Electronic Preprint for Journal of Information Processing Vol.28

Regular Paper

Branch Divergence Reduction Based on Code Motion

Junji Fukuhara[1,a)] Munehiro Takimoto[1,b)]

Received: August 30, 2019, Accepted: December 20, 2019

Abstract: The Single Instruction Multiple Data (SIMD) execution model on GPUs enables a program to execute efficiently. Nevertheless, the efficiency may decrease via branch divergence that occurs when SIMD threads follow different paths in branches. Once the branch divergence occurs, some threads have to wait until the completion of other threads. This inefficiency on GPU is caused by instructions included in branches, which may be increased by some traditional code optimizations based on code motion. Partial Redundancy Elimination (PRE) is one of such code motions methods. The PRE causes some insertions of expressions into some paths in branches and increases branch divergence. Thus, we propose a new PRE approach, called Speculative Sparse Code Motion (SSCM), which not only removes redundant expressions but also reduces branch divergence. The SSCM achieves them based on both properties of Sparse Code Motion (SCM) that reduces the static numbers of expressions in addition to PRE and speculative code motion that hoists some expressions in branches out of them. The SCM property of SSCM reduces branch divergence since it also hoists all the expressions in the true and false paths in a branch as a single expression. Moreover, the speculation property helps to hoist all the expressions not hoisted by the SCM, which removes more redundant expressions where speculation is not harmful in branches with branch divergence. Furthermore, the SSCM also enables the selective application of speculative code motion to improve programs with divergent and or non-divergent branches. To prove the effectiveness of our method, we applied it to some benchmarks with divergent branches. Our experimental results demonstrate more than 8% improvement in some program efficiency.

Keywords: GPU, branch divergence, code optimization, partial redundancy elimination, speculative code motion

1. Introduction

Generally, GPUs are increasingly used not only for image processing but also for general-purpose applications. Most of GPU technologies implement *Single Instruction Multiple Data* (SIMD) execution model, which executes processes in parallel by applying single instruction to multiple data simultaneously. The SIMD execution model enables a program to execute efficiently, while the efficiency may be decreased by *branch divergence*. The branch divergence occurs when SIMD threads in a warp follow different paths in a branch. In the face of the divergence, some threads have to wait until the completion of the other threads. Thus, the branch divergence results in the decrease of GPU performance. In addition, some traditional code optimizations based on code motion may increase branch divergence. *Partial Redundancy Elimination* (PRE), which is effective code optimization that not only removes partially redundant expressions but also moves invariant expressions out of loops, is one of such code motions approach. The PRE causes some insertions of expressions into some paths in branches which increases branch divergence; hence, makes it difficult to apply PRE to GPU programs.

In this paper, we propose a novel PRE approach, called *Speculative Sparse Code Motion* (SSCM), which not only removes redundant expressions but also reduces branch divergence. The GPU programs execute both true and false sides of divergent branches accordingly. Using this property, the SSCM speculatively hoists the expression that is on one side of a branch before it without a decrease in execution efficiency. Similarly, PRE with speculative code motion, which is called *speculative PRE* (SPRE) [11], [12], speculatively hoists some expressions with less penalty cost based on profile information. The SSCM removes more redundant expressions without the profile information due to the speculative code motion in the divergent branch with no penalty cost. Notice here that the SPRE also has the essential property of PRE, which inserts some expressions into some paths in branches; hence, it cannot be applied to programs easily, including divergent branches. The SSCM has a property of *Sparse Code Motion* (SCM) [1] that reduces the static numbers of expressions in addition to the speculation property. The SCM property of SSCM, which hoists all the possible expressions in both sides of a divergent branch out of it, as a single expression, contributes to reducing the branch divergence. Also, the speculation property enables hoisting expressions missed by SCM, so that the SSCM can remove more redundant expressions. However, the speculative code motion may decrease execution efficiency for non-divergent branches because most of GPU programs have both divergent branches and non-divergent branches. Consequently, the SSCM enables selective application of speculative code motion to improve programs with divergent and or non-divergent branches.

The contributions of this paper are as follows:

(1) The SSCM enables PRE to remove partially redundant expressions in GPU programs without increasing branch divergence.

---

[1] Department of Information Sciences, Graduate School of Science and Technology, Tokyo University of Science, Noda, Chiba 278–8510, Japan

[a)] 6318527@ed.tus.ac.jp

[b)] mune@rs.tus.ac.jp

(2) The SSCM removes more redundant expressions through speculative code motion and contributes to reducing branch divergence by decreasing the static numbers of expressions.

