Faculty of Computer Science, Electrical Engineering and Mathematics

Department of Computer Science

An LLVM based toolchain for transparent acceleration of digital image processing applications using FPGA overlay architectures

Master Thesis

Supervisors:

Jun. Prof. Dr. Christian Plessl

Lukas Funke

Prof. Dr. Marco Platzner

EMPTY version

Paderborn, July 2015

Abstract

THIS IS MY ABSTRACT

Declaration

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Paderborn, 16.8.2015 Lukas Funke

Preface

Hier entsteht das Vorwort

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Accelerating the execution of software is an important task, either from an economical point of view (e.g. energy costs, hardware utilization), because runtime is a fundamental requirement (e.g. real-time computing), or because data-sets are so huge, that they cannot be handled in reasonable time any more. For this purpose, hardware accelerators like GPUs and FPGAs are more and more used in general purpose computing as an additional resource.

Applications can benefit from the use of hardware accelerators. They manage the use of accelerators and even require the presence of a certain accelerator type. This behaviour might be undesired, either because the computing resource is unavailable or blocked, or because there is an additional resource from which the application could benefit more, in terms of performance and energy-efficiency. Thus migrating software tasks, or at least parts of it, between different accelerator types is a desired feature for computing in heterogeneous environments.

Migrating binary programs on-the-fly between heterogeneous computing resource is still challenging. This holds in particular for non instruction-based architectures such as FPGAs A special challenge in this area is the avoidance of synthesis overhead. Mapping

and synthesizing binary applications to an equivalent FPGA implementation still takes a disproportional amount of time. As a consequence, tasks have to run longer to amortize these costs.

One approach to address this issue is to use domain specific overlay-architectures. An overlay is an FPGA configuration, which implements a set of predefined, usually coarse-grained elements, which are themselves reconfigurable. Their granularity can vary from networks of fine grained operations like adders and multipliers , to vector-processors or even many-core architectures. An overlay is considered as domain specific, if it is customized for a certain application domain. As an example, one can think of a generic convolution filter with a configurable filter matrix. One major advantage of such architectures is that they can typically be reprogrammed much faster compared to re-synthesizing a complete FPGA-design. On the other hand, a single overlay might be too specific for a wide range of tasks. For this reason, there is no "one overlay fits all"-solution, but one has to choose from many overlays, depending on the application and its parameters. This, together with the overlay configuration, is a manual process.

Chapter 2

Related Work

And the second real chapter.

Chapter 3

Algorithm recognition

The goal of this thesis is to identify resource intensive parts of a computer program which can be replaced by call to a semantically equivalent accelerator function with a lower execution time.

3.1 Problem definition

To achieve our goals we first have to define what a *part of a computer program* is. Therefore we have to know the language and structure of the input program. In this theses the input consists of *LLVM IR*. This is an intermediate language which is used in the LLVM compiler framework. The advantage of an IR is the strict separation of compiler frontend, optimizer and backend. While the frontend can generate IR code from several source languages, the optimizer and backend only have to deal with a common intermediate language.

The smallest organization unit of LLVM IR is an *instruction*. LLVM IR uses Static single-assignment form (SSA) and three address code. Each instruction defines a variable with a unique name, which can be used multiple times. This introduces a *def-use* chain, which we will use later to verify a valid replacement.

Instructions are grouped into *basic blocks*. Each basic block contains a maximal-length sequence of instructions that execute without a branch. Basic blocks are grouped into a

function where each function lives inside a *module*. A module corresponds to a translation unit of the source language, e.g. a .c/.cpp-file.

All basic blocks of a function form a *control-flow graph*(CFG). This is a directed graph $G = (V, E)$ where each node $n \in V$ corresponds to a basic block and each edge $(u, v) \in E$ represents a possible transfer of control (branch) from block u to block v [3]. An example of a simple control flow graph is illustrated in figure 3.1.

