Abstract

Acknowledgements

Biographical Sketch

Table of Contents

Related Works

* Diospyros

Current draft borrows formatting from Samwise Parkinson’s MS Thesis.

Change citations to ACM style citations

Introduction

This thesis,,,,

Explain what thesis does

Explain why problem is important

Explain solution

Add some motivation with LLVM and why it is useful

Chapter 1

Introduction

Problem and Motivation

Developments in computer architecture have led to domain specific architectures that deliver performance and energy benefits, such as digital signal processors (DSP). DSPs often have many functional units to compute instructions in parallel. One feature of DSPs is the ability to compute instructions of the same type at the same time, for example, four additions at the same time. This ability, provided by vector instructions, allows programmers to write more performance efficient code, because there is more computation in any one clock cycle.

General purpose programming languages, like the C programming language, can be compiled to these domain specific architectures, such as those with vector instructions, but often, the full performance is not enabled. The compiled code may fail to detect parallelism, or fail to generate profitable parallel assembly instructions. There remains a gap between the performance the compiler generated code has, versus the optimal code for the architecture.

Bridging this performance gap is a central idea in compilers research. Compilers seek to translate code from high level to low level languages, with capability for optimizing the code and using features of the target architecture. To fully utilize the features of the target architecture, today’s compilers rewrite code aggressively. Rather than running a fixed sequence of passes, compilers now use an array of methods to generate better code ranging from reducing a problem to a problem solvable optimally by a boolean-satisfiability or integer linear programming solvers, to applying a rewrite engine, or even using machine learning to guide the search for more performant programs.

All these methods find better programs by searching a larger space of possible programs, compared to the single program that is found by applying a linear sequence of compiler optimization passes. As a result, today’s research compilers can find code that is an order of magnitude faster compared to traditional compilers like Clang [??]. However, a major tradeoff is that compilation slows down, because the compilation process contains a search of many candidate programs.

One area where a performance gap exists in compilation is with linear algebra kernels, small functions that map array inputs to array outputs. Often, there is implicit parallelism in these kernels. An auto-vectorizing compiler will try to find similar operations that are repeated, such as multiple additions, and turn these instructions into a single vector addition instruction. Finding opportunities to use vector instructions is difficult, either because the parallelism is difficult for the compiler to detect, or because the compiler fails to search a large enough space of equivalent programs to vectorize the program.

Previous work has identified several approaches to harness parallelism and generate vector programs for kernel code. Loop vectorization identifies loops without dependencies, and transforms several iterations of the loop into a single iteration using vector instructions. Superword level parallelism (SLP) vectorization is an orthogonal approach to loop vectorization. SLP vectorization takes advantage of repeated instructions of the same type, such as multiple additions, which can be joined together into a vector addition instruction. SLP vectorization joins instructions together based on a greedy heuristic, and thus sometimes does not find vectorization opportunities.

Diospyros presents a novel method of auto-vectorization by reducing the problem to one of equality saturation [VanHattum…]. In equality saturation, rewrite rules are repeatedly applied to a program to transform it. Equality saturation uses a data structure called an E-graph to store all possible orders of rewrites of the original program. By storing all possible orders of rewrites, equality saturation avoids the pitfalls of ordinary compilation, in which only one sequence of rewrites are applied to a program, resulting in a suboptimal result.

Diospyros has the following compilation process. First, Diospyros will discover a specification of what is calculated in each output address of the kernel, by applying symbolic evaluation from Rosette. Next, Diospyros will apply custom rewrite rules, saturating the E-graph with many equivalent programs, some of which use vector instructions. After applying custom rewrite rules, Diospyros then extracts an optimal program by greedily choosing a program with from E-graph with the lowest cost, using a custom cost model. Finally, Diospyros will lower the chosen program to C++ code using compiler intrinsic instructions for a specific digital signal processor’s architecture.

Diospyros is currently implemented in a domain specific language, and only targets a specific Tensilica digital signal processor. My goal for this research is to expand Diospyros to apply to a wider set of kernel programs, by allowing for inputting in C language programs. In addition, this thesis will discuss using the ideas of vectorization on LLVM intermediate representative, rather than from a high-level language or a domain specific language. Finally, this thesis will investigate whether it is possible to vectorize a program using Diospyros for a non-Digital signal processor. This thesis will try to vectorize for an architecture that does not have cost-effective gather and scatter operations and will instead try to avoid these instruction, which are central to Diospyros.

