Develop and prototype code generation techniques for a clause-based GPU

Johan Ju, Niklas Jonsson
Develop and prototype code generation techniques for a clause-based GPU

Johan Ju
johanj93@gmail.com

Niklas Jonsson
niklasandersjonsson@gmail.com

June 2, 2017

Master’s thesis work carried out at ARM Sweden AB.

Supervisors: Jonas Skeppstedt, Jonas.Skeppstedt@cs.lth.se
Dmitri Ivanov, Dmitri.Ivanov@arm.com
Markus Lavin, Markus.Lavin@arm.com

Examiner: Krzysztof Kuchcinski, Krzysztof.Kuchcinski@cs.lth.se
Abstract

Processors grow more and more complex, even more so in the field of GPUs, where the instruction set, in general, is not publicly available. This enables them to change more rapidly than CPUs, since backwards compatibility is not an issue. In this thesis multiple approaches are investigated for code generation for a clause-based GPU. An optional pre-scheduler is implemented, together with two approaches for register allocation and one post-scheduler. It is found that a pre-scheduler can lower the register pressure considerably which makes a major impact on register allocation. The differences between the two register allocators is minimal and they can be considered to be equal.

Keywords: code generation, gpu, register allocation, instruction scheduling
Acknowledgements

We would like to thank ARM for giving us the opportunity to do our thesis in cooperation with them. Furthermore, we would like to thank our supervisors at ARM, Markus Lavin and Dmitri Ivanov, going out of their way to support us with everything from implementation details to thesis writing. We would also like to thank Jonas for his support and comments, both for this thesis but also the preceding literature study that is the basis for the theoretical parts of this thesis. Lastly, both of the authors would like to thank their respective family and friends for their love and support.
Contents

1 Introduction ........................................... 7
   1.1 Background ........................................ 7
   1.2 Bifrost architecture ............................... 7
   1.3 Problem ........................................... 9
   1.4 Goals and Limitations ............................. 9

2 Theory .................................................. 11
   2.1 SSA Form .......................................... 12
   2.2 Register Allocation ................................. 12
      2.2.1 Linear Scan .................................. 13
      2.2.2 Linear Scan on SSA form .................... 13
      2.2.3 Chaitin's algorithm ......................... 13
      2.2.4 Graph coloring on SSA form ................ 14
   2.3 Instruction Scheduling ........................... 14
      2.3.1 List Scheduling ................................ 15
      2.3.2 Modulo Scheduling ......................... 15
   2.4 Previous work ...................................... 16

3 Approach ............................................... 19
   3.1 Pre-scheduler ..................................... 20
      3.1.1 Model ......................................... 21
      3.1.2 Heuristics (priority function) .............. 21
   3.2 Live range-based register allocation ............. 22
      3.2.1 Shuffle insertion .............................. 23
      3.2.2 Wide variables as arguments ................ 25
   3.3 Coupled register allocator ....................... 25
      3.3.1 Spilling ....................................... 25
      3.3.2 Coloring ....................................... 26
   3.4 Post scheduling ................................... 26
Chapter 1
Introduction

1.1 Background

The thesis is carried out at ARM, a company that licences their hardware designs (intellectual property) to hardware manufacturers for use in System on Chips (SoCs). The office in Lund, where the thesis project has been carried out, is a part of the Media Processing Group which is responsible for developing the Mali GPU family as well as Video Processing Units (VPUs) and Display Processing Units (DPUs), together with their accompanying drivers. Like most modern GPUs, Mali features a programmable pipeline where some of the stages run shader programs whilst others are fixed in hardware. An application may either submit a shader written in a high level shader language which the driver needs to compile at runtime or a pre-compiled binary that the driver simply runs on the GPU. ARM has recently released a new hardware architecture called Bifrost and our task is to design a code generator to explore different approaches to code generation for shaders in Bifrost.

1.2 Bifrost architecture

The compiler in this thesis is targeting ARM’s Bifrost architecture which is their latest GPU design [19]. Bifrost is a scalar architecture where arithmetic instructions use scalar 32-bit registers as operands. This is different from the previous Midgard architecture which uses wide vector register operands. There are two arithmetic functional units in Bifrost called FMA and ADD/SF shown in fig. 1.1. Some instructions can only execute in the FMA unit while others can only execute in the ADD/SF unit. But there are also instructions that can execute in both units.

Instructions are represented in pairs so that every instruction pair has an instruction for each unit. Those instruction pairs are then inserted into a clause that can contain multiple instruction pairs. A clause is a series of instruction pairs that can execute without
switching thread. The clause can be of varying length. Every clause can only contain one instruction with uncertain latency e.g. memory access. Within a clause, instructions can fetch data from a temporary (bypass) register that contains the result from the latest executed instruction see fig. [1.2]. One benefit of using the temporary register is that a slot in the register file does not always need to be overwritten. Temporary registers can be seen as forwarding the result and can’t hold the value for an extended period of time.

Furthermore, the ISA (Instruction Set Architecture) of Bifrost supports wide instructions for memory access, i.e. instruction that write to more than one register. Whilst these are optional in general, the efficiency of code is generally increased if they are used. Take for example a load instruction that reads from memory and writes to a register. Instead of several such loads we can use a wide load that writes to several registers, saving one or more memory accesses.
1.3 Problem

The general problem is the classical task of combining instruction scheduling and register allocation to produce effective code. But the goal for scheduling for GPUs is different to traditional CPU scheduling. Because of the nature of graphical workloads, GPUs are designed to switch threads with little overhead. This feature is used to hide memory access latency as other threads can run instead of stalling. The Bifrost architecture introduces some important features that will affect the scheduler considerably. Firstly, as described above, there are clauses that bind together instructions with fixed latency that they can execute all the way without worrying about switching threads. The compiler is solely responsible for scheduling the clauses and allocating the bypass registers. There is also some benefit of keeping values in adjacent registers which needs to be considered by the register allocator. Together with this there are several constraints on how constant may be encoded into instructions, how clause execution may be interleaved, specific features of graphics hardware etc. that the scheduler and register allocator have to take into account when attempting to produce efficient code.

1.4 Goals and Limitations

The goal of this thesis/report is to develop and prototype code generation techniques for the Bifrost architecture. It will also present whether the traditional strategies for doing register allocation and instruction selection can be applied effectively to the architecture and if not, how one should perform them differently to generate code that performs well. For this purpose, a compiler back-end is developed consisting of instruction selection, an optional pre-scheduler, register allocation, post-scheduling and instruction emission. This is further detailed in chapter 3.

