Texture Filter Memory – A Power-efficient and Scalable Texture Memory Architecture for Mobile Graphics Processors

B. V. N. Silpa*, Anjul Patney†, Tushar Krishna†, Preeti Ranjan Panda*, and G. S. Visweswaran†
*Department of Computer Science and Engineering
†Department of Electrical Engineering
Indian Institute of Technology Delhi
Hauz Khas, New Delhi 110016, India
Email: {silpa,panda}@cse.iitd.ac.in, apatney@ucdavis.edu, tkrishna@princeton.edu, gswaran@ee.iitd.ac.in

***Abstract*— With increasing interest in sophisticated graphics capabilities in mobile systems, energy consumption of graphics hardware is becoming a major design concern in addition to the traditional performance enhancement criteria. Among the different steps in the graphics processing pipeline, we have observed that memory accesses during texture mapping – a highly memory intensive phase – contribute 30-40% of the energy consumed in typical embedded graphics processors. This makes the texture mapping subsystem an attractive candidate for energy optimization. We argue that a standard cache hierarchy, commonly used by researchers and commercial graphics processors for texture mapping, is wasteful of energy, and propose the *Texture Filter Memory*, an energy efficient architecture that exploits locality and the relatively high degree of predictability in texture memory access patterns. Our architecture consumes 75% lesser energy for texturing in a fixed function pipeline, incurring no performance overhead and a small area overhead over conventional texture mapping hardware.**

I. Introduction

Traditionally, 3D graphics applications have been developed for either desktops or dedicated gaming consoles, but with enhancements in computational capacity of mobile platforms, graphics applications such as gaming, GPS-backed maps, screen savers, and animated chats emerge as possible applications for mobile devices. The challenge in porting complex 3D applications onto mobile platforms is posed not as much by performance as power consumption. Mobile devices are powered by battery and since battery capacity is not increasing on par with processing power of chips, the gap between the demand and supply of power is widening. This motivates us to focus on power optimizations in graphics processors for mobile devices. Modern graphics processors (GPUs) show a trend towards supporting general purpose computation, but mobile GPUs are not as general purpose as their desktop counterparts due to tighter constraints on power and area. Since our focus is on low-power options for embedded and mobile graphics, we only consider the graphics operations in the pipeline as opposed to general purpose ones.

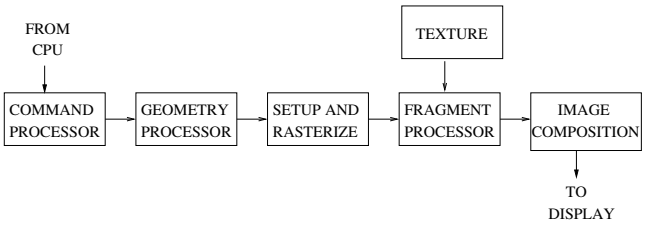

Fig. 1. Simplified Graphics Pipeline

The basic steps carried out in a graphics pipeline are summarized in Figure 1. A command processing unit receives the vertices representing the scene from the CPU, which first go through a *geometry processing* phase, in which they are first transformed from world space to eye space. The next step is to process the lighting of the vertices of the polygons in the scene according to the light sources present and reflections from various other scene objects. The part of the scene that falls out of the viewing frustum is then clipped off. At this stage, the entire scene is represented in terms of basic primitives - points, lines, and triangles. Corresponding to each pixel on the screen, there is a location in the *color buffer* that stores the color to be displayed at that pixel and a location in *depth buffer* (or *z-buffer*) that stores the depth of the pixel. These two buffers together form the *frame buffer*. The rasterization phase scan converts the primitives into a set of parallel horizontal lines called *scanlines*, which are further divided into discrete points (*fragments*), each representing a screen pixel. The fragments are then assigned a color by either interpolating the vertex colors or from a texture image stored in memory. The latter, called *texture mapping*, illustrated in Figure 2, which shows the modeling of a globe by wrapping an image of a world map (texture in Figure 2(b)) around a spherical object (Figure 2(a)). The texture mapped object is shown in Figure 2(c). As can be seen, this operation adds significant detail and contributes greatly to the realism of the final image. Adding such detail using only geometric primitives is both slower and inefficient. Hence, in most modern 3D applications texture mapping is

978-1-4244-2820-5/08/$25.00 ©2008 IEEE

(a) Object (b) texture (c) textured object

Fig. 2. Texture mapping example

one of the primary operations. The image composition stage concentrates on the task of compositing the final image which includes hidden surface elimination using depth-testing and the final transfer of the image to a display device.

