学位論文

Compiler Optimization for GPU Based on Code Motion

(コード移動に基づくGPU向けコンパイラ最適化)

2023年3月

福原　淳司

Abstract

As the computing capability of Central Processing Units (CPUs) has been gradually reaching the peak, Graphics Processing Units (GPUs) are utilized for general purpose applications that require high computational power. While CPUs have tens of computing cores at most, GPUs have hundreds to thousands of cores enabling thousands of threads to run in parallel, resulting in its tremendous computing power. Additionally, the programmability of GPUs has been improved rapidly through providing special programming languages, libraries, and tools. Thus, GPUs are playing an important role in various application domains.

The more GPUs are widely utilized, the more sophisticatedly GPU compilers need to optimize GPU programs to execute the programs efficiently. Typical GPU compilers separately compile CPU host and GPU device codes, and a function written in the device code is called a kernel. Optimizations for GPUs are mainly classified into intra-kernel optimizations and inter-kernel ones. Intra-kernel optimizations transform a single kernel into more efficient one while inter-kernel ones improve overall performance of GPU programs by analyzing multiple kernels and the host code.

This thesis proposes both intra- and inter-kernel optimizations inspired by traditional code motion based techniques. First, as intra-kernel optimizations, two methods that remove not only redundant expressions but also branch divergence, which occurs when GPU software threads follow different paths in some conditional branches and decreases the performance of GPU kernels, are proposed. Their approaches are to take advantage of the property where both true and false sides of divergent branches are executed. Considering the property, the proposed methods speculatively hoist expressions in one side of divergent branches out of them without decreasing performance. Thus, the speculative code motion contributes to branch divergence reduction. Second, as an inter-kernel optimization, this thesis proposes a code motion based kernel fusion approach, which exposes the fusibility of kernels by moving them along control flows, so that it increases the opportunities of fusing them. Although traditional methods consider only data dependency between candidate kernels, the proposed method considers not only the data dependency but also a control flow of a program. Therefore, the proposed method can fuse more kernels than traditional methods. The experimental results show that both intra- and inter-kernel optimizations of the proposed methods improve the performance of GPU programs in some benchmarks.

Acknowledgments

I would like to thank my supervisor Professor Munehiro Takimoto. He gave me many significant opinions and insightful comments for this study. He got me interested in compiler optimization, taking me to the exciting path of studying it. Without him, I could not have completed the work mentioned in this thesis.

I would like to thank my thesis committee members, Professor Hiroshi Takemura, Professor Satoru Miyazaki, Professor Joji Maeda, and Professor Kouichi Katsurada. Their valuable feedback significantly improved the thesis. Moreover, I would like to thank Professor Emeritus Tetsuo Tamai of the University of Tokyo and his seminar members. Their useful comments and sharp questions motivated me a lot and made this study much better. I would also like to thank all the people who gave me a lot of guidance in carrying out this study.

Finally and most importantly, I would like to thank my family, especially my parents. They welcome me warmly whenever I return parents' home and support me at all times. I will not be here without unconditional support from them. I would like to express my sincere gratitude to all the people who have supported me.

Contents

Chapter 1

Introduction

High-performance computing has been in great demand for modern applications, such as machine learning, image processing, data mining, and virtual reality. Although the applications are traditionally implemented on Central Processing Units (CPUs), the required computing capability exceeds the CPU performance. Therefore, to meet the high demand of computing capability, *Graphics Processing Units* (GPUs) are utilized for general purpose applications. GPUs, which have thousands of processing cores, give the tremendous computing power, playing a central role in modern high-performance computing.

The programmability of GPUs has been improved rapidly for the last decade. Programmers write GPU programs using programming languages such as *Compute Unified Device Architecture* (CUDA) [76] and OpenCL [5]. The LLVM compiler infrastructure [2, 57] supports GPU programs in each *intermediate representation* (IR) level, Multi-Level IR (MLIR) [4] and LLVM-IR [3]. Most GPU compilers such as CUDA *nvcc* [78] and ROCm HIPCC [6] separately compile a CPU host and GPU device code. In the compilation process, a CPU and GPU code are respectively compiled into IR codes and optimized to be executed efficiently on a target system. The rest of this chapter describes the CUDA programming model, introduces compiler optimizations for GPU programs, and gives the proposed approaches in the thesis.

1.1 CUDA Programming Model

CUDA is the most commonly used GPU programming language, based on the Single Instruction Multiple Thread (SIMT) execution model. The CUDA programmers write the host code processed on CPUs and the device code processed on GPUs separately in a program. The function written for the device code is specially called a *kernel*. In the kernel, there may be performance bottlenecks for GPU programs as described in the followings.

Branch Divergence

In the SIMT execution model, SIMT threads are grouped into *warps*, and threads in a warp must execute the same instruction. In a GPU kernel, if the threads in a warp evaluate the branch condition of a branch instruction, and some threads in a warp evaluate the condition to true while the others evaluate it to false, both true and

false sides of the branch are executed in sequential. This is called *branch divergence*. The threads that evaluate the condition to true execute the true-path while the other threads that evaluate the condition to false wait without doing anything, and vice versa. Consequently, branch divergence takes both costs of the true and false sides of a divergent branch, significantly reducing performance of a kernel.

In the execution environments, some code motion based approaches such as *Partial Redundancy Elimination* (PRE) [70, 54, 55, 16, 48] and *scalar replacement* [12, 13, 18, 98] may increase branch divergence, because the destinations of the code motion may include divergent branches. In this case, they may increase executed instructions compared with original ones, decreasing execution efficiency of a kernel. Therefore, the approaches cannot simply be applied to a kernel with branch divergence.

1.2 Compiler Optimization for GPU Programs

Compiler optimization techniques for GPU programs can be categorized into two main types based on the difference of their optimization scope: *intra-kernel optimizations* and *inter-kernel optimizations*. Intra-kernel optimizations aim to optimize each kernel in a device code while inter-kernel ones analyze several kernels and the host code with callers of them, improving overall performance of the program. Generally, intra-kernel optimizations are orthogonal to inter-kernel ones, so that these two types of optimizations can be applied to GPU programs simultaneously.

1.3 Proposed Approach in the Thesis

This thesis proposes both intra-kernel and inter-kernel optimizations based on code motion techniques. They improve performance of GPU programs by reducing branch divergence and enhancing kernel fusion in terms of respectiveness.

As discussed in Section 1.1, branch divergence is the major source of performance bottlenecks of a GPU kernel. Additionally, although PRE is an effective code optimization technique that not only removes partially redundant expressions but also moves invariant expressions out of loops, it may increase branch divergence and reduce performance of a kernel. The code motion based technique proposed in the thesis as an intra-kernel optimization, which is called *Speculative Sparce Code Motion* (SSCM), extends PRE not only to remove redundant expressions but also to reduce the divergence through speculative code motion. In a divergent branch, where both true and false sides are executed, SSCM speculatively hoists the expression that exists only on one side of a divergent branch before the branch without decreasing performance. The property of speculative code motion contributes to reducing branch divergence. Additionally, speculative code motion makes more expressions redundant [40, 35, 11, 112, 118], so that SSCM eliminates more expressions than traditional methods, resulting in further performance improvement. However, most of GPU kernels have both divergent branches and non-divergent branches, and speculative code motion for non-divergent branches may decrease performance. Therefore, SSCM enables selective application of speculative code motion to improve the performance of kernels with divergent and/or non-divergent branches.

As well as PRE, scalar replacement, which removes inter-iteration redundant memory accesses in a loop, may also increase branch divergence because it inserts memory accesses into some program paths. Therefore, it is difficult to effectively apply scalar replacement to a GPU kernel. Another code motion based technique proposed in the thesis as an intra-kernel optimization, which is called *Speculative Scalar Replacement based on Question Propagation* (SSRQP), makes scalar replacement effective in a kernel by not only removing redundant memory accesses in a loop but also reducing branch divergence through speculative code motion. It is based on *question propagation* [88], which checks whether each expression is redundant while propagating questions on a control flow graph, in terms of redundancy checking. As well as SSCM, SSRQP also takes advantage of the property of a divergent branch to effectively apply speculative code motion without decreasing performance, and hence applies it to only divergent branches. Furthermore, SSRQP checks whether each insertion point of an expression is included in divergent branches and does not insert any expression into the destinations of divergent branches, so that it does not increase branch divergence.

On the other hand, kernel fusion based methods, which is one of inter-kernel optimizations, improve performance by combining several kernels into a single one. Because GPU kernels must store their input and output data in global memory that is the costliest for accessing, many GPU applications have a bottleneck for accessing memories. Fusing these kernels into one and utilizing faster on-chip memory instead of global memory improve performance. Thus, traditional kernel fusion based methods fuse kernels in which one kernel is data-dependent on the other. However, traditional methods miss many fusion opportunities because they focus only on data dependency between consecutive kernels. That is, traditional methods can fuse only consecutive kernels on a control flow graph. This thesis proposes a novel kernel fusion method as an inter-kernel optimization, called *Kernel Fusion based on Code Motion* (KFCM). KFCM exposes the fusibility of kernels based on code motion, which is implemented by dataflow analyses based on the traditional dataflow equations used in PRE [54, 55] and *partial dead code elimination* (PDE) [56]. PRE and PDE are code optimizations that eliminate partially redundant expressions and partially dead statements, respectively. These properties enable KFCM to consider not only flow insensitive fusibility between consecutive kernels but also flow sensitive one between kernels on a control flow of a program. Thus, KFCM can fuse more kernels than traditional methods, reducing more kernel launch overhead and increasing other code optimization opportunities.

Contributions

As mentioned above, this thesis presents two types of a code motion based branch divergence reduction technique and the kernel fusion technique based on code motion. More specifically, their contributions are as follows:

- **Code motion based branch divergence reduction**: SSCM and SSRQP enable PRE and scalar replacement to remove partially redundant expressions and redundant memory accesses in GPU kernels, respectively. Furthermore, by taking advantage of the property of branch divergence, both methods speculatively hoist an expression out of a divergent branch without decreasing performance, contributing to branch divergence reduction. Besides, the methods selectively apply speculative code motion to only divergent branches but not non-divergent

ones because speculative code motion for non-divergent branches may decrease performance. The experimental evaluation shows the effectiveness of SSCM and SSRQP across a number of benchmarks.

- **Kernel fusion based on code motion techniques**: A new kernel fusion based method, KFCM, is introduced by a flow-sensitive approach based on code motion. KFCM is designed based on dataflow analyses utilizing dataflow equations of traditional code motion based optimizations. Thus, KFCM can fuse more kernels based on flow-sensitive fusibility than traditional methods. Experimental results show the effectiveness of KFCM for well-known benchmarks.

1.4 Organization of the Thesis

This thesis consists of the following chapters.

- **Chapter 1: Introduction** gives brief explanations of a GPU programming language with the SIMT execution model and branch divergence as its problem. This chapter also introduces two types of compiler optimizations for GPU programs and describes how the proposed methods improve performance of GPU programs.
- **Chapter 2: Preliminaries** introduces the basic terminologies for compiler optimization techniques used in the rest of the thesis. Moreover, the chapter introduces the NVIDIA GPU architecture and the details of CUDA programming model. Finally, the chapter details types of compiler optimizations for GPU programs.
- **Chapter 3: Code Motion** introduces traditional redundancy elimination techniques based on code motion and shows how the techniques remove a redundant expression in a program.
- **Chapter 4: Code Motion Based Branch Divergence Reduction** discusses how PRE and scalar replacement are extended to reduce branch divergence. This work was published in [29] and [30].
- **Chapter 5: Kernel Fusion Based on Code Motion** discusses how KFCM is realized based on traditional dataflow equations. The experimental evaluation shows how effective KFCM is and how many kernels KFCM can fuse compared with a traditional method. This work was published in [31].
- **Chapter 6: Related Work** presents different branch divergence reduction techniques and kernel fusion based methods. This chapter also discusses other intra- and inter-kernel optimizations for GPU programs.
- **Chapter 7: Conclusion** concludes the thesis by summarizing the proposed methods and shows the future work.

Chapter 2

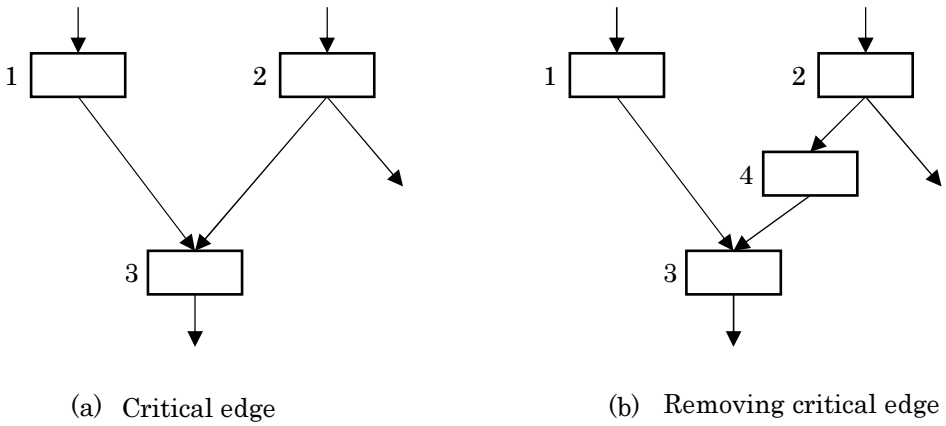
Preliminaries

In this chapter, first, the basic terminologies for a compiler optimization used in the thesis are described. Second, the details of the architecture of NVIDIA GPUs are presented. Third, the terminologies of CUDA and its programming model are described. Finally, types of compiler optimizations for GPU programs are detailed.

2.1 Basic Terminology

This thesis assumes that a *Control Flow Graph* (CFG) has already been generated for each function defined in a program. A CFG is a directed graph $G(N, E, \mathbf{s}, \mathbf{e})$ with a node set N and an edge set $E \subset N \times N$. Each node $n \in N$ represents a *basic block* consisting of continuous statements without any branch in the middle or represents one statement. Each edge $(n, m) \in E$ represents the control flow between nodes n and m. $\mathbf{s}$ and $\mathbf{e}$ denote the unique *start node* and *end node* of G, which are assumed not to have any predecessors and successors, respectively. Every node $n \in N$ is assumed to lie on a path from $\mathbf{s}$ to $\mathbf{e}$. $pred(n) =_{df} \{m \mid (m, n) \in E\}$ and $succ(n) =_{df} \{m \mid (n, m) \in E\}$ denote the set of all predecessors and successors of a node n, respectively. Moreover, *critical edges* [54, 55], which lead from nodes with more than one successor to nodes with more than one predecessor, may block an effective code motion. For example, in Figure 2.1a, the edge leading from node 2 to node 3 is a critical edge. The critical edges are assumed to be eliminated by inserting synthetic nodes on the edges, as illustrated in Figure 2.1b.

It is assumed that a basic block is divided into two partitions, as shown in Figure 2.2. The division point of a basic block is defined as the point immediately after the last *modification statement*, which modifies some operands of a considered expression e. The former partition is called an *entry part*, and the latter is called an *exit part*. If there is no modification statement in a basic block, the entire basic block is defined as an entry part, while an exit part is defined as empty. The first expression e of an entry part is called *entry computation*, and the expression of an exit part is called *exit computation*. When inserting an expression, it is inserted at either an entry part or an exit part. The insertion points at an entry part and an exit part are called an *entry insertion point* and an *exit insertion point*, respectively. Thus, the insertion point is immediately before entry computation or exit computation if there is the computation, or immediately before the first modification statement if there is no entry computation, and there is a modification statement. Otherwise, the insertion point is the end of each



(a) Critical edge (b) Removing critical edge

Figure 2.1: Critical edges and their elimination

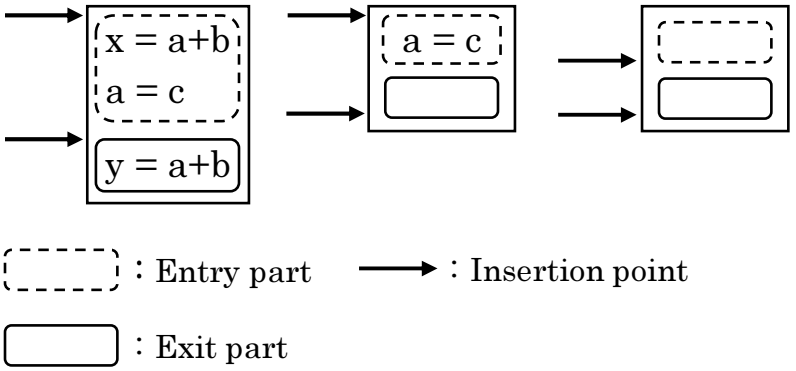


Figure 2.2: How to divide a basic block

part.

A node m *dominates* a node n if and only if every path from **s** to n contains m. A node m *postdominates* a node n if and only if every path from n to **e** contains m. Additionally, a node m is called the *immediate dominator* of a node n if and only if m dominates n, m is not equal to n, and the node dominating n does not exist on every path from m to n except for m. Inversely, a node m is called the *immediate post-dominator* of a node n if and only if m postdominates n, m is not equal to n, and the node postdominating n does not exist on every path from n to m except for m [7]. A node m is *control-dependent* on a node n if and only if there is a non-empty path from n to m such that m postdominates all the nodes except n on the path [71]. Control dependency for an *augmented CFG*, which is the CFG augmented with a special node ENTRY that has one edge going to the start node **s** and another edge going to the end node **e** [26], is assumed to be computed. Notice here that the node ENTRY is non-divergent.

Two memory references have *dependence* if there exists some paths of CFG from the first reference to the second reference and both references access the same memory location [13]. The dependence is called *loop-carried dependence* if two dependent references are in different iterations of a loop. If two dependent references are in the same iteration of a loop, the dependence is called *loop-independent dependence*. The

threshold of a loop-carried dependence is the number of loop iterations between two dependent references. If the threshold is constant throughout the execution of the loop, the threshold is called *consistent threshold*.

An expression e is *available* at node n if each path from the start node **s** to n includes node m that has e and any operands of e are not modified on the path between m and n. Further, e is *partially available* at node n if e is available at some nodes of the predecessors of n. If e is available at node n, n is *up-safe* for e. If an expression e exists at node n and is available immediately before n, e is *totally redundant* at n and can be eliminated by replacing it with the variable that holds the value of e. On the other hand, if e exists at node n and is partially available immediately before n, e is *partially redundant* at n and cannot simply be removed as totally redundant expressions. An expression e is *anticipated* at node n if each path from n to the end node **e** includes node m that has e and any operands of e are not modified on the path between n and m. Moreover, e is *partially anticipated* at node n if e is anticipated at some nodes of the successors of n. If e is anticipated at node n, n is *down-safe* for e.

2.2 GPU Architecture

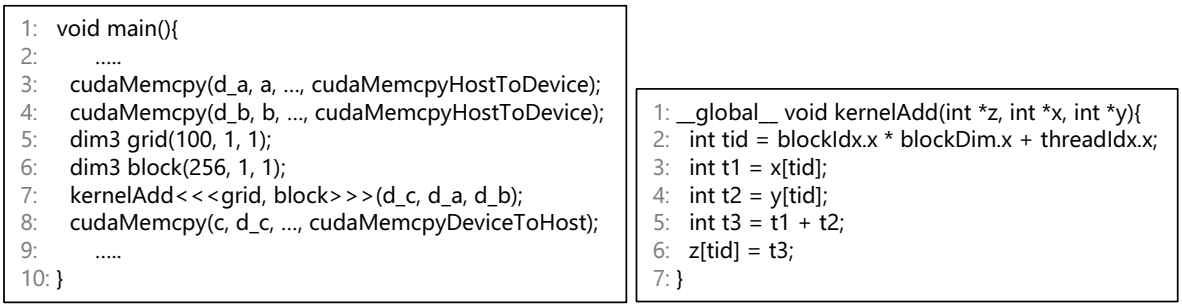
NVIDIA has been introducing GPUs for not only graphics applications but also general purpose computing. GPUs have thousands of processing cores, high arithmetic throughput, and high memory bandwidth. Thus, the tremendous computing power has made GPUs the standard accelerator in many scientific domains.

The NVIDIA A100 GPU is based on the Ampere architecture [75]. A100 GPU consists of 108 *streaming multiprocessors* (SMs), and each SM has 64 FP32 cores, 32 FP64 cores, 64 INT32 cores, and 4 third-generation Tensor cores. Each SM also contains 256 KB of register files and 192 KB of on-chip memory that is configured as L1 cache and shared memory. Besides, A100 GPU has 40 MB of L2 cache and 40 GB of global memory. The memory bandwidth is 1555 GB/sec with a 5120-bit memory interface.

2.3 CUDA

As mentioned in Section 1.1, CUDA [76] is the most used GPU programming language with the SIMT execution model. CUDA provides extensions to C and C++ languages to allow programmers to write the code executed on GPUs. The function executed on GPUs is called a kernel. Programmers launch each kernel in the host code with a hierarchical execution configuration, called a *grid*. A grid consists of three-dimensional *thread blocks* (TBs). TBs also consist of three-dimensional *threads*, which run concurrently on a single SM. Each TB in a grid and each thread in a TB has a unique ID, respectively. The threads in a TB are grouped into warps. Each warp consists of 32 threads with the datapath of Single Instruction Multiple Data (SIMD). The NVIDIA H100 GPU introduces a *thread block cluster* that is a group of TBs, enabling efficient cooperation of threads across multiple SMs [80, 79]. *Occupancy* is defined as the ratio of active warps on a GPU to the maximum number of active warps that can be run on the GPU.

A kernel can access multiple GPU memories during execution, including three kinds of off-chip memories and one kind of on-chip memory. Off-chip memory includes *global*,



(a) Host code processed on CPUs (b) Device code processed on GPUs

Figure 2.3: (a) Host code processed on CPUs; (b) device code processed on GPUs.

constant, and *texture memory* that are shared by all GPU threads and the CPU. On-chip memory includes *shared memory* that is shared among threads within a TB on the GPU.

Figure 2.3 shows a simple CUDA code that adds one vector to another vector. In the host code, first, the data on a host-side memory is copied to a device-side memory (lines 3-4). Second, the values of the variables `grid` and `block` are set to (100,1,1) and (256,1,1), respectively (lines 5-6). The type of these variables is `dim3` that represents a three-dimensional structure. In this simple code, both `grid` and `block` have one-dimension. Third, the kernel `kernelAdd` is launched with the two launch configurations, `grid` and `block` (line 7). Finally, the result of `kernelAdd` is copied to a host-side memory (line 8). Figure 2.3b shows the kernel `kernelAdd`, which adds the vector `x` to the vector `y` and stores the result into the vector `z`. In the device code, first, the flattened index of a thread within its one-dimensional TB is computed (line 2). Second, each thread loads the data `t1` and `t2` from the memory location to which `x[tid]` and `y[tid]` correspond, respectively (lines 3-4). Third, each thread adds `t1` and `t2` and stores the result into `t3` (line 5). Finally, each thread stores `t3` into the memory location to which `z[tid]` corresponds (line 6). As shown in the code, global memory is used for the data communication between CPUs and GPUs.

2.4 Types of Compiler Optimizations for GPU Programs

As mentioned in Section 1.2, compiler optimization techniques for GPU programs are categorized into two main types, intra-kernel optimizations and inter-kernel ones. In this section, the two types of optimizations are described in detail.

2.4.1 Intra-kernel Optimization

Intra-kernel optimizations analyze a single kernel and transform it into the more efficient one. A lot of traditional optimizations such as peephole optimizations and dataflow analysis based optimizations [17, 51, 45, 46] are included in this category, although they are not GPU-specific optimizations. As intra-kernel optimizations for GPUs, there are several types as follows:

- **Branch divergence reduction** techniques reduce the divergence in each kernel, resulting in performance improvement. Many techniques have been proposed, classified into software-based approaches [19, 36, 116, 110, 37, 8, 72, 86, 64, 21, 20, 90] that reduce the harmful influence of branch divergence on performance and hardware-based approaches [33, 69, 32, 24, 10, 87, 25, 50, 41, 107, 113, 114] that try not to cause branch divergence.

- **Intra-kernel data locality improvement** approaches [108, 15, 59, 62, 101, 100] improve GPU cache utilization in a single kernel. Most approaches expose data reuse opportunity between TBs within a kernel by scheduling TBs.

- **Dimensionally redundant instruction elimination** techniques [52, 111, 106, 65, 53, 115] remove redundant instruction executions at the grid, TB, and warp level. The warp-, TB-, and grid-wide redundant instructions often exist in a multi-dimensional TB and grid, causing performance degradation.

- **Occupancy optimizations** [92, 93, 38] try to retain high occupancy, which requires more GPU resources such as registers. Improving occupancy increases thread- and instruction-level parallelism and promotes hiding memory latency.

2.4.2 Inter-kernel Optimization

This type of optimization analyzes multiple kernels and improve overall performance of a GPU program. Some optimizations analyze both a host code and a device code to collect inter-kernel information that indicates whether the program can be optimized integrally. There are several types of inter-kernel optimizations for GPUs as follows:

- **Kernel fusion** methods [61, 102, 103, 104, 85, 84, 109, 27, 28, 67] achieve performance improvement by combining two or more kernels into a single one. Fusing kernels reduces kernel launch overhead and expensive data communication with GPU memory.

- **Inter-kernel data locality improvement** approaches [63, 42, 105] improve energy efficiency and performance by exploiting inter-kernel data reuse when multiple kernels operate on the same data.

- **Kernel scheduling and co-execution** techniques [105, 83, 117, 58, 91, 66] execute multiple kernels concurrently by scheduling kernels and co-execute the kernels, resulting in full utilization of GPU resources.

Chapter 3

Code Motion

Modern optimizing compilers transform an input program into the optimized one to execute the program efficiently. In the process, to remove redundant expressions, the compilers perform *code motion*. The code motion based techniques improve performance of a program by avoiding unnecessary recomputations of an expression. This is achieved by replacing the original expressions with the variable that holds the result of the expression calculated at suitable program points. The one of code motion based approach is *Partial Redundancy Elimination* (PRE) [70, 54, 55, 16, 48], which not only removes partially redundant expressions but also moves invariant expressions out of loops. PRE performs *safe code motion*, i.e., does not introduce a new computation on any execution path to preserve the semantics of the original program. Thus, PRE does not perform *speculative code motion*, which introduces a new computation on some execution paths. Although speculative code motion may decrease performance because it costs the new computation introduced on some paths, it can increase totally redundant expressions and remove more expressions. Therefore, speculative code motion can remarkably improve performance in some cases.

In the rest of this chapter, the detail of PRE and speculative code motion are described in Section 3.1 and Section 3.2, respectively.

3.1 Partial Redundancy Elimination

The first algorithm of PRE is proposed by Morel and Renvoise [70]. The key idea of the algorithm is to obtain *computationally optimal* results, that is, the results where the number of expressions on each execution path cannot be decreased any further as long as safe code motion is performed. In the process of obtaining the optimal results, the algorithm places expressions *as early as possible* in a program. This strategy performs unnecessary code motion and increases *register pressure*. To extend it in terms of minimizing the register pressure, Knoop et al. proposed *Lazy Code Motion* (LCM) [54, 55], which is widely adopted by modern optimizing compilers. The key idea of LCM is to obtain computationally and *lifetime optimal* results, i.e., to place an expression *as late as possible* while maintaining computational optimality. First, as well as the algorithm developed by Morel and Renvoise, LCM computes the earliest insertion points. Second, by delaying the earliest points while preserving the optimality, it finds the latest insertion points. Finally, it moves an expression into the latest points and replaces the original expression with the variable that holds the result of the expression.

