國 立 交 通 大 學

資訊科學與工程研究所

碩 士 論 文

繪圖處理器之材質貼圖下有效率之材質記憶體系統設計

The efficient texture memory system design for texture mapping in GPU

研 究 生：張 辰 瑋

指導教授：鍾 崇 斌 博士

中 華 民 國 九 十 六 年 八 月

繪圖處理器之材質貼圖下有效率之材質記憶體系統設計

The efficient texture memory system design for texture mapping in GPU

研 究 生：張 辰 瑋　　Student： Chen-Wei Chang

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

國 立 交 通 大 學
資 訊 科 學 與 工 程 研 究 所
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

July 2007

Hsinchu, Taiwan, Republic of China

中華民國 九十六 年 七 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 張辰瑋 君

所提論文:

繪圖處理器之材質貼圖下有效率之材質記憶體系統設計

The efficient texture memory system design for texture mapping in GPU

合於碩士資格水準、業經本委員會評審認可。

口試委員： [illegible] 謝萬雲

單智君 鍾崇斌

指導教授： 鍾崇斌

所　　長： 曾文貴

中 華 民 國 九十六 年 七 月 二十五 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研究所 系統設計 組， 95 學年度第 2 學期取得碩士學位之論文。

論文題目：繪圖處理器之材質貼圖下有效率之材質記憶體系統設計
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	■ 中華民國 101 年 8 月 31 日公開
校外網際網路	■ 中華民國 101 年 8 月 31 日公開

■ 全文電子檔送交國家圖書館

授 權 人：張辰瑋

親筆簽名：張辰瑋

中華民國 96 年 8 月 31 日

國　立　交　通　大　學

博碩士紙本論文著作權授權書

（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研究所 系統設計 組， 95 學年度第 2 學期取得碩士學位之論文。

論文題目：繪圖處理器之材質貼圖下有效率之材質記憶體系統設計
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：張辰瑋

親筆簽名： 張辰瑋

中華民國 96 **年** 8 **月** 31 **日**

國家圖書館
博碩士論文電子檔案上網授權書

（提供授權人裝訂於紙本論文本校授權書之後）

ID:GT009455600

本授權書所授權之論文為授權人在國立交通大學資訊科學與工程研究所 95 學年度第 2 學期取得碩士學位之論文。

論文題目：繪圖處理器之材質貼圖下有效率之材質記憶體系統設計
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：張辰瑋

親筆簽名： 張辰瑋

民國 96 **年** 8 **月** 31 **日**

繪圖處理器之材質貼圖下有效率之材質記憶體系統設計

學生：張辰瑋　　　　　　　　　　　　　　　　　　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

材質貼圖於現今繪圖處理器架構中是一常見普遍的技術。此技術除了處理材質過濾外，還必須先執行其他前置運算：座標產生，位置轉換及材質像素的查詢。此部分運算於整體執行時間中有相當比例，且隨著材質過濾複雜程度上昇而上昇。

在這篇論文中，我們以材質記憶體放置法為切入點改善在三種可能且常見材質快取記憶體的支援下，此部分所需的時間（拿取材值像素的平均時間），而此材質記憶體放置法是以材質快取記憶體命中率，位置轉換所需時間，及平均查詢快取記憶體次數為目標來達成目標。

由結果顯示，新的材質放置法於三種可能且常見快取記憶體支援下可改善約略 2~9% 的平均拿取材值像素時間。

The efficient texture memory system design for texture mapping in GPU

Student：Chen-Wei Chang Advisor：Dr, Chung-Ping Chung

Institute of Computer Science and Engineering
National Chiao-Tung University

Abstract

Texture mapping is a common rendering technique in current Graphic Processing Unit architecture. In order to have better synthesized picture, many texture data will be referenced and accessed. We found that the technique may take significant part of total scene execution time on preliminary operations to access these referenced data. The operations could be coordinate generation, address translation and texels look up. And the time is increased as filtering algorithms are more complex.

In this thesis, we are going to improve this part of time (average texels access time of texture filtering) by proposing new texture placement under three possible and common texture cache supports. And the placement is target to achieve the goal by improving the texture cache hit rate, average cache access counts and address translation time.

As the result shows that the new placements could gain 2~9% upgrade in average texels access time of texture filtering under the three common texture cache support in current GPU architecture.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

In Three-Dimensional (3-D) computer graphics, texture mapping is a common and one of the successful techniques in high quality image synthesis. It is responsible for rendering the 3-D scene by adding detail, surface texture, pattern, surface normal or color to a 3-D object and become more and more complex due to the requirement of 3-D scene realism and special effect [1][2].

Basically, in order to have quality of synthesized image, more texels data will be referenced, and more computation will be invoked. We found that the complex texture mapping technique may take a significant part of scene total execution time on the preliminary operations. The operations are accessing the required referenced texels data in the texture memory system for texture filtering. They contain address calculations, coordinate generations and texel look ups for those required texels in the texture memory system. Thus, whether the texture memory system is well design or not may affect the average texels access time of texture filtering.

In this thesis, we are going to improve the average texels access time of texture filtering under three possible and common texture cache supports. In order to achieve the objective, We are going to improve texture cache hit rate, average cache access counts and address translation time by proposing the new placement for saving the average texels access time of texture filtering.

1.1 Motivations

Texture placement, placing the texture in the texture memory, is what we consider the most important and fundamental solution, as the following reasons.

1. Texture placement will affect the texture cache hit rate.
2. Texture placement will affect average cache access counts.
3. Texture placement will affect the address translation complexity.

In first reason, since the texture placement is the decision of how to place the texture in the texture memory, if the placement is well design, the cache hit rate could be improve and average texels access time will also be improve. If not, it may introduce cache hit rate loss and increase average texels access time of texture filtering.

The third reason, due to some complex texture mapping techniques, i.e. bilinear filtering, need more than one texel data, the required texels maybe scatter over many texture cache lines, i.e. 2, 4, cache lines, according to the placement algorithm. Moreover, it will also affect the continuousness of required texels within a cache line.

The second reason, if the placement has regular property, it can be translated through some fast bit-wise logic circuit. If not, the address translation time will increase due to the abnormality of placement and also increase average texels access time of texture filtering.

If we have the hardware support to help us to retrieve the required texels in the same cache line, we may retrieve them in one cache access. If not, we may have to access them in another cache access. However, if we do not have such hardware support, the continuousness factor could be an important cause. If the required texels are within a cache line and continuousness, we can retrieve them by using wider bus or a common technology, burst mode. If they are not continuous, we may retrieve them in another time of cache access.

1.2 Objectives

We are going to propose the new texture placement that is how to place the texture in the texture memory. And the placement is aim to save the average texels access time of texture filtering under three possible texture cache supports by improving the three aspects:

1. The placement could improve the texture cache hit rate.
2. The placement could be easy to translate through some easy ideas.
3. The placement could improve the average cache access counts.

1.3 Organization about this thesis

In Chapter 2, we explain the graphic processing flow and texture mapping techniques. In Chapter 3, proposed the new placement concept, the address translation idea, possible fast address translation logic circuit and we will list the three possible and common texture cache supports. In Chapter 4, we will describe our experiment goal, environment and methodology; evaluate average texels access time of texture filtering under three kinds of possible texture cache support. In Chapter 5, there are discussion, future work and conclusion.

Chapter 2 Background and Related research

In section 2.1, we will give a brief concept of rendering pipeline in Graphic Processing Unit (GPU). And we'll find that our research is focus on pixel processing, the third pipeline stage. In section 2.2, we are going to explain the texture mapping techniques which include the topic of the texture data structure, and the responsible function unit, called texture unit and processing flow of texture mapping. Finally, some related research will be study.

2.1 GPU rendering flow

The rendering flow in current GPU can be roughly divided into four parts which are vertex processing, rasterization, pixel processing, depth processing based on its pipeline stage, as shown in figure 2-1-1.

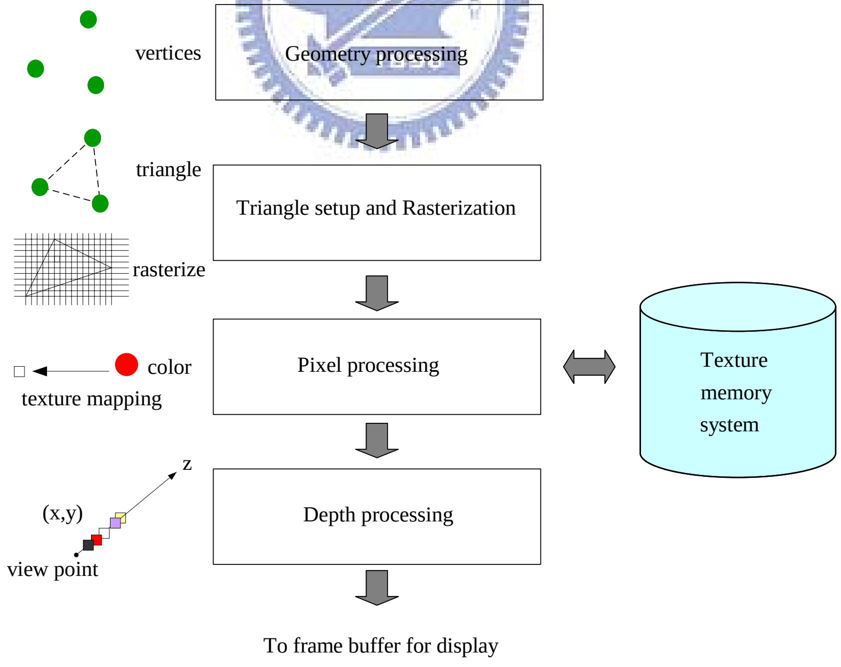


Figure 2-1-1 rendering pipeline of GPU

In figure 2-1-1, the vertex processing is done in vertex shaders. The majority works in them is performing vertex's coordinate translations. These translations actually is a serial of coordinate translations from vertex’s local coordinate to global environment coordinate and finally translate to view point coordinate.

After vertex processing, the following stage is triangle setup and rasterization. Triangle setup is responsible for assembling primitive according to their view point coordinate. That is finding three vertices which are valid to be assembled into a triangle (primitive). Based on the primitive, rasterization is responsible for interpolating this primitive. In another word, rasterization interpolate each primitive into some fragments. Thus, we obtain the fragments, pixels before output to frame buffer are called fragments.

The pixel processing is done in pixel shaders. Its majority work is coloring each fragment with the texture which is usually stored in the texture memory system, i.e. memory, cache, through the dedicate function units, texture units.

The final processing is depth processing. Since the are many fragments have the same x, y coordinate in the screen but are different in z coordinate, we are target to find out which fragment will not be covered (closest to the view point) and will be final displayed on the screen. Thus, the works in depth processing is simply comparison the depth value (Z value) of each fragment which has the same x, y and pass these fragments to frame buffer for display on the screen.

2.2 The Texture mapping techniques

Texture mapping technique usually invokes multiple textures or MIP maps as samples and also invokes the other techniques, such as bilinear interpolations or trilinear interpolations to produce different amounts of realism. Moreover, the major process of the whole texture mapping is done in special function units in the stage of pixel processing, called texture unit. We will introduce them respectively as following organization:

In section 2.2.1, we will first give an overview of texture data structure.

In section 2.2.2, we will introduce the processing flow of texture mapping within the texture unit which is responsible for coloring the fragment. And it contains coordinate generation, address translation, texels look up and texture filtering.

2.2.1 The textures

Texture is simply a data structure which is used as color reference in pixel shader and can be viewed as a picture or bitmap image. Its dimensions are usually restricted to power of 2 for hardware implementation. Moreover, the width and the height of the texture can be different [4].

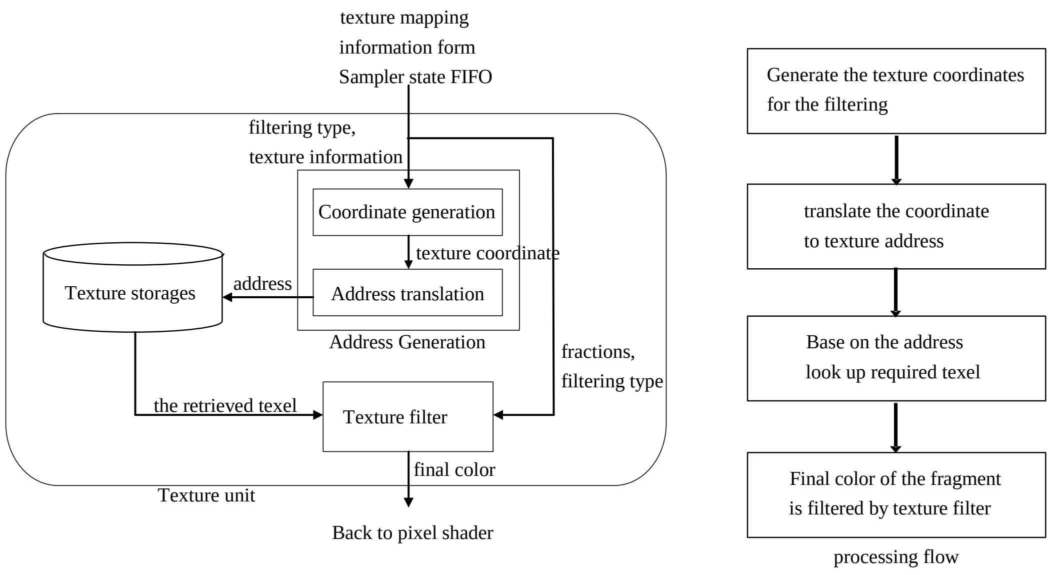
A pixel of a texture, call a texel, is a basic cell of a texture and is usually made up of four components, which is R (Red), G (Green), B (Blue), A (Alpha) respectively. And each component is usually one byte width. However, with the High Dynamic Range (HDR) introduced in DirectX 10, a texels can be up to 16 bytes, which each component is up to 4 bytes for more precision

Textures are usually stored in the off-chip large texture memory and on-chip fast texture cache for quickly retrieval in GPU. When the pixel shader needs to paint the fragment, it needs the color information in the textures, thus goes to the texture storage to get the required texels for that fragment.

2.2.2 Texture mapping process flow within Texture Unit

Texture mapping is done within the texture unit. The texture unit is usually in the Pixel shader, since it is responsible for color the fragment according to the filtering type. The processing flow can be roughly classified into four operations, as shown in figure 2-2-2-1.

From Sampler State FIFO [14], we know the required information of how to color the fragment. The information may contain texture filtering type, texture coordinates, base texture address of the required texture, etc.



texture mapping information form Sampler state FIFO
filtering type, texture information
Coordinate generation
texture coordinate
Texture storages
address
Address translation
Address Generation
fractions, filtering type
the retrieved texel
Texture filter
final color
Texture unit
Back to pixel shader
Generate the texture coordinates for the filtering
translate the coordinate to texture address
Base on the address look up required texel
Final color of the fragment is filtered by texture filter
processing flow

Figure 2-2-2-1 the texture unit and texture mapping processing flow

After we have the information, the coordinate generator will generate the required texture coordinates based on the filtering type. These texture coordinates may be further translated into texture addresses by the following address translation unit. These translated addresses will be used to look up the required texel in the texture cache next to the texture unit. The texture cache is a fast SRAM storage space, it store the texels information, and can be any traditional cache configuration. After we have retrieved the required texels, we can perform the texture filtering algorithm based on the filtering type, texels, and other information in the texture filter. The final color will be sent back to the pixel shader.

In current high-end graphic card, there are multiple texture units in the pixel shader for performance issue [11]. Moreover, most of the texture units also have multiple texture address units and texture filters which allow processing more filtering algorithms or more complicated filtering algorithm in parallelism [11]. Texture units are allowed to generate the final color of filtering algorithm per cycle.

As mentioned in chapter one, we are target to save average texels access time of filtering. The average texels access time may contain the time spending on coordinate generation, address translation and cache look up.

2.2.2.1 Address translation

In GPU processor, the texel is indexed through texture coordinates, i.e. u, v, coordinates, But in texture memory system, texture is indexed through texture memory address. Since the indexing methods are different between GPU processor and memory system;, thus we need a special function unit which is target to perform the address translation, as shown in figure 2-2-2-1-1.