A plot (Figure 3) of the energy consumption in different benchmarks at various stages of the graphics processor pipeline using Qsilver [1] shows that the texture mapping component is among the most power hungry, being responsible for 30-40% of the total energy consumption, which makes it an attractive candidate for optimization. This step is dominated by a large number of memory accesses, which account for most of the energy consumption. Published architectures for texture mapping typically employ a standard cache hierarchy. In this paper we argue that a standard cache hierarchy is wasteful of energy, and propose and evaluate a memory architecture that is customized to exploit the typical reference patterns occurring during texture mapping. Since this step is a significant contributor to graphics processor energy, the optimization is a promising candidate for reducing overall energy in 3D graphics-enabled mobile devices.

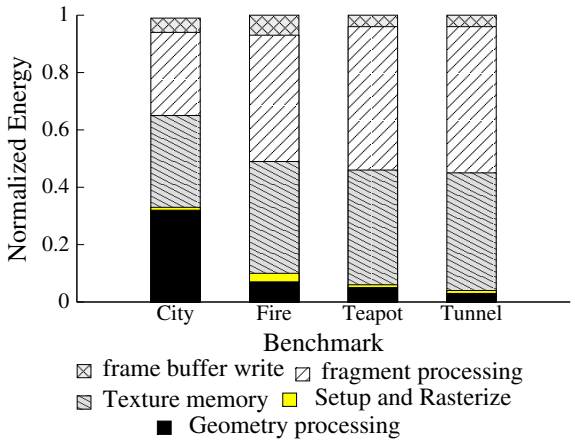

Fig. 3. Fraction of energy consumed in texture unit

II. Related Work

Related research can be classified into two major classes: texture memory architectures and power optimizations in multimedia and graphics processing.

Since texture mapping is a memory intensive step, in [2], the authors aim to reduce the memory bandwidth requirement of texture mapping and hence achieve improved performance by using a blocked representation of the texture map (Section III) and introducing a cache between the Texture memory and fragment generator. This was based on the observation that texture mapping exhibits a good amount of spatial and temporal locality. The L1 cache, due to its small size, only handles intra primitive locality and not inter triangle or inter frame locality. Hence in [3] the authors suggest an external texture cache (L2 cache) between the internal L1 cache and the texture memory. They organize the L2 cache as a virtual memory, with a mechanism to translate from texture addresses to physical addresses. In [4] the authors suggest a mechanism to pre-fetch textures into the cache in an attempt to hide memory latency. Igehy et al. [5] studied the benefits of texture caching in a parallel architecture. Serial rasterization architectures benefited from texture cache because of locality of accesses. But in an architecture with parallel rasterization units with local texture caches, this locality decreases. Hence they propose a shared texture memory architecture for effective bandwidth utilization by avoiding duplication of textures. In [6] the authors evaluate the effect of three hybrid access cache systems: victim cache, half-and-half cache and cooperative cache on conflict misses. They observed that for an 8KB cache, victim cache performs better that the others but for a 16KB cache the performance of victim cache and half-and-half cache are comparable.

In the above works, the objective was performance enhancement and no specific attention was paid to power and energy consumption. A few research efforts to reduce power in the graphics system have been reported recently. In [7], the authors suggest replacing the Booth multiplier in the MAC unit of the pipeline with a low power asynchronous multiplier. In [8], the authors propose the processing of the difference of adjacent pixel values instead of directly on the pixel values. Spatial correlation in typical images lead to the difference being a small number on an average. Similar tonal locality observations are also exploited in [9], [10] to assign codes in order to reduce total bit transition count during serial transmission over the Liquid Crystal Display (LCD) bus to the LCD display device. In [11], [12], the authors observe that the eye's visual perception depends on the intensity and the transmittance characteristic of the LCD panel. It is possible to adjust these parameters without affecting the perception quality; specifically, it is possible to reduce the intensity and thereby reduce power. Usage of fixed point arithmetic instead of floating point has been suggested for low power implementations of graphics processors in mobile applications [13]. Low power features in such implementations include using the *valid instruction* signal in the vertex shader to clock-gate the register files, preventing writes (reads can

still proceed). In [14], [15], [16], the authors observe that Graphics applications show a significant variation of workload with resolution, level of detail, lighting and texture mapping and hence employ dynamic voltage and frequency scaling for optimizing power.

