MASTER'S THESIS | LUND UNIVERSITY 2016

Investigating Different Register Allocation Techniques for a GPU Compiler

Max Andersson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-23

Investigating Different Register Allocation Techniques for a GPU Compiler

Max Andersson

June 22, 2016

Master's thesis work carried out at ARM Sweden.

Supervisors: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se
Dmitri Ivanov, Dmitri.Ivanov@arm.com

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

Register allocation is one of the most critical parts of an optimizing compiler. Although a great effort has been put into researching how to allocate registers, not much of it has been focused on vector registers. This report seeks to find out what fundamentally new problems arise when allocating vector registers rather than scalar registers, how the previously known problems change in vector registers and what methods can be used to tackle these issues. Furthermore, an attempt to use a combined scheme of register allocation and instruction scheduling is made, to see how it performs with vector registers. This thesis presents the results of an investigation of how two commonly used register allocation techniques, Linear scan and Graph coloring, perform relatively. Furthermore, it presents generalizations to a commonly used algorithm used in graph coloring, Chaitin's algorithm.

Using the internal performance suites of ARM Midgard compiler, our investigation revealed that linear scan can in fact speed up code generation quite significantly, up to 12.5% compared to graph coloring. However, this comes at the cost of reduced code quality. The generalized spill criterion resulted in an significant reduction in spill code inserted, where 10% less spill code was inserted using the derived criterion. This however did not equate to 10% reduction in execution time, although execution time of the generated code was overall decreased by 0.5%. The combined scheme reached comparable efficiency compared to graph coloring, however, since it was only used for single basic block shaders, it is difficult to say how efficient the register allocation would be for larger shaders.

Keywords: Register allocation, vector registers, graph coloring, compiler optimization.

Acknowledgements

First of all, I would like to thank ARM Sweden for offering me the possibility to do this thesis work. I would especially like to thank my supervisor at ARM, Dmitri Ivanov, for invaluable discussions, comments and help throughout the entire thesis work. I would also like to thank my supervisor at LTH, Jonas Skeppstedt, for his enthusiasm, proofreading and support through this thesis.

I would like to thank my family for their support throughout my time at LTH. Lastly, I want to thank Ellen Rieloff for her love and help throughout our time in Lund.

Contents

1 Introduction **7**
- 1.1 Research Questions . . . 8
- 1.2 Contributions . . . 9

2 Background **11**
- 2.1 Compiler infrastructure in general . . . 11
- 2.2 Compiler Back-end . . . 12
  - 2.2.1 Instruction Scheduling . . . 12
  - 2.2.2 Liveness Analysis . . . 14
- 2.3 Register Allocation . . . 14
  - 2.3.1 Graph Coloring . . . 14
  - 2.3.2 Chaitin's Algorithm . . . 15
  - 2.3.3 Chaitin-Briggs algorithm . . . 17
  - 2.3.4 Linear Scan . . . 18
- 2.4 Vector Registers . . . 18

3 Hypotheses **23**
- 3.1 Required Adjustments To Chaitin-Briggs Algorithm . . . 23
  - 3.1.1 Criterion for Trivial Colorability . . . 23
  - 3.1.2 True Criterion For Trivial Colorability . . . 26
  - 3.1.3 Criterion for Potential Spill . . . 28
- 3.2 Subgraph Coloring . . . 30
- 3.3 Linear Scan . . . 36
- 3.4 Combining Register Allocation with Instruction Scheduling . . . 37
  - 3.4.1 Top-down or Bottom-up . . . 37
  - 3.4.2 Avoiding Fragmentation . . . 38
  - 3.4.3 Placement Strategy . . . 39
  - 3.4.4 Fragmentation and Live Range Splitting . . . 39

4 Evaluation Methodology **41**

5 Results and Discussion **43**
5.1 Linear Scan 43
5.2 Graph Coloring 45
5.2.1 True Criterion For Trivial Colorability 45
5.2.2 Criterion For Potential Spill 46
5.2.3 Subgraph Coloring 47
5.3 Combined Scheduling and Register Allocation 49
5.3.1 Top-down or Bottom-up 49
5.3.2 Efficiency of a combined scheme 49

6 Conclusions **51**
6.1 Summary 51
6.2 Future Work 52

Bibliography **53**

Chapter 1
Introduction

A graphics processing unit, or *GPU* abbreviated, is a processor which handles displaying graphics. The reason why this is done on a separate processor, and not the CPU, is because the task of displaying graphics is fairly different from normal computation. First of all, much of graphics can be computed in parallel. Another difference is the extensive use of *vectors*, which motivates the use of *SIMD*-instructions. SIMD is an abbreviation of *single instruction, multiple data*, and simply means that instead of using the two separate instructions *add r0.x, r1.x, r2.x* and *add r0.y r1.y, r2.y*, have a single instruction *add r0.xy r1.xy, r2.xy* [11]. For a GPU, this is practically invaluable, since computer graphics heavily uses vectors for many reasons, e.g. to model the light sources, characters, texture positions, pixel positions etc. Being able to do these operations in a single clock cycle results in a vast improvement of performance.

This have been done with SIMD on e.g. ARM NEON [2] or SSE [11], but a vital difference between those and GPU-vector registers is the use of *swizzling*. Swizzling is the act of modifying which components is read/written in a register, so that we may for instance have instructions such as add r0.x~z r1.y~w, r2.x~x, where the tilde on a component signals that that components is not read or written. In this instruction, we would add the y-value of r1 and the x-value of r2 and store it as the x-value in r0, while simultaneously add the w-value of r1 to the x-value of r2 and store it as the z-value in r0. It is possible to do this in SSE and NEON, however in order to do so, we need an vector permute, which is often an instruction. Swizzling is common in computer graphics, and for this reason, using an instruction to reorganize the registers would be quite expensive. While swizzling may not seem to be a big difference, it allows for much flexibility in how the components can be allocated. Furthermore, it allows for increased optimization properties, as discussed in detail later. Using a SIMD architecture does not only offer improvements. Some issues with using a SIMD architecture are that it typically uses more power and instructions have a higher latency.

Programs that run on a GPU are called *shaders*. Shaders are of course not written in machine code, and thus need to be compiled into machine code by a compiler. A compiler

is typically divided into three stages, called front-end, middle-end and back-end. The front-end parses source code and creates an *intermediate representation*, (IR), which is sent to the middle-end. The middle-end then performs several high-level optimizations on the code, for instance to reduce redundancy. Finally, the representation enters the back-end, which converts the IR into machine code. The back-end is dependent on the target machine and its characteristics while the front-end is dependent on the program language. One of the most crucial parts of the back-end is *Register allocation*, which is the main focus of this thesis.

The range between the first definition of a variable and the last use of it is called its live range. Register allocation attempts to assign each variable a register for the entirety of its live range. There are many ways of allocating registers to variables. One of the most commonly used methods is to construct a graph, called the *Register Interference Graph*, or RIG abbreviated, which one then attempts to color using at most as many colors as there are registers available. Since graph coloring is a *NP-complete* [8] problem, it is not feasible to use an optimal algorithm. Another way of performing register allocation is by linear scan, which simply attempts to order the live ranges in some order allocate them in that order. The clear advantage of this is faster compilation time, but it may result in worse register allocation. There is also a possibility to allocate registers during the *Instruction scheduling* pass. Such a scheme would perhaps prove useful, since it eliminates the need to perform liveness analysis and register allocation after the scheduling. It is also worth mentioning that efforts have been made to use *constraint programming* to achieve optimal register allocation. However, due to the amount of computational time it would take to find such a solution, it is not feasible on a GPU compiler [9]. There is a strong incentive to design architectures with as few registers as possible, since register memory is very expensive. Being able to reduce the amount of registers while not significantly reducing performance is thus very beneficial for hardware design.

Finally, there are different types of compilers, two of which are *ahead-of-time*, abbreviated AOT, and *just-in-time*, abbreviated JIT, compilers. The difference lies in that AOT-compilers compile source code prior to execution whereas JIT-compilers compile during runtime. There are advantages and disadvantages of both types, but the main difference is speed of compilation. Since JIT-compilers compile during runtime, it is vital that they are quick, whereas AOT-compilers are allowed to take longer time to compile in order to better improve code efficiency.

1.1 Research Questions

This thesis aims to investigate how the vector registers in a GPU affect register allocation, and how the current techniques can be improved. Furthermore, other techniques will be investigated to see how they perform. This has motivated the following research questions, which the thesis revolves around:

- How does Linear Scan perform relative to Graph Coloring in a GPU compiler?
- Can the current implementation of Graph Coloring be improved?
- How do vector registers affect the possibility to perform register allocation during instruction scheduling?

- How can fragmentation be avoided when performing register allocation during instruction scheduling? Fragmentation will be formally defined in Section 2.4.

1.2 Contributions

The contributions in this thesis comes in three fields, namely *linear scan*, *graph coloring* and *combining register allocation with instruction scheduling*.

I have compared how the, for GPUs, commonly used technique linear scan performs relative to graph coloring. I have also investigated how different orderings of linear scan perform with vector registers.

Register allocation by graph coloring has been in an ad-hoc manner with vector registers. I present a mathematical generalization of the ad-hoc method for finding trivially colorable nodes. I then prove the generalization to be non-optimal, and formulate a true generalization of trivial colorability. I then present a new criterion for spill node selection. Lastly, I present an improved version of Chaitin-Briggs algorithm, called Subgraph coloring, which is meant to decrease the amount of potential spill nodes which are spilled.

Lastly, I have implemented a combined scheme of register allocation and instruction scheduling. The effectiveness of such a scheme in a GPU compiler has been investigated. I have also found some problems which arise when combining the two methods. Many of these problems occur in register allocation in general, since they are related to register fragmentation.

Chapter 2
Background

In this chapter a detailed description of the register allocation pass, as well as the necessary information of compiler infrastructure will be presented. Lastly, the characteristics of a SIMD-vector register are presented and formulated.

Throughout this thesis, the terms *variable*, *live range*, *register* and *output of instruction* will be used interchangeably. A variable is the output of an instruction which is stored in a register during its live range. This background presumes scheduling is done before register allocation.

2.1 Compiler infrastructure in general

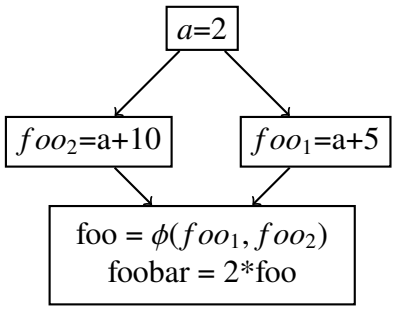
As mentioned in the introduction, a compiler consists of three parts: front-end, middle-end and back-end. We will be focusing on the back-end, since that is where instruction scheduling and register allocation are performed. For the purpose of this thesis, we simply assume that we will be given a *Control Flow Graph*, (CFG), which we will do these passes on. A more formal definition of a graph is found in Section 2.3.1. The CFG is a directed graph, whose nodes are *basic blocks*, and there exists an edge between two nodes in the CFG if there exists a branch instruction from the origin basic block to the target basic block. A basic block is a set of instructions which contains an entry point and only a single branch instruction at the end of the basic block. There are two special basic blocks in the CFG; the entry and the exit blocks. The entry block is the source of all flow in the CFG, and the exit block is the sink of all flow in the CFG [13].

In order to make better optimizations, the variables are transformed into *Single Static Assignment* form, or SSA for short. In SSA, each variable is only defined once. Once a variable is updated, the output of such an instruction results in a new variable rather than an updated variable. This is done in the middle-end, and its purpose is to make it easier to handle definitions and uses of a variable. An issue which arises due to this, is that we sometimes define a variable once, but there are two different definitions. Take the

following C-code for example:

```
1:  function Example C-code
2:      int a = 2;
3:      int foo;
4:      if bar then
5:          foo = a + 5;
6:      else
7:          foo = a + 10;
8:      end if
9:      foobar = 2 * foo;
10: end function
```

In this case, there exist two definitions of the variable *foo*. We have previously claimed that a redefinition creates a new variable. The issue is that there are two definitions and we do not know which of these reaches the use in *foobar*. To solve this issue, the concept of *phi-nodes* is introduced. A phi-node is a pseudo-instruction which takes all the possible values of the defined variable, and assigns the correct value to the variable according to the actual control transfer CFG edge. From the example code, we can construct a CFG, which would look like this:



2.2 Compiler Back-end

In this section, the parts of a compiler back-end which are vital for this report will be presented.

