12-2014

Improving GPU Shared Memory Access Efficiency

Shuang Gao
University of Tennessee - Knoxville, sgao3@vols.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Shuang Gao entitled "Improving GPU Shared Memory Access Efficiency." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Gregory D. Peterson, Major Professor

We have read this dissertation and recommend its acceptance:

Michael W. Berry, Michah Beck, Charles Collins

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Improving GPU Shared Memory Access Efficiency

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Shuang GAO

December 2014

DEDICATION

I dedicate this dissertation to my family and the faculty and staff who have supported me on every step of this journey.

ACKNOWLEDGEMENTS

I would like to thank Greg Peterson for being a fantastic advisor, my committee members, and NSF for funding much of my graduate career.

Soli Deo Gloria

ABSTRACT

Graphic Processing Units (GPUs) often employ shared memory to provide efficient storage for threads within a computational block. This shared memory includes multiple banks to improve performance by enabling concurrent accesses across the memory banks. Conflicts occur when multiple memory accesses attempt to simultaneously access a particular bank, resulting in serialized access and concomitant performance reduction. Identifying and eliminating these memory bank access conflicts becomes critical for achieving high performance on GPUs; however, for common 1D and 2D access patterns, understanding the potential bank conflicts can prove difficult. Current GPUs support memory bank accesses with configurable bit-widths; optimizing these bit-widths could result in data layouts with fewer conflicts and better performance.

This dissertation presents a framework for bank conflict analysis and automatic optimization. Given static access pattern information for a kernel, this tool analyzes the conflict number of each pattern, and then searches for an optimized solution for all shared memory buffers. This data layout solution is based on parameters for inter-padding, intra-padding, and the bank access bit-width. The experimental results show that static bank conflict analysis is a practical solution and independent of the workload size of a given access pattern. For 13 kernels from 6 benchmarks suites (RODINIA and NVIDIA CUDA SDK) facing shared memory bank conflicts, tests indicated this approach can gain 5%-35% improvement in runtime.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

In recent two decades, graphical processing unit (GPU) evolved from a graphics-oriented processor to a general-purpose parallel processor. NVIDIA CUDA (Compute Unified Device Architecture) and OpenCL[1] are two commonly used GPU programming models. Through such programing models, many HPC applications and libraries can exploit GPU accelerators to obtain performance improvements.

When developing GPU kernels, optimizing memory access efficiency is one of the main schemes for improving execution performance [2]. Among the different memory types defined in the CUDA programming model, shared memory plays a key role as a software manageable on-chip storage. As figure 1 shows, a shared memory buffer is allocated for one thread block and all threads in this thread block have access to it. The access latency of shared memory is much less than GPU device memory. Normally, shared memory is used for caching data to improve temporal locality [3], holding the data shared inside one thread block [4], and being temporary storage for data layout transforms to achieve better global memory performance [5]. A primary concerns of using shared memory is the penalty of potential bank conflicts for different memory access strides [1, 2].

CUDA shared memory is organized into banks. The bank mapping function is based on conventional low-order bank mapping [6], which maps n successive words to n successive banks. To improve bank access efficiency for different data types, it supports dynamic configurable bank access bit-width [7]. For example, 32 of 4-Byte elements can be uniformly mapped to 32 banks; and 32 of 8-Byte elements can also be uniformly mapped to these banks. In addition, multiple accesses to the same layer of the same bank cause no conflict.

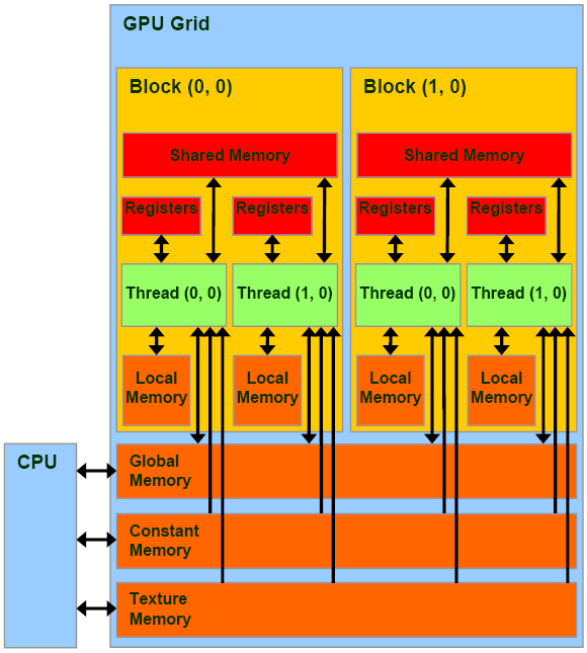


Figure 1.1 CUDA programming model and memory hierarchy [1]

Bank conflict analysis of interleaved memory has been well studied since the 1970s [8-14]. When two accesses of the same bank occur inside the period of the bank response latency, the conflict happens and the two requests are processed serially. For non-vector accesses, by instruction scheduling and adding buffers, bank conflicts can be reduced [15, 16]. For parallel access, stride analysis is necessary to deal with bank conflicts [15, 16]. Based on stride analysis, many solutions have been proposed to obtain better support for different strides. CUDA shared memory bank organization is explained in programming guide. When designing CUDA kernels, developers should be aware of potential bank conflict issue and they can reduce or eliminate conflicts by modifying data layout, or changing memory access pattern. Researchers presented different data layout transformations to deal with shared memory bank conflict problem [3, 5, 17-19]. Among them, array padding is the easiest and most frequently used [5, 18] .

Array padding has been used to solve varies issues related to memory access efficiency. Typical cases include cache conflicts, false sharing, and bank conflicts. There are two types of array padding: inter-padding and intra-padding [20]. Inter-padding adds dummy space between array variables; intra-padding inserts unused spaces inside one array. To deal with bank conflicts problem, intra-padding can be used to change the array

access stride and in turn impacts the conflict degree. Normally, CUDA kernel programmers try to use different padding sizes and choose the one that causes the least bank conflicts.

The motivation of this dissertation is to improve CUDA shared memory bank access efficiency. In the CUDA parallel execution model, since the memory access pattern of one grid's (1D to 3D) access requests is relatively complex, shared memory bank conflict analysis is not obvious. A warp is the CUDA hardware parallel execution unit; it includes a small group of threads. Based on the shape of a thread block, the threads in a warp could be organized in a 1D vector, 2D array, or even 3D array. Different transformations of these warp shapes can be mapped to arrays stored in memory, resulting in different memory access patterns.

This thesis analyzes CUDA shared memory bank conflict for 1D and some 2D array access patterns, and proposes a heuristic optimization solution. (1) Given an access pattern and a hardware-supported mapping function, the bank conflict degree is evaluated. (2) To reduce or eliminate bank conflicts, a heuristic scheme seeks an improved data layout through optimizing parameters of inter-padding, intra-padding, and different bank mapping function configurations. (3) Finally, the source code is transformed according to the chosen solution.

The contribution of this work includes:

- 1D/2D stride and bank conflict analysis of dynamic bank addressing;
- A method that calculates the overall conflict number of one pattern within a limited period which doesn't depend on memory access workload size of this pattern;
- Given the shared memory space limitation of one thread block, a model driven heuristic parameter optimization method that looks for a solution in a potential parameter optimization space:
    - Bank mapping functions supported by hardware dynamic bank addressing;
    - Inter-variable-padding and changing variable definition sequences; and
    - Array intra-padding size.

This thesis is organized into 8 chapters. Chapter 2 describes the prior work related to interleaved memory bank conflict analysis, memory access pattern study, and padding

related schemes. Chapter 3 introduces the framework of the proposed approach. Chapter 4 describes the single vector/warp conflict analysis, which is the kernel module of the work. In CUDA programming model, one memory access expression drives concurrent threads to access a sequence of data in parallel. Normally these threads belong to multiple vectors (warps). Chapter 5 takes single expression as a unit and analysis it bank conflict. Chapter 6 presents the parameter optimization strategy for inter-padding, intra-padding, and bank access bit-width configuration. Chapter 7 briefly lists the applications used for experiments and chapter 8 presents the experimental results.

CHAPTER 2 PRIOR WORK

2.1 Introduction

This chapter studies three research areas that are related to the proposed project. We studied the prior works of interleaved memory bank conflict solutions, and then investigated the common 1D and 2D memory access patterns. Finally, we studied the prior works of padding. The proposed tool of this work uses padding as one of the main data layout transformation methods.

2.2 Interleaved Memory and Bank Conflict

Interleaved memory is used to improve memory throughput by dividing memory into multiple modules/sections/banks to allow them to work simultaneously. This is especially straightforward for vector processors; it enables parallel access to memories. [9, 21]

Bank conflict is one of the main concerns for designing efficient, interleaved memory. It occurs when multiple concurrent memory requests are issued to the same module/bank. In such a situation, the bank has to serve one request after another, degrading performance.

Bank conflicts exist in various system/processor designs. A vector processor has a bank conflict when one vector access request operates on data in the same banks. An example system is the Cray-1 [9]. It had 16 banks, and each bank was 64-bits wide. This design had bank conflicts when the access stride size was 8 or 16 words. Superscalar processors such as Intel's Sandy Bridge [22] also have the same issue when multiple memory accesses are grouped together. Normally some software/hardware modules are added to reduce or eliminate conflicts. Bank conflicts also impact other types of processors. VLIW processors, basically rely on compile time schemes to reduce the impact of bank conflict [23]. Multimedia processors and other application driven

hardware design also have bank conflict concerns due to their memory access patterns [24, 25].

Many hardware solutions have been proposed to deal with the memory bank conflict problem. Most existing solutions define a better bank mapping function that can support as many different strides as possible or provide perfect support for some special access patterns. Normally these bank mapping functions are designed based on the knowledge of some frequently used access patterns, such as the ones appearing in linear algebra calculations. Normally the number of banks is a power of two. Some researchers proposed using a prime number of memory banks to reduce bank conflict [10, 11]; however, prime number arithmetic is hard to implement in hardware. Some other research targets the bank conflict caused by multiple memory access instructions. Some well-designed scheduling schemes and extra buffers are used to avoid conflict or reduce bank conflict impact[15, 16].

In addition to hardware solutions, software solutions can also be used to reduce bank conflict. For example, by changing instruction sequences generated by a compiler [22], the memory operation instructions that cause bank conflict can be separated. From a high level programming perspective, changing memory access patterns in source code can also help to avoid or reduce bank conflict. [26]

Conventional Low-order Bank Mapping Scheme and Analysis

As shown in Figure 2.1, a conventional mapping function maps array elements sequentially on to N banks. The function can be described as $bank\_id[i] = \mathrm{mod}(i, N)$. This function maps the ith word of data on to bank of $bank\_id[i]$. The bank conflict degree for different 1D access strides can be determined by $\gcd(S, N)$, in which N is the number of banks and S is the constant stride value. For a system which has power-of-two banks, the result of $\gcd(S, N)$ equals to 1 for any odd stride. In other words, odd stride access of N words has no bank conflicts. However, for even strides, the conflict exists because $\gcd(S, N)$ does not equals to 1. For example, when the stride is 2, the conflict is 2-way conflict since $\gcd(2, N) = 2$.

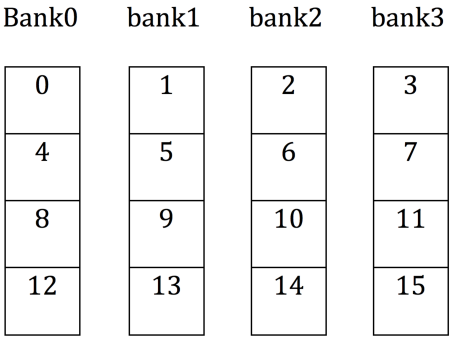


Figure 2.1 Conventional bank mapping

The notation "stride family"is defined for stride analysis purposes [4]. Basically, a stride family is described as $S = \sigma \times 2^e$, where σ is the sequence of odd numbers, and e is an integer which denotes a distinct family. For example, $S = \{2,6,10,14,18,...\}$ is the family with $e = 2$. For conventional mapping functions on a system with power-of-two banks, the stride family indicates the degree of conflict. For example when $e = 2$, the conflict is always 2. Oed and Lang presented detailed conflict analysis of conventional mapping function in [12].

GPU shared memory is based on a conventional low-order bank mapping function [2, 18]. The bank number is a power-of-two and the parallel access request number is the same as the bank number. To improve flexibility, it supports dynamic bit-width bank access. To avoid bank conflicts, programmers need to organize the data in proper ways. The most common ways to deal with bank conflict are: (1) choose a proper access mode provided by the GPU programming model [27]; (2) add padding to change access strides [5, 18]; (3) change the array index functions to make vector access uniformly mapped to every bank; and (4) algorithm level memory layout optimization [3]

Variant Mapping Schemes

Various bank mapping functions have been proposed to solve bank conflicts. The two main categories are XOR schemes [28] and skewing schemes [10, 29].

The **XOR** scheme was first proposed by Failong [13]. This scheme normally has the form of a linear transformation: $y = Tx$. Its input is an address x which is represented as a bit vector. This transformation maps x into another bit vector y which is the address

to which the data is mapped to. Some bits of such bit vectors indicate the bank index. The matrix T is the transform matrix. Each element in T is one-bit of data and its value is 0 or 1. This transform is realized by modulo 2 arithmetic, which is easy for hardware implementation.

The XOR scheme has better flexibility compared to skewing methods. By changing T, XOR can support different mapping strategies. For example, some existing transformations are designed to generate pseudorandom numbers to realize uniform data distribution across banks [14, 30]; some other transformations produce periodic sequences that can avoid bank conflict for some strides or access patterns [31].

Based on the XOR scheme, Gou presents SAMS [32] to support some stride families for 1D access. This method uses the XOR transform to reduce conflict degree and then increase the bank bit-width to remove remaining conflicts. This work also presents a 2D scheme 2DSMM, that uses two bank mapping functions, T_h and T_v, to support some 2D access patterns such as unit-stride/stride visit of row, column, diagonal, and block.

Harper proposed a dynamic strategy based on the XOR method [33]. Given a known stride, a proper XOR transform is selected to meet requirements.

The **Skewing** method was presented by Budnick and Kuck [10]. Normally it realizes conflict-free access for a subset of strides. Shapiro presented a review of the skewing method [29]. Basically a skewing method can be described as a linear mapping that maps consecutive data to banks resulting in less bank conflicts. Since no single skewing method can eliminate conflict for all different strides, many skewing methods have been proposed to support different stride types [29] [34-36].

No single skewing method can support all strides. Instead of eliminating all bank conflict, Harper proposed a skewing-based solution that optimizes overall performance by reducing bank conflict [37]. In addition, Harper presented a dynamic strategy based on the skewing method [8]. Based on known stride knowledge, this approach chooses a proper skewing scheme to eliminate conflict. Aho et al [38] presented a runtime changeable skewing method, which determines the skewing scheme based on runtime stride information.