(3) The SSCM selectively applies speculative code motion to divergent branches but not non-divergent branches.

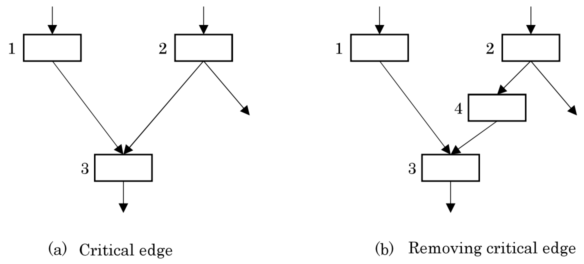
The rest of this paper is organized as follows: Section 2 presents the preliminaries of our approach. Sections 3 and 4 provide brief explanations of the PRE and the branch divergence, respectively. Section 5 describes the SCM and presents the proposed method (i.e., SSCM), which is an improvement of the SCM. Our experimental results are presented in Section 6. Section 7 discusses related works. Finally, Section 8 concludes our paper and presents possible future works.

2. Preliminaries

We assume that a *Control Flow Graph* (CFG) has already been created for each function defined in the source program. The 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 nodes $n \in N$ represents a *basic block* consisting of continuous statements without any branch in the middle. Each edge $(n, m) \in E$ represents the flow of control between basic blocks n and m. $\mathbf{s}$ and $\mathbf{e}$ denote the unique *start node* and *end node* of G. Every node $n \in N$ is assumed to lie on a path from $\mathbf{s}$ to $\mathbf{e}$. p_i denotes the i-th node on the execution path p in a CFG and l_p denotes the length of p. $P[m, n]$ denotes the set of all execution paths from a node m to a node n. $pred(n) =_{df} \{m \mid (m, n) \in E\}$ and $succ(n) =_{df} \{m \mid (n, m) \in E\}$ denote sets of all predecessors and successors of a node n, respectively.

Like the other code motion methods, *critical edges*, which lead from nodes with more than one successor to nodes with more than one predecessor, may block the effective code motion. For example, in **Fig. 1** (a), the edge leading from node 2 to node 3 is a critical edge. We assume that such critical edges are eliminated through inserting a new node as illustrated in Fig. 1 (b).

Also, similarly to Ref. [4], it is assumed that a basic block is divided into two partitions, as shown in **Fig. 2**. The division point of a basic block is defined as the point immediately after the last *modification statement*, which modifies some operands of the considered expression e. The former partition is called the *Entry part* and the latter is called the *Exit part*. If there is no modification statement in the basic block, the entire basic block is defined as an entry part, where 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*. In our algorithm, we insert an expression at either an entry part or an exit part. The insertion points at an entry part and an exit part are called *entry insertion point* and *exit insertion point*, respectively. Therefore, the insertion point is immediately before an entry computation or an exit computation if there is the computation, immediately before the first modification statement if there is no entry computation and there is a modification statement, otherwise the end of that part.



(a) Critical edge (b) Removing critical edge

Fig. 1 Critical edges and their elimination.

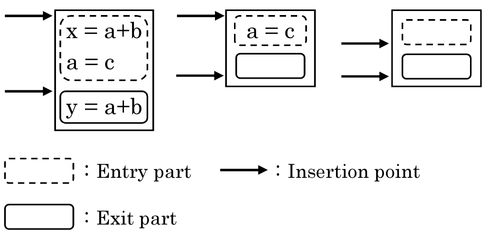


Fig. 2 How to divide a basic block.

3. Partial Redundancy Elimination

If an expression e exists at a program point p and any operands of e are not modified on the execution path P from a program point p to a program point q, e is *available* on P. Furthermore, if e is available on all execution paths from the start node $\mathbf{s}$ to q, e is available at q. Also, if e is available on at least one execution path in the paths from $\mathbf{s}$ to q, e is *partially available* at q. If an expression e is available at a program point q, q is *up-safe* for e. If an expression e exists at a program point p and is available immediately before p, e is fully redundant at p, and can be eliminated by replacing it with the variable that holds the value of e. On the other hand, if an expression e exists at a program point p and is partially available immediately before p, e is partially redundant at p, and cannot simply be removed as with fully redundant expressions.

If an expression e exists at a program point p and any operand of e is not modified on the execution path P from a program point q to a program point p, e is *anticipated* at the program point q on P. If e is anticipated on all execution paths from q to the end node $\mathbf{e}$, e is anticipated at q. Moreover, e is anticipated on at least one execution path from q to $\mathbf{e}$, e is *partially anticipated* at q. If an expression e is anticipated at a program point q, q is *down-safe* for e.