2.2.1 Instruction Scheduling

Instruction scheduling is done in order to use the hardware efficiently. This needs to be done in the back-end, since it is highly dependent on the characteristics of the target machine. Processors heavily use *pipelining* to achieve better performance. Pipelining is an act of splitting computations into sequential stages. In a traditional 5-stages pipeline, there are the following stages [11]:

1. Instruction Fetch

2. Instruction Decode
3. Execute
4. Memory Access
5. Write Back

During stage 1, the next instruction is fetched. During stage 2, the fetched instruction is decoded, so that the processor knows what to do. Stage 3 is where the actual computation is done. During stage 4, memory is accessed, which is not always necessary. Finally, during stage 5, the output of the instruction is written to a register. There are *dependencies* [11, 13], which determine how instructions may be issued. For instance, there are *data dependencies*, such that an instruction i defines a value, which is then read by instruction j. We can clearly not issue instruction j before the instruction i has written its output to registers.

Typically, instruction scheduling is done on a basic block level, so that each basic block in the CFG is scheduled individually. It is possible to construct a *dependency graph* [13] from the instructions in a basic block. All instructions in the basic block then forms nodes, and there is a directed edge from a node to another if there exists a dependency from the origin instruction to the target instruction, e.g. if the origin instruction produces a result which is used by the target instruction. There can be no cycle in such a graph, which makes it a *directed acyclic graph* (DAG), and the dependency graph is henceforth referred to as the DAG. Although it is possible to traverse the DAG in many ways, we will focus on two, which are *Top-down* and *Bottom-up*.

Essentially, every time we select an instruction to schedule, we select an instruction from a set of instructions which are available for scheduling. What instructions are available for scheduling is dependent on whether the scheduling is top-bottom or bottom-up. Which instruction to choose is not important for the purpose of this thesis, but it not a trivial problem (in fact, it is also NP-complete [13]).

Top-down is perhaps the most intuitive way to schedule, since it follows the flow of the program. Essentially, instructions which have all operands they use defined, are available for scheduling, and may be scheduled. In terms of the DAG, this is equivalent to a node having no incoming edges. After an instruction has been scheduled, all outgoing edges are removed from the DAG. This is then done until the DAG is empty, at which point the entire basic block has been scheduled.

In bottom-up scheduling however, an instruction is available for scheduling if it has no outgoing edges, i.e. there is no instruction dependent on it which has not yet been scheduled. When an instruction has been scheduled, all incoming edges from its corresponding DAG-node will be removed. Much like in top-down, whenever the DAG is empty, the entire basic block has been scheduled.

Finally, the scheduler needs to take *register pressure* into account. Register pressure is an estimate of how many physical registers are currently live. If the register pressure is high, then it is likely that we need to spill variables. Every instruction may use a number of variables and may write to a register. An instruction which writes its value to a register clearly requires register space. If there is currently a shortage of registers, then the scheduler should focus on freeing register space, whereas if the is a large amount of register space, the scheduler should focus on consuming register space.

In a VLIW, an abbreviation of *very long instruction word*, or a superscalar architecture, several instructions are issued the same clock cycle [11]. There are a number of functional units which the instructions can utilize, and if there for instance only exists a single adder, then two instructions which utilize an adder may not be scheduled at the same clock cycle. Using a superscalar or VLIW architecture means that the scheduler needs to take all the instructions in the instruction word into account when calculating how register pressure is affected by this instruction word, while at the same time optimizing functional unit usage, for instance using the adders as efficiently as possible.

2.2.2 Liveness Analysis

Liveness analysis is determining when a variable is *live*. Formally, a variable is live between its definition and its last use. This duration is called its *live range* [13], and that is what we will use henceforth. It is not necessarily easy to determine a variables live range. A variable may for instance have different last uses depending on paths taken in the CFG. If two variables at some point in time can not share register, then they are said to *interfere*.

We may split the live ranges into *local* and *global* live ranges. A local live range is a live range which is only live during a single basic block, whereas a global live range is live during several basic blocks. This division will be used later when describing the combined scheme of register allocation and instruction scheduling.

2.3 Register Allocation

Register allocation is the act of assigning all the variables in the CFG a register to be stored in through their live ranges. Typically, live ranges are ordered and then they are allocated according to that ordering. There are two main ways in which the ordering is done, which is *Graph Coloring* and *Linear Scan*.

2.3.1 Graph Coloring

A graph G is said to consist of *vertices*, sometimes referred to as *nodes*, and *edges*, sometimes referred to as *arcs*. The two entities are denoted V and E respectively, and the graph itself is denoted $G(V, E)$. Two nodes are said to be adjacent if there exists an edge which connects the two nodes. Additionally, the set of all nodes adjacent to a node is called its *neighborhood* [8].

During graph coloring, the nodes of a graph are colored, such that no adjacent nodes share the same color. Generally, we attempt to find a coloring using k colors, which is referred to as a *k-coloring*. The lowest number of colors needed to color a graph is said to be its *chromatic number* and is denoted $\chi(G)$. An example graph, called the *Diamond Graph*, is displayed in Figure 2.1. We see in Figure 2.1 that the chromatic number for the diamond graph is 2.

How does graph coloring correlate to register allocation? To answer that question, we first look into how variables can be stored in registers. Assume that we have a value in register R0. Then it must be safe to overwrite that value with another value if the original value did not have any more uses. Analogously, if we overwrite the value but still need to

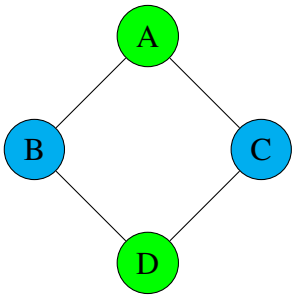


Figure 2.1: Diamond graph.

use it at some point in the future, we need to write that value to a register again. This of course means that we either need to have the value stored in memory, so that we can load it from memory, or have the possibility to compute the value again. Regardless of which way we restore the value to a register, it comes at a cost. Ideally, we would not want to need to restore any values in such a manner. We thus come to the conclusion that if two values are ever live simultaneously, then we do not want them to share the same register. This is equivalent to allowing two variables to share a register if they do not interfere. In graph coloring, we want to find a coloring such that no node shares color with an adjacent node. Assume we let a node represent a variable, and let there be an edge between two nodes if their corresponding variables interfere. Finally, if we let the number of colors equal the number of registers present, then we see that we can rephrase register allocation as "We attempt to assign registers to all variables, such that no interfering variables share the same register". We thus conclude that if we can find a coloring of the constructed graph, called the *Register Interference Graph* (RIG), then we can successfully allocate the variables registers [5].

It is possible to find optimal coloring in polynomial time when the graph is in SSA form. Performing register allocation in SSA form however results in non-optimal register allocation for the phi-nodes in the graph, since they have not been removed [3]. For this reason, graph coloring is not done in SSA form in this thesis. Finding a k-coloring for the interference graph is thus *NP-complete* [6]. If $P \neq NP$, then there are no optimal polynomial solutions to the problem. Proving that graph coloring is NP-complete is not included, but the idea is that in order to find a k-coloring, what is needed is to split the set of nodes V into k independent sets. There have been much research on finding efficient NP-solutions of graph-coloring, but it is generally still expensive.

2.3.2 Chaitin's Algorithm

Chaitin's algorithm is an iterative algorithm for approximating solutions of graph coloring, which is based on the notion of trivial colorability [5]. If we let the number of edges from a node be the *degree* of that node and the number of registers be N, then we note that a node v_i can be trivially colored if the following holds:

Definition 2.3.1. *A node v_i is deemed to be trivially colorable if $N > degree(v_i)$ holds.*

Chaitin's algorithm is based on graph reductions. If a node is trivially colorable, then it can be colored regardless of how its adjacent nodes are colored. In essence, this means

that such a node plays no role in the coloring, and as such, it can be removed. Removed nodes are pushed to a stack. The clever part of the algorithm is that all nodes which are pushed on the stack can definitely be assigned a color if all other nodes it is adjacent to have already been assigned colors. Since all adjacent nodes will be stored on top on the trivially colored node, we can simply pop from the stack and allocate in that order, after the graph has been completely reduced. There is however a possibility that the graph is *blocked*. The graph is blocked if there are no more trivially colorable nodes. This is illustrated in Figure 2.2. If there are no nodes which are trivially colorable, a node needs to be selected to be *spilled*. Spilling a node means that that node will be stored in memory and then loaded into registers when it is needed but not present in registers. This is costly, and we thus want to do this as little as possible. There are two things which makes a node a good choice for spilling. One is that it should cost little in terms of spill code inserted. The other is that removing the node should free up the graph as much as possible. In Chaitin's algorithm, we let the degree of a node determine how much removing a node frees up the graph. This means that in Figure 2.2, we would have selected B or C as spill nodes since they both have the highest degree, which is equal to three.

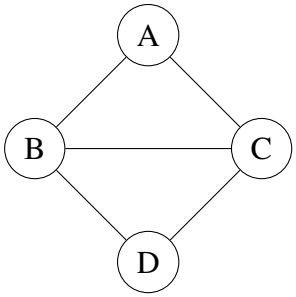


Figure 2.2: Blocked graph when coloring with 2 colors.

Chaitin's algorithm is outlined below [5].

```
1:  procedure CHAITIN'S ALGORITHM(V)
2:      S = ∅
3:      while V ≠ ∅ do
4:          for v ∈ V do
5:              if v is trivially colorable then
6:                  S ← R + v
7:                  V ← V − v
8:              end if
9:          end for
10:         if No node was removed this iteration then
11:             v ←best spill node
12:             V ← V − v
13:         end if
14:     end while
15:     return S
16: end procedure
```

In the algorithm outlined above, V is the set of nodes in the RIG, and S is the stack which all the removed nodes are pushed to. Additionally, for the stack, the + operator means stack pushing, whereas for the set, the − operator means removing the element from the set.

It is important to note that the actual register allocation is not done during the graph coloring, but rather afterwards, during register assignment. The graph coloring creates an ordering of the live ranges, so that the allocator should be able to iterate through the stack and successfully allocate the live ranges. During register assignment, nodes are popped from the stack and then assigned a color. Furthermore, the algorithm presumes that spilling does not fail.

2.3.3 Chaitin-Briggs algorithm

In Chaitin's algorithm, one would always spill if the graph got blocked. Briggs et al [4]. found that some graphs which Chaitin's algorithm would spill in, do not necessarily need to spill. An excellent example of such a case is the diamond graph, which is illustrated in Figure 2.1. In the diamond graph, we clearly can color the graph with two colors. However, since no node in the graph has a degree lower than two, one node would be spilled.

Briggs et al. suggested that the decision of spilling should be postponed until the actual assignment, and when removing a node from the graph it is only marked as a *potential* spill [4]. During the actual allocation, we attempt to allocate even nodes which were marked as a potential spill. If such a node could be allocated, then it is allocated and otherwise it is spilled. This is an improvement which makes the Briggs style allocator at least as efficient as the Chaitin allocator. If Briggs spills, then Chaitin also spills. If Chaitin spills, then maybe Briggs will not spill.

This addition to the allocator does not introduce any dangers of failing allocation of trivially colorable nodes, since all nodes which are allocated after a potential spill node are trivially colorable, whilst the ones above it will be allocated before the spill node. This will guarantee that the nodes which the potentially spilled node could affect the color of, will already have been allocated when the potential spill node is allocated.

The revised algorithm for graph coloring thus is:

```
1:  procedure Chaitin-Briggs colorer(V)
2:      S = ∅
3:      while V ≠ ∅ do
4:          for v ∈ V do
5:              if v is trivially colorable then
6:                  S ← S + v
7:                  V ← V − v
8:              end if
9:          end for
10:         if No node was removed this iteration then
11:             v ←best spill node
12:             V ← V − v
13:             S ← S + v
14:             Mark v as potential spill
15:         end if
16:     end while
17:     return S
18: end procedure
```

2.3.4 Linear Scan

Graph coloring is a fairly expensive technique to order the live ranges for register allocation, both in terms of memory usage and compilation time. First, the RIG needs to be constructed, which is done in $O(n^2)$, where n is the number of live ranges in the program, which is equivalent to the number of nodes in the RIG. Then, we apply Chaitin-Briggs coloring algorithm which in itself runs in $O(n^2)$.

Linear scan uses the idea that all that is ever really done during register allocation with graph coloring is to order the live ranges in an order which makes allocation go as smooth as possible, and then simply allocate as well as it can. This sounds exactly like the reasoning behind a greedy algorithm. If one can do something clever with the ordering of the list from the beginning, one perhaps does not need to use graph building, traversal and stack pushing just to get a good ordering of the live ranges. There are many different ways to order live ranges and each has its strengths and weaknesses. For instance, Poletto and Sarkar suggested ordering the live ranges by first definition [12]. If one for instance were to allocate the ranges with longest live range first, then that would avoid fragmentation caused by ranges which are short lived. A completely different strategy would be to allocate shorter live ranges first.

