Automatic Data Layout Transformation for Heterogeneous Many-core Systems

Ying-Yu Tseng[1], Yu-Hao Huang[1], Bo-Cheng Charles Lai[1], Jiun-Liang Lin[1]

[1] Department of Electronics Engineering, National Chiao-Tung University
1001 Da-Hsueh Rd, Hsinchu, Taiwan

yingyu.ee99@nctu.edu.tw, pcco001.ee99@nctu.edu.tw,
bclai@mail.nctu.edu.tw, qazhphphphp3@gmail.com

Abstract. Applying appropriate data structures is critical to attain superior performance in heterogeneous many-core systems. A heterogeneous many-core system is comprised of a host for control flow management, and a device for massive parallel data processing. However, the host and device require different types of data structures. The host prefers Array-of-Structures (AoS) to ease the programming, while the device requires Structure-of-Arrays (SoA) for efficient data accesses. The conflicted preferences cost excessive effort for programmers to transform the data structures between two parts. The separately designed kernels with different coding styles also cause difficulty in maintaining programs. This paper addresses this issue by proposing a fully automated data layout transformation framework. Programmers can maintain the code in AoS style on the host, while the data layout is converted into SoA when being transferred to the device. The proposed framework streamlines the design flow and demonstrates up to 177% performance improvement.

Keywords: heterogeneous systems, data layout transformation, many-core, GPGPU.

1 Introduction

Heterogeneous many-core systems have demonstrated superior performance by leveraging the benefits from processing units with different characteristics. A heterogeneous system consists of a host and a device, where the host takes charge of sophisticated algorithm flow while the device performs massively parallel data processing [10]. Fig. 1 illustrates a widely adopted architecture of modern heterogeneous many-core systems. The system applies high-end CPUs as the host, and a GPGPU (General Purpose Graphic Processing Unit) as the device. CPUs perform control flow management and enable ease of programming, while GPGPUs support massive parallel execution to achieve high throughput data processing. Tasks of an application can then be dispatched to the best-suited processing resources to achieve efficient and high performance computing.

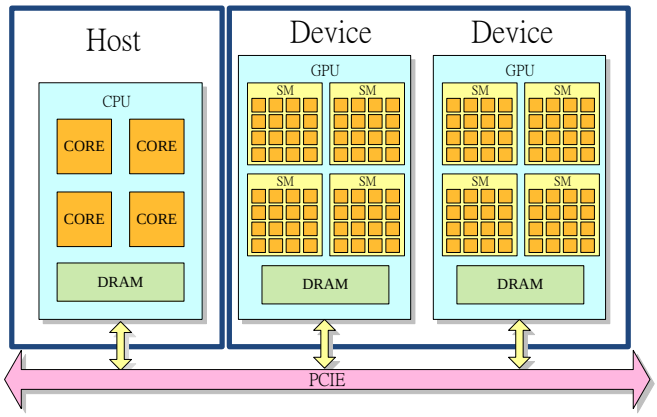

Fig. 1. The organization of a heterogeneous many-core system.

The heterogeneity of the system has caused disparate design philosophy and resultant execution behavior between the host and device. A common design flow divides an application into two parts. The control of the execution flow is taken care of by the host processor, while the part with massive parallelism is transferred to the many-core engines on the device for high throughput computing. Since the massive parallel computation usually poses intensive data accesses, developing a proper data structure for the corresponding many-core device is a critical performance factor.

The preferred data structures of tasks are different on the two sides of the system. The programming paradigm of the host processor usually applies conventional object-oriented scheme, which tends to pack the associated elements of the same object into the same class. This scheme enhances both code readability and maintainability. However, the same data layout scheme does not provide efficient data accesses to the throughput processors on the device side.

