Survey on Analysing and Optimizing the LLVM Divergence Analysis

Yashwant Singh[1], Gunashekar Reddy K[2], Mohammed Tajuddin[3]

[1,2,3]*Department of Computer Science Dayananda Sagar College of Engineering, Bengaluru, India*

---***---

Abstract—In contrast to previous generation designs, which utilized special-purpose hardware units, the present generation of Graphics Processing Units (GPUs) comprises a large number of general-purpose processors. This fact, together with the widespread use of multi-core general purpose CPUs in modern machines, argues that performance-critical software like digital forensics tools should be massively threaded to make the most of all available computational resources. The LLVM Core libraries offer a modern source and target-independent optimizer, as well as code generation support for a wide range of CPUs. These libraries are based on the LLVM intermediate representation (LLVM IR), which is built around well-defined code representation. To make the code execute or compile quicker, an LLVM backend performs ”passes” on the intermediate code representation (LLVM IR). The LLVM backend introduces a pass called ”Divergence Analysis,” which is a combination of Divergence Analysis and Syncdependence Analysis, to aid the compiler in determining the divergent instruction ahead of time. This paper conducts a comprehensive review of the literature in order to address new ideas and difficulties in the field of divergence analysis, as well as the adoption of better methodologies. This was accomplished by evaluating previously published articles. This review’s findings show a variety of techniques, recent advances, and current approaches for analysing and optimizing the LLVM Divergence Analysis. Based on the findings, we make conclusions on the efficacy of LLVM Divergence Analysis from the results discussed in the selected review articles, and the report identifies future research challenges and demands for optimizing the Divergence Analysis.

***Index Terms*—CPU, GPU, Compiler, Multi-core machines, SPMD instruction, SIMD instruction, Thread, LLVM, Optimization pass, Divergence Analysis**

I. INTRODUCTION

For execution on a CUDA core, threads from a block are bundled into fixed-size warps, and threads within a warp must follow the same execution path. All threads must execute the same instruction at the same time i.e., threads cannot diverge within a warp. Branching for conditionals in an if- then-else statement is the most typical code construct that can cause thread divergence. When certain threads in a warp evaluate to ’true’ while others evaluate to ’false,’ the ’true’and ’false’ threads branch to distinct instructions. Some threads will want to go to the ’then’ command, while others will want to go to the ’else’ instruction. We can assume that statements in the then and else clauses should be executed in parallel. This, however, is not possible due to the need that threads in warp cannot diverge. The CUDA platform has workarounds that fix the problem, but it has a negative impact on performance. Divergence analysis is used by vec- torizing compilers to determine whether programme points have uniform variables, meaning that all SPMD threads that execute this programme point have the same value. They use uniformity to keep branching and defer calculations to scalar processor units to avoid branch divergence. Non-interference and binding time are closely related to divergence, which is a hyper property. There are various divergence, binding time, and non-interference studies already available, but in order to attain soundness, they either sacrifice precision or impose considerable limits on the program’s syntactic structure.

The phrase divergence stems from two different lanes on a branch whose predicate is variable, i.e., non-uniform, taking different objectives. As a result, in the literature, the phrases uniform, variable, and non-divergent, divergent are occasion- ally used interchangeably. The term uniformity is used in this study to underline that the purpose of the analysis is to find uniform variables. Non-uniform branch variables are the cause of branch divergence. The most important part of divergence analysis is selecting the proper collection of control-flow split nodes that influence the uniformity of the abstract transformers for control flow joins and loop exits. Existing analyses simplify the problem in a variety of ways, for example, by putting severe limits on the program’s syntactic structure, such as supporting only structured syntax (while loops, if-then-else) or prohibiting branches with more than two successors. The majority of formal treatments of divergence analysis rely on structured syntax established inductively.

II. MOTIVATION

The motivation for this literature survey comes from the shortcoming of the current Divergence Analysis pass and see- ing what all work has been gone into writing the current pass. Before we go into actual shortcoming, let's first dive into what divergence is and how it is calculated. Any instruction that evaluates to different values for different threads is considered as divergent, since in a multi-threaded program all threads execute an instruction in a single lock step whenever a program counter hits a divergent instruction it needs to be handled carefully. For example if we have a divergent instruction as a if condition which evaluates to true for some threads and false for other threads in that case the threads are no longer synchronized and are awaiting to execute different instructions. One way the compiler handles this is by keeping executing instructions in order and if some threads don't have to execute certain instructions it switches off those threads and turns them on their relevant instructions and switches off the other threads, this process is called masking.