2.4 Vector Registers

In this section, the properties of vector registers are presented. The biggest difference between CPU register allocation and GPU register allocation is the usage of vector registers. A vector register has a size, i.e. a number of bytes of data it can hold. There is nothing inherently which states that a vector register is in fact larger than a scalar register but in

general, vector registers should be larger, so they may hold more data.

Definition 2.4.1. *A vector register is a register of a given number of bytes, capable of holding several variables. Furthermore, the set of all vector registers present during compilation is denoted $\mathcal{R}$.*

The way vector registers differ from normal scalar registers is their division into register components. Data can be read from these register components. There exists a smallest lane which data can be read through. Analogously, there exists a largest lane which data can be read through. Typically, these buses are aligned. If we for instance have 8 bits as smallest components size, then we can read 32-bit values from component 0 to 3, 4 to 7 etc.

Definition 2.4.2. *A vector register is divided into several minimal register components. Each register component has the same number of bytes, denoted B, which is constant and equal for all vector registers in $\mathcal{R}$. The number of register components is denoted N. All of these register components can be accessed through swizzling, so that no reordering of the register is necessary due to accessing its components.*

An example register where some register components are occupied is presented in Figure 2.3.

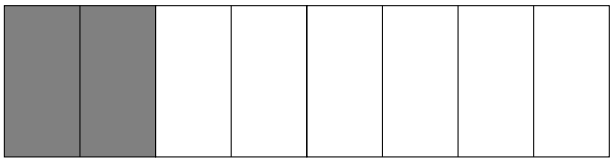


Figure 2.3: Example vector register, with N=8, where components 0 and 1 are occupied.

Definition 2.4.3. *A variable has two values: The number of components the variable has, denoted m, and the number of register components a single component occupies, denoted n. All valid values of n need to be powers of two. In practice, a variable v_i has m_i components, each requiring n_i register components to allocate.*

Definition 2.4.4. *A variable is said to be scalar if m=1.*

All components of a variable need to be allocated in the same register. A new behaviour in vector registers is that the allocator needs to handle register fragmentation. In Figure 2.3, we may for instance have had an allocation of a variable v_i with n_i=2. It is important to note that v_i now blocks the entire bottom half of the vector register for a variable v_j where n_j=4. Since the bus is between 0 through 3, and 4 through 7, we can not place the first two components of the variable v_j in register component 2 and 3, and then the other two components in register component 5 and 6. This means that efficiently, the placement of v_i has lead to 4 register components being occupied, because the allocation blocks the usage of the data bus. This is illustrated in Figures 2.4 and 2.5.

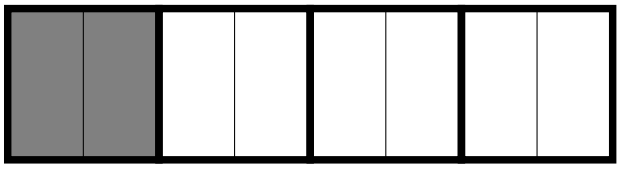


Figure 2.4: Vector register allocation of a variable with n_i=2, as seen from the point of view of a variable with n_i=2. Thick lines mark start and end of input buses.

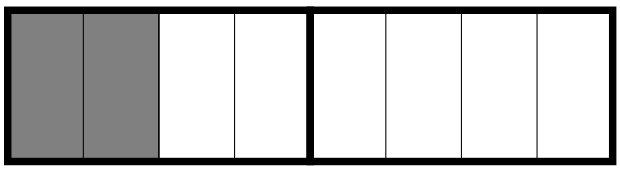


Figure 2.5: Vector register allocation of a variable with n_i=2, as seen from the point of view of a variable with n_i=4. Thick lines mark start and end of input buses.

Lemma 2.4.5. *For a scalar variable v_i, an allocation of the scalar variable v_j will block $max(n_i, n_j)$ register components for v_i.*

Proof. To prove this claim, we distinguish between two cases,

1. $n_i < n_j$
2. $n_i \geq n_j$

   If 1 holds, then the allocation of v_j will block exactly $\frac{n_j}{n_i}$ input buses for v_i. It will however fully block all of these buses. In this case, the allocation thus occupies n_j vector register components. If 2 holds, then the allocation of v_j will partially block a single input bus for v_i. This in turn means that the allocation of v_j will make an allocation of v_i impossible for that bus, which means that v_i will block n_i vector register components.

□

Lemma 2.4.5 is illustrated in the following two figures:

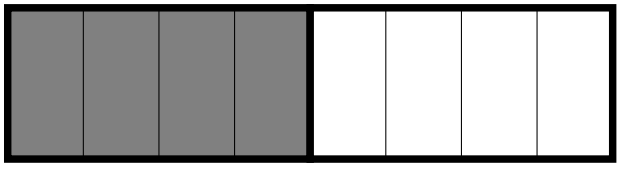


Figure 2.6: Register allocation of a variable with n=4.

As seen in Figure 2.6, this allocation blocks the first 4 register components. If we e.g. were to allocate another scalar variable which has n=2, the allocation still blocks 4 vector register components.

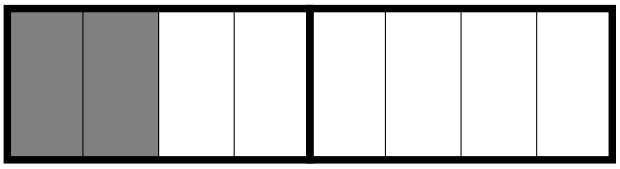


Figure 2.7: Register allocation of a variable with n=2, seen from the point of view of a variable with n=4

In the other case, where $n_i \geq n_j$, we get the following situation:

As illustrated in Figure 2.7, we have a variable allocated with n=2. If we now want to allocate a variable which has n=4, we find that the allocation blocks the usage of the data bus from register components 0 through 3, since register components 0 and 1 are occupied. There is no bus which can read from register components 2 through 5, so it is not possible to allocate this larger variable in register components 2 to 5. The allocation of the smaller variable thus effectively makes allocation impossible in vector components 0 through 3.

So what changes with these new characteristics? Essentially, three parts which become vastly different are instruction scheduling, liveness analysis, and of course, the register allocation. Instruction scheduling now needs to adjust register pressure, since issuing some instructions which write to registers will not necessarily write to a new register. The liveness analysis needs to be adjusted, since the components of a variable may have different live ranges. This means that we now not only need to keep track of when a variable dies, but also keep track of when every of its components goes dead. Graph coloring in itself needs to be adjusted to handle the new behaviour, both with the aforementioned alignment issues, but also how to handle registers being capable of holding several values.

Chapter 3
Hypotheses

In this chapter, the contributions of this thesis from a theoretical point of view are presented. Initially much focus will be put in an attempt to generalize Chaitin's algorithm to work for general purpose vector registers. Then follows a mechanism to maximize the chance of allocating potential spills. Finally, some theories on how to perform register allocation during instruction scheduling with vector registers will be discussed.

3.1 Required Adjustments To Chaitin-Briggs Algorithm

In this section the adjustments needed to generalize Chaitin-Briggs graph coloring algorithm to hold for vector registers will be presented. First follows a generalization of the criterion for trivial colorability, which has been performed in a ad-hoc manner up to this point. After that a more exact method of determining trivial colorability will be presented. Using the ideas from the trivial colorability, a new spilling metric is presented. Lastly, a method for reducing the amount of potential spills which fail allocation is presented.

3.1.1 Criterion for Trivial Colorability

As stated in the previous chapter, the algorithms and heuristics provided by Chaitin and Briggs are only focusing on scalar registers and as such do not handle the complications inherent to vector registers. Smith, Ramsey and Holloway generalized how register allocation may be performed when the registers are of different register classes [14], which have been useful for insights regarding this generalization.

Lemma 3.1.1. *The number of vector register components a single variable v_i requires in order to be allocated into a vector register is denoted N_i, and is given by, $N_i = n_i * m_i$.*

Proof. Each variable component requires n_i vector register components, so m_i variable components requires $n_i * m_i$ register components to be allocated. □

Lemma 3.1.2. *The maximum amount of vector register components which can be blocked by other variables without hindering an allocation of a variable v_i is denoted S_i, and is given by, $S_i = N - N_i$.*

Proof. There are N register components in a vector register. If v_i require N_i register components, then clearly the maximum amount of registers components which can be blocked is equal to $N - N_i$, which literally means that we can just fit v_i into that register. □

Using the lemmas outlined above, we arrive at the following theorem:

Theorem 3.1.3. *The worst-case number of vector components which can be blocked by other variables which hinder an allocation of a variable v_i in a register is $S_i + 1$.*

Proof. From Lemma 3.1.2, we know the maximum amount of vector register components which can be blocked by other variables is equal to S_i. If we could allocate v_i with $S_i + 1$ vector components blocked, then S_i would not be the maximum amount, which is a contradiction. □

Definition 3.1.4. *The total number of vector register components which are blocked by interfering variables across all vector registers, for a given variable v_i is denoted I_i.*

The value I_i is distributed over all register available. The interference from any one node is however calculated to only hold for a single vector register. This is from the fact that a variable needs to be fully allocated to a single register, and can not be distributed amount several registers. Theorem 3.1.1 and Definition 3.1.4, can be combined to form the following formula for trivial colorability:

Theorem 3.1.5. *The least number of register components which needs to be blocked in order to prevent an allocation an allocation of a node v_i to any register is $(S_i + 1) * R$.*

Proof. Blocking $S_i + 1$ register components is the least amount of register components which needs to be blocked in order to fully block a register. Then the least number of register components which needs to be blocked to prohibit an allocation to any register must be $(S_i+1)*R$, since in the worst case, all registers then have S_i+1 register components blocked. □

Definition 3.1.1 mentions that there exists a value I_i which is the amount of register components blocked by all adjacent nodes. Each of the adjacent nodes will contribute with some interference, which we conservatively approximate, meaning that we assume that all nodes are allocated in the worst possible manner. From Lemma 2.4.5, we know that the worst case interference a scalar variable will cause another scalar variable. In fact, the lemma may be generalized into theorem 3.1.6.

Theorem 3.1.6. *Any variable v_i will have the worst case interference by a scalar variable v_j to be $WC^1(v_i, v_j) = max(n_i, n_j)$, where the superscript signals that it is one component of v_j interfering with v_i.*

Proof. The fact that v_i is a vector does not mean that v_j can block more register components, so it must hold. □

Using all of the above theorems, we arrive at the following theorem.

Theorem 3.1.7. *The worst way a vector variable v_j can interfere with another variable v_i is given by*
$I_{ij} = min(m_j * WC^1(v_i, v_j), S_i + 1)$

Proof. In essence, the worst possible way m_j components can be allocated will be no worse than m_j worst-case single component allocations. The other case originates for the cases where a single variable blocks an entire register for another variable. Assume we have a vector register with N=8. Then, assume that we have a variable we want to allocate, v_i, with n_i=4 and m_i=1, and another variable allocated, v_j, with m_j=3 and n_i=2. The register state could in that case be as illustrated in Figure 3.1.

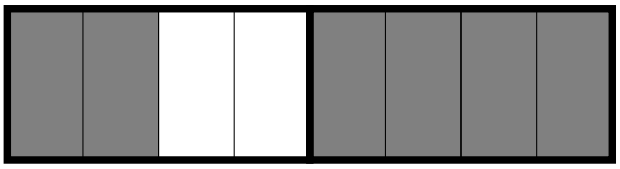


Figure 3.1: Illustration of an allocation of a variable with n=2 and m=3.

