國立臺灣大電機資訊學學院資訊網路與多媒體研究所

碩士論文

Department or Graduate Institute of Networking and Multimedia

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

應用於行動繪圖處理器的著色語言編譯器後端之暫存器配置與指令調度研究探討

Shading Language Compiler Backend for a Mobile GPU: Case Study on Register Allocation and Code Scheduling

蘇建豪

Jian-Hao Su

指導教授：廖世偉 博士

Advisor: Shin-Wei Liao, Ph.D.

中華民國 98 年 7 月

July, 2009

誌謝

時間過的好快,轉眼間人生又過了一個階段-碩士,許多夜晚、project 等成為了這段時間的代表,在這段日子裡,我得到了很多人的幫忙與協助,讓我在其中學習成長,在此我衷心的感謝這群同伴。

首先我必需感謝簡韶逸老師對在硬體上對我的指教與容忍,使我能夠循序的完成工作。也感謝廖世偉老師的指導,讓我對於我的研究有充分的了解,教導我如何做研究。

當然很重要的要感謝實驗室的成員,讓我們實驗室不時充滿歡樂的氣息,也對於很多不懂的地方給了很多幫助!強烈直率風格的大師,造謠王冠軍的 Pete 和歐文學長,沈睡的 mint,忠肯的禮俊,神人 QQ 學弟,嘉恆學長、以及 Peegoo、injan、便當魚,漢興感謝你們陪我做過數個 Project 與這兩年寶貴的時間!

最後感謝很照顧我的好朋友(院代,glare,軍長,7j 與吳大師…)與永遠支持我的父母與妹妹!謝謝你們!

摘要

今日 3D 電腦繪圖越來越顯得重要，電腦遊戲、虛擬實境等廣泛的使用 3D 電腦繪圖。對於電腦及行動裝置上，3D 繪圖加速已經是一個必備的功能。

繪圖處理器的產生代表者 GPU 接管了幾乎所有 3D 繪圖相關的工作，在新一代的繪圖處理器中，甚至提供了可程式性以應付更複雜的效果需求。著色語言便是提供繪圖處理器可程式化的關鍵，它抽象化了硬體同時提供更多彈性 程式設計師因此不必使用組合語言來對繪圖處理器寫作程式

著色語言編譯器將著色語言轉成硬體的執行格式。我們開發了一個針對 OpenGL ES 著色語言與 Media IC & System and DSP IC 實驗室開發的行動繪圖處理器的著色語言編譯器。

此論文中，我們敘述了該著色語言編譯器的實作，並對相關硬體的後端作了探討，我們針對該繪圖處理器的架構提出了最佳化方法–在指令調度方面，編譯器實作了資料前饋的偵測與程式碼的產生，另外針對著色語言與硬體特性亦提出了一個暫存器配置方法，目前實驗結果顯示最高可以省下 28%的使用量

ABSTRACT

3D computer graphic has become more and more important today, applications like games, virtual reality and so on prevail the way for 3D computer graphic. Accelerator for 3D Graphic has become a necessary component in computer and mobile device.

GPU is the new name of graphic hardware since it has taken nearly all workload of 3D graphic processing form CPU. As the evolution of GPU, programmability is now provided for more complex computation. Shading language is the key to release the power of GPU programmability. It provides the high hardware abstraction and more flexibility programmability rather than programming GPU in low-level hardware related assembly.

Shading language compiler translates shader codes to low level instructions. We have developed an OpenGL ES shading language compiler for a mobile GPU developed by Media IC & System and DSP IC Lab at NTU.

In this thesis, we describe the compiler we have done and study the backend of shading compiler for the mobile GPU hardware. Optimization methods are proposed to improve the performance of current hardware. Code scheduling in compiler detects data forwarding and generates code to use hardware forwarding path. Register allocation supports packing to use remain part of vector type in shading language. In current experiment, at most 28% register usage can therefore shrink.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

1.1 Motivation

3D graphic has played a major role nowadays. More and more game, movie, and other applications use 3D graphic technology to attract your attentions. It thrives in recent years because of requirements from users and hardware evolution. Graphic Processor Unit (GPU) has born to bring the colorful and realistic world. To utilize the power of GPU, shading language is provided to programmer for higher hardware abstraction and a more flexible programming model.

Media IC & System and DSP/IC Lab at NTU have studied GPU hardware for several years. A mobile GPU [21][22] has been developed by them. We are here to develop shading language compiler and explore the potential of hardware.

Figure 1-1 Cape NO 7

Figure 1-2 Crisis

1.2 Introduction

3D graphic is a branch of computer graphic used to generate realistic image or visual effect. Users build the 3D models by primitives formed by triangles, lines, points and so on as the data of graphic processing. Through graphic pipeline, the results are generating to

either display or to store.

1.2.1 Graphic Pipeline

Processing to get 2D images from 3D models and related configuration is called rending. Several algorithms are developed to render a picture such as Ray Tracing [26], “Radiosity” [23], and Rasterization. Rasterization is the one widely used among them. The algorithm of rasterization shown in figure 1.3 is called as graphic pipeline. Users input vertexes, primitives (which was formed by vertex. E.g. triangle is consist by 3 vertexes) and settings into graphic pipeline. After processing, the images formed by pixels are generated as output. Functions inside pipeline do special tasks which are explained in the following paragraph..

1. T&L (Transform & Lighting): Transform converts spatial coordinates, which in this case involves moving three-dimensional objects in a virtual world and converting the coordinates to a two-dimensional view. Clipping was used to draw things that might be visible to viewer; it can therefore improve performance by removing the invisible object. Lighting takes light objects in a virtual scene and calculates to result color of surrounding objects as the light falls upon them.

2. Primitive Setup: Primitive Setup collects vertices and converts them into primitives (lines, triangles). Information is generated that will allow later stages to accurately generate the attributes of every pixel associated with the primitives.

3. Rasterization: Rasterization is responsible for taking an image described in a vector graphics format (shapes) and converting it into a raster image (pixels or dots) .

4. Pixel processing: Pixel processing processes color of a pixel (aka Texture operation). Filtering is also the task of pixel processing

5. Frame Buffer Blend: This step processes multi-frame effect, such as alpha blending, BitBLT so on.

Other methods can be added into graphic pipeline to get effect you want. “Software Rendering” do all things on CPU.

1.3 Graphic Processing Unit

From Figure 1-3, we can see the evolution of rasterization graphic pipeline. Render by software was used originally. But later graphic pipeline was implemented by specific circuit and named as fix-function pipeline. T&L is the technique that makes Graphic Processing Unit born. Before popularization of DirectX [9] 7, T&L was computed by CPU.

But Nvidia Geforce 256 offloads T&L works to graphic accelerator and graphic accelerator was named Graphic Process Unit since GPU takes all jobs of graphic pipeline. Later, the arrival of DX 8 has provided programmable graphic pipeline to us. This turns a new leaf of the GPU; programmers are able to produce much more the effect they want then.

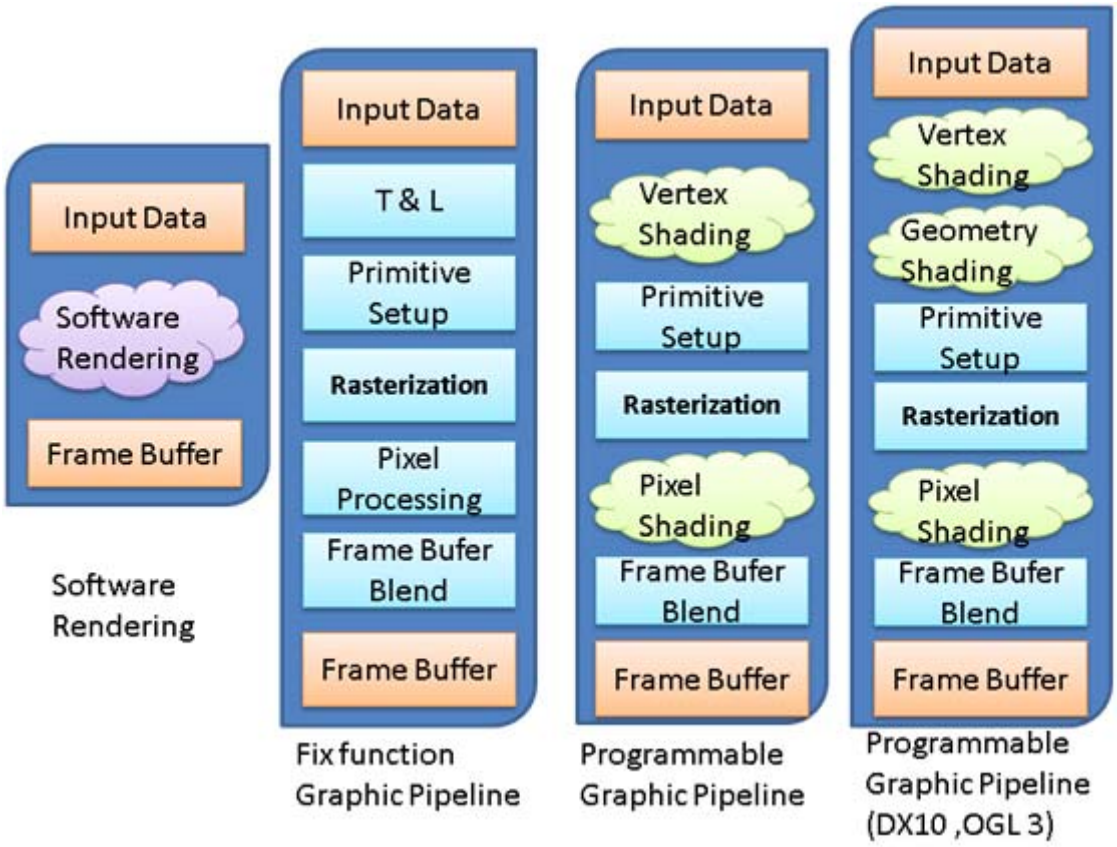


Figure 1-3 graphic pipeline

1.4 Shading Language

A shader in the field of computer graphics is a set of software instructions, which is

used primarily to calculate rendering effects on graphics hardware with a high degree of flexibility. Shading languages was developed from the work of Cook, who described how shade trees [16] could provide a flexible, programmable framework for shading computations. The RenderMan [28] is the most common shading language for production-quality rendering. Except offline render system like RenderMan which render production-quality image, real-time rendering systems is focus more on real-time requirement. Shading languages for real-time rendering systems is used to provide high hardware abstraction and the ability to program the GPU. Because the strong computation power and programmability of GPU today, shader is not merely used for graphic. You can also use it for science computation or something else, works of this kind are named General Purpose GPU- GPGPU. Because of some constraints in older GPU which make it hard for GPGPU [16], the new architecture along with CUDA [29] was later proposed for GPGPU computing.