For masking to work, the compiler needs to know be- forehand which all instructions will be divergent for optimal handling of thread divergence. This is where the Divergence Analysis pass comes in. The Divergence analysis pass is target independent analysis pass i.e does not make any changes to instructions or the control flow graph of the program unlike transformation passes and marks all the appropriate instructions divergent.

The current algorithm for marking an instruction divergent was introduced in the paper [7]Improving Performance of OpenCL on CPUs by Ralf Karrenberg and Sebastian Hack. The paper introduces three scenarios when an instruction can be marked divergent. If an instruction i1 is marked diver- gent(for example some instructions as inherently divergent such as an instruction is directly dependent on the thread id which evaluates to different values for different threads) then another instruction i2 can be marked divergent if one of the below 3 conditions are satisfied.

- Instruction i2 is directly dependent on instruction i1 for example consider the below code snippet

  ```
  %i1 = call i32 @llvm.amdgcn.workitem.id()

  %i2 = icmp eq i32 %0, %i1
  ```

  if i1 is divergent and i2 is direct successor of it hence will also be marked divergent.

- i1 is a terminating condition in a block and there exist at least 2 disjoint paths starting from i1 and ending at i2, we call it as sync dependent divergence.

For example consider Fig.1, The branch condition in block A is divergent since both the value it depends on is divergent (%p), hence there are 2 different routes a thread can take to reach :join block i.e via block B and block C and in both the cases the variable %x will be assigned different value depending on the path thread took. Hence it will be marked divergent.

- i1 is involved in the exit condition of a loop and causing threads to exit the loop at different blocks, if you roll the loop out this looks similar to sync dependent divergence. For example consider Fig.2, If the loop exit condition in block C is divergent then the thread can take any combination of paths given in the right figure hence it will cause divergence in block D, since it is reached by different threads at different times.

III. CONTRIBUTION

The current divergence analysis pass is not iterative, what we mean by that is once the pass gives the result of running the Divergence Analysis pass on the instruction set any changes made to the current instruction set no matter how small by

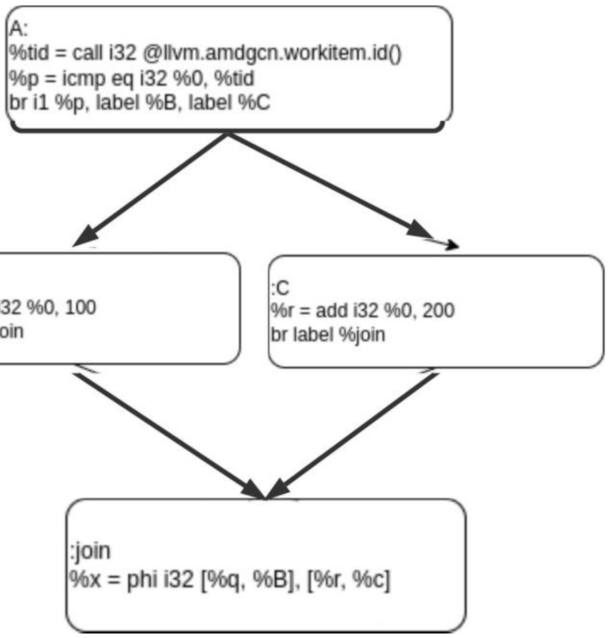
A:
%tid = call i32 @llvm.amdgcn.workitem.id()
%p = icmp eq i32 %0, %tid
br i1 %p, label %B, label %C
:B
%q = add i32 %0, 100
br label %join
:C
%r = add i32 %0, 200
br label %join
:join
%x = phi i32 [%q, %B], [%r, %c]

Fig. 1. Code Snippet

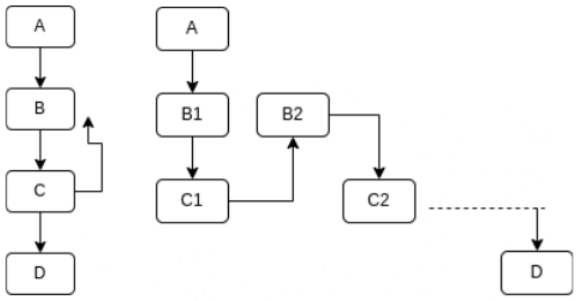

Fig. 2. Control Flow Structure

Either the compiler while doing other optimization or by the user will cause the current results to be nullified and hence we need to compute the pass result again. We aim to propose a solution to this problem by analysing the working of the current Divergence analysis pass and proposing a new way to compute the analysis more efficiently to accommodate changes being made to the control flow graph on the go. We here present our all related findings that are previously done on the Divergence and Divergence analysis pass which will be essential in proposing a new efficient solution.