In [17], the authors reduce power consumption by replacing the 16KB 2-Way associative cache with a very small (128-256 Byte) direct mapped cache. This reduces average power as the direct-mapped cache avoids several tag comparisons of set-associative caches, but at a considerable performance penalty (50%) due to high miss rates. In this paper, we propose a texture memory architecture that attempts to improve the power consumption characteristics of architectures such as [2], [17].

III. Texture Mapping Access Pattern

Texture mapping is the process of mapping an image (in texture space) on to a surface (in object space) using some mapping function [18]. Since texture space and object space could be at arbitrary distance and orientation with respect to each other, there is no one-to-one correspondence between the pixels on the object and texels of the texture. This necessitates the use of some texture filtering mechanism to attribute the best color to a pixel.

Bilinear filtering and *Trilinear filtering* [18] are two most common filtering techniques used for texture mapping. In bilinear filtering the weighted average of four texels nearest to the pixel center gives the color of the pixel (Figure 4). In order to produce good results for various levels of depth (*lod*) at which the object could be viewed, the texture image is stored at various resolutions called *mip-maps* [18] and the nearest one is picked up for filtering at run time based on the *lod*. In trilinear interpolation, the bilinearly interpolated values from the two nearest mip-map levels are averaged to give the color of the pixel.

Texture mapping with bilinear filtering exhibits high spatial and temporal locality. This is because:

- to compute the color of a pixel we need to fetch four neighboring texels
- consecutive pixels on the scan line map to neighboring texels, and
- consecutive scanlines of a primitive share texels.

In addition to locality, texture mapping also exhibits predictability in access pattern. As seen in Figure 4, access to texel t1 is followed by accesses to texels t2,t3 and t4. Thus access to texel t1 gives us the information about the accesses to the next three pixels. A conventional 4-way associative cache architecture for texture memory as suggested in [2] is oblivious to such predictability in accesses. The four required reads from the tag and data arrays in addition to the four tag comparisons for each texel fetch makes it very power hungry. We propose a customized memory architecture that exploits spatial locality and predictability in the access stream resulting in a low power solution without compromising on performance.

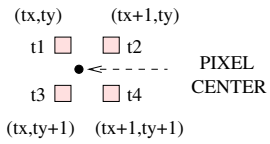

Fig. 4. Footprint of a Bilinear filter

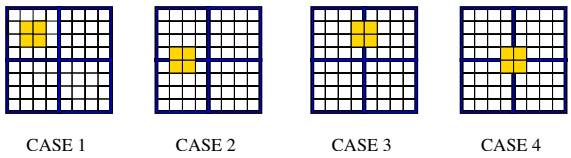

Fig. 5. Scenarios to which the footprint could be mapped

Since the direction of accesses in texture memory is arbitrary, a blocked representation of texture maps in memory is generally used (illustrated in Figure 6). Each block resides in contiguous memory space. The overhead of block address computation is offset by the performance gained by the reduced cache miss rates by selecting the line size equal to the block size [2].

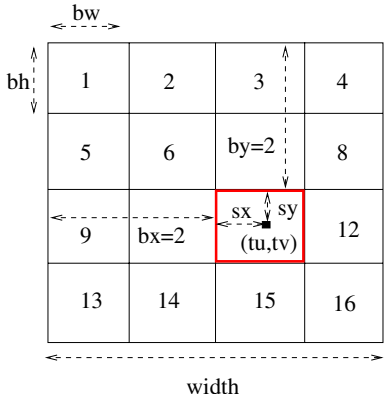

Fig. 6. Blocked representation of texture

Since texture mapping exhibits high spatial locality, we propose to buffer the blocks of texture expected to be accessed in near future in a set of registers. The number of blocks to be buffered depends on the type of filter being used. In a bilinear filtering operation, the texels could be in one, two, or four of the neighboring blocks as shown in Figure 5. Also, the next set of texels could fall in one of these four blocks with a high probability. Hence we would require to buffer upto four blocks of texture. For trilinear filtering we would need to buffer eight blocks – four blocks from each of the two nearest mipmap levels.

