Fusion of calling sites

Douglas do Couto Teixeira Sylvain Collange Fernando Magno Quintão Pereira
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais, Brazil
{douglas,sylvain.collange,fernando}@dcc.ufmg.br

***Abstract*—The increasing popularity of Graphics Processing Units (GPUs), has brought renewed attention to old problems related to the Single Instruction, Multiple Data execution model. One of these problems is the reconvergence of divergent threads. A divergence happens at a conditional branch when different threads disagree on the path to follow upon reaching this split point. Divergences may impose a heavy burden on the performance of parallel programs. In this paper we propose a compiler-level optimization to mitigate this performance loss. This optimization consists in merging function call sites located at different paths that sprout from the same branch. We show that our optimization adds negligible overhead on the compiler. It does not slowdown programs in which it is not applicable, and accelerates substantially those in which it is. As an example, we have been able to speed up the well known SPLASH Fast Fourier Transform benchmark by 11%.**

***Keywords*-compilers; parallelism; optimization**

I. Introduction

Graphics Processing Units (GPUs) are becoming a staple hardware in the high-performance world. They provide a simple, cheap, and efficient platform in which parallel applications can be developed [1]. Since the release of CUDA, in early 2006 [2], a plethora of programming patterns and algorithms have been designed to run in this environment, touching multiple fields of knowledge, including Biology, Chemistry and Physics [3].

The basic operating principle of this hardware consists in running the threads of *Single Program, Multiple Data* (SPMD) programs in lockstep, so to execute their identical instructions on *Single Instruction, Multiple Data* (SIMD) units. This execution model is, nowadays, known as *Single Instruction, Multiple Threads* (SIMT), a term coined by Nvidia's engineers [1]. SIMT execution has gained momentum beyond the graphics processing ecosystem. SPMD programming environments like OpenCL[1], OpenACC[2] or OpenMP 4.0[3] can target SIMD architectures like GPUs, multi-core CPUs with SIMD extensions, and even Intel Xeon Phi accelerators.

Nevertheless, in spite of all these advances, programming SPMD applications for SIMD architectures remains a challenging task. One of the reasons behind this difficulty is a phenomenon known as *Thread Divergence*. When facing a conditional branch, two threads diverge if they disagree on which path to take. Divergences are a problem because they have an impact on the program's performance. In other words, a divergence splits threads into two groups, upon reaching a conditional branch. Only one of these groups contain threads that do useful work at a given point in time.

We have designed, implemented and tested a compiler optimization that mitigates this performance loss. We name this optimization *Fusion of Calling Sites* (FCS). Our optimization relies on a simple idea: threads should enter functions in lockstep to minimize the effects of divergences. Therefore, whenever a function is invoked at the two different paths that stem from a conditional test, we merge the two calling sites into one single invocation of that function. This optimization can benefit implicit SIMD architectures, such as those found in GPUs, and explicit SIMD hardware like the Xeon Phi. In the latter case, the compiler merges threads together to form SIMD instruction, handling divergence with mask-predicated instructions [4].

As we show in Section III, our algorithm scans blocks of code within the program, performing the merging whenever it is possible. In this paper, we demonstrate that our optimization is: (i) easy to implement, (ii) innocuous when non-applicable and (iii) effective when used. Our optimization has low computational complexity in practice. In other words, it always applies a constant number of operations per pair of calling sites that it merges. If a program does not present any opportunity for this merging to happen, then we do not impose any runtime overhead onto the compiler, nor onto the executable program, once it is deployed. In Section IV, we show the potential of our optimization through a toy benchmark, and show its applicability in the well-known implementation of Fast Fourier Transform available in

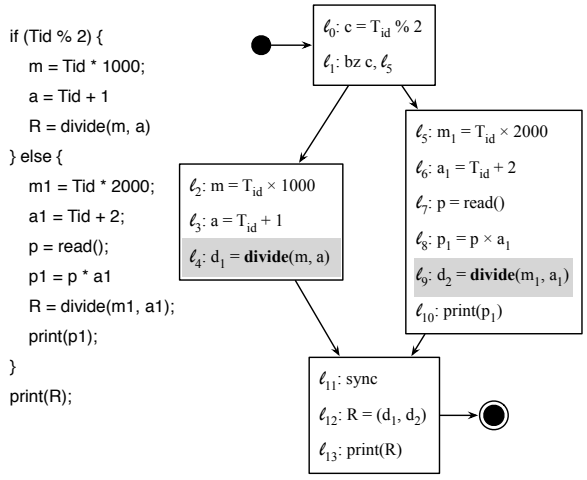
[1]http://www.khronos.org/opencl/
[2]http://www.openacc.org/
[3]http://openmp.org/

SPLASH[4]. In the former benchmark, FCS reduces the number of divergent instructions by 55%, and on the latter by 11%.

