

An access pattern based adaptive mapping function for GPGPU scratchpad memory

Feng Han, Li Li^{a)}, Kun Wang, Fan Feng, Hongbing Pan, Jin Sha, and Jun Lin

*School of Electronic Science and Engineering, Nanjing University,
Nanjing 210023, China*

a) lili@nju.edu.cn

Abstract: As modern GPUs integrate massive processing elements and limited memories on-chip, the efficiency of using their scratchpad memories becomes important for performance and energy. To meet bandwidth requirement of simultaneously accessing of a thread array, multi-bank design, dividing a scratchpad memory into equally-sized memory modules, are widely used. However, the complex access patterns in real-world applications can cause the bank conflicts which comes from different threads accessing the same bank at the same time, and the conflicts hinder the performance sharply. A mapping function is a method that redistributes the accesses according to access addresses. To reduce bank conflicts some scratchpad memory mapping functions are exploited, such as XOR based hash functions and configurable functions. In this paper, we propose an adaptive mapping function, which can dynamically select a suitable mapping function for applications based on the statistics of first block executing. The experimental results show that 94.8 percent bank conflicts reduced and 1.235× performance improved for 17 benchmarks on GPGPU-sim, a Fermi-like simulator.

Keywords: GPGPU, scratchpad memory, adaptive mapping function, bank conflict reduction

Classification: Integrated circuits

References

- [1] NVIDIA Corporation: NVIDIA CUDA C Programming Guide 6.0 (2014) <http://docs.nvidia.com/cuda/>.
- [2] AMD: CodeXL profiler 1.4 (2014) <http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-tools-sdks/codexl/>.
- [3] H. Vandierendonck and K. De Bosschere: “XOR-based hash functions,” IEEE Trans. Comput. **54** (2005) 800 (DOI: 10.1109/TC.2005.122).
- [4] G.-J. van den Braak, *et al.*: “Configurable XOR hash functions for banked scratchpad memories in GPUs,” IEEE Trans. Comput. **65** (2016) 2045 (DOI: 10.1109/TC.2015.2479595).
- [5] T. Givargis: “Improved indexing for cachemiss reduction in embedded

- systems,” Des. Autom. Conf. (2003) 875 (DOI: [10.1145/775832.776052](https://doi.org/10.1145/775832.776052)).
- [6] K. Patel, *et al.*: “Reducing conflict misses by application-specific reconfigurable indexing,” IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **25** (2006) 2626 (DOI: [10.1109/TCAD.2006.882588](https://doi.org/10.1109/TCAD.2006.882588)).
 - [7] H. Salwan: “Eliminating conflicts in a multilevel cache using XOR-based placement techniques,” High Performance Computing and Communications and 2013 IEEE International Conference on Embedded and Ubiquitous Computing (2013) 198 (DOI: [10.1109/HPCC.and.EUC.2013.37](https://doi.org/10.1109/HPCC.and.EUC.2013.37)).
 - [8] J. Fang, *et al.*: “Aristotle: A performance impact indicator for the openCL kernels using local memory,” Sci. Program. **22** (2014) 239 (DOI: [10.1155/2014/623841](https://doi.org/10.1155/2014/623841)).
 - [9] A. Bakhoda, *et al.*: “Analyzing CUDA workloads using a detailed GPU simulator,” IEEE Int. Symp. Perform. Anal. Syst. Softw. (2009) 163C174 (DOI: [10.1109/ISPASS.2009.4919648](https://doi.org/10.1109/ISPASS.2009.4919648)).
 - [10] S. Che, *et al.*: “Rodinia: A benchmark suite for heterogeneous computing,” Proc. IEEE Int. Symp. Workload Characterization (2009) 44 (DOI: [10.1109/IISWC.2009.5306797](https://doi.org/10.1109/IISWC.2009.5306797)).
 - [11] J. A. Stratton, *et al.*, “Parboil: A revised benchmark suite for scientific and commercial throughput computing” (2012) 1048 (DOI: [10.1109/IPDPSW.2012.128](https://doi.org/10.1109/IPDPSW.2012.128)).

