Microprocessors and Microsystems 38 (2014) 717–729

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Improved GPU SIMD control flow efficiency via hybrid warp size mechanism

Xingxing Jin, Brian Daku, Seok-Bum Ko *

Department of Electrical and Computer Engineering, University of Saskatchewan, 57 Campus Drive, Saskatoon, S7N 5A9 SK, Canada

ARTICLE INFO

Article history:
Available online 28 June 2014

Keywords:
SIMD
GPU
Warp
Branch divergence

ABSTRACT

High single instruction multiple data (SIMD) efficiency and low power consumption have made graphic processing units (GPUs) an ideal platform for many complex computational applications. Thousands of threads can be created by programmers and grouped into fixed-size SIMD batches, known as warps. High throughput is then achieved by concurrently executing such warps with minimal control overhead. However, if a branch instruction occurs, which assigns different paths to different threads, one warp will be broken into multiple warps that have to be executed serially, consequently reducing the efficiency advantage of SIMD. In this paper, the contemporary fixed-size warp design is abandoned for a hybrid warp size (HWS) mechanism. Mixed-size warps are generated according to HWS and are scheduled and issued flexibly. The simulation results show that this mechanism yields an average speedup of 1.20 over the baseline architecture for a wide variety of general purpose GPU applications. The paper also integrates HWS with dynamic warp formation (DWF), which is a well-known branch handling mechanism used to improve SIMD utilization by forming new warps out of split warps in real time. The simulation results show that the combination of DWF and HWS generates an average speedup of 1.27 over the DWF-only platform with an estimated area increase of about 1% of DWF.

1. Introduction

Thread level parallelism (TLP) when compared with instruction level parallelism (ILP) is becoming the dominant technique to satisfy increasing computation demand, as single thread performance improvement are getting slow. Intel's Larrabee [1], IBM's Power7 [2], NVIDIA's Tesla GPU [3] and AMD's Fusion APU [4] all employ TLP in various ways. These devices typically implement TLP within graphic processing units (GPUs). GPUs have become the dominant parallel architecture in these devices because of GPUs' significant computational power, large bandwidth and high energy efficiency.

GPUs are characterized by numerous simple yet energy-efficient computational cores, thousands of simultaneously active fine-grained threads and large off-chip memory bandwidth [5]. These threads are grouped into fixed-size single instruction multiple data (SIMD) batches, known as warps. Generally warp size is equal to or a multiple of SIMD width. Correlated threads within a warp execute the same instruction in sequence on different registers in parallel. This organization amortizes the overhead of instruction fetch and decode, and therefore, more processing units can be integrated onto a single chip. Contemporary GPUs employ the fine-grained multithreading organization, to hide stalls that arise from long-latency operations. When any thread within a warp experiences a long-latency operation or a data hazard, the entire warp is stalled. However, other warps that are ready to be executed will be issued to pipelines. Multiple warps will occupy pipelines concurrently and throughput loss will be reduced. For example, for NVIDIA's GPUs, the latency of read-after-write dependencies is approximately 24 cycles. If there are more than 192 active threads (8 GPU cores per multiprocessor $\times$ 24 cycles of latency = 192 active threads, or 6 interweaved active warps of size 32), the latency can be completely hid through this multithreading technique [6].

GPUs have tremendously accelerated many applications. For example, up to February 2012, NVIDIA had listed 1287 GPU applications with 214 of these applications obtaining a speedup of 50 or more and 135 of the 214 obtaining a speedup of 100 or more [7]. However, there are many applications that can achieve only limited performance improvement or no improvement at all. One major barrier to performance improvement is branch divergence.

The SIMD organization saves control overhead and increases computation density. However, when a branch instruction is

* Corresponding author. Tel.: +1 306 966 5456.
E-mail addresses: xing.jin@usask.ca (X. Jin), brian.daku@usask.ca (B. Daku), seokbum.ko@usask.ca (S.-B. Ko).

http://dx.doi.org/10.1016/j.micpro.2014.06.007
0141-9331/

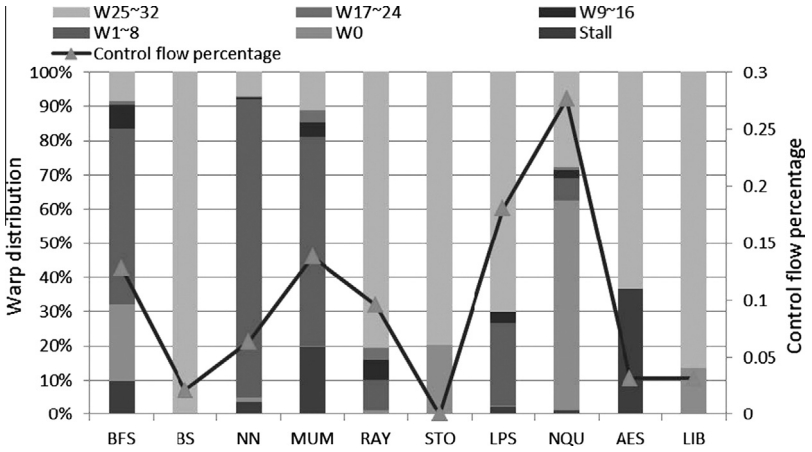

Fig. 1. Warp occupancy and control flow weight.

