their ability to extend percolation beyond another `MPI` call. Since a GPU kernel call can have multiple synchronization calls surrounding it, and synchronization calls (by semantics) wait for all previous kernel calls to finish in CPU-GPU heterogeneous systems, GSOHC requires a data-flow analysis to facilitate the percolation of multiple synchronization calls together to an optimal program point. Furthermore, as in the case of TLP, `MPI_wait` calls are mere blocking calls, unlike the ones with data transfer in our case.

A number of past works have also proposed transformation techniques for addressing synchronization overheads and improving program performance. Sarkar et al. [50] provided an approximation algorithm to select an ordering of the fork operations and determine the placement of join operations in a given program, allowing independent computations to run simultaneously and thereby improving the overall parallelism. Zhu et al. [56] provided a code motion compiler optimization technique to reduce the communication overhead caused by remote reads/writes (accessing dynamically allocated data structures) in parallel C programs. Their analysis determines the optimal communication mode (blocking or pipelined) and propagates remote writes in the program that may be subject to coalescing. Diniz et al. [19] aimed at redundant lock elimination to reduce lock overheads through the percolation of synchronization calls in a parallel program. Unfortunately, however, their approach may reduce the overall parallelism of a given program. Nandivada et al. [39] proposed optimizations such as `finish` elimination, loop chunking and `forall`-coarsening to reduce the number of synchronization operations in X10 programs. Nayak et al. [40] introduces ScopeAdvice, a tool designed to detect over-synchronization in CUDA programs, enabling performance optimization by analyzing memory access and synchronization traces using NVIDIA's NVBit library. The cuSync framework [24] provides fine-grained tile-level synchronization to address GPU underutilization caused by uneven tile distribution when the number of computation tiles is not a multiple of the GPU's execution units, enabling concurrent execution of independent tiles with user-defined synchronization policies. While we also address synchronization statements to improve performance, unlike these approaches that directly reduce synchronization overheads, our focus is on percolating synchronization calls to enhance CPU-GPU computation overlap in CUDA programs.

In summary, the existing works on synchronization optimization are inadequate to address the problem of hetero-sync motion due to the semantic differences in synchronization operations. To the best of our knowledge, our work is the first to focus on automatic synchronization motion optimization in CPU-GPU heterogeneous systems.

Static Analyses for Optimizating Device Code in Heterogeneous Computing. Recent efforts [51, 2, 1] focus on accelerating CUDA programs by improving GPU kernels (device-side computations) through static analysis and compiler optimization approaches. DARM [51] presents a static analysis-based framework that identifies and automatically merges divergent blocks in a CFG to reduce the effects of thread divergence in device computation. GPUDrano [1] automatically identifies uncoalesced memory accesses in a GPU kernel code using data flow analysis. GPU Block Independence Analysis [2] represents a static analysis framework that can automatically identify whether a GPU kernel code is independent of the block size, allowing the compiler to tune block size to improve performance. As GSOHC, they are all built on LLVM [31], a general framework for building compilers. These optimizations differ from ours, but they can be effectively combined with GSOHC to improve the performance of the input program further.

Load Balancing of Heterogeneous Workloads. The optimal performance due to hetero-sync motion is attained when the input workload to the source program is load-balanced across various architectures in a heterogeneous system. This can be attributed to our framework's ability to optimize the source code in a manner that allows/improves concurrent execution of host-side and device-side computations. The performance study conducted by White et al. [53] shows that their manually optimized CPU-GPU implementation with workload distributed between CPU and GPU results in better performance. Lee et al. [32] proposed a run time technique that enables the effective utilization of available CPU and GPU computing resources in a heterogenous system through a workload distribution module and a global scheduling queue. Khalid et al. proposed Troodon [28], a machine learning-based framework that automatically schedules jobs to CPU and GPU in a load-balanced way, increasing throughput with high device utilization.