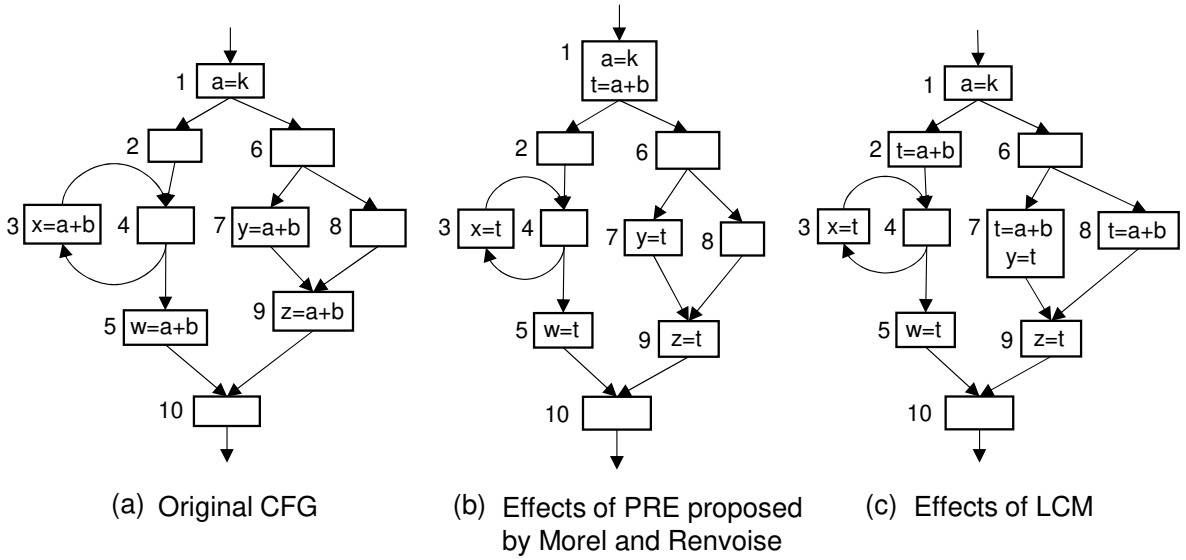


Figure 3.1: (a) Original CFG; (b) the CFG after applying PRE proposed by Morel and Renvoise to the CFG in (a); (c) the CFG after applying LCM to the CFG in (a).

PRE proposed by Morel and Renvoise transforms the CFG in Figure 3.1a into the one in Figure 3.1b by moving the expression `a+b` to node 1 and replacing the original expression with the variable `t`. In contrast, LCM transforms the CFG in Figure 3.1a into the one in Figure 3.1c by moving the expression to nodes 2, 7, and 8. Although both the PRE and LCM remove the redundancy of the expression `a+b`, unnecessary register pressure is avoided in the CFG shown in Figure 3.1c compared to the CFG in Figure 3.1b.

3.2 Speculative Code Motion

Speculative code motion introduces a new computation on some execution paths, which may decrease performance but can increase totally redundant expressions. Although speculative code motion is not performed by PRE because of safety, it is significantly effective in some cases. Several methods that extend traditional PRE to perform speculative code motion have been proposed [40, 35, 11, 112, 118]. They perform speculative code motion if performance benefit is expected for the speculation by entirely decreasing the execution frequency.

Consider the application of speculative code motion to the CFG shown in Figure 3.2a. It moves the expression `a+b` to node 1 as shown in the CFG in Figure 3.2b although the exit of node 1 is not down-safe. This code motion introduces the new computation on the execution path $1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7$, so that it may decrease performance. However, it is expected to increase performance if the execution path $3 \rightarrow 2 \rightarrow 3$ is executed many times. Once the expression `a+b` is moved to node 1, the expression in node 5 becomes totally redundant. Thus, the redundancy is removed by simply applying *Common Subexpression Elimination* (CSE) as shown in Figure 3.2c. In contrast, traditional PRE such as LCM performs no transformation for the CFG in Figure 3.2a because neither the code motion from node 2 to node 1 nor the one from

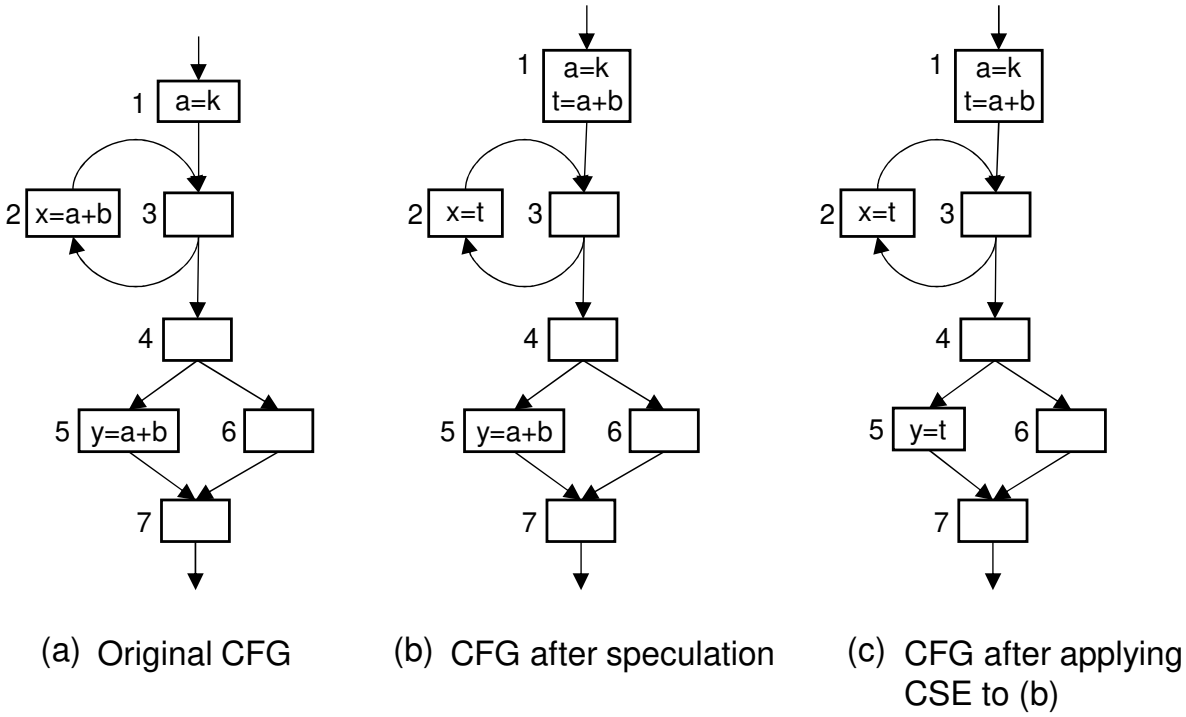


Figure 3.2: (a) Original CFG; (b) the CFG after moving the expression `a+b` from node 2 to node 1; (c) the CFG after applying Common Subexpression Elimination (CSE) to the CFG in (b).

node 5 to node 1 is safe, even though it may give performance improvement.

Chapter 4

Code Motion Based Branch Divergence Reduction

In this chapter, to improve performance of GPU kernels by reducing branch divergence, two code motion based techniques as intra-kernel optimizations, SSCM and SSRQP, are proposed. SSCM and SSRQP enable the traditional PRE and scalar replacement approaches to be applicable to GPU kernels that cause branch divergence, respectively. By exploiting the property of branch divergence, both methods speculatively hoist an expression out of a divergent branch without decreasing performance. The speculative code motion makes more expressions redundant, so that the proposed methods can remove more redundant expressions than traditional approaches. Further, both SSCM and SSRQP selectively apply speculative code motion to only divergent branches but not non-divergent branches because speculative code motion for non-divergent branches may decrease performance. To prove effectiveness of the methods, they are evaluated across a number of benchmarks.

The rest of the chapter is organized as follows. Section 4.1 gives the behavior of branch divergence and its problem in further detail. Section 4.2 and Section 4.3 describe SSCM and SSRQP, respectively. Finally, Section 4.4 summarizes this chapter.

4.1 Branch Divergence

As mentioned in Section 1.1, branch divergence can become significant performance bottlenecks of GPU kernels with the SIMT execution model. In the model, a kernel must execute the statements in both true and false sides of a divergent branch because each warp has just a single control flow.

In Figure 4.1a, assume that the conditional branches at nodes 1 and 2 cause branch divergence. As illustrated in Figure 4.1c, a warp has eight threads, which are represented by squares. The black squares represent executing threads, and the gray ones represent waiting threads. Consider the execution of the code shown in Figure 4.1a. First, all the threads execute statements at node 1 in parallel. At node 1, the threads from `t0` through `t3` evaluate the branch condition `p` to true while the threads from `t4` through `t7` evaluate it to false. Thus, branch divergence occurs at node 1. Second, the threads from `t0` through `t3` execute statements at node 2 in the true side of node 1. During the execution of the true side of node 1, the threads from `t4` through `t7` wait without doing anything, as shown in Figure 4.1c. At node 2, `t0` and `t1` evaluate the branch

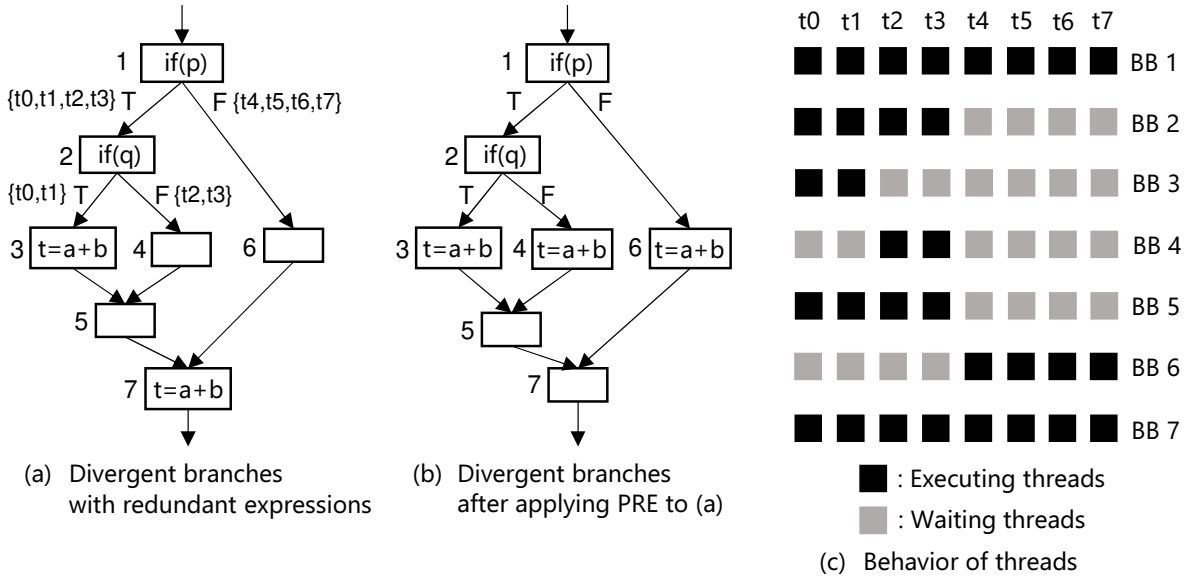


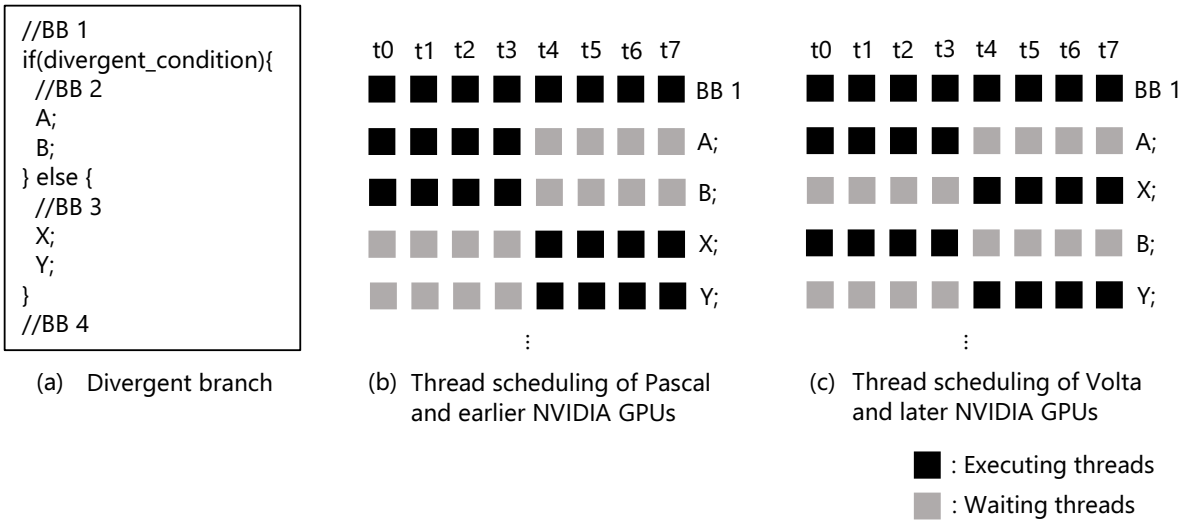
Figure 4.1: Divergent branches and behavior of threads

condition `q` to true while `t2` and `t3` evaluate it to false. The threads `t0` and `t1` execute statements at node 3 in the true side of node 2. At this time, `t2` and `t3` wait until the completion of the execution at node 3. Next, `t2` and `t3` execute statements at node 4 in the false side of node 2. At this time, `t0` and `t1` wait. After the execution at node 4, the threads from `t0` through `t3` execute statements at node 5. After that, the threads from `t4` through `t7` execute statements at node 6 in the false side of node 1. During the execution, the threads from `t0` through `t3` wait. Finally, all the threads execute statements at node 7.

The NVIDIA Volta GV100 and later GPUs support *Independent Thread Scheduling* (ITS) [73, 74]. It enables threads in the same warp to perform finer-grain synchronization and cooperation between the threads. ITS can interleave execution of statements in divergent branches, reducing the overhead of branch divergence in a program. Figure 4.2 compares the thread scheduling of Pascal and earlier GPUs to one of Volta and later GPUs. In Figure 4.2a, the conditional branch is assumed to be divergent. Pascal and earlier GPUs execute statements in the true-path of the branch and then the false-path in order (Figure 4.2b). In contrast, Volta and later GPUs interleave the execution of the true-path with the execution of the false-path (Figure 4.2c). Thus, ITS makes threads in a warp more flexible and gives higher efficiency for fine-grained parallel algorithms. However, the execution is still in SIMT fashion. In other words, active threads in a warp execute the same instruction. Therefore, branch divergence still decreases performance of a kernel.

As discussed above, branch divergence takes both costs of the true and false sides of a divergent branch to execute it, decreasing performance of a kernel. Code motion based approaches, such as PRE and scalar replacement, may increase branch divergence and decrease performance. Therefore, they cannot simply be applied to GPU programs with branch divergence.

As shown in Figure 4.1a, expressions originally exist in one side of the divergent branch node 2 and in node 7, and the expression in node 7 is partially redundant.



(a) Divergent branch

(b) Thread scheduling of Pascal and earlier NVIDIA GPUs

(c) Thread scheduling of Volta and later NVIDIA GPUs

Figure 4.2: Independent thread scheduling

The traditional PRE transforms Figure 4.1a into Figure 4.1b. In Figure 4.1b, the redundancy is removed by PRE. However, the expressions appear in both sides of the divergent branches at nodes 1 and 2. Therefore, execution efficiency is reduced compared with Figure 4.1a because of the divergence at nodes 1 and 2.

4.2 Speculative Sparse Code Motion

This section describes the detailed algorithm of SSCM. SSCM is realized by extending *Sparse Code Motion* (SCM) [89] so that SCM takes branch divergence into account. In the rest of this section, first, the algorithm of SCM is described. Second, SCM is extended to SSCM.

4.2.1 Sparse Code Motion

SCM [89] is realized by extending LCM [54, 55], which is a traditional PRE approach. In the following, the concept of SCM is defined, and behavior of SCM for an input program is described.

SCM consists of the following four steps:

1. Application of LCM: SCM applies LCM to an input program.
2. Computation of safe insertion points: SCM computes up-safe program points, down-safe points, and earliest points.
3. Computation of down-safety closure and down-safety region: SCM computes the down-safety closure $\rho(n)$ and down-safety region R.
4. Determination of sparse insertion points: SCM finds the insertion points of an expression and transforms a program based on the insertion points.

Each step is explained below, where the details and significance of the steps 2 and 3 are particularly described, and the outlines of the steps 1 and 4 are given.

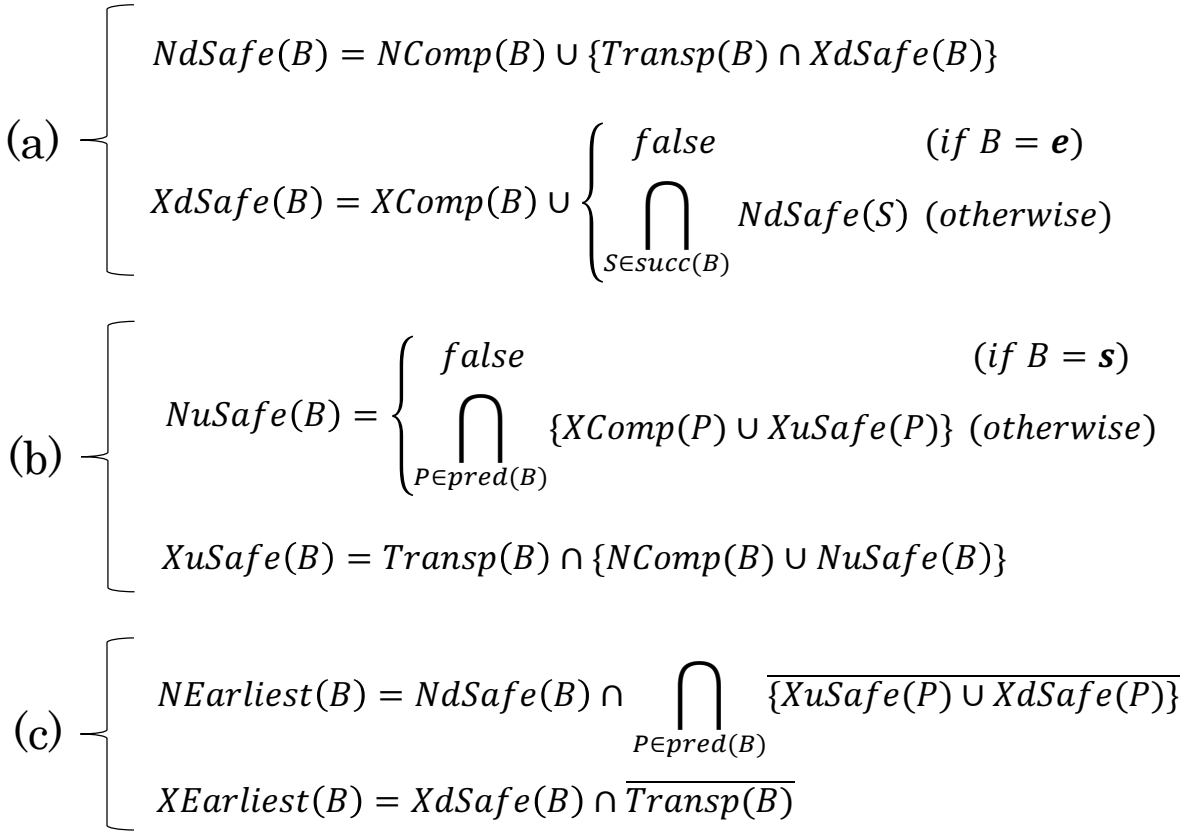


Figure 4.3: Dataflow equations for SCM

Application of LCM

SCM first applies LCM, which suppresses unnecessary code motion of PRE, to an input program. SCM inserts an expression at the optimal program point considering register pressure, as well as LCM. For the details of LCM, see Ref. [54, 55].

Computation of Safe Insertion Points

To compute safe insertion points, SCM defines the following predicates such as *NdSafe*, *XdSafe*, *NuSafe*, *XuSafe*, *NEarliest*, and *XEarliest* using the dataflow equations in Figure 4.3, where the predicates *NComp*, *XComp*, and *Transp* are locally determined. Each predicate denotes the followings:

- **NComp(B)**: B has an entry computation.
- **XComp(B)**: B has an exit computation.
- **Transp(B)**: B does not have a modification statement.
- **NdSafe(B)**: the entry insertion point of B is down-safe.
- **XdSafe(B)**: the exit insertion point of B is down-safe.
- **NuSafe(B)**: the entry insertion point of B is up-safe.

- **XuSafe(B)**: the exit insertion point of B is up-safe.
- **NEarliest(B)**: the entry insertion point of B is down-safe, but an expression cannot be moved to any predecessor of B.
- **XEarliest(B)**: the exit insertion point of B is down-safe, but an expression cannot be moved to the entry insertion point of B.

These predicates and dataflow equations used in SCM are the same as ones used in LCM. For ease of understanding, the notations *Comp*, *DnSafe*, *UpSafe*, and *RelComp* are also used as defined in the followings:

- $Comp =_{df} NComp$
- $DnSafe =_{df} NdSafe \cup XdSafe$
- $UpSafe =_{df} NuSafe \cap XuSafe$
- $RelComp =_{df} Comp \setminus UpSafe$

Computation of Down-safety Closure and Down-safety Region

In addition to down-safety, SCM defines *down-safety closure* $\rho(n)$ and *down-safety region* R. The definitions quoted from Ref. [89] are shown in the following. The down-safety closure $\rho(n)$ represents a set of nodes that is considered as the program points where an expression can be moved from the CFG nodes n and $succ(n)$. Down-safety region R represents a set of nodes that is considered as the program points where an expression can be moved in the entire CFG. These are based on the idea of adjusting insertion points determined by LCM; SCM considers down-safe and non-up-safe points as the program points where an expression is hoistable. Down-safety closure $\rho(n)$ is defined as the minimum set satisfying the following properties for CFG nodes $n \in DnSafe \backslash UpSafe$:

1. $n \in \rho(n)$
2. $\forall m \in \rho(n) \backslash Comp.succ(m) \subseteq \rho(n)$
3. $\forall m \in \rho(n).pred(m) \cap \rho(n) \neq \phi \Rightarrow pred(m) \backslash UpSafe \subseteq \rho(n)$

With down-safety closure, down-safety region R is defined as follows:

1. $RelComp \subseteq R \subseteq DnSafe \backslash UpSafe$
2. $\rho(R) = R$

Furthermore, SCM defines a set of program points *R-Earliest* that is the closest to the start node **s** in down-safety region R. The definition from Ref. [89] is shown below.

$$R - Earliest(n) \Leftrightarrow_{df} n \in R \cap ((n = \mathbf{s}) \cup \exists m \in pred(n). \neg Transp(m) \cup m \notin R \cup UpSafe)$$

SCM finds sparse insertion points of an expression in the down-safety region.

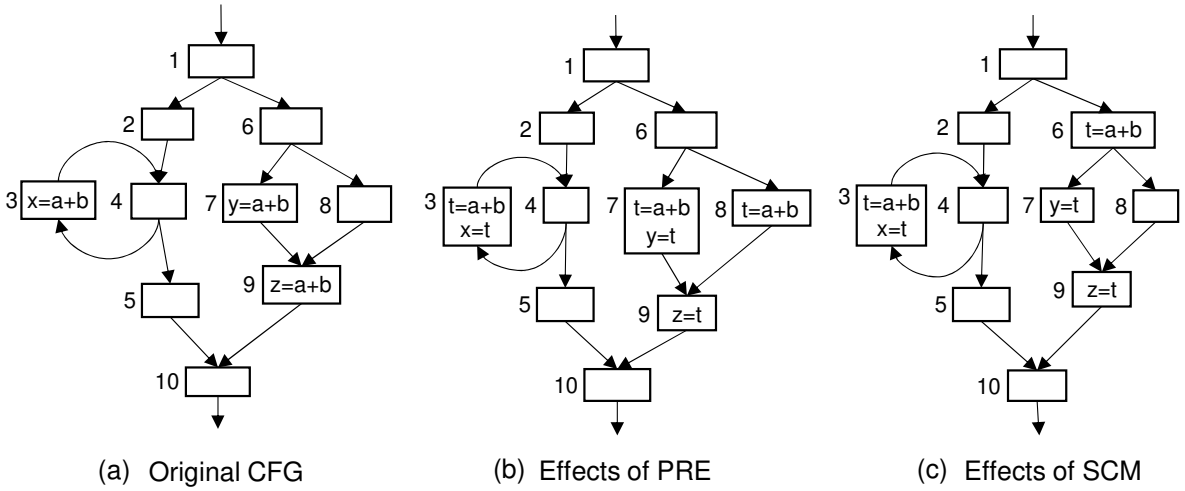


Figure 4.4: Effects of PRE and SCM

Determination of Sparse Insertion Points

Using the predicates and down-safety region mentioned above, SCM finds sparse insertion points of an expression, transforming a program based on them. First, SCM selects the down-safety region where the number of *R-Earliest* points for *RelComp* points is the fewest. Second, SCM inserts an expression at *R-Earliest* points in the down-safety region and removes expression occurrences at *RelComp* points. Through the transformation, SCM reduces the number of occurrences of an expression. In this process, because SCM inserts expressions into program points in the down-safety region as well as LCM, SCM does not perform speculative code motion that may increase the execution of expressions on some execution paths. See Ref. [89] for details of how to find insertion points in SCM.

LCM transforms the CFG shown in Figure 4.4a into the one shown in Figure 4.4b. SCM finds the down-safety region for the CFG in Figure 4.4b, getting two regions that consist of node 3 and nodes 6, 7, and 8, respectively. Finally, SCM finds sparse insertion points of an expression for the CFG in Figure 4.4b. Consequently, SCM gets node 6 as the insertion point, i.e., SCM transforms the CFG in Figure 4.4b into the one in Figure 4.4c by inserting an expression into node 6 and removing expression occurrences in nodes 7 and 8. Compared with the number of occurrences of the expression in Figure 4.4b, SCM can decrease it from three to two, as shown in Figure 4.4c. Notice that the number of the expression occurrences can be decreased to one if the insertion to node 1 is allowed. However, because node 1 is not down-safe, it results in speculative code motion, which introduces a new expression on the execution path $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 10$. LCM and SCM do not perform such speculative code motion.