1.5 Contribution

In this work, we implement shading language compiler and also study the register allocation and code scheduling for current hardware. We have found that traditional register allocation algorithm may lose some performance benefit in shading

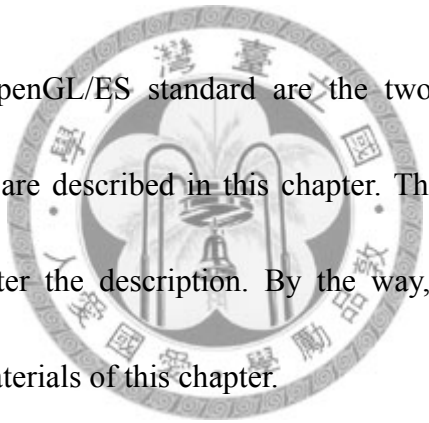
language. Therefore we propose other register allocation method –Tetris Allocation, it can pack remain part of vector variable to fully use register allocation. Experiments tell us that we can potentially reduce register usage for some case. Another thing we have done is the modification of scheduling algorithm to support the forwarding method provided by hardware.

1.6 Organization:

The overall contents in this thesis are organized as follow. Chapter 1 gives very brief introductions. Chapter 2 introduces GPU hardware and OpenGL ES [25] system. And then chapter 3 describes the inner of compiler we implement. Tetris Allocation and code scheduling methods are also described in this chapter. Experiments are in chapter 4 to show efficiency of the optimization methods. Finally, we gave conclusion and future work in chapter 5.

Chapter 2

Preliminaries



GPU hardware and OpenGL/ES standard are the two important elements in 3D graphic. The characteristics are described in this chapter. Their relations to our compiler work are also explained after the description. By the way, other implementations and related works are also the materials of this chapter.

2.1 OpenGL /OpenGL ES

OpenGL is a royalty-free, cross-platform API for full-function 2D and 3D graphics. It creates a flexible and powerful low-level interface between software and graphics acceleration. The “ES” is the version targeted to embedded system, which was modified and enhanced from OpenGL desktop/workstation. For OpenGL ES, there is EGL™

specification for portably binding to native windowing systems.

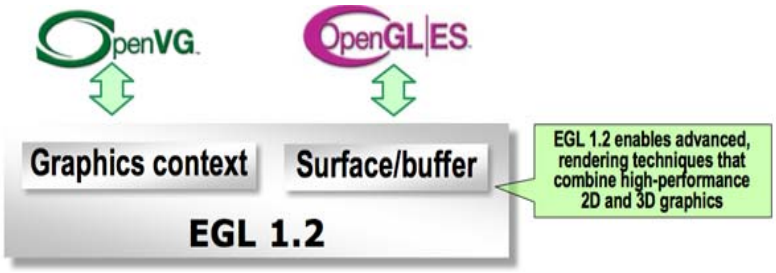


Figure 2-1 relation of OpenGL ES & EGL Source: OpenGL ES 2.0 website

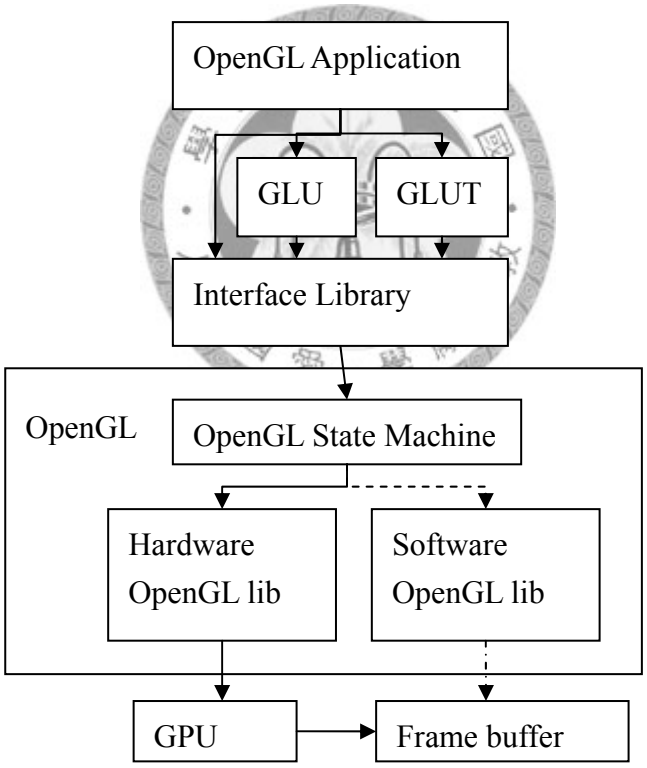


Figure 2-2 overall infrastructure of OpenGL API

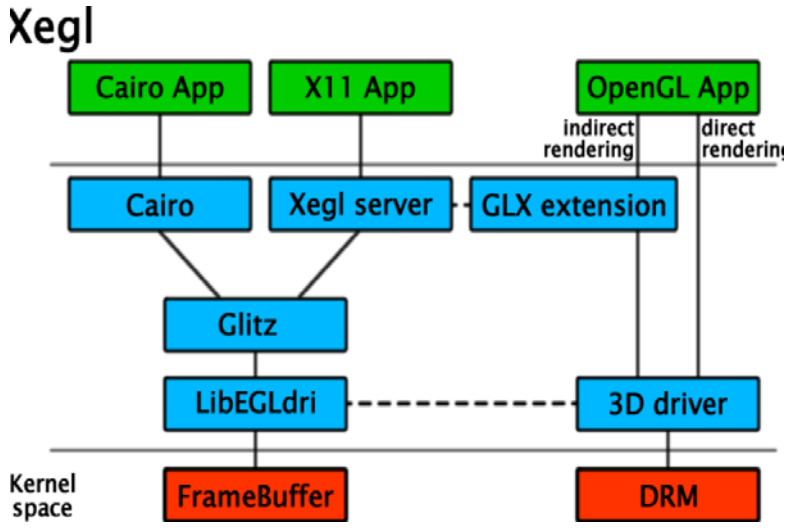


Figure 2-3 Xegl windows system

As shown in figure 2-2, OpenGL is an API which roughly contains a frontend and a backend. Applications use the frontend to compile application program into an assembly-like language. This assembly-like language assumes the underlying hardware as a virtual machine with a number of states, which are visible to the application. The OpenGL backend then converts this assembly-like language into the instruction set which can run on the GPUs.

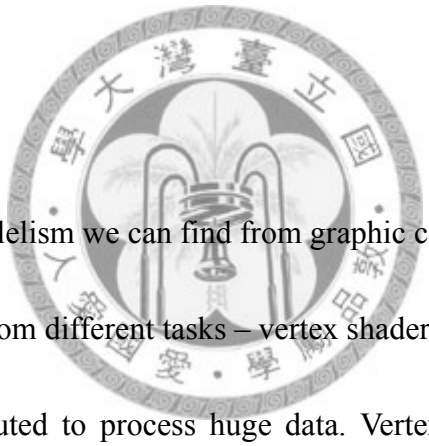
From the view of window system, as shown in figure 2-3, several layer are needed to run OpenGL. “GLX extension” in figure is the interface implementation of OpenGL. “3D driver” and “LibEGLdri ” are the drivers provided by vender. Applications use OpenGL will process by “GLX_extension” and may use the hardware ability to accelerate. It is fun to see that 2D and X applications are possible to accelerate by hardware. Take example, Glitz, Glitz is a software library for 2D graphics which provides hardware acceleration.

Notice that vendors often implement OpenGL on their own instead of using the built-in OpenGL to get higher performance.

2.2 GPU Hardware

GPU are originally designed for graphic, therefore the features of graphic deeply affect design of GPU. Graphic is the computation full of parallelism.

2.2.1 Parallelism



There are four kinds of parallelism we can find from graphic computation.

- TLP: The parallelism from different tasks – vertex shader and pixel shader.
- DLP: Program is executed to process huge data. Vertexes are processed by vertex shader, and pixels are processed by pixel shader.
- SIMD: There are up to four channels in vertex datum or pixel datum, because color are presented by r,g,b,a elements and ordinate are presented by x,w,z,w.
- ILP: Independent instructions can be executed together.

2.2.2 Architecture

GPU architecture is designed to use the parallelism. Designer can use VLIW or superscalar architecture to utilize different type of ALU in GPU. SIMD always exists inside PEs (Processing Elements). DLP is explored by the number of PEs, which is the major computational resource. TLP is the key for unified shader shown in figure 2-4(b). When some PEs are idle due to the unbalance workload between vertex and fragment shader, the thread controller in unified shader dispatches some workload to the idle PE. The result is the improvement of performance,

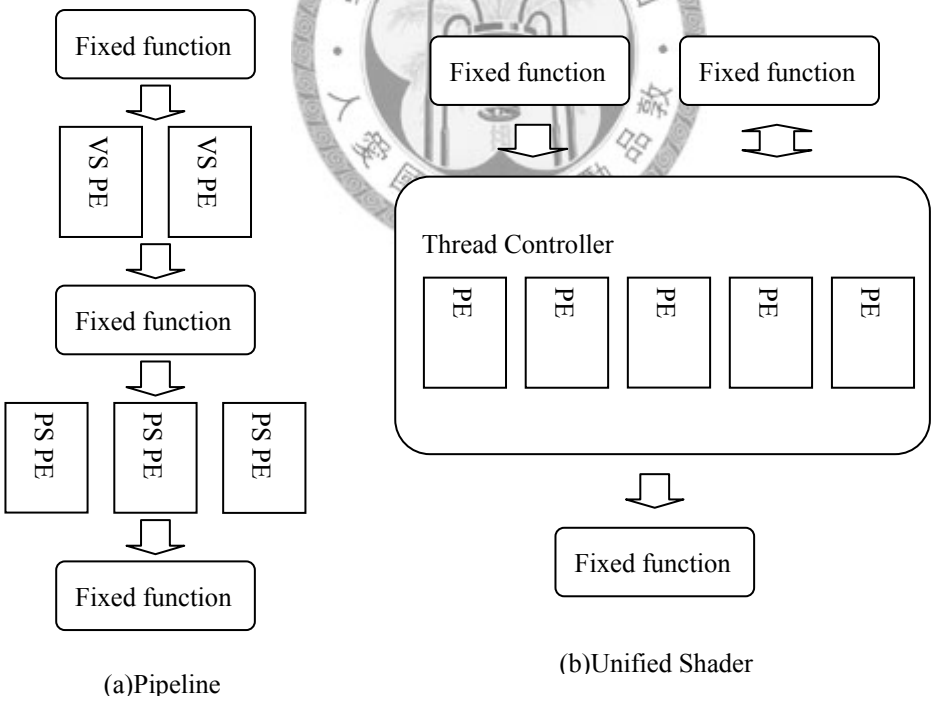


Figure 2-4 Unified Shader

2.2.3 Our Hardware

Comparing to GPUs in desktop or workstation, mobile GPUs consider more in power than performance. Media IC & System and DSP IC lab at NTU have studied GPU for several years. They have developed a mobile GPU. The features are listed below:

1. Single core is a 2-way VLIW, 4-stage pipeline, 4-channel SIMD and there are special instructions to support for codec processing (ME).
2. It uses OpenGL ES 2.0 as its implementation standard, but it has its own ISA, rather than vs, ps ARB ISA.
3. There are 2 cores to form an unified shader. One shader for vertex shader and one for fragment shader, but each can take another job depended on the workload.
4. CMA registers and hardware thread are used for low-power yet good performance.

