國立交通大學

資訊科學與工程所

碩 士 論 文

可動態重組之 shader unit 於頂點與像素處理

A dynamically reconfigurable shader unit for vertex and pixel processing

研 究 生：陳 逸 麒

指導教授：鍾 崇 斌 博士

中 華 民 國 九 十 五 年 九 月

可動態重組之處理單元於頂點與像素處理

A dynamically reconfigurable shader unit for vertex and pixel processing

研 究 生：陳 逸 麒　　Student：Yi-Chi Chen

指導教授：鍾 崇 斌　教授　　Advisor：Dr. Chung-Ping Chung

國 立 交 通 大 學

資 訊 工 程 學 系

碩 士 論 文

A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master
In

Computer Science and Information Engineering

September 2006

Hsinchu, Taiwan, Republic of China

中華民國 九十五 年 九 月

國立交通大學

研究所碩士班

論文口試委員會審定書

本校 資訊科學與工程 研究所 陳逸麒 君

所提論文:

可動態重組織之 shader unit 於頂點與像素處理

A dynamically reconfigurable shader unit for vertex and pixel processing

合於碩士資格水準、業經本委員會評審認可。

口試委員：

指導教授：

所　　長：

中 華 民 國 九十五 年 9 月 6 日

國 立 交 通 大 學

博碩士論文全文電子檔著作權授權書

（提供授權人裝訂於紙本論文書名頁之次頁用）

本授權書所授權之學位論文，為本人於國立交通大學 資訊科學與工程 系所 系統設計組，九十四 學年度第 二 學期取得碩士學位之論文。

論文題目：可動態重組之處理單元於頂點與像素處理

指導教授：鍾崇斌

■ 同意

本人茲將本著作，以非專屬、無償授權國立交通大學與台灣聯合大學系統圖書館：基於推動讀者間「資源共享、互惠合作」之理念，與回饋社會與學術研究之目的，國立交通大學及台灣聯合大學系統圖書館得不限地域、時間與次數，以紙本、光碟或數位化等各種方法收錄、重製與利用；於著作權法合理使用範圍內，讀者得進行線上檢索、閱覽、下載或列印。

論文全文上載網路公開之範圍及時間

本校及台灣聯合大學系統區域網路	■ 立即公開
校外網際網路	■ 立即公開

■ 全文電子檔送交國家圖書館

授 權 人：陳逸麒

親筆簽名：陳逸麒

中華民國 95 年 9 月 7 日

國 立 交 通 大 學

博碩士論文紙本著作權授權書

（提供授權人裝訂於全文電子檔授權書之次頁用）

本授權書所授權之學位論文，為本人於國立交通大學 資訊科學與工程 系所 系統設計組，九十四 學年度第 二 學期取得碩士學位之論文。

論文題目：可動態重組之處理單元於頂點與像素處理

指導教授：鍾崇斌

■ 同意

本人茲將本著作，以非專屬、無償授權國立交通大學，基於推動讀者間「資源共享、互惠合作」之理念，與回饋社會與學術研究之目的，國立交通大學圖書館得以紙本收錄、重製與利用；於著作權法合理使用範圍內，讀者得進行閱覽或列印。

本論文為本人向經濟部智慧局申請專利(未申請者本條款請不予理會)的附件之一，申請文號為：_____，請將論文延至____年____月____日再公開。

授 權 人：陳逸麒

親筆簽名：陳逸麒

中華民國 95 年 9 月 7 日

國家圖書館
博碩士論文電子檔案上網授權書

ID:GT009323622

本授權書所授權之學位論文，為授權人於國立交通大學 資訊科學與工程 系所 系統設計組，九十四 學年度第 二 學期取得碩士學位之論文。

論文題目：可動態重組之處理單元於頂點與像素處理

指導教授：鍾崇斌

茲同意將授權人擁有著作權之上列論文全文（含摘要），非專屬、無償授權國家圖書館，不限地域、時間與次數，以微縮、光碟或其他各種數位化方式將上列論文重製，並得將數位化之上列論文及論文電子檔以上載網路方式，提供讀者基於個人非營利性質之線上檢索、閱覽、下載或列印。

※ 讀者基於非營利性質之線上檢索、閱覽、下載或列印上列論文，應依著作權法相關規定辦理。

授 權 人：陳逸麒

親筆簽名：陳逸麒

中華民國95年 9月 7日

可動態重組之處理單元於頂點與像素處理

學生：陳逸麒　　指導教授：鍾崇斌 博士

國立交通大學資訊科學與工程研究所碩士班

摘　要

在頂點與像素的處理中，頂點與像素的工作量，在執行過程中有大量的變化。然而在固定的硬體資源分配下，頂點處理器以及像數處理器經常有一方閒置，而另一方則發生資源不足的情況。為此，我們提出了一個新的 shader unit: DR-shader unit，可針對工作量的變化，動態分配處理器於頂點或像素處理之數量，以提升硬體資源之使用率，並縮短執行時間。

在本論文中，首先分析處理器的架構，決定可動態重組處理器中，各元件是否能讓兩種組態所共用。其中我們利用**最小繞線代價**、**最多共用邏輯**以及**最佳面積與時間**三種演算法，幫助我們決定運算邏輯是否應作共用設計，以組合成運算單元。並且設計工作量監測邏輯，根據工作量的變化控制各可動態重組處理器之組態。最後得到於速度上有 60%之提昇，以及 30%使用率提昇。

A dynamically reconfigurable shader unit for vertex and pixel processing

Student：Yi-Chi Chen Advisor：Dr. Chung-Ping Chung

Institute of Computer Science and Engineering
College of Computer Science

Abstract

In vertex and pixel processing, the workloads of vertices and pixels vary greatly during run time. However, in fixed resource allocation between vertex shaders and pixel shaders, many vertex or pixel shaders may be idle while the other type of shaders are insufficient. Therefore, we propose a dynamically reconfigurable shader unit (DR-shader unit) which can distribute shaders for vertex and pixel processing according various workloads during run time. By the way, shader utilization can be upgraded, shortening execution time

In this thesis, we firstly analyze the architecture of shaders and determine shared units between vertex and pixel shader type in DR-shader. We use three algorithms: **minimum routing overhead**, **maximum sharing logic**, **and optimal area-time** to determine how logics be shared and complete sharable computation unit. Besides, we design workload monitor logic to control the configuration of each DR-shader by workloads. Finally we gain 60% upgrade in speed and 30% upgrade in utilization

誌　謝

首先感謝我的指導老師 鍾崇斌教授，在老師的諄諄教誨、辛勤指導與勉勵下，我得以順利完成此論文，並且順利通過畢業口試。同時感謝我的口試委員 單智君教授、 蕭勝夫教授以及 周賜福教授，由於他們的指導與建議，讓這篇論文更加完整和確實。

此外，感謝實驗室的學長姊、以及學弟妹們，經常在各種問題上給予我不同的意見，並且不斷的鼓勵我，得以持續地努力下去，也給予我心情上的抒發。特別感謝的是我唯一的同學汪威定，從大二起就不斷的支持我，並在各種時候給予我任何幫助，同時也鼓勵我提早入學，最後也陪我一起提早畢業。

在此，我必須向我的女朋友說聲道歉。在忙碌的時候，我並沒有很多時間陪她，在她生日的這天，我必須先跟她說聲生日快樂，同時也說聲抱歉，因為我有太久沒有好好陪她了。

最後感謝我的家人，謝謝你們在背後全心全意地支持我，讓我在這研究的路上走得更順利，進而能無後顧之憂的學習，讓我追求自己的理想。

謹向所有支持我、勉勵我的師長與親友，奉上最誠摯的祝福，謝謝你們。

陳逸麒

2006. 9. 7

Table of contents

List of Figures

List of Tables

Chapter 1 Introduction

Programmable graphics pipeline is the most popular type of graphics hardware nowadays. The program lengths and execution time of vertex and pixel processing may vary from scene to scene. However, this kind of variation in the execution time will lower the utilization of graphics hardware. In this thesis, we propose a dynamically reconfigurable shader unit (DR-shader unit) for vertex and pixel processing. DR-shader unit can dynamically allocate its hardware resources to harmony with the computation requirements of vertices and pixels at runtime. By this kind of flexibility, we can increase the utilization of graphics hardware and shorten the execution time of scenes.

1.1 Vertex and pixel shaders

In vertex and pixel processing, there are number of vertex and pixel shaders, which are in the form of programmable processors. The function of the two shaders is to execute the entire vertex or pixel shader codes respectively on each individual vertex or pixel and shader codes vary from pass to pass. However, the workloads of vertices and pixels for the two shaders may be very various during run-time. Number of pixels will be produce by each primitive (composed of three vertices) may have a range from zero to whole pixels in a scene, according to its position. In different situations the execution time of each vertex and pixel may be very diverse

The workloads of vertices and pixels for the vertex and pixel shaders may be very various during run time. Traditionally, number of vertex and pixel shaders are fixed and the various workloads are partially be adapted by pixel queue, which is a buffer in front of pixel processing and stores pixels for the inputs of pixel shaders. However, the problem is that the

degrees of variation during run time often exceed the adaptability of pixel queue. When pixel queue is full, there is no space to store the result of vertex processing and all stages in vertex processing will be idle, including vertex shaders. When pixel queue is empty, there is no input for pixel processing and all stages in pixel processing will be idle, including pixel shaders. When both of these two situations happen too frequently, the utilization loss of graphics processing unit will be very low.

1.2 Dynamically reconfigurable system

We can basically classify reconfigurable systems into two different categories: dynamically reconfigurable system and static reconfigurable system. The most important difference between the two systems is that dynamically reconfigurable system can change its configuration during runtime. Dynamically reconfigurable system not only can be used for reducing the requirement of hardware in a design, but also can be used for circuit specialization based on the information known only during runtime. This feature does not exist in both static reconfigurable system and ASIC design. Moreover, by means of dynamically reconfiguration, we can optimize the resource allocations in hardware to meet the computation requirements at runtime.

1.3 Motivation

The workloads of vertices and pixels for vertex and pixel shaders may be very various during run time. It is difficult for any architecture with fixed resource allocation between vertex shaders and pixel shaders to deal with such a big variation. If there are some multi-function shaders which can change their functions between vertex shader and pixel shader, we can easily distribute hardware resource according to the workloads of vertices and pixels. Besides, the architectures of vertex shader and pixel shader are very the same and lots

of the hardware resources can be shared to each other. It gives us a very good chance to use reconfigurable architecture to design them

1.4 Objective

Design a dynamically reconfigurable shader unit (DR-shader unit) to adapt various workloads between vertices and pixels

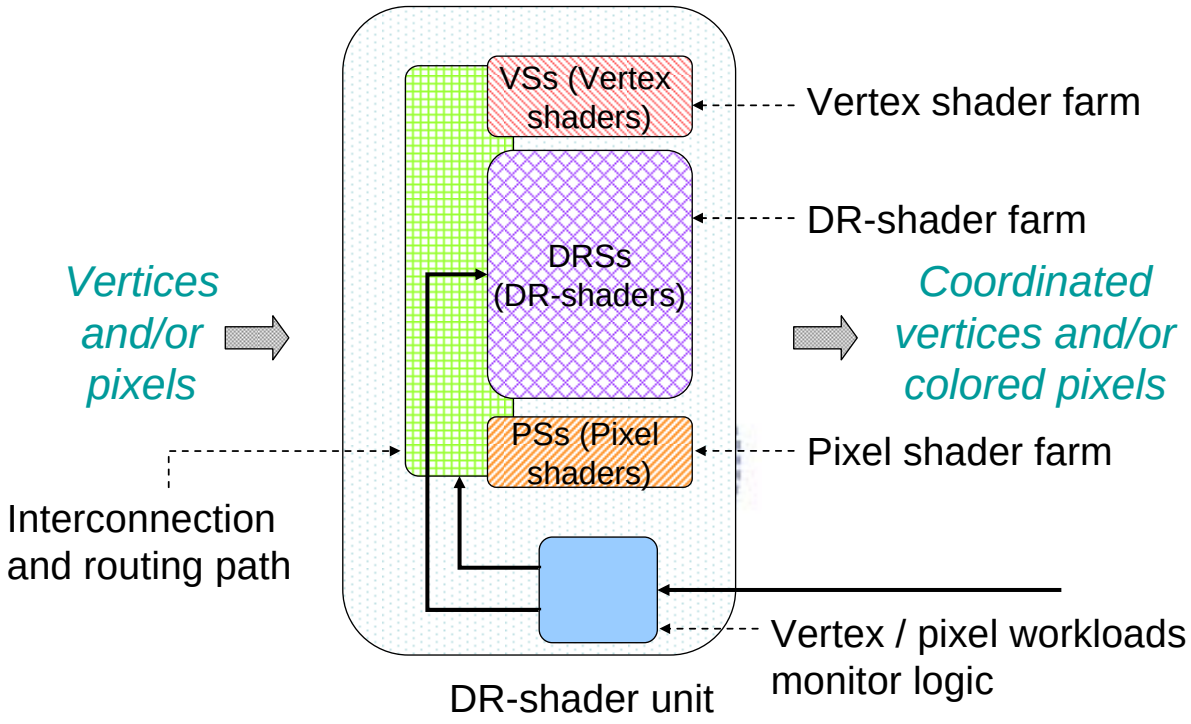


Fig.1-1 The architecture of DR-shader unit

1.5 Organization of this thesis

The organization of this thesis is as follow:

In Chapter 2, the background about graphics pipeline is presented.

In Chapter 3, we analyze the architecture of vertex and pixel shaders with their computation requirement and design DR-shader with workloads monitor logic

In Chapter 4, we show our simulation result with environment and decide a proper

proportion within vertex, pixel and dynamically reconfigurable shaders.

In Chapter 5, there are discussion, future work and conclusion.

Chapter 2 Background

2.1 Graphics pipeline

We can simply see graphics pipeline as separable into four distinct and sequential steps: vertex processing, rasterization, pixel processing, and writeback. In below, we will use a table to show inputs, output and explain their operations of these four steps and to give a mainly explanation.

