國立臺灣大學電機資訊學院電子工程學研究所

碩士論文

Graduate Institute of Electronics Engineering
College of Electrical Engineering & Computer Science
National Taiwan University

Master Thesis

適用於行動繪圖之可重組執行緒濾除器設計

Architecture Design of Configurable Thread Culling Unit for Mobile GPUs

余孟璘

Yu, Meng-Lin

指導教授：簡韶逸 博士

Advisor: Chien, Shao-Yi, Ph.D.

中華民國 100 年 12 月

December, 2011


國立臺灣大學
人格教育・學生愛國

誌謝

能順利完成這本碩論，到現在回想起來還是覺得不可思議。首先最要感謝的是我的指導教授簡韶逸老師，願意收留我這個從外校來的學生，面對初次接觸這領域什麼都還懵懵懂懂的我，老師總是非常有耐心地一步步教導著，雖然老師是個大忙人，但老師都會盡力協助每位同學，讓我覺得能夠來到這間實驗室真的非常的幸運，也學到了很多。

還要感謝能撥冗擔任我口試委員的強大老師們，有陳維超老師、陳彥光老師還有范倫達老師。承蒙您們的教導，讓我在口試當下領悟了很多知識!

再來要感謝 GPU 團隊，有家銘、Steve、豪哥、春益、彥璋、嘉洋、崇耀還有柏舜，因有學長學弟們的幫忙，我才能完成這份研究。謝謝家銘學長即使自己忙翻了，還是願意抽空提供我技術上的指導。謝謝 Steve 一路的陪伴，從最初沒有題目到最後的成果展現，全賴你的幫助與分享，我才能順利走完。真的非常感謝每位團員在這幾年來不厭其煩地提供我許多建議與協助，能跟大家一起合作讓我覺得非常開心。此外還要感謝 421 的每位同學們，跟你們一起在實驗室相處的時光總令人難忘，因為實驗室的氣氛融洽，大家都會互相幫忙，所以都可以放鬆心情地一起做研究。

接著要感謝我的家人還有朋友們一路上給我的支持與鼓勵。感謝爸爸、媽媽、哥哥與妹妹在我遇到困難心情低落的時候，你們給了我最大安慰讓我備感窩心。也要感謝奕翔這幾年來的陪伴，在我最無助的時候你總是在身邊陪著我，分擔我的煩惱，陪我度過許多難關。還有富貴、姚姚跟母婷，跟你們一起度過的時光總是很溫馨又愉快，讓我在繁忙之餘仍不時感到朋友的溫暖。還要謝謝我的新室友婉菁，雖然我跟你原本都不認識的，但你總是給我一些貼心的鼓勵，真的很感動。

在此謝謝所有幫助過我的人，因為有大家的陪伴，我才能順利完成碩士的研究，衷心感謝大家!


國立臺灣大學
人格教育・學生愛國

中文摘要

因 3D 繪圖處理器(GPU)提供了強大的運算能力，在目前市售的電腦甚至是手持裝置上有愈來愈多它的蹤跡。大部分的繪圖處理器屬於多核心架構，相當適合用來處理平行運算。因而如此，除了傳統的繪圖功能外，有許多適合平行運算的其他演算法也相繼實現在繪圖處理器上。

然而在手持裝置上因功率成本的考量下，繪圖器所能提供的運算資源仍然相當有限，因此應該藉由某些技術來減少手持裝置上應用的運算量。其中有些概念其實已經普遍實現在目前的繪圖處理器上，像是在場景中會有許多被擋住的物件，這些物件並不會顯示在最後的螢幕上，因此可以在繪圖的流程中移除這些物件的運算以節省資源。除此之外，我們也觀察到，在某些多媒體應用下只有部分畫素(pixel)的運算結果被視為重要，我們稱之為 ROI 應用，而那些位於非重要區域內的運算也應該可以被提早移除。因此基於以上的概念，在這篇論文中我們提出了一個可重組執行緒濾除器，將之整合在繪圖的內插流程(rasterization)中，用來減少手持裝置上繪圖處理器多餘的運算。

可重組執行緒濾除器可支援兩種運算模式，且這兩種模式分別都在方塊(tile)以及畫素這兩層執行條件測試。第一種，在 3D 繪圖的模式下，條件測試會移除被擋住的物件以及標示出可見的物件。我們的實驗結果顯示，有 14%的執行緒可以被移除，有 15%的執行緒會被標示為可見並可減少頻寬，且相比於沒有濾除器時可加速 1.1 倍。第二種，在 ROI 應用的模式下，條件測試會移除非重要區域內的執行緒運算。實驗結果顯示，在 Viola-Jones 人臉偵測的演算法下，相較於之前做法最多可以有 25 倍的加速，而在 Gabor 應用下，可以有 6 倍的加速。

最後，我們使用 TSMC 65 奈米製程來驗證我們的硬體設計，而我們所提的可重組執行緒濾除器所帶來的面積成本少於 5%。此外，我們也完成了此繪圖系統在 FPGA 平台上的驗證，約佔用了 FPGA 上 40K 個運算單元 (slice)。


國立臺灣大學
人格教育・學生愛國

Architecture Design of Configurable Thread Culling Unit for Mobile GPUs

Meng-Lin Yu

Advisor: Shao-Yi Chien

Graduate Institute of Electronics Engineering

National Taiwan University

Taipei, Taiwan, R.O.C.

December 6, 2011


國立臺灣大學
博愛 · 學術 · 人格

Contents

List of Figures

List of Tables

Abstract

GPU becomes popular in modern computing devices due to its outstanding processing ability. Modern GPUs usually consist of multiprocessors that are suitable for parallel processing. As the design of GPU becomes more general in recent years, not only traditional 3D graphics rendering but also many general applications tend to utilize the plentiful resources on GPU.

For mobile GPUs, the computing resources are rather limited; therefore some techniques should be established to reduce workload. It is known that the rendering operations of occluded objects should be avoided in 3D graphics applications. Moreover, for some applications which exploit region of interest (ROI) processing, only the operations in the ROI should be executed. In order to reduce the computation with the above-mentioned concepts, in this thesis, a configurable thread culling unit (TCU) is proposed to enhance the performance of mobile GPUs.

TCU can be configured into two operating modes and performs at both tile and pixel levels. First, for 3D rendering applications, an early Z test is employed to remove occluded regions and detect visible regions. Experimental results show that in average 14-% of pixels are discarded, 15-% of pixels are marked as visible and the speed up of 1.1× can be achieved compared to the case where no culling is carried out. Second, an early stencil-like test is executed to discard non-interested regions in ROI processing. Experimental results show that compared to predicate execution, the performance is enhanced by 25× and 6× for Viola-Jones face detection and Gabor feature

extraction in salient region, respectively.

Finally, the proposed design is integrated into GPU and is verified in hardware implementation. Designed with TSMC 65-nm technology, less than 5% total area is increased with TCU, including a 1KB cache, which shows that the hardware cost overhead of proposed TCU is quite small. Furthermore, the GPU system is prototyped on a FPGA platform and the function has been verified, where about 40K slices are utilized in Xilinx Vertex-5 XC5VLX330.

Chapter 1

Introduction

GPU(graphics processing unit) is developed to accelerate graphics rendering applications. As the complexity of rendering a scene increases, the heavy workload exhausts the resource of CPU. GPU then evolves to provide outstanding processing power. In the beginning, GPU is a dedicated ASIC targeting at the fixed pipeline of rendering. With the improvements in technology and researches, GPU has become programmable through some specific programming models.

Nowadays, GPU becomes an important computing resource in either commodity desktops or mobile devices. It evolves rapidly to deliver significant processing power for supporting the growing demands on high-throughput and complex applications. Owing to the property that a large amount of data usually run the same shader program in rendering, SIMD architecture with either scalar processors or vector processors is usually adopted in most GPUs to exploit data parallelism. Among the latest graphics card in desktops, NVIDIA GTX 590 has 1024 scalar processors and AMD Radeon HD 6990 has 3072 processors. Modern products tend to provide amazing power of parallel processing in multi-processor architecture.

1.1 Graphics Rendering

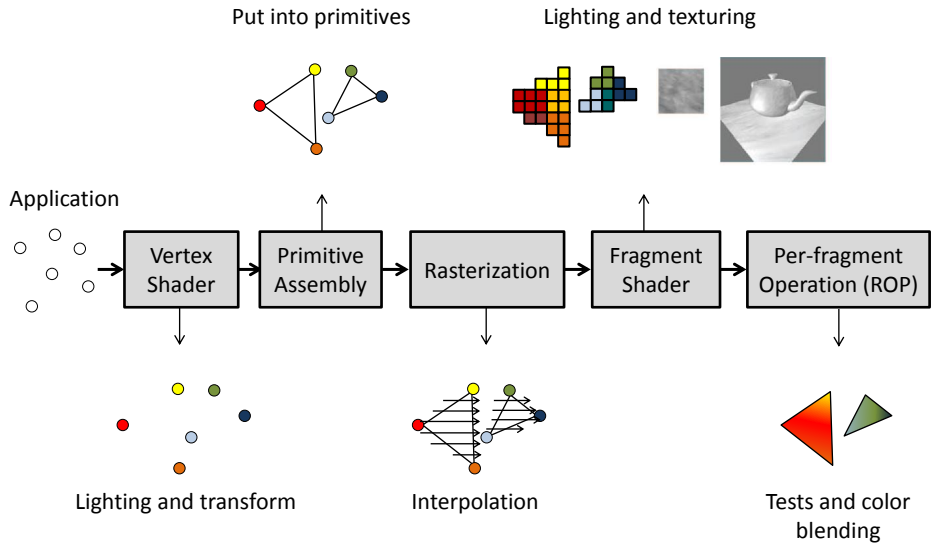


Figure 1.1: Graphics pipeline.

The traditional rendering pipeline contains five stages as shown in Figure 1.1. In the beginning, input is the geometry information of the scene provided by programmers, such as the positions, colors and other attributes of vertices together with the environmental setups of scenes. These data are fetched and transferred to vertex shader. Vertex shader is a programmable stage that processes each vertex with the shader program. In this stage, vertices can be transformed, rotated and shaded based on the shader program. Next, primitive assembly puts those processed vertices into a primitive, which can be a point, a line or a triangle. Then in rasterization stage, vertices are transformed from world coordinates into camera coordinates first. Then all values, including color, position, depth and attributes are interpolated for each pixel (or fragment) that a primitive is projected to in the screen. The interpolation is calculated with plane equations setup by the attributes of vertices of a primitive. Before executing pixel shader, some techniques such as depth test, frustum culling, small-primitive culling and back-face culling

can be applied to this stage to reduce the workload. Fragment shader then processes lighting and texturing on each interpolated fragment. Texturing is a procedure that shades a pixel with a texel fetched from a texture image. It can present dedicated scenes without complicated computations so is frequently used in applications. Fragment shader and vertex shader are the only two stages that are programmable in this pipeline. Programmers can define the operations on vertices and pixels in shading languages such as Cg, openGL shading language (GLSL) [2] and high level shading language (HLSL) [3]. In GPU architecture, a vertex or a fragment is assigned as an active *thread* which is scheduled to processors and run through the whole shader program independently. Finally, per-fragment operations are performed on shaded fragments in ROP stage. This stage contains scissor test, depth test (Z test), stencil test and color blending. Scissor test defines a rectangular region on the screen that is writable. Z test compares the depth values of the coming fragment and the corresponding pixel on the depth buffer. If the coming fragment is farther, it will not be drawn on the screen. Stencil test is used to determine whether a fragment can be updated or not with a per-pixel mask. It is usually used for special effects such as shadows, reflections and cut-outs when rendering a scene. In the end, the color buffer will be updated if a fragment passes all the tests.

Early Z test which performs Z test before ROP stage to reduce redundant work is commonly adopted in GPUs. It is proposed to discard occluded regions as shown in Figure 1.2 as early as possible during the pipeline stages. In Figure 1.2, the darkened region of Triangle 2 is occluded by Triangle 1 so it will not be shown on the screen. Early Z test detects these occluded regions and discards them so that the processing power and bandwidth of shader, texturing, and per-fragment operations can be saved.

In a similar way, early stencil test performs stencil test before ROP stage for saving computing resources. The stencil buffer stores a per-pixel mask

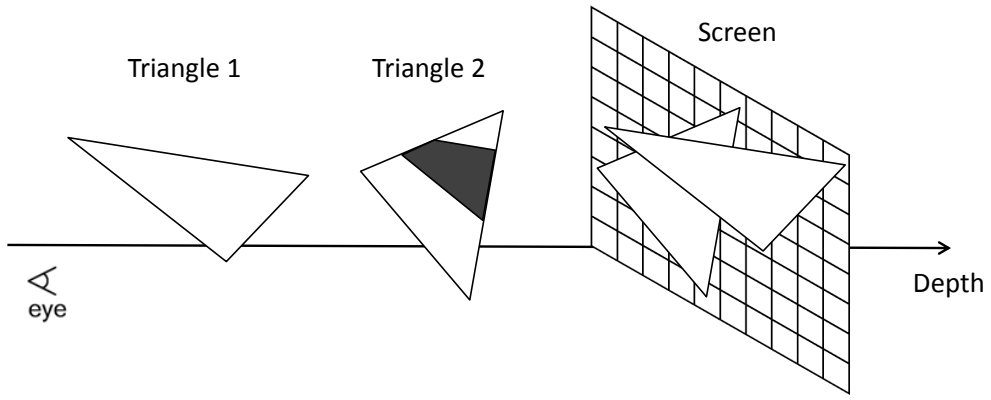


Figure 1.2: Early Z test Example : Occluded region(darkened region) in triangle 2 can be discarded during early Z test.

rendered in last frame and is accessed in early stage to remove failed fragments. It appears in the products of ATI [4] and NVIDIA [5] [6], however is less concerned in researches.

1.2 GPGPU

To better utilize the plentiful processing resources on GPUs, *General-Purpose computation on Graphics Processing Units* (GPGPU) is proposed to exploit the advantage of parallel processing ability. Many researches in the fields of physics simulation [7], linear algebra [8], multimedia applications [9] endeavor to figure out the method of implementation which gets the best gain.

Several programming models are designed to support GPGPU, starting from earlier Cg, GLSL, HLSL, which are originally dedicated for graphics rendering pipeline, to current tailored general purposed programming models - CUDA [10] and OpenCL [11]. Among them, CUDA and OpenCL draw most of the attention. As CUDA arises, many researchers focus on exploiting

the parallelism inside GPUs for computationally intensive applications. Such general-purpose programming model drives the popularity of GPGPU. Parallel threads resources are allocated hierarchically, where *block* and *thread* are the basic execution units used in CUDA. OpenCL, which is a general-purpose computing framework across heterogeneous platforms, supports parallel processing as well. Through the API setting, the idea behind OpenCL is similar to CUDA. *Work-group* and *work-item* are the corresponding basic units used in OpenCL. While CUDA is not available on mobile devices now, OpenCL establishes an embedded profile which is a subset of full standard running on embedded systems with more restrictions [12].