Fig. 2 illustrates an example of multi-object operations on GPGPUs. This operation has been widely applied to various applications, such as image processing. As in Fig. 2, there are three types of data elements, R, G, and B. The application can simultaneously operate on the data belonging to the same type, and would process the data of different types in turns. An object can be represented by combining the associated data from different types. For example, a pixel in an image consists of color elements from red, green, and blue. To have better code readability, programmers tend to pack the elements for the same object into the same class. Such data layout is referred as the Array-of-Structure (AoS). However, AoS is not an efficient data layout for GPGPUs. The parallel tasks in a GPGPU are clustered into execution groups, called warps. Tasks in a warp perform the same operation on different data points. To enable efficient data accesses, the data elements of the same type are required to be arranged in a consecutive manner. This data layout scheme is referred as Structure-of-Arrays (SoA). Programmers need to transform the data layout from AoS to SoA before passing the execution to the device side. However, SoA is usually against the intuition of understanding an object, and makes the code hard to maintain. This issue has involved

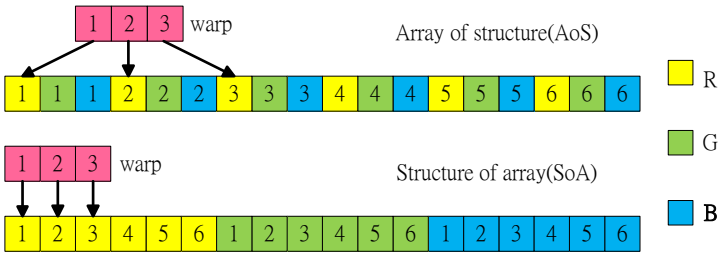

Fig. 2. The data layout of AoS style and SoA style.

the tradeoff of programmers between the performance on devices and the readability and maintainability of codes. The former would degrade the benefits of heterogeneous many-core systems, while the latter would significantly burden programmers.

This paper proposes an automatic data layout transformation framework to streamline the design flow as well as alleviate the overhead. The data is maintained as AoS structure on the host, and automatically transformed into the ASTA (Array-of-Structure-of-Tiled-Arrays) structure [1] during the data transfer from the host to device. ASTA arranges data into tiled arrays, which can be more effectively utilized by a GPGPU. The proposed framework involves the design of several novel hardware modules. The hardware Data Layout Transformer performs the data layout transformation to ASTA during the data transfer on the PCIe interface. In this way, the runtime overhead of data layout transformation can be hidden. When receiving the data on the device, a pipelined adapter is implemented to transpose the data efficiently. Specialized load and store units are also developed to translate the assigned data addresses to the target addresses. The proposed framework enables automatic data layout transformation between the host and device, and is transparent to programmers. The experimental results have demonstrated up to 177% performance improvement with the proposed framework.

The rest of the paper is organized as follows. Section 2 discusses the related work of data accesses of heterogeneous many-core systems. Section 3 introduces the hardware data layout transformation framework proposed in this paper. Section 4 shows the simulation results and analyzes the performance enhancement. Section 5 will concludes this paper.

2 Related Works

A heterogeneous system has applied disparate design philosophy and resultant execution behavior between the host and device. The corresponding data structure of an application also needs to be adapted to the characteristics and requirements of the system. GPGPUs on the device side have implemented memory coalescing, which combine consecutive data accesses into a single transaction. It has been demonstrated that to retain the benefit provided by memory coalescing, the data structure needs to be arranged as the SoA scheme [1][2][3][8].

The data layout transformation has been studied on both CPUs and GPGPUs. For CPUs, Karlsson [4] discussed an in-place transposition way and Gustavson and Swirszcz [5] proposed an OpenMP-based approach. On GPGPUs, early researches gave the out-of-place transposition way and the performance was limited due to the ineffective usage of GPGPU DRAM [6]. The Dymaxion framework proposed by Che et al. [7] is a library-based software approach. It performs data transformation on the CPU side and overlaps with data transfers on PCI-e. Since the data transformation is performed by CPUs, the transformation speed is limited by the CPU memory bandwidth. Sung et al. [1] proposed another library-based approach that uses in-place algorithm transforming data layout from AoS to ASTA (Array-of-Structure-of-Tiled-Arrays) on GPGPUs. The ASTA arranges data into tiled arrays, which can be more effectively utilized by a throughput processor. An in-place marshaling algorithm was also developed for transforming the data layout, and has demonstrated to be faster than the optimized traditional out-of-place transformations while avoiding doubling the GPGPU DRAM usage. However, this software approach induces runtime overhead of transforming the data layout.