Some existing skewing schemes target 2D/3D access patterns [37, 39-41]. Harper et al [37] analyzes skewing performance for some commonly existing stride types from

linear algebra applications. Kaufman et al [40] presented a skewing method that supports 3D vector access of 26 different directions. To support different sub-array patterns in 2D space, Liu et al [41] uses linear skewing in the horizontal direction and non-linear skewing in the vertical direction.

Bank Conflict for Multimedia Processors

Multimedia accelerators generally require high memory bandwidth due to parallel 2D access patterns and fast multimedia processing speed. How to avoid or reduce bank conflict for these 2D-stride access patterns has been investigated to improve memory system performance. Some solutions are provided based on 2D-stride access analysis. Kuzmanov et al [24, 25] presented a parameterized pattern for a type of 2D parallel access; an interleaved memory organization is proposed accordingly. Different parameter configuration patterns use different bank mapping functions. Lentaris et al [42] presented a non-linear skewing based method to achieve efficient memory access for some 2D access patterns. This work also optimized the bank access efficiency for a typical correlation existing among consecutive parallel memory access requests for image access.

GPU Bank Conflicts

In the past decade, the traditional graphics process unit (GPU) architecture was adapted to support general purpose computing and became widely used for massively data parallel computing. Memory system efficiency is crucial for this massive parallel device. Sung et al [43] presented a bank conflict study of GPU global memory access. Using micro-benchmark with different access strides, they studied the bank conflict characteristics of global memory. By combining this knowledge with analysis of application memory access stride information, they could find optimized data layout transforms to improve access efficiency.

The bank conflict problem is a primary issue for making efficient use of GPU shared memory [1, 2, 27]. This memory is composed of power-of-two banks. Based on the conventional low-order mapping scheme, different generations of GPU devices have

different mapping functions. The newest improvement is dynamic bank access mode [7]. This mode provides different bit-width access mode to all banks. The motivation of this design is to support efficient parallel memory access for data types with different bit-widths. For example, when bank number equals to N, if data type size is 4-Byte, N sequential elements are mapped to N different banks; if data type size is 8-Byte, N sequential elements can still be mapped to N different banks. For both cases, there is no bank conflict.

Like conventional memory module mapping, these consecutive mapping functions have similar bank conflict issues. However, since they support different consecutive mapping strategies, they have different conflict characteristics regarding different stride size [27]. Moreover, unlike conventional mapping methods, these strategies sometimes have conflicts due to bank offsets of the base address (the address of the first element that is visited). Examples are described in chapter 4.

The GPU programming model leaves the shared memory bank conflict problem to programmers. With the knowledge of different mapping functions, programmers need to design their data layout carefully to achieve efficient data access to the banks. For many applications, it is not easy to understand how data are mapped to banks. Generally, programmers try to add a small padding to change the access stride, or redesign the data organization to improve the efficiency.

In chapter 4, we will discuss the stride analysis of this bank mapping solution.

2.3 Memory Access Patterns and Strides

For scientific computing applications, array access pattern analysis is very helpful for improving memory access efficiency. Related compile-time optimizations include loop transformation [44], loop prefetching [45, 46], and array padding [47-50]. Besides general optimization for various array access patterns, some previous work provides automatic analysis and optimization for code that have similar array access patterns [51]. Jaejer and Barthou proposed a stencil kernel generator which is based on access pattern analysis [50], it searches for better data layout transforms to improve memory access

efficiency. Sung et al [43] presented an automatic data layout transform scheme based on common access patterns of PDE solvers and structured grids. A source-to-source subscript transformation module is designed accordingly.

Program level optimizations regarding access pattern and memory efficiency have been well studied for different memory systems. Lee et al [52] summarized common array access patterns of typical applications. Corresponding optimization advice is presented as well.

For interleaved memory, bank access stride patterns directly impact the parallel access efficiency. Besides bank mapping functions, for linear array data layout (as in FORTRAN, C/C++), the knowledge of array access patterns is crucial to obtain the bank access stride information. For vector processors and multimedia/graphics accelerators, bank access efficiency directly depends on 1D/2D array access patterns.

Two aspects determine array access patterns: array definitions and array sub-indexing functions. Determining how to extract array access patterns from source code and properly represent them is the first step. The polyhedral model uses matrices to represent the sub-indexing functions that are based on loop iterator variables [18]. Each sub-indexing function is linear combinations of iterator variables. Sung et al uses a similar way to represent array sub-indexing for GPU parallel thread access [43]. In this work, we continue to use their representation for sub-indexing functions.

Commonly used array access patterns in linear algebra applications include row, column, backward/forward diagonal, and block. These patterns are generally considered when designing interleaved memory for vector processors. For multimedia processors/accelerators, 2D access patterns are more common. They include different block based patterns [25, 42] and even regular sampling patterns [42]. Please refer Figure 2.2, Figure 2.3, and Figure 2.4.

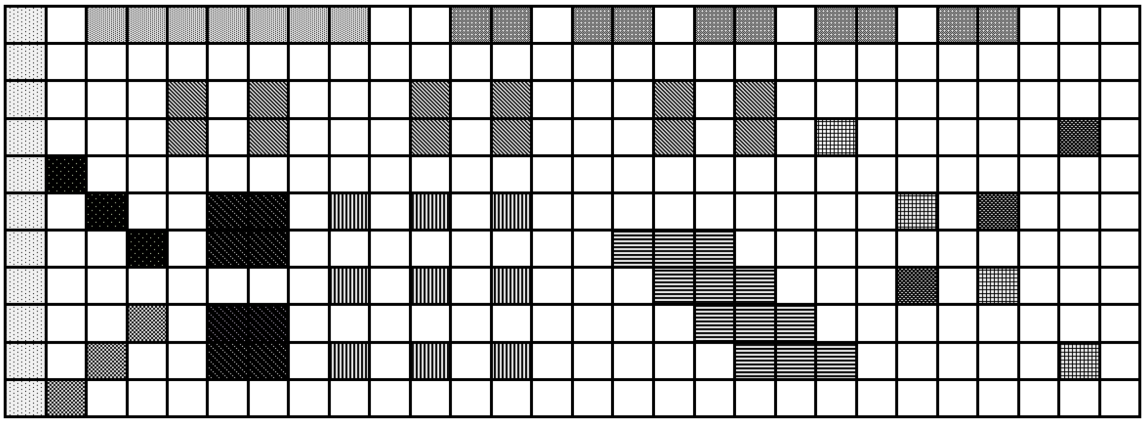


Figure 2.2 Typical access patterns

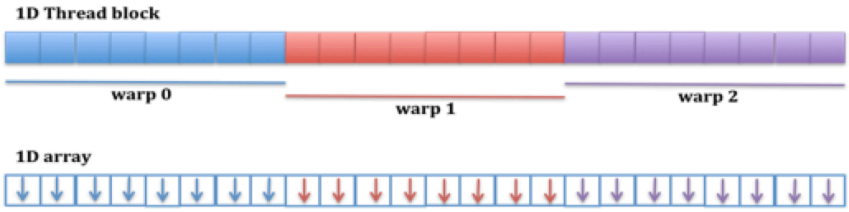


Figure 2.3 Memory access by 1D warp

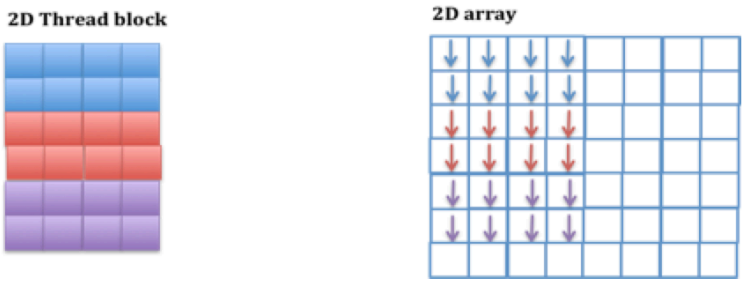


Figure 2.4 Memory access by 2D Warp

The CUDA GPU programming model supports parallel memory access by executing one parallel memory instruction for a group of threads. Concurrent threads can be organized in 1D to 3D grids. It makes thread execution structure match the array dimension and the code becomes easier to be manipulated. Beneath this structured

parallel execution model, GPU hardware executes instructions in units of warps, which are a thread array of 16 or 32 threads. Each thread block is composed of one or more warps. Threads in one thread block are linearly mapped to a sequence of warps, and each warp could be mapping to a 1D to 3D array of threads. Figure 2.3 presents two basic cases of using thread block to visit arrays. Fig. 2.3 (a) is a 1D thread block composed of 3 warps; when it visits a 1D array, each warp visits a 1D sub-array. Fig 2.3 (b) is an example for 2D warps. Each warp visits a 2×4 block in the 2D array. From these basic examples, it can be seen that for GPUs, the thread grid structure is another factor that impacts data access patterns.

The CUDA GPU memory system is composed of different types of memory. Various constraints are imposed on the programmer to obtain high memory access efficiency. GPU global memory is fixed length (such as 32-Byte or 128-Byte) vector access depending on cache or related configurations. These memory operations could achieve maximum bandwidth when the access patterns guarantee that a sequence of threads access consecutive data elements and they are properly aligned. Baskaran et al [18] presented code transforms based on Polyhedral models. Che et al [53] designed a set of APIs to reorganize the data to improve global memory access efficiency. Extra GPU kernels are used for data layout transform and memory access patterns are changed accordingly.

As previously mentioned, GPU shared memory bank conflict is a primary concern when designing GPU kernels and their data access patterns. How to avoid shared memory bank conflict is left to programmers to solve. To achieve better access efficiency, array access patterns need to be carefully designed to avoid or reduce bank conflict. Baskaran et al [18] presented a heuristic searching method to deal with this issue. It searches for the best padding width by examining bank conflict with the function $\gcd(stride, bank\_number)$. However, this solution is not enough. First of all, it only considers the basic conventional low-order mapping function. For mappings that support dynamic memory bank access, it requires a broader analysis scheme to estimate the bank conflict degree. Secondly, this solution cannot support other memory access patterns/strides besides constant 1D stride. This is not practical regarding the various access patterns used in GPU kernels [54].

Based on analysis of single parallel memory access patterns, the correlation information among consecutive parallel accesses can also be used to achieve better efficiency. In [42], after presenting the design of the bank-mapping scheme, Lentaris and Reisis also presented the definition of "MacroSquares", the area visited by a sequence of correlated memory accesses. He demonstrated that the proposed schemes work well for such a group-access pattern.

Beside memory access optimization for each GPU memory type, Jiang et al [54] proposed a scheme to choose among different memory types according to memory access pattern analysis.

2.4 Padding Transformations

Array padding is a commonly used method for data layout transformation. It is very useful for dense numerical algorithms such as linear algebra and iterative solvers. Padding is also a common optimization scheme adopted by compilers[55] . By adding unused spaces, the related memory access patterns are changed to improve memory visit efficiency. There are two types of padding: inter-padding and intra-padding [20]. Inter-padding adds dummy space between array variables; intra-padding inserts unused spaces inside one array.

Padding is commonly used for improving memory usage efficiency. Cache/TLB conflict is one of the problems that array padding can be applied to [56-62]. By adding extra blank space at proper locations, the cache conflict due to memory accesses can be reduced. False sharing is another example. By adding padding, data near each other that cause the false sharing can be separated [63, 64]. Array padding is also an important method to deal with the memory bank conflict problem [26, 47-49]. By adding padding in one of the inner array dimensions, the memory access stride is changed, which has direct impact on concurrent bank access patterns. Taking conventional bank mapping as an example, when padding is added to make a stride change from an even number to an odd one, gcd($stride$,$bank\_number$) equals to 1 and conflicts are eliminated. By adding padding before an array definition, it changes the offset of the array's base address. For

some bank mapping functions such as the ones supported by GPU dynamic bank access scheme, changing this offset might impact the bank conflict degree.

Array padding is easy to apply and it is practically efficient. In addition, unlike some other data transform methods, padding normally involves no extra source code transformation for array sub-indexing functions. Although it consumes some extra spaces, many problems can be solved with a relatively modest padding. Padding is generally used combined with other optimization schemes such as tiling [65-69], and prefetching [20, 70]. For example, Rivera et al presented a combination solution, which use tiling and padding to improve memory efficiency for some 3D iterative solvers [65]. In [68], a method combining intra-padding and tiling is used and proved to be efficient for matrix multiply. In [70] padding is used to avoid or reduce prefetching conflict.

Various automatic padding solutions are proposed for different purposes. Many of them are based on problem modeling [49, 57, 58, 65, 67], and then use heuristic methods [20, 71] or other searching methods [47, 60, 69] to find optimized solutions [66]. In [71] the author raised concerns with the relation between applying intra and inter padding; their solution always applies intra-padding prior to inter-padding.

Array padding is also commonly used for GPU program optimization. The two main purposes include improving global memory efficiency and reducing shared memory bank conflict. Based on the coarse-access principle for global memory access, padding could be used to transform data layout in global memory and coarse global memory access requests [72, 73]. Based on the GPU shared memory bank organization knowledge, small amounts of padding could be used to change memory access stride and in turn reduce or eliminate potential bank conflict [18, 19].

A compiler auto-padding solution for shared memory is proposed in [18]. A heuristic searching algorithm is used to search for a proper padding. For each candidate padding size, gcd($stride$,$bank\_number$) is evaluated to get the conflict degree for each parallel array access. The padding size that results in the least conflict number will be used. Some other auto-padding schemes are developed for application-domain related optimizations. Jaeger et al [50] proposed an auto-padding scheme for stencil calculations. This work extends the usual padding into a multi-padding method, which uses non-

uniform but periodic padding at different locations. This extended padding method is shown to be efficient for alignment issues on different CPU/GPU architectures.

2.5 Summary

This chapter presents three areas of previous research that are related to this dissertation. First of all, since this work target GPU shared memory bank conflict, we studied the prior work on solving interleaved memory bank conflict solutions. Secondly, we investigated common 1D and 2D memory access patterns, which is widely studied for automatic optimization techniques. In order to transform source code to obtain better efficiency, the proposed tool needs to be able to recognize common patterns and find proper data layout solution. Finally, we studied the prior work on padding. Although it is a basic and simple optimization, it is commonly used for data layout transform, especially for GPU shared memory optimizations. The proposed tool uses padding as one of the main data layout transformation methods.