This thesis will cover the following areas: first, I will discuss background ideas relating to auto-vectorization, equality saturation, and the Diospyros approach to vectorization. Next, I will share the implementation of the auto-vectorization Finally, I will discuss results of my implementation of the auto-vectorizer on benchmarks, and conclude.

Core Ideas

My Contributions

Thesis layout

Chapter 2

Background

2.1 Auto-vectorization

Auto-vectorization is the process of automatically converting sequential code into parallel code by using vector instructions to do multiple identical operations at the same time. Unlike a scalar instruction, such an addition operation, a vector addition operation can do multiple addition operations at the same time. Using vector instructions is desirable because vector instructions can do a factor more work in the same amount of time as a single scalar instruction. In addition, vector instructions can use less energy compared to similar parallel instructions [citation needed].

Manually transforming code to use vector instructions is tedious and time-consuming. Programmers using vector instructions are often forced to spend time to change their code into a form that can use vector instructions. Making code reach a performance goal using vector instructions can take many iterations of programming using many man-hours of work. When code is finally transformed from a scalar version to a vectorized version, the code often lacks the original high-level nature, and instead is filled with implementation details, such as unrolling or calls to compiler vector intrinsic functions. The actual meaning from the code is obscured behind optimization-specific code. Finally, code that is manually vectorized is not portable. Manually vectorized code targets a single architecture, and retargeting another architecture requires more experimentation and engineering to make sure that the same level of performance is still achieved.

As a result, compiler auto-vectorization is crucial, because the compiler will automatically discover vectorized forms of a sequential program the programmer gives, without the need for programmer intervention. In some scenarios, compiler vectorized code can explore more alternatives that the programmer might not consider, and generate assembly code that performs at least as well as hand-optimized code using vector instructions. Moreover, auto-vectorized code does not require the programmer to change the original scalar algorithm, and therefore preserves the original high-level nature of the code, that is uncluttered by optimization specific code. A final advantage for auto-vectorized code is that a compiler backend can use target vectorized code to many architectures, allowing for portability of the code.

There are two common types of auto-vectorization. The first, loop vectorization, takes advantage of the fact that across iteration of the loop, instructions at the same location in the loop are identical, and can sometimes be vectorized. One challenge to loop vectorization is when there exist inter-iteration dependencies between statements that are to be vectorized, which stop vectorization because vectorization would break the dependencies. Worse, compilers sometimes cannot prove there are no inter-iteration dependencies even when the dependencies do not exist, and therefore, the loop cannot be vectorized.

A different form of auto-vectorization is super-word-level parallelism vectorization otherwise called SLP vectorization [Larsen and Amarasingh, 2001]. SLP vectorization takes advantage of natural repetition of operations that exist in code, outside of loops. To create natural repetition, SLP begins by unrolling loops in the program. SLP groups these repeated sequences of instructions together and tries to vectorize them greedily. First, groups of aligned and consecutive loads and stores are created. Then, following use-def and def-use chains, more instructions are vectorized, until an operand mismatch occurs, in which vectorization stops. Then, the benefit of the vectorization is measured using a cost model, and if the vectorization is deemed beneficial, vector code is emitted.

Due to the greedy vectorization strategy used in SLP vectorization, sometimes, profitable vector code is not generated, despite the fact that the program might contain code that could be beneficially vectorized. This challenge has been addressed in several ways: some research has extended SLP to vectorize instructions that failed by taking advantage of commutative instructions or other properties of instructions in the vectorization tree that allow for rearrangement to continue vectorization [Porpordas..]. Another approach is to try other greedy heuristics for vectorization. For example, there is research that hierarchically considers short chains of programs to vectorize, selects the better chains, then prunes the better chains [..] Another greedy heuristic considered was to consider vectorization nodes with the most number of uses and defs [Shin?]. These greedy approaches generally perform better than SLP vectorization on standard benchmarks, but do not study the entire space of vectorization.

