Ranking Functions in a Program for Hardware Acceleration

Ali H.A. Abyaneh
a.abyaneh@uwaterloo.ca
University of Waterloo
Waterloo, Ontario

Abstract

FPGAs allow us to accelerate some applications with much more energy efficiency compared to GPUs while providing the same or even more performance improvement. However, deciding which part of the program should be accelerated using FPGAs, especially in large programs, has not been appropriately answered yet. This report motivates the problem, provides an under-development framework to automatically rank functions based on their suitability for hardware acceleration, and explore future development opportunities. The LLVM based tool extract several features of each function in C code. Then, a ranker algorithm, which is motivated by PageRank, ranks the blocks based on those features. The ranker algorithm is a graph processing algorithm, and function calls in the code map to directed edges between the callee and caller. The ranker traverses the graph until it converges to an almost fixed value for each function. Extracted features are loop information, dependency structure of the code, and function call network. Necessary future steps are learning the weight of each feature, and disambiguating pointer ranges to calculate data transfer sizes.

CCS Concepts: • **Computer systems**;

Keywords: FPGA, LLVM, acceleration,

ACM Reference Format:
Ali H.A. Abyaneh. 2020. Ranking Functions in a Program for Hardware Acceleration. In *Proceedings of Waterloo, Ontario (CS644 Winter 2020).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The growing and considerable power consumption of state-of-the-art datacenters on the one hand and the performance requirement of cloud services, on the other hand, has heightened the need for acceleration in clouds. As a result, deploying hardware accelerators such as FPGAs has gained increasing attention in both academia and industry in the past few years. Nowadays, many cloud providers such as Microsoft and Amazon offer FPGA accelerators to their costumers [2, 11]. Microsoft uses FPGA to accelerate its search engine[9]. Compared to GPUs, FPGAs offer much less power consumption, while providing high performance. Also, unlike ASICs, FPGAs are reconfigurable, which makes their use financially more reasonable.

All the merits notwithstanding, deploying FPGAs as a reconfigurable fabric is still challenging in that the design cycle requires vast knowledge, and it is time-consuming. However, the emergence of High-Level Synthesis tools that translate higher-level languages like C to HDL mitigate this challenge. Although HLS tools provide a framework to translate codes into HDL, there is still a question which is needed to be answered. Given that our resources are limited, which part of the program should be accelerated to gain more performance? Note that, previous studies [3] has been shown that sometimes hardware accelerating can result in huge performance degradation. This problem is much more severe when it comes to accelerating interactive tasks. Note that interactive tasks, in which tail latency is the main performance measure, form a significant portion of datacenters workload. Examples of such tasks are search engines, speech recognition, memory caching systems, Apache Spark application, etc. The problem becomes more interesting if we limit the resources. Given that a node has limited area on some FPGAs, which part of the program should be accelerated to gain higher performance?

Furthermore, detecting subject-to-hardware acceleration regions of the program for programmers who are not the developer of the very program is challenging. This challenge becomes more significant when acceleration resources are limited, and programmers need to find the best permutation of code blocks to accelerate leading to the highest performance achievement. This procedure is not trivial for several reasons, namely, different usage frequency of blocks, required data transfer and available bandwidth, computation intensity of blocks, and parallelism level inside each block. To the best of our knowledge, current studies accelerate the whole application. Also, there is no framework to mitigate

CS644 Winter 2020, University of Waterloo

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the procedure of finding subject-to-acceleration blocks of code.

In this project, we leverage LLVM to analyze the code and extract several features. Then, using a PageRank-like algorithm, we rank the functions. Using LLVM, we disambiguate pointers, calculate the dependencies in the control flow graph of the program, and score each function based on its dependency structure. Also, we find parallelizable loops in functions, and include that in our calculation. The usage frequency of each function is another important factor. If a function is rarely used, by accelerating that function, the FPGA will be underutilized. Moreover, we instrument the code and calculate function calls and loop iterations dynamically. Also, we study how to analyze the range of pointers, but unfortunately, the implementation is not complete yet. Finding the range of arrays is in our interest because the required transferred data between the host and the kernel directly affects the performance. This factor is more critical when it comes to accelerating latency critical tasks.

The following sections are organized as follows. The tool structure is briefly described. We explain the features we are extracting. Then, we summarize some previous studies which are dealing with some of our challenges. Additionally, the ranker algorithm is explained. Moreover, we describe a road-map for the future of this research.