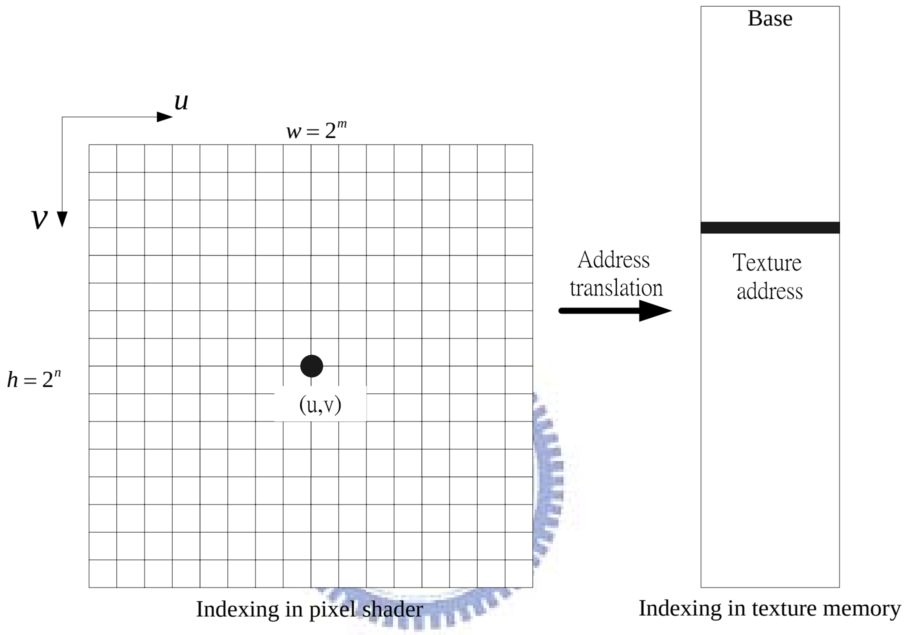


Figure 2-2-2-1-1 the concept of address translation.

The address translation could be viewed as a translation function with texture coordinates (i.e. u, v), texture dimensions, base address of texture as input and generate the translated address as output [4].

Thus, the complexity of address translation may relate to the texture placement algorithm. If the placement is well design, the address translation could be easy to translate. If not, the address translation complexity could be complicated.

2.2.2.2 Texture filtering

Texture filtering is the method used to obtain the color for a fragment by using the colors of nearby texels in some texture. In another words, it is an attempt to find a value at some point by giving a set of discrete samples at nearby points. Thus, texture filtering is a kind of process that for any given fragment, it goes to loop up some required texels, and calculated the final color for that fragment.

Since one fragment may not usually correspond exactly to one texel, there can be different types of correspondence between a fragment and the texel/texels depend on the position of the textured surface relative to the viewer.

For example, one fragment is exactly the same as one texel of the texture, that is one to one mapping. Closer than that, the texels are larger than fragments. Texels are needed to be scaled up appropriately, known as texture magnification. Farther away, each texel is smaller than a fragment, that is one to many. In this case an appropriate color has to be picked based on the covered texels, via texture minification.

Because the different correspondence between fragments and texels mentioned before, that may necessitate reading all of entire texels and combining their values to correctly determine the fragment color. This process would be a potentially expensive operation. Mipmapping technique is introduced in [12]. It can avoid this by pre-calculating, recursively sampling the texture and storing it in a quarter down to a single texels. As the textured surface moves farther away, the texture being applied switches to the pre-sampled size. Different sizes of the mipmap are referred to as 'levels', with Level 0 being the largest size (used closest to the viewer), and increasing levels used at increasing distances. As shown in figure 2-2-2-2-1, we have an example of how the mipmaps looks like.

The filtering method can be roughly classified according to the image quality and computation complexity.

The first one is nearest neighbor interpolation. It is the fastest and crudest filtering method - it is only look up the closest texels' color for the mapped fragment. While fast, this results in a large number of artifacts, thus image quality is the worst.

The second one is nearest neighbor with mipmapping. According to the fragment's Z value, we select the two closest mipmaps first. For each mipmap, by applying Nearest neighbor interpolation, we got two selected texels. Finally, the final color for that fragment is the result of weighted average of those two texels. This reduces the aliasing and shimmering significantly, but does not help with blockiness.

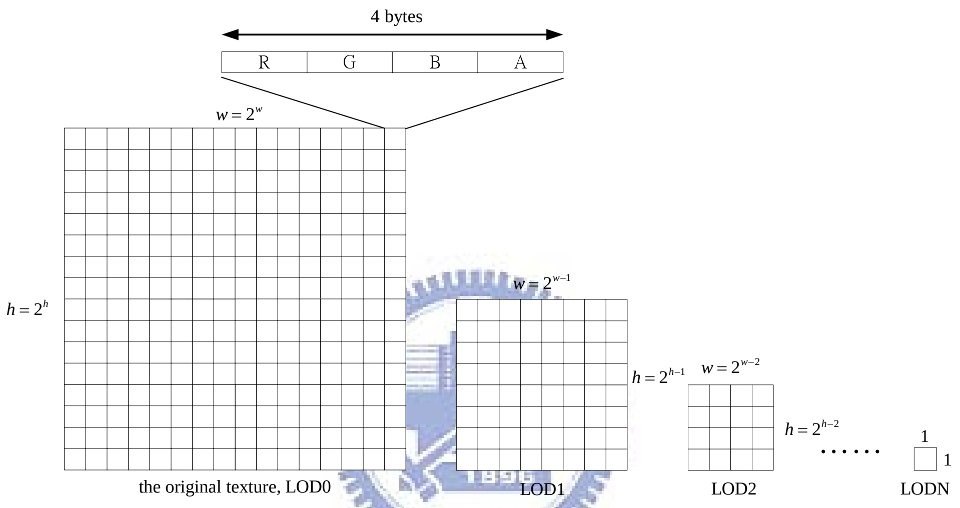


Figure 2-2-2-2-1 the mipmaps and data structure of a texel

The third on is bilinear filtering. In this method the four closest texels on a nearest mipmap level to the fragment center are chosen, and final color for that fragment is the color of weighted average among them. Figure 2-2-2-2-2 shows the concept of bilinear filtering algorithm. Bilinear filtering is almost invariably used with mipmapping; though it can be used without, it would suffer the same aliasing problems as nearest neighbor. Moreover, bilinear filtering is the basic component of the following filtering method. And they can be viewed as several pieces of bilinear filtering

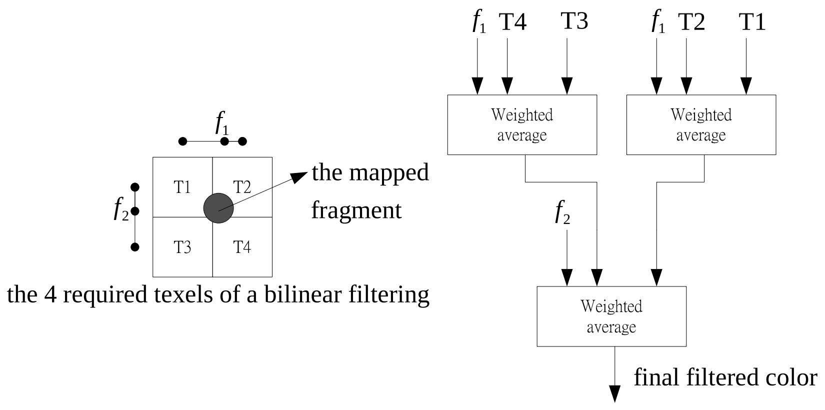


Figure 2-2-2-2-2 the concept of bilinear filtering

The fourth one is trilinear filtering. It can be treated as a weighted average of two pieces of bilinear filtering. For each of two closest mipmap levels, perform the bilinear filtering. And the final color for that mapped fragment is the color which is the weighted average of the two bilinear filtering results. Of course, closer than Level 0 there is only one mipmap level available, and the algorithm reverts to bilinear filtering.

The final one is anisotropic filtering. It is the highest quality filtering available in current consumer 3D graphics cards. If we need to color a plane which is at an oblique angle to the camera, bilinear or trilinear filtering would give us insufficient horizontal resolution and extraneous vertical resolution. Anisotropic is a method of enhancing the image quality of textures on surfaces that are far away and steeply angled with respect to the camera. The final color of that mapped fragment is the color which is the "trilinearly" average of the n pieces of trilinear filtering results. The value n called anisotropic ratio, horizontal direction to vertical direction, is defined by application.

Finally, we summarize a table of texture filtering methods as shown below.

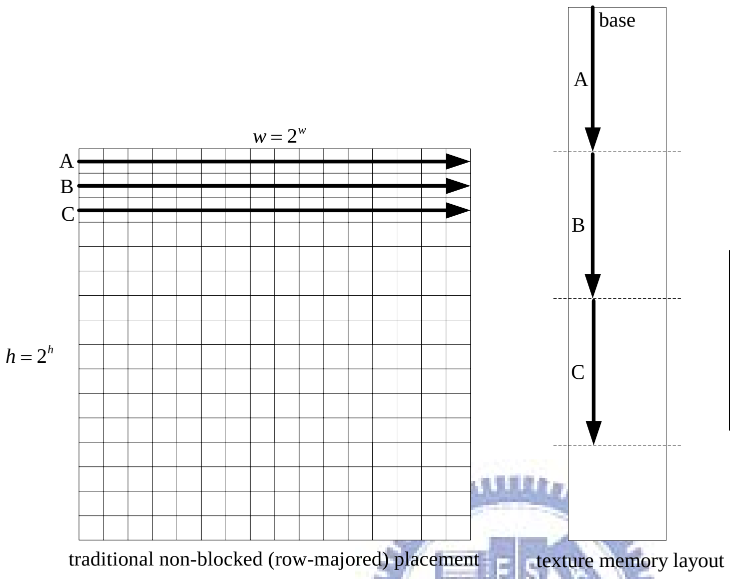
Filtering Type	# of MipMap	# of Texel / MipMap	# of Texels (# of Bi)	Filtering Algorithm
nearest neighbor interpolation	1	1	1(0)	Apply color of the closest texel respect to that fragment center
nearest neighbor with mipmapping	2	1	2(0)	Weighted average of two nearest neighbor interpolation.
Bilinear	1	4	4(1)	Weighted average among nearest four texels on the closest mipmap.
Trilinear	2	4	8 (2)	Weighted average of two bilinear filtering which are on two closest mipmaps respectively.
n:1 Anisotropic n=2,4,8,16	2	4n	8n (2n)	Weighted average of n trilinear filtering which are on two closest mipmaps respectively.

Table 2-2-2-2-1 summary of texture filtering algorithms

2.3 Related Research

2.3.1 Nonblock placement

Traditionally, texture is placed in the texture memory by using row-major concept, as shown in figure in 2-3-1-1. This is also known as Nonblock placement.



Address translation equation :
$A = base + u + [v << \log_2 w]$
where A is the translated address
base is the base address of the texture
u, v is texture coordinate
w is texture width

Address transltion equation

Figure 2-3-1-1 Nonblock placement and address translation equation

The concept of placement is straightforward and intuitional. Address translation is also straightforward. However, since texture filtering have spatial locality, that is the required texels of a bilinear filtering is in a 2 by 2 region, and the required texels of next bilinear filtering is usually closed to the current one, Nonblock placement could be considered as a non-efficiency placement due to the long texture's width and always row-major.

Among these four required texels, the upper and lower two will in two adjacent rows respectively, as shown in figure 2-3-1-2. However, if the row of texture is very long, the required texels will be separated far away in the texture memory.

Moreover; when the cache line size is smaller or equal to the size of a single row data structure, the required texels which are in two adjacent rows will be placed in two different cache lines. Thus the upper/lower two required texels will be in different cache lines and for those texels in the same cache line, they are continuous.

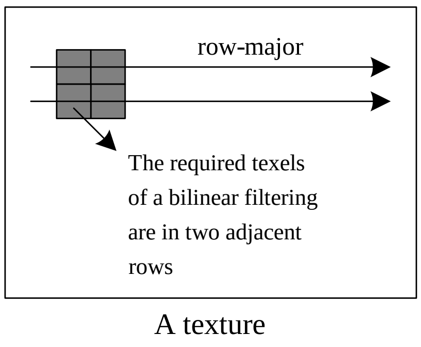
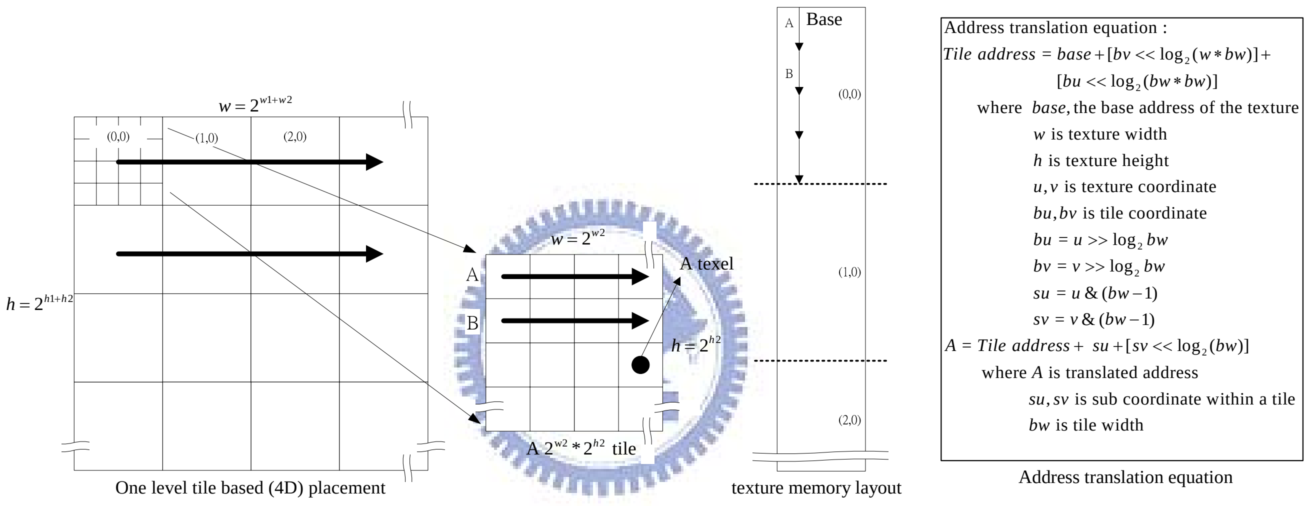

row-major
The required texels of a bilinear filtering are in two adjacent rows
A texture

Figure 2-3-1-2 access condition of Nonblock placements

2.3.2 4D placement



Address translation equation :

$Tile\ address = base + [bv << \log_2(w * bw)] + [bu << \log_2(bw * bw)]$

where $base$, the base address of the texture
w is texture width
h is texture height
u, v is texture coordinate
bu, bv is tile coordinate
$bu = u >> \log_2 bw$
$bv = v >> \log_2 bw$
$su = u \,\&\, (bw - 1)$
$sv = v \,\&\, (bw - 1)$

$A = Tile\ address + su + [sv << \log_2(bw)]$

where A is translated address
su, sv is sub coordinate within a tile
bw is tile width

Figure 2-3-2-1 4D placement and address translation equation

4D placement [4] is also known as tile-based placement, as shown in figure 2-3-2-1. The concept of 4D placement is row-majored and one level tile-based: original texture is divided into some squared tiles and inter/intra-tile is row-major.

Since texture filtering has spatial locality, the placement which place the texels in a form of group could get better cache performance. This is because the required texels of a filtering may be fall into a 4D tile and they are placed in the texture memory nearby according to the tile size.

However, since inter-tile is also the row-major, the required four texels of bilinear filtering will have strongly chances to cross two adjacent tiles in the column or four different tiles, shown in figure 2-3-2-2. Thus these required texels may be placed separately in texture memory and may introduced conflict miss in direct mapping cache. In [4], they say when the size which is texture width multiplies tile width is multiple of cache size and cache line size is multiple of tile size, conflict misses will occur due to the upper and lower tile will have the same cache index number. By padding the unused tile to form another new column, the problem can be solved. However, texture memory spaces will waste.

