EXTENDING DIOSPYROS AUTO-VECTORIZATION TO LLVM

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Jonathan Tran

August 2023

ABSTRACT

Short vector instruction sets are ubiquitous on computers. Auto-vectorizing compilers help programmers bridge the gap by automatically mapping domain specific codes to efficient sequences of vector instructions on domain specific architectures. This thesis builds on the Diospyros project to implement a Diospyros LLVM pass translating linear algebraic kernels to SIMD vector architectures. The pass uses the idea from the Diospyros project to apply a search-based equality saturation method to find vector instructions. Results show that the Diospyros LLVM pass generates vector code with good runtime performance on stencil and convolution kernels, but poor results on other kernels indicate future work is required.

BIOGRAPHICAL SKETCH

Jonathan Tran is a master's student at Cornell University. He studied at Cornell for his undergraduate degree from 2017 to 2021, majoring in statistics and computer science. At Cornell, he was an undergraduate consultant and teaching assistant for six semesters, and a graduate teaching assistant for four semesters. As part of Professor Anne Bracy's course staff for CS 3410 and CS 1110, he contributed to new assignments and the auto-grader infrastructure. Jonathan will be heading to Green Hills Software as a test engineer, where he hopes to contribute to producing high quality embedded operating systems, compilers and debuggers.

ACKNOWLEDGEMENTS

I am incredibly grateful to Professor Adrian Sampson and Dr. Alexa VanHattum for their guidance, patience, and wisdom they gave me throughout this project. They provided me with a unique academic experience, a chance to contribute to research, and an opportunity that allowed me to find a job after college. I am thankful for Professor Zhiru Zhang for serving as a minor advisor. I owe much to Jasper Liang, who was an excellent partner on the Diospyros project. Finally, I am indebted to my parents and my sister for their support and motivation they provided me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Developments in computer architecture has made short vector architectures common, such as with SIMD extensions or in digital signal processors. These short vector architectures have the ability to compute instructions of the same type at the same time; for example, four additions can be computed the same time. This ability, provided by vector instructions, allows programmers to write more performance efficient code, because there is more computation in every clock cycle.

General purpose programming languages, like the C programming language, can be compiled to these short vector architectures, but often, the full performance is not enabled. The compiler may fail to detect parallelism, or fail to generate profitable parallel assembly instructions. Compiler generated code still has room to improve runtime performance.

Bridging this runtime performance gap is a central focus 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. These methods range 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 fixed 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 or vendor written code, as the Diospyros paper shows [24]. However, a major tradeoff is that compilation time takes longer, because the compilation process searches a space of numerous candidate programs.

One area where a runtime performance gap exists from compilation is with linear algebraic 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 using a greedy heuristic, and thus sometimes does not find vectorization opportunities.

Diospyros presents a novel method of auto-vectorization of kernel functions

for digital signal processors [24]. Diospyros finds more efficient vector code by reducing the compilation problem to equality saturation. In equality saturation, rewrite rules are repeatedly applied to a program to transform it. Equality saturation uses a special data structure 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 is currently implemented in a domain specific language, and only targets a Tensilica digital signal processor. This thesis extends prior research on Diospyros to apply to a wider set of programs, by allowing for C language programs to be compiled. In addition, this thesis will apply vectorization on the LLVM intermediate representation, rather than on a high-level language or a domain specific language. Finally, this thesis will investigate whether using Diospyros to vectorize a program for a SIMD architecture can be profitable, by considering alignment and consecutiveness of memory addresses.

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 Diospyros auto-vectorization pass for LLVM. I will discuss results of the Diospyros auto-vectorization pass on benchmarks. Finally, I will discuss related works, and conclude.

CHAPTER 2

BACKGROUND

The sections of the background chapter will discuss aspects of auto-vectorization, equality saturation, the Diospyros project, and LLVM, relevant to the thesis.

2.1 Auto-vectorization

Auto-vectorization is the process of automatically converting sequential code into parallel code by using vector instructions. Unlike a scalar instruction, which does one operation at a time, a vector instruction can do multiple identical 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.

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, consuming 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; instead the code 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 exper-

imentation 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 without 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, without being cluttered by optimization specific code. A final advantage for auto-vectorized code is that the compiler backend can 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 instructions are repeated across iterations of a loop and some of these instructions can be vectorized. One challenge to loop vectorization is when there exist inter-iteration dependencies between statements that are to be vectorized, which hinders vectorization because vectorization would alter 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 second form of auto-vectorization is super-word-level parallelism vectorization, otherwise called SLP vectorization [5]. SLP vectorization takes advantage of commonly found short sequences of repeated of operations in basic blocks of code. Each sequence of repeated operations can be transformed to use parallel hardware instructions. To create more repetition of operations, SLP

unrolls loops in the program. SLP groups sequences of isomorphic and independent 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 case vectorization stops. Last, the benefit of the vectorized program is measured using a cost model, and if the vectorized program is deemed beneficial, vector code is emitted. While SLP can be effective, a major limitation is the greedy approach to vectorization, which can miss more profitable vector packing strategies.

A different formulation of auto-vectorization is as a search problem. Compared to greedy methods, such as SLP, this form of auto-vectorization expands the ability to find profitable and more-optimal vector code by searching a larger space of programs. Besides reduction to program synthesis or integer linear programming, another approach is by reduction to equality saturation [24]. This approach is the central method used in this thesis.

2.2 Equality Saturation

Prior approaches to compilation suffer from the problem of destructive rewrites. Running one sequence of rewrites may result in a better solution than another sequence of rewrites. However, some compilers try only one sequence of rewrites. As a result, the rewrites are ”destructive”; the compiler does not maintain the old state before the rewrites were applied.

Equality saturation is a compilation technique that avoids the problem of destructive rewrites by representing all possible combinations of applications

of rewrite rules. A special data structure called the equality graph (E-graph) is used to store all of the combinations in a compact manner.

The E-graph is a data structure that compactly stores equivalences between terms. E-graphs remain compact by not having duplicate nodes via node reuse. Structurally, an E-graph is composed of equality classes, or E-classes. Each E-class is a set of terms. Each term in an E-class is equivalent. Every term in a E-class represents a function with zero or more arguments, and each argument points to one E-class.

An E-graph is manipulated with rewrite rules. A rewrite rule is a pair of a term to match and the result of the match. If a rewrite rule matches some set of nodes in the E-graph, the result of the match is generated and added to the E-graph. Adding a new result to the E-graph may cause E-classes to merge together when two classes are shown to be equivalent via a rewrite rule.

Equality saturation operates by filling an E-graph with equivalent program representations. Starting from an initial set of program representations in the E-graph, rewrite rules match nodes in the E-graph and add more equivalent programs to the E-graph. When no additional equivalent programs can be added via the rewrite rules, the E-graph has been saturated, which means all combinations of rewrites in any order are all represented. After this, an optimal program is extracted from the E-graph.

Extraction routines can be greedy, such as the routine employed in Diospyros, which traverses e-classes and chooses the term with minimal cost in each E-class, while also considering attributes of the child e-classes of the current e-class [24]. More sophisticated extraction routines can consider the relationships

between terms in the E-graph and use integer linear programming and boolean satisfiability solvers to guarantee optimality of the extracted program, under some criterion of optimality [25] [4].

As an example of equality saturation applied to vectorization, consider figure 2.1. The left hand side shows the original E-graph, with a program containing two independent additions that could be vectorized. The rewrite rule Vec(Add(?A, ?B), Add(?C, ?D)) ⇝ VecAdd(Vec(?A, ?C), Vec(?B, ?D)) is then applied. The modified E-graph is shown on the right hand side, capturing the original program, and a vectorized variant.

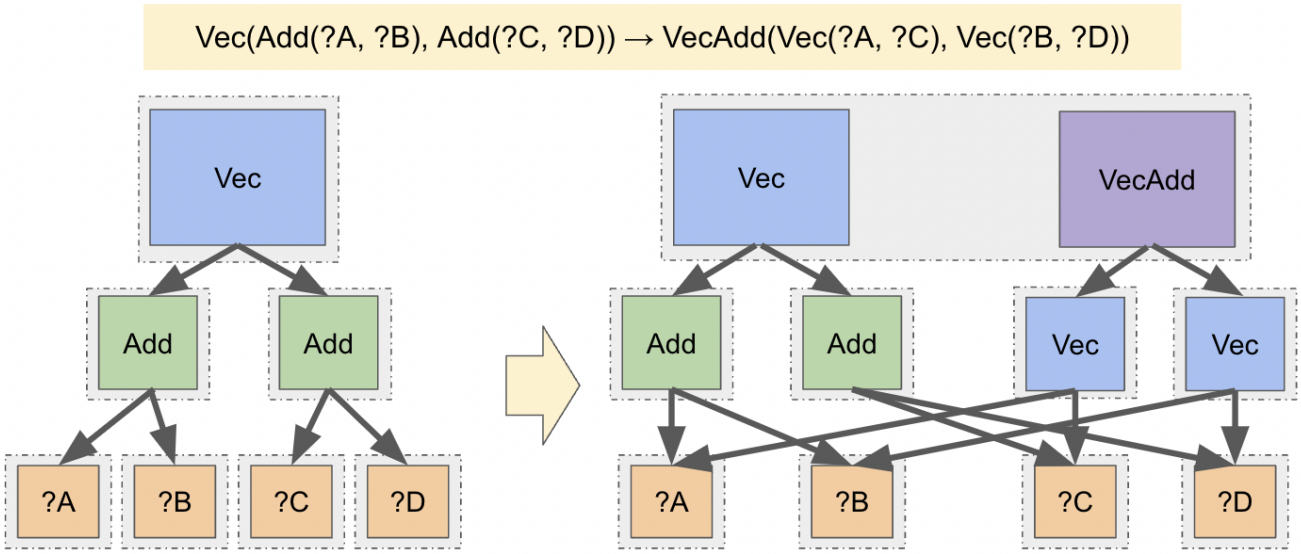

Figure 2.1: Example of rewriting vector instructions under the given rewrite rule. Each dashed box represents an E-class. The Vec and VecAdd nodes are in the same E-class.

Continuing this example, we can extract a program from the E-graph. Figure 2.2 shows greedy extraction of the program with lowest cost from the E-graph. Extraction begins by traversing E-classes and considering the cheapest cost E-node within the class.