2.2.3.1 Stream Processing model:

Stream sperates a program into kernels and streams. Streams are the data to process, and kernels are the codes to exectue streams. Vertexs shader and Fragement shader are two kernels. Cordinates, Colors are streams. In our hardware,the hierarchy of memory is splite as two. CMA handles Temporary register(t), Constant Registe(c), StreamOut register(u), Input register(v). The Const Registers are related to OpenGL ES constant and uniform variables. The Input Registers ared related to the inputs of vertex shader. The

output registers of fragement is Stream Out Register. Texture instructions are the kind of instructions that can process data loaded directly form memory. Following figure is the processing model of 1 shader. .

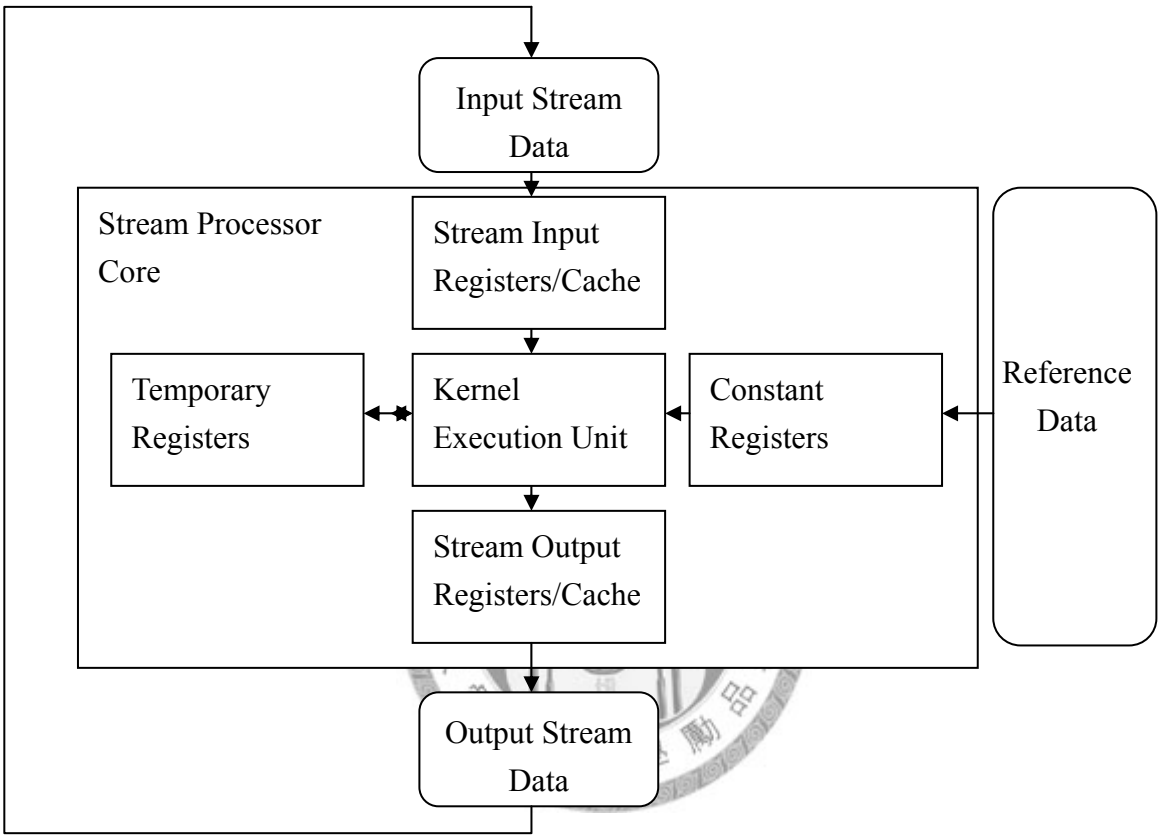


Figure 2-5 Stream model

2.2.3.2 CMA (Configuarable Memory array)

The register file is named as CMA (Configurable Memory Array), because it can configure its memory (the registers for a thread) for different applications. It has 8 banks and 16 long-words each bank, totally 128 long words in CMA. Following is a simple configuration, 4V4P means it uses 4 banks for vertex shader and 2 banks for pixel shader

while 6V2P means that 6 banks for vertex shader and 2 for pixel shader. Thread capacity means that how many treads are; 8V8P means 8 vertex threads and 8 pixel threads. We can count registers that 1 thread use.

The number is banks by a thread* long words per banks (16 here)/thread.

Therefore 16V8P means that
4*16/16 = 4 registers per thread
4*16/8 = 8 registers per thread

Physical Bank Configuration	Thread Capacity
4V4P	8V8P
	8V16P
	16V8P
	16V16P
6V2P	12V4P
	12V8P

Figure 2-6 CMA configure

It is needed to notice that CMA is shared by all cores.

2.2.3.3 Thread Controller

The cores in this hardware are not target for vertex or pixel, this means that core can execute vertex or pixel program depended on the workload. In this hardware, they propose some skills – AMT、ATS, AMS (Adaptive Multi-thread Switch) means that is will change thread when it meet a long latency instruction like textureload. ATS (Adaptive Task Switch) is a thread mechanic to balance the workload of two cores.

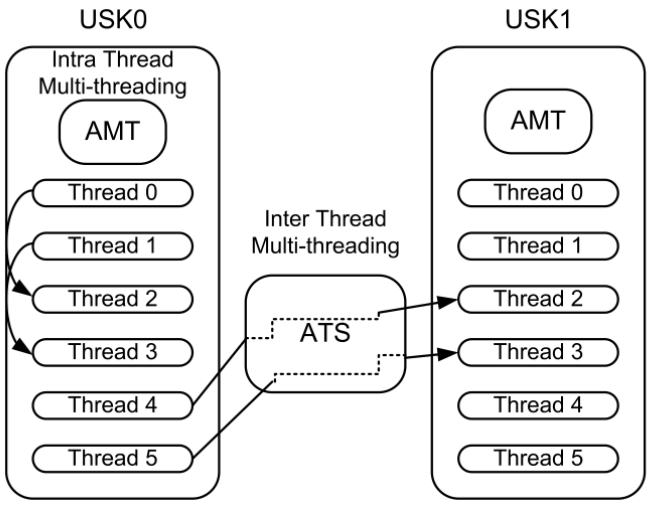


Figure 2-7 ATS

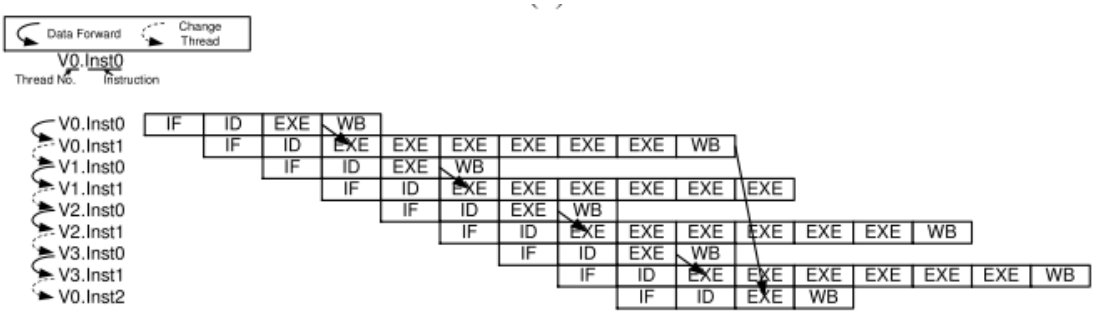


Figure 2-8 AMS

2.3 Shading Language Compiler and Related Work

2.3.1 History

In real-time rendering systems, support for user programmability has evolved with the underlying graphics hardware. The UNC PixelFlow[5][6] and its accompanying PFMan

procedural shading language demonstrated the utility of real-time procedural shading capabilities from 1992 to 1998. Commercial graphic hardware was configurable that time, but not user programmable. Multi-pass rendering techniques [1] are used by the related system. To program graphics hardware, higher-level tools are provided to user.

Graphics architects began to incorporate programmable processors into both the vertex-processing and fragment-processing stages of single-chip. The Stanford RTSL [3] system was designed for this type of programmable graphics hardware. Recent generation of PC graphics hardware continue the trend of adding additional programmable functionality. Of greater significance for languages and compilers, the vertex processor in some of these architectures departs from the previous SIMD programming model, by adding conditional branching functionality and the vertex processors. .Since branching capability cannot be easily supported by RTSL, NVIDIA and Microsoft collaborated on the design of a new language. Cg [4] and HLSL was born then.

2.3.2 Related Work:

The Stanford RTSL system is a shading compiler designed for Nvidia register combiner architecture (Geforce 1~Geforce 3). Shader codes without branch are compiled to fit the hardware. The internal implementation is to find the correct operations for VLIW

slots form DAG representation. Specialized methods of code scheduling and register allocation are used to utilize the ability of hardware. Issue related to performance is also revealed in this work.

