Fusion of calling sites
Douglas Do Couto Teixeira, Caroline Collange, Fernando Magno Quintão Pereira

To cite this version:
Douglas Do Couto Teixeira, Caroline Collange, Fernando Magno Quintão Pereira. Fusion of calling sites. International Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD), Oct 2015, Florianópolis, Santa Catarina, Brazil. 10.1109/SBAC-PAD.2015.16 . hal-01410221

HAL Id: hal-01410221
https://hal.science/hal-01410221
Submitted on 6 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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%.

1 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-

---

\(^{1}\)http://www.khronos.org/opencl/

\(^{2}\)http://www.openacc.org/

\(^{3}\)http://openmp.org/
predicated instructions [4].

As we show in Section 3, 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 4, 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 SPLASH\textsuperscript{4}. In the former benchmark, FCS reduces the number of divergent instructions by 55\%, and on the latter by 11\%.

2 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 \textit{warps} in NVIDIA’s jargon, or \textit{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 \textit{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 \textit{v} in a CFG post-dominates a node \textit{u} if any path from \textit{v} to the end of the CFG must go across \textit{u}. In Figure 1, basic block \textit{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

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{(Top) A program whose performance may experience a slowdown due to divergences. (Bottom) An execution trace of the program. If a thread \textit{t} executes an instruction at cycle \textit{j}, we mark the entry \((t, j)\) with √. Otherwise, we mark it with ⬤. In this example we assume that each invocation of function \textit{divide} takes one hundred cycles to execute.}
\end{figure}

\footnote{\url{http://www.caps1.udel.edu/splash/}}
3 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 \( \ell_b \) and that lead to different calls of the same function \( F \). If the cost of transforming the program’s control flow graph is lower than a certain threshold, then the prospective branch is sent to function `merge_cfg`, which is in charge of transforming the program.

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 \( \ell_1 \). 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 \( \ell_1 \to \ell_2 \to \ell_3 \to \ell_4 \). The second path if formed by the sequence \( \ell_1 \to \ell_5 \to \ell_6 \to \ell_7 \to \ell_8 \to \ell_9 \). 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., \( \ell_r \), that will join the two call sites that we want to fuse. We use the \( \phi \)-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]. Nowadays, almost every compiler uses this intermediate representation to manipulate programs. The SSA format relies on \( \phi \)-functions to join different variables into common names. Going back to Figure 4, an instruction such as \( a_{11} = \phi(a_{11}, a_{21}) \) will assign to variable \( a_{11} \) the value of \( a_{11} \) if the program flow reaches that operation through label \( \ell_1 \), and will assign \( a_{21} \) to \( a_{11} \) if the program flow comes through label \( \ell_2 \). 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 \( \ell_1 \). The branch immediately after the new label \( \ell_r \) 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: \( \ell_0 \to \ell_1 \to \ell_2 \to \ell_3 \to \ell_4 \to \ell_r \to \ell_{r-1} \to \ell_{l0} \). In this case we say that the program is not strict. Strictness is a very important requirement imposed by the Static Single Assignment (SSA) form to ensure that each variable is defined only once.
merge_cfg():
  input: Program P, Label l₀, l₁, l₂
  output: Program P'
  let r₁ = F(p₁₁, ..., p₁ₙ) be at l₁ in P
  let r₂ = F(p₂₁, ..., p₂ₙ) be at l₂ in P
  let p₁ = F(p₁₁, p₂₁)
  ...
  let pₙ = F(pₙ₁, pₙ₂)
  replace l₁ by "goto l₄" in P
  replace l₂ by "goto l₄" in P
  create "l₄: p₁₁; ..., pₙ₁; t = F(p₁, ..., pₙ)"
  create "l₄₊₁: branch equal to l₁"
  targeting such(l₁) and succ(l₂)
  rename every use of r₁ to r in P
  rename every use of r₂ to r in P
  P' = strictify_program(P)
  return P'

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

merge_call_site():
  input: Program P, two function call sites R₁, R₂
  output: Program P'
  let R₁ = F(p₁₁, ..., p₁ₙ) be at R₁ in P
  let R₂ = F(p₂₁, ..., p₂ₙ) be at R₂ in P
  let p₁ = F(p₁₁, p₂₁)
  ...
  let pₙ = F(pₙ₁, pₙ₂)
  let r₁ = F(p₁₁, ..., p₁ₙ) be at l₁ in P
  let r₂ = F(p₂₁, ..., p₂ₙ) be at l₂ in P
  let p₁ = F(p₁₁, p₂₁)
  ...
  let pₙ = F(pₙ₁, pₙ₂)
  replace l₁ by "goto l₄" in P
  replace l₂ by "goto l₄" in P
  create "l₄: p₁₁; ..., pₙ₁; t = F(p₁, ..., pₙ)"
  create "l₄₊₁: branch equal to l₁"
  targeting such(l₁) and succ(l₂)
  rename every use of r₁ to r in P
  rename every use of r₂ to r in P
  P' = strictify_program(P)
  return P'