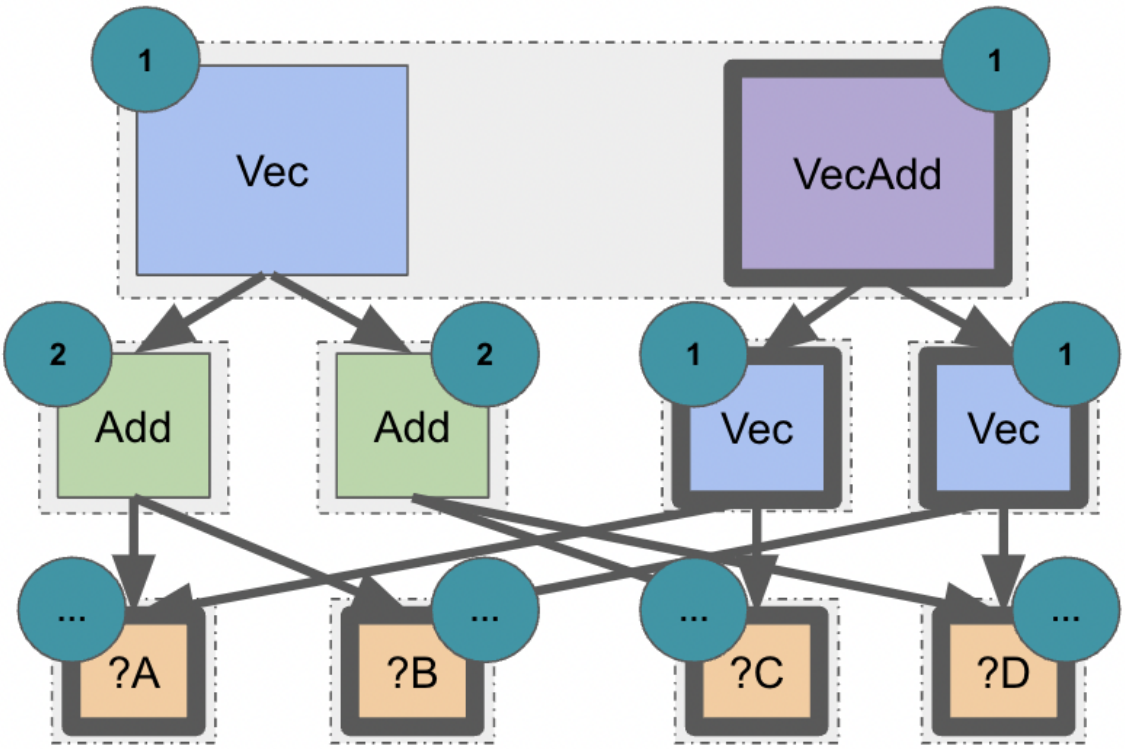

Figure 2.2: Example of extracting program from E-graph. Costs are shown in circles and the greedily selected program is shown with bolded edges.

This thesis uses the Egg software package [25] for equality saturation. Egg is a customizable equality graph rewriting package, in which users can define a domain specific language on which rewrites are performed. The user can also define rewrites. Egg contains the rewrite engine, which applies rewrites to the E-graph, tools for matching and applying rewrite rules, and custom extraction logic that can use greedy cost models or integer linear programming (ILP) solvers.

2.3 Diospyros

Diospyros, created by VanHattum et al., is a vectorizing compiler targeting digital signal processors [24]. Diospyros uses equality saturation to search for optimized vector programs for digital signal processors. The input program to Diospyros is a finite kernel. A kernel is a function that maps array inputs to array outputs. Diospyros begins by using Rosette to symbolically execute the kernel and extract a high-level specification [22]. The specification is a mathematical formula that describes how each output address of the kernel program is computed, as a function of the input array addresses. Diospyros then converts the specification to a program in the vector language, a minimal language that can be optimized. The vector language program is passed through the Egg rewriting engine, which applies custom rewrite rules to transform the program into a vectorized form. After the E-graph is saturated, the best program is chosen from the E-graph via a cost model. Finally, Diospyros will lower the chosen program to C++ code using Tensilica digital signal processor intrinsic instructions.

Diospyros holds several advantages over other vectorization strategies. First, for small and irregular sized kernels, Diospyros is able to find vectorized programs where other vectorization algorithms fail. Using rewrite rules, Diospyros considers a larger space of programs compared to greedy vectorization heuristics. Diospyros also effectively uses data shuffling operations, like gather instructions, for digital signal processors. Diospyros's strategy means that more programs are explored, and some of these programs are profitably vectorized. Even when a greedy heuristic will vectorize a program, Diospyros can vectorize the program more effectively, by finding better assignments

to vector registers. Second, Diospyros also simultaneously considers a smaller space of transformations compared to other techniques enumerative techniques. For example, when compared to vectorization using an ILP solver, which might consider all pairs of vectors, Diospyros only considers equivalent transformations of programs via rewrite rules. As a result, fewer programs are sampled. This means the compilation is faster, yet the vectorized programs remain efficient at runtime.

2.4 LLVM

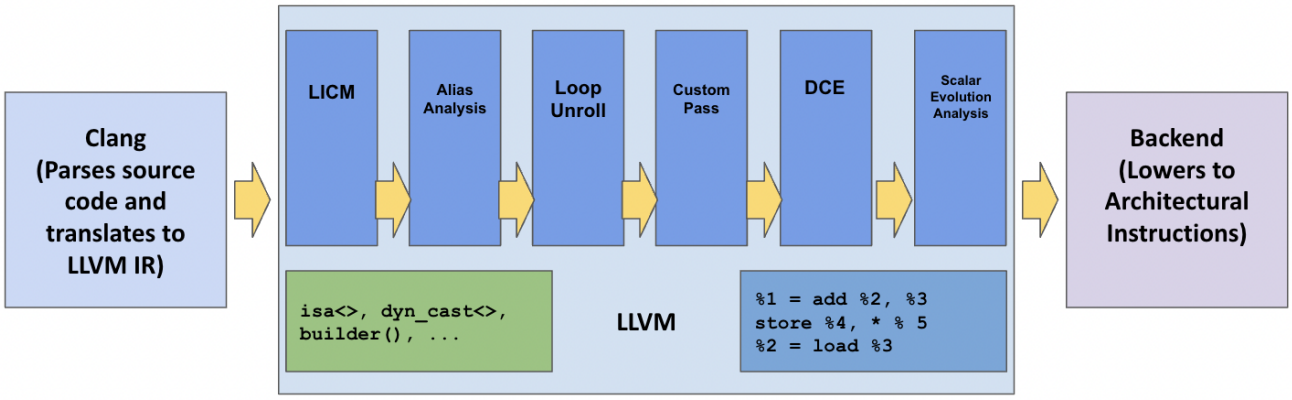

Figure 2.3: LLVM Infrastructure.

LLVM is a customizable middle-end infrastructure to develop compilers [6], as shown in figure 2.3. LLVM contains an internal representation language (LLVM IR), which is used to represent C programs. The IR contains standard assembly instructions, such as arithmetic instructions, vector instructions, and vector intrinsic instructions. LLVM has many utility functions to interact with the IR, such as functions to build and insert new instructions. LLVM also provides optimization and analysis routines. These routines operate on the IR,

either transforming the program or logging analysis results. This thesis uses LLVM's alias analysis and scalar evolution analysis passes, and uses dead code elimination, loop unrolling and other common transformation passes. Due to the customizable nature, LLVM allows insertion of user-written passes to manipulate the IR. User-written passes are written in C++, and linked into the LLVM. LLVM allows the user to schedule when a custom pass is run, and what other passes also run alongside the custom pass. Finally, LLVM also contains various compiler backends to lower IR code to different assembly languages for different architectures.

CHAPTER 3

IMPLEMENTATION

This chapter discusses the implementation of the Diopsyros auto-vectorization pass in LLVM. At a high level, there are six stages to compilation, as shown in figure 3.1. First, pre-processing is applied to increase opportunity to apply vector instructions. The Load-Store-Movement pass then moves memory instructions in basic blocks to allow for larger regions of instructions without memory dependencies. Next, LLVM code is translated to the Vector DSL, the Vector DSL is rewritten via the Egg engine, and the Vector DSL code is translated back to LLVM IR instructions. Finally, clean-up passes are applied.

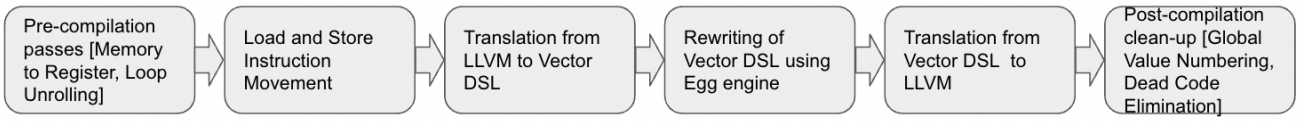

Figure 3.1: Compilation pipeline for vectorization.

3.1 Load-Store-Movement Pass

To begin the vectorization process, the first pass to run is the Load-Store-Movement pass. The Load-Store-Movement pass moves load and get-element-pointer (gep) instructions to be located as early as possible in a basic block, while moving store instructions as late as possible. The Load-Store-Movement pass respects dataflow dependencies and memory dependencies. Simple def-use analysis ensures dataflow dependencies in the basic block still hold. Mem-

ory dependencies are maintained by making queries LLVM's Basic Alias Analysis pass. If two instructions point to potentially aliasing memory addresses, then instruction movement does not occur. Also, instructions are not moved across function calls, because such calls could change memory dependencies, if the function body accesses memory. As a future extension, memory instructions could be moved before or after special functions such as the square root or absolute value functions, because these functions do not modify memory.

To simplify the analysis, the Diospyros pass is only applied on basic blocks that contain certain LLVM IR 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 be optimized by the Diospyros pass.

When the Load-Store-Movement 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.