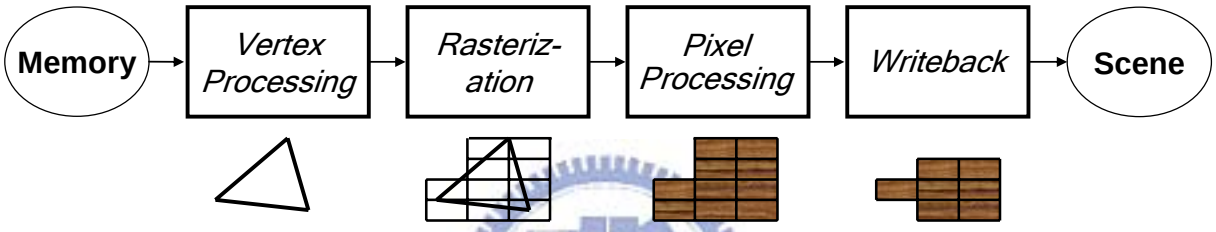


Fig.2-1 Four steps of graphics pipeline

	Input	Output	Operation
Vertex processing	Vertices with 3D coordinates	Vertices positioned in the 2D scene	Transforms each 3D vertex in world space to 2D vertex on scene
Rasterization	Primitives (triangles) assembled by vertices	Fragments	Interpolations each primitives into numbers of fragments
Pixel processing	Fragments	'Finalized' pixel with final color value	Colors each fragment according to its information
Writeback	'Finalized' pixel with final color value	Image composed of 'finalized' pixel	Uses frame buffer storing pixels to assemble a frame

Table.2-1 Input, output and operation in each step of graphics pipeline

In the following sections, we will completely descript the details in vertex and pixel

processing.

2.2 Vertex processing

At the input of vertex processing step, each primitives consists of three vertex coordinates, vertex normal values and other information, such as lighting and texture coordinates. At the beginning, all vertices are represented in the 3D coordinates with three dimension values {x, y, z}. In order to using a uniform matrix representation to represent affine transformation, we convert the Cartesian coordinates (3D coordinates) to the homogeneous coordinates, which are quadruples of the form {X, Y, Z, W}, where {X, Y, Z, W} = {xW, yW, zW, W} and in most case W is 1. After the conversion, we can use a sequence of matrix operations easily to transform the coordinates of vertices. Figure3 shows the steps of vertex processing in a typical graphics pipeline which consists of the following stages:

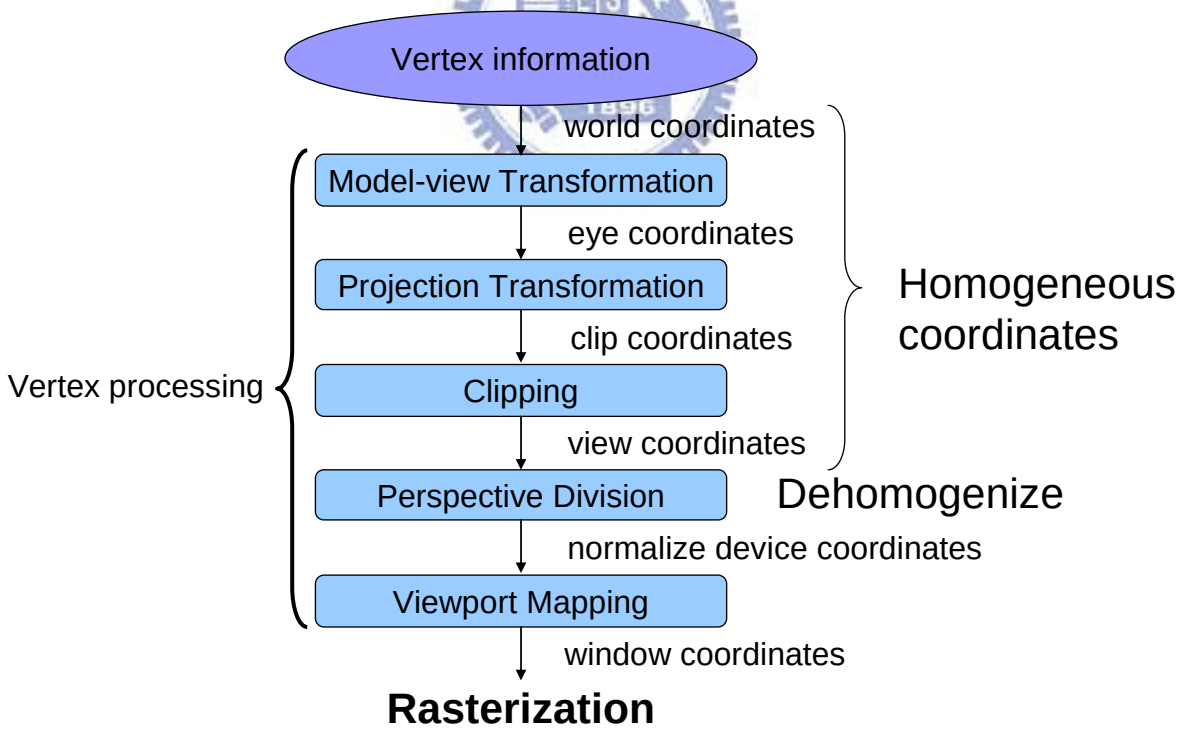


Fig.2-2 The steps of vertex processing

2.2.1 Model-view transformation

Modeling transformation may reshape and move primitives with respect to the position of viewer (eye position: {eye_x, eye_y, eye_z}) because the position of the viewer often does not locate at the origin of the world coordinates. Therefore, we must use move the position of the viewer to the origin and also move all the vertices with the movement of the origin. Formula 1 shows the matrix that we use to transform the position of viewer to the origin.

$$T\begin{bmatrix} eye_x \\ eye_y \\ eye_z \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \Rightarrow T = \begin{bmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix} ----------(1)$$

Fig.2-3 Formula1

Besides the movements of the position, we must change the directions of x-axis y-axis and z-axis with respect to the orthogonal direction (u), the up-direction vector (v), and the viewing direction (n) of the viewer. Fig4 shows the relations of u, v and n. Formula2 shows the matrix that we use to do the transformation.

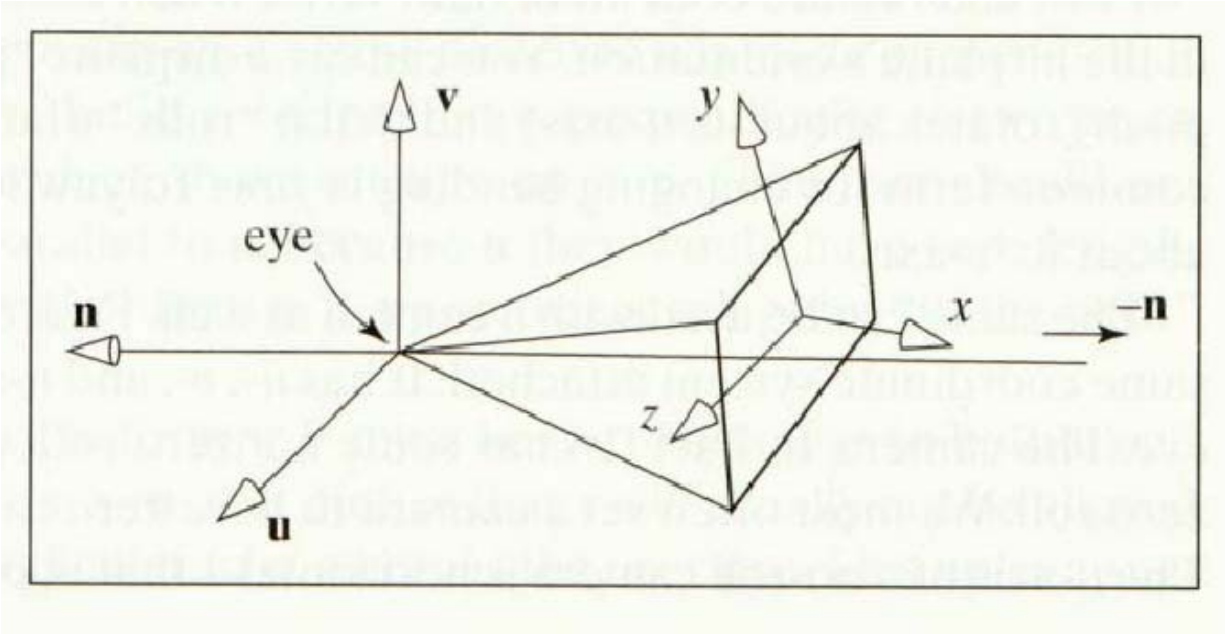


Fig.2-4 The relations between (u, v, n)

$$B\begin{bmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \Rightarrow B = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} --(2)$$

Fig.2-5 Formula2

This new orthogonal coordinate system usually is called as the viewing-coordinate system or the u-v-n system. Because these two transformations are both multiplications with a 4×4 matrix in the homogenous coordinates, they can be combined into a single multiplication (Formula 3), which is implemented by 16 floating point multiplications and 12 floating point additions. As the result, the model-view transformation carries us to eye coordinates, where the viewer is at the origin and the directions of the x-axis, y-axis, and z-axis have changed. In the model-view transformation, we translate all vertices from the world coordinates to the eye coordinates. Then, we need to project all vertices on the view plan.

$$BT = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix} ----------(3)$$

Fig.2-6 Formula3

2.2.2 Projection transformation

Like a real camera, once we decide the position and the directions of the viewer, all the objects (consist of vertices) will be projected on a plane (view plane, which is defined by $\{x_{max}, x_{min}, y_{max}, y_{min}, z_{max}, z_{min}\}$ six numbers) to show what we see. There are also near plane and far near to limit the space we can see and we usually call the limited space as view volume. Here we have two types of projections: orthogonal (orthographic) projection and perspective projection.

The orthogonal projection is a simple projection, in which the projector is perpendicular to the view plane. In this projection, the z values of objects just define the depth of objects. The only thing we must do is to normalize the view volume and let the view volume to be a cube with ranges from -1 to 1 (canonical view volume). The projection transformation will be like Formula 4.

$$P = \begin{bmatrix} \frac{2}{x_{\max} - x_{\min}} & 0 & 0 & -\frac{x_{\max} + x_{mix}}{x_{\max} - x_{\min}} \\ 0 & \frac{2}{y_{\max} - y_{\min}} & 0 & -\frac{y_{\max} + y_{mix}}{y_{\max} - y_{\min}} \\ 0 & 0 & \frac{2}{z_{\max} - z_{\min}} & -\frac{z_{\max} + z_{mix}}{z_{\max} - z_{\min}} \\ 0 & 0 & 0 & 1 \end{bmatrix} --------(4)$$

Fig.2-7 Formula4

The perspective projection is a more complicated transformation than the orthogonal projection but it can produce more realistic images by changing the sizes of objects according to their distances. Therefore, an object far away will be smaller than in the near. In this

projection, the x value and y value of an object may be divided by its z value. In the homogeneous coordinates, this kind of divisions can be implemented by just change the w value. Formula 5 shows the perspective projection matrix. This matrix also can translate the view volume to canonical view volume.

$$P=\begin{bmatrix} \frac{2z_{\min}}{x_{\max}-x_{\min}} & 0 & \frac{x_{\max}+x_{mix}}{x_{\max}-x_{\min}} & 0 \\ 0 & \frac{2z_{\min}}{y_{\max}-y_{\min}} & \frac{y_{\max}+y_{mix}}{y_{\max}-y_{\min}} & 0 \\ 0 & 0 & -\frac{z_{\max}+z_{\min}}{z_{\max}-z_{\min}} & -\frac{2z_{\max}z_{\max}}{z_{\max}-z_{\min}} \\ 0 & 0 & -1 & 0 \end{bmatrix} --------(5)$$

Fig.2-8 Formula5

Each of these projection transformations are both consist of a 4×4 matrix multiplication. Therefore, we also can combine the projection transformation with the model-view transformation. At this time, we have a canonical view volume (clip coordinates), and then we can easily to check whether objects are in the eyesight of the viewer.

2.2.3 Clipping

Although we transform all objects from world coordinates to the clip coordinates, there are many objects which are outside of the canonical view volume and won't be showed on the scene. Therefore, we must clip those objects to reduce the workloads of behind stages. Clipping in the homogenous coordinates isn't completely necessary, but it makes the clipping clean, fast, and simple. Besides, after dehomogenizing, the signs of the x value, y value, z value and w value will be lost $\{(x,\ y,\ z,)=\left(\frac{X}{W},\ \frac{Y}{W},\ \frac{Z}{W}\right)\}$. Therefore, we can't know whether objects are in front of or behind the viewer.

We first ignore the objects with w values smaller than zero because they are behind the viewer. Then, we can apply Cyrus-Beck clipping to test if a vertex V in the canonical view volume. Formula 6 shows the testing. By this testing, we clean some vertices out of the sight and others will continue into next steps.

$$\frac{a_i}{a_w}>-1\Rightarrow(a_w+a_i)>0,\frac{a_i}{a_w}<1\Rightarrow(a_w-a_x)>0\quad i\in\{x,y,z\}--(6)$$

Fig.2-9 Formula6

2.2.4 Perspective division

Finally, all vertices have been transform from world coordinates to eye coordinates, and some vertices out of the sight have also been clean. At this step, we try to transform objects from 3D- coordinates to 2D-coordinates and decide the position of each vertex on scene. In projection transformation, we have defined how vertices be projected on 2D coordinates and the information has been store in *w* value. Therefore, the function of perspective division is just to divide (*x, y, z*) by *w* value and discard *w* value. So, we dehomogenize each vertex using the Formula7.

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} \Rightarrow \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \frac{X}{W} \\ \frac{Y}{W} \\ \frac{Z}{W} \end{bmatrix} \qquad W \neq 0$$

Fig.2-10 Formula7

2.2.5 Viewport matrix

Finally, we decide the positions of each vertex on scene and the position of each vertex will be scaled by resolution of scene. Therefore, we transform the normalized (*x, y*) position of each vertex to scene position. Assume that the resolution of scene is *w* x *h*, then we will transform (*x, y*) from (*-1, -1*) to (*0, 0*) and from (*1, 1*) to (*w, h*). We use Formula8 to do this transformation.

$$\begin{bmatrix} w_x \\ w_y \end{bmatrix} = \begin{bmatrix} \frac{w}{2}(x+1) \\ \frac{h}{2}(y+1) \end{bmatrix}$$

Fig11. Formula8.