II. Overview of the Approach

Figure 1 will let us illustrate thread divergences. This phenomenon characterizes the Single Instruction, Multiple Data execution model typical of Graphics Processing Units. These processors organize threads in groups that execute in lockstep. Such groups are called

[4]http://www.capsl.udel.edu/splash/



Cycle	Instruction	t_0	t_1	t_2	t_3
14	$c = \texttt{T}_{id}\%2$	✓	✓	✓	✓
15	$\texttt{bz}\ c, then$	✓	✓	✓	✓
16	$m = \texttt{T}_{id}\ \times\ 1000$	✓	•	✓	•
17	$a = \texttt{T}_{id}\ +\ 1$	✓	•	✓	•
18	$d1 = divide(m, a)$	✓	•	✓	•
...					
118	$m1 = \texttt{T}_{id}\ \times 2000$	•	✓	•	✓
119	$a1 = \texttt{T}_{id}\ +\ 2$	•	✓	•	✓
120	$p = read()$	•	✓	•	✓
121	$p1 = p\ \times a1$	•	✓	•	✓
122	$d2 = divide(m, a)$	•	✓	•	✓
...					
222	$print(p1)$	•	✓	•	✓
223	sync	✓	✓	✓	✓
224	$R = phi(d1, d2)$	✓	✓	✓	✓
225	$print(R)$	✓	✓	✓	✓

Figure 1: (Top) A program whose performance may experience a slowdown due to divergences. (Bottom) An execution trace of the program. If a thread t executes an instruction at cycle j, we mark the entry (t, j) with ✓. Otherwise, we mark it with •. In this example we assume that each invocation of function `divide` takes one hundred cycles to execute.

warps in NVIDIA's jargon, or *wavefronts* in AMD's. We can imagine that threads in the same warp use different arithmetic and logic units, but share the same instruction control logic. Control flow divergences happen when threads in a warp follow different paths after processing the same branch. If the branching condition is data divergent, then it might be true to some threads, and false to others. In face of divergences, some threads will take the "then" part of the branch in Figure 1, and others will take the "else" part. Due to the shared instruction control logic, only one group of threads will be allowed to do useful work at a given instant. The execution trace at the bottom of Figure 1 shows which threads are active at each cycle, assuming an architecture that allows four threads simultaneously in flight.

When two threads diverge, the hardware should reconverge them as earlier as possible to maximize the amount of active workers per cycles. A *reconvergence point* is the earliest instruction in the program where we can expect control flow paths to join regardless of the outcome or target of the divergent branch. Fung *et al.* have shown that the *post-dominator* of a branch is – usually – the best place to reconverge threads [5]. We say that a node v in a CFG post-dominates a node u if any path from v to the end of the CFG must go across u. In Figure 1, basic block `end` is the post-dominator of every other block. Yet, as Fung *et al.* themselves have also shown, reconverging threads at the post-dominators of branches is far from being a perfect solution to divergences. Figure 1 illustrates this situation particularly well.

The `divide` function is invoked at both sides of the branch in Figure 1. Even though this function must be executed by all the threads that reach the divergent branch, these threads will be entering the function at different execution cycles due to the divergence. Consequently, the instructions that constitute function `divide` will be called twice: once for the threads in the "then" part of the branch, and another time for the threads in the "else" part. In this case, reconverging threads at post-dominators of divergent points will not avoid the redundant execution of `divide`. If `divide` runs for a long time, then we will be missing the opportunity to share many execution cycles among different threads. The goal of FCS is to reconverge threads at the entry points of functions. We will accomplish this goal by changing the structure of the program's control flow graph, as we will explain in the next section.