It is known that most low-level image processing and computer vision algorithms can be effectively accelerated with GPUs because of the characteristic of independence between regions or pixels, such as image filtering and linear feature extraction; however, it is not the case for high-level applications, where object-based processing and processing in ROI (Region of Interest) are often involved. That is, not every extracted feature is significant for later processing stages, and substantial feature information is generally concentrated in salient regions [13] or ROI. For example, the stage rejection process in Viola-Jones face detection framework [14] can incrementally eliminate possible face candidate positions. Feature extraction [15] [16] computes the description vectors for only interested points. These ROI processing on GPU will cause branch divergence and may degrade the performance due to distinct behavior between threads especially in processors with wide SIMD.

1.3 Limitation of GPUs on Mobile Devices

Under the consideration of power consumption and area cost, the computing resource of mobile GPUs is more limited compared to the GPUs on desktops. For example, NVIDIA Tegra2 has only 8 cores and PowerVR SGX

series 5 has only 2 to 16 cores (SGX543, SGX544, SGX554). It is then more challenging to realize the mentioned applications efficiently by GPUs on mobile devices.

On nowadays mobile devices, the programming of GPGPU is done through a light-weight open graphics standard – OpenGL ES 2.0 instead. Before CUDA and OpenCL become mature on mobile devices, it is common to utilize the general-purpose processing power of GPUs with shading language. In such programmable graphics pipeline, users can write vertex programs and pixel programs running on multiprocessors. Initially, vertex programs operate on geometry models. Next, the rasterizer takes charge of a scan conversion procedure to convert transformed vertex data into fragments. The generated fragment threads then activate the shader processors to run fragment programs. Identically, it is straightforward to associate fragment threads as 2-dimensional threads, and the allocated thread number is determined by the size of frame buffer. Similar to CUDA and OpenCL, the same programming concepts can also be performed with shading language.

1.4 Motivation

In this work, we want to reduce redundant evocation of threads for both graphics rendering and ROI processing of GPGPU, targeting at mobile GPUs where resource is not sufficient. A configurable thread culling unit is proposed to improve the efficiency of GPUs. When rendering a scene, an early Z test is performed; and when processing ROI of multimedia applications, an early stencil-like test is performed. The experimental results show that the performance of GPU is enhanced and the hardware cost is relatively small through hardware sharing.

1.5 Thesis Organization

The thesis is organized as follows. Algorithms of early Z tests of previous works are introduced and summarized in Chapter 2. Besides, some related solutions to the ROI processing on GPU are also presented in Chapter 2. In Chapter 3, we explain the architectures of proposed configurable thread culling unit and simulation model. Two operating modes and their experimental results are also presented in this chapter. Then in Chapter 4, the hardware analysis and implementation results are demonstrated. Finally, Chapter 5 concludes the thesis.


國立臺灣大學
人格教育・學生愛國

Chapter 2

Related Works

In this chapter, some previous works of early Z test algorithms will be discussed in Section 2.1. Moreover, issues and current solutions of branch divergence on GPU will be discussed in Section 2.2.

2.1 Early Z Test Algorithm

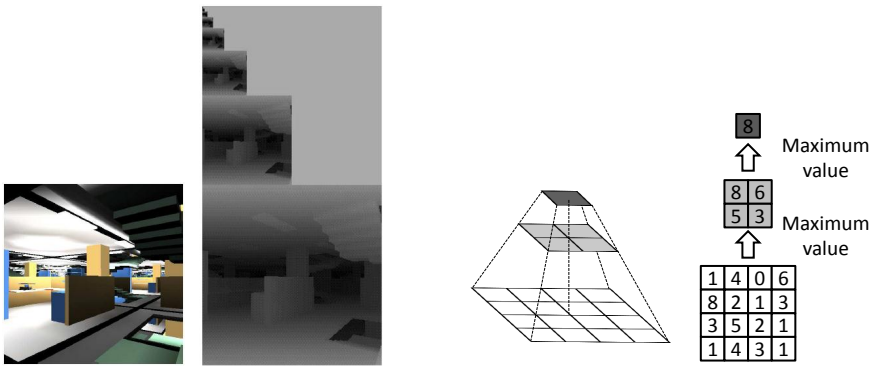
Early Z test algorithms can be classified by their design targets; Z-max algorithm reduces five times of memory access by detecting occluded pixels; Z-min algorithm saves one time of memory access for each visible pixel; hybrid algorithm contains Z-max and Z-min algorithms to take advantages of saving memory accesses for both occluded and visible pixels.

2.1.1 Z-Max Algorithm

As explained in Figure 1.2, Z-max algorithm is designed to skip redundant works by finding occluded regions in an object. During the process, the maximum of Z-values for each region of currently unfinished scene is stored in an additional memory. Usually maximums are stored for every tile (8×4 pixels in our system) to save memory area. When the Z-value of the coming object is farther than the corresponding stored Z-value, which indicates it is

occluded by other objects that are already drawn, it is discarded from the pipeline. Usually, Z-max algorithm is implemented before, during or after rasterization depending on the culling level, which will be explained later. When an object is culled, redundant stages of pixel shader and ROP can be saved; besides, texture read, color read/write, and depth read/write can be saved.

Hierarchical Z-buffer (HZ-buffer) [17] is the most popular Z-max algorithm proposed in 1993. They hierarchically examines whether an object is occluded by using a Z-pyramid. Z-pyramid contains many layers of Z-values where each upper layer is downsampled from the lower layer by a factor of 2 on each side as shown in Figure 2.1. When downsampled from the lower layer, a 2×2 maximum filter is issued to ensure that every stored Z-value is the farthest depth in its corresponding region. Besides, the bottom layer of a Z-pyramid is the external depth buffer accessed in ROP stage. Whenever a new depth is updated after the Z test in ROP stage, a maximum filter is employed from bottom layer to top layer to update the entire Z-pyramid with correct Z-values. However, this introduces substantial overhead of memory bandwidth and computing latency.



(a) The Z-pyramid of a scene. (b) Construction of a Z-pyramid.

Figure 2.1: Illustration of Z-pyramid.

In HZ-buffer, comparison tests are performed between the coming Z-value and several stored Z-values, which are fetched gradually from the upper layer to the lower layer of the Z-pyramid, until the object is confirmed to be rejected or it reaches the bottom layer of the Z-pyramid. The advantage of HZ-buffer is that it is efficient to cull large occluded objects since fewer comparison tests are needed in upper layers. However, it costs more comparison tests for non-occluded objects because tests are terminated only when the bottom layer is reached.

Many works [18] [19] [20] [21] are originated from HZ-buffer. Since the management of HZ-buffer is hard, they implement only a few layers of Z-pyramid. Usually, an additional memory is allocated with the size of the lower resolution than that of the depth buffer. That is, maximum Z-values are stored for each tile, not for each pixel instead. Updates are then accomplished by managing this additional memory, without accessing to the original depth buffer. However, in this way, additional mechanisms are required to record the Z-values in a tile for correct updates. As illustrated in Figure 2.2, it is needed to make sure we update the Z-pyramid with the maximum Z-value within the range a tile covers. If we do not know the range the entire tile covers, updates can not be carried out. In [21], a cache is implemented to store Z-values in a tile. The maximum Z-value can be updated to the additional memory when all values in a tile have already be drawn and cached.

Among these works, different culling levels are selected. Culling level is the granularity of an object that is used to perform Z test, varying from primitive, tile to pixel. For example, every tile an object covers will go through Z test if the culling level is at tile. Finer the culling level is, the higher culling rate can be achieved, but on the other hand, more processing time is taken.

Depth filter [22] proposed in 2008 uses the concept different from HZ-

(a) Z-values of a fully-covered tile. (b) Z-values of a partially-covered tile.

Figure 2.2: Illustration of Z-max update. The current stored Z-max value is 9 for example. In case (a), 7 is updated since it is smaller than the stored Z-max. However, in case (b), 7 can not be updated because the depth values of those darken pixels are not known. It can not be sure that 7 is the maximum of entire tile.

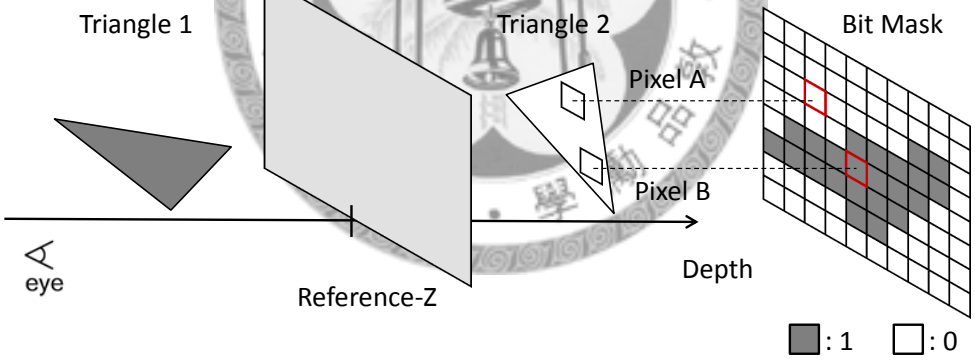


Figure 2.3: Illustration of depth filter. Triangle 1 encodes the pixels it covers as 1 because it is in front of *reference-Z*. Although Triangle 2 is farther than *reference-Z*, Pixel A can not be discarded since the corresponding value on bit-mask is 0, which means no pixel is drawn in front of Pixel A. However, Pixel B on Triangle 2 can be discarded with the corresponding value on bit-mask being 1.

buffer. In order to reduce hardware overhead, a bit-mask instead of a Z-pyramid is utilized as shown in Figure 2.3. In the beginning, a Z position (called *reference-Z*) is chosen and a bit-mask is created for each pixel on screen with initial value - 0. Then if the Z-value of coming pixel is in front of *reference-Z* , corresponding position on the bit-mask is encoded as 1. Besides, if it is farther than *reference-Z* and the corresponding value on the bit-mask is 1, the pixel is occluded by other objects so is discarded. The advantage is that memory is simply managed and is of small size. But the disadvantage is that it is hard to find a proper position of *reference-Z*, which is influential to the culling rate. [22] records the distribution of Z-values in ROP stage and calculated the best value as the *reference-Z* for the next frame.

2.1.2 Z-Min Algorithm

Z-min algorithm [23] is proposed to skip Z read in ROP stage by indicating visible regions. Different from Z-max algorithms, the minimum of Z-values for each region (usually per tile) is stored in an additional memory. If the Z-value of the coming object is nearer than the stored Z-value, the object is marked as visible so that Z read in ROP stage can be saved because the Z test in ROP will certainly pass. It is beneficial especially for pixels that are first drawn. [23] claims that more bandwidth can be saved in Z-min algorithm than that in Z-max algorithm for scenes whose depth complexity is smaller than 4. Besides, the management of memory update is quite simpler compared to that in Z-max algorithm. In Z-min algorithm, the coming Z-value can be updated as long as it is nearer than the stored Z-min value. However in Z-max algorithm, it is needed to be sure that the coming Z-value is not only nearer than stored Z-max value but also is the maximum in that region (or tile).

[24] [25] adopt both Z-max algorithm and Z-min algorithm in their early Z tests. Both advantages mentioned can be taken to save more bandwidth.

2.1.3 Summary of Previous Works

The collection of previous works on early Z test are summarized in Table 2.1. Check mark indicates the selected culling level in each work. And in the memory column, an $n \times m$-buffer is the additional memory of the size of the original Z-buffer downsampled by $n \times m$. A LR-buffer indicates the size is not specified in that work. Besides, some works use masks to manage updates in Z-max algorithm. The disadvantages of each work are also listed in the table. Culling at primitive or tile level is fast but leads to low culling rate. In the opposite, culling at pixel level results in high culling rate but is less efficient. Although [21] culled at both primitive and pixel level, a primitive is actually hard to be culled entirely. Furthermore, there are two layers of Z-pyramid with a cache for updates in [21], and also a compression technique is exploited to reduce memory usage. The hardware cost is relatively high.

2.2 Branch Divergence on GPU

Modern GPUs typically bundle multiple scalar threads into a batch, regarded as a warp by NVIDIA CUDA, and execute them by the same program in a lockstep on SIMD architecture. While more non-graphics GPGPU applications are realized on GPUs, complex control flows make threads in a warp divergent, leading to efficiency degradation on SIMD processors. Branch divergence occurs when threads in the same warp take different executing paths after a conditional statement such as *if-then-else*, *for*, *switch* and *while*. For some multimedia applications as mentioned in Section 1.2, processing of ROI causes thread divergence in a warp. In the following sections, possible implementations will be discussed, including jump instruction, predicate execution, warp re-scheduling and stream reduction.

Table 2.1: Summary of previous works of early Z test algorithm.

Previous works	Primitive	Tile	Pixel	Memory	Disadvantage
AHV [18], 1999		✓		LR-buffer	Low culling rate
Delay stream [19], 2003	✓			8x8-buffer	
Z-min culling [23], 2003		✓		LR-buffer	
Blocked-Z [20], 2009		✓		LR-buffer	
ATDF [24], 2006			✓	8x4-buffer Mask	Slow
Depth filter [22], 2008			✓	Bit mask	
UEZT [25], 2010			✓	8x4-buffer Mask	
Two-level HZ-buffer [21], 2003	✓		✓	8x8-buffer 16x16-buffer	1. Low culling rate (primitive level) 2. Hard management of memory

2.2.1 Jump Instruction

Jump instructions are commonly used in sophisticated processors such as CPUs or multiple instruction multiple data (MIMD) architectures when encountering conditional statements. An example program and its assembly pseudo code is shown in Table 2.2. "**beq**" and "**j**" are used to control the branch path of a thread. However, in SIMD architecture, jump instructions usually lead to complicated scheduling mechanism and degrade the performance of GPUs.

Table 2.2: Program example with jump instruction.

Program code	Assembly pseudo code
If (i != j)	**beq i j Label1** // *branch to Label1 if i equals to j*
Data = i + j;	**add Data i j**
else	**j Label2** // *jump to Label2*
Data = i - j;	**Label1 : sub Data i j**
	Label2 : ...

2.2.2 Predicate Execution

Predicate execution [26] is another scheme to deal with thread divergence within a warp. When encountering conditional branches, a predicate is allocated for each thread and is referred to as a flag for indicating taking or not-taking the current instruction. The operation of predication within a warp is shown in Figure 2.4. Conditional statements occur among the segment A-F in a program. Processors execute these segments one by one where some active threads will take calculating results while other inactive threads will not. Another simple program and its corresponding pseudo code in assembly is shown in Table 2.3. "**PSET**", "**PFLIP**" and "**PEND**" are generated to control the predicate flag for each thread. "**PSET**" sets the

flag as true if the condition matches. "**PFLIP**" reverses the flag when encountering "else" statement. Finally, "**PEND**" indicates the flag is invalid when conditional statement ends. During this period, only threads with positive flag will take results. Predicate execution is easy to be accomplished on SIMD organization; however, it introduces redundant computation since instructions with false predicates in a thread are still executed.