Consider a running example of vector-vector addition, shown in panel (A) of figure 3.2. Panel (B) shows LLVM instructions before the load-store-movement pass has been applied. Panel (C) shows the result after the load-store-movement pass has been applied, with load instructions at the beginning, addition operations in the center, and store instructions at the end. There were no dependencies between memory instructions in this example, the load and store instructions could be moved.

```
#define SIZE 4

void sum(float a_in[restrict SIZE], float b_in[restrict SIZE], float c_out[restrict SIZE]) {

    for (int i = 0; i < SIZE; i++) {

        c_out[i] = a_in[i] + b_in[i];

    }
}
```

(A): Vector-Vector addition code

```
%4 = load float, float* %0, align 4
%5 = load float, float* %1, align 4
%6 = fadd float %4, %5
store float %6, float* %2, align 4
%8 = load float, float* %7, align 4
%10 = load float, float* %9, align 4
%11 = fadd float %8, %10
store float %11, float* %12, align 4
%14 = load float, float* %13, align 4
%16 = load float, float* %15, align 4
%17 = fadd float %14, %16
store float %17, float* %18, align 4
%20 = load float, float* %19, align 4
%22 = load float, float* %21, align 4
%23 = fadd float %20, %22
store float %23, float* %24, align 4
```

(B): Code before load store movement

```
%6 = load float, float* %5, align 4
%8 = load float, float* %7, align 4
%11 = load float, float* %10, align 4
%13 = load float, float* %12, align 4
%16 = load float, float* %15, align 4
%18 = load float, float* %17, align 4
%19 = load float, float* %1, align 4
%20 = load float, float* %0, align 4
%21 = fadd float %20, %19
%22 = fadd float %18, %16
%23 = fadd float %13, %11
%24 = fadd float %8, %6
store float %24, float* %4, align 4
store float %23, float* %9, align 4
store float %22, float* %14, align 4
store float %21, float* %2, align 4
```

(C): Code after load store movement

Figure 3.2: Example of the load store movement pass applied to a vector-vector addition program. Certain LLVM instructions have been omitted.

3.1.1 Motivation for the Load-Store-Movement Pass

Why is the Load-Store-Movement pass used? 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, with all loads and stores from the same address. Panel (A) of figure 3.3 shows a small segment of code with load and store instructions. Note that the second load instruction reads from the address of the first store instruction.

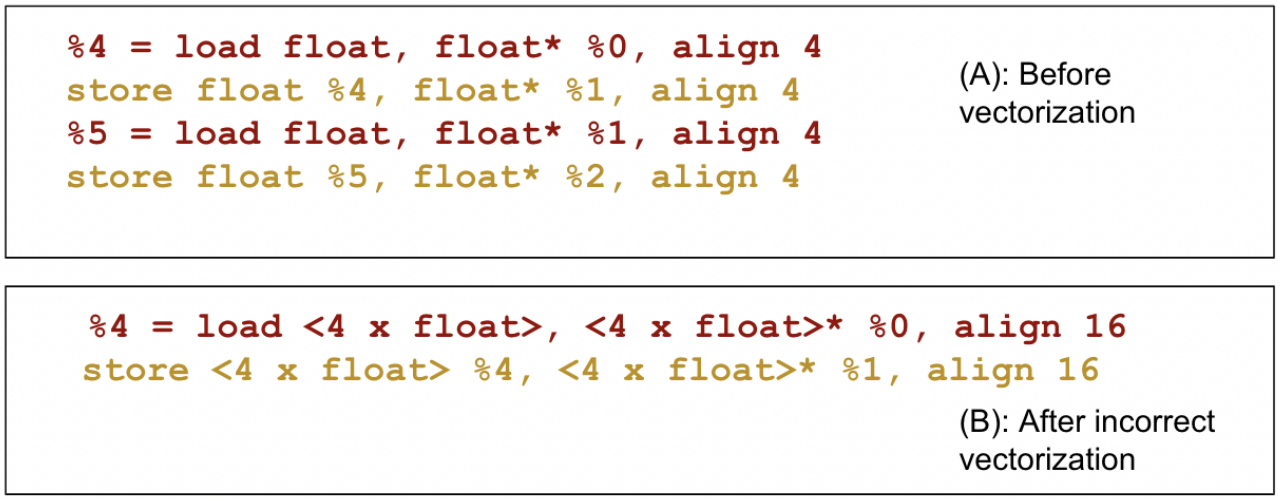

Figure 3.3: Example of incorrect vectorization.

After incorrect vectorization, both load instructions are executed at the same time with a vector load instruction. A similar situation occurs for the stor instructions. This is shown in panel (B) of figure 3.3.

In the vectorized version of the program, a memory dependency between the second load instruction and the first store instruction has been eliminated, when 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. Four outcomes are necessary for vectorization to be correct, and these outcomes consider all four combinations of write and read dependencies. For the first outcome, we require the region must not contain a store followed by a load. This condition ensures that any read-after-write dependencies hold after vectorization.

For the second outcome, we require no two stores in the region can be to the same address. This condition guarantees 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.

Third, write-after-read dependencies are maintained after vectorization, because a vector store or store instruction is always generated after the vector loads or loads instructions are created.

Fourth, read-after-read dependencies also remain in the program, and they can be reordered when the loads are moved around during vectorization. Changing read-after-read dependencies does not change the semantics of the program. Hence, all memory dependencies are maintained in the vectorized form of the program.

In practice, regions of ordered loads and stores in typical code are observed to be short, so vectorization is limited. To increase the potential for vectorization, regions are made larger. One way to 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 Load-Store-Movement pass.

3.2 Diospyros Pass

The main LLVM pass for vectorization is the Diospyros pass. The Diospyros pass finds chunks of instructions that are sent to the Egg rewriting engine, from

which optimized LLVM IR code is emitted.

To allow more opportunity for vectorization in the Diospyros pass, several transformations are applied to the code. These transformations are typical for vectorization routines and include loop unrolling and inlining [5]. The code transformations allow basic blocks to have more instructions, and possibly more opportunities for vectorization.

After the transformations, instructions are sorted into chunks, which are then divided into store trees. Chunks and store trees are defined in the following subsections.

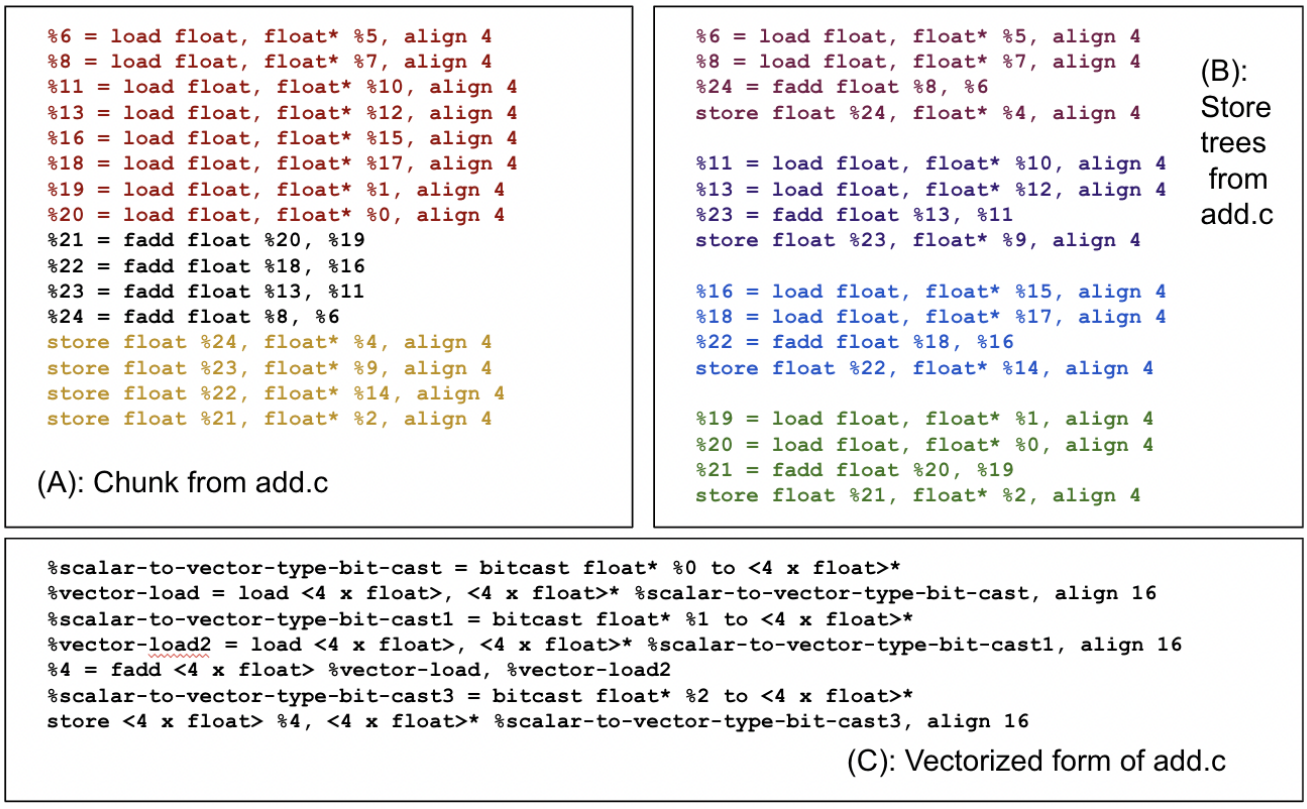

Figure 3.4: Example of chunks and store trees extracted from a vector-vector addition program. Certain LLVM instructions have been omitted.

3.2.1 Chunk Description

Vectorization is applied to chunks of LLVM instructions. A chunk is a subset of a single basic block. 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 nor terminator instructions, such as branches, jumps or returns. Each chunk begins 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. Each chunk ends at the last store which is immediately followed by a non-store instruction. Panel (A) of figure 3.4 demonstrates a chunk from a program adding two vectors together.

Store instructions in chunks can be traversed backwards recursively via use-def relationships. The recursive traversal accumulates a subset of instructions within the chunk which compute a value that the store instruction stores into memory. Those subset of instructions, called Store Trees, are the units of computation to be vectorized. Chunks may also contain instructions that do not belong to any Store Tree, and these instructions will not be targeted for vectorization. No other instructions are affected by the Diospyros vectorization process.

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.

3.2.2 Store Tree Description

Each Store Tree is a tree comprised of LLVM values. The root must be a store instruction, and its data value is composed of the subtree formed from the data field of the store instruction. The store instruction also has an address, computed by a gep, bitcast, argument value, or another LLVM instruction. Panel (B) of figure 3.4 demonstrates four different store trees from a program adding two vectors together.

The internal nodes to the tree, which do not include leaves or the root, must be instructions with a floating point return type. 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 might disappear, meaning that the instruction corresponding to that node is never generated. If an instruction disappears during rewriting, it is not regenerated by the pass, and therefore cannot be referenced by a future chunk. To simplify the analysis, internal nodes are required 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, and can be a gep, a bitcast, or an argument value.