4.2.2 Extension of SCM

In this section, SCM described in the previous section is extended to SSCM to allow speculative code motion. As well as SCM, SSCM consists of the following four steps:

1. Application of LCM,

(a)

$$NdSafe(B) = NComp(B) \cup \{Transp(B) \cap XdSafe(B)\}$$

$$XdSafe(B) = XComp(B) \cup \begin{cases} false & (if\ B = \boldsymbol{e}) \quad \text{— (1)} \\ \bigcup_{S \in succ(B)} NdSafe(S) & (if\ B \in Div) \quad \text{— (2)} \\ \bigcap_{S \in succ(B)} NdSafe(S) & (otherwise) \quad \text{— (3)} \end{cases}$$

Div : Set of a basic block ends with a divergent branch

(b)

$$NuSafe(B) = \begin{cases} false & (if\ B = \boldsymbol{s}) \\ \bigcap_{P \in pred(B)} \{XComp(P) \cup XuSafe(P)\} & (otherwise) \end{cases}$$

$$XuSafe(B) = Transp(B) \cap \{NComp(B) \cup NuSafe(B)\}$$

(c)

$$NEarliest(B) = NdSafe(B) \cap \bigcap_{P \in pred(B)} \overline{\{XuSafe(P) \cup XdSafe(P)\}}$$

$$XEarliest(B) = XdSafe(B) \cap \overline{Transp(B)}$$

Figure 4.5: Dataflow equations for SSCM

2. Computation of safe insertion points,
3. Computation of down-safety closure and down-safety region, and
4. Determination of sparse insertion points.

Steps 1, 3, and 4 are similar to SCM; their explanations are omitted in this section, and the step 2 is presented in detail.

Computation of Safe Insertion Points

In SSCM, the dataflow equations of SCM shown in Figure 4.3 are changed to ones in Figure 4.5. As shown in Figure 4.5, the equation (2) in Figure 4.5a is added to the equations of down-safety in Figure 4.3a, where the other equations are the same as ones in SCM. The equation (2) in Figure 4.5a is computed if B ends with a divergent branch instruction. In this case, if B is partially anticipated at the exit insertion point, it is regarded as a down-safe node. In other words, if branch divergence occurs, statements included in the true and false sides of the branch are executed. Thus, speculatively hoisting an expression before the branch does not decrease execution efficiency, which

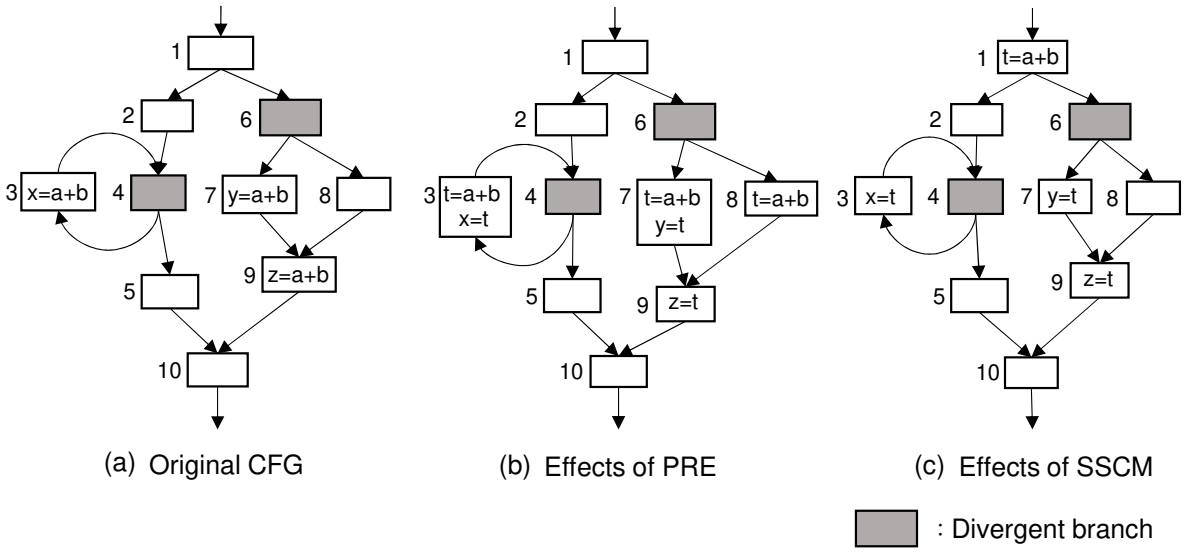


Figure 4.6: Effects of speculative sparse code motion

means that an expression can be safely hoisted out of the branch. In SCM, regardless of branch divergence, a node that is partially anticipated at the exit insertion point is not down-safe. That is, SSCM considers a larger down-safety region than SCM does. In particular, this property enables SSCM to achieve more sparse code motion than SCM.

Moreover, for a non-divergent branch, SSCM applies the same equation as one in SCM, as shown in Figure 4.5a (3), which does not perform speculative code motion that may decrease execution efficiency. The extension also contributes to the selective application of SSCM and SCM depending on the occurrence of branch divergence.

Figure 4.6a shows the CFG in which the shaded nodes cause branch divergence. Consider the application of SSCM to the CFG. First, the application of LCM results in the CFG shown in Figure 4.6b. Computing the dataflow equations in Figure 4.5, XdSafe at node 4 becomes true through Figure 4.5a (2) because node 4 is a divergent branch. Once XdSafe becomes true, which means the exit insertion point at node 4 is down-safe, the entry insertion points of nodes 2 and 6 also become down-safe by computing the XdSafe, so that the exit insertion point at node 1 becomes down-safe though node 1 is non-divergent branch. Second, the down-safety region for the CFG in Figure 4.6b, which consists of nodes 1, 2, 3, 4, 6, 7, and 8, is found. Exploring the sparse insertion point of an expression in the down-safety region, node 1 is found as the insertion point. Finally, the CFG shown in Figure 4.6c is obtained by inserting an expression into node 1 and removing expressions at nodes 3, 7, and 8 in the CFG in Figure 4.6b. Comparing the number of occurrences of the expression in Figure 4.6b with the one in Figure 4.6c, it is found that the number of them decreases from three to one. Moving $a+b$ at node 3 in Figure 4.6b before the branch at node 4 is speculative code motion that introduces new computations on the execution path $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 10$. In PRE and SCM, such speculative code motion is not allowed. Furthermore, because branch divergence also occurs at the branch at node 6, both expressions at nodes 7 and 8 in Figure 4.6b are executed in order. SSCM hoists the expressions out of the branch with them, as well as SCM, shown in Figure 4.6c, which directly contributes

to branch divergence reduction.

Although the termination of this algorithm is not discussed, SCM and SSCM have monotonicity of the dataflow analysis [7], which guarantees that a solver of the dataflow analysis terminates. Moreover, the computational complexity of SSCM is of order $O(N^{\frac{7}{2}})$, where N is the size of an input program. This is because SSCM consists of LCM and SCM, which are accomplished in time $O(N^2)$ and $O(N^{\frac{7}{2}})$, respectively [54, 55, 89].

4.2.3 Evaluation

To evaluate the effectiveness of the proposed method, two kinds of experiments were conducted. One is a comparison of execution efficiency for the eight benchmarks, and the other one is a comparison of execution efficiency based on the number of threads. The open-source software Ocelot CUDA compiler [23] was used for the experiments. The Ocelot is a backend for PTX [82] similar to a GPU assembly code and also works as a PTX optimizer. Furthermore, the divergence analysis [19] implemented in Ocelot was utilized to identify divergent branches. The description of an environment in the experiments are as follows:

- OS: Ubuntu 16.04 LTS,
- CPU: Intel Core i7-4770K,
- GPU: Geforce GTX TITAN Black, and
- CUDA Toolkit 5.0.

Experiment A

In the experiment, the proposed SSCM and traditional methods were implemented, compared in terms of the execution time of object codes for the eight benchmarks. The benchmarks are two programs (*barnshut*, *knn*) of Treelogy benchmark [39], one program (*cfd*) of Rodinia benchmark [14], two programs (*FDTD3D*, *eigenvalues*) of NVIDIA SDK Sample code [77], and two programs (*histo*, *mri-q*) of Parboil benchmark [95]. Moreover, two kernel functions for *eigenvalues* were measured. The program of *barnshut* is the n-body simulation that computes forces acting on a body due to its interaction with other bodies by using kd-trees. *knn* finds the k-nearest neighbors to each query point by using kd-trees. *cfd* is an unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow. *FDTD3D* applies a finite differences time domain progression stencil on a 3D surface. *eigenvalues* computes all eigenvalues of a tridiagonal symmetric matrix of arbitrary size. *histo* computes 2-D saturating histogram with a maximum bin count of 255. *mri-q* computes a matrix Q representing the scanner configuration for calibration used in a 3D magnetic resonance image reconstruction algorithms.

First, these methods were applied to the PTX code obtained by the NVIDIA CUDA compiler, *nvcc* [78], with the optimization option O3. Figure 4.7 shows the results of the experiment. In the figure, *O3* represents the execution time of the object code generated by nvcc with the optimization option O3, where *O3* is used as a baseline

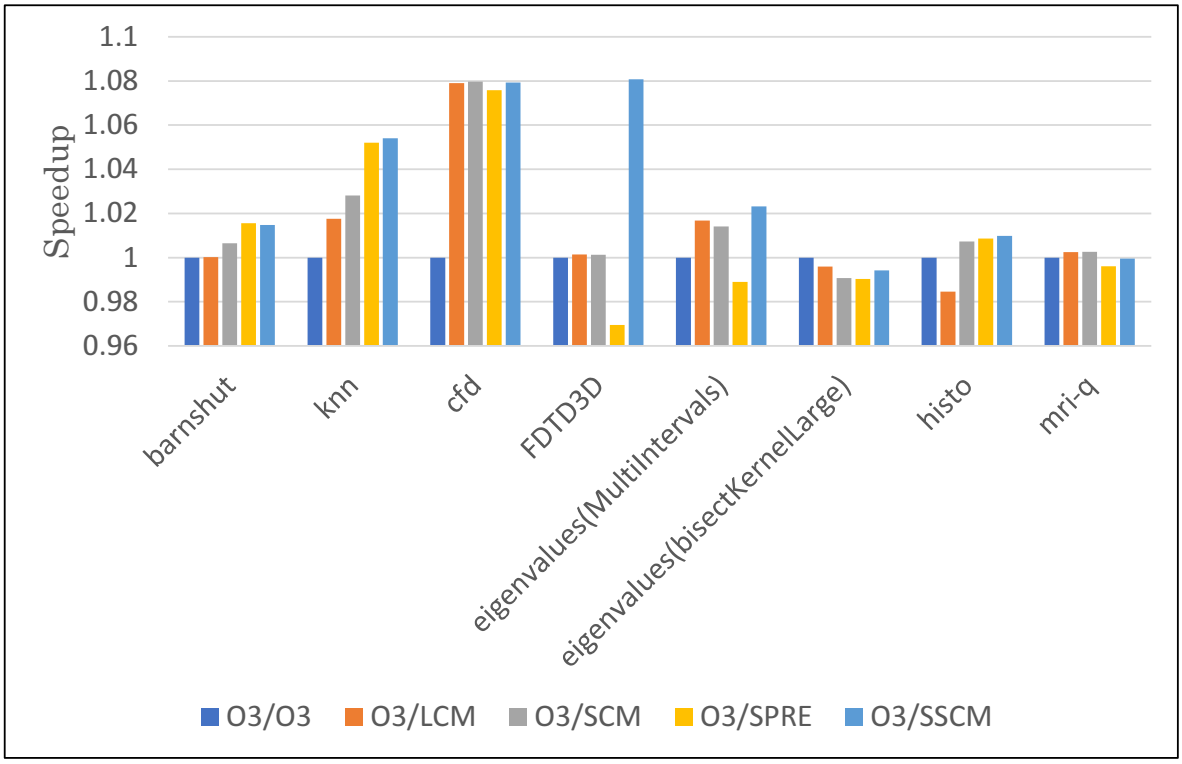


Figure 4.7: Experiment A: comparison of execution speed

in the evaluation. The *LCM*, *SCM*, *SPRE*, and *SSCM* in the figure represent the execution time when applying LCM, SCM, speculative PRE, and SSCM, respectively, to the PTX code generated by nvcc with O3. Each result is shown as the ratio of each execution time to the baseline. As shown in Figure 4.7, the proposed method SSCM improved performance for six programs: *barnshut*, *knn*, *cfd*, *FDTD3D*, *eigenvalues(MultiIntervals)*, and *histo*. For the program of *barnshut*, the proposed method performs speculative code motion that is never performed by LCM and SCM at divergent branches. Thus, it could remove redundancy and suppress branch divergence more than LCM and SCM, which contributes to performance improvement. On the other hand, LCM could not perform any transformation. SCM statically decreased the number of expressions, but it achieved less efficiency than SSCM because it does not perform speculative code motion for divergent branches. SPRE performed speculative code motion without considering branch divergence, but it achieved the same result as SSCM. For the program of *knn*, SPRE and SSCM improved the performance by 5% approximately. These methods hoisted expressions speculatively for divergent branches, removing redundancy and reducing branch divergence more than LCM and SCM. SPRE did not decrease the performance because of speculative code motion without considering branch divergence. For the program of *cfd*, all the methods of LCM, SCM, SPRE, and SSCM improved the performance by 8% approximately by removing many redundant expressions. SPRE and SSCM did not perform speculative code motion. For the program of *FDTD3D*, SSCM improved the performance by 8.07% while LCM and SCM could not perform any transformation and get performance gain. SSCM hoisted many expressions in a loop speculatively for divergent branches, contributing to redundancy elimination and branch divergence reduction. In contrast, SPRE achieved less effi-

ciency than other methods because it speculatively hoisted many expressions without considering branch divergence and introduced new expressions into some paths. For the program of *eigenvalues(MultiIntervals)*, SSCM improved the performance through speculative code motion for divergent branches in a loop, which removed redundancy and reduced branch divergence more than LCM and SCM. SPRE performed speculative code motion without considering branch divergence, decreasing the performance. For the program of *histo*, LCM decreased the performance because it increased branch divergence by inserting expressions into the destinations of divergent branches. However, SCM, SPRE, and SSCM achieved performance improvement by hoisting the expressions before the branches.

Although the proposed method improved the performance of the above mentioned six programs, it was not able to improve that of *eigenvalues(bisectKernelLarge)* and *mri-q*. For the program of *eigenvalues(bisectKernelLarge)*, LCM and SCM eliminated some redundant expressions in a loop but they could not improve performance. SPRE and SSCM performed speculative code motion for divergent branches, which could not contribute to performance improvement. For the program of *mri-q*, all the methods removed redundancy, and SCM, SPRE, and SSCM hoisted expressions before divergent branches. However, all the methods could not improve the performance. SSCM performs code motion based on the results of the divergence analysis, which pessimistically determines whether each branch causes branch divergence or not [19]. Therefore, the proposed method might perform speculative code motion for non-divergent branches, decreasing the performance. In summary, this experiment has indicated that the proposed method is particularly effective for the tree traversal program (*knn*) and the simulating programs (*cfd* and *FDTD3D*). Generally, tree traversal and simulating programs tend to have more divergent branches than image processing and linear algebra programs. Thus, for programs with the property, SSCM is particularly more effective than other methods.

Experiment B

In this experiment, the improvement in execution efficiency was compared by changing the number of TBs and the number of threads. In GPU programs, as the number of threads per TB increases, the number of registers that can be used by one thread decreases. Because the method with code motion tends to extend the lifetime of variables and increase register pressure, this experiment is important to know the effect of the method with code motion on GPUs. The sample program, MCML (Monte Carlo Modeling of Light Transport in Multi-Layered Tissues) [44], which is relatively less tuned than the benchmarks used in Experiment A, was used. The proposed method was applied to the PTX code of the sample program obtained by using nvcc with the optimization option O3 and compared the performance of them for settings of the number of TBs in starting the kernel function to 1, 30, 60, 120, and 240, and settings of the number of threads to 32, 64, 128, 192, and 256.

Figure 4.8 shows the results of the experiment. The execution efficiency decreased when the number of TBs was 30 and the number of threads was 128 and 192, when the number of TBs was 60 and the number of threads was 32, and when the number of TBs was 120 and the number of threads was 32. The decrease in execution efficiency was up to 8.9%. However, in other cases, the efficiency was improved. In particular, 66.4% improvement was obtained when the number of TBs was 240, and the number

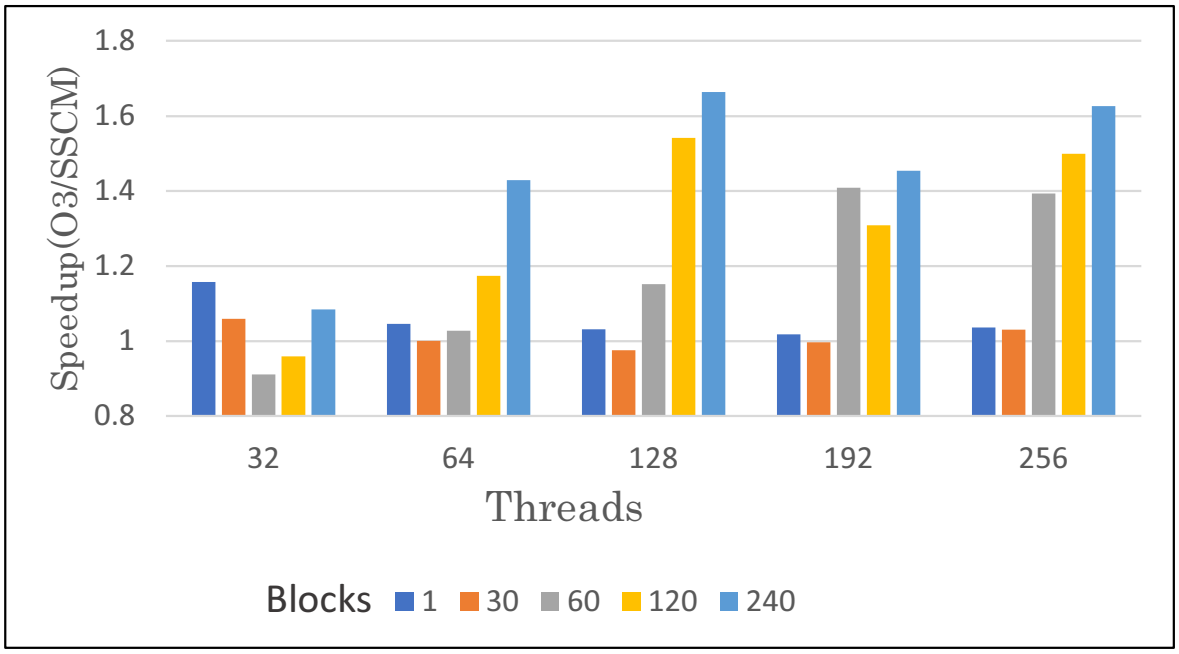


Figure 4.8: Experiment B: comparison of execution speed

of threads was 128. The reason why execution efficiency decreased is that some register spilled over because of the lack of register resources shared by threads in one TB. In contrast, the reason why the efficiency increased is that the efficiency improved by the proposed method for suppressing branch divergence exceeded the efficiency decreased by the lack of register resources.

4.3 Speculative Scalar Replacement based on Question Propagation

In this section, the algorithm of SSRQP is described in detail. First, an outline of question propagation [88] is introduced. Second, the algorithm of *scalar replacement based on question propagation* (SRQP) [96], which is the basis of SSRQP, is described. Thrid, SRQP is extended to SSRQP by considering branch divergence. Finally, SSRQP and related methods are evaluated through the well-known benchmarks.

4.3.1 Question Propagation

Question propagation [88] is a method that checks whether each expression is redundant or not by propagating questions on a CFG. As methods using the question propagation, *partial redundancy elimination based on question propagation* (PREQP) [99, 97] and SRQP [96] have been proposed. These methods apply a sequence of processes to each CFG node, which consists of a statement at most, while traversing a CFG in topological sort order. If the current node has an expression e, they check whether e can be eliminated by propagating a query about availability backwardly in order to determine whether the lexically same expressions as e exist at the node where the query is prop-

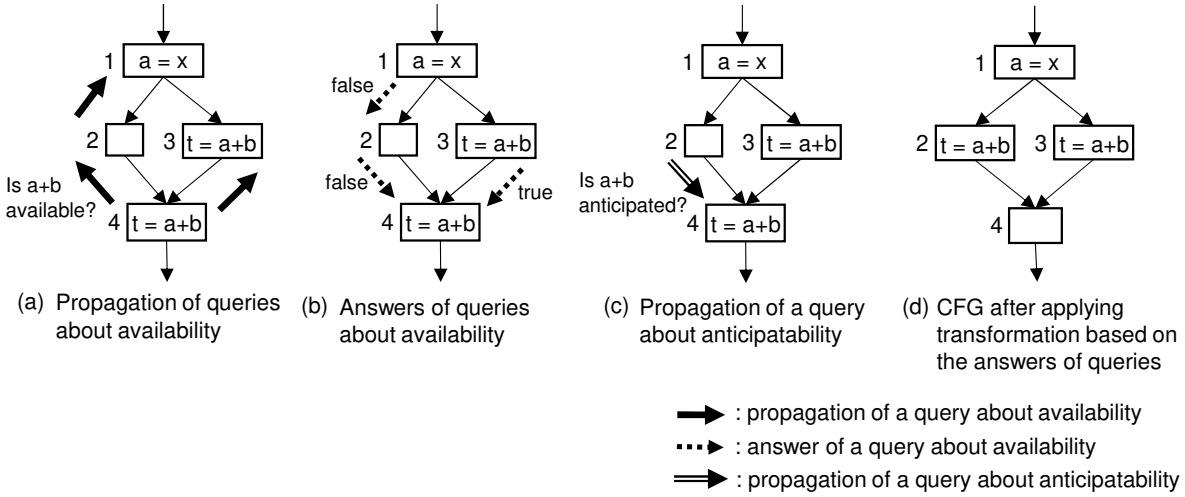


Figure 4.9: Example of PREQP

agated. This expression e is called a *questionary expression*. If a query is propagated to a node that has the same expression as a questionary expression, the node returns *true* as its answer which is forwardly propagated to the originating node of the query. Conversely, if a query is propagated to a node that has the statement that modifies the operands of a questionary expression, the node returns *false*. If both true and false are returned to a node n from the predecessors, another question propagation checks whether n is down-safe through propagating a query about anticipatability forwardly. If the query is propagated to a node that has the same expression as the questionary expression, the node returns *true*. Once n is found to be down-safe, the questionary expression is inserted into the node that returns *false* as its answer about availability to make the questionary expression totally redundant at n, so that n returns *true* as its answer about availability toward the originating node of the propagation. Consequently, e is decided to be redundant if the originating node gets *true* as its answer about availability.

The question propagation in PREQP about the expression $a + b$ at node 4 is illustrated in Figure 4.9a. First, it propagates queries about availability to check whether $a + b$ is available to the predecessors of node 4. Second, the query propagated to node 3 returns *true* as its answer because $a + b$ is available in node 3, as shown in Figure 4.9b. On the other hand, the query propagated to node 2 does not solve its answer because there are no statement in node 2. In this case, the query is propagated further to the predecessor of node 2. The query propagated to node 1 returns *false* because the operand of $a+b$ is modified by the statement $a = x$ at the node. As a result, both *true* and *false* are obtained at node 4, which denotes the questionary expression is partially redundant. To remove the redundancy, as illustrated in Figure 4.9c, a query about anticipatability is propagated in order to check whether node 4 is down-safe. Consequently, *true* as the answer is obtained, so that node 4 is found down-safe. Finally, the expression $a+b$ at node 4 is removed by inserting the same expression to node 2, which results in the CFG shown in Figure 4.9d.

4.3.2 Scalar Replacement Based on Question Propagation

Scalar replacement based on question propagation (SRQP) is an extension of PREQP to scalar replacement. SRQP checks whether each array reference expression e at each CFG node is redundant over iterations through question propagation.

In SRQP, the answer of a query about availability is a tuple of *isAvail* and *isReal*, where *isAvail* represents whether a questionary expression is available and *isReal* represents the fact of reaching some occurrences of the same expression as a questionary expression. The expression that originates question propagation is an array reference expression such as $A[i]$, of which the operands of the questionary expression include A and i. In the propagation process, the answer of a query about availability at node n is determined as follows:

(1) If n is the start node **s**, the answer is (*false*, *false*).

(2) If n is the node where the query has already been propagated twice, the answer is (*false*, *false*).

(3) If n is the node where the same query has already been propagated, the answer is (*true*, *false*).

(4) If n contains a statement that may modify the operands of a questionary expression or a memory location accessed through it, the answer is (*false*, *false*).

(5) If n has the same expression as a questionary expression, the answer is (*true*, *true*).

(6) If n has definitions of the operands of a questionary expression and is visited for the first time, the operands are replaced with the right-hand side of the definition statement for algebraic conversion.

(7) If any above rules are not applied, the query is propagated to all of the predecessors of n.

The above rules are applied in order from the top to the bottom. In the process of question propagation, an answer can be solved locally in each node if any of rules (1) to (6) is satisfied. If an answer cannot be solved locally, SRQP propagates a query according to the rule (7) and applies the rules to the predecessors.

As shown in the rule (2), different queries are allowed to visit a node at most twice to analyze loop-carried availability and remove inter-iteration redundant expressions. As shown in the rule (3), the answer of *isAvail* is *true* if the same query has already been propagated to n. This is because the result of question propagation corresponds to the maximum fixed point of the dataflow equations. Moreover, as shown in the rule (4), a questionary expression must not be propagated beyond a modification statement of the questionary expression. The modification includes a store statement to the same memory location as the array reference. In general, it is difficult to decide whether two memory references access the same memory location because one may be an alias of the other. In order to expose such alias relation, pointer analysis or alias analysis may be required. If a store statement may modify the memory location referenced, SRQP conservatively considers that the referenced memory location is modified. As shown in the rule (6), the operand i of a questionary expression is replaced with the right-hand

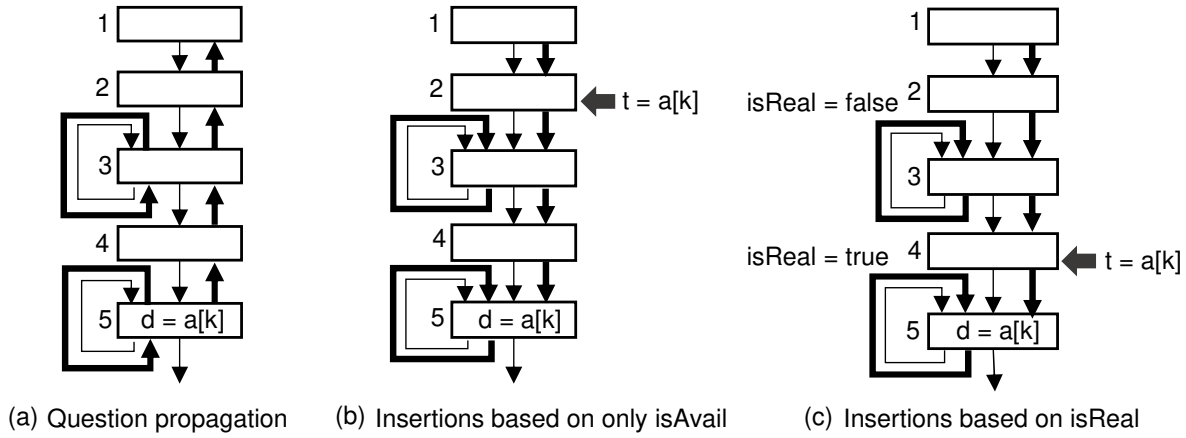


Figure 4.10: An example of unnecessary code motion

side of a definition statement such as $i = i + 1$. That is, the operand i is replaced with $i + 1$. By means of this rule, SRQP can analyze loop-carried availability.

