Basic-block vectorization for graphics compilers

by

Robert Foss

Department of Computer Science
Faculty of Engineering at Lund University

June 2013

Master's thesis work carried out at *ARM*.

Supervisors	Jonas Skeppstedt	jonas.skeppstedt@cs.lth.se
	Markus Lavin	markus.lavin@arm.com
Examiner	Jonas Skeppstedt	jonas.skeppstedt@cs.lth.se

Abstract

Increasingly complex graphics shaders and new use cases like OpenCL provide increased opportunities for vectorization, due the the larger codebases they provide. Unlike general-purpose microprocessors, graphics microprocessors can feasibly be equipped with just vector registers. By already having data in vector registers, some of the cost of vectorization can be avoided, which leads to vectorization being simpler on graphics microprocessors than on most general-purpose microprocessors.

Graphics compilers mostly do compilation during the run-time of applications, which makes compilation time a serious aspect of any GPU compiler transformation.

In this thesis two basic-block vectorizers suitable for graphics compilers and microprocessors are presented and evaluated.

Keywords: gpu, compiler, basic-block, vectorization

Acknowledgments

I would like to thank my on-site supervisor Markus Lavin for tracking down compiler issues and helping me navigate the ins and outs a new compiler. Dmitri Ivanov for proofreading this paper, performance diagnostics, and being a great source of knowledge. Hal Finkel for the optimization pass BBVectorize in LLVM, related talks, references and email correspondence. Johan Grönqvist for helping out with debugging tools, ideas and providing interesting reading material. Mikael Nilsson, who was doing his master thesis on a different optimization pass for the same compiler, for being a great sounding board. Krister Walfridsson for proofreading this paper and interesting discussions. Jonas Skeppstedt for proofreading this paper and planting the seed of an interest in compilers. ARM Holdings, Olof Dellien and Philippe Coucaud for hosting this work in the graphics compiler team at ARM Lund.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Much research has been dedicated to vectorization since the introduction of the earliest vector supercomputer Solomon, which could handle up to 1024 operands[1]. Current general-purpose microprocessors often support multimedia extensions like AltiVec[2] for the PowerPC instruction set, AVX[3] for the x86 instruction set and NEON[4] for the ARMv7 and ARMv8 instruction sets. Common general-purpose microprocessor workloads are rarely very vectorizable. Because of this and the diverse set of multimedia extensions, code written for general purpose microprocessors is often disregarding performance gains that could be achieved with vectorization.

Like general-purpose microprocessors, modern personal computer graphics processing units (henceforth GPUs) have support for executing general purpose code on vector hardware. The focus however is on computational throughput which makes GPUs ideal for batch calculations of many kinds. General-purpose GPU (henceforth GPGPU) support has evolved in four generations of GPU hardware. The first generation of GPUs capable of 3D graphics, like the 3dfx Voodoo Graphics, are based on a simple fixed function pipeline. Geometry is calculated by the central processing unit (henceforth CPU) and pixels are calculated by the GPU. The first generation GPU pipeline basically consists of three stages; rasterization, texturing and blending. Second generation GPUs, like the ATI Radeon 7500[5] and the nVidia GeForce 256[6], introduced hardware support for; geometry processing, transform & lightning and clipping. The increased hardware support led to a large increase in performance and the first thoughts of GPGPUs. GPGPUs were still infeasible since the hardware was just configurable and not programmable. Third generation GPUs, like the ATI Radeon 9700[7] and the nVidia GeForce FX[7], introduced a separate vertex shader and fragment shader both of which were programmable. The hardware implementation of the vertex and fragment shaders were different since they had

different requirements. This hardware is the first hardware generation for which GPGPU is possible. However the floating point implementations were not standards compliant, which caused practical issues. Fourth generation GPUs, like the ATI Radeon X1800[7] and the nVidia GeForce 8800[8], introduced unified shaders. A unified shader unit handles the calculations of both the vertex and the fragment shader. Additionally support for longer programs, branching and floats based on the full IEEE-754 floating point specification were introduced. Unified shaders are well suited for GPGPU calculations and are what is used by OpenCL and CUDA today.

Modern GPUs do not include multimedia extensions and are generally optimized for vector inputs. This design is a result of graphics shaders often computing scalars or small vectors both of which are ideal candidates for simultaneous computation on a wide vector unit. Graphics shaders are also becoming increasingly complex and the use cases for GPUs are becoming more diverse with additions like OpenCL. Both leading to larger code-bases and thereby larger opportunities for optimizations.

Computation of scalars and small vectors can be done more efficiently if vector-registers and vector-units are fully utilized. For graphics microprocessors the hardware is rarely exposed to the programmer, which places the responsibility of vectorization largely on the graphics compiler. If vectorization is to be used and effectively utilize the hardware the compiler has to have full knowledge of the hardware architecture and be able to maximize the utilization of all resources.

However there are many costs associated with simply combining two operations into a vector operation (henceforth fusing) and they can be divided into compile-time costs and run-time costs.

Compile-time costs arise from deciding which vectorization alternatives are legal and profitable, where the legality of an operation is determined by if it is hardware-wise possible and logically equivalent to the unvectorized code. Cyclical data dependences have to be found and avoided. Legal vectorization options have to be found, sorted and determined to be profitable. Vectorization options are rarely independent and selecting one option might prevent every other option from being legal or profitable. Knowing that one option might exclude other options and that run-time of the compilation is important, we are forced to resort to heuristics to determine which options are preferable.

Run-time costs arise when data has to be moved into a specific register, re-ordered inside a vector register or the results from an operation has to be moved into another register. Move operations are not always required and should be avoided where possible. Register pressure may also be increased, which can have a severe performance impact.

Common vectorization techniques fall into one of two categories, loop transformations or basic-block vectorization.

Loop transformations are based on analyzing loop boundaries and data dependences. Algorithms like Fourier-Motzkin elimination[9] can be used to determine which loop-iterations are independent and therefore vectorizable. The GNU Compiler Collection uses data dependence analysis in the form of a Static Single Assignment [10][11] graph in combination with a Strongly Connected Components search using a linear algorithm like that presented by Tarjan[12]. Loop iterations that are found to have no data dependence can be vectorized by simply doing multiple iterations of the loop by replacing operations with their vector equivalent. If the original loop runs a number of iterations which isn't evenly divisible by the vectorization factor, a cleanup loop is needed. The cleanup loop runs the remaining loop iterations, which haven't been computed by the vectorized loop.