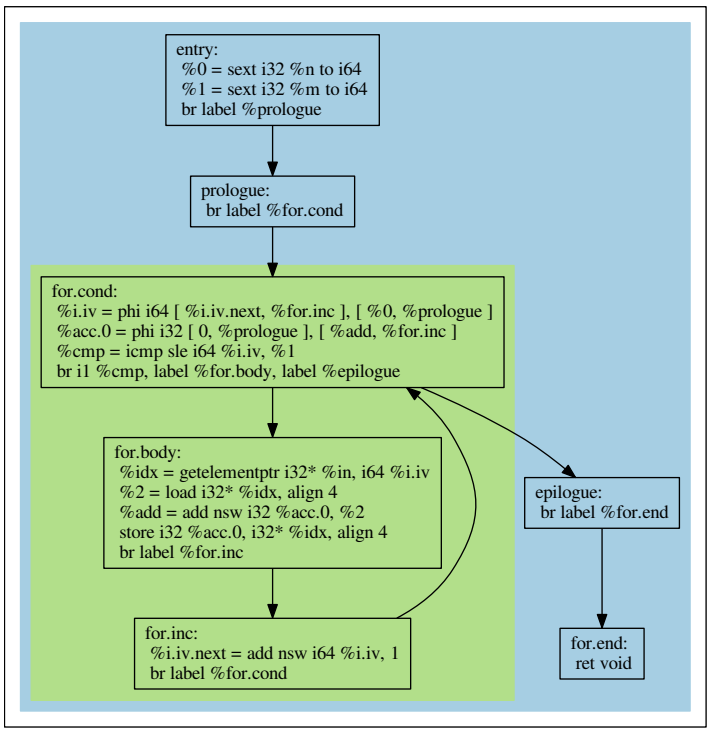


Figure 3.1: Example of a control-flow graph with basic blocks and their corresponding instructions.

Definition 1. Let H be a subgraph of G, where G is the complete control-flow graph of a function F. A part P of a computer program, which is a candidate for replacement, is a subgraph H consisting of as set of basic blocks. Each part P has to suffice the following conditions:

- There exist at least one value in the successor blocks of P, argument list of F or constant inside P which is a used by P. These values are considered as *input parameters*.

- There exist zero or more instruction in P that have an external users in one of the predecessor blocks of the induced graph G/H. These are considered as *output parameters*.

In figure 3.1 this part, further referred to as *candidate*, could be the set of basic blocks containing $C = \{for.cond, for.body, for.inc\}$, the input parameters %0 and %1 and no output parameter. In this example they implement a simple loop. Epilogue and prologue are place holders for arbitrary code.

After defining what a valid candidate is, we also have to define what we mean by *semantically equivalent*. We follow the definition in [7] in a slightly modified form:

Definition 2. Let P and P' be two program parts, where P' is called a *pattern* for P. Further let I_i and I'_i with $i = 0, \ldots, n$ be the set of input parameters and O_j and O'_j with $j = 0, \ldots, m$ be the set of output parameters.

P and P' are semantically equivalent if and only if they there is an binding of parameters with equal types with the $I_i = I'_j \quad \forall i, j \in 0 \ldots n$ such that when P' is called, one of the following is true:

- P and P' will not halt or

- $O_i = O'_i \quad \forall i \in 0 \ldots n$.

In short definition 2 means: if two program parts have the same input, they always produce the same output.

We can now express the problem as followed: given an accelerator interface A, search a candidate P and a binding of input parameters to A such that they are semantically equivalent. Unfortunately there is no algorithm for testing the semantic equivalence of P

and A since it is an undecidable problem.

Instead of testing if P is semantically equivalent to A, we only check if P is an *algorithmic equivalent* of A. Algorithmic equivalent was defined by Metzger et al. [7].

Definition 3. Let P and P' be two program parts. Define P algorithmically equivalent to P' ($P \equiv P'$) if and only if one can be derived by another by the following operations:

- Simultaneous renaming of variables
- Semantic preserving reordering of expressions
- Permutation of commutative operations

Listings 3.1 and 3.2 showing an example of algorithmic equivalence. The right-hand side is derived from the left-hand side by renaming the variables in their declaration and use in both statements, exchanging the order in both statements (since there is not data dependency) and changing the operators in the multiplication statement.

```
1 int i, j
2 float A[n][m], B[n][m]
3 for i = 2, i < n, i++
4   for j = 2, j < m, j++
5     A[i][j] = A[i][j-1]
6     B[i-1][j-1] = 42 * B[i-2][j-2]
```

Algorithm 3.1: Original code

```
1 int a, b
2 float X[r][t], Y[r][t]
3 for a = 2, a < r, a++
4   for b = 2, b < t, b++
5     Y[a-1][b-1] = Y[a-2][b-2] * 42
6     X[a][b] = X[a][b-1]
```

Algorithm 3.2: Derived code

Metzger et al. also define the more general term of an *algorithmic instance* which also covers the substitution of induction values. As our goal isn't to solve algorithmic recognition in general we will leave this definition by side.

3.2 Algorithmic pattern recognition

Our goal is to determine if there exist a *candidate* in the input program which is algorithmically equivalent to the behaviour of one of our accelerator function. Therefore we

require for each accelerator function one or more equivalent implementations. These implementations will be further referred to as *patterns*. We will first take a look at pattern representation in section 3.2.1.