1 Introduction

In past decades, graphics processing units (GPUs) have emerged as a popular platform for non-graphics computation. To improve the performance, modern general purpose graphics processing units (GPGPUs) employ hundreds of processing elements that cause considerable bandwidth for data transaction. Using the scratchpad memories with adequate number of banks can effectively improve the efficiency of data reuse. One key factor for GPGPU programmers to achieve such improvements is taking advantage of the on-chip software-controlled scratchpad memories, such as shared memory in compute unified device architecture (CUDA), a parallel computing platform and application programming interface model created by Nvidia, and local memory in OpenCL [1, 2]. To achieve high bandwidth, a scratchpad memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. Unfortunately, a proper use of these on-chip memories requires programmers to have a certain level of expertise.

Mapping functions are used in GPGPUs for mapping memory address into different banks to exploit the bandwidth of multi-bank memories and caches [3, 4, 5, 6, 7]. The main aim of these functions is to evenly spread the memory accesses of a running application to avoid bank conflicts in interleaved memory access collections. A simplest mapping function, select banks according to the least significant address bits, is used in regular architectures. However, for various access patterns in real-world applications, it is obvious that a specific mapping function could not always be appropriate. Thus, how to choosing a suitable hash function for a specific application is studied. The selection of bank indexing bits

can be performed by exhaustive search or with heuristics as is proposed in [4, 5]. Nevertheless, in their works lots of statistics and calculations are required before actual running applications.

In this paper we propose an adaptive mapping function to find a suitable mapping for applications based on the statistics of first block executing. In GPU architectures the memory spaces of scratchpad memory are independent of different thread blocks. Such architecture allows configuring different mapping function for blocks. The configurations are generated dynamically after first batch of blocks executed. The advantage of adaptive mapping is to reduce the number of bank conflicts without programmers' memory scheduling and additional computing. The main contributions of this paper are the following:

- Various access patterns of GPU applications are introduced, and a further analysis of access patterns are made.
- We propose a heuristic to discover a suitable mapping function according to the address collections of thread warps. Compared with isolated addresses, the address collections keep the relationship among the address requests at the same time.
- Based on the algorithm we develops a adaptive mapping function, which update its mapping function according to the first batch of blocks executing of the kernel.
- A study of the hardware cost of the proposed adaptive mapping function module is carried out. Based on a 40 nm standard cell library, the area and power costs for different target clock frequency are calculated.

The rest of the paper is organized as follows. First, Section 2 describes the motivation that adaptive mapping function is a way to avoid bank conflicts without software control or configuration before applications. In Section 3 a memory access pattern classification for scratchpad memory accesses is shown. Different classes mapping functions are introduced in Section 4, and our adaptive mapping function are introduced in Section 5. The hardware costs of adaptive mapping functions and results of the different mapping functions are presented in Section 6. Finally, conclusions are stated in Section 7.

2 Motivation

Scratchpad memory, as a high speed on chip memory can be controlled by programmers, plays an important role in increasing performance. Evenly distributing accesses to all banks is a crucial issue to ensure the bandwidth of scratchpad memories. Unfortunately, in GPU applications, the memory requests are varied, and access distribution is hardly to be even. To take the advantage of the bandwidth of multi-bank, programmers of GPU applications need to schedule the scratchpad memories carefully, and this requires sufficient GPU architecture knowledge which is unnecessary for software level programmers.