Basic-block vectorization is done by determining which operations are independent and can be combined into a vector operation. The Vienna MAP vectorizer [13] introduced the idea of finding pairs of operations that can be built into trees of vector operations, to avoid having to move data around. The Vienna MAP vectorizer is the basis of basic-block vectorizer of LLVM called BBVectorize[14], which was introduced in LLVM version 3.1 by Hal Finkel. BBVectorize does basic-block vectorization by doing dependence analysis, pairing independent operations and chaining the pairs into trees. If the trees are long enough, the best trees of pairs are converted into vector operations [14]. Heuristics are used to determine what constitutes a good tree. Some heuristics used in the LLVM basic-block vectorizer are not applicable or needed for graphics microprocessors, since the hardware architecture is based on vector-registers only.

This paper takes the basic ideas of the LLVM basic-block vectorizer and adapts them to suit vector register based microprocessors and provide compilation speeds for shaders that are fast enough not to be an issue or bottleneck. A second simpler basic-block vectorization algorithm is introduced and compared to the LLVM based basic-block vectorizer.

Chapter 2

Approach

2.1 Data dependence

If an operation writes to a variable that is read or written by another operation, changing their execution order can change the results. Such pairs of operations are called data dependent operations. To create a legal vectorization, the operations that will be fused into a vector operation can have no dependence on each other. If a chain of operation pairs is to be fused, there can be no dependence between the pairs other than a pair depending on the previous pair in the chain. Both of these rules are used to prevent cyclical dependences in which input requirements of one or more operations are unsatisfiable. There are three kinds of data dependence[9][15].

True dependence, flow dependence or read after write:
When a write to a location is followed by a read from the same location, the read depends on the value written by the write.

Antidependence or write after read:
When a read from a location is followed by a write to the same location, the write does not depend on the read as long as the internal ordering is intact. But if the operations were to be internally reordered a true dependence would arise.

Output dependence or write after write:
When a write to a location is followed by another write to the same location, the ordering of the operations defines the results.

True dependences and output dependences must be avoided to produce legal vectorizations. Antidependence can be worked around by storing the

values in different places, however that is beyond the scope of this optimization. Dependence analysis can be very expensive if implemented in a naive way. The cost of dependence analysis is tied to the basic-block size and the amount of operations of a single type that exist in that basic-block. Therefore unneeded pairs are pruned as early as possible as not to incur unnecessary dependence checks.

2.2 LLVM based basic-block vectorization

This implementation is based on the ideas of the Vienna MAP vectorizer[13] and the LLVM basic-block vectorizer[14]. The overarching idea is to find fusible pairs of operations which can be profitably fused. To achieve this, trees of operations are built and heuristically determined to be either profitable to vectorize or not.

The algorithm is designed to be called in a fixed-point iteration manner, so that already fused operations can be fused again until the maximum vector length that the hardware supports has been reached.

Dictionaries or key-value maps will be described as dict{key} in the following algorithms.

Algorithm 1 Sort operations by type.

```
function SORT(basicBlock)
    opSets ← ∅
    for all op ∈ basicBlock do
        if ISFUSIBLE(OPTYPE(op)) then
            opSets{OPTYPE(op)} ← opSets{OPTYPE(op)} + op
    return opSets
```

The purpose of Algorithm 1 is to sort operations into sets where every operation in a set is possible to fuse with another from the same set. **IsFusible** is a filtering function which prevents operations which cannot be fused from being added to *opSets*. Operations that cannot be fused can be filtered out by determining two criteria; vector length and operation type. An operation of vector length 5 cannot be successfully vectorized on a microprocessor which has support for vector a length of at most 4, and is ignored. Certain operation types, like jumps, aren't meaningful to vectorize and can also be ignored. **OpType** is simply a function that returns the operation type of an operation. The result of **OpType** is used as an index or key for retrieving and storing all operations of a type in the same set.

Algorithm 2 Pair fusible operations.

```
function BuildPairs(opTypes, opSets)
    opPairs ← ∅
    for all opSet ∈ opSets do
        for i ← 0; i < Length(opSet); i ← i + 1 do
            for k ← i + 1; k < Length(opSet); k ← k + 1 do
                if AreFusible(opSet[i], opSet[k]) then
                    opPairs ← opPairs + Pair(opSet[i], opSet[k])
    return opPairs
```

Algorithm 2 considers every possible combination of two operations in a set for each *opSet* in *opSets*. **Length** is used to determine the number of elements in the set *opSet*. Combinations of operations are filtered through the function **AreFusible** to remove variants that are not fusible on the grounds of the following criteria; combined vector length, subtypes and dependences between the two operations.

If the combined vector length of the two operations is larger than what the microprocessor supports, ignore that operation combination.

If the combination contains operations of differing subtypes, like compare equals and compare less than, ignore that combination.

Finally, if there is a data dependence between the two operations, the operations can't safely be paired and have to be ignored. Internally **AreFusible** implements its dependence check using the function **HasDependence**, which will be discussed in Algorithm 7.

Operation combinations are paired using **Pair** and then added to the set of all pairs called *opPairs*.

Algorithm 3 Find directly dependent pairs.

```
function FindPairUses(opPairs)
    pairUses ← ∅
    for all pair ∈ opPairs do
        pairsUsingOp1 ← DirectDependences(Op1(pair))
        pairsUsingOp2 ← DirectDependences(Op2(pair))
        pairUses{pair} ← pairsUsingOp1 ∩ pairsUsingOp2
    return pairUses
```

Algorithm 3 finds all pairs which use the outputs of both operations in *pair* as input and adds them to a set in *pairUses*. **Op1** and **Op2** simply fetch the first and second operation from a pair. **DirectDependences** fetches all pairs which directly depend on the output of an operation. The intersection

of *pairsUsingOp1* and *pairsUsingOp2* will consist only of pairs that directly depend on both outputs of *pair*.

Algorithm 4 Build unpruned trees.

```
function BuildUnprunedTrees(opPairs, pairUses)
    pairTrees ← ∅
    for all pair ∈ opPairs do
        treeRoot ← Tree(pair)
        pairTrees{pair} ← treeRoot
        Recurse(treeRoot, pair, pairUses)
    return pairTrees

function Recurse(treeParent, pair, pairUses)
    for all childPair ∈ pairUses{pair} do
        treeNode ← Tree(childPair)
        AddTreeBranch(treeParent, treeNode)
        Recurse(treeNode, childPair, pairUses)
```

Algorithm 4 uses the *pairUses* information to construct trees, where each child of a tree node directly depends on that tree node. Each pair is made into a tree, even if the height of that tree would be just one pair. The **Tree** function constructs a tree node out of a pair. **AddTreeBranch** is used to append a tree node as a child to another tree node.

