THESIS TITLE

M.Tech Project Stage 1 Report

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Gazula Srinivas Bapaiah Naidu
(Roll No. 203079007)

Under the guidance of
Prof. Virendra Singh

Department of Electrical Engineering
Indian Institute of Technology Bombay
October 2022

Acknowledgement

I express my gratitude to my guide Prof. Virendra Singh for providing me the opportunity to work on this topic.

Srinivas Gazula
Electrical Engineering
IIT Bombay

Abstract

XX.

Contents

List of Figures

Chapter 1

Introduction

GPUs are being used to implement many algorithms and applications which are highly irregular/unstructured and data intensive. Graph algorithms which are highly irregular and data intensive can be executed efficiently on GPU. The reasons for low execution efficiency of graph algorithms on GPUs are control flow divergence, non-uniform work distribution and large number of memory accesses with divergence.

The GPU performance is effected by the low utilization of the resources. Many applications often does not use all the available registers There are several unused physical registers in GPU[1].

The texture cache is an essential component of modern GPUs and plays an important role in achieving real-time performance when generating realistic images. The texture cache is a read-only cache. It stores the image of the texture which has to be applied over other surfaces. This image is written once but read several times to perform *texture mapping*[2].

Chapter 2

Literature Survey

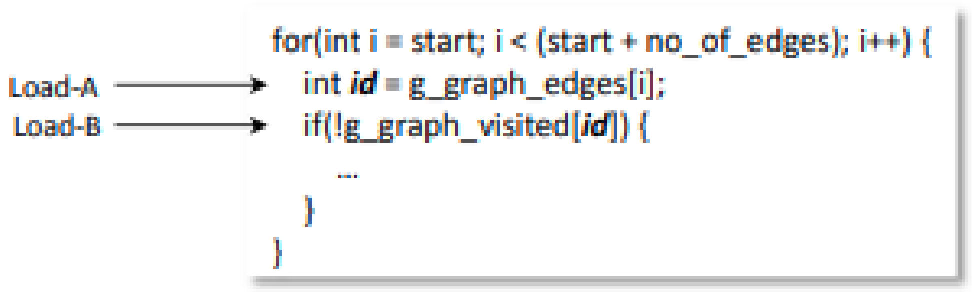
2.1 Prefetching for Graph Algorithms

The highly irregular memory accesses and data-dependencies in graph algorithms will increase the latency. Prior prefetching mechanisms use caches to store prefetched data. To improve the performance we can optimize cache hierarchy to improve the performance in CPU[3]. But in GPUs there will be problem of early eviction. Due to excessive threads, the amount of cache available per thread is smaller than that of a CPU. Hence Spare Registers can be used to prefetch. In a GPU, there are two types of spare registers:

1. *Unallocated registers:* Even though there are abundant registers, registers will be allocated to a thread block only if all the required resources are available. Hence there will always be some unallocated registers.

2. *Terminated registers:* If a thread inside a warp is terminated, its registers can be allocated to other threads in the warp.

The unallocated type of spare registers are be used to prefetch the data. The target loads are shown in the figure1.1. Load-A is dependent on i, which increments in each iteration of *for* loop. Therefore load-A has strided access. The load-B depends on Load-A. Hence Load-B is a random access.



```
for(int i = start; i < (start + no_of_edges); i++) {
  int id = g_graph_edges[i];
  if(!g_graph_visited[id]) {
     ...
  }
}
```

Figure 2.1: Target Loads

2.1.1 Hardware Prefetching

- **Step-1:** Identify the loop. A loop is detected whenever a backward branch is encountered by a warp. This is the necessary condition for a loop.

- **Step-2:** Average iteration count is calculated at the start of the iteration to decide the prefetch distance.
- **Step-3:** Two sets of tags are used to detect the target loads (the stride access and the random access). Once the tags are set, the data of next iteration loads are stored in spare registers.
- **Step-4:** In the next iteration, convert the *load* instruction to *mov* and prefetch the data into spare register for the iteration to follow.

2.1.2 Software Prefetching

- The compiler can identify the target loads and can perform the prefetching and also determines the iteration count before entering into the loop.

2.1.3 Points to note:

- The Software prefetching is faster than the Hardware because the hardware has to generate tag values to prefetch and it takes few iterations.
- Spare register aware prefetching is effective when there is only one target load pair
- Spare register aware prefetching is not effective when there is only strided accesses are present[1]

2.2 Texture Cache

The texture cache is a *read-only* cache. It will store the texture image data which has to be applied over other surfaces. This texture image data will be written once and read many number of times. It is placed closer to the shader-unit to reduce the latency. The hit rate of texture cache is high because of heavy reuse between neighbouring pixels.