The patent of DX9 shader compiler describes the internal and compilation passes. Code scheduling and register allocation are still mentioned in this work. DX 9 shader compiler considers the register usage when scheduling. Optimizations are separated by 2 types, scalar and vector. The compiler of Microsoft generates some standard profiles to abstract the hardware layer. Vender needs to write a JIT to translate profile output to codes used by its hardware.

AMD in CGO 2008[27] presents slides for what they have done to optimize performance. It first de-optimized what Microsoft compiler has done. Optimized are somehow not clearly described. But fast compilation is the thing they focus. Register allocation and code scheduling for hardware are important issues.

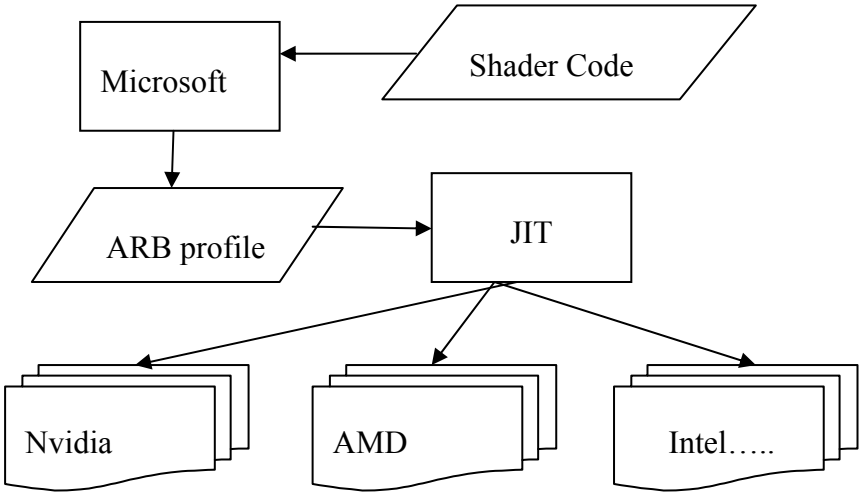


Figure 2-9 shader compiler processing of DX9

2.3.3 Features in Shading Language

Then we briefly compare real-time rendering shading language with general purpose programming language. The features are shown below.

1. It doesn't support recursive and pointer.

2. Branch ability has limited support.

3. Specialized built-in function for graphic and vector type computation.

4. The program limited the memory accessing. Input and output are constrained.

Shader compiler implementation is affect by the feature. Since less control, it is much simpler than general purpose compiler.

2.3.4 Compiler for Hardware

In this work, shader codes are directly into binary related to hardware. The work of us is to implement and proposes some optimization to fully utility hardware. In current system, compiler directly translates shader code to ISA of our hardware as shown in figure 2-10. Since the ISA is not compliant to ARB, this is the efficient way for implementation and user usually does care about the low-level of our hardware.

As RTSL, we do optimizations for the graphic hardware. CMA is the thing we need to

consider more. Since CMA is shared by all thread and core, it is possible to lift the number of threads by reduce register usage. We need to generate code with the possible best CMA configure also. Then thread controller with AMS, ATS can benefit from the extra threads. Data forwarding is another thing that compiler has to do since data forwarding path is existed but not by handle automatically by hardware. We can therefore have a result that major things in this work are register allocation and code scheduling.

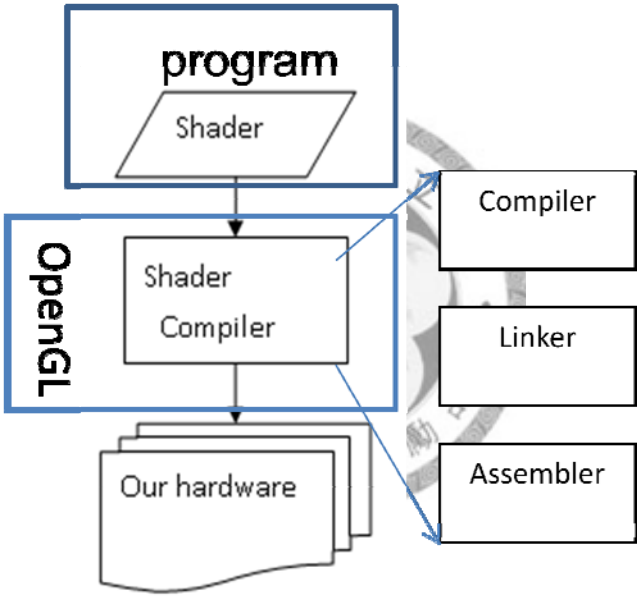


Figure 2-10 Whole system

Chapter 3

Compiler Architecture and Algorithm

Shading language compiler is different comparing with general CPU language compilers since it is designed only for shading language. More clearly to say, optimizations need to consider the features of shading language and hardware. In this chapter, we describe the compiler internal first and later describe the special optimization methods for the hardware architecture.

3.1 Overview:

Figure 3-1 lists the internal design of our shading language compiler. We use the

frontend from Vincent 3D [14] 2.x as our parser and code generator. Codes generated from the Vincent 3D front-end are modified from ARB instruction to the self define ISA. Passes for optimization and analysis are added after the code generator.

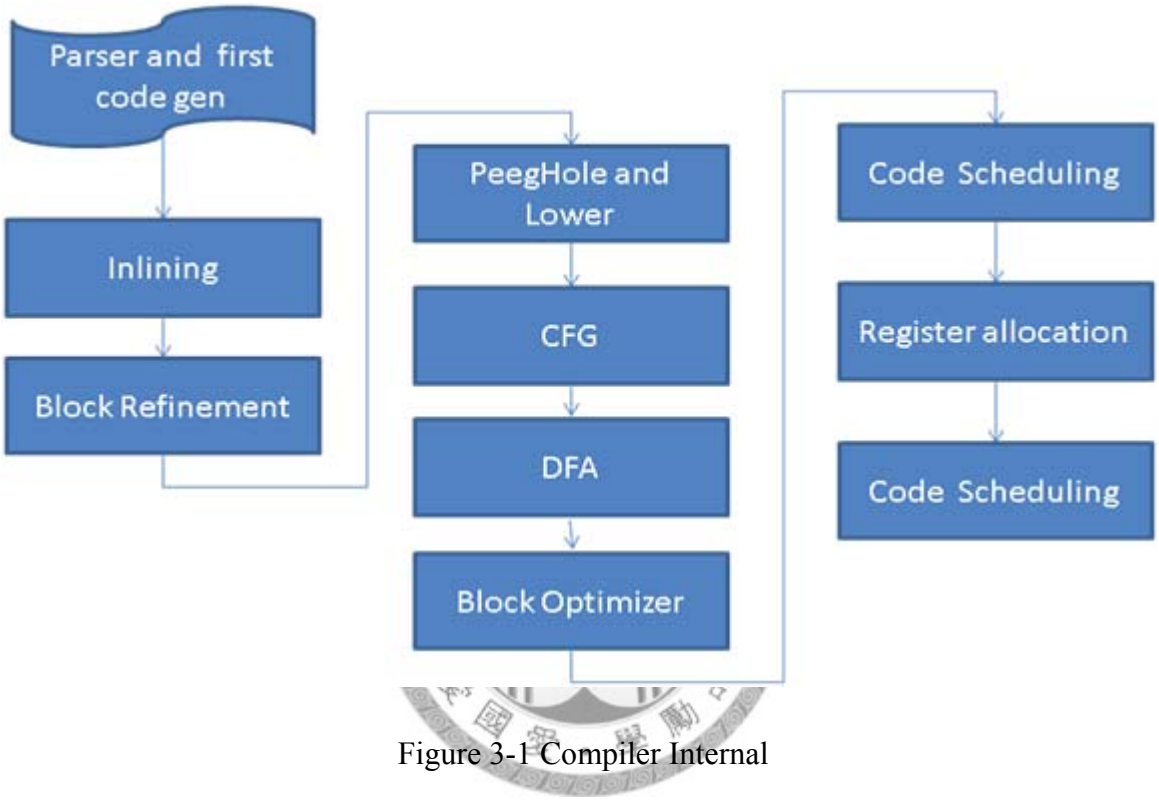


Figure 3-1 Compiler Internal

The original Vincent 3D front-end (the compiler and parser.) generate codes while parsing. After initial codes was generated, Instruction List is the Intermediate Representation for processing. Instruction List contains 1. Instructions representation contains op-code, register information. 2. Block is a list of instructions that represent a basic block. 3. BlockList is a list of Block which represents a program. Our major work starts by use original IL form. The major works are listed below.

3.1.1 Passes

- Inlining Functions:

No stack frame and no call instruction in our GPU, the function is limited supported. User can't t write recursive function. Compiler here embedded the function body to the place function was called.

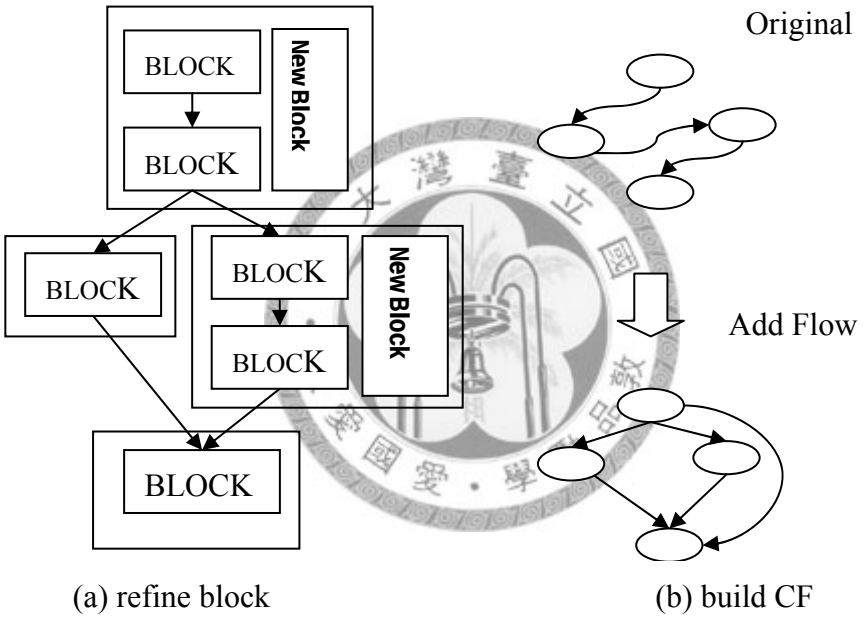


Figure 3-2 Refine Basic Block and Build Control Flow

- Refine Basic Block:

In this pass, basic blocks without instructions are removed. Blocks without branch or jump and are aggregated to a bigger block. The reason to aggregate bigger block is because there are more instructions to avoid stall caused by dependence.