Using mapping functions to redistribute accesses is a simple and effective way to separate software and hardware works. A basic mapping function used in regular GPU is using the lower bits of address as bank indexes. XOR (exclusive or) based hash functions are often used as hash functions because of their relative good hash

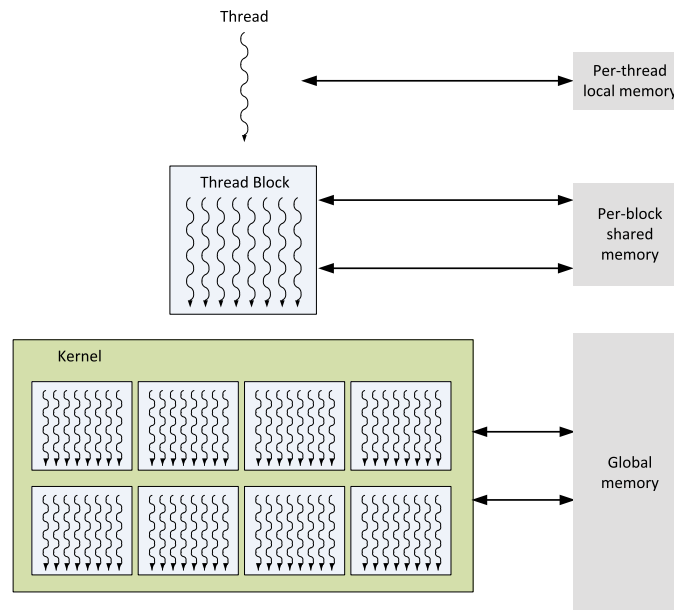


Fig. 1. Hierarchic architecture of GPU thread memory.

properties and their low computational costs. Although the overall performance is improved, in some cases the XOR hash function hampers the performance. Both the basic and fixed hash mapping functions are designed to be used in all situations, but different applications usually have different requirements of the mapping function. Therefore, a fixed mapping function cannot be suitable for all applications.

Some works try to choose a suitable hash function for a specific application [4, 5]. In work of Braak, a configurable XOR based hash function is proposed, and their results of experimentation show that their hash function can achieve bank conflict reduction. Nevertheless, for applications adopted different access patterns, a certain mapping function cannot always be useful for all patterns. The configurable hash function can provide proper method to different program with hardware profiles and software compile information. The configurable mapping functions still face some challenges. In some cases, an application has more than one kernels, and a kernel can also adopt more than one access patterns. So it is hard to find a suitable mapping function for complex memory access situations. Both the heuristic algorithms of Givargis and Braak calculate the hash functions from access addresses statistics of the entire applications executing. It is notable that the statistics through the whole process will hide the conflict information among the access collection of a thread warp.

As shown above, previous works still face some challenges. This encourages us to propose a dynamic adaptive mapping function that improves scratchpad memory efficiency without programmers' scheduling or off line calculation before applications. In GPU memory architecture, includes global, local, shared, texture, constant and so on, shared memory have much higher bandwidth and much lower latency than local or global memory. CUDA threads may access data from multiple memory spaces during their execution as illustrated by Fig. 1. Each thread has private local memory. Each thread block has shared memory visible to all threads of

the block and with the same lifetime as the block. All threads have access to the same global memory. A thread block is a group of threads which execute on the same multiprocessor. Threads within a thread block access to shared memory and can be explicitly synchronized. Take this advantage we propose a dynamic mapping module that automatically chooses a suitable mapping function, bases on the first batch of blocks, for the other blocks.

3 Scratchpad memory access patterns

In this section, we focus on scratchpad memory access patterns that mean a collection of addresses generated by one scratchpad memory transaction of a thread warp [8]. In [4], the author classified access patterns to linear, stride, block and random. Fig. 2 shows some examples of access patterns.

Linear is the simplest class where all memory accesses in a warp are consecutive.

Stride is an access pattern that every access is separated with a stride factor S . Note that linear is a special case of stride where $S = 1$. The Fig. 2b and Fig. 2c show the stride access with $S = 2$ and 3, relative prime of number of banks.

Block is a class including two dimension access pattern where the accesses are separated with a stride factor S , and each part have one more accesses. As shown in Fig. 2d, the access pattern is: 0, 1, 2, 3; 32, 33, 34, 35; ..., 98, 99, 100, 101.