8 Conclusion

We have developed a novel compiler analysis and optimization framework, GSOHC that automatically analyzes CPU-GPU heterogeneous programs and relocates poorly placed synchronization barriers within them. This optimization, which we named hetero-sync motion, facilitates overlapping CPU and GPU computations, thereby improving program performance. GSOHC consisting of three distinct phases employs an interprocedural context- and flow-sensitive data-flow analysis to accurately identify optimization opportunities in the input programs. We have also performed an evaluation of our framework on 56 benchmark programs drawn from several widely-recognized benchmark suites. Our evaluation shows that our static analysis can successfully identify all the actual optimization opportunities within these benchmark programs. Furthermore, our evaluation confirms significant performance improvements in the optimized code, underscoring the practical impact of our approach.

References

1 Rajeev Alur, Joseph Devietti, Omar S. Navarro Leija, and Nimit Singhania. Gpudrano: Detecting uncoalesced accesses in gpu programs. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 507–525, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-63387-9_25.

2 Rajeev Alur, Joseph Devietti, and Nimit Singhania. Block-size independence for gpu programs. In *International Static Analysis Symposium*, pages 107–126. Springer, 2018. doi:10.1007/978-3-319-99725-4_9.

3 Andmax. Github - gpufilter: Gpu recursive filtering. URL: https://github.com/andmax/gpufilter.git.

4 Matthew B Ashcraft, Alexander Lemon, David A Penry, and Quinn Snell. Compiler optimization of accelerator data transfers. *International Journal of Parallel Programming*, 47(1):39–58, 2019. doi:10.1007/S10766-017-0549-3.

5 Michal Babej and Pekka Jääskeläinen. Hipcl: Tool for porting cuda applications to advanced opencl platforms through hip. In *Proceedings of the International Workshop on OpenCL*, IWOCL '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3388333.3388641.

6 David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994. doi:10.1145/197405.197406.

7 Dip Sankar Banerjee and Kishore Kothapalli. Hybrid algorithms for list ranking and graph connected components. In *2011 18th International Conference on High Performance Computing*, pages 1–10, 2011. doi:10.1109/HiPC.2011.6152655.

8 Gautam Chakrabarti, Vinod Grover, Bastiaan Aarts, Xiangyun Kong, Manjunath Kudlur, Yuan Lin, Jaydeep Marathe, Mike Murphy, and Jian-Zhong Wang. Cuda: Compiling and optimizing for a gpu platform. *Procedia Computer Science*, 9:1910–1919, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012. doi:10.1016/j.procs.2012.04.209.

9 Gautam Chakrabarti, Vinod Grover, Bastiaan Aarts, Xiangyun Kong, Manjunath Kudlur, Yuan Lin, Jaydeep Marathe, Mike Murphy, and Jian-Zhong Wang. Cuda: Compiling and optimizing for a gpu platform. *Procedia Computer Science*, 9:1910–1919, 2012. doi:10.1016/J.PROCS.2012.04.209.

10 Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009. doi:10.1109/IISWC.2009.5306797.

11 Zhe Chen, Andrew Howe, Hugh T. Blair, and Jason Cong. Clink: Compact lstm inference kernel for energy efficient neurofeedback devices. In *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED '18, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3218603.3218637`.

12 Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintao Pereira, and Wagner Meira Jr. Divergence analysis and optimizations. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 320–329, 2011. `doi:10.1109/PACT.2011.63`.

13 Csl-Iisc. Github - csl-iisc/scor: A (sco)ped (r)acey benchmark suite, containing cuda code for many different gpu benchmarks. URL: `https://github.com/csl-iisc/ScoR.git`.

14 L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998. `doi:10.1109/99.660313`.

15 A. Danalis, Ki-Yong Kim, L. Pollock, and M. Swany. Transformations to parallel codes for communication-computation overlap. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 58–58, 2005. `doi:10.1109/SC.2005.75`.

16 Anthony Danalis, Lori Pollock, Martin Swany, and John Cavazos. Mpi-aware compiler optimizations for improving communication-computation overlap. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 316–325, New York, NY, USA, 2009. Association for Computing Machinery. `doi:10.1145/1542275.1542321`.