executed within a warp resulting in different paths for different threads, this warp will be broken into multiple warps which have to be executed serially. Meanwhile, warp occupancy[1] will be decreased and throughput will be reduced significantly. Fig. 1 shows warp occupancies for a set of general purpose applications. The weight of the branch instructions is also shown. The warp size here is set to 32. Each stacked bar represents an application. Within each bar, every block indicates the percentage of cycles corresponding to a certain number of active threads. The figure shows that benchmarks BFS, NN, MUM, LPS and NQU (see Table 3 on page 22) have relatively higher numbers of under-filled warps. Meanwhile, they all have comparatively more control flow instructions. This indicates that control flow intensive applications will more likely suffer from branch divergence leading to more idle computation resources.

This paper makes the following contributions:

1. It proposes a novel mechanism to overcome throughput loss due to branch divergence. It abandons current fixed-size warp design and introduces a hybrid warp size (HWS) mechanism. Warp size is set dynamically by hardware with the aim of achieving as high a throughput as possible.
2. It demonstrates the realistic hardware implementation of HWS and its estimated area overhead.
3. It combines HWS and dynamic warp formation (DWF), a well-known technique that deals with the GPU control flow issues. Other than DWF, it introduces a new squeeze algorithm to make individual warps denser before combining them with other warps. Meanwhile, it modifies the pattern of warp formation to better tolerate warp conflicts.[2]
4. It gives a theoretical method to estimate throughput loss due to low warp occupancy and, furthermore, approximates the potential room for performance improvement.

The remainder of this paper is organized as follows: Section 2 describes the baseline GPU architecture. Section 3 discusses the mainstream branch handling methods. Section 4 describes the proposed HWS mechanism and outlines the integration with DWF. Section 5 shows the simulation method of this work. Section 6 gives the simulation results. Section 7 discusses related work and Section 8 provides a summary.

[1] Warp occupancy is defined as the percentage of active threads within a warp [12].
[2] Warp conflict: Each warp has multiple slots. If two warps both have at least one active thread in the same slot, we call this a warp conflict.

2. Baseline GPU architecture

In this section, the baseline GPU architecture described is based on NVIDIA devices of compute capability 1.x, where 1 is for devices based on G80 architecture and x represents the minor revision number. However, the method discussed in this paper can be extended to other SIMD architectures.

Contemporary GPU cores are organized into a hierarchy. The top level of GPU is composed of an array of processors referred to as streaming multiprocessors (SMs) [8]. All SMs are connected to multiple memory modules through an interconnect network, as shown in the left part of Fig. 2 [11].

The right part of Fig. 2 gives the details of a single SM architecture [9]. It is mainly composed of a shared instruction fetch unit, an instruction decode unit, a highly banked register file and multiple ALUs. Before execution, individual threads are grouped into fixed-size warps [8], which are the granularity used for scheduling inside a SM. In fetch stage, the scheduler selects a warp from the scheduling pool using a round-robin policy. Then the instruction cache is accessed and the instruction decode is performed. Next, multiple register values are read synchronously and then fed into ALUs, where the computation is finished in parallel. Once a warp reaches the final stage of the pipeline, it will be committed and put into the scheduling pool again for future scheduling. However, if any threads in a warp encounter a long latency operation (such as a DRAM access), the warp will be taken out of the scheduling pool until the warp is committed, meanwhile other warps will be issued. As a result, long latency can be hidden and throughput loss can be reduced.

3. Branch divergence handling

Branch divergence is a key issue for general purpose GPU applications. It occurs when threads within a warp take different paths. For equal length paths, an if-else branch instruction loses 50% efficiency. To facilitate user programming, contemporary GPUs allow threads to branch and execute independently, and therefore, threads with different paths within a warp can be directly serialized, as shown in Fig. 3(a).

Fig. 3(d) gives the example program. Once the divergent point A is reached, the two warps $W0$, $W1$ are split into four fragments $W0:A-B$, $W0:A-C$, $W1:A-B$, $W1:A-C$. Next, the four segments will continue executing until the end of the program, even though they have an opportunity to converge (at merge point D).