This paper proposes a hardware-based data layout transformation framework that is transparent to programmers. With the proposed hardware modules, the data layout transformation and address translation have been fully automated. The programmer can benefit from the low overhead transformation, and also be able to enhance the productivity by using the more intuitive SoA object-oriented code in both CPUs and GPGPU kernels.

3 Hardware Data Layout Transformation Framework

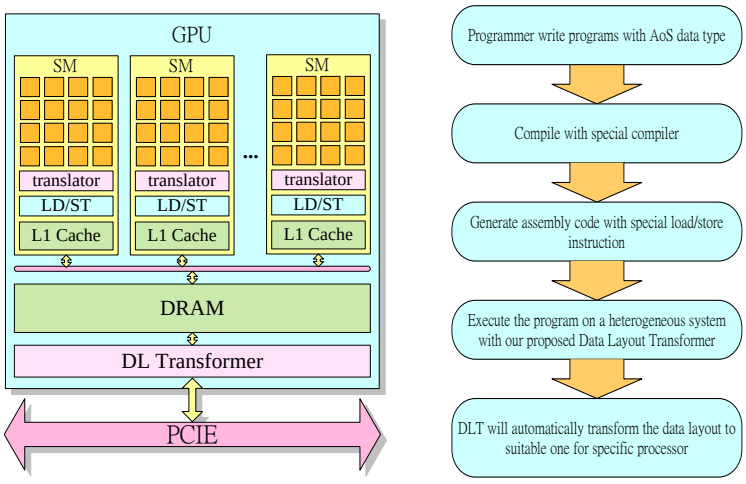

Fig. 3. (a) The hardware modules of the data layout transformation. (b) Design flow with the proposed data layout transformation framework.

To achieve the best performance on a heterogeneous many-core system, programmers are required to take considerable effort to transform the data layout to enable data accesses for the throughput processor on the device side. This paper proposes a fully automated data layout transformation framework to streamline the design flow as well as alleviate the overhead. This paper applies the proposed data transformation on a widely adopted heterogeneous system with CPU-GPGPU organization to demonstrate the fundamental functions and attained benefits.

3.1 Overview of System and Design Flow

The proposed data layout transformation consists of two hardware modules. As shown in Fig. 3(a), the first module is the Data Layout Transformer (DL Transformer) that is cascaded to the PCIe interface. The DL Transformer transforms the AoS data structure from the host to the ASTA structure, and stores the new data structure into the DRAM of the GPGPU. The second module is the specialized load/store (LD/ST) unit in GPGPUs. These LD/ST units are able to translate the data access addresses automatically for programmers. With the proposed hardware modules, the design flow of the heterogeneous program is illustrated in Fig. 3(b). Programmers maintain the more readable AoS codes for both GPGPU and CPU kernels. The AoS data structure can avoid discrete data layout with better code readability. Programmers only need to specify the AoS data structure that would be used by the parallel kernels on the device GPGPU. A simple function can be added to the current available compiler to recognize these data structures, and automatically send these data to the hardware DL Transformer. The appropriate PTX code for GPGPU will be generated to control the data receiving and address translation on the GPGPU side. By inserting the special

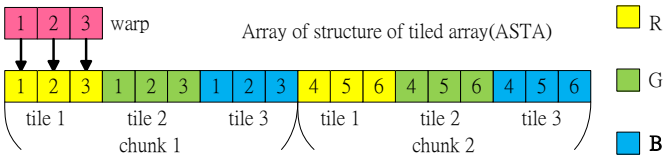

Fig. 4. The ASTA data layout.

flag into the PTX code, the load/store unit would access the data through a hardware address translator to get the transformed addresses.