CHAPTER 3 PROPOSED APPROACH

3.1 Background and Motivation

CUDA shared memory is software manageable on-chip storage, it is faster than device memory and its size is limited. Shared memory is commonly used for the following purposes: (1). Caching data to improve temporal locality. For the data which are visited multiple times, they can be cached in shared memory to avoid the long latency of global memory access. (2). Hosting data shared among threads of one thread block. An example is producer-consumer warps inside one thread block; they can communicate through shared memory. (3). Temporary storage for improving global memory access efficiency. In the kernel of matrix transpose, by using shared memory, access pattern of global memory can be changed to unit-stride row-major access. This helps to improve the performance by optimizing global memory access efficiency.

Bank conflict is a primary concern when using CUDA shared memory. Programmers are responsible to reduce or avoid bank conflict given the bank organization information [1, 2]. In earlier generations, CUDA shared memory used conventional low-order mapping; the value of bank number (denoted as $bank\_num$) equals to the vector access length (denoted as vec_length). Programmer could use $\gcd(stride, bank\_num)$ to calculate the bank conflict degree and use array padding or other data layout transformation to avoid bank conflicts. However, the NVIDIA Kepler GPUs enables dynamic bank access mode, which is designed to improve efficiency for different bit-width accesses. This makes the bank conflict issue more complex for programmers. Traditional analysis methods of conventional mapping are not enough to solve the bank conflict problem for new bank access modes. Generally it is not obvious to understand how data layout causes bank conflicts; people just try different padding sizes or choose different mapping access bit-width settings. On the other hand, shared memory space is limited. When changing data layout to solve the bank conflict issue, the space constraint needs to be considered. Otherwise, device occupancy might decrease and performance might drop significantly. Based these observations, we believe more effort

should be invested to reduce the difficulty of improving shared memory access efficiency.

This dissertation studies bank conflict issues of CUDA shared memory that supports dynamic bank access. Based on a generalized description of the bank organization and its access policy, the bank conflict analysis method is presented. Given bank conflict estimation results, a heuristic perimeter optimization algorithm is presented to find an efficient data layout solution. The first dimension of the searching space is bank mapping function options provided by the programming model; the second dimension is intra-variable padding solutions; and the third one is the storage sequence of variables and potential inter-variable padding solutions. The heuristic perimeter optimization method looks for optimal or sub-optimal solution with the following two questions: (1). Does it reduce the overall bank conflict of one kernel? (2). How much extra space is needed? Does it exceed the space limitations?

Figure 3.1 presents the relation between this work and related research areas. First of all, it is a project dealing with bank conflict issues of interleaved memory. It targets CUDA dynamic bank access mapping functions. In future, it could be used for other interleaved memory types with similar features. Secondly, it is based on memory access pattern knowledge of different data layouts. Thirdly, it tries to find an efficient solution based on intra-padding, inter-padding, and bank mapping function configuration. The goal is to make an automatic software solution that works at the programming level, and it realizes optimization through source code transformation.

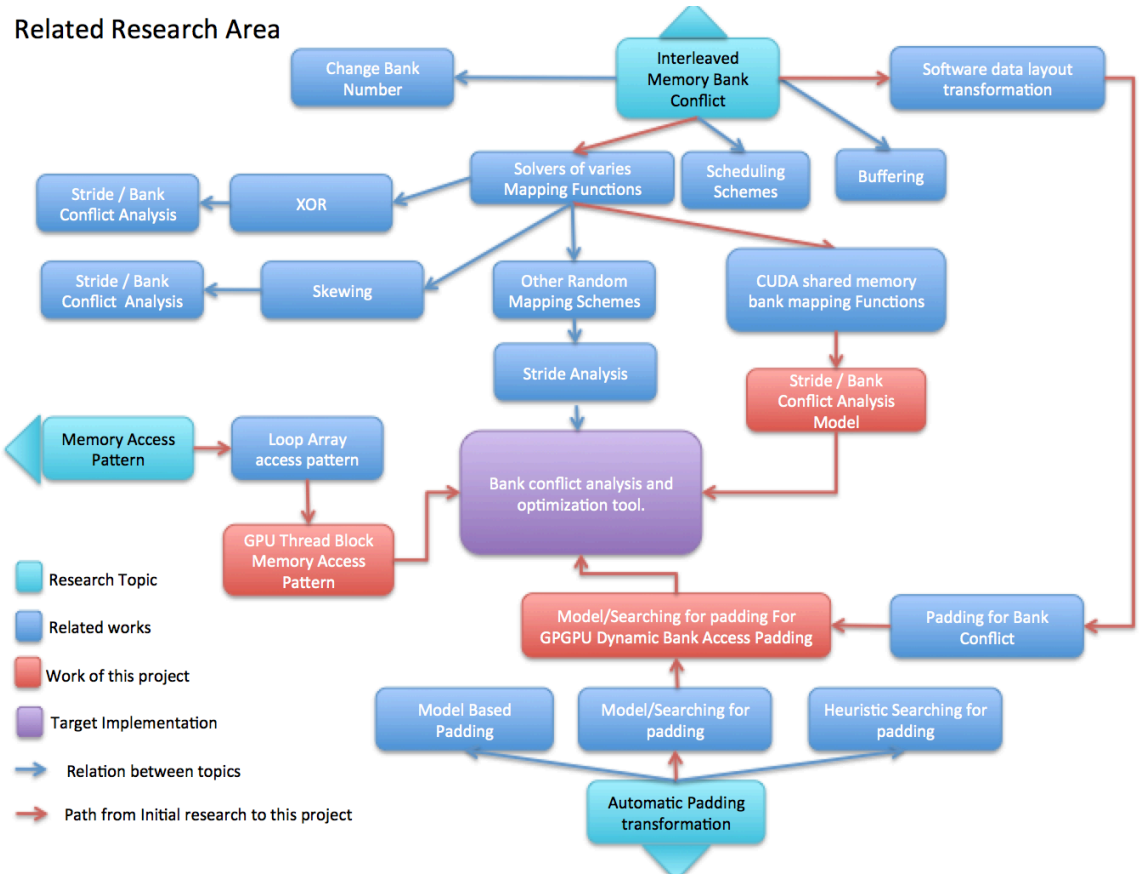


Figure 3.1 The relations of this project and related research areas

3.2 Project Assumptions

The following list is assumptions of this project:

(1). This is a static analysis.

(2). The bank number equals to the vector access length and its value is power-of-two.

(3). The target application uses dense memory access only. Indirect accesses (such as those used in sparse matrix and irregular mesh processing) are not included.

(4). This project targets C/C++ CUDA programming. By default array data is stored in row-major style. Any array used in this work is based on row-major data layout.

(5). Some typical 2D patterns are supported, but not all. For loops, the assumption is that the consecutive memory access requests have similar patterns.

(6). Based on the situation that programmers are responsible for solving bank conflict, here we don't assume the existence of any particular shared memory bank conflict related compiler optimizations.

(7). When an auto transformation tool looks for an optimal/sub-optimal solution, it will consider using less extra space, but it cannot guarantee that extra space chosen to be added will not cause decreasing device occupancy. To avoid such penalty, programmers should give a space limitation for each thread block.

(8). The current implementation targets at transforming C/C++ CUDA programs. Implementation for OpenCL can be added in similar way.

3.3 Project Framework

Figure 3.2 is the description of proposed approach. The work starts from a kernel source code and its memory access pattern description including: (1). Shared memory variable definitions; (2) Bank mapping functions defined by the programming model; (3) GPU thread block definitions; (4) array access stride; (5) other control information related to memory accesses. Basically, for different bank access bit-width settings, the tool analyzes the conflict replay number for each array, and then optimized the intra-padding size to obtain the optimal or sub-optimal solution. Among different bank access bit-width settings, the best option is the one that has the least conflict number and uses less memory. If the total conflict replay number is not zero, the tool looks for a proper inter-padding size for each array. Finally, according to the data layout solution, a source code transform is performed to modify the source code accordingly.

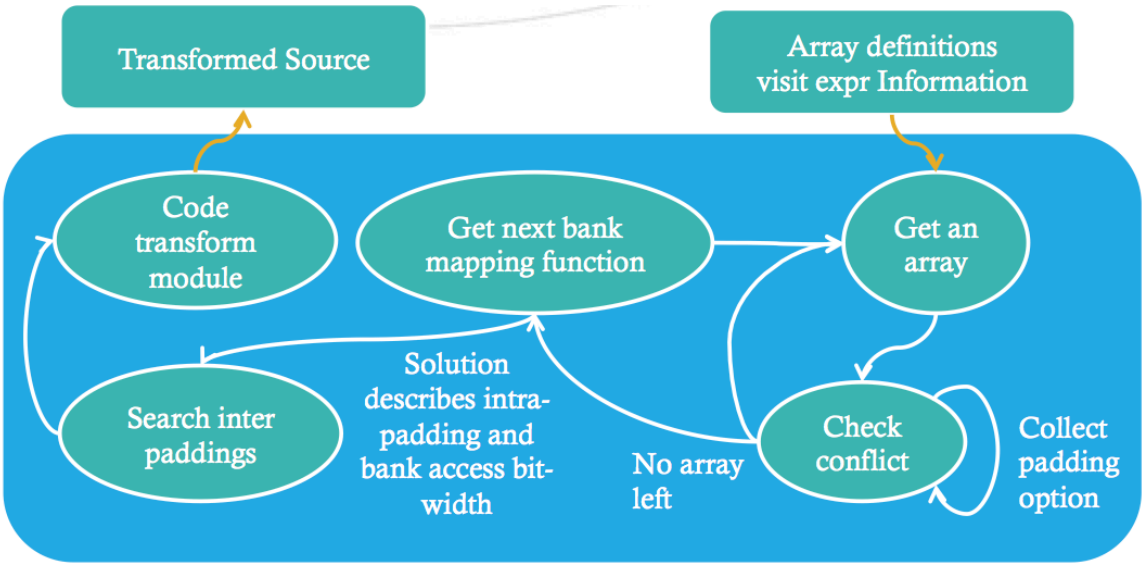


Figure 3.2 Framework of the project

3.4 Performance Improvement Expectation

The tool is designed to be able to improve GPU shared memory efficiency under following circumstances:

For a certain application kernel that uses 1D or 2D common shared memory access patterns, if its bank conflict problem can be eliminated or reduced by a combination optimization of (1) bank mapping function selection, (2) inter-padding, (3) intra-padding, then this tool can find an optimal or suboptimal solution automatically and transform data layout accordingly.

For an optimal/sub-optimal solution that can reduce the conflict degree from N to M, the memory access instruction replay number is reduced by $\frac{N-M}{N}\times 100\%$. It means that fewer cycles will be used for same purpose shared memory access. For applications that has bottleneck caused by shared memory bank conflicts, this transform may improve overall execution time.

3.5 Summary

This chapter introduces the background of GPU shared memory conflict issues, as well as the framework of this research. Due to the importance of shared memory access efficiency in GPU kernel performance tuning, and the difficulties of investigating the bank conflict for common access patterns, we made this effort to achieve automatic bank conflict optimization. The proposed work performs automatic source code transformation to optimize the data layout. This work includes a static bank conflict analysis and a heuristic parameter optimization method to find optimal or sub-optimal solutions.

CHAPTER 4 SINGLE VECTOR ACCESS BANK CONFLICT

This chapter analyzes the bank conflict of single vector (warp) access. Firstly, the bank mapping functions are introduced, and the impacts of inter-padding, intra-padding, and bank access bit-width on bank conflict are presented. Then the bank conflict analysis module is designed to estimate the degree of 1D stride and 2D stride access patterns. The work in this chapter is the core in the proposed framework. Any bank conflict optimization task will finally be divided into sub-tasks of single vector conflict analysis.

4.1 Bank Mapping Function Descriptions

Based on bank mapping functions supported for current commercial GPU shared memory, we define a generalized description. N is the bank access mapping width in bytes, M is the instruction access data type size in bytes, and W is the layer width size of one bank in bytes. We call N the N-Byte mode, and M the M-Byte element. To describe the target problem we add the following constrains:

1. Values of M , N , and W are power-of-two.
2. $4 \le M \le W$, $4 \le N \le W$
3. W is constant in one system
4. $vec\_length >> \frac{W}{M}$

There are following four different mapping functions as following:

- **Case One:** $M = N$ and $M < W$ **.** $M = N$ means that the instruction access bit-width matches the bank mapping access bit-width. $M < W$ means that this instruction access bit-width is smaller than the size of one layer of one bank. An example is $M = N = 4$,and $W = 8$. In this dissertation we call it case **row-major mapping**.
- **Case Two:** $M = N = W$. $M = N$ means that the instruction access bit-width matches the bank mapping access bit-width. $M = W$ means that this instruction access bit-width equals to the size of one layer of one bank. An example is $M = N = 8$, $W = 8$.

- **Case Three:** *M>N, M=W*. *M>N* means that the instruction access bit-width is larger than the bank mapping access bit-width configuration. *M=W* means that the instruction access bit-width equals to the size of one layer of one bank. An example is *N=4*, *M=W=8*.
- **Case Four:** *M<N*, *N=W*. *M<N* means that the instruction access bit-width is smaller than the bank mapping access bit-width. *N=W* means that the bank access bit-width equals to the size of one layer of one bank. An example is *M=4*, *N=W=8*. This case is called **column-major mapping**.

Table 4.1 describes features of these four bank-mapping functions. When we describe the stride analysis, following definitions are used:

- ***vector (or warp)***: execution unit of parallel memory access;
- ***vector length***: the element number visited by each vector (warp) access;
- ***offset***: the memory offset of the first element visited by a single vector access.
- ***stride family***: a stride can be described as stride=$\sigma \times 2^e$ (σ is an odd, and, $e \in Z$). For all strides that have same e, they belong to the same stride family. For example, $stride = \{2,4,6,10,14,...\}$ is the stride family that has $e = 2$.
- l***ayer***: One layer of a bank is a unit of space that multiple simultaneous accesses of it will cause no conflict. For example, for a layer size of 8B, the access of the upper 4B and the access of the lower 4B cause no conflict.
- ***row***: For case one, two, and three, a layer of a bank has $R = \frac{W}{M}$ rows. For example, for a layer size of 8B and a bank mapping access width of 4B, there are two rows in each layer. For case four, a layer of a bank has $R = \frac{W}{M}$ rows.

Figure 4.1 is array data mapping examples for these bank mapping functions (Bank number is 4, *W=8*). This literature focuses on the **row-major data mapping (case one)** and **column-major data mapping (case four)**. Case-three is similar to row-major mapping function; case-two is conventional low-order bank mapping function.