As for author contribution to this thesis, Niklas was mainly responsible for the sections on Graph Coloring algorithms, Modulo scheduling, section 3.2 section 3.4 whilst Johan wrote most of the sections on Linear Scan, List Scheduling, section 3.1 section 3.3 and section 5.2.
Chapter 2
Theory

Most compilers can be divided into three parts: front-end, middle-end and back-end. The front-end handles parsing of the input program, checking so that it follows the applicable syntax rules and translating the input program into some form of intermediate representation (IR) that is used for the rest of the compilation. The middle-end performs optimization that are not architecture-specific, e.g. dead code elimination. The back-end (also known as code generator) translates the IR to the assembly required for the target platform, thus the back-end is architecture-specific. As this thesis is mainly concerned with the back-end, the front-end and middle-end are not described further.

Up until the back-end, variables have been used to hold values but one task of the back-end is to map these to hardware registers and memory locations. In general, there are not enough hardware registers so that each register is only assigned to one variable, instead variables need to share registers. A variable is assigned to a register and whilst that variable (or rather the value it represents) is still needed the register is allocated to that variable only. The duration a variable is needed is called its lifetime or live range, and the variable is said to be live during this. The amount of live variables with respect to it’s width at a specific point in the program is denoted the register pressure. If there are not enough free hardware registers to hold all variables, variables need to be spilled, i.e. stored in memory. Spilling a variable is usually done by storing it in memory directly after its definition and loading it before each uses. Each load is replaced with different variable, thus the lifetime of the variable is shortened to only small intervals and register pressure is reduced.

The back-end may also perform instruction scheduling in order to increase the performance of the program. Some instructions may have latency, i.e. that their results are only available after a number of cycles and thus unrelated instructions may be executed in the meantime. Depending on which instructions have been previously scheduled, no unrelated instructions may be available and the scheduler might have to insert NOP (no-operation) instructions, effectively stalling the processing unit, and wasting cycles that may have been used to execute instructions. Scheduling unrelated instructions increases the register pres-
2. Theory

Figure 2.1: Example program, not on SSA-form (a) and on SSA-form (b)

sure and thus might lead to spilling. For the architecture investigated in this thesis, another goal of the scheduler is to allocate instructions to the functional units so as to keep them as busy as possible. The classical approaches to code generation and their application to a generalized Bifrost architecture were explored in a report [11]. The relevant parts will be reused in this chapter.

2.1 SSA Form

A program is said to be on SSA-form if each variable is only defined once, and each variable is defined before being used. SSA-from simplifies several compiler optimizations and most modern compilers make use of it today [17]. Converting a program to SSA-form usually involves renaming any re-definitions of a variable and changing the subsequent uses appropriately. As illustrated in fig. 2.1a this introduces a problem when a use can be reached from two different definitions, depending on which control flow path is taken during execution. The solution to this is to introduce \( \phi \)-functions, a function which produces a different variable depending on which predecessor was executed. The same program can be seen in fig. 2.1b now on SSA-form, with \( \phi \)-functions where appropriate.

2.2 Register Allocation

Chaitin described [5] how register allocation process can be modeled as graph coloring. Graph coloring is the problem of, given a graph, coloring it such that no two neighbouring nodes have the same color. His algorithm has been the basis of several other papers, which aim to improve upon it [3][7]. In general graph coloring is NP-complete (given Chaitin’s assumptions) and as such, heuristics have to be used to solve register allocation within reasonable time. Among them are Linear scan [16] and Chaitin’s algorithm, described further below. In some cases we can make assumptions about the graph which makes it possible to color it in polynomial time. This is further described in section 2.2.4.
2.2 Register Allocation

2.2.1 Linear Scan

The basic Linear Scan method is presented in [16]. This version is based on keeping track of which variables are live at the same time at a program point. There is a list with live ranges for all the variables. A live range starts when a variable is defined and ends with the last use. Redefining a variable counts as a new live range. A variable is called active if it’s live at a certain point in execution. A list with active variables and their live ranges is also stored so that interference between variables can be found. The live range list is sorted according to start of the live range and the active list is sorted according to end of live range for efficient computation. The algorithm scans the live range list from start to end and inserts them into the active list. If the program counter has passed the last use of a variable it’s live range is removed from the active list. A spilling heuristic is also presented that is based on spilling the live range with the latest end time which was chosen as it’s fast and produces acceptable result.

2.2.2 Linear Scan on SSA form

SSA form is a common way to structure intermediate representation. In SSA form a variable can only be assigned once and all variables must be assigned before use. The register allocator can exploit these properties to be more efficient. Without SSA a data flow analysis has to be made which often increases compilation time. SSA form implies that live ranges are continuous as variable can’t be redefined. The SSA graph needs to be deconstructed after allocation as usual with parallel copy. [20]

2.2.3 Chaitin’s algorithm

When using Chaitin’s algorithm [5] to allocate registers, each live range is a node in a graph and edges between nodes are added when the two live ranges interfere, i.e. when two variables (virtual registers) are alive at the same time. This graph is called the interference graph. Whether or not two variables are alive at the same time is determined with liveness analysis, usually involving dataflow analysis.

When coloring a graph with $K$ colors, Chaitin’s algorithm works by successively reducing the graph by removing nodes which are known to be colorable, i.e. nodes which have less than $K$ neighbours, and putting them on a stack. If a node has fewer neighbours than the number of colors (i.e. the number of available registers) then even if all neighbours have been assigned different colors, there will still be at least one color left to assign to the node. When all nodes have been removed only an empty graph is left, which of course is colorable. Each node on the stack is popped, added to the graph, and assigned a color. Thus the nodes are added in the reverse order in which they were removed.

If at some stage of this algorithm no more removable nodes, i.e. nodes with fewer than $K$ neighbours, can be found, we choose a node to spill and split up its live range into several very small live ranges. Before each use a load from memory is inserted and after each definition of a variable a store is inserted. After this the interference graph is reconstructed and the algorithm is performed again. Choosing which node to spill has to be done carefully, for performance reasons, e.g. we might prefer to not spill a node with a
2. Theory

Chaitin’s algorithm also includes a coalescing stage where the nodes of the argument and the result of a copy instruction are merged in the interference graph (if they do not already interfere). The new node might have more than K neighbours though, which might make the graph uncolorable. Coalescing without regarding the colorability of the graph is called aggressive coalescing and it is what Chaitin employs [5]. Briggs et al. propose a more conservative approach [3]. George & Appel believes this to be too conservative and propose a solution that performs a more aggressive coalescing [7].

2.2.4 Graph coloring on SSA form

Hack et al. [10] describe a novel way to perform graph coloring register allocation. If we can assume the code is in SSA form as described by Cytron et al. [6], we can utilize the fact that the interference graph of an SSA-program is chordal. Since the time it takes to color chordal graphs is quadratic, the time it takes to find the chromatic number, the number of colors needed to color a graph, is also quadratic.

Hack proposes an algorithm where all the necessary spilling is done first by computing the chromatic number for the graph and then, if it is not already colorable with the available registers, the necessary variables are spilled using some heuristic (since the problem of spilling is NP-complete). Here Hack uses Belady’s MIN algorithm [1] for basic block spilling but others may be used as well. After this stage the interference graph is colorable according to an order defined by the dominance relation of the program, which is proven to be an optimal coloring by Hack.