3.2 ASTA Data Layout

Fig. 4 illustrates Array of Structure of Tiled Array (ASTA) data layout proposed by Sung et al. [1]. It is a type of AoS data structure optimized for GPGPUs. To achieve efficient data accesses on GPGPUs, it is not necessary to apply the SoA data layout since only tasks in the same warp will be executed concurrently. Therefore, one can only adjoin the data elements required by a warp to achieve the same data access efficiency as SoA. In ASTA, a tile refers the data elements of the same type that have been allocated consecutively. The tiles of different data types will join together and form a mini AoS structure, named a chunk. The ASTA data layout is an array of these chunks. The hardware DL Transformer proposed in this paper also utilizes ASTA data layout on GPGPUs. With ASTA, the design of DL Transformer no longer needs to gather all the same elements together. This paper also proposes a pipeline design of DL Transformer that can achieve higher performance with low hardware complexity.

3.3 Data Layout Transformer

Fig. 5 illustrates the data layout transformation on the proposed DL Transformer module. The DL Transformer is designed to transform the AoS data layout to ASTA style while transferring data from CPUs to GPGPUs. Because the procedure between different chunks is independent, the following discussion will focus on one chunk. First, the buffer A in DL Transformer gathers a new chunk from PCIe. As it finishes receiving the entire chunk, it will send the chunk to the next stage of buffer B. The data layout of this chunk will be transformed from AoS to ASTA tile by tile. It is done by K iterations, where K is the number of data elements in a class. The first iteration is generating the first tile of this chunk. A set of multiplexers is controlled by K to determine which elements to form a new tile. In each cycle of this iteration, the first bit of these selected elements is sent to buffer C and the data in both buffer B and C are shifted by one bit simultaneously. Until all bits in buffer C are ready, the new tile will be sent to buffer D so that buffer C can do the next iteration of gathering the next tile. The buffer D is used to store the tile to the DRAM in a GPGPU.

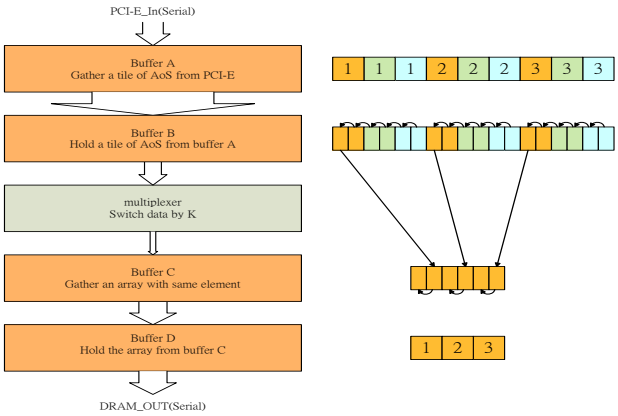

Fig. 5. Procedure of data layout transformation.

Due to the limited bandwidth of PCIe and the concern of hardware cost, the iteration between buffer B and C should be separated by some proper number of cycles. The execution cycles of these stages of one chunk workload is shown in Table 1. The design of the pipeline structure is more efficient when the execution cycles of these stages are the same. That is, the multiplexer between buffer B and C should transfer $(bandwidth/W)$ bits in each cycle to balance the latency. The parameter W is the length for a tile. For example, the bandwidth is 128bit/transaction and W is 32, DL Transformer should have 4 copies of multiplexer sets to transfer 4 bits, and the data in both buffer B and C are also shifted by 4 bits simultaneously.

Another transformation needed in our module is to transform the ASTA back to AoS when the data is transferred from a GPGPU to a CPU. The function can be achieved easily by reversing the flow of the previous transformation. The data will be sent from the GPGPU DRAM to Buffer D, and the multiplexer can be replaced by a decoder or de-multiplexer.

A possible design issue happens when the size of the whole array is not a multiple of W. It makes the last chunk of this array incomplete. To solve this issue, we choose to pad redundant bits to make the size of chunk as multiples of W. It will slightly increase the memory usage of the array. However, the number of objects of an array is usually much larger than W, and this overhead becomes negligible.

Table 1. Execution cycles in different stages.