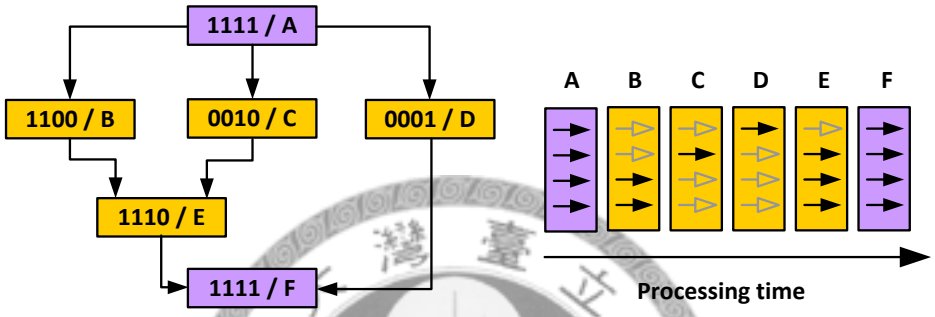


Figure 2.4: Predicate execution. For simplicity, it is assumed that four threads are executed in a lock step. Left : A-F indicate different segments of a program. All threads are valid in segment A. Only left two threads are valid in segment B. Similar concept in other segments. Right : Segments are executed successively. Only darkened threads will take results after each segment is processed.

2.2.3 Warp Re-Scheduling

To mitigate the performance degradation caused by branch divergence on CUDA architectures, [27] proposes a reformulation mechanism to generate a new warp when threads diverge. Based on a coarse-grained multi-threading architecture, interleaving multiple warps is commonly exploited in modern GPUs to hide memory latency. If branch occurs and threads in a warp take different paths, a new warp is generated by collecting inter-warp threads

Table 2.3: Program example with predicate excution.

Program code	Assembly pseudo code
If (i != j)	**Cmp equal i j** // *equal = (i==j)*
Data = i + j;	**PSET equal** // *set predicate = 1 if equal is TRUE*
else	**sub Data i j**
Data = i - j;	**PFLIP** // *flip the value of predicate if encounters "else"*
	add Data i j
	PEND // *predicate is invalid*
	...

that branch to the same path. It is crucial to balance the inter-warp processing speed and avoid bank conflicts in registers file when re-grouping a new warp. Therefore, [28] proposes to partition a diverged warp into two warp-splits, each of which gathers threads that branch to the same path and thus contains fewer threads than the original warp. These warp-splits are additional scheduling entities to hide latency. To sustain high utilization of SIMD lanes, properly re-converging warp-splits is necessary. Both these algorithms endeavor to alleviate the performance overhead of conditional divergence, but complicated scheduling mechanisms cause substantial hardware design efforts.

2.2.4 Stream Reduction

Stream reduction, also known as stream compaction, is commonly used in parallel algorithms to eliminate unwanted elements from a data stream. It is usually employed in multi-pass GPGPU applications, and the collected valid elements from stream reduction are sent to the next pass for further processing. This approach can be employed for many applications, such

as tree construction, traversal [29], ray tracing on GPU [29] and feature extraction [30]. Stream reduction can also be treated as a technique to discard branching statements in shader programs and further improve the workload balancing among shader processors [29].

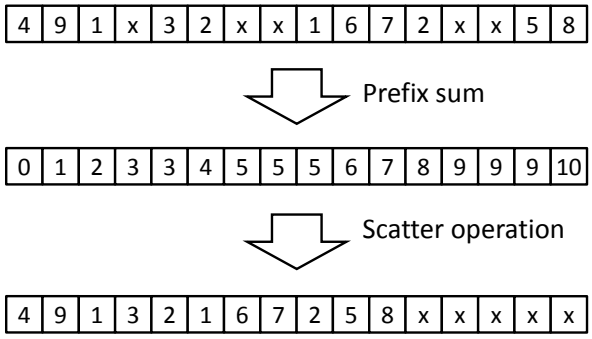


Figure 2.5: Illustration of stream reduction. Input stream contains both wanted and unwanted (marked as '×') elements. After the reduction program, wanted elements are written out sequentially in an array.

Usually, stream reduction is composed of a two-pass procedure: prefix sum (scan) computes the displacement by accumulating the number of valid elements before each element; in the second pass, the calculated displacement is used to write each valid element to the correct position in the output stream, as shown in Figure 2.5. General shading langauges support only gather operations that enable a thread to access arbitrary data but lack the ability of writing to random positions. [31] and [32] use binary search to locate valid element for each corresponding output with the complexity of $O(N \log N)$. [1] improves the complexity to $O(K \log K)$ by subdividing input stream into chunks of size K. However, with the emerging of CUDA architecture, scatter capability is also available on recent GPUs. The searching procedure in the second pass can be replaced with flexible scatter operations.

Being implemented with CUDA, [33] achieves a complexity of $O(N)$ and [34] gets even better performance. Although many efficient stream reduction algorithms are developed, it still takes additional efforts to realize, especially for GPGPU with shading language.

Chapter 3

Proposed Configurable Thread Culling Unit

3.1 Overview

Tile-based rasterization [35] as shown in Figure 3.1 is implemented in our system. It consists of three stages, setup, tile-based traversal and interpolation. In setup stage, edge equations, plane equations and the bounding box are calculated for later stages. Tile-based traversal finds tiles that are intersected with the primitive by checking the corners of each tile in the bounding box against edge equations. Only when four corners of a tile are detected to be outside the primitive, the tile will not be passed to the next stage. Then, the visibility of every pixel in a intersected tile is checked against edge equations. Besides, the varyings of each fragment are calculated by interpolation with plane equations. During the whole process, setup stage appears once for a primitive; tile-based traversal and interpolation are executed several times, depending on how many tiles a primitive covers. Among these stages, interpolation is the most time-consuming part since plenty of floating-point operations are needed for interpolating 32 fragments in a tile.

Thus, a configurable two-level thread culling unit (TCU) is proposed and

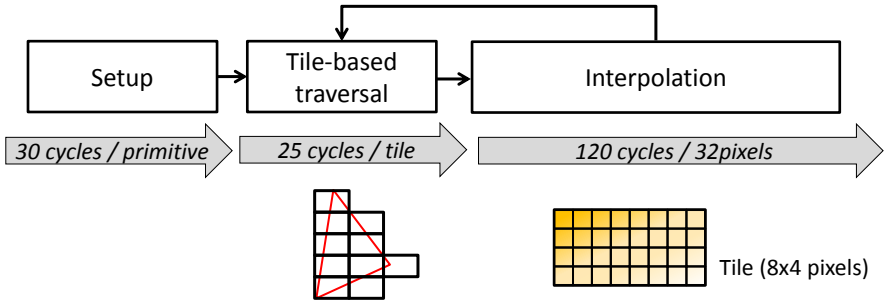


Figure 3.1: Tile-based rasterization.

introduced into our system. The modified pipeline is in Figure 3.2. TCU is configurable for two operating modes; an early Z test is performed for graphics rendering and an early stencil-like test is performed for ROI processing in GPGPU. In either of them, two levels of tests are carried out; in tile-level test, comparison test is performed for every tile after tile-based traversal in order to enhance the efficiency of rasterization by saving interpolation, which costs most of the time; in pixel-level test, comparison test is performed for every pixel after interpolation to reach high culling rate.

3.2 GPU Simulation Architecture

In this work, a scalable mobile GPU simulator is developed, and the ISA (instruction set architecture) is compatible with OpenGL ES 2.0 shading language. The GPU architecture is composed of a rasterizer, programmable unified cores, texture units, ROP, vertex/pixel scheduler, Configurable Memory Arrays (CMA) [36] and TCU as shown in Figure 3.3. The number of shader processors can be configured flexibly. The simulator is configured with eight unified cores for later experimental results.

Each unified core contains a 4-way vector processor that supports instructions in either vertex or pixel program. These instructions operating on one-dimensional array of data are beneficial for vector operations in graphics

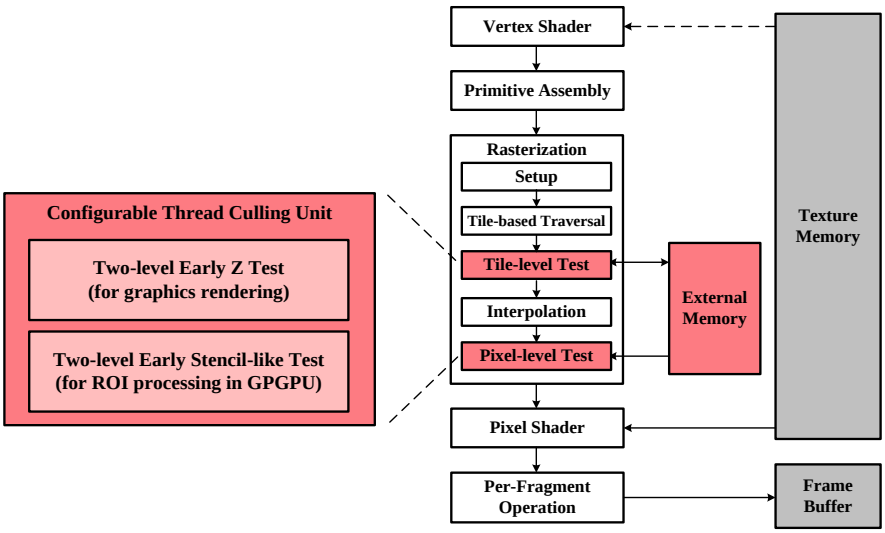


Figure 3.2: Pipeline overview with proposed TCU.

rendering. Different from CUDA architecture where several scalar threads are packed together as a warp, a vector thread is generated instead and then assigned to a vector processor in our simulation model. To balance heterogeneous workloads among unified cores, vertex/pixel scheduler is designed for coordinating operations among modules and dispatching threads to shader processors. Vertex scheduler fetches vertex data and stores into CMA, and then distributes vertex programs to the idle unified shader processors. CMA acts as local memory buffers and can be dynamically configured for storing input data or intermediate processed results, depending on how scheduler manages threads. As long as vertex threads of a primitive are completed, rasterizer is triggered to generate pixel threads from transformed vertex and meanwhile interpolates the attributes of each pixel thread. Tile-based rasterization [35] is adopted and proved to be efficient for enhancing the data locality in texture cache. It leads to bandwidth reduction for external texture memory access. Texture unit is an auxiliary module of shader processors, typically coupled with an L1 texture cache while supporting texture fetch with

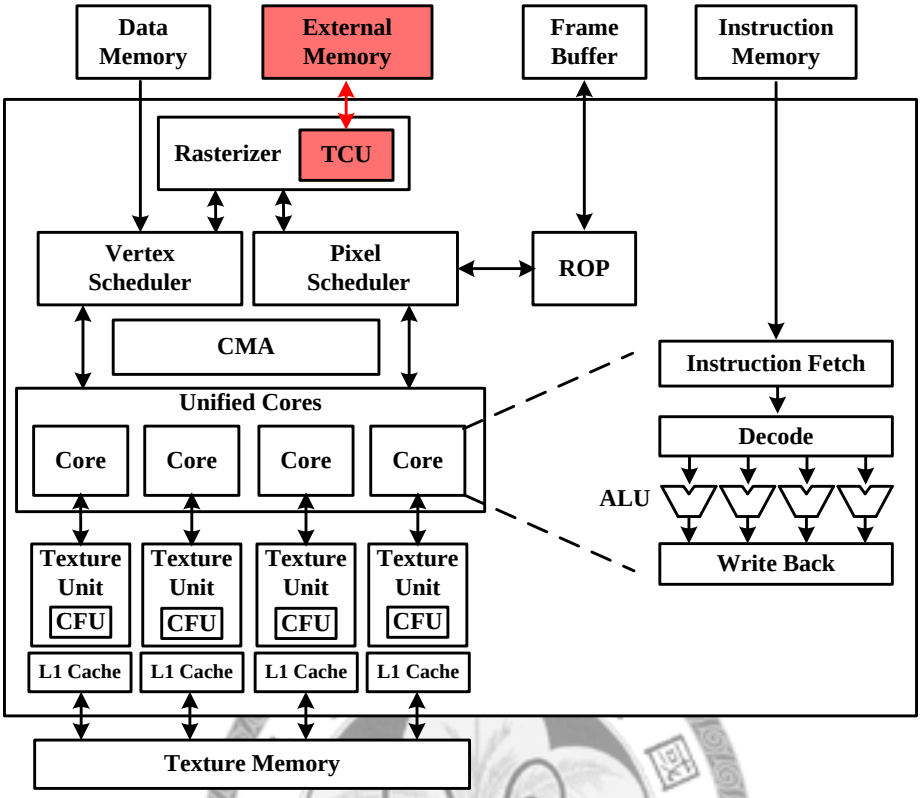


Figure 3.3: GPU simulation architecture.

specific filtering patterns. For each texture sample request, texture unit takes cycles to fetch required texels from the off-chip texture memory and calculate the filtered value. Similar to the vertex scheduler, pixel scheduler dispatches generated pixel threads to available shader processors. Once a shader processor finishes executing pixel program, ROP collects shaded pixel data for stencil/depth test, anti-aliasing and alpha/color blending. Post-processed pixels are then written out to the external frame buffer.

3.3 Two-Level Early Z Test

TCU is configured to perform two-level early Z test when rendering 3D scenes. Both Z-max and Z-min algorithms are exploited in our early Z test

to save shader processing and memory accesses.

3.3.1 Scheme

Two external memories are required for storing the maximum and minimum of Z-values for each tile, called **Zmax-buffer** and **Zmin-buffer** respectively. The size of **Zmax-buffer**/**Zmin-buffer** is that of the original Z-buffer downsampled by 8 and 4 on each side, with the consideration that the tile size is 8x4 pixels in our system. That is, a value in **Zmax-buffer**/**Zmin-buffer** represents the maximum/minimum Z-value in that tile region. In the beginning, the values on **Zmax-buffer** and **Zmin-buffer** are initialized with 0×FFFF (each Z-value is of 16 bits).

In our scheme, two levels of tests are performed. In tile-level test, each tile will go through Z-max and Z-min tests. A tile is discarded if the depth range it covers fails the Z-max test and a tile can skip Z read in ROP stage if the depth range it covers passes the Z-min test. In pixel-level test, every pixel will go through Z-max test only. A pixel is discarded if its depth fails the Z-max test. Z-min test is not implemented in this level in order to be consistent with the operation of ROP stage in our system. In order to hide memory latency, as long as there is a visible tile generated during rasterization, the memory access unit in ROP stage will be issued to fetch the corresponding Z-values from external Z-buffer, a tile at a time. Thus performing Z-min test for every pixel is actually unable to save Z reads in ROP.