- Lower:

There are some pseudo or high level instructions for some purposes (such as simplify the complexity of code generation and so on), they can't run by target hardware directly. In this pass, instructions that can't run by hardware are translate to instructions can directly run by hardware (called as low level instruction).

- Build Control Flow:

Control flow graph is then built after refining basic blocks. As shown in figure 3-3, the information of control is known after this pass. The control flow information is used later by data flow analysis.

- Data Flow Analysis:

Data Flow Analysis analyzes the information of data (like which variable is used, which is dead) through blocks on flow graph. Currently we use live-variable analysis to collect information use for Block Dag.

Live-variable analysis is used to calculate variables that are live at the exit from each program point. This is used for some optimizations.

- Block DAG(Block Optimizer):

The instructions are transforming to Direct Acyclic Graph representation. We then use read/write relation of instructions to build DAG. Optimizations like common

sub-expression elimination, local deal code elimination and so on are done in DAG form. For example, in the processing of building DAG, we can also remove some common sub-expression because common sub-expressions have same node in DAG. Something like register renaming is done in this pass since we build a new node when destination registers in instruction is difference.

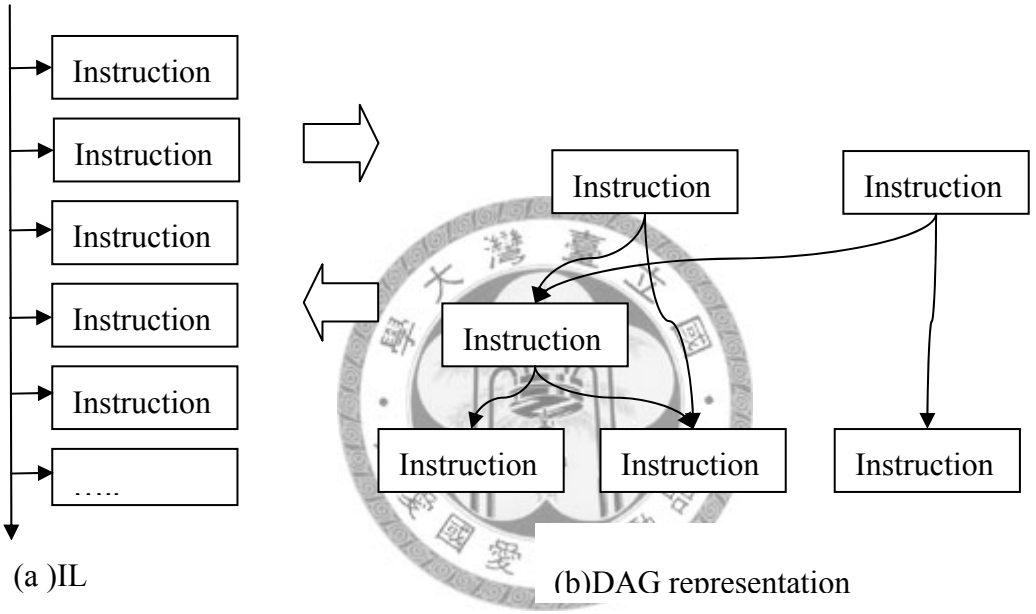


Figure 3-3 DAG Representation

Optimizations for our hardware architecture are called backend. In our work, the major pass in our work are listed in following.

- Code scheduling:

Code scheduling changes instruction order to reduce some overhead like Nops and hardware stall. It can also discovery ILP to fit something like VLIW and help aggressive

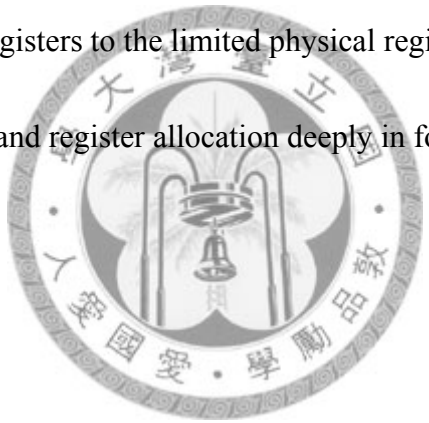
hardware design (super-scale than have more than 1 execution unit).

In this work, we have use 2 pass code scheduling. The first pass detects data-forwarding. And the later does the general code scheduling. Nops are inserted in second pass code scheduling.

- Register allocation:

After middle-end and before register allocation pass, the register is called virtual register that represents a variable and some renaming information. Register allocation maps the unlimited virtual registers to the limited physical register.

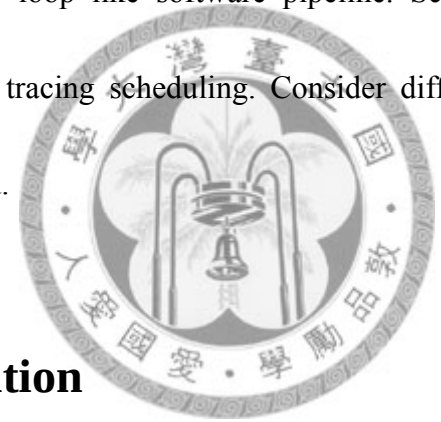
We discuss code scheduling and register allocation deeply in following sections.



3.2 Code Scheduling

Code scheduling also known as instruction scheduling is a compiler optimization used to improve instruction-level parallelism, which improves performance on machines with instruction pipelines, very long instruction word and so on. By means of rearranging the order of instructions, pipeline stall can be avoided maximally.

There are several types of scheduling method. Scheduling in a block like list scheduling. Scheduling for loop like software pipeline. Scheduling across blocks and scheduling for VLIW like tracing scheduling. Consider different situation we may use different scheduling method.



3.2.1 Implementation

We modify list scheduling algorithm for target hardware. The algorithm also takes care of instruction clocks. In following content, we don't talk about the clocks of instruction and how to model timing of instruction because it depends on target hardware. But it is still important for implementation. By the way, list scheduling is a simple and flexible algorithm to modify for different proposes. AMD uses a modified listed scheduling for their VLIW hardware.

3.2.2 Dependence Relation

The rearrangements will fail if the dependence relation is violated. Several dependence relations are described below.

RAR: Previous instructions read data. Later instructions read same data. Reorder instructions having RAR dependence will not cause error.

RAW: Previous instructions write data. Later instructions read same data. This also called true dependence.

WAR: Previous instructions read data. Later instructions write same data.

WAW: Previous instructions write data. Later instructions write same data.

If we reorder instruction, dependence relations such as WAR, RAW, WAW are kept to avoid wrong result. Other relations are introduced when necessary, for example, controls dependence is defined to keep the correct control flow. Compiler can also define the relation when it is required.

Scheduling algorithm starts form build the dependence information of instruction. The function can be done by following algorithm.

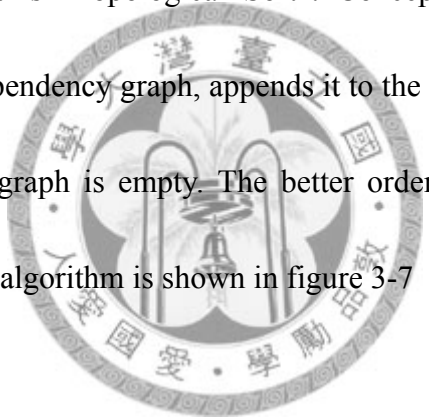
```
DAG = record {
Nodes, Roots: set of integer;
Edges: set of (interger x integer),
Label: (integer x integer) -> integer
}
Procedure Build_DAG(m . Inst) returns DAG
        m:integer
        Inst: in array[1..m] of LIRInst
begin
        D := <Nodes:Φ, Edges: Φ ;Roots: Φ>: DAG
Conf: set of integer
j,k: integer
||determine nodes, edges, labels and roots of a basic-block scheduling DAG

for j:= 1 to m do
    D.Nodes ∪ = {j}
    Conf : =Φ;
    for k := 1 to n do
        if Conflict(Inst[k], Inst[j])then
            Conf ∪ = {k}
        fi
    od
    if   Conf =Φ then
        D.Roots ∪ = {j}
    else
        for each k ∈ Conf do
            D.Edges ∪ = {k->j}
            D.Labels(k,j) := Latency(Inst[k] ,1 ,Inst[j] ,IssueLatency+1);
        od
    fi
od
return D
end || Build_DAG
```

Figure 3-4 Build_DAG Source : Advanced compiler design and implementation ch 9

The function “Latency” returns the clock interval between start of inst[K] and issueLatency+1 clocks after executing inst[j]

3.2.3 List Scheduling

In our work, list scheduling is the method we currently used for the reason of conservative design in flow control. List scheduling is the simplest algorithm for code scheduling. The spirit of it is “Topological Sort”. Conceptually, it repeatedly selects (schedules) a node of the dependency graph, appends it to the current scheduled instruction. Finally it terminates if the graph is empty. The better order of instructions is therefore generated. The flow chart of algorithm is shown in figure 3-7



3.2.4 List Scheduling for VLIW:

When we don't consider aggressive design like trace scheduling and region scheduling, list scheduling is a very simple algorithm to support VLIW. The change of instruction scheduling is selecting independent nodes in dependence DAG. In figure 3-5, B,C can be choose for VLIW slots if they fit the requirement of slots attribute. In following figure 3-5, independent nodes are schedule to some instruction, different slots. If there are dependent

instructions, it will schedule to different VLIW instructions.

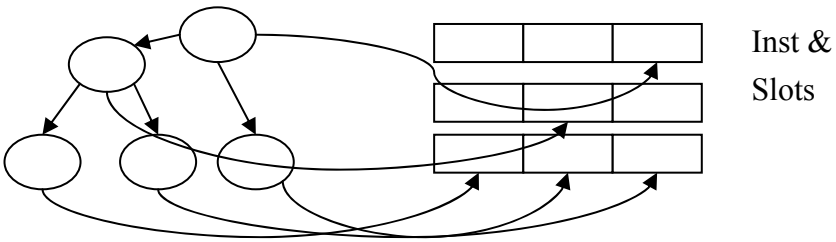


Figure 3-5 List Scheduling for VLIW

3.2.5 Data Forwarding Detection by Compiler

In the situation we face, compiler takes the responsibility to detect hazards and insert non operation to keep hardware work properly. Data hazards cause by data dependence can be prevent by nop or forwarding data. Data forwarding gives ALU the needed data immediately to prevent stall. Hardware provides a mechanism to forward data.