A different formulation of auto-vectorization is as a search problem. This form of auto-vectorization expands the ability to find profitable vector code within a larger space of programs considered, compared to greedy methods. One method of reduction was to integer linear programming problems, in which pairwise constraints between instructions are encoded and the ILP solver finds an optimal pairwise solution [Mendis & Amarasingh]. This approach finds the best pairwise solution in a tractable amount of time. A second type of reduction is to a synthesis problem, in which a specification is generated, and a synthesizer generates code to satisfy that specification. This vectorization approach was used in an early version of the Diospyros project [VanHattum, …]. Another approach considered is by reduction to equality saturation [VanHattum, …]. This approach is the central method used in this thesis.

2.2 Equality Saturation

Equality saturation is a compilation technique that represents the results of rewrite rules in all possible orders. This technique was used to find superoptimized programs in the Denali project [Joshi …], and was used as a general compilation technique by [Tate…] Furthermore, many compilers research projects have used equality saturation as a central compilation technique [Willsey, Chandakandi?, ….] The equality graph is the central data structure in equality saturation.

The equality graph, or E-Graph, is a data structure that compactly stores equivalences between terms. An E-graph is composed of equality classes, or E-classes, which are sets comprised of terms. Each term in an E-class is considered equivalent. A term in a E-class represents a function applied to arguments, which the E-class point to. A useful package for equality saturation is Egg, which is used in this thesis [Willsey..] Egg is a customizable equality graph rewriting package, in which users can act domain specific rules.

Equality saturation was used as a central technique for optimization in the Denali project. [Joshi…] The goal of Denali is to take an assembly program and find a shorter sequence of assembly instructions that can compute the same result. Prior approaches to find the shortest sequence of instructions to compute a result, termed Superoptimization by Alexa Massalin [Massalin…], generally used time-consuming enumeration and failed to scale to larger programs. To solve this problem, the designers of Denali applied to E-Graph to the domain of optimization. Programs are instantiated in the E-Graph, and rewrite rules are applied to the E-Graph until the E-Graph is saturated, representing all possible programs equivalent to the original program under the set of rewrite rules. Then, a maximum number of cycles to compute the program is selected. A boolean expression is computed based on the E-Graph and the selected maximum number of cycles. The Boolean expression is either satisfied or not, based on a query to a SAT solver. Finally, the number of cycles is either decreased, if the expression is satisfied, or increased if not, to find the lower number of cycles needed to compute the program. Termination occurs when a lower bound is found, and the assembly instructions corresponding to that lower bound are emitted.

2.3 Diospyros

Diospyros is a vectorizing compiler that uses equality saturation to find a vectorized program, building on work from Denali to the domain of auto-vectorization. The input program to Diospyros must be a kernel, that is, a function that maps array inputs to array outputs. Diospyros works by first extracting a specification of a given program. A specification is a mathematical formula that explains how a certain output address of the kernel program is computed. Diospyros then converts the specification to a program in the vector language, a minimal language that can then be optimized. The vector program is sent through the Egg rewriting engine, which applies hand-written rewrite rules to transform the program. Finally, the best program is chosen from the E-graph via a cost model, and from there, the program is lowered to the specified architecture, which is a Tensilica machine.

Diospyros has several advantages over other vectorization strategies. First, Diospyros considers a large space of transformations compared to greedy heuristics for vectorization. This means that more programs can be profitably vectorized, and even when a greedy heuristic will vectorize a program, Diospyros can vectorize the program more effectively, by keeping more values in vector registers. Secondly, Diospyros also simultaneously considers a smaller space of transformations compared to other techniques. For example, compared to reduction of vectorization to an ILP solver, which might consider all pairs of vectors, Diospyros only considers legal transformations of programs via rewrite rules. As a result, fewer programs are sampled, meaning vectorization is faster, yet the vectorized programs are still very efficient and nearly optimal.

2.4 LLVM

LLVM is a customizable middle-end infrastructure to develop compilers. LLVM contains an internal representation language (LLVM IR), which is like RISC-V assembly, and can represent C programs. The IR language contains standard assembly instructions, such as arithmetic instructions, boolean instructions, memory load and store instructions, as well jumps and branches. The IR also has vector instructions and vector intrinsic instructions. LLVM contains many utility functions to interact with the IR, such as instruction builders. LLVM also provides a multitude of optimization and analysis passes, which operate on the IR, either transforming the IR or logging analysis results. Due to the customizable nature, LLVM allows insertion of user-written passes. User-written passes are written in C++, and linked into the LLVM compilation schedule. LLVM allows the user to control when a custom pass is run, and what other passes also run alongside the custom pass. Finally, LLVM also contains various compiler backend to lower code to different assembly languages for different architectures.