Of course there is a potential for slight variations in the input program. If we take a look at the statement level there are multiple ways to represent an arithmetic expression. E.g. $a = b * (c + d) \equiv a = b * c + b * d$. These variations also exist on the control-flow level. In order to reduce the number variations in the input we first have to canonicalize it. LLVM provides a brought range of canonicalization passes, which we will revise in section 3.2.2

To check if one of our patterns is included in the input, we first have to check whether there exists a subgraph in the input control-flow graph which can be mapped to one of our patterns. This problem is known as *subgraph isomorphism*. The isomorphism gives us a injective mapping of nodes from the candidate CFG to the pattern CFG. We will later see a formal definition of a subgraph isomorphism, common algorithms to find them and an examples in section 3.2.3.

If we found a mapping from the pattern control-flow to a possible candidate, we have to check if this assignment is valid. For each potential transfer of control the branch has to happen under the same condition as in the pattern. Further we have to check for each expression in the pattern if there exist a corresponding expression in the candidate. We will deal with this problem in section 3.2.4.

3.2.1 Pattern representation

A pattern includes the required control-flow of basic blocks, the condition for each possible transfer of control and all computational statements. Additionally a pattern includes which hardware interface is represented. For an interface a pattern also defines what values of the statements were bound to which argument. All information are encoded in the *JavaScript Object Notation* (JSON) format. JSON is an open, text based, human-readable data interchange format with a small set of formatting rules **??**. Although patterns can be generated

from llvm functions, the specific binding to an interface can not be automatically derived. The properties of JSON, in particular the human-readability, facilitate the postprocessing of patterns.

A set of patterns can be combined to form a database.

```
Pattern := '{' VersionAttr ',' NameAttr ',' HWIface ',' Graph '}'

HWIface := 'hwiface:' '{' NameAttr Binding '}'
Binding := 'binding:' '[' ('[' int, llvmtype']' Sep )+ ']'

Graph := 'graph:' '{' '[' NodeObj ',' EdgeObj ']' '}'
NodeObj := '{' 'nodes:' '[' (Node Sep)+ ']' '}'
EdgeObj := '{' 'edges:' '[' (Edge Sep)+ ']' '}'
Edge  := '{' 'src:' int, 'dst': int '}'

Node  := '{' IDAttr ',' Attributes  '}'

Attributes  := 'attributes:' '{' (CFGAttributes | StmtAttributes) '}'

CFGAttributes := LabelAttr? ExpressionsAttr?
StmtAttributes  :=
        '{' (OpcodeAttr | ArgAttr | ConstAttr) ','
          IValAttr? ',' TypeAttr  '}'

ExpressionsAttr := '[' Graph+ ']'

VersionAttr := 'version:' int
NameAttr  := 'name:' string
IDAttr    := '_id:' int
OpcodeAttr  := 'opcode:' '['  ']'
ArgAttr   :=  'argument:' string
ConstAttr :=  'const' (int | float | double)
TypeAttr  := 'type:' llvmtype
IValAttr  := 'i_val:' [int, int, string]
LabelAttr := 'labels:' '[' (int Sep)+ ']'

Sep := ',' | ϵ
```

Format 3.3: Pattern format description

Due to the text based nature of JSON, one major drawback is the size of a pattern as file. They become large even for relative simple program parts. An instance of the format 3.3 can be found in appendix **??**.

3.2.2 Program canonicalization

There are several ways to write down an abstract operation like convolution in a high-level language like C. Besides the fact that there are several algorithms, each algorithm can be expressed in a different way. Similar formulations with minor differences in the high level language can in turn produce different outcomes in LLVM IR. E.g. computational statements like $x = 8 * (a + b)$ can be expressed as $x = 8 * a + 8 * b$ or $x = a << 3 + b << 3$ and so on. Although these variations can also be covered by increasing the number of patterns, the recognition process will be faster if the number of patterns is small. One idea to reduce the number of variations is to transform the program into a *canonical* form as suggested in [7].

In order to canonicalization the LLVM IR, one can make use of the already available transformation passes, implemented in the optimizer. There are several passes that can be used for normalization. In the following we will look at some of LLVMs optimizations which we can use for canonicalization.

Interprocedual canonicalization

As our recognition pass does not look across a functions boundary we have to inline eventually called functions. This is accomplished by using llvms inlining pass. In practice the inliner calculates the *costs* for each potential inlining. A threshold then decides whether the callee gets inlined or not. The costs are calculated based on the number of arguments, their method of passing, the number and type of instructions and so on. Thus the number of patterns we can recognize also depends on the inlining costs, which can be given as an input to the optimizer. As we will see in the evaluation the default cost threshold of 255 turn out to be valid trade-off.