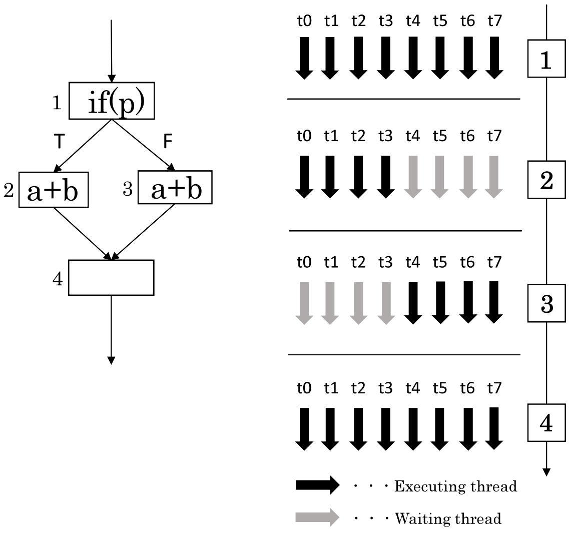
The PRE removes partially redundant expressions by inserting expressions into the appropriate program points. Then, since PRE inserts expressions at the down-safe program points, it can remove partially redundant expressions without increasing the number of executed expressions on any execution path.

When the PRE removes an expression, it stores the value of an available expression in a temporary variable and replaces the expression with the temporary variable. It is important for the temporary variables introduced by PRE to be allocated to registers, so as not to reduce the effectiveness of the PRE. Therefore, in order to suppress register pressure, it is necessary for the PRE not to perform unnecessary code movement with no effect of redundancy removal. Such an extension of PRE is called Lazy Code Motion (LCM) [2]. The LCM shortens the lifetime of the temporary variable introduced for eliminating a partially redundant expression by inserting expressions into the nodes farthest from the start node so that it suppresses some register spills.

4. Branch Divergence

Generally, branch divergence is caused by the SIMD execution model, reducing the execution efficiency of GPU programs. In the SIMD execution model, GPU programs must execute the instructions in both true and false sides of a divergent branch accordingly because each warp, which is a group of threads, has a single control flow.

Example. Consider a conditional branch where the branch divergence occurs at node 1 in **Fig. 3** (a). As shown in Fig. 3 (b), a warp has eight threads, which are represented by arrows. The black arrows represent the executing threads, and the gray arrows represent the waiting threads. Figure 3 (b) shows the case where the threads from t0 through t3 are active at node 2 in the true side, and the other threads are active at node 3 in the false side, depending on the result of p at node 1. In detail, first, all the threads execute instructions at node 1 in parallel. Second, the threads from t0 through t3 execute instructions at node 2 on the true side. Then, the threads from t4 through t7 wait without doing anything. Next, the threads from t4 through t7 execute the instructions at node 3 on the false side. Also, the threads from t0 through t3 wait. Once the execution of the branch is completed, all the threads execute the instructions at node 4. ☐



(a) CFG with divergence (b) Behavior of threads

Fig. 3 CFG with branch divergence and behavior of threads.

Thus, the branch divergence costs time to execute the true and false sides of a branch in order; hence, reducing the execution efficiency. Several methods have been proposed to reduce the branch divergence and improve execution efficiency [3], [8]. However, since traditional code motion-based approaches such as PRE may increase the branch divergence, they cannot simply be applied to programs with branch divergence.

Example. **Figure 4** assumes that the conditional branch at node 6 causes branch divergence. As shown in Fig. 4 (a), expression originally exists at node 6 only on one side of the branch. In this case, applying PRE to Fig. 4 (a) transforms to Fig. 4 (b). In Fig. 4 (b), an expression appears on both sides of the branch, compared to Fig. 4 (a), so that execution efficiency is reduced because of the divergence in node 6. ☐

5. Proposed Method

In this section, first, we explain the SCM that is a traditional method, which decreases the static number of expressions, and then extend it to SSCM.

5.1 Sparse Code Motion

Here, we explain SCM [1], which is the basis of SSCM. We define a concept and describe behavior for an input program in the SCM.