Algorithm 5 Prune low-scoring trees.

```
function PruneLowScoringTrees(pairTrees, minScore)
    for all tree ∈ pairTrees do
        if TreeScore(tree) < minScore then
            pairTrees ← pairTrees \ tree
    return pairTrees
```

Algorithm 5 has the task of removing all trees with a score lower than *minScore* from *pairTrees*. *minScore* is an implementation defined value which provides a cut-off point beneath which no vectorization will take place. This pruning is needed to minimize the number of trees Algorithm 7 has to process. This is important because Algorithm 7 can be very expensive and depends on the size of the basic-block. The pruning is also used to remove trees from *pairTrees* to enable iteration over *pairTrees* until it does not contain any trees. **TreeScore** is the function containing the heuristics that determine

which trees are profitable to vectorize and which are not, and it will be further discussed in Section 2.4.

Algorithm 6 Prune duplicated operations.

```
function PruneDuplicatedOperations(pairTrees)
    for all treeRoot ∈ pairTrees do
        seenOperations ← ∅
        Recurse(treeRoot, seenOperations)
    return pairTrees

function Recurse(treeParent, seenOperations)
    for all treeChild ∈ TreeChildren(treeParent) do
        if Op1(treeChild) ⊂ seenOperations then
            RemoveTreeBranch(treeParent, treeChild)
            continue
        else if Op2(treeChild) ⊂ seenOperations then
            RemoveTreeBranch(treeParent, treeChild)
            continue
        else
            seenOperations ← seenOperations ∩ Op1(treeChild)
            seenOperations ← seenOperations ∩ Op2(treeChild)
            Recurse(treeChild, seenOperations)
```

The purpose of Algorithm 6 is to remove pairs which contain operations that are found elsewhere in the tree. **TreeChildren** returns a set of all child nodes to the tree node that is the argument. **RemoveTreeBranch** simply removes a tree node from the list of children of another tree node.

When two or more duplicated operations are found, every operation after the first one found will be pruned from the tree. This is a very simple heuristic and could be improved to make more intelligent decisions.

Algorithm 7 Prune intradependent pairs.

```
function PruneIntradependentPairs(pairTrees)
    for all treeRoot ∈ pairTrees do
        Recurse(treeRoot)
    return pairTrees

function Recurse(treeParent)
    for all treeChild ∈ TreeChildren(treeParent) do
        if HasDependence(Op1(treeChild), Op2(treeChild)) then
            RemoveTreeBranch(treeParent, treeChild)
            continue
        Recurse(treeChild, seenOperations)
```

Algorithm 7 is used to prune pairs which contain operations that have intra-dependences. This has already been done once when building the trees, but new dependences might have been introduced when fusing pairs in Algorithm 10. **HasDependence** simply checks whether there exists an intra-dependence between the operations in the pair of a tree node. If dependences are found, that branch is pruned from the tree.

Algorithm 8 Fetch the best branch of the best tree.

```
function GetBestBranch(pairTrees)
    bestTree ← ∅
    for all tree ∈ pairTrees do
        if TreeScore(tree) > TreeScore(bestTree) then
            highestTree ← tree
    branchPairList ← CreateList(Pair(TreeRoot(bestTree)))
    return Recurse(TreeRoot(bestTree), branchPairList)

function Recurse(treeParent, branchPairList)
    bestBranch ← ∅
    for all treeChild ∈ TreeChildren(treeParent) do
        if TreeScore(tree) > TreeScore(bestBranch) then
            bestBranch ← treeChild
    AppendToList(branchPairList, treeParent)
    if bestBranch = ∅ then                          ▷ This node is a leaf.
        return branchPairList
    return Recurse(bestBranch, branchPairList)
```

The purpose of Algorithm 8 is to find the best tree and extract the best branch from it, for this lookup **TreeScore** is used as the main heuristic.

Pair returns the pair of a tree node, **CreateList** creates a new list consisting of its argument and **AppendToList** simply appends a pair to a list. The returned list is ordered in the same order as the tree nodes were.

Algorithm 9 Prune fused operations.

```
function PruneFusedOperations(pairTrees, branchPairList)
    for all treeRoot ∈ pairTrees do
        Recurse(treeRoot, branchPairList)
    return pairTrees

function Recurse(treeParent, branchPairList)
    for all treeChild ∈ TreeChildren(treeParent) do
        if Op1(treeChild) ⊂ Ops(branchPairList) then
            RemoveTreeBranch(treeParent, treeChild)
            continue
        else if Op2(treeChild) ⊂ Ops(branchPairList) then
            RemoveTreeBranch(treeParent, treeChild)
            continue
        Recurse(treeChild, branchPairList)
```

Algorithm 9 prunes branches which contain pairs that use operations that have already been fused. **Ops** returns the set of operations contained within all pairs in *branchPairList*. The operations that are removed have also been removed from the program and replaced with their vector counterpart by the **FusePair** function in Algorithm 10. However the new vector operation can be fused later during the fixed-point iteration of this entire optimization.

Algorithm 10 Fuse operation pairs into vector operations.

```
function FuseBranch(branchPairList)
    for all pair ∈ branchPairList do
        FusePair(pair)
```

The purpose of Algorithm 10 is to fuse all operation pairs in the list into vector operations. The **FusePair** function is used to fuse the two operations of a pair. When the first pair is fused data usually has to be moved into the vector registers that are the arguments of the newly fused vector operation. Similarly the result of the last newly fused vector operation usually has to be moved into other registers.