17 Dibyendu Das, Manish Gupta, Rajan Ravindran, W. Shivani, P. Sivakeshava, and Rishabh Uppal. Compiler-controlled extraction of computation-communication overlap in mpi applications. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2008. `doi:10.1109/IPDPS.2008.4536193`.

18 Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. A practical interprocedural dominance algorithm. *ACM Trans. Program. Lang. Syst.*, 29(4):19–es, August 2007. `doi:10.1145/1255450.1255452`.

19 Pedro Diniz and Martin Rinard. Synchronization transformations for parallel computing. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 187–200, New York, NY, USA, 1997. Association for Computing Machinery. `doi:10.1145/263699.263718`.

20 M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966. `doi:10.1109/PROC.1966.5273`.

21 Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10, 2012. `doi:10.1109/InPar.2012.6339595`.

22 HyTruongSon. Github - graphflow: Deep learning framework in c++/cuda that supports symbolic/automatic differentiation, dynamic computation graphs, tensor or matrix operations accelerated by gpu and implementations of various state-of-the-art graph neural networks and other machine learning models including covariant compositional networks for learning graphs [risi et al]. URL: `https://github.com/HyTruongSon/GraphFlow.git`.

23 Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic cpu-gpu communication management and optimization. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 142–151, New York, NY, USA, 2011. Association for Computing Machinery. `doi:10.1145/1993498.1993516`.

24 Abhinav Jangda, Saeed Maleki, Maryam Mehri Dehnavi, Madan Musuvathi, and Olli Saarikivi. A framework for fine-grained synchronization of dependent gpu kernels. In *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '24, pages 93–105. IEEE Press, 2024. `doi:10.1109/CGO57630.2024.10444873`.

25 Zheming Jin. HeCBench. `https://github.com/zjin-lcf/HeCBench.git`, 2022. Retrieved: 2022-11-10.

26 Jaehoon Jung, Daeyoung Park, Youngdong Do, Jungho Park, and Jaejin Lee. Overlapping host-to-device copy and computation using hidden unified memory. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pages 321–335, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3332466.3374531.

27 Christoph Kessler. Global optimization of operand transfer fusion in heterogeneous computing. In *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems*, pages 49–58, 2019. doi:10.1145/3323439.3323981.

28 Yasir Noman Khalid, Muhammad Aleem, Usman Ahmed, Muhammad Arshad Islam, and Muhammad Azhar Iqbal. Troodon: A machine-learning based load-balancing application scheduler for cpu–gpu system. *Journal of Parallel and Distributed Computing*, 132:79–94, 2019. doi:10.1016/J.JPDC.2019.05.015.

29 Khronos, 2020. URL: https://registry.khronos.org/SYCL/specs/.

30 Gwangsun Kim, Jiyun Jeong, John Kim, and Mark Stephenson. Automatically exploiting implicit pipeline parallelism from multiple dependent kernels for gpus. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 341–352, 2016. doi:10.1145/2967938.2967952.

31 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

32 Changmin Lee, Won Woo Ro, and Jean-Luc Gaudiot. Boosting cuda applications with cpu–gpu hybrid computing. *International Journal of Parallel Programming*, 42(2):384–404, 2014. doi:10.1007/S10766-013-0252-Y.

33 Yunsup Lee, Ronny Krashinsky, Vinod Grover, Stephen W. Keckler, and Krste Asanović. Convergence and scalarization for data-parallel architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, 2013. doi:10.1109/CGO.2013.6494995.

34 LLVM. LLVM Dominator Tree Wrapper Pass. URL: https://llvm.org/doxygen/classllvm_1_1DominatorTreeWrapperPass.html.

35 LLVM. LLVM LoopInfoWrapperPass Class Reference. URL: https://llvm.org/doxygen/classllvm_1_1LoopInfoWrapperPass.html.

36 LLVM. LLVM Clang. https://clang.llvm.org/, 2024. Retrieved: 2024-03-19.

37 Sparsh Mittal and Jeffrey S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4), July 2015. doi:10.1145/2788396.

38 Steven Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

39 V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. A transformation framework for optimizing task-parallel programs. *ACM Trans. Program. Lang. Syst.*, 35(1), April 2013. doi:10.1145/2450136.2450138.