2.3 Programmable graphics pipeline

Programmable graphics pipeline is the most popular solution for the requirements of both performance and flexibility in computer graphics nowadays. With the rapidly development of computer graphics, such as 3D games, virtual realities and digital lives, the requirements of computer graphics in effects and performance become higher. To meet all kinds of users' requirements, programmable graphics pipeline have been introduced into graphics hardware and many complicated function units have been put in. Different from fixed-functionality (non-programmable) graphics pipeline, programmable graphics pipeline has new graphics processing units: vertex shader unit and pixel shader unit. These two new processing units give graphics pipeline the flexibility to deal with all kinds of computation requirements while retaining the capability of complicated computation.

Chapter 3 Design

3.1 Analysis of shaders

The architecture of vertex/pixel shader in DirectX(spec. of GPU) is below:

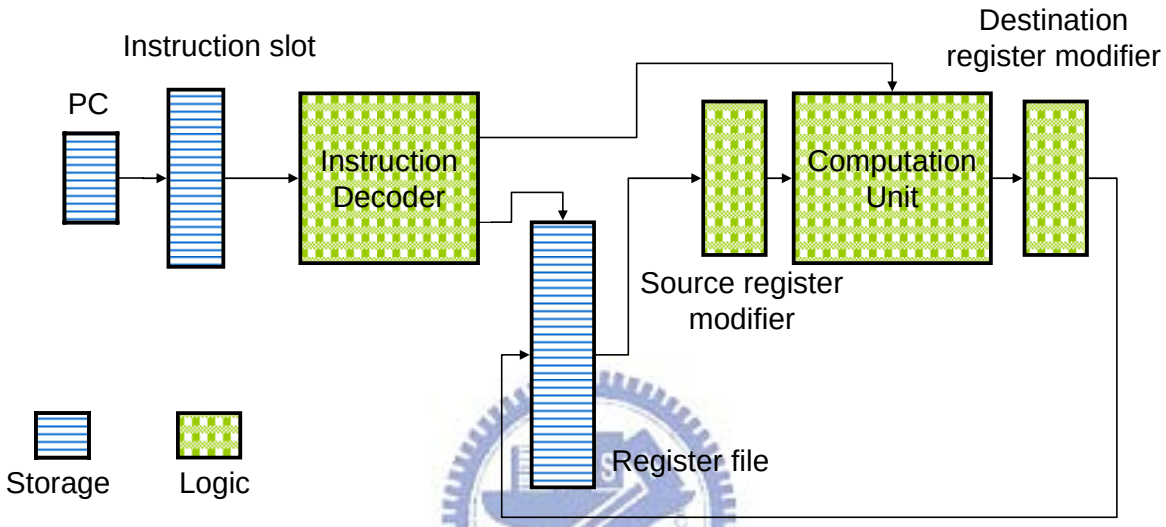


Fig.3-1 The architecture of vertex/pixel shader

There are several units in both vertex shader and the pixel shader, which are:

1. Program counter: a register which stores the address of the instruction being executed.

2. Instruction slot: a storage unit which stores all shader codes for vertex/pixel shader(s).

3. Instruction decoder: a combinational circuit to translate an instruction into the control signals of the data path.

4. Register file: a storage unit which contains all inputs and outputs of any computations for each vertex or pixel.

5. Source register modifier: a simple computation unit which can swizzle or negate source data.

6. Computation unit: the main computation unit which process complex operation (ex. add, mul, mad …).

7. Destination register modifier: a simple computation unit which is similar to source register modifier, but its target is destination register.

DR-shader must support all functions in both two shader types to be a multi-function shader. Therefore, it also contains those units and it must have to double the units which can't be shared between vertex shader type and pixel shader type.

Firstly, we consider which units in DR-shader can be shared between vertex shader type and pixel shader type to reduce the hardware overhead of DR-shader. The sharing policies are:

1. If and only if a storage unit must store data, which may be states, instructions or temporary results, for vertex shader and pixel shader simultaneously, it can't be shared.
2. All logic units are sharable.

Under these policies, we decide source register modifier, computation unit, and destination register modifier are sharable units because all of them are logic units. Instruction slot is non-sharable unit, for it must store vertex shader codes and pixel shader codes in the same time. Besides, we can't decide whether program counter and register file can be shared. We will make the decision for them when we discuss the architecture and flexibility of DR-shader.

Secondly, we deliberate upon how to design those sharable units for both two shader types. In those sharable units, source modifier and destination modifier are the same in vertex shader type and pixe shader type. Therefore, we will focus on how to design a sharable computation unit in the following sections. There are some assumptions of vertex and pixel shaders' architecture for us to design a sharable computation unit, listed below:

- Single issue and single execution: because shaders expose the parallelism of data better than the parallelism of instructions for single issue and multi-shaders respectively execute instead of multi-execution
- The widths of all operations in the computation unit are $i*v$ bits in vector form, where i is currently 32 (most probable), and v may be 1 or 4: for the precision

requirement described in DirectX.

3.2 Analysis of Computation requirements

Before design a sharable computation unit for vertex shader type and pixel shader type, we need to understand using data, function units and processing flow in all vertex and pixel instructions individually to decide how to design the computation unit in DR-shader. We divide all vertex and pixel shader instructions in DirectX into three types by their using data and processing flows, which are:

1. Vector type: separately computes four fields (x, y, z, w) of source registers and produces four results.
2. Scalar type: only does a computation on one field of a source register and produces one result. In this type of instructions we use a changed second Taylor formula to reduce the complexity of their computations. (See Appendix A)
3. Non-computation type: only send the data of source register to bus without any computation.

In the below, we will show what instructions are in the three types with their operations and computation requirements.

Instruction	Belong	Operations	Requirements
add sub	VS, PS	Dst.x = Src0.x + Src1.x Dst.y = Src0.y + Src1.y Dst.z = Src0.z + Src1.z Dst.w = Src0.w + Src1.w	2in-fpSUM$_{32}$ ＊4
cmp	PS	Dst.x = (Src0.x >= 0)? Src1.x : Src2.x Dst.y = (Src0.y >= 0)? Src1.y : Src2.y	2in-MUX$_{32}$ ＊4

		Dst.z = (Src0.z >= 0)? Src1.z : Src2.z Dst.w = (Src0.w >= 0)? Src1.w : Src2.w	
dp2add	VS	Dst = Src0.x ∗ Src1.x + Src0.y ∗ Src1.y + Src2.w	$\underline{fpMUL_{32}}$ ∗2 $\underline{3in\text{-}fpSUM_{32}}$ ∗1
dp3 (vs)	VS	Dst = Src0.x ∗ Src1.x + Src0.y ∗ Src1.y + Src0.w ∗ Src1.w	$\underline{fpMUL_{32}}$ ∗3 $\underline{3in\text{-}fpSUM_{32}}$ ∗1
dp3 (ps) dp4	VS, PS	Dst = Src0.x ∗ Src1.x + Src0.y ∗ Src1.y + Src0.w ∗ Src1.w + Src0.w ∗ Src1.w	$\underline{fpMUL_{32}}$ ∗4 $\underline{4in\text{-}fpSUM_{32}}$ ∗1
max	VS, PS	Dst.x = (Src0.x > Src1.x)? Src0.x : Src1.x Dst.y = (Src0.y > Src1.y)? Src0.y : Src1.y Dst.z = (Src0.z > Src1.z)? Src0.z : Src1.z Dst.w = (Src0.w > Src1.w)? Src0.w : Src1.w	$\underline{2in\text{-}fpSUM_{32}}$ ∗4 $2in\text{-}MUX_{32}$ ∗4
min	VS, PS	Dst.x = (Src0.x < Src1.x)? Src0.x : Src1.x Dst.y = (Src0.y < Src1.y)? Src0.y : Src1.y Dst.z = (Src0.z < Src1.z)? Src0.z : Src1.z Dst.w = (Src0.w < Src1.w)? Src0.w : Src1.w	$\underline{2in\text{-}fpSUM_{32}}$ ∗4 $2in\text{-}MUX_{32}$ ∗4
mul	VS, PS	Dst.x = Src0.x ∗ Src1.x Dst.y = Src0.y ∗ Src1.y Dst.z = Src0.z ∗ Src1.z Dst.w = Src0.w ∗ Src1.w	$\underline{fpMUL_{32}}$ ∗4
mad	VS, PS	Dst.x = Src0.x ∗ Src1.x + Src2.x Dst.y = Src0.y ∗ Src1.y + Src2.y Dst.z = Src0.z ∗ Src1.z + Src2.z Dst.w = Src0.w ∗ Src1.w + Src2.w	$\underline{fpMUL_{32}}$ ∗4 $\underline{2in\text{-}fpSUM_{32}}$ ∗4

sge	VS	Dst.x = (Src0.x >= Src1.x)? 1.0f : 0.0f Dst.y = (Src0.y >= Src1.y)? 1.0f : 0.0f Dst.z = (Src0.z >= Src1.z)? 1.0f : 0.0f Dst.w = (Src0.w >= Src1.w)? 1.0f : 0.0f	2in-fpSUM$_{32}$ ＊4 2in-MUX$_{32}$ ＊4
slt	VS	Dst.x = (src0.x < src1.x)? 1.0f : 0.0f Dst.y = (src0.y < src1.y)? 1.0f : 0.0f Dst.z = (src0.z < src1.z)? 1.0f : 0.0f Dst.w = (src0.w < src1.w)? 1.0f : 0.0f;	2in-fpSUM$_{32}$ ＊4 2in-MUX$_{32}$ ＊4
sgn	VS	Dst.x = (Src0.x > 0)? 1.0f : (Src0.x = 0)? 0.0f : -1.0f Dst.y = (Src0.y > 0)? 1.0f : (Src0.y = 0)? 0.0f : -1.0f Dst.z = (Src0.z > 0)? 1.0f : (Src0.z = 0)? 0.0f : -1.0f Dst.w = (Src0.w > 0)? 1.0f : (Src0.w = 0)? 0.0f : -1.0f	Compare to 0$_{32}$ ＊4 2in-MUX$_{32}$ ＊8

Table.3-1 Requirements for vector type instructions

Instruction	Belong	Operations	Requirements
add sub	VS, PS	Dst.x = Src0.x + Src1.x Dst.y = Src0.y + Src1.y Dst.z = Src0.z + Src1.z Dst.w = Src0.w + Src1.w	2in-fpSUM$_{32}$ ＊4
cmp	PS	Dst.x = (Src0.x >= 0)? Src1.x : Src2.x Dst.y = (Src0.y >= 0)? Src1.y : Src2.y	2in-MUX$_{32}$ ＊4

		Dst.z = (Src0.z >= 0)? Src1.z : Src2.z Dst.w = (Src0.w >= 0)? Src1.w : Src2.w	
dp2add	VS	Dst = Src0.x ∗ Src1.x + Src0.y ∗ Src1.y + Src2.w	$\underline{fpMUL_{32}}$ ∗2 $\underline{3in\text{-}fpSUM_{32}}$ ∗1
dp3	VS	Dst = Src0.x ∗ Src1.x + Src0.y ∗ Src1.y + Src0.w ∗ Src1.w	$\underline{fpMUL_{32}}$ ∗3 $\underline{3in\text{-}fpSUM_{32}}$ ∗1
dp4	VS, PS	Dst = Src0.x ∗ Src1.x + Src0.y ∗ Src1.y + Src0.w ∗ Src1.w + Src0.w ∗ Src1.w	$\underline{fpMUL_{32}}$ ∗4 $\underline{4in\text{-}fpSUM_{32}}$ ∗1
max	VS, PS	Dst.x = (Src0.x > Src1.x)? Src0.x : Src1.x Dst.y = (Src0.y > Src1.y)? Src0.y : Src1.y Dst.z = (Src0.z > Src1.z)? Src0.z : Src1.z Dst.w = (Src0.w > Src1.w)? Src0.w : Src1.w	$\underline{2in\text{-}fpSUM_{32}}$ ∗4 $2in\text{-}MUX_{32}$ ∗4
min	VS, PS	Dst.x = (Src0.x < Src1.x)? Src0.x : Src1.x Dst.y = (Src0.y < Src1.y)? Src0.y : Src1.y Dst.z = (Src0.z < Src1.z)? Src0.z : Src1.z Dst.w = (Src0.w < Src1.w)? Src0.w : Src1.w	$\underline{2in\text{-}fpSUM_{32}}$ ∗4 $2in\text{-}MUX_{32}$ ∗4

Table.3-2 Requirements for scalar type instructions

Instruction	Belong	Operations	Requirments
branch	VS	PC = (Src0 !=0)?PC+1 : Src1	Bus to program counter
texld	PS	Dst = Mem#Src1(Src0)	Bus to texture memory

Table.3-3 Requirements for non-computation type instructions

From above tables, we conclude that there are only five kinds of computations with maximum requirements for each kind to execute any instruction, as shown in the following tables:

Computation	fpMUL	2in-fpSUM	3in-fpSUM	4in-fpSUM	Compare to 0
Maximum requirement	4	4	1	1	4

Table.3-4 Maximum requirement of each computation

3.3 Design of computation unit

In this section, we want to implement the computation unit with its area as small as possible while keeping one-cycle execution. The tradeoff in the design of computation unit is that the more sharing we want the more routing overhead we may have. Therefore, we must carefully decide whether functions of any computation unit can be shared by others. To solve this problem, we divide each computation into sub-function nodes with requirement of each node individually to discover potential sharing possibility and then use an algorithm to choose nodes covering all computations. The computations we divide are called the tree of computation requirements.