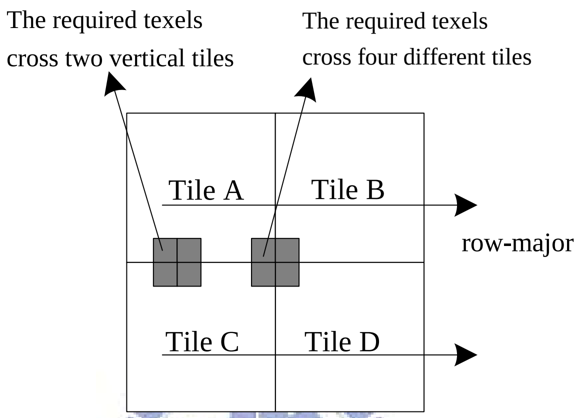

The required texels cross two vertical tiles
The required texels cross four different tiles
Tile A
Tile B
row-major
Tile C
Tile D

Figure 2-3-2-2 access condition of 4D placement

Moreover, if cache line size is equal to the tile size, for those four requited texels in the same cache line, they are two and two continuous or all continuous due to 4*4 tile size. If two texels are in the same cache line, they are continuous like Nonblock placement or discontinuous due to the two texels are placed on different rows in the tile.

The address translation of 4D placement proposed in [4] invokes many arithmetic operations, such as ADD operation. Due to texture address is 32-bits or 64-bits [4] in current GPU architecture, the ADD operation may have long carry propagation according to the hardware implementation. Thus, the propagation could be the critical path of the address translation.

2.3.3 6D placement

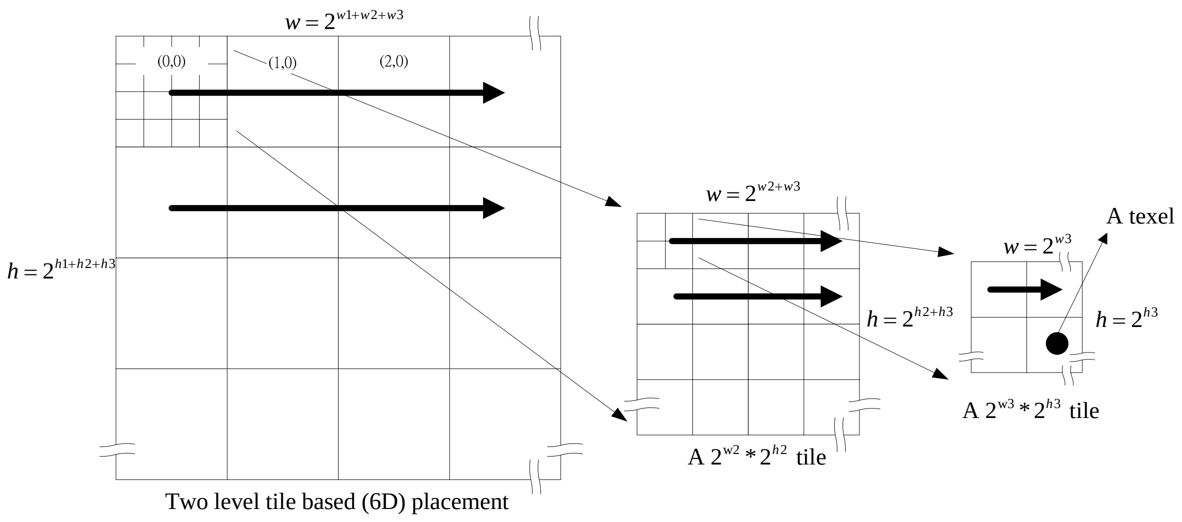


Figure 2-3-3-1 the related work of 6D placement

6D placement [4] is known as two-level tile-based placement, as shown in figure 2-3-3-1. The original texture is divided into some squared larger tiles and inter-larger tile is row majored. Within a larger tile, 4D placement is applied to it.

The placement is proposed to improve the conflict miss which occurs in 4D placement. Unlike the padding unused tiles to form a new column, 6D placement will not waste the memory space. However, the address translation idea proposed in [4] is still following the concept of 4D placement. It invokes arithmetic operations, such as ADD operation.

Finally, we have a table 2-1 to summarize the three placement algorithms in term of address translation time, cache hit rate, average cache access counts based on cache lines and average cache access counts based on cache lines and continuousness. We expect the address translation time of Nonblock is better than 6D placement. The cache hit rate of 6D is better than Nonblock placement. Average cache access counts based on cache lines of 6D/4D placement is better than Nonblock placement. Finally, average time of cache access based on cache lines and continuousness of 6D/4D placement is better than Nonblock placement.

	Nonblock	4D	6D
Placement concept	Row/column-major	One level tile based + row/column major	Two level tile based + row/column major
Address translation concept	Base + offset	Base + level1 tile offset + offset within a l1 tile (*)	Base + level 2 tile offset + level1 tile offset + offset within a l1 tile (*)
Address translation time	better	medium	worse
Cache performance, hit rate (**)	98.893/99.018 (%)	99.3139/98.7078 (%)	99.558/99.728 (%)
Average cache access counts based on cache lines (**)	2.194/2.099	2.078/1.909	2.078/1.909
Average cache access counts based on cache lines and continuousness (***)	2.194/2.107	2.408/2.408	2.408/2.408
(*) Waste memory space when texture height is smaller than tile width. (**) Direct mapping, 8K, cache line size is 32/64 bytes. (***) Direct, 8K, line size is 32/64 bytes, burst mode support with max data length is 16 bytes			

Table 2-3-1 summary of three placement algorithms

Chapter 3 Design

3.1 Design Overview

Our design can be roughly divided into two topics: the first one is focus on the new placement algorithm that is how to place the texture in the texture memory. The second one is focus on the possible texture cache supports in the GPU.

In the placement topic, motivated by the related work proposed in [4], we will propose the new texture placement algorithm by using the recursive concept. The new placement is called Recursive Z placement, and can be viewed as multi-level row-major placement which is extended from 4D/6D placement. Later on, we will try to further improve the RZ placement in term of the continuousness of required texels with in a cache line. We have two main ideas. The first one is motivated from shape. We can try another shape instead of Z shape. The other is motivated from tile size. We can try larger base tile size instead of 2*2 to gain more continuousness.

After we have the placement, we should develop the address translation idea of these placements. The idea should be easy. And the logic should also easy to implement. It may use bit-wise logic operations to accomplish the translation.

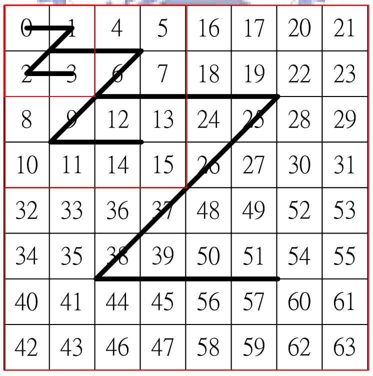
In the possible texture cache support topic, we list three possible texture cache supports in current GPU architecture which are baseline texture cache support, texture cache support1 and texture cache support2. The baseline texture cache support and texture cache support 1 are common in current texture cache. And our design is focus on texture cache support 2 which can retrieve the required texels of bilinear filtering in the same cache line.

3.2 Texture placement

In the section, we will design the new placement algorithm. That is how to place the texture in the texture memory. The new placement algorithm will be design in three aspects which are cache hit rate, average cache access counts and address translation time. Finally, we will propose a possible logic implementation for the address translation idea.

3.2.1 Recursive Z placement (RZ)

Our new placement is called Recursive Z placement. The placement strategy is placing the texel in the recursive z scan line, as shown in figure 3-2-1-1. In the term of iteration, we have the base case (1*1) which only invokes one texel. The next case (2*2) is iteratively integrated with the four previous cases by using Z shape placement Recursive Z can also be viewed as multi-level row-major placement which is extended from 4D/6D placement proposed in [4].



0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

Figure 3-2-1-1 recursive z placement

Since the required texels of bilinear filtering has spatial locality, that is bilinear filtering itself is a form of 2*2 region and the required texels of current bilinear filtering is close to the next one, tile-based placement can avoid placing texels continuously along one u/v direction, i.e. row-major/column-major, like Nonblock placement. Thus, the required texels of filtering may not be separated far away.

RZ placement can also avoid row-major of inter-tile like 4D/6D placement. In figure

3-2-1-2, the required texels of filtering can be cross two/four tiles/Z in RZ/4D/6D. If we have four required texels, say A, RZ can place them more closely than inter-tile is row-major (4D/6D). B, C is the same, too. However, if we have D/E/F, RZ may be worse than 4D/6D placement. But, as mention before, filtering have spatial locality. We expect RZ placement have better cache performance in average.

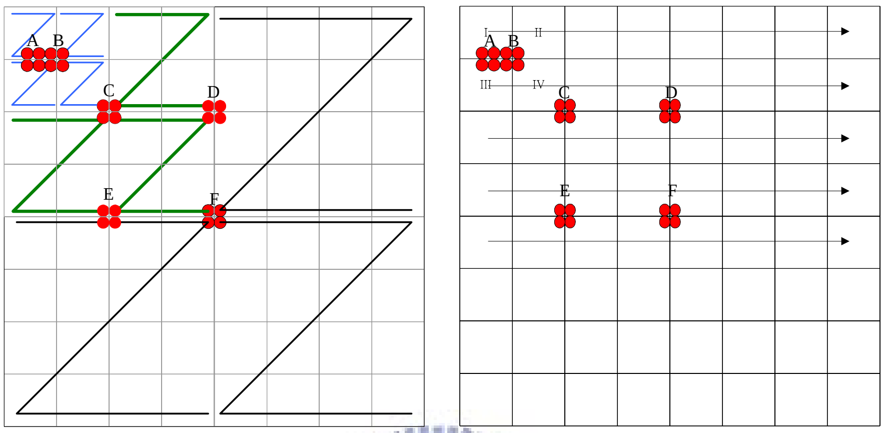


Figure 3-2-1-2 cross tile condition in RZ and 4D/6D

RZ placement can also improve average cache access counts compare to the 4D/6D placement. For a given cache line size, RZ placement can fit those texels which are in that cache line size into the square-liked region. But, row-major of inter-tile may fit them into the rectangular-liked region. In figure 3-2-1-3, if we have cache line size is cable of 4 tiles/Z, we will not cross another cache line when access A or B texels in RZ placement. But, it may will in 4D/6D placement.

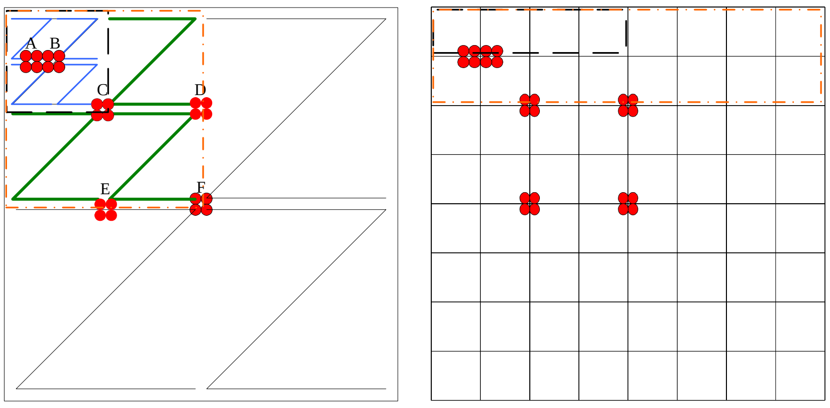


Figure 3-2-1-3 multiple cache lines condition in RZ and 4D/6D

3.2.2 Different placement policies

Under some texture cache system with burst mode technology support, texture cache support1, it may retrieve the required texels of bilinear filtering in one cache access if they are all continuous. It can be done by sending the start address of the required texels and the data offset for the required texels in continuousness. If they are discontinuous, the cache may not retrieve them in one cache access. Thus, the consideration of continuousness is also important.

3.2.2.1 Recursive Z with U (RZU)

Although the required four texels within a base 2*2 Z-shape and 2*2 u-shape are all continuousness, if the required texels are crossing two z or two u in horizontal/vertical, U-shape could be potentially have more benefits than z-shape. This is because the U shape has three directions of continuousness benefits. But the z-shape only has two. Thus, we may change the z-shape to the u-shape.

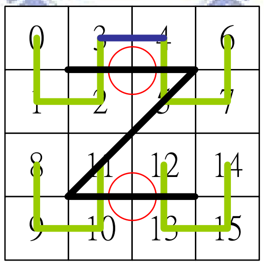


Figure 3-2-2-1-1 Recursive Z with U placement

RZU placement is obtained by changing the 2*2 Z-shape to 2*2 u-shape as shown in figure 3-2-2-1-1. And the placement policy between 2*2 tile is the same as RZ placement. It is also the multi-level tile-based like RZ placement. Thus, we expect the cache hit rate is equal to RZ placement under three kinds of cache support.

By changing z-shape to u-shape, we may have the required four texels of bilinear filtering as shown the red circle in figure 3-2-2-1-1 continuous, but may texels 1/2/8/11 discontinuous. But, in average, we may improve the average cache access counts under texture cache support 1. However, the average cache access counts may equal to RZ in texture cache support 2.

3.2.2.2 Recursive Z with Flipped-U (RZFU)

We can further improve the RZU by flipping the lower U over in order to have the bottom of the upper and lower U edge to edge, as shown as red circle in figure 3-2-2-2-1. However, we may have texels covered by blue circle discontinuous as shown in figure 3-2-2-2-1. And we can also try to flip the upper two U over as shown in figure 3-2-2-2-2. However, by doing this, we may have some required texels discontinuous.

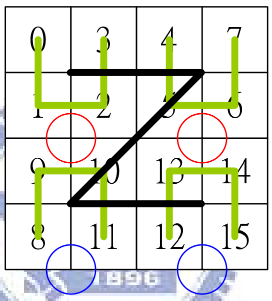


Figure 3-2-2-2-1 Recursive Z with Flipped-U v1

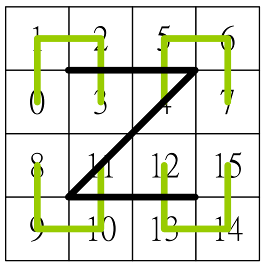


Figure 3-2-2-2-2 Recursive Z with Flipped-U v2

RZFU1/2 placement can also be viewed as the multi-level tile-based like RZ placement. Thus, we expect the cache hit rate is equal to RZ placement under three kinds of cache support. Whether the average cache access counts under texture cache support 1 of RZFU1 is better than RZFU2 may dependent on the probability. If the required four texels are always happen to red circle in figure 3-2-2-2-1, the RZFU1 could be better. If the required four texels are always happen to blue circle in figure 3-2-2-2-2, the RZFU2 could be better. The average cache access counts may equal to RZ in texture cache support 2.

3.2.2.3 Recursive Z with Snake (RZS)

In section of 3-2-2-1 and 3-2-2-2, we improve the RZ placement by changing z-shape to u-shape. In this section, we improve the RZ placement by changing base 2*2 tile size to larger n*n tile size. We found that the larger tile size we choose, the probability of required four texels of bilinear filtering crossing two/four tiles is lower. If the required four texels of bilinear filtering cross two/four tiles, the discontinuous may occur. Another reason for larger tile size is that we have more placement policy within the larger tile size.

We propose a new placement, called Recursive Z with snake. The snaked-tile can be viewed as row-major instead the direction of odd row and we take 4*4 snaked tile size as example shown in figure 3-2-2-3-1. And the placement policy between 4*4 snaked-tile is also the same as RZ placement.

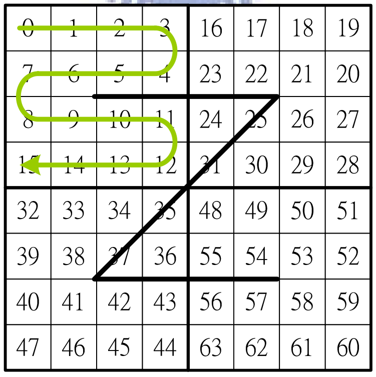


Figure 3-2-2-3-1 Recursive Z with Snake placement

However, we can not increase our base tile size unlimited. The larger snaked tile size we have, the placement within that tile is more like Nonblock placement. The spatial locality of required four texels of bilinear filtering may decrease. Thus, it may affect cache hit rate.