4. Implementation

-- Give an architectural diagram

-- Refer to figures

4.1 Load Store Movement

Before the Diospyros pass is run, first, the LoadStoreMovement pass occurs. The LoadStoreMovement pass will move load and get-element-pointer (gep) instructions

to be located as far forward as possible in a basic block, while moving store instructions as far backward as possible. The LoadStoreMovement pass respects dataflow dependencies and

memory dependencies. Simple def-use analysis ensures dataflow dependencies in the basic block still hold. I also apply LLVM’s Basic Alias Analysis to ensure memory dependencies remain identical to the original program. Also, instructions are not moved across function calls, because such calls could change memory dependencies, if the function body accesses memory. As an extension, I could later add support for moving instructions across functions like the square root or absolute value functions, because these functions do not access program memory.

To simplify the analysis, we only apply the Diospyros pass on basic blocks that contain certain LLVM internal representation instructions. The instructions are limited to floating point addition, subtraction, multiplication, division, stores, loads, get-element-pointer (gep), bitcast and allocation (alloca) instructions. These instructions occur often in kernel programs and can also map down to architectural instructions in a predictable manner.

When the LoadStoreMovement pass is completed, basic blocks are organized so that there are phi nodes at the beginning, following by multiple repeated sections of organized instructions. Each section consists of load and gep instructions, followed by other instructions, followed by store instructions. Terminators are located at the end of the basic block.

Phi Nodes

(

Loads / Geps

Other Instructions

stores

)\* - Kleene star

Terminators

Why do we need the LoadStoreMovement pass? In a naive scheme, each store instruction in a basic block would form the root of a tree of instructions computing the stored value at the store address. These trees would be fed into Egg, which would apply rewrite rules to optimize vectorized code. A shortcoming to this approach is that memory dependencies are not respected when the code is vectorized.

Consider code that features interleaved load and store instructions, such all loads and stores are to the same address. A figure is shown below.

load1

...

store1

load2

...

store2

Assume that Store 1's data is computed as a function of load1, while store2's data is computed as a function of load2.

When we vectorize, we do load1 and load2 at the same time. We also do store1 and store2 at the same time. A figure showing the parallel operations for loading and storing is shown below:

load1 load2

...

store1 store2

In the vectorized version of the program, a memory dependency between store1 and load2 has been eliminated, compared to the original program. Hence, the vectorized code is incorrect relative to the original specification.

One safe alternative to this incorrect procedure is to only vectorize in a continuous region of instructions between loads and stores, inclusive of the loads and stores, such that the region does not contain a store followed by a load. This condition ensures that any read-after-write dependencies hold after vectorization. We place no restrictions on the load instructions, but we add the following stipulation on stores: no two stores in the region can be to the same address. By adding this stipulation, we guarantee that there are no write-after-write dependencies that are eliminated by vectorization. If two stores were to the same address, a write-after-write dependency could be changed under vectorization, and the result of the vector store instruction to the address would be ambiguous, based on the semantics of the vector instruction. Write-after-read dependencies are maintained after vectorization, because we always generate a vector store or store instruction after the vector loads or loads instructions are created. The only dependencies we will have are read-after-read dependencies, caused by possibly reordering the loads after vectorization. Hence, all memory dependencies are maintained in the vectorized form of the program.

However, these regions of ordered loads and stores in typical code are typically very short, so there is not much vectorization that can be done. In order to increase the potential for vectorization, we make regions larger. One way we make regions larger is to move loads to be as early as possible in a basic block and stores to be as late as possible in a basic block. This motivates the creation of the LoadStoreMovement pass.

4.2 Diospyros

The main pass for vectorization is the Diospyros pass. The Diospyros pass works by finding chunks of instructions that can be sent to the Egg rewriting engine, from which optimized LLVM IR code can be emitted.

To allow more opportunity for vectorization in the Diospyros pass, I apply several transformations to the code, that are typical for vectorization routines, such as loop unrolling and inlining [Larsen & Amarasingh…] These transformations allow basic blocks to be larger, and allow for identification of more vectorization opportunities.

