Common Subexpression Convergence: A New Code Optimization for SIMT processors

Sana Damani and Vivek Sarkar

Georgia Institute of Technology

Keywords: SIMT processors, thread divergence, compiler optimizations, GPUs

Abstract. On SIMT processors, when threads in a warp diverge at a branch, the hardware scheduler serializes their execution, thereby resulting in reduced SIMT efficiency. We propose a new compiler optimization, Common Subexpression Convergence (CSC), that uses cross-block scheduling to ensure that expression trees that are common across diverged paths are moved to convergent regions and executed by more/all threads in parallel, thereby improving SIMT efficiency and execution time. Our optimization framework is based on a dynamic programming algorithm for finding maximally profitable common expression subgraphs. We also introduce a general approach to test the legality of our optimization based on the program dependence graph, and a heuristic-based cost model to decide when the optimization should be applied. We demonstrate the potential benefits of our approach through a preliminary hand-optimized evaluation using synthetic benchmarks and a BitonicSort example program.

1 Introduction

On Single Instruction Multiple Thread (SIMT) processors, such as a GPU, all threads in a warp execute the same instruction at the same program counter location in parallel on different data. However, when a branch instruction is encountered such that the branch condition evaluates to *taken* for some threads and *not taken* for others, then the threads in the warp are said to have *diverged*. To handle this situation, the hardware scheduler in a SIMT processor serializes the execution of threads within the divergent region, and the warp no longer issues instructions in parallel across threads in a warp, which results in reduced SIMT efficiency and thereby hurts overall execution time [6].

We propose a code motion optimization, called Common Subexpression Convergence (CSC), that helps reduce the cost of thread divergence in programs where *taken* and *not taken* paths of a divergent branch contain common operations. Our optimization moves such operations to a convergent region of execution where more/all threads can execute them in parallel.

For the remainder of this paper, we use Nvidia's thread and warp terminology and profile metrics [8], without loss of generality. To maximize SIMT efficiency, a measure of the proportion of time threads in a warp execute in parallel, we must minimize the number of instructions executed by threads in divergent regions.

The main contributions of this paper include:

- A dynamic programming based algorithm to find profitable common expression subgraphs across divergent branches.
- An approach to testi the legality of CSC based on the Program Dependence Graph (PDG).
- A cost model to decide when CSC is profitable.
- A preliminary hand-optimized evaluation of CSC on synthetic benchmarks and a BitonicSort example program.

We emphasize that this is a work in progress and that the main direction for future work is to build a complete automatic compiler implementation of the CSC optimization, and evaluate it on a large set of GPU benchmark programs.

2 Overview of Approach

In this section, we summarize the three code motion transformations used in our framework for optimizing common subexpression convergence. All three transformations are guaranteed to result in a SIMT efficiency that is greater than or equal to the prior SIMT efficiency. However, it is possible to sometimes see performance degradation due to increases in variable live ranges, introduction of pipeline stalls, or changes in cache behavior.

Hoist. This transformation moves common code to the nearest convergent control flow point before threads diverge at the branch, as illustrated in Listings 1.1 and 1.2. This is only legal if all incoming definitions are located at or before the branch, or can also be hoisted.

Listing 1.1: Before Hoist

```
b = ...;
c = ...;
if (threadId % 2)  {
    a = b * c;
    use a;
} else {
    a = b * c;
}
```

Listing 1.2: After Hoist

```
b = ...;
c = ...;
a = b * c;
if (threadId % 2)  {
    use a;
} else {
    ...
}
```

Sink. This transformation moves common code to the nearest convergent control flow point after threads reconverge at the postdominator, as illustrated in Listings 1.3 and 1.4. This is only legal if all uses of the instruction and redefinitions of its operands can also be moved to the join point.