The tile-level Z test contains several steps.

- Step1 : Find the maximum(*TileZmax*) and minimum(*TileZmin*) of the depth range a tile covers.
- Step2 : Fetch *StoredZmax*/*StoredZmin* from **Zmax-buffer**/**Zmin-buffer**
- Step3 : Perform Z-max/Z-min test.
- Step4 : Update **Zmax-buffer**/**Zmin-buffer**.

First, the depth range a tile covers should be found to perform Z-max test. More accurate *TileZmax* and *TileZmin* are, a higher culling rate at tile level could be achieved. Because a tile is more likely to be discarded if its depth range is narrower, a loose bound is easy to miss-culled tiles that are actually occluded. [20] calculates the intersections of tile and primitive, and then finds the extremums among interpolated Z-values of intersections. Extremums are accurate but the hardware cost is high. Instead of calculating intersections, [18] directly uses the the Z-values of vertices for partially-covered tiles, which is too coarse and leads to low culling rate. In this work, an intermediate method is proposed to make trade-off between cost and accuracy. For a fully-covered tile, as shown in Figure 3.4, *TileZmax* and *TileZmin* are found among the Z-values of four corners of the tile. The Z-values of four corners are interpolated by plane equations, which are usually processed in interpolation stage (Figure 3.1) but now executed earlier. On the other hand, for a partially-covered tile, *TileZmax* and *TileZmin* are found among the corners that are inside the primitive and the vertices of the edges that are intersected with the tile. It is simple to detect which edges are intersected with the tile by checking the sign bit of the edge equation corresponding to the corner position. This is usually processed in tile-based traversal stage (Figure 3.1) so no extra calculation is needed. Therefore the proposed method can achieve higher culling rate than [18] and lower hardware cost than [20].

After *StoredZmax* is fetched from **Zmax-buffer**, Z-max test is performed according to the conditions in Table 3.1. If a tile fails the Z-max test, indicating it is occluded by other objects due to its depth range falls behind *StoredZmax*, it is discarded from the pipeline. If a tile passes the update condition, its *TileZmax* will be updated to the **Zmax-buffer**. Since we need to ensure the value in **Zmax-buffer** is the maximum within the entire tile, only fully-covered tile can update. That is, if we update the Z-value of a partially-covered tile, other tiles in the following may be culled incorrectly. Here, the

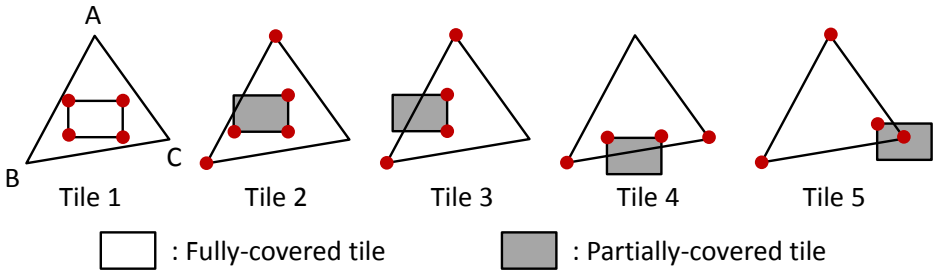


Figure 3.4: **TileZmax** and **TileZmin** are calculated from the selected candidates(red points) in different cases.

Table 3.1: Condition of tile-level Z test.

Z-max Test	Z-min Test
Pass condition : 1. **TileZmin** $\leq$ **StoredZmax**	Pass Condition : 1. **TileZmax** $<$ **StoredZmin**
Update condition : 1. Fully-covered tile 2. **TileZmax** $<$ **StoredZmax** 3. **TileZmin** ≥ 0	Update condition : 1. **TileZmin** $<$ **StoredZmin**

update mechanism is simple thus does not require additional management as in [21] where a cache was implemented.

If a tile passes Z-max test, Z-min test will be performed in the following by repeating the similar procedure as Z-max test does (Step2 $\sim$ Step4). Noted that the update condition is more instinctive since Z-min test is used to detect visible regions. Even if there is only one visible pixel in a tile with Z-value smaller than *StoredZmin*, its value should be updated to ensure the following Z-min tests for other tiles work correctly. The concept is quite different from that in Z-max test.

Table 3.2: Condition of pixel-level Z test.

Z-max Test	Z-min Test
Pass condition : 1. Interpoated Z-value $\leq$ **StoredZmax**	Not implement

The pixel-level test performs Z-max test (step3 in level one) only. At this stage, the Z-value of each pixel is already interpolated and directly used to check the pass condition in Table 3.2. No updates are required since fully-covered tile is detected at level one and updated at that time. Remind that Z-min test is not implemented in pixel-level test with the consideration of the operation in ROP stage. The complete algorithm of two-level early Z test is shown in Table 3.3.

3.3.2 Experimental Results

Test scenes are in Figure 3.5 with screen size of 512×512. Detection rate is calculated by Equation (3.1) and the results are in Figure 3.6(a). In average, 14.8% threads are culled by Z-max test and 15.0% threads are marked as visible by Z-min test. Besides, saved bandwidth is calculated by Equation (3.2) and the results are in Figure 3.6(b). In average, 13.6% bandwidth are saved by Z-max test and 19.6% bandwidth are saved by both Z-max test and Z-min test.

$$Detection\ rate\ (\%) = 1 - \frac{Number\ of\ total\ threads\ (after\ cull)}{Number\ of\ total\ threads\ (no\ cull)} \tag{3.1}$$

$$Saved\ BW\ (\%) = 1 - \frac{R/W\ of\ ROP\ +\ R/W\ of\ Early\ Z\ Test\ (after\ cull)}{R/W\ of\ ROP\ (no\ cull)} \tag{3.2}$$

Table 3.3: Pseudo code of two-level early Z test.

Tile-level Z Test :

```
01  For every tile covered by a primitive
02    Calculate TileZmax and TileZmin
03    Fetch StoredZmax from Zmax-buffer
04    Perform Z-max test
05    if (Pass condition of Z-max test in Table 3.1 is TRUE)
06    {
07      if (Update condition of Z-max test in Table 3.1 is TRUE)
08            Update TileZmax to Zmax-buffer
09      Transfer StoredZmax to pixel-leve Z test
10      Fetch StoredZmin from Zmin-buffer
11      Perform Z-min test
12      if (Pass condition of Z-min test in Table 3.1 is TRUE)
13      {
14            Mark current tile as visible
15            if (Update condition of Z-min test in Table 3.1 is TRUE)
16                  Update TileZmin to Zmin-buffer
17      }
18    }
19    else
20      Discard this tile
```

Pixel-level Z Test :

```
01  For every pixel generated from passed tiles
02    Perform Z-max test
03    if (Pass condition of Z-max test in Table 3.2 is TRUE)
04      ;
05    else
06      Discard this pixel thread
```

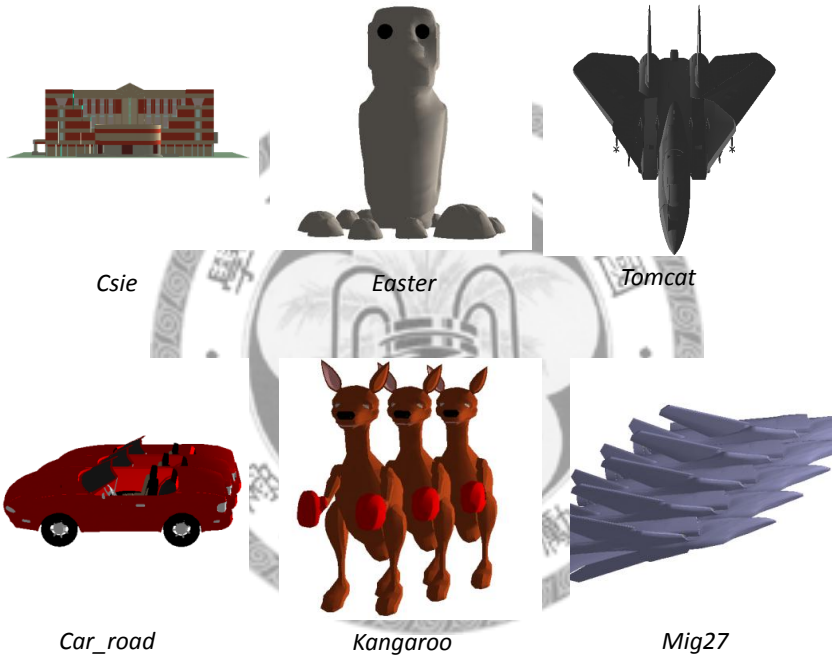
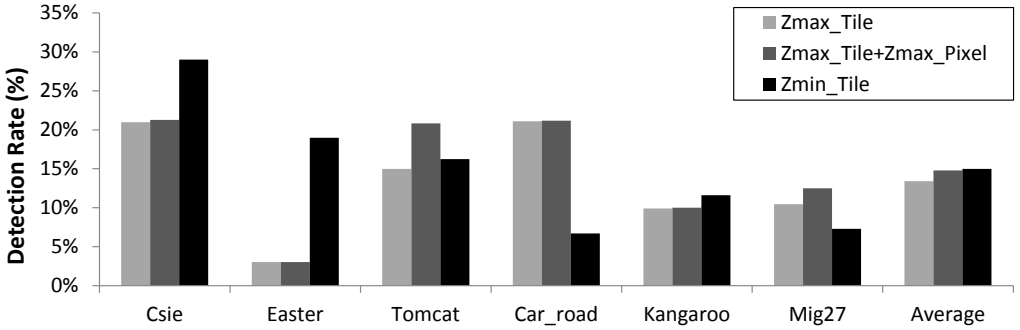
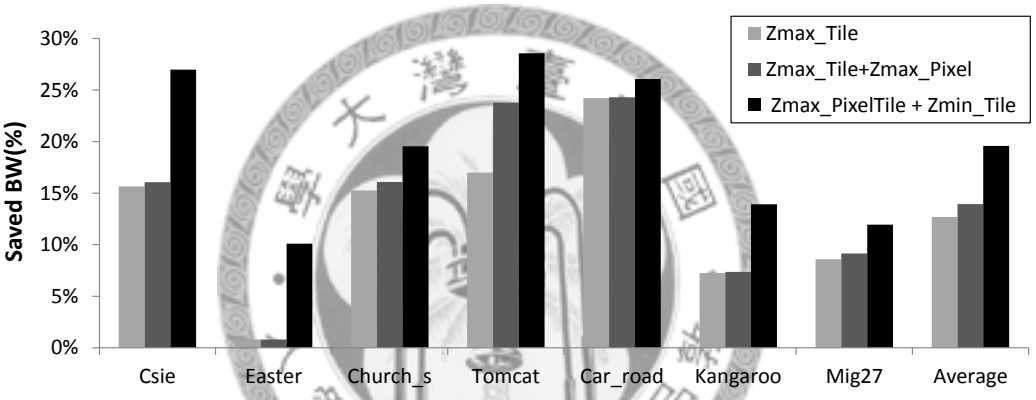


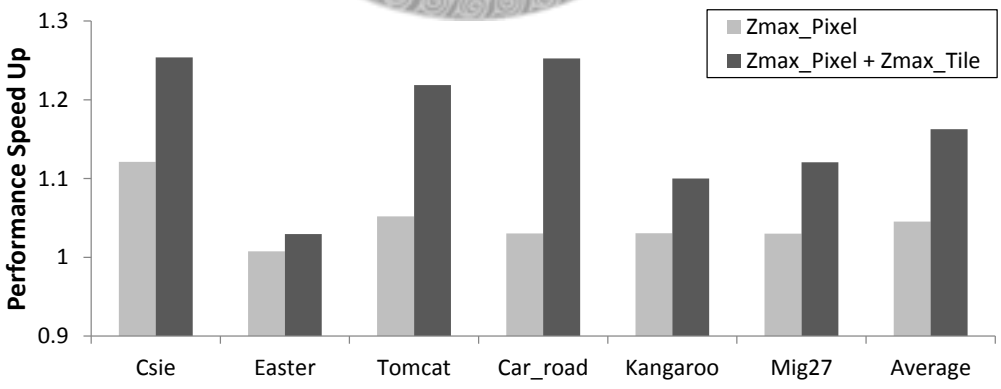
Figure 3.5: Test scenes.



(a) Detection Rate.



(b) Saved bandwidth.



(c) Performance speedup.

Figure 3.6: Simulation results of early Z test.

Finally, speedups are in Figure 3.6(c), normalized to the performance of no culling. Since Z-min test does not reject any pixel and results in subtle effect on the overall performance, only the Z-max test at pixel level and at both tile and pixel level are discussed. In average, 1.05× speedup is achieved by pixel-level Z test and 1.16× speedup is achieved by two-level Z test. The results show that the proposed two-level Z test outperforms pixel-level Z test by 1.1× by improving the efficiency of rasterization.

3.4 Two-Level Early Stencil-Like Test

Inspired by scissor and stencil test, TCU is configured to perform early stencil-like test to mitigate the overhead of ROI processing in GPGPU. Originally, scissor and stencil tests are dedicated for special effects in graphics rendering algorithms and are implemented in ROP stage. Stencil test can be used to determine whether a pixel should be updated or not with a per-pixel mask; and scissor test provides an additional clipping level by specifying a rectangular region idicating which pixels in the framebuffer are writable. Applying the same rendering concepts of these ROP tests to ROI processing, TCU is intended to reject invalid pixel threads in early stage. Since rasterization is the stage for generating pixel threads before shader execution, stencil-like culling can immediately intercept pixel threads and determine which pixels should be execute or not.

To illustrate the functionality of TCU, a simple program is taken as an example in Figure 3.7. The program contains a branch of *if-then-else*. In conventional graphics pipeline of SIMD architecture, branch divergence occurs among threads with *MaskValue* greater and less than *threshold*, respectively. Even if in MIMD architecture where jump instruction is adopted and thus no divergence occurs, redundant shader evocations are still issued. The per-pixel mask which consists of *MaskValue* is pre-determined and stored in external

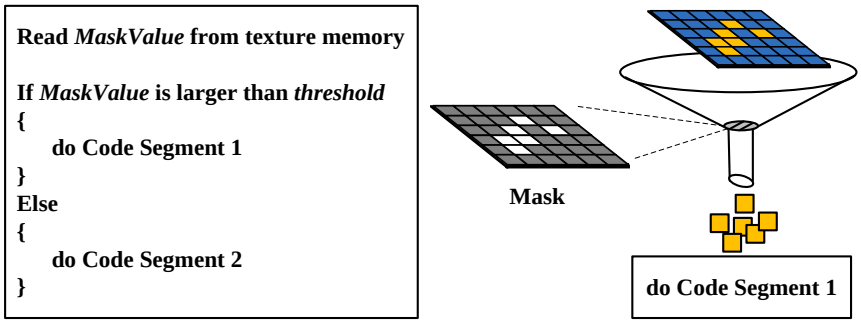


Figure 3.7: Program example. Left : Original shader program. Right : With early stencil-like test.

memory. It can be used in early stage to reject redundant threads. With the introduction of TCU, condition-failed threads are discarded, and condition-passed threads can execute the fraction of taken code segment. That is, the discarded threads would not be scheduled to be processed by shader cores while only condition-passed threads are scheduled to increase the hardware utilization.