Algorithm 11 Build and fuse trees.

```
function BuildAndFuseTrees(basicBlock, opTypes, minScore)
    opSets ← Sort(basicBlock)
    opPairs ← BuildPairs(opTypes, opSets)
    pairUses ← FindPairUses(opPairs)
    pairTrees ← BuildUnprunedTrees(opPairs, pairUses)
    pairTrees ← PruneLowScoringTrees(pairTrees, minScore)

    while Length(pairTrees) > 0 do
        bestBranch ← GetBestBranch(pairTrees)
        FuseBranch(bestBranch)
        pairTrees ← PruneFusedOperations(pairTrees, bestBranch)
        pairTrees ← PruneLowScoringTrees(pairTrees, minScore)
        pairTrees ← PruneIntradependentPairs(pairTrees)
        pairTrees ← PruneLowScoringTrees(pairTrees, minScore)
```

Algorithm 11 is the starting point of the LLVM based vectorization algorithm and combines previous algorithms into two parts. Part one builds and prunes *pairTrees*. Part two selects which pairs are to be fused, prunes *pairTrees* and is looped until there are no more trees in *pairTrees*. *PruneLowScoringTrees* is called twice inside the loop to prevent more expensive algorithms from being run on trees which won't be used anyway.

This algorithm is designed to be called in a fixed-point iteration manner for each *basicBlock*, so that already fused operations can be fused again until the maximum vector length that the hardware supports has been reached.

2.3 Pair based basic-block vectorization

The pair based vectorizer is based on evaluating the profitability of pairs of operations directly. This is a much simpler approach than the LLVM based basic-block vectorizer. Fewer data structures and fewer functions for building and maintaining these data structures are needed. However the trade-off is optimality. Since the LLVM based approach builds entire sets of pairs that can be fused and determines which of these sets are the most profitable to fuse, it can avoid fusing some pairs that later would have hindered some other profitable pair from being fused.

Algorithm 12 Put pairs into a tree sorted by profitability.

```
function PutIntoTree(opPairs)
    pairTree ← ∅
    for all pair ∈ opPairs do
        pairScore ← PairScore(pair)
        InsertIntoTree(pairTree, pairScore, pair)
    return pairTree
```

Algorithm 12 sorts the pairs in *opPairs* based on the score given by **PairScore**. Just like **TreeScore**, **PairScore** is a function which uses heuristics to return a *pairScore* which measures the profitability of fusing this pair. **InsertIntoTree** is a function that inserts *pair* into *pairTree*. If there already exists a pair with the same *pairScore* in *pairTree*, the new pair is inserted into a set of pairs sharing that specific *pairScore* in the *pairTree*.

Algorithm 13 Fuse the most profitable pairs.

```
function FusePairs(pairTree, minScore)
    while pairTree ! = ∅ do
        pair ← GetBestPair(pairTree)
        if HasDependence(Op1(pairTree), Op2(pairTree)) &&
        PairScore(pair) >= minScore then
            FusePair(pair)
```

Algorithm 13 fuses all pairs without internal dependences and a *pairScore* larger or equal to *minScore*. **GetBestPair** returns and removes the pair with the highest *pairScore* from *pairTree*.

Algorithm 14 Build and fuse pairs.

```
function FusePairs(basicBlock, minScore)
    opSets ← Sort(basicBlock)
    opPairs ← BuildPairs(opTypes, opSets)
    pairTree ← PutIntoTree(opPairs)
    FusePairs(pairTree, minScore)
```

Algorithm 14 combines previous algorithms into a function that builds and combines the most profitable operation pairs.

This algorithm is designed to be called in a fixed-point iteration manner for each *basicBlock*, so that already fused operations can be fused again until the maximum vector length that the hardware supports has been reached.

2.4 Heuristics

TreeScore and **PairScore** are the main heuristic functions of the respective optimization passes. The score is the metric of our heuristics and is used to determine profitability of vectorization options. The higher the score, the more profitable the vectorization option is. The way it scores trees can range from simplistic to very intricate and hardware dependent.

The heuristics employed by **TreeScore** and **PairScore** differ from each other. At a basic level they both evaluate the profitability of fusing a pair. But **TreeScore** has to take additional information about the current tree into account. Another difference is that many theoretically possible pairs have already been filtered out when building the trees. This prevents the heuristics from being very conservative, if any vectorization options are to be found.

The simplest way is just to determine the height of a tree and return that as the score. While this is a relevant metric it is not fine-grained nor does it take any hardware aspects into consideration. The idea is that the higher the tree the lower the amount of moves per fused operation we have to do, since data is already in place in vector registers.

Operations can also be scored differently depending on operation type. Some operations might be cheap in terms of hardware utilization, energy consumption or microprocessor cycles. The cumulative cost of an operation effects its score inversely.

Hardware restrictions for how data can be stored and moved can be taken into consideration. Alignment requirements for operations may force additional moves to be generated. If data is used by multiple operations, moves may be forced to be generated. Moves may differ in cost, where some are free and some have a relatively high cost, depending on how the hardware is constructed.

2.5 Examples

Figure 2.1 depicts a simple graph of operations and it will be used to illustrate the different routes of vectorization the two approaches take. For the sake of simplicity, operations that rearrange data have been omitted from all graphs. These graphs are not intended to be used to guage performance of code generated with either approaches to vectorization. The behaviors found in these examples are highly dependent on the implementation of the heuristic functions **TreeScore** and **PairScore**.

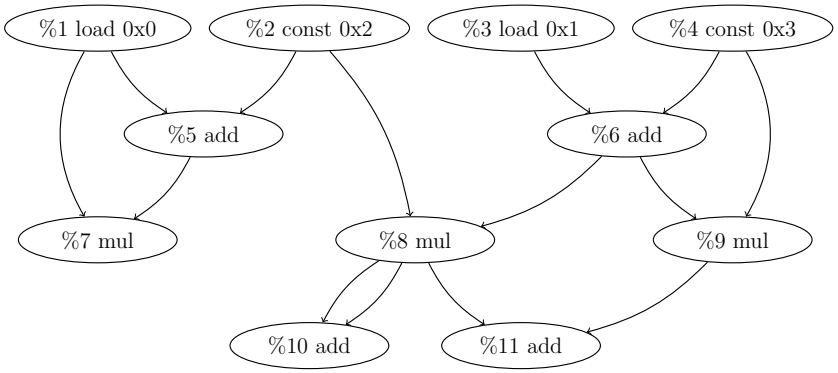

Figure 2.1: Graph of operations that have not been vectorized.

Figure 2.1 is used as a input to the two vectorization approaches and Figures 2.2, 2.3 and 2.4 are the resulting vectorizations.