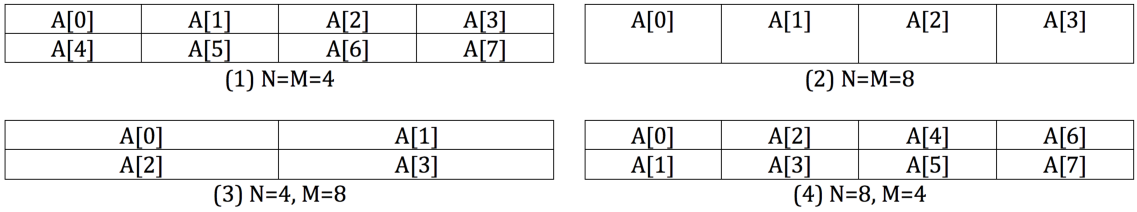


Figure 4.1 1D Array data mapping for different mapping functions.

Table 4.1 Summary of features of bank mapping functions

Case	***Example Architecture***	***Bank Num***	***N***	***M***	***W***
One	NVIDIA Kepler	32	4B	4B	8B
Two	NVIDIA Tesla	16	4B	4B	4B
	NVIDIA Fermi	16	4B	4B	4B
	NVIDIA Kepler	32	8B	8B	8B
Three	NVIDIA Kepler	32	4B	8B	8B
Four	NVIDIA Kepler	32	8B	4B	8B

4.2 Data Layout Transform and Bank Conflict

In this section some basic experiments is used to exam the impact of data layout on bank conflict degree. The platform information is as following:

- GPU device: Tesla K20c,
    - Shared memory:
        - Bank number is 32;
        - W=8B.
    - Warp size: 32 threads.
    - Compute capability: 3.5
- Programming model: CUDA 5.0
- Profiler: NVIDIA NVVP, release 5.0

4.2.1 Inter-padding

Inter-padding is a method used to change the memory access offset. This experiment shows the impact of access offset on bank conflict degree for the row-major data mapping function. Basically, the program reads shared memory by $stride = 2^e$. When the offset varies, the conflict number changes. In the example code in Figure 4.2, by inserting dummy variable of different sizes, we can change the offset. Figure 4.3 shows the impact of offset for different strides: the effect of changing offset is "+0" or "+1" to the existing conflict degree. It means that for larger strides, the impact of offset is smaller. Figure 4.4 compares the offset impact for two different bank access bit-width configurations. For this example, when offset is larger than 3, column major data mapping scheme is better than row-major data mapping scheme. Similarly, for the column-major data mapping function, the offset also could change the conflict degree. The detail will be discussed in chapter 6.

```
__shared__ DATATYPE dummy[N];
__shared__ DATATYPE A[SIZE];
Array_in_global[threadIdx.x] = A[threadIdx.x * stride];
__syncthreads();
```

Figure 4.2 Sample code

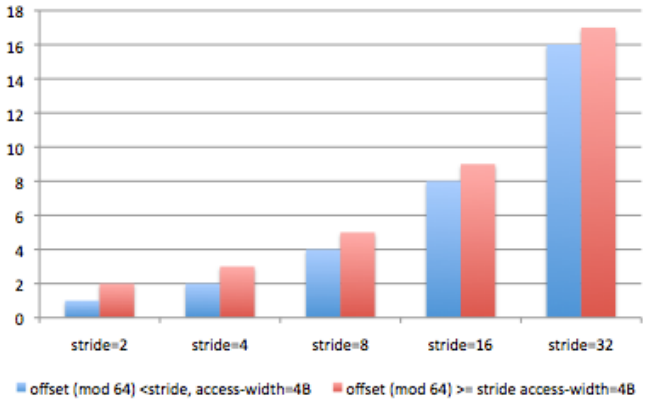


Figure 4.3 Conflict degree of different offsets for different stride values

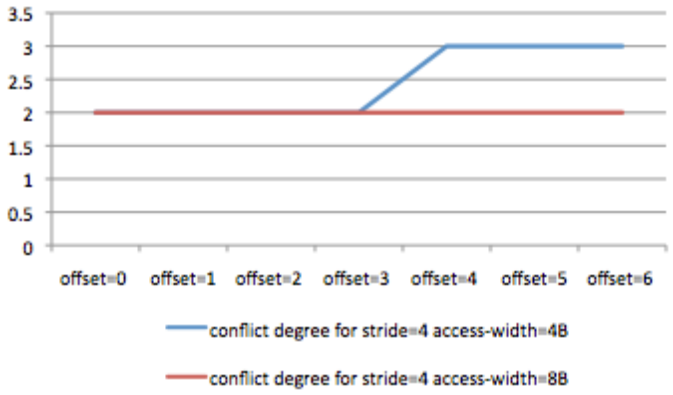


Figure 4.4 Impact of offset for different access bit-width setting

4.2.2 Intra-padding

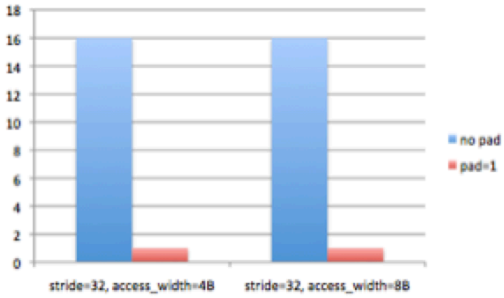
This experiment exams the impact of array intra-padding. When the padding varies, the conflict degree changes. For example code in **Figure 4.5** (a) and (b), we change the value of the macro PAD and check the conflict degree **Figure 4.5** (c) and (d) shows the impact of intra padding on bank conflict degree.

```
__shared__ DATATYPE A[32][32+PAD];
A_global[threadIdx.x] = A[threadIdx.x][0];
__syncthreads();
```

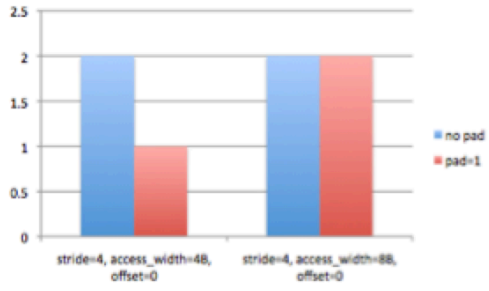
(a) Sample code A

```
__shared__ DATATYPE A[4][32+PAD];
A_globa[threadIdx.y][threadIdx.x] =
A[threadIdx.y][threadIdx.x*4];
__syncthreads();
```

(b) Sample code B



(c) Conflict degree of code A



(d) Conflict degree of code B

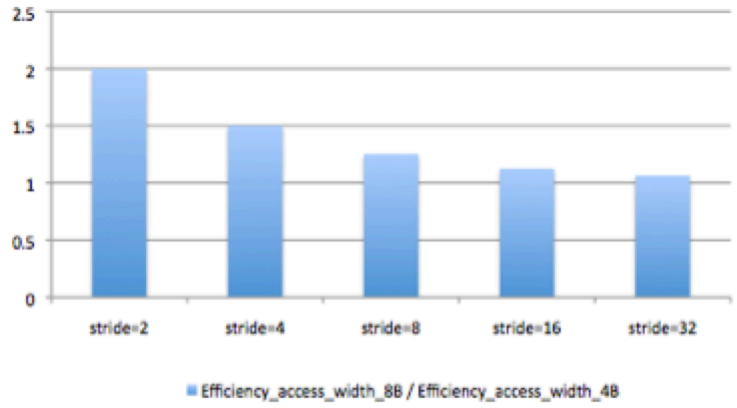
Figure 4.5 Impact of intra-padding on bank conflict numbers

4.2.3 Bank Access Bit-width

This experiment shows the impact of bank access bit-width on bank conflict degree. When access bit-width changes, the conflict degree changes. In the example code in Figure 4.6 (a), we change bit-width through the API provided by the CUDA programming toolkits. Figure 4.6 (b) shows the difference of the conflict degree when offset=32B. Figure 4.4 also shows the difference when $offset > 3$ and $stride \geq 4$: when $N = 8$ it has better efficiency.

```
Host code:
cudaDeviceSetSharedMemConfig(cudaSharedMemBankSizeEightByte)
or
cudaDeviceSetSharedMemConfig(cudaSharedMemBankSizeFourByte)
Kernel code:
__shared__ DATATYPE dummy[N];
__shared__ DATATYPE A[SIZE];
A_global[threadIdx.x] = A[threadIdx.x*stride];
__syncthreads();
```

(a) Sample code



(b) Rate of conflict degrees between 4B and 8B bank access modes

Figure 4.6 Impact of bank access width on bank conflict degree

4.3 Single Vector Bank Conflict Analysis

4.3.1 1D Access Pattern and Bank Conflict Analysis

Table 4.2 describes the features of bank conflict problem for **row-major bank mapping function** and **column major bank mapping function**. Basically, the bank conflict analysis module realizes following function:

bank_conflict_degree = func(bank_num, W, N, M, stride, offset)

Based on the input parameters, the bank mapping type is determined, and the task is assigned to the routine that perform the analysis. Detailed analysis description of each type can be found in appendix A-1.

Table 4.2 Summary of features of bank conflict problem

Type	***Conflict analysis features***
Row-major data mapping	Analysis process is based on *gcd(stride, bank_number)*. However, since one layer of all banks has multiple rows, only accesses to different layers cause bank conflict. The analysis routine take offset and result of *gcd(stride, bank_number)* to calculate the bank conflict number.
Column-major data mapping	Since each layer of all banks has multiple rows, and data are mapped in column major direction, both odd stride and even stride could cause bank conflict. A routine is designed for odd stride bank conflict analysis. Even stride analysis can be transformed to either odd stride problem or conventional bank mapping problem.

4.3.2 2D Access Pattern and Bank Conflict Analysis

For 2D parallel memory access, we can describe a parallel execution unit by two types: a 1D vector, or a 2D rectangular grid (Figure 4.7 (a)). When such a vector accesses data in an array, we describe access pattern as:

$$< stride\_x, repeat\_x, stride\_y, repeat\_y >$$

(a) 1D vector and 2D grid (b) Basic memory access patterns

Figure 4.7 shape of parallel memory access unit and base memory access patterns

By defining an affine transform matrix in Homogeneous Coordinates, we can get the transformed 2D access patterns based on the basic access pattern in Figure 4.7 (b). We describe this transform matrix T as:

$$v' = Tv = \begin{pmatrix} a_{11} & a_{12} & c_1 \\ a_{21} & a_{22} & c_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} threadIdx.x \\ threadIdx.y \\ 1 \end{pmatrix}$$

When the width of the 2D memory access execution unit is less than *vec_ legnth*, the 2D stride pattern can be obtained from the functions in Table 4.3. In these functions, the *blockDim.x* denotes the width of execution unit; width denotes the width of 2D array. Figure 4.8 shows some examples of the transformed access patterns.

Table 4.3 2D stride pattern calculation

$stride\_x$	$repeat\_x$	$stride\_y$	$repeat\_y$
$a_{21} \times array\_width + a_{11}$	$blockDim.x$	$a_{22} \times array\_width + a_{12}$	$\frac{vec\_length}{blockDim.x}$

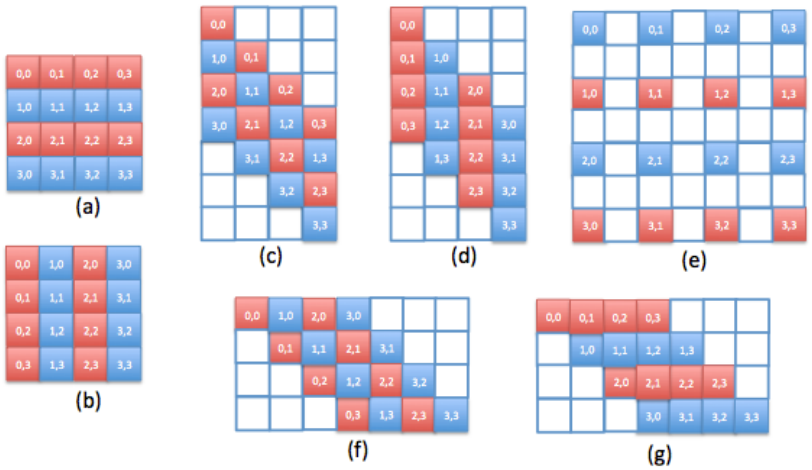


Figure 4.8 Examples of the base 2D access pattern transformation

When we shift the offset of current parallel access, an observation is that the bank mapping of all visited elements repeat after a certain number of steps. For example, when row-major mapping function is in use, for

$$offset_{new} = offset_{original} + \frac{W \times bank\_num}{M}$$

it has

$$conflict_{new} = conflict_{original}.$$

This means that we can firstly calculate bank conflict for a small and fixed number of offset values, and then for other offsets, get the conflict degree by mapping it to a known offset value. We call this small group as $base\_set$. In many GPU kernels, one parallel memory operation is executed for many times with different offsets. By computing the conflict for a small $base\_set$, this method can obtain the overall conflict in a limited time period which is independent of vector access number. For detail information about 2D bank conflict analysis, please refer appendix A-3.

4.4 Algorithm Analysis

The single vector analysis is the basic component of the proposed framework. It works at the center of this work in that other modules are built upon it. This section introduces the algorithms used to realize single vector analysis; their time complexities are discussed as well.

4.4.1 1D Analysis Algorithm

Row-major Bank Mapping Function

For row-major bank mapping function, when stride is odd, for any W, N, and M, there is no bank conflict. For even strides, we divide them into two categories: (1) stride is power-of-two, (2) Other even stride.

When stride is power-of-two, the algorithm (algorithm 4.1) uses the result of $\gcd(stride, bank\_num)$ and the vector access offset to calculate the conflict degree. Since

the time complexity of gcd($stride, bank\_num$) is $O(stride)$, this function's time complexity is also $O(stride)$. Basically, (1) when stride is larger than the element number that can be stored by one layer of all banks, all visited elements lie in different layers of the same bank, and any pair of them has conflict. (2) Otherwise, if the stride can be divided by $bank\_num$, all visit sites lines in same bank of one or multiple layers. Based on the value of offset, the conflict degree can be deduced from the result of gcd($stride, bank\_num$). (3). Otherwise, it means that the $bank\_num$ can be divided by stride, the result of gcd($stride, bank\_num$) and the offset is used to calculate the conflict degree. Figure 4.9 describes this in details.

Given $S = 2^s$, $bank\_num = 2^b$, and $row\_num\_per\_layer = \frac{W}{N} = 2^e$, when

$S \geq 2^b \times 2^r = 2^{b+r}$. It has $\frac{S}{2^{b+r}} = 2^{s-b-r} \geq 1$. So each layer hosts at most one visited site. So the conflict degree is a constant which is as same as *vec_length*.

$S < 2^b \times 2^r = 2^{b+r}$. There are two possibilities:

$2^s < 2^b$. Which means $s < b$. So the visited site number in each row is 2^{b-s}.