We would get that the worst-case number of register components blocked by v_j would be 3*(max(4,2))=12. The major issue is that 12 register components blocked implies that v_j blocks register components in several registers, since there are only 8 register components in each register. Since the interference caused by a single variable is contained within a single register, this is clearly not correct. It is thus not possible for a variable to ever block more than N register components. We can then let there be an upper bound of N register components. Alternatively, we can let the upper bound be $S+1$. Both bounds would work. The reason why I choose to use $S_i + 1$ is that in doing so, no other modifications needs to be done. We know that blocking $S_i + 1$ register components exactly blocks a register fully. If we instead had used N to signal that a variable fully blocks a register for allocation, then we would have needed to reduce the amount of registers available accordingly. As an example, imagine that we have a variable v_i with $n_i = 1$ and $m_i = 3$ and a variable v_j with $n_j = 2$ and $m_j = 2$ with a vector register with $N = 8$. This would in turn mean that $S_i = 8 - 3 * 1 = 5$ and $S_j = 8 - 2 * 2 = 4$. Then the interference v_i inflicts on v_j is given by $I_{ji} = min(m_i * WC^1(n_i, n_j), S_j + 1) = min(3 * 2, 4 + 1) = 5$. The worst case interference is illustrated in Figure 3.2. As seen in the figure, the worst case interference caused from v_i will in the worst case block an allocation in a whole register. The interference v_j inflicts on v_i is given by $I_{ij} = min(m_j * WC^1(n_i, n_j), S_i + 1) = min(2 * 2, 5 + 1) = 4$, meaning that v_i in the worst case blocks 4 register components for v_j. The worst case interference is illustrated in Figure 3.3. As seen in the figure, the interference from v_j does not fully block an entire register for v_i. If however, another variable v_k blocks 2 register components for v_i interferes, then the combined interference from v_j and v_k prohibits an allocation of v_i in the same register as they are allocated to, in the worst case.

□

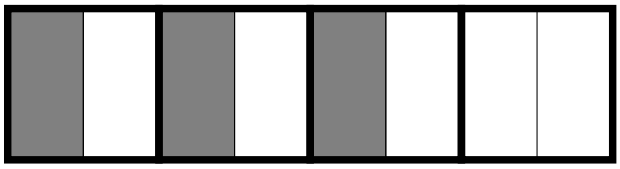


Figure 3.2: Illustration of the worst case interference caused by a variable with $n = 1, m = 3$ for a variable with $n = 2, m = 2$.

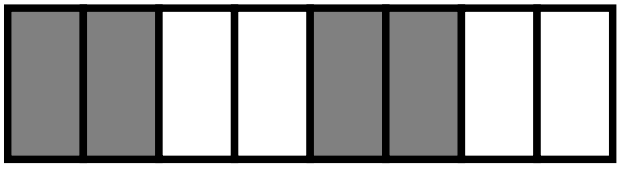


Figure 3.3: Illustration of the worst case interference caused by a variable with $n = 1, m = 3$ for a variable with $n = 2, m = 2$.

Now we have all the necessary information to define how to calculate the interference defined in Definition 3.1.4.

Definition 3.1.8. *The interference all the nodes adjacent to a given node v_i affects v_i with is given by $I_i = \sum_{v_j \in V_i} I_{ij}$ where V_i is the set of all nodes adjacent to v_i.*

Using Definition 3.1.8, we can extend Theorem 3.1.5 into the generalized form of trivial colorability for vector registers.

Definition 3.1.9. *We define that a node v_i is trivially colorable if the following criterion hold:*
$(S_i + 1) * R > I_i = \sum_{v_j \in V_i} I_{ij}$.

Finally, I would like to prove that it is possible to reduce Definition 3.1.9 into the regular Chaitin criterion for trivial colorability. This would imply that a register has only a single register component, i.e. N=1. This in turn implies that n=1 and m=1 for all variables. Using the equation in Definition 3.1.9, we get:

$$\begin{aligned}
&(S_i + 1) * R > \sum_{v_j \in V_i} WC^{m_j}(v_i, v_j) \iff \\
&(0 + 1) * R > \sum_{v_j \in V_i} WC^{1}(v_i, v_j) \iff \\
&R > \sum_{v_j \in V_i} 1 \iff \\
&R > |V_i| = deg(v_i)
\end{aligned} \tag{3.1}$$

which is equivalent to the Chaitin criterion for trivial colorability.

3.1.2 True Criterion For Trivial Colorability

The generalized criterion does not find any not trivially colorable node to be trivially colorable. It however does not find all the trivially colorable nodes. This is illustrated by the

following example. Imagine a case where we have two vector registers, and that a vector register has four components. Then assume we have a node that requires one vector register component, which is labelled A. Furthermore, assume that it has three adjacent nodes, all requiring three vector register components in order for them to be allocated, labelled B, C and D respectively. The interference graph is illustrated in Figure 3.4.

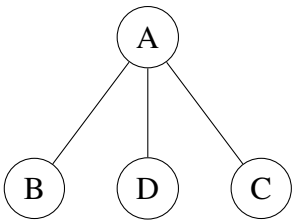


Figure 3.4: Interference graph which disproves the generalized criterion of colorability.

In Figure 3.4, the node A is the node which requires one register component, and the other three nodes requires three register components. This is not the full RIG, since if it was, then the nodes B, C and D would have been trivially colorable by the generalized criterion. For simplicity's sake however, the rest of the RIG has been left out. Then we can apply the generalized of trivial colorability for the node A, yielding

$$(3 + 1) * 2 > 3 + 3 + 3 \implies 8 > 9 \tag{3.2}$$

Since the criterion does not hold, we would regard node A to not be trivially colorable. However, if we imagine how the components of the variables can be placed in the vector registers, the nodes B and C would in the worst case be allocated as in Figure 3.5. As

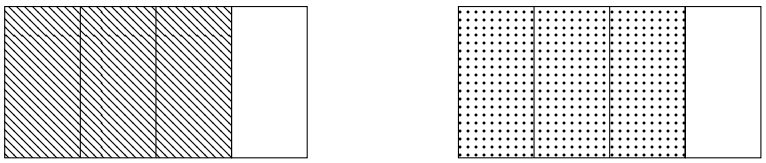


Figure 3.5: Worst-case register allocation of B and C, where B is allocated to the left register and C to the right register.

seen in Figure 3.5, there is a free register component in each of the two registers. Since D requires all of its components to be allocated in the same register, we can definitely fit A in whichever register D is not allocated to. We can thus see that A in fact is trivially colorable.

Finding such trivial colorability is more difficult than using the generalized criterion for trivial colorability. In this case, in order to block A from allocation in a single register, 4 register components needs to be blocked in that register. The set of interference which affects A is $I_i = \{3, 3, 3\}$. If there are no two disjoint subsets of I_i, such that the sum of the values in the subsets surpass 4, then the node is trivially colorable. This is the *true criterion for trivial colorability*. We need two subsets since there are two registers and the sum needs to surpass 4 since 4 register components needs to be blocked in a register to prevent allocation to that register. The subsets needs to be disjoint, such that no interference caused

by a single node, is in two of the subsets, since a variable only interferes in a single register. In the example case, we can not find two such subsets, the closest we can get is the two sets $\{3, 3\}$ and $\{3\}$, so the node must be trivially colorable. We can additionally see that if A was adjacent to another node which blocks a single register component, then we could find two such subsets, e.g. the sets $\{3, 3\}$ and $\{3, 1\}$. The generalization of this criterion is found in Definition 3.1.10.

Definition 3.1.10. *A node v_i is trivially colorable if it is not possible to partition I_i into R disjoint subsets, such that sum of the interference of all nodes in each subset is at least $S_i + 1$.*

An issue is that finding the subsets that make allocation of the node impossible requires us to iterate through all possible subsets to find an optimal subset, which of course takes NP-time. There must however be an upper bound on how high degree a node maximally may have in order to possibly have such subsets, and an upper bound of how many members the subset maximally may have. The latter value is $S_i + 1$, since each node blocks at least one register component, so in the worst case, we would need $S_i + 1$ nodes in the subset. If a node thus have a degree of more than $R * (S_i + 1)$, then clearly we can partition the nodes in such subsets.

This technique should only be used if the alternative is to select a potential spill node, since it is far more computationally demanding than the generalized criterion for trivial colorability.

3.1.3 Criterion for Potential Spill

Chaitin proposed that the node to be removed from the graph was selected by weighing a spill cost and the number of adjacent nodes it had in the graph. The rationale for using the number of adjacent nodes as a measurement is that a node with many adjacent nodes will be more likely to increase the colorability of the graph. The rationale behind spill choice does not change when using vector registers. However, simply using the number of adjacent nodes does not suffice. What we need is to incorporate the sizes of the variables when we select a spill node. A simple measurement for a given node v_i would be to use $n_i * m_i * deg(v_i)$ as a metric, where $\deg(v_i)$ is the number of nodes v_i is adjacent to. The rationale for such a criterion is that both the degree and the size of a variable affect the colorability of the graph. As in the Chaitin criterion, the higher degree of a node, the more likely it is that removing the node unblocks the graph. A larger variable occupies more register components, so removing a large variable should free more register components. To illustrate the issue with such criterion, imagine the following:

Assume we have two nodes, v_i which requires two vector register components and has five adjacent nodes, and v_j requiring only a single vector register component, but has nine adjacent nodes. By using the above criterion, we would select v_i as the potential spill. In general, it is better to spill the larger node, since it frees two register components for five nodes. Imagine however that v_j has only adjacent nodes which require a full register in order to allocate. Then effectively v_j blocks a whole register for all of its adjacent nodes. Removing the v_i will then effectively free a whole register for all nine adjacent nodes. It is reasonable that the spill criterion is modified to take the number of register components a node blocks for its adjacent nodes into account.

It is possible to rephrase the rationale for the traditional spill node criterion as "select the node which has the highest ratio between spill cost and how much it frees up in the graph". Spill cost is largely unchanged, so it is the latter part of the criterion which we will focus on. The node which frees up the most in the graph is reasonably the node which blocks the total highest amount of register components. This motivates the following definition:

Definition 3.1.11. *The total amount of register components a given node v_i blocks for all its adjacent nodes is denoted $\overline{I_i}$ and is called the spill degree of the node v_i. The bar signals that this is a separate value than the original I_i.*

Definition 3.1.12. *The worst-case amount of register components a node v_i can block for an adjacent node v_j is given by*

$$\overline{I_{ij}} = \begin{cases} m_i * WC^1(v_j, v_i), \ \textit{if } m_i * WC^1(v_j, v_i) + n_j * m_j < N \\ N \ \textit{otherwise} \end{cases}$$

The first case is in essence the same as for trivial colorability. The condition where the first case is used can be translated into "if v_i does not block enough register components to prevent an allocation to a register by itself". The second case simply states that we should count the amount of register space the node blocks as N, i.e. blocks an entire register, if v_i in itself prohibits an allocation to a single register. From Definition 3.1.12, we can then arrive at the following definition:

Definition 3.1.13. *The spill degree of a node v_i is given by $I_i = \sum_{v_j \in V_i} \overline{I_{ij}}$.*

I would like to finish this section as with the generalized criterion for trivial colorability, and prove that reduction from Definition 3.1.13 results in the Chaitin criterion for spill node selection.

Once again, scalar registers means that N=1, and for all nodes that m=1 and n=1. This would make the $\overline{I_{ij}}$ equal

$$\begin{aligned} \overline{I_{ij}} &= \begin{cases} m_i * WC^1(v_j, v_i), \ \text{if } m_i * WC^1(v_j, v_i) + n_j * m_j < N \\ N \ \text{otherwise} \end{cases} \\ &= \begin{cases} 1 * 1, \ \text{if } 1 * 1 + 1 * 1 < 1 \\ 1 \ \text{otherwise} \end{cases} = 1 \end{aligned} \tag{3.3}$$

This means that the spill degree of a node v_i will be,

$$\overline{I_i} = \sum_{v_j \in V_i} \overline{I_{ij}} = \sum_{v_j \in V_i} 1 = |V_i| = deg(v_i)$$

which is equivalent to Chaitin's criterion.

3.2 Subgraph Coloring

In this section a variation to the traditional Chaitin-Briggs scheme is developed, which attempts to reduce the amount of potential spill nodes which actually are spilled during allocation.

To illustrate inefficiencies of Chaitin-Briggs algorithm, here follows an example. Assume that we attempt to color the graph in Figure 3.6 with three colors. Since no node have a degree of less than three, we select a potential spill. Since A has the highest degree, it is selected as the potential spill node.

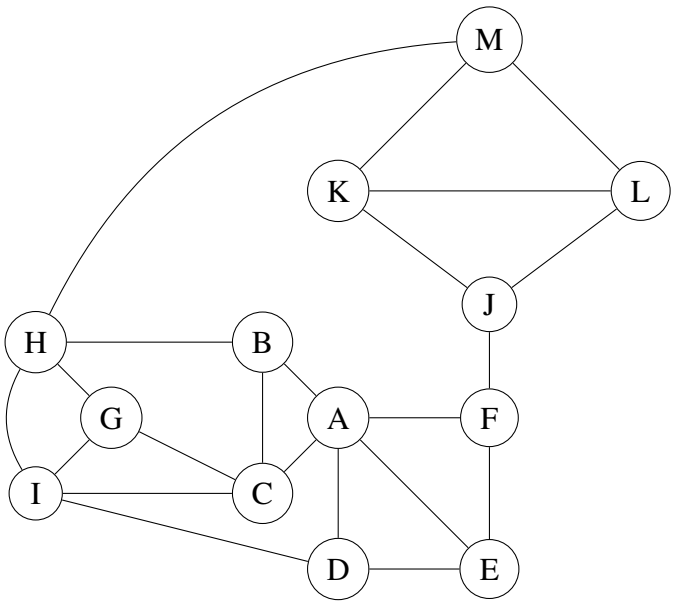


Figure 3.6: Example graph where Chaitin-Briggs fails.