A chunk is a subset of a single basic block, consisting of instructions. A basic block might have 0 or more chunks, and no two chunks can overlap; that is, no instruction is shared between chunks. A chunk cannot include phi nodes or terminator instructions, such as branches, jumps or returns. Each chunk begins at after the last store in the prior chunk, and if there was no prior chunk, then the chunk starts at the beginning of the basic block after the phi nodes, and ends at the last store which is followed by a non-store instruction.

Chunks are primarily composed of instructions that compute a value stored at a certain address via a store instruction in the chunk. These computations that are stored, called Store Trees, are the units of computation that will be vectorized. Chunks may also contain instructions that do not belong to a Store Tree, and these instructions will not be targeted for vectorization.

No other instructions are affected by the Diospyros vectorization process.

As a result, after the vectorization procedure, because each chunk’s optimized instructions are unchanged in relative position within each chunk, and between different chunks, no memory nor dataflow dependencies are affected, leaving the altered program having the same semantics as the original program.

Store Tree Description

Each Store Tree is a tree with a single root. The root must be a store instruction, and its data value is computed from the subtree of the store instruction. The store instruction also has an address, computed by a gep, bitcast, argument value, or another LLVM instruction.

The internal nodes to the tree, which do not include leaves or the root, must be instructions that have a return type that is floating point. Currently, each internal node of the tree must be either an floating point addition, subtraction, multiplication, or division. instruction. Every internal node of the tree must be part of the same chunk and cannot be a part of another chunk. The reason for this constraint is because when the Egg engine rewrites the nodes corresponding to the Store Trees, internal nodes instructions might disappear. If an instruction disappears during rewriting, it is not regenerated by the pass, and therefore cannot be referenced by a future chunk. Thus, to simplify our analysis, we require that internal nodes belong to a single chunk.

The leaves of each Store Tree are either a floating-point number, or a load instruction. The address of the load instruction can be any instruction, not necessarily in the same chunk, and can be a gep, a bitcast, or an argument value. We can assume a gep or bitcast instruction can form the address computation because an address operation does not need to be in the same chunk, as the computation to build the address calculation instruction will never be part of a Store Tree, and hence, it will remain in the basic block after vectorization.

4.3 Rewriting of Egg Code

-- Add a Section at the End, appendix, on the EGG DSL grammar.

-- Add a section in the appendix on Egg Rewrite rules

-- Add a section on the Egg cost model

Next, I discuss how the rewriting procedure occurs. For each chunk in a basic block, a builder is placed after the last instruction in the chunk. Next, all Store Trees in the chunk are collected, in a list of LLVM instructions representing all instructions in all the Store Trees in the current chunk.

Each chunk list is fed to the Rust rewriting program. Each store in the chunk list now corresponds to a unique Store Tree.

Each Store Tree of LLVM instructions is recursively translated to a directed graph of Egg instructions, which is a tree. Computations are duplicated, such that if some Store Tree shares instructions with another Store Tree in the same chunk, then the shared instructions will be cloned recursively, copying the entire subtree. This allows the Egg rewriting engine to apply as many rewrites as possible, unlike when no duplication is allowed.

The Egg graphs are joined together by vector and concatenation operations.

The Egg rewriting engine then applies vector rewrite rules. Using a cost model, the best rewriting program is selected from the saturated E-Graph. At the end, the best rewritten program is translated from Egg instructions into LLVM instructions, proceeding recursively from the root of the Egg graph and building LLVM instructions along the way.

We must be careful to rewrite the topmost Egg Vector nodes into individual LLVM stores instructions. Topmost vector instructions correspond to non-vectorized store instructions or padded constants. For example, we might have (Vec (store ...) 0 0 0) where the last 3 Store Trees are not trees from the original graph, but are constants generated via padding or rewriting to fill all the vector lanes. We replace the Vec node with a NoOptVec node. We individually generate LLVM instructions for each subtree. No store instructions are translated, and since the instructions never modify memory, these instructions and any instructions they transitively depend on can be eliminated easily by LLVM's dead code elimination pass.

At the end of the rewriting process, the original stores are pruned, to also facilitate ded code elimination.