III. Fusion of Calling Sites

Figure 2 provides a high-level view of FCS. The function **merge_call_site** tries to join call sites, until

this action is no longer possible. If a merging happens, then the function invokes itself recursively, otherwise the optimization terminates. Candidate branches are found via the function **find_joinable_calls**, which is also depicted in Figure 2. This procedure looks for paths that stem from the same branch ℓ_b and that lead to different calls of the same function F.

```
merge_call_site:
  input: Program P
  output: Program P'
  if (ℓb, ℓ1, ℓ2) = find_joinable_calls(P):
    P' = merge_cfg(P, ℓb, ℓ1, ℓ2)
    return merge_call_site(P')
  else:
    P' = P
    return P'

find_joinable_calls():
  input: Program P
  output: (Label ℓb, Label ℓ1, Label ℓ2)
  for each branch ℓb in P:
    let ℓp = post-dominator of ℓb
    if ∃ path t1 from ℓb to ℓp
      ∃ path t2 from ℓb to ℓp
      t1 ∩ t2 = ∅
      t1 ⊃ call to F at ℓ1
      t2 ⊃ call to F at ℓ2
      return (ℓb, ℓ1, ℓ2)
```

Figure 2: Main routines that perform the fusion of calling sites.

We use the program from Figure 1 as an example to illustrate our transformation. The program in that figure has one candidate branch, at label ℓ_1. Our function **find_joinable_calls** will detect two paths from this branch leading to invocations of the same function. The first path is formed by the sequence of labels $\ell_1 \rightarrow \ell_2 \rightarrow \ell_3 \rightarrow \ell_4$. The second path if formed by the sequence $\ell_1 \rightarrow \ell_5 \rightarrow \ell_6 \rightarrow \ell_7 \rightarrow \ell_8 \rightarrow \ell_9$. After finding the paths, our routine **merge_cfg** will produce a new version of the program.

Figure 4 provides an overview of the transformation that **merge_cfg** performs. This function creates a common label, e.g., ℓ_r, that will join the two call sites that we want to fuse. We use the ϕ-functions of the *Static Single Assignment* [6] (SSA) form to join function arguments. SSA form is a program representation in which each variable has only one definition site [6].

```
1  merge_cfg():
2    input: Program P, Label ℓb , ℓ1 , ℓ2
3    output: Program P'
4    let r1 = F(p11, ..., p1n) be at ℓ1 in P
5    let r2 = F(p21, ..., p2n) be at ℓ2 in P
6    let p1 = φ(p11, p21)
7    ...
8    let pn = φ(p1n, p2n)
9    replace ℓ1 by "goto ℓr" in P
10   replace ℓ2 by "goto ℓr" in P
11   create "ℓr: p1; ...; pn; r = F(p1, ..., pn);
12   create "ℓr+1: branch equal to ℓb
13           targeting such(ℓ1) and succ(ℓ2)
14   rename every use of r1 to r in P
15   rename every use of r2 to r in P
16   P' = strictify_program(P)
17   return P'
```

Figure 3: Routine that transforms the program's control flow graph. We let succ(ℓ) be the unique successor of label ℓ.

Nowadays, almost every compiler uses this intermediate representation to manipulate programs. The SSA format relies on ϕ-functions to join different variables into common names. Going back to Figure 4, an instruction such as $a_1 = \phi(a_{11}, a_{21})$ will assign to variable a_1 the value of a_{11} if the program flow reaches that operation through label ℓ_1, and will assign a_{21} to a_1, if the program flow comes through label ℓ_2. Figure 5 shows the program that we obtain after applying the FCS optimization onto the function seen in Figure 1. This time we have only one invocation site for function **divide**, which will be reached independent on the way that we branch at ℓ_1. The branch immediately after the new label ℓ_r is used to preserve the program flow after the execution of **divide**.

Ensuring Strictness: We notice that the transformed program contains a path in which variable p_1 is used without being defined: $\ell_0 \rightarrow \ell_1 \rightarrow \ell_2 \rightarrow \ell_3 \rightarrow \ell_4 \rightarrow \ell_r \rightarrow \ell_{r+1} \rightarrow \ell_10$. In this case we say that the program is not *strict*. Strictness is a very important requirement imposed by the Static Single Assignment form. It ensures the key SSA property: the definition of a variable dominates all its uses. After our transformation, we may have programs that are not strict, as we have seen in the example. To obtain strictness back, we apply the function **strictify_program** in the transformed code. This function inserts dummy definitions to all the

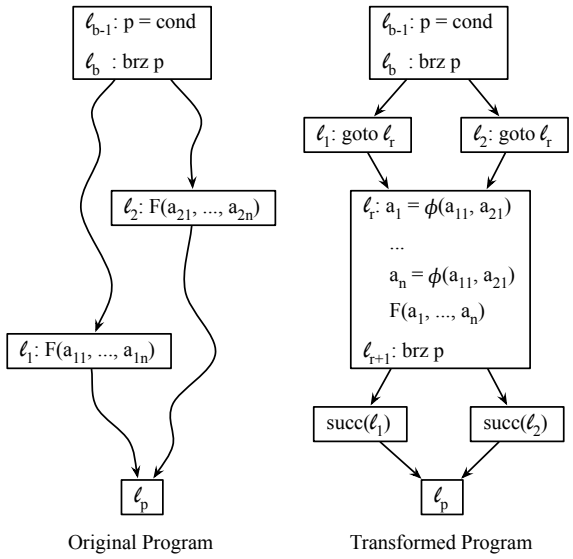


Figure 4: Overview of the transformation that we perform to join function calls.