After the coloring phase, coalescing is performed using $\phi$-functions. Hack describes an approximation to optimal coalescing (since this also is NP-complete): Maximize the number of $\phi$-operands that share color with the destination of the $\phi$-function. This will minimize the number of moves this $\phi$-function will result in but does not guarantee optimal coalescing in the global sense. This does not change the colorability of the interference graph - no additional register demand arises and thus no iteration is needed (as might be the case with Chaitin’s and it’s derivatives) and the algorithm terminates.

Pereira F. published a technical report [14] detailing the implementation of a register allocator on SSA-form. It differs from Hack’s on several points but it retains the same basic structure with pre-spilling, coloring and coalescing. The main difference is that Pereira’s algorithm allows $\phi$-virtuals to be mapped to memory, giving the compiler extra freedom to choose values to spill, but requires a more complex $\phi$-deconstruction algorithm. Other differences include: the global register coalescing heuristics, the handling of pre-colored variables and the spilling heuristics.

2.3 Instruction Scheduling

Instruction scheduling is the process of rearranging the order of instructions to minimize pipeline stalls, which is usually done by distancing a producer and a consumer of a value as much as needed. Finding an optimal schedule is NP-complete [17].
When doing instruction scheduling the algorithm has to address two main issues. First one is to consider how to express the constraints so that the result is correct, i.e., any dependencies between instructions have to be taken into account. Then the algorithm must determine how to order the instructions while bounded by the constraints [8].

Instruction scheduling is most often run before register allocation in order to not burden the scheduler with more constraints. If registers had already been allocated, the scheduler would have less freedom in how the instructions could be reordered. When scheduling is done before register allocation, care has to be taken as to not incur too much register pressure. Too much register pressure would lead to difficulties coloring the interference graph and in turn would lead to more spill code and possibly a performance decrease compared to if no scheduling had been done. Scheduling is sometimes done both before and after register allocation as well, in order to optimize any spill code generated during register allocation [18].

### 2.3.1 List Scheduling

List scheduling is a technique to schedule instructions within a basic block [17]. Firstly a dependency graph is built, in which the nodes are instructions and the edges between them determine scheduling order. The source node (instruction) must execute before the target of the edge. When the graph has been built there are one or more instructions that have no dependencies, e.g., instructions that depend on instructions from a previous basic block or on program input. These are the candidates, one of which will be the next instruction to be scheduled. Since optimal scheduling is NP-complete, heuristics are employed to choose which of the candidates to schedule, one of which will be detailed further below. Scheduling can be done either top-down or bottom-up.

When one of these candidates has been scheduled, instructions dependent on it might become available as well (since their dependencies have been resolved) and if so, they are added to the set of candidates. The process repeats until there are no more candidates and the algorithm terminates.

Gibbons et al. [8] present a heuristic that picks an instruction that is not dependent on the previous one. If there are multiple choices an instruction that has many other instructions dependent on it should be picked. This heuristic biases towards giving the scheduler many options.

### 2.3.2 Modulo Scheduling

Modulo scheduling, as described in [17], is a variant of software pipelining. The idea is to interleave loop iterations to improve pipeline utilization, i.e., the next iteration is started before the current one has finished. This is achieved by the compiler attempting to find a modulo schedule of a loop with the lowest initiation interval (II). The initiation interval is the number of clock cycles between the start of two consecutive iterations. Whether the compiler can find a more efficient schedule depends on data dependencies and hardware resources. If an iteration depends on data from the preceding iteration or all hardware resources are fully utilized in the original loop, a more effective schedule might not be possible. In other words both instruction level and loop level data dependencies are relevant for modulo scheduling.
2. Theory

The algorithm begins by computing the bounds of the possible values of the initiation interval. The maximum bound is the execution time of a regularly scheduled loop body. The minimum is computed by taking into account the available hardware resources, required resources and data dependencies on both instruction and loop levels.

The algorithm then iteratively, beginning from the lower bound, computes whether or not a schedule can be made with the current II. If it can, the loop is copied and scheduled as such, if not the next II is evaluated. Prologue and epilogue are also generated if a new schedule is found, which consists of instructions not being part of the new loop, i.e. the iterations currently in progress after exiting the new loop (epilogue) and the first instructions which do not constitute a complete iteration (prologue).

Modulo scheduling may also employ Modulo variable expansion and Heirarchical reduction [12]. Modulo variable expansion is the process of duplicating variables that are used exclusively within one loop iteration (but names are shared across iterations). These variables are only used to hold temporary values and are written to and read from during the same iteration. In order to lessen the constraints on the scheduler, they can be duplicated and renamed for each concurrent iteration in the new loop body.

Hierarchical reduction enables modulo scheduling to optimize loops containing any kind of conditional statement. When modeling instruction dependencies a graph is often built containing information about instruction level data dependencies and loop level dependencies. Nodes are instructions and edges are constraints on execution order. Hierarchical modeling schedules the innermost constructs, e.g. nested loop or if-statements, first and adds them to the graph as new nodes. They become essentially equivalent to instructions, with their own dependency edges to other instructions. They can now be treated as any other node (instruction) in the graph and the outer construct can be scheduled. In other words, loops containing branches are scheduled bottom up where each inner construct, when it’s contents have been scheduled, becomes a node in the other constructs dependency graph, which can be scheduled when all of it’s inner constructs have been scheduled.

2.4 Previous work

Some work has been done on combining instruction scheduling and register allocation, among others by D. Bradlee, S. Eggers and R. Henry in their paper Integrating Register Allocation and Instruction scheduling for RISCs [2]. In this paper, three representative methods for solving both instruction scheduling and register allocation were presented. The first is the classical approach: separate instruction scheduling and register allocation, where scheduling is done after register allocation, called Postpass. The second strategy consists of invoking the scheduler before register allocation and ensuring that it never schedules outside a local register limit (IPS, Integrated Prepass Scheduling). Thirdly a novel technique they call RASE (Register Allocation with Schedule Estimates) where the register allocator is augmented with cost estimates that enable it to quantify how the register allocation will affect the subsequently generated schedule.

They found that the two latter strategies consistently outperformed the former, showing that coupling instruction scheduling and register allocation is beneficial. Another conclusion is that RASE, where instruction scheduling and register allocation are the most cou-
pled, only yield marginal improvements over the IPS and thus the increased complexity and compilation expense of RASE might not be worth the performance gain. The authors do concede that this may change over time as basic blocks grow larger though. The insights that may benefit our report is that, judging from this paper alone, a too tightly coupled register allocation and instruction scheduling might be too expensive for the runtime compilation environment that is GPU programming, but further testing is required.