Listing 1.3: Before Sink

```
c = ...;
if (threadId % 2)  {
    b = 10;
    a = b * c;
} else {
    a = b * c;
}
use a;
```

Listing 1.4: After Sink

```
c = ...;
if (threadId % 2)  {
    b = 10;
    ...
} else {
    ...
}
a = b * c;
use a;
```

Split. To handle operations that can neither be hoisted nor sunk to a convergent region, we introduce an intermediate temporary reconvergence point within the divergent region, as illustrated in Listings 1.5 and 1.6. The common code can be moved to this region before threads diverge once again. As with Hoist and Sink, this transformation is guaranteed to result in a SIMT efficiency that is greater than or equal to the prior SIMT efficiency. However, Split can sometimes introduce more overhead compared to the Hoist and Sink transformations due to branch duplication and extra synchronization instructions.

Listing 1.5: Before Split

```
b = ...;
c = ...;
if (threadId % 2)  {
    a = b * c;
    use a;
} else {
    b = 10;
    a = b * c;
}
use a;
```

Listing 1.6: After Split

```
b = ...;
c = ...;
if (threadId % 2)  {
    ...
} else {
    b = 10;
}
a = b * c;
if (threadId % 2)  {
    use a;
} else {
    ...
}
use a;
```

Extension with Operand Renaming. Statements 2 and 5 in Listing 1.7 can be executed in parallel by different threads because they have the same opcode, even though their source and result operands do not match. This is legal because by virtue of the SIMT execution model, each thread performs the same operation *on different data.*

To optimize this code pattern, we can insert MOV instructions and rename operands so that statements (1) and (2) can be issued as a single instruction as seen in Listing 1.8. However, extra care is needed when performing the code convergence optimization with operand renaming, to account for the extra cost of MOV instructions.

Listing 1.7: Before Renaming

```
if (threadId % 2)  {
    a = b * c;
     ...
} else {
    a = d * c;
    ...
}
use a;
```

Listing 1.8: After Renaming

```
if (threadId % 2)  {
    x = b;
} else {
    x = d;
}
a = x * c;
use a;
```

Problem Statement: *Given a GPU program, identify the best combination of hoist, sink and split optimizations to move common operations to a convergent region such that no dependences are violated.*

Legality: A reordering transformation that preserves every dependence preserves the meaning of the program. As described above, Hoist and Sink are only performed if the transformation preserves all data dependences and Split does not alter the order of statements in the program. CSC is therefore a legal reordering optimization.

3 Algorithmic Details

At a high level, our algorithm finds common expression sub-graphs that are control dependent [2] on the same divergent branch and computes the profitability and legality of code motion. We first describe a dynamic programming solution for optimal sub-graph matching given two computation graphs. Next, we describe a region-based graph traversal approach that incrementally discovers and optimizes divergent branches.

3.1 Common Sub-Expression Detection

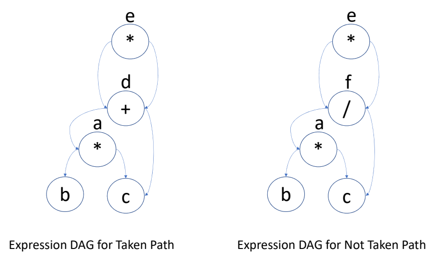
Given two expression DAGs that are control dependent on the same divergent branch, our goal is to detect the maximally profitable matching subgraph. We shall use Listing 1.9 as our running example to explain this algorithm.

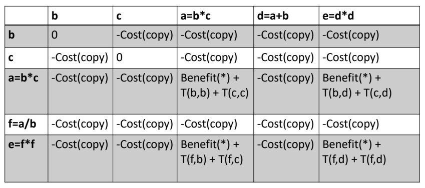
Listing 1.9: Dynamic Programming: Code Example

```
if (condition) { a = b * c;    d = a + c;    e = d * d; }
else           { a = b * c;    f = a / c;    e = f * f; }
```