2 Tool structure

The general structure of the tool is described in Fig. 1. Using clang, we create LLVM intermediate representation–IR. The developed passes require some pre-analysis, namely, Alias Analysis, Scalar Evolution[1], and Loop Information analysis[2]. Afterward, we detect parallelizable loops, 4. Also, we score functions based on their data dependency structure, 5. Moreover, we instrument the code to calculate number of function calls, and loops' iterations[1].

3 Dependence Analysis

In the first place, we study a representation of program control and data flow called Program Dependence Graph, PDG, because PDG has been the foundation of most of the compiler techniques for dependence analysis for almost last four decades. And no need to mention that dependence analysis is the foundation of many optimization techniques, among which parallelism, pointer disambiguation, and array range analysis are of our interest.

Using LLVM, first we disambiguate ambiguous pointers. Then, we perform dependence analysis inside each function. The dependence analysis detects flow dependency, input dependency, output dependency, anti-dependency, and true dependency. Note that, LLVM dependency analysis is conservative. Consequently, the analysis result may overlook many features. This can be improved in the future.

3.1 Program Dependence Graph

Program Dependence Graph is a graph representation of a program that clarifies both data and control dependency. Using PDG, the compiler can perform many optimizations like vectorization, uniformly for both control and data dependency. Ferrante et al. in [5] introduce an incremental flow algorithm to simplify incremental analysis and transformations using PDG. **–add this later!**

4 Parallelism Detection

For any kind of acceleration, detecting parallelism in sequential programs is of great importance, and this job is no exception. Therefore, in the first place, we focus on state-of-the-art techniques to detect a parallel loop. In fact, substantial research has been devoted to automatic parallelization, resulting in compilers able to detect parallel loops to a great extent. Note that, for the purpose of our research, not only parallel loops are important, but also programs with small-scale dependency are essential.

Unlike CPU and GPGPU, in FPGA, exploiting loop level parallelism is more straightforward in that FPGAs can implement any custom logic. Therefore, we can easily use different loop parallelism methodologies in FPGA, namely, DOALL, Distributed, and DOACROSS parallelism. However, at the moment, for ranking, we just detect DOALL loops. Also, since we still do not have the dynamic range analysis described in the section 8, and LLVM analysis are conservative, we instrument the code to obtain information about loops header. For every loop, we create a data structure which keeps the information about the nesting level, size of the loop body, computation score of loop body, and loop header information if applicable. Note that, if there is a dependency in an inner loop, outers can still be parallelized. Another factor which is needed to be learned in the future is the exact score of loop based on these values. For now, we use a simple function to calculate loops score based on these numbers.

5 Score Dependency

Not only are loops subject to FPGA acceleration, but also blocks of code with small-scale data dependency are suitable candidates for hardware acceleration. Consider a block of code with a huge if-else statement. Obviously, there is a control dependency in this block, and in worst case scenario, it takes several clock for the CPU to calcualte the result– which affects the tail latency. However, in FPGA, it can be done in only one clock. In contrast, for data dependency, the best thing that can be done is pipelining to utilize the FPGA. Note that accelerating such a block with huge data dependency can lead to huge performance degradation.

[1] This pass understands loop expressions, and scalar which evolves by loop iteration.
[2] LoopInfo identifies natural loops.

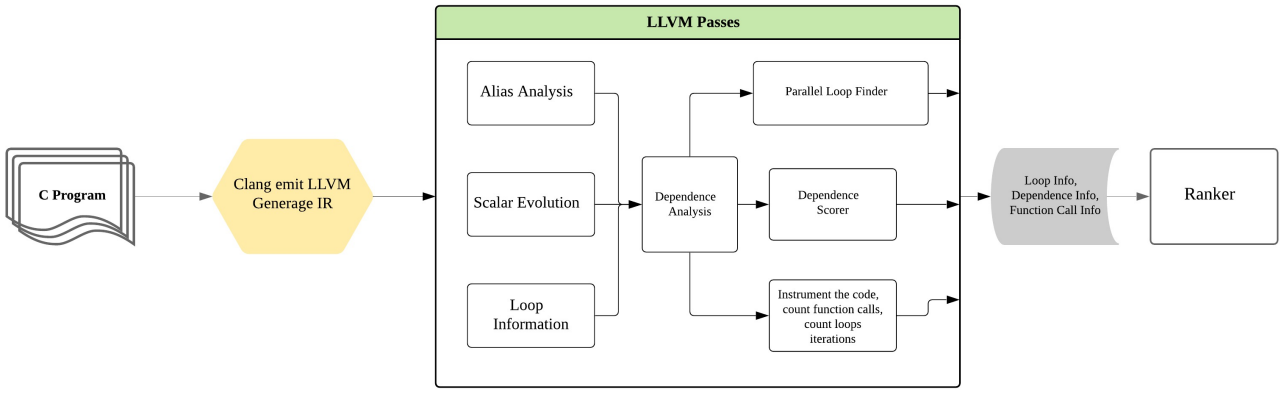

Figure 1. Tool General Structure - Dynamic Array Range Analysis should be added in the future.