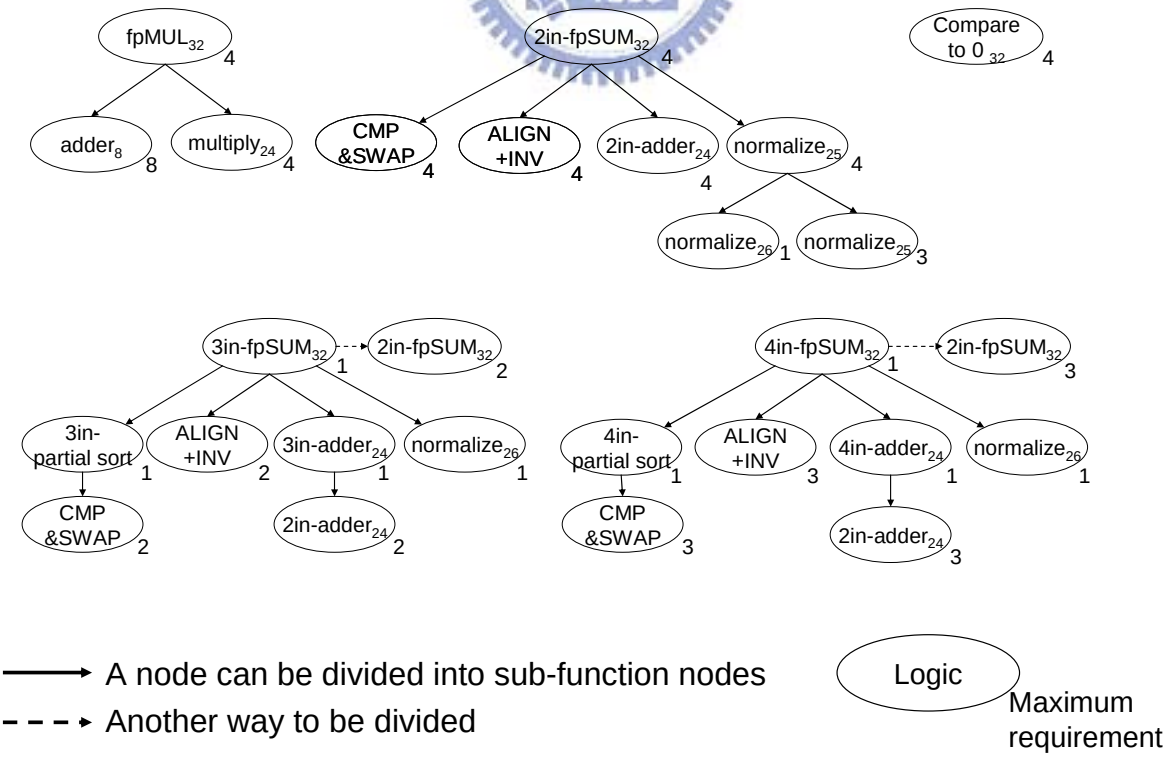


Fig.3-2 Trees of computation requirements

The meaning of covering is that if we choose a node in the tree of computation requirements, we can say the node has been covered. Besides, if all children of a node have been covered, the node also is covered. We will compare the average and maximum area requirement of all vertex and pixel instructions and choose the one with smallest average and maximum area requirement.

3.3.1 Sharing all units within *n*in-fpSUM

In *n*in-fpSUM, we find that there are some possible sharing logics when we divide *n*in-fpSUM into many sub-function nodes. There are two possible partitions of *n*in-fpSUM , which are: 1. partition 3 or 4in-fpSUM to several 2in-fpSUMs

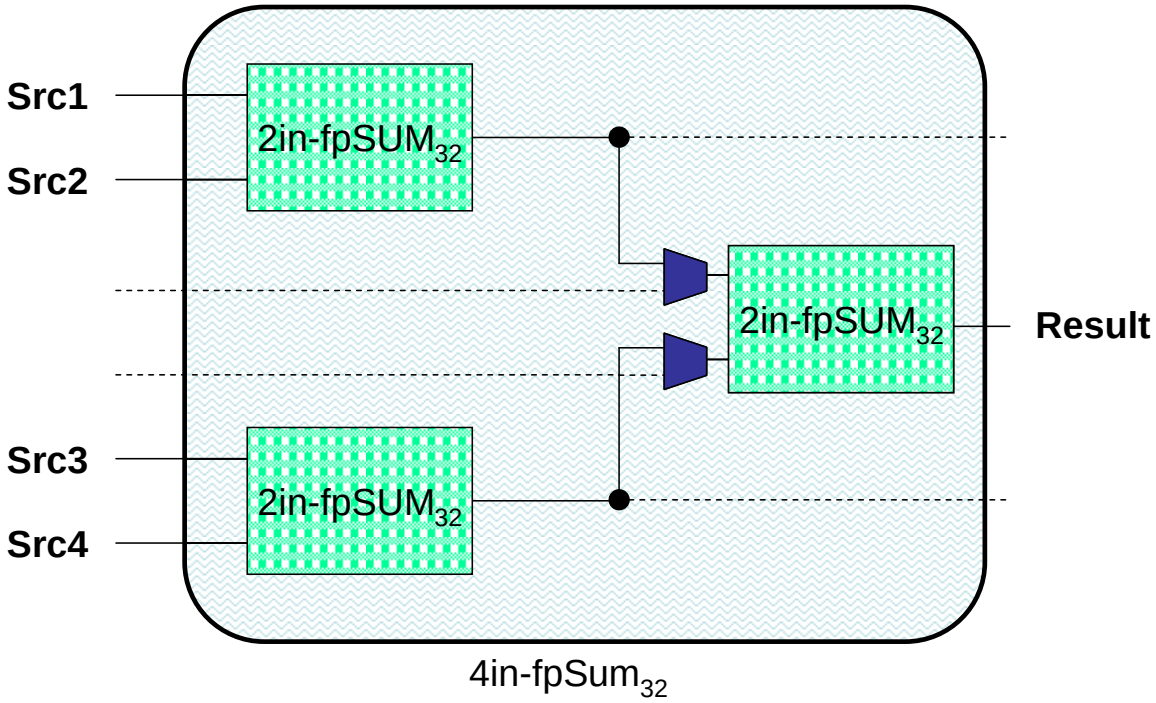


Fig.3-3 How three 2in-fpSUM$_{32}$s be reconfigured to one 4in-fpSUM$_{32}$

2. Sharing all units of *n*in-fpSUM within each other

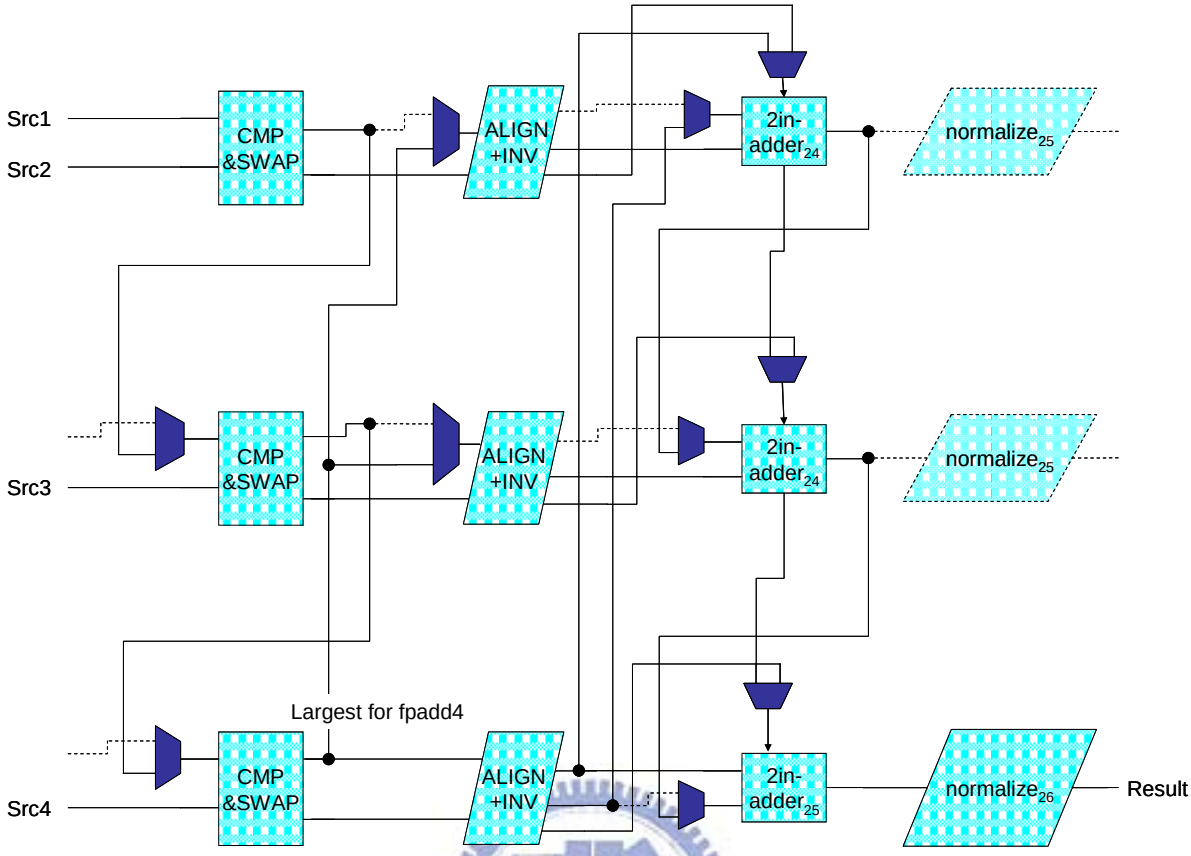


Fig.3-4 How three 2in-fpSUM$_{32}$s be reconfigured to one 4in-fpSUM$_{32}$

Although “CMP&SWAP”, “ALIGN+INV”, “normalize” can be easily shared within *n*in-fpSUM , the problem is in adders, especially at we use three 2in-adders to form a 4in-adder. In these three 2in-adders, two adders will be carry-save adders and the last one will be normal adders to add the carry and sum of the second carry-save. However, there are three problems we need to get over for this kind of design, which are:

1. We need to add four 24-bit numbers by two 24-bit carry save adder and one 24-bit normal adder. Is there any extension in adder?
2. After “ALIGN + INV”, there may be three carry-ins from the inverters. How do we add the three carry-ins by existent adders
3. The result has 24+2 bits and the sources may be minus from inverters. How do we solve the sign-extensions of minuses?

In problem1, the first carry-save adder adds three 24-bit summands from “ALIGN +

INV”, so it doesn’t need to extend. Besides, the normal adder must give 26-bit result, so it must be extend to 25-bit adder. However, in the second carry-save adder, do we need to extend it to 25-bit adder? The answer is no because we doesn’t need to process the highest bit of the carry from first carry-save adder. The figure below can give us more carefully concept:

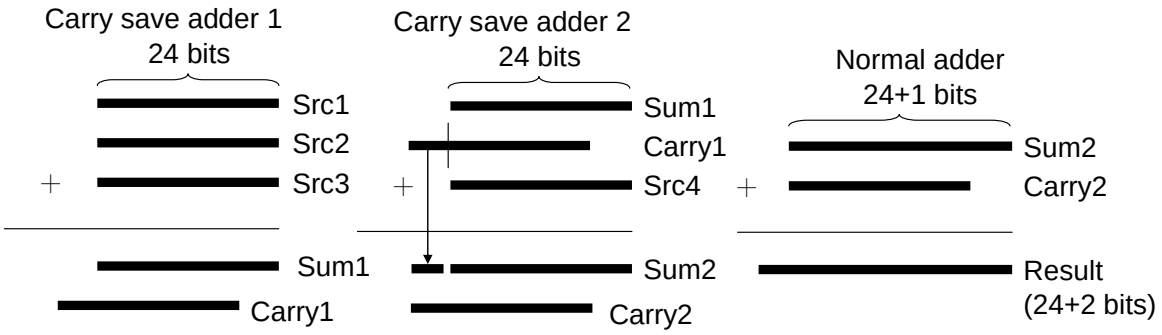


Fig.3-5 The solution of problem1

In problem2, “ALIGN + INV” may send 1-bit carry-in to adder for the negation of 2’s complement. How can we sum carry-ins, which are at most three, to summands without any additional logics? To solve this problem, we use the vacant position in the carries of the two carry-save adder to add two carry-ins. Then, the last carry-in will be added as normal carry-in by the normal adder.

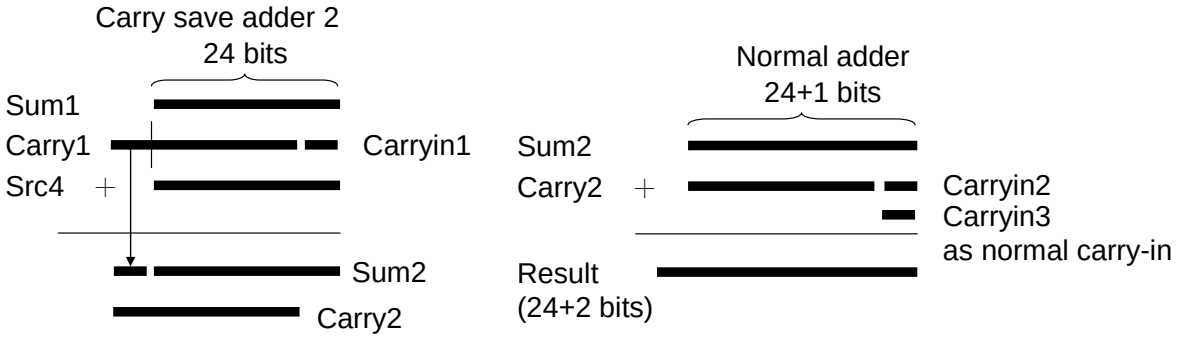
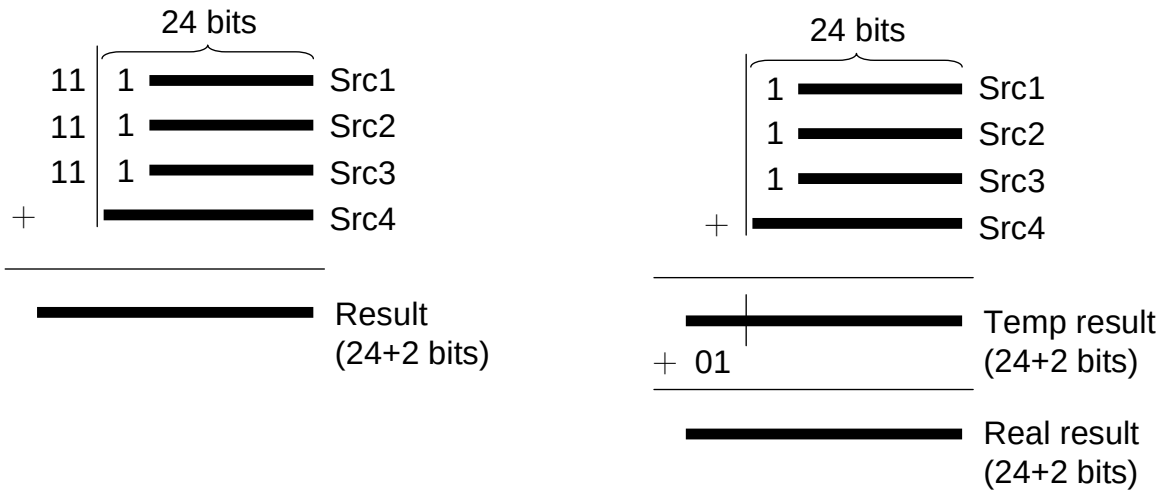


Fig.3-6 The solution of problem2

In problem3, the final solution is 26 bits and summands may be minus. Therefore, we must add compensation, which we call sign-compensation, to temporary result and get correct

result.