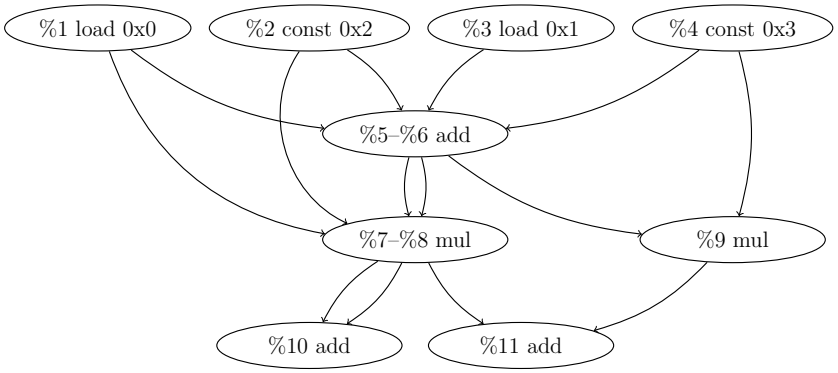

Figure 2.2: Graph of operations that have been vectorized with the LLVM based vectorizer.

As it can been seen from Figure 2.2, the LLVM based approach will find two pairs, %5–%6 and %7–%8. The output of the %5–%6 is used by both in operations in %7–%8. This is the kind of tree that is built and pruned by Algorithm 11. %10–%11 will not be a part of the tree since both operations do not depend on the output of %7–%8. Note that %10 and %11 could be fused if the heuristics allow trees of height 1 to be vectorized.

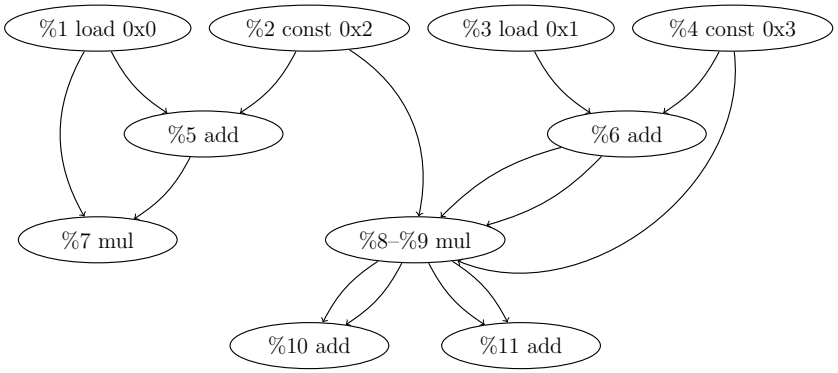

Figure 2.3: Graph of operations that have been vectorized one iteration with the pair based vectorizer.

The implementation of **PairScore** dictates that only pairs which use constants or use the result of the same operation will be fused. As it can been seen from Figure 2.3, the pair based approach will only find the pair %8–%9 in the first iteration.

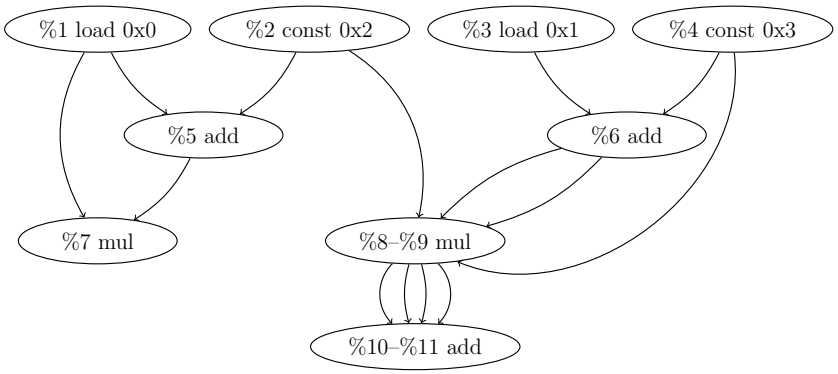

Figure 2.4: Graph of operations that have been vectorized two itertions with the pair based vectorizer.

In the second iteration of the pair based approach, seen in Figure 2.4, the pair %10–%11 is found. %10–%11 could not have been found in the previous iteration since %10 and %11 did not both use the result of the same operation. However when %8–%9 were fused, %10–%11 met all the requirements for vectorization.

Chapter 3

Evaluation

3.1 Benchmark Programs

Three benchmarks were chosen, each containing a number of OpenGL ES 2.0 vertex and fragment shaders. The benchmarks center around the usage scenarios of a smartphone; graphical user interfaces and games.

BenchA is a benchmark concerned mostly with gaming graphics. It consists of 37 fragment shaders and 38 vertex shaders.

BenchB is a benchmark concerned mostly with animating graphical user interface components. It consists of 60 fragment shaders and 30 vertex shaders.

BenchC is a benchmark concerned mostly with gaming graphics. It consists of 60 fragment shaders and 30 vertex shaders.

3.2 Measurement Details

For the compilation time benchmark, the compiler was timed when compiling all shaders in a benchmark 10000 times for each benchmark.

When measuring the compiled code benchmark the instructions of the generated binary were used. By counting the number of cycles used in the arithmetic logic unit and counting the maximum number of registers needed at any time two measurements were obtained.

3.3 Compilation Benchmark

The time the compiler needs to compile shaders in the benchmarks, less time is better. The results are presented as a ratio against the same benchmark

compiled with no basic-block vectorization enabled.

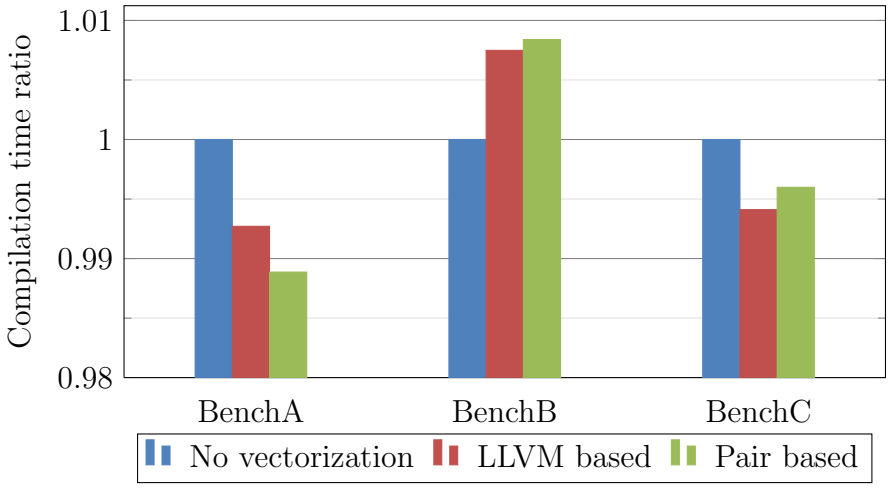

Figure 3.1: Compilation time benchmark for three auto-vectorization alternatives. Results are cumulative for each benchmark.