$2^s \geq 2^b$. Which means $s \geq b$. So all visited sites lie in the same bank, and every 2^{s-b} rows have one visited site. In addition, 2^{s-b} can be divided by 2^r.

Figure 4.9 for power-of-two stride

When stride is other even numbers, the time complexity is also $O(stride)$. For an even stride in stride family $\sigma \times 2^e$, the visited sites can be divided into 2^e groups, each groups occupies σ rows. For the i[th] row of all groups, they visit same banks. So there must be conflict if not all of them lie in same layer. Inside each group, there is no conflict possibility. Based on such observation, the task becomes to check the conflict among i[th] rows of all groups (algorithm 4.2).

Algorithm 4.1 *func_row_major_power_of_two_stride*
Input: bank_num, W, N, M, stride, offset
Output: res -- bank conflict degree.
//----------
gcd_res = gcd(stride, bank_num);
R = W/N; res=1;

```
layer_size = bank_num * W;
vec_length = bank_num;
offset = offset % layer_size;  //calculate offset in one layer.
If (stride >= bank_num * R)
   res = vec_length;
else
   If(stride % bank_num == 0)
      res = (gcd_res*(stride/bank_num) + (R-1)) / R;
   else
      res = (gcd_res + (R-1)) / R;
   end if
   if(offset_impact==true)
      res += 1;
   end if
end if
```

Algorithm 4.2 *func_row_major_other_even_stride*

```
Input: bank_num, W, N, M, stride, offset
Output: res -- bank conflict degree.
//-------------------------------------------------------
gcd_res = gcd(stride, bank_num);
R = W/N; res=1;
layer_size = bank_num * W;
vec_length = bank_num;
offset = offset % layer_size;  //calculate offset in one layer.
tau = tau(stride);   //calculate stride family parameter tau
e=e(stride);  //calculate stride family parameter e.
if(2^e >= bank_num)
   if(stride > W*bank_num/M)
      res = bank_num;
   else
      res = ceil(offset/M + (vec_length-1) * stride + 1, bank_num*W/M) / (bank_num * W / M);
   end if
end if
for row I in {0,..,tau-1}
   for group in {0,..., 2^e-1}
      if current_layer(group) != previous_layer(group)
         conflict = true; res++;
      end if
   end for
end for
```

Column-major Bank Mapping Function

For this bank mapping function, no matter stride is odd or even, there could be potential conflict, and we need to calculate conflict degree (algorithm 4.3). The time

complexity of this method is $O(R \times vec\_length)$. Compares to enumerating every visit site of one vector access, this time complexity is worse because the later one has $O(vec\_length)$. However, it doesn't mean that this method is not helpful. Actually, it gives useful clues to take short cut for some cases. One important conclusion based on this method is about current commercial GPU shared memory. In appendix A, it is approved that the conflict is always 2-way when $R = 2$, with $R = \frac{W}{M}$. For other $R \neq 1$, the condition about $layer\_scope$ could terminate the loop and help to avoid unnecessary calculations for non-valid pairs.

Algorithm 4.3 *func_column_major_odd_stride*
Input: bank_num, W, N, M, stride, offset
Output: res -- bank conflict degree.

```
//---------------------------------
    For vector visit start from r in 0 to R-1
    //initialize bank_layer_info
        For i=0 to bank_num-1
                std::pair<unsigned, std::set<unsigned int> > curr_pair;
                curr_pair.first = i;
                curr_pair.second.clear();
                bank_layer_info.push_back(curr_pair);
        Endfor

        //.....................................................
        //step 1:
        row_idx = (offset % W) / M;

        //.....................................................
        //step 2:
        For i = 0 to R-1
                calculate row_offset[i]
        Endfor

        For i = 0 to R-1
                calculate imm_col_offset[i]
        Endfor

        //.....................................................
        //step 3:
        For i = 0 to R-1
                calculate row_scope_num[i];
        Endfor

        //.....................................................
```

```
//step 4 (CASE-A): check every row-pair of CASE-A
For i=0 to R-2
        For j=i+1 to R-1
                //For each pair of rows, check existence of conflicts.
                bool conflict = false;
                For diff_y_x = 0 to vec_length/R-1
                        calculate dist: the offset difference between visit x and visit y
                        If ((dist > 0) && ((dist % bank_num) == 0))
                                layer_scope = dist / bank_num;
                                If (layer_scope <= (row_scope_num[i] - 1))
                                        diff_res = diff_x_y;
                                        conflict = true;
                                        break;
                                Endif
                        Endif
                Endfor
                If (conflict)
                        For each pair of x and y that has difference of diff_res
                                calculate bank_id;
                                calculate layer_y;
                                calculate layer_x;
                                save these conflict information to bamk_info[bank_id];
                        Endfor

                Endif
        Endfor
Endfor

//.....................................................
//step 4 (CASE-B): check every row-pair of CASE-B

For i=0 to R-2
        For j=i+1 to R-1
                //For each pair of rows, check existence of conflicts.
                bool conflict = false;
                For diff_y_x = 0 to vec_length/R
                {
                        calculate dist: the offset difference between visit x and visit y
                        If ((dist > 0) && ((dist % bank_num) == 0))
                                int layer_scope = dist / bank_num;
                                If (layer_scope <= (row_scope_num[j] - 1))
                                        diff_res = diff_y_x;
                                        conflict = true;
                                        break;
                                Endif
                        Endif
                Endfor

                If(conflict)
                        For each pair of x and y that has difference of diff_res,
                                calculate bank_id;
```

```
                                    calculate layer_y;
                                    calculate layer_x;
                                    save these conflict information to bamk_info[bank_id];
                                Endfor
                            Endif
                        Endfor
                    Endfor

                    For I in 0 to bank_num-1
                        If bank[i].layer_num > max;
                        Max = bank_info[i].layer_num;
                        Endif
                    Endfor
Endfor
```

For even stride, the problem can be transformed either to odd stride problem or directly to conventional bank access problem. Then the time complexity is either as same as the one for odd stride problem, or the one for conventional bank conflict problem. The table A-1-5 describes the rules of problem transformation.

4.4.2 2D Access Analysis Algorithm

When array is visited through a 2D stride, there are two cases: (1). Each warp accesses array in 2D pattern; (2). Each warp accesses array in 1D pattern. A simple example of the first case is 8x4 access by a warp of 32 threads. It means that for the first stride the repeat times is 8 and for the second stride it is 4. For the second case, even the access of a whole thread block is 2D, but since the repeat time of the first stride can be divided by vector length, then the problem can be transferred to a 1D cases for each warp. The algorithm 4.4 is the bank conflict calculation for row-major bank mapping. For column-major bank mapping function, the algorithm is similar except that different functions are used to calculate the bank indices and layer indices. This is a simple and straightforward solution. At the beginning, an array of bank information are defined and initialized, it is used to store the bank access information. Then, for each visited element, calculate its bank index and its layer offset, and record the distinct layer indices of same bank. Finally, it goes through all banks and finds the bank that has maximum distinct layer number. This number is the bank conflict degree of the current single warp 2D access.

Algorithm 4.4 *2D_row_major_stride*
Input: bank_num, W, N, M, stride, offset
Output: res -- bank conflict degree.
//--
For I in 0 to bank_num-1
    Initialize bank_info [i]
Endfor

For I in 0 to rep_y-1
    For j in 0 to rep_x-1
        Calculate bank_idx and layer_idx based on row-major bank mapping function
    Done.
Done.

For I in 0 to bank_num-1
    If bank[i].layer_num > max;
        Max = bank_info[i].layer_num;
    Endif
Endfor

For this algorithm the time complexity is composed of three parts are: $O(bank\_num)$. $O(vec\_length)\,O(bank\_num)$. Since we assume vec_length equals to $bank\_num$, the overall time complexity of this algorithm is $O(bank\_num)$.

4.5 Summary

In this chapter we describe the bank conflict problem of single vector access, and introduced the method for bank conflict analysis. Section 4.1 introduces the information of bank mapping functions. In section 4.2, experimental evidences are used to show the impact of data layout on bank access efficiency. By changing bank access bit-width, adding inter padding, or adding intra padding, the data layout transformations reduce or eliminate the bank conflict. In section 4.3 and 4.4, the conflict analysis algorithms are presented.

Based on this single vector analysis module, in chapter 5, the analysis method of single expression memory access is constructed; in chapter 6, a heuristic parameter optimization method is built to look for the optimal or sub-optimal data layer solution.

CHAPTER 5 SINGLE EXPRESSION ACCESS BANK CONFLICT

In GPU programming model, one memory access expression drives concurrent threads to access a sequence of data in parallel. Normally these threads belong to multiple vectors/warps. In this chapter, we take single expression as a unit and analysis its bank conflict. Given an array access expression, the tool analyzes the overall bank conflict number of multiple warps that execute the memory operation. This work is based on the single vector bank conflict analysis presented in chapter 4. Since programmers determine the warp number, the ideal solution should be able to estimate conflict number and its time complexity shouldn't depend on the warp number. In this chapter, section 5.1 analyzes the bank conflict of basic array access expression. Section 5.2 analysis the "for" wrapped memory access, which is normally used to increase workload of each thread. Section 5.3 and 5.4 analyze "if" and "for-if" wrapped memory access expression, they are normally used to filter the threads and control the memory access ranges/patterns. Figure 5.1 presents the relations of these conflict analysis modules.

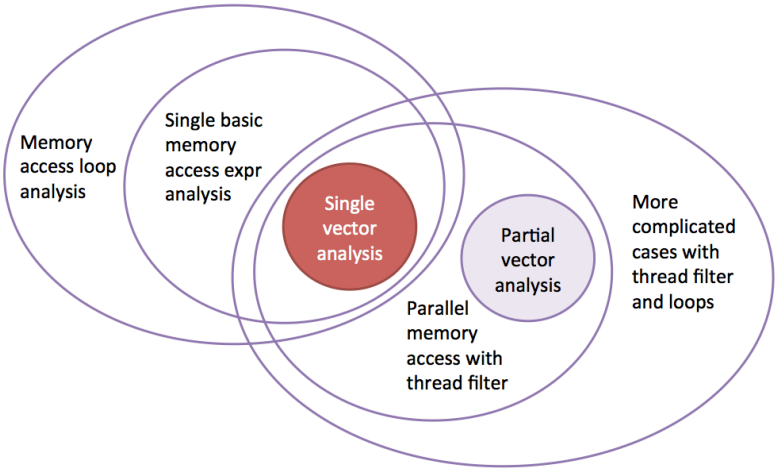


Figure 5.1 Conflict Analysis Modules

5.1 Basic Bank Conflict Analysis

5.1.1 1D Access Analysis and Algorithm

This section explains how to calculate the bank conflict number of single memory access expression when multiple warps are involved. This work is based on the algorithm for 1D single vector analysis presented in section 4.3. After the bank conflict of the first warp is obtained, the conflict result of other warps could be different from it in that their access offsets could be different. The memory access offset of the ith warp is:

$offset[i] = offset[0] + i \times stride \times vec\_length$, $i = [1,2,...,warp\_num\_per\_block)$

This formula shows that the offset of warp i is linear to the warp index i. In one layer of all banks, the in-layer offset is:

$$offset\_in\_layer[i] = \mathrm{mod}(offset[i], bank\_num \times \frac{W}{M})$$

This formula shows that the relative offset is periodic. For example, for all warp i that have $\mathrm{mod}(offset[i], bank\_num \times \frac{W}{M}) = 0$, they have same relative offset which is the beginning of a layer. Based on this observation, we design the conflict analysis for row-major bank mapping function and column-major bank mapping function as following.

Row-major bank mapping function

For row-major bank mapping function, when the stride is odd, there is no bank conflict. For even strides, they can be divided into two categories and each uses a different analysis method. The first group is stride values that are power-of-two; the second group includes all other even strides.

Stride is power-of-two

1. When $stride \geq R$, since each of them is power-of-two, then $\mathrm{mod}(stride, R) = 0$. And since vec_length equals to $bank\_num$, we have:

$$\mathrm{mod}(i \times stride \times vec\_length, R \times bank\_num) = 0$$

Based on this equation, for warp i, the in-layer offset is:

$$offset\_in\_layer[i] = \mathrm{mod}(offset[0] + i \times stride \times vec\_length, R \times bank\_num) = offset\_in\_layer[0]$$

with $i = [1,2,...,warp\_num\_per\_block)$. Since all these warps have same value of in-layer offset, they have same bank conflict estimation result. The total conflict number can be obtained through multiplying single warp conflict number by the warp number.

2. When $stride < R$, since both R and stride are power-of-two, R can be divided by stride. Let's define $P = \frac{R}{stride}$, it has:

$$offset[i+P] = offset[i] + P \times stride \times vec\_length = offset[i] + R \times vec\_length$$

Since *vec_length* equals to *bank_num*, it has:

$$offset\_in\_layer[i+P] = \mathrm{mod}(offset[i+P], R \times bank\_num) = \mathrm{mod}(offset[i], R \times bank\_num) = offset\_in\_layer[i]$$

This equation shows that after every P warps, the in-layer offset repeats. Based on this observation, the algorithm goes through following three steps to obtain the final conflict number:

- Calculate P.
- Obtain the conflict number for the P distinct warps. To get the offset of each cases, variable *sub_offset* is defined as:

$$sub\_offset = \mathrm{mod}(offset, R \times bank\_num)$$

Then relative offset in one layer can be obtained by:

$$offset\_in\_layer[k] = \mathrm{mod}(sub\_offset + k \times stride \times vec\_length, R \times bank\_num)$$

with $k = [0,...,P)$

- Calculate the warp number that belongs to each case k, $k = [0,...P)$.
- Get the overall conflict number by:

$$total\_conflict = \sum_{k=0}^{P-1} conflict_k \times warp\_num_k$$

Other even strides

When stride is other even number, it can be described as a member of stride family $\sigma \times 2^e$ with $\sigma = [3,5,7...)$ and $e = [1,2,3,...)$. Then the offset of warp i can be calculated as:

$$offset[i] = offset[0] + i \times stride \times vec\_length = offset[0] + i \times \sigma \times 2^e \times vec\_length$$

1. When 2^e is larger than or equal to R:

$$\mathrm{mod}(offset[i], R \times bank\_num) = \mathrm{mod}(offset[0], R \times bank\_num)$$

It means that for all warps, they have same in-layer offset, and have same conflict number.

2. When 2^e is smaller than R, we need to find the value of P for which it has:

$$\mathrm{mod}(offset[i+P], R \times bank\_num) \equiv \mathrm{mod}(offset[i], R \times bank\_num)$$

First of all, since *vec_length* equals to *bank_num*, it has:

$$P \times \sigma \times 2^e \times vec\_length = N \times R \times bank\_num$$

This equation can be simplified as:

$$P \times \sigma = N \times \frac{R}{2^e}$$

with P and N are non-zero positive integers.