Why inlining is required is illustrated in figure 3.4. Even if one can recognize that the *matrix_multiply(...)* is candidate for hardware migration, the speedup would be rather small, as the matrices in image processing are usually $3x3$ or $4x4$. However if we inline this operation and involve the two other loops, we may gain a speedup.

```
1 for (j = 0; j < ny; ++j){
2   for (i = 0; i < nx; ++i) {
3   ...
4   matrix_multiply(...)
5   ...
6   }
7 }
```

Algorithm 3.4: Potential inlining opportunity

Control flow canonicalization

A critical part of the control-flow are loops. LLVM detects natural loops using the *loop* analysis pass. A natural loop has exactly one entry point, the *header* block, which dominates all blocks in the loop. The header has exactly one *backedge* entering it. A backedge is an edge coming from one block of the loop body [2, p.662-665].

The first transformation pass we apply is the *loop-simplify* pass. This pass guarantees that the resulting loop has the following properties:

- The header block has a single, non-critical input edge coming from a *pre-header* block outside the loop.
- A loop must have a *latch*, the source node of the only backedge. This block will be executed in before every new iteration. The latch is not required to be a separate block. It can be the same block as the header.
- There is a single exit block which will always be executed after loop exit.

Loop invariant code motion (*licm*) is another optimization pass which contributes to canonicalization. This pass tries to move as much invariant instructions as possible from inside of a loop body to its pre-header or exit block. Thus moving expression which have the same value at each iteration out of the loop.

```
1 for (j = 0; j < n; ++j)
2   for (i = 0; i < m; ++i) {
3     a = b + c
4     C[j][i] = a
5   }
```

Algorithm 3.5: Before licm

```
1 a = b + c
2 for (j = 0; j < n; ++j)
3   for (i = 0; i < m; ++i) {
4     C[j][i] = a
5   }
```

Algorithm 3.6: After licm

Loop unrolling is a well known optimization technique, however it makes algorithm recognition difficult. When the iteration count of a loop is known at compile time the compiler may replicate the loop body as many times as the loop gets iterated and builds instances of the induction values where they were used [3, p. 441]. However this requires to have a pattern for each potential unrolling count. In order to reduce the number of patterns Metzger et. al [7] suggest to re-roll the loops. However there is currently not general loop-rerolling pass in llvm. Thus we will only recognize program parts which were not unrolled.

Basic Block canonicalization

There are several optimizations that work in the scope of one basic block. In the following we list the most common.

Memory to register (mem2reg)

switch: -mem2reg

This pass promotes alloca instruction into scalar registers (SSA values). Load and store instructions are replaced by register access. This enables a set of further optimizations.

```
1  define i32 @foo(i32 %n) {
2  entry:
3    %n.addr = alloca i32, align 4
4    store i32 %n, i32* %n.addr, align 4
5    br label %some_bb
6
7  some_bb:
8    %0 = load i32* %n.addr, align 4
9    %1 = add i32 %0, 1
10 ...
```

Listing 3.7: Before mem2reg

```
1  define i32 @foo(i32 %n) {
2  entry:
3    br label %some_bb
4
5  some_bb:
6    %1 = add i32 %n, 1
7
8
9
10 ...
```

Listing 3.8: After mem2reg

Combining redundant instructions

switch: -instcombine

This pass combines redundant instructions to generate fewer instruction in the output llvm code. There is also canonicalization performed, e.g. all constant operands were moved to the right hand side, multiplications with power of two arguments are translated to shifts, and so on. The full list is provided in [1].

```
1  %1 = add i32 %0, 5
2  %2 = add i32 %1, 3
3  ...
```

Listing 3.9: Before instcombine

```
1  %2 = add i32 %0, 8
2
3  ...
```

Listing 3.10: After instcombine

Common subexpression elimination

switch: -early-cse

A common subexpression is a part of an expression which has been previously computed. E.g. $y = n*11+(m+(n*11))$ can be reduced to $x = n*11$ and $y = x+(m+x)$. This opens an opportunity for further reductions such, e.g. $y = 2 * x + m$ where strength reduction can be applied: $y = x << 2 + m$.