Register Allocation

The results of Poletto and Sarkar [16] in their original paper on linear scan are that it is "significantly faster" than graph coloring algorithms and that it in general produces code that is within 10% of the performance of code generated by an aggressive graph coloring algorithm. Furthermore they state that it should be well suited for environments with runtime compilation. Linear scan on SSA form [20] by Wimmer et al. makes for an even faster compilation time while still retaining the performance of the generated code, making it even more suitable for the environment of GPU programming.

As for graph coloring algorithms, F. Pereira and J. Palsberg implemented a graph coloring algorithm exploiting the properties of chordal graphs [15]. Their compiler had the ability to handle both chordal and non-chordal graphs and was successful due to the fact that many interference graphs have this property (at least in the Java 1.5 library implementations). S. Hack mentions their implementation in his paper on SSA form graph coloring [10] and comments that since they do not require the input code to be on SSA form, thus not requiring chordal graphs, they cannot fully exploit the advantages of SSA form in graph coloring. The compilation time of Pereira and Palsbergs register allocator is similar to that of the Iterated Register Coalescing of George & Appel [7], which in general is not regarded as viable for runtime compilation. Pereira has also implemented a register allocator that does require the input program to be on SSA form, which is described in a technical report [14] in which he is the sole author. See section 2.2.4 for further details on this compiler.

Hack mentions in his PhD thesis [9] that his SSA based graph coloring register allocator might be usable in JIT (Just-In-Time) compilation environments, as there are potential compile time benefits. This is due coloring order given by the dominance relation and thus the creation of the interference graph, which is usually time consuming and thus unsuitable for JIT environments, does not need to be performed. This is what Pereira and Palsberg fail to exploit since they do not require chordal interference graphs.
2. Theory
Chapter 3

Approach

A compiler framework was provided with a minimal back-end implementation which included infrastructure for Intermediate Representation (IR), partial support for instruction selection and code emission together with basic register allocation. It could compile minimal shaders (programs written in GLSL for a programmable graphics pipeline).

Our work consisted of firstly extending this back-end to support real world shaders in order to create a test suite that ensures our modifications are valid and produce correct code. This included a register allocator that could handle instructions that write to several registers, proper spilling etc. Secondly, various strategies for register allocation and instruction scheduling were implemented, replacing parts of the original back-end, in order to see whether or not modifications enabled the compiler to produce code that performed better. The structure of the back-end is as follows:

**Instruction selection**  Converts from the intermediate representation (IR) used in the previous parts of the compiler to that of our back-end. E.g. data structures for representing instructions and basic blocks.

**Pre-scheduling**  This step is optional and one implementation has been attempted, described further in section 3.1. It aims to minimize instruction count, maximize clause length and keep the execution units (ADD and FMA) as busy as possible.

**Register allocation**  Two attempts have been made: One with wide virtual registers based on [14], an SSA-based register allocator described in section 2.2.4. The other is developed by the authors from scratch and is also SSA-based. The former is detailed in section 3.2 and the latter in section 3.3.

**Post-scheduling**  Up until now the input program has been in the form of basic blocks and instructions, but the bifrost architecture is based on clauses, and has two execution units (ADD and FMA) which are assigned instructions by the compiler as part of the scheduling. This step creates instruction pairs, trying to reduce the number of
NOP (no-operation) as much as possible, and inserts these pairs into clauses. There is one implementation, that only optimizes to a small extent, since it mostly preserves the instruction sequence as received from the register allocator. It is further detailed in section 3.4.

**Code emission** Converts the internal data structures to a file in assembly format together with the relevant metadata.

Furthermore the register allocator has several other constraints to take into account:

- Predefined registers for parameter passing
- Predefined registers for return values
- ISA-related constraints
- Instructions that need to write to a register that it reads from as argument
- Some instructions may write to or read from several consecutive registers

For the last item, the two approaches to register allocation use different ways to represent instructions that write to several registers. The one based on Pereira’s work (section 3.2) uses wide variables, i.e. each variable has a size equal to that of the number of hardware registers it occupies. In the other approach (section 3.3) variables always occupy one register and thus instructions that write to several registers define several variables. Instruction scheduling is less concerned with how one represents instructions that write to several registers, as long as dependencies between instructions are modeled correctly.

### 3.1 Pre-scheduler

There are some differences between schedulers in compilers for a GPU and a CPU. GPU workloads have massive parallelism so that memory latency can be hidden by switching threads. This means that removing pipeline stalls from memory access is not a goal of the scheduler but it is often one of the goals for a CPU scheduler. On the other hand the Bifrost pipeline have constraints that the scheduler must have in mind to generate effective code. There are two different functional units and the scheduler must fill them to fully utilize the hardware.

The scheduler is using SSA-form and is based on a list scheduler because the benefit of modulo scheduling would not be worth the additional complexity. After higher level optimizations have been done the intermediate representation doesn’t show the need for modulo scheduling for our benchmark suite. Modulo scheduling would also be less beneficial on Bifrost compared to a classic RISC pipeline.

Scheduling is done top down for each basic block and begins with setting up dependencies for each instruction. There are two important dependencies which are defined operands and usage left. Defined operands are used to track when an instruction has all its operands defined so that it is ready for scheduling. Memory dependencies can be modeled as a pseudo operand in the scheduler to enforce the memory access order. Usage left is calculated on basic block level to track when a virtual register can be deallocated. This
enables the scheduler to keep track of the live variables and current register pressure. Every time an instruction is scheduled, the scheduler inserts it into the model to support the heuristics for choosing instruction. Instructions are scheduled by keeping a single list of instructions with all the dependencies fulfilled. A score is calculated for every instruction based on information from the model and some other heuristics.

![Figure 3.1: Example of scheduling](image)

In this fictional example mul and div can only be executed in FMA unit while add and sub can only be executed in the ADD unit. For the figures in clauses the instructions are put in instruction pairs with the format "x-unit" where x is the pair index and unit, which functional unit. A constraint in this example is that the result of a div is not available in the same instruction pair which would make scheduling mul first sub-optimal. The downside with the scheduling is that d can’t be in the same register as a and b which may increase register pressure.

### 3.1.1 Model

The scheduler uses a model of the hardware to be able to generate efficient code. The model keeps track of how long the clauses are, if an operand can be read as a bypass and other constraints. Because of the constraints the model is necessary for efficient code generation. Without a model the scheduled instructions may not work in the pipeline directly and sub-optimal fixes may be necessary. Because of the two distinct arithmetic functional units the model keeps track of which unit that is available to fill both units with useful instructions.

### 3.1.2 Heuristics (priority function)

Two important considerations for the scheduler are to fill the functional units and minimize register pressure. Filling functional units would benefit from having many instructions ready so that the probability of finding one that fits the pipeline is high. The problem of keeping many instructions ready for scheduling is that it often increases register pressure. This problem is solved by focusing on filling functional units when the register pressure is low and try to lower register pressure when register pressure is high. Some heuristics for lowering register pressure are the following:

- Prioritize instructions that kill live variables.