4.4 Post-Vectorization Process

After vectorization, we also apply other optimizations like common subexpression elimination to further clean up repeated code. In addition, aggressive dead code elimination is applied, in order to remove any redundant code that occurs from vectorization. Dead code exists primarily because the Diospyros pass removes any stores that are vectorized, but the pass does not remove any other instructions that compute the stored values. Applying the aggressive dead code elimination pass will transitively remove any instructions that compute the stored values. Then the Diospyros pass is completed. Finally, the vectorized program is then lowered down to hardware specific instructions by the LLVM backend.

5. Evaluation

6. Related and Future Work

This thesis builds on work from auto-vectorization and equality saturation. Work in auto-vectorization includes loop vectorization, super word level parallelism, and solver, rewriting or synthesis-based methods. Work in equality saturation spans many fields, including numerical analysis, graphics, program optimization and other areas of computer science.

6.1 Related Work

There is a rich body of literature studying auto-vectorization. Early attempts at auto-vectorization focused on finding loops that could be parallelized into vector programs. Finding loops without dependencies, to allow for parallelization, was central to the work of […]. Loop vectorization was further developed to target SIMD processors, and this work includes alignment and consecutiveness constraints for loop vectorization, as well as approaches to outer loop vectorization. [Nuzman]

A different approach to auto-vectorization takes advantage of commonly repeated isomorphic and independent operations, called Superword Level Parallelism. [Larsen and Amarasinghe, 2000] The original version SLP uses a greedy algorithm to pair instructions together as vectorizable, beginning at seed instructions, such as consecutive load instructions. [Larsen and Amarasinghe, 2000] Other greedy heuristics were developed to overcome the original algorithm, and consider the total number of reuses a vector pack will have, or work hierarchically, first considering numerous small chains of instructions that can be vectorized, then reducing this set of chains and choosing an instruction chain that leads to the most vectorization. [Shin], [Penn State]. These auto-vectorization approaches find good vector programs on benchmarks, and run efficiently due to the greedy nature of the algorithms, but may fail to find optimal vectorization plans, because the space of vector programs is not fully explored.

Current approaches to auto-vectorization tradeoff compilation time to find near-optimal vectorization opportunities by searching a larger space of programs. goSLP translates functions into integer linear programs representing possible vector packs, consisting of pairs of instructions. [Mendis and Amarasinghe, 2018] goSLP applies a ILP solver to then find an optimal solution in terms of reusing vector packs of size 2. A greedy routine is used to scale up to larger vector pack sizes. As solving ILP problems is NP-Complete, some instances of goSLP can take a long time to solve. However, the authors find this technique still scales well, due to only finding optimality when considering vector packs of pairs. A contrasting approach to using a solver is using machine learning. Mendis explored an alternative, using imitation learning methods to train an algorithm to decide whether instructions should be paired in vector packs [Mendis et al, 2019]. This approach scales much better than an ILP solver, but loses the optimality that goSLP presents. In addition, vectorized programs from this approach are created by a blackbox algorithm and lose interpretability. Another approach is to apply rewriting to find better programs. Diospyros uses Equality Saturation to find near-optimal vectorization solutions. [VanHattum, et al., 2021] However, Diospyros focuses on vectorization for Digital Signal Processors, and therefore assumes that shuffle, gather and scatter instructions are performance efficient. Program synthesis has also been used successfully for auto-vectorization. An work in progress paper for Diospyros explains how synthesis can be used to find better vector programs, but this approach was abandoned because it did not scale up well. [VanHattum, et al., 2020] Another work using synthesis focuses on finding performance efficient backend sequences of architectural instructions, and uses a three phase method of lifting intermediate representation instructions to an abstract set of instructions, lowering to architectural instructions, and finally considering data movement, permutations and shuffling. This approach is found to scale well to … [Ahmad et al., 2022].

E-Graphs and rewriting

E-Graphs are a common technique for solving problems in compilers research. The Denali project used E-Graphs to find near-optimal programs in terms of program size. [Joshi, et al., 2002] E-Graphs were also applied intermediate representation code as a means of implementing compiler optimizations. [Tate, et al., 2009] Diospyros applied E-Graphs to find better vectorized programs for digital signal processors. [VanHattum, et al., 2021].