Then, based on this equation, P can be calculated as:

$$P = \frac{LCM(\sigma, \frac{R}{2^e})}{\sigma}$$

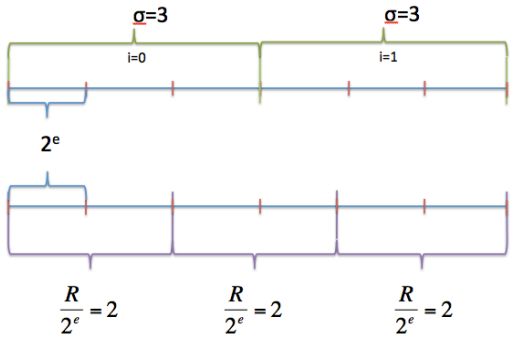


Figure 5.2 an example

Figure 5.2 is an example. After replacing i with P, the $offset[P]$ becomes:

$$offset[P] = offset[0] + LCM(\sigma, \frac{R}{2^e}) \times 2^e \times vec\_length$$

$$= offset[0] + N \times \frac{R}{2^e} \times 2^e \times vec\_length$$

with $N \times \frac{R}{2^e} = LCM(\sigma, \frac{R}{2^e})$. This indicates that:

$$\text{mod}(offset[0], R \times bank\_num) \equiv \text{mod}(offset[P], R \times bank\_num)$$

Based on this observation, the algorithm uses three steps to obtain the final conflict number:

- Obtain the conflict estimation of the P distinct cases. To get the offset of each cases, it defines sub_offset as:

$$sub\_offset = \text{mod}(offset, stride \times bank\_num)$$

Then each offset can be obtained through:

$$offset[k] = sub\_offset + k \times stride \times bank\_num \text{, with } k = [0, ..., P)$$

Then the in-layer offset can be obtained by:

$$offset\_in\_layer[k] =$$
$$\text{mod}(sub\_offset + k \times stride \times bank\_num, R \times bank\_num)$$

with $k = [0, ..., P)$

- Calculate the warp number that belongs to each case k, $k = [0, ..., P)$.
- Get the overall conflict number by:

$$total\_conflict = \sum_{k=0}^{P-1} conflict_k \times warp\_num_k$$

Column-major Bank Mapping Function

For column-major bank mapping function, when an offset makes a vertical shifting among different rows in a layer, it impacts the conflict number. Horizontal shift only moves conflicts from one bank to the other bank, so it doesn't change the conflict number. We define vertical row offset as:

$$offset\_row[i] = \mathrm{mod}((offset\_row[0] + i \times stride \times vec\_length), R)$$

Since *vec_length* and R are power-of-two and *vec_length* is larger than R, we have

$$\mathrm{mod}(i \times stride \times vec\_length, R) \equiv 0$$

This indicates that for any value of i and any value of stride, it has

$$\mathrm{mod}(offset\_row[i], R) \equiv \mathrm{mod}(offset\_row[0], R)$$

Then, the overall conflict number can be obtained by:

$$total\_conflict = conflict_o \times warp\_num$$

5.1.2 2D Access Analysis and Algorithm

Similar to 1D solution, for multiple warps that access memory in 2D patterns, it goes through these steps to obtain the overall conflict number. In section 4.4, when we calculate conflict number for 2D single vector access, the conflict degree is calculated for $offset \in [0, \frac{W}{M} \times bank\_num)$, and the results are stored in a table. This table can be reused here to look up the conflict number for a certain offset. Here we divide 2D access patterns into two categories, and then discuss the solution for each of them.

1. for a 2D access pattern <*stride_x, repeat_x, stride_y, repeat_y*>, when *repeat_x* is less or equals to *vec_length*, we use four steps to obtain overall conflict number:

Calculate P as:

$$P = \frac{lcm(scope\_of\_single\_warp\_access, R \times bank\_num)}{scope\_of\_single\_warp\_access}$$

- Then for each distinct $k \in [0,...,P)$, calculate the $offset\_in\_layer[k]$ and look up the conflict table to get the conflict.
- Calculate the warp numbers that belongs to case $k = [0,...,P)$
- Sum up the overall conflict number:

$$total\_conflict = \sum_{k=0}^{P-1} conflict_k \times warp\_num_k$$

2. When *repeat_x* can be divided by *vec_length*:

- For each warp i in the first dimension stride access, $i \in [0,..., \frac{repeat\_x}{vec\_length})$, use following four steps to calculate the $total\_conflict[i]$:
  - Calculate P which is the number of distinct offset cases;
  - Calculate conflict for each $k = [0,...,P)$,
  - Calculate the warp number that belong to case $k = [0,...,P)$
  - Summary the conflict numbers and save as $total\_conflict[i]$:

$$total\_conflict[i] = \sum_{k=0}^{P-1} conflict_k \times warp\_num_k$$

- Finally, use a reduction to get the summary of elements in array *total_conflict*, which is the total conflict number.

5.2 "for" Loop Wrapped Single Access Expression

5.2.1 Motivation

"for" loops are frequently used to increase the workload of each thread. Figure 5.3 shows two loop examples. In each of them, iterate variable *i* is used to change the memory access offset for current iteration. In case (a), the offset increment of each iteration is the first dimension length of array A; in case (b), it is the value of *blockDim.x*.

```
For i in 0..M
    Operate on A[i][threadIdx.x]
End for loop
```
(a) 2D example

```
For i in 0..M
    Operate on A[i*blockDim.x+threadIdx.x]
End for loop
```
(b) 1D example

Figure 5.3 Examples of for loop wrapped memory accesses

Given the solution for single expression conflict estimation, a basic method is to estimate conflict for each iteration, and then get overall conflict number through a reduction. However, the workload of this solution depends on the loop iteration number. For kernels that have large number of iterations, it is not a practical static processing. The ideal solution should be able to complete the analysis within a certain period which doesn't depend on the iteration number.

5.2.2 Solution

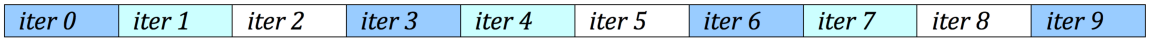
Given the solution of multiple warp analysis, similar solution could be used to deal with memory access with "for" loop wrappers. We use function *lcm()* to find distinct iterations that have different offset from one another, and then calculate the overall conflict. This helps to optimize the workload from *O(iteration_number)* to *O(elem_number_per_layer)*. The first one depends on application kernel design, and the later one depends on memory bank architecture. The algorithm uses four steps to obtain the final result:

- Obtain the number of distinct iterations as P:
- $$P = \frac{LCM(iter\_offset\_increment, bank\_num \times R)}{iter\_offset\_increment}$$
- iter_offset_increment denotes the offset increment for each iteration.
- For P distinct cases, get the offset of each cases.
- Calculate the iteration number that belongs to each case.
- Get the overall conflict number by:

$$total\_conflict = \sum_{k=0}^{P-1} conflict_k \times iter\_num_k$$

Figure 5.4 is an example. There are totally 10 iterations, and P is 3. Then, for $k = 0$, there are 4 iterations; for $k = 1$, there are 3 iterations; and for $k = 2$ there are 3

iterations. For row-major bank mapping function and col-major bank mapping function, the methods used to calculate $offset[k]$ $(k=[0,...,P))$ are different.



$$total\_conflict = conflict[0]\times 4 + conflict[1]\times 3 + conflict[2]\times 3$$

Figure 5.4 An example of loop which has 10 iterations, and P=3

For 2-level nested loops, this solution can be extended in similar way. Firstly, the conflict of the inner loop is calculated by the introduced method. Then, the P for the outer loop is calculated, and conflict of each case $k=[0,...,P)$ is calculated. Finally, the overall conflict number of the 2-level nested loop is calculated. For other multi-level loops, as long as the iterate variable has similar impact on the memory access offset, they can be analyzed in the same way. The execution time depends on Loop level number and one layer size of all banks: the single expression analysis is executed for $P_0\times P_1\times ...\times P_{i-1}$ times, with $P_i \le bank\_num\times\frac{W}{M}$ $(i\in[0,...,l))$, and l is the loop level number.

5.3 "If" Condition Wrapped Single Access Expression

5.3.1 Motivation

In GPU kernels, "if" statement is sometimes used to filter threads and only some threads are allowed to execute.Figure 5.5 is an example: the code has a branch and two groups of threads do different jobs: the first group visits array A, and the second group visits array B.

```
If threadIdx.x < X
      Operate on A[threadIdx.x]
Else
      Operate on B[threadIdx.x-X]
End if
```

Figure 5.5 An example of "if" statement used to filter the threads by thread ID

These thread filter conditions are designed according to the purpose of the program. It means that the boundary of work threads and idle threads could be anywhere and it is not guaranteed to be aligned to *vec_length*. When this happens, the proposed conflict estimation method is not applicable any more. One basic problem is related to gcd*(stride, bank_num)*. In chapter 4 the result of this function is used to estimating conflict degree. When vector access length is random instead of equal to bank number, gcd*(stride, bank_num)* cannot be used for this purpose any more. To design a proper method, we need to study row-major bank mapping function and column-major bank mapping function separately. For each of them, a routine is designed for bank conflict estimation.

5.3.2 Solution

Divide Threads into Groups

First of all, for a "if" statement that allows m threads to execute, these threads could be divided into 3 potential groups:

- 1st group (**G1**): thread number is less than *vec_length* and the threads belong to the later part of a vector. (Figure 5.6 (a))
- 2nd group (**G2**): thread number can be divided by *vec_length*, and thread index of the first thread is aligned to *vec_length* (Figure 5.6 (b))
- 3rd group (**G3**): thread number is less than *vec_length* and the threads belong to the first part of a vector. (Figure 5.6 (c))

For example, when a thread group's thread number is less than *vec_length*, and the first thread's index is *vec_length* aligned, it has one group which is G3. For a thread group that has thread number more than *vec_length* and the first thread's index is not aligned to

vec_length, it could have two groups {G2, G3}, or three groups {G1, G2, G3}. For a thread group that has more thread number than vector length and starts from an aligned thread ID, it has two groups {G2, G3}.

(a). G1 is the blocks in orange (b) G2 is the blocks in blue (c) G3 is the blocks in green

Figure 5.6 Dividing threads into groups

Bank conflict for groups G2 can be calculated by existing solutions. For the group G1 and G3, solutions are designed as following.

Estimating Conflict for G1 and G3.

For **row-major bank mapping function**, when stride is odd, there is no conflict. When stride is even, the offset in current layer is calculated, and then the stride family parameters σ and 2^e are calculated.

1. **When thread number is smaller than** $\frac{bank\_num}{2^e}$, the scope of the visit can be calculated as:

$$visit\_scope = stride \times thread\_num = \sigma \times 2^e \times thread\_num$$
$$< \sigma \times 2^e \times bank\_num / 2^e = \sigma \times bank\_num$$

As we know, for a stride $S = \sigma \times 2^e$, the visited layers can be divided into chunks each of which has σ rows. Inside each chunk there is no conflict. Since this *visit_scope* has only σ rows, it has no conflict.

2. **When thread number is larger or equals to** $\frac{bank\_num}{2^e}$, there could be conflicts.

1) When bank number can divide the stride, it means that all visited sites lie in the same bank:

a) If the stride can be divided by $bank\_num \times R$, it means that all visited elements lie in different layers, and the conflict number equals to the thread number.
b) Otherwise, there is at least one visit in each layer. For this case, we can calculate the layer number that contains visited sites.

2) When bank number cannot divide the stride, we calculate the layer number that has been visited. Combined with the value ofσ, we can calculate the conflict degree.

For **column-major bank mapping function**, as introduced in appendix A, the existence of conflict is calculated between every two rows that have different indices, when there is conflict, an index difference of x and y exists (x and y are the indices of visit sites in each row). The value of this difference needs to be smaller than the visiting scope of a vector access. Here, when the thread group has less thread number than *vec_length*, after the index difference is obtained, it needs to be inside the scope of current visiting area. In conclusion, after the condition for the visiting scope check is changed accordingly, the original algorithm can be reused. For even stride, the problem can be transformed to either the odd stride problem, or the conventional low-order bank mapping problem,

5.4 "for-if" Statement Wrapped Single Access Expression

5.4.1 Motivation

In some kernels, "for-if" combination is used to allow different amount of threads work in different iterations. Basically, the iterate variable is used as the condition in the "if" statement which filter the threads by thread indices. An example is shown in Figure 5.7. Such code is normally used to process array data by an increasing/decreasing number of threads.

```
for i in 0..M
   If threadIdx.x < i
      Operate on A[threadIdx.x]
   endif
End for
```

Figure 5.7 An example of "for-if "wrapped single expression access

A basic method is to estimate the conflict number of each iteration one after another. With existing solutions, for each iteration, estimating conflict number does not depend on the thread number. However, the time complexity for processing all iterations depends on iteration number. Since the iteration number is determined by kernel design, this solution is not practical. The ideal solution should have relatively bounded execution time no matter how many iterations the loop has.

5.4.2 Solution

By studying the threads activities of all iterations, we can find clues to reduce the workload of conflict estimation. Figure 5.8 is a figure of thread activities across all iterations. The *Y* direction is the iteration index. The *X* direction is the thread index. In this example, since the iterate variable ($i = [0,...,15]$) is used as the boundary of active thread index, the maximum index of active threads is 15.

For this example, assume the bank number and *vec_length* is 4, there are:

- 28 *vec_length* aligned accesses (In the figure, the start point of each aligned vector access is marked by "*").
- 4 of single-thread accesses;
- 4 of double-thread accesses;
- 4 of triple-thread accesses.