---

21
3. Approach

- Prioritize arithmetic instructions over memory load.

- Prioritize predecessors of successors to instructions that have been scheduled see fig. 3.2

![Figure 3.2: When 0 is scheduled 1 will be prioritized](image)

There are also some Bifrost specific heuristics like maximize bypass usage and following constraints in Bifrost. If there are constraints that are not fulfilled, NOP may need to be inserted which lowers performance. Some of the heuristics for filling the functional units are:

- Prioritize instructions that can be scheduled only in the current unit.

- Prioritize instructions that many instructions are waiting for.

- Count the number of instructions that could be executed in each unit and schedule instructions that can be executed in both, in the unit with less schedulable instructions.

3.2 Live range-based register allocation

The register allocation algorithm used is that of Pereira F. described in [14]. It is a SSA-based register allocator that uses the live range representation commonly found in linear scan allocators and is further detailed in section 2.2.4. It has been extended to support the register allocation constraints in the following way:

- An instruction that only uses part of a wide variable as input, uses a sub index to access the specific register, e.g. a wide variable of size two can be accessed with either sub index zero or one.

- A wide variable live range ends with the last of its uses, i.e. it cannot partially die, which means that registers allocated to a wide variable cannot be reused before the end of the wide variables’ lifetime.

- The spill phase does not take into account that wide variables need to be consecutive, it only spills on account of total registers usage. When the wide variables are to be colored, there are enough registers available, but not necessarily enough consecutive registers.

- Instructions reading from more than one consecutive register are implemented as reading from a preceding merge node, further detailed in section 3.2.2.
3.2 Live range-based register allocation

3.2.1 Shuffle insertion

In order to handle the constraints placed on a register allocator as listed in chapter 3, shuffle code is inserted before the instruction to free the required registers (essentially a register file permutation). Mohr et al. [13] use a shuffle instruction to implement $\phi$-functions and the same concept can be reused here. An instruction that shuffles registers is defined as $ra \rightarrow rx, rb \rightarrow ry$ which copies the contents of register $ra$ to $rx$ and $rb$ to $ry$. It may also include swaps $rx \leftrightarrow ry$ and circular dependencies $rx \rightarrow ry, ry \rightarrow rz, rz \rightarrow rx$. Whilst Mohr et al. implements a shuffle instruction as a register file permutation hardware instruction, it usually needs to be implemented as several move or swap instructions in a similar way to $\phi$-deconstruction. If the hardware does not support swap instructions, a swap can be implemented in the classical way with three XOR instructions. From here on it will be assumed that the more moves a shuffle instruction contains, the more expensive it will be in terms of computation cost and furthermore, swaps are more expensive than moves.

An issue with inserting shuffle code to handle the register allocation constraints is that the register values need to be reshuffled in order to preserve program correctness. To illustrate this, an example program which has been partially colored can be seen in fig. 3.3. The notation $a^{(r)}$ is used to signify that a variable $a$ has been allocated to $r$. This example showcases the implementation regarding wide variables but the same techniques are used for the rest of the constraints.

The register allocator has assigned register $rx$ to the variable $a$ and $rx+1$ to $b$. When it arrived at the definition of $c$ in basic block 1, no two consecutive registers were free to hold $c$, so $rx$ and $rx+1$ were freed by moving their contents to $ry$ and $rz$, which are free for the lifetime of $c$. In order for the uses of $a$ and $b$ in basic block 2 to use the correct values, the contents of $rx$ and $rx+1$ need to be shuffled from $ry$ and $rz$ before those instructions are executed but only if the original shuffle has been encountered during execution. Two approaches to ensuring that all registers get the correct values and that reshuffles are inserted appropriately are:

1. Reshuffle directly after the instruction
2. Reshuffle right before entering the dominance frontier of $c \leftarrow def<2>$, i.e.
3. Approach

Figure 3.4: Fully colored program, approach 1 (a) and 2 (b)

dominance frontier of basic block 1 (see either [17] or [6] for more information on SSA form and dominance relations).

The result of the first approach can be seen in fig. 3.4a and the second approach in fig. 3.4b. As can be seen, the first approach always generates swap instructions whilst the second approach only generates move instructions if needed, i.e. in the example program, if $b$ is still live as exit the dominance frontier of basic block 1. Furthermore if $c$ was defined in basic block 0 or basic block 2, then no uses of $a$ or $b$ would outside the dominance tree and no reshuffles would need to be inserted. If the a program is extended from this simple example to include loops, the second approach would still be favorable since for each execution of the original shuffle only one of the reshuffles would be executed. This is since we never enter into the a dominance tree, if we have exited it, without traversing the root node of the dominance tree. This follows from the definition of a strict program as defined by Budimlic [4].

With both of these approaches, live ranges need to be updated accordingly. This is since the register allocator determines which hardware registers are available by comparing the live ranges of hardware registers with variables, if they are disjoint, the hardware register is available. Thus it is vital to update live ranges correctly, otherwise registers might not be available when they ought to be. For the first approach the live range bookkeeping is fairly simple. The register $r_y$ and $r_z$ have the same live range as $c$ and the live ranges of $r_x$ and $r_{x+1}$ remain unchanged. It becomes slightly more complex with the second approach as $r_y$ and $r_z$ should only be live for as much as needed during the range of the shuffle. For $r_y$ and $r_z$, the live range is computed as $\text{live range}(r_y) = \text{live range}(r_z) \cup (\text{live range}(r_x) \cap \text{shuffle range})$ where shuffle range is the linearised live range for which start with the original shuffle and ends with each reshuffle. As for $r_x$ and $r_{x+1}$, $\text{live range}(r_x) = (\text{live range}(r_x) - \text{shuffle range}) \cup \text{live range}(c)$. The subtraction is needed to remove all the live range segments of $a$ and $b$ within the shuffle live range.
3.2.2 Wide variables as arguments

A merge node takes several variables as arguments and produces as variable of a size equal to the sum of the sizes of the arguments like, \( w \leftarrow \text{merge}<2> a, b \). Thus a merge with two arguments, both with a size of one, will produce a variable of size two. When the register allocator reaches a merge instruction, it is treated similarly to any instruction that writes to several registers. In the case of a merge though, shuffles are only inserted if the arguments of the merge do not match up to the assigned registers. For example, a merge takes four arguments of size one that have been assigned to hardware registers 4, 5, 16 and 7. If the register allocator assigns 4, 5, 6 and 7 to this merge node, then only a \( r_{16} \leftrightarrow r_7 \) needs to be inserted.

When coloring any variable, a preference list listing all possible hardware registers ordered from most preferable to least preferable for this instruction is created. The purpose of this list in Pereira’s compiler [14] is to allow opportunities for \( \phi \) register coalescing, i.e. choosing a register for the \( \phi \) node that matches one or several of its arguments. This preference list can also be used to reduce the number of instructions generated by the merge.