In two benchmarks out of three, the compilation time is decreased. Overall, the both algorithms perform equally well. The reasons for this will be discussed Section 4.2.

3.4 Compiled Code Benchmark

3.4.1 Arithmetic logic unit

The sum of the number of the arithmetic cycles used in the benchmark, fewer arithmetic cycles is better. The results are presented as a ratio against the same benchmark compiled with no basic-block vectorization enabled.

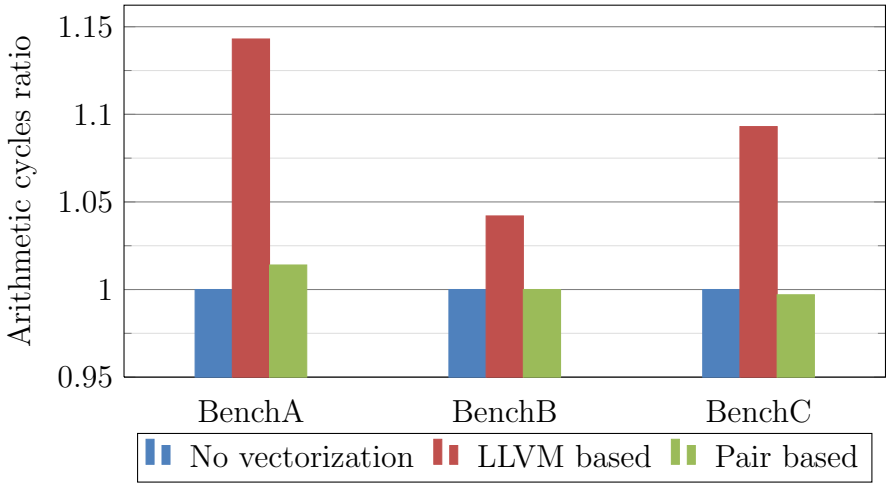

Figure 3.2: Arithmetic cycle benchmark for three auto-vectorization alternatives. Results are cumulative for each benchmark.

The LLVM based algorithm has worse arithmetic cycles performance than the pair based algorithm. The pair based algorithm does to a large extent not affect the arithmetic performance.

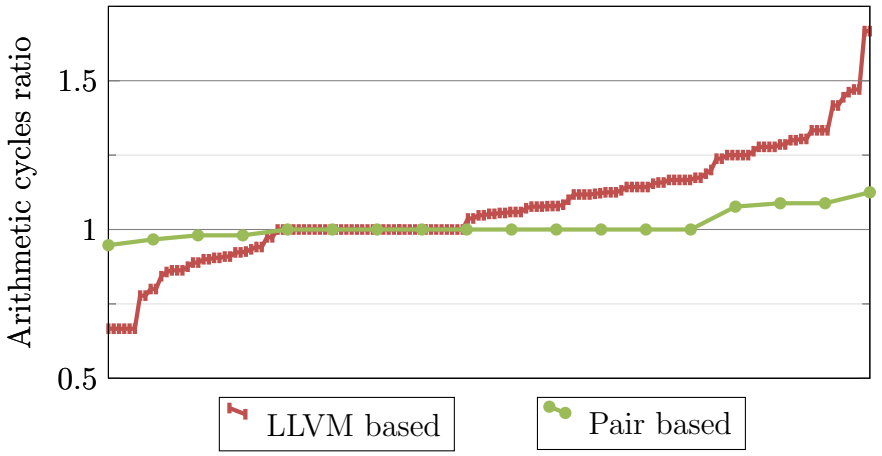

Figure 3.3: Arithmetic cycle benchmark comparing the performance of shaders changed by vectorization to their unvectorized equivalents.

The arithmetic cycles performance impact of the LLVM based algorithm is very volatile and overall negative. The arithmetic cycles performance impact of the pair based algorithm is slightly negative but in most cases unchanged.

3.4.2 Register usage

The sum of the number of registers needed to run the benchmark, fewer registers is better. The results are presented as a ratio against the same benchmark compiled with no basic-block vectorization enabled.

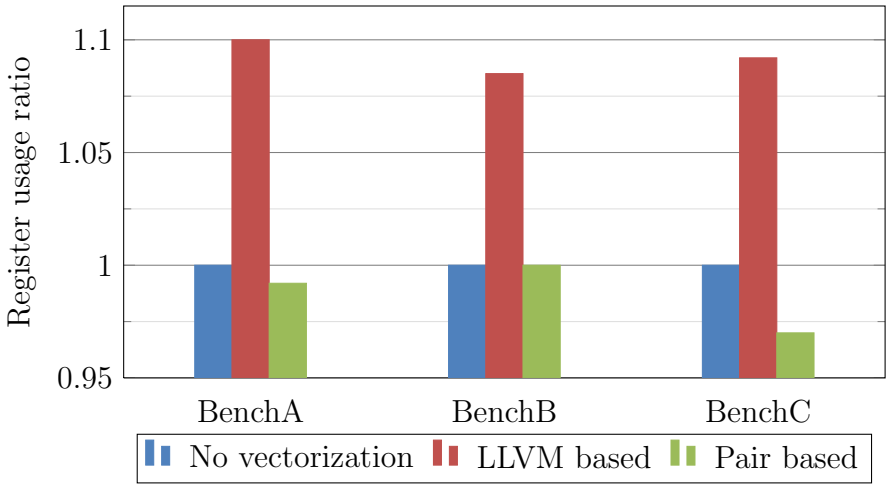

Figure 3.4: Register usage benchmark for three auto-vectorization alternatives. Results are cumulative for each benchmark.

The overall register usage impact of the LLVM based algorithm is negative, while the pair based approach is has a slightly positive register usage impact.