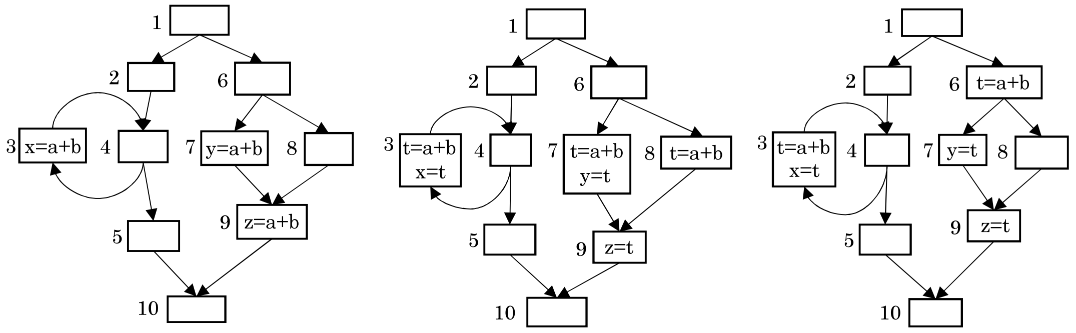
The SCM consists of the following four steps:

(1) Application of LCM; we apply the LCM to an input program.
(2) Computation of a safe moving point; we compute up-safe program points, down-safe points, and earliest points.
(3) Computation of down-safety closure and down-safety region; we compute the down-safety closure $\rho(n)$ and down-safety region R.
(4) Determination of sparse insertion points; we find the insertion points of an expression and transform a program for sparse code motion.

Then, each step is explained below, where we especially give details and significance of (2) and (3) as it applies to SSCM, and then outline (1) and (4).

5.1.1 Application of LCM

The SCM first applies LCM, which suppress unnecessary code motion of the PRE, to an input program. The SCM inserts an expression at the optimal position considering register pressure, as



(a) Original code (b) Effects of PRE (c) Effects of SCM

Fig. 4 Effects of PRE and SCM.

$$\text{(a)}\ \begin{cases} NdSafe(B) = NComp(B) \cup \{Transp(B) \cap XdSafe(B)\} \\ XdSafe(B) = XComp(B) \cup \begin{cases} false & (if\ B = \boldsymbol{e}) \\ \bigcap\limits_{S \in succ(B)} NdSafe(S) & (otherwise) \end{cases} \end{cases}$$

$$\text{(b)}\ \begin{cases} NuSafe(B) = \begin{cases} false & (if\ B = \boldsymbol{s}) \\ \bigcap\limits_{P \in pred(B)} \{XComp(P) \cup XuSafe(P)\} & (otherwise) \end{cases} \\ XuSafe(B) = Transp(B) \cap \{NComp(B) \cup NuSafe(B)\} \end{cases}$$

$$\text{(c)}\ \begin{cases} NEarliest(B) = NdSafe(B) \cap \bigcap\limits_{P \in pred(B)} \overline{\{XuSafe(P) \cup XdSafe(P)\}} \\ XEarliest(B) = XdSafe(B) \cap \overline{Transp(B)} \end{cases}$$

Fig. 5 Dataflow equations for SCM.

well as the LCM. For the details of LCM, see Ref. [2].

5.1.2 Computation of a Safe Moving Point

To compute a safe moving point, we define the following predicates such as NdSafe, XdSafe, NuSafe, XuSafe, NEarliest, and XEarliest using the dataflow equations in **Fig. 5**, where predicates NComp, XComp, and Transp are locally determined. Then, 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)**: It is down-safe at the entry insertion point of B.
- **XdSafe(B)**: It is down-safe at the exit insertion point of B.
- **NuSafe(B)**: It is up-safe at the entry insertion point of B.
- **XuSafe(B)**: It is up-safe at the exit insertion point of B.
- **NEarliest(B)**: It is down-safe at the entry insertion point of B, but an expression cannot be moved to any predecessor of B.
- **XEarliest(B)**: It is down-safe at the exit insertion point of B, but an expression cannot be moved to the entry insertion point of B. □

These predicates and their dataflow equations used in SCM are the same as ones used in LCM [4]. For ease of understanding, we also use notations *Comp*, *DnSafe*, *UpSafe*, and *RelComp* defined as follows:

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

5.1.3 Computation of Down-safety Closure and Down-safety Region

In addition to down-safety, the SCM defines *down-safety closure* $\rho(n)$ and *down-safety region* R. We show the definitions quoted from Ref. [1]. The down-safety closure $\rho(n)$ represents a set of nodes to be considered as movable points of an expression from the CFG nodes n and $succ(n)$. Down-safety region R represents a set of nodes to be considered as movable points of an expression in the entire CFG. These are based on the idea that in adjusting insertion points determined by LCM; we consider down-safe and not up-safe points as hoistable points. Down-safety closure $\rho(n)$ is defined as the minimum set satisfying the following properties for CFG nodes $n \in DnSafe \setminus UpSafe$:

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

Down-safety region R is defined using down-safety closure as follows:

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

Also, the SCM defines a set of the program points *R-Earliest* that is the closest to the start node **s** in down-safety region R. The definition from Ref. [1] is shown below.