Random is the last class and contains all cases which cannot be captured by the other classifications.

3.1 Mapping functions selection analyze of different access patterns

After access patterns have been classified, we try to analyze how the mapping functions impact different access patterns. The address bits, which changed in one collection of access, can help us to infer the type of access patterns. Hence, we analyze two statistic values about this. One is the changed bits in a collection of access, the other is the union set of changed bits of a collection in the entire kernel processing.

Linear patterns can be easily distributed to banks evenly by base function. According to [1], relative prime will not cause conflicts. Generally, the number of banks is an integral power of 2, if the stride is not the relative prime of number of banks, the common factor must be integral power of 2.

For block access patterns, when stride is integral power of 2 the changed bits can be used as mapping function like the stride access. It is complex to select bits from address bits when stride is relative prime of the number of banks.

3.2 Examples of access pattern classifications

In real cases, applications may adopt more than one access patterns. For example, in the FWT (fast Walsh transform), an application in cuda SDK, when off-chip memory data are written to shared memory and the result data written back, linear access patterns are used. However, the block accessing is used in computing period. To reduce the bank conflicts for more than one access patterns, the mapping functions need to take all the access patterns used in the kernels into consideration.

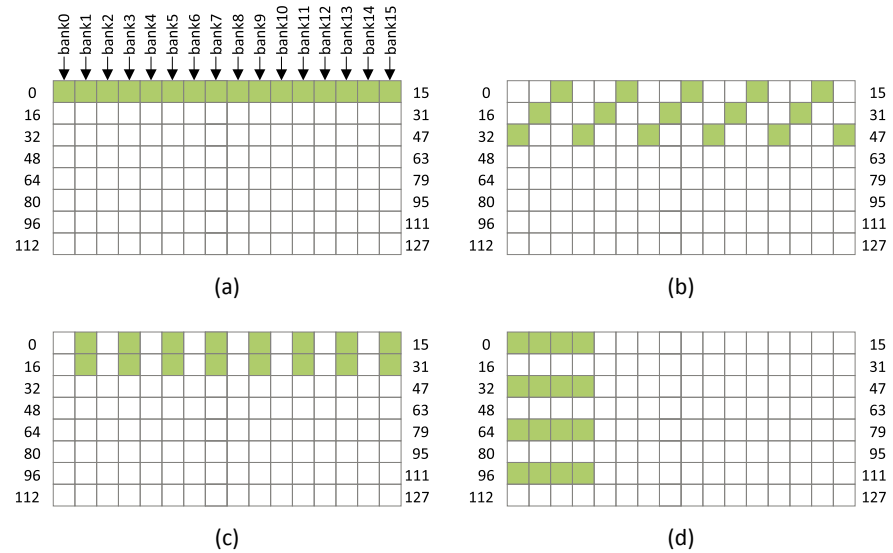


Fig. 2. Scratchpad memory access patterns.

To understand the access pattern in real-world benchmarks, the memory access classification is helpful. We analyze the applications with shared memory utilizing in three frequently used benchmarks. The access pattern of FWT is a typical hybrid access. Linear accessing is used by data transfer between off-chip memories and shared memories, and block accessing is used in computing period, as:

```
s_data[tx] = src_data[tx]
tx_lo = tx & lowerbitmask
reg_data = s_data[(tx - tx_lo) >> 2 + tx_lo]
s_data[(tx - tx_lo) >> 2 + tx_lo] = foo(reg_data)
dst_data[tx] = s_data[tx].
```

In above code, the `tx` is a thread identify, the `src_data` and `dst_data` are off-chip memory pointers, the `lowerbitmask` is a mask to select the lower bits, and `reg_data` is a data in register file.