A standard cache based memory architecture could be used for the texture memory accesses, but this is expensive in terms of power, as each access results in a lookup operation where both tag and data arrays of the cache are read, with the number

Algorithm 1 Kernel For Bilinear Filtering

Input: Texel Co-ordinates (tu,tv), Base - Starting address of Texture

Compute texel addresses corresponding to texel co-ordinates (tu,tv), (tu+1,tv), (tu,tv+1) and (tu+1,tv+1)

for $I = 1$ to 4 **do**

  $texelI \leftarrow$ CacheLookup(texeladdressI)

$color \leftarrow$ WeightedAverage($texel1, texel2, texel3, texel4$)

return color

of such memory accesses proportional to the associativity (lower power cache architectures exist, but they compromise on the performance). The predictability of the texture access pattern can be used to reduce the average number of memory accesses. We propose a novel memory architecture for textures, where cache-style lookups are minimized. The information about which of the four cases in Figure 5 applies to a texture access, can be obtained by comparing the block co-ordinates of the texels. If the accesses belong to case 1, where all the texels are mapped to same block, a lookup for texel 1 could be followed by fetching the texels 2, 3, and 4 from the same block. Thus, we need only one lookup for fetching four texels. Similarly for cases 2 and 3, two lookups are sufficient for four texel accesses. Only case 4 requires four lookups. For this to be possible, our buffering unit should be designed such that it allows both lookup operation like a cache and a direct register access.

We modified the conventional kernel for bilinear filtering as shown in Algorithm 1 to the one shown in Algorithm 2 so as to take advantage of the above buffering mechanism.

Though we use two additional comparison operations for classifying the accesses to different cases, at the same time we reduce the number of block address computations and eliminate the texel address computations. From our experiments on various benchmarks we observed that on an average, 58% of the accesses are to the same block (case 1), 36% to two blocks (cases 2,3) and 6% of the times to four of the blocks (case 4). Thus, only one lookup is required in 58% of the texture accesses. Even though this single lookup can consume the same power as in an associative cache (though smaller in magnitude because our buffers have only 4 registers), the remaining three accesses do not require any lookup/comparison operation because the register containing the block is already known. On an average, the number of memory accesses and comparisons is drastically reduced.

IV. Architecture of Texture Filter Memory

In a conventional texturing unit, the address generator computes the block address and the offsets of the four texels to be bi-linearly filtered. The four texels are fetched from the cache by the fetch unit and the filtering unit performs the bi-linear interpolation. There are two filtering units so that during tri-linear interpolation the bilinear filtering of both the mipmap levels could be done in parallel. We include a Texture Filter Memory (TFM) in the texturing unit which acts as an

Algorithm 2 Modified Bilinear Texture Filtering

Input: Texel Co-ordinates (tu,tv), Base - Starting address of Texture

Output: Color

$bx \leftarrow tu >> log_2(bw)$

$by \leftarrow tv >> log_2(bh)$

$bx1 \leftarrow (tu+1) >> lbw$

$by1 \leftarrow (tv+1) >> lbh$

$c0 \leftarrow (bx = bx1)?\ 0 : 1$

$c1 \leftarrow (by = by1)?\ 0 : 1$

Calculate offset1, offset2, offset3 and offset4

if $c0 = 0$ *and* $c1 = 0$ **then**

  compute block address1

  $texel1 \leftarrow$ LookupBuffer(block address1, offset1)

  Read texels 2,3 and 4 from the same block

else if $c0 = 0$ *and* $c1 = 1$ **then**

  compute block address1 and block address 3

  $texel1 \leftarrow$ LookupBuffer(block address1, offset1)

  Read texels 2 from the same block

  $texel3 \leftarrow$ LookupBuffer(block address3, offset3)

  Read texels 4 from the same block

else if $c0 = 1$ *and* $c1 = 0$ **then**

  compute block address1 and block address 2

  $texel1 \leftarrow$ LookupBuffer(block address1, offset1)

  Read texels 3 from the same block

  $texel2 \leftarrow$ LookupBuffer(block address2, offset2)

  Read texels 4 from the same block

else

  compute block addresses of all the four texels

  for $I = 1$ to 4 **do**

    $texelI \leftarrow$ LookupBuffer(blockaddressI, offsetI)

$color \leftarrow$ WeightedAverage($texel1, texel2, texel3, texel4$)

return color

interface between the texturing unit and the texture memory. TFM consists of three components (Figure 7): (i) Texture Buffer Array, (ii) Address Comparators, and (iii) Controller.