If all of the queries propagated to predecessors of n have *true* as their answers of *isAvail*, the answer of *isAvail* at n is also *true*. If the answers of *isAvail* from predecessors of n have both *true* and *false*, a questionary expression at n is partially available. At this time, SRQP inserts a questionary expression into the predecessors where the answers are *false* if the expression is anticipated at n, so that the answer at n becomes *true*. Notice here that if predecessors that have *true* as their answers of *isAvail* are caused by repropagating a query to the same point without reaching an occurrence of the same expression as a questionary expression, unnecessary insertions at predecessors with *false* result in such as ineffective code motion through empty loops (called *hoisting-through-the-loop effect* [22]), as illustrated in Figure 4.10b. To avoid the unnecessary insertions, SRQP defines a predicate *isReal*, which represents the fact of reaching some occurrences of the same expression as a questionary expression. SRQP inserts new expressions only if some of predecessors with *true* as their answers of *isAvail* also have *true* as *isReal*. As shown in Figure 4.10c, the propagation to node 3 does not satisfy this requirement, whereas the one to node 5 satisfies *isReal*. As a result, SRQP inserts a new expression into node 4. Note that the result is the same as that of LCM [54, 55], and the predicate *isReal* works the same way as the predicate *Latest* used in LCM.

SRQP checks anticipatability through propagating another query as well as availability. The query about anticipatability is forwardly propagated contrary with the one about availability. In addition, a questionary expression of a query about anticipatability contains all the operands that have been algebraically converted during the propagation of a query about availability. For example, it may contains i and $i + 1$, where the operand i have been converted to $i + 1$. At this time, this means that SRQP checks the anticipatability of $A[i]$ and $A[i + 1]$ at the same time. The answer of a query about anticipatability at node n is decided as follows:

(1) If n is the end node **e**, the answer is *false*.

(2) If n is the node where the same query has already been propagated, the answer is *true*.

(3) If n contains a statement that may modify the operands of a questionary expression or a memory location accessed through it, the answer is *false*.

(4) If n has the same expression as a questionary expression, the answer is *true*.

(5) If any above rules are not applied, the query is propagated to all of the successors of n.

As described above, SRQP checks availability and anticipatability for each array reference expression e at n. Let ($isAvail_p$, $isReal_p$) be the value returned as the result of propagation from the predecessors of n, and let *isDownSafe* be the result of propagation of a query about anticipatability at n. The condition for the availability of e at n is as follows:

$$\prod_{p \in pred(n)} isAvail_p \vee isDownSafe \wedge \bigcup_{p \in pred(n)} isReal_p$$

If the above condition is *true*, n returns *true* toward the originating node of the propagation.

Consider the question propagation for the expression $A[i]$ in node 8 in Figure 4.11a, where an array reference $A[i]$ denotes the memory access to the i-th element of the array A. First, SRQP propagates a query about availability to the predecessors of node 8 to check the availability of $A[i]$. When the query visits node 2, it is propagated further to nodes 1 and 9. The query propagated to node 9 is propagated further to node 8. At this time, SRQP performs algebraic conversion of the questionary expression because node 8 has the statement $i = i + 1$. Hence, SRQP converts the questionary expression $A[i]$ to $A[i + 1]$. Next, the query about $A[i + 1]$ is propagated to the predecessors of node 8 as shown in Figure 4.11b. The query propagated to node 7 is propagated further to the predecessors of node 7. The one about $A[i + 1]$ propagated to node 5 gives *true* as its answer because the expression $A[i+1]$ exists in the node. On the other hand, the one propagated to node 6 is propagated further to the predecessors of node 6, resulting in *false* as its answer according to the rules (2) and (4). Consequently, at node 7, both *true* and *false* as answers of queries about availability are obtained as illustrated in Figure 4.11c. Since the information of down-safety is required to generate an answer at node 7 from the answers at predecessors, SRQP forwardly propagates a query about anticipatability to check whether node 7 is down-safe. In the process of the propagation, it checks whether the expression $A[i]$ or $A[i + 1]$ is anticipated. As a result, the query about anticipatability gives *true* as the answer of node 8, which denotes that node 7 is down-safe. Thus, node 7 returns *true* as its answer of the query about availability to node 8, and node 6, which returns *false* as its answer of a query about availability, is marked as the destination of insertion of the questionary expression $A[i+1]$. Remember that the query for $A[i+1]$ is propagated from node 8 to node 3. It is propagated further to the predecessors of node 3, and it gives *false* as its answer as well as node 6. Consequently, as shown in Figure 4.11d, both *true* and *false* as answers of queries about availability are obtained at node 8. Similarly to the case in node 7, SRQP checks whether node 8 is down-safe through propagating a query about anticipatability. The query results in *true* for down-safety at node 8, so that node 8 returns *true* as its answer of the query about availability to node 9, and node 3 is marked as an insertion point of the questionary expression $A[i + 1]$. As shown in Figure 4.11e, node 2 obtains *true* as the answer of node 9 and *false* as the answer of node 1. Similarly to the case in

nodes 7 and 8, SRQP checks whether node 2 is down-safe through propagating a query about anticipatability. The query results in *true* for down-safety at node 2, so that node 2 returns *true* as its answer of the query about availability, and node 1 is marked as an insertion point of the questionary expression $A[i]$. As a result, node 8 that is the originating node of a query obtains *true* as the answer of the query about availability. Based on the result of the question propagation, SRQP transforms the CFG in Figure 4.11a to the one in Figure 4.11f through inserting $t2 = A[i]$ into node 1 and $t = A[i+1]$ into nodes 3 and 6, which are the nodes marked during the propagation.

In the CFG illustrated in Figure 4.11f, the redundancy of an array reference between $A[i]$ and $A[i+1]$ is removed. However, if the branch nodes 2 and 4 are divergent, branch divergence is increased compared to the CFG in Figure 4.11a. This is because of the insertion into nodes 3 and 6, which are the destinations of divergent branches at nodes 2 and 4.

4.3.3 Extension of SRQP

In this section, SRQP is extended in the following points:

(1) insertion of expressions considering branch divergence, and

(2) propagation of a query about speculation and code motion based on the answer.

Insertion of Expressions Considering Branch Divergence

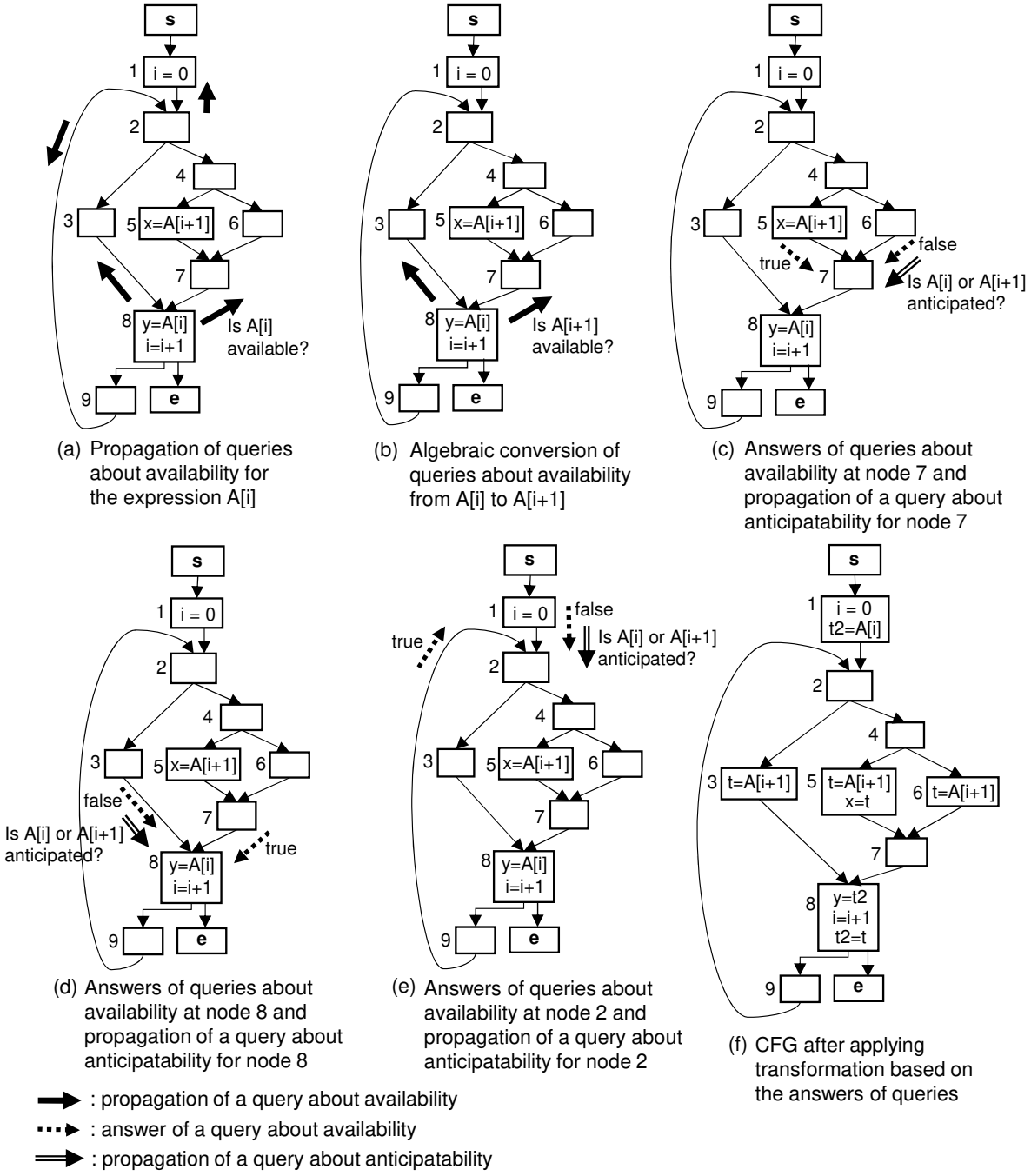
As mentioned in the previous section, the application of SRQP may increase branch divergence. The increase is suppressed through insertion of expressions based on a control dependency property. Here, a *control dependence region of non-divergence* (CDRND) is defined as follows:

definition. *A control dependence region of non-divergence is the set of nodes that are control-dependent on only non-divergent branches.*

A CDRND denotes a region where branch divergence is not increased by insertion of expressions, which makes a partially redundant expression totally redundant. When inserting an expression to a node, whether the node is included in a CDRND is checked. If a node where an expression is inserted is not in a CDRND, the insertion is suppressed in order not to increase branch divergence.

Consider a CDRND of the CFG in Figure 4.11, where the branch nodes 2 and 4 are divergent while the node 8 is non-divergent. In addition, a control dependency is computed for the augmented CFG [26], of which the special node ENTRY to be non-divergent. Nodes 3, 4, and 7 are control-dependent on node 2, and nodes 5 and 6 are control-dependent on node 4. On the other hand, nodes 2, 8, and 9 are control-dependent on node 8, and nodes **s**, 1, 2, and 8 are control-dependent on the special node ENTRY. Thus, the CDRND of the CFG consists of nodes **s**, 1, 2, 8, and 9. Therefore, the insertion of expressions into nodes 3 and 6 is suppressed in Figure 4.11f.

SRQP is extended to *Extended SRQP* (ESRQP) through suppression of insertion based on a CDRND. As mentioned above, if a node obtains *true* and *false* as the answers of queries about availability from the predecessors, SRQP inserts a questionary expression into the node that returns *false*. On the other hand, ESRQP checks whether



(a) Propagation of queries about availability for the expression A[i]

(b) Algebraic conversion of queries about availability from A[i] to A[i+1]

(c) Answers of queries about availability at node 7 and propagation of a query about anticipatability for node 7

(d) Answers of queries about availability at node 8 and propagation of a query about anticipatability for node 8

(e) Answers of queries about availability at node 2 and propagation of a query about anticipatability for node 2

(f) CFG after applying transformation based on the answers of queries

Figure 4.11: Propagation of queries about availability and anticipatability in SRQP

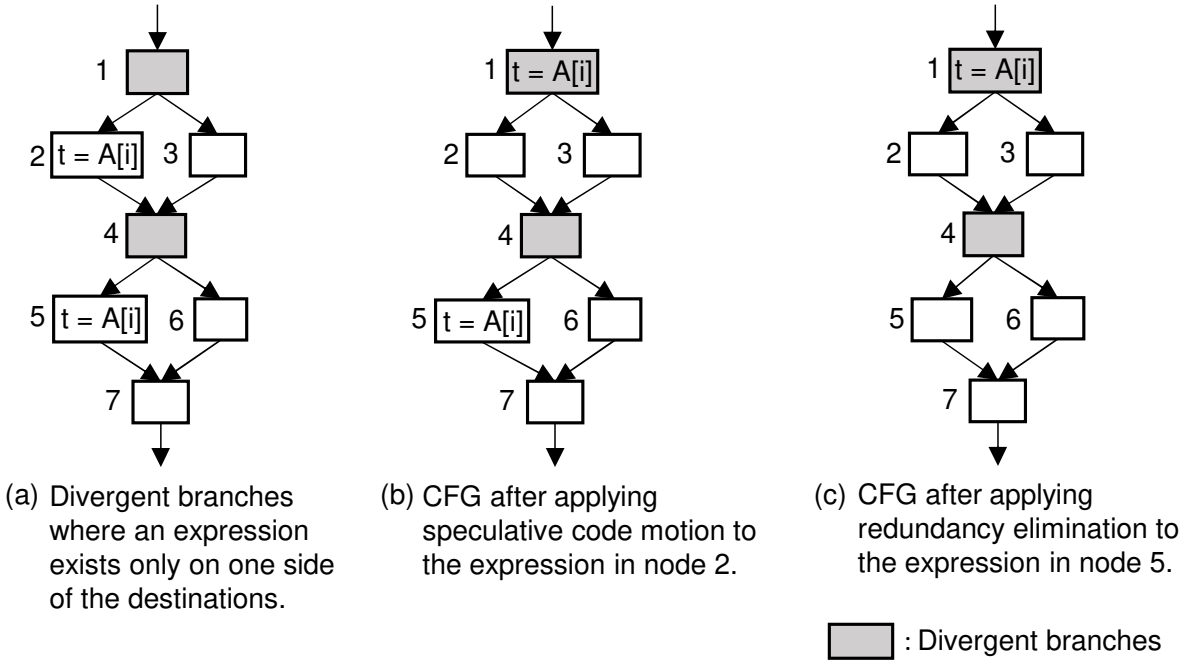


Figure 4.12: Speculative code motion for a divergent branch

the node that returns *false* is included in a CDRND. It inserts a questionary expression into the node only if it gives *true* for whether the node is included in a CDRND, resulting in suppression of increase in branch divergence. The pseudocode of the algorithm of ESRQP is shown in Appendix 4.5.1.

Propagation of a Query About Speculation and Code Motion Based on the Answer

Utilizing the property that both destinations of a divergent branch are executed, an expression that exists in only one side of a divergent branch can be speculatively hoisted out of it without decreasing execution efficiency. Through adding the speculative code motion, ESRQP is extended, which is called Speculative SRQP (SSRQP). SSRQP is realized by propagating a query about speculation.

The effectiveness of speculative code motion for a divergent branch is illustrated in Figure 4.12. In the figure, the shaded nodes 1 and 4 cause branch divergence. In Figure 4.12a, expressions exist in only one side of the destinations of these branches, and the expression in node 5 is partially redundant. Traditional PRE or scalar replacement cannot remove the redundancy because of safety. On the other hand, utilizing the property of branch divergence, the expression in node 2 can be hoisted to node 1 speculatively as shown in Figure 4.12b. In consequence of this, the expression $A[i]$ becomes available at node 5, so that it can be eliminated as shown in Figure 4.12c. In addition, branch divergence of the CFG in Figure 4.12c is reduced more than that in Figure 4.12a because the statements at nodes 2 and 5, which are control-dependent on divergent branches, are suppressed. Thus, speculative code motion for a divergent branch makes more expressions available without decreasing execution efficiency and reduces branch divergence.

A query about speculation is propagated to check whether an expression can be hoisted out of a divergent branch speculatively. The propagation realizes speculative code motion from node m to node n based on a control dependency relation between m and n. As discussed above, branch divergence can be reduced by hoisting an expression from a node that is control-dependent on a divergent branch out of the branch. The answer of a query about speculation is a tuple of an answer that denotes whether speculative code motion can be performed and a set of nodes to which a questionary expression e is inserted if the answer is *true*. The query about speculation (e, m, n) is backwardly propagated based on the following rules:

(1) If n is the start node **s**, the answer is (*false*, $\emptyset$).

(2) If n is a node where the same query has already been propagated, the answer is (*true*, $\emptyset$).

(3) If n contains a store statement which may modify the memory location referenced by the questionary expression, or a definition statement of the operands of the questionary expression, the answer at n is decided by the following rules:

  (i) If n is a divergent branch node, the answer is decided by the following rules:
    (a) If m is control-dependent on n, the answer is (*true*, $\{n\}$).
    (b) If m is not control-dependent on n, the answer is (*false*, $\emptyset$).
  (ii) If n is not a divergent branch node, the answer is (*false*, $\emptyset$).

(4) If n is a divergent branch node, the answer is decided by the following rules:

  (i) If m is control-dependent on n, the query is propagated further to the predecessors of n to check whether e can be hoisted to an earlier node; therefore m is replaced with n. Let q_{ans} be the answer in which the propagation results. The answer at n is decided by the following rules:
    (a) If q_{ans} is (*true*, x), the answer is also q_{ans}.
    (b) If q_{ans} is (*false*, x), the answer is (*true*, $\{n\}$).
  (ii) If m is not control-dependent on n, the answer is the result of the query propagated further.

(5) If n is a non-divergent branch node, the answer is decided by the following rules:

  (i) If n is down-safe, the answer is decided by the following rules:
    (a) If m is control-dependent on n, the answer is the result of the query propagated further after replacing m with n.
    (b) If m is not control-dependent on n, the answer is the result of the query propagated further.
  (ii) If n is not down-safe, the answer is (*false*, $\emptyset$).

(6) If any above rules are not applied, the query is propagated to all of the predecessors of n.

The above rules are applied in order from the top to the bottom. Here, m and n are initialized to the node where a query about speculation originates, A query is propagated to the predecessors p of n while n is replaced with p, so that n represents the node where the query is currently being propagated. Let (*canHoist*, *Node*) be the answer of a query about speculation. If *canHoist* is *true*, e can be speculatively hoisted to the nodes in *Node*. Conversely, any transformation cannot be performed if *canHoist* is *false*.

In the process of propagation, it is blocked by a store statement which may modify the memory location referenced by e or a definition statement of the operands of e because e cannot be hoisted beyond such a statement. Thus, as shown in the rule (3), the answer at node n is decided based on whether n with a store statement or a definition statement is a divergent branch node. If n is a divergent branch node, whether m is control-dependent on n is also checked. If m is control-dependent on n, the answer is (*true*, $\{n\}$) because the hoisting from m to n contributes to reducing branch divergence and making more expressions available. If m is not control-dependent on n, the hoisting from m to n is ineffective, so that the answer is (*false*, $\emptyset$). These correspond to the rule (3)(i). On the other hand, If n is not a divergent branch node, the hoisting from m to n is also ineffective, so that the answer is (*false*, $\emptyset$). This corresponds to the rule (3)(ii).

As shown in the rule (4), if n is a divergent branch node and m is control-dependent on n, the query is propagated further to the predecessors of n after m is replaced with n to check whether e can be hoisted to an earlier node than n. This means that the rule checks whether n is control-dependent on an earlier node that is a divergent branch node. If the result of the further propagation is (*true*, x), e can be hoisted to the nodes in x. Conversely, if it is (*false*, x), e cannot be hoisted to the nodes in x. However, e can be speculatively hoisted to n, so that the answer at n is (*true*, $\{n\}$). These correspond to the rule (4)(i). On the other hand, if m is not control-dependent on n, the hoisting from m to n is ineffective. Thus, whether m is control-dependent on an earlier node that is a divergent branch node is checked through the further propagation. The answer at n depends on the result of the propagation, corresponding to the rule (4)(ii).

As shown in the rule (5), if n is a non-divergent branch node, the rule needs to check whether n is down-safe. If n is down-safe, e can be hoisted from m to n safely, so that the rule also checks whether m is control-dependent on n. If m is control-dependent on n, m is replaced with n to check whether n is control-dependent on an earlier node that is a divergent branch node, and then the query is propagated further to the predecessors of n. If m is not control-dependent on n, the query is propagated further without replacing m with n to check the control dependence between m and an earlier node than n. These correspond to the rule (5)(i). On the other hand, if n is not down-safe, e cannot be hoisted to n safely, so that the answer is (*false*, $\emptyset$), corresponding to the rule (5)(ii).

A program is transformed based on the result of propagation of a query about speculation after ESRQP finds that an expression e which originates a query about availability is not redundant. The pseudocode of the algorithm of propagation of a query about speculation is shown in Appendix 4.5.2.

Consider the application of SSRQP to the CFG in Figure 4.13. In the figure, the shaded nodes 2 and 4 cause branch divergence. First, SSRQP visits node 5 and propagates a query about availability for the expression $A[i+1]$. The result of the propagation is *false*, so that as illustrated in Figure 4.13a, SSRQP propagates a query about spec-

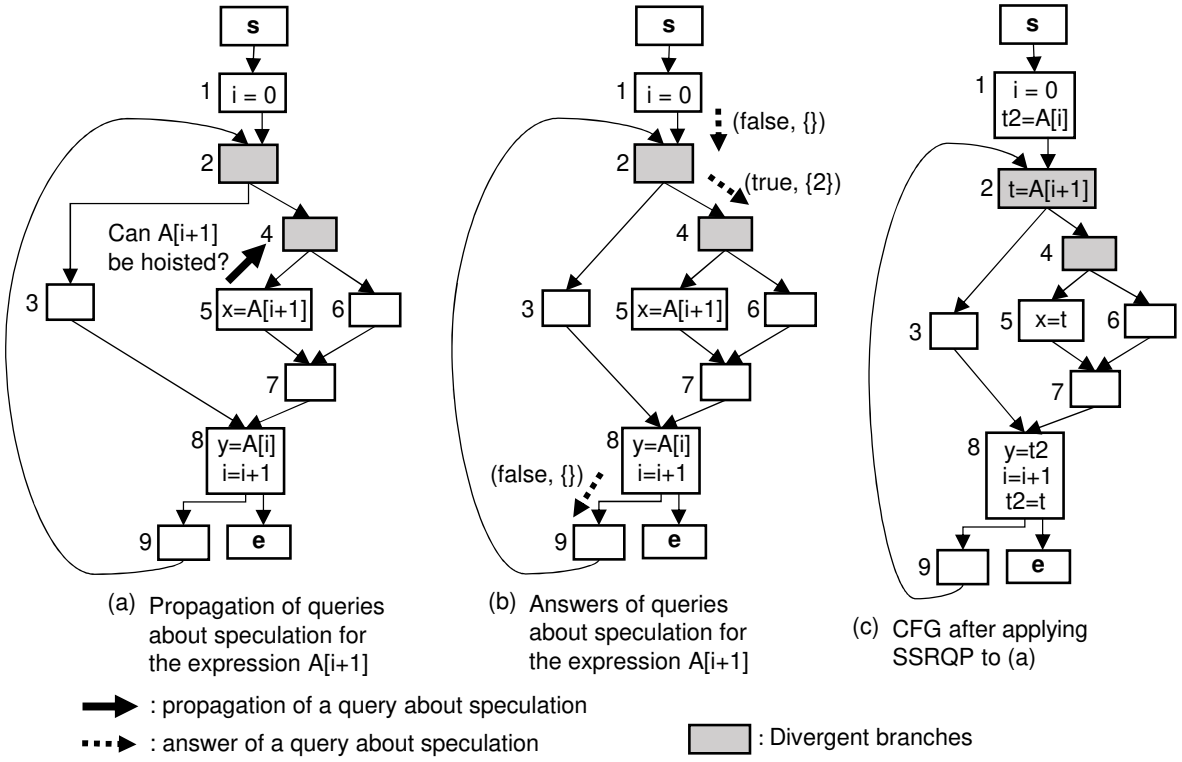


Figure 4.13: Example of application of SSRQP

ulation ($A[i+1]$, 5, 4) to the predecessor of node 5 in order to check whether it can speculatively hoist the expression to an earlier node than node 5. When the query visits node 4, the rule (4)(i) is applied because node 4 is a divergent branch node and does not have a store statement or a definition statement, so that the query ($A[i+1]$, 4, 2) is propagated to node 2. Next, at node 2, the rule (4)(i) is applied as well as node 4, so that the query ($A[i+1]$, 2, 1) is propagated to node 1, and the query ($A[i+1]$, 2, 9) is propagated to node 9. At node 1, the rule (3)(ii) is applied because the statement $i = 0$ is a definition statement of $A[i+1]$ and node 1 is not a divergent branch node. Thus, the former query gives the answer (*false*, $\emptyset$) at node 1 as shown in Figure 4.13b. On the other hand, at node 9, the rule (6) is applied, so that the query ($A[i+1]$, 2, 8) is propagated to node 8. At node 8, the query gives the answer (*false*, $\emptyset$) because the rule (3)(ii) is applied. Consequently, at node 2, both answers obtained from the predecessors are (*false*, $\emptyset$) as illustrated in Figure 4.13b. In this case, the rule (4)(i)(b) is applied, so that the answer (*true*, $\{2\}$) is returned toward node 5. In consequence, the originating node 5 obtains the answer (*true*, $\{2\}$), so that SSRQP hoists $A[i+1]$ from node 5 to node 2 speculatively. Second, SSRQP propagates a query about availability for the expression $A[i]$ at node 8. As a result, it can remove the inter-iteration redundancy between $A[i+1]$ and $A[i]$ because of the speculative code motion of $A[i+1]$ from node 5 to node 2. Finally, it obtains the CFG in Figure 4.13c.

Remember that SRQP increases branch divergence because it inserts $A[i+1]$ into nodes 3 and 6 as shown in Figure 4.11f. On the other hand, as illustrated in Figure 4.13c, SSRQP hoists the expression speculatively, so that it does not increase branch divergence and furthermore makes $A[i+1]$ available across the loop iteration at node 8.

Computational Complexity

As described above, SSRQP is realized by extending SRQP in terms of a CDRND and propagation of a query about speculation. SRQP is based on question propagation [88] whose worst-case time is $O(N^3)$, where N is the size of an input program. To utilize a CDRND, a control dependency needs to be computed, and it can be computed in time $O(N^2)$ [26]. Additionally, propagation of a query about speculation is also based on question propagation, so that its worst-case time is $O(N^3)$. Therefore, SSRQP is accomplished in time $O(N^3)$.