When the preference list for an instruction \( i \) is generated, a check is done if any merge instructions read from \( i \). If \( i \) is the first argument to this merge (i.e. if this is the first time this merge is visited as a use during preference list generation), then a bias register is reserved for this merge. The next time this merge is visited as a use of an instruction, the register allocator tries to allocate a register, using the bias, such that the merge will result in as few instructions as possible when it is colored.

3.3 Coupled register allocator

This approach scalarizes all nodes in the internal representation. This means that loads that fetch to more than one register will get multiple virtual registers assigned to it. Stores that store more than one register need to keep track of all the virtual registers they store and their internal order.

Another difference of this approach is that it’s tightly coupled with the scheduler. Metadata from the scheduler is used to keep track of lifetimes instead of calculating live ranges. This means that this approach can be used for scheduling and allocation in unison. There are however some practical benefits of doing scheduling and register allocation sequentially. Some benefits is of doing it sequentially are that the scheduler can be used with the wide virtual registers approach. It is also a bit simpler to implement and debug the code when they are separated. Some parts of the text in section [3.3] are specific for when the allocator is run after the scheduler, modifications needed for running those two simultaneously will be described in section [5.2].

3.3.1 Spilling

When instructions are scheduled an interference graph is constructed if there are more live variables than registers. This graph is the same as the graph in Chaitin’s[5] graph coloring algorithm but is only constructed when register pressure is too high. The spilling
3. Approach

Implementation also keeps track of intervals that need spilling. This makes it possible for uses of a spilled variable to use the same reload if it’s outside a high register pressure interval. Registers are spilled as 32 bit scalars but it’s possible to schedule spills to leverage wide memory instructions.

3.3.2 Coloring

An important feature of the coloring algorithm is the ability to support data that must be in adjacent registers. If virtual registers can’t be allocated to adjacent physical registers directly this pass will insert shuffles to correct it. The algorithm colors basic blocks in reverse post order and goes top to bottom within a basic block.

The current physical register a virtual register occupies will be called effective register. Every basic block has a data structure that maps every virtual register to its effective register which enables shuffling around physical register effectively. If a virtual register is live outside of a shuffles dominance tree the register must be shuffled back to ensure correctness. With the effective register system, back-shuffles can be inserted as a shuffle from a virtual register to a physical register.

**φ**-functions are deconstructed in the coloring phase by inserting a shuffle from the argument variable to **φ**-function’s physical register. If the **φ**-function does not have a physical register it will be assigned the same as one of its operands physical register. If the effective register is the same as the destination of a shuffle the shuffle will simply be ignored. When the effective register is different from the physical register moves or swaps needs to be inserted depending on if the physical destination is holding a live variable.

One property of reverse post order is that all predecessors excluding loops are colored before the current basic block. This means that all operands from other basic blocks that are not used in a **φ**-function will already have a color. Because of this property, the effective register can be propagated by setting the same effective register in the successors where the correspondent variable is live.

Look ahead

To minimize unnecessary shuffles the coloring algorithm will check if the a variable has a use with constraints. If a use is a wide store the algorithm will try to allocate a register that have as many free adjacent registers as the store is wide. The algorithm will also look if a **φ**-function uses the variable and would preferably chose the same physical register to avoid shuffling.

3.4 Post scheduling

The Bifrost architecture expects assembly code in the form of clauses and instruction pairs. The task of this stage is to schedule instructions in the FMA or ADD unit and create clauses with these scheduled instruction pairs. Each basic block is divided into several clauses and a clauses never contains instructions from two different basic blocks.

Barely any optimizations are done in this stage, it is assumed this is done in the scheduler before register allocation. This stage also handles the various constraints of Bifrost.
instruction encoding, such as which instructions need to be passed constants through reg-
isters and which can use them directly as arguments etc. It tries to pack the instructions
into as few pairs as possible and rearranging of instructions is only done within a pair,
e.g. if an instruction is scheduled in the ADD unit, and the following instruction may be
executed in the FMA unit, provided it is not dependent on the result of the ADD, it is
scheduled in the FMA unit.
Chapter 4

Results

The test suite consists of 228 shaders that our back-end can compile and the generated code produce correct output. In order to be displayed, the tests have been grouped according several categories to differentiate how the back-end performs for different types of shaders. Firstly, they may be grouped according to size (number of instructions directly after instruction selection), the size limits for each group can be seen in table 4.1. Furthermore, the test cases are also be divided into shaders that include 16 bit computations and those that only perform 32 bit computations. Also, they may be divided into types of shader: fragment shader, compute shader and vertex shader, which correspond to shaders for different parts of the graphics pipeline.

The following metrics are used for the evaluation of the back-end:

**Executed instruction pairs** Gives a rough estimate how how many cycles that the shader runs for

**Number of instructions** Statically counted number of instructions

**Number of clauses** Statically counted number of clauses

<table>
<thead>
<tr>
<th>Table 4.1: Limits for size categories, number of instructions after instruction selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>category</td>
</tr>
<tr>
<td>very small</td>
</tr>
<tr>
<td>small</td>
</tr>
<tr>
<td>medium</td>
</tr>
<tr>
<td>large</td>
</tr>
<tr>
<td>very large</td>
</tr>
</tbody>
</table>
4. Results

The reason why not, for example, the run time of the shader program was used as metric is that it was not deemed worthwhile to start optimizing for execution time from the start. Instead focus was on reducing the number of executed instruction pairs, and the plan was to start comparing execution time when the difference in the number of executed instruction pairs was low enough. Unfortunately, due to time constraints, this was never done.

Whilst the number of clauses and the number of instructions do not give a exact measure of performance, they can still be used to compare the effectiveness of code generation compared to the reference compiler which is a compiler developed at ARM. Some metrics may be presented as normalized with regards to the reference compiler, e.g. if the code generated by our back-end results in 30 executed instruction pairs and the same shader compiled by the reference compiler results in 20 executed instruction pairs, then the displayed value would be \( \frac{30}{20} = 1.5 \).

4.1 Metrics

As can be seen in fig. 4.1, there is a significant improvement in the number of executed instruction pairs using the pre-scheduler. To evaluate why the scheduler produces faster code, a static count of instruction types is done. The result have been divided into Non-NOP, NOP and spill related instructions. The measurements are done with the live range based register allocator to isolate the impact of the scheduler.

With pre-scheduling the backend can generate code for all the shaders without spilling even for very large shaders. If pre-scheduling is disabled spilling is needed for very large shaders and the negative impact is considerable. The difference in NOP instructions is mostly because the scheduler tries to interleave instruction in both functional units while fulfilling architectural constraints.

The cause for the difference in non-NOP instructions for small shaders have to do with architectural constraints that require non-NOP insertions. The scheduler tries to fulfill those constraints to minimize inefficient fixes in the post scheduler. For larger shaders, instructions used for shuffling may also contribute to the difference in non-NOP instructions.