variables defined within the scope of the branch, and that were used after the fused call. In our example, p_1 is the only such variable. If a variable is used at one side of the branch, then the dummy definition is inserted in the other side. Figure 6 shows this transformation.

Termination: The function **merge_call_site** always terminates due to a simple argument: the fusion of two function call sites do not enable the fusion of further calls. In other words, if a program has a number N of branches that pass the profit test performed by the function **find_joinable_calls**; then no more than N branches will be fused by our optimization. Therefore, as the number of branches in a program is limited, our algorithm is guaranteed to terminate.

Complexity: We call function **merge_call_site** recursively at most once per potentially profitable branch in the program. Each call of this function scans all the conditional tests in a program, looking for the most profitable fusion (see line 14 of **find_joinable_calls** in Figure 2). If we have $O(N)$ blocks in the program, we may have to inspect $O(N)$ branches. Each inspection is $O(N)$, as it involves a traversal of the paths that sprout away from the conditional. If two calls must be merged, then we resort to function **merge_cfg**, whose complexity is bound by **strictify_program**. This last function is standard in compilers, and runs in $O(N)$. Therefore, our functions (**merge_call_site**, **merge_cfg** and **strictify_program**) run – together – in $O(N^4)$.

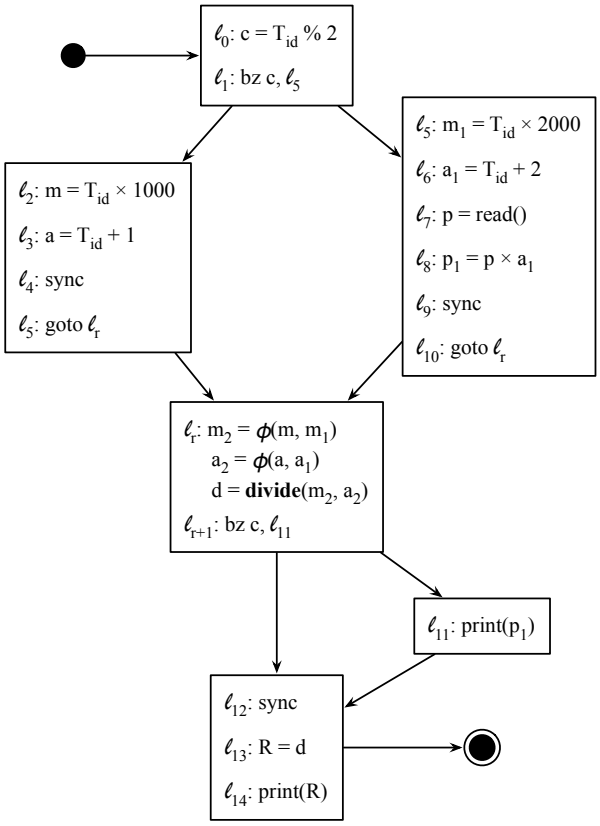


Figure 5: Transformed version of the program earlier seen in Figure 1.

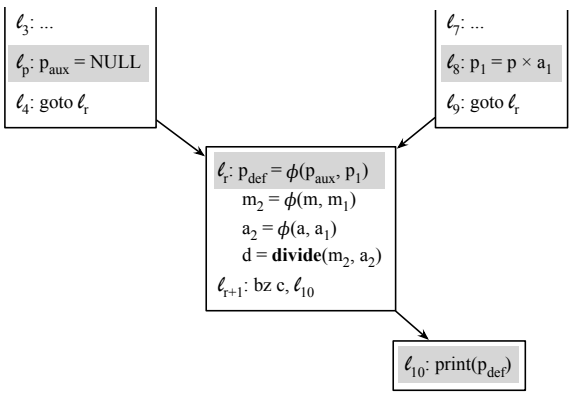


Figure 6: New definitions of p_1 to ensure strictness.

This complexity may seem very high at a first glance. However, in practice only a handful of branches lead to two different calls of the same function. Furthermore, if we order the branches by profit, then we can inspect

each one of them in line 14 of **find_joinable_calls** at most once. Therefore, in practice our optimization runs in $O(N^2)$, where N is the number of basic blocks in the program code.