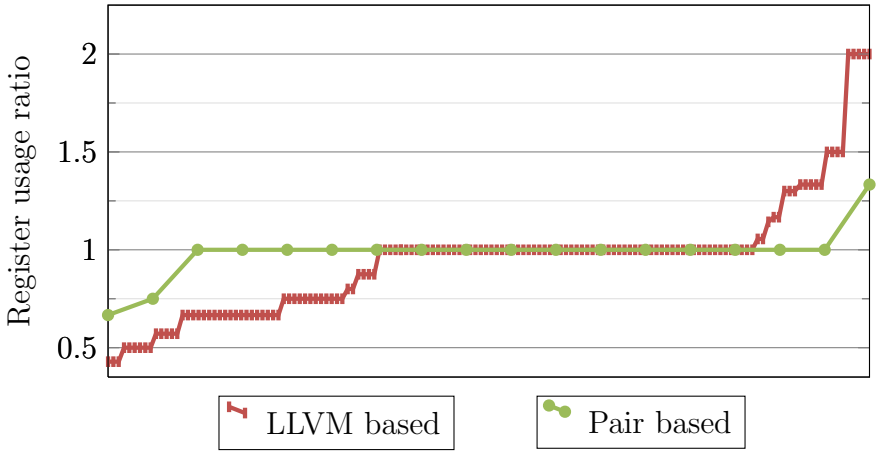

Figure 3.5: Register usage benchmark comparing the performance of shaders changed by vectorization to their unvectorized equivalents.

The register usage impact of the LLVM based algorithm is volatile and overall negative. The register usage of the the pair based algorithm is overall slightly decreased, but in most cases unchanged.

Chapter 4

Discussion

The pair based basic-block vectorizer is much simpler than the LLVM based basic-block vectorizer and as a result the heuristics employed by it can be more exact and much easier to develop. On the other hand the LLVM based basic-block vectorizer compares sets of pairs to fuse. That gives it the opportunity to consider what the consequences of a single fused pair will be, and avoid alternatives that may seem profitable on the surface but are not in the context of further vectorization.

A serious issue for machines with vector-registers only, one can expect the vector instructions to be scheduled on the same hardware as non-vector instructions. That means that the reduced number operations needed when changing two operations into one vector operation is entirely offset by a single move operation. This means that we have to rely on vectorization options that are known to be essentially free or at least cheaper than the operation that is removed by fusing a pair of operations. This alters the notion of heuristics and scores into something more like a filter or a white-list, where only known cheap or free options are allowed. This is especially problematic in the context of GPU programs since they tend to be very small and often offer few opportunities for vectorization to begin with.

The LLVM basic-block vectorizer is based on the idea of amortizing the cost of moving data into vector register by making sure that data that has been moved is used at least a number of times before it has to be moved out again. This assumption works well in the traditional context of multimedia extensions on CPUs with separate register and vector register banks, but has very little meaning in the context of an all vector-register machine since data is already located in vector registers. However incorporated in that same idea is the notion of fusing a set of operations that are deemed more profitable than another set. This is valuable and can be used to avoid fusing pairs of operations that later will prohibit more important pairs from being fused.

4.1 Compiler

One issue when developing heuristics and trying to determine their effectiveness is the noise created by other optimization passes, transformations and the back-end. Simply replacing an operation with a new and equivalent operation might change the instruction scheduling. If larger changes are made, it is likely that later optimization passes and transformations will interact with the changes and create ripples of changes that ultimately are likely to alter instruction selection and scheduling even further. The noise could be minimized during the development of a specific optimization pass or transformation by running as little other functionality in the compiler as possible. But this would create an issue of selecting which behavior is correct when integrating the new optimization or transformation pass with the full compiler.

When two operations are performed separately they can be shuffled around to a larger extent and are likely to use register resources at different points in time. However when both operations are combined into a vector operations they will require at least the same amount of register space, but now at the exact same time, which makes register allocation harder and more likely to spill.

Another effect of combining two operations is making instruction scheduling harder because more dependences are concentrated in a single operation. When scheduling these operations, fewer alternatives are possible and the instruction scheduler may be forced to create a more expensive schedule.

4.2 Evaluation

The cases of increased compilation speed in Figure 3.1 are surprising, but can be explained by the number of operations being removed by vectorization. This correlates well with the results in Figure 3.2 and Figure 3.4, where performance of BenchB is not changed other than the compilation time is being increased.

Figure 3.2 shows an overall decreased arithmetic performance of the LLVM based algorithm. Specifically Figure 3.3 shows that about half of the vectorizations made are unprofitable and about a fourth of them are profitable. In most cases this is due to the heuristics, which are not very conservative. There clearly are cases which cause bad performance that are not taken into account when computing the **TreeScore**.

Figure 3.2 shows almost unchanged arithmetic performance for the pair based algorithm. Specifically Figure 3.3 shows that most of the cases where

vectorization is performed the result is unchanged performance. This is due to vectorized operations not always translating into faster binaries. The changes produced are fed into the back-end of the compiler, which is highly tuned for the parameters that it usually sees. Vectorization causes parameters like register pressure and scheduling tightness to be changed which might cause the back-end to generate sub-optimal solutions.

Figure 3.4 shows an overall increase in register usage of the LLVM based algorithm. Specifically Figure 3.5 shows that the register usage is quite volatile, this can be attributed to the heuristics of **TreeScore** not being very conservative.

Figure 3.4 shows an overall decrease in register usage of the pair based algorithm. This is not something to be expected from vectorization in general, but some heuristics of **PairScore** look for cases where register usage can be decreased. However Figure 3.5 shows that in most cases register usage is unchanged, which is the best case scenario while vectorizing, due to the increased register pressure.

4.3 Implementation

While pruning duplicated operations in Algorithm 6 there is a decision about which pair to prune if pairs containing the same operations are found. The current solution of simply pruning all pairs after the first is a simplistic one. A better heuristic would be to compare which of the conflicting pairs can be removed to minimize the decrease in **TreeScore**. However if there are multiple conflicting sets of pairs the problem gets harder due to the interdependences of conflicts, which makes this a compile-time vs. run-time trade off. The increase in complexity and compile-time was deemed not worth it for the few cases where conflicts were found.

The heuristics are the most important factor for runtime performance of programs compiled with these algorithms. For different compiler types the implementation of **TreeScore** and **PairScore** should vary greatly.

Compilers which compile for many hardware architectures should have an implementation of the heuristics that is very general. This can be accomplished in one of two ways. The first one is to give this algorithm access to architecture specific information. The second option is to make sure that the heuristics make no architecture specific assumptions. If the second option is elected, there are still some fundamental architecture specifics that are needed. **IsFusible** and **AreFusible** both depend on the maximum supported vector length of the hardware. A maximum vector length longer than what the hardware supports could be used and the compiler back-end could

simply break illegal vector operations into smaller parts. While this is possible, it creates more work for the algorithm and the results may need to be truncated in an uncontrolled and suboptimal way.