Early common subexpression elimination removes these trivially redundant instructions in an early stage of the optimization.

```
1 %n1 = load i32* n, align 4  ; readonly
2 %1 = add i32 %n1, 5
3 %n2 = load i32* n, align 4  ; readonly
4 %2 = add i32 %n2, 0
5 %3 = mul i32 %2, 3
6 ...
```

Listing 3.11: Before early-cse

```
1 %n1 = load i32* n, align 4  ; readonly
2 %1 = add i32 %n1, 5
3 %3 = mul i32 %n1, 3
4
5
6 ...
```

Listing 3.12: After early-cse

Reassociates commutative expressions

switch: -reassociate

This pass reassociates commutative expression. E.g. $a = (x + 4) + 9$ will be transformed into $a = x + (4 + 9)$. A subsequent pass constant propagation, or instcombine pass will lower this expression into $a = x + 13$.

3.2.3 Matching the control flow

As mentioned in the overview of this section, matching the control-flow of a possible candidate to the pattern control-flow, is the problem of finding the subgraph isomorphism of two graphs. First, we formally define what a subgraph isomorphism is, afterwards we revise a well known algorithms to solve this task in practice.

Definition 4. Let $G = (V, E)$ and $H = (V', E')$ be two graphs. A subgraph isomorphism of H into G is an injective relation $A \subseteq V \times V'$ such that for every pair $(v_i, v_j) \in V'$ and $(w_i, w_j) \in V$ with $(v_i, w_i) \in A$ and $(v_j, w_j) \in A$, $(w_i, w_j) \in E$ if $(v_i, v_j) \in E'$. A is then called the subgraph isomorphism of H in G [5].

Figure 3.2 illustrates the subgraph isomorphism of two graphs. Checking whether two graphs are isomorphic to another is not known to be in P or NP. The program is a special case of the general subgraph isomorphism, which is an NP-complete problem [6].

Since finding a subgraph isomorphism is required in a wide range of disciplines, from chemistry to circuit design, there exist a couple of algorithms to cope with this task. They usually give a lower bound for the runtime and perform quite good on graphs with a small

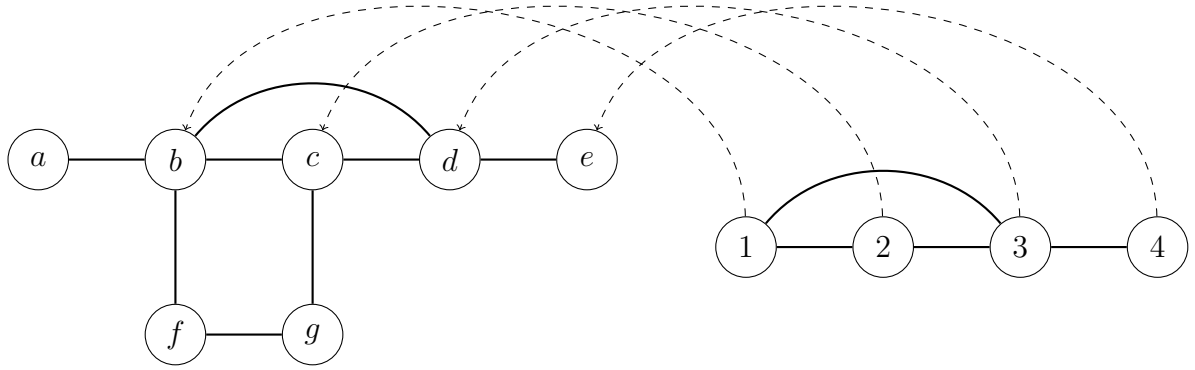


Figure 3.2: Example of a subgraph isomorphism

number of nodes. The number of nodes in control-flow graphs is in practice usually not as large as it would produce a huge runtime for isomorphism-algorithms. However this statement does not hold for all input programs. We have to keep in mind that finding an isomorphism might have exponential runtime.

Ullmann

A naive algorithm would enumerate all possible assignments of nodes from H to nodes of G and check for each assignment whether it is valid, according to the edge sets of both graphs. It is obvious that the search space becomes exponential in the number of nodes.

A more efficient algorithms for subgraph isomorphism was invented by Ullmann [4] in 1976. His algorithm extends the naive approach by continually removing infeasible assignments using a *refine procedure*. This narrows the search space and therefore reduces the runtime.

