Journal of Systems Architecture 87 (2018) 49–63

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

Last level cache layout remapping for heterogeneous systems

Licheng Yu*,[a], Tianzhou Chen[a], Minghui Wu[b], Xueqing Lou[a]

[a] College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang, China
[b] School of Computer & Computing Science, Zhejiang University City College, Hangzhou, Zhejiang, China

ARTICLE INFO

Keywords:
Memory layout
LLC
GPGPU
Heterogeneous system

ABSTRACT

Heterogeneous systems with CPU and GPGPU sharing the last level cache (LLC) provide viability and flexibility. However, the different programming models lead to conflicting memory layouts, which are required for best performance of different processors. Software converting that directly accesses target layout is subject to sub-optimal localities. Converting in GPGPU shared memory also incurs copying and synchronization overhead.

In this paper, we analyze the memory layout requirement and propose to remap the memory layout in the shared LLC. A remap controller in LLC executes a simple program that calculates target requests from an LLC request in the source memory space. The LLC request is thus remapped to the target memory space with the generated requests. Consequently, all processors always access memory in their optimal data layouts. The locality is thus kept through all the private caches, and software remapping overhead is also eliminated.

The tiled-matrix multiplication is discussed as a case study and benchmarks from Polybench/GPU and Rodinia are modified to take advantage of the LLC layout remapping. The experiment results show the average benchmark execution time is decreased to 69%. Compared with CPU software layout converting, the CPU time is decreased to 41%–73%.

1. Introduction

Heterogeneous systems are widely adopted from super computers to mobile devices. Among them, general-purpose computing on graphics processing unit (GPGPU) complements CPU with high throughput and excellent power efficiency, which are achieved via the massive parallel architecture. Traditionally, as a peripheral device, GPGPU is connected to CPU through a peripheral bus such as PCI Express. The system memory access from GPGPU incurs high latency and bus overhead, and thus a discrete GPGPU has its private memory for high speed memory access. Shared data must be copied between system memory and GPGPU's memory for collaboration. On the other hand, closely-coupled CPU-GPGPU architecture with shared last level cache (LLC) enables CPU to offload workload to GPGPU in fine-grained. Since the memory underneath the LLC are also shared, data copy is eliminated.

Fig. 1 gives the targeted system with shared LLC between CPU and GPGPU. The CPU has its private L1 cache and L2 cache. The latter connects to the system memory through the shared LLC. The GPGPU consists of multiple *stream multiprocessors* (SMs), each of which has a private L1 data cache and a scratchpad memory. The scratchpad memory is directly addressable and is named *shared memory* (SHM in Fig. 1). All SMs are further connected to the memory system with multiple L2 cache banks, which are shared among all SMs but are private to the GPGPU.

A program (or *kernel*) on GPGPU is executed as hundreds of hardware threads running simultaneously. New programming models such as CUDA are developed to natively support GPGPU architecture. Without loss of generality, we use CUDA's terminology in the paper. Threads are grouped into *blocks* to enable affordable inter-thread synchronization inside each block. Further, threads in small groups execute in lock-step (or single instruction multiple data, SIMD) to simplify the control logic, and each group is a *warp*. A memory instruction executed by a warp generates as many memory requests as active threads of the warp. Memory request coalescing merges requests with a good spatial locality into one request and is employed by GPGPU to alleviate the high memory bandwidth requirement.

On the other hand, a conventional CPU focuses on performance of single thread and algorithms on CPU exhibit different memory access patterns from algorithms on GPGPU. In single thread CPU algorithms, intra-thread memory locality is always kept in mind. For GPGPU, the memory coalescing among threads of a warp renders inter-thread memory locality more important for better memory performance. A typical scenario is CPU prepares the shared data set and starts GPGPU for further processing. For a given loop-based CPU algorithm, one common way to implement a GPGPU algorithm is to execute one iteration of the loop in each GPGPU thread. This causes different

* Corresponding author.
E-mail addresses: yulicheng@zju.edu.cn (L. Yu), tzchen@zju.edu.cn (T. Chen), mhwu@zucc.edu.cn (M. Wu), xqlou@zju.edu.cn (X. Lou).

https://doi.org/10.1016/j.sysarc.2018.05.002
Received 13 February 2018; Received in revised form 6 May 2018; Accepted 8 May 2018
Available online 16 May 2018
1383-7621/

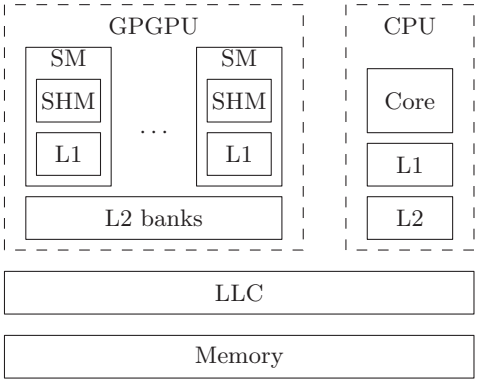

Fig. 1. System with shared LLC between CPU and GPGPU.

locality requirements from algorithms on CPU that prepares the shared data set. These mismatch requirements lead to conflict optimal memory layouts of the shared data and result in inefficient data sharing.