Each SM has a dedicated L1 Texture cache as shown in figure-1.2, It is connected to L2 cache via a crossbar. To avoid the orientation dependency textures are stored in memory in a tiled manner. Each texture is broken into a series of n x n tiles and all the texels in the tile are stored before moving onto the next tile in the tile are stored before moving onto the next tile.
The internal architecture of a texture cache is shown in figure-1.3. In addition to shader units, there will be texture units. When a shader performs a texture read, it sends a request to the texture unit. The texture unit batches up a bunch of requests and performs the addressing math on them—selecting mip levels and anisotropy, converting UVs to texel coordinates. Once it knows which texels it needs, it reads them through the texture cache.

2.2.1 Points to note:

- In recent years, using GPUs for general purpose parallel computing has become common place and is typically referred to as GPGPU
- In modern read/write L2 cache GPU architectures, the texture cache doesn't offer a great advantage over the shader load/store L1 cache

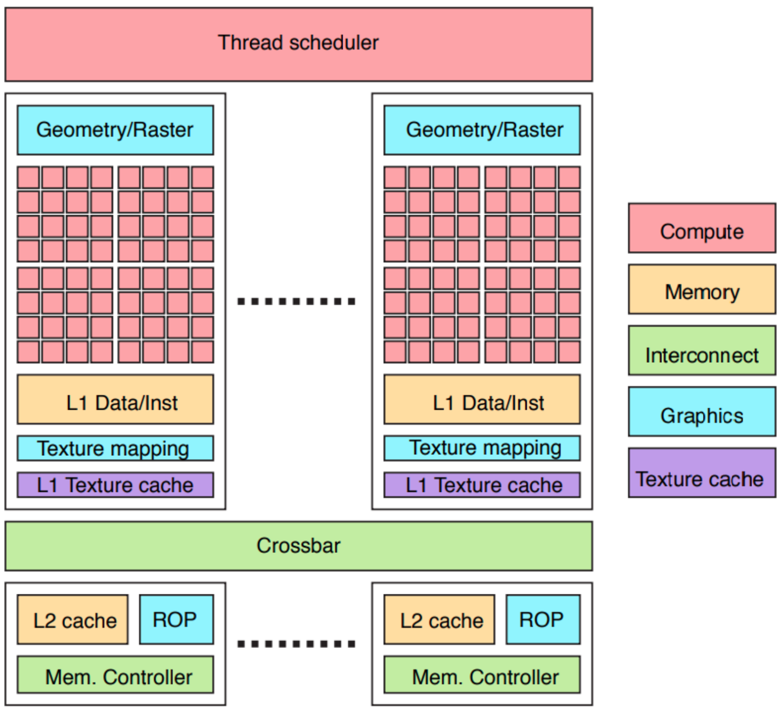


Figure 2.2: Modern GPU Architecture

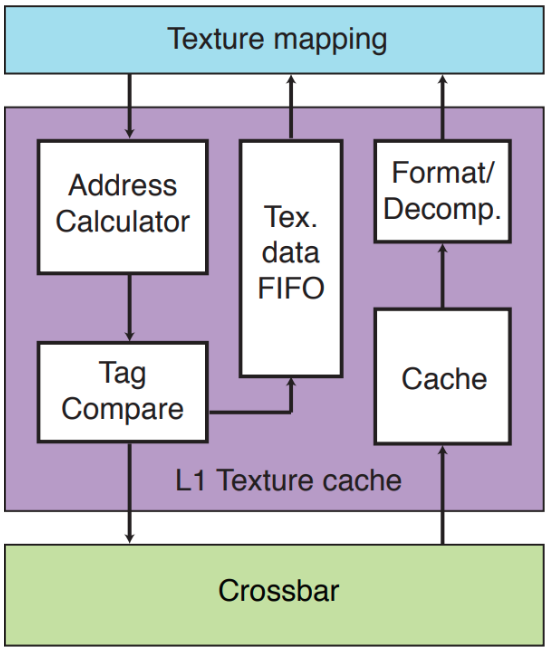


Figure 2.3: Texture Cache Architecture

- The texture filtering custom hardware can be put to good use in GPGPU applications, if a good match for the filtering operations can be found.[2]
- Since data is written once and read many times, we can use to texture cache to optimize the performance of WORM (write once read many) applications.

References

[1] N. B. Lakshminarayana and H. Kim, “Spare register aware prefetching for graph algorithms on gpus,” in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 614–625, 2014.

[2] M. Doggett, “Texture caches,” *IEEE Micro*, vol. 32, no. 3, pp. 136–141, 2012.

[3] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, “Analysis and optimization of the memory hierarchy for graph processing workloads,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 373–386, 2019.