4 Mapping functions

The goal of a hash function is to distribute the memory accesses over the different memory banks to avoid conflicts. However, a mapping function works well for some access patterns, but cause bank conflicts for other access pattern. Mapping functions can be divide into several classes. In addition, some heuristic algorithms are proposed to calculate a specific mapping function for certain applications.

4.1 Static mapping function

The bits selection is the simplest mapping function, such as, for N banks scratchpad memory, choosing $\log_2 N$ bits in address to select the memory banks. For example the basic mapping function described in Section 2 is a typical static mapping function. For these mapping functions, it is a reasonable scheme to select the lower bits as the bank indexes.

According to the analysis in Section 3 we find that spreading the accesses to all banks need more bits than the basic mapping function. The XOR operations are

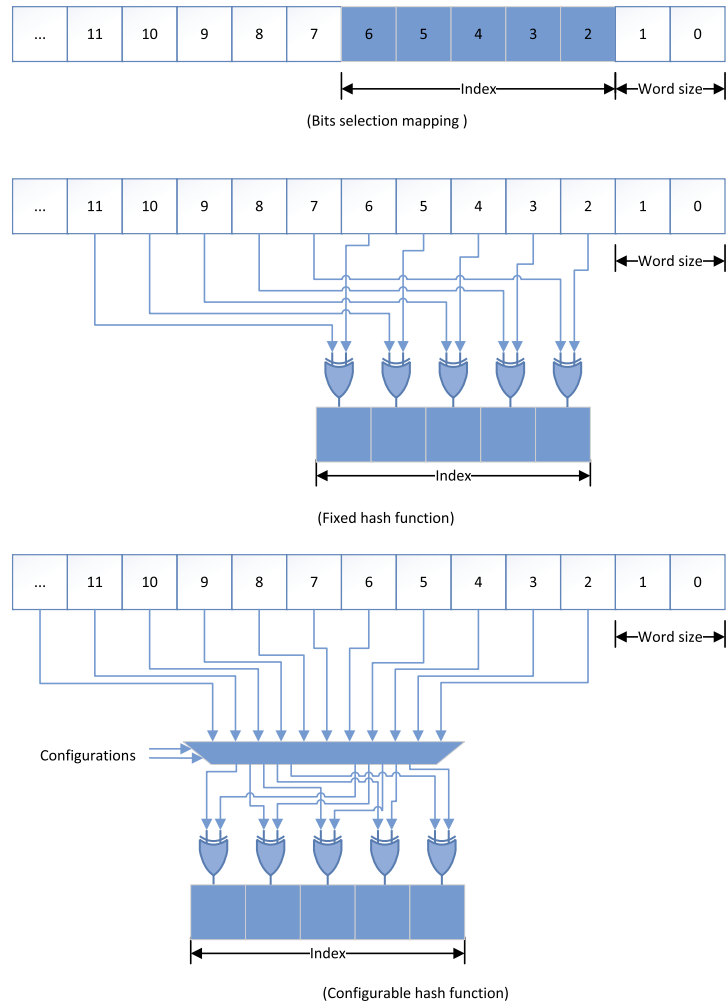


Fig. 3. Scratchpad memory mapping functions.

often used in mapping functions because of their good hash properties and low costs. Hashing two fixed address bit sets as mapping function can contain more information. However, on the other hand, hashed two bits together could cause additional bank conflicts in some cases.

Both the basic and fixed hash mapping functions are static mapping functions that fixed in hardware for all applications.

4.2 Configurable mapping function

As Fig. 3 shown, the basic mapping function and the fixed hash mapping function are implemented in hardware design. Unlike the static mapping the configurable mapping function can select different mapping functions according to configurations. Using configurable mapping functions is an approach to make the mapping function more flexible. With specific configurations applications can employ different mapping functions to find the most suitable mapping function.

Givargis introduced a heuristic algorithm to choose the best address bits to index a cache. A corresponding quality measure which is real number ranging from 0 to 1, is used to select the index bits. The quality measure is calculated by taking the ratio of zeros and ones of a certain bit.