Moreover, the external memory is accessed with a TCU cache. This centralized accessing scheme can increase cache hit rate and relieve the overhead of overlapped off-chip memory accesses when the branching operation is handled by the shader cores. Without TCU, threads are dispatched to several cores and *MaskValues* are accessed through texture operations among cores. In this way, the footprints of memory accesses are separated and overlapped, which is less efficient compared to centralized accesses in TCU. Therefore, TCU provides a simple yet effective approach to alleviate branching overhead in SIMD architecture and also reduce the redundant threads in MIMD architecture, too.

Table 3.4: API functions

glMaskImage *(GLenum format, GLsizei width, GLsizei height, GLenum type, const void* mask_values)*	
format	the format of the image data, can be: GL_RGBA GL_RGB GL_LUMINANCE_ALPHA GL_LUMINANCE GL_ALPHA
width	the width of the image in pixels
height	the height of the image in pixels
type	the type of the pixel data, can be: GL_UNSIGNED_BYTE GL_UNSIGNED_SHORT_4_4_4_4 GL_UNSIGNED_SHORT_5_5_5_1 GL_UNSIGNED_SHORT_5_6_5
mask_values	the array of pixel data for the image

glMaskFunc *(GLenum func, GLint ref_value)*	
func	the comparison function for early stencil-like test, can be: GL_EQUAL GL_NOTEQUAL GL_LESS GL_GREATER GL_LEQUAL GL_GEQUAL GL_ALWAYS GL_NEVER
ref_value	the comparison value for early stencil-like test

3.4.1 API Control

To be integrated to the framework of OpenGL ES 2.0, two new API functions are introduced for early stencil-like test to set hardware configuration as shown in Table 3.4.

Similar to *glTexImage2D* in OpenGL ES 2.0, *glMaskImage* function binds a user-defined mask and loads the mask into the external memory. Note that only one mask of storage is supported since it is adequate for most of the applications. In addition, if a mask value has more than one channel, GL_RGB for example, only the first component r will be valid for comparison test. The conditional constraint is then established by applying *glMaskFunc*, which is analogous to *glStencilFunc*. A comparison test is performed by fetching the mask value of corresponding position. Conditional comparison can be configured as in depth and stencil test, such as GL_EQUAL, GL_LESS, GL_LEQUAL, GL_GREATER, GL_GEQUAL, and GL_NOTEQUAL through setting the *func* parameter. Also, a reference value is set by the *ref_value* parameter as a threshold for comparison.

3.4.2 Scheme

To enhance the efficiency of rasterization, two levels of stencil-like tests are performed in our scheme. It is noted that for some images whose ROI occupies only a small fraction of total threads, many shader cores are idle and waiting for interpolated threads. That is, rasterization can not produce threads in time and thus becomes the system bottleneck. It costs a lot to enhance the processing power of rasterization on mobile devices which requires plenty of floating-point calculations. Thus the two-level culling is proposed to relieve the workload by early culling. In tile-level test, comparison test is performed on every tile. If a tile fails the user-defined constraint, it is then discarded and the interpolation effort is saved. In pixel-level test, comparison test is performed on every thread. Similarly, if a thread fails the user-defined

constraint, it is then discarded.

Two external memories are required for tile-level and pixel-level tests, called **LR-buffer** (low-resolution buffer) and **Ori-buffer**, respectively. **Ori-buffer** stores the per-pixel mask provided by users. **LR-buffer** stores the low-resolution image downsampled from the original mask by the factor of 8×8. Since a tile is of the size of 8×4 and the implementation for generating LOD of textures can be utilized, the factor of 8×8 is chosen. Downsampling is accomplished by calculating the maximum or minimum of 8×8 pixels in **Ori-buffer**, depending on the user-defined constraint. For example, the maximums of every 8×8 pixels are stored into **LR-buffer** if GL_GREATER or GL_GEQUAL is set when calling API functions. On the other hand, the minimums of every 8×8 pixels are stored if GL_LESS or GL_LEQUAL is set. The overall configuration for each API function is shown in Table 3.5. In this way, a value in **LR-buffer** represents the extreme value in the corresponding region, and thus can be easily used to perform tile-level test.

In tile-level test, mask values are fetched from **LR-buffer** for every tile. Similarly, in pixel-level test, mask value is fetched from **Ori-buffer** for every pixel. Comparisons are performed on fetched values and *ref_value* according to the user-defined constraint. Noted that, GL_NOTEQUAL performs only pixel-level test because the pass condition in tile-level comparison is not conservative with either max or min configuration of **LR-buffer**.

3.4.3 Experimental Results

To evaluate the performance of the proposed TCU performing early stencil-like test, Viola-Jones face detection and linear feature extraction in salient region are implemented for analysis. Results are compared to the methods of jump instructions, predicate execution and reduction program as mentioned in Chapter 2.2. Reminded that our GPU simulator (Chapter 3.2) generates vector threads and schedules them to four-way processors independently.

Table 3.5: The configuration of different functions. In LR-buffer, max or min indicates the downsampling method. A tile or pixel is culled if it fails the pass condition in the last two columns respectively.

func	LR-buffer	Tile-level pass condition	Pixel-level pass condition
GL_EQUAL	max or min	fetched value $\geq$ *ref_value* or fetched value $\leq$ *ref_value*	fetched value = *ref_value*
GL_NOTEQUAL	$\times$	$\times$	fetched value $\neq$ *ref_value*
GL_LESS	min	fetched value $<$ *ref_value*	fetched value $<$ *ref_value*
GL_GREATER	max	fetched value $>$ *ref_value*	fetched value $>$ *ref_value*
GL_LEQUAL	min	fetched value $\leq$ *ref_value*	fetched value $\leq$ *ref_value*
GL_GEQUAL	max	fetched value $\geq$ *ref_value*	fetched value $\geq$ *ref_value*
GL_ALWAYS	$\times$	all pass	all pass
GL_NEVER	$\times$	all fail	all fail

Then, a processor executes a single vector-thread at a time. Thus the program assembled with jump instructions are reasonably executed, which is different from that in CUDA architecture. Besides, in order to realize the overhead of predicate execution in those architectures which execute a bundle of threads in a lock step, assembly codes with predicate instructions are also generated in our architecture for analysis.

Finally, the solution by using reduction program is also considered for comparison. As shown in Figure 3.8, a mask image is processed by a reduction kernel first and valid pixels are written out in a continuos array. Then the face detection or Gabor filter kernel is issued. The processing time of the reduction program in Figure 3.9 is referenced from [1] which is realized in OpenGL. The results in [1] are measured using a GeForce 8800 GTS of 96 cores at 1.2GHz, so the processing time is scaled for fair comparison. Also, the processing time is scaled with the number of stream elements but with a little variation under different ratios of valid elements. For each test image, the processing time of the reduction kernel is referenced from Figure 3.9(b) according to their resolutions and ratios of valid elements. Noted that the second kernel is implemented in our system and the total processing time is accumulated from two kernels.

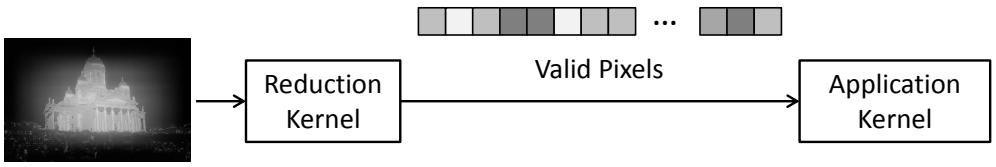
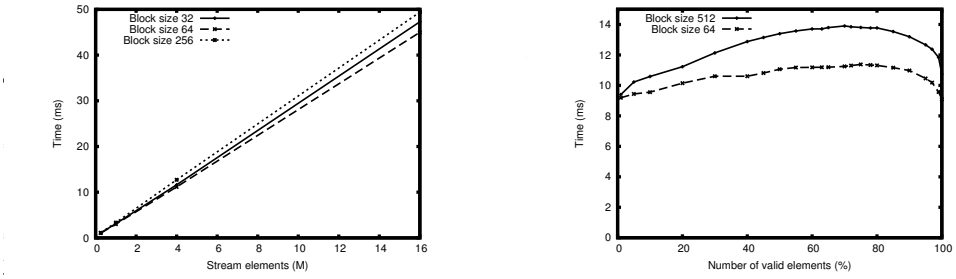


Figure 3.8: The operation flow of face detection or salient-region feature extraction with reduction program. Reduction kernel writes out valid pixels of a mask into an array. Then the application kernel is launched to process these valid pixels.



(a) Processing time with respect to the number of stream size.

(b) Processing time with respect to the ratio of valid elements in s stream of size 4M.

Figure 3.9: Processing time of the reduction program in [1].

3.4.3.1 Viola-Jones Face Detection

A well-known face detection framework, Viola-Jones face detection [14], requires great amount of computational efforts. This approach, as shown in Figure 3.10, utilizes a set of pre-trained Haar-basis functions as facial features. With the training process of AdaBoost, few to hundreds of features form a weak classifier within each stage, and multiple weak classifiers constitute an ultimate classifier called cascaded classifier. Each feature window of different sizes slides over the image and consequently determines if the potential candidates can pass the stage or not. Those passed candidates will be indicated as possible face positions for subsequent stages. Candidates can be gradually eliminated and the left ones after the last stage are detected as facial regions.

To achieve even higher performance, [37] has presented a thorough analysis on parallelizing Viola-Jones face detection framework using CUDA on desktops. Three different approaches to parallelize face detection are discussed: parallelizing computation by features, by scales and by windows, respectively. Parallelizing by windows performs better than the other two solutions. Thread for each feature window can independently examine face candidates through the image, which is suitable for parallel implementation

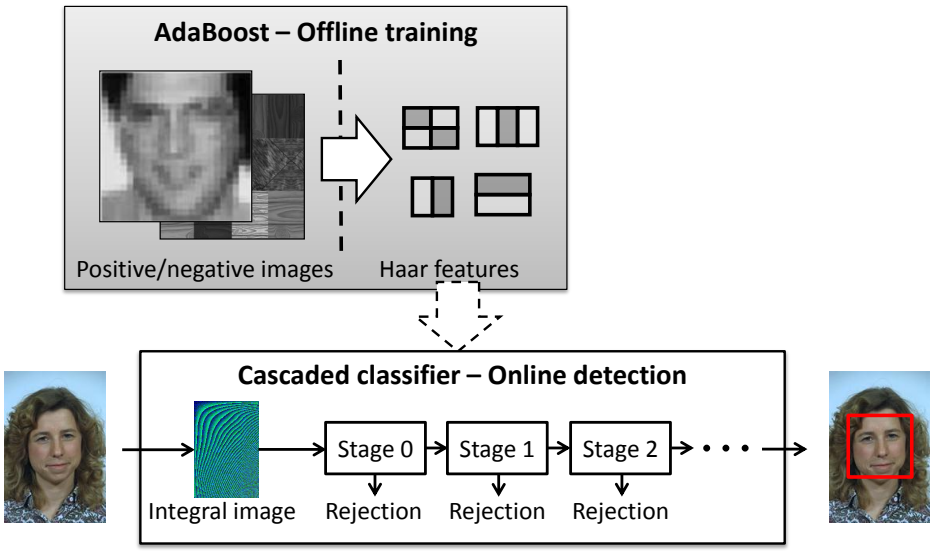


Figure 3.10: Viola-Jones face detection.

and optimization. However, these three strategies have the same problem of unbalanced load distribution among threads. Because most portions in an image are non-face regions, those regions would lead to idle threads. Synchronization problem makes the workload unevenly distributed and causes the resource waste, which leads to serious performance degradation for resource-limited mobile devices.

To evaluate the performance of face detection with our proposed TCU, face detection is implemented with OpenGL ES 2.0 shading language. Figure 3.11 illustrates the detecting procedure. In each stage, a Haar feature response is calculated for each remaining face candidates. Meanwhile, a mask is stored in the external memory, composed of those calculated feature values. For the next stage (next pass of shader code), a new kernel is launched and the mask image from previous stage is fetched by TCU. Comparison tests are executed according to the threshold value and pass condition of the previous stage to remove those non-face candidates.

Because of the small variation of the selected test images, the amount

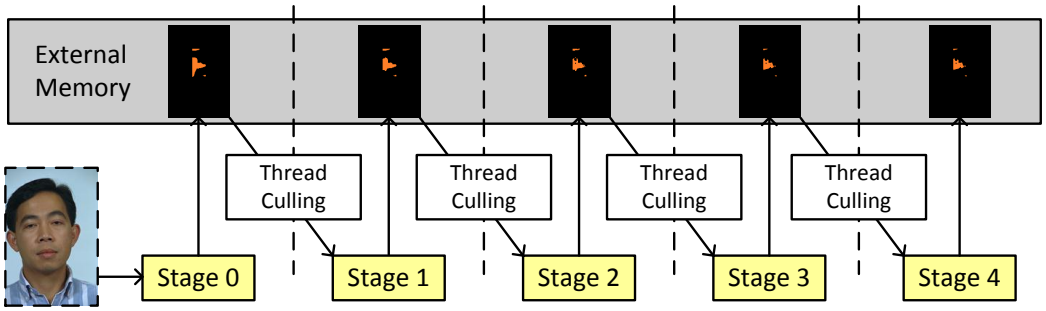
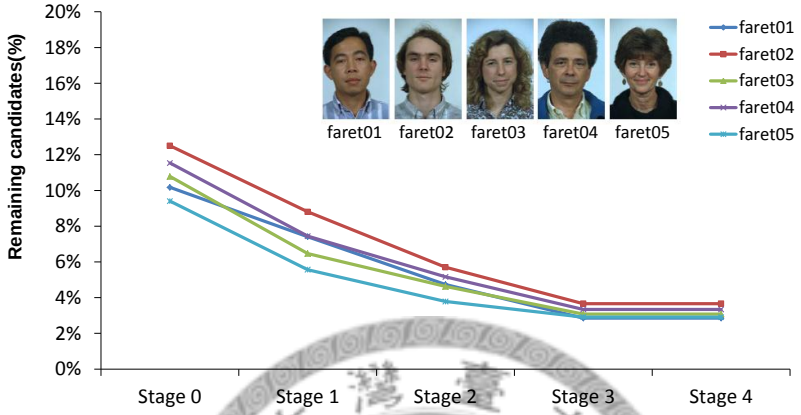
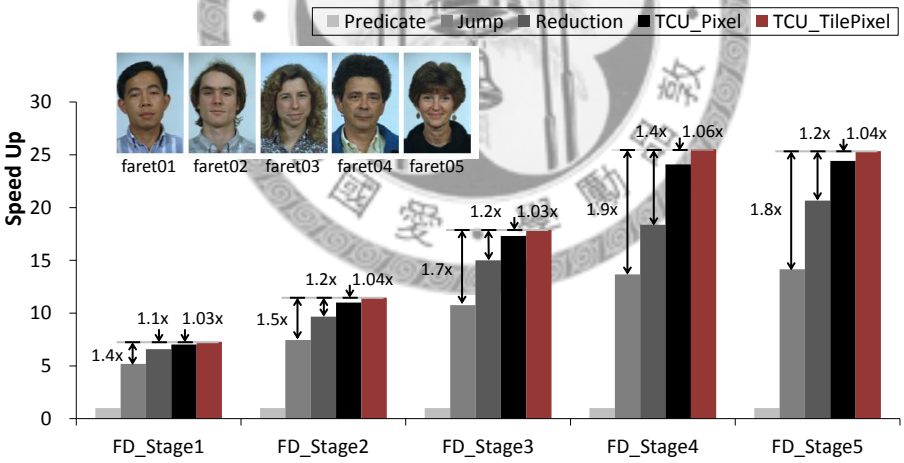


Figure 3.11: Operation flow of Viola-Jones face detection with proposed TCU.

of passed candidates of each stage do not have much difference. Therefore, the result is presented by the average of the face data set, which is chosen from [38], as shown in Figure 3.12. Figure 3.12(a) shows the ratio of remaining candidates after each cascaded stage. For most of the images, 90 percent of the candidates are rejected after stage1. Figure 3.12(b) illustrates the performance comparison of our proposed architecture with conventional predicate scheme, jump instruction and reduction. The performance speedups are normalized to the processing cycle of predicate scheme. It can be seen that in the best case, two-level TCU outperforms predicate by 25.5×, jump by 1.9×, reduction by 1.4× and pixel-level TCU by 1.06×. The shader program is quite complex for calculating feature values, especially in later stages where hundreds of Haar features are required. Although only a few threads are generated to cores, rasterizer can still catch up the speed that cores finish threads with. Thus the effort of two-level test is less obvious for this kind of application.