E-Graphs are also applied to other domains, other than program optimization. E-Graphs were used to find floating point programs with less bits of error in the Herbie Project. [Panchecka et al, …] Another area E-Graphs were used was to find high-level human readable and editable CAD code from mesh compilers in the Szalinski project. [Nandi et al…] E-Graphs were also used to create “compilers” that generate instructions on how a carpenter can cut wood. [Wu et al…] Finally, Ruler used E-Graphs to find small sets of re-write rules. [Nandi et al…] Many of these projects utilize the Egg package, developed to make it easier to use E-Graphs in various settings. [Willsey et al…]

6.2 Future Work

--- IDK cannot do this rn--

7. Conclusion

8. Bibliography

[Ahmad et al., 2022] Ahmad, M. et al. (2022). Vector Instruction Selection for Digital Signal Processors using Program Synthesis. *APSLOS*.

[Allen and Kennedy, 1987] Allen, R. and Kennedy, K. (1987). Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Program. Lang. Syst. 9(4): 491-542.*

[Huh and Tuck, 2017] Huh, J. and Tuck, J. (2017). Improving the effectiveness of searching for isomorphic chains in superword level parallelism*. MICRO*.

[Joshi et al., 2002] Joshi, R. et al. (2002). Denali: A Goal-directed Superoptimizer. *PLDI*.

[Larsen and Amarasinghe, 2000] Larsen, S. and Amarasinghe, S. (2000). Exploiting Superword Level Parallelism with Multimedia Instruction Sets. *PLDI*.

[Liu et al., 2012] Liu, J. (2012). A Compiler Framework for Extracting Superword Level Parallelism*. PLDI*.

[Massalin, 1987] Massalin, H. (1987). Superoptimizer: a look at the smallest program. *ASPLOS*.

[Mendis and Amarasinghe, 2018] Mendis, C. and Amarasinghe, S. (2018). goSLP: Globally Optimized Superword Level Parallelism Framework. *OOPSLA*.

[Mendis et al., 2019] Mendis, C. et al. (2019). Compiler Auto-Vectorization with Imitation Learning. *NeurIPS*.

[Nandi et al., 2020] Nandi, C. et al. (2020). Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations. *PLDI.*

[Nandi et al., 2021] Nandi, C. et al. (2021). Rewrite Rule Inference Using Equalit Saturation. *OOPSLA*.

[Nuzman et al., 2006] Nuzman, D. et al. (2006). Auto-vectorization of interleaved data for SIMD. *PLDI*.

[Nuzman et al., 2008] Nuzman, D. et al. (2008). Outer-loop vectorization : revisited for short SIMD architectures. *PACT*.

[Panchekha et al., 2015] Panchekha, P. et al. (2015). Automatically Improving Accuracy for Floating Point Expressions. *PLDI*.

[Porpodas et al., 2015] Porpodas, V. et al. (2015). PSLP : Padded SLP Automatic Vectorization. *CGO*.

[Porpodas, 2017] Porpodas. V. (2017). SuperGraph SLP Auto-Vectorization. *PACT.*

[Porpodas, 2018] Porpodas, V. et al. (2018). Look-Ahead SLP : Auto-Vectorization in the Presence of Commutative Operations*. CGO*.

[Porpodas et al., 2019] Porpodas, V. et al. (2019). Super-Node SLP : Optimized Vectorization for Code Sequences Containing Operators and Their Inverse Elements*. CGO*.

[Tate et al., 2009] Tate, R. et al. (2009). Equality Saturation: a New Approach to Optimization. *POPL.*

[VanHattum et al., 2020] VanHattum, A. et al. (2020). A Synthesis-Aided Compiler for DSP Architectures (WiP Paper). *LCTES*.

[VanHattum et al., 2021] VanHattum, A. et al. (2021). Vectorization for Digital Signal Processors via Equality Saturation. *ASPLOS*.

[Willsey et al., 2021] Willsey, M. et al. (2021). egg: Fast and Extensible E-graphs. *POPL*.

[Wu et al. 2019] Wu, C. et al. (2019). Carpentry Compiler. *ACM Trans. Graph, 38(6).*

9. Appendix

Egg Domain Specific Language Grammar

Egg ::=

| Num(n)

| Arg(a)

| Load()

| Store()

| VecAdd()

| VecSub()

| VecMul()

| VecDiv()

| VecMAC()

N ::= Int

Egg Rewrite Rules