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

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

5.1.4 Determination of a Sparse Insertion Point

Using the predicates and down-safety region mentioned above, we find sparse insertion points of expression, transforming a program based on them. First, we select the down-safety region where the number of *R-Earliest* points for *RelComp* points is the fewest. Second, we insert an expression at *R-Earliest* points in the down-safety region and remove expression occurrences at *RelComp* points. Through the transformation, we reduce the number of occurrences of the expression most. In this process, because the SCM inserts expressions into program points in the down-safety region as well as the LCM, it does not perform speculative code motion that may increase the execution of expressions on some execution paths. See Ref. [1] for details of how to find insertion points in SCM.

Example. The LCM transforms a program shown in Fig. 4 (a) into the program shown in Fig. 4 (b). Then, we find the down-safety region for the CFG in Fig. 4 (b) so that we get two regions that consist of node 3 and nodes 6, 7, and 8, respectively. Finally, we find sparse insertion points of expression for the CFG in Fig. 4 (b). Consequently, we get node 6, i.e., the SCM transforms the CFG in Fig. 4 (b) into the CFG in Fig. 4 (c) by inserting an expression into node 6 in Fig. 4 (b), and removing expression occurrences in nodes 7 and 8. Compared with the number of occurrences of the expression in Fig. 4 (b), the SCM can decrease it from three to two, as shown in Fig. 4 (c). Notice that the number of the expression occurrences can decrease to one if insertion to node 1 is allowed. However, because it is not down-safe, it results in speculative code motion, which introduces a new expression on an execution path 1 → 2 → 4 → 5 → 10. The LCM and SCM do not perform such speculative code motion. □

5.2 Speculative Sparse Code Motion

In this section, we extend the SCM described in the previous section to SSCM to allow speculative code motion. Like the SCM, SSCM consists of the following four steps:

(1) Application of LCM,
(2) Computation of a safe moving point,
(3) Computation of down-safety closure and down-safety region, and
(4) Determination of a sparse insertion point.

Steps (1), (3), and (4) are similar to the SCM; we omit their explanation in this section but present step (2) in detail.

5.2.1 Computation of a Safe Moving Point

In SSCM, the dataflow equations shown in Fig. 5 of the SCM are changed to ones in **Fig. 6**. As shown in Fig. 6, Eq. (2) in Fig. 6 (a) is added to the equations of down-safety in Fig. 5 (a), where the other equations are the same as ones in the SCM. Equation (2) in Fig. 6 (a) is computed when 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 expression before the branch does not decrease execution efficiency, which means that an expression can be safely hoisted out of the branch. In SCM, regardless of the branch divergence, a node that is partially anticipated at the exit insertion point is not down-safe. That is, the SSCM considers a larger down-safety region than the SCM does. Particularly, this property enables the SSCM to achieve more sparse code motion than the SCM.

Also, for a non-divergent branch, the SSCM applies the same equation as one in the SCM, as shown in Fig. 6 (a) (3), which does not perform speculative code motion that may decrease the execution efficiency. The extension also contributes to the selective application of SSCM and SCM depending on the occurrence of branch divergence.

(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)

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

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

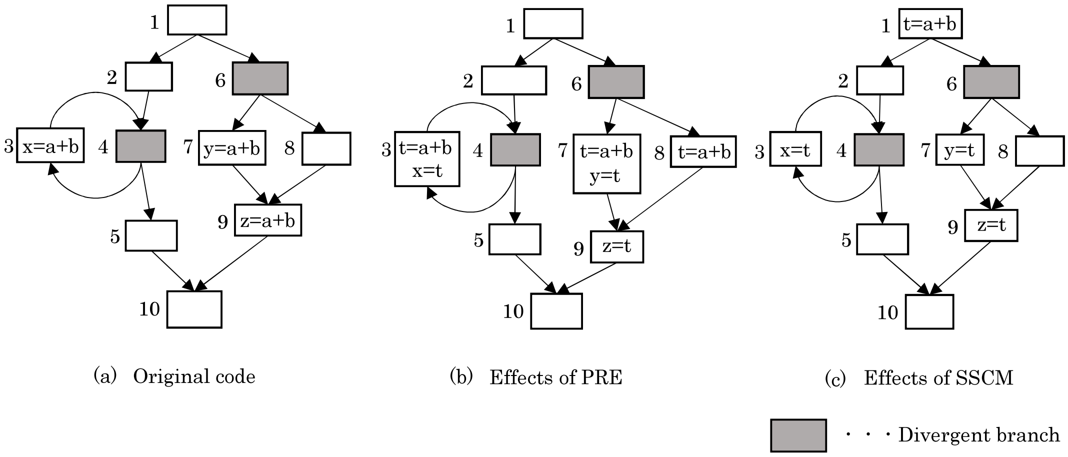
Fig. 6 Dataflow equations for SSCM.