(a) Remaining candidates after each stage.



(b) Performance speedup of face detection.

Figure 3.12: Experimental results of Viola-Jones face detection.

3.4.3.2 Linear Feature Extraction in Salient Region

Salient regions indicate the positions human would pay more attention to. In such visually salient regions [13], feature information is more meaningful than those of unaware regions, for example, [39] employs this idea in object recognition. Therefore, unnecessary feature extraction and matching efforts can be prevented with saliency information. In addition, computational complexity can thus be reduced without sacrificing accuracy, which is beneficial for mobile devices. We turn this concept into a mobile GPGPU application for efficient processing, where TCU can use saliency map to perform thread culling. Gabor feature extraction is selected as the linear feature for performance evaluation.

Combining with TCU, the feature extraction flow in salient region is illustrated in Figure 3.13. A saliency map is provided and stored in the external memory, and then TCU can examine saliency with a user-defined threshold. Only the passed pixel threads are permitted to trigger kernel program of Gabor feature extraction.

The test data set and the normalized speedups for each image are illustrated in Figure 3.14. The percentage below each image is the ratio of saliency region with the mask value larger than 0.5. Take *mule* as an example, if the salient regions are revealed as sparsity and occupy small portion in the image, more processing efforts can be saved. In average, two-level TCU can achieve 6.3×, 2.1×, 3.1× and 1.4× improvement compared to predicate, jump, reduction and pixel-level TCU respectively. Cores tend to idle when running Gabor feature extraction with only pixel-level TCU since the program is simpler than that of Viola-Jones face detection. In this case, the two-level TCU provides remarkable enhancement in the efficiency of rasterization.

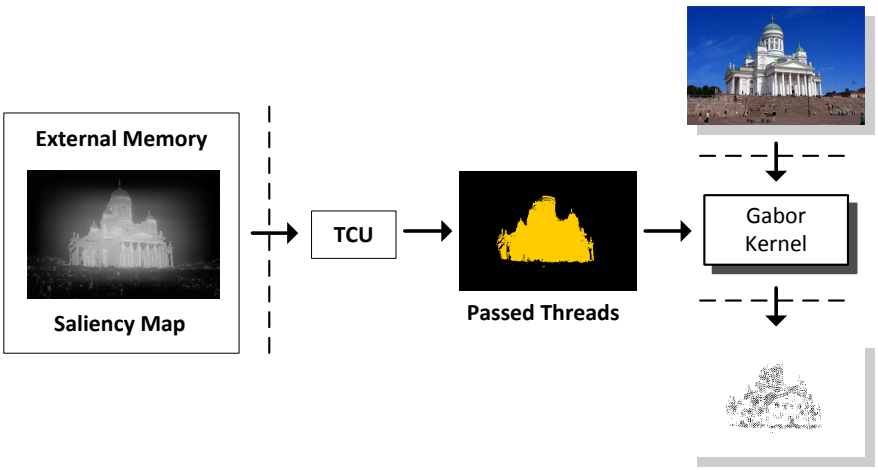


Figure 3.13: Operation flow of salient Gabor feature extraction with proposed TCU.

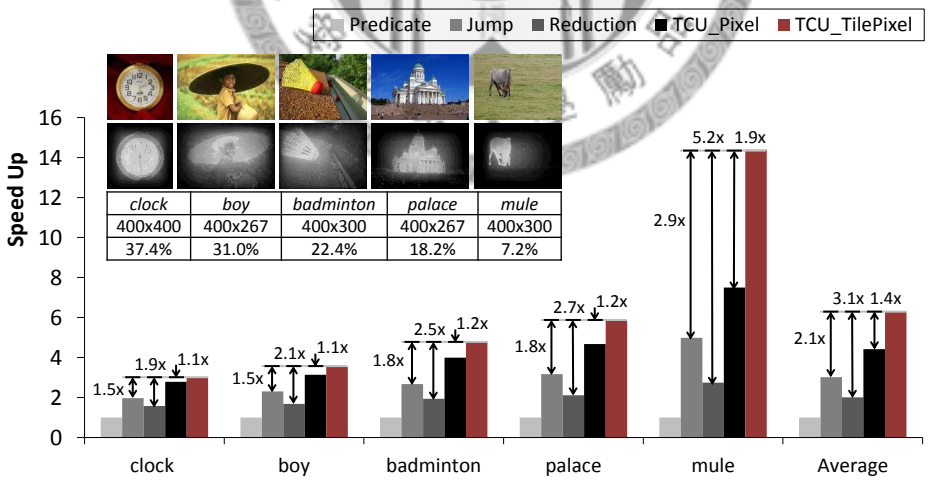


Figure 3.14: Test images and performance speed up of Gabor feature extraction.

Chapter 4

Hardware Analysis and Design of the Proposed Configurable Early Thread Culling Unit

4.1 Architecture

The proposed TCU is integrated into the rasterization unit with the architecture shown in Figure 4.1. The architecture contains level-one culling and level-two culling which are implemented after tile-based traversal stage and interpolation stage in rasterization, respectively. Moreover, two caches are allocated to reduce the overhead of accessing external memory. Through hardware sharing, two operating modes, early Z test and early stencil-like test, can be executed under different control schemes.

Figure 4.2(a) illustrates the active path for early Z test. Two caches are utilized to store Z-max and Z-min values. In level-one culling, Z-max test is performed by fetching depths through Z-max cache and Z-min test is performed through Z-min cache. After tests are performed, update modules write new depths to caches if update condition is passed. Then the fetched Z-max value is transferred to level-two culling directly to perform Z-max test

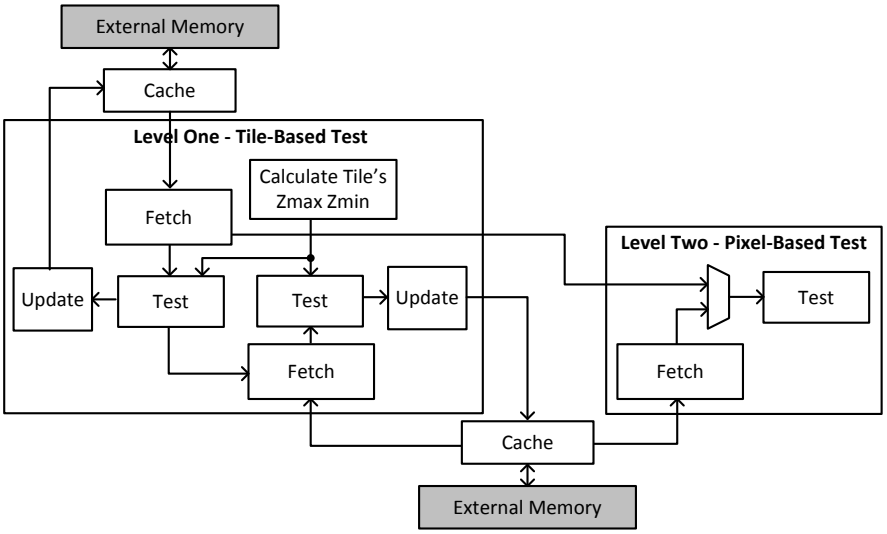
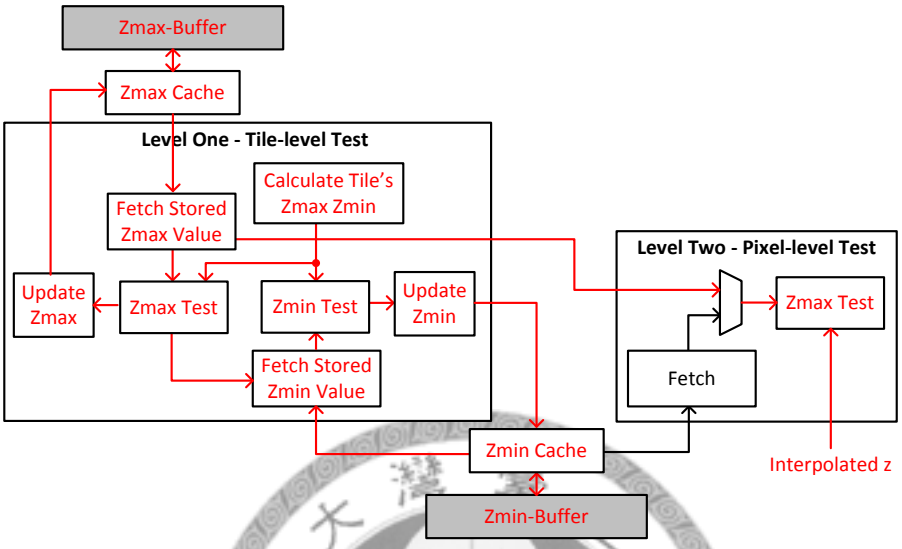


Figure 4.1: Configurable architecture of proposed TCU.

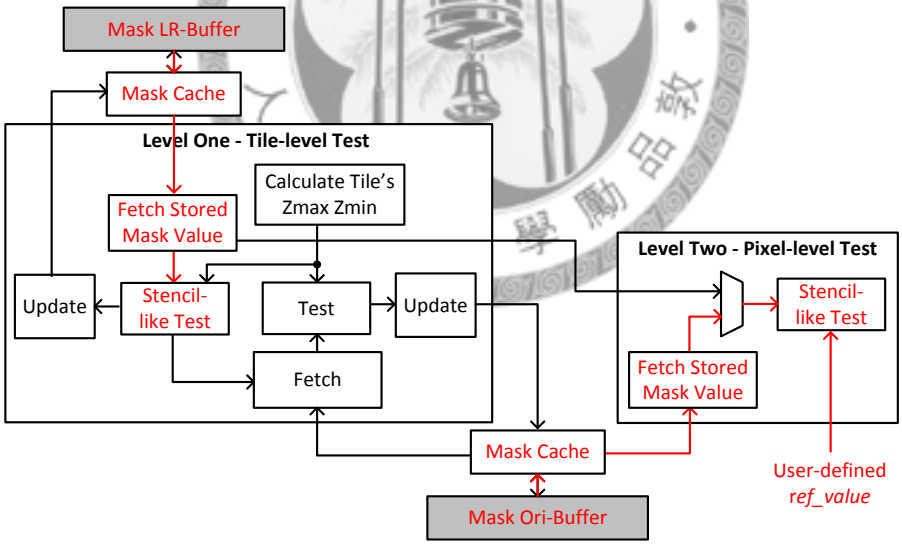
against interpolated depth for each thread.

Figure 4.2(b) illustrates the active path for early stencil-like test. One cache is used to store low-resolution mask for level-one culling and the other one is used to store original mask for level-two culling. The data path is less complex since there is unnecessary to calculate the tile values and write out cache to external memory, which is due to no update is carried out.

Beside, the functionality of ROP is modified for Z-min test where tiles that pass Z-min test can skip Z reads. In our system, there is a depth cache in ROP stage to enhance the performance of accessing external Z-buffer. A tile of depth values are fetched into cache at once. The management of depth cache is adapted to handle the situation when tile is coming with a must-visible flag. For example, if current tile is marked as visible, the depth values of this tile will not be loaded from external Z-buffer into cache. That is, the cache is filled with values of last tile. A 32-bit mask is thus created for this tile to indicate which positions are drawn by the coming pixels. When cache miss occurs, only the values in the cache with a positive bit can be written



(a) TCU in early Z test.



(b) TCU in early stencil-like test.

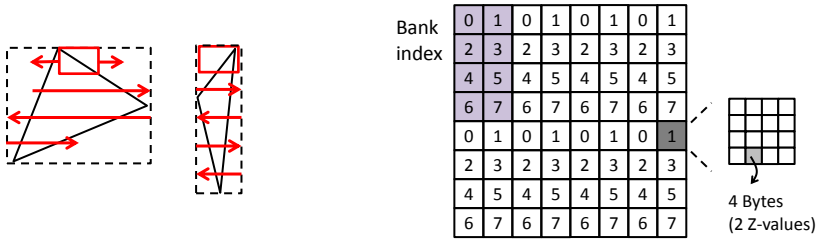
Figure 4.2: TCU architecture in different operating modes.

to the external Z-buffer to ensure the correctness. Since the emphasis is put on the implementation of rasterization, the implementation details in ROP stage will be omitted.

4.2 Cache Analysis

For early Z test, both caches are utilized in tile-level test, thus the path of tile-based traversal is taken into consideration for designing caching mechanism. In our system, tile-based traversal starts from the middle of top of the bounding box. It goes right first until current tile is already outside the primitive or bounding box, and then it goes left, terminating with the same condition. To the next row, it starts from the position corresponding to the final position on previous row and then follows the same direction, to the right and to the left, as shown in 4.3(a). It is noted that the geometry of primitive varies when scene changes. For some scenes, primitives are flat and wide; however for some others scenes, primitives are long and narrow. In order to benefit both cases, a 2D-blocked caching is exploited as shown in Figure 4.3(b). Considering the efficiency of burst requests to bus transfer, a bank is designed to contains 4×4 32-bit words, which is equal to 8×4 Z-values. All values in a bank are read or written at once when cache miss occurs. The analysis of hit rate is listed in Table 4.1. Cache size varies from 64 bytes to 8K bytes, indicating from 1 bank to 128 banks with the placement as in Figure 4.3(b). According to the simulating results, the size of 512 bytes with 8 banks is selected with the hit rate of 0.96 in average for each cache, thus 1K bytes with 16 banks in total are used.