A block of texture consists of 4×4 texels. Hence we need a buffer of 16 registers (called *Texture Buffer Array* – TBA) to save one block of texture and an array of four buffers to store four blocks of texture for bilinear filtering and eight buffers to support trilinear filtering also. We arrange the eight buffers as two sets of four buffers each. For bilinear filtering we use only one of the sets, and turn off the other in order to reduce power. In trilinear filtering, we map the two sets to the two different mipmap levels. By doing so, each texel lookup would require us to search four buffers instead of eight. The address bus width of the TBA is 7 bits wide: one bit to select between two of the sets, two bits to select the buffer in a set, and four bit offset into the buffer.

TFM has two blocks of address comparators, each associated with a set of buffers in the TBA. A comparator block consists of four registers to store the addresses of the blocks

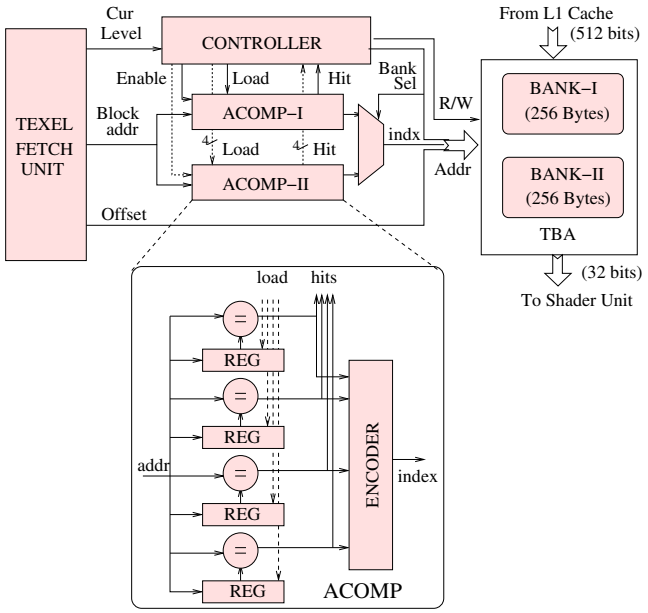

Fig. 7. TFM architecture

present in the buffers of that set. When the address comparator receives an address, it compares it with the addresses saved in the four registers in parallel. The output of the comparators are sent to the controller and also encoded to give the address of the buffer in which the texture block resides in case of a hit.

The Texel fetch unit provides the block address, the offset of the texel along with the mipmap level from which the texel is to be fetched. The access to TFM could be: (i) a direct access when the fetch is to the same block as the previous texel (ii) a lookup when it is not known in which of the buffers the texel would reside. The texel fetch unit determines the type of each access and provides this information to the controller. In case of a lookup, the controller enables the appropriate address comparator and the TBA entry. For bilinear filtering, only one of the banks and comparator associated with it are enabled. For trilinear filtering, the controller compares the mipmap level of the current access with that of the previous one and when it changes, toggles comparator enable and bank select signals. Thus the texels of two different mipmap levels would always be mapped to two different sets reducing the interference.

The controller combines the outputs of the address comparator to determine if the lookup resulted in a hit or a miss. Upon hit, the buffer address generated by the comparator is registered so that for the successive access to the same block it is sufficient to provide only the offset and the costly comparisons can be avoided. When there is a TBA miss, the controller issues a read signal to the L1 cache and a block of texture is moved from L1 cache to TBA. From synthesis of TFM, we observed that access time to the TFM is about half that of the cache. Thus we can fill the buffer in two cycles when there is a miss, and still stay within an L1 cache access time.

V. Experiments and Results

We describe in this section our experiments conducted to validate our proposed architecture on typical rendering examples used for evaluating graphics hardware and algorithms. We have developed a trace driven simulator for the proposed architecture. We have instrumented Mesa code (software renderer for the graphics pipeline [21]) for generating the memory traces. We developed a synthesizable VHDL model for the design and used Synopsys Design Compiler and Synopsys PrimePower [24] for synthesis and power/energy simulation. We have also used CACTI models [22] for estimating the energy of caches and SRAMs in the designs.