Stage	Cycles
PCIe to buffer A	$K \times W \times Size \times 8/bandwidth$
buffer B to C	$K \times Size \times 8/bit\_per\_cycle$

3.4 Specialized Load and Store Unit

The specialized load/store (LD/ST) unit is proposed to automatically translate the original data address to the target data address (ASTA). This LD/ST module relieves the programmers from reorganizing the complex transformed data accesses when programing GPGPU kernels. When performing the transformation from AoS to ASTA, the transformed addresses can be obtained by the following equations (**1**) and (**2**). Equation (**1**) derives the index of the datum (*Index*) from *Addr_origin* and *Addr_begin*. With *Index*, one can calculate the transformed data address ($Addr'$) by adding three offsets to the beginning of the array (*Addr_begin*). The offsets are respectively listed in equation (**2a**), (**2b**), and (**2c**). Equation (**2a**) represents the element index in its tile while equation (**2b**) gives the offset by the order of tile in the chunk. The chunk offset is represented by equation (**2c**) as well. With *Addr_begin* and these three offsets, one can translate an address to the transformed ASTA style data address.

$$Index = (Addr\_origin - Addr\_begin)/Size \quad (1)$$

$$\begin{aligned} Addr' = Addr\_begin + [& \\ & (Index/K)\%W + \quad (2a) \\ & (Index\%K) \times W + \quad (2b) \\ & (Index/K/W) \times K \times W \quad (2c) \\ &] \times Size \end{aligned}$$

4 Experiment Results

This section compares the performance of the proposed DLT framework. The experiment setup will be shown in section 4.1. Section 4.2 will illustrate the performance with no hardware delay time. Section 4.3 will explore the impact of different lengths of a tile, and section 4.4 will discuss the performance effect when adding hardware delays.

4.1 Experiment Setup

The performance of the proposed data layout transformation framework is verified with GPGPU-Sim, a cycle-accurate performance simulator for GPGPU [9]. The architecture parameters of GPGPU-Sim are configured to model NVIDIA GTX480, which consists of 15 streaming multiprocessors. Each warp contains 32 concurrent threads. The benchmarks used in this paper are listed in Table 2. The Black-Scholes benchmark is adopted from CUDA SDK [11] and the other two benchmarks are from Parboil Benchmark suite [12].

Table 2. Descriptions of test benchmark.

Benchmarks	Description
Black-Scholes	This benchmark evaluates fair call and put prices for a given set of European options by Black-Scholes formula.
LBM	A fluid dynamics simulation of an enclosed, lid-driven cavity, using the Lattice-Boltzmann Method.
SPMV	Computes the product of a sparse matrix with a dense vector. The sparse matrix is read from file in coordinate format, converted to JDS format with configurable padding and alignment for different devices.

4.2 Performance Comparison with Different Data Layouts

Fig. 6 shows the normalized performance of designs with different data layout schemes. The CUDA_AoS and CUDA_ASTA apply only AoS and ASTA data structures respectively. The CUDA_AoS_DLT runs the CUDA_AoS on the platform with the proposed DLT framework. Note the performance is measured with no translator delay time. One can observe that CUDA_ASTA outperforms CUDA_AoS mainly because the GPGPU can access data efficiently with ASTA data structure. The benchmarks LBM and BlackScholes show more significant performance gain because these two applications pose regular data access behavior. In these applications, warps generate multiple accesses to the same cache line or adjacent memory locations, and therefore the performance benefits more from the ASTA structure. The SPMV benchmark, on the other hand, has irregular data access behavior. In this case, the ASTA data layout only provides minor performance gain.

Although having better performance, CUDA_ASTA applies the data structure that is not intuitive to programmers. Transforming the coding styles between AoS and ASTA requires extra programming effort. The proposed DLT automatically transforms the data layout from AoS to ASTA without changing the coding style of kernel functions. An interesting observation is that the CUDA_AoS_DLT even outperforms CUDA_ASTA in all the benchmarks. This is because the coding of AoS structure has fewer instructions in the kernel function than the ASTA structure. The AoS data layout needs only one array pointer to manipulate the whole data while the ASTA data layout needs as many array pointers as the number of arrays. The code of ASTA style needs to pass more parameters into the kernel function and also requires more instructions to calculate the addresses of different structure elements. These overheads are not involved in the proposed DLT hardware since the code still retains the AoS data layout. Therefore the CUDA_AoS_DLT can return better performance than CUDA_ASTA.