3.2.3 Address translation idea of RZ

The address translation can be viewed as a translation function with inputs, m,n,U,V,B and generates the output, A, which are dimension of texture's width and texture's height, u coordinate, v coordinate, base address of the texture and the translated address as defined in figure 3-2-2-1. So, we are going to find a RZ function which is $A = RZ(m,n,U,V,B)$

Texture's width is $Wn = 2^m \le 2^d, m,d \in N$
Texture's height is $Hn = 2^n \le 2^d, n \in N$
The u-coordinate is U, $0 \le U \le Wn$
The v-coordinate is V, $0 \le V \le Hn$
The translated address is $A \le 2^s$
The base address of texture is $B \le 2^s$

Figure 3-2-2-1 definition of terms

There are three cases in RZ, which are m equal to n, m smaller than n, and m larger than n, respectively. However, the main concept of these cases is the same, that is recursive translation.

The first case is m equal to n, that is texture's width is equal to texture's height. As shown in Figure 3-2-2-2, the base case only invokes one texel, and the translated address A is 0.

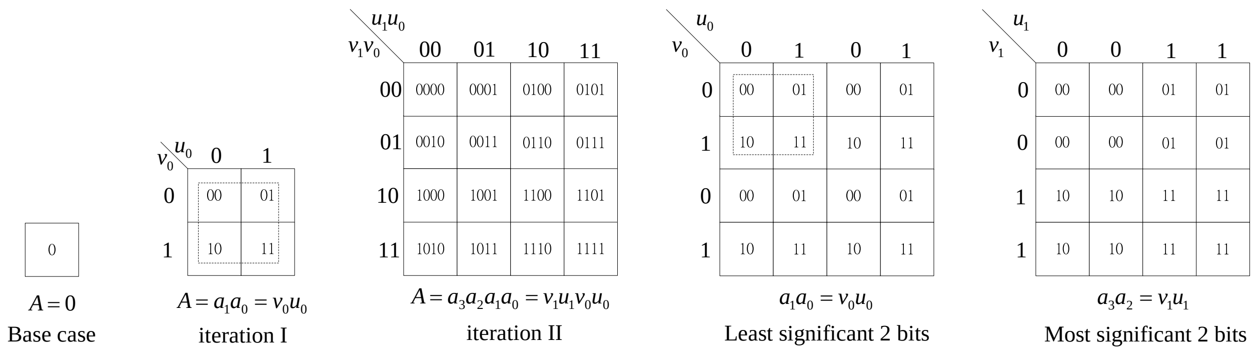


Figure 3-2-2-2 the case I of address translation

In iteration I, the translated address, a_1a_0, could be found by using Karnaugh Map. Thus

$a_1 a_0$ could be $v_0 u_0$, which u_0 and v_0 are the least significant bit of U and V, respectively.

In iteration II, we first focus on the least significant 2 bits of each translated address. And we found that each of the bit pattern in dotted rectangle is corresponding to the bit patterns found in the iteration I. Thus, we can suggest that the least significant 2 bits of translated address in iteration II may equal the bit pattern in iteration I, that is $v_0 u_0$.

We now look at the most significant 2 bits of each translated address. And we can also use Karnaugh Map to translate $a_3 a_2$. The result shows that $a_3 a_2 = v_1 u_1$. So the translated address of iteration II, $a_3 a_2 a_1 a_0$ is $v_1 u_1 v_0 u_0$.The bit pattern can be view as the form which is iteratively cross interleaving each u and v coordinate bit, respectively. In the term of recursive concept, the translated address bit pattern of base case is the subset in iteration I. And the translated address bit pattern of iteration I is also the subset in the iteration II, iteration II is the subset in iteration III, etc.

Now, we can suppose that when the texture is 8 by 8, the translated address $a_5 a_4 a_3 a_2 a_1 a_0$ is $v_2 u_2 v_1 u_1 v_0 u_0$ by cross interleaving each least significant 3 bits of u and v coordinate.

Example: $Rz(3,3,4,7,B)$	$U =$		1		0		0	$= 4$
$m = n = 3$	$V =$	1		1		1		$= 7$
$U = 4 = 100_2$	$A' =$	1	1	1	0	1	0	
$V = 7 = 111_2$	$A = (A' << 2) + B$							

Figure 3-2-2-3 example of address translation case I.

For example, since we are going to translate the pair of (4, 7), all we need to do is cross interleaving each least significant 3 bits of u and v coordinate, respectively. In figure 3-2-2-3, the result of cross interleaving is 58.

However, texture filtering may sample the texture with n by m dimension which is not equal , but is power of 2, respectively. The translation idea mentioned before may need to modify slightly. In figure 3-2-2-4, the texture's width is larger than texture's height. This is $m > n$. In the case, since texture's height is shorter than texture's width, for any texel, we do not have enough v-coordinate bits to cross interleave with u-coordinate bits. On the other words, after perform cross interleaving, some u-coordinate bits are left. These left bits should be followed by the cross interleaved result, in order to obtain the correct translated address.

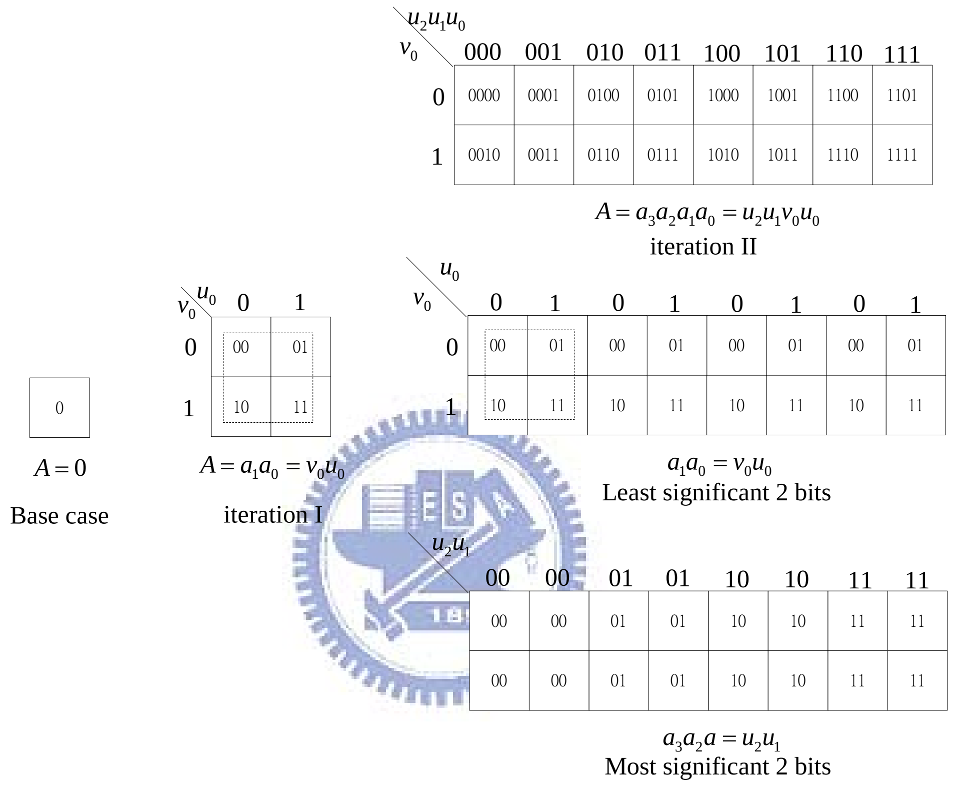


Figure 3-2-2-4 the case II of address translation

In figure 3-2-2-4, we have 3 u-coordinate bits and 1 v-coordinate bit to cross interleave for translated address. After cross interleaving, we have a part of translated address a_1a_0 equal to v_0u_0. However, the translated address should have four bits to index the required texel. The part of address, a_3a_2, we do not assign them yet. So, we focus on the most significant 2 bits of the translated address. The bit pattern of each texel is exactly the same as the unused 2 u-coordinate bits, u_2u_1.

Example: $Rz(4,2,9,3,B)$

$m > n$

$U = 9 = 1001_2$

$V = 3 = 0011_2$

$$\begin{array}{lcccccccl} U = & 1 & 0 & & 0 & & 1 & & = 9 \\ V = & & & 1 & & 1 & & & = 3 \\ A' = & 1 & 0 & 1 & 0 & 1 & 1 & & \end{array}$$

$A = (A' << 2) + B$

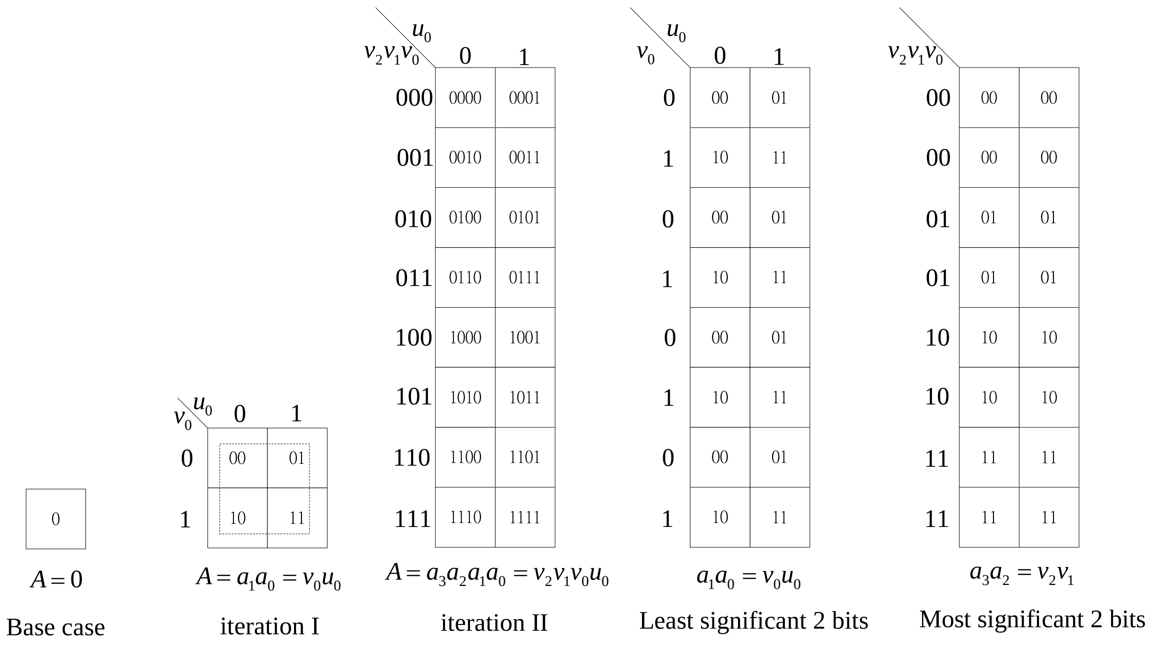
Figure 3-2-2-5 example of address translation case II

For example, since we are going to translate the pair of (9, 3) and m is larger than n, we should need to cross interleave 2 bits of u and v, respectively. And the left 2 bits, u_3u_2, should be followed by the cross interleaved result. In figure 3-2-2-5, the result of cross interleaving is 43.

The conclusion is that, when the texture's dimension is not equal, the cross interleaving is still work, but the remaining coordinate bits should be followed by the cross interleaved result. In this case, the two bits should be followed by the cross interleaved result in order to obtain the correct translated address.

Moreover, in terms of recursive concept, the inequality of two dimensions means incompletely recursive texture. The iteration of placement will break when the short side is met. So, the recursive bit pattern will be limited when the coordinate bits of shorter side is exhausted.

The final case is shown in figure 3-2-2-6. That is texture's width is smaller than texture's height. After we cross interleave them, the left v-coordinate bits, v_2v_1, should be followed by the cross interleaved result, say v_0u_0, in order to obtain the correct translated address.



Base case: $A = 0$

0

iteration I: $A = a_1a_0 = v_0u_0$

v_0 \ u_0	0	1
0	00	01
1	10	11

iteration II: $A = a_3a_2a_1a_0 = v_2v_1v_0u_0$

$v_2v_1v_0$ \ u_0	0	1
000	0000	0001
001	0010	0011
010	0100	0101
011	0110	0111
100	1000	1001
101	1010	1011
110	1100	1101
111	1110	1111

Least significant 2 bits: $a_1a_0 = v_0u_0$

v_0 \ u_0	0	1
0	00	01
1	10	11
0	00	01
1	10	11
0	00	01
1	10	11
0	00	01
1	10	11

Most significant 2 bits: $a_3a_2 = v_2v_1$

$v_2v_1v_0$		
00	00	00
00	00	00
01	01	01
01	01	01
10	10	10
10	10	10
11	11	11
11	11	11

Figure 3-2-2-6 the case III of address translation

Example: $Rz(2,4,3,9,B)$

$m < n$

$U = 3 = 11_2$

$V = 9 = 1001_2$

$V = 1\ 0\ 0\ \ 1\ \ = 9$

$U = \ \ \ \ 1\ \ 1\ = 3$

$A' = 1\ 0\ 0\ 1\ 1\ 1$

$A = (A' << 2) + B$

Figure 3-2-2-7 example of address translation case III

For example, since we are going to translate the pair of (3, 9) and m is larger than n, we should need to cross interleave 2 bits of u and v, respectively. And the left 2 bits, v_3v_2, should be followed by the cross interleaved result. In figure 3-2-2-7, the result of cross interleaving is 39.

So far as here, the translated address mentioned before is not the final address we are going to use. This is because we did not take base address of the texture as a consideration. So the translated address will be added with base address and be left shifted 2 bits for 4 bytes a texel. Final address is $(A' << 2) + B$

Finally, we have a figure 3-2-2-8 to summarize the address translation idea under three cases.

Address translation of Recursive Z :

case I $(Wn = Hn, m = n = r)$

$A' = v_{r-1}u_{r-1}v_{r-2}u_{r-2}\cdots v_1u_1v_0u_0$

case II $(Wn > Hn, m > n)$

$A' = u_{m-1}u_{m-2}\cdots u_{m+1}u_m v_{n-1}u_{n-1}\cdots v_1u_1v_0u_0$

case III $(Wn < Hn, m < n)$

$A' = v_{n-1}v_{n-2}\cdots v_{n+1}v_n v_{m-1}u_{m-1}\cdots v_1u_1v_0u_0$

Final address $A = (A' << 2) + B$, for one texel is 4 bytes

Thus we have address translation function $A = Rz(m, n, U, V, B)$

Figure 3-2-2-8 summary of RZ address translation function

3.2.4 Address translation of other placements

Since the inter-tile placement policy of RZU, RZFU1/2 and RZS is identical to the Recursive Z placement, the difference of address translation among these placements could be the least significant bits. We summarize a table to list the difference of translated address bit pattern among these placements.

	Recursive Z	Recursive Z with U	Recursive Z with Snake	Recursive Z with Flipped U 1/2
a_3	v_1	v_1	v_1	v_1
a_2	u_1	u_1	v_0	u_1
a_1	v_0	u_0	$v_0 \oplus u_1$	u_0
a_0	u_0	$v_0 \oplus u_0$	$v_0 \oplus u_0$	$u_0 \oplus v_0 \oplus v_1 \,/\, \overline{u_0 \oplus v_0 \oplus v_1}$

Table 3-2-4-1 Summary of least significant four bits of address among placements

3.2.5 Address translation logic implementation

Since texture mapping is a time consuming operation in current graphic processing unit, the new address placement's algorithm should not take too much time to obtain the required address. In [4], the address translation time of Nonblock and 4D and 6D may take long time to translate, since they may invoke arithmetic operations, such add operation, and the propagation path in add operation may up to 32 bits or 64 bits, due to texture address is 32 bits of 64 bits and hardware implementation of Adder. We are going to use bit-wise logic operations to translate the address, Since there is the regularity in the Recursive Z. And we expect that the address translation time is short.

In the pervious section, no matter what translation case it is, the translated address can be viewed as combination of common address field and differential address field, as shown in figure 3-2-5-1. The common field only invokes the bit pattern which is cross interleaved form some bit toggles of U and V coordinate. And the Differential field invokes only U or V coordinates or simply zero. The combination invokes bit-wise OR operations.