In this project, we calculate the relative number of data dependency for each block. Then, we score blocks relatively based on that number.

6 Computation Score

In addition to loops, blocks of code which are computation intensive are suitable candidate for hardware acceleration. For example, floating point division can take up to 60 cycle in CPU, while it can be done in FPGA in one cycle. Therefore, we calculate how much computation each block has, and extract that information as a feature. This number, in addition to the dependency score of functions can help us to build a more accurate model for ranking.

7 Related Works

7.1 DawnCC, Automatic Annotation for Data Parallelism and Offloading

Motivated by the difficulty of code annotating for programmers, G Mendoca et al., in [7], provide a framework for automatic annotation of sequential programs to directive-based programming models. The significant challenges in doing so are ambiguous pointers for two main reasons. First, aliasing pointers restrict the ability of automated parallelism. Second, data transfer primitives are limited by those pointers, the bounds of which are not explicit.

The main contributions of this paper are three folded. In the first place, DawnCC disambiguate pointers, and performs the symbolic bound analysis to mitigate data transfers. In the second place, DawnCC maps information gained in IR to source code using the scope tree [3]. Third, by using the program dependence graph, DawnCC avoids redundant data transfers between host and kernel and vice versa. The DawnCC pipeline is as follows, detecting doall loops, then injecting data transfer primitives. When in fact, doall loops only form a small portion of loop level parallelism. Put it another way, being doall is too strong a condition for a loop. DOACROSS parallelism, pipeline parallelism, distributed parallelism, to name but a few, are other kinds of common parallelizable loops in programs. Moreover, LLVM static analysis for memory dependency detection is innately conservative. Therefore, any analysis done using that framework results in a risk-averse tool that may underestimate quite a few parallelization opportunities in the program.

All of the analysis is static and can be divided into two categories, that is, dependence analysis, and symbolic range analysis. Dependence analysis is the first step in the static analysis phase of DawnCC. Dependence analysis is done using the Program Dependence Graph [5]. Symbolic range analysis is a technique that calculates a lower and upper bound for every integer variable in a program [8]. A symbol is a variable the value of which cannot be determined as a function of other variables. To find these bounds for every variable, a series of range inference rules are defined for every arithmetic operation and control flow construct in LLVM IR. The main goal of such rules is tracking the changes in the range of variables in the program flow. The outcome of symbolic range analysis is a map from integer variables to a tuple (lower bound, upper bound). Although implementing the Symbolic Range Analysis described in [7] is trivial, it is not suitable for our project, the main reasons for which are provided at the section 8. Another question which is not answered in DawnCC is the convergence of the symbolic range analysis algorithm. Instead, they use a widening operator to prevent divergence in loops.

Pointer Aliasing—the situation in which a memory location can be accessed using different symbols is another challenging problem in languages like C and C++, where opaque pointers can point to any location in the program memory space. Although DawnCC handles aliasing using the information gained in symbolic range analysis, static

[3]Scope Tree is a data structure, each node of which keeps a map from a scoping statement to its valid scope range, that is, line number in the actual code.

techniques are conservative. As a result, quite a few pointers are confused in compile-time, whereas, in reality, they are guarded from one another. On top of everything else, the extent to which pointers address spaces are confused is enormously determined by the programmer. Consequently, the tool may not behave as appropriate as expected in some cases.

The last obstacle addressed here is mapping the analysis result from LLVM IR back to C code, which is done using the scope tree.

8 Array Range Analysis

Precise array range analysis plays an important role in our research in that it is needed to determine the required amount of data transfer between blocks of code. Therefore, imprecise methods cannot satisfy our requirements. Furthermore, even state-of-the-art compile-time approaches in range analysis are conservative, which can miss many acceleration opportunities. Above all, compile-time range analysis methods cannot determine the size of dynamically allocated arrays. Unfortunately, now, the array range analysis pass is not implemented thoroughly, and it requires much more work in the future. Note that, current trends in FPGA technology for cloud services are making data transfers between Host CPU and FPGA much faster. Currently, FPGA vendors offer FPGA with High Bandwidth Memory, HBM, which provide around 500 GB bandwidth, which is about 25 times faster than DDR4 memory. Also, new developments like CCIX–Cache Coherent Interconnect for Accelerators– mitigate the data transfer overhead. As a result, FPGA are becoming popular for in-memory processing in databases[4].