To support operand re-use, we use expression DAGs rather than trees as shown in Figure 1a. As in other work, we formally define the expression DAG to be a directed graph G such that leaves represent variables, interior nodes represent operators, and an edge from a to b indicates that b is an input to a. We assume an SSA representation so that each variable is defined only once. Our goal is to find a subgraph matching that maximizes (Benefit - Cost), where Cost and Benefit are defined as follows:

- Cost = Increase in number of copy instructions needed to handle operand renaming.
- Benefit = Decrease in number of divergently executed instructions. The benefit of moving a function call or loop into common code path may be much


e
*
d
+
a
*
b
c
Expression DAG for Taken Path
e
*
f
/
a
*
b
c
Expression DAG for Not Taken Path
(a) Expression graph for Listing 1.9



	b	c	a=b*c	d=a+b	e=d*d
b	0	-Cost(copy)	-Cost(copy)	-Cost(copy)	-Cost(copy)
c	-Cost(copy)	0	-Cost(copy)	-Cost(copy)	-Cost(copy)
a=b*c	-Cost(copy)	-Cost(copy)	Benefit(*) + T(b,b) + T(c,c)	-Cost(copy)	Benefit(*) + T(b,d) + T(c,d)
f=a/b	-Cost(copy)	-Cost(copy)	-Cost(copy)	-Cost(copy)	-Cost(copy)
e=f*f	-Cost(copy)	-Cost(copy)	Benefit(*) + T(f,b) + T(f,c)	-Cost(copy)	Benefit(*) + T(f,d) + T(f,d)

(b) Dynamic Programming for Listing 1.9

Fig. 1: Dynamic Programming

greater than the benefit of moving an arithmetic instruction, and we therefore introduce a weighted model for benefit computation, biased towards convergent execution of more expensive instructions.

Our dynamic programming solution, while optimal for the given cost model, may not always generate optimal code due to the limitations of the heuristic-based cost mode. In particular, we do not take into consideration the impact of code motion on scheduling and register pressure in Hoist and Sink transforms, or the cost of additional branch and synchronization instructions in the Split transform. Furthermore, Split may result in suboptimal application of compiler optimizations due to smaller basic blocks. We leave the exploration of these additional heuristics to future work and focus only on copy instructions in our cost model.

Table entries: Let $T(i,j)$ be the cost of matching node i and node j.

Initialization: $T(i,i)$ where i is a leaf node is trivially matched as having 0 cost while $T(i,j)$ where both i and j are leaf nodes is matched as having $+1$ cost required for the copy instruction.

Recurrence: We then define the recurrence to compute the profit of matching interior nodes, $T(i,j)$ as:
$T(i,j) =$ cost of copy insertion, if i and j have an operator mismatch
$T(i,j) =$ benefit of matching operator $+$ sum(max(match subgraph, insert copy)) where, i and j belong to different expression DAGs.

Figure 1b shows the application of our dynamic programming approach on the program described in Listing 1.9.

3.2 Code Motion

We begin by building a program structure tree (PST) which is a hierarchical representation of single-entry single-exit regions of a control flow graph as described in [5]. We traverse this PST from leaves to the root, thereby ensuring that the innermost regions are handled first. This helps optimize programs with nested conditionals and loops.

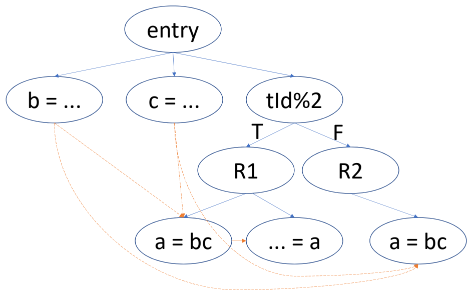
At each region, we build a program dependence graph, a directed graph where nodes represent instructions in the region and edges between nodes represent control or data dependence between instructions. We use the control dependence edges to find operations that are control dependent on true and false edges of the same divergent branch, and data dependence edges to generate expression graphs and determine legality of code motion. Figures 2a and 2b represent the PDGs for Listing 1.1 and Listing 1.2 respectively.

```
Input   : Program P, threshold
Output: Optimized Program P'
Function CommonSubexpressionConvergence(P)
    BuildPST();
    for each region R in the PST in bottom up order do
        BuildPDG(R);
        if BranchIsDivergent(R.root) then
            for each instruction I1 backwards in R.child0 do
                for each instruction I2 backwards in R.child1 do
                    D1 = BuildDAG(I1);
                    D2 = BuildDAG(I2);
                    Profit = MatchProfit(D1, D2);
                    if Profit < threshold then
                        continue;
                    if LegalToHoist(D1, D2) then
                        Hoist(D1, D2, R);
                    else if LegalToSink(D1, D2) then
                        Sink(D1, D2, R);
                    else
                        Split(D1, D2, R);
        FlattenBranches();
```