4.3.4 Evaluation

To evaluate the effectiveness of the proposed method, it is compared with traditional approaches about execution efficiency for three benchmarks. The proposed method has been implemented on the open-source software Ocelot CUDA compiler [23]. The Ocelot is a backend for PTX [82] corresponding to a GPU assembly code and also works as a PTX optimizer. Furthermore, the divergence analysis [19] implemented in Ocelot is used to identify divergent branches. The descriptions of experimental environments are as follows:

- Environment 1
  - OS: Ubuntu 16.04 LTS,
  - CPU: Intel Core i7-4770K,
  - GPU: Geforce GTX TITAN Black, and
  - CUDA Toolkit 5.0.
- Environment 2
  - OS: Ubuntu 18.04 LTS,
  - CPU: Intel Core i9-9900K,
  - GPU: NVIDIA TITAN RTX, and
  - CUDA Toolkit 11.1.

In the experiments, the proposed method and the related methods, SRQP, ESRQP, and SSRQP_S, were implemented. SRQP neither considers a CDRND nor applies propagation of a query about speculation. ESRQP considers a CDRND without applying propagation of a query about speculation. SSRQP_S considers a CDRND and applies propagation of a query about speculation to move an expression speculatively without considering branch divergence. The execution time of object codes is compared for the three benchmarks, Rodinia [14], Treelogy [39], and NVIDIA SDK sample code [77]. For *SobelFilter* and *bilateralFilter* in NVIDIA SDK sample code, the memory access pattern is changed, and execution efficiency is compared before and after the change. The change will be described later. Each program was executed 10 times. The average of all execution time is reported. Execution time is measured by the CUDA profiler *nvprof* [81].

First, these methods are applied to the PTX code generated by the NVIDIA CUDA compiler, *nvcc*. Figures 4.14 and 4.16 show the results of the experiments in the environment 1 for Rodinia and Treelogy benchmark, respectively. Figures 4.15 and 4.17 show the experimental results in the environment 2 for these benchmarks, respectively. The baseline for comparison is the execution time of the object code generated by nvcc with the optimization option O3. In the rest of this section, this baseline is called *O3*. In the figures, *SRQP*, *ESRQP*, *SSRQP_S*, and *SSRQP* respectively represent the execution time when applying SRQP, ESRQP, SSRQP_S, and SSRQP to the PTX code used by the baseline *O3*. Each result is normalized by *O3*.

As illustrated in Figure 4.14, the proposed method SSRQP improved performance for five programs in Rodinia benchmark in the environment 1: *b+tree*,*bfs*, *huffman*, *lavaMD*, and *leukocyte*. For the program of *b+tree*, SRQP, ESRQP, and SSRQP improved the execution efficiency by 21%, 21%, 22%, respectively. These mothods moved loop-invariant expressions out of the loop. SSRQP could move more loop-invariant expressions out of the loop than SRQP and ESRQP did through hoisting some expressions speculatively. On the other hand, SSRQP_S achieved less efficiency than other methods did because it hoisted many expressions without considering branch divergence and introduced new expressions into some paths. For the program of *bfs*, SSRQP_S and SSRQP improved the execution efficiency by 19.6%, 22.5%, respectively through moving loop-invariant expressions out of the loop. SRQP and ESRQP could not have an opportunity to transform, so that they could not get performance gain. For the program of *huffman*, all methods improved the execution efficiency by 7% through removing intra-iteration redundancy. For the program of *lavaMD*, SRQP, ESRQP, and SSRQP improved the execution efficiency by about 1.5% through removing intra-iteration redundancy. SSRQP_S hoisted many expressions speculatively through the propagation of a query about speculation, but it could not improve the execution efficiency. For the program of *leukocyte*, SRQP and ESRQP could improve the execution efficiency by 1% through removing intra-iteration redundancy. SSRQP_S could remove a lot of intra-iteration redundancy through speculative code motion without considering branch divergence, resulting in the improvement of the efficiency by 4%. Since SSRQP performs speculative code motion that considers branch divergence, it could remove less redundancy than SSRQP_S, resulting in the improvement of the efficiency by 3%. For the programs of *backprop* and *srad*, SSRQP_S could improve the execution efficiency while the other methods could not transform the programs. On the other hand, for the programs of *hotspot*, *lud*, and *particlefilter*, SSRQP_S reduced the execution efficiency of them. This is because SSRQP_S hoists expressions speculatively for not only divergent branches but also non-divergent branches. For the programs of *heartwall*, *kmeans*, *myocyte*, *nw*, and *pathfinder*, SSRQP and other methods had an opportunity to transform, but they could not improve the execution efficiency of those programs. For the other programs, all methods did not have an opportunity to transform, so that they could not get performance gain.

As shown in Figure 4.15, the result in the environment 2 showed the almost same tendency as the one in the environment 1. However, for the program of *b+tree*, SSRQP_S improved more execution efficiency than SRQP and ESRQP did. In addition, SSRQP could improve the most efficiency of the four methods. SRQP, ESRQP, SSRQP_S, and SSRQP improved the efficiency by 58.5%, 58.5%, 84.9%, 88%, respectively.

As shown in Figure 4.16, SSRQP improved performance for three programs *barnshut*,

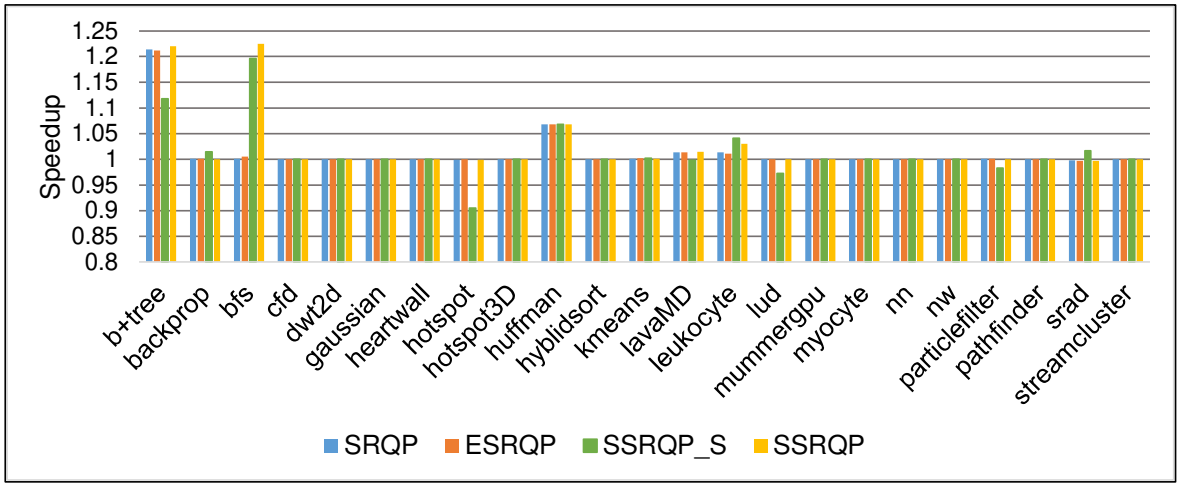


Figure 4.14: Comparison of execution speed of Rodinia benchmark in the environment 1. Each result is normalized by the baseline *O3*.

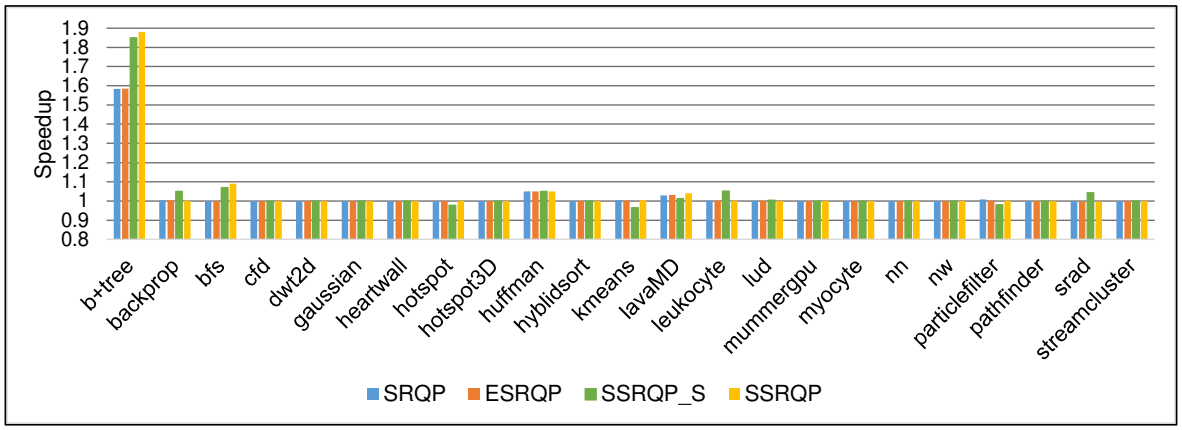


Figure 4.15: Comparison of execution speed of Rodinia benchmark in the environment 2. Each result is normalized by the baseline *O3*.

kmeans, and *knearestneighbor* in Treelogy benchmark in the environment 1. SSRQP_S and SSRQP improved the execution efficiency of them by about 10%, 40%, and 3%, respectively while SRQP and ESRQP could not transform the programs. This is because the speculative code motion that SSRQP_S and SSRQP performed made some expressions totally redundant.

As illustrated in Figure 4.17, in the environment 2, all methods could not improve performance for five programs in Treelogy benchmark: *barnshut*, *fastmultipole*, *knearestneighbor*, *nearestneighbor*, and *pointcorr*. For the program of *kmeans*, SSRQP_S and SSRQP improved the execution efficiency by about 10%, but it was less efficient than ones in the environment 1.

In these two benchmarks, geometric mean speedup in the environment 1 for SRQP, ESRQP, SSRQP_S, and SSRQP is 1.011, 1.011, 1.026, and 1.034, respectively, and one in the environment 2 is 1.02, 1.02, 1.032, and 1.032, respectively. These methods could not improve performance of many programs because benchmark programs are well known

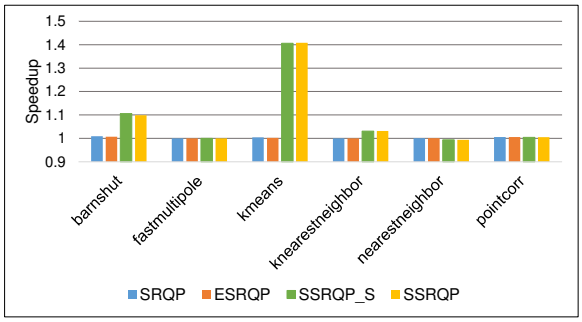


Figure 4.16: Comparison of execution speed of Treelogy benchmark in the environment 1. Each result is normalized by the baseline *O3*.

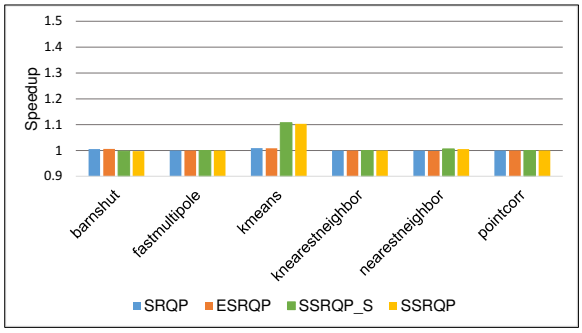


Figure 4.17: Comparison of execution speed of Treelogy benchmark in the environment 2. Each result is normalized by the baseline *O3*.

and have already been heavily optimized. Nevertheless, the proposed method could get great performance gains in three programs (*b+tree* and *bfs* of Rodinia benchmark, and *kmeans* of Treelogy benchmark). The three programs include the graph traversal algorithm that has many memory operations in a loop. Thus, the proposed method was significantly effective for them. On the other hand, ESRQP obtained the same performance gain as SRQP. Moreover, although SSRQP_S obtained performance gains a little greater than SSRQP in some cases, it obtained much less execution efficiency than SSRQP in some cases. Therefore, SSRQP_S is not practical. From the above, the proposed method, which considers a CDRND and speculative code motion based on branch divergence, is the best of these methods.

As mentioned above, for *SobelFilter* and *bilateralFilter* in NVIDIA SDK sample code, the memory access pattern was changed, and experiments are conducted with them. The access pattern of the original programs was coalesced, so that there were redundant memory accesses among threads, which could not be removed. Therefore, the memory access pattern was changed, which transformed redundant accesses among threads into ones within each thread. Figures 4.18a and 4.18b show the pattern of the memory accesses before and after the modification. The figure shows the 2nd to 10th elements of an array are calculated on three threads with three iterations of a loop. Assume that calculating the i-th element needs to access the i-1th and i+1th elements. In Figure 4.18a, thread `t1` calculates the 2nd, 5th, and 8th elements, `t2` does the 3rd, 6th, and 9th ones, and `t3` does the 4th, 7th, and 10th ones for each loop iteration. There are no redundant memory accesses within each thread, but there are redundant ones among threads. That is, at the first iteration of the loop, since thread `t1` accesses the 1st, 2nd, and 3rd elements, and `t2` does the 2nd, 3rd and 4th ones, the accesses to the 2nd and 3rd elements are redundant between these two threads. As well, since thread `t3` accesses the 3rd, 4th, and 5th elements, the accesses to the 3rd and 4th ones are redundant between `t2` and `t3`. This type of redundancy exists at each iteration, and it is difficult to remove the redundancy. On the other hand, in Figure 4.18b, in the loop iterations, thread `t1` calculates the 2nd, 3rd, and 4th elements, `t2` does the 5th, 6th and 7th ones, and `t3` does the 8th, 9th and 10th ones, respectively. In this case, there is no redundancy among threads in the same iteration. Instead, there is redundancy across iterations within each thread. Since thread `t1` accesses the 1st, 2nd, and 3rd elements at the 1st iteration of the loop, and it accesses the 2nd, 3rd and 4th

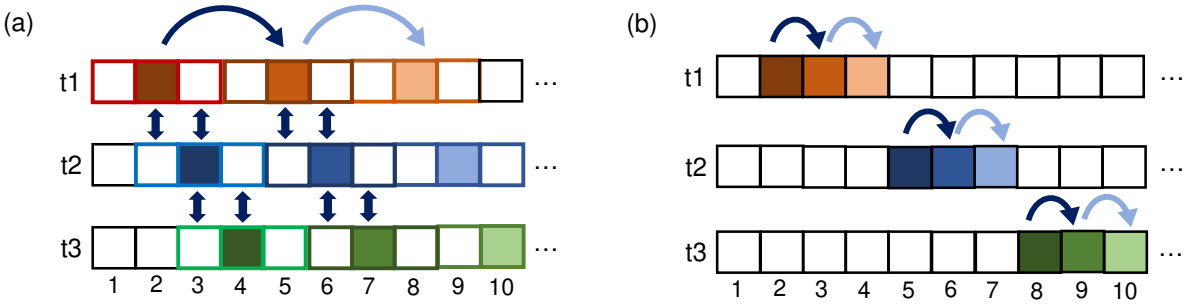


Figure 4.18: (a) Coalesced memory accesses. There are redundant memory accesses among threads in each iteration, (b) Access pattern after the modification. There are redundant memory accesses within threads in across iterations.

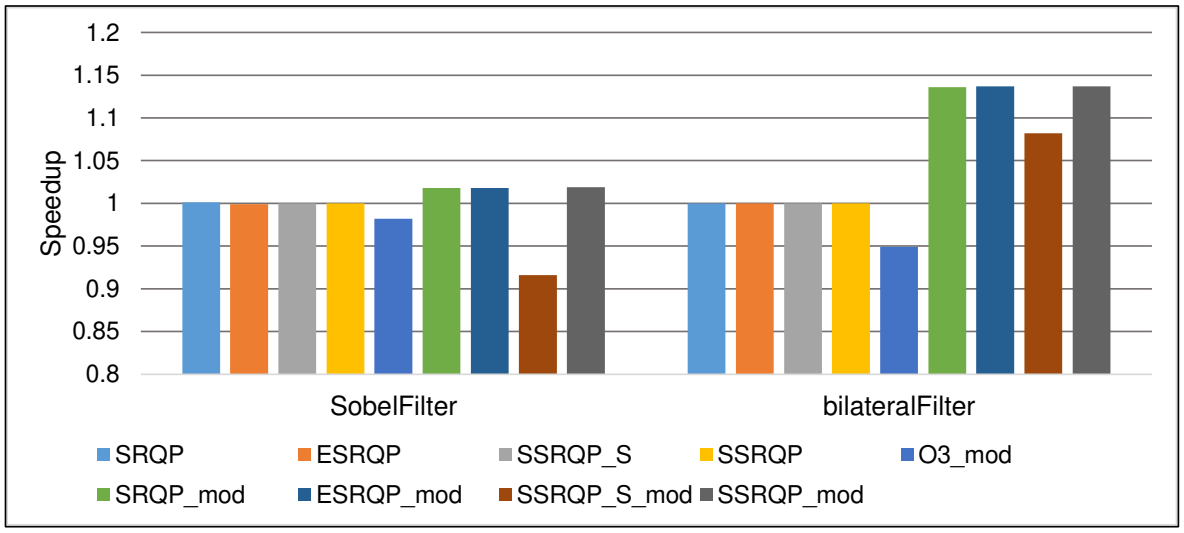


Figure 4.19: Comparison of execution speed of *SobelFilter* and *bilateralFilter* in Nvidia SDK in the environment 1. Each result is normalized by the baseline *O3*.

ones at the 2nd iteration, the accesses to the 2nd and 3rd ones are redundant in this iteration. This type of redundancy exists in each thread, and it can be removed by scalar replacement.

The execution efficiency before and after changing the memory access pattern as described above was compared. Figures 4.19 and 4.20 show the experimental results of *SobelFilter* and *bilateralFilter* in the environments 1 and 2, respectively. Each result is normalized by *O3*, which represents the execution time of the object code before the modification of the access pattern with the optimization option O3. *SRQP*, *ESRQP*, *SSRQP_S*, and *SSRQP* represent the same ones as in Figures 4.14, 4.15, 4.16, and 4.17. *O3_mod* represents the execution time of the object code after the modification of the access pattern with the optimization option O3. *SRQP_mod*, *ESRQP_mod*, *SSRQP_S_mod*, and *SSRQP_mod* represent the execution time when respectively applying SRQP, ESRQP, SSRQP_S, and SSRQP to the PTX code of *O3_mod*.

For the program of *SobelFilter*, SRQP, ESRQP, SSRQP_S, and SSRQP could not transform and improve the PTX code of O3. In the environment 1, as shown in Fig-

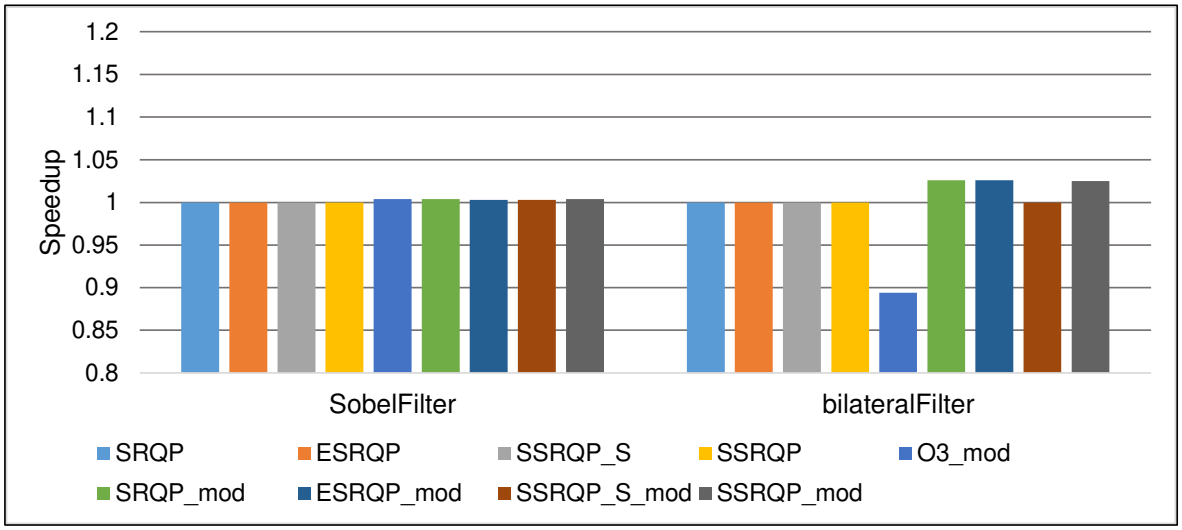


Figure 4.20: Comparison of execution speed of *SobelFilter* and *bilateralFilter* in Nvidia SDK in the environment 2. Each result is normalized by the baseline *O3*.

ure 4.19, O3_mod decreased the execution efficiency by 2% because the change of the memory access pattern prevents the memory coalescing optimization. SRQP_mod, ESRQP_mod, SSRQP_S_mod and SSRQP_mod could remove inter-iteration redundancy. However, SSRQP_S_mod hoisted expressions speculatively without considering branch divergence, and the number of statements to be executed increased, so that it decreased the execution efficiency by 9% in the environment 1. SRQP_mod, ESRQP_mod and SSRQP_mod could improve the efficiency by 2% in the environment 1 because SRQP_mod and ESRQP_mod do not perform speculative code motion, and SSRQP_mod suppresses speculative code motion for non-divergent branches. In the environment 2, as shown in Figure 4.20, all methods could not improve the efficiency after modifying the memory access pattern. O3_mod and SSRQP_S_mod did not decrease the efficiency. The modification that prevents the memory coalescing optimization and speculative code motion did not affect the performance of this program in the environment 2. For the program of *bilateralFilter*, as well as *SobelFilter*, SRQP, ESRQP, SSRQP_S and SSRQP could not transform and improve the PTX code of O3. In the environment 1, for the same reason as *SobelFilter*, O3_mod decreased the execution efficiency by 5% as shown in Figure 4.19. SRQP_mod, ESRQP_mod, SSRQP_S_mod and SSRQP_mod could remove inter-iteration redundancy and improve the efficiency by 14%, 14%, 8%, and 14%, respectively. SSRQP_S_mod was less efficient than other methods for the same reason as *SobelFilter*. In the environment 2, as shown in Figure 4.20, O3_mod decreased the execution efficiency by 11%. SRQP_mod, ESRQP_mod, and SSRQP_mod could improve the efficiency by 3%, and SSRQP_S_mod could not improve the efficiency.

Most GPU programs access memory through the pattern shown in Figure 4.18a. This pattern coalesces memory access, so that it has few memory transactions. In addition, the pattern has a high rate of cache hit because the spatial locality of the pattern is high at the same iteration. In general, the strided memory access shown in Figure 4.18b is less efficient than the coalesced access shown in Figure 4.18a. However, since the access to a register is faster than the memory access, the execution speed of a program can be faster if the data is stored in a register as many as possible. The results

of this experiment show that the proposed method can improve the execution efficiency of GPU programs by changing the pattern of the memory access and applying scalar replacement to increase accesses through registers.

4.4 Summary

SSCM and SSRQP are introduced to reduce branch divergence and improve performance of a GPU kernel. SSCM is designed by extending the dataflow equations used in traditional PRE. The extension allows a partially anticipated expression at a divergent branch to be considered as fully anticipated, enabling speculative code motion without penalty by utilizing the property of branch divergence. Based on the code motion, SSCM makes more expressions redundant than traditional method, so that it can improve performance further. SSRQP is proposed by making SRQP consider branch divergence when propagating a query. As well as SSCM, SSRQP performs speculative code motion without decreasing performance at a divergent branch, eliminating more redundancy and improving performance. Moreover, SSRQP introduces a CDRND as a condition in inserting a compensation code. Based on the CDRND, SSRQP suppresses inserting an expression into the node where branch divergence may increase. Therefore, SSRQP does not increase branch divergence in addition to eliminating redundant memory accesses and reducing the divergence through speculative code motion.

4.5 Appendix

4.5.1 Algorithm of Extended SRQP Based on a CDRND

Algorithm 1.1 Extended SRQP based on a CDRND: Propagate

1: **function** Propagate(n, q)
2: let $isDownSafe \coloneqq antqp(n, q)$
3: let $N_f \coloneqq \emptyset$
4: **for all** $p \in pred(n)$
5: let $(isAvail_p, isReal_p) \coloneqq Local(p, q)$
6: **if** $\overline{isAvail_p}$ **then add** p **to** N_f
7: let $isAvail \coloneqq \prod_{p \in pred(n)} isAvail_p$
8: let $isReal \coloneqq \bigcup_{p \in pred(n)} isReal_p$
9: **if** both *true* and *false* are in $isAvail_p$ **then**
10: **if** $N_f \subseteq CDRND$ **then**
11: **if** $isAvail \vee (isDownSafe \wedge isReal)$ **then**
12: **add** N_f **to** *insert_dst*
13: $isAvail \coloneqq true$
14: **if** $isAvail \vee (isDownSafe \wedge isReal)$ **then return** ($isAvail$, $isReal$)
15: **else return** (*false*, *false*)

Algorithm 1.2 Extended SRQP based on a CDRND: Local

```
1: function Local(n, q)
2:   if n = s then return (false, false)
3:   if answer[n] ≠ ⊥ then return answer[n]
4:   if visited[n] > 1 then return (false, false)
5:   if query[n] = q then return (true, false)
6:   query[n] := q
7:   visited[n]++
8:   for i=instSize(n) to 0
9:     let inst := getInstruction(n,i)
10:    if mayAlias(q, inst) then
11:      answer[n] := (false, false)
12:      return (false, false)
13:    if isSameVal(q, inst) then
14:      answer[n] := (true, true)
15:      return (true, true)
16:    if isDefVal(q, inst) ∧ visited[n] = 1 then updateQuery(q, inst)
17:  let rlt := Propagate(n, q)
18:  answer[n] := rlt
19:  return rlt
```

4.5.2 Algorithm of Propagation of a Query About Speculation

Algorithm 2.1 Propagation of a query about speculation: PropagateSpeculativeQuery

```
1: function PropagateSpeculativeQuery(e, m, n)
2:   for all p ∈ pred(n)
3:     let (canHoist_p, Node_p) := Local_Spec(e,m,p)
4:     if canHoist_p is false then add p to N_p
5:   let canHoist := ∏_{p∈pred(n)} canHoist_p
6:   let Node := ⋃_{p∈pred(n)} Node_p
7:   if (both true and false are in canHoist_p) ∧ (N_p ⊆ CDRND) ∧ (n is down-safe)
     then
8:     Node := Node ∪ N_p
9:     canHoist := true
10:  if canHoist then return (true, Node)
11:  else return (false, ∅)
```

Algorithm 2.2 Propagation of a query about speculation: Local_Spec

```
1: function Local_Spec(e, m, n)
2:   if n = s then return (false, ∅)
3:   if answer[n] ≠ ⊥ then return answer[n]
4:   if query[n] = q then return (true, ∅)
5:   query[n] := (e, m)
6:   let rlt := ⊥
7:   if containsMayAlias(e,n) ∨ containsDefVal(e,n) then
8:     if n is a divergent branch node then
9:       if m is control-dependent on n then rlt := (true, {n})
10:      else rlt := (false, ∅)
11:    else rlt := (false, ∅)
12:  else
13:    if n is a divergent branch node then
14:      if m is control-dependent on n then
15:        m := n
16:        let (canHoist, Node) := PropagateSpeculativeQuery(e, m, n)
17:        if canHoist is false then (canHoist, Node) := (true, {n})
18:        rlt := (canHoist, Node)
19:      else rlt := PropagateSpeculativeQuery(e, m, n)
20:    else if n is a non-divergent branch node then
21:      if n is down-safe then
22:        if m is control-dependent on n then
23:          m := n
24:          rlt := PropagateSpeculativeQuery(e, m, n)
25:        else rlt := PropagateSpeculativeQuery(e, m, n)
26:      else rlt := (false, ∅)
27:    else rlt := PropagateSpeculativeQuery(e, m, n)
28:  answer[n] := rlt
29:  return rlt
```

Chapter 5

Kernel Fusion Based on Code Motion

In this chapter, KFCM is introduced, which is the code motion based kernel fusion technique that can fuse more kernels than traditional methods.