A possible solution is to convert the memory layout of the shared data set with either CPU or GPGPU. For CPU converting, it inevitably becomes a bottleneck due to serialization. Despite that SIMD execution such as SSE or AVX is also supported in modern CPUs, the mismatching SIMD width between CPU and GPGPU prevents CPU from generating the optimal target layout for GPGPU. Further, the sub-optimized layout data also contaminate the CPU's private caches and degrades CPU performance.

GPGPU can convert the layout with shared memory during processing or with a dedicated kernel before real processing. The former method loads data into shared memory in a coalesced way and accesses the shared memory in the algorithm's way but incurs shared memory management overhead. The latter usually needs high-overhead global atomic operations and introduces extra kernel launch overhead. Since the converting kernel expects full input data, it also makes overlap of converting and data generation impossible.

To provide proper memory layout for efficient memory access on both CPU and GPGPU, while mitigate the overhead of layout converting, we propose the layout remapping in the LLC. A remap controller is added in the LLC to generate requests in target memory space from a request in source memory space with a remap program. Since the converting is done during LLC access, GPGPU and CPU are free to scheduling other threads to overlap useful work with it, and the resource occupation such as the shared memory in GPGPU is also lowered. Further, algorithms running on different processors can access the shared data in their optimal layouts, and keep the locality along all their private caches. Therefore, compared with the traditional software converting, our method eliminates the extra converting work on either CPU or GPGPU, and provides efficient memory access for both processors.

The contributions of this paper are as follows:

- We analyze the memory access patterns of typical CPU and GPGPU algorithms and reveal the conflict in optimal memory layout caused by different memory locality behaviors. We also show the problems in software layout converting.
- LLC layout remapping is proposed to convert the memory layout in the LLC level to alleviate the problems of software layout converting.
- A case study of tiled-matrix multiplication is presented to demonstrate the proposed LLC remapping method.
- Performance and overhead of the LLC remapping is evaluated with more benchmarks. We also compare our solution with the GPGPU converting proposed by Sung et al. [1].

The paper is organized as follows. Section 2 gives the experiment setup in this paper. Section 3 discusses the memory access patterns and traditional memory layout data converting methods. Section 4 introduces the layout remapping, and Section 5 presents the design and implementation of the LLC layout remapping. Section 6 shows a case study on tiled-matrix multiplication with the proposed method, which is further evaluated in the experiment of Section 7. Section 8 gives the related work. Finally, Section 9 concludes the paper.

Table 1
Configuration parameters.

CPU	Core frequency	2 GHz
	Data L1 cache	64KB
	L2 cache	2MB
GPGPU	SMs	15
	SM frequency	700 MHz
	L1 cache	16KB, write-evict, private to each SM
	Shared memory	48KB, private to each SM
	NoC	Butterfly
	L2 cache	6 banks, 128KB each, shared by all SMs
LLC	Remap frequency	2 GHz
	Remap ALUs	8
	Block size	128 bytes
	Associative	8 ways
	Size	16MB
	Hit Latency	20 ns
	Replacement	LRU
Memory	Bandwidth	12.8GB/s
	Latency	50 ns

2. Experiment methodology

Before present any experiment results, we first introduce the experiment setup in this paper. Our experiment platform is based on a cycle-accurate simulator that simulates the shared LLC system of CPU and GPGPU. The simulator implements the CPU and memory system with a full system simulator gem5 [2] and adds the GPGPU simulation from GPGPU-Sim v3 [3], which mimics Nvidia Fermi architecture. We adopt x86-64 CPU core and setup the GPGPU with Nvidia GTX480 configuration included in the GPGPU-sim. The purposed remap controller is implemented along the LLC controller in the simulator. Table 1 shows the main configuration parameters.

3. Memory access patterns and motivation

3.1. Memory layout

When data structures are stored in an array, it becomes a multi-dimension data set, which has to be rearranged in a one-dimension layout to store in the linear memory space. When the array element is structure instances and elements of each instance are stored continuously, the layout is named *array of structure* (AoS). For example, lines 1–5 of Fig. 2 define an array `aos` of 100 structure instances of `foo`. Meanwhile, the same structure elements from different structure instances can also be stored continuously as shown in lines 7–11, where a single structure instance `soa` contains arrays of its original structure's elements. This is named *structure of array* (SoA).

Since a structure array data set can be viewed as a 2D matrix, each row of which is one structure instance, we present the matrix representation. To store in the linear memory space, a 2D matrix is usually arranged in either *row-major* or *column-major*, corresponding to AoS and SoA, respectively. For a matrix with m rows and n columns $A_{m\times n} = \{a_{i,j}\}$, where $i \in [0, m)$, $j \in [0, n]$, the row-major stores neighboring elements of a row consecutively, row-by-row ($\{a_{0,0}, a_{0,1}, ..., a_{0,n-1}, a_{1,0}, ..., a_{m-1,n-1}\}$). Correspondingly, column-major places the column elements together ($\{a_{0,0}, a_{1,0}..., a_{m-1,0}, a_{0,1}, ..., a_{m-1,n-1}\}$).