Number of minuses	0	1	2	3
Sign-compensation	00	11	10	01

Fig.3-7 The solution of problem3

3.3.2 Algorithm1 & 2 to choose nodes

Here, we propose two algorithms to choose node covering all computations:

- Algorithm1- minimum routing overhead: use the fewer choices to cover all computation requirements. The advantage of this algorithm is that there may be fewer routing overhead with enough sharing logic. However, the disadvantage is that it may loss some possible sharing opportunity for smaller area requirement. The steps of the algorithm are described in below:

  Step1: collect nodes with the same logic (sharable nodes) and indicate the most maximum requirement.

  Step2: group nodes into several sets and let there are no links or the same nodes within different sets and ignore the sets which only have one computation. See below:

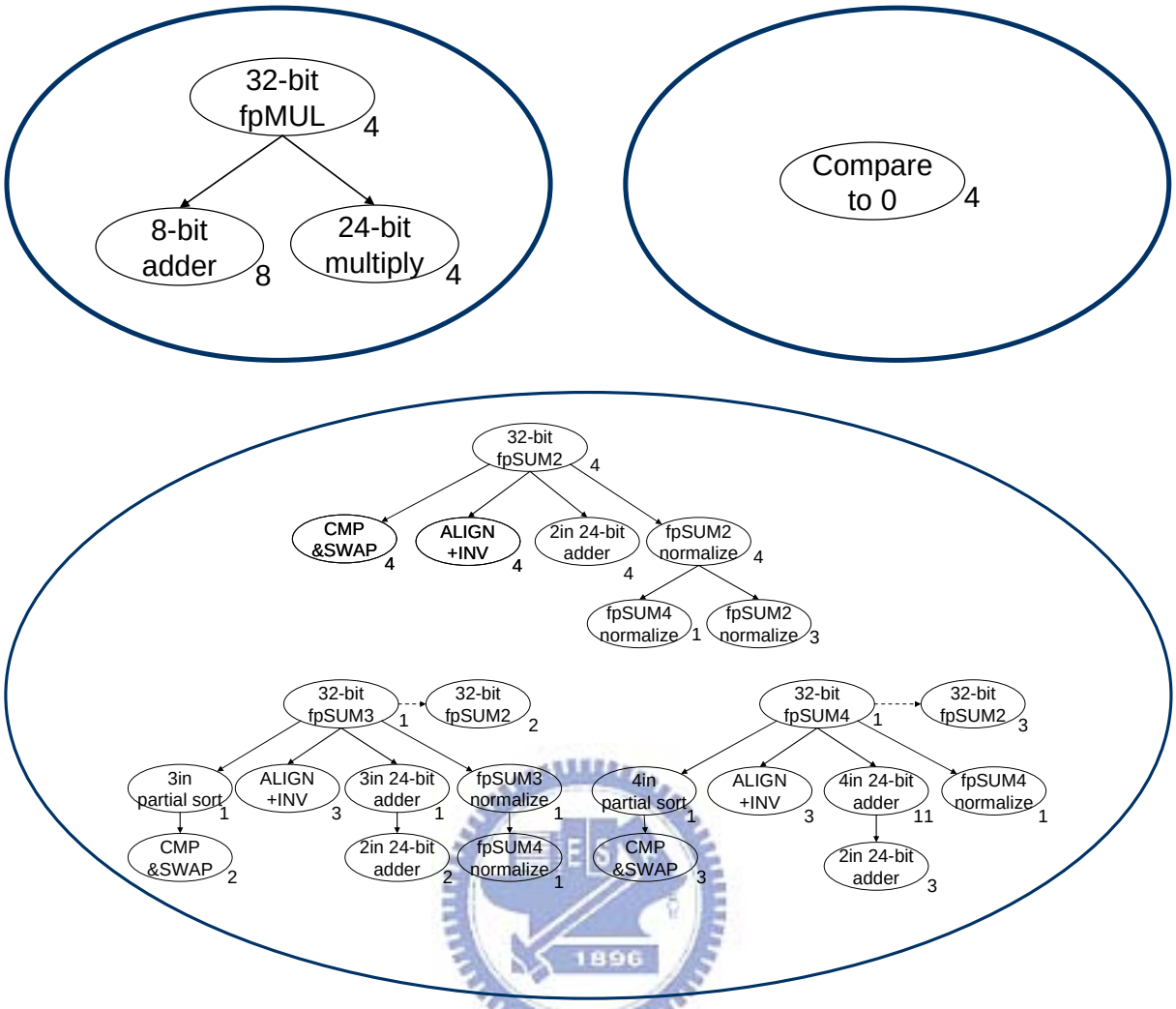


Fig.3-8 Different sets of computation trees

Step3: For each set we do that, we firstly choose a sharable node in the highest level and see if all computations have been covered. If there are computations haven't been covered, delete chosen nodes with all their children and choose another sharable node in highest level. Recursively, all computation requirements have been covered or no sharable node.

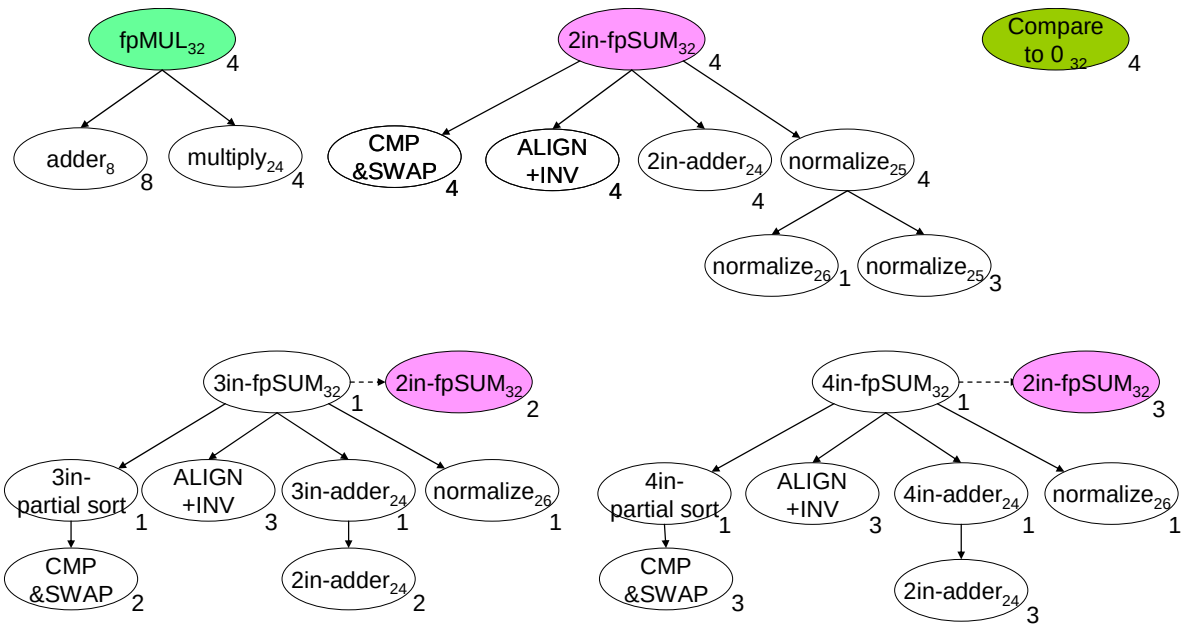


Fig.3-9 The result of minimum routing overhead

- Algorithm2-Maximum sharing logics: find more sharing choices to cover all computation requirements. The advantage of this algorithm is that there are the most sharing logics. However, the routing overhead may become more serious.

  Step1.2: the same as step1 and step2 in minimum routing overhead to group nodes into several sets.

  Step3: For each set we do that, we firstly choose a sharable node in the lowest level and see if all computations have been covered. If there are computations haven't been covered, delete chosen nodes with all their children and choose another sharable node in lowest level. Recursively, all computation requirements have been covered or no sharable node.

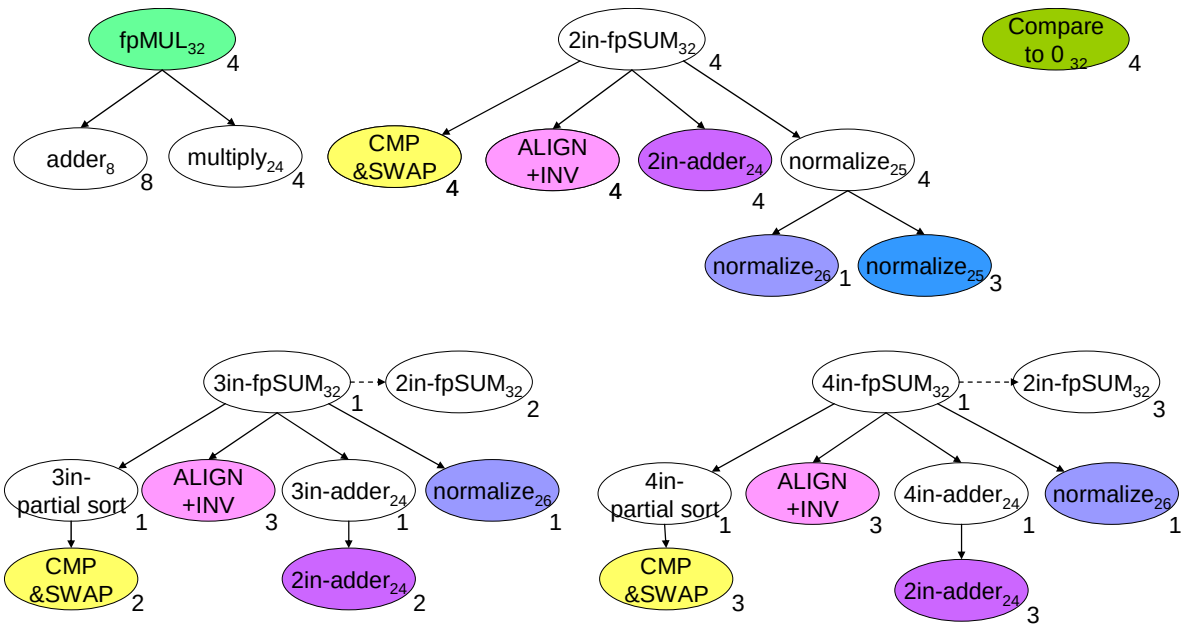


Fig.3-10 The result of maximum sharing logic

The following figures show how three 2in-fpSUM$_{32}$s be reconfigured to one 4in-fpSUM$_{32}$ in these two algorithms as an example:

3.3.3 Algorithm3-optimal area-time:

In minimum routing overhead and maximum sharing logic, we find that some factors for sharing logic haven't been considered. In these two algorithms, we choose nodes as basic unit but we don't consider about different proportions within nodes. Besides, the silicon area is not the same in all nodes. Therefore, we use a new algorithm.

- Search by integer programming: weight each sharable node with hardware cost and use integer programming to minimize total cost. Here, we estimate hardware cost of each node by number of multiplexer area it may need. The cost function has two cases. If one sharable node with the most maximum requirement it means that logic of the node will be shared to other nodes which needed the same logic. Therefore, the cost will be its implementation area without routing overhead divided by area of

a multiplexer. Except those nodes, other sharable nodes will have a cost equal to three meaning the routing overhead on two input multiplexers and one output multiplexer.

$$\text{cost} = \begin{cases} \dfrac{\text{area of an implementation}}{\text{area of a multiplexer}} & \text{randomly choose one nodes with the most maximum requirement} \\ 3 & \text{otherwise (meaning routing overhead)} \end{cases}$$

Fig.3-11 Cost function of search by integer programming

The advantage of this algorithm both consider sharing logic and routing overhead. However, the disadvantage is that the qualities of results depend on the precision of cost. For the integer programming, we change the display of computation trees and give more information.

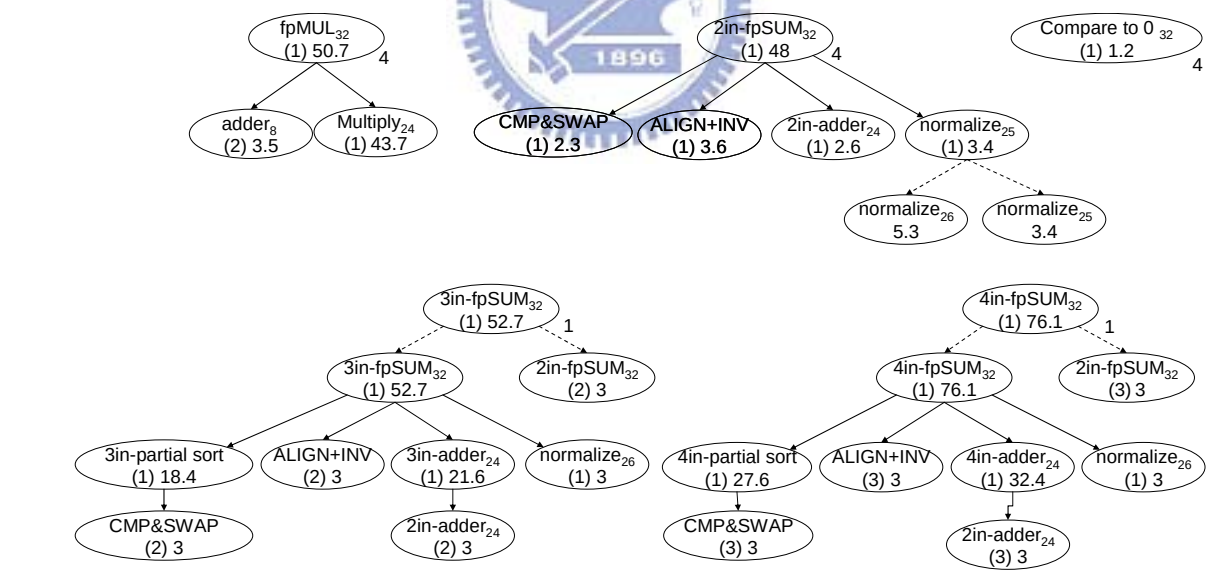


Fig.3-12 The computation trees for integer programming

Step1.2: the same as step1 and step2 in minimum routing overhead to group nodes into several sets.

