Data Layout Optimization for GPGPU Architectures

Jun Liu, Wei Ding, Ohyoung Jang, Mahmut Kandemir
The Pennsylvania State University, University Park, PA 16802, USA
{jxl1036, wzd109, oyj5007, kandemir}@cse.psu.edu

Abstract

GPUs are being widely used in accelerating general-purpose applications, leading to the emergence of GPGPU architectures. New programming models, e.g., Compute Unified Device Architecture (CUDA), have been proposed to facilitate programming general-purpose computations in GPGPUs. However, writing high-performance CUDA codes manually is still tedious and difficult. In particular, the organization of the data in the memory space can greatly affect the performance due to the unique features of a custom GPGPU memory hierarchy. In this work, we propose an automatic data layout transformation framework to solve the key issues associated with a GPGPU memory hierarchy (i.e., *channel skewing*, *data coalescing*, and *bank conflicts*). Our approach employs a widely applicable strategy based on a novel concept called *data localization*. Specifically, we try to optimize the layout of the arrays accessed in affine loop nests, for *both* the device memory and shared memory, at *both* coarse grain and fine grain parallelization levels. We performed an experimental evaluation of our data layout optimization strategy using 15 benchmarks on an NVIDIA CUDA GPU device. The results show that the proposed data transformation approach brings around 4.3X speedup on average.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation, Compilers, Optimization

General Terms Algorithms, Design, Performance, Experimentation

Keywords GPGPU, Data Layout Transformation, CUDA, Optimization

1. Introduction

The CUDA [1, 5] programming model can greatly improve the programmer productivity. However, fully utilizing the specific features of the underlying architecture can be very challenging. One of the most important reasons for this is that the customized memory hierarchy in GPGPUs needs to be explicitly managed at an application level by the CUDA programmer. We believe that automated compiler support can play a critical role in exploiting the memory hierarchy of emerging GPGPU systems. It is also to be noted that loop nests constitute a significant fraction of application execution time in high performance computing. The unique characteristics of the GPGPU architectures pose new challenges as well as opportunities for traditional loop optimization techniques. As shown in Figure 1, the CUDA GPU memory hierarchy consists of a (global) device memory and shared memories on the streaming multiprocessors. Clearly, patterns exhibited by data accesses to different memory components can greatly affect the overall performance of the applications running on this architecture.

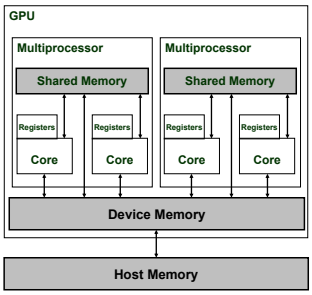

Figure 1. Memory hierarchy of a GPU architecture.

In this work, we target the codes already parallelized for CUDA GPGPUs, and focus on *data layout transformations* to improve application performance by taking into account the underlying memory hierarchy. Our approach employs a *general* data layout optimization strategy based on a novel concept called *data localization*. Specifically, we first consider the data access patterns exhibited by the parallel processing units (thread blocks/threads), followed by identifying localized arrays and partitioning them into data blocks that are mostly accessed by corresponding processing units. We then apply both an affine data transformation and a non-affine data transformation to change the layout of the localized arrays to solve three specific problems associated with the GPGPU memory hierarchy: *channel skewing*, *memory coalescing* and *shared memory bank conflicts*.

2. Problem Definition

As shown in Figure 1, the device memory of the GPU can be accessed by all thread blocks mapped to different streaming multiprocessors. Each multiprocessor has its own shared memory, which can only be accessed by the threads assigned to that multiprocessor. The device memory can be accessed through different memory channels at the same time to increase memory level parallelism. In addition, if the data accessed by different threads on the same multiprocessor are aligned and continuous in the device memory, they can be coalesced into a single memory operation to increase memory bandwidth. On the other hand, the most frequently reused data are likely to be copied into the shared memory to reduce access latency, and the banks of the shared memory provide parallel data accesses. However, there exist three issues as follows regarding this GPU memory hierarchy that need to be addressed in order to exploit its full potential.

PPoPP'13, February 23–27, 2013, Shenzhen, China.

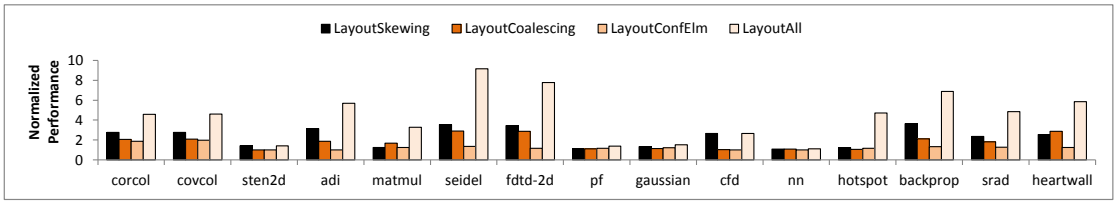

Figure 2. Performance of our data layout transformations normalized to the Original version.