To ensure correctness, address computations consist of LLVM instructions satisfying several conditions. First, the instructions in the computation tree for the address must be contained in one basic block. Second, the address computation tree cannot contain memory instructions such as load instructions.

3.3 Rewriting of Egg Code

Next, the LLVM Code is rewritten into a vectorized form. For each chunk in a basic block, a builder is placed after the last instruction in the chunk. All Store Trees in the chunk are collected as a list of LLVM instructions. Store Trees are grouped in groups of size four, the vector lane width. In these groups, called a subchunk, all four store instructions are consecutive, and the first store is to an aligned address.

Each subchunk is passed to the rewriting routine. Each Store Tree within the subchunk is recursively translated to a directed graph of Egg instructions, which also forms a tree. Computations are duplicated, such that if some Store Tree shares instructions with another Store Tree in the same subchunk, 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 operations. Figure 3.5 shows the result on the running example.

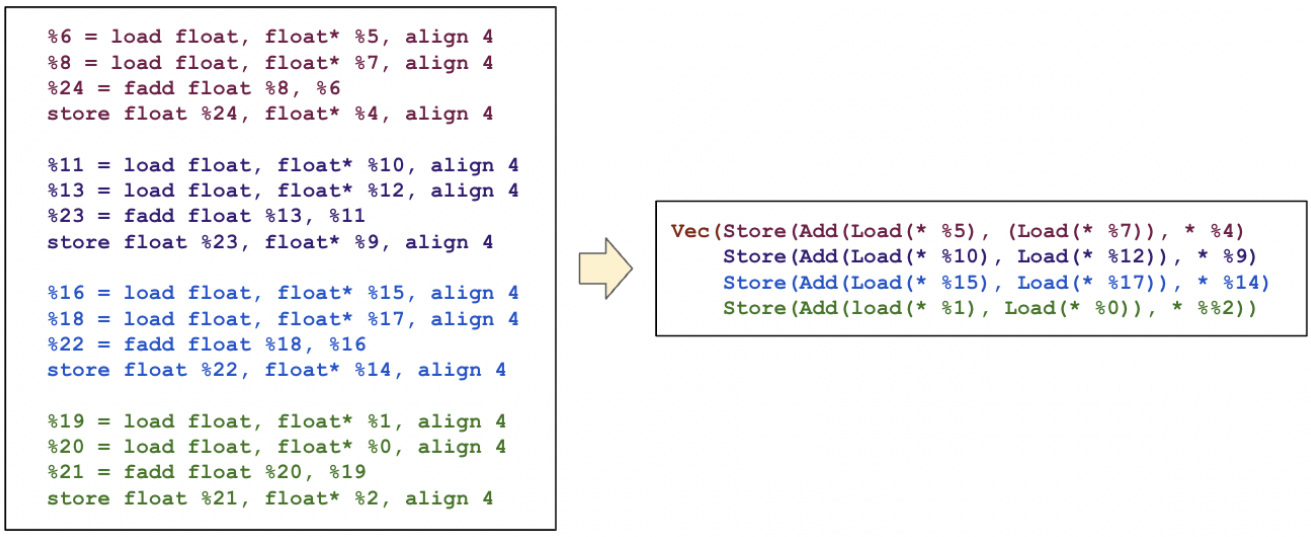

Figure 3.5: Translation from LLVM to Egg DSL.

The Egg rewriting engine then applies vector rewrite rules. Using a cost model, the best rewriting program is selected from the saturated E-graph. Figure 3.6 shows the selected rewrite on the example. 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. Figure 3.7 shows translation back to llvm.

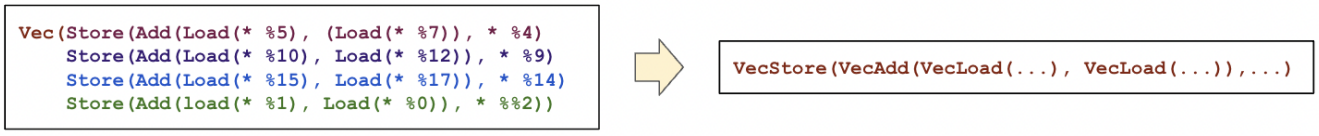

Figure 3.6: Rewriting of Egg DSL.

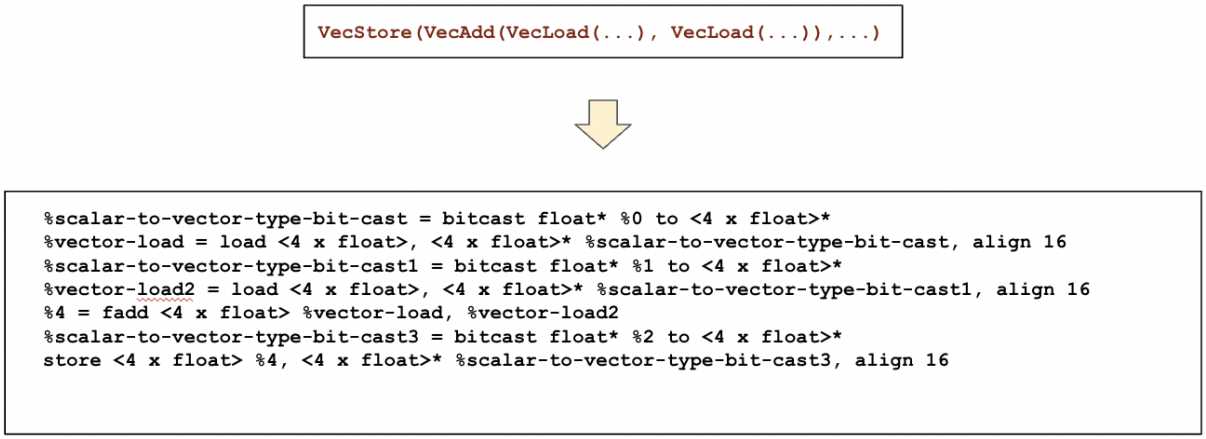

Figure 3.7: Translation from Egg DSL to LLVM.

The topmost Egg Vector nodes are rewritten into individual LLVM stores instructions. Topmost vector instructions correspond to non-vectorized store instructions or padded constants. For example, the rewritten engine might generate the node (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. The vector node is replaced with a node indicating

no vectorization is done. After this, LLVM instructions are generated 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.

After each chunk is vectorized by the rewriting process, the vectorized result is cached. When a chunk has identical store trees, modulo the leaves of the tree, the cached result can be reused. Figure 3.8 demonstrates how the caching occurs for a chunk in the add.c example. Empirically, certain types of kernels, like stencils, lead to many cached results, saving a lot of time in the compilation process.

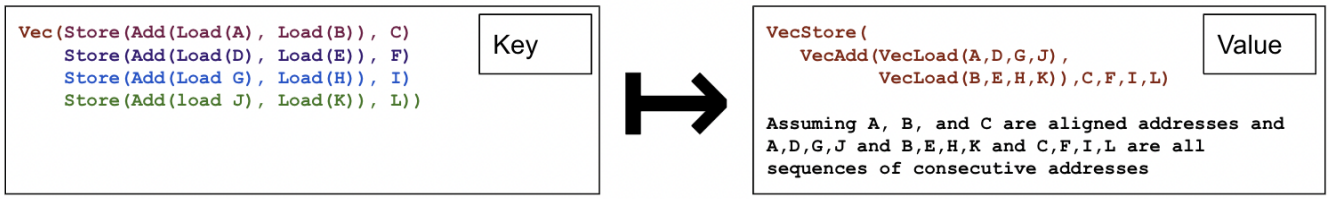

Figure 3.8: Caching Vectorized Chunks.

At the end of the rewriting process, the original stores are pruned, to also facilitate dead code elimination. Panel (C) of figure 3.4 demonstrates the vectorized code from the the original program using loops to add two vectors together.

3.4 Post-Vectorization Process

After vectorization, other optimizations are applied, such as common subexpression elimination to further clean up repeated code. Aggressive dead code elimination is applied, 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. Finally, the vectorized program is then lowered down to hardware specific instructions by the LLVM backend.

3.5 Testing

A test suite of 120 programs were used to validate the compilation pass. Each program contains assertions that must hold after vectorization is complete. Some of these programs were small vectorizable kernels, while others were larger programs added in via regression testing. Using the small kernels helped debug common problems, while the larger kernels, with randomly populated matrix values, often detected more subtle mistakes in the implementation of the compilation pass. The test suite was an invaluable method to incrementally develop the compilation pass.

CHAPTER 4

EVALUATION

To evaluate the vectorization results, the Diospyros pass is compared to the baseline program, and the LLVM SLP vectorization pass, and the runtime performance speedup is calculated. Results are discussed, and methods for potential improvement are discussed.

4.1 Experiment Setup

Each benchmark is compiled in three ways. The first way is to use the O3 optimization level, but without any vectorization. The second way uses LLVM's SLP vectorization pass. The third way uses the Diospyros vectorization pass. Intermediate passes in compilation, such as loop unrolling, are done identically.

After each kernel is vectorized, the kernels are executed for ten million times each. The wall clock time for execution over the iterations is counted, and the average time per iteration is reported. Then, the performance improvement is measured by comparing the average runtime for SLP and Diospyros vectorization strategies to the baseline.

An Apple M1 computer using MACOS 12.2 with 8 GB RAM and 251 GB Flash memory was used for experiments.

4.1.1 Benchmark Descriptions

The original set of kernel benchmarks were taken from the Diospyros project and are common kernel operations. The benchmarks were 2d-convolution, 2d-stencil, qr-decomposition, matrix multiplication, and quaternion-product. Various matrix dimensions are chosen for each benchmark, and are shown in table 4.1.

Each benchmark is written in the C language. Input data to the benchmark is generated randomly and populated into arrays. The type of the data being computed on is always the float type. The input arrays and output array are passed as arguments into the main kernel function, and these arguments are marked with the restricted keyword, to allow the compiler to know the arrays addresses do not alias. In addition, some kernels call user-defined functions. These user-defined functions, which include matrix transpose and dot product routines, are always inlined into the main kernel, via compiler inlining directives. No other compiler directives are used. There are some kernels that also call other predefined functions. These functions include math library functions like the absolute value function, but also include standard library functions like *calloc* and $free$.