Example. Fig. 7 (a) 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 Fig. 7 (b). Then, computing the dataflow equations in Fig. 6, XdSafe at node 4 becomes true through Fig. 6 (a) (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, we find the down-safety region for the CFG in Fig. 7 (b), which consists of nodes 1, 2, 3, 4, 6, 7, and 8. Exploring the sparse insertion point of an expression in the down-safety region, we find node 1 as the insertion point. Finally, we obtain the CFG shown in Fig. 7 (c) by inserting an expression in node 1 and removing expressions in nodes 3, 7, and 8 from the CFG in Fig. 7 (b). Comparing the number of occurrences of the expression in Fig. 7 (b) with Fig. 7 (c), we find that the number decreases from three to one. Moving $a+b$ at node 3 in Fig. 7 (b) before the branch at node 4 is a speculative code motion that introduces new computations on the execution path 1 → 2 → 4 → 5 → 10. In the 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 Fig. 7 (b) are executed in order. The SSCM hoists such expressions out of the branch with them, as well as SCM, shown in Fig. 7 (c), which directly contributes to reducing branch divergence. □

Although we do not discuss the termination of this algorithm, the SCM and our extended version, SSCM, have monotonicity of the dataflow analysis [13], which guarantees that a solver of the dataflow analysis terminates.

6. Experiments

To evaluate the effectiveness of our method, we conducted two kinds of experiments: a comparison of execution efficiency for the eight benchmarks and a comparison of execution efficiency based on the number of threads. We used the open-source software Ocelot CUDA compiler [9] for the experiments. The Ocelot



(a) Original code (b) Effects of PRE (c) Effects of SSCM

Fig. 7 Effects of speculative sparse code motion.

is a backend for PTX similar to GPU assembly code, and also works as a PTX optimizer. Further, we used the divergence analysis [3] implemented in Ocelot to identify divergent branches. The description of an environment where we conducted 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.

6.1 Experiment A

In the experiment, we implemented the proposed SSCM method and the traditional methods, with which we compared the execution time of object code for the eight benchmarks. The benchmarks are two programs (barnshut, knn) of Treelogy benchmark [5], one program (cfd) of Rodinia benchmark [6], two programs (FDTD3D, eigenvalues) of Nvidia SDK Sample code, and two programs (histo, mri-q) of Parboil benchmark [7]. Moreover, we measured two kernel functions for eigenvalues.

First, we applied these methods to the PTX code obtained by the Nvidia CUDA compiler, nvcc, with optimization option O3. **Figure 8** shows the results of the experiment. In that figure, *O3* represents the execution time of object code generated from the benchmarks with nvcc using the optimization option O3, where *O3* is used as a baseline in the evaluation. The *LCM*, *SCM*, *SPRE*, and *SSCM* represent the execution time when applying LCM, SCM, speculative PRE, and SSCM, respectively, to PTX generated by nvcc with O3. Each result is shown as the ratio of each execution time for *O3*. As shown in Fig. 8, our method achieved more efficient execution than *O3*, *LCM*, *SCM*, and *SPRE* for six programs: barnshut, knn, cfd, FDTD3D, eigenvalues(MultiIntervals), and histo. In particular, we improved by 8.07% with FDTD3D. For the programs of barnshut, knn, FDTD3D, and eigenvalues(MultiIntervals), our method performs speculative code motion that is never performed by the LCM or SCM at divergent branches. Therefore, it could decrease the static number of expressions more in addition to suppressing branch divergence, which contributes to execution efficiency. For the program of cfd, all the methods of *LCM*, *SCM*, *SPRE*, and *SSCM* improved the execution efficiency by removing redundant expressions. Also, comparing *SPRE* with *SSCM*, they have almost the same effect for the programs of barnshut, knn, cfd, histo, and mri-q. However, for the FDTD3D program, the execution efficiency of *SPRE* is lower than the one of *SSCM*. The SPRE performed speculative code motion without considering the occurrences of divergent branches. Besides, our method can improve the execution efficiency by performing speculative code motion for divergent branches. For the program of histo, the execution efficiency decreased when applying the LCM, especially. This is due to the increase of branch divergence by inserting expressions into the divergent branches. However, *SCM*, *SPRE*, and *SSCM* improve the efficiency of histo by hoisting an expression before the branches.