As long as the nodes are trivially colorable when they are removed, it is possible to get an arbitrary order of the stack from Chaitin-Briggs algorithm. The resulting stack might, for instance, be as in Figure 3.7.

If we have the colors red, green and blue, then we may for instance color the nodes with the following coloring scheme:

- Color the node red if red is available.
- Color the node green if red is not available and green is.
- Color the node blue if no other color is available.

If we were to assign colors to the nodes in the graph using the scheme above, then we would get the graph in Figure 3.8.

As seen in Figure 3.8, the neighbors of the potential spill node have been colored using all available colors, meaning that the potential spill node, A, would be spilled. The chromatic number of the graph is however three, as can be seen in Figure 3.9.

Why did coloring fail when using Chaitin-Briggs for this graph? The only part which affects whether we successfully color the potential spill or not, is how we color the nodes

I
G
C
H
B
M
L
K
J
F
D
E
A

Figure 3.7: One of the possible resulting stacks when running Chaitin-Briggs algorithm.

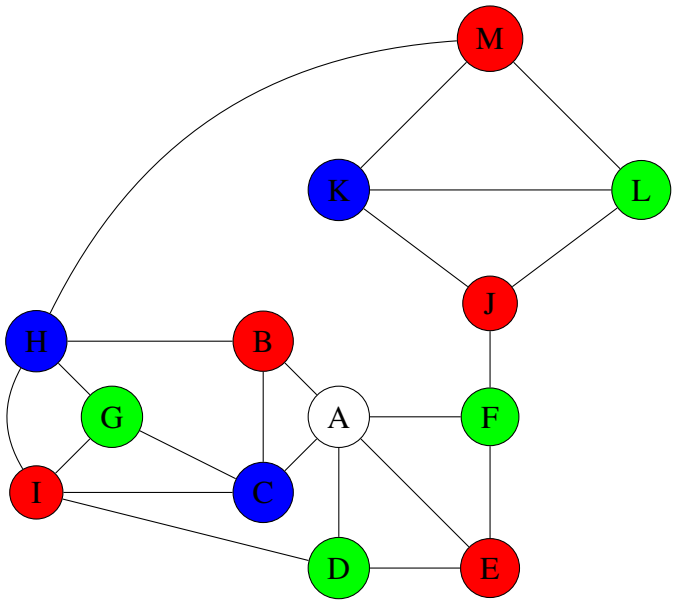


Figure 3.8: Coloring of the example graph with the stack in Figure 3.7

.

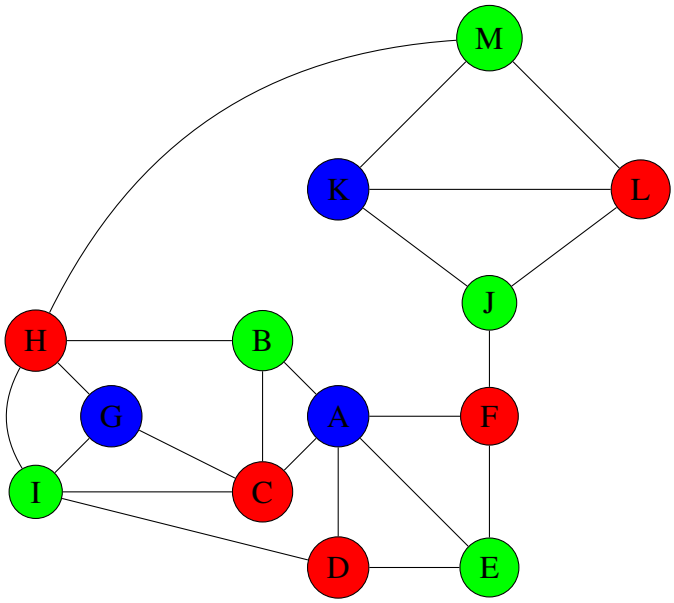


Figure 3.9: 3-coloring of the example graph.

adjacent to the potential spill. If the nodes adjacent to a potential spill node were to share more colors, as in Figure 3.9, then we would be able to color the potential spill node. It is more likely that two such nodes can share the same color if they are colored early, since in that case less constraints will be put as to which color they can be colored with. It is this notion that founds the basis of *Subgraph Coloring*. Subgraph coloring consists of partitioning the graph into two graphs, V and V', where V', the *subgraph*, consists of the nodes adjacent to a potential spill node and V, the *disjoint graph*, is the rest of the graph. In the example graph, we would have $V' = \{B, C, D, E, F\}$ and $V = \{G, H, I, J, K, L, M\}$ We then remove nodes from V as normal, however, we only remove nodes from V' when we have no other choice. This should in theory make the nodes in V' to be placed higher up in the stack than they would have been, had not Subgraph Coloring been used. The earlier a node is removed from V', the more likely it is that it is assigned a color which makes coloring the potential spill node impossible. For this reason, we want to remove as few nodes from V' as possible. Due to this, when we need to remove a node from V', we remove the node which increases the colorability of the disjoint graph the most, i.e. the node with the highest *disjoint degree*, which is the number of nodes it is adjacent to which are members of V. Removing such a node makes it the most likely that more nodes becomes trivially colorable in the disjoint graph. Selecting such a node however also means that the disjoint graph at worst will block a high amount of colors for the node, which may force the node to be colored such that we fail coloring the potential spill node.

Using the above heuristics, we attempt to color the graph in Figure 3.6. We would thus select the node with the highest disjoint degree, which in this case is a tie between the nodes B, D and F, since they all have a disjoint degree of 1. We see that C has a disjoint degree of 2, however, we can not remove it since it is not trivially colorable. We arbitrarily select B for removal. Removing B does not make any disjoint nodes trivially colorable, however it makes C trivially colorable. Since C has disjoint degree of 2, we remove it from the graph. Removing C makes I and G trivially colorable. It is then possible to remove all nodes from the disjoint graph, so that the only remaining nodes are the nodes in the

F
E
D
J
L
K
M
H
G
I
C
B
A

Figure 3.10: One of the possible resulting stacks when running subgraph coloring algorithm.

subgraph. We may then arbitraily remove the nodes in the subgraph. A possible resulting stack from running subgraph coloring on the example graph is presented in Figure 3.10 and the coloring of the graph is in Figure 3.11.

We could instead have removed F in the beginning, which would have made it possible to remove J, K, L and M. The graph would then have been blocked, after which another node from V' would have been removed. If we attempt to color the example graph with subgraph coloring and remove F first, then the resulting stack may be as illustrated in Figure 3.12.

We can compare the stack produced by subgraph coloring and the stack produced by ordinary Chaitin-Briggs algorithm. This is presented in Figure 3.13. We can see that, in essence, subgraph coloring moves the nodes in the subgraph upwards in the stack. Moreover, we see that three nodes from the subgraph are colored first by the subgraph coloring algorithm. It is not possible that the coloring of these nodes by themselves prevents coloring of the potential spill. This means that the only nodes which may be colored such that we can not color the potential spill, are B and F.

The above example only has a single potential spill node. If there are several potential spill nodes, then we add each of their neighbors to the subgraph. We may want to bias from which of the potential spill nodes we want to remove a neighbor. The basis of subgraph coloring is that coloring is easier the higher up in the stack a node is placed. For this reason, we may want to avoid removing nodes neighboring to a potential spill node which was removed from the graph early, since removing one of its neighbors makes it more likely that we fail coloring it. Conversely, we may want to avoid removing nodes neighboring the potential spill which was removed last, since it is the most likely that we can still color it. Removing one of its neighbors may result in it becoming non-colorable. These are two different strategies which are interesting in subgraph coloring. A completely different strategy would be to use the heuristics from spill node selection. This is based on the fact that we want to avoid spilling nodes which are expensive to spill. We let the nodes in the subgraph carry the spill cost of the potential spill node it neighbors, and then select nodes which have low cost. The spill cost criterion is what was used in this thesis.

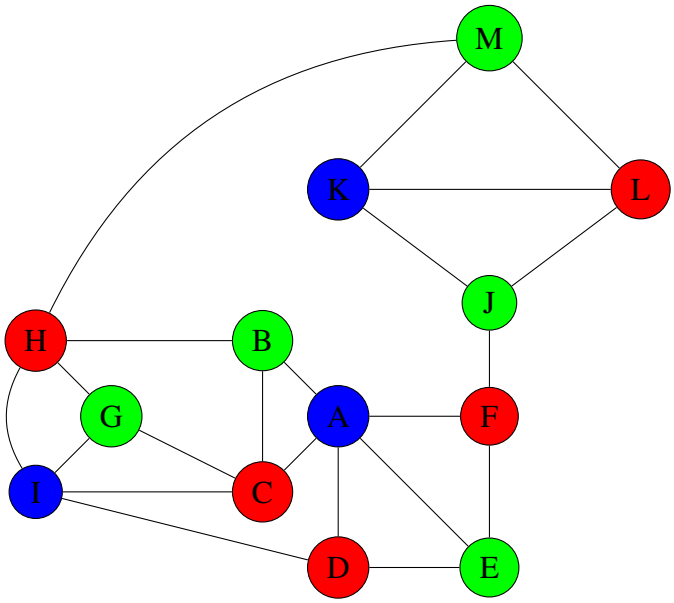


Figure 3.11: Coloring of the example graph with the stack produced by subgraph coloring, found in Figure 3.10

D
E
C
I
G
H
B
M
L
K
J
F
A

Figure 3.12: One of the possible resulting stacks when running subgraph coloring algorithm when removing *F* first.

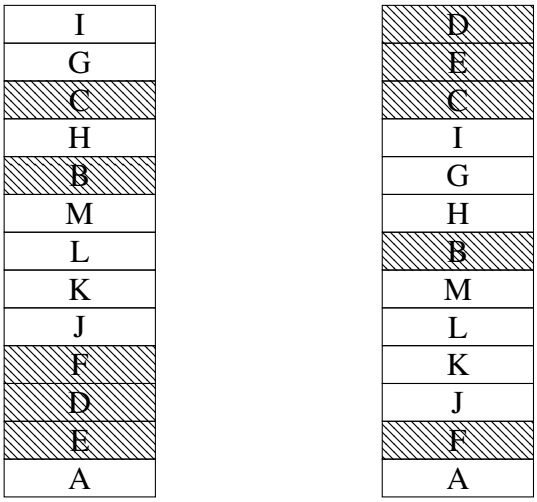


Figure 3.13: Comparison between the stacks produced by Chaitin-Briggs algorithm and subgraph coloring. The nodes which are adjacent to the potential spill are hatched for easier comparison. Left: Stack produced by Chaitin-Briggs algorithm. Right: Stack produced by Subgraph coloring

Another problem which arises in subgraph coloring when having several potential spills, is that some nodes in the subgraph may be adjacent to several potential spill nodes. If using heuristics dependent on when a node in the subgraph was inserted in the subgraph, then no adjustments are needed. If we however use the spill cost heuristic, then we can make an adjustment. In the worst case scenario, removing a node from the subgraph which is adjacent to several potential spill nodes will make coloring impossible for all of the potential spills it is adjacent to. We can thus let every node in the subgraph carry the sum of the spill costs of all potential spill nodes it is adjacent to.

The revised Chaitin-Briggs colorer is presented in Algorithm 1. The main structure of Chaitin-Briggs algorithm is unchanged. The new part lies in the case where no nodes in V are trivially colorable. In that case, we either select the best trivially colorable node to remove in V', or if no nodes in V' are trivially colorable, then a new potential spill node is selected. If V is the empty set, and there are nodes left in V', then the graph V' can be colored using the subgraph coloring algorithm. In this thesis, we used the highest disjoint degree and spill cost of the associated potential spills to select which node to remove from the V'.

The amount of potential spills should not be affected. The subgraph colorer will only remove a node from V' if no disjoint node is trivially colorable. But we will only select a new potential spill if no node in V' or V is trivially colorable, which is the scenario where we spill in a classic Chaitin-Briggs scheme. The amount of spill code inserted should however be affected, since in theory, this scheme should reduce the amount of potential spills which are spilled during allocation.