3.2.5.1 Data Forwarding Mechanism of Hardware

Following table shows how to use data forwarding mechanism.

without hardware forwarding	software data forwarding
add t2 t1,t0 NOP NOP add t3 t2,t0 …….	add GroupA t1,t0 add t3 GroupA,t0

Table 3-1 example of forwarding mechanism

GroupA is the Flip-Flop of the execution unit of Add. Different instructions have their own

execution unit, SOP instruction has execution unit named GroupC. Compiler here needs to do data forwarding detection and change instructions to use FF of ALU unit.

The characteristic and limitation of data-forwarding supported by this hardware is listed.

1. GroupX (X is the name of execution unit) FF can only hold 1 clock cycle.
2. By 1, it cans only forwards data that is used only once later. Because it does not write to register
3. The maximal speedup is 2x in current model. Instruction with Nops is 3 cycles long, and without Nops is 1 cycle.

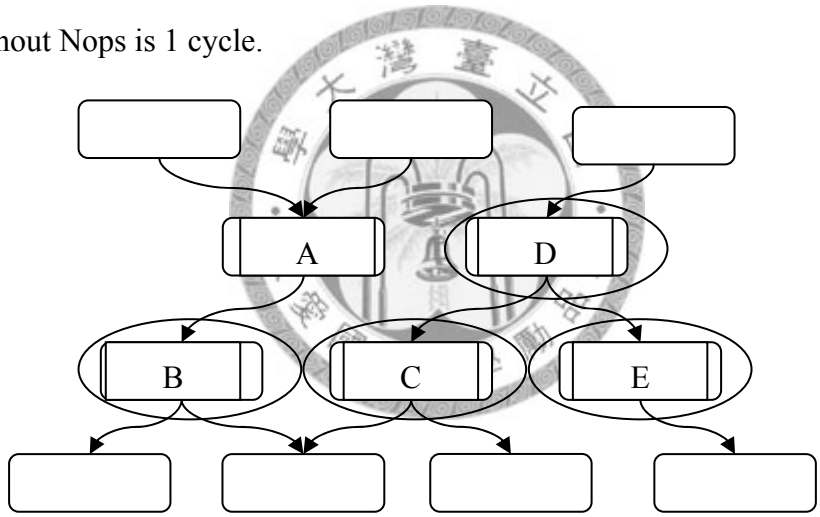


Figure 3-6 Dependence DAG

To do data forwarding detection, we can find the instruction to forward DAG used in list scheduling. The character of data forwarding node are

1. The successors are all scheduled. This can be solved by scheduling.
2. The last scheduled node has only one predecessor.

As the attributes are known by us, we can schedule for data forwarding. In the Figure 3-6, the nodes circled are the node possible to do data-forwarding.

There are two ways we can do this:

1. Passive forwarding: If the node is suitable to do data forwarding, just forward it.
2. Positive forwarding: Finding the node have data forwarding node as possible as we can

The pseudo codes are listed in following.

```
While ( candidate_is_still_available() ){

        //general scheduling code …..

        /*passive data forwarding*/
        If(      data_fordwarding_check
        (    last_schedule_instruction,current_schedule_instruction)    )
        {
              Modify_SrcReg_of_current_schedule_instruction();
              Modify_DstReg_of_ last _schedule_instruction();
        }
        Else{
        // general scheduling code …..
        }
        // general scheduling code …..
}
```

Figure 3-7 Passive Data Forwarding

This method forward data when the previous instruction can forward data to current instruction. But this is still not enough. Forwarding data is always good for performance in current. We therefore need to find all possibility of forwarding. Positive data-forwarding forwards data of all possibility by some extra overhead. The pseudo code is listed in following.

```
While ( candidate_is_still_available() ){
  /*positive data forwarding*/
if(#_of _RAW_successor_of_last_scheduled_instruction == 1 ){
        Can_data_forwarding = TRUE;
        Forward_node = the_only_RAW_succussor;


}


If(! can _data_forwarding){
        //general scheduling code …..
}{
          Modify_SrcReg_of_current_schedule_instruction();
        Modify_DstReg_of_ last _schedule_instruction();

}
```

Figure 3-8 Positive Data Forwarding

Figure 3-9 shows t flow chart of algorithm. Notice that we need only “Positive Data-forwarding Inst Selection”, “Passive data-forwarding relation Detection” is useless because “Positive Data-forwarding Inst Selection” is abetter choice. The “General instruction selection” is the function that selects an instruction to schedule. The policies we used are 1.instruction without independence first. 2. Destination register will be used first. 3. The head nodes in the candidate queue first.

3.3 Register Allocation

Register allocation is the process of multiplexing a large number of target program variables (or virtual registers) onto a small number of CPU registers. The goal is to keep as many operands as possible in registers to maximize the execution speed of programs.

Register allocation can happen over a basic block (local register allocation), over a whole function/procedure (global register allocation), or in-between functions as a calling convention (interprocedural register allocation).

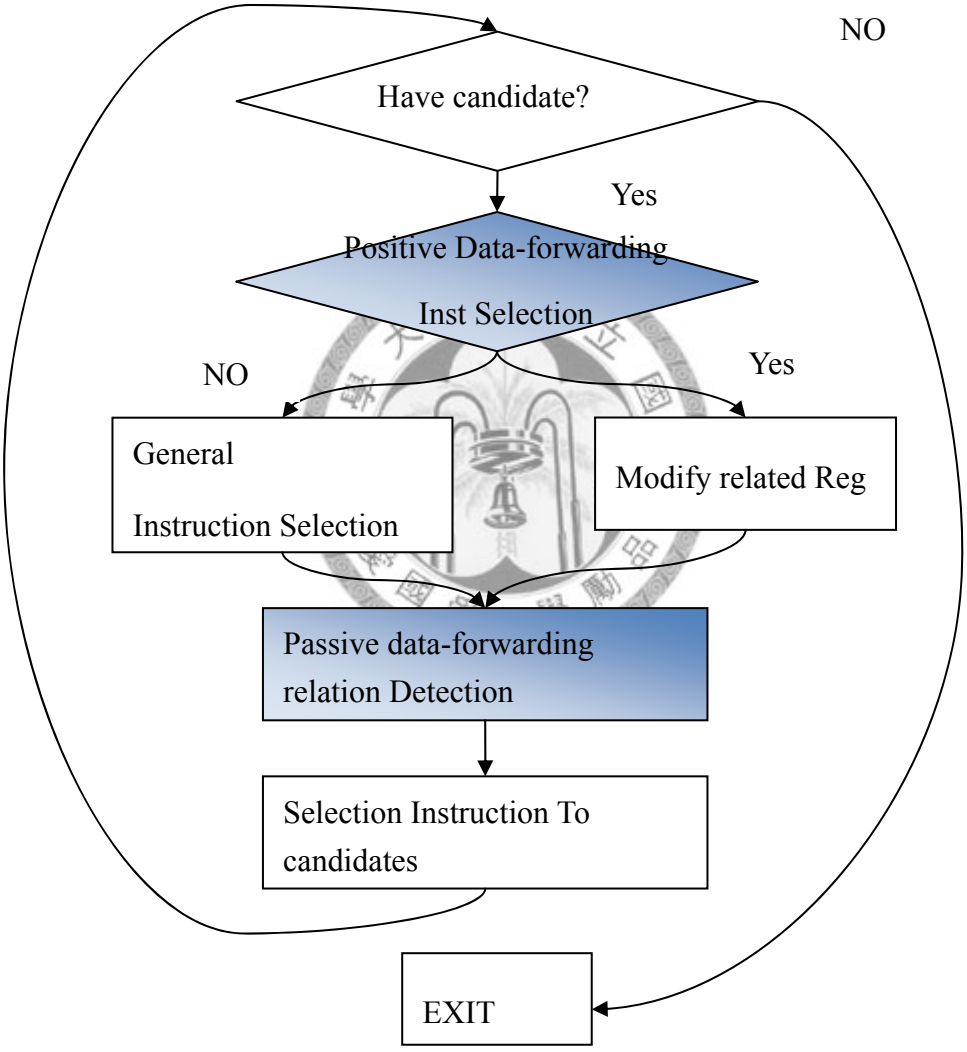


Figure 3-9 Process of instruction scheduling

When a compiler is generating machine code and there are more live variables than

the machine registers, it has to "spill" some variables from registers to memory. This incurs a certain cost, as access from memory is typically slower than access from a register.

3.3.1 Implementation

The algorithm we implemented is a modified linear scan allocation. We also call it as “Tetris Allocation”. The modification can pack variables without full width of vector type. Following content describes the algorithm and why we use this algorithm. Related materials are also described in following content.

3.3.2 Graph Coloring

Graph coloring is one of the most used register allocation methods because it generates the possible best register allocation. After IBM develops it, most compilers use this method for register allocation. Graph coloring maps register allocation problem to graph problem. It generates interference graph first, which graph have nodes represent virtual registers and edges represent that virtual registers live in some time interval. Then it colors on interference graph, which lists in figure 3-10. The key insight to graph coloring algorithm is called the degree < R rule. Given a graph G which contains a node N with

degree less than R, G is R-colorable iff the graph G', where G' is G with node N removed, is R-colorable.

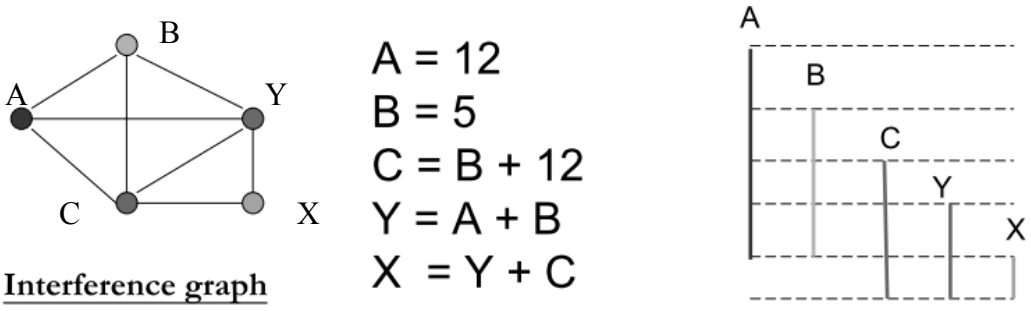


Figure 3-10 Interference graph and live interval

Graph Coloring

```
While G cannot be R-colored
   While graph G has a node N with degree less than R
      Remove N and its associated edges from G and push N on a stack S
   End While
   If the entire graph has been removed then the graph is R-colorable
      While stack S contains a node N
         Add N to graph G and assign it a color from the R colors
      End While
   Else graph G cannot be colored with R colors
      Simplify the graph G by choosing an object to spill and remove its node N
from G
      (spill nodes are chosen based on object's number of definitions and references)
End While
```

Figure 3-11Graph Coloring Algorithm

3.3.3 Linear Scan Allocation

Linear scan allocation [2] is another algorithm that has been recently developed. Comparing to graph coloring, it have some attributes.