Braak proposed the minimum imbalance heuristic (MIH) to minimize the imbalance of the accesses to banks. In their work, the access numbers to all addresses are recorded to compute the best address bits sequentially.

4.3 Access patterns based heuristic

Unlike achieve the configuration with extra computing, the adaptive mapping function searches for the best possible configuration with hardware logic while the kernel executing. Hence, the heuristics mentioned above are too complex to be implemented in hardware. We make some simplification for an access pattern based heuristic to be used in adaptive mapping. The changed bits of an access pattern are recorded to reduce the possible bits when update mapping function. We develop a adaptive algorithm, shown as algorithm 1, to find suitable configurations dynamically. When the first batch of blocks are executing, if new access patterns are detected it will be saved in a vector of access pattern. After that new configurations are searched based on all the access patterns in the vector.

5 Adaptive mapping function

A certain mapping function hardly suits for all of the access patterns generated by varies applications. Cooperated with heuristic algorithm configurable mapping functions can effectively reduce bank conflicts for a specific application. However, there are still some challenges to face for previous works. In the works of [4] and [5], the total access times of each addresses are used to calculate the mapping functions. In this way the conflicts of a collection could be covered by the final results of the entire process. Another difficulty to face is that addresses are large amount of data, so recording and processing these data is a huge overhead which is hard to be accepted by hardware designs. Furthermore, for these applications, which adopt more than one kernels, the configured mapping functions specified for a certain application could not be appropriate for all kernels. Take the above questions into account, we propose an adaptive index mapping function which is dynamically configured according to the first batch of thread block executing.

Generally, GPGPU applications have numerous of blocks which have independent shared memory data. In other words, different blocks will execute the same kernel instructions with different block identifier, so they employ similar memory operations.

Take the advantage of this view, the data are recorded while the first block executing to calculate a reasonable mapping function configurations for the later blocks in the same kernel. Unlike the Givargis or Braaks works, our adaptive mapping function automatically searching a suitable configuration while a thread block executing. The adaptive hash function bases on bitwise XOR approach to combine pairs of bits which are determined by a bit selection algorithm. A fixed hash mapping function is adopted as a base mapping function for the first batch of thread blocks. The selection algorithm of mapping function is described as Algorithm 1.

6 Experimental results

6.1 Hardware design and evaluation

Different from static mapping functions have a negligible latency, the configurable functions latency cannot be ignored. A module with additional logical costs is used in our adaptive mapping function to decide the configurations of mapping function. To evaluate the overhead, the proposed adaptive mapping function is implemented in Verilog hardware description language. As Fig. 4 shown, the latency, area and power cost are obtained by using Synopsys Design Compiler based on the 40 nm standard cell library of TSMC. Different from CPU (central processing unit), GPGPU's shader clock frequency most are between 1.2 GHz to 1.5 GHz. The figures in Fig. 4 show the target clock frequency from 1.2 GHz to 2.6 GHz and the clock period just below it. Take NVIDIA GTX 480 as an example, its shader clock frequency is 1.4 GHz and it instantiate 15 streaming multiprocessors. At 1.6 GHz clock frequency the hardware overhead of proposed module is 0.19 mm² and 10.2 mW. The total power consumption of the proposed mapping module is about 153 mW, and the corresponding area costs is 2.9 mm².

6.2 Bank conflict reduction comparison

To evaluate the impact of the adaptive mapping function, fixed, configurable and adaptive hash mapping functions are implemented in GPGPU-Sim version 3.2.0 [9], which is configured as an NVIDIA GTX 480 (Fermi) GPU. The benchmarks are taken from CUDA SDK, Rodina and Parboil [10, 11].

The bank conflict reduction of the adaptive mapping function and the other mapping functions are compared against the regular GPU which does not use a hash function in addressing the shared memory banks. This fixed hash function calculates the bank index as: $bank = addr[0 \dots 4] \oplus addr[5 \dots 9]$.