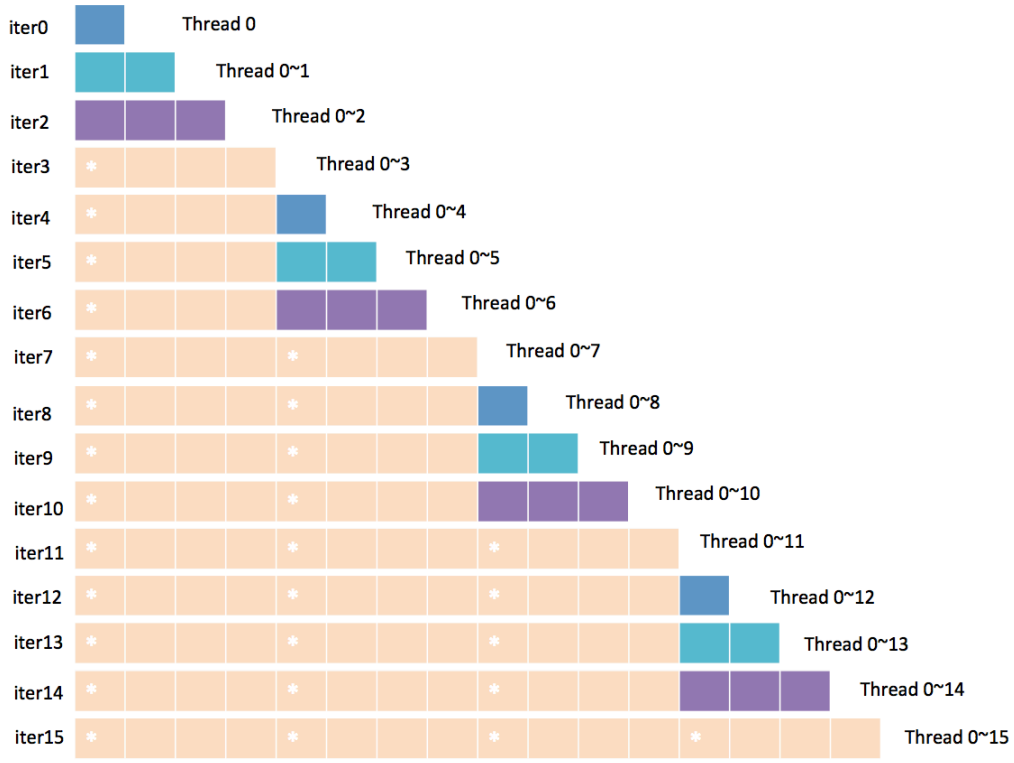


Figure 5.8 the example of "for-if " wrapped 1D array access

Through this example, we know that by counting the number of vector accesses which have same length x ($x = [1,...,vec\_length]$), the final conflict result can be obtained by a reduction:

$$total\_conflict = \sum_{i=1}^{vec\_length} C_i \times conflict_i$$

$conflict_i$ is the conflict number of the vector access that has i active threads. C_i is the number of vector access which length is i. The time complexity of this method doesn't depend on the iteration number; it is only related to the *vec_length* which is determined by hardware design. This method requires a preprocess procedure to calculate C_i ($i = [1,...,vec\_length]$). As shown in Figure 5.8, there are certain distribution patterns for these vector accesses, and it is not hard to calculate. Algorithm 5.1 describes the method:

algorithm 5.1 *for_if_analysis*
Input: bank_num, W, N, M, stride, offset
Output: res -- bank conflict number.

```
//-------------------------------------------------------
    if iter_num <=vec_length)
        conflict_sum = frac_1D_single_block();  //calculate fractional warp access conflict
```

```
        else
//calculate max warp aligned access number for one iteration.
            int line_max_aligned_num = iter_num/vec_length - 1;
            //calculate distinct offset case number
            sample_num = distinct_case_num (vec_length, stride, W,
elemSize, bank_num, line_max_aligned_num);
//calculate repeat number of each case:
            for (int i=0; i<sample_num; i++)
                num_per_case[i] = get_case_repeat_num(line_max_aligned_num, sample_num);
            end for

            for (int i = 0; i < sample_num; i++)
        cur_conflict=single_block(bankNum, W, N, gridInfo, blockInfo,
       arrayInfo, pad);
                total_conflict_num = get_case_conflict(cur_conflict, num_per_case);
            end for
end if
```

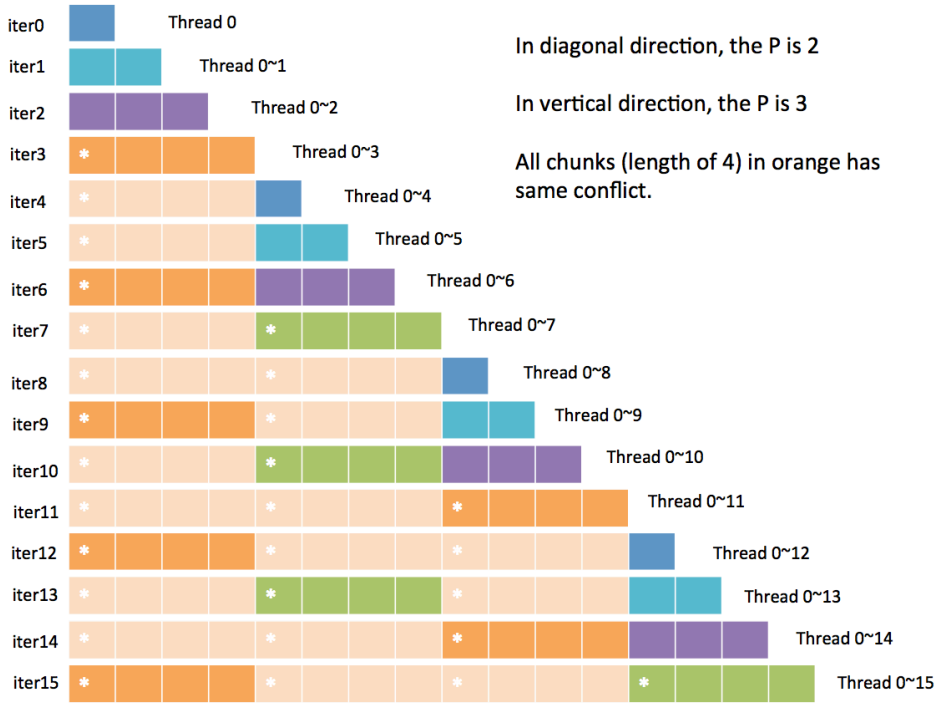


Figure 5.9 Analysis of the example of "for-if "scenario for 2D triangular access

When array is 2D, and the "for-if" filter is used access a triangular area, it becomes more complex to calculate the number of vector accesses that have same thread number. Figure 5.9 shows an example for this case (The vertical direction is the iteration index, the horizontal direction is the active thread index). In this figure, the vector accesses that use part of a vector/warp are marked in blue, light blue, and purple. For each of these

colors, the distance between consequential two vector accesses is constant. It means that we can find a P (distinct case number) and then get the total conflict number.

For the vector accesses that use all threads of a warp, they appear periodically in two dimensions: one is in the diagonal direction, the other is in the vertical direction. Following steps can be used to calculate the total conflict:

1) Calculate P_d, calculate P_v.
2) For each case j ($j = [1,...,P_v]$)
   a) Calculate conflict for P_d cases;
   b) Calculate repeat times of case k ($k = [1,...,P_d]$)
   c) Calculate conflict summary of k cases and save in $conflict\_d[j]$
3) Calculate summary of array *conflict_d*, which is the final result.

The proposed method calculates P_d which denotes the distinct case number in diagonal direction, and P_v which denotes the distinct case number in vertical direction. For the example in figure 5.12, the Pd in this direction is 2, and P_v is 3. In the second step, for each k in [1,.., P_v], calculate the conflict summary in diagonal direction. In the final step, add up all the conflict numbers and get the final conflict result. For the example in figure 5.12, all of the dark orange warps have same access pattern and same conflict number; all the green warps have same access pattern and same conflict number. The repeat number of each color can be calculated given P_d, P_k, and *iter_num*.

5.5 Summary

This chapter presents the conflict analysis of single array access expression. In the GPU programming model, one expression can drive many threads from different warps to visit memory. In addition, by using "for" loops, "if" condition thread filter, or "for-if" combination wrapper, the program can control the working thread number and control the area to be visited. This chapter presents the solutions for these scenarios. By studying how the access offset varies for different vectors/warps and for different iterations, the

proposed methods can realize conflict analysis while its time complexity is independent of warp number and iteration number. With these solutions, now we can process the array access expressions in a kernel one after another.

CHAPTER 6 PARAMETER OPTIMIZATION STRATEGY

In this chapter, the parameter optimization strategy is presented to obtain the optimal or sub-optimal inter-padding size, intra-padding size, and bank access bit-width. Based on the conflict analysis modules introduced in chapter 4 and chapter 5, this parameter optimization procedure looks for an optimal or sub-optimal data layout solution for all arrays in a kernel. Section 6.1 introduces the parameter optimization space. In this space each solution could have different value of inter-padding size, intra-padding size, and bank mapping functions. Section 6.2 studies the impact of offset on conflict number; this information is helpful for inter-padding size optimization. Section 6.3 studies the potential intra-padding size searching boundaries. It helps to clarify the maximum workload size for intra-padding size optimization. In section 6.4, the overall framework of parameter optimization engine is presented, and some related optimizations are discussed.

6.1 Parameter Optimization Space

As mentioned in chapter 3, this space is three-dimensional in that the bank mapping function, the inter-padding size, and the intra-padding size varies. Each of these parameters is related to one another and could have impact on each other. For example, for different bank mapping function, to eliminate conflict, the optimal intra-padding and inter-padding size could be different; by changing the intra-padding size of one array, the base address of other arrays could be changed and in turn the inter-padding size for them could be different.

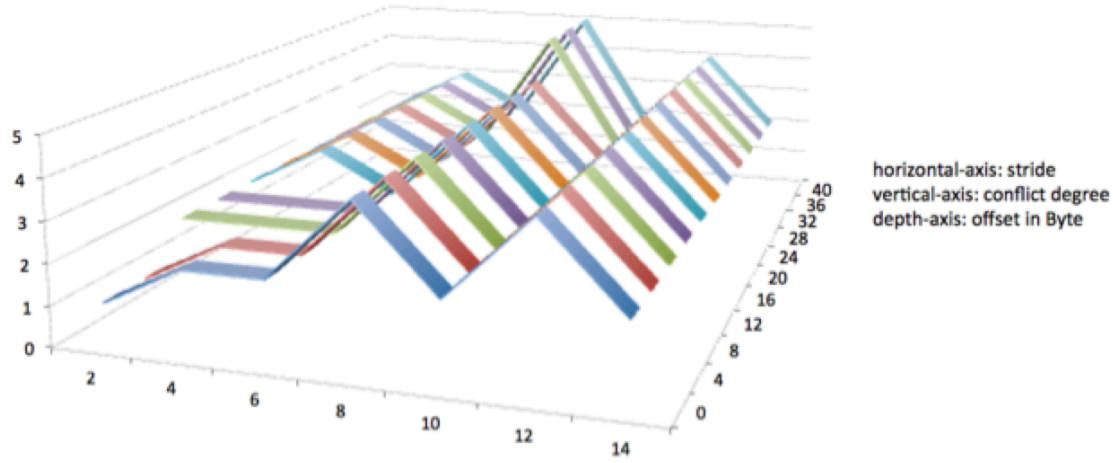
There are limitations related to this parameter optimization space. First of all, bank-mapping function is unique for a whole kernel. In other word, all arrays of same kernel share the same bank access bit-width. Secondly, intra-padding of an array impacts all accesses of this array. So the decision of intra-padding size needs to be made based on the overall conflict number of this array. The available memory size is limited, which sets

a limitation for inter-padding and intra-padding size optimization. These limitations impact the structure of parameter optimization engine.

6.2 Inter-padding Optimization

6.2.1 1D Strides

Inter-padding changes the array base address by adding dummy space in front of the array. As mentioned in chapter 4, for the conventional mapping function, offset has no impact on conflict degree. However, for dynamic bit-width bank access, offset could cause extra bank conflict. Figure 6.1shows the impact of offset. The vertical axis is conflict degree number; horizontal axis is even stride value, and depth axis is offset varies from 0 to 40. As shown in this figure, for some strides, conflict degree changes when offset value increases. In this section, the offset impact for row-major bank mapping function and column-major bank mapping function are briefly described. The purpose is to (1). Figure out the cases for which the inter-padding doesn't change conflict number; (2). Understand the potential padding size boundary. This is helpful to understand the inter-pad optimization workload and to reduce the workload.



Row-major bank mapping, *N=M=4*, *W=8*, *stride=[2,…,14]*, *offset=[0,4,8,……,40]*

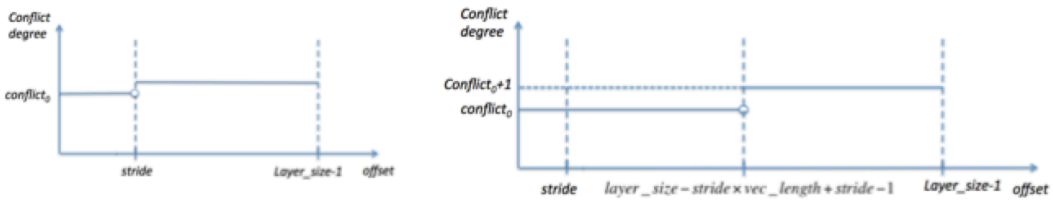
Figure 6.1 offset impacts on conflict number

Row-major Bank Mapping Function

For this mapping function, odd strides cause no conflictfor any offset. For even strides, we divide them into two categories: power-of-two strides, and other even strides.

Power-of-two strides

When $stride \geq \frac{W}{N}$, the scope of current vector access is layer-size aligned. This can be proved by $\mathrm{mod}(vec\_access\_scope, layer\_size) = 0$. Figure 6.2 (a) is how conflict degree changes with increasing offset. In this figure, $layer\_size = \frac{W}{N} \times bank\_num$, and $conflict_0$ is the conflict degree when $offset = 0$. When $stride < \frac{W}{N}$, the vector visiting scope is smaller than a layer. Then there are three ranges (as Figure 6.2 (b)).



(a). $stride \geq \frac{W}{N}$ (b). $stride < \frac{W}{N}$

Figure 6.2 impact of offset on conflict degree for power-of-two strides

Other even strides

For other even strides, we calculate the parameters in stride family expression $\sigma \times 2^e$, with $e > 0$, and $\sigma = \{3,5,7,9,11,...\}$. It has following features:

- Its visit scope across $\sigma \times 2^e$ rows; among every σ rows, there is no conflict.
- The shortest distance between a conflict pair is:
  - $\frac{vec\_length}{2^e} \times stride = vec\_length \times \sigma$ in unit of element;
  - and σ in unit of row;

- Each element conflicts with $2^e - 1$ other elements.

Figure 6.3 is an example. When $stride = 6$, the first three rows has no conflict, and the next three rows also have no conflict. When $\sigma > \frac{W}{N}$, each of the visited sites that conflict in the same bank is located in a different layer. It means that applying any offset cannot change the conflict degree. Otherwise, the conflict needs to be calculated.