The rest of this chapter is organized as follows. Section 5.1 describes a traditional kernel fusion based method and its problem. Section 5.2 presents the proposed method, KFCM. In Section 5.3, the experimental results of KFCM are presented. Finally, Section 5.4 gives the summary.

5.1 Kernel Fusion

Kernel fusion based methods [104, 84, 85, 28, 27, 102, 103, 109, 67] achieve performance improvement by combining two or more kernels into one. Because the fusion decreases the number of kernels, it reduces kernel launch overhead, improving performance. However, the major potential performance gain of kernel fusion comes from increasing opportunities for optimizations by enlarging each kernel. For example, kernel fusion can effectively reduce expensive data communication with global memory. Typical kernels have input/output data, which must be stored in the global memory with the highest communication cost of memories in a GPU. Kernel fusion enables kernels to share some data on other memories at a lower cost, reducing traffic with global memory.

Figure 5.1 shows a simple fusion scenario of two CUDA [76] kernels. As shown in Figure 5.1a, the kernels `Kernel1` and `Kernel2` first read input data from global memory and write output data to global memory at the end. On the host side, the function `main` serially launches the kernels. Here, the output of `Kernel1` is used by `Kernel2` as the input. As shown in Figure 5.1b, traditional kernel fusion based methods combine `Kernel1` and `Kernel2` into `fusedKernel`. Once they are fused, `out1[threadIdx.x]` in the right-hand side of the statement at line 7 can be replaced with `rlt1`. This will enable the fusion to eliminate a load statement from global memory, as indicated by a copy assignment at line 8. Furthermore, the statement that writes to `out1` at line 6 can be eliminated if a compiler knows that `d_out1` is never used in the rest of the function `main`. As a result, the fusion decreases the number of kernel launches from two to one, improving the performance.

Thus, traditional kernel fusion based methods are effective for various GPU appli-

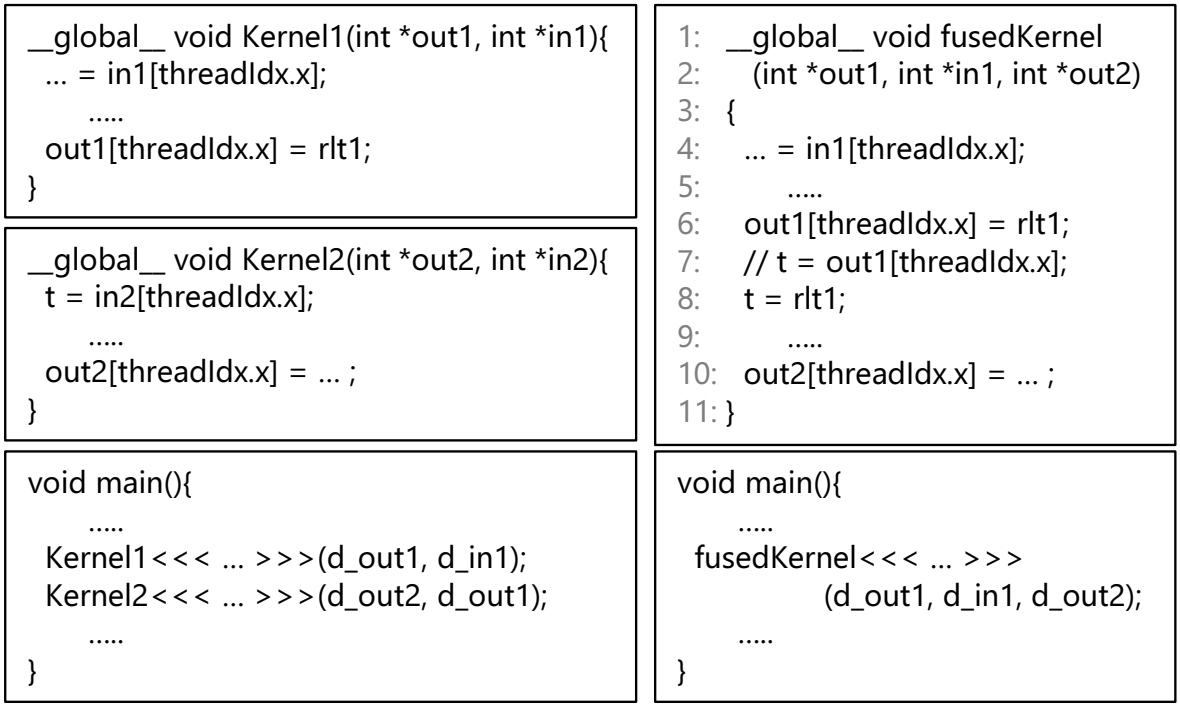


Figure 5.1: Example of applying a traditional kernel fusion based method to a simple CUDA code.

cations but miss fusion opportunities in cases as shown in Figure 5.2a. As shown in the figure, `Kernel1` and `Kernel2` cannot simply be fused using traditional methods because `Kernel1` is not always executed before `Kernel2`. In such cases, the proposed method assumes that the launch statement of `Kernel2` has been hoisted to *then* and *else* sides of the if-statement. At this time, `Kernel1` and `Kernel2` can be fused in the *then* side by combining them into `fusedKernel`, as shown in Figure 5.2b. The proposed method achieves the kernel fusion based on code motion, as explained in Section 5.2.

5.2 Kernel Fusion Based on Code Motion

This section describes the details of KFCM. KFCM comprises two fusion phases, and each phase consists of a fusibility analysis and transformation based on the result. The first fusion phase performs a backward fusibility analysis, which hoists each kernel launch statement to the earliest program points without lengthening any execution path. The phase checks whether the hoisted launch statement is adjacent to some other launch statements. Once adjacent ones are detected, they are fused as a single kernel. In contrast, the second fusion phase performs a forward fusibility analysis, which sinks each kernel launch statement to the latest program points without lengthening any execution path. The phase checks adjacency as well as the first phase and fuses adjacent launch statements. The second phase is performed only when the first phase does not result in any fusion. These fusibility analyses are implemented as dataflow

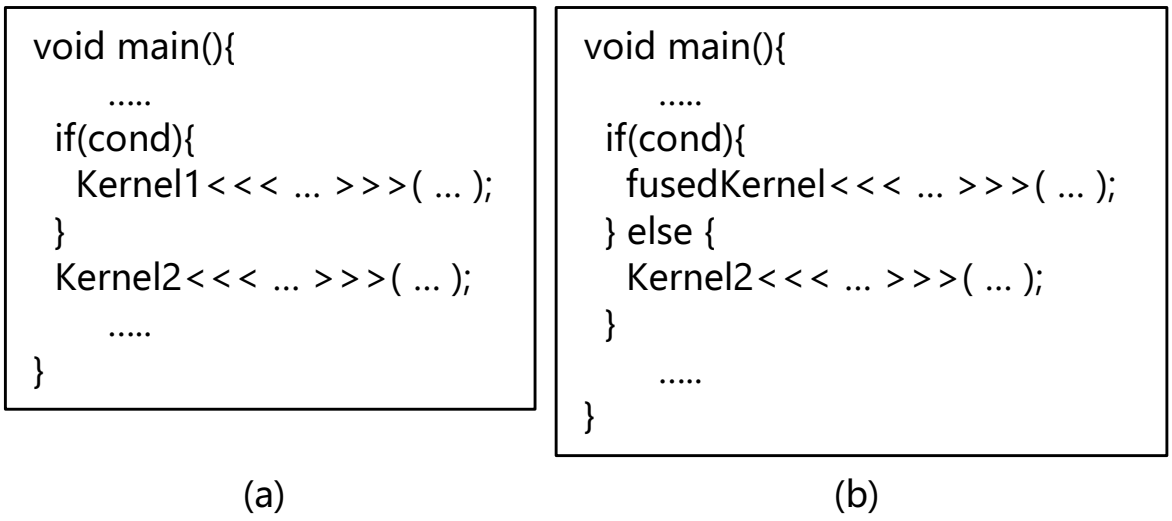


Figure 5.2: (a) Simple code in which traditional kernel fusion based methods cannot fuse the kernels `Kernel1` and `Kernel2`; (b) code after applying the proposed method to the program shown in (a). The proposed method can combine `Kernel1` and `Kernel2` into `fusedKernel` on the true-path of a conditional branch and insert `Kernel2` as a compensation code on the false-path.

analyses. Their definitions are given by dataflow equations in Sections 5.2.1 and 5.2.2, respectively. The fusion phases are applied to each node in a CFG until there is no change. The behaviors of the overall algorithm of KFCM are illustrated in Section 5.2.3.

In the rest of this section, the CFG in Figure 5.3a is used as a motivation. As shown in the figure, `K1`, `K2`, `K3`, and `K4` are kernels, of which kernel launch statements are represented by kls_{K1}, kls_{K2}, kls_{K3}, and kls_{K4}. KFCM eventually transforms the CFG given in Figure 5.3a into the one given in Figure 5.3b.

5.2.1 Backward Fusion Phase

At the first phase of the proposed kernel fusion, the proposed method applies the *backward kernel fusion* (BKF), which determines fusible kernels backwardly and fuses them, to a program. BKF is based on a dataflow analysis similar to PRE, which is a code optimization that eliminates partially redundant expressions [54, 55]. BKF solves a dataflow equation for each kernel launch statement kls_K in a program and transforms the program based on the result. The goal of this phase is to hoist kls_K and fuse the kernel K with other kernels existing at earlier program points. In the hoisting process, some single kernel launch statements may be inserted as a compensation code to hold the program semantics if necessary. This phase consists of the following steps:

(1) Computing local predicates;

(2) Computing down-safety points;

(3) Finding program points where a kernel K is fusible with other kernels by hoisting kls_K;

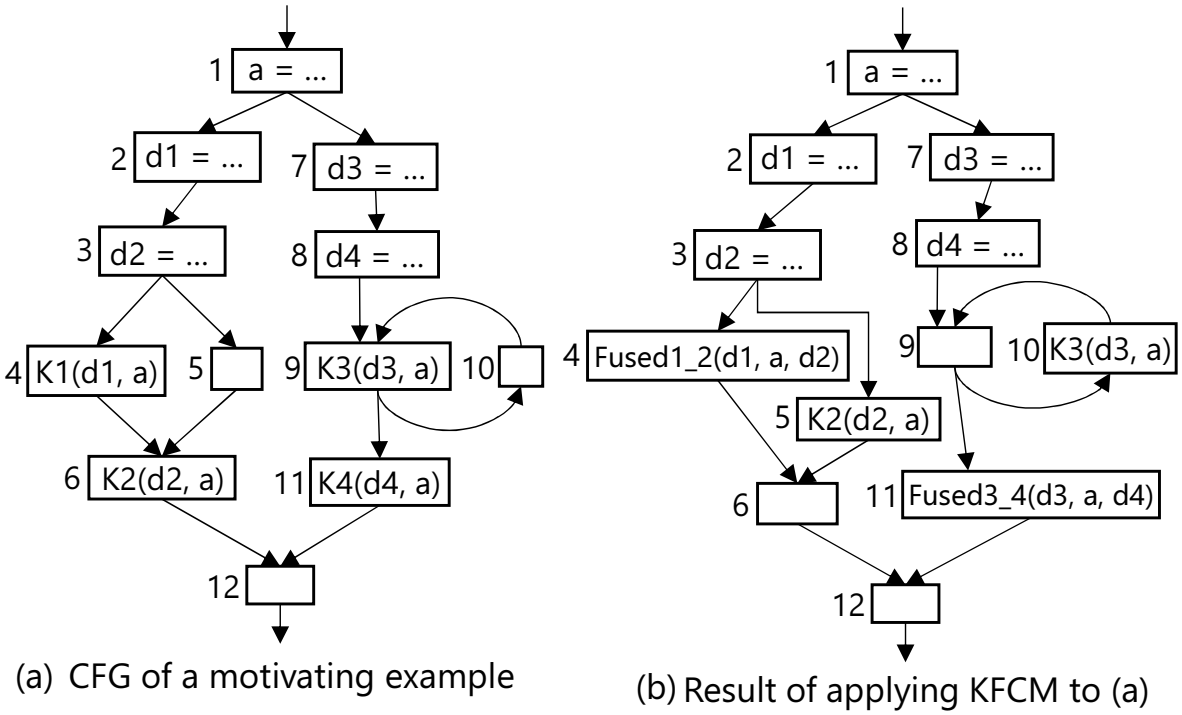


Figure 5.3: (a) CFG used as a motivation; (b) result of applying KFCM to the CFG in (a).

(4) Computing insertion points of kls_K; and

(5) Hoisting kls_K and fusing kernels.

The details of the above steps are described as follows.

Computing Local Predicates

First, BKF computes the local predicates *Launch* and *Transp* for a kernel launch statement kls_K. Each predicate is represented as follows:

- $Launch(n)_{kls_K}$: n has kls_K.
- $Transp(n)_{kls_K}$: n has neither a modification statement of kls_K nor the other kernel launch statements.

For example, consider `K1(d1,a)` in Figure 5.3a. The predicate $\text{Launch}(4)_{kls_{K1}}$ is *true*, and $\text{Launch}_{kls_{K1}}$ for the other nodes are *false*. Moreover, the predicates $\text{Transp}_{kls_{K1}}$ for the nodes {3, 4, 5, 7, 8, 10, 12} are *true*. However, $\text{Transp}_{kls_{K1}}$ for the nodes {1, 2, 6, 9, 11} are *false* because there are modification statements in the nodes {1, 2} and the other kernel launch statements in the nodes {6, 9, 11}. Thus, the Transp property of kls_K holds the order between kernels by killing Launch of the other kernel launch statements.

$$\begin{cases} NdSafe(n)_{kls} = Launch(n)_{kls} \vee \{Transp(n)_{kls} \wedge XdSafe(n)_{kls}\} \\ XdSafe(n)_{kls} = \begin{cases} false & (if\ n = \boldsymbol{e}) \\ \prod_{s \in succ(n)} NdSafe(s)_{kls} & (otherwise) \end{cases} \end{cases}$$

Figure 5.4: Dataflow equations for a down-safety analysis.

Computing Down-Safety Points

The local predicates defined in the previous section are used to compute down-safety program points for kls_K. The down-safety program points indicate the region where kls_K can be hoisted without lengthening any execution path. BKF seeks the program points where to hoist kls_K in the region. The down-safety is defined by the predicates *NdSafe* and *XdSafe* by solving the dataflow equations shown in Figure 5.4. Each predicate for kls_K represents the following.

- $NdSafe(n)_{kls_K}$: the entry of n is down-safe with respect to kls_K.
- $XdSafe(n)_{kls_K}$: the exit of n is down-safe with respect to kls_K.

Figure 5.5a shows the result of the down-safety analysis for the kernel `K2` at node 6. As shown in the figure, NdSafe$(5)_{kls_{K2}}$, NdSafe$(6)_{kls_{K2}}$, XdSafe$(4)_{kls_{K2}}$, and XdSafe$(5)_{kls_{K2}}$ are *true*. On the other hand, NdSafe$(4)_{kls_{K2}}$ is *false* because kls_{K2} is killed by kls_{K1} at node 4.

Finding Kernel Fusion Points

Once down-safety points for a kernel launch statement kls_K are computed, BKF checks where a kernel K can be fused with other kernels. Assume that kls_K and kls_{K2} are launch statements for different kernels. When the exit of a node n is down-safe with respect to kls_K and n has kls_{K2}, kernels K and *K2* are fusible at the exit of n because kls_K can be hoisted immediately after kls_{K2}. To characterize the program points, BKF introduces the predicate *XFusible* shown in Figure 5.6. XFusible(n, kls_K, kls_{K2}) indicates that a kernel K is fusible with another kernel *K2* at the exit of a node n by hoisting kls_K to the exit of n.

For example, in Figure 5.5a, the kernel `K2` is fusible with `K1` at the exit of node 4 because XFusible(4, kls_{K2}, kls_{K1}) is *true*. This means that the exit of node 4 is down-safe with respect to `K2(d2,a)`, and node 4 has `K1(d1,a)`. However, fusing the two kernels requires hoisting `K2(d2,a)` to the exit of node 4. This hoisting must insert a compensation code to preserve the semantics of the program. The insertion points are computed as follows.

Computing Insertion Points

This section shows how BKF decides insertion points of a kernel launch statement kls_K as a compensation code. The process of deciding the insertion points is similar to LCM

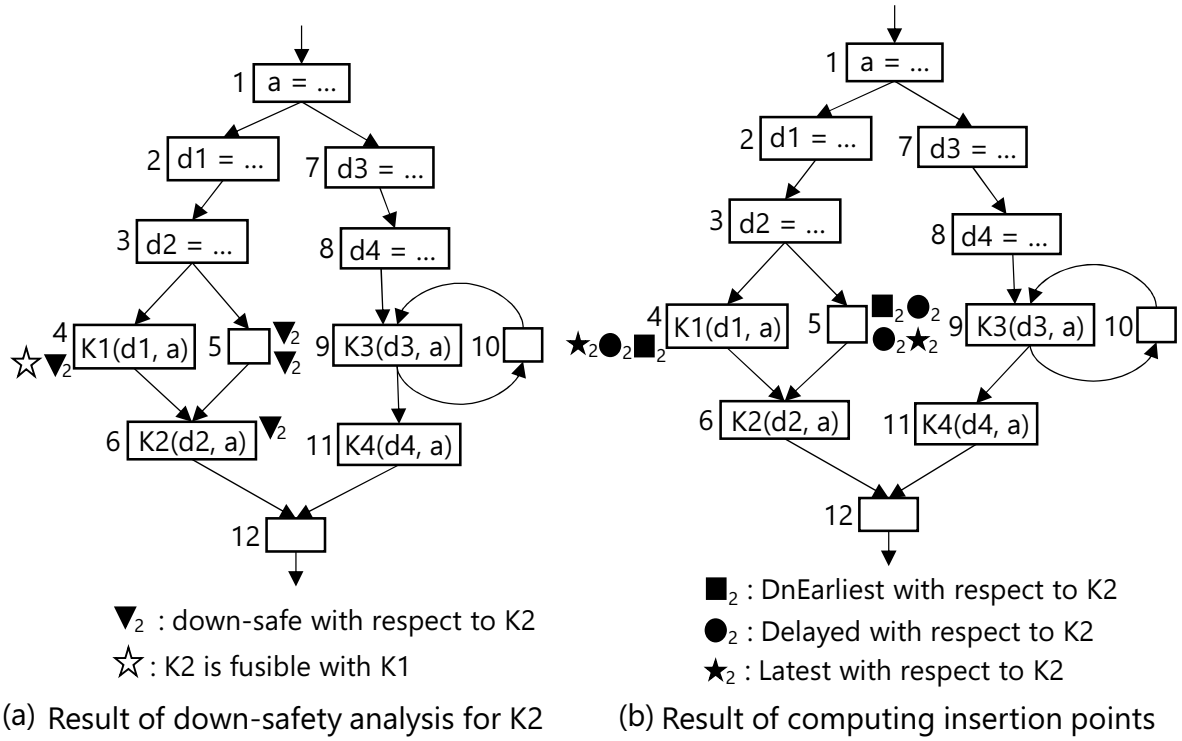


Figure 5.5: (a) Result of applying the down-safety analysis to the CFG in Figure 5.3a and finding a kernel fusion point; (b) result of computing insertion points in the backward fusion phase.

$$XFusible(n, kls, kls_2) = XdSafe(n)_{kls} \wedge Launch(n)_{kls_2}$$

Figure 5.6: Dataflow equation for the predicate *XFusible*.

[54, 55]. LCM is a traditional PRE approach comprising two analyses of earliestness and latestness based on down-safety and delayability, respectively. In the latestness part, BKF is the same as LCM. On the other hand, BKF gives a new definition to the earliestness. Each analysis step is described as follows.

First, BKF computes the earliest points of down-safety. Here, earliestness is defined by the predicates *NDnEarliest* and *XDnEarliest* in the dataflow equations shown in Figure 5.7. Each predicate for kls_K represents the following.

- $NDnEarliest(n)_{kls_K}$: the entry of n is down-safe with respect to kls_K, but kls_K cannot be moved to any predecessor of n.
- $XDnEarliest(n)_{kls_K}$: the exit of n is down-safe with respect to kls_K, but kls_K is killed in n.

Notice that BKF does not regard different occurrences of the same kernel as the same, unlike redundant expressions in LCM. Therefore, the earliest points of BKF are computed within each down-safety region without considering up-safety.

Second, BKF computes the delayability of kls_K. The delayability analysis is essentially the same as that of LCM. Notice here that the statement kls_K decided to be

$$\begin{cases} NDnEarliest(n)_{kls} = NdSafe(n)_{kls} \wedge \sum_{p \in pred(n)} \overline{XdSafe(p)_{kls}} \\ XDnEarliest(n)_{kls} = XdSafe(n)_{kls} \wedge \overline{Transp(n)_{kls}} \end{cases}$$

Figure 5.7: Dataflow equations for an earliestness analysis.

$$\begin{cases} NDelayed(n)_{kls} \\ = NDnEarliest(n)_{kls} \vee \begin{cases} false & (if\ n = \boldsymbol{s}) \\ \prod_{p \in pred(n)} \left\{ XDelayed(p)_{kls} \wedge \prod_{k \in KLS} \overline{XFusible(p, kls, k)} \right\} & (otherwise) \end{cases} \\ XDelayed(n)_{kls} = XDnEarliest(n)_{kls} \vee \left\{ NDelayed(n)_{kls} \wedge \overline{Launch(n)_{kls}} \right\} \end{cases}$$

Figure 5.8: Dataflow equations for a delayability analysis.

fused with other kernel launch statements in the previous section cannot be delayed beyond the kernel fusion point. The delayability is defined by the predicates *NDelayed* and *XDelayed* in the dataflow equations shown in Figure 5.8. Each predicate for kls_K represents the followings.

- $NDelayed(n)_{kls_K}$: kls_K can be delayed to the entry of n from *NDnEarliest* or *XDnEarliest* point.
- $XDelayed(n)_{kls_K}$: kls_K can be delayed to the exit of n from *NDnEarliest* or *XDnEarliest* point.

Finally, BKF computes the latest program points based on delayability. The latest points represent program points where kls_K cannot be delayed any further. The latestness is defined by the predicates *NLatest* and *XLatest* in the dataflow equations shown in Figure 5.9. Each predicate for kls_K represents the followings.

- $NLatest(n)_{kls_K}$: kls_K can be delayed to the entry of n, but it cannot be delayed further.

$$\begin{cases} NLatest(n)_{kls} = NDelayed(n)_{kls} \wedge Launch(n)_{kls} \\ XLatest(n)_{kls} = XDelayed(n)_{kls} \wedge \sum_{s \in succ(n)} \overline{NDelayed(s)_{kls}} \end{cases}$$

Figure 5.9: Dataflow equations for a latestness analysis.

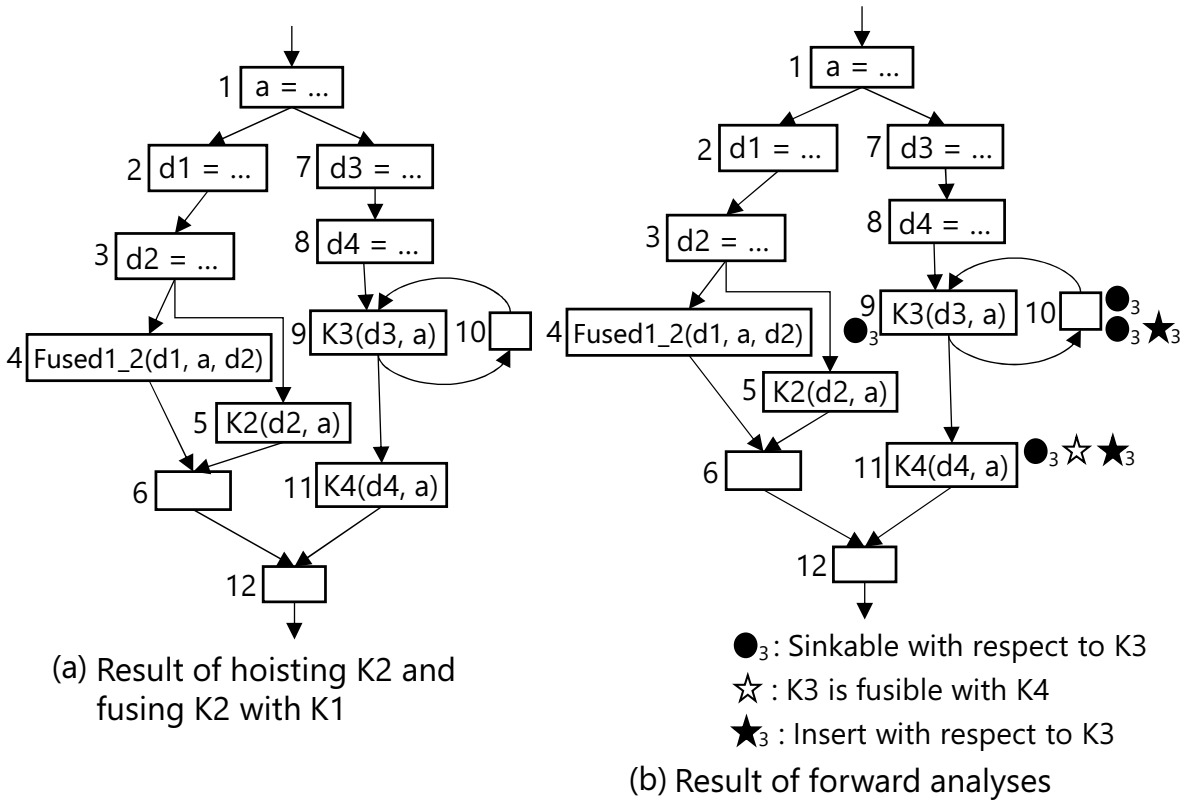


Figure 5.10: (a) Result of backward analyses and transformation based on them with respect to `K2`; (b) result of forward analyses with respect to `K3`.

- $XLatest(n)_{kls_K}$: kls_K can be delayed to the exit of n, but it cannot be delayed further.