The algorithm

```
LinearScanRegisterAllocation
active ← {}
foreach live interval i, in order of increasing start point
    ExpireOldIntervals(i)
    if length(active) = R then
        SpillAtInterval(i)
    else
        register[i] ← a register removed from pool of free registers
        add i to active, sorted by increasing end point

ExpireOldIntervals(i)
foreach interval j in active, in order of increasing end point
    if endpoint[j] ≥ startpoint[i] then
        return
    remove j from active
    add register[j] to pool of free registers

SpillAtInterval(i)
    spill ← last interval in active
    if endpoint[spill] > endpoint[i] then
        register[i] ← register[spill]
        location[spill] ← new stack location
        remove spill from active
        add i to active, sorted by increasing end point
    else
        location[i] ← new stack location
```

Figure 3-12 Linear Scan Register Allocation

1. Faster, simple: The complexity of graph coloring is O(N^2), but Linear scan is less than 0(N^2). This is because that it have only 1 major pass.
2. The allocation efficiency is not good as graph coloring.

The algorithm is shown in figure 3-9.

3.3.4 Packing

Shading language has vector type. There are vec4, vec3, vec2, int4, int3, int2, bool4, bool3, bool2 , the longer width of a variable is 4. When we consider the hardware issue, register width is depended on hardware implementation. In this work, the register width is 4 just as same as the longer width of shading language's variable. For the vector type, it needs to do packing: An example of packing is list in follow.

vec3 a3, b3,c3;
float a,b,c;
 	
c3= a3+b3;
c= a*c;
 	
It may costs 6 registers.
	After packing….
	
	C4.xyz = a4.xyz+b4.xyz;
	c4.w = a4.w*b4.w
	
	It may cost just 3 register after packing.

Table 3-2 Comparison of register usage with and without packing.

3.3.4.1 Packing efficiency factor:

Through the example shows the improvement of 2x. The packing in real case can get

that as much benefit as example do. Because following factors.

1. Number of vector type :

   If all registers are all vec4, there are not any improvements of register usage reduce.

   Consider all variable live during all program life, the efficiency can be mode as

   Probility(1)+ probility(2,2)/2 if probility of 3 > 1

   Probility(3)+ probility(2,2)/2 + (Probility(1)- Probility(3))/4

2. Live interval :

   Live interval is another effect of packing. If we don't consider the live interval, the register allocation algorithm will produce performance since it is live interval sensitive.

3. Implementation:

   You can define how to pack variables. 2 variables together, 3 variables togethers, even pack 4 variables together.

3.3.4.2 Bitwidth Aware

The first related work that has some differences is "Bitwidth Compilation [20]", it finds and analyses the possible shortest width of variables and use it to pack a word length long variables. The importance of the paper is the method to analysis. Another way is "Bitwidth Aware Global Register Allocation [20]". It analyzes and uses coalescing to pack. In our situation, we don't need such complex methods. Programs of shading language have

already point out “Bitwidth” attribute, what we have to do is packing the possible virtual vector register together. If we use related methods, it may not run enough fast because methods are complex and may not benefit much from this algorithm.

3.3.4.3 Discussion about Packing with Graph Coloring

Packing in graph coloring is not an easy work. Since it uses interference graph, relation of live intervals are transformed to interference relation. Packing may be opposite to interference relation. Packing relation in interference graphic is the circle in figure 3-13, but two circles in figure 3-13 have a same node. If we build packing relation first and later do graph coloring, it cannot work if there are nodes circle by more than 1 circle because interference graph has no time information and width information. Circles in graph coloring are coalescing, but circles with same node cannot view as coalescing. So packing with graph coloring is not easy. If you need use graph coloring with packing, redesign and modify algorithm is needed.

If we use pre-packing pass, the register allocation will produce poor performance since the live interval become longer which is shown in figure 3-14. If packing posts graph coloring, it needs to another allocation. And there are possible miss for fine grain packing. The shortages to support packing in graph color are shown in figure 3-14, figure 3-15.

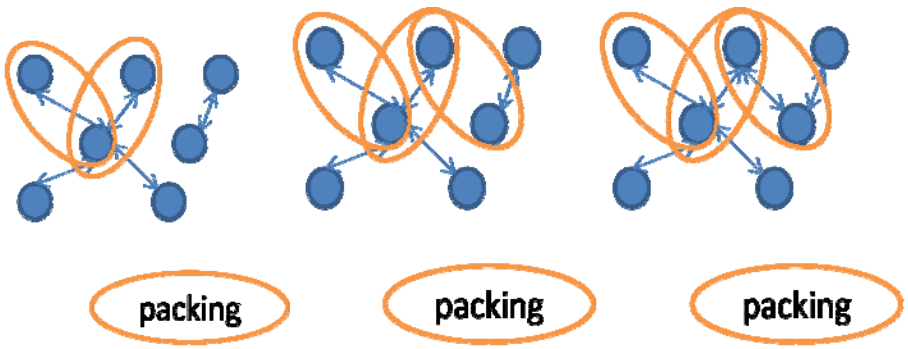


Figure 3-13 Packing in interference graph

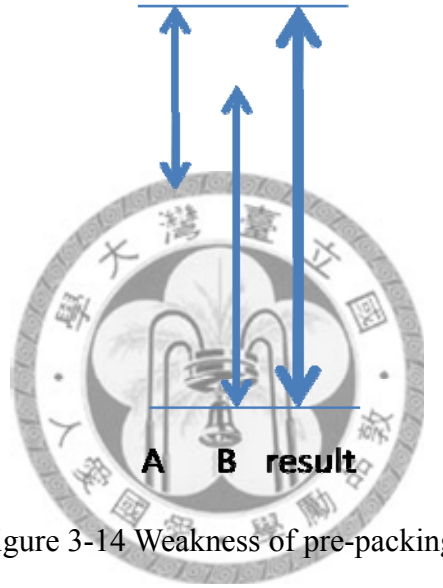


Figure 3-14 Weakness of pre-packing

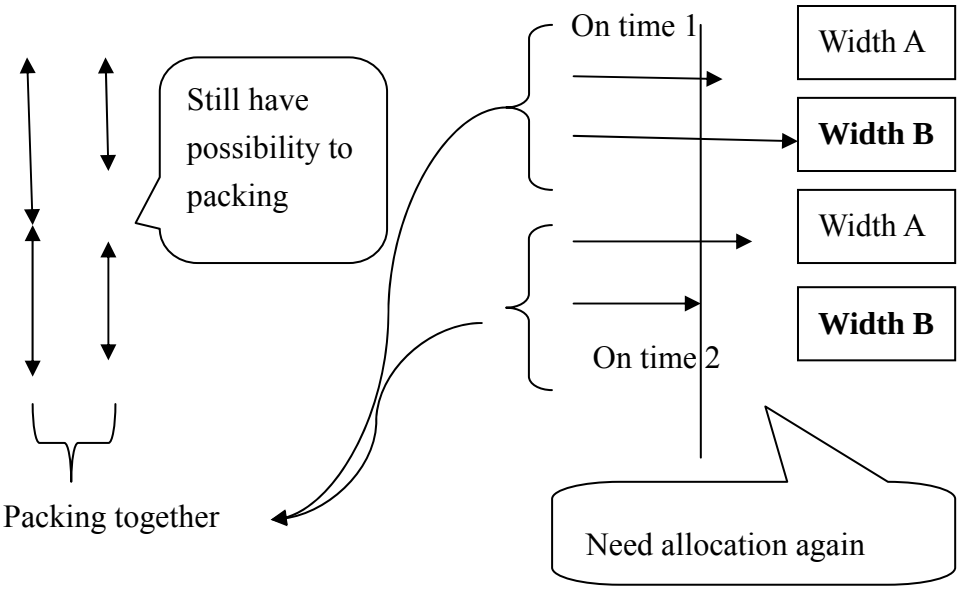


Figure 3-15 Weakness of post-packing

Support packing with graph coloring is not easy, it always need extra overhead. However packing is a NP-Complete program. A heuristic algorithm is enough to solve this problem.

3.3.4.4 Modified Linear Scan Allocation - Tetris Allocation

So we modify Linear Scan Allocation to support packing and called it as “Tetris Allocation”. It is easier and faster to support packing in graph coloring.

There are 4 kind of type in register pools after modification- Scalar_Pool, Width_2Pool, Width_3Pool, Width_4Pool. If the type of variables can’t mix with other type, Pools represent Float, Integer, Boolean are needed. The spirit of algorithm is that original linear scan allocation will allocate and free for proper width of register.

Major algorithm:

Like linear scan algorithm, but Allocation and “ExpireOldIntervals” are modified for packing.

```
TetrisAllocation
active ← {}
foreach live interval i, in order of increasing start point
    ExpireOldIntervals(i)
    if length(active) = R then
        SpillAtInterval(i)
    else
        Allocation(i)
        add i to active, sorted by increasing end point
```

Figure 3-16 Tetris Allocation

Allocation Process:

The function of allocation is changed as following. Besides allocation, adding the reminding partial to proper free pools is need in this modification. TypeWidth(i) returns type(float /integer…) and width(1~4) of variable that has liveness i. FulltypeWidth(i) returns full width..

```
If numofElementInFreePool( typeWidth(i) )!= 0
Register[i]< - removeOneFromFreePool( typeWidth(i) )
   ModifyRegisterChannel( i )
Else numElementInFreePool( FulltypeWidth(i) )
   Register[i]< - removeOneFromFreePool( FulltypeWidth(i)    )
   Insert the remind width of register to the proper free pool
```

Figure 3-17 Allocation of Tetris Allocation

ExpireOldIntervals Process:

We also need to modify the ExpireOldIntervals Process. Some to notice here, we need check if there are free pools have the partial width register. If there are, we need to collect the partial and add to proper free pool with the aggregated width.

```
foreach interval j in active, in order of increasing end point
      if endpoint[j] ≥ startpoint[i] then
            return
      remove j from active
      If there are partial width    of register[j] in free pool
         Aggregate the partial width of register [j]
    Add register[j] with proper width to proper memory pool.
```

Figure 3-18 ExpireOldIntervals of Tetris Allocation

We can see the method is very simple. Comparing with supporting packing in graph coloring, the shortages meet with doing packing in post-pass or pre-pass can be easily prevented. And yet it is efficient.