A. Evaluation of the TFM architecture

The overall energy of a texture memory architecture depends on two main factors:

- Hit rates to the lower levels of the hierarchy
- Access energy of the corresponding levels

In the case of our proposed architecture, the hit rates are lower than those obtained using a large L1 cache, but the energy per access is much smaller. The overall energy for TFM is the lowest among the evaluated architectures.

1) Hit rate comparison:

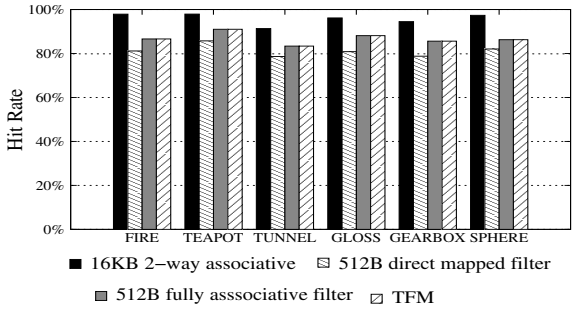

Fig. 8. Hit rate comparison

Figure 8 shows the hit rate into TFM and compares it with the hit rates obtained by the following architectures:

1) The conventional 16KB, 2-way set associative cache as L1 and 256KB 4-way set associative L2 cache
2) 512B direct mapped cache as L1 and 256 KB L2 cache
3) 512B direct mapped filter cache along with L1 and L2
4) 512B fully associative filter cache along with L1 and L2
5) Texture Filter Memory along with L1 and L2

Case 1 above is the original proposal in [2]. Case 2 is the architecture proposed in [17] and cases 3-4 are variants of the filter cache proposal in [23]. Cases 2 and 3 have the same hit ratio for the lowest level of cache hierarchy, so are shown only once in Figure 8. We observe that for 80% of the accesses we would hit into the TFM, which is significant in comparison to 96% hit rate into the L1 cache. Since the majority of the accesses would be to the smaller TFM rather than the cache, we achieve a significant energy saving. We find that the TFM

gives about 4.5% better hit rate than the direct mapped cache of same size used as a filter and equals the miss rate for a fully associative cache of the same size, at much lower comparisons per access than the fully associative cache.

2) Access energy comparison: Comparing the access energy of various architectures for texture memory (Figure 9), we observe that by buffering the texels in the texturing unit we obtain significant reduction in the average access energy. We also observe that average access energy when TFM is used is 25% less than using a direct mapped cache of the same size and 60% lesser than using a fully associative cache at a lower level of hierarchy (below L1). We see that using a direct mapped cache instead of 2 way associative cache , does not result in reduction in power due to lower hit rate. An architecture with TFM consumes about 77% lower energy than the conventional architecture. The energy overhead due to the extra computation in TFM has been included in the reported energy numbers.

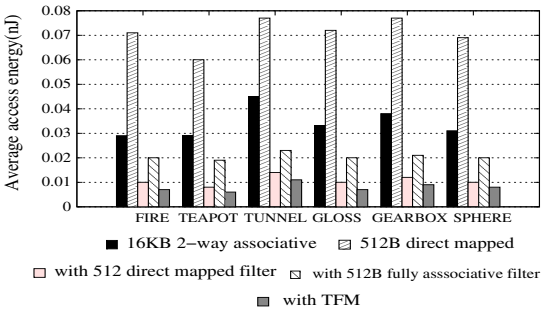

Fig. 9. Average Access Energy

3) Access time and area comparison: A comparison of the access times of the different architectures shows that TFM performs better than the other architectures (with the 512B direct mapped filter coming closest) in terms of access time also because of the simpler circuit, hence presents no performance overhead for the rest of the pipeline. An area comparison shows that adding the TFM would incur an area overhead of 0.48% over the conventional texture memory (L1: 16 KB, 2-way; L2: 256 KB, 4-way).