Cycle	Instruction	t_0	t_1	t_2	t_3
14	$c = \mathtt{T}_{id}\%2$	✓	✓	✓	✓
15	$\mathtt{bz}\ c, then$	✓	✓	✓	✓
	...				
16	$m = \mathtt{T}_{id}\ \times\ 1000$	✓	•	✓	•
17	$a = \mathtt{T}_{id}\ +\ 1$	✓	•	✓	•
	...				
25	$m1 = \mathtt{T}_{id}\ \times 2000$	•	✓	•	✓
26	$a1 = \mathtt{T}_{id}\ +\ 2$	•	✓	•	✓
27	$p = read()$	•	✓	•	✓
28	$p1 = p\ \times\ a1$	•	✓	•	✓
	...				
45	sync	✓	✓	✓	✓
46	$P = phi(m, m1)$	✓	✓	✓	✓
47	$P2 = phi(P, P1)$	✓	✓	✓	✓
48	$d1 = divide(P, P1)$	✓	✓	✓	✓
	...				
49	$print(P2)$	•	✓	•	✓
	...				
51	sync	✓	✓	✓	✓
52	$print(d1)$	✓	✓	✓	✓

Figure 7: (Top) The same program from Figure 1 after being optimized. (Bottom) An execution trace of the program. If a thread t executes an instruction at cycle j, we mark the entry (t, j) with the symbol ✓. Otherwise, we mark it with •.

IV. Experimental Results

Experimental setup.: The fusion of calling sites may be applied onto SPMD programs running on SIMD machines, following the SIMT execution model. There are several different computer architectures that fit into this model, from GPUs and vector units (SSE, MMX, etc) to Long's Minimal Multi-Threading architectures [7]. As the transformation is performed at source code level, our technique makes no assumption about the way the SPMD code is later transformed into SIMD instructions. Evaluating FCS separately on each programming environment and each platform would be tedious. Furthermore, this approach would produce results that are hard to generalise. Therefore, we chose to evaluate FCS on general-purpose parallel applications in a micro-architecture-agnostic simulator which models an ideal SIMT machine. This simulator has been implemented by Milanez *et al.* [8], who have made it publicly available. The simulator is implemented on top of the PIN binary instrumentation framework[5]. The Pin tool reads the binary and produces traces representing every instruction that each thread executes. Then, we replay the traces using different heuristics (that we describe in the next paragraph) to re-converge threads. To perform the code transformation, we have used the LLVM compiler [9]. Our performance numbers have been obtained in the following way: we run the PIN-based simulator on the original program that LLVM produces at its -O3 optimization level. Then, we apply FCS on that binary, and re-run the simulator.

Heuristics for Thread Reconvergence: our simulator accepts different thread reconvergence heuristics. Such heuristics determine the next instruction to be fetched in an SIMT architecture. We chose to simulate four heuristics: MinPC, MinSP-PC, MaxFun-MinPC, and Long-MinSP-PC. These heuristics are described below:

Min-PC: this technique, due to Quinn *et al.* [10], is the thread reconvergence heuristics typically adopted in the implementation of graphics processing units: in face of divergent lines of execution, the fetcher always chooses the heuristics with the smallest Program Counter (PC) to process. The rational behind this heuristics is simple: in the absence of backward branches, given two program counters: n and $n+1$, the latter will be executed after the former. By fetching the instruction at the lowest PC, the hardware maximizes the chance of keeping the threads in lockstep execution.

MinSP-PC: this approach was proposed by Collange [11] and is built on top of Quinn's Min-PC heuristic [10]. It is used in programs that contain function invocations, and its bedrock is the fact that when a function is called, the stack grows down. Thus, threads running more deeply nested function calls have smaller Stack Pointers (SP). Based on this observation, this heuristic fetches instructions to threads with the smallest SP, because it assumes that these threads are behind in the program's execution flow.

MaxFun-MinPC: this heuristic, Maximum Function Level - Minimum PC, is similar to MinSP-PC, but instead of choosing the smallest SP, it chooses the thread that has the highest number of activation records on the stack. Therefore, this heuristic would have the same behavior of MinSP-MinPC if all activation records had the same size.