Case I ($m = n = r$)

$$\begin{array}{lrl} & \text{differential field :} & 0 \\ \text{Bit-wise OR} & \text{common field :} & v_{r-1}u_{r-1}\cdots v_1u_1v_0u_0 \\ \hline & & v_{r-1}u_{r-1}\cdots v_1u_1v_0u_0 \end{array}$$

Case II ($m > n$)

$$\begin{array}{lrll} & \text{differential field :} & u_{m-1}u_{m-2}\cdots u_{n+1}u_n & 0 \\ \text{Bit-wise OR} & \text{common field :} & 0 & v_{n-1}u_{n-1}\cdots v_1u_1v_0u_0 \\ \hline & & \multicolumn{2}{r}{u_{m-1}u_{m-2}\cdots u_{n+1}u_nv_{n-1}u_{n-1}\cdots v_1u_1v_0u_0} \end{array}$$

Case III ($m < n$)

$$\begin{array}{lrll} & \text{differential field :} & u_{n-1}u_{n-2}\cdots u_{m+1}u_m & 0 \\ \text{Bit-wise OR} & \text{common field :} & 0 & v_{m-1}u_{m-1}\cdots v_1u_1v_0u_0 \\ \hline & & \multicolumn{2}{r}{u_{n-1}u_{n-2}\cdots u_{m+1}u_mv_{m-1}u_{m-1}\cdots v_1u_1v_0u_0} \end{array}$$

Figure 3-2-5-1 concept of address translation unit

The address translation unit can be roughly divided into three parts based on the concept of combination mentioned in figure 3-2-5-1, which are common field generator, differential field generator and additional operations as shown in figure 3-2-5-2.

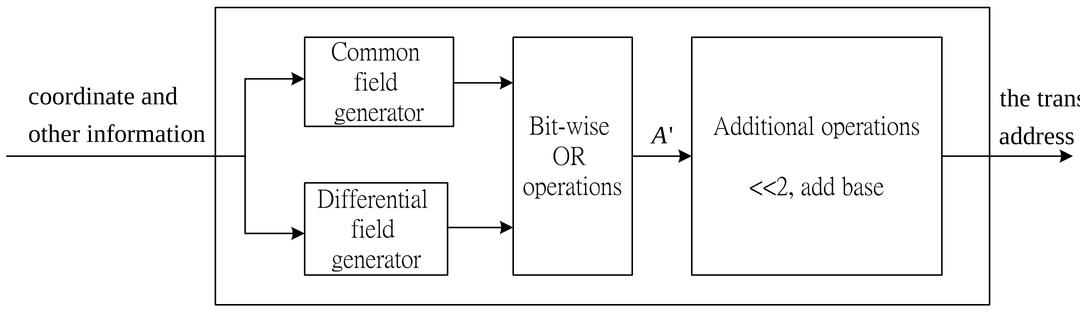


Figure 3-2-5-2 global view of address translation logic

Common field generator is responsible for the generation of the bit pattern by cross interleaving some bit toggles of each coordinate. Differential field generator is responsible for the generation of the bit pattern by concatenating the bit toggles which are not cross interleaved. The obtained two filed are then combined by bit-wise OR operations. Finally, the additional operations are responsible for left shift and add the base address.

In common field generator, what we concern about is how many least significant bit toggles of each U, V coordinate should we cross interleave? Intuitively, by comparing m with n and choosing the smaller one, we have the number of bit toggles that should be cross interleaved of each coordinate. For example, if m=7 and n=3, there are three least significant bit toggles of each coordinate we should cross interleave. As a result, the common field generator could have a compare and select logic as shown in figure 3-2-5-3 and a flexible cross interleaving logic which can perform cross interleaving operation under any number of least significant bit toggles.

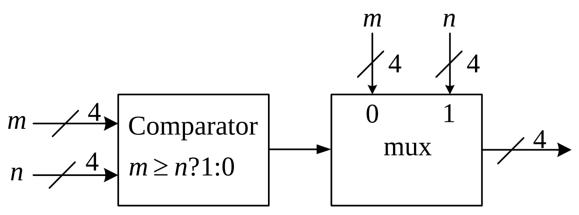


Figure 3-2-5-3 compare and select logic

The compare and select logic is the comparator combined with a 2 to 1 mux. The logic can tell us the smaller one, say m or n. Thus, we know how many bit toggles we should cross interleave. And the result is passed to the flexible cross interleaving logic. It could be implemented in a form of mux. However, the mux can be the critical path in common field generator. How about use the concept of integration by smaller and unique cell to implement the common field generator? If we can design a cell which can cross interleave the two inputs, one bit toggle from U and the other from V, by a control signal, we can concatenate many of them of form our the common field generator.

The control signal for each cell is generated through an encoder which can encode the output from compare and select logic, i.e. if we have the output form compare and select logic, say 3, that is we should cross interleave least significant three bits form U and V, thus the encoder will generate the enable bit pattern, 111_2. Based on the enable bit pattern, we should cross interleave from each significant three bit toggles of U and V, i.e. $u_2 \sim u_0$ and $v_2 \sim v_0$.

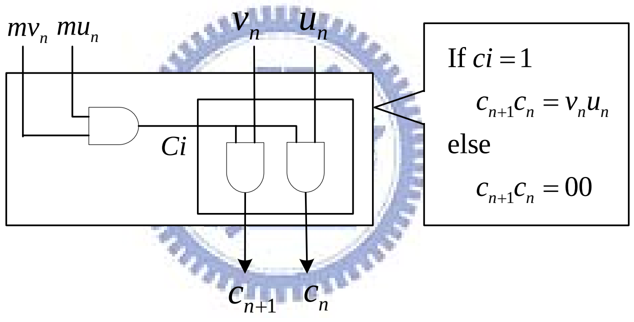


Figure 3-2-5-3 one cell of common field generator

Figure 3-2-5-3 represents the one cell of common field generator. The control signal, Ei, is from the corresponding bit position in enable bit pattern. Figure 3-2-3-4 shows the common field generator which is obtained by integrating n cells.

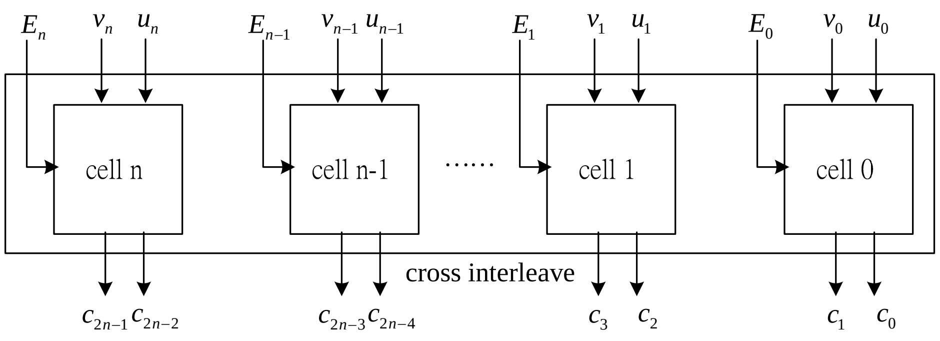


Figure 3-2-5-4 common field generator with n cells

The differential field generator is responsible for generation the differential field of the translated address under any given m and n. For example, if we have m=7 and n=3, the differential field can be obtained by setting bit toggles $u_2u_1u_0$ to zero and left shifting 3 bits, like $\cdots u_5u_4u_3000000$.

So what we concern about is the differential field is from most significant bit toggles of U or V and How many bit toggles will be used and their bit position? Since we have known who the smaller one is(m or n), by using the result, we can select the desired coordinate $(\mathrm{m} \geq \mathrm{n}?\mathrm{U}{:}\mathrm{V})$. And we also have the enable bit pattern generated from the encoder.

Thus, we can use the information to generate the differential field. In figure 3-2-5-5, the comparison result is used to select the desired coordinate (U or V) by using another 2 to 1 mux. After that the selected result, say A and the result from enable encoder, say B, is passed to the Bit filter. It can set the unnecessary bit field of B to zero. Finally, the left shifter can left shift any number of bit of B' based on the smaller one value of m or n .

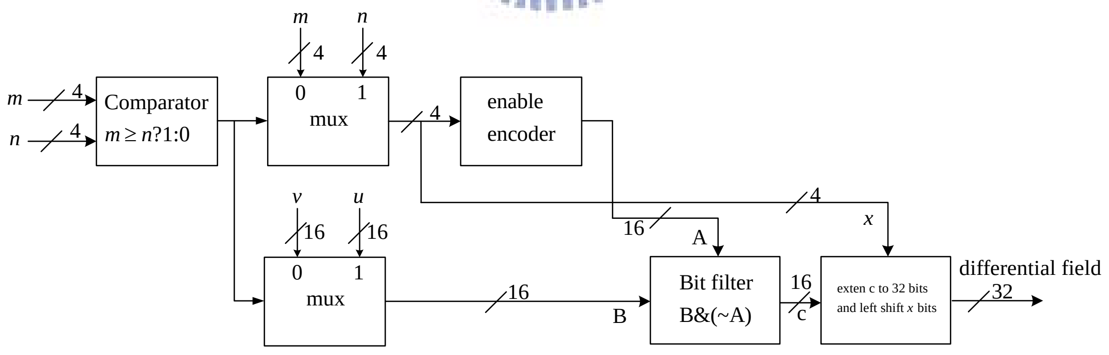


Figure 3-2-5-5 differential field generator

For example, if we still have m=7 and n=3, the input A, B to the bit filter is coordinate U and bit pattern 111_2 form enable encoder. The bit filter will filter out the least three significant bit of coordinate U. the left shift will left shift 3 bits based on the pattern 111_2. The differential field could be like $\cdots u_5u_4u_3000000$.

3.3 Three possible texture cache supports

Although, the average texels access time of bilinear filtering is affected by the texture placement, it is also affected by the hardware design (texture unit/texture cache). We have three possible texture cache supports in different hardware cost. And each of them has different texels retrieval capability.

3.3.1 Baseline texture cache support

The baseline texture cache support is straightforward. The texture cache can retrieve one required texel data with a address request. In this kind of system, only the address translation time and texture cache miss rate will affect the average texels access time of bilinear filtering. This is because every address request can only retrieve one texel data to the texture filter. Thus, average cache access counts of bilinear filtering are always four.

3.3.2 Texture cache support 1

The texture cache support 1 is a common texture cache with burst mode support. The burst mode technique is done by sending a start address and the maximum required data offset; the receiver can get the required data as soon as possible. Since the required texels of bilinear filtering are four, the maximum data offset length is 16 bytes. In the other words, if the required texels are adjacent to each other within a cache line, the cache can retrieve all of the required texels in one cache access. Under the texture cache support, the average cache access counts may affected by whether the required four texels are adjacent to each other.

3.3.3 Texture cache support 2

Since the required texels of bilinear filtering could be potentially in the same cache line, for those texels in the same cache line, we can retrieve them in one cache access no matter whether they are continuous or not. This kind of texture cache support is more flexible than the previous one. Thus, we can retrieve more required texels in one cache access. However, for those texels are not in the current been accessed cache line, we still have another cache access to get them.

3.3.3.1 Possible design of texture cache support2

Base on the concept mentioned in 3.3.3, we need case identifier, texels router and the modified coordinate generator to accomplish the task.(Shown in figure 3-3-3-1-1) Each of them is describe in the following sections.

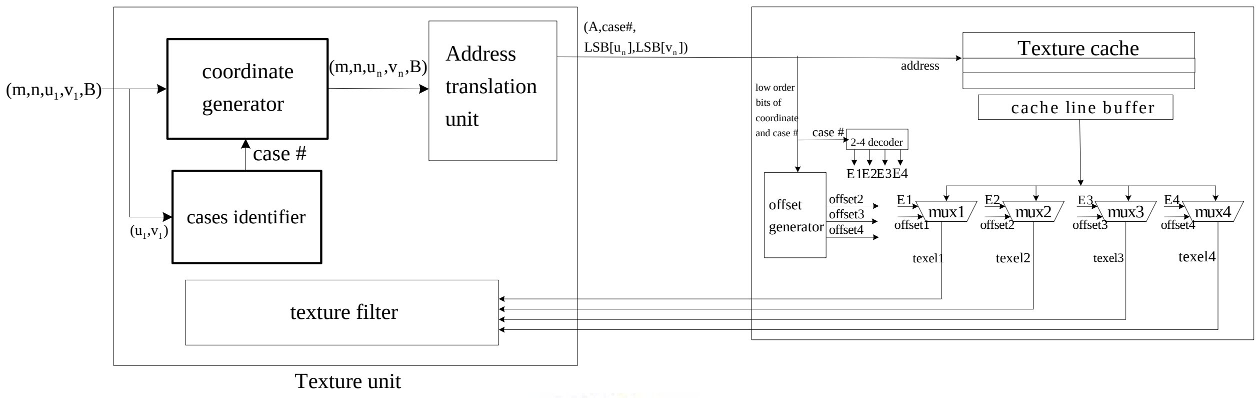
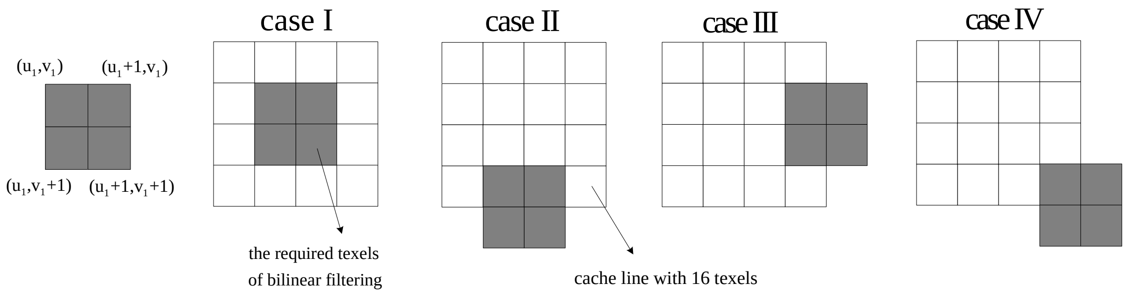


Figure 3-3-3-1-1 Texture cache support 2

3.3.3.1.1 Case identifier

Since the required texels of bilinear filtering is a form of 2 by 2 texels, these four texels can be potentially in one, two, four cache lines. We can identify the case condition through coordinate (u_1, v_1), as shown in figure 3-3-3-1-1. In figure 3-3-3-1-1, w supposes that cache line size is 64bytes (16 texels) and the condition can be roughly classified into 4 types.

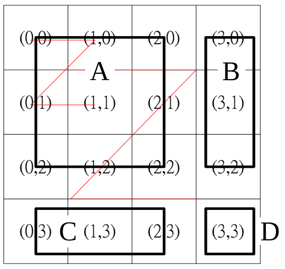
Case I is the required texels are fall into a single cache line. Case II is two of the required texels in the row are fall into a cache line, and the other two are fall into another cache line. Case III is two of the required texels in the column are fall into a cache line, and the other two are fall into another cache line. Case IV is four required texels are in different cache lines.



case I
case II
case III
case IV
(u1,v1)
(u1+1,v1)
(u1,v1+1)
(u1+1,v1+1)
the required texels of bilinear filtering
cache line with 16 texels

Figure 3-3-3-1-1-1 multiple cache lines conditions

Since we know the placement algorithm and cache configuration, i.e. RZ placement, cache line size, the case condition can be obtained though identification of coordinate (u_1, v_1). The identification is easy and straightforward. If we have RZ placement and cache line size is 16 texels, the 16 texels can be shown as the 16 white squares in the figure 3-3-3-1-1-1. We can partition the 16 texels into 4 regions; say A, B, C and D, as shown in figure 3-3-3-1-1-1.



assume cache line size is 64 bytes (16 texels)

```
if $(u_1\%3 = 0$ and $v_1\%3 = 0)$
    case IV
else if $(u_1\%3 = 0$ and $v_1\%3 \neq 0)$
    case III
else if $(u_1\%3 \neq 0$ and $v_1\%3 = 0)$
    case II
else
    case I
```

Figure 3-3-3-1-1-2 operation of case identifier

If the (u_1, v_1) is fall into region A, the other three coordinates will also in the same cache line. Thus it could be case I, all required texels are in the same cache lines. If the (u_1, v_1) is fall into region B, it is the case III, two texels in the column are in the same cache line, the other two texels are in the other cache line. The worst case is (u_1, v_1) fall into region D; all required texels are in different cache lines.