4.2 Results and Discussion

Table 4.1 shows average execution times for all three compilation methods over all benchmarks. For QR Decomposition and Matrix Multiplication, various sizes of matrices are used. For stencils, various sized matrices are used, and the sten-

cil neighborhood is also changed for small neighborhood sizes. Similarly, different sized matrices are tried for the two matrices the convolution is being applied to.

Benchmark name	Baseline avg (s)	SLP avg (s)	Diospyros avg (s)
2×2 qr decomp	5.1900e-05	1.8770e-04	2.0250e-04
3×3 qr decomp	6.2180e-04	6.7100e-04	6.9320e-04
4×4 qr decomp	1.0648e-03	1.1612e-03	1.1410e-03
5×5 qr decomp	1.5842e-03	1.8499e-03	1.8311e-03
6×6 qr decomp	2.2060e-03	2.5880e-03	2.6048e-03
2×2 mat mul	1.5000e-06	1.7000e-06	2.3000e-06
3×3 mat mul	4.6000e-06	7.3000e-06	6.8000e-06
4×4 mat mul	1.1100e-05	1.2500e-05	7.8000e-06
5×5 mat mul	2.1900e-05	1.9400e-05	2.4600e-05
6×6 mat mul	3.8500e-05	3.9900e-05	3.7300e-05
7×7 mat mul	6.4300e-05	2.8300e-05	6.7200e-05
8×8 mat mul	9.4400e-05	3.2100e-05	3.7100e-05
9×9 mat mul	1.2640e-04	8.9600e-05	1.2870e-04
10×10 mat mul	1.7470e-04	1.2520e-04	1.3810e-04
11×11 mat mul	2.3830e-04	1.7330e-04	2.3410e-04
12×12 mat mul	3.0320e-04	1.9280e-04	1.6260e-04
15×15 mat mul	6.2240e-04	3.4610e-04	5.8120e-04
16×16 mat mul	7.7340e-04	3.6520e-04	3.5800e-04
4×4,2×2 stencil	3.0000e-06	3.9000e-06	3.3000e-06
5×5,2×2 stencil	6.4000e-06	7.9000e-06	7.5000e-06
6×6,2×2 stencil	1.1100e-05	1.4300e-05	1.2900e-05
8×8,2×2 stencil	2.5700e-05	3.1000e-05	2.2700e-05
12×12,2×2 stencil	6.9500e-05	8.4900e-05	6.0800e-05
16×16,2×2 stencil	1.7330e-04	1.6550e-04	1.1930e-04
4×4,3×3 stencil	6.5000e-06	8.0000e-06	7.7000e-06
5×5,3×3 stencil	1.6500e-05	1.7400e-05	1.6600e-05
6×6,3×3 stencil	2.6300e-05	3.1300e-05	2.5100e-05
8×8,3×3 stencil	5.7800e-05	6.8700e-05	4.9800e-05
12×12,3×3 stencil	1.6850e-04	1.9080e-04	1.5690e-04
16×16,3×3 stencil	3.2760e-04	3.7110e-04	3.0720e-04
3×3,2×2 conv	9.9000e-06	1.1100e-05	9.0000e-06
3×3,3×3 conv	3.4000e-05	2.4300e-05	1.7400e-05
3×3,4×4 conv	6.5000e-05	4.2800e-05	2.7800e-05
4×4,2×2 conv	2.5700e-05	1.9400e-05	1.3900e-05
4×4,4×4 conv	2.6520e-04	7.7400e-05	5.0500e-05
5×5,2×2 conv	3.5800e-05	2.9500e-05	2.1200e-05
5×5,3×3 conv	8.8500e-05	6.6600e-05	4.3200e-05
5×5,4×4 conv	3.7260e-04	1.2300e-04	7.0500e-05
6×6,2×2 conv	5.0900e-05	4.2200e-05	2.7700e-05
6×6,3×3 conv	2.5460e-04	9.7300e-05	6.0300e-05
6×6,4×4 conv	5.2610e-04	1.7950e-04	1.1050e-04
8×8,2×2 conv	1.5540e-04	7.5400e-05	4.6400e-05
8×8,3×3 conv	4.1850e-04	1.6920e-04	9.2500e-05

8×8,4×4 conv	9.7880e-04	3.2590e-04	1.7870e-04
qprod	5.4000e-06	6.2000e-06	5.8000e-06

Table 4.1: Runtime Performance

Figures 4.1, 4.2, 4.3, 4.4 and 4.5 aggregate each type of benchmark together and show the performance speed up of SLP and Diospyros relative to the baseline.

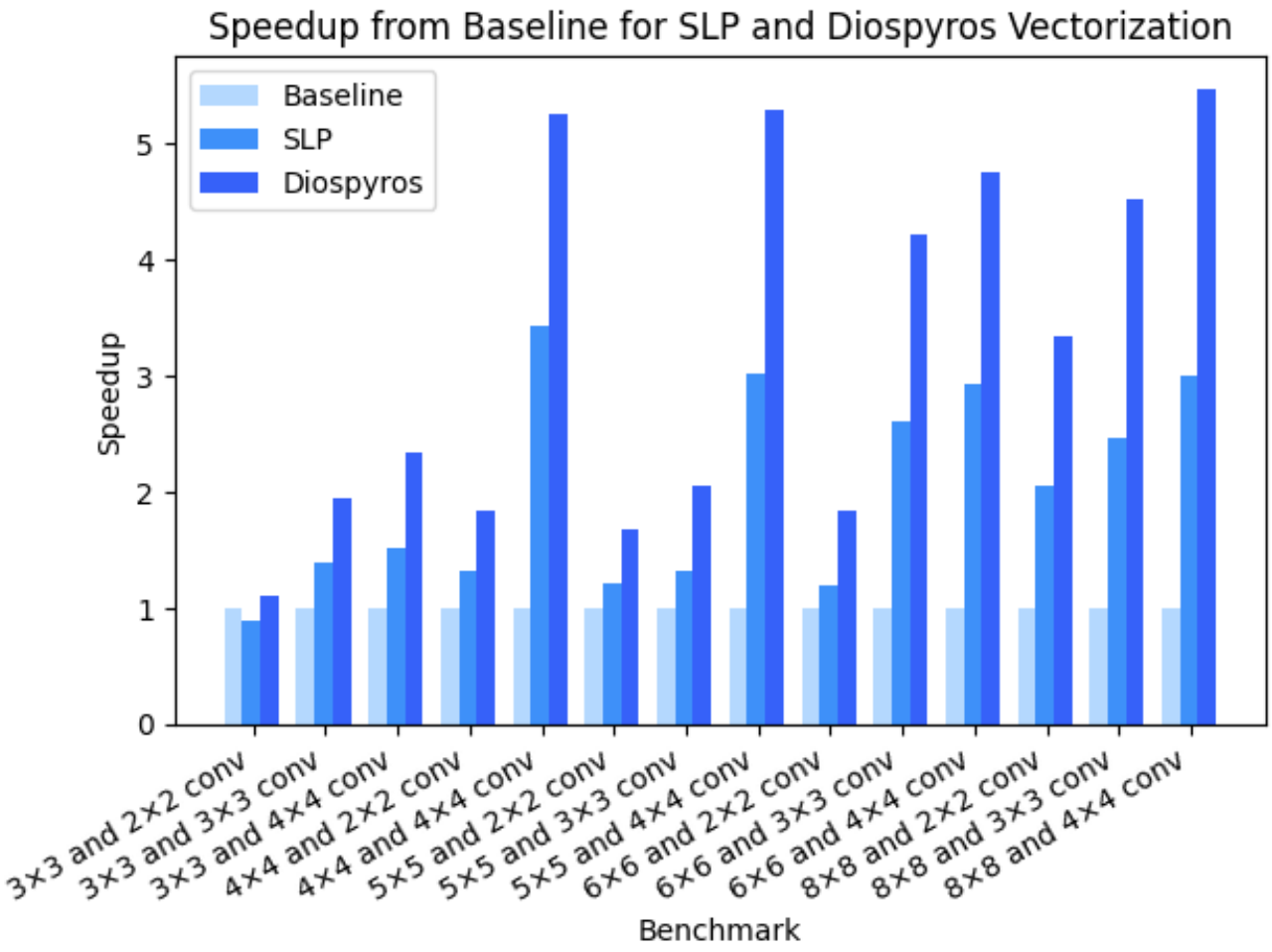

Figure 4.1: Convolution Speedup over Baseline for SLP and Diospyos

The Diospyros pass performs the best on the convolution benchmarks. Figure 4.1 shows how the Diospyros has a larger speedup than the baseline and SLP on all benchmarks. As the matrices in the convolution become larger, the

speedup increases, as demonstrated by the larger speed ups for the convolution of 8 by 8 matrices and 4 by 4 matrices. On the average, speed ups for Diospyros compared to the baseline was 4.26 times faster, and 1.67 times faster compared to SLP vectorization. One reason that Diospyros performs better than SLP for the 8 by 8 with 4 by 4 matrix convolution benchmark is that Diospyros is able to vectorize the code successfully and finds 32 instances of aligned and consecutive load instructions in the vectorization, while SLP does not find profitable vectorization opportunities.