Long: Long *et al.* [7] have created a heuristic to analyze redundancies in SIMD programs. Their key idea is to add memory to each thread. One thread uses the memory of the others to advance or stall. If the current PC of a thread t_0 is in the recent history of another thread t_1, then thread t_0 is probably behind t_1. In this case, t_0 needs to progress to catch up with t_1. The idea of Longs heuristic can be used to create other heuristics. When multiple threads have the same highest priority, the original heuristic executes these threads alternately, but the variations of Long's heuristic use other policies to choose the next thread to execute. In this paper we used Long's with Min-PC: when multiple threads have

[5]http://www.pintool.org/

Benchmark	LoC	Inst	Trace	Merge
Divide	51	112	581	1
FFT	1,291	2,854	697	4
Fluidanimate	5,712	6,357	5,586	3
Swaptions	1,309	3,742	1,123	3

Figure 8: The benchmarks that we have analyzed. **LoC**: number of lines of code, including comments; **Inst**: number of assembly instructions; **Trace**: number of instructions in millions, that each benchmark executes with its standard input **Merge**: number of call sites that we have merged.

> the highest priority, the basic block of the threads with smallest PC is executed first. Then, all priorities are recounted again for the next choice.

The Benchmarks: To probe the effectiveness of the FCS optimization, we chose to apply it on general purpose SPMD applications from the PARSEC/SPLASH benchmark suite [12]. We have used three programs from these collections: `FFT`, `Fluidanimate` and `Swaptions`. Figure 8 shows some characteristics of these benchmarks. These are the PARSEC programs that we manage to compile using LLVM 3.4 without having to acutely modify the benchmark's source code. These programs are large, and contain only a handful of branches that touch the same function call through different program paths. Therefore, the benefits that we can expect from the application of FCS on these benchmarks is limited. Hence, to demonstrate the possibilities of our optimization, we shall add to this suite the program first seen in Figure 1.

Performance analysis: Figure 9 shows the result of combining our optimization with different heuristics and different numbers of available threads. Numbers above bars show relative speedup compared to not using our optimization. We performed these experiments on an Intel Xeon CPU E5-2620 2.00GHz processor with 16 GB of DDR2 RAM running Linux running Ubuntu 12.04 (Kernel 3.2.0). Nevertheless, our results do not dependend on these features, as they have been produced through simulation. Our experiments let us draw some conclusions. MinPC-based heuristics (MinSP-PC and MinPC) tend to benefit more from FCS. This advantage exists because such heuristics favour the synchonization of threads before a function call. For instance, in Figure 5 both, MinPC and MinSP-PC, will fetch the instructions in all the smaller labels, e.g., $\ell_2 - \ell_9$, before grabbing ℓ_r, which lays further ahead in the program's binary layout. Consequently, the heuristic published by Long *et al.* does not benefit as much, because it has

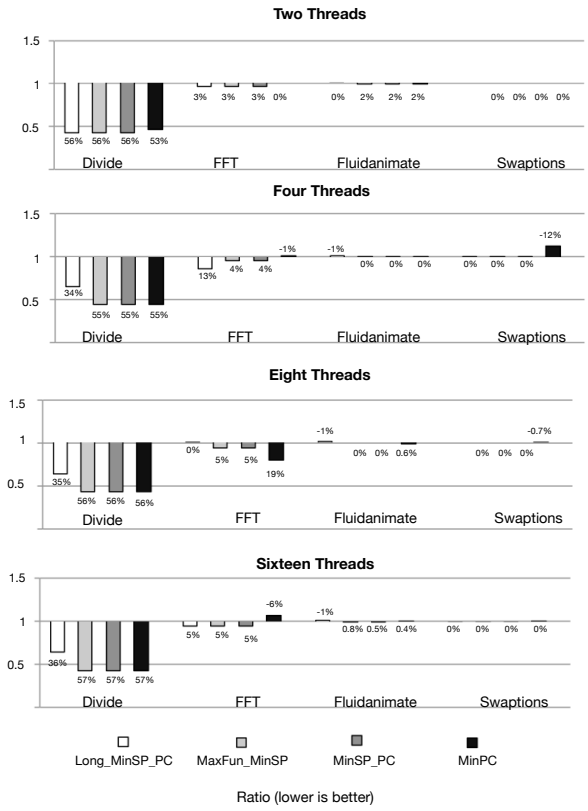


Figure 9: Execution time reduction after applying FCS.

been designed in a way that is totally oblivious to the invocation of functions. Figure 9 also shows that our optimization does not impact negatively the benchmarks that we have tested. The only exceptions are due to MinPC in Swaptions (four threads) and in FFt (16 threads). This negative impact is due to back-edges, e.g., jumps that lead the program's flow back to the beginning of a loop. In face of repeat-style iterators, which test the exit condition at the end, MinPC may fail to reconverge threads within the loop. In this case, the slightly larger code that we produce ends up causing an increase on the number of instructions that are not shared among threads. We have not observed this behavior in the other heuristics.