IV. RELATED WORK

1. The methods given allow parallel loops and serial sections to be executed repeatedly without any blocking or explicit serialisation. It is possible to combine computations involving sections that are mutually independent. Processes are synchronised in such a way that they just wait for work in a specific section to end, not for every process running theprogram to arrive at the synchronisation point, hence we don't have process joins except at barrier points. A delayed process will not prevent other processes from executing the program. When a program is assigned many processes and run on a multiprocessor machine, the job execution time can be sped up.

Authors	Technique/problem targeted	Performance gains/result
F.Darema, D.A.George, V.A.Norton and G.F.Pfister [1]	SPMD model for Fortran	Parallelization of programs and with keeping check on perfor- mance issues
Chris Lattner, Vikram Adve [2]	Original proposal for LLVM com- piler	Description and design of LLVM compiler tackling lot of previous compiler problems
Jaewook Shin [3]	More efficient SIMD code generation in a vectorizing compiler	Speedup factor of 1.99 over previ- ous technique
Torben Amtoft[4]	Slicing	Irrelevant loop elimination
Ralf Karren- berg, Sebas- tian Hack [5]	Whole function vectorization on SSA code emitted by LLVM	Average speedup factor of 3.9 on different OpenCL kernels
Ralf Karren- berg, Sebas- tian Hack [6]	OpenCL/CUDA performance on CPU	Speedup factor of 1.21 against naive vectorization, 1.15–2.09 against suited kernels and 2.5 speedup factor against intel driver
Aniruddha S. Vaidya, Anahita Shayesteh, Dong Hyuk Woo, Roy Saharoy, Mani Azimi [7]	Optimising SIMD vectorization	Reduction in execution cycles in divergent applications by upto 42% and on subset of divergent work- loads 7% reduction on today's gpu and 18% on future gpus
Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, Steve Zdancewic [8]	LLVM optimiza- tions	The SSA based extraction gener- ates code which is on par with LLVM's unverified implementation
Diogo Sam- paio,Rafael Martins de Souza,Caroline Col- lange,Fernando Magno Quintão Pereira [9]	Divergence anal- ysis on SIMD ex- ecution model	Divergence aware spiller produced 26.51% faster GPU code
Ralf Karren- berg [10]	SIMD Vectoriza- tion of SSA bases CFGs	Consistent improved performance of the generated vectorized code
Anirban Ghose, Soumyajit Dey, Pabitra Mitra, Mainak Chaudhuri [11]	OpenCL workloads partitioning	Improved partition results over pre- vious ML based static partitioning
Charitha Saumya, Kirshanthan Sundarara- jah, and Milind Kulkarni [12]	SIMT thread di- vergence reduc- tion	A new analysis/transformation tool(DRAM) that reduces performance decline caused by CFG divergence

2. Working prototype of the low-level virtual machine (LLVM) compiler, on which we are now working on one of the optimization passes (Divergence Analysis). LLVM is a compiler system that provides high-level information to compiler transformations at compile-time, link-time, run-time, and idle time between runs, allowing for transparent, lifetime program analysis and transformation for arbitrary programs. LLVM defines a simple, language-independent type system that exposes the primitives commonly used to implement high-level language features; an instruction for typed address arithmetic; and a simple mechanism that can be used to uniformly and efficiently implement the exception handling features of high-level languages (and setjmp/longjmp in C). The LLVM compiler framework and code representation work together to provide a set of key features for practical, long- term program analysis and transformation. To our knowledge, no other compilation method offers all of these features. We describe the design of the LLVM representation and compiler framework, and evaluate the design in three ways: (a) the size and effectiveness of the representation, including the type information it provides; (b) compiler performance for several intraprocedural problems; and (c) illustrative examples of the benefits LLVM provides for several challenging compiler problems.

3. ”Branches-on-superword-condition-code” (BOSCC) is introduced, which is closely related to divergence analysis. This method reintroduces control flow into vectorized code in order to take advantage of instances where the predicates for particular parts of the code are totally true or false at run- time. While this avoids superfluous code execution, it has some overhead for the dynamic test and does not eliminate the problem of increased register pressure because the code must still care for cases where the predicate isn’t totally true or false.