Algorithm 1 Mapping selection algorithm for scratchpad memory

Require: AP ▷ Access patterns get from kernel executing
Require: $AP_v = \text{NULL}$ ▷ A list of Access patterns of this kernel

- 1: **if** AP matches the i th AP in AP vector **then**
- 2: $count_i ++$
- 3: **else if** AP is not belong to the AP vector **then**
- 4: Add AP to AP vector
- 5: **end if**
- 6: $CB_v \leftarrow AP_v$ ▷ get the changed bits from AP
- 7: **for** $hash1 \in CB$ **do**
- 8: **for** $hash2 \in CB_v - hash1$ **do**
- 9: $bankconflict = \text{false}$
- 10: **for** AP = $AP_v.begin:AP_v.end$ **do**
- 11: **if** AP causes bank conflicts **then**
- 12: $bankconflict = \text{true};$
- 13: **end if**
- 14: **end for**
- 15: **if** No APs cause bankconflict **then**

```

16:     break;
17:   end if
18:   update hash2
19: end for
20: if No APs cause bankconflict then
21:   break;
22: end if
23: update hash1
24: end for
25: return index  $\leftarrow$  hash1  $\oplus$  hash2

```

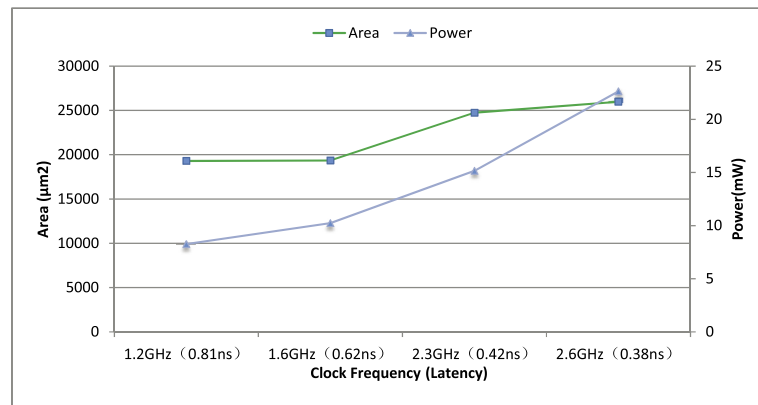


Fig. 4. Area and power versus latency results for a range of target clock frequency.

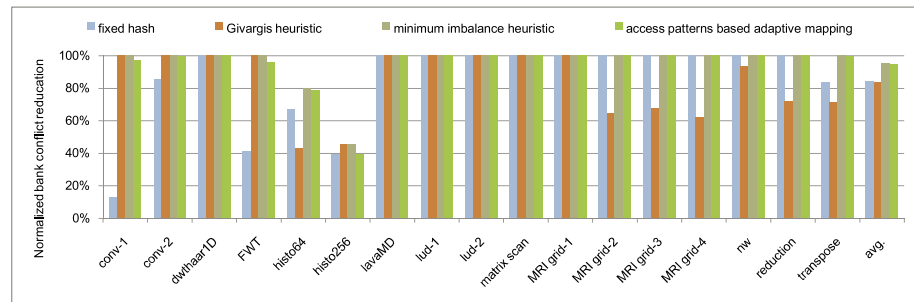


Fig. 5. Relative number of bank conflict reduction normalized by a basic mapping function of various mapping functions.