The identification algorithm is shown in the figure 3-3-3-1-1-2. If both u_1 and v_1 mod 3 are equal to 0, region D. If u_1 but not v_1 mod 3 are equal to 0, region B. If not u_1 but v_1 mod 3 are equal to 0, region B. If neither u_1 nor v_1 mod 3 are equal to 0, region A. The mod operation can be implemented through Bit-wise logic AND operation of two lower order bits of coordinate u_1 and v_1, i.e. u_1 mod 3 is equal to 0 can be implemented through the result of AND u_1^1 and u_0^1 is equal to 0. Thus all we need is low order two bits of coordinate u_1 and v_1 to identify the case conditions. Since we have total four cases, we can encode the cases by using 2 bits signal. 00 means case I. 01 means case II. 10 means case III. 11means case IV.

However, the texture dimension can be any magnitude of power of two. That is the texture height/width can be smaller or equal to 2. In these cases, the 16 texels which are in the cache line will not be square-like region any more in the texture. It could be the rectangular with narrow width or wider height dimension. As a result, the identification algorithm in figure 3-3-3-1-1-2 should be modified. The values A, B which is the magnitude u_1 mod A and v_1 mod B in figure 3-3-3-1-1-2 should be changed based on the texture dimension.

In the original, the 16 texels are in the 4*4 rectangular, the magnitude of A should be 3, and B should be 3, as shown in figure 3-3-3-1-1-2. However, if the texels are in a form of 16*1 rectangular, A should be 15, B should be 0. If they are in a form of 1*16 rectangular, A should be 0, A should be 15. If 8*2, A should be 7, B should be 1. If 2*8, A should be 1, A should be 7.

Thus we have the prefix operation which can tell the case identifier what the magnitude of A and B should the identifier use. And there are five cases if cache line size is 64 bytes (16texels). Which are corresponding to 4*4, 8*2, 2*8, 16*1 and 1*16 rectangular. The classification can be done through the m, n which is power of width and height, respectively. By comparing m and n, we know the case and can enable one of the five enable signals. And the enabled case can perform future case identification based on the coordinate (u_1, v_1). The overview of case identifier can be shown in figure 3-3-3-1-1-3.

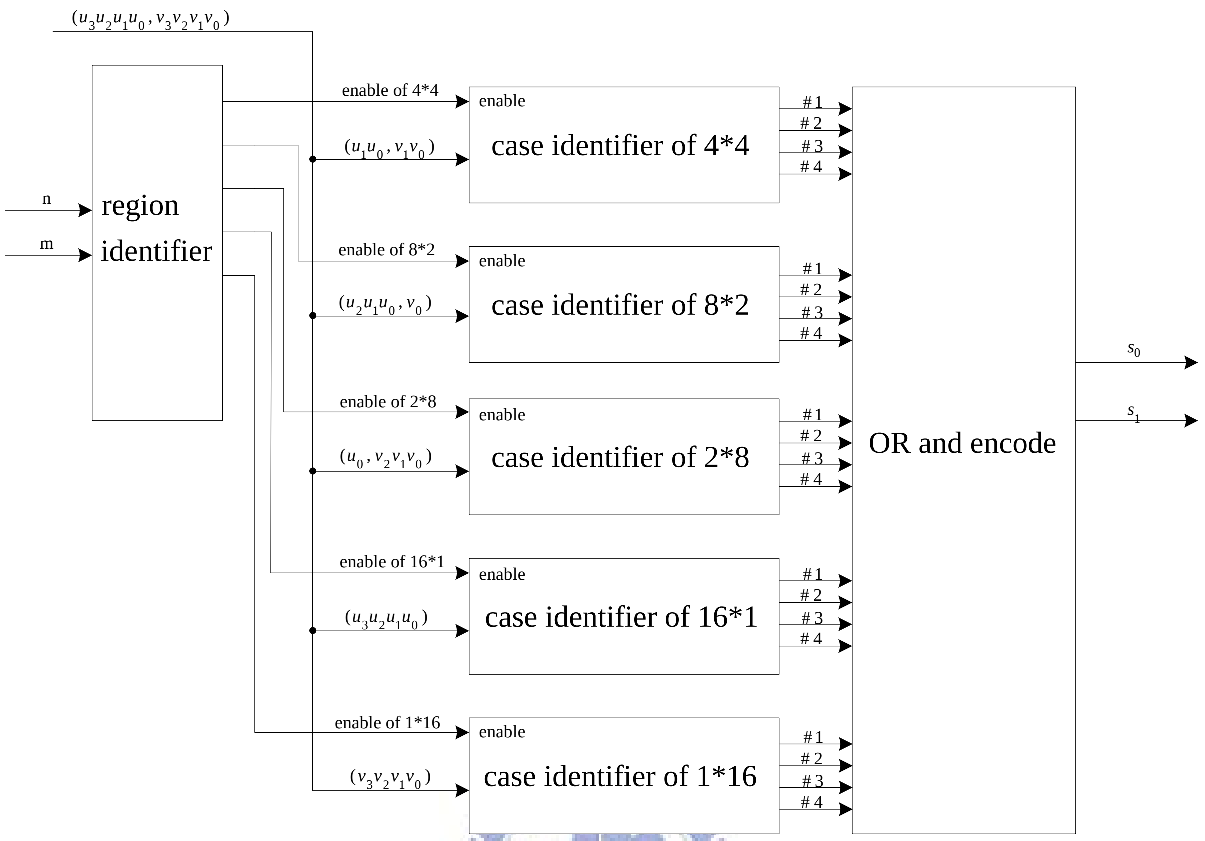


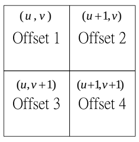
Figure 3-3-3-1-1-3 overview of case identifier

3.3.3.1.2 Coordinate generator

The coordinate generator is responsible for generate the required coordinates based on the filtering types and one of the coordinate, say (u_1, v_1). Since the required texels of filtering can be potentially in the same cache line, we can only generate the coordinate of explicit texels. For those implicit texels, we choose not to generate them, i.e. if we have case I, we only generate coordinate (u_1, v_1) as explicit texel, for the other three texels, say (u_1+1, v_1), (u_1, v_1+1) and (u_1+1, v_1+1), can be viewed as implicit texels and not to generate these coordinates.

We modified the original coordinate generator. As a result, the generator can generate the coordinates based on the case condition obtained from the case identifier. Based on these cases, the generator may generate one, two or four coordinate pairs. The obtained coordinates are then sending to the queue for further address translation.

The algorithm of coordinate generator is shown in figure 3-3-3-1-2-1. In figure 3-3-3-1-2-1, if we have case I, the generator will do nothing but the original coordinate (u_1, v_1). If case II, the generator will generate the other coordinate $(u_1, v_1 + 1)$. If case III, it will generate the other coordinate $(u_1 + 1, v_1)$. If case IV, it will generate the other three coordinates $(u_1 + 1, v_1)$, $(u_1, v_1 + 1)$ and $(u_1 + 1, v_1 + 1)$.



```
if (caseI)
   need coordinate (u1,v1)
else if (case II)
   need coordinate (u1,v1)
                   (u1,v1+1)
else if (case III)
   need coordinate (u1,v1)
                   (u1+1,v1)
else //case IV
   need coordinate (u1,v1)
                   (u1,v1+1)
                   (u1+1,v1)
                   (u1+1,v1+1)
```

Figure 3-3-3-1-2-1 operation of coordinate generator

3.3.3.1.3 Texels router

In order to retrieve those implicit texels, we need to generate extra information to notify the cache to retrieve them back to the texture filter in one time of cache access. Thus we have texels selector and offset generator to accomplish the task, as shown in figure 3-3-3-1-3-1. For those implicit texels, offset generator will generate the corresponding offset field of those texels in the same cache line. These offsets information will be sent to the texels selector for further selection.

Texels selector is worked as four independent muxs, i.e. mux1, mux2, mux3, mux4, as shown in figure 3-3-3-1-3-1. Each of them is responsible for selecting the desired texel from the cache line buffer based on the offset field and enable signal. The offset fields are generated from the offset generator which is based on the case condition and low order bits of coordinate of the explicit texel. And the enable signal of each mux is from enable generator. The line buffer size is based on the cache line size and the input of the mux is line buffer size divided by four for one texel is four bytes. Thus the delay of texels selector is dependent on the mux. We show the Mux delay in Appendix A.1

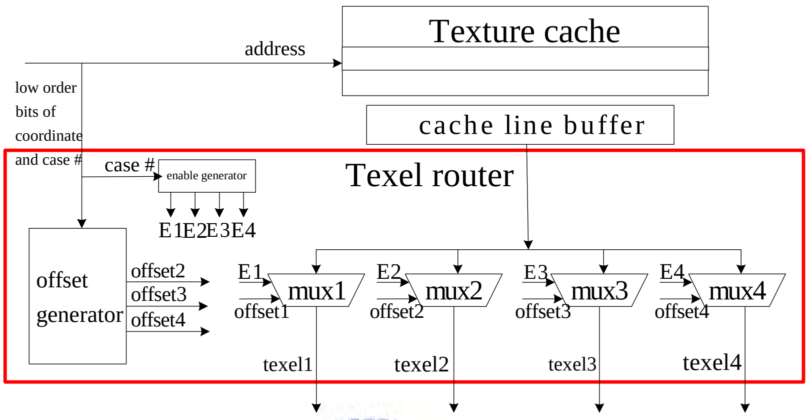
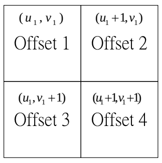


Figure 3-3-3-1-3-1 texels router

The offset generator is responsible for extra information. This information is used to notify the texels selector to select the implicit texels. The input to the offset generator is case number, low order bits of coordinate of explicit texels and region case number. For example, if we have case one with coordinate (1,1) of explicit texel, the offset generator will generate the other three offsets, 6, 9, 12 which are correspond to coordinates (1,2), (2,1) and (2,2).

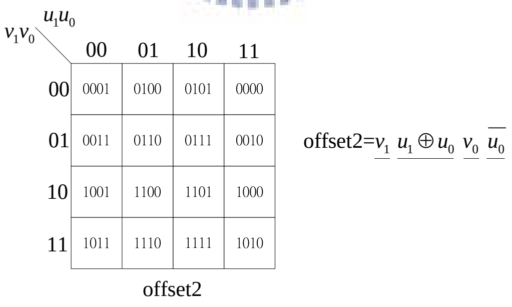
As shown in figure 3-3-3-1-3-2, if case I, we will generate the offset2, offset3 and offset4 which is sending to the mux2, mux3 and mux4. And the offset1 is implicit in the texture address. If case II, we will generate offset2. if case III, we will generate offset3. if case IV, the output of offset generator are don’t care.



```
if (case I)
    generate  offset2, offset3, offset4
else if (case II)
    generate  offset2
else if (case III)
    generate  offset3
else // case IV
    do nothing
```

Figure 3-3-3-1-3-2 operation of offset generator

The Boolean equation of offset2/3/4 can be obtained by using Karnaugh Map. As shown in figure 3-3-3-1-3-3, offset2 can be obtained through low order bits of coordinate of explicit texels, say (u_1u_0, v_1v_0) under cache line size is 16 texels and fitted in 4*4 square-liked region. If we have explicit the texel of coordinate $(0,0)$, the offset2 should be the 1, If we have coordinate $(0,1)$, the offset2 should be the 3. If we have coordinate $(1,1)$, the offse2 should be 6. After we have enumerated all the cases from $(0,0)$ to $(3,3)$, a Karnaugh Map can be obtained like in figure 3-3-3-1-3-3. Thus, we have the Boolean equation of offset2 shown in the figure.



v_1v_0 \ u_1u_0	00	01	10	11
00	0001	0100	0101	0000
01	0011	0110	0111	0010
10	1001	1100	1101	1000
11	1011	1110	1111	1010

offset2

offset2=$\underline{v_1}\ \underline{u_1 \oplus u_0}\ \underline{v_0}\ \underline{\overline{u_0}}$

Figure 3-3-3-1-3-3 Boolean equation of offset2

However, the 16 texels which are in the same cache line may be fitted into the 8*2/2*8/16*1/1*16 rectangular-liked region due to the dimension of the texture. We can use the same methodology to obtain the Boolean equations for them. We summarize a table to enumerate the Boolean equations in table 3-3-3-1-3-1.

The enable generator is responsible for enable signal of column mux selector. Inputs are case conditions and outputs are enable signals to the corresponding mux. As shown in figure 3-3-3-1-3-4, if case I, enable 1 is set. If it is case II, enable 2/3 is set. If it is case III, enable 1/3 is set. If it is case IV, enable 1/2/3/4 is set.

	4*4 region	8*2 region	2*8 region
offset2	$\underline{v_1}\ \underline{u_1 \oplus u_0}\ \underline{v_0}\ \underline{\overline{u_0}}$	$\underline{u_1(u_2 \oplus u_0)+\overline{u_1}u_2}\ \underline{u_1 \oplus u_0}\ \underline{v_0}\ \underline{\overline{u_0}}$	$\underline{v_2}\ \underline{v_1}\ \underline{v_0}\ \underline{\overline{u_0}}$
offset3	$\underline{v_1 \oplus v_0}\ \underline{u_1}\ \underline{\overline{v_0}}\ \underline{u_0}$	$\underline{u_2}\ \underline{u_1}\underline{\overline{v_0}}\ \underline{u_0}$	$\underline{v_1(v_2 \oplus v_0)+\overline{v_1}v_2}\ \underline{v_1 \oplus v_0}\ \underline{\overline{v_0}}\ \underline{u_0}$
offset4	$\underline{v_1 \oplus v_0}\ \underline{u_1 \oplus u_0}\ \underline{\overline{v_0}}\ \underline{\overline{u_0}}$	$\underline{u_1(u_2 \oplus u_0)+\overline{u_1}u_2}\ \underline{u_1 \oplus u_0}\ \underline{\overline{v_0}}\ \underline{\overline{u_0}}$	$\underline{v_1(v_2 \oplus v_0)+\overline{v_1}v_2}\ \underline{v_1 \oplus v_0}\ \underline{\overline{v_0}}\ \underline{\overline{u_0}}$

	16*1 region	1*16 region
offset2	$\underline{u_3(\overline{u_2}+\overline{u_1}+\overline{u_0})+\overline{u_3}u_2u_1u_0}$ $\underline{u_2(\overline{u_1}+\overline{u_0})+\overline{u_2}(u_0u_1)}\ \underline{u_1 \oplus u_0}\ \underline{\overline{u_0}}$	--
offset3	--	$\underline{v_3(\overline{v_2}+\overline{v_1}+\overline{v_0})+\overline{v_3}v_2v_1v_0}$ $\underline{v_2(\overline{v_1}+\overline{v_0})+\overline{v_2}(v_0v_1)}\ \underline{v_1 \oplus v_0}\ \underline{\overline{v_0}}$
offset4		--

Table 3-3-3-1-3-1Boolean equation of offset field

case label	encode s_1s_0	E_1	E_2	E_3	E_4
I	11	1	1	1	1
II	00	1	1	0	0
III	01	1	0	1	0
IV	10	1	0	0	0

$$E_1 = 1$$

$$E_2 = S_1$$

$$E_3 = S_0$$

$$E_4 = S_1S_0$$

Figure 3-3-3-1-3-4 Boolean equation of enable signal

Chapter 4 Experiment and Results

4.1 experiment goal, environment and methodology

We are going to know the average texels access time of bilinear filtering under three different possible texture cache supports mentioned in section 3.3. That is how many performance improvements we have under three different kinds of texture cache support.

We trace the texture coordinate pattern form the Alila simulator which is proposed in [10]. The simulator architecture is based on the design of ATI GPU's architecture and support OpenGL based benchmarks, i.e. Doom3[19], Quake4[20], the 3-D based computer games. The texture coordinate pattern is recorded in the file when the Atila is rendering frames of the Doom3/Quake4. The screen resolution we have could be 640*480/1240*1028/1600*1200 pixels.