40 Ajay Nayak and Arkaprava Basu. Over-synchronization in gpu programs. *MICRO*, 2024.

41 Van Man Nguyen, Emmanuelle Saillard, Julien Jaeger, Denis Barthou, and Patrick Carribault. Automatic code motion to extend mpi nonblocking overlap window. In *High Performance Computing: ISC High Performance 2020 International Workshops, Frankfurt, Germany, June 21–25, 2020, Revised Selected Papers 35*, pages 43–54. Springer, 2020. doi:10.1007/978-3-030-59851-8_4.

42 Alexandru Nicolau, Guangqiang Li, and Arun Kejariwal. Techniques for efficient placement of synchronization primitives. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 199–208, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1504176.1504207.

43 Alexandru Nicolau, Guangqiang Li, Alexander V. Veidenbaum, and Arun Kejariwal. Synchronization optimizations for efficient execution on multi-cores. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 169–180, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1542275.1542303.

44 NVIDIA. CUDA Sample SDK. `https://github.com/NVIDIA/cuda-samples.git`, 2022. Retrieved: 2024-03-19.

45 David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, December 1986. `doi:10.1145/7902.7904`.

46 Anthony Pajot, Loic Barthe, Mathias Paulin, and Pierre Poulin. Combinatorial Bidirectional Path-Tracing for Efficient Hybrid CPU/GPU Rendering. *Computer Graphics Forum*, 30(2):315–324, April 2011. `doi:10.1111/j.1467-8659.2011.01863.x`.

47 Song Jun Park, James Ross, Dale Shires, David Richie, Brian Henz, and Lam Nguyen. Hybrid core acceleration of uwb sire radar signal processing. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):46–57, 2011. `doi:10.1109/TPDS.2010.117`.

48 Daniel P Playne, Mitchell Johnson, and Kenneth A Hawick. Benchmarking gpu devices with n-body simulations. *CDES*, 9:132, 2009.

49 Martin Rinard and Pedro Diniz. Eliminating synchronization bottlenecks in object-based programs using adaptive replication. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 83–92, New York, NY, USA, 1999. Association for Computing Machinery. `doi:10.1145/305138.305167`.

50 V. Sarkar. Instruction reordering for fork-join parallelism. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 322–336, New York, NY, USA, 1990. Association for Computing Machinery. `doi:10.1145/93542.93590`.

51 Charitha Saumya, Kirshanthan Sundararajah, and Milind Kulkarni. Darm: Control-flow melding for simt thread divergence reduction. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '22, pages 28–40. IEEE Press, 2022. `doi:10.1109/CGO53902.2022.9741285`.

52 Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. Hetero-mark, a benchmark suite for cpu-gpu collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

53 J.B. White III and J.J. Dongarra. Overlapping computation and communication for advection on hybrid parallel computers. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 59–67, 2011. `doi:10.1109/IPDPS.2011.16`.

54 Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. Gpucc: An open-source gpgpu compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 105–116, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2854038.2854041`.

55 Ming Xu, Feiguo Chen, Xinhua Liu, Wei Ge, and Jinghai Li. Discrete particle simulation of gas–solid two-phase flows with multi-scale cpu–gpu hybrid computation. *Chemical Engineering Journal*, 207-208:746–757, 2012. 22nd International Symposium on Chemical Reaction Engineering (ISCRE 22). `doi:10.1016/j.cej.2012.07.049`.

56 Yingchun Zhu and Laurie J. Hendren. Communication optimizations for parallel c programs. *SIGPLAN Not.*, 33(5):199–211, May 1998. `doi:10.1145/277652.277723`.

57 Zjin-Lcf. Github – hecbench/src/entropy-cuda. URL: `https://github.com/zjin-lcf/HeCBench/`.

58 zjin lcf. Github – hecbench/src/xlqc-cuda, 2020. URL: `https://github.com/zjin-lcf/HeCBench/`.