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

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 variables defined within the scope of the branch, and that were used after the fused call. In our example, p₁ 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_sites 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_sites recursively at most once per potentially profitable branch in the program. Each call of this function scans all the conditional tests in a program, look-
ing for the most profitable fusion (see line 14 of \texttt{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 traversing the branches that sprout away from the conditional. If two calls must be merged, then we resort to function \texttt{merge.cfg}, whose complexity is bound by \texttt{strictify_program}. This last function is standard in compilers, and runs in $O(N)$. Therefore, our functions ($\texttt{merge.call_site}$, \texttt{merge.cfg} and \texttt{strictify_program}) run \texttt{N} – together – in $O(N^4)$.

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 \texttt{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.

4 Experimental Results

\textbf{Experimental setup.} The fusion of call 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 Milanes 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.

\textbf{Heuristics for Thread Reconvergence:} our simulator accepts different thread reconvergence heuris-

\begin{figure}[h]
\centering
\begin{tabular}{|l|l|l|l|l|}
\hline
\textbf{Cycle} & \textbf{Instruction} & \textbf{t_0} & \textbf{t_1} & \textbf{t_2} & \textbf{t_3} \\
\hline
14 & $c = T_d \times 2$ & ✓ & ✓ & ✓ & ✓ \\
15 & \texttt{bz } c, \texttt{then} & ✓ & ✓ & ✓ & ✓ \\
\hline
16 & $m = T_d \times 1000$ & ✓ & ● & ✓ & ● \\
17 & $a = T_d + 1$ & ✓ & ● & ✓ & ● \\
\hline
25 & $m_1 = T_d \times 2000$ & ● & ✓ & ● & ✓ \\
26 & $a_1 = T_d + 2$ & ● & ✓ & ● & ✓ \\
27 & $p = \text{read}()$ & ● & ✓ & ● & ✓ \\
28 & $p_1 = p \times a_1$ & ● & ✓ & ● & ✓ \\
\hline
45 & \texttt{sync} & ✓ & ✓ & ✓ & ✓ \\
46 & \texttt{phi}(m, m_1) & ✓ & ✓ & ✓ & ✓ \\
47 & \texttt{phi}(P,P_1) & ✓ & ✓ & ✓ & ✓ \\
48 & \texttt{divide}(P,P_1) & ✓ & ✓ & ✓ & ✓ \\
\hline
49 & \texttt{print}(P_2) & ● & ✓ & ● & ✓ \\
\hline
51 & \texttt{sync} & ✓ & ✓ & ✓ & ✓ \\
52 & \texttt{print}(d_1) & ✓ & ✓ & ✓ & ✓ \\
\hline
\end{tabular}
\caption{(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 ●.}
\end{figure}

$^5$\url{http://www.pintool.org/}
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. 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 apply it also onto 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 synchronization 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 $\ell_r$, 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 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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LoC</th>
<th>Inst</th>
<th>Trace</th>
<th>Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divide</td>
<td>51</td>
<td>112</td>
<td>581</td>
<td>1</td>
</tr>
<tr>
<td>FFT</td>
<td>1,291</td>
<td>2,854</td>
<td>697</td>
<td>4</td>
</tr>
<tr>
<td>Fluidanimate</td>
<td>5,712</td>
<td>6,357</td>
<td>5,586</td>
<td>3</td>
</tr>
<tr>
<td>Swaptions</td>
<td>1,309</td>
<td>3,742</td>
<td>1,123</td>
<td>3</td>
</tr>
</tbody>
</table>

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.
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.

![Figure 9: Execution time reduction after applying FCS.](image)

5 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.

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 Reconcure Threads: there exist different heuristics implemented at the hardware level that enforce early re-convergence of divergent threads [19, 20, 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 reconcile threads. Nevertheless, as we have observed in Section 4, some of these heuristics yield greater benefit when combined with our approach.

6 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](https://github.com/dougct/function-call-fusion).

**References**