Step3: To find optimal result using integer programming, we set

- **Variables**

$$I_i = \#\text{of implementation from node}_i \quad \forall I_i \geq 0$$

- **Constraints from**

$$\text{if node}_i \text{ has real lines linking to children}$$

$$I_i + \frac{1}{R_c} * \text{Req}_c \geq \text{Req}_i \quad \text{for each node}_c \in \text{children of node}_i$$

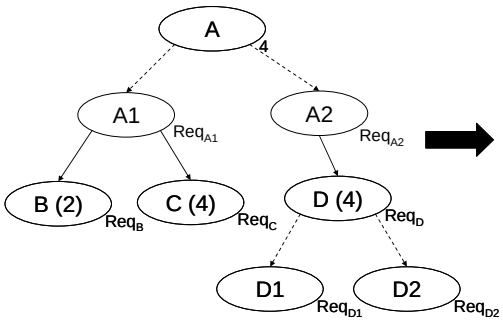
$$\text{if node}_i \text{ has dot lines linking to children}$$

$$I_i + \sum \frac{1}{R_c} * \text{Req}_c \geq \text{Req}_i \text{ for all node}_c \in \text{children of node}_i$$

- **Objective**

$$\text{minimize} \sum I_i * \text{cost of node}_i \quad \forall \text{node}_i$$

Step4: reduce all *Reqs* from leaves to roots and get all constraints for integer programming.



$$I_A + \text{Req}_{A1} + \text{Req}_{A2} \geq 4 \text{ --node}_A$$

$$\begin{cases} I_{A1} + \frac{1}{2}\text{Req}_B \geq \text{Req}_{A1} \\ I_{A1} + \frac{1}{4}\text{Req}_c \geq \text{Req}_{A1} \end{cases} \text{--node}_{A1}$$

$$I_{A2} + \frac{1}{4}\text{Req}_D \geq \text{Req}_{A2} \text{ --node}_{A2}$$

$$I_D + \text{Req}_{D1} + \text{Req}_{D2} \geq \text{Req}_D \text{ --node}_D$$

$$\Rightarrow \begin{cases} I_A + \text{Req}_{A1} + \text{Req}_{A2} \geq 4 \\ I_{A1} + \frac{1}{2} I_B \geq \text{Req}_{A1} \\ I_{A1} + \frac{1}{4} I_c \geq \text{Req}_{A1} \\ I_{A2} + \frac{1}{4}(I_D + I_{D1} + I_{D2}) \geq \text{Req}_{A2} \end{cases} \Rightarrow \begin{cases} I_A + I_{A1} + \frac{1}{2} I_B + I_{A2} + \frac{1}{4}(I_D + I_{D1} + I_{D2}) \geq 4 \\ I_A + I_{A1} + \frac{1}{4} I_c + I_{A2} + \frac{1}{4}(I_D + I_{D1} + I_{D2}) \geq 4 \end{cases}$$

Fig.3-13 An example for *Req*s reducing

Step5: apply integer programming and get the result with minimum cost

The result of optimal area-time is in below:

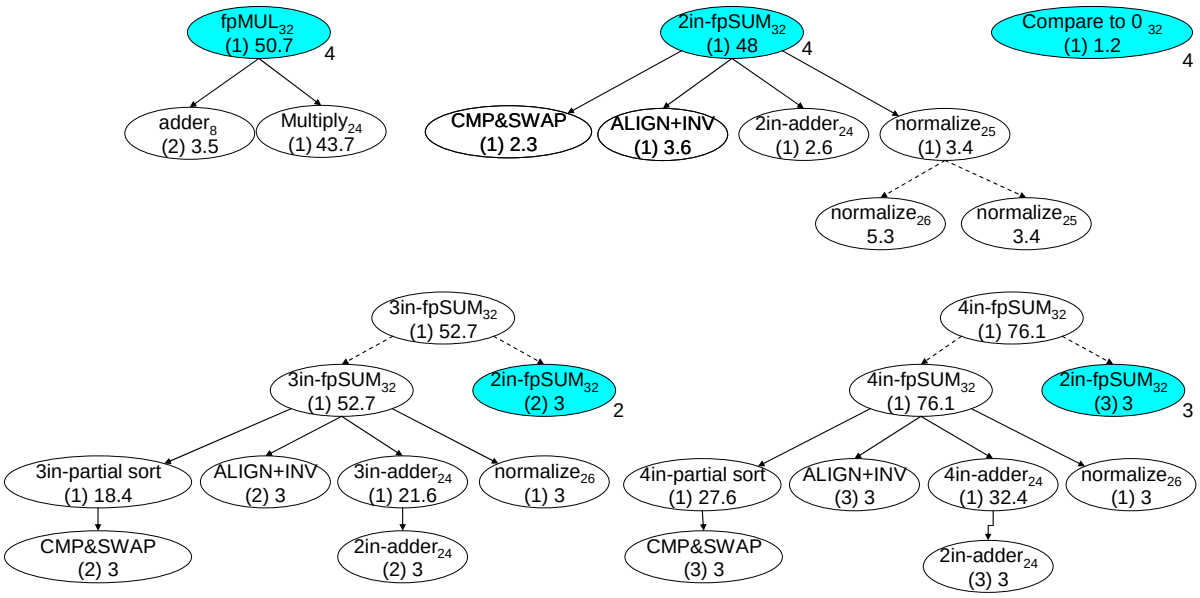


Fig.3-14 The result of optimal area-time algorithm

We find that the result of optimal area-time algorithm is the same as minimum routing overhead because of too few possible solution. Then, we compare the result of three algorithms

3.3.4 Comparison within algorithms

Firstly, we show the comparison within three algorithms

	Average area requirement (um^2)	Maximum area requirement (um^2)
Minimum routing overhead	985793.5938	2,000,547.5
Maximum sharing logic	986,095.4688	2,105,722.5
Optimal area-time	985793.5938	2,000,547.5

Table.3-5 Average and maximum area requirement of three algorithms

Finally choose the result of minimum routing overhead/optimal area-time because of the smallest average and maximum area requirements. To compare the two kinds of result in detail, we find they only differ in the choices of how to reconfigure 2in-fpSUM$_{32}$s to 3in-fpSUM$_{32}$ or 4in-fpSUM$_{32}$. In addition, we analyze the sharing logics between 2in-fpSUM$_{32}$S and 4in-fpSUM$_{32}$ and find out the failure of result2 is in the routing overhead. The critical path of minimum routing overhead/optimal area-time is:

Delay time = CMP&SWAP ➔ ALIGN+INV ➔ 2in-add$_{24}$ ➔ normalize$_{25}$ ➔ $\underline{\text{MUX}}_{32}$ ➔ CMP&SWAP ➔ ALIGN+INV ➔ 2in-add$_{24}$ ➔ normalize$_{25}$

= 3.93 + 6.14 + 4.47 + 4.72 + $\underline{0.76}$ + 3.87 + 6.01 + 2.54 + 7.56 (ns)

= 40ns with time overhead 0.76ns (1.9%)

The area requirement of minimum routing overhead/optimal area-time is:

Area = 2in-fpSUM$_{32}$*3 + $\underline{\text{MUX}}_{32}$*2

= 313425 + $\underline{8837.5}$ (um^2)

= 332027.5um^2 with area overhead 8837.5um^2 (2.66%)

The critical path of maximum sharing logic is:

Delay time = $\underline{\text{MUX}}_{32}$ ➔ CMP&SWAP ➔ $\underline{\text{MUX}}_{32}$ ➔ CMP&SWAP ➔ $\underline{\text{MUX}}_{32}$ ➔ CMP&SWAP ➔ ALIGN+INV ➔ 2in-add$_{24}$ ➔ $\underline{\text{MUX}}_{32}$ ➔ 2in-add$_{24}$ ➔ $\underline{\text{MUX}}_{32}$➔ 2in-add$_{25}$ ➔ normalize$_{26}$

= $\underline{0.6}$ + 3.9 + $\underline{0.75}$ + 3.59 + $\underline{0.77}$ + 5.08 + 6.24 + 1.15 + $\underline{0.86}$ + 1.42 + $\underline{0.85}$ + 9.31 + 6.63 (ns)

= 43.86ns with time overhead 2.12ns (4.83%)

The area requirement of maximum sharing logic is:

Area = 2in-fpSUM$_{32}$ * 3 + $\underline{\text{MUX}}_{32}$* 6

= 108972.5 + 106872.5 + 107345.0 + $\underline{18882.5}$ (um^2)

= 332307.5um^2 with area overhead 18882.5 um^2 (5.68%)

Because the time overhead and area overhead of maximum sharing logic are much more than those overhead of minimum routing overhead/optimal area-time, we finally choose the result of minimum routing overhead/optimal area-time as our design of the computation unit.

3.4 Architecture of DR-shader

After finish the computation unit, we can build DR-shader. The architecture of DR-shader is below:

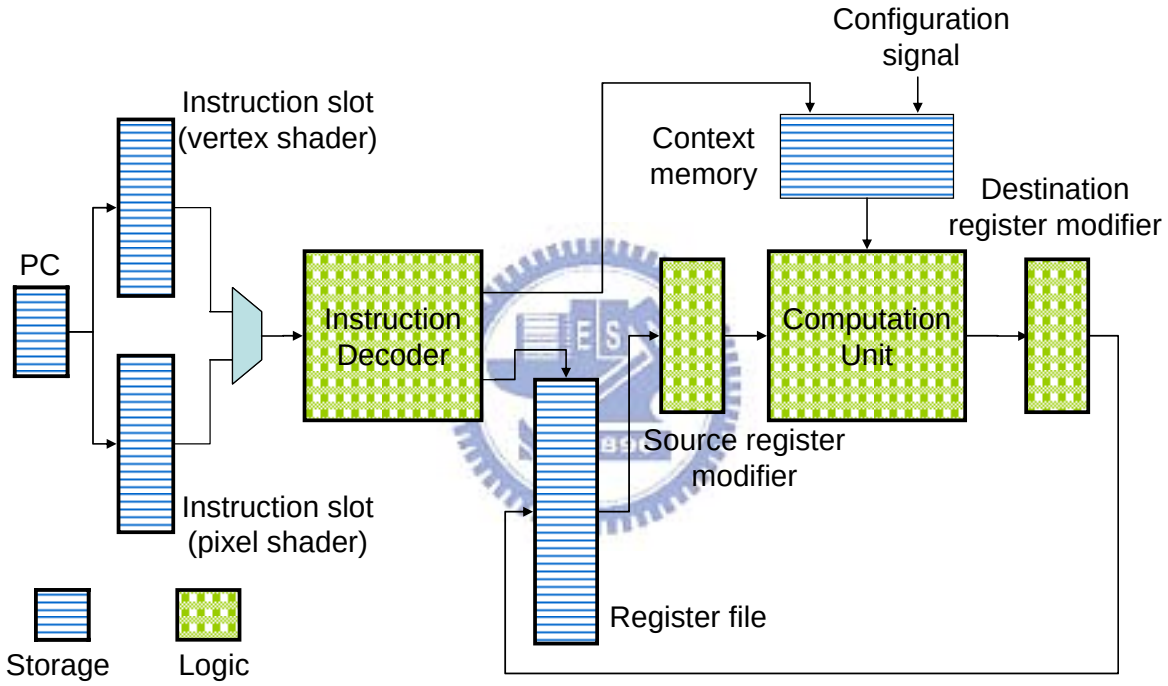


Fig.3-15 The architecture of DR-shader

In the architecture, there are some necessary hardware overheads:

- More logic in the sharable computation unit to support all vertex and pixel instructions with routing overhead
- Context memory to store the configuration of each instruction
- One more instruction slot to store vertex and pixel shader codes simultaneously

Therefore, the area of DR-shader may be larger than the area of vertex shader or pixel shader

for the ability to reconfigure between vertex shader type and pixel shader type. However, we will find whether its flexibility deserve be added to upgrade shader utilization in our simulations.

3.5 Design of workloads monitor logic

In this section, we firstly descript the properties of DR-shader and the hardware overhead. Then, we will descript the design of vertex/pixel workloads monitor logic. There are two assumptions of reconfigure property for DR-shader:

1. Order of processing: In the beginning, all DR-shaders will be reconfigure to vertex shader type because of no workload in pixels. Then, DR-shaders will be often reconfigured to vertex shader type or pixel shader type according to the various in the workloads between vertices and pixels until all vertices have been processed. Finally, all DR-shaders will be reconfigured to pixel shader type for remaining pixels.
2. Reconfiguring timing: The configuration of each DR-shader only can be changed after it finish a vertex/pixel to avoid needing one more register file for temporary results.

The purpose of the workloads monitor logic is to control number of DR-shaders with pixel shader type in DR-shader unit and let stall cycles of all shaders as few as possible. To achieve this goal, we base on three kinds of information to control number of DR-shaders with pixel shader type, which are:

- Expected number of DR-shaders with pixel shader type is equal to number of used intervals in pixel queue. (the size of intervals will be determined later)
- Current number of DR-shaders with pixel shader type is recorded in workload monitor logic.
- Job end signal is sent by each DR-shader, telling workload monitor logic which

DR-shaders finish their job.

At every cycle, we count the difference between expected and current number of DR-shaders with pixel shader type. If the expected number is bigger than current number, we change finishing DR-shaders with vertex shader type to other type by the difference. Otherwise, we change finishing DR-shaders with pixel shader by the difference.