On the applicability of Fusion Call Fusion: The programs seen in Figure 9 contain only a few situations in which it is possible to merge function calls to reduce the effects of divergences. Yet, the pattern that is needed to enable FCS is not rare. To demonstrate this last statement, we have performed a study on the programs available in the SPEC CPU 2006 CINT suite, and in the LLVM test suite. These programs cannot

Benchmark	Instrs	Branches	Candidates
mcf	2,560	183	1
libquantum	6,446	317	15
astar	8,662	534	10
bzip2	17,439	1,052	6
sjeng	30,275	3,238	30
omnetpp	91,822	4,211	67
hmmer	67,735	4,992	43
h264ref	144,266	7,570	122
gobmk	145,670	13,326	95
xalancbmk	593,895	26,146	758
perlbench	284,039	26,651	228
gcc	801,918	78,316	935

Figure 10: Applicability of the fusion of calling sites in the integer programs available in SPEC CPU 2006.

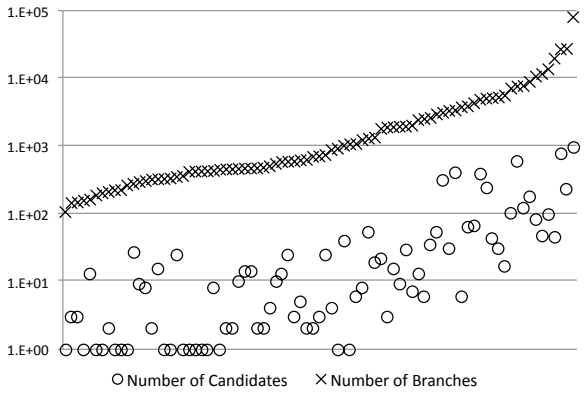


Figure 11: Comparison between number of Branches and number of candidate branches. Each tick on the X-axis represents a program with more than 100 branches.

benefit from FCS, because they are not coded to run in parallel. Nevertheless, they will gives us an idea about how applicable is our optimization. Table 10 shows the number of branches, and the number of branches that are *candidates* for FCS. A branch is a candidate if it gives origin to two paths along which the same function is called at least once. We found 2,310 candidates among 166,536 branches; hence, about 1.4% of the branches found in SPEC CINT are candidates to FCS.

This proportion of candidates can also be observed in smaller programs. For instance, Figure 11 shows the number of branches and candidates in the 84 programs present in the LLVM test suite, including SPEC CPU 2006, that have more than 100 branches. In total, we analyzed 308,999 branches and found out 5,333 candidates. The highest proportion of candidate branches has been observed in SPEC CFLOAT `soplex`, which gave us 398 candidates out of 3,298 branches.

V. Related Work

Other Divergence Aware Optimizations: this paper introduces a new optimization to mitigate the performance loss caused by divergences in GPGPU applications. There are a number of different optimizations that serve the same purpose; however, they reduce divergences in different ways [13], [14], [15], [16], [17], [18]. For instance, Han *et al.*'s [15] *Branch Distribution* hoists instructions up or down divergence paths to join them at common program points – we can perform this merging in the middle of divergence paths. *Branch Fusion* [14] is a generalization of Branch Distribution; however, it does not merge function calls. In other words, the optimization of Coutinho *et al.* bails out when faced with divergent branches that contain call instructions – this is the exact case that we handle. Another optimization in this group is if-conversion. However, if-conversion requires evaluating both sides of the conditional branch in every case, possibly discarding the computations of the other side. In SIMT architectures, only paths that are executed by at least one thread are visited. This implies that branches paths are kept in different basic blocks throughout compiler transformations, so optimizations have to work at the CFG level. If-conversion merge paths to form a single basic block, enabling the use of straight-forward intra-block optimizations.

There are other divergence aware optimizations that target loops, instead of branches, as we do. For instance, Carrillo *et al.* [13] have designed a code transformation called *Branch Splitting*, which divides parallelizable loops enclosing multi-path branches. In this way, they produce multiple loops, each one with a single control flow path. In similar lines, Lee *et al.* [16] have proposed *Loop Collapsing*, a technique that reduces divergences by combining multiple divergence loops into common iterators. Han *et al.* [15] have further extended Lee's approach with the notion of *Iteration Delaying*. This transformation recombines loops containing divergent branches, so that threads tend to remain together for a longer time. None of these optimizations is designed to handle function calls specifically, and, more importantly: none of them would be able to carry out the optimization that we discuss in this paper.