As for the two register allocators, the results can be seen in fig. 4.2. There is a very small difference in the number of executed pairs between the two, in favor of the coupled allocator, but when compared to the reference compiler, they are largely equal.
4.1 Metrics

**Figure 4.1:** Static instruction count with and without prescheduler

**Figure 4.2:** The number of executed instruction pairs for each register allocator with pre-scheduling enabled for both allocators
4. Results
Chapter 5
Discussion

In general, the two approaches to register allocation are equal but as can be seen in fig. 4.2 there are slight differences. These can be seen more clearly in Appendix B and are partly due to the coupled allocator producing less shuffle code. It has implemented more optimizations toward register coalescing with $\phi$-functions (i.e. bias for the coloring of the $\phi$) which the live range allocator only implements for wide variables. In the latter, the registers allocated to a wide variable are allocated for the entire live range of the wide variable, whilst in the former the registers which are no longer needed, may be used by other instructions. This may increase register pressure which in turn might create the need for more shuffle code. The difference is not due to one of the allocators spilling though, since with the pre-scheduler, neither of the two do any spilling.

The graph in Appendix A shows that the difference in performance between our back-end and the reference in the number of executed instruction pairs is related to the difference in the number of instructions (statically counted) generated by the two compilers, which is confirmed when examining the differences in the generated code as well. Essentially, the instruction selection of our back-end does not measure up to that of the reference compiler. The difference in the number of executed instruction pairs is not merely the product of instruction selection differences thoughs since, as can be seen in Appendix A, the number of NOPs in the generated code differs as well. This is a product of the scheduling algorithms (both pre- and post-scheduling).

5.1 Coupled register allocator

The coupled registers allocator was designed with the following requirements in mind:

- There are wide memory instructions that require adjacent registers.
- There are hardware benefits from using bypass registers within a clause.
5. Discussion

• Other constraints in the hardware that need to be addressed.

Splitting a wide memory instructions into multiple variables can lower the register pressure. Uses of parts in a wide memory load may be far apart and it’s unnecessary to keep a part that won’t be used live, see fig. 5.1. One downside of splitting variables is that it increases number of variables and may affect compile time negatively.

1: \( a \leftarrow \text{def}<2> \)
2: \( \leftarrow a[0] \)
... 
x: \( \leftarrow a[1] \)

**Figure 5.1:** \( a[0]\) don’t have to be live after 2

Because of those wide memory instructions it’s sometimes necessary to shuffle registers so that the required registers can be adjacent. The "effective register" approach is designed to simplify shuffling. With it, shuffles are defined as moving a variable to a physical register. This enables the compiler to shuffle registers without doing complex calculations when a variable is shuffled multiple times. It is also a causal system that can be done without knowing the future which is a prerequisite for doing register allocation with instruction scheduling.

The choice of letting the scheduler set up information of when a variable can be killed is also done so that the register allocation can be done simultaneously. Much focus has been put on the ability to allocate register with instruction scheduling to build clauses and use bypass registers. The register allocator can be seen a just-in-time linear scan allocator with support for wide memory instructions.

Spilling can be avoided in our test suite with the pre-scheduler so this approach didn’t put much effort in optimizing spilling. It can output correct code for all the tests that spill without the pre-scheduler but performance may suffer.

5.2 Unified Scheduling and Register Allocation

Scheduling and Register allocation can be done in unison with the coupled register allocator. The post scheduler is also an integrated part of this approach. When an instruction is scheduled, a register will be allocated and the instruction will be inserted in a clause immediately. There are three parts that are executed every time an instruction is selected for insertion. The scheduler will first choose a instruction based on a modified heuristic with register continuity in mind. After that the allocator will assign registers and if shuffling is needed the shuffled code will be inserted before the instruction. A post pass will then be done after every insertion to make sure that all constraints are fulfilled. The post pass will create instruction pairs and clauses directly so that the model will mirror the current state accurately. The spilling algorithm for register allocation after scheduling won’t work now but possible solutions will be described later.

Implementation of this unified code generator is unfortunately not mature enough to produce efficient code yet. It can run all of the shaders used in our benchmark suite but
the instruction pairs count is around 10% worse. There are however lessons learned about the advantages and disadvantages with a combined code generator.

After some experimentation with this unified approach it was noted that the increased freedom and model precision was too much for our limited time frame. The heuristics became quite complex and conflicts in the heuristics sometimes produced code with lower quality. The additional freedom has the potential to produce better code but the cost in developing heuristics and compilation time may be high.

Wide memory instructions

One disadvantage of doing register allocation and instruction scheduling simultaneously is that the register allocator cannot look ahead to see how long a variable is alive. In fig. 5.2 at cycle 1 a will be allocated to \( r_0 \) and \( r_1 \) will be reserved for \( b \) because of the store in cycle \( z \). Without knowing when \( c \) will die the register allocator won’t allocate \( c \) to \( r_1 \) which may be the optimal solution. The advantage on the other hand is that the scheduler can actively optimize the problem by killing problematic variables. The register allocator can also see if a variable is ready to be killed by looking if it’s last use is ready for scheduling.

```
1: a ← def<1>
2: c ← def<1>
...
x: ← c
y: b ← def<1>
z: ← a, b
```

Figure 5.2: \( c \) can be allocated to \( b \)’s reserved register

Spilling

Spilling benefits from knowing interference between variables to do a relatively optimal spill. It is impossible to know how the interference for instructions that haven’t been scheduled yet, which may be a problem. It may also be better to insert spill in a position that already has been scheduled. If that is done the spill will probably be less optimal compared to scheduling the spill in a post scheduler. The benefits may include that reloading a spill can be scheduled more optimally and that spill may be avoided with use of bypass registers. The simple way is to spill a variable directly when the register pressure gets too high. But that approach may spill variables inside a loop which is sub-optimal. It’s possible to make better spills in other places of the code but that scheme may be quite complex because of different constrains.

Heuristics

When scheduling is done in unison with the register allocation the heuristic can be made more optimally and more complex. Because clauses are created directly, the model is no longer an approximation so the heuristic can be much more precise. This will for example make it possible for the scheduler to optimize for long clauses. As discussed before the
schedulers heuristic may now consider register allocation problems and the allocators look ahead may look into the scheduler.

More efficient code could be generated if the scheduler scheduled a few instructions at a time because some constraints that depend on following instructions could be optimized. This may increase the compilation time depending on how many instruction in the future that will be evaluated.

Conclusions

Doing Register Allocation and Instruction Scheduling in unison gives a lot of freedom and a better model. The downside is the increased complexity and problematic spilling. The improved heuristics and spilling haven’t been implemented yet and are candidates for future work.