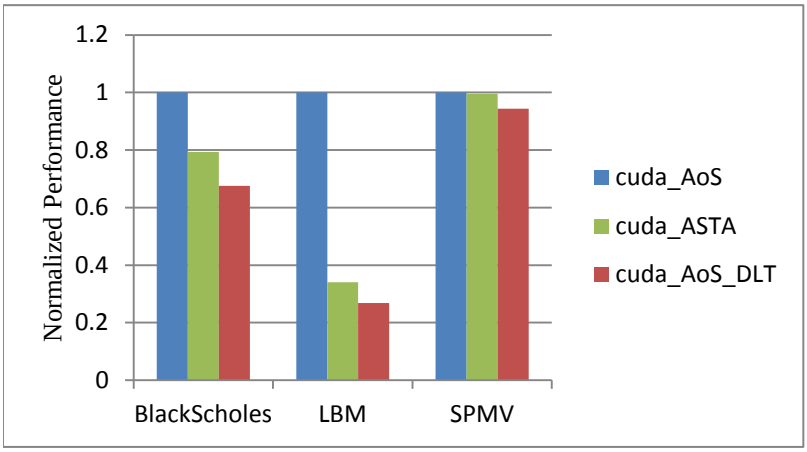

Fig. 6. The normalized performance of designs with different data layout schemes.

4.3 Impact of different lengths of a tile

Fig. 7 shows the normalized performance when the tile length is changing from 16, 32, to 64. The tile length is the parameter *W* discussed in section 3. This is an important factor since it affects not only the amount of transformer buffers and multiplexers but also the performance of the ASTA layout. The design with ASTA layout behaves like AoS layout when the parameter *W* is small. One can notice that the performance with *W = 16* is worse than 32 and 64. However, increasing the *W* from 32 to 64 does not return more performance enhancement. This is because the performance does not gain further benefit from applying larger *W*, while the overhead of supporting larger *W* starts compromising the performance.

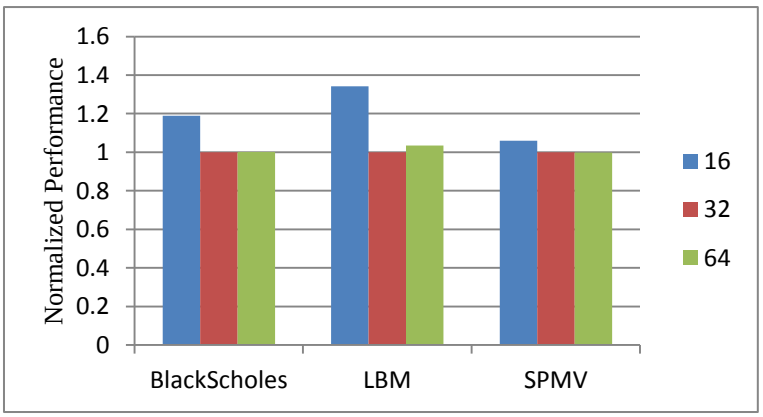

Fig. 7. The normalized performance of designs with different tile lengths (parameter *W*).

4.4 Performance effect when adding hardware delays

The proposed DLT framework uses hardware modules to perform the automatic data layout transformation. The experiments so far did not take hardware delays into account. Fig. 8 shows organization of the arithmetic units for data layout transformation. This paper models these modules in the GPGPU-Sim. The address translator hardware is added into the streaming multiprocessor (SM) of the simulator. The address of GTX480 is 32bit and therefore the parameter *K* is 5bit. In this case, the translator needs a divider, a multiplier, and three 32-bit adders. The divider needs to support 32-bit divided by 5-bit, and the multiplier should support 32-bit multiplied by 5-bit. The delay of the hardware is estimated based on the integer ALU from the GTX480 configuration file. The latency of the DLT is modeled as 75 cycles in the simulator. Note that the PCIe module between CPUs and GPGPUs is not implemented in GPGPU-Sim. Fig. 9 shows the performance when the hardware delay is concerned. As shown in Fig. 9, the proposed design has achieved up to 177% performance improvement.