After we have the trace, we also implement the L1 texture cache and the pipelined address translation unit which are referenced from ATI GPU architecture environment [10]. The input to the simulator we implement is the trace we obtain mentioned before. Thus, we can obtain the cache hit rate, average cache access counts and average texels access time of bilinear filtering under three different kinds of texture cache support.

The configuration of L1 texture cache is referenced from research [7]. In [7], they say direct mapping and 8K texture cache is sufficient to cache the required texels of bilinear filtering. If the cache misses, the system will stall and we use a linear equation to describe the miss penalty: Miss penalty = constant + (cache line size) / (bus width between texture cache and texture memory) * (cycle/per byte). And we have the constant is 100 cycles and bus width between texture cache and memory is 8 bytes which is a common configuration in the current GPU architecture.

For the address translation unit design of related work and RZ-based placements, we use Verilog [17] to describe the equation proposed in [4] and designs in section 3-2. And we use Max Plus II [16] to perform functional verification. Currently, most desktop graphic cards' texture size does not exceeded in 4096 * 4096 texels. Thus, we select 16 bits for texture dimension and coordinate. And texture address is 32 bits for most GPU architecture.

Moreover; In order to obtain the address translation time, we synthesize the address translation unit by Design compiler [18] and choose the TSMC 130nm technology as the parameter, since it is a reliable technology for many years and there are many consumer products of ATI and Nvidia using the technology. The clock rate using 130nm die processing technology can up to 400 MHz, as shown in appendix A.2. Thus, the cycle time could be 2.5 ns.

Finally, we assume that we can divide the address translation unit into stages perfectly according to the address translation time divided by cycle time and the address queue size between address translation unit and L1 texture cache is infinity.

4.2 Experiment results

In section 4.2.1, we obtain the time result of address translation of different placement algorithms, which include the Nonblock/4D/6D/RZ-based placements. In section 4.3.2, we show the result of the cache miss rate and average texels access time of bilinear filtering under baseline texture cache. In section 4.3.3 and 4.3.4, we show the result of the cache miss rate, average cache access counts and average texels access time of bilinear filtering under texture cache support1/2.

4.2.1 Results of address translation time

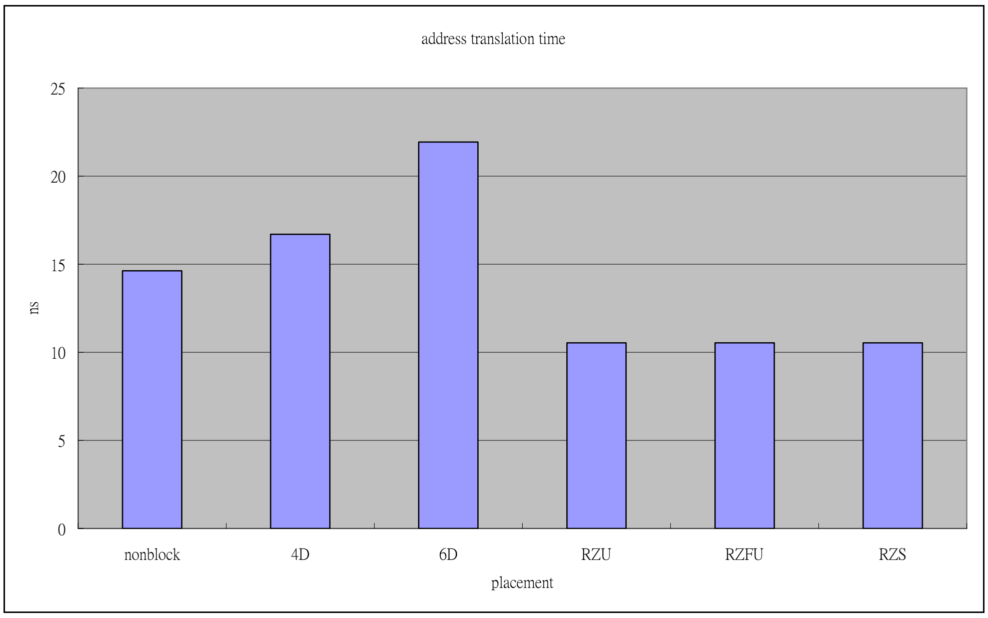


Figure 4-2-1-1 address translation time of different placements

As shown in figure 4-2-1-1, the address translation time of RZ-based placement is better than address translation concept proposed in [4]. This is because their address translation concept is the summation of multi-level offsets instead of bit-wise logic operations, i.e. Bit-wise ADD, OR or Shifter.

The difference of translated address bit pattern between RZU/RZFU and RZ or RZS and RZ placement could be only least significant two or three or four bits. Although these difference could be complicated than RZ placement, the address translation time spend on them can be hided by the critical path which is the time spend on generation of differential field address. Thus, the address translation time of RZU, RZFU and RZS is equal to RZ placement.

4.2.2 Results under baseline texture cache

In [4], they indicates that for a given cache line size, the lowest miss rate is happen to the placement algorithm which tile size is most fit the line size, i.e. tile size is 4 by 4 texels under cache line size is 64 bytes, and they also mention that the level one tile size of 6D placement should fit the cache size. Thus, we have 4D4 and 6D32_4 placement as the configuration of related works. 4D4 means the 4D placement with tile size is 4 by 4 texels. 6D32_4 placement means level one tile size is 32 by 32 texels and level two tile size is 4 by 4 texels for the cache configuration.

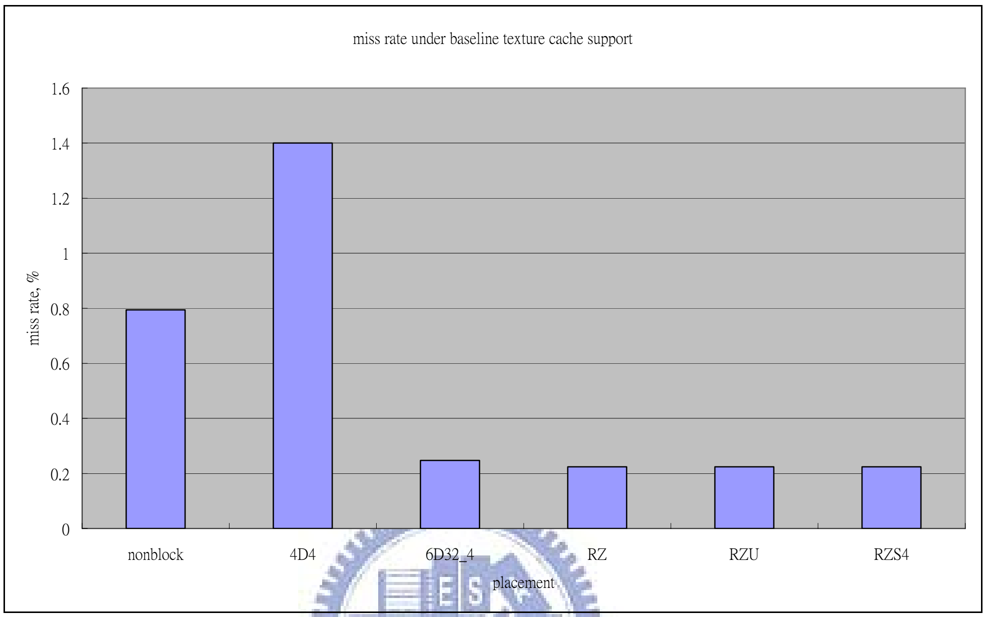


Figure 4-2-2-1 miss rate in baseline texture cache

In figure 4-2-2-1, the miss rate of 4D4 placement may even worse than Nonblock placement. This is because when the size which texture width multiply the tile width is multiple of cache size and cache line size is multiple of tile size and the required four texels of bilinear filtering are crossing over two adjacent vertical tiles or different four tiles as shown in figure 4-2-2-2, 4D placement will have serious conflict misses. However, 6D and Recursive-based placement can eliminate it

Figure 4-2-2-1 shows that the miss rate of RZ-based placement is improved ~0.02% compare to 6D32_4 placement, ~1.18% compare to 4D4 placement and ~0.57%compare to Nonblock placement in the baseline texture cache support.

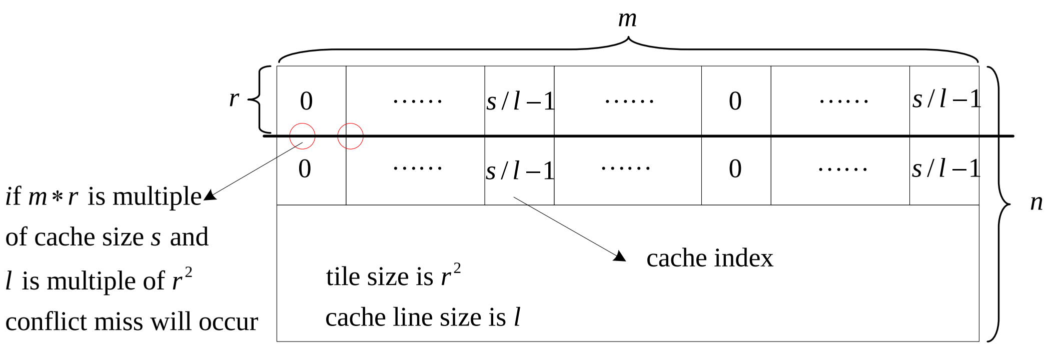


Figure 4-2-2-2 conflict miss under direct mapping with 4D placement

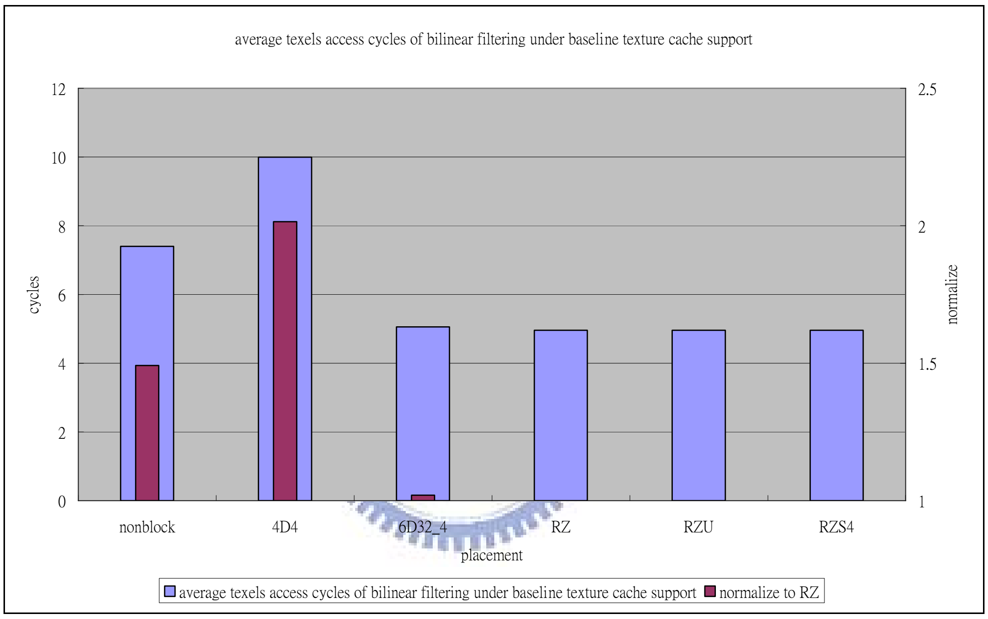


Figure 4-2-2-3 average texels access time of bilinear filtering in baseline texture cache support

Figure 4-2-2-3 shows that the average texels access time of bilinear filtering of RZ-based placement is improved ~2% compare to 6D32_4 placement, ~101% compare to 4D4 placement and ~49%compare to Nonblock placement under baseline texture cache support.

4.2.3 Results under texture cache support 1

Under texture cache support 1, the placement which places the required texels continuous in the same cache line can improve the average cache access counts. In figure 4-2-3-1 shows that RZU can improve ~6% of average counts by changing z-shape to u-shape and improve ~22% of average counts compare to RZ placement, ~8% of average counts compare to 4D/6D placement by using 4*4 snaked-tile size. However, Nonblock placement could be the best. This is because the required four texels of bilinear filtering are almost always two and two continuous and rarely discontinuous.

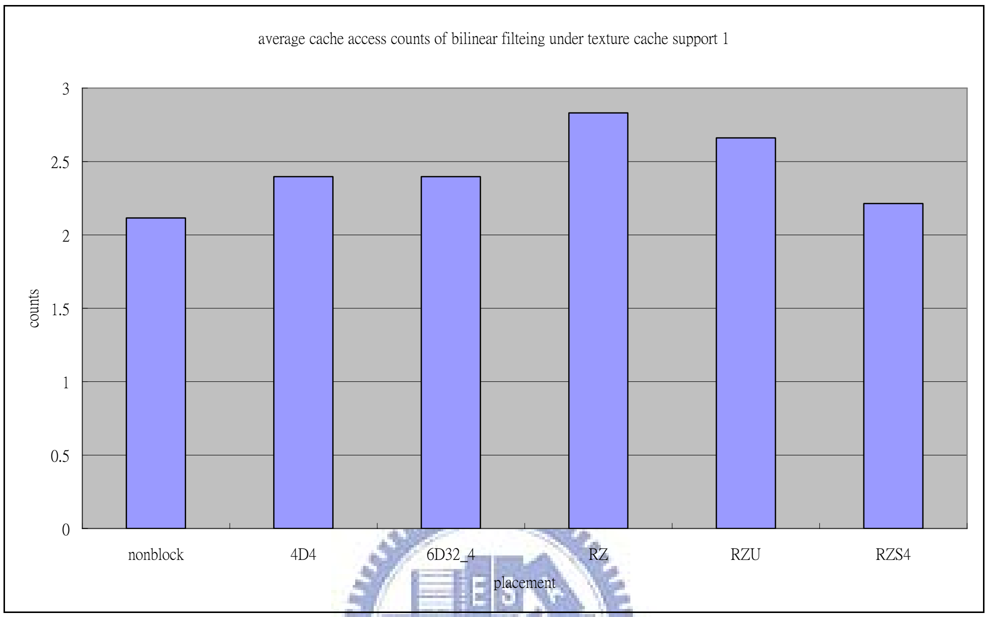


Figure 4-2-3-1 average cache access counts in texture cache support 1

However, in figure 4-2-3-1, we only know the average cache access counts. We don't know how many portion of the average counts is cache miss and how many portion of it is cache hit.

Figure 4-2-3-2 shows the cache miss rate, total cache access counts and hit counts under the texture cache support 1. The blue bar shows the total cache access counts, red bars shows the miss counts and yellow bars shows the miss rate under texture cache support 1. Although the average cache access counts of bilinear filtering of Nonblock is best, the miss rate is worse than 6D/RZ-based placement. Thus, the average texels access cycle of bilinear filtering may not be the best. On the contrast, the miss rate of RZ placement is best, but the total cache access count is worse than the other placement. The average texels access cycle of bilinear filtering may not be the best, too.