Algorithm 1 Subgraph Coloring algorithm.

```
1:  procedure SUBGRAPH COLORING(V)
2:      S ← ∅
3:      V′ ← ∅
4:      while V ≠ ∅ do
5:          for v ∈ V do
6:              if v is trivially colorable then
7:                  S ← S + v
8:                  V ← V − v
9:              end if
10:         end for
11:         if No node was removed this iteration then
12:             if can remove node in V′ then
13:                 v ← best node in V′ to remove
14:                 V′ ← V′ − v
15:                 S ← S + v
16:             else
17:                 v ←best spill node
18:                 for all v_i ∈ E_v do
19:                     if v_i ∉ V′ then
20:                         V′ ← V′ + v_i
21:                         V ← V − v_i
22:                     else
23:                         Increment spill cost of v_i
24:                     end if
25:                 end for
26:                 V ← V − v
27:                 S ← S − v
28:                 Mark v as potential spill
29:             end if
30:         end if
31:     end while
32:     if V′ ≠ ∅ then
33:         S ← S+ Subgraph Coloring(V′)
34:     end if
35:     return S
36: end procedure
```

3.3 Linear Scan

In linear scan, the only thing we are interested in at this point, is how to order the live ranges for an efficient allocation. The orderings we investigate are,

1. Earliest definition
2. Longest life span

3. Longest life span*size

Earliest definition is the criterion suggested in [12]. The longest life span can be interesting, since those live ranges are the most likely to affect other live ranges. The last criterion is an attempt to make an easy ordering which takes size of variables into account, since that is not done in the previous two.

3.4 Combining Register Allocation with Instruction Scheduling

In this section, methods developed for a combined register allocation and instruction scheduling scheme are discussed.

3.4.1 Top-down or Bottom-up

In a combined scheme for register allocation and instruction scheduling, scheduling direction is of major importance. If scheduling top-down, register space will be allocated upon scheduling a definition of a variable and deallocated when the last instruction which has a data dependency upon it is scheduled. If scheduling bottom-up, register space will be allocated when we schedule the first instruction which has a dependency towards that instruction and deallocate when scheduling variable's definition. A difference between regular registers and vector registers is that the various components of a variable may be defined by different instructions, which is why there may be bias as to which strategy is best. To illustrate this bias, here follows two examples.

Assume we are using top-down scheduling, and that we have a variable v, with three components, namely $v.x$, $v.y$ and $v.z$, and that they are defined at instructions i_1, i_2 and i_3 respectively. Finally, assume there is an instruction i which kills *all* components of v. Instruction i_1 is scheduled first, followed by i_2, followed by i_3. Instruction i is scheduled last. When we schedule i_1, we need to allocate $v.x$. We do not need to allocate $v.y$ or $v.z$ right now, since we have not yet scheduled i_2 or i_3. However, we may only allocate other variables to the register space they will occupy if we can guarantee that those variables die before scheduling of i_2 or i_3. Since we do not know when we will schedule i_2 or i_3, we resort to allocating all of v during scheduling of i_1. In this case, we are effectively wasting the register space which $v.y$ and $v.z$ occupies until we schedule i_2 and i_3 respectively. Assume now that we would schedule bottom-up, so that i is scheduled first, followed by i_3, followed by i_2 and lastly i_1. Then when scheduling i, we would allocate register space for all components of v. Upon scheduling the definitions of v, we will then deallocate their respective component. This means that we will not waste any register space in this case. Since we prefer not to waste register space, we prefer bottom-up to top-down.

Now assume the opposite. We have a variable v with three components, all of which are defined by a single instruction i. There are three instructions i_1, i_2 and i_3, that kill $v.x$, $v.y$ and $v.z$ respectively. Assume that we are using top-down scheduling. Upon scheduling i, we will allocate register space for v. The components will then be deallocated upon scheduling of their respective last use. Unlike the previous example, top-down scheduling will *not* waste any register space. Now assume we use bottom-up scheduling. The order of

scheduling is then first e.g. i_3, followed by i_2, followed by i_1 and lastly i. Upon scheduling of i_3, we notice that we face the same issue as we faced in the previous example for top-down scheduling. We can only allocate any variable in the components occupied by $v.x$ or $v.y$ if that variable dies before we schedule i_1 or i_2. We still do not know when either i_1 or i_2 is scheduled, so we will allocate all components of v upon scheduling i_3, resulting in register wastage until i_1 and i_2 are scheduled. This results in register wastage, so in this case, we prefer top-down scheduling.

To find which scheduling direction is most beneficial, we should investigate how commonly these two scenarios occur.

3.4.2 Avoiding Fragmentation

Register allocation during instruction scheduling should only be performed for local live ranges, which are live ranges that only are live within a single basic block. The reason for this is that it is difficult to keep the global live ranges allocated efficiently, especially with live ranges in phi-nodes. For this reason, we will use a scheme as proposed by D. Ivanov in [7].

Disregarding allocating global variables, the main issue with performing register allocation during instruction scheduling with vector registers is avoiding fragmentation. To illustrate with an example, assume we have vector registers with 4 components. Assume we need to allocate two variables which require three components, and three variables which require a single component. It seems to be a good idea to allocate one variable requiring three components together with one which requires one component, since such a grouping together occupies a whole vector register. Doing so would result in the register state illustrated in Figure 3.14.

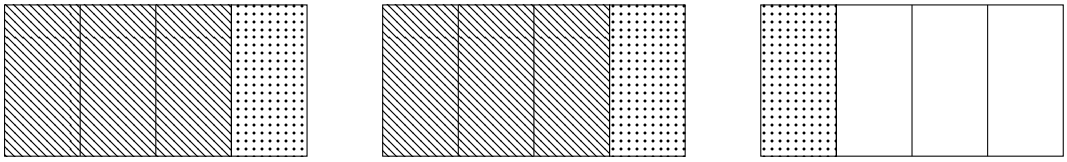


Figure 3.14: Example of optimal temporal register allocation.

With this allocation we are subject to minimal fragmentation if we allocate in this manner. The issue however is that this allocation is optimal in regards to fragmentation for this particular clock cycle. It may however be the case that the two 3 component variables dies the next clock cycle. If that were to happen, the register state would be as in Figure 3.15 the next clock cycle, presuming no other variables are allocated to the registers.

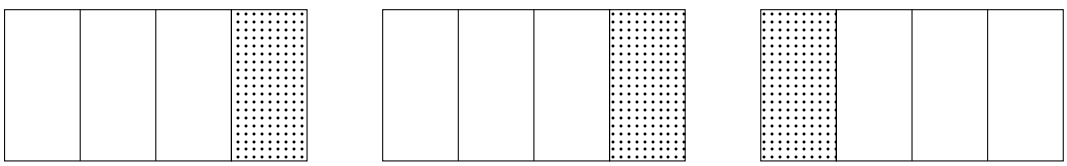


Figure 3.15: Resulting register state from temporal optimal register allocation.

The allocation has now introduced a lot of fragmentation in the registers. In fact, this allocation prevents an allocation of a variable which require a whole register in all three registers.

3.4.3 Placement Strategy

The main problem in the previous section was that we did not take the liveness into account. Of course, one issue with performing register allocation during instruction scheduling is that we have not determined the liveness yet. We therefore must approximate the liveness. Let L_i be the approximation of the liveness of variable v_i, and that it is equal to the exact number of instructions we at least need to schedule in order to kill v_i.

Assume we now have a set of variables V allocated to a given vector register. We may then estimate the worst-case register waste from that allocation. We let W denote the worst-case register waste, and it is given by:

$$W(V) = L * (N - N_V) + \sum_{v_i \in V} (L - L_i) * N_i \tag{3.4}$$

where L is the maximum approximated life length of all variables in V and N_V is the amount of register components occupied in the vector register.

Assume we have a set of variables I, which is the set of variables that should be allocated right now. Furthermore, assume that we have R registers, and that V_i is the set of variables currently allocated to R_i. Then we want to allocate all members of I, such that we minimize

$$W(I) = \sum_{R_i \in R} (W(V_i)) \tag{3.5}$$

If we can find allocations of I such that we minimize $W(I)$, then we are less likely to get fragmentation in the vector registers. There are two primary shortcomings of using such a method to find a good allocation. The first is that it seems to be NP-complete. The second is that we do not use any information of what will be scheduled next. Since we have information regarding what instructions are available for scheduling, we effectively know roughly what we will need to allocate the next cycle. The issue is that we still can not use that information effectively. We may force the scheduler to schedule a given instruction because it improves allocation, but that may have consequences later during scheduling.

3.4.4 Fragmentation and Live Range Splitting

Even though we preemptively work towards minimizing the fragmentation in the registers by following the placement strategy, this work may not be ideal. Instead of spilling or rematerializing a variable, we may perform *live range splitting* instead. Rematerialization is the act of recalculating a value. Live range splitting is the act of splitting a live range into several subranges and letting the subranges be allocated in different registers [10], or in the case of vector registers be allocated to different components within a register.

If there exists no available instruction whose output can be stored in a register due to register fragmentation, then we may still not need to spill. Imagine the case where there,

for instance, exists enough space in a vector register, but the register is so fragmented that it would not be possible to allocate a certain live range in it. We may instead do some clever reorganization of the vector registers, which would make live range fit. Among the currently allocated variables, we want to select one, or perhaps more, variables to reallocate in order to decrease fragmentation. This is hard, since we in essence want to minimize the amount of moves that we insert, i.e. the amount of variables that we reallocate. Furthermore, we want to reduce the fragmentation as much as possible. How to manage optimal live range splitting for fragmentation is an interesting topic, but no answer is presented in this thesis.

Chapter 4
Evaluation Methodology

The target compiler is the ARM Midgard compiler, used in Mali GPU. The hardware is the ARM Midgard architecture [1]. Primarily the focus lies on the Mali-T760, however when more data is needed, more available versions of Mali is used. The benchmarks which the efficiency of the algorithms are tested on, are the internal performance suites of ARM Midgard. They consist of a large amount of commonly used programs, representative for normal GPU computation. The programs are compiled, after which the compiled code is run. After running the program, data regarding the amount of load cycles spent, which spilling mostly affects, the execution time and the number of registers used, is gathered. This data then forms the basis of the evaluation. Since shaders are typically small, there are no larger differences with any techniques, which makes min and max differences less interesting. For this reason, we will focus on averages to see trends.

For linear scan, the anticipated result is that the amount of register used, load cycles and the execution time of the compiled programs is increased, compared to programs compiled using graph coloring. We however anticipate that the compilation time will decrease when using linear scan, compared to using graph coloring. Linear scan will thus be measured in terms of amount of registers used, loads executed and execution time, as well as compilation time.

The true criterion of trivial colorability will be measured in reduction in amount of potential spills. There should be some situations where the generalized criterion for trivial colorability results in a blocked graph, but by using the true criterion of trivial colorability we can find more nodes to be trivially colorable and thus make the graph not blocked. For this reason, the amount of potential spills should be reduced when using the true criterion for trivial colorability.

For the new method of spill node selection, the main metric will be the amount of clock cycles spent on loads and execution time. Intuitively it seems like a good idea to use amount of spill code inserted as an evalutation, however this may be misleading. Reductions in inserted spill code may still result in an increase in load cycles spent, since the spill code inserted may have been inserted in a loop for instance.

Subgraph coloring is, much like the improved spill criterion, meant to decrease spills. For this reason they have the same evaluation metric. It would be interesting to see what effect subgraph coloring has on compilation time as well, since it should make compilation slower. This will however not be investigated, due to limitations in the compiler which makes the implementation far from being optimized, which would result in an unrealistic measurement of compilation time.

For the combined scheme for instruction scheduling and register allocation, the main metrics will be average amount of registers used and compilation time. The combined scheme is a prototype, so many of the values are preliminary.

Chapter 5
Results and Discussion

In this chapter, the results of the investigations are presented and discussed.

5.1 Linear Scan

The results of the different linear scan orderings are presented in Figure 5.1. In the figure, a higher value than 1, in for instance registers used, means that the ordering used more registers than graph coloring. The same holds for load cycles and execution time, where a value higher than 1 equates to having more load cycles and longer execution time than graph coloring. Furthermore, data from the benchmark Mali-T760 GLES was gathered. The result is presented in Figure 5.2.

As seen in both Figure 5.1 and Figure 5.2, the ordering of the live ranges which resulted in the best allocation was longest live range first. The reason why this was the best may be connected to the fact that a variable which is live for a long time is more likely to have more overlapping live ranges. This would mean that such a node would have a high degree in the RIG, if it was constructed. In general, the stack produced by Chaitin's algorithm is ordered in decreasing original degree, so that the longest living live ranges are in general placed on top of the stack. Linear scan with the live ranges ordered in decreasing life length thus approximates the graph coloring solution. This is likely the reason why it also performed the closest to it. Another interesting observation is that life length*size does not perform as well as only longest life length. This is quite interesting and suggests that how hard it is to allocate a variable is more dependent on life length than size. Size seems to matter, but not as much as life length.