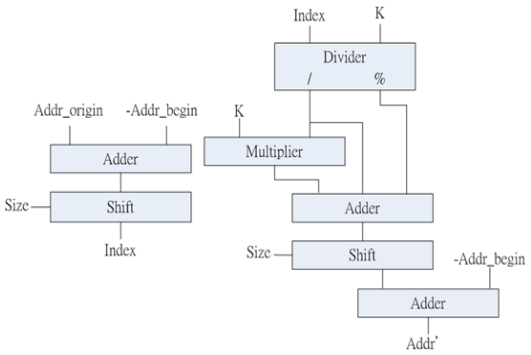

Fig. 8. Hardware architecture of address translator

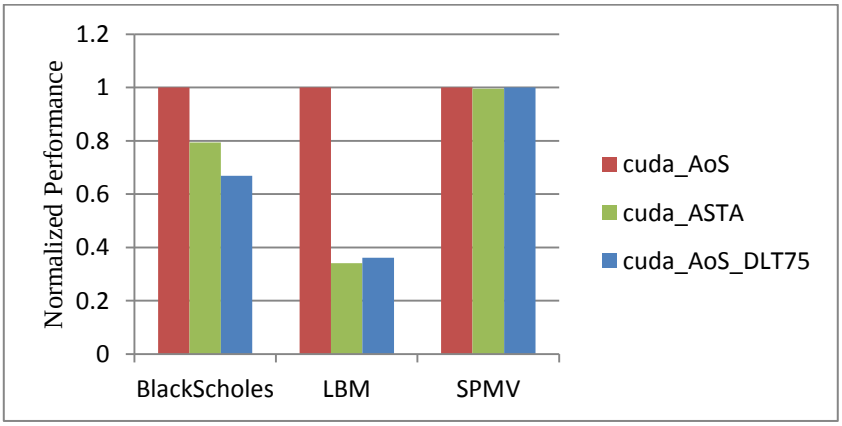

Fig. 9. The normalized performance of designs with hardware delays.

5 Conclusion

This paper proposes a fully automated data layout transformation framework to help streamline the design flow as well as alleviate the overhead. Our programmer-friendly framework is composed of Data Layout Transformer and specialized Load/Store Unit. Our proposed framework is evaluated using three different applications with multiple input datasets. The results have demonstrated to achieve up to 177% performance improvement while retaining good program readability and maintainability.

References

1. I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu., "DL: A Data Layout Transformation System for Heterogeneous Computing," In Proc. IEEE InPar, San Jose, May 13, 2012, pages 513–522, 2012.
2. Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli, "Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures" in Proc. IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 22, NO. 1, JANUARY 2011
3. S. Che, J.W. Sheaffer, and K. Skadron, "Dymaxion: optimizing memory access patterns for heterogeneous systems", in Proc. SC, 2011, pp.13-13.
4. L. Karlsson., "Blocked in-place transposition with application to storage format conversion." Technical report, 2009.
5. F. Gustavson, L. Karlsson, and B. Kagstr□m. Parallel and cache-efficient in-place matrix storage format conversion. ACM Transactions on Mathematical Software.
6. G. Ruetsch and P. Micikevicius. Optimizing matrix transpose in CUDA. January 2009.
7. S. Che, J.W. Sheaffer, and K. Skadron, "Dymaxion: optimizing memory access patterns for heterogeneous systems", in Proc. SC, 2011, pp.13-13.
8. "CUDA C programming guide" , Available: http://docs.nvidia.com/cuda/cuda-c-programmingguide/index.html
9. A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPGPU Simulator," Ispass 2009: Ieee International Symposium on Performance Analysis of Systems and Software, pp. 163-174, 2009.
10. Michael Garland, Scott Le Grand, John Nickolls, " PARALLEL COMPUTING EXPERIENCES WITH CUDA" , IEEE Computer Society, 2008.
11. "GPU Computing SDK" , Available: https://developer.nvidia.com/gpu-computing-sdk
12. "Parboil Benchmarks**" ,** Available**:**http://impact.crhc.illinois.edu/Parboil/parboil.aspx