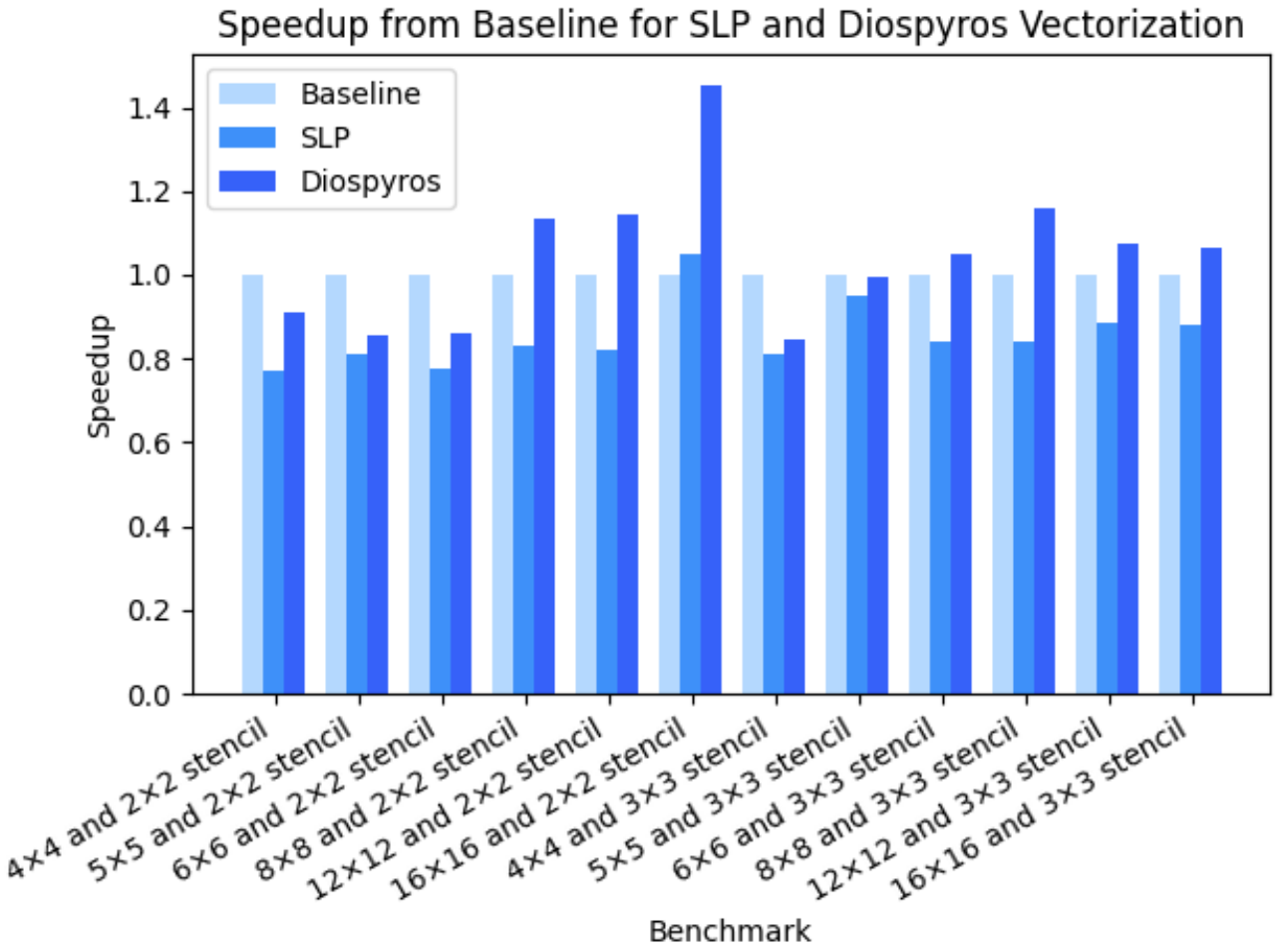

Figure 4.2: Stencil Speedup over Baseline for SLP and Diospyos

The Diospyros pass also performs well on stencil benchmarks. In figure

4.2, for smaller benchmark sizes, Diospyros performs worse than the baseline. However, for larger stencil neighborhoods and matrices, Diospyros outperforms the baseline. On all stencil benchmarks, Diospyros vectorization outperforms SLP vectorization. Overall, on stencil benchmarks, Diospyros has an average 1.13 speedup over the baseline and an average 1.26 speedup over SLP vectorization. Another aspect on which Diospyros performs well for stencil benchmarks is during compile time. Many of the chunks being vectorized by Diospyros for stencil benchmarks are identical to each other, allowing the cache-reuse feature to be used repeatedly to speed up compile time performance.

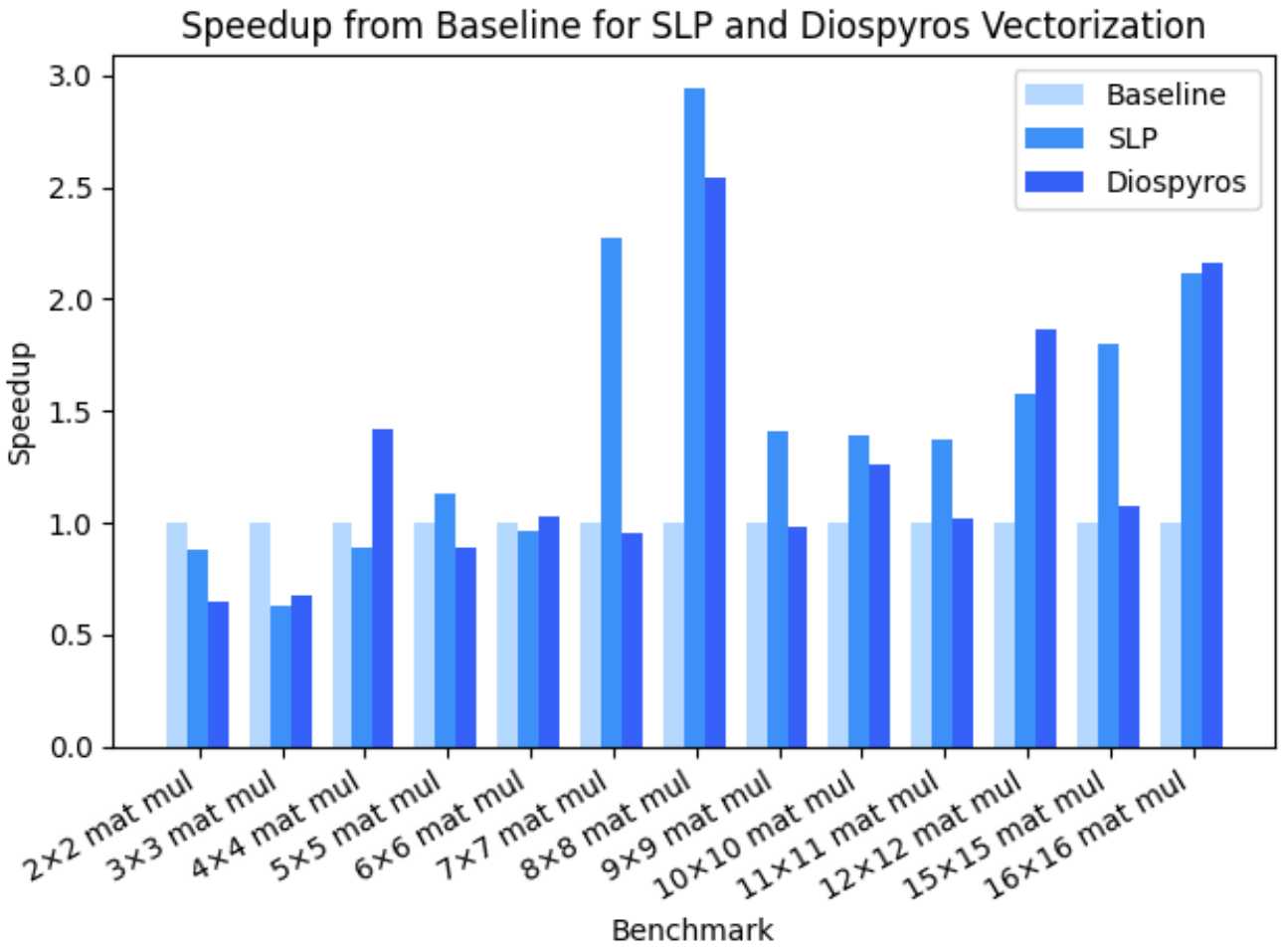

Figure 4.3: Matrix Multiplication Speedup over Baseline for SLP and Diospyos

On matrix multiplication benchmarks, figure 4.3 reveals Diospyros has similar behavior to the stencil benchmarks. For smaller sized matrices, Diospyros has a performance slow down compared to the baseline. However, on larger matrix dimensions, Diospyros starts to match the performance of the baseline, particularly on matrices with dimensions that are even numbers. On most of the benchmarks, however, Diospyros performs worse than SLP vectorization; in particular, Diospyros has a 0.80 slow down compared to SLP vectorization, but has an overall speed up of 1.39 over the baseline.

An interesting trend is that when the dimension of the square matrix is a multiple of the vector width, Diospyros out performs the baseline by quite a large factor, and often outperforms SLP vectorization as well. One reason might be that for these specific benchmarks, many aligned-consecutive load instructions are generated. For example, 432 aligned-consecutive loads are generated for the 12 by 12 benchmark, allowing for much faster runtime performance compared to using gather instructions. Also, the cache-reuse feature also speeds up compile time performance noticeably for matrix multiplication.

Diospyros performs poorly for the qr decomposition benchmarks as figure 4.4 shows. Diospyros has a slow down of 0.85 relative to the baseline and 0.99 relative to SLP. Diospyros can only find small chunks to vectorize, because loop unrolling still produces multiple basic blocks in the body of the qr decomposition function. In addition, many shuffle instructions are emitted in the LLVM IR, to create a vector splat, which may limit the profitability of any vectorization from Diospyros.

For the quaternion product benchmark shown in figure 4.5, Diospyros performs worse than the baseline. Diospyros finds limited chunks that can be vec-