4. A program transformation technique i.e., Slicing was proposed. Slicing allows the user to concentrate on the sections of a program that are relevant to the task at hand. Compiler optimizations, debugging, model checking, and protocol understanding are all examples of when program slicing has been used. The underlying Control Flow Graph must have a single end node for slicing to work. This constraint makes it difficult to handle control structures with several exit points.

5. A single piece of code run in parallel on distinct components of an array of data does not indicate that each instance of that code performs the exact same instructions. The programmer or compiler must substitute control flow with data flow in order to employ SIMD instructions in the context of diverging control flow. The proposed approach provides efficient blending code for architectures without hardware support for predicted execution and can deal with irreducible control-flow without duplication. It is based on control to data flow conversion and can deal with irreducible control-flow without duplication.

6. SIMD utility units on GPUs are increasingly being employed in standard purpose applications for high perfor- mance and accelerated power savings. However, the impacts of SIMD flow control fragmentation can result in a decrease in the efficiency of GPGPU programs, which are considered independent applications. Improving SIMD efficiency has the potential to assist a wide range of similar data applications by providing considerable performance and power gains. The subject of SIMD fragmentation has recently gotten increasing attention, and different micro-architectural techniques to solve various elements of the problem have been presented. How- ever, because these procedures are frequently sophisticated, they are unlikely to be implemented in practise. We propose a two-dimensional configuration for GPGPU architecture in this research, which leverages basic compression cycle techniques in the instruction stream where specific closed line groups are present. Basic cycling pressure (BCC) and swizzled pressure (SCC) are the terms used to describe these settings. Additional requirements for implementing this option in the context of GPGPU-trained architecture are given in this work. Their analysis of loads of different SIMD functions from OpenCL (GPGPU) and OpenGL applications (images) shows that BCC and SCC reduced execution cycles in different systems by approximately 42% (20% on average). In a subset of various operating loads, performance time is reduced by 7% in today’s GPUs or by 18% in future GPUs with better memory providing better performance. Their work makes a significant contribu- tion by simplifying micro-architecture in order to promote diversity while giving a variety of benefits to complicated systems.

7. The phi representation mentioned in the third condition of the motivation section is a result of the Static Single Assignment instruction representation format (SSA). Modern moderators, including as LLVM and GCC, use Static Single Assignment (SSA) intermediate representation (IR) to simplify and enable more complex setups, as described in this work. However, claiming the correctness of SSA-based development is difficult because the properties of SSA are dependent on the full work flow graph. This study addresses this issue by presenting proven techniques for demonstrating the consis- tency and efficiency of SSA-based systems. In Coq’s proof-of- concept, we’re employing this

method to extract mechanical evidence of a few ”small” alterations that serve as building blocks for the formation of massive SSAs. We officially verify the mem2reg LLVM variation in Vellvm, Coq-based formal semantics for LLVM IR, to show how this approach might be used. The resulting launches produce code and performance that are comparable to untested LLVM usage.

8. The Single Instruction Multiple Data (SIMD) signature paradigm has resurfaced as a result of the growing interest in image processing units. App developers now have more integration options thanks to SIMD technology, but self-organization remains a difficulty. Developers must cope with memory fragmentation and control flow in particular. These occurrences are caused by a phenomenon known as data fragmentation, which occurs when two processing elements (PEs) detect the same variable word with different values. Breakdown analysis, a statistical analysis that detects data fragmentation, is discussed in this article. This research, which is presently being invested in a high-quality compiler, is beneficial in a number of ways: it enhances the translation of SIMD code into non-SIMD CPUs, it assists developers in improving their SIMD applications, and it drives SIMD ap- plication automation. They demonstrate this last argument by presenting the concept of the division register’s disintegration. This spiller leverages data from their analytics to restructure or distribute data among PEs. They tested it on a set of 395 CUDA kernels from well-known benchmarks as confirmation of its effectiveness. The divergence-aware spiller generates GPU code 26.21% faster than the code-generated code used in the basic moderator.