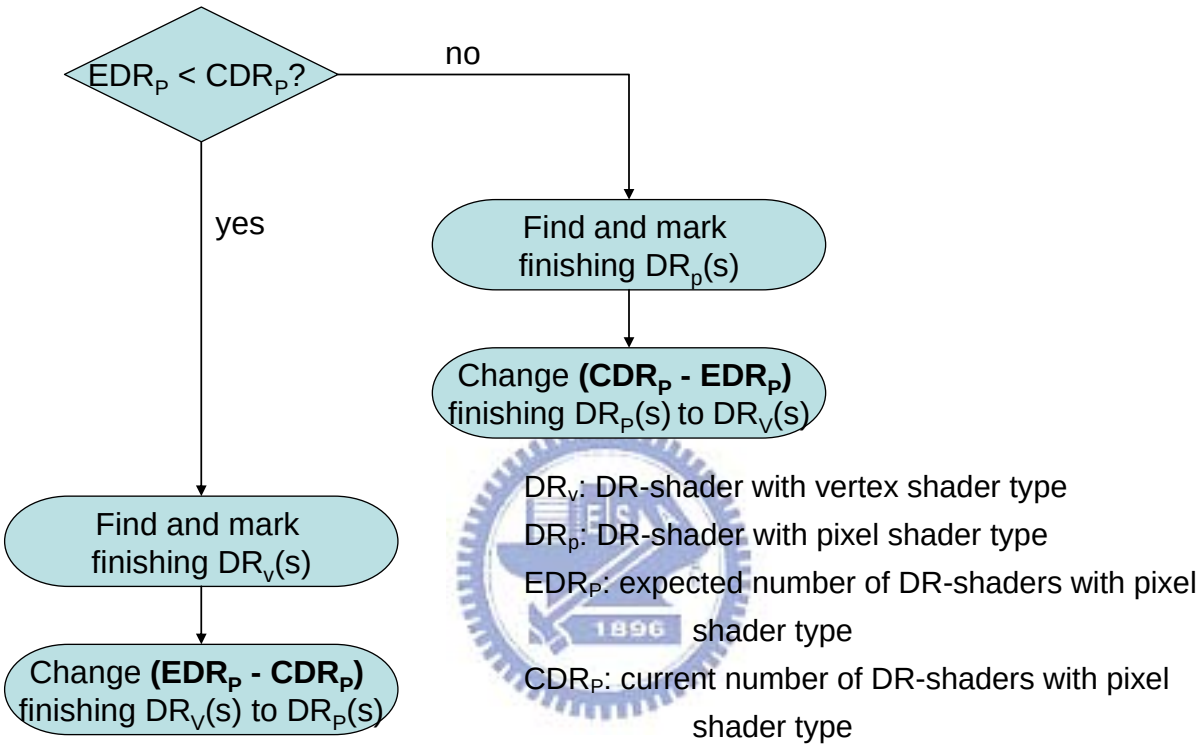


Fig.3-16 Flowchart of workloads monitor logic

Chapter 4 Simulation

4.1 Simulator of DR-shader

For this thesis, we build a cycle-based simulator referenced from SiS. The input of the simulator is 3Dmark05, we consider about information which is listed below:

- If a primitive is clipped (culled) or pass
- Number of tiles produced from each primitive
- If a tile is blocked by preZ or not
- Number of pixels can be produced from each pass tile
- Vertex shader codes and pixel shader codes

The output of simulator is the execution time from vertex processing to pixel processing of a frame with the information about shader utilization. There are also some parameters we can set for different environments we want, listed below:

- Clip information
  - Throughput of the clipping unit
- PreZ information
  - Throughput of the PreZ unit
- Shader information
  - Throughput of the vertex input
  - Size of pixel queue
  - Numbers of DR-shader, vertex shaders and pixel shaders
  - Number of batches in each shader
  - Latencies of each instruction
- Texture information

- Texture unit access cycles
- Miss rate of the texture memory
- Miss penalty
- Throughput of texture units

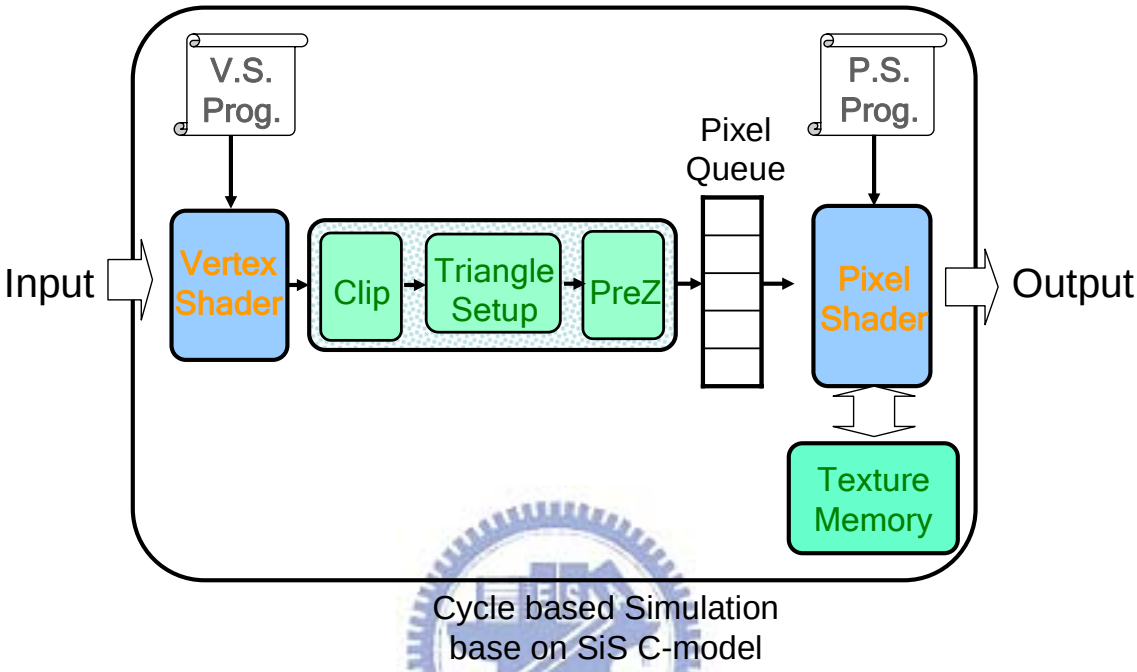


Fig.4-1 The cycle based simulator base on SiS

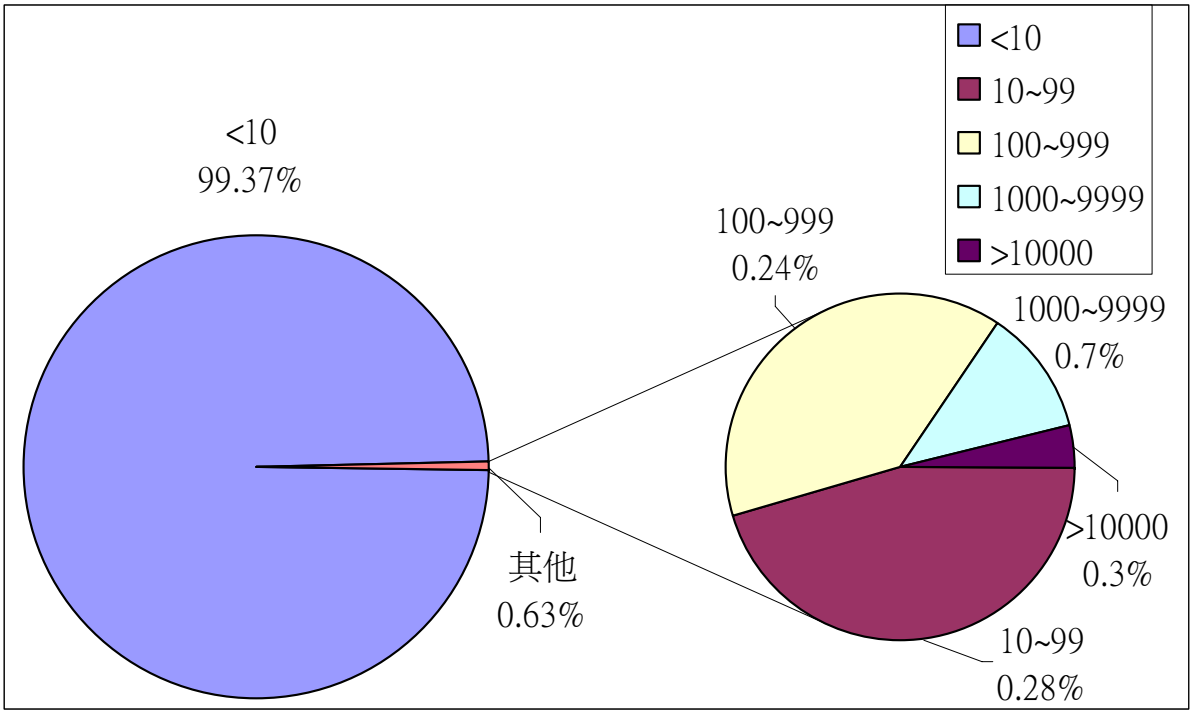
4.2 Simulation1

In this section, we will decide a proper proportion between vertex shaders and pixel shaders and the size of pixel queue. For the goal, we assume number of vertex shaders is three, and other parameter setting listed below:

- Clip information
  - Throughput of the clipping unit = **unlimited**
- PreZ information
  - Throughput of the PreZ unit = **unlimited**
- Shader information
  - Throughput of the vertex input = **unlimited**

- Number of vertex shaders = **3**
- Latencies of each instruction = **8**
- Number of batchs in each shader = **8**

Then, we gather workload statistics of pixels in every cycle. The workload in each cycle is counted as number of pixels in the cycle product with their execution time. We display the pie chart of pixels' workload:



Average = 36.353
Standard deviation = 279.114

Fig.4-2 The pie chart of the pixels' workload in every cycle

We choose the average workload as number of pixel shaders when there are three vertex shaders. Under 3 vertex shaders with 37 pixel shaders, we simulate the relation between the size of pixel queue and execution time:

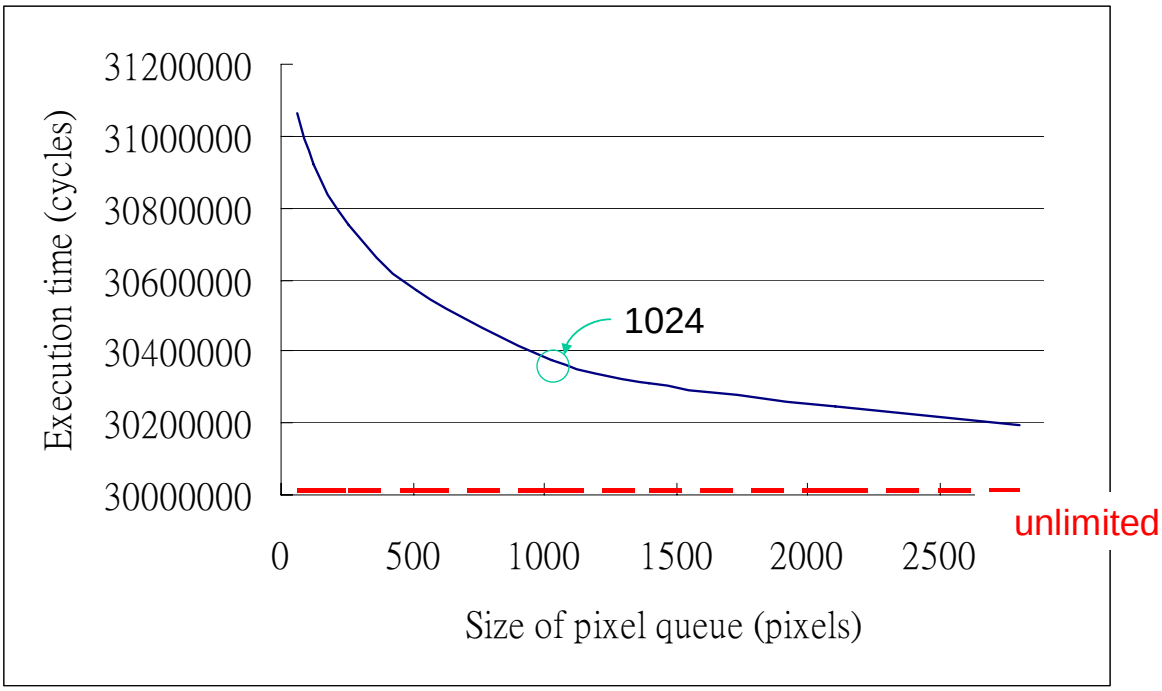


Fig.4-3 The relation between the size of pixel queue and execution time

By the graph, we choose the size of pixel queue is 1024 (pixels).