For early stencil-like test, one cache is used in tile-level test and the other is used in pixel-level test. In a similar way, path of tile-based traversal is considered when designing the first cache. Generally, the primitives in GPGPU applications are usually wide as shown in Figure 4.4(a). Thus it is



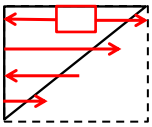
(a) Tile-based traversal in different primitives.

(b) Data caching in early Z test.

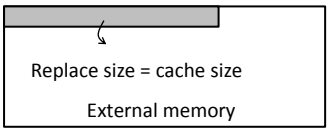
Figure 4.3: Cache design of early Z test.

Table 4.1: Cache hit rate of early Z test with the configuration in Figure 4.3(b).

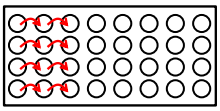
Render scene	Cache Size (Byte)							
	64	128	256	512	1K	2K	4K	8K
Csie	0.71	0.87	0.93	0.97	0.99	0.99	1.00	1.00
Easter	0.72	0.83	0.89	0.91	0.91	0.94	0.94	0.96
Tomcat	0.70	0.81	0.87	0.95	0.95	0.99	0.99	1.00
Car_road	0.73	0.90	0.97	0.98	0.99	0.99	1.00	1.00
Kangaroo	0.71	0.86	0.92	0.97	0.97	0.99	0.99	0.99
Mig27	0.76	0.85	0.96	0.99	0.99	0.99	1.00	1.00
Average	0.72	0.84	0.91	0.96	0.97	0.98	0.99	0.99



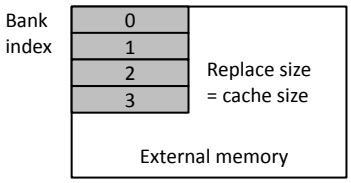
(a) Tile-based traversal.



(b) Data caching in level one test.



(c) Interpolation of a tile.



(d) Data caching in level two test.

Figure 4.4: Cache design of early stencil-like test.

adequate to cache data with simple indexing where a line is fetched at one time as in Figure 4.4(b). Besides, it is interesting to realize that mask values that have already undergone stencil-like tests are no longer used later. That is, each mask value is demanded at most once. The hit rate is influenced by how many data are fetched at one time. As a consequence, it is designed to read and replace all the values within a cache at a time. The analysis of hit rate is in Table 4.2. The cache size here also represents the amount of data that is read from external memory at a time. Although there is higher hit rate with larger cache size reasonably and the maximum size can be used is 512 bytes, which is determined before for rendering scenes, it costs longer latency since more data are requested. Thus the access latency of different cache size is also considered in Table 4.3. A simple latency model is adopted with 10 cycles of latency for the first data comes and 1 cycle of latency for the subsequent burst requests, as in Equation 4.1. After observing both hit rate and access latency, only 64 bytes out of 512 bytes with 1 bank out of 8 banks are used when performing tile-level stencil-like test. It is mentioned that, for different systems, more accurate latency model can be employed

and the cache size can be adapted to different considerations.

$$Miss\ latency\ (cycle) = (\ 10\ +\ Number\ of\ burst\ requests\ ) \times Number\ of\ transactions \tag{4.1}$$

For the other cache in pixel-level test, the data path of interpolation is considered. In our system, a column of pixels in a tile are processed simultaneously as in Figure 4.4(c). In order to provide mask values for four pixels at a time, a 4-bank cache is proposed as in Figure 4.4(d). Four banks of values are replaced at a time due to the same reason that mask values will not be demanded twice. The hit rate and the access latency are in Table 4.4 and Table 4.5, respectively. Recall that the cache size is equivalent to the replacing size. Based on the simulation results, 256 bytes out of 512 bytes with 4 banks out of 8 banks, are used when undergoing pixel-level stencil-like test.

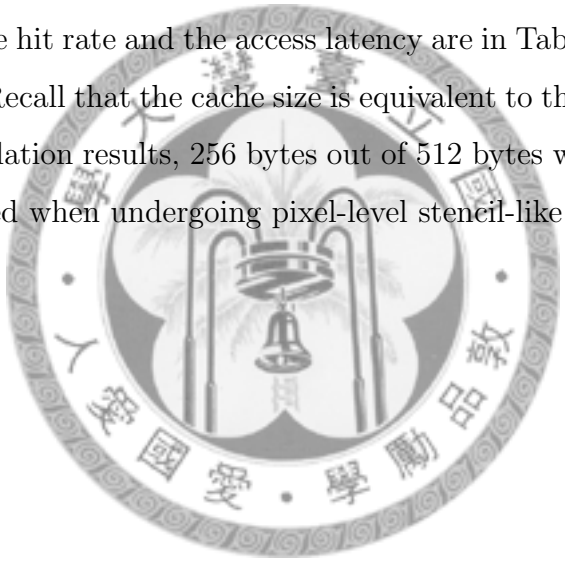


Table 4.2: Cache hit rate of level one early stencil-like test with the configuration in Figure 4.4(b).

Render	Cache Size (Byte)							
scene	16	32	64	128	256	512	1K	2K
clock	0.74	0.86	0.92	0.95	0.97	0.98	0.99	1.00
boy	0.74	0.85	0.91	0.94	0.97	0.98	0.99	0.99
badminton	0.74	0.86	0.92	0.95	0.97	0.98	0.99	1.00
palace	0.74	0.85	0.91	0.94	0.97	0.98	0.99	0.99
mule	0.74	0.86	0.92	0.95	0.97	0.98	0.99	1.00
faret01∼faret05	0.75	0.87	0.94	0.97	0.99	0.99	1.00	1.00
Average	0.74	0.86	0.92	0.95	0.97	0.98	0.99	1.00

Table 4.3: Access latency of level one early stencil-like test with the configuration in Figure 4.4(b)(cycle count/pixel).

Render	Cache Size (Byte)							
scene	16	32	64	128	256	512	1K	2K
clock	3.57	2.47	2.04	2.56	3.16	3.18	3.10	2.94
boy	3.67	2.61	2.23	2.88	3.56	3.62	3.62	4.34
badminton	3.64	2.57	2.17	2.78	3.27	3.46	3.46	3.46
palace	3.67	2.61	2.23	2.88	3.56	3.62	3.62	4.34
mule	3.64	2.57	2.17	2.78	3.27	3.46	3.46	3.46
faret01∼faret05	3.45	2.27	1.53	1.53	1.53	1.53	2.04	1.02
Average	3.61	2.52	2.06	2.57	3.06	3.14	3.21	3.26

Table 4.4: Cache hit rate of level two early stencil-like test with the configuration in Figure 4.4(d).

Render	Cache Size (Byte)							
scene	16	32	64	128	256	512	1K	2K
clock	0.75	0.87	0.94	0.97	0.98	0.99	1.00	1.00
boy	0.75	0.87	0.94	0.97	0.98	0.99	0.99	1.00
badminton	0.75	0.87	0.94	0.97	0.98	0.99	1.00	1.00
palace	0.74	0.87	0.94	0.97	0.98	0.99	0.99	1.00
mule	0.75	0.87	0.94	0.97	0.98	0.99	1.00	1.00
faret01∼faret05	0.75	0.87	0.93	0.97	0.98	0.99	0.99	1.00
Average	0.74	0.87	0.94	0.97	0.98	0.99	1.00	1.00

Table 4.5: Access latency of level two early stencil-like test with the configuration in Figure 4.4(d) (cycle count/pixel).

Render	Cache Size (Byte)							
scene	16	32	64	128	256	512	1K	2K
clock	11.10	6.07	3.54	2.23	1.64	1.70	1.84	1.97
boy	11.17	6.13	3.60	2.26	1.73	1.85	2.10	2.32
badminton	11.13	6.08	3.57	2.20	1.74	1.91	2.01	1.99
palace	11.23	6.16	3.63	2.25	1.72	1.96	2.21	2.95
mule	11.10	6.07	3.56	2.13	1.70	2.04	1.80	3.18
faret01∼faret05	11.20	6.14	3.66	2.11	1.77	1.96	2.21	1.89
Average	11.15	6.11	3.59	2.20	1.71	1.90	2.03	2.38

4.3 Verification Flow

Verification of the proposed TCU consists of several stages in Figure 4.5. Initially, the functionality is proved by the GPU simulator (Chapter 3.2) in C language. The simulator models the detailed hardware architecture with cycle-approximated accuracy. Then the hardware architecture is realized in Verilog. We use *VCS* as our simulator to perform functional verification and use *nWave* to view waveform behavior. After the functional verification, the architecture is synthesized with *Synopsys Design Complier* as in traditional ASIC design flow to evaluate the area overhead of the proposed TCU. Finally, through the processes of synthesis with *Synopsys Synplify*, and place and route with *Xilinx ISE*, the system is fulfilled in FPGA implementation.

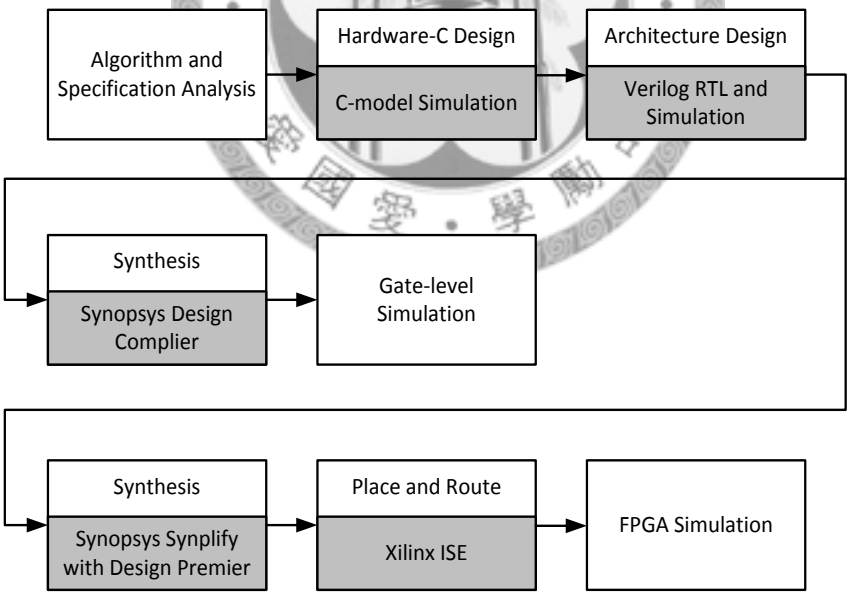


Figure 4.5: Verification flow.

4.3.1 Area Results

The specification of the GPU synthesized with Design Compiler is in Table 4.6. The GPU system is fabricated with TSMC 65nm technology and operated at 200MHz. The total area of GPU with proposed TCU is 4379265.03 um^2, where logic circuit occupies 1627170.86um^2(37%) and memory occupies 2752094.18 um^2(63%). 20 KBytes of on-chip SRAM are used in total.

The area of individual module with original version and that with TCU version are listed in Table 4.7. For operating the functionality of TCU, areas are increased in ROP and raster unit; and additional 1K bytes of cache are implemented in raster unit. From the results, TCU increases 3.9% of logic area, 4.7% of memory area and 4.2% of total area. After all, TCU introduces quite small area overhead into the entire system.

4.3.2 FPGA Implementation

We use Socle MDK-3D EVB as our emulating platform, which is a highly integrated, pre-verified and silicon-proven System-on-Chip (SoC) design platform. Its block diagram is shown in Figure 4.6. Two major features are mentioned here. First, L6021 is a 16/32-bit RISC microprocessor, which is designed to provide high performance processing for multimedia and general applications. Both ARM1176JZF and ARM926EJ processor cores are built-in and either one of them could be activated when operating system. Second, the on-chip FPGA, supporting Xilinx Virtex-5 XC5VLX330FF1760, can be used to verify the hardware design with programmable logic cells. IPs are communicated through the protocol of AHB/AXI bus.

To emulate the functionality of our GPU on the SoC platform, kernel

Table 4.6: System specification.

Technology	TSMC 65 nm
Working Frequency	200 MHz
Total Area	4,379,265.03 um^2
Logic	1,627,170.86 um^2
Memory	2,752,094.18 um^2
Gate Count (Logic)	1,129,979
Memory Usage	20,128 Bytes
Memory Type	8b×40b SRAM ×8
	16b×40b SRAM ×8
	32b×32b SRAM ×6
	32b×40b SRAM ×8
	32b×256b SRAM ×1
	96b×40b SRAM ×1
	128b×16b SRAM ×1
	128b×32b SRAM ×4
	128b×64b SRAM ×2
	128b×128b SRAM ×3
	128b×256b SRAM ×1

applications are programmed and executed by ARM1176JZF and GPU is fabricated in FPGA with required external memories loaded into DRAM. The software program in C language is built by the cross compiler to become compatible with ARM core under Linux operating system. Then the driver is carried out to configure control registers in GPU though the method of memory-mapped I/O (MMIO). Next, GPU is triggered to start the execution. In the system, GPU serves as both a master and a slave. Being a master IP, GPU will send requests to the bus for accessing external memories such

Table 4.7: Synthesis area of each module.

Modules		Area(um^2)	
		Original	With TCU
Texture unit		213373.08	213373.08
Scheduler		61659.36	61659.36
Shader cluster		676839.61	676839.61
ROP		323446.32	**332820.72**
Raster unit	Total	530465.40	**695413.08**
	TCU Cache	0	**101905.92**
Data fetch		95030.64	95030.64
Global register		60808.32	**61732.08**
CMA		2242396.46	2242396.46
Total		4204019.20	**4379265.03**

as instructions, texture images and color buffer, which are stored in DRAM in advance. Besides, GPU is also a slave IP for responding to the requests from the ARM processor. After execution is finished in GPU, an interrupt signal will be triggered to notify the ARM core. Finally, the rendered scene is displayed on the LCD screen.

The specification of GPU with FPGA implementation is shown in Table 4.8. The system is operated at 33MHz and 40K slices are utilized. In order to minimize the utilization of LUT, floating-point multiplication and addition are synthesized with DSP48E elements. Otherwise, the LUTs in FPGA can not afford the amount that GPU requires. Finally, the layout is in Figure 4.7.