However, we study one Pointer range analysis techniques and summarize it in the following section. Also, we provide notes on how we plan to extend that approach in our dynamic analysis[4]. Note that even with instrumentation, determining the size of memory for function parameters is not trivial since pointers can point to any region of memory, arrays can be accessed using pointers, and this is true not only inside the function but also at the call location.

8.1 Array Range Analysis*[10]

8.1.1 Motivations and Problem Statement. Yong and Horwitz in [10] compute a Pointer-Range Analysis algorithm to estimate memory access bounds of each pointer safely. The goal of such an analysis is determining the range of array indices. This information can be used in many aspects, some of which are determining possible out of bound access to memory, dead code elimination, and aliasing analysis. Furthermore, determining the range of pointer dereferences can be used to determine the size of data transfer when it comes to accelerating functions. The importance of such an analysis notwithstanding, C-like languages impose new challenges for range analysis. Arrays in C can be accessed using pointers. Also, pointers of one type can access to other types using typecasting. Moreover, deciding whether a pointer is pointing to an array or not is involved. To address the second problem, Yong and Horwitz describe pointer-range using a data structure containing three components, namely, target location, target type, and offset range. From now on, by range data, we mean the mentioned structure. Furthermore, to simplify the analysis, to deal with numerical values, Integer Interval Domains are used. With all this taken into account, authors in [10] suggest demonstrating every variable, including arrays, as pointers. Hence, accessing variables can be done using dereference.

8.1.2 Implementation. Pointer-Range Analysis is applied to the Control Flow Graph of the program. For each location of a pointer, x, a range representation is kept to keep track of x range data. Note that, for a range of data to be safe, it must be a superset of the actual range.

As mentioned before, to understand the object to which a pointer points—target—for each pointer, we keep a target type, target location, and target offset separately. The example shown in Table 1 is self-explanatory.

Code	pointer info

int a[10], b[8];
int* p;
 	
if(condition)
  p = &a[6];
else
  p = &b[4]
 	
*p = 0
	
	p → <a: int [10], [6,6]>
	
	p → <a: int [8], [3,3]>
	
	p → <unknown: int [8], [3,6]>

Table 1. range info example

8.1.3 Pointer Arithmetic. Handling pointer arithmetic is another vital aspect of range analysis for C like languages. Pointer arithmetic can be divided into two categories, that is, well-type and mismatched typed arithmetic. Well-typed pointer arithmetic is a type of operation in which types of arguments of the arithmetic expression are their expected types. Note that arguments type do not necessarily need to be identical in C pointer arithmetic operations. Note that, for well-typed pointer arithmetic operations, we can simply update the offset range in the range data structure. Accordingly, the type is what is expected from the very operation. To address the problem associated with mismatched-type arithmetic, we can either tag the pointer as confused or weaken the safety condition. The safety condition enforces us to validate the types. In other words, a range is safer than another range if they have the same type, and the offset

[4]This is a part that I really appreciate if I receive some feedback on it.

range of the second one is the subset of the first one multiply by the ratio of their target object type size. Removing the same type of condition, authors in [10] alleviate the problem. All in all, using the mentioned data structure for range, and considering the rules for pointer arithmetic, we can keep track of the range of pointers.

8.2 CGCM

Also, there are other studies that face the very similar problem from other aspects, one of which is CGCM[6], which stands for CPU-GPU Communication Manager. Authors in [6] suggest a framework to manage communication between CPU and GPU automatically. The main contribution of CGCM is a set of static analysis and runtime libraries that manage communication between CPU and GPU without relying on compile-time analysis. The main motivation behind dynamically managing communication is similar to our motivation, which opts us to analyze array's range dynamically; that is, in C-like languages, any argument in a function can be casted to a pointer, and pointers can point to any arbitrary object. Note that when facing such a problem, the compile-time analysis does not yield any useful information.

CGCM runtime library derives the size and shape of structures in runtime. To do so, instead of just looking at arrays, the runtime library keeps track of allocation units. An allocation unit is a block of memory that is allocated as a single memory unit. For example, malloc allocates an allocation unit. CGCM runtime library stores the base and size of every allocation unit in a tree data structure. This tree is indexed by the base addresses of allocation units, and keys are the size of allocation units. To keep track of the size of allocation units in pointers, CGCM instruments every memory allocation instruction, namely, malloc, calloc, realloc, and free. Every time one of these instructions get called, the allocation unit tree will be updated. The drawback is that the paper does not specifies how pointer arithmetic is handled in run time.