Figure 5.5b shows the result of computing earliestness, delayability, and latestness. As shown in the figure, NDnEarliest$(5)_{kls_{K2}}$ and XDnEarliest$(4)_{kls_{K2}}$ are *true*. Therefore, NDelayed$(5)_{kls_{K2}}$, XDelayed$(5)_{kls_{K2}}$, and XDelayed$(4)_{kls_{K2}}$ are *true*. However, NDelayed$(6)_{kls_{K2}}$ is *false* because XFusible(4, kls_{K2}, kls_{K1}) is *true*. The statement kls_{K2} cannot be delayed beyond the fusion point, although XDelayed$(4)_{kls_{K2}}$ is *true*. Consequently, XLatest$(4)_{kls_{K2}}$ and XLatest$(5)_{kls_{K2}}$ are *true*.

Hoisting a Statement and Fusing Kernels

Once all predicates are computed, BKF applies transformations, i.e., actual insertions and kernel fusion of kls_K, to the program based on the predicates. First, kls_K is inserted into the entry (or exit) of a node n that satisfies NLatest$(n)_{kls_K}$ (or XLatest$(n)_{kls_K}$) = *true*. Second, the kernels K and *K2* are fused at n that satisfies XFusible(n, kls_K, kls_{K2}) = *true*.

Figure 5.10a shows the result of applying the insertion based on latestness and the fusion of the kernel `K2` with `K1` to the CFG in Figure 5.3a. According to the latestness analysis, the predicates XLatest$(4)_{kls_{K2}}$ and XLatest$(5)_{kls_{K2}}$ are *true*; therefore, `K2(d2,a)` is inserted into the exit of nodes 4 and 5. Furthermore, the kernels `K2` and `K1` are fused at node 4 because XFusible(4, kls_{K2}, kls_{K1}) is *true*. The name of the fused

$$\begin{cases} NSinkable(n)_{kls} = \begin{cases} false & (if\ n = \boldsymbol{s}) \\ \prod_{p \in pred(n)} XSinkable(p)_{kls} & (otherwise) \end{cases} \\ XSinkable(n)_{kls} = Launch(n)_{kls} \vee \{Transp(n)_{kls} \wedge NSinkable(n)_{kls}\} \end{cases}$$

Figure 5.11: Dataflow equations for a sinkability analysis.

kernel is `Fused1_2` in the figure.

5.2.2 Forward Fusion Phase

The proposed kernel fusion finds fusible kernels for each kernel launch statement backwardly and forwardly in this order. If some fusible kernels are found by BKF, KFCM applies BKF to the next statement; otherwise, the *forward kernel fusion* (FKF) is applied to the current statement. FKF finds fusible kernels forwardly and fuses them based on the dataflow equations similar to PDE, which is a code optimization that eliminates partially dead statements [56]. This phase aims to sink a kernel launch statement kls_K and fuse K with kernels that exist in later program points. This phase includes the following steps:

(1) Computing sinkability of kls_K;

(2) Finding program points where a kernel K is fusible with other kernels by sinking kls_K;

(3) Computing insertion points of kls_K; and

(4) Sinking kls_K and fusing kernels.

The details of the above steps are described as follows.

Computing Sinkability

As an initial step, FKF analyzes sinkability of kls_K, which gives the program points where kls_K is sinkable. Because kls_K can be sunk until it is killed, it may be sunk immediately before other kernel launch statements at a later program point. In this case, these kernels can be candidates to be fused. The details of fusion points are presented in the following step. The sinkability is defined by the predicates *NSinkable* and *XSinkable* in the dataflow equations shown in Figure 5.11. Each predicate for kls_K is represented as follows:

- NSinkable$(n)_{kls_K}$: kls_K is sinkable at the entry of n.
- XSinkable$(n)_{kls_K}$: kls_K is sinkable at the exit of n.

Figure 5.10b shows the result of the sinkability analysis for the kernel `K3`. As shown in the figure, NSinkable$(10)_{kls_{K3}}$, NSinkable$(11)_{kls_{K3}}$, XSinkable$(9)_{kls_{K3}}$, and XSinkable$(10)_{kls_{K3}}$ are *true*.

$$NFusible(n, kls, kls_2) = NSinkable(n)_{kls} \wedge Launch(n)_{kls_2}$$

Figure 5.12: Dataflow equation for the predicate *NFusible*.

$$\begin{cases} NInsert(n)_{kls} = NSinkable(n)_{kls} \wedge \overline{Transp(n)_{kls}} \\ XInsert(n)_{kls} = XSinkable(n)_{kls} \wedge \sum_{s \in succ(n)} \overline{NSinkable(s)_{kls}} \end{cases}$$

Figure 5.13: Dataflow equations for the predicates *NInsert* and *XInsert*.

Finding Kernel Fusion Points

Once the program points where kls_K is sinkable are given by a sinkability analysis, FKF finds the program points where K can be fused with other kernels. Assume that kls_K and kls_{K2} are launch statements for different kernels. Kernels K and *K2* are fusible at the entry of a node n where kls_K is sinkable and kls_{K2} exists. To characterize the program points, FKF introduces the predicate *NFusible* shown in Figure 5.12. NFusible(n, kls_K, kls_{K2}) means that a kernel K is fusible with another kernel *K2* at the entry of n by sinking kls_K to the entry of n.

Figure 5.10b shows that the kernel `K3` is fusible with `K4` at the entry of node 11 because kls_{K3} is sinkable, and kls_{K4} exists there. It means that NFusible(11, kls_{K3}, kls_{K4}) is *true*. However, to fuse them without changing semantics, FKF has to insert kls_{K3} as a compensation code as well as BKF. The insertion points are computed as described in the followings.

Computing Insertion Points

In FKF, insertion points of kls_K as a compensation code are decided as a result of the sinking. Here, kls_K cannot be sunk across the program point where a kernel K can be fused with other kernels. Thus, the predicates *NInsert* and *XInsert* are defined in the dataflow equations shown in Figure 5.13. Each predicate for kls_K is represented as follows:

- NInsert$(n)_{kls_K}$: kls_K can be sunk to the entry of n, where it cannot be sunk further.
- XInsert$(n)_{kls_K}$: kls_K can be sunk to the exit of n, where it cannot be sunk further.

As shown in Figure 5.10b, kls_{K3} cannot be sunk to the later program point than the entry of node 11 because the kernel `K3` is fusible with `K4` at the point. Moreover, kls_{K3} cannot be sunk to the later point than the exit of node 10 because it is not sinkable at the entry of the successor of node 10. Therefore, NInsert$(11)_{kls_{K3}}$ and XInsert$(10)_{kls_{K3}}$ are *true*.

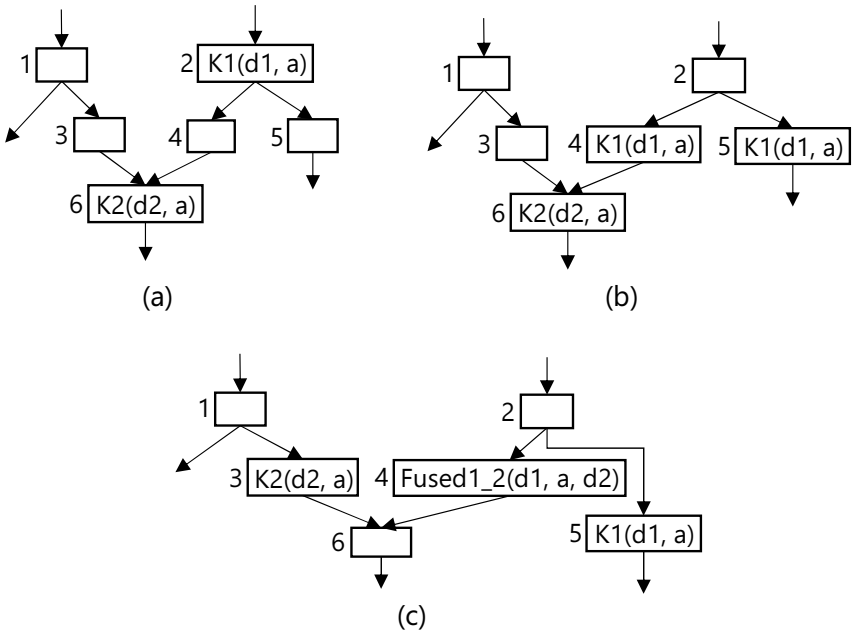


Figure 5.14: (a) Motivating example of capturing second order effects; (b) result of sinking the kernel `K1`; (c) result of hoisting the kernel `K2` and fusing it with `K1`.

Sinking a Statement and Fusing Kernels

As the final step, FKF applies transformations, i.e., actual insertions and fusion, based on the predicates obtained in the above steps. First, kls_K is inserted into the entry (or exit) of a node n that satisfies NInsert$(n)_{kls_K}$ (or XInsert$(n)_{kls_K}$) = *true*. Second, the kernels K and *K2* are fused at n that satisfies NFusible(n, kls_K, kls_{K2}) = *true*.

Figure 5.3b shows the result of inserting kls_{K3} and fusing `K3` with `K4` in the CFG in Figure 5.10a. The predicates NInsert$(11)_{kls_{K3}}$ and XInsert$(10)_{kls_{K3}}$ are *true*; hence, `K3(d3,a)` is inserted to the entry of node 11 and the exit of node 10. The kernel `K3` is fused with `K4` at node 11 because NFusible(11, kls_{K3}, kls_{K4}) is *true*. As shown in the figure, `Fused3_4` is a new name given to a single kernel in which the fusion results.

5.2.3 Application of KFCM

Each KFCM phase has a *second order effect* on the other phase. For example, in the case shown in Figure 5.14a, any kernel cannot be fused by only applying either of the phases. However, once the kernel `K1` is sunk to node 4 (Figure 5.14b), it can be fused with the kernel `K2` at node 4 by hoisting `K2` (Figure 5.14c). To effectively capture the second order effect, both phases are applied to each statement one by one while visiting nodes in a CFG from the start node to the end node. The process is repeated to capture all effects until there is no change.

The computational complexity of KFCM is $O(N^2)$, where N is the size of an input program. This is because KFCM consists of BKF and FKF that are uni-directional analyses and thus accomplished in time $O(N^2)$ [49]. To capture all the second order effect, BKF and FKF must be applied repeatedly until there is no change, so that the complexity of KFCM becomes $O(N^3)$.

5.3 Evaluation

This section shows the effectiveness of the proposed method through the results of experiments conducted for a well-known benchmark, NVIDIA SDK sample code, and real-world applications. The main purpose of the experiments is to investigate the followings:

- How effective KFCM is compared with a traditional method.
- How many kernels KFCM can fuse compared with a traditional method.

In the followings, first, the experimental environment and the implementation of KFCM are introduced. Next, the details of the experimental results are described.

5.3.1 Environment and Implementation

The proposed method has been implemented on LLVM [2, 57]. To compare the proposed method with a traditional method, the traditional method was implemented based on a flow insensitive approach that fuses only originally consecutive kernels. In what follows, the traditional one is called KF_{trad}. In the experiments, first, a CUDA code is compiled to the LLVM *intermediate representation* (IR) code using Clang [1]. Second, KFCM and KF_{trad} are applied to the IR code to generate an optimized IR code, respectively. Finally, the optimized IR code is compiled to an object code by LLVM. The experiments were conducted in the following environment.

- OS: Ubuntu 18.04 LTS,
- CPU: Intel Core i9-9900K,
- GPU: NVIDIA TITAN RTX,
- CUDA Toolkit 11.1, and
- Clang/LLVM 11.1.0.

The experiments assume that it is unnecessary to insert any global synchronization, which is a synchronization between thread blocks, between kernels to be fused. It must be manually checked because it is difficult to analytically decide whether to insert the synchronization. In the experiments, all programs do not need any global synchronization except for the program *gaussian* of the Rodinia benchmark. For *gaussian*, a global synchronization is manually inserted between kernels to be fused. On the other hand, a local synchronization is conservatively inserted, which is a synchronization between the threads in the same thread block, between fused kernels, even if it is unnecessary.

As mentioned in Section 5.2.3, second order effects can be generated after applying both phases of KFCM to one kernel launch statement. To capture the effects, KFCM was implemented such that it visits nodes in a CFG in the reverse postorder. The application of KFCM to all the nodes is repeated until there is no change.

Table 5.1: Comparison of speedup in Rodinia benchmark [14], NVIDIA SDK sample code [77], and GPU Computing Gems (GCG) [43]. Each result is normalized by the baseline *O3*.

Rodinia	KF_{trad}	KFCM	KFCM over KF_{trad}
bfs	0.99x	0.99x	-
cfd	1.35x	**1.48x**	**1.10x**
gaussian	1.02x	1.02x	-
huffman	1.00x	1.00x	1.00x
hybridsort	1.02x	1.02x	-
lud	1.03x	**1.04x**	**1.01x**
particlefilter	1.03x	1.03x	-
srad_v1	1.06x	**1.11x**	**1.05x**
srad_v2	1.22x	1.22x	-
NVIDIA SDK	KF_{trad}	KFCM	KFCM over KF_{trad}
eigenvalues	1.00x	**1.25x**	**1.25x**
lineOfSight	1.00x	**1.22x**	**1.22x**
mergeSort	1.01x	1.01x	-
newdelete	1.02x	1.02x	-
GCG	KF_{trad}	KFCM	KFCM over KF_{trad}
barneshut	1.16x	**1.21x**	**1.05x**
bspline	1.02x	1.02x	-
IFS	1.00x	**1.05x**	**1.05x**
pathReg	1.18x	**1.60x**	**1.35x**
rna_folding_gpu	1.00x	1.00x	1.00x

5.3.2 Experimental Results

KFCM and KF_{trad} are compared for the execution time of the well-known Rodinia benchmark [14], NVIDIA SDK sample code [77], and the real-world applications in GPU Computing Gems (GCG) [43]. Each program is executed 10 times, and the average of all the execution time is reported. Moreover, the number of reduced kernel launches because of kernel fusion was measured. The CUDA profiler *nvprof* [81] was used to measure them. The baseline for comparison is the code generated by Clang with the optimization option O3. In what follows, this baseline is called *O3*.

Performance Comparison

Table 5.1 presents the speedup of KF_{trad} and KFCM by ratio for the baseline *O3*. For each program in the first column, the speedup of KF_{trad} and KFCM are shown in the second and third columns, respectively. The last column represents the speedup given by KFCM compared with KF_{trad}. In the Rodinia benchmark, eight programs of 23 programs have some fusible kernels. Notice that the program *srad* of the eight programs has two implementations, *srad_v1* and *srad_v2*; therefore, both versions are reported. In the other 15 programs, the seven programs (*heartwall*, *hotspot*, *hotspot3D*, *lavaMD*, *nn*, *pathfinder*, and *streamcluster*) have only one kernel, and the other eight

programs (*b+tree*, *backprop*, *dwt2d*, *kmeans*, *leukocyte*, *mummergpu*, *myocyte*, and *nw*) have only kernels with no fusion opportunity, which have some modification statements between them or are executed exclusively. The program of *bfs* implements a breadth-first search algorithm which traverses all the connected components in a graph. *cfd* is an unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow. *gaussian* solves systems of equations using the gaussian elimination method. *huffman* implements a parallel variable-length encoder of Huffman coding on a GPU. *hybridsort* is a parallel sorting algorithm by utilizing a bucket sort and a merge sort. *lud* is an LU decomposition algorithm to calculate the solutions of a set of linear equations. *particlefilter* is a statistical estimator of the location of a target object when noisy measurements of the target's location and an idea of the object's path in a Bayesian framework are given. *srad* is speckle reducing anisotropic diffusion, which is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations. *srad_v1* processes a real image, and *srad_v2* randomizes the inputs. In NVIDIA SDK sample code, experiments were conducted for the programs in *6_Advanced*. The four programs that have some fusible kernels are reported; the other programs in *6_Advanced* have either only one kernel or kernels with no fusion opportunity. The program of *eigenvalues* computes all eigenvalues of a tridiagonal symmetric matrix of arbitrary size. *lineOfSight* is a simple line-of-sight algorithm; when a height map and a ray originating at an observation point are given, the program computes all the points along the ray that are visible from the observation point. *mergeSort* is a sorting algorithm using a merge sort on a GPU. *newdelete* is a sample program that demonstrates dynamic global memory allocation. In the real-world applications of GCG, the five programs of nine available programs have some fusible kernels. In the table, the program *iteratedfunctionsystems* is abbreviated as *IFS*. In the other four programs, the three programs (*GenomeSearch*, *LDPCdec*, and *ch4estatics*) have only one kernel, and the program *MultiSVM* has no CUDA implementation. The program of *barneshut* is a n-body simulation that computes forces acting on a body due to its interaction with other bodies. *bspline* is a deformable volumetric registration algorithm using B-splines. *iteratedfunctionsystems* implements a real-time rendering of the fractal flame algorithm. *pathReg* traces photon paths by utilizing path regeneration. *rna_folding_gpu* is an RNA folding algorithm using dynamic programming.

As presented in the table, for the eight programs, *bfs*, *gaussian*, *hybridsort*, *particlefilter*, *srad_v2*, *mergeSort*, *newdelete*, and *bspline*, KF_{trad} and KFCM transformed them similarly, resulting in the same speedup. For the other 10 programs, KFCM could fuse more kernels than KF_{trad}, resulting in more performance gain. In particular, for the five programs, *huffman*, *eigenvalues*, *lineOfSight*, *IFS*, and *rna_folding_gpu*, KFCM could fuse some kernels although KF_{trad} could not fuse any kernel. For the Rodinia benchmark, NVIDIA SDK sample code, and GCG, KF_{trad} achieves geometric mean speedups of 1.07, 1.01, and 1.07, respectively. In contrast, KFCM achieves geometric mean speedups of 1.09, 1.12, and 1.16 for the three benchmarks, respectively. Overall, KFCM achieves a geometric mean speedup of 1.12, whereas KF_{trad} achieves a geometric mean speedup of 1.06.

The Number of Reduced Kernel Launches

The number of kernel launches affects the overall performance. Therefore, the number of kernel launches reduced by kernel fusion was measured. Table 5.2 presents the degree

Table 5.2: Comparison of the number of kernel launches in Rodinia benchmark [14], NVIDIA SDK sample code [77], and GPU Computing Gems (GCG) [43].

Rodinia	*O3*	KF_{trad}	KFCM	$Reduction_{KF_{trad}}$	$Reduction_{KFCM}$
bfs	24	15	15	37.5%	37.5%
cfd	14003	8003	6003	42.9%	**57.1%**
gaussian	30	15	15	50.0%	50.0%
huffman	46	46	44	-	**4.4%**
hybridsort	68	64	64	5.9%	5.9%
lud	46	31	30	32.6%	**34.8%**
particlefilter	396	99	99	75.0%	75.0%
srad_v1	502	402	301	19.9%	**40.0%**
srad_v2	4	2	2	50.0%	50.0%
NVIDIA SDK	*O3*	KF_{trad}	KFCM	$Reduction_{KF_{trad}}$	$Reduction_{KFCM}$
eigenvalues	300	300	201	-	**33%**
lineOfSight	403	403	304	-	**24.6%**
mergeSort	49	37	37	24.5%	24.5%
newdelete	6	5	5	16.7%	16.7%
GCG	*O3*	KF_{trad}	KFCM	$Reduction_{KF_{trad}}$	$Reduction_{KFCM}$
barneshut	93	78	66	16.1%	**29.0%**
bspline	96	72	72	25.0%	25.0%
IFS	1275	1275	1200	-	**5.9%**
pathReg	10026	7512	6420	25.1%	**36.0%**
rna_folding_gpu	75774	75774	75771	-	0.004%

of the reduction. As presented in the table, for each program in the first column, the number of kernel launches in *O3*, KF_{trad}, and KFCM are shown in the second, third, and fourth columns, respectively. The fifth and sixth columns show the ratio of the number of kernel launches reduced by KF_{trad} and KFCM to the one reduced by *O3*. For the eight programs, *bfs*, *gaussian*, *hybridsort*, *particlefilter*, *srad_v2*, *mergeSort*, *newdelete*, and *bspline*, since KF_{trad} and KFCM transformed them similarly, both methods achieved the same reduction. For the other 10 programs, KFCM reduced the number of kernel launches more than KF_{trad} because KFCM could fuse more kernels in those programs than KF_{trad}. For the Rodinia benchmark, NVIDIA SDK sample code, and GCG, KF_{trad} reduced the number of kernel launches by a geometric mean of 39.2%, 10.9%, and 14.0%, respectively. KFCM reduced the number of kernel launches by a geometric mean of 43.5%, 24.9%, and 20.3% for the three benchmarks, respectively. Overall, KFCM reduced the number of kernel launches by a geometric mean of 33.8%, whereas KF_{trad} reduced it by a geometric mean of 27.1%. The experimental result shows that KFCM has the potential to fuse more kernels than KF_{trad}.

5.3.3 Discussions of the Experiments

KFCM performs well in the experiments. It reduces more kernel launches than a traditional method by fusing more kernels, improving the performance. The experimental

results show that KFCM is particularly effective compared with KF_{trad} for the fluid dynamics (*cfd*), linear algebra (*eigenvalues*), and simulating programs (*lineOfSight*, *barneshut*, and *pathReg*). The programs have many kernels and launch them in branches and loops. Since KFCM exposes the fusibility of kernels by a flow sensitive approach, KFCM is effective for the programs that has many kernel launches in branches and loops. However, the current KFCM has some issues that may decrease performance through fusing all fusible kernels. That is, fusing kernels may not always give better performance than leaving the kernels original. For example, when the kernels include ones consuming a lot of GPU memories, fusing them may decrease performance. To address the issue, it would be effective for KFCM to introduce an objective function that estimates the benefits of each kernel fusion, as reported by Qiao et al. [85] and Wahib et al. [102, 103]. Besides, the effectiveness of KFCM may depend on the application order of the two fusion phases, BKF and FKF. The current KFCM first applies BKF to an input program and then applies FKF, but, in some cases, reversing the application order may give better performance. Furthermore, the current KFCM fuses candidate kernels only vertically. As reported by Li et al. [60, 61], horizontal kernel fusion is beneficial when fused kernels include some memory-intensive kernels and some compute-intensive ones, which require different types of GPU resources. The horizontal kernel fusion would enable KFCM to result in better performance.

KFCM creates the fused kernel in a device code, so that it increases the size of the device code. In the experiments, KFCM increased the size of a device code by a geometric mean of 35.6%. However, the average size of a device code before applying KFCM was 32.0 kilobytes while the one after applying KFCM was 39.8 kilobytes. Thus, the increased size was insignificantly small compared to the size of the input data of GPU programs.

5.4 Summary

This chapter describes a novel method that fuses more kernels in GPU programs than a traditional kernel fusion based method by a flow sensitive approach. The proposed method exposes the fusibility of kernels based on dataflow analyses utilized in traditional code optimizations. The experimental results have shown that the proposed method can fuse more kernels and perform better than the traditional method for the well-known benchmark programs, NVIDIA SDK sample code, and real-world applications. The proposed method has achieved a geometric mean speedup of 1.12 and reduced the number of kernel launches by a geometric mean of 33.8%.

Chapter 6

Related Work

Many techniques that improve the performance of GPU programs have been developed. In this chapter, first, existing branch divergence reduction techniques are discussed. Second, kernel fusion based methods are discussed. Finally, some other optimizations for GPU programs are described, categorized as intra- and inter-kernel optimizations.

6.1 Branch Divergence Reduction

Coutinho et al. [19] proposed *branch fusion* that reduces the computational cost of a divergent branch by combining computations with the same operator in the true and false sides of the branch into a single statement. However, this method may need to insert new branches and select statements. The insertion may decrease performance of GPU programs. The number of the insertions depends on the order of statements in an original branch. Saumya et al. [90] proposed a control flow melding technique, called *Divergence-Aware-Region-Melder* (DARM), which can handle more general control flow structures than branch fusion. DARM can meld similar control flow subgraphs inside a if-then-else region. In the melding process, DARM recursively melds divergent control flow at the level of subgraphs of a CFG, allowing threads to reconverge early and reducing branch divergence. Han and Abdelrahman proposed *branch distribution* [36] that hoists the computations with the same operator in a divergent branch as a single computation out of the branch. When the method finds the computation with the same operator in a branch, it inserts a new branch with the same condition as the original branch and then moves the target computations out of the branch. If the operands of target computations are different, it is necessary to introduce temporary variables to retain the suitable values. Thus, branch distribution may insert many new branches; therefore, it is effective only when the effectiveness of hoisting statements out of a branch is greater than the cost of the inserted branches and the concentration of hoisted statements to one place. Damani et al. [21] proposed *Common Subexpression Convergence* (CSC). When true and false sides of a divergent branch contain common expressions, CSC moves such expressions to a convergent program point, enabling more threads to execute them in parallel.

Wu et al. [110] transform an unstructured control flow graph to a structured one, which contributes to the reduction of branch divergence. In an unstructured CFG, some basic blocks may be executed 2 times or more because of the divergence. On the other hand, in a structured CFG, such redundant execution does not occur. However,

this method increases the code size exponentially because it performs transformation by copying code. Anantpur et al. [8] transform an unstructured CFG to a structured one through the linearization based on the idea of guarded execution of basic blocks. For each basic block of a CFG, the linearization method creates a guard basic block to guard its execution. The mechanism reconverges divergent threads as early as possible. In addition, it does not duplicate code, so that it incurs only a linear increase in the number of basic blocks. Reissmann et al. [86] proposed a control flow restructuring technique that consists of loop restructuring and branch restructuring. Loop restructuring converts all loops to tail-controlled loops, and branch restructuring ensures proper nesting of control flow. These restructuring techniques work by adding predicates and branches to a CFG, so that they avoid the risk of exponential code inflation.

Han and Abdelrahman proposed *iteration delaying* [36], which is applied to a divergent branch within a loop. In each iteration of the loop, the method delays some statements such that they can be executed together with other unexecuted statements on the same side of the branch in subsequent iterations. It enables more threads to execute in parallel, reducing branch divergence and improving performance. Zhang et al. [116] proposed *G-Streamline* to reduce dynamic irregularities in both branch divergence and memory references. An irregular memory reference occurs when the requested data lies on multiple memory segments, resulting in more memory transactions than necessary. Their framework reduces both types of irregularities through data reordering and job swapping. Lin et al. [64] proposed a thread-data remapping technique to achieve runtime branch divergence reduction. Their technique performs source-to-source transformation on each kernel. The thread-data remapping is performed via shared memory and within warps. By remapping data, their technique reduces both branch divergence and memory coalescing overhead. Damani et al. [20] proposed a user-guided speculative reconvergence mechanism that enables threads in a warp to be reconverged at an earlier program point than the immediate post-dominator of the divergent branch. Their mechanism utilizes user-guided information to identify common code paths within a program and exploit reconvergence opportunities.

6.2 Kernel Fusion

Qiao et al. [85, 84] proposed a kernel fusion method to find fusible kernels based on the minimum cut technique for an image processing DSL. Inspired by traditional loop fusion techniques [9, 34, 47, 94, 68], their method improves data locality on GPUs. Their algorithm first constructs a data dependence graph among kernels. Second, it estimates the benefit of fusing kernels connected by an edge in the graph and assigns the benefit to the edge as a weight. Finally, the algorithm performs the minimum cut on the graph, maximizing the benefit of the fusion. Filipovič et al. [27] proposed a kernel fusion for basic linear algebra subprograms (BLAS) applications. They restrict the type of fusible kernel to *map* and *reduce* because an automatic fusion of generic kernels is difficult. Lutz et al. [67] proposed *Helium*, which builds a dynamic task and data dependency graph for an OpenCL program. Their system analyzes data dependency between kernels using the runtime information and performs horizontal and vertical kernel fusion. Wahib and Maruyama [102, 103] proposed a method to formalize a kernel fusion as an optimization problem. Their method searches the space of possible kernel fusions to identify optimal fusions.