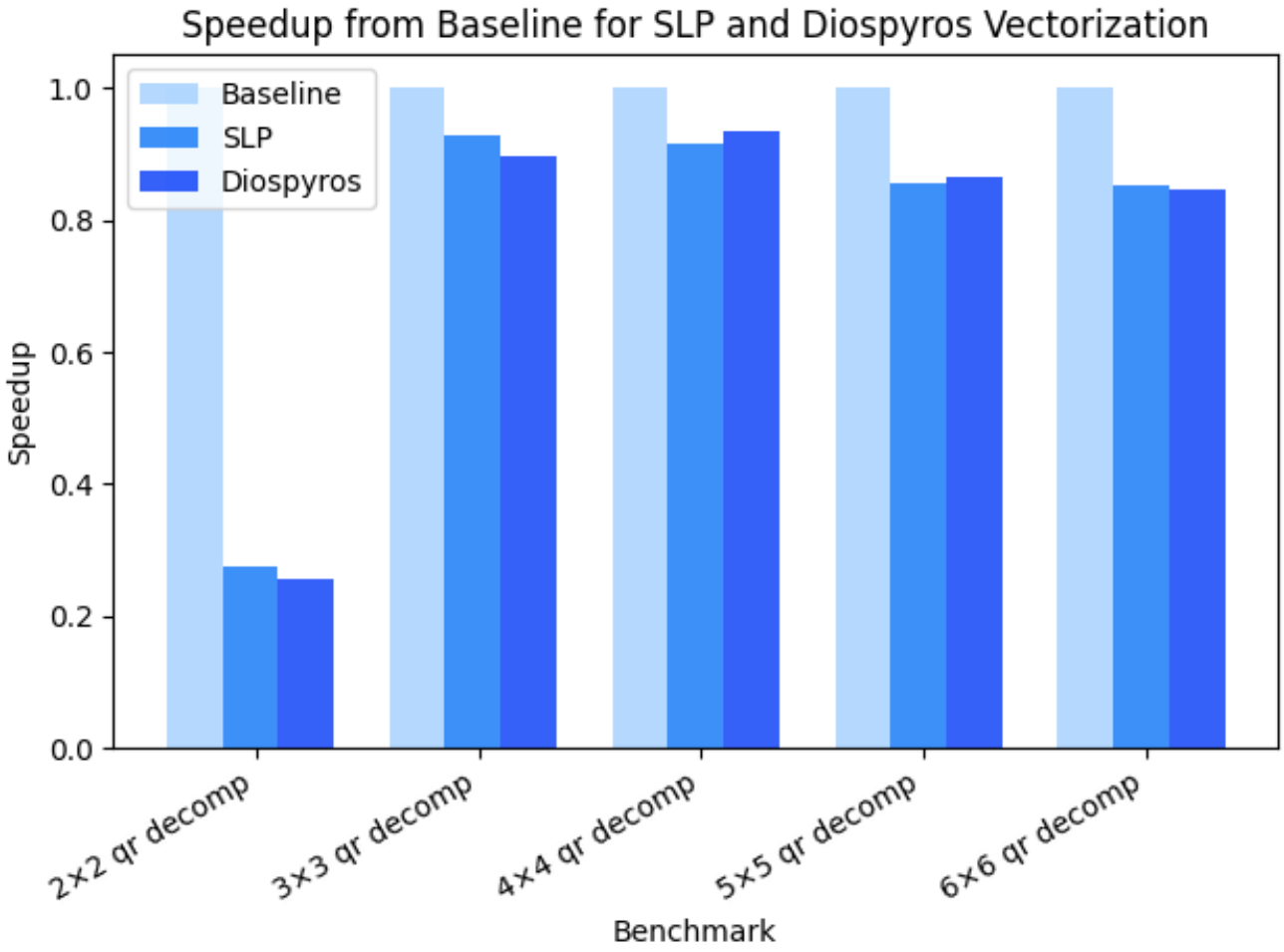

Figure 4.4: QR Decomposition Speedup over Baseline for SLP and Diospyos

torized, and has to use gather operations. Diospyros has a slow down of 0.93 compared to the baseline.

These benchmarks show that Diospyros can find speedups in certain cases, particularly for larger matrix sizes, especially for stencil, convolution and matrix-multiplication kernels. However, for other benchmarks, Diospyros's choice of vectorization causes the code to be slower than the baseline. More work to improve Diospyros, such as considering different load instruction optimizations, adding new rewrite rules, and modifying the cost model during program extraction, may lead to better results.

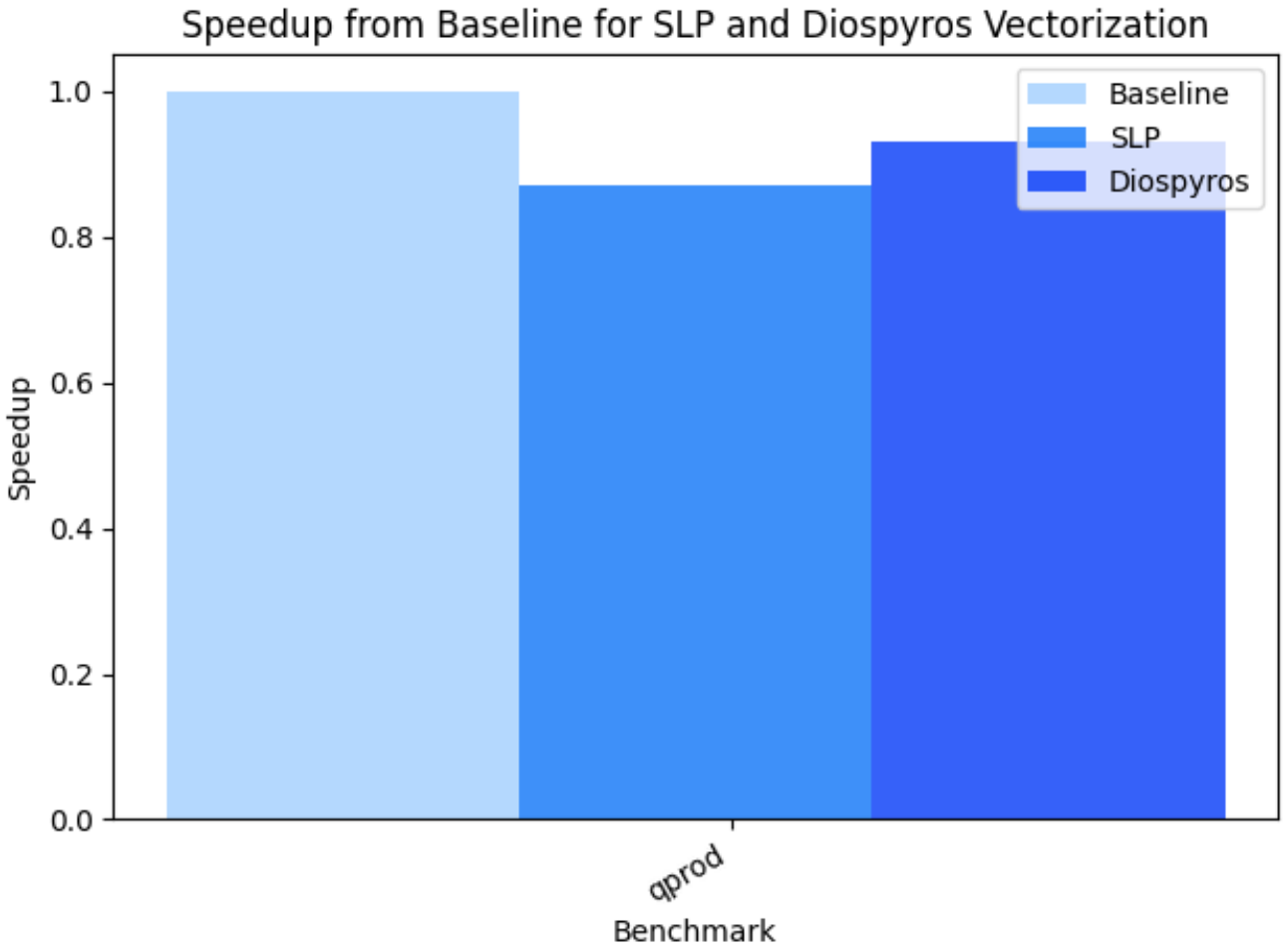

Figure 4.5: Quaternion Product Speedup over Baseline for SLP and Diospyos

4.3 Future Work

Based on the results, the Diospyros pass could be improved in various ways, including the approach for vectorization, the use of gather instructions, and the choice of rewrite rules and calculation of costs in E-classes.

The approach for vectorization in this thesis is to find store trees to consecutive vector addresses, so that a consecutive and aligned store vector instruction can be emitted. However, this leads to slow downs in code after vectorization,

such as in the matrix multiplication benchmarks. An alternate approach would be to consider vectorization beginning with seeds of consecutive load operations, as defined by the SLP vectorization algorithm [5]. Combining this load address heuristic with equality saturation might yield better results, because there are potentially many more load operations, compared to store operations, as shown in the benchmarks. Emitting more consecutive and aligned load instructions could yield faster vector programs.

In addition, it would be interesting to consider memory address optimizations for gather instructions that are emitted by the compilation pass. On some of the benchmarks, many of the gather instructions display strong spatial locality or repetition of addresses. Addresses are often fetched in near-consecutive groupings. For instance, 5-by-5 matrix multiplication features gather instructions to addresses in four address subsets for every five consecutive addresses, as well as repeated fetches of the same addresses, incrementing in a strided fashion. A simple procedure combining consecutive and aligned loads with shuffle was used to optimize the gather instructions, but led to almost two times worse code for 5-by-5 matrix multiplication. More sophisticated optimization procedures or use of more specialized instructions could yield better results.

Adding in different rewrite rules could also lead to better vectorization. One type of rewrite rule that could be added includes permutation of vector instructions. This rule might lead beneficial data movement, allowing for consecutive and aligned loads to be generated. However, care would have to be taken to make sure that the permutations are limited in application, because having general permutation rules would rapidly increase the search space beyond a tractable size. Another type of rewrite rule to consider would be a rule to split

vectors into smaller shorter vectors. This rule could also lead to profitable vector programs being found.

Finally, improving the cost model might lead to better results. The cost model could be tuned specifically for a particular architecture. In addition, a more accurate extraction approach for the Egg rewriting engine, relying on an ILP solver, rather than a greedy approach, could lead to better results as well. However, using the ILP solver could also lead to longer extraction times.

CHAPTER 5

RELATED WORK

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

5.1 Auto-Vectorization

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 Allen and Kennedy [2]. Loop vectorization was further developed to target SIMD processors, and this work includes alignment and consecutiveness constraints for loop vectorization [14], as well as approaches to outer loop vectorization [15].

A different approach to auto-vectorization takes advantage of commonly repeated isomorphic and independent operations within a basic block, called Superword Level Parallelism [5]. The original version of SLP uses a greedy algorithm to pair instructions together as vectorizable, beginning at seed instructions, such as consecutive load instructions [5]. SLP vectorization has proven to be successful; LLVM and GCC have implementations of SLP vectorization. Additionally, the original SLP greedy algorithm has also been used in other contexts, such as taking older vector code and mapping the code to use modern

vector instructions [10].

Extensions to the original SLP algorithm take advantage of structure in basic blocks, such as the existence of commutative operations [18], smaller-width vectors [20], padding with extra instructions [17], and looking ahead to guide the vectorization process [19]. Other greedy heuristics were developed to overcome the original algorithm, and consider the total number of reuses a vector pack will have [7], 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 [3]. These auto-vectorization approaches find profitable 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, as Mendis et al. notes in the goSLP paper [9].

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 [9]. goSLP applies an 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, because optimality is only guaranteed when considering vector packs of pairs.