8.3 To Do

To do an array range analysis dynamically, we suggest a hybrid of the mentioned papers. The idea of using a balanced map in [6] is interesting for us. Furthermore, looking at allocation units rather than memory units can facilitate our job. Note that, by doing so, we will not need to keep track of the type of data at run time. However, we still need rules in [10] to handle pointer arithmetic. Note that by dynamically analyzing the code, unlike [10], we can manage non-constant malloc as well.

9 Ranker Algorithm

Ranker collect data from LLVM passes. For now, the collected data includes loop details, code data dependency information, function call patterns, number of calls, and number of loops iterations. To rank the code, we create a graph, each node of which corresponds to a function in the program. For every call inside a function, we create an edge between the very function and the callee. In fact, if we decide to accelerate a function, we have to accelerate it as a whole. Note that if we want to implement an inner function in the host CPU, it is highly likely that we degrade the performance. Hence, a function score is the summation of its own score and the score of the functions it calls. Therefore, the ranker algorithm must iteratively traverse the graph and update every function score. Note that in our implementation, parent functions do not necessarily obtain higher score compared to the functions they call; because the dependence structure of the parent can decrease its score. To ensure the convergence of the ranking algorithm, we define a dynamic learning rate for updating functions score at each iteration.

As mentioned before, we score the compute-intensity of functions and loops bodies. For now, for computational operations like add and multiply, we assume a constant score. The scoring is based on the fact that, for example, multiplication in CPU takes about 6 cycle, whereas it can be done in FPGA in one cycle. Moreover, existence of each data dependency in the code delays the operation and decreases the maximum clock in FPGAs. Thus, for data dependencies, we assign a negative score. Furthermore, each loop block score is the number of iterations it get to run multiply by its computation score. For functions, the score is the summation of computation score and dependency score.

10 Future Works

There are quite a few things to be done in the future. First, we need to instrument the code to obtain the range of arrays. Also, to calculate the score, at the moment, we have intuitively assigned weights to data gathered via LLVM. However, we propose to learn those weights. Note that, to do so, we have to create a relatively large data set for codes mapped to FPGA, and measure the speed-up for each function. By doing so, we can create an analytical model for the scorer. Moreover, information gathered in LLVM passes can be expanded. Take, for instance, different kind of dependencies may have different weights in the final result. Also, as mentioned before, other loop level parallelism can be exploited.

11 Conclusion

We explained the problem of ranking functions based on the possible performance gain achieved by accelerating them. We used LLVM to extract features of codes which can help us to build a model for such a ranking. The features are data dependency structure, computation score, parallel loops, and their computation score, number of each loop iterations, and usage of each function. Moreover, we analyzed some research that faced similar problems. The ranker algorithm is a graph processing algorithm that traverses the graph and updates the score of each function until all scores converge. To the

best of our knowledge, this is the first attempt in ranking functions for acceleration in a program. Our hope is that this project results in a research.

References

[1] [n.d.]. LLVM Tutor. https://github.com/banach-space/llvm-tutor.

[2] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*. 1–10.

[3] Yu-Ting Chen, Jason Cong, Zhenman Fang, Jie Lei, and Peng Wei. 2016. When Spark Meets FPGAs: A Case Study for Next-Generation {DNA} Sequencing Acceleration. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*.

[4] Jian Fang, Yvo TB Mulder, Jan Hidders, Jinho Lee, and H Peter Hofstee. 2020. In-memory database acceleration on FPGAs: a survey. *The VLDB Journal* 29, 1 (2020), 33–59.

[5] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[6] Thomas B Jablin, Prakash Prabhu, James A Jablin, Nick P Johnson, Stephen R Beard, and David I August. 2011. Automatic CPU-GPU communication management and optimization. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 142–151.

[7] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. 2017. DawnCC: automatic annotation for data parallelism and offloading. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.

[8] Vitor Paisante, Maroua Maalej, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. 2016. Symbolic range analysis of pointers. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 171–181.

[9] Jing Yan, Zhan-Xiang Zhao, Ning-Yi Xu, Xi Jin, Lin-Tao Zhang, and Feng-Hsiung Hsu. 2012. Efficient query processing for web search engine with FPGAs. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 97–100.

[10] Suan Hsi Yong and Susan Horwitz. 2004. Pointer-range analysis. In *International Static Analysis Symposium*. Springer, 133–148.

[11] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. 2017. The Feniks FPGA operating system for cloud computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*. 1–7.