The original algorithm introduced by Ullmann in intended to work with adjacent matrices of undirected graphs, however our input looks different. The input, as illustrated in figure 3.1, is a directed graph, where each node only knows its successor nodes. Thus we will translate the original algorithm to work directed graphs an adjacent lists such that we

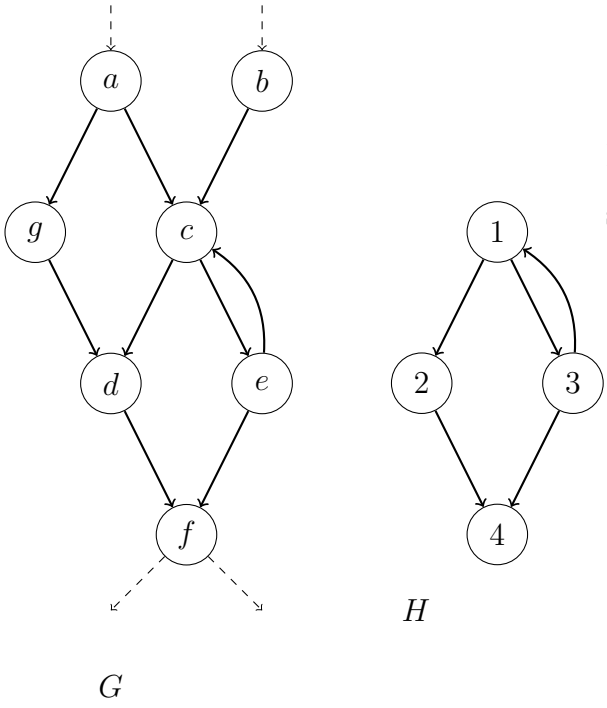
can find a subgraph isomorphism of a given pattern control-flow H in the input control-flow G.

In the implementation we made use of the *graph-traits* concepts of LLVM. This will make our algorithms work with arbitrary graphs, not only control-flow graphs. Graph-traits can be thought of as a generalized interface for graphs, which decouples their actual representation from algorithms which work on them.

Like Ullmanns algorithm, procedure *calculate_possible_assignments*(...) (listing 3.14) first calculates all possible assignment from the nodes of V' of H to the nodes V of G, which we denote as the function P. For each node $u \in V'$ this is the set $V'' = \{ \quad v \in V \quad | \quad \forall u \in V' : deg(v) \geq deg(u) \wedge nfn(u, v)\}$ where $nfn(\dots)$ is an additional narrow function. This function will further thin out the number of possible assignments in order to speed up our matching process. Figure 3.3 illustrates this operation for an exemplary control-flow.

```
1  find_isomorphism(GraphT G, GraphT H, list<map<NodeT* NodeT*>> A, nfn(NodeT*,NodeT*))
       -> bool
2
3      map<NodeT*, set<NodeT*>> P
4      calculate_possible_assignments(G, H, P, nfn)
5      found_assignment ← true
6      while(found_assignment)
7          found_assignment ← false
8          map<NodeT*, NodeT*> tmp
9
10         if(recursive_find_isomorphism(G, H, P, tmp))
11             A.add(tmp)
12             found_assignment ← true;
13             foreach(pair p : tmp)
14                 NodeT* Y ← p.first
15                 NodeT* X ← p.second
16                 P[Y].remove(X)   // Remove assigned node to find no isomorphism twice
```

Algorithm 3.13: Find isomorphism



Let $H = (V', E')$ and $G = (V, E)$ then a possible assignments is a function $P : V' \to V''$

- $P(1) = \{c, e, f, \dots\}$
- $P(2) = \{a, b, c, d, g, \dots\}$
- $P(3) = \{c, e, \dots\}$
- $P(4) = \{c, f, \dots\}$

Figure 3.3: Possible assignments for a subgraph.

```
1  calculate_possible_assignments(GraphT G, GraphT H, map<NodeT*, set<NodeT*>> P, nfn(
       NodeT*,NodeT*)) -> void
2
3      foreach(NodeT* u : H)
4        set<NodeT*> candidates
5          foreach(NodeT* v : G)
6              if(deg(v) ≥ deg(u) and nfn(u,v))
7                  candidates.add(v);
8        P.add((N, candidates))
```

Algorithm 3.14: Calculate possible assignments

After calculating possible assignments the algorithm uses $recursive\_find\_isomorphism(\dots)$ to recursively test each possible assignments. Line 4-9 in listing 3.15 test whether an assignment is feasible. Referring to definition 4 an assignment is feasible if and only if for each edge (v_i, v_j) in the subgraph H there is also an edge $(A(v_i), A(v_j))$ in G. In other words, for each edge in H there must be corresponding edge in G. If an assignment is not feasible the algorithm returns false and another path in the search tree will be taken.