For compilers that are used for a narrow range of hardware architectures the heuristics implementation should take as many hardware aspects as possible into consideration. Some aspects may be expensive to find or verify which leads to a compile-time vs. run-time trade off.

When combining pairs of operations into trees, only pairs where the output of both operations in the pair are used directly by the operations in the next pair are added to the tree. A possible option is to let pairs which only use the output of one of the previous pairs be added to the tree. This would allow for longer trees to be built, however this would require more moves which are expensive.

4.4 Problems Encountered

Since this optimization is new in the context of this compiler, some performance-wise incompatibility is to be expected. For example the back-end is finely tuned to input which has a certain register pressure and might perform worse if the pressure is increased. Issues like these made development of general heuristics very difficult and a white-list/filter approach was mainly used instead. Overall, the back-end and the code generation have the biggest issues in terms of developing a vectorization algorithm that improves performance. The workarounds that have been employed to achieve the current level of performance have also made the implementation of this algorithm very specific to the compiler it was developed for.

Multiple issues relating to scalability exist. Given a set of operations of a single type with size n the number of possible combinations is $\frac{n^2}{2}$. Most of the possible operations are fusible but might have internal dependences.

A single call to **HasDependence** has the time-complexity $\mathcal{O}(2^{\frac{h_{bb}}{2}})$, where h_{bb} is the height of the basic-block dependence tree and each operation is assumed to have a direct dependence on two operations. h_{bb} does in turn depend on the size of the basic-block. This time-complexity models the basic block as single tree, producing a single output. In realistic scenarios there are likely multiple outputs and the direct dependence graph is a directed acyclic graph, not a tree.

When combined the time-complexity of initially building a tree for a single operation type is $\mathcal{O}(\frac{n^2}{2} * 2^{\frac{h_{bb}}{2}})$. This is not a large problem for GPU compilers at the moment, but basic-block sizes are likely to increase in the future. For increasing sizes of n and h_{bb} bailouts or fast-paths need to be implemented.

Since much of the compilation of a GPU compiler is done during the run-time of applications, compilation time is a very important factor.

Chapter 5

Conclusion

As it can be seen from the run-time charts, the LLVM based vectorizer performs worse than no vectorization at all throughout all the tests. The pair based vectorizer performs better in three out of six benchmarks, produces no changes in BenchB and increases the arithmetic cycle count for BenchA. Combined with the run-time charts, the compilation-time chart hints at vectorization increasing the compilation speed for cases where profitable vectorization options are found. Another conclusion is that BenchC lends itself very well to vectorization.

In general the performance of the output of the pair based vectorizer is unchanged. This is due to the heuristics being very careful as to not do unprofitable vectorization, still unprofitable vectorizations are performed. If more cases of profitable vectorization were found, more performance gains could be had.

The goal of this master's thesis was to find and develop a basic-block auto-vectorization optimization pass suitable for a graphics compiler. Two optimization passes have been developed, however no large performance gains have been found with either of them, but compilation speed increases have been found.

Improving performance for a graphics compiler on an all vector register machine is difficult for a few reasons. Move operations are likely to be carried out on the same hardware as the vector operations, which in principle prevents the usage of any moves. The same issue of sharing hardware resources is likely to be valid for operations and their vector operations counterparts, which means that a successful vectorization only saves a single cycle. Register pressure is increased which can have very severe consequences in the context of graphics applications and performance. Additionally the instruction scheduler will be faced with a tighter schedule as a result of two operations that previously could be chronologically separate, but now need their

dependences to be met at the same time.

5.1 Future Work

Further exploration into heuristics is needed to find more cases of guaranteed or almost guaranteed profitable vectorization options. Many vectorization options can be free in some or most cases but are not always free. Some performance gains could be had if more bad vectorization options were known.

HasDependence could be improved to have a better time-complexity using an algorithm like Tarjan's strongly connected components (henceforth SCC) algorithm [12]. If the operations in the pair are temporarily fused and SCC does find a group of size larger than one, a dependence cycle has arisen. SCC has the time-complexity $\mathcal{O}(|V| + |E|)$, where V is the set of operations in the basic-block and E is the set of direct dependences between operations.

Caching the results of **HasDependence** is another alternative that is straight forward to implement and would yield a performance improvement. However the cache would need to be cleared after every fused operation pair, which limits its effectiveness.

Bibliography

[1] J. R. Ball, R. C. Bollinger, T. A. Jeeves, R. C. McReynolds, and D. H. Shaffer. On the use of the solomon parallel-processing computer. In *Proceedings of the December 4-6, 1962, fall joint computer conference*, AFIPS '62 (Fall), pages 137–146, New York, NY, USA, 1962. ACM.

[2] Keith Diefendorff, Pradeep K Dubey, Ron Hochsprung, and HASH Scale. Altivec extension to powerpc accelerates media processing. *Micro, IEEE*, 20(2):85–95, 2000.

[3] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel avx: New frontiers in performance improvements and energy efficiency. *Intel White paper*, 2008.

[4] ARM. Architecture and implementation of the arm cortex-a8 microprocessor. *ARM White paper*, 2005.

[5] Randy Fernando, Mark Harris, Matthias Wloka, and Cyril Zeller. Programming graphics hardware. *EUROGRAPHICS (Tutorial)*, 2004.

[6] nVidia Corporation. Geforce 256, the world's first gpu, 1999. `http://www.nvidia.com/page/geforce256.html`.

[7] Guennadi Riguer. The radeon x1000 series programming guide, 2006.

[8] David Luebke and Greg Humphreys. How gpus work. *Computer*, 40(2):96–100, 2007.

[9] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools with Gradiance*. Addison-Wesley Publishing Company, USA, 2nd edition, 2007.

[10] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[11] Dorit Naishlos. Autovectorization in gcc. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004.

[12] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[13] Juergen Lorenz, Stefan Kral, Franz Franchetti, and Christoph W Ueberhuber. Vectorization techniques for the blue gene/l double fpu. *IBM Journal of Research and Development*, 49(2.3):437–446, 2005.

[14] Hal Finkel. The LLVM basic-block autovectorization pass. *llvm-3.1/lib/Transforms/Vectorize/BBVectorize.cpp*, 2012.

[15] David J Kuck, Robert H Kuhn, David A Padua, Bruce Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–218. ACM, 1981.