4.3 Simulation2

In this section, we decide the size of intervals in pixel queue and number of vertex shaders and pixel shaders be changed to DR-shaders. We use the parameters decided above, listed below:

- Clip information
  - Throughput of the clipping unit = **unlimited**
- PreZ information
  - Throughput of the PreZ unit = **unlimited**
- Shader information
  - Throughput of the vertex input = **unlimited**
  - Latencies of each instruction = **8**

- Number of batchs in each shader = **8**
- Number of vertex shaders = **3**
- Size of pixel queue = **1024**
- Total number of shaders = **40 (3 + 37)**

➢ Texture information

- Texture unit access cycles = **8**
- Miss rate of the texture memory = **0**
- Throughput of texture unit = **unlimited**

Firstly, we simulate the relation between the size of intervals and execution time and get below graph:

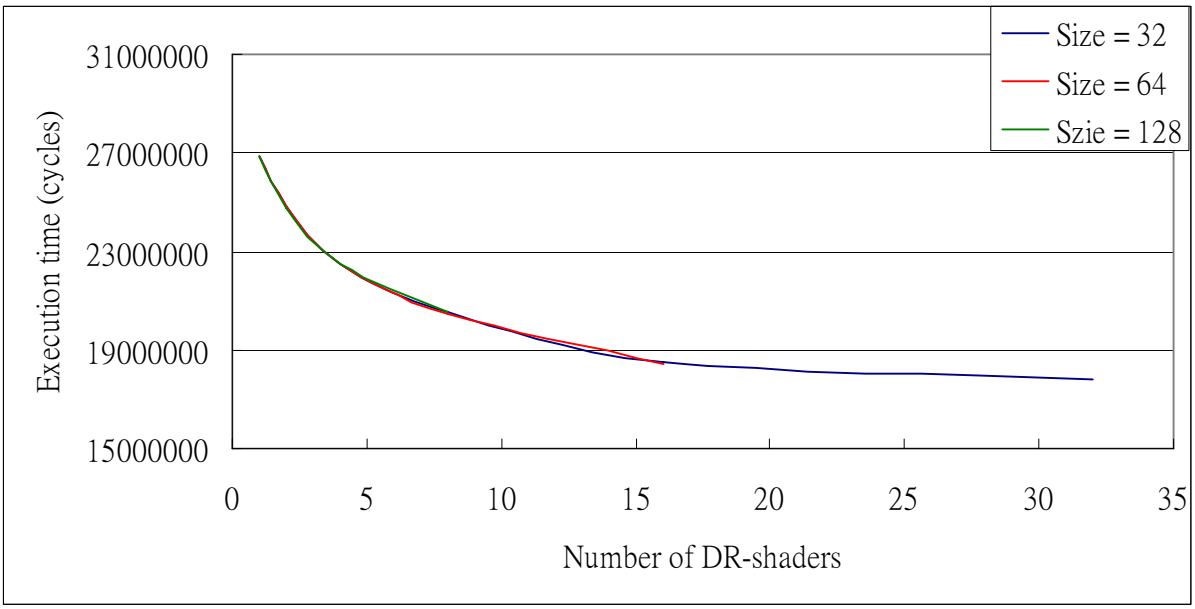


Fig.4-4 The relation within the size of intervals, number of DR-shaders, and execution time

It is apparent that the size of intervals doesn't have a great influence on the execution time. Therefore, we choose the size of intervals is equal to 32 (pixels) for the flexibility.

Secondly, we simulate the relation between the number of DR-shaders and time-area product with the size of intervals equal to 32:

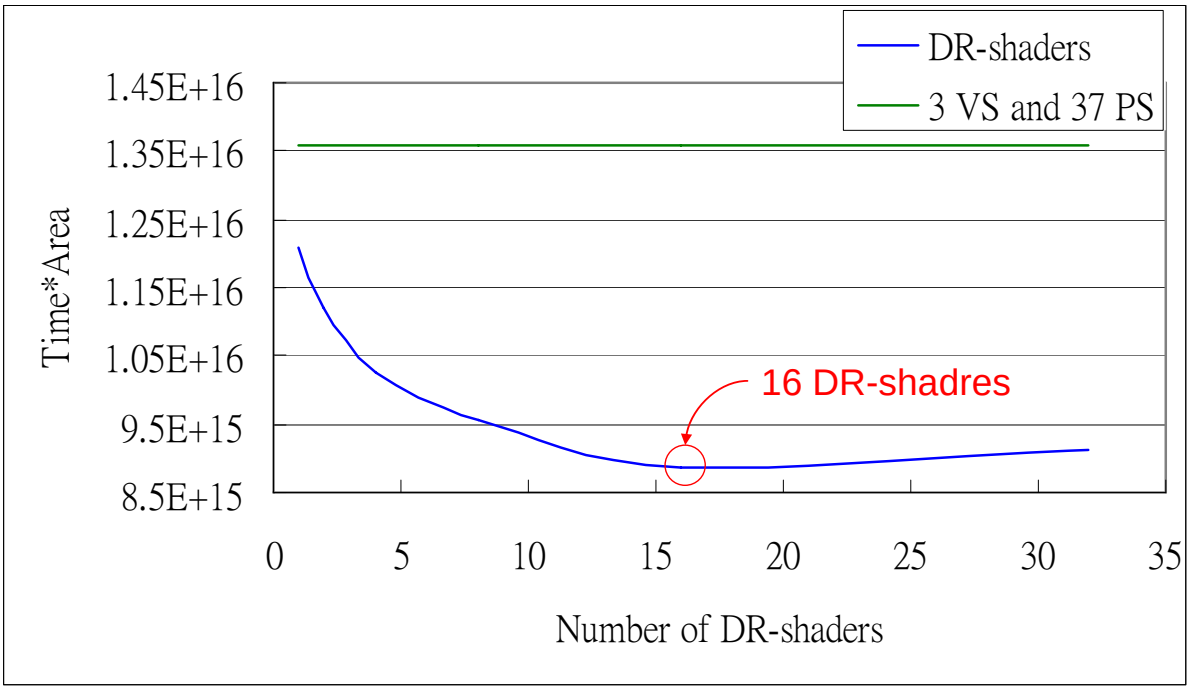


Fig.4-5 The relation between the number of DR-shaders and area-time product

The time-area products have a minimum value at number of DR-shaders equal to 16. For the analysis in detail, we list time, area, and utilization of each shader type in below:

Number of each kind of shader			Time (cycles) [Speed up]	Area (um^2) [Ratio]	Time*Area [Ratio]
DR	VS	PS			
0	3	37	30,373,118 [1]	447,827,831.4 [1]	13,601,927,566,796,306.56 [1]
16	0	24	18,474,735 [1.644]	480,609,124 **[1.073]**	8,879,126,204,482,140 **[0.653]**

Table.4-1 The time, area, and area-time product

Number of each kind of shader			Stall cycles [Utilization] (DR-shaders)	Stall cycles [Utilization] (Vertex shaders)	Stall cycles [Utilization] (Pixel shaders)
DR	VS	PS			
0	3	37		45,056,703 [0.50552]	544,974,677 [0.515063]
24	0	16	35,664,243 [0.879348]		78,431,817 [0.82311]

Table.4-2 The utilization of each shader type

We choose 24 DR-shaders with 16 pixel shaders in DR-shader unit and the size of intervals in pixel queue is 32 pixels as our final result. This kind of design will have a great improvement in shader utilization and execution time with a few of hardware overhead and area-time product will be reduced to **65.3** %.

Chapter 5 Conclusion

5.1 Discussion

To design hardware by reconfigurable architecture, we need to consider sharable logic, hardware overhead from routing path, sharing time and usable opportunity, etc. However, this kind of problem may be very complex and we couldn't consider all causes at once. The priorities of those causes must be carefully decided for computation time and better result. There may be a trade-off between sharable logic and routing overhead. So, how to decide whether a logic be shared or not will be one of the most important problems in the reconfigurable architecture.

5.2 Future work

Utilization loss in texture load misses:

In our observation, long texture load miss penalty will cause shader utilization loss greatly. Although DR-shaders can be reconfigured at finishing, they stalled a long time when load misses. We may reconfigure those load-miss DR-shaders with pixel shader type to vertex shader type and try to reduce utilization loss in texture load miss. To solve this problem, we may need one more register file to buffer its temporary result and one more program counter for current state as hardware overhead. The reconfigure timing may be changed from an end of a vertex or pixel to any cycle. The workload monitor logic may need to change the configuration of load-miss DR-shaders with pixel shader type to vertex shader type. The proposed architecture is below:

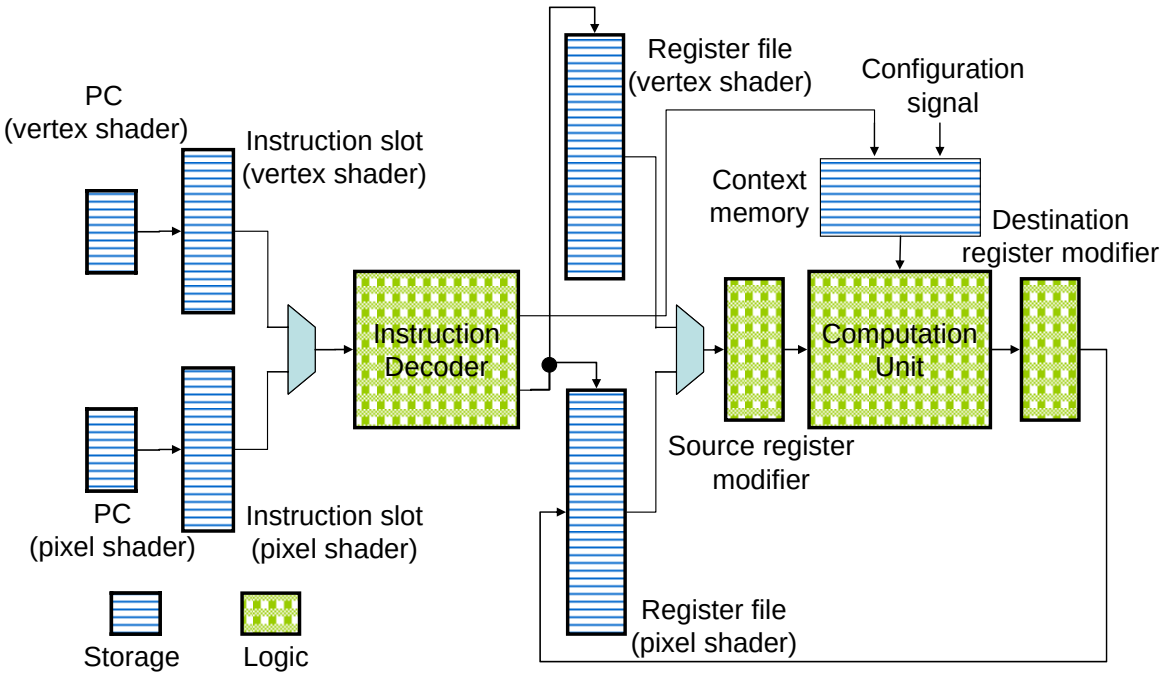


Fig.5-1 The proposed architecture to reduce utilization loss in texture load misses

5.3 Conclusion

In this thesis, we have prove that besides reducing the hardware cost by sharing logic, the flexibility of reconfigurable architecture can be used to adapt various workloads everywhere and try to upgrade the utilization of whole system. In our design, the execution time has been greatly shortened with limited hardware overhead.

The level of reconfigurable architecture can be anywhere and used in different levels in the same time. In our design, besides DR-shader can be reconfigured between vertex shader type and pixel shader type, the computation unit of DR-shader also can be reconfigured to execute all vertex and pixel instructions for area saving.

Reference

[1] DirectX 9.0 Programmer's Reference © 1995-2002 Microsoft Corporation.

[2] E. Lindholm, M. J. Kilgard, and H. Moreton. A userprogrammable vertex engine. In Proceedings of ACM SIGGRAPH 2001, pages 149–158, August 2001.

[3] M. D. McCool. SMASH: A next-generation API for programmable graphics accelerators. Technical report CS-2000-14,Computer Graphics Lab, University of Waterloo,2000.

[4] M. Olano. A Programmable Pipeline for Graphics Hardware .Ph.D. thesis, University of North Carolina at Chapel Hill, 1998.

[5] Chris J. Thompson Sahngyun Hahn Mark Oskin : Using Modern Graphics Architectures for General-Purpose Computing：A Framework and Analysis ,International Symposium on Microarchitecture archive Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture table of contents .Istanbul, Turkey ,Pages: 306 – 317.

[6] Jiawen Chen1 Michae I. Gordon1 William Thies Matthias Zwicker Kari Pulli Frédo Durand：A Reconfigurable Architecture for Load-Balanced Rendering ,Massachusetts Institute of Technology ,Nokia Research Center .Graphics Hardware (2005)M. Meissner, B.- O. Schneider (Editors).

[7] Victor Moya, Carlos González, Jordi Roca Agustín Fernández, Roger Espasa：A Single (Unified) Shader GPU Microarchitecture
for Embedded Systems , Department of Computer Architecture, Universitat Politècnica de Catalunya ,HiPEAC 2005

[8] Alireza Shoa and Shahram Shirani Dept. of Electrical and Computer Eng., McMaster University, Hamilton, Canada：Run-time Reconfigurable Systems For Digital Signal Processing Applications: A Survey

[9] Austin Robison and Abe Winter：An Overview of Graphics Processing Hardware, March 14, 2006

[10] Karl Hillesland and Anselmo Lastra University of North Carolina, at Chapel Hill：GPU Floating-Point Paranoia

Appendix A. Reducing for second order Taylor formula (reference from SiS)

$$f(x) \cong f(x\ ) + f'(x_p)(x - x_p) + \frac{f''(x_p)}{2!}(x - x_p)^2 \text{ where } x_p = \frac{1}{2}(x_0 + x_1)$$

$$= \frac{f''(x_p)}{2}x^2 + (f'(x_p) - x_p f''(x_p))x + f(x_p) - x_0 f'(x_p) + \frac{x_p^{\ 2}}{2} f''(x_p)$$

$$\Rightarrow x = x_0 + dx$$

$$= \frac{f''(x_p)}{2}dx\left[dx + \frac{2f'(x_p)}{f''(x_p)} + 2(x_0 - x_p)\right] + \left[f(x_p) + (x_0 - x_p)f'(x_p) + \frac{(x_0 - x_p)^2}{2} f''(x_p)\right]$$

$$\Rightarrow \text{let, } a = \frac{f''(x_p)}{2}, \quad b = \frac{2f'(x_p)}{f''(x_p)} + 2(x_0 - x_p) = \frac{2f'(x_p)}{f''(x_p)} + 2(x_0 - x_1),$$

$$c = f(x_p) + (x_0 - x_p)f'(x_p) + \frac{(x_0 - x_p)^2}{2} f''(x_p) = f(x_p) + \frac{(x_0 - x_1)}{2} f'(x_p) + \frac{(x_0 - x_1)^2}{8} f''(x_p)$$

$$\text{So, } f(x) \cong \begin{cases} c + dx * a * (b + dx) & \text{if } a \geq 0 \text{ and } b \geq 0 \\ c - dx * a' * (b + dx) & \text{if } a < 0 \text{ and } b \geq 0 \\ c - dx * a * (b' - dx) & \text{if } a \geq 0 \text{ and } b < 0 \\ c + dx * a' * (b' - dx) & \text{if } a < 0 \text{ and } b < 0 \end{cases} \text{ where } a' = -a \text{ and } b' = -b$$