Let A be an assignment in the example of figure 3.3 with $A(1) = e$, $A(2) = b$ and so

on. A is not a feasible, because $(e, b) \notin E$.

```
recursive_find_isomorphism(GraphT G, GraphT H, map<NodeT*,set<NodeT*>> P, map<NodeT*,
    NodeT*> A) -> bool

    refine_assignments(P);
    foreach(NodeT* uH : H)
        foreach(NodeT* vH : children(u))
            NodeT* uG ← A[uH]
            NodeT* vG ← A[vH]
            if(!is_edge(uG, vG)) // check if (uG, vG) is an edge in G
                return false

    if(|A| == |H|)
        return true

    foreach(NodeT* Y : H)
        if(A.contains(Y))  // skip node if already assigned
            continue
        set<NodeT*> candidates = P[Y];
        foreach(NodeT* X : candidates)
            A.add((Y, X))
            map<NodeT*, set<NodeT*>> P' ← P.copy()
            // update_possible_assignments(...)
            // Removes a node X from all possible assignments, except for Y
            update_possible_assignments(Y, X, P')
            if(recursive_find_isomorphism(G, H, P', A))
                return true
            A.remove((Y,X))
    return false
```

Algorithm 3.15: Recursive find isomorphism

The actual speedup comes from Ullmanns refinement procedure which is implemented in the $refine\_assignments(\dots)$ function. The function checks for each node $u \in V'$ and each possible assignment $v = P(u)$ if there is least one neighbour $u' = children(u)$ is in the set $P(v')$, where $v' = children(v)$. This check will be executed until no further deletions are possible.

```
1  refine_assignments(map<NodeT*, set<NodeT*>> P) -> void
2      changes ← true
3      while(changes)
4          changes ← false
5          foreach(pair p : P)
6              NodeT* Y = p.first
7              set<NodeT*> candidates = p.second
8
9              foreach(NodeT* X : candidates)
10                 match ← true if #children(Y) == 0 otherwise false
11
12                 foreach(NodeT* ny : children(Y)) // neighbour of Y
13                     set<NodeT*> A ← P[ny]
14                     foreach(NodeT* nx : children(X)) // neighbour of X
15                         if(A.contains(nx))
16                             match ← true
17                 if(!match)
18                     candidates.remove(X)
19                     changes ← true
```

Algorithm 3.16: Refine possible assignments

Let us again consider the example in figure 3.3 and look at the assignment $P(2) = g$. Node 4 is a neighbour of 2, but none of the neighbours of g (which is d) is in the set of possible assignments of 4. Thus the assignment $2 \rightarrow g$ will never be possible and g will be removed from the set $P(2)$.

If an assignment is feasible and all nodes of H were assigned, the recursive function finally returns true. Because there might be multiple subgraph isomorphisms the function $find\_isomorphism(\dots)$ will delete the previous found assignment from the possible assignments and will search again, until no more isomorphisms were found.

The more is known about the nodes the further we can restrict number of possible assignments before running the algorithm. Hence some nodes are associated with one or more *labels*. The label describes a role in the control-flow. Possible labels are *entry*, *exit*, *loop_header*, *loop_inc*, *loop_body* and so on. A node with a *loop_inc* label can never be mapped to a *entry* node.

3.2.4 Matching Statements

In the previous section we showed that it is possible to find a control-flow in the input which is equal to one of our patterns control-flow. Now we have to check if the assignment is valid, according the branch conditions and computations carried out by the nodes of the input CFG with respect to our pattern.

As described in 3.2.1 each node is associated with a list of statements. Since we only consider programs of a restricted domain we look at statements which are either a branch expression or a expressions which sink in a store instruction.

From 3.2.2 we known that each statement consists of a canonicalized directed acyclic graph (DAG). Each node of the DAG (in the figure ?? represented as a rectangular) represents an instruction carried out by a statement.

To finally match a statement from our pattern to a statement of a candidate we have to traverse both DAGs simultaneously in depth-first order to compare whether each value suffice the following conditions:

- Each instruction from the pattern statement has the same opcode as the candidate instruction.
- For each induction variable (IV), aka phi-node, in the pattern the *start*, *step* and *exit* expression is equal to the candidate IV.
- The resulting data type of each value in the candidate statement is the same as the data type of our pattern value.

For each value of the pattern statement we establish a bidirectional mapping to the matched value in the candidate statement. Thus we know which constant or function argument, (which are the input values of the hardware interface) is bound to the corresponding value in the candidate statement (and vice versa). We will use these bindings in section 3.3 in order to bypass the input parameters of the matched program part to the hardware interface. This binding is illustrated in figure ?? using the dotted bidirectional arrows between both statements.