A contrasting approach to using a solver is using machine learning. Mendis et al. explored an alternative, using imitation learning methods to train an algorithm to decide whether instructions should be paired in vector packs [11].

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 vectorization solutions [24]. 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. A work in progress paper for Diospyros explains how synthesis can be used to find better vector programs, but this approach was replaced by equality saturation because synthesis did not scale up well for larger kernels [23]. Another work using synthesis focuses on finding performance efficient 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 [1]. This approach is able to find efficient sequences of vector assembly instructions for relatively large programs.

5.2 Equality Saturation

Equality saturation is a common technique for solving problems in compilers research by allowing for searching a large space of programs. Early techniques in optimization included brute-force enumeration of many possible programs, such as in Superoptimizer [8]. Later work revealed using rewrite rules to search for programs equivalent to an original program scaled better to larger program

inputs. The Denali project used equality saturation to find near-minimal programs in terms of number of machine cycles required to execute the program [4]. To extract the final program from the E-graph, Denali paired equality saturation with a SAT solver, querying to solver to see if a boolean expression representing a upper bound of the number of cycles needed was satisfiable.

Equality saturation was also applied intermediate representation code as a means of implementing compiler optimization by Tate et al. [21]. In a domain specific setting, Diospyros applied equality saturation as a central technique to find better vectorized programs for digital signal processors [24].

Equality saturation has also been applied to other domains beyond program optimization. Equality saturation was used to find floating point programs with fewer bits of error in the Herbie Project [16]. Another area equality saturation was used was to find high-level human readable and editable CAD code from mesh compilers in the Szalinski project [12]. Equality saturation was also used to create "compilers" that generate instructions on how a carpenter can cut wood [26]. Finally, Ruler used equality saturation to derive minimal sets of re-write rules [13]. Many of these projects utilize the Egg package, developed to make it easier to use equality saturation in various settings [25].

CHAPTER 6

CONCLUSION

This thesis extends Diospyros to the LLVM compiler architecture. Several benchmarks are compiled via the LLVM Diospyros pass to target a Mac M1 chip. Results show that the Diospyros can produce code with better runtime performance than SLP vectorization or baseline compilation for stencil and convolution benchmarks, but fails to produce better code than the baseline for other benchmarks. New rewrite rules and better consideration of address alignment and consecutiveness are proposed as means for future improvement.

APPENDIX A

EGG GRAMMAR

⟨*number*⟩ ::= {32 bit floating point numbers}

⟨*address*⟩ ::= {LLVM Values}

⟨*register*⟩ ::= {LLVM Values}

⟨*alignment*⟩ ::= $\mathbb{N} \cup \{-1\}$, where -1 is for unknown alignment

⟨*offset*⟩ ::= $\mathbb{N} \cup \{-1\}$, where -1 is for unknown offset

⟨*scalar-op*⟩ ::= Number(⟨*number*⟩)

| Register(⟨*register*⟩)

| Load(⟨*address*⟩, ⟨*alignment*⟩, ⟨*offset*⟩)

| Store(⟨*scalar-op*⟩, ⟨*address*⟩)

| Neg(⟨*scalar-op*⟩)

| Add(⟨*scalar-op*⟩, ⟨*scalar-op*⟩)

| Sub(⟨*scalar-op*⟩, ⟨*scalar-op*⟩)

| Mul(⟨*scalar-op*⟩, ⟨*scalar-op*⟩)

| Div(⟨*scalar-op*⟩, ⟨*scalar-op*⟩)

⟨*address-vec*⟩ ::= AddressVector(⟨*address*⟩, ⟨*address*⟩, ⟨*address*⟩, ⟨*address*⟩)

⟨*alignment-vec*⟩ ::= AlignmentVector(⟨*aligment*⟩, ⟨*aligment*⟩, ⟨*aligment*⟩, ⟨*aligment*⟩)

⟨*offset-vec*⟩ ::= OffsetVector(⟨*offset*⟩, ⟨*offset*⟩, ⟨*offset*⟩, ⟨*offset*⟩)

⟨*vector-op*⟩ ::= ConstantVector(⟨*number*⟩, ⟨*number*⟩, ⟨*number*⟩, ⟨*number*⟩)

| VectorLoad(⟨*vector-op*⟩, ⟨*address-vec*⟩, ⟨*alignment-vec*⟩, ⟨*offset-vec*⟩)

| AlignedConsecutiveVectorLoad(⟨*vector-op*⟩, ⟨*address*⟩)

| VectorNeg(⟨*vector-op*⟩)

| VectorAdd(⟨*vector-op*⟩, ⟨*vector-op*⟩)

| VectorSub(⟨*vector-op*⟩, ⟨*vector-op*⟩)

| VectorMul(⟨*vector-op*⟩, ⟨*vector-op*⟩)

| VectorDiv(⟨*vector-op*⟩, ⟨*vector-op*⟩)

⟨*kernel*⟩ ::= Vector(⟨*scalar-op*⟩, ⟨*scalar-op*⟩, ⟨*scalar-op*⟩, ⟨*scalar-op*⟩)

| VectorStore(⟨*vector-op*⟩, ⟨*address*⟩), for aligned and consecutive addresses

APPENDIX B

EGG REWRITE RULES

B.1 Base Scalar Rules

- Add(a, 0) $\rightsquigarrow$ a and Add(0, a) $\rightsquigarrow$ a
- Mul(a, 0) $\rightsquigarrow$ 0 and Mul(0, a) $\rightsquigarrow$ 0
- Mul(a, 1) $\rightsquigarrow$ a and Mul(1, a) $\rightsquigarrow$ a
- Neg(Neg(a)) $\leftrightsquigarrow$ a
- Neg(0) $\leftrightsquigarrow$ 0
- Neg(a) $\leftrightsquigarrow$ Sub(0, a)

B.2 Associative and Commutative Rules

- Add(a, b) $\rightsquigarrow$ Add(b, a)
- Mul(a, b) $\rightsquigarrow$ Mul(b, a)

B.3 Vector Rules

- Custom matching for binary operations.
  - Vector(Add(a, b), Add(c, d), Add(e, f), Add(g, h)) $\rightsquigarrow$ VectorAdd(Vector(a, c, e, g) Vector(b, d, f, h))
  - Vector(0, Add(c, d), Add(e, f), Add(g, h)) $\rightsquigarrow$ VectorAdd(Vector(0, c, e, g), Vector(0, d, f, h)), where any of the lanes can be a 0.

- Vector(Neg(a), Neg(b), Neg(c), Neg(d)) $\rightsquigarrow$ VectorNeg(a, b, c, d)
- Vector(Load(a, b, c), Load(d, e, f), Load(g, h, i), Load(j, k, l) $\rightsquigarrow$ AlignedConsectiveVectorLoad(a), if a is aligned, and a, d, g and j are all consecutive addresses.
- Vector(Load(a, b, c), Load(d, e, f), Load(g, h, i), Load(j, k, l) $\rightsquigarrow$ VectorLoad(AddressVector(a, d, g, j), AlignmentVector(b, e, h, k), OffsetVector(c, f, i, l))
- Vector(Store(a, b), Store(a, b), Store(a, b), Store(a, b)) $\rightsquigarrow$ VectorStore(Vector(a, c, e, g), b)

BIBLIOGRAPHY

[1] Maaz Bin Safeer Ahmad, Alexander J. Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. Vector instruction selection for digital signal processors using program synthesis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 1004–1016, New York, NY, USA, 2022. Association for Computing Machinery.

[2] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, oct 1987.

[3] Joonmoo Huh and James Tuck. Improving the effectiveness of searching for isomorphic chains in superword level parallelism. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 718–729, New York, NY, USA, 2017. Association for Computing Machinery.

[4] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, page 304–314, New York, NY, USA, 2002. Association for Computing Machinery.

[5] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 145–156, New York, NY, USA, 2000. Association for Computing Machinery.

[6] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis amp; transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[7] Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, and Mahmut Kandemir. A compiler framework for extracting superword level parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 347–358, New York, NY, USA, 2012. Association for Computing Machinery.

[8] Henry Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Pro-*

gramming Languages and Operating Systems, ASPLOS II, page 122–126, New York, NY, USA, 1987. Association for Computing Machinery.

[9] Charith Mendis and Saman Amarasinghe. Goslp: Globally optimized superword level parallelism framework. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[10] Charith Mendis, Ajay Jain, Paras Jain, and Saman Amarasinghe. Revec: Program rejuvenation through revectorization. In *Proceedings of the 28th International Conference on Compiler Construction*, CC 2019, page 29–41, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Charith Mendis, Cambridge Yang, Yewen Pu, Saman Amarasinghe, and Michael Carbin. *Compiler Auto-Vectorization with Imitation Learning*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[12] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured cad models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 31–44, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. Rewrite rule inference using equality saturation. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[14] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for simd. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, page 132–143, New York, NY, USA, 2006. Association for Computing Machinery.

[15] Dorit Nuzman and Ayal Zaks. Outer-loop vectorization: Revisited for short simd architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 2–11, New York, NY, USA, 2008. Association for Computing Machinery.

[16] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 1–11, New York, NY, USA, 2015. Association for Computing Machinery.

[17] Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. Pslp: Padded slp automatic vectorization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, page 190–201, USA, 2015. IEEE Computer Society.

[18] Vasileios Porpodas, Rodrigo C. O. Rocha, Evgueni Brevnov, Luís F. W. Góes, and Timothy Mattson. Super-node slp: Optimized vectorization for code sequences containing operators and their inverse elements. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 206–216. IEEE Press, 2019.

[19] Vasileios Porpodas, Rodrigo C. O. Rocha, and Luís F. W. Góes. Look-ahead slp: Auto-vectorization in the presence of commutative operations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 163–174, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Vasileios Porpodas, Rodrigo C. O. Rocha, and Luís F. W. Góes. Vw-slp: Auto-vectorization with adaptive vector width. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, New York, NY, USA, 2018. Association for Computing Machinery.

[21] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, page 264–276, New York, NY, USA, 2009. Association for Computing Machinery.

[22] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 135–152, New York, NY, USA, 2013. Association for Computing Machinery.

[23] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. A synthesis-aided compiler for dsp architectures (wip paper). In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '20, page 131–135, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. Vectorization for digital signal processors via equality

saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 874–886, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021.

[26] Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. Carpentry compiler. *ACM Trans. Graph.*, 38(6), nov 2019.