3.3.4.5 Packing Analysis

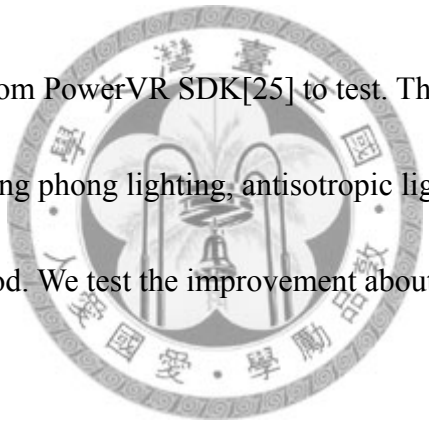
The algorithm can also analysis what variables to pack from, the information can be used by other allocation algorithm such as graph coloring and so on. The objective of this analysis is collecting what variables can pack together. The spirit of it is that variables in their own live interval can only pack with variables in active. The information needs more process to make it useful.

```
Analysis
packingRelation:Hash{a,Set}
active ← {}
foreach live interval i, in order of increasing start point
    ExpireOldIntervals(i)
    packingRelation(j)  = packingRelation(j)  ∪ active
add i to active, sorted by increasing end point

ExpireOldIntervals(i)
foreach interval j in active, in order of increasing end point
    if endpoint[j] ≥ startpoint[i] then
        return
    remove j from active
```

Figure 3-19 Packing Analysis

Chapter 4

Experiments



We use code samples from PowerVR SDK[25] to test. The codes contain small and fundamental shaders, including phong lighting, antisotropic lighting, envmap, fasttnl, , reflection, simple , toon, wood. We test the improvement about register allocation and code scheduling.

4.1.1 Efficiency of Data Forwarding by Compiler

In this experiment, we compare the instructions with and without data forwarding. The number in table is instruction count. Since current compiler takes care of nop generation and forwarding detection. The instruction number can show information about the relative performance, programs with less instruction is better for performance.

with/without	phong lighting	antisotropic lighting	envmap	fasttnl
vertex	97/112	66/71	79/112	40/50
fragment	2/2	5/10	18/26	8/20

with/without	reflection	simple	toon	wood
vertex	63/82	12/24	33/46	33/47
fragment	5/10	2/4	32/36	40/43

Table 4-1 Instruction number used with and without applying data forwarding

The result shows that vertex is much more complex while fragment is simpler. The vertex shader have much more ILP, therefore the gain is less the best condition. Fragment code is simple in common, thus it is closed to the best theoretical speedup.

4.1.2 Efficiency of Packing with Register Allocation

In this experiment, we compare the register usage with and without packing. The register usage is reported after compiling the shader code. And it is used to configure CMA. The table in following shows the tests that have improvements.

	phong lighting	envmap	reflection
packing	5	5	4
Without packing	7	6	5

Table 4-2 register usage with and without packing

We discover that the fragment shader code in SDK is so simple that it doesn't have any gain by packing. When we check the result of vertex, we found only 3 of them have improvement. There are phone lighting and so on. By checking the code, we found that variables have all program life are primarily sources for packing in SDK. To make a simple conclusion, the major performance gain by packing comes from the variables of all program life scope. If the programs are complex in variables width, it will gain much more. By the way, the forwarding technique hardware provided can also save 1~2 registers. Because it doesn't need to write to a general register but write to a pipeline register.

4.1.3 Discussion:

Here we discuss the relation of register allocation and code scheduling. In currently scheduling algorithm, it will found ILP as possible. But the effect brings the mass usage of register. An example is showed in table 4-3.

For register allocation, case 1 only uses 2 temporal registers to get the result. BUT case 2 uses 4. In this architecture, very limit registers are shared by all thread and there is no memory to spill. The result is that we need to take care of register usage sometimes to prevent the shortage of threads.

Case 1:Genernel Scheduling	Case 2:Schedule for ILP
//mat3* vec3	//mat3* vec3
Mul t0,c0,v0	Mul t0,c0,v0
Mul t1,c1,v0	Mul t1,c1,v0
NOP	Mul t2,c3,v1
NOP	Mul t3,c4,v1
Add t1,t1,t0	Add t1,t1,t0
Mul t0,c2,v0	Mul t0,c2,v0
NOP	Add t3,t2,t3
NOP	Mul t2,c5,v1
Add t1,t0	Add t1,t0
.....	NOP
//another mat3 * vec3	Add t3,t2,t3
Mul t2,c3,v1	
Mul t3,c4,v1	
NOP	
NOP	
Add t3,t2,t3	
Mul t2,c5,v1	
NOP	
NOP	
Add t3,t2,t3	

Table 4-3 RA vs CS

Chapter 5 Conclusion

In our work, a shading language compile for a mobile GPU with self-defined ISA has been developed. Considering register allocation, packing is a solution to reduce usage of registers and increasing performance. Due to some shortages to do allocation under graph coloring allocation, we propose a simple and efficient method to support packing under linear scan allocation. Experiments show that some vertex programs can therefore reduce the register usage. When it comes to discuss about code scheduling, we detect the data forwarding in compiler and generate code that hardware can directly forward data. It is necessary for the current hardware to improve performance. These optimizations are applied without performance regress

Chapter 6 Future Work

There are still some works that don't finish yet. Numbers of optimizations like memory optimization, loop optimization haven't done in current implementation. Considering only register allocation and code scheduling, there are several possible ways for enhancement.

1. Register control by compiler

   Because the usage of register is known by compiler, there are chances to adjust CMA register usage dynamically for better performance.

   a. Compiler can suppress ILP to get register usage for the near best register settings.

   b. If there are instructions to dynamic adjust register setting during runtime, it can get more threads when register usage is low on some program phase.

2. Software forwarding and hardware data forwarding.

   Current hardware needs compiler to do data forwarding detection. We can also combine with hardware data forwarding detection to take both of their goodness afterward.

Bibliography

[1] Eric Chan, Ren Ng, Pradeep Sen, Kekoa Proudfoot, and Pat Hanrahan ,“Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware,” in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*,pp. 69 – 78,2002.

[2] Samuel Larsen, Saman Amarasinghe ,“Exploiting superword level parallelism with multimedia instruction sets,” *ACM SIGPLAN Notices*, Volume 35 , Issue 5, pp. 145-156 ,2000.

[3] William R. Mark, Kekoa Proudfoot ,“Compiling to a VLIW fragment pipeline,” in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* ,pp. 47 - 56, 2001.

[4]William R. Mark, R. Steven Glanville, Kurt Akeley, Mark J. Kilgard ,”Cg: a system for programming graphics hardware in a C-like language,” *ACM Transactions on Graphics (TOG),* Volume 22 , Issue 3, July 2003.

[5]Marc Olano, Anselmo Lastra ,“A programmable pipeline for graphics hardware,” UMI

Order Number: AAI9840971, The University of North Carolina at Chapel Hill ,1998

[6]Marc Olano, Anselmo Lastra “A shading language on graphics hardware: the pixelflow shading system,” in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques,* pp.159-168, July 1998

[7] Brian Guenter, Todd B. Knoblock, Erik Ruf, “Specializing shaders,” in *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pp. 343-350, 1995.

[8]Michael D. McCool ,Zheng Qin Tiberiu S. Popa ,”Shader metaprogramming,” in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware,* pp. 57-68, 2002.

[9]http://msdn.microsoft.com/directx.

[10]http://www.opengl.org/

[11] https://renderman.pixar.com/products/rispec/

[12]http://www.opengl.org/documentation/glsl/

[13] http://www.khronos.org/opengles/

[14]http://www.vincent3d.com/Vincent3D/software/ogles2/ogles2.html

[15]Steven S. Muchnick, “Advanced compiler design and implementation,” Morgan Kaufmann Publishers Inc., San Francisco, CA, 1998.

[16]http://www.gpgpu.org

[17] R. L. Cook,“Shade Trees.” *ACM SIGGRAPH Computer Graphics*, Volume 18, Issue 3, pp. 223-231, 1984.

[18] Gregory J. Chaitin, Mark A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein, "Register allocation via coloring," C*omputer Languages 6*, pp. 47-57, 1981.

[19] Massimiliano Poletto, Vivek Sarkar. “Linear scan register allocation,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Volume 21, Issue 5, pp. 895-913, 1999.

[20] M. Stephenson, J. Babb, and S. Amarasinghe, “Bitwidth Analysis with Application to Silicon Compilation,” *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*

[21] You-Ming Tsao, “Scalable and Reconfigurable Stream Processor for Mobile Multimedia System,” PH.D thesis, 2008.

[22] Yu-Cheng Lin, “Hardware Architecture Design and Implementation of Universal Vertex/Pixel Shader for 3D Graphics System,” Master Thesis ,2007.

[23] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, Greenberg and Bennett Battaile, “Modeling the interaction of light between diffuse surfaces," *ACM SIGGRAPH*

Computer Graphics ,Volume 18, Issue 3, pp. 213-222, July 1984.

[24] http://en.wikipedia.org/wiki/ARB_(GPU_assembly_language)

[25] http://www.imgtec.com/powervr/insider/sdk/KhronosOpenGLES2xSGX.asp

[26] Turner Whitted, "An improved illumination model for shaded display,"

Communications of the ACM, Volume 23, Issue 6, June1980.

[27] Norm Rubin, "Issues and challenges in compiling for graphics processors, "*in*

Proceedings of the sixth annual IEEE/ACM international symposium on Code generation

and optimization, pp 2-2, 2008.

[28] https://renderman.pixar.com/

[29] http://www.nvidia.com/object/cuda_home.html

[30] Sriraman Tallam, Rajiv Gupta "Bitwidth aware global register allocation" in

Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of

*programming language*s, pp. 85 – 96, 2003.