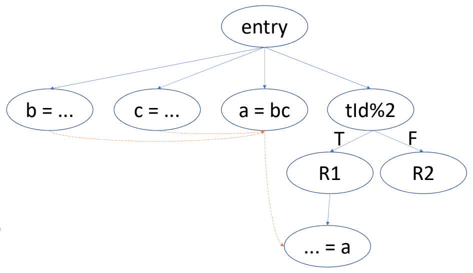
Algorithm 1: Overall Algorithm for Common Subexpression Convergence

Next, we determine if the branch condition is divergent by checking if the branch depends on a thread-varying value using the method described in [1]. Finally, after each region is processed, we flatten the branches within the region to handle common sub-regions nested within outer regions. Branch flattening or predication eliminates branches in the region and converts control dependences to data dependences. Our algorithm treats these flattened branches as regular expression trees when handling the parent region. See Listing 1.11 for an example where branch flattening enabled additional optimization.

Branch flattening is insufficient for loops nested within regions which require special handling. When comparing two sub-regions within a region, we compare the loop bodies and iteration domains to detect common code and opportunity for optimization. If the loop iteration domains are non-identical, we may use loop peeling or index-set splitting to generate identical loops that can be hoisted. If the loop bodies are only partially common, we use loop distribution to separate out the common operations before performing code motion.



(a) PDG for Listing 1.1 before Hoisting.



(b) PDG for Listing 1.2 after Hoisting.

Fig. 2: PDGs before and after Hoist (solid and dashed edges correspond to control and data dependences respectively)

Listing 1.10: Nested Common Code

```
if (threadId % 2)  {
    // common code
    a = b * c;
    use a;
} else {
    if (threadId % 3) {
        // common code
        a = b * c;
    }
}
```

Listing 1.11: Branch Flattening

```
if (threadId % 2)  {
    p = true
} else {
    p = threadId % 3;
}
// flattened common code
(p) a = b * c;
if (threadId % 2)  {
    use a;
}
```

3.3 Time Complexity Analysis

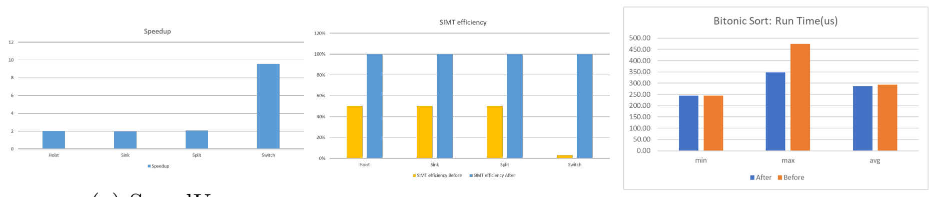
We present an approximate solution to optimal expression DAG matching using a heuristic-based dynamic programming approach. The 2-dimensional table *Profit* takes O(n^2) time to build, where n is the number of instructions within an expression DAG. This procedure is repeated for each instruction within the divergent branch. Hence, for each region R, the algorithm takes O(n^2) time in the worst case because the *Profit* table caches matching data, thereby ensuring that each pair of instructions within the divergent branch is only compared once. If the number of regions is m, the overall optimization takes O(mn^2) time.