VI. Conclusion and Future Work

In addition to high spatial and temporal locality, texture mapping in the graphics rendering pipeline also exhibits considerable predictability with respect to the memory access pattern. We have utilized these properties of texture mapping to build a low power memory architecture for texture mapping. By buffering the blocks of texture in registers, we have replaced the high power cache lookups with low power register reads. We have proposed a smart lookup mechanism to maximize the hits into the registers with a small number of arithmetic operations. Our architecture consumes 75% lesser energy than existing texture cache architectures in a pipeline with single texturing unit. Thus, for pixel texturing, we claim that our proposed architecture consumes lower energy than those reported in literature, with no performance overhead, and insignificant area overhead.

Currently our architecture only supports pixel texturing. The vertex texturing phase [18] can not take advantage of our proposed architecture as it does not exhibit similar spatial locality. In the future we plan to investigate ways to adapt our architecture so as to be able to reduce power for vertex texturing also.

Acknowledgment

We gratefully acknowledge the contributions of Chaitanya Rajguru, Shabbir Topiwala, and Kalyan Bhairavabhatla from Intel towards this work. This work was partially supported by a research grant from Intel.

References

[1] J. W. Sheaffer, et al. A flexible simulation framework for graphics architectures. ACM EUROGRAPHICS conference on Graphics hardware, pp 85–94, 2004.
[2] Z. S. Hakura and A. Gupta. The design and analysis of a cache architecture for texture mapping. *ISCA*, pp 108–120, 1997.
[3] M. Cox, et al. Multi-level texture caching for 3D graphics hardware. *ISCA*, pages 86–97, 1998.
[4] H. Igehy,et al. Prefetching in a texture cache architecture. ACM EUROGRAPHICS workshop on Graphics hardware, pp. 133–ff, 1998.
[5] H. Igehy, et al. Parallel texture caching. ACM EUROGRAPHICS workshop on Graphics hardware, pp. 95–106,1999.
[6] C. J. Choi, et al. Performance comparison of various cache systems for texture mapping. *HPCA*, 1:374–379 vol.1, 2000.
[7] K. P. Acken, et al. Architectural optimizations for a floating point multiply-accumulate unit in a graphics pipeline. *ASAP*, 0:65, 1996.
[8] U. Zangi and R. Ginosar. A low power video processor. *ISLPED '98*, pp. 136–138.
[9] W.-C. Cheng and M. Pedram. Chromatic encoding: A low power encoding technique for digital visual interface. *DATE* , pp.10694–10699, 2003.
[10] S. Salerno, et al. Limited intra-word transition codes: an energy-efficient bus encoding for lcd display interfaces. *ISLPED '04*, pages 206–211,2004.
[11] Y.-Y. Chang, et al. Design and analysis of low-power cache using two-level filter scheme. *IEEE Trans. VLSI*, 11(4):568–580, 2003.
[12] A. Iranli and M. Pedram. Dtm: dynamic tone mapping for backlight scaling. *DAC '05*, pages 612–617, 2005.
[13] J.-H. Sohn, et al. A 155-mw 50-m vertices/s graphics processor with fixed-point programmable vertex shader for mobile applications. *IEEE Journal of Solid-State Circuits*, 41(2):1081–1091, May 2006.
[14] Y. Gu, S. Chakraborty, and W. T. Ooi. Games are up for DVFS. *DAC*, pages 598–603,2006.
[15] B. Mochocki, et al. Power analysis of mobile 3D graphics. *DATE '06*, pages 502–507, 2006.
[16] B. Mochocki, et al. Signature-based workload estimation for mobile 3D graphics. In *DAC '06* , pages 592–597, 2006.
[17] I. Antochi, et al. Trading efficiency for energy in a texture cache architecture. *Euromicro conference on Massively-parallel computing systems*, April 2002.
[18] T. Möller and E. Haines. *Real-time rendering*. A. K. Peters, Ltd., Natick, MA, USA, 1999.
[19] C.-L. Su and A. M. Despain. Cache designs for energy efficiency. *ISLPD*, 1995.
[20] http://www.patentstorm.us/patents/7069387-claims.html.
[21] http://www.mesa3d.org/.
[22] S. J. E. Wilton, et al. An enhanced access and cycle time model for on-chip caches. *HP Labs Technical Report*, WRL-93-5, 1994.
[23] J. Kin, et al. The filter cache: An energy efficient memory structure. In *International Symposium on Microarchitecture*, pp. 184–193, 1997.
[24] http://www.synopsys.com