- **Channel Skewing** If the accesses to the memory channels of the device memory are not well balanced, some of the channels may be congested, and as a result, bandwidth utilization can be affected.
- **Coalescing** Threads from the same warp may access the data elements stored non-contiguously in the device memory. In such cases, the memory coalescing instructions provided by the underlying architecture cannot be utilized.
- **Bank Conflicts** If at the same time, different data accessed by different threads to the shared memory are located in the same bank, conflicts occur and the accesses involved need to be serialized.

3. Data Layout Optimization

3.1 Parallelization

In this work, we mainly target at optimizing the CUDA kernels that are transformed from *affine loop nests*. We assume that the array references and loop bounds are affine functions of the enclosing loop indices and loop-independent variables. Our optimizations are thus based on polyhedral model and mainly focus on optimizing direct/regular data accesses, which are more common in most applications. We further assume that our target loop nests have already been parallelized for the CUDA. Specifically, how the computations are divided into thread blocks and how each thread block is partitioned into threads, are assumed to be known to our approach.

3.2 Layout Optimization

For a kernel extracted from an affine loop nest under the parallelization described above, to achieve high performance, we need to solve the three problems discussed above. Considering the unique features of the CUDA programming model and memory hierarchy, each layout optimization applied in our work mainly consists of two basic steps, *localization* and *data relocation*. The basic idea of the first step (data localization) is to divide the entire data space of an array into *data blocks*, such that the data elements in each data block are mostly accessed by only one processing unit (thread block/thread). In other words, data localization ensures that the data elements accessed by the same processing unit are not spread over the entire data space. The second step then performs the actual transformation that maps the data from its current layout to the desired layout. Overall, our layout optimization employs a two-level transformation, i.e., coarse level (thread block) and finer level (thread). Specifically, the data optimization at the coarse level is used to minimize channel skewing in the device memory. Our goal of this optimization is to obtain a data layout so that the data accesses made by different thread blocks will go to different memory channels of the device memory. In this way, the thread blocks mapped to different multiprocessors will have balanced accesses to the memory channels of the device memory, lessening the channel skewing problem. Our data optimization at the finer level tries to increase the opportunities for data coalescing in the device memory and reduce the number of bank conflicts in the shared memory. Intuitively, in the device memory, we apply affine transformation to obtain a data access pattern such that the threads of a warp will access (at each time step) data elements that are stored in consecutive memory locations. In other words, our layout transformation targets at improving data locality among the threads of a warp, i.e., *inter-thread locality*. In the shared memory, we apply strip-mining and permutation to obtain the desired data layout where data accessed by a thread will be only stored in its own associated bank.

4. Experimental Evaluation

We implemented our proposed data layout optimization strategy in Pluto 0.6.2-CUDA [2, 3], a source-to-source transformation framework for affine loop nests based on the polyhedral model with CUDA support. The optimized kernel code is then compiled with the CUDA compiler (nvcc release 4.0, V0.2.1221) into binaries for execution on the GPU. We performed our experiments on an NVIDIA Tesla C2050 GPU device. The device is equipped with 2.8 GB of device (global) memory and 14 streaming multiprocessors clocked at 1.15 GHz.

We evaluated our strategy on 15 benchmarks from Pluto [2] and Rodinia [4]. Each benchmark is compiled into five versions, namely, *Original*, *LayoutSkewing*, *LayoutCoalescing*, *LayoutConfElm*, and *LayoutAll*. Specifically, the *Original* version is the original parallel CUDA kernel code without our data layout optimization; the *LayoutSkewing* version is obtained by applying our data optimization (to the Original version) for reducing channel skewing in the device memory; the *LayoutCoalescing* version is generated by applying our data optimization (to the Original version) for increasing coalescing instructions; the *LayoutConfElm* version is formed by applying our data optimization (to the Original version) for reducing bank conflicts in the shared memory; and the *LayoutAll* version is obtained by applying all of our three optimizations to the Original version. Figure 2 gives performance numbers for different versions, *normalized* with respect to the Original version. The average normalized performance values (when all 15 applications are accounted for) of LayoutSkewing, LayoutCoalescing, LayoutConfElm and LayoutAll versions are 2.3, 1.8, 1.3 and 4.3, respectively.

Acknowledgments

This research is supported in part by NSF grants #1213052, #1152479, #1147388, #1139023, #1017882, #0963839, #0811687 and a grant from Microsoft Corporation.

References

[1] CUDA. `http://www.nvidia.com/object/cuda_home_new.html`.

[2] PLUTO. `http://pluto-compiler.sourceforge.net/`.

[3] U. Bondhugula et al. A practical automatic polyhedral parallelizer and locality optimizer. *Proc. of PLDI*, 2008.

[4] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. *IISWC*, 2009.

[5] M. Garland et al. Parallel computing experiences with CUDA. *MICRO*, 2008.