The bank conflict reduction of various mapping functions compared with a basic one is shown in Fig. 5. The MIH do the best in bank conflicts reducing, and the average value is 95.6 percent. After MIH the proposed adaptive mapping function get 94.8 percent bank conflicts removed with real-time configuration computing. Both without off line configuration, the proposed reduces 10.7 more percent bank conflicts than the fixed hash function. For histogram applications, a substantial part of memory access is dependent on the input data. Therefore the proposed adaptive mapping function does not find a suitable configuration for histo265 benchmark after the first batch of thread blocks executed, and keep

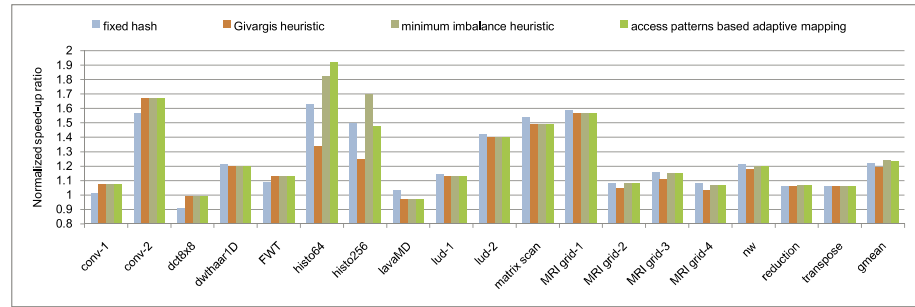


Fig. 6. Speed-up ratios normalized by a basic mapping function of various mapping functions.

using the fixed hash function for last blocks. Some applications do not cause bank conflicts with basic mapping function or the others, such as, back propagation, hotspot, sradd, and so on. It is worth mentioning that the fixed hash function causes bank conflicts in `dct8x8`. Hence, the adaptive mapping function also causes bank conflicts at the first batch of blocks executing, but it get a conflict free mapping after the first batch of blocks finished.

The speed-up obtained by the hash functions over a baseline GPU is shown in Fig. 6 for a set of benchmarks. Executing a kernel takes a lot of other operations besides scratchpad memory accessing. Therefore, the speed-up difference of various mapping functions is not as obvious as bank conflict reduction. The geometric mean of the speed-up ratios for the fixed hash and Givargis heuristic mapping functions are $1.218\times$ and $1.206\times$, respectively. The MIH performs best with $1.241\times$ and the proposed mapping function gets $1.235\times$ speed-up. For some applications both the MIH and our proposed mapping functions reduce more bank conflicts and get better performance.

For some applications which get bank conflicts in a small part of scratchpad memory accessing, flexible mapping functions experience a slowdown due to an extra cycle to configure the functions, such as the `lavaMD` benchmark in Fig. 6. For the benchmarks that get well bank conflict reduction both from fixed and flexible hash functions, the fixed hash function outperforms the others. Because extra bank conflicts are generated, the fixed hash function causes about 9 percent performance loss for the `dct8x8` benchmark. Scratchpad memory accessing is the main operation in histogram algorithm and the memory locations are related to the input data. Therefore, the `histo64` and `histo256` get outstanding improvement by bank conflict reduction.

7 Conclusions

In this work we analyze the typical memory access patterns, and bases on the merit of massively thread computing propose an access pattern based adaptive mapping function for scratchpad memory. The adaptive mapping function can automatically search for suitable mapping configurations according to information from the first batch of thread blocks executing. To evaluate the overhead an RTL module is implemented. The improvement of bank conflict reducing are tested on 17 benchmarks with bank conflicts. Take 1.2 GHz clock frequency as an example, this

adaptive mapping function can get 94.8 percent bank conflict reduction and $1.235\times$ speed-up with 0.2 mm^2 area and 8.2 mW power cost for each streaming multi-processors.

Acknowledgments

This work was supported by the National Nature Science Foundation of China under Grant No. 61176024; Research Fund for the Doctoral Program of Higher Education of China under Grant No. 20120091110029; The project on the Integration of Industry, Education and Research of Jiangsu Province BY2015069-05; The project Funded by the Priority Academic Program Development of Jiangsu Higher Education Institutions (PAPD); Collaborative Innovation Center of Solid-State Lighting and Energy-Saving Electronics; And the Fundamental Research Funds for the Central Universities.