Using linear scan instead of graph coloring reduced compilation time by roughly 12% on average. It also reduced the register allocation time by roughly half.

Much like predicted, linear scan reduced compilation time, at the cost of lower performance of machine code. For an AOT-compiler, there is no question as to what to choose; graph coloring is preferred. However, even for a JIT-environment, I would suggest using

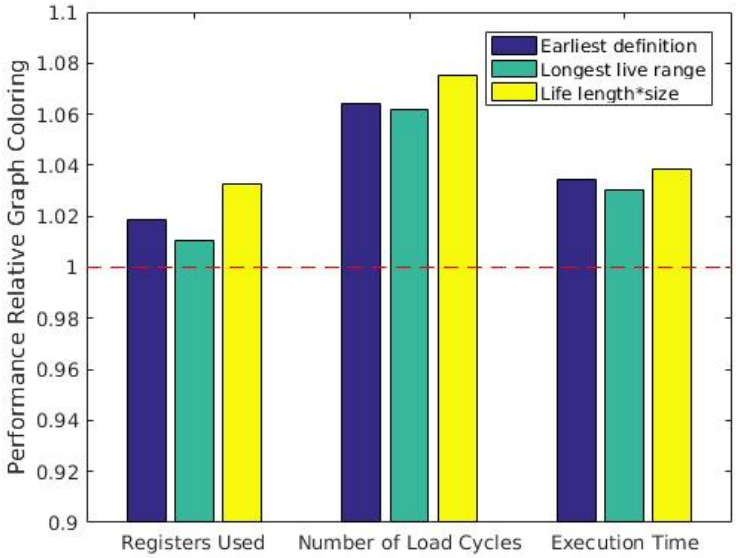


Figure 5.1: Performance of Linear Scan relative to Graph Coloring, in respect to amount of registers used, load cycles executed and execution time. The bars are the different orderings of linear scan. The left bar is earliest definition, the middle bar is longest live range and right bar is length*size.

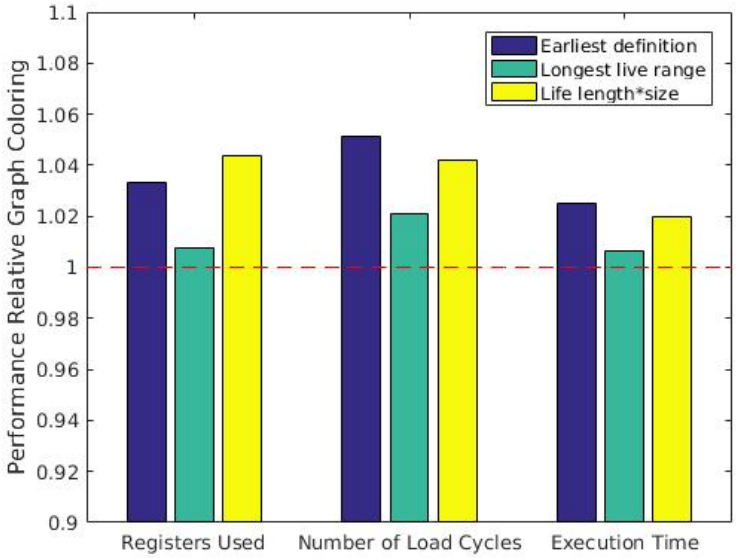


Figure 5.2: Performance of Linear Scan relative to Graph Coloring for Mali-T760. The bars are the different orderings of linear scan. The left bar is ordering by earliest definition, the middle bar is ordering by longest life range and right bar is ordering by life length*size.

the graph coloring algorithm. One of the most problematic aspects of register allocation when using vector registers is avoiding fragmentation and handling the different sizes of variables. Adjusting the linear scan algorithm to take this into consideration would make it slower, which reduces the attractiveness of the algorithm. Chaitin's algorithm inherently handles fragmentation by ordering the live ranges with regard to interference. For this reason, I would say that graph coloring is preferred over linear scan for a linear scan register allocator.

5.2 Graph Coloring

In this section, the results specific to graph coloring improvements are discussed.

5.2.1 True Criterion For Trivial Colorability

Using the true criterion for trivial colorability yielded **no** difference in the amount of potential spill nodes. This is quite interesting, because there were many nodes which could be removed with this improved way of determining colorability. A reasonable explanation why this still did not lead to any reduction in potential spills may be that it is rarely so that such nodes are bottlenecks which later trigger more nodes to be trivially colorable. The graph in Figure 5.3 might not be commonly found in a RIG and we instead have then graph in Figure 5.4. In such a graph, there is no way to avoid spilling, so the effort to remove A from the graph is futile since we need spilling regardless.

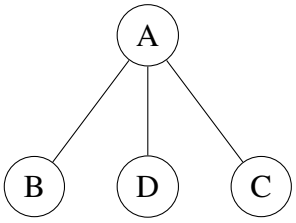


Figure 5.3: Graph where removing a node using the true criterion for trivial colorability unblocks the graph.

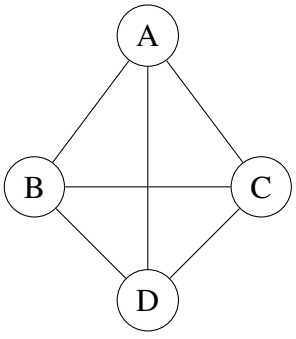


Figure 5.4: Graph where the true criterion for trivial colorability does not matter.

5.2.2 Criterion For Potential Spill

The results gathered for changing the criterion of potential spill node is represented in Figure 5.5.

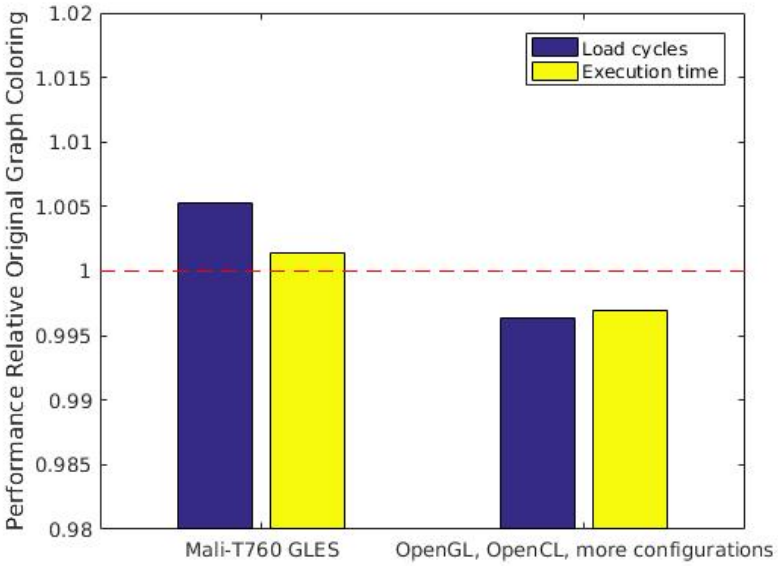


Figure 5.5: Results of changing the Spill criterion to my criterion. The left bar is loads relative original criterion and the right bar is execution time relative to original criterion.

As seen in Figure 5.5, the effect of changing the spill criterion was fairly unsubstantial. For most suites, the new criterion outperformed the original, so Mali-T760 GLES is a unrepresentative measurement. The positive effect likely comes from the fact that it is impossible to say how much register space a node blocks by simply looking at the number of adjacent nodes and space required. The new criterion for potential spills is dynamic, and has its foundation in the amount of register space blocked. This result however, does not mean that using my criterion is guaranteed to outperform the original criterion for any shader. In fact, as seen in Figure 5.5, the new criterion performed slightly worse for the benchmark Mali-T760 GLES. It is also not unlikely that the two criteria chooses the same potential spill nodes in some cases, since the size increases the amount of register components blocked, as does the degree. In that case, there will be no difference between the two criteria.

One reason why my criterion would result in more spill code inserted may be that in general, the nodes selected with the original criterion may be easier to allocate than nodes selected with my criterion. It is possible that choosing nodes with the stricter criterion may select nodes which can not be allocated during allocation, whereas nodes selected with the other criterion can be allocated. In that case, we are taking a risk. The node selected by my criterion will insert less spill code if spilled, but at a higher risk of failed allocation. This may explain why some shaders performs better with the original criterion, whereas in general, the new criterion performs better in terms of spilling and execution time.

5.2.3 Subgraph Coloring

The results from using the Subgraph Coloring algorithm, while only taking disjoint interference into account are presented in Figure 5.6.

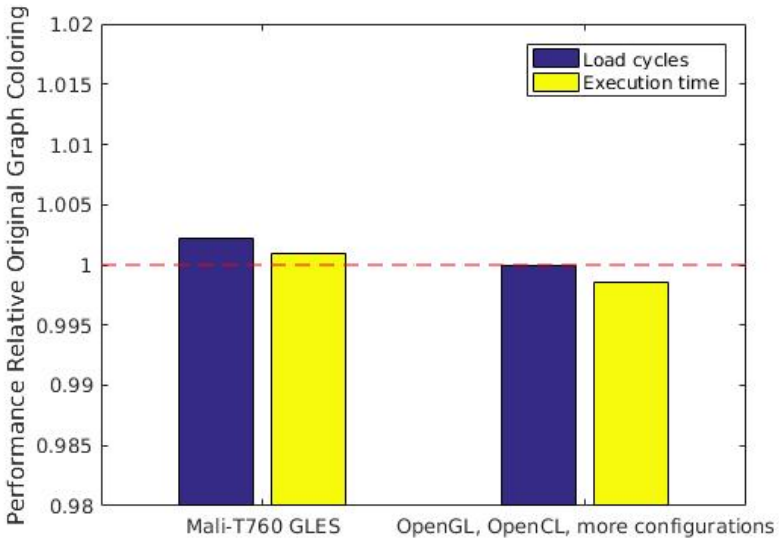


Figure 5.6: Results of extending Chaitin-Briggs with Subgraph Coloring. Left bar is loads relative Chaitin-Briggs and right bar is execution time relative to Chaitin-Briggs.

As seen in Figure 5.6, the benefits of only utilizing the disjoint interference part of Subgraph Coloring is insignificantly small, if any. If we however use the internal interference as well, we get Figure 5.7.

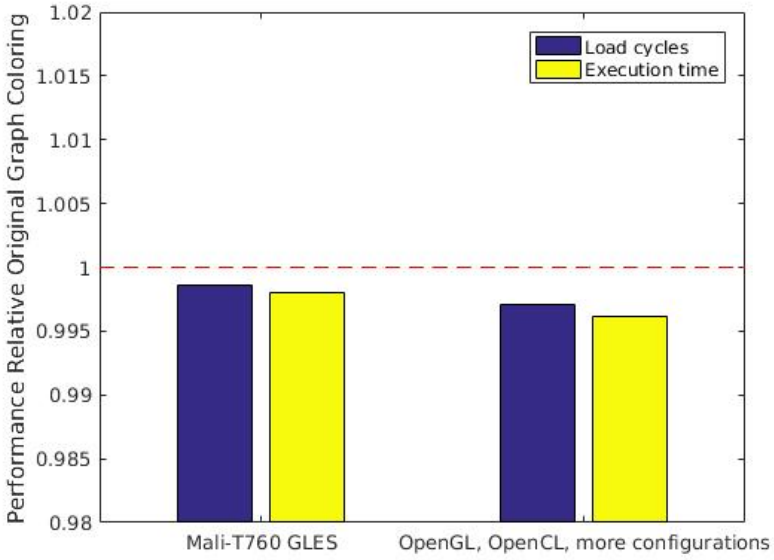


Figure 5.7: Results of extending Chaitin-Briggs with Subgraph Coloring. Left bar is loads relative Chaitin-Briggs and right bar is execution time relative to Chaitin-Briggs.

As we see, the improvement is now larger, with an average decreased execution time of 0.5%. Combining the spill criterion and Subgraph Coloring results in the Figure 5.8.