Wang et al. [104] proposed three types of fusion targeted for different situations to reduce energy consumption on GPUs. The three fusions contain the standard vertical fusion and the horizontal distribution of the computation among different threads or thread blocks. Li et al. [60, 61] proposed a horizontal kernel fusion based technique that increases the thread-level parallelism to hide instruction latencies. The technique searches for the best thread space partition when it fuses kernels horizontally. Their experimental results show that horizontal fusion is beneficial when fused kernels require different types of GPU resources.

6.3 Other Optimizations for GPUs

6.3.1 Intra-kernel Optimization

Many dimensionally redundant SIMT instruction elimination techniques [52, 111, 106, 65, 53, 115] have been proposed. With the SIMT execution model, it is well-known that there are many redundantly executed instructions at the grid, TB, and warp level. The techniques find and remove the redundancy, saving energy and improving performance. Recently, Yeh et al. [115] proposed a non-speculative instruction skipping mechanism. It skips redundant instructions within TBs before they are fetched through static information markings from a compiler and dynamic information, such as the dimension size of the TBs known at kernel launch time.

Shobaki et al. [92] proposed a scheduling algorithm that improves occupancy and Instruction-Level Parallelism (ILP). First, the algorithm searches for a maximum occupancy schedule. Next, the algorithm searches for the shortest schedule that provides maximum occupancy. Although minimizing register pressure conflicts with exploiting ILP, the proposed algorithm separates the two problems and efficiently solves one problem at a time. Shobaki et al. [93] proposed a register-pressure-aware instruction scheduling algorithm using Ant Colony Optimization (ACO) to solve the compiler optimization problem of balancing ILP and Register Pressure (RP). Maximizing ILP conflicts with minimizing RP, and these two objectives must be balanced in pre-allocation scheduling because both ILP and RP affect the performance.

Intra-kernel data locality improvement approaches [108, 15, 59, 62, 101, 100] improve data cache utilization in a single kernel. Recently, Tripathy et al. [100] proposed the approach that exploits the inter-TB data cache locality by creating a graph of TBs and cutting the graph to decide which TBs to group in the same SM, maximizing the cache utilization between the TBs and improving performance.

6.3.2 Inter-kernel Optimization

Inter-kernel data locality improvement approaches [63, 42, 105] exploit inter-kernel shared data and improve performance. Huzaifa et al. [42] proposed four TB schedulers that optimize inter-kernel data reuse by scheduling threads that require some data on the same core as threads from a previous kernel which shared the data. Li et al. [63] proposed the framework that reduces the impact caused by the memory oversubscription. With unified virtual memory and demand paging, both CPUs and GPUs can access some data with the same virtual address on demand. Their framework reuses the inter-kernel shared data on unified virtual memory, reducing the data

communication between CPUs and GPUs.

Kernel scheduling and co-execution techniques [105, 83, 117, 58, 91, 66] try to utilize full GPU resources by scheduling independent kernels and executing concurrently. In recent work, López-Albelda et al. [66] proposed a software scheduler that performs a specific TB allocation of co-executing kernels to SMs with low overhead. With the online profiler that they implemented to obtain information of co-executing kernels, the scheduler establishes a suitable SM partitioning for concurrent kernels.

Chapter 7

Conclusion

The thesis proposed intra- and inter-kernel optimizations to improve performance of GPU programs. In this chapter, the thesis is concluded by presenting the summary in Section 7.1 and describing the future work in Section 7.2.

7.1 Summary

First, as intra-kernel optimizations, the thesis proposed two methods, SSCM and SSRQP, which reduce branch divergence in each GPU kernel and improve performance. Taking the property of branch divergence into account, SSCM decreases the static number of expressions and performs speculative code motion for only divergent branches by extending traditional dataflow equations. The experimental results showed that SSCM could improve the performance of a kernel with branch divergence although traditional PRE and speculative PRE that performs speculative code motion for both non-divergent branches and divergent branches could not improve it. In the experiments, speculative PRE decreased performance because of speculative code motion for non-divergent branches. In contrast, SSCM selectively applied speculative code motion for branches, resulting in branch divergence reduction and removing redundancy more than traditional PRE without penalty. In the best case, SSCM achieved 1.08x speedup for the well-known and heavily optimized benchmark program compared with the O3 option of the NVIDIA compiler. Additionally, SSRQP performs scalar replacement and speculative code motion to remove redundant memory accesses in a loop and reduce branch divergence by extending traditional question propation. SSRQP propagates a query about speculation to check whether an expression can be hoisted speculatively. As well as SSCM, SSRQP performs speculative code motion for only divergent branches. Moreover, utilizing control dependency, SSRQP introduces a CDRND in order to suppress inserting expressions into the destinations of divergent branches, resulting in the suppression of increasing branch divergence. The experimental results have indicated that SSRQP can improve more execution efficiency of GPU kernels with branch divergence than traditional related techniques. In the best case, SSRQP achieved 1.88x speedup compared with the O3 option of the NVIDIA compiler.

Second, as an inter-kernel optimization, the thesis proposed KFCM, which is the code motion based kernel fusion that can fuse more kernels than traditional methods. KFCM exposes the fusibility of kernels based on dataflow analyses utilized in traditional code optimizations. KFCM comprises two fusion phases, and each phase consists of

a fusibility analysis and transformation based on the result. The first fusion phase performs a backward fusibility analysis, which hoists each kernel launch statement to the earlier program points without lengthening any execution path. The phase checks whether the hoisted launch statement is adjacent to some other launch statements. Once adjacent ones are detected, they are fused as a single kernel. The second fusion phase performs a forward fusibility analysis, which sinks each kernel launch statement to the latest program points without lengthening any execution path. As well as the first phase, this phase checks adjacency and fuses adjacent launch statements. The experimental results have shown that KFCM can fuse more kernels and perform better than the traditional method for the well-known benchmark programs, NVIDIA SDK sample code, and real-world applications. In the best case, KFCM achieved 1.60x and 1.35x speedup compared with the O3 option of Clang and a traditional method, respectively.

The intra-kernel optimizations proposed in the thesis tackle the problem of reducing branch divergence, which causes a significant performance penalty if a GPU kernel has many divergent branches. On the other hand, the proposed inter-kernel optimization enhances existing kernel fusion methods by increasing the fusibility of kernels. Although the benchmark programs used in the experiments are well-known and heavily optimized in terms of intra-kernel, they tend not to be well-optimized in terms of inter-kernel. Thus, the performance gains of the proposed inter-kernel optimization were relatively bigger than those of the proposed intra-kernel ones. Furthermore, the intra- and inter-kernel optimizations are orthogonal, and hence, they can be applied simultaneously. KFCM enlarges each kernel in a GPU program by kernel fusion and increases opportunities for intra-kernel optimizations, making SSCM and SSRQP more effective. The more important role GPUs play in general purpose applications, the more important compiler optimizations for GPUs become. Therefore, both types of optimizations proposed in the thesis have the potential to significantly improve performance of GPU applications, contributing to the development of modern high-performance computing.

7.2 Future Work

SSCM utilizes traditional dataflow equations, so that the insertion points are computed efficiently. However, SSCM performs LCM before computing insertion points, and hence, it may increase branch divergence. In contrast, SSRQP introduces a CDRND to suppress the increase in branch divergence. However, it may suppress harmless insertion of SRQP. The insertion into the destinations of divergent branches does not always increase execution cost of them. That is, the code motion that SSRQP performs is too conservative. Hence, it is desirable to develop a method that analyzes an actual execution cost of the programs before and after performing code motion in the presence of branch divergence. Additionally, the divergence analysis used in the experiments is a pessimistic analysis, that is, it statically considers a branch that is non-divergent at runtime to be divergent. Because both SSCM and SSRQP use the result of the divergence analysis, they may apply speculative code motion to non-divergent branches. To solve the problem, it is desirable to achieve selective application based on the dynamic checking of branch divergence.

Although the proposed KFCM outperforms a traditional kernel fusion based method, the current KFCM may not always increase performance because it fuses fusible ker-

nels without estimating them. Besides, the effectiveness of KFCM may depend on the application order of the two fusion phases. In addition, the current KFCM performs only vertical kernel fusion. Combining different types of fusion may enable it to perform better. It would open a way for introducing an objective function that estimates the benefits of each kernel fusion before performing it, making the application order of fusion phases changeable, and utilizing other types of fusion.

Bibliography

[1] Clang: a c language family frontend for llvm, 2022. URL: https://clang.llvm.org/.

[2] The llvm compiler infrastructure, 2022. URL: https://llvm.org/.

[3] Llvm language reference manual, 2022. URL: https://llvm.org/docs/LangRef.html.

[4] Multi-level intermediate representation overview, 2022. URL: https://mlir.llvm.org/.

[5] Opencl, 2022. URL: https://www.khronos.org/opencl/.

[6] Rocm compiler sdk, 2022. URL: https://rocmdocs.amd.com/en/latest/ROCm_Compiler_SDK/ROCm-Compiler-SDK.html.

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[8] Jayvant Anantpur and Govindarajan R. Taming control divergence in gpus through control flow linearization. In Albert Cohen, editor, *Compiler Construction*, pages 133–153, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[9] U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT'10, pages 343–352. ACM, 2010. doi:10.1145/1854273.1854317.

[10] Nicolas Brunie, Caroline Collange, and Gregory Diamos. Simultaneous branch and warp interweaving for sustained gpu performance. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 49–60, 2012. doi:10.1109/ISCA.2012.6237005.

[11] Qiong Cai and Jingling Xue. Optimal and efficient speculation-based partial redundancy elimination. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 91–102, 2003. doi:10.1109/CGO.2003.1191536.

[12] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 53–65, New York, NY, USA, 1990. Association for Computing Machinery. doi:10.1145/93542.93553.

[13] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software-Practice & Experience*, 24(1):51–77, 1994. doi:10.1002/spe.4380240104.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, W. J. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC'09, pages 44–54, 2009.

[15] Li-Jhan Chen, Hsiang-Yun Cheng, Po-Han Wang, and Chia-Lin Yang. Improving gpgpu performance via cache locality aware thread block scheduling. *IEEE Computer Architecture Letters*, 16(2):127–131, 2017. doi:10.1109/LCA.2017.2693371.

[16] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on ssa form. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 273–286, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/258915.258940.

[17] John Cocke. Global common subexpression elimination. In *Proceedings of a Symposium on Compiler Optimization*, pages 20–24, New York, NY, USA, 1970. Association for Computing Machinery. doi:10.1145/800028.808480.

[18] Keith Cooper, Jason Eckhardt, and Ken Kennedy. Redundancy elimination revisited. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 12–21, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1454115.1454120.

[19] B. Coutinho, D. Sampaio, M. Q. F. Pereira, and W. Meira Jr. Divergence analysis and optimizations. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT'11, pages 320–329, 2011. doi:10.1109/PACT.2011.63.

[20] S. Damani, D. R. Johnson, M. Stephenson, S. W. Keckler, E. Yan, M. McKeown, and O. Giroux. Speculative reconvergence for improved simt efficiency. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, pages 121–132. Association for Computing Machinery, 2020. doi:10.1145/3368826.3377911.

[21] Sana Damani and Vivek Sarkar. Common subexpression convergence: A new code optimization for simt processors. In *Languages and Compilers for Parallel Computing*, pages 64–73, Cham, 2021. Springer International Publishing.

[22] D. M. Dhamdhere. Practical adaption of the global optimization algorithm of morel and renvoise. *ACM Trans. Program. Lang. Syst.*, 13(2):291–294, apr 1991. doi:10.1145/103135.214520.

[23] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT'10, pages pages 353–364, 2010.

[24] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. Simd re-convergence at thread frontiers. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 477–488, 2011.

[25] Ahmed ElTantawy, Jessica Wenjie Ma, Mike O'Connor, and Tor M. Aamodt. A scalable multi-path microarchitecture for efficient gpu control flow. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 248–259, 2014. doi:10.1109/HPCA.2014.6835936.

[26] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, jul 1987. doi:10.1145/24039.24041.

[27] J. Filipovič, M. Madzin, J. Fousek, and L. Matyska. Optimizing cuda code by kernel fusion: application on blas. *J Supercomput*, 71:3934–3957, 2015.

[28] J. Fousek, J. Filipovič, and M. Madzin. Automatic fusions of cuda-gpu kernels for parallel map. In *Second International workshop on highly-efficient accelerators and reconfigurable technologies (HEART)*, pages 42–47, 2011.

[29] J. Fukuhara and M. Takimoto. Branch divergence reduction based on code motion. *Journal of Information Processing*, 28:302–309, 2020. doi:10.2197/ipsjjip.28.302.

[30] J. Fukuhara and M. Takimoto. Scalar replacement considering branch divergence. *Journal of Information Processing*, 30:164–178, 2022. doi:10.2197/ipsjjip.30.164.

[31] Junji Fukuhara and Munehiro Takimoto. Automated kernel fusion for gpu based on code motion. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2022, pages 151–161, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519941.3535078.

[32] Wilson W. L. Fung and Tor M. Aamodt. Thread block compaction for efficient simt control flow. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 25–36, 2011. doi:10.1109/HPCA.2011.5749714.

[33] Wilson W.L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 407–420, 2007. doi:10.1109/MICRO.2007.30.

[34] G. R. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 281–295. Springer-Verlag, 1992.

[35] R. Gupta, D.A. Berson, and J.Z. Fang. Path profile guided partial redundancy elimination using speculation. In *Proceedings of the 1998 International Conference*

on Computer Languages (Cat. No.98CB36225), pages 230–239, 1998. doi:10.1109/ICCL.1998.674173.

[36] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in gpu programs. In *Proceedings of 4th Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU'11, 2011. doi:10.1145/1964179.1964184.

[37] Tianyi David Han and Tarek S. Abdelrahman. Reducing divergence in gpgpu programs with loop merging. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 12–23, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2458523.2458525.

[38] Ari B. Hayes, Lingda Li, Daniel Chavarría-Miranda, Shuaiwen Leon Song, and Eddy Z. Zhang. Orion: A framework for gpu occupancy tuning. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2988336.2988355.

[39] N. Hegde, J. Liu, K. Sundararajah, and M. Kulkarni. Treelogy: A benchmark suite for tree traversals. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS'17, pages pages 227–238, 2017.

[40] R.N. Horspool and H.C. Ho. Partial redundancy elimination driven by a cost-benefit analysis. In *Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering*, pages 111–118, 1997. doi:10.1109/ICCSSE.1997.599882.

[41] Yun-Chi Huang, Kuan-Chieh Hsu, Wan-shan Hsieh, Chen-Chieh Wang, Chia-Han Lu, and Chung-Ho Chen. Dynamic simd re-convergence with paired-path comparison. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 233–236, 2016. doi:10.1109/ISCAS.2016.7527213.

[42] Muhammad Huzaifa, Johnathan Alsop, Abdulrahman Mahmoud, Giordano Salvador, Matthew D. Sinclair, and Sarita V. Adve. Inter-kernel reuse-aware thread block scheduling. *ACM Trans. Archit. Code Optim.*, 17(3), aug 2020. doi:10.1145/3406538.

[43] W. W. Hwu. *GPU COMPUTING GEMS Emerald Edition*. Morgan Kaufmann, 2011.

[44] T. Ito. *Introduction to GPU Programming, Implementation by CUDA 5*. Kodansha, 2013. (in japanese).

[45] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, jan 1976. doi:10.1145/321921.321938.

[46] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977. doi:10.1007/BF00290339.

[47] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320. Springer-Verlag, 1993.

[48] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in ssa form. *ACM Trans. Program. Lang. Syst.*, 21(3):627–676, may 1999. doi:10.1145/319301.319348.

[49] Uday P. Khedker and Dhananjay M. Dhamdhere. A generalized theory of bit vector data flow analysis. *ACM Trans. Program. Lang. Syst.*, 16(5):1472–1511, sep 1994. doi:10.1145/186025.186043.

[50] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Efficient warp execution in presence of divergence with collaborative context collection. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 204–215, 2015. doi:10.1145/2830772.2830796.

[51] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. Association for Computing Machinery. doi:10.1145/512927.512945.

[52] Ji Kim, Christopher Torng, Shreesha Srinath, Derek Lockhart, and Christopher Batten. Microarchitectural mechanisms to exploit value structure in simt architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 130–141, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2485922.2485934.

[53] Keunsoo Kim and Won Woo Ro. Wir: Warp instruction reuse to minimize repeated computations in gpus. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 389–402, 2018. doi:10.1109/HPCA.2018.00041.

[54] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI'92, pages 224–234. ACM, 1992. doi:10.1145/143095.143136.

[55] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, 1994.

[56] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI'94, pages 147–158. ACM, 1994. doi:10.1145/178243.178256.

[57] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, CGO'04, pages 75–86, 2004. doi:10.1109/CGO.2004.1281665.

[58] Haeseung Lee and Mohammad Abdullah Al Faruque. Gpu-evr: Run-time event based real-time scheduling framework on gpgpu platform. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2014. doi:10.7873/DATE.2014.233.

[59] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. Improving gpgpu resource utilization through alternative thread block scheduling. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 260–271, 2014. doi:10.1109/HPCA.2014.6835937.

[60] A. Li, B. Zheng, G. Pekhimenko, and F. Long. Automatic horizontal fusion for gpu kernels, 2020. arXiv:2007.01277.

[61] A. Li, B. Zheng, G. Pekhimenko, and F. Long. Automatic horizontal fusion for gpu kernels. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 14–27, 2022. doi:10.1109/CGO53902.2022.9741270.

[62] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. Locality-aware cta clustering for modern gpus. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 297–311, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3037697.3037709.

[63] Xuanyi Li, Chen Li, Yang Guo, and Rachata Ausavarungnirun. Improving inter-kernel data reuse with cta-page coordination in gpgpu. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2021. doi:10.1109/ICCAD51958.2021.9643535.

[64] H. Lin, C. Wang, and H. Liu. On-gpu thread-data remapping for branch divergence reduction. *ACM Transactions on Architecture and Code Optimization*, 15(3), October 2018. doi:10.1145/3242089.

[65] Zhenhong Liu, Syed Gilani, Murali Annavaram, and Nam Sung Kim. G-scalar: Cost-effective generalized scalar execution architecture for power-efficient gpus. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 601–612, 2017. doi:10.1109/HPCA.2017.51.

[66] Bernabé López-Albelda, Francisco M. Castro, José M. González-Linares, and Nicolás Guil. Flexsched: Efficient scheduling techniques for concurrent kernel execution on gpus. *The Journal of Supercomputing*, 78:43–71, 2022. doi:10.1007/s11227-021-03819-z.

[67] Thibaut Lutz, Christian Fensch, and Murray Cole. Helium: A transparent inter-kernel optimizer for opencl. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, pages 70–80. ACM, 2015. doi:10.1145/2716282.2716284.

[68] K. S. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, jul 1996. doi:10.1145/233561.233564.

[69] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 235–246, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1815961.1815992.

[70] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, feb 1979. doi:10.1145/359060.359069.

[71] R. Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.

[72] Roman Novak. Loop optimization for divergence reduction on gpus with simt architecture. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1633–1642, 2015. doi:10.1109/TPDS.2014.2324587.

[73] NVIDIA. Inside volta: The world's most advanced data center gpu, 2017. URL: https://developer.nvidia.com/blog/inside-volta.

[74] NVIDIA. Nvidia tesla v100 gpu architecture, 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[75] NVIDIA. Nvidia a100 tensor core gpu architecture, 2020. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[76] NVIDIA. Cuda c++ programming guide, 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[77] NVIDIA. Cuda samples, 2022. URL: https://docs.nvidia.com/cuda/cuda-samples/index.html.

[78] NVIDIA. Nvidia cuda compiler driver nvcc, 2022. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[79] NVIDIA. Nvidia h100 tensor core gpu architecture, 2022. URL: https://www.hpctech.co.jp/catalog/gtc22-whitepaper-hopper_v1.01.pdf.

[80] NVIDIA. Nvidia hopper architecture in-depth, 2022. URL: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/.

[81] NVIDIA. Profiler user's guide, 2022. URL: https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[82] NVIDIA. Ptx: Parallel thread execution isa, 2022. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[83] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 407–418, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2451116.2451160.

[84] B. Qiao, O. Reiche, F. Hannig, and J. Teich. Automatic kernel fusion for image processing dsls. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, SCOPES 2018, pages 76–85. ACM, 2018.

[85] B. Qiao, O. Reiche, F. Hannig, and J. Teich. From loop fusion to kernel fusion: A domain-specific approach to locality optimization. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 242–253. IEEE, 2019.

[86] N. Reissmann, T. L. Falch, B. A. Bjørnseth, H. Bahmann, J. C. Meyer, and M. Jahre. Efficient control flow restructuring for gpus. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 48–57, 2016. doi:10.1109/HPCSim.2016.7568315.

[87] Minsoo Rhu and Mattan Erez. The dual-path execution model for efficient gpu control flow. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 591–602, 2013. doi:10.1109/HPCA.2013.6522352.

[88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'88, pages 12–27, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/73560.73562.

[89] Oliver Rüthing, Jens Knoop, and Bernhard Steffen. Sparse code motion. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 170–183, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/325694.325715.

[90] Charitha Saumya, Kirshanthan Sundararajah, and Milind Kulkarni. Darm: Control-flow melding for simt thread divergence reduction. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–13, 2022. doi:10.1109/CGO53902.2022.9741285.

[91] S.-Kazem Shekofteh, Hamid Noori, Mahmoud Naghibzadeh, Holger Fröning, and Hadi Sadoghi Yazdi. ccuda: Effective co-scheduling of concurrent kernels on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 31(4):766–778, 2020. doi:10.1109/TPDS.2019.2944602.

[92] G. Shobaki, A. Kerbow, and S. Mekhanoshin. Optimizing occupancy and ilp on the gpu using a combinatorial approach. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, pages 133–144. ACM, 2020. doi:10.1145/3368826.3377918.

[93] Ghassan Shobaki, Vahl Scott Gordon, Paul McHugh, Theodore Dubois, and Austin Kerbow. Register-pressure-aware instruction scheduling using ant colony optimization. *ACM Trans. Archit. Code Optim.*, 19(2), jan 2022. doi:10.1145/3505558.

[94] S. K. Singhai and K. S. McKinley. A parametrized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6):340–355, 1997. doi:10.1093/comjnl/40.6.340.

[95] John A. Stratton, Christopher Rodrigrues, I-Jui Sung, Nady Obeid, Liwen Chang, Geng Liu, and Wen-Mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, mar 2012.

[96] Y. Sumikawa, R. Ojima, and M. Takimoto. Demand-driven scalar replacement. *Computer Software*, Vol.32(No.2):93–113, 2015. (in Japanese). doi:10.11309/jssst.32.2_93.

[97] Y. Sumikawa and M. Takimoto. Effective demand-driven partial redundancy elimination. *IPSJ Transactions on Programming*, Vol.6(No.2):33–44, 2013.

[98] R. Surendran, R. Barik, J. Zhao, and V. Sarkar. Inter-iteration scalar replacement using array ssa form. In *Compiler Construction*, CC'14, pages 40–60, Berlin, Heidelberg, 2014. Springer, Berlin, Heidelberg. doi:10.1007/978-3-642-54807-9_3.

[99] M. Takimoto. Speculative partial redundancy elimination based on question propagation. *IPSJ Transactions on Programming*, Vol.2(No.5):15–27, 2009. (in Japanese).

[100] Devashree Tripathy, Amirali Abdolrashidi, Laxmi Narayan Bhuyan, Liang Zhou, and Daniel Wong. Paver: Locality graph-based thread block scheduling for gpus. *ACM Trans. Archit. Code Optim.*, 18(3), jun 2021. doi:10.1145/3451164.

[101] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B. Gibbons, and Onur Mutlu. The locality descriptor: A holistic cross-layer abstraction to express data locality in gpus. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 829–842, 2018. doi:10.1109/ISCA.2018.00074.

[102] M. Wahib and N. Maruyama. Scalable kernel fusion for memory-bound gpu applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 191–202, 2014. doi:10.1109/SC.2014.21.

[103] M. Wahib and N. Maruyama. Automated gpu kernel transformations in large-scale production stencil applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC'15, pages 259–270. ACM, 2015. doi:10.1145/2749246.2749255.

[104] G. Wang, Y. Lin, and W. Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350, 2010. doi:10.1109/GreenCom-CPSCom.2010.102.

[105] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. Laperm: Locality aware scheduler for dynamic parallelism on gpus. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 583–595, 2016. doi:10.1109/ISCA.2016.57.

[106] Kai Wang and Calvin Lin. Decoupled affine computation for simt gpus. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 295–306, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3079856.3080205.

[107] Yaohua Wang, Dong Wang, Shuming Chen, Zonglin Liu, Shenggang Chen, Xiaowen Chen, and Xu Zhou. Iteration interleaving–based simd lane partition. *ACM Trans. Archit. Code Optim.*, 12(4), jan 2016. doi:10.1145/2847253.

[108] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 119–130, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2751205.2751213.

[109] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanshili, and S. Chakradhar. Optimizing data warehousing applications for gpus using kernel fusion/fission. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW, pages 2433–2442. IEEE, 2012. doi:10.1109/IPDPSW.2012.300.

[110] Haicheng Wu, Gregory Diamos, Si Li, and Sudhakar Yalamanchili. Characterization and transformation of unstructured control flow in gpu applications. In *Proceedings of the 1st International Workshop on Characterizing Applications for Heterogeneous Exascale Systems*, 2011.

[111] Ping Xiang, Yi Yang, Mike Mantor, Norm Rubin, Lisa R. Hsu, and Huiyang Zhou. Exploiting uniform vector instructions for gpgpu performance, energy efficiency, and opportunistic reliability enhancement. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 433–442, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2464996.2465022.

[112] Jingling Xue and Qiong Cai. A lifetime optimal algorithm for speculative pre. *ACM Trans. Archit. Code Optim.*, 3(2):115–155, jun 2006. doi:10.1145/1138035.1138036.

[113] Yaohua Yang, Shiqing Zhang, and Li Shen. Control divergence optimization through partial warp regrouping in gpgpus. In *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, CSAI '18, pages 369–374, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3297156.3297201.

[114] Yaohua Yang, Shiqing Zhang, and Li Shen. A lightweight method for handling control divergence in gpgpus. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2019, page 120–127, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293320.3293331.

[115] T. T. Yeh, R. N. Green, and T. G. Rogers. Dimensionality-aware redundant simt instruction elimination. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2020, pages 1327–1340. ACM, 2020. doi:10.1145/3373376.3378520.

[116] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 369–380, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1950365.1950408.

[117] Jianlong Zhong and Bingsheng He. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1522–1532, 2014. doi:10.1109/TPDS.2013.257.

[118] Hucheng Zhou, Wenguang Chen, and Fred Chow. An ssa-based algorithm for optimal speculative code motion under an execution profile. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 98–108, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993498.1993510.