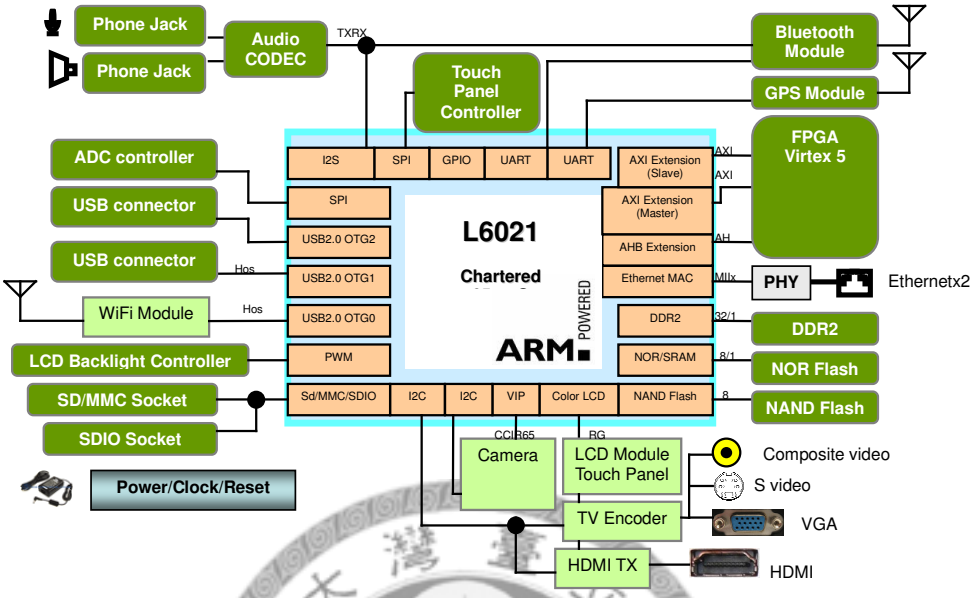


Figure 4.6: MDK-3D EVB system block diagram, captured from Socle document.

Table 4.8: Results of place and route in FPGA implementation.

	Specification	Utilization
Platform	Xilinx Virtex-5 XC5VLX330	
Working Frequency	33MHz	
DSP48Es	188	97%
Slices	40,873	78%
Slice Registers	38,780	18%
Slice LUTs	112,510	54%

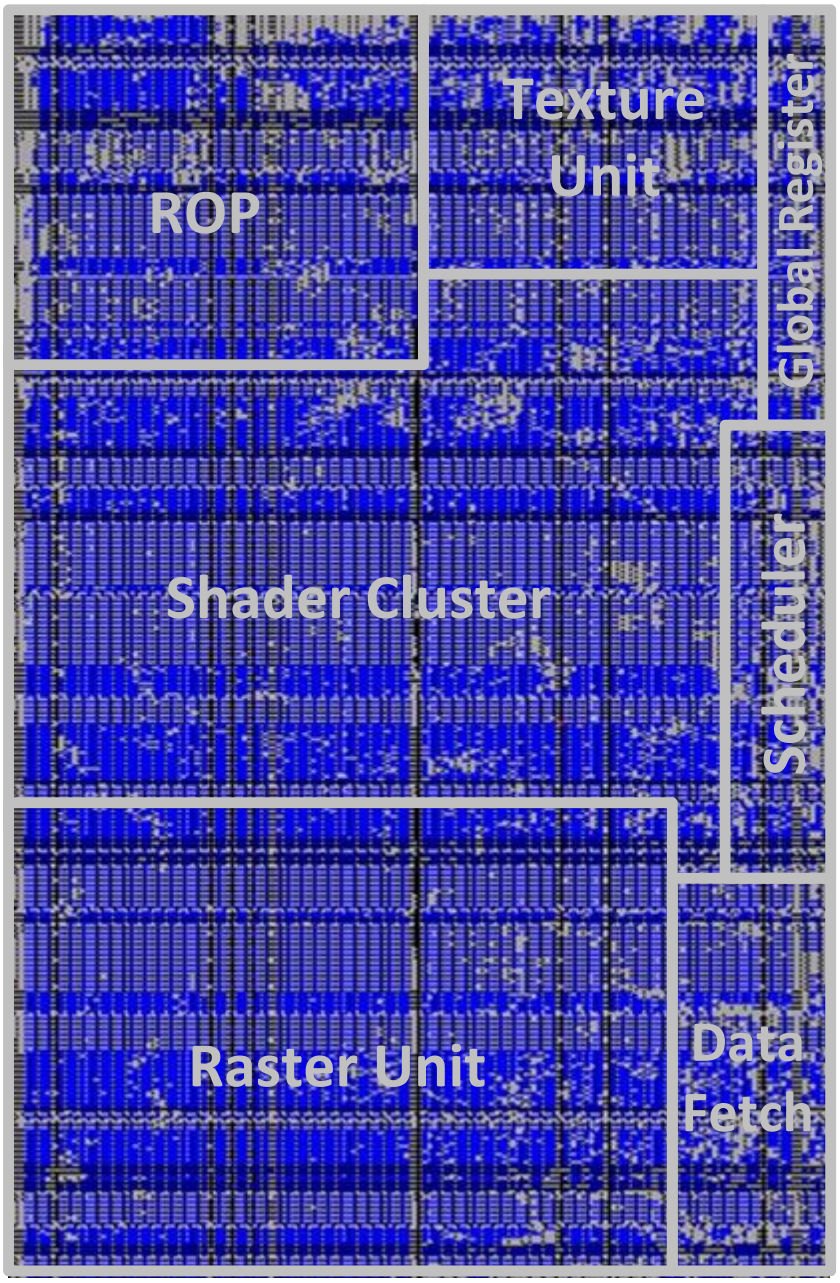


Figure 4.7: FPGA layout.


國立臺灣大學
人格教育・學生愛國

Chapter 5

Conclusion

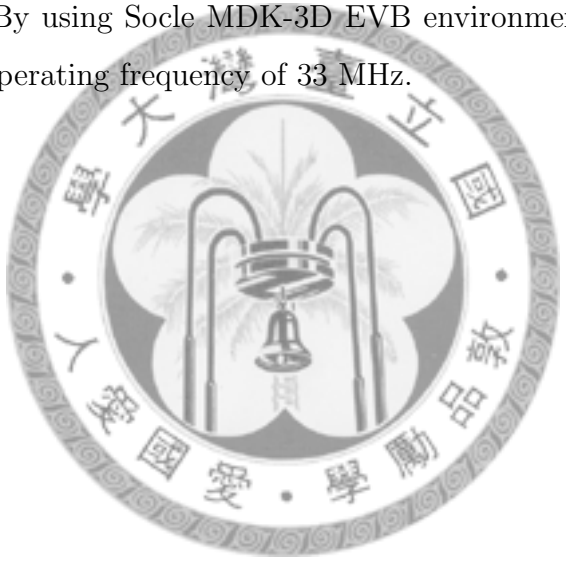
Considering the limited computing resources on mobile devices, we aim to enhance the processing ability of mobile GPU. It is noted that in traditional graphics rendering, many pixels are occluded by other objects in a scene, and will not be drawn on the screen. Computations are wasted for those unnecessary threads. In addition, general-purpose processing on GPU is popular and is suitable for regular applications where threads run the identical routine. However, for ROI processing in certain multimedia applications, different behavior between threads degrades the performance of GPU. Based on the observation, we proposed a configurable thread culling unit (TCU) to discard redundant threads in early stage and thus enhance the performance of GPU.

The proposed thread culling unit is configured to perform early Z test when rendering 3D scenes. In rasterization stage, Z-max test is executed at both tile and pixel levels to remove hidden pixels and Z-min test is executed at tile level to save Z reads for visible pixels. Experimental results show that 14.8% threads are rejected by Z-max test and 15.0% threads are marked as visible by Z-min test. Finally, speedup of 1.1× can be achieved by TCU compared to no culling mechanism.

In addition, thread culling unit is configured to perform early stencil-like test while processing ROI calculations in multimedia applications. Stencil-

like test is carried out according to the mask value at both tile and pixel levels. For Viola-Jones face detection, performance is improved by 25.5×, 1.9× and 1.4× compared to predication, jump and reduction program, respectively. Moreover, performance is enhanced by 6.3×, 2.1× and 3.1×, respectively for Gabor feature extraction in salient region.

Finally, the hardware implementation of the proposed configurable thread culling unit is integrated in a multi-core GPU system. The system is fabricated with TSMC 65nm technology working at 200 MHz, and the total area was 4,378,341.27 um^2. Through hardware sharing, the area overhead is less than 5% of the total area. Besides, the system is also verified on the FPGA platform. By using Socle MDK-3D EVB environment, 40,873 slices are consumed at operating frequency of 33 MHz.



Bibliography

[1] D. Roger, U. Assarsson, and N. Holzschuch, "Efficient stream reduction on the GPU," in *Proceedings of Workshop on General Purpose Processing on Graphics Processing Units*, Oct. 2007.

[2] `http://www.opengl.org/documentation/glsl/`.

[3] `http://msdn.microsoft.com`.

[4] `http://developer.amd.com/media/gpu_assets/Depth_in-depth.pdf`.

[5] D. A. VOORHIES, J. M. VAN DYKE, and J. E. MARGESON, III, "System, method and article of manufacture for Z-value and stencil culling prior to rendering in a computer graphics processing pipeline." Patent, 2006.

[6] S. D. TZVETKOV, "Early stencil test rejection." Patent, 2007.

[7] A. Kolb, L. Latta, and C. Rezk-Salama, "Hardware-based simulation and collision detection for large particle systems," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 123–131, ACM, 2004.

[8] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," in *ACM SIGGRAPH 2005 Courses*, ACM, 2005.

[9] K. Moreland and E. Angel, "The FFT on a GPU," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 112–119, 2003.

[10] `http://developer.nvidia.com/category/zone/cuda-zone`.

[11] `http://www.khronos.org/opencl/`.

[12] J. Leskela, J. Nikula, and M. Salmela, "OpenCL embedded profile prototype in mobile device," in *Proceedings of IEEE Workshop on Signal Processing Systems (SiPS 2009)*, pp. 279 –284, Oct. 2009.

[13] L. Itti, C. Koch, and E. Niebur, "A model of saliency-based visual attention for rapid scene analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, pp. 1254–1259, Nov. 1998.

[14] P. Viola and M. J. Jones, "Robust real-time face detection," *International Journal of Computer Vision*, vol. 57, pp. 137–154, 2004.

[15] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, pp. 91–110, 2004.

[16] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded up robust features," in *Proceedings of ECCV*, vol. 3951, pp. 404–417, 2006.

[17] N. Greene, M. Kass, and G. Miller, "Hierarchical Z-buffer visibility," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pp. 231–238, ACM, 1993.

[18] F. Xie and M. Shantz, "Adaptive hierarchical visibility in a tiled architecture," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pp. 75–84, ACM, 1999.

[19] T. Aila, V. Miettinen, and P. Nordlund, "Delay streams for graphics hardware," *ACM Transactions on Graphics*, pp. 792–800, 2003.

[20] C.-P. Chung, H.-W. Chen, and H.-C. Yang, "Blocked-z test for reducing rasterization, z test and shading workloads," *IEEE International Conference on Computational Science and Engineering*, vol. 2, pp. 402–407, 2009.

[21] C.-H. Chen and C.-Y. Lee, "Two-level hierarchical Z-buffer with compression technique for 3D graphics hardware," *The Visual Computer*, vol. 19, pp. 467–479, 2003.

[22] C.-H. Yu, D. Kim, and L.-S. Kim, "An area efficient early Z -test method for 3-D graphics rendering hardware," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 55, pp. 1929 –1938, Aug. 2008.

[23] T. Akenine-Möller and J. Ström, "Graphics for the masses: a hardware rasterization architecture for mobile phones," *ACM Transactions on Graphics*, vol. 22, pp. 801–808, Jul. 2003.

[24] Y.-M. Tsao, C.-L. Wu, S.-Y. Chien, and L.-G. Chen, "Adaptive tile depth filter for the depth buffer bandwidth minimization in the low power graphics systems," in *Proceedings of IEEE International Symposium on Circuits and Systems*, pp. 5023–5026, 2006.

[25] H.-Y. Kim, C.-H. Yu, and L.-S. Kim, "A memory-efficient unified early z-test," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, pp. 1286–1294, 2011.

[26] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 177–189, ACM, 1983.

[27] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proceedings*

of IEEE/ACM International Symposium on Microarchitecture, pp. 407–420, 2007.

[28] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proceedings of the 37th annual international symposium on Computer architecture*, pp. 235–246, ACM, 2010.

[29] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time KD-tree construction on graphics hardware," *ACM Transactions on Graphics*, vol. 27, Dec. 2008.

[30] N. Cornelis and L. V. Gool, "Fast scale invariant feature detection and matching on programmable graphics hardware," *Computer Vision and Pattern Recognition Workshop*, pp. 1–8, 2008.

[31] D. Horn, *GPUGems2 : Stream Reduction Operations for GPGPU Applications*. Addison-Wesley, 2005.

[32] S. Sengupta, A. Lefohn, and J. D. Owens, "A work-efficient step-efficient prefix sum algorithm," in *Proceedings of Workshop on Edge Computing Using New Commodity Architectures*, pp. 26–27, May 2006.

[33] M. Harris, J. D. Owens, S. Sengupta, S. Tzeng, Y. Zhang, A. Davidson, and R. Patel, "Cudpp : Cuda data parallel primitive library," 2007.

[34] M. Billeter, O. Olsson, and U. Assarsson, "Efficient stream compaction on wide SIMD many-core architectures," in *Proceedings of the Conference on High Performance Graphics 2009*, pp. 159–166, ACM, 2009.

[35] C.-H. Sun, Y.-M. Tsao, K.-H. Lok, and S.-Y. Chien, "Universal rasterizer with edge equations and tile-scan triangle traversal algorithm for graphics processing units," in *IEEE International Conference on Multimedia and Expo, 2009.*, pp. 1358 –1361, Jul. 2009.

[36] Y.-M. Tsao, C.-H. Chang, Y.-C. Lin, S.-Y. Chien, and L.-G. Chen, "An 8.6mW 12.5Mvertices/s 800MOPS 8.91mm2 stream processor core for mobile graphics and video applications," in *IEEE Symposium on VLSI Circuits, 2007*, pp. 218 –219, Jun. 2007.

[37] D. Hefenbrock, J. Oberg, N. Thanh, R. Kastner, and S. Baden, "Accelerating Viola-Jones face detection to FPGA-level using GPUs," in *IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM2010)*, pp. 11 –18, May 2010.

[38] P. Phillips, H. Moon, S. Rizvi, and P. Rauss, "The FERET evaluation methodology for face-recognition algorithms," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, pp. 1090 – 1104, Oct. 2000.

[39] S. Lee, J. Oh, J. Park, J. Kwon, M. Kim, and H.-J. Yoo, "A 345 mw heterogeneous many-core processor with an intelligent inference engine for robust object recognition," *IEEE Journal of Solid-State Circuits*, vol. 46, pp. 42 –51, Jan. 2011.