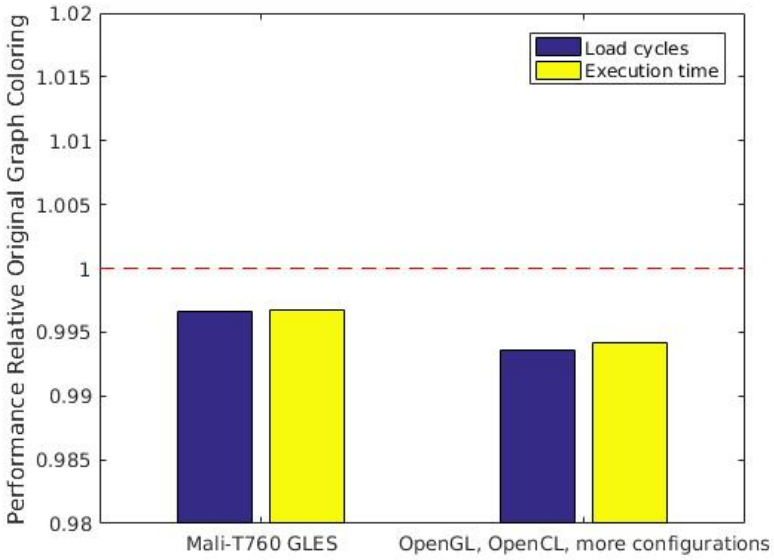


Figure 5.8: Results of extending Chaitin-Briggs with Subgraph Coloring. Left bar is loads relative Chaitin-Briggs and right bar is execution time relative to Chaitin-Briggs.

As seen in Figure 5.8, the effect of combining the two methods decreased both time spent on load cycles and execution time. This seems reasonable, since they both contribute

to decreasing spill code insertion.

One reason why the Subgraph coloring algorithm does not improve performance by much may lie in the fact that shaders are short programs, often not more than a hundred clock cycles long. In such programs, there is generally not much spilling, so the effect of subgraph coloring might not be as clear as in longer programs

5.3 Combined Scheduling and Register Allocation

In this section, results gathered from the combined scheme are presented and discussed.

5.3.1 Top-down or Bottom-up

The effects under investigation was having several defining instructions and a single killing instruction, and having a single defining instruction and several killing instructions. Investigation revealed that 9.74% and 12.1% of all live ranges had the first and second characteristic respectively. One might argue that due to this, there is no significant difference between the two cases and as such it does not matter. It is however pivotal that we investigate how substantial these scenarios are. If we let one byte-cycle (abbreviated BC henceforth) measure the amounts of bytes occupied each cycle, then we may measure how many bytes are wasted in the vector register based on the different scenarios. It is here that the two scenarios vastly differ. Data gathered when using the internal performance tests revealed that on average 196 BC were wasted due to effect the first case, however 1064 BC were wasted due to second case, which is more than 5 times as much. For a combined scheduling and register allocation scheme, it would thus be better to use top-to-bottom scheduling to minimize the amount of register space wasted. The vast difference between the two characteristics can be explained by the nature of shader programs. It is common that vectors use all their components in dot products. Afterwards, it is possible that the x and y component is used for map lookups, which would make it so that the z (and potentially w component) would have died by the dot product, but the x and the y component died by the texture lookup. Furthermore, it is common to use depth, which typically is a single component of a vector, which might make all other components of the vector die early.

5.3.2 Efficiency of a combined scheme

The results of using a combined scheme for single basic-block shaders are presented in Figure 5.9.

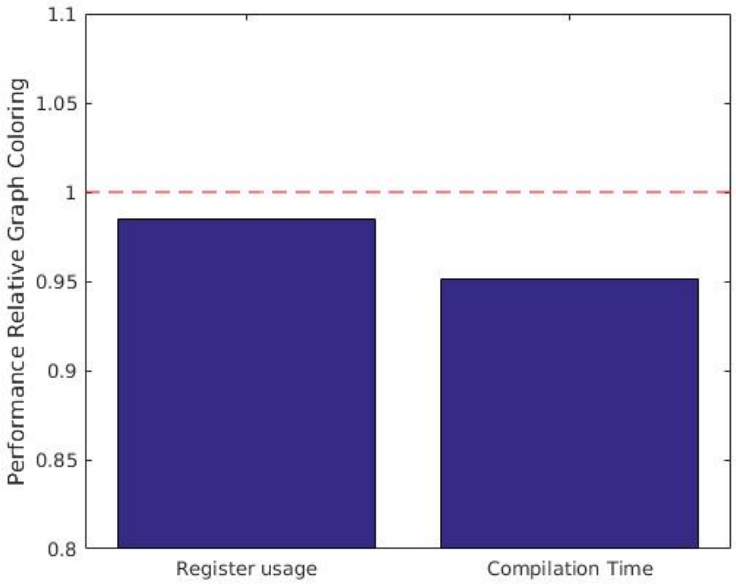


Figure 5.9: Comparison of how the combined scheme performs for single basic-block shaders.

We see that in Figure 5.9, the compilation time decrease is fairly significant. The register usage reduction is a bit misleading, since there are a couple identical shaders in which the combined scheme performs better than graph coloring. Regardless, the two methods are quite similar in terms of register usage.

Since the results presented in this thesis are based on only single basic-blocks, it is likely that the relative compilation time is increased when using more basic blocks. This has got to do with using a conventional allocation technique, such as linear scan or graph coloring, for global variables, which is more time consuming. Furthermore, it is difficult to decrease fragmentation in a combined scheme, since we do not exactly know the liveness of the variables which are allocated. Additionally, we do not know which variables which needs to be allocated shortly. We may force the scheduler to schedule variables which fit into the registers and reduces fragmentation, however this might come at the cost of worse scheduling.

Chapter 6
Conclusions

In this chapter, the key insights of the thesis will be presented. Lastly, the future work will be discussed.

6.1 Summary

As seen in the results, using linear scan for register allocation results in faster compilation at the cost of reduced code quality compared to graph coloring. It is difficult to state that one is superior to the other, since it is fairly situational. If the compiled code is only run once, then it may be worth using linear scan. If the compiled code however is run many times, then the increased code quality will result in better performance.

Using the true criterion of trivial colorability does not seem to offer any improvements, despite the fact that it does find more trivially colorable nodes than the generalized criterion for trivial colorability. The reason for this might be in the characteristics of RIGs and that the case where removing nodes with the true criterion for trivial colorability unblocks the graph simply does not occur.

Generalizing the spill criterion seems to offer improvements to graph coloring. Less spill code was inserted and in general, both the time spent on loading from memory and execution time decreased.

Using subgraph coloring seems to be able to improve graph coloring by reducing the amount of potential spill nodes which are actually spilled during allocation. The improvement would come at the cost of longer compilation time. The increased compilation time was not investigated in this thesis, since the result would be inaccurate regardless.

Using a combined scheme of register allocation and instruction scheduling might prove useful in a GPU compiler. Both the amount of registers used and compilation time was reduced, however this can be misleading since the programs which were compiled only consisted of single basic blocks. In order to make more accurate conclusions, we would need to allow the combined scheme to be used for any program.

6.2 Future Work

There has not been enough time to implement Subgraph coloring on a CPU compiler. Since the rationale holds for both a GPU and CPU compiler, it would be very interesting to investigate whether similar improvements can be found in CPU compilers.

Two of the most crucial elements to the combined scheduling and register allocation scheme can be utilized by any register allocation technique for vector registers. These two are the live range splitting for fragmentation and the preemptive solution to avoid fragmentation. For a CPU, the choices we have when spilling are simply to either rematerialize or load spilled instructions. In a GPU compiler, we also have the possibility to perform split live ranges to ensure better packing of the variables if we fail to allocate a potential spill node. In fact, when using VLIW instruction words, it is not impossible that we can reallocate a register at a low cost. This may happen if there is an instruction word which we can issue a move instruction, which is a very cheap way to avoid spilling. The scheme presented to preemptively reduce amount of fragmentation could also be adjusted to be used in the register allocator for linear scan or graph coloring. In theory, it would be even better to do so during normal register allocation, since we then have access to all liveness information.

Bibliography

[1] AnandTech. Arm's mali midgard architecture explored. http://www.anandtech.com/show/8234/arms-mali-midgard-architecture-explored. Accessed: 2016-05-18.

[2] ARM. Neon-arm. http://www.arm.com/products/processors/technologies/neon.php. Accessed: 2016-05-07.

[3] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. *Languages and Compilers for Parallel Computing: 19th International Workshop, LCPC 2006, New Orleans, LA, USA, November 2-4, 2006. Revised Papers*, chapter Register Allocation: What Does the NP-Completeness Proof of Chaitin et al. Really Prove? Or Revisiting Register Allocation: Why and How, pages 283–298. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[4] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, May 1994.

[5] Gregory J. Chaitin. Register allocation & spilling via graph coloring. *SIGPLAN Not.*, 17(6):98–101, June 1982.

[6] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Comput. Lang.*, 6(1):47–57, January 1981.

[7] Dmitri S. Ivanov. Register allocation with instruction scheduling for vliw-architectures. *Programming and Computer Software*, 36(6):363–367, 2010.

[8] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[9] Roberto Castañeda Lozano, Mats Carlsson, Frej Drejhammar, and Christian Schulte. *Principles and Practice of Constraint Programming: 18th International Conference,*

CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings, chapter Constraint-Based Register Allocation and Instruction Scheduling, pages 750–766. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[10] Takuya Nakaike, Tatsushi Inagaki, Hideaki Komatsu, and Toshio Nakatani. Profile-based global live-range splitting. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 216–227, New York, NY, USA, 2006. ACM.

[11] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2007.

[12] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, September 1999.

[13] Jonas Skeppstedt. *An Introduction to the Theory of Optimizing Compilers*. Skeppberg, 2012.

[14] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. *SIGPLAN Not.*, 39(6):277–288, June 2004.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2016-06-03

EXAMENSARBETE Investigating Different Register Allocation Techniques for a GPU Compiler
STUDENT Max Andersson
HANDLEDARE Jonas Skeppstedt (LTH), Dmitri Ivanov (ARM)
EXAMINATOR Flavius Gruian (LTH)

Optimering av användning av registerminne för grafikprocessorer

POPULÄRVETENSKAPLIG SAMMANFATTNING **Max Andersson**

Register-allokering är en av de viktigaste delarna för optimerande kompilatorer. En förbättring av register-allokering skulle leda till inte bara ökad prestanda, utan även minskad energiförbrukning av datorer. I detta arbete har olika tekniker undersökts för att utföra register-allokering, samt nya tekniker utvecklats för att hantera problem som uppstår för grafikprocessorer.

En *processor* är datorns beräkningsenhet och utför alla instruktioner i ett program. En *GPU* är en processor som hanterar visning av grafik på elektroniska enheter, bland annat på datorer och telefoner. För att kunna visa grafiken snabbt är det skillnad på en vanlig processor och en GPU. I en vanlig processor finns det register som är kapabla att lagra ett värde. Till skillnad från vanlig beräkning använder grafik *vektorer* väldigt mycket. En vektor är flera tal som tillsammans beskriver något, exempelvis kan tre värden användas för att beskriva en position i ett spel. Därför använder en GPU *vektor-register*, vilket är register som är kapabla att lagra flera värden. Det är då möjligt för processorn att exempelvis göra beräkningen "addera värde ett och tre i register 1 med värde två och fyra i register 2 och lagra resultatet som värde ett och två i register 3".

Program skrivs normalt inte i *maskinkod*, som en processor förstår, utan i *högnivå-språk*, vilket är språk mer lika våra mänskliga språk. Programmen måste då omvandlas till maskinkod, vilket sker i en process som kallas *kompilering*. En nödvändig process i kompilering är *register-allokering*, vilket tilldelar varje variabel i ett program ett register att lagras i. Dålig register-allokering leder till försämring av prestanda. Att hitta en optimal allokering är väldigt svårt, så därför används oftast ungefärliga metoder för register-allokering. De två vanligaste metoderna är *Graph coloring* och *Linear scan*.

I mitt arbete jämfördes hur graph coloring och linear scan presterar med vektor-register. Vidare utvecklades teori som generaliserar Chaitin-Briggs algoritm, vilket är det vanligaste sättet att allokera med graph coloring. Slutligen undersöktes hur möjligheten att kombinera register-allokering med instruktion-schemaläggning, en tidigare optimering, påverkas av vektor-register. Graph coloring gav 2% snabbare kod, men 12% långsammare kompilering relativt linear scan. Teknikerna jag utvecklat minskade tiden det tog att köra programmen med ungefär 0.5% i snitt. Tänk exempelvis på att ett datacenter förbrukar lika mycket energi som 180000 hus[1]. Tiden som program körs kan löst översättas till elförbrukning. Energin som sparas med teknikerna skulle därför kunna försörja 1000 hus med el per datacenter om samma resultat skulle uppnås där.

[1] http://science.time.com/2013/08/14/power-drain-the-digital-cloud-is-using-more-energy-than-you-think/