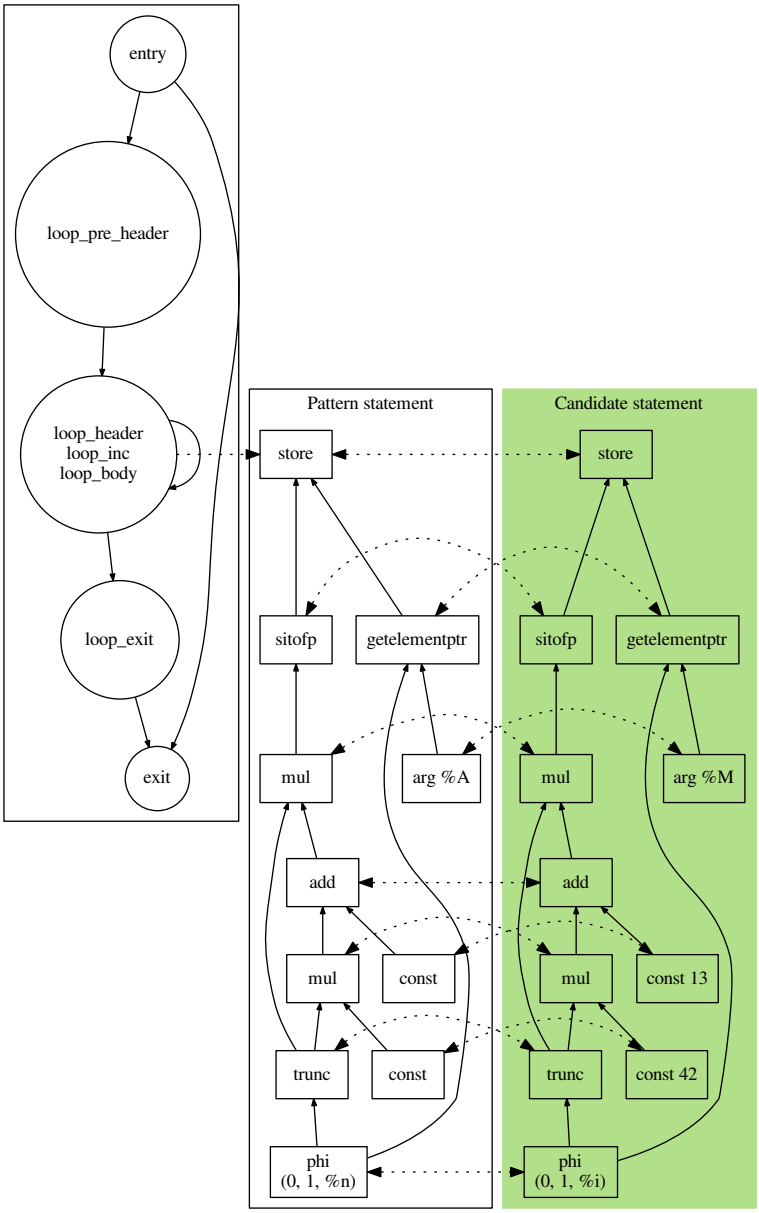


Figure 3.4: Abstract control-flow graph for the pattern of function *foo*.

3.3 Pattern extraction

In this section we will describe how a recognized algorithm can bound to a hardware interface. In general it doesn't matter what particular accelerator lives behind an interface, whether its a GPU or an FPGA. In this thesis we use the Maxeler dataflow computer as our main platform, thus in our case an FPGA lives behind the interface **??**.

Binding a program part to a hardware interface

What we know so far is that a particular program part is an algorithmic instance of one of our patterns. Further we know the binding of arguments and constants in our statements

Chapter 4

Evaluation

Chapter 5

Conclusions

Write your conclusions here.

Bibliography

[1] Llvms analysis and transform passes. 14

[2] A.V. Aho. *Compilers : principles, techniques, and tools*. Always learning. Pearson Education, 2014. 12

[3] K.D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann. Morgan Kaufmann, 2012. 6, 13

[4] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976. 16

[5] G. Valiente. *Algorithms on Trees and Graphs*. Springer, 2002. 15

[6] I. Wegener. *Complexity Theory: Exploring the Limits of Efficient Algorithms*. Springer, 2005. 15

[7] Z. Wen. *Automatic Algorithm Recognition and Replacement: A New Approach to Program Optimization*. MIT Press, 2000. 7, 8, 11, 13

Appendix A

My First Appendix

In this file (appendices/main.tex) you can add appendix chapters, just as you did in the thesis.tex file for the 'normal' chapters. You can also choose to include everything in this single file, whatever you prefer.