Function-Aware Heuristics to Reconverge Threads: there exist different heuristics implemented at the hardware level that enforce early reconvergence of divergent threads [19], [20], [21], [7]. In particular, Milanez *et al.* [8] have proposed the Min-SP-PC technique, one of

the heuristics that we use in this paper. We emphasize that our work is orthogonal and complementary to these research efforts. Our optimization can be applied on programs independent on the heuristic used to reconverge threads. Nevertheless, as we have observed in Section IV, some of these heuristics yield greater benefit when combined with our approach.

VI. Conclusion

This paper has introduced Fusion of calling sites, a new compiler optimization that mitigates the negative impact caused by divergences on applications running in SIMD fashion. This optimization consists in rearranging the control flow graph of a program, so to merge different function call sites at common program points. In this way, the merged function can be invoked together by divergent threads. There exists, presently, a great deal of effort to develop techniques, at the hardware and software level, to reduce the effects of divergences. Our work is complementary to these efforts: our gains are cumulative with the increasing performance of graphics cards, and it adds a negligible cost over compilation time. More importantly, we believe that optimizations such as Fusion of calling sites contribute to shield application developers from particularities of the parallel hardware, such as divergence and reconvergence of threads.

Software: the software used in this paper, including our simulator and binary instrumentation tool, is available at https://github.com/dougct/function-call-fusion.

References

[1] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE Micro*, vol. 30, pp. 56–69, 2010.

[2] M. Garland, "Parallel computing experiences with CUDA," *IEEE Micro*, vol. 28, pp. 13–27, 2008.

[3] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, "A survey on parallel computing and its applications in data-parallel problems using GPU architectures," *Communications in Computational Physics*, vol. 15, no. 2, pp. 285–329, 2014.

[4] R. Karrenberg and S. Hack, "Improving performance of opencl on cpus," in *CC*. Springer, 2012, pp. 1–20.

[5] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *MICRO*. IEEE, 2007, pp. 407–420.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *TOPLAS*, vol. 13, no. 4, pp. 451–490, 1991.

[7] G. Long, D. Franklin, S. Biswas, P. Ortiz, J. Oberg, D. Fan, and F. T. Chong, "Minimal multi-threading: Finding and removing redundant instructions in multi-threaded processors," in *MICRO*. IEEE, 2010, pp. 337–348.

[8] T. Milanez, S. Collange, F. M. Q. Pereira, W. Meira, and R. Ferreira, "Thread scheduling and memory coalescing for dynamic vectorization of SPMD workloads," *Parallel Computing*, vol. 40, no. 9, pp. 548–558, 2014.

[9] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*. IEEE, 2004, pp. 75–88.

[10] M. J. Quinn, P. J. Hatcher, and K. C. Jourdenais, "Compiling C* programs for a hypercube multicomputer," *SIGPLAN Not.*, vol. 23, pp. 57–65, 1988.

[11] S. Collange, "Stack-less SIMT reconvergence at low cost," ENS Lyon, Tech. Rep., 2011.

[12] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *PACT*. ACM, 2008, pp. 72–81.

[13] S. Carrillo, J. Siegel, and X. Li, "A control-structure splitting optimization for gpgpu," in *Computing frontiers*. ACM, 2009, pp. 147–150.

[14] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. M. Jr., "Divergence analysis and optimizations," in *PACT*. IEEE, 2011, pp. 320–329.

[15] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in gpu programs," in *GPGPU-4*. ACM, 2011, pp. 3:1–3:8.

[16] S. Lee, S.-J. Min, and R. Eigenmann, "Openmp to gpgpu: a compiler framework for automatic translation and optimization," in *PPoPP*. ACM, 2009, pp. 101–110.

[17] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen, "Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping," in *ICS*. ACM, 2010, pp. 115–126.

[18] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for GPU computing," in *ASPLOS*. ACM, 2011, pp. 369–380.

[19] M. Dechene, E. Forbes, and E. Rotenberg, "Multi-threaded instruction sharing," North Carolina State University, Tech. Rep., 2010.

[20] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili, "SIMD re-convergence at thread frontiers," in *MICRO*. ACM, 2011, pp. 477–488.

[21] J. González, Q. Cai, P. Chaparro, G. Magklis, R. Rakvic, and A. González, "Thread fusion," in *ISLPED*. ACM, 2008, pp. 363–368.