Although our method improved the execution efficiency of the above mentioned six programs, it was not able to improve that of eigenvalues(bisectKernelLarge) and mri-q. The SSCM performs code motion based on the results of divergence analysis, which determines each branch that causes branch divergence in unconservative [3]. Therefore, our method might perform speculative code motion for branches that do not cause branch divergence to decrease the execution efficiency.

6.2 Experiment B

In this experiment, we compared the improvement in execution efficiency by changing the number of blocks and the number of threads. In GPU program, as the number of threads per block 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 GPU. We used the sample program, MCML (Monte Carlo Modeling of Light Transport in Multi-Layered Tissues) [10], which is relatively less tuned than the benchmarks used in Experiment A. We applied our method to the PTX code obtained by using nvcc with the optimization option O3 to the sample program and compared the execution efficiency of them for setting the number of blocks in starting the kernel function to 1, 30, 60, 120, and 240, and setting the number of threads to 32, 64, 128, 192, and 256.

Figure 9 shows the results of the experiment. The execution efficiency decreased when the number of blocks was 30 and the number of threads was 128 and 192, when the number of blocks was 60 and the number of threads was 32, and when the number of blocks 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, we obtained

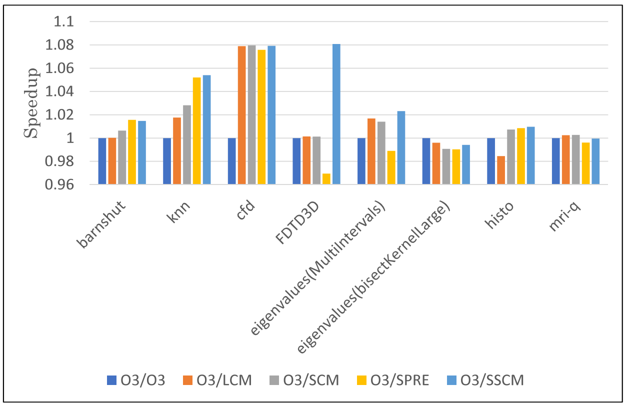


Fig. 8 Experiment A: comparison of execution speed.

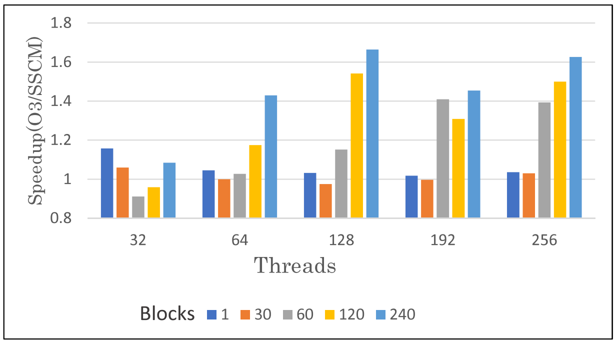


Fig. 9 Experiment B: comparison of execution speed.

66.4% improvement when the number of blocks was 240, and the number 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 block. Then, the reason why the efficiency increased is that the efficiency improved by applying our method and suppressing branch divergence exceeded the efficiency decreased by the lack of register resources.

7. Related Works

The Branch Fusion [3] 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 instruction. However, this method may need to insert new branches and select instructions. The number of the insertions depends on the instruction order in the original branch. Even though there are many expressions with the same operator at the branch, it may be necessary to insert many branches for sequences of expressions. However, since our method hoists expressions with the same operator and operands before a divergent branch, as a single expression, combining it with the branch fusion may suppress more branch divergence than the branch fusion.

The iteration delaying method [8] is applied to the 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 in subsequent iterations. It enables more threads to execute in parallel so that it can suppress branch divergence, and improve execution efficiency. Then, because our method can be applied with the iteration delaying, it is possible to suppress more branch divergence.