4 Preliminary Results

We evaluated our algorithm by hand transformations on microbenchmarks designed to provide optimization opportunities for the (1) hoist, (2) sink, and (3) split transformations, as well as for the use of (4) a multi-way switch statement. Figure 3 shows increased SIMT efficiency, fewer memory operations and improved runtimes. In particular, moving code to convergent paths reduced the number of DRAM reads by up to 26×. We validated these results by created an automated implementation in the LLVM compiler for the Hoist transformation, which delivered comparable performance to our hand-coded implementation. For BitonicSort, our results show variability in execution times, with improvements in Max times and no degradation in Min and Average times with CSC. Listing 1.12 shows the relevant step of bitonic sort which has common code across divergent branches for a swap operation. We use branch flattening and sinking to move the common code to a convergent path and test our optimization using the GPU implementation of bitonic sort available on GitHub at [3].

Listing 1.12: Bitonic Sort

```
bitonic_sort(ascending)
{
    if (ascending) {
      // sort in ascending order
      if (array[i]>array[j]) {
        swap(array[i],array[j]);    // common across diverged paths
      }
    }
    else {
      // sort in descending order
      if (array[i]<array[j]) {
        swap(array[i],array[j]);    // common across diverged paths
      }
    }
}
```



(a) SpeedUp (b) SIMT efficiency (c) Bitonic sort: Runtime

Fig. 3: Benchmark Results

5 Related Work

Branch Fusion. Coutinho et al describe branch fusion, a divergence optimization that uses a variation of sequence alignment, called instruction alignment, to identify common code across diverged paths [1] and merge divergent code using the *Split* transformation. They do not consider the use of code hoisting or sinking as a means to avoid the introduction of additional branch instructions. Further, their instruction alignment algorithm is restricted to common code discovered in a specific order, a limitation overcome by our graph-based approach.

Branch Distribution. Han and Abdelrahman propose branch distribution [4], an optimization to reduce branch divergence by factoring out structurally similar code from diverged paths thereby reducing the total number of dynamic instructions executed in the divergent region. They do not however provide an algorithm to detect and move common code in divergent branches; also, it is unclear whether their approach extends to loops and nested conditionals.

Partial Redundancy Elimination. CSC is similar to PRE in that they both aim at eliminating redundant execution of instructions in conditional code regions using code motion. However, while PRE eliminates redundant recomputation of a scalar value in single-threaded execution [7], CSC eliminates redundant re-execution of a vector operation by preventing temporal misalignment of an instruction across threads in a warp.

6 Conclusion and Future Work

In conclusion, we find that significant opportunity exists to improve SIMT efficiency in parallel programs with divergent branches. We have described a new $O(mn^2)$ time code motion optimization called Common Subexpression Convergence (CSC), which is designed to improve SIMT efficiency by moving common code from divergent to convergent paths of execution.

We believe that further opportunities for CSC may be found when targeting multicore CPU code (written using OpenCL or OpenMP, for example) to run on GPUs. Unlike hand-written CUDA code that is optimized for execution on the GPU, automatically generated GPU code from multicore CPU code may be suboptimal since divergence does not pose a performance penalty on multicore CPUs. Whereas common code across conditional branches is not redundant in CPU execution, the same code may perform poorly on SIMT architectures without the application of CSC. Additionally, the optimization can further be extended to interprocedural common subexpression convergence using analysis across function calls.

References

1. B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. M. Jr. Divergence analysis and optimizations. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011.
2. R. Cytron, J. Ferrante, and V. Sarkar. Compact representations for control dependence. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, 1990.
3. M. Endler. Bitonic Sort on CUDA.
4. T. D. Han and T. S. Abdelrahman. Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, 2011.
5. R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994.
6. E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 2008.
7. E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 1979.
8. NVIDIA. *SIMT Efficiency.*