5.3 Live range based register allocator

As mentioned above, there are a number of points to improve on the live range based register allocator,

- Faster compilation time
- Better optimizations for shuffle code reduction (probably bug)
- \( \phi \) bias for coloring

The first point could be addressed by improving how the instructions access register arguments that may have been shuffled as described by section 3.2. The current implementation is based on the live ranges used to represent instruction live ranges, and when the arguments of an instruction are to be hardened into their hardware register equivalents, a look up is done to find if an instruction is within the region defined by a shuffle. These regions may be layered, which further increases the time it takes to find the correct register. Furthermore, the creation of the preference list used for register coalescing is relatively unoptimized as well.

The general evaluation for this allocator is that, at least for the shaders comprising the test suite, it is not necessary for the register allocator to be complex as long as the pre-scheduling lowers the register pressure enough. As for future work, something interesting to investigate would be allocating register to variables in decreasing size, ensuring that wide variables are always allocated to wide registers and less shuffle code would be needed.

5.4 Post scheduling

In the post-scheduler, an opportunity for optimization is with regards to clause length. As stated in section 3.4 the post-scheduler assumes that the instructions are already scheduled well and merely attempts to pack into as few instruction pairs as possible. The issue with this is that there are some instructions in the Bifrost ISA that need to be in the last instruction pair of the clause and others which result is not available before the next clause
executes, i.e. their result cannot be used before the clause has finished executing. At the moment, the post scheduler handles these types of instructions in the same way: they both, when scheduled, force the scheduler to create a new clause in which the rest of the instructions may be inserted to. For the instructions that do not need to end a clause but for which the result is not available in the current clause, unrelated instructions may be inserted to increase the length of the clause. This would require dependency analysis, which the post-scheduler does not have.

The post-scheduler might also benefit from rearranging instructions as well, so as to not depend on scheduling done by the pre-scheduler, since it cannot schedule instructions introduced by the register allocator. For the shaders used in the test suite, no spilling was done but some shuffle code was introduced so the register allocator does not introduce large amounts of code and whether or not it would result in large gains to include instructions rearranging in the post-scheduler is hard to determine. In the general case, especially if the register allocator inserts much spill and shuffle-code, instruction rearranging in the post-scheduler would definitely be beneficial.
5. Discussion
Chapter 6
Conclusions

In this thesis multiple approaches have been presented for back-end code generation and their respective pros and cons evaluated. We found that a pre-scheduler can lower the register pressure considerably which makes a major impact on part of our tests. Furthermore the pre-scheduler can also increase the utilisation level of the hardware by executing instruction in a more optimal way.

There was no major difference in the quality of the generated code between the two register allocation algorithms for our test suite. Generated register shuffling for both approaches was minimal and was not responsible for any significant portion of the difference in executed instruction pairs.

Instruction selection and higher level optimisations made a major impact on the result for some of our tests but does not explain all of the differences in executed instructions. The rest of the differences in executed instructions are probably because of suboptimal scheduling in the pre- and post-scheduler.

6.1 Future work

Improvements can be done for all parts of the back-end presented in this thesis. There is a lot of room for improvement in the pre-scheduler heuristics. Example of improvements include a better balance between register pressure and number of ready instruction to increase utilization of the hardware. The register allocators can benefit from a better coloring algorithm for more complex shaders to avoid unnecessary shuffling and phi deconstruct instructions. Spilling could also be improved as our benchmark only covers the correctness and not the efficiency of spilling. The post scheduler can be improved by allowing it to schedule a basic block instead of only an instruction pair.

It would also be interesting to fully implement combined scheduling and register allocation for comparison. There are a lot of architecture-specific improvements to be made in both pre- and post-scheduler. Those improvements may include scheduling memory...
6. Conclusions

access, fulfilling constraints and minimize the number of clauses. Instruction selection could also be improved with more advanced optimizations.
Bibliography


Appendices
Appendix A

Test suite listing 1

This appendix lists all test cases with the number of executed instruction pairs, number of instructions excluding NOP and number of clauses for each shader, with and without the pre-scheduler for the live range register allocator. They are divided into the 16 and 32 bit categories as explained in chapter [4] for readability.
Figure A.1: With 16bit instructions
Figure A.2: Only 32bit instructions
A. Test suite listing 1
Appendix B

Test suite listing 2

This appendix lists all test cases with the number of executed instruction pairs for the live range allocator, live range allocator without pre-scheduler and coupled allocator.
Figure B.1: With 16bit instructions
Figure B.2: Only 32bit instructions
Kodgenerering för grafikprocessor

Grafikprestanda blir en allt viktigare faktor i köpet av en mobiltelefon, dock begränsas förbättringsmöjligheterna av batterikapaciteten och förmågan att leda bort värme. Hårdvaruföretag pressas allt mer för att förbättra prestandan utan att förkorta batteritiden och generera för mycket värme. Ett sätt att uppnå detta är att flytta logik från hårdvara till mjukvara. Ett exempel för att illustrera logik i hårdvara kan vara att jämföra grafikprocessorn med muskler och mjukvaran med hjärnan. Att ha logik i hårdvara innebär att jag kan tänka ’flytta min hand’ medan logik i mjukvaran innebär att jag måste tänka ’Flytta armbågen så många grader, vrid underarmen och böj handleden etc.’. Kodgenerering handlar om att generera instruktioner i hjärnan som styr armen.

Detta arbete har undersökt kodgenerering för en grafikarkitektur av det senare laget vilket betyder att mycket av logiken har flyttats till mjukvaran. Vi har implementerat traditionella och specialpassade tekniker för kodgenerering och utvärderat dessa.

Kodgenerering kan framförallt delas upp i två steg, instructionsschemaläggning och registerallokering. Målet med instructionsschemaläggning är att utnyttja beräkningsresurser i hårdvaran på ett effektivt sätt. Man kan tänka sig att instructionsschemaläggning är som om det fanns en kö till två automater. Om ena automaten endast tar emot kort och den andra endast tar emot kontanter så vill man att köna ska vara varannan med kontant och varannan med kort för att inte en automat ska stå och vänta.

Registerallokering innebär att placera den mest använda datan i så kallade register (liten lagringplats med snabb access, tänk skrivbord) och resterande i datorns minne (stor lagringplats med långsam access, tänk bibliotek). Då kan registerallokering jämföras med att planera sina besöck till biblioteket så att de blir så få som möjligt och lägga de mest använda böckerna på skrivbordet för att komma åt dem snabbt.

Resultaten som vi kom fram till visar på att det går att göra kodgenereringen mer än dubbelt så snabbt som en referens-kodgenerator. Dock så är koden inte alltid lika effektiv men vi tror på att kodkvaliteten kan förbättras genom att förbättra schemaläggningen. Vi har även sett att schemaläggning kan ha betydlig påverkan på registerallokeringen i båda riktningarna. Traditionellt sätt så brukar instructionsschemaläggning och registerallokering ske sekventiellt men vi har även undersökt ett sätt att göra det samtidigt under vissa förutsättningar vilket kan innebära att kodgenetorats bli bättre. Vi har även funnit att det kan vara en god ide att göra schemaläggning både före och efter registerallokeringen.