Furthermore, the branch distribution method [8] hoists the computations with the same operator in a divergent branch as a single computation out of the branch. When the distribution 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 the target computations are different, it is necessary to introduce temporary variables to retain the suitable values. Thus, the branch distribution may insert many new branches; therefore, it is effective only when the effectiveness of hoisting instructions out of a branch is greater than the cost of the inserted branches and the concentration of hoisted instructions to one place. Applying our method before the branch distribution, it suppresses the insertions of branches for branch distribution by hoisting expressions with the same operator and operand to overcome additional branch divergence.

8. Conclusions

In this paper, we presented an SSCM method to suppress branch divergence by decreasing the static number of expressions without decreasing execution efficiency. Moreover, our method can selectively apply SCM, i.e., a kind of PRE decreasing the static number of expressions and SSCM, i.e., SCM with speculative motion, to the programs with both of non-divergent and divergent branches. We implemented our method and traditional approaches with code motion, and conducted experiments with them. The results show that our method can improve the performance of the programs with branch divergence. However, because our method uses the result of the static divergence analysis, it may apply speculative motion to non-divergent branches. In the future, we hope to perform selective application based on dynamic checking of branch divergence to address the same issue.

References

[1] Rüthing, O., Knoop, J. and Steffen, B.: Sparse Code Motion, *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL 2000*), pp.170–183, ACM (2000).

[2] Knoop, J., Rüthing, O. and Steffen, B.: Lazy Code Motion, *Proc. ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (*PLDI 1992*), pp.224–234, ACM (1992).

[3] Coutinho, B., Sampaio, D., Pereira, M.Q.F. and Meira, Jr., W.: Divergence Analysis and Optimizations, *Proc. 2011 International Conference on Parallel Architectures and Compilation Techniques* (*PACT 2011*), pp.320–329, IEEE (2011).

[4] Nakata, I.: Compiler Construction and Optimization, Asakura Publishing Co., Ltd. (1999).

[5] Hegde, N., Liu, J., Sundararajah, K. and Kulkarni, M.: Treelogy: A Benchmark Suite for Tree Traversals, *Proc. IEEE International Symposium on Performance Analysis of Systems and Software* (*ISPASS 2017*), pp.227–238, IEEE (2017).

[6] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, W.J., Lee, S. and Skadron, K.: Rodinia: A Benchmark Suite for Heterogeneous Computing, *Proc. IEEE International Symposium on Workload Characterization* (*IISWC 2009*), pp.44–54, IEEE (2009).

[7] Stratton, A.J., Rodrigues, C., Sung, I., Obeid, N., Chang, L., Anssari, N., Liu, D.G. and Hwu, W.W.: Parboil: A REvised Benchmark Suite for Scientific and Commercial Throughput Computing, *IMPACT Technical Report, IMPACT-12-01, University of Illinois, at Urbana-Champaign*, March 2012, available from ⟨http://impact.crhc.illinois.edu/parboil/parboil.aspx⟩.

[8] Han, D.T. and Abdelrahman, S.T.: Reducing Branch Divergence in GPU Programs, *Proc. 4th Workshop on General Purpose Processing on Graphics Processing Units* (*GPGPU-4*), pp.1–8, ACM (2011).

[9] Diamos, G., Kerr, A., Yalamanchili, S. and Clark, N.: Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems, *Proc. 19th International Conference on Parallel Architectures and Compilation Techniques* (*PACT 2010*), pp.353–364, ACM (2010).

[10] Ito, T.: Introduction to GPU Programming, Implementation by CUDA5, Kodansha Ltd. (2013).

[11] Cai, Q. and Xue, J.: Optimal and Efficient Speculation-Based Partial Redundancy Elimination, *Proc. International Symposium on Code Generation and Optimization* (*CGO 2003*), pp.91–102, IEEE (2003).

[12] Gupta, R., Berson, A.D. and Fang, Z.J.: Path Profile Guided Partial Redundancy Elimination Using Speculation, *Proc. 1998 International Conference on Computer Languages* (*ICCL 1998*) (1998).

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

Junji Fukuhara received his B.S. degree from Tokyo University of Science in 2018. He is currently in the second year of the master's course at Tokyo University of Science.

Munehiro Takimoto is a professor in the Department of Information Sciences from Tokyo University of Science. His research interests include theory and practice of programming languages, and the various things derived from them, which include mobile agent systems and their applications. He received his Ph.D., M.S., and B.A. in Engineering from Keio University. He is a member of ACM, IEEE Computer Society, IPSJ, JSSST, and IEICE.