9. A static analysis was presented that identified specific limitations on variable values in distinct concurrent instances. This Vectorization Analysis classifies memory access patterns and determines which values are the same for all parallel instances, which values require sequential execution, which programme points are always executed by all instances, which loops may have instances exiting in different iterations or over different exits, and which loops may have instances exiting in different iterations or over different exits. To detect reductions, existing vectorizing compilers rely on a set of hard- coded patterns. These loop vectorization implementations are intended for loops where the reduction operations are critical to the loop’s overall performance. Finally, WFV was utilised in a compiler to construct a loop vectorization transformation that allows users to designate which optimization stages should be conducted on specific code areas.

10. Multi-partitioning is a crucial step in mapping and integrating compatible data applications on computer networks with both CPUs and GPUs. A number of automatic cate- gorization approaches, both vertical and dynamic, have been proposed for this purpose in recent years. The current study looks at regulatory unpredictability and how it affects the quality of those system variables. They recognise the number of variations in the system as a critical performance factor and train Machine Learning (ML)-based separators to calculate the OpenCL load split of several platforms with one CPU and one GPU statistically. When compared to previous ML- based classification algorithms for the same data load, their performance reports improved classification results in terms of time performance.

11. DARM is a framework for analysing and transforming compilers that can combine divergent control-flow topologies with comparable instruction sequences. DARM has been found to mitigate the performance deterioration caused by control-flow divergence. Divergence-Aware-Region-Melder (DARM) is a control-flow melding implementation that uses a hierar- chical sequence alignment technique to identify advantageous melding opportunities in divergent if-then-else portions of the control flow and then melds these regions to decrease control- flow divergence. According to the requirements, the method takes an SPMD function F and iterates over all basic blocks in F to see if the basic block is an entry to a meldable divergent region (R). Then, inside R, use Simplify to convert all sub- regions to simple regions.

V. CONCLUSION

The focus of this study was to carry out a systematic analy- sis of current research work on the LLVM Divergence analysis. The current Divergence Analysis pass returns the analysis of IR instruction set in linear time. However, any change in the IR (by any optimization pass) nullifies the results of the previous analysis pass. Hence the entire computation needs to be redone no matter the extent of changes in the IR. Hence a new Divergence analysis pass which can smartly determine the changes and only recomputes the required information will result in noticeable performance enhancement. This divergent pass depends on SyncDependnece Analysis to determine the join blocks for a divergent terminator block. The current SDA algorithm is static (need to be recomputed for any change to IR) and runs in time complexity O(E + V), where E is the set of edges in the control flow graph and V depicts the set of vertices in the CFG.

REFERENCES

[1]F. DAREMA, D.A. GEORGE, V.A. NORTON and G.F. PFISTER: A single-program-multiple-data computational model for EPEX/FORTRAN, Computer Sciences Department, IBM T.L Watson Research Centre, Yorktown Heights, NY 10598, U.S.A. https://doi.org/10.1016/0167-8191(88)90094-4 - 1988

[2] LLVM: A Compilation Framework for Lifelong Program Analysis Transformation. Chris Lattner Vikram Adve(2004)

[3] Shin, J.: Introducing Control Flow into Vectorized Code. In: PACT. pp. 280–291. IEEE Computer Society (2007)

[4] Torben Amtoft. 2008. Slicing for modern program structures: a theory for eliminating irrelevant loops.

[5] Ralf Karrenberg, Sebastian Hack: Whole-function vectorization, 2011, IEEE https://doi.org/10.1109/CGO.2011.5764682

[6] Karrenberg R., Hack S. (2012) Improving Performance of OpenCL on CPUs. In: O'Boyle M. (eds) Compiler Construction. CC 2012. Lecture Notes in Computer Science, vol 7210. Springer, Berlin, Heidelberg.

[7] Aniruddha S. Vaidya, Anahita Shayesteh, Dong Hyuk Woo, Roy Sa- haroy, Mani Azimi (2013). SIMD divergence optimization through intra- warp compaction.

[8] Formal verification of SSA-based optimizations for LLVM Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, Steve Zdancewic(2013)

[9] Divergence analysis. Diogo Sampaio,Rafael Martins de Souza,Caroline Collange,Fernando Magno Quintão Pereira(2014)

[10] Ralf Karrenberg. 2015. Automatic SIMD Vectorization of SSA-based Control Flow Graphs. Springer.

[11] Divergence Aware Automated Partitioning of OpenCL Workloads. Anir- ban Ghose, Soumyajit Dey, Pabitra Mitra, Mainak Chaudhuri(2016)

[12] Charitha Saumya, Kirshanthan Sundararajah, and Milind Kulkarni. DARM: Control-Flow Melding for SIMT Thread Divergence Reduction.