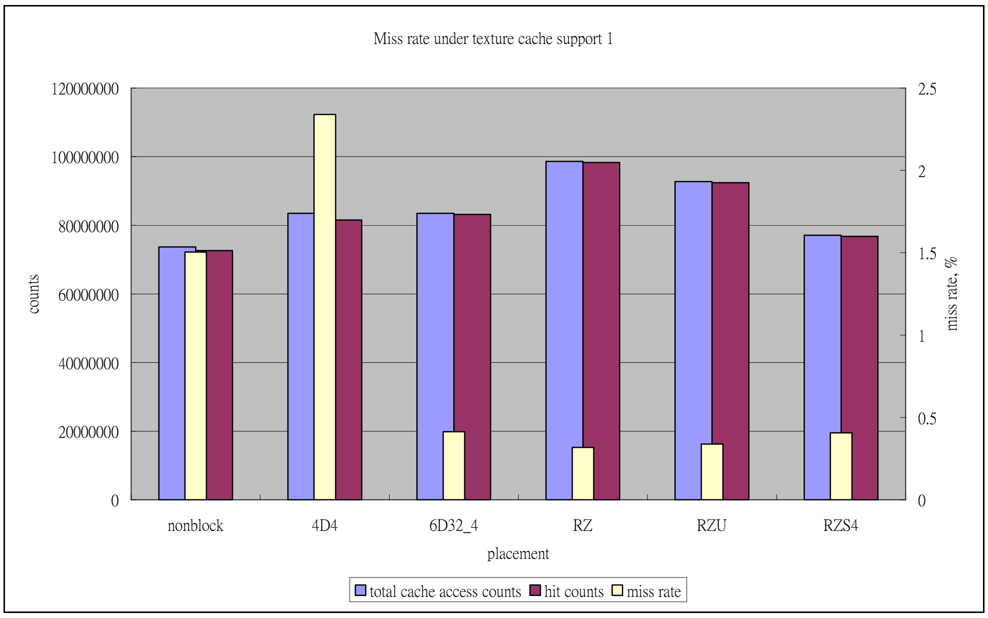


Figure 4-2-3-2 miss rate in texture cache support 1

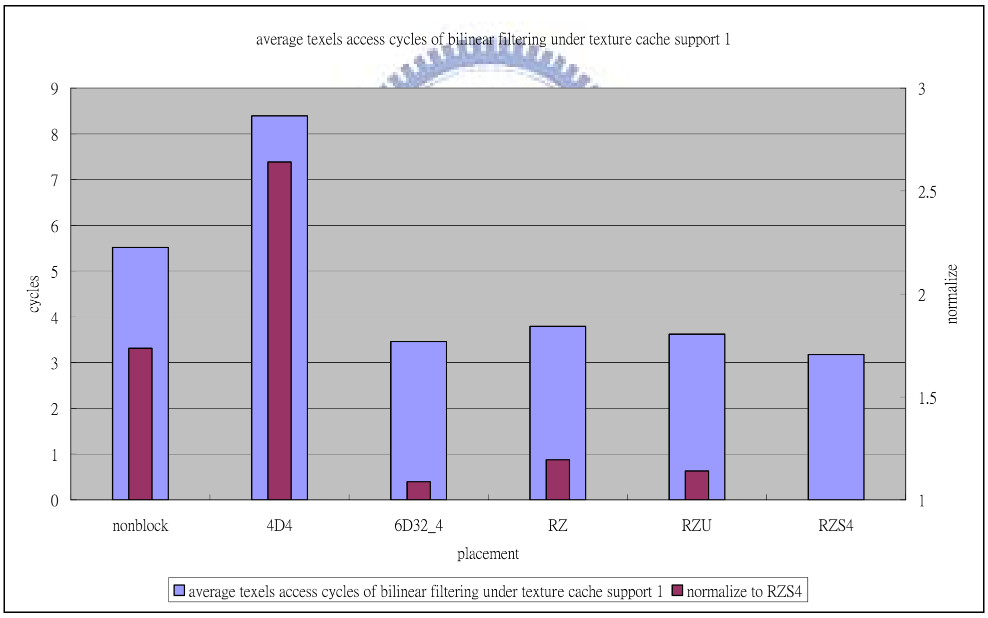


Figure 4-2-3-3 average texels access time of bilinear filtering in texture cache support 1

Figure 4-2-3-3 shows the average cache access counts of RZS4 is best and can improve ~9% compare to 6D32_4 placement, ~164% compare to 4D4 placement and ~74%compare to Nonblock placement under texture cache support 1. And we also notice that by changing z-shape to u-shape in RZ, we can improve ~5% of average texels access cycles of bilinear filtering of RZU placement compare to RZ placement. And by changing 2*2 z-tiled size to 4*4 snaked-tile size in RZ, we can improve ~19% of average texels access cycles of bilinear filtering of RZS4 placement compare to RZ placement

4.2.4 Results under texture cache support 2

Under texture cache support 2, the placement which places the required texels within the same cache line can improve the average cache access counts. In figure 4-2-4-1 shows that under cache line size is 64 bytes, the average cache access counts of bilinear filtering of 4D/6D/RZ-based placement could be the same. And improve ~11% compare to Nonblock placement. The average cache access counts of them are the same due to cache line size 64 bytes can place 4*4, 16 texels like a square-like in the texture.

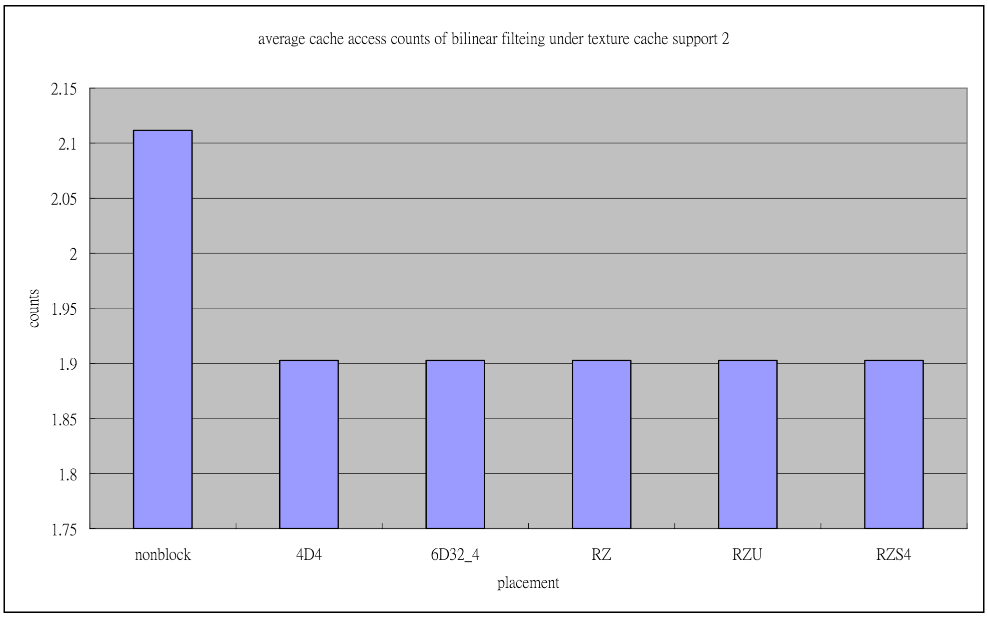


Figure 4-2-4-1 average cache access counts in texture cache support 2

Figure 4-2-4-2 shows the cache miss rate, total cache access counts and hit counts under the texture cache support 2. And the blue bar shows the total cache access counts, red bars shows the miss counts and yellow bars shows the miss rate under texture cache support 2. Although the average cache access counts of bilinear filtering of Nonblock is worse then 4D placement, but the miss rate is better than 4D placement. Thus, the average texels access cycle of bilinear filtering could be better than 4D placement.

And the figure also shows that RZ-based placement can improve ~0.05% on miss rate compare to 6D32_4 placement, ~2.47% compare to 4D4 placement and ~1.03% compare to Nonblock placement under texture cache support 2.

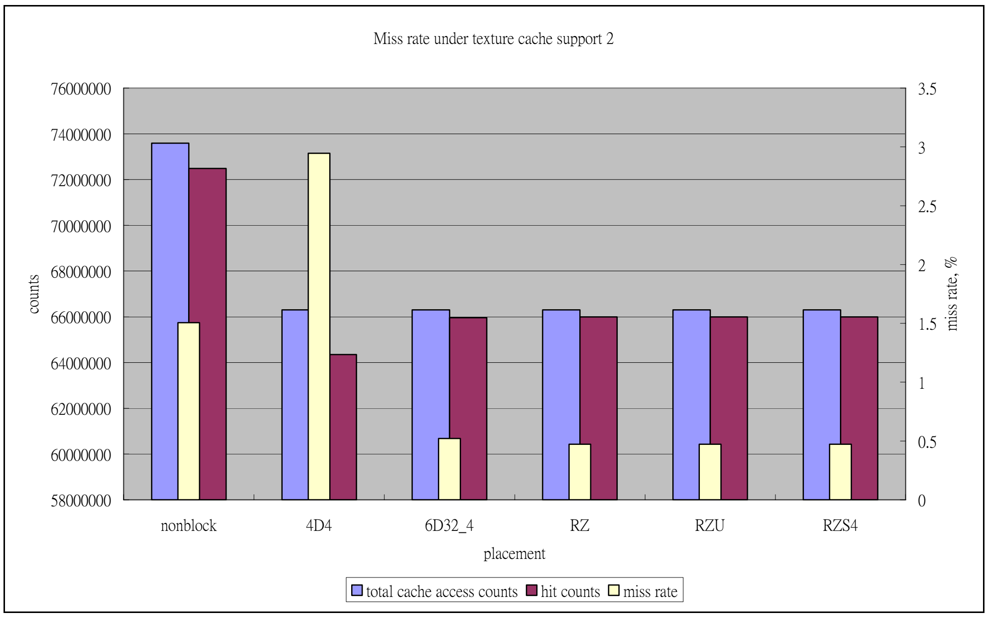


Figure 4-2-4-2 miss rate under texture cache support 2

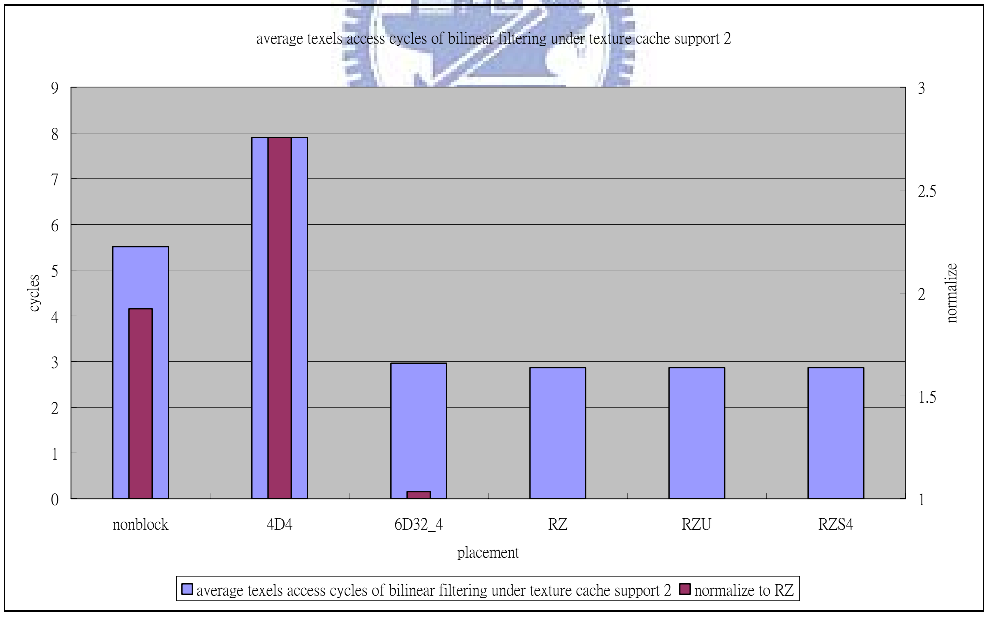


Figure 4-2-4-3 average texels access time of bilinear filtering in texture cache support2

Finally, under texture cache support2, figure 4-2-4-3 shows the RZ-based placement can improve ~3.5% of average texels cycle of bilinear filtering compare to 6D32_4 placement, ~101% compare to 4D4 placement and ~49%compare to Nonblock placement under texture cache support 2.

Chapter 5 Conclusion

5.1 Conclusion

In this thesis, we propose the new placements which is target to improve the average texels access time of bilinear filtering by improving cache hit rate, address translation time, average cache access counts under three kinds of possible texture cache support.

In the baseline texture cache support, by using recursive concept, we can improve the spatial locality of the required four texels of bilinear filtering. Thus, the miss rate of RZ-based placement is improved and the average texels access cycle of bilinear filtering is improve ~2% compare to 6D placement.

In texture cache support 1, by changing shape and tile size and also adapt the recursive concept, we can not only improve the miss rate but also the average cache access counts of bilinear filtering. Thus, the average texels access cycle of bilinear filtering is improved ~ 9% compare to 6D placement.

Finally, although the average cache access counts of 4D/6D/RZ-based is the same, 2, we can still take the advantage of recursive concept to improve the hit rate. The average texels access cycle of bilinear filtering is improved ~ 3.5% compare to 6D placement.

5.2 Future work

Since the bilinear filtering may have spatial locality, in fully associative cache, LRU replacement policy may have chance to be improved by using the other strategies. We found that the locations/addresses of the currently required four texels of bilinear filtering in the texture maybe far away than previous required four. And the addresses of required four texels of the next bilinear filtering maybe close to the previous nearby region. Thus, time strategy in LRU can be changed by using distance strategy in the replacement to gain more cache performance benefits.

Reference

[1] Foley J, van Dam A, Feiner SK, Hughes JF, “Computer graphics: principles and practice”, 2nd ed. Reading MA: Addison-Wesley, 1990

[2] Watt A, “3D computer graphics”, 3rd Edition. Addison-Wesley: Harlow, England. 2000.

[3] Hennessy JL, Patterson DA. “Computer architecture: a quantitative approach”,
3rd edition. Morgan Kaufmann: San Francisco. 2003.

[4] Ziyad S. Hakura and Anoop Gupta, “The design and analysis of a cache architecture for texture mapping”, 24th International Symposium on Computer Architecture, 1997.

[5] Michael Cox, Narendra Bhandari and Michael Shantz ,“Multi-level texture caching for 3D graphics hardware”, ACM/IEEE International Symposium on Computer Architecture, 1998.

[6] Homan lgehy, Matthew Eldridge and Kekoa Proudfoot, “Prefetching in a texture cache architecture”, Eurographics/SIGGRAPH Workshop on Graphics Hardware, 1998.

[7] Igehy H, Eldridge M, Hanrahan, P, “parallel texture caching”, SIGGRAPH/Eurographics Workshop on Graphics Hardware. 1999.

[8] Se-Jeong Park, Jeong-Su Kim, Ramchan Woo, Se-Joong Lee, Kang-Min Lee, Tae-Hum Yang, Jin-Yong Jung and Hoi-Jun Yoo, “A reconfigurable multilevel parallel texture cache memory with 75-GB/s parallel cache replacement bandwidth”, journal of solid-state circuits, vol. 37, no. 5, may 2002.

[9] Chun-Ho Kim and Lee-Sup Kim, “Adaptive selection of an index in a texture cache”, the IEEE International Conference on Computer Design, 2004.

[10] Victor Moya del Barrio, Carlos González, Jordi Roca, Agustín Fernández, “ATTILA: a cycle-level execution-driven simulator for modern GPU architectures”, 2006 IEEE International Symposium on Performance Analysis of Systems and Software.

[11] Chris Y. Chung, RaviA. Managuli and Yongmin Kim, “Design and evaluation of a multimedia computing architecture based on a 3D graphics pipeline”,

IEEE ,Application-Specific Systems, Architectures and Processors, 2002.

[12] Williams L, “Pyramidal parametrics” ,Computer graphics and interactive techniques. 1983.

[13] J. Chittamuru, J. Euh, and W. Burleson, "An Adaptive Low Power Texture Mapping Architecture", IEEE Mid West Symposium On Circuits and Systems 2002

[14] Microsoft, Microsoft DirectX9 Software Development Kit, Microsoft Corporation.

[15] John Montrym, Henry Moreton, “NVIDIA GeForce 6800”, NVIDIA Corporation

[16] MAX+PLUS II Development Tools manuals, http://www.altera.com/literature/lit-mp2.jsp

[17] Verilog design guide, http://www.doulos.com/knowhow/verilog_designers_guide/

[18] Synopsys, design vision, http://www.synopsys.com/sps/sps.html

[19] Benchmark, Doom3, http://www.doom3.com/

[20] Benchmark, Quake4,

Appendix

A.1 The time delay of mux

	1-1	2-1	4-1	8-1	16-1	32-1	64-1	128-1	256-1	512-1	1024-1	2048-1
Time (ns)	0	0.09	0.14	0.25	0.35	0.45	0.55	0.65	0.75	0.85	0.95	1.05

A.2 The spec of current GPU architecture

	Die processing technology	Clock rate	Cycle time
Nvidia GeForce 6800 Ultra	130nm	400Mhz	2.5ns
ATI Radeon 9600	130nm	325Mhz	~3ns
ATI Radeon 9600Pro	130nm	400Mhz	2.5ns