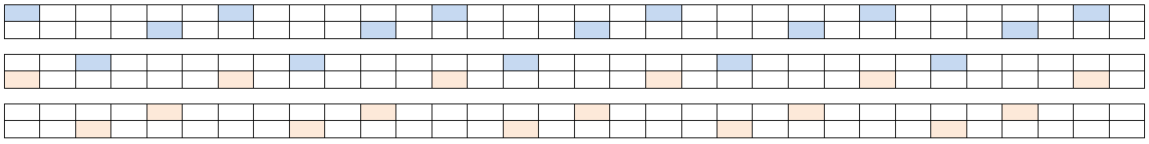


Figure 6.3 Conflict of stride=3, W=8, N=4, M=4

In conclusion, for even strides, when it is power-of-two, the offset impacts the conflict number, and the conflict need to be calculated for each different offset. For other even strides, when $\sigma > \frac{W}{N}$, the offset has no impact on conflict degree; otherwise, the conflict needs to be calculated for each different offset. Whenever inter-padding size optimization is necessary, the maximum padding size is $layer\_size - 1$ ($layer\_size$ is the number of elements that can be stored in one layer of all banks).

Column-major Bank Mapping Function:

For this bank mapping function, in each layer, the data is mapped to banks in column-major direction. Each column is one layer of one bank, which can host $\frac{W}{M}$ elements. For each pair of elements that conflict, when an offset is added, two elements might get different shifting distances in column direction and in horizontal direction. Remember in chapter 4, when calculate bank conflict for column-major bank mapping function, we consider $R = \frac{W}{M}$ cases: case i starts its visit from ith row. For each case i, the first R visited sites are calculated as saved for further usage. Here, when an offset is added, we can obtain the new start row index as $i' = \mathrm{mod}(offset + i, R)$. Then the conflict

result becomes as same as the case i except that conflicts is shifted in horizontal direction. Figure 6.4 is an example with $R = 4$. After adding an offset, it has:

$$i' = \mathrm{mod}(i + offset, R) = \mathrm{mod}(i + c, R) \xrightarrow{i=3, c=2} i' = 1$$

The new conflict pattern is of $i' = 1$ and is shifted to the right. This means the maximum offset we need to check is R.

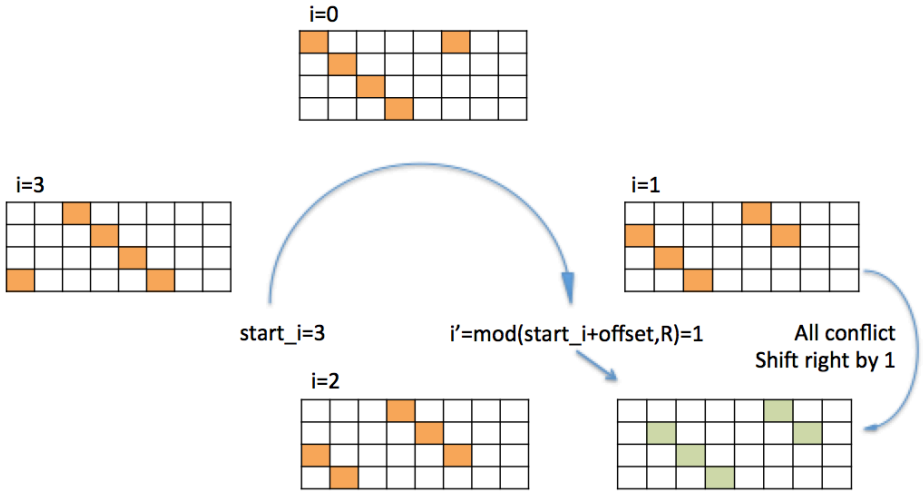


Figure 6.4 Map the conflict pattern of a offset to one of *R* known distinct cases

6.2.2 2D Strides

For both row-major mapping function and column major bank mapping function, the solution is to reuse the single vector conflict table (introduced in section 4.3.2) to calculate conflict for different offset values.

6.3 Intra-padding Optimization

Intra-padding optimization is looking for a stride that causes no conflict or least conflicts. With padding, the array layout is changed, and the memory access stride is changed as well.

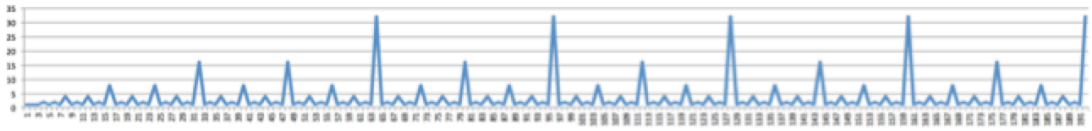
In this procedure, one of the concerns is about the upper bound of the intra-padding size searching. Normally the padding size is small when it reaches the goal. In this

section, we briefly discuss the padding size upper bound. This is helpful to understand the workload of the intra-padding optimization.

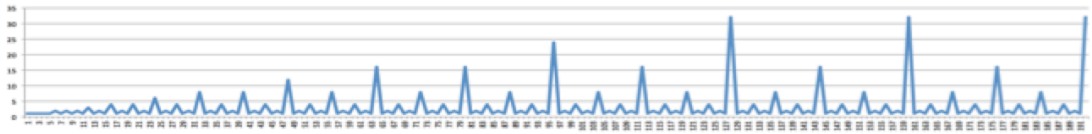
6.3.1 1D Strides

Row-major Bank Mapping Function:

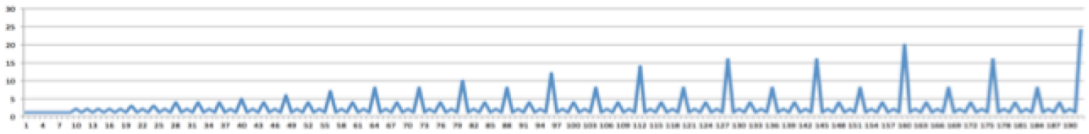
Figure 6.5 is how conflict degree varies when stride changes. As mentioned, when the stride is odd, there is no conflict. For any even stride, by replacing it with the first odd stride that is larger than it, the conflict degree is reduced to "1".



(a). Row-major bank mapping, *N=M=4*, *W=8*, *stride=[1,...,192]*, *max(pad)=4B*



(b). Row-major bank mapping, *N=M=4*, *W=16*, *stride=[1,...,192]*, *max(pad)=4B*



(c). Row-major bank mapping, *N=M=4*, *W=32*, *stride=[1,...,192]*, *max(pad)=4B*

Figure 6.5 Conflict degree examples for row-major bank mapping function

Column-major Bank Mapping Function:

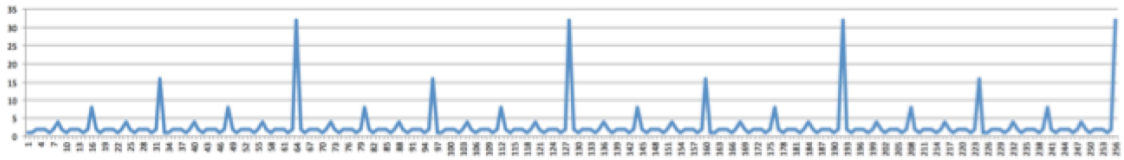
Figure 6.6 shows how conflict degree changes while stride increases. Different from row-major bank mapping functions, both even and odd strides could cause conflict. However, in chapter 4, we mentioned that there are even strides that only access elements that are located in the first row of each layer. In table A-1-5, this case is described and its conflict estimation method is presented. Basically, when stride is larger than $\sigma \times R$ ($R = \frac{W}{M}$, and $\sigma = \{1,3,5,7,9,11,...\}$), the visits locations are fall into the first row of all layers, and the problem is transformed to the one of conventional bank mapping function,

with stride is replaced by $\frac{stride}{R}$. An example is $R=2$, $stride=6$ and $\sigma=3$: all visited sites lie in the first row of layers, and it becomes an odd stride access on conventional interleaved banks.

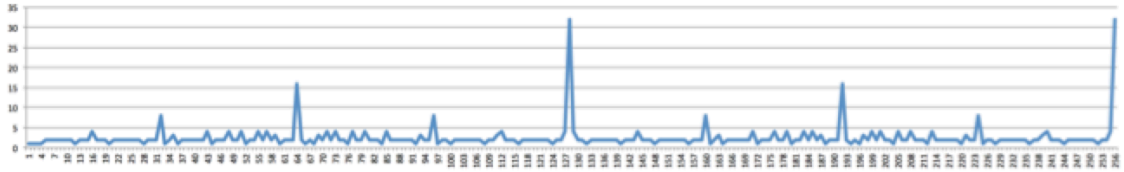
This knowledge helps to determine the intra-padding size searching upper bound. For any stride, we can locate the next stride equals to $\sigma \times R$, for which the conflict degree is 1. The distance between two such zero-conflict strides is

$$\sigma_2 \times R - \sigma_1 \times R \xrightarrow{\sigma_2-\sigma_1=2} 2 \times R$$

It means that when current stride causes conflict, the maximum padding size upper bound is $2 \times R - 1$. In figure 6.6, the case (a) has $R=2$ and the padding upper bound is 3; (b) has $R=4$ and padding upper bound is 7; (c) has $R=8$ and the padding upper bound is 15.



(a). Column-major bank mapping, *N=8, M=4, W=8, stride=[1,...,256]. max(pad)=3 × 4B*



(b). Column-major bank mapping, *N=16, M=4, W=16, stride=[1,...,256]. max(pad)=7 × 4B*

(c). Column-major bank mapping, *N=32, M=4, W=32, stride=[1,...,256]. max(pad)=15 × 4B*

Figure 6.6 Conflict degree examples for column-major bank mapping function

6.3.2 2D Strides

Basically, for row-major bank mapping function, the intra-padding size upper bound equals to *bank_num*. Given a 2D stride <*stride_x, repeat_x, stride_y, repeat_y*>, we

denote the index of one element as <*idx_x, idx_y*> ($idx\_x = [0,...,repeat\_x - 1]$, and $idx\_y = [0,...,repeat\_y - 1]$). When the conflict happens between a pair of elements that belong to different *idx_y*, the horizontal distance between these two elements is periodic and the period is *bank_num*. For example, when the horizontal difference of two sites is 3, and $bank\_num = 32$, after adding 32 to the distance between them, the horizontal difference of these two sites becomes the same. Intra-padding size optimization is to find the padding size that eliminates a conflict pair by changing the horizontal distance. This means that the maximum intra-padding size should be less than *bank_num*.

For column-major bank mapping function, the upper-bound is $bank\_num \times \frac{W}{M} - 1$. The reason is similar. After the stride is added by $bank\_num \times \frac{W}{M}$ (which is also the number of elements that can be stored by one layer), the horizontal distance between a pair of conflict elements remains the same. So, the maximum padding size should be less than this.

6.4 Parameter Optimization Algorithm

Figure 6.7 is the framework of the parameter optimization procedure. The outer most loop iterates over different bank access bit-width. Then, for each array, an initial investigation is used to collect information, which will be used for inter-padding and intra-padding size optimization. When optimize the intra-padding size, a range of padding sizes are applied to this array, and the corresponding conflict number is calculated and stored the in the intra-padding option list of this array. (If multiple arrays have same access pattern, only the first one is processed, and other array can share the same padding size.) After obtained the intra-padding option lists for all arrays, the next step is to find a solution that meet following requirements:

1) The total memory size used by intra-paddings of all arrays is within the maximum free memory size.
2) For each array, this solution gives an optimal or sub-optimal intra-padding solution.

The procedure is in the left-bottom part of Figure 6.7. Normally the option number for each array is relatively small, and we can calculate the final solution by exhaustive enumeration. Otherwise, extra strategies need to be adopted to reduce the workload.

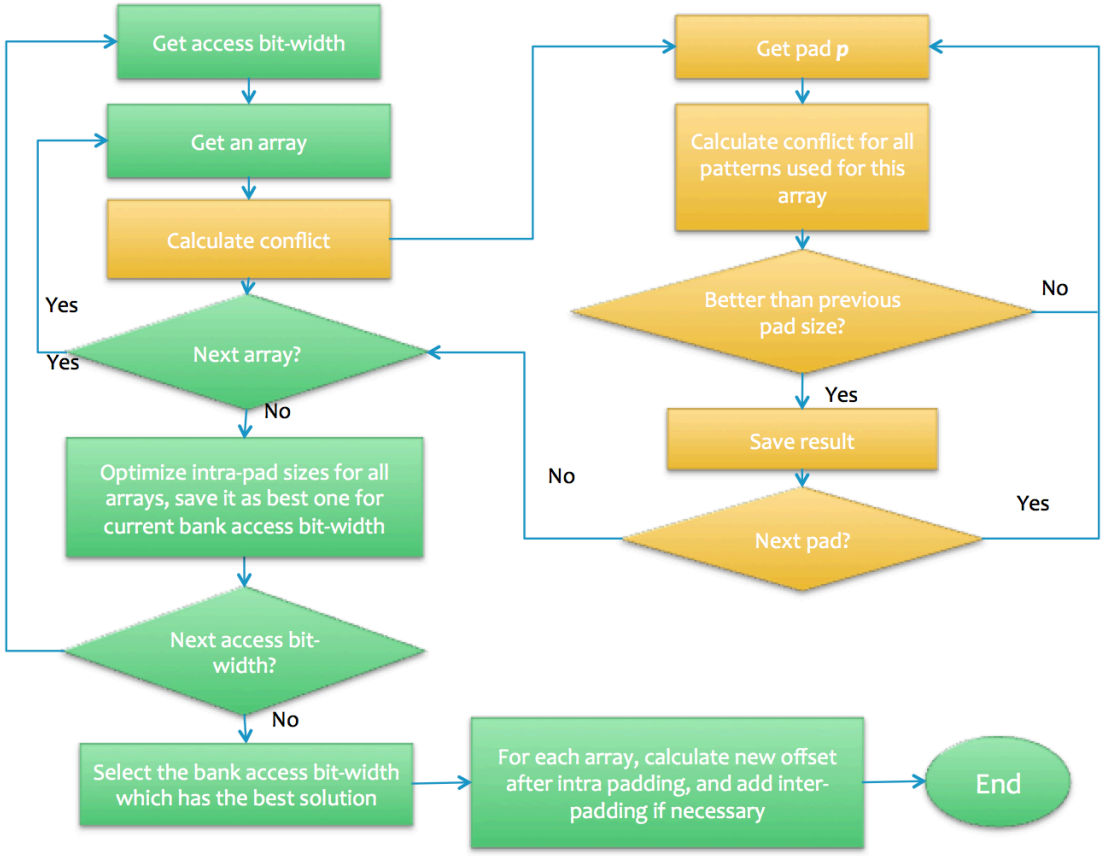


Figure 6.7 Parameter optimization strategy

After intra-padding sizes of all arrays are determined, the inter-padding is used to reduce the remaining conflict. This intra-first-inter-second padding strategy was mentioned in [54], and it is adopted here. Before intra-padding optimization, a certain size of memory is reserved before intra-padding size optimization. After intra-padding, following steps are used to determine the optimal offset for each array:

1) Get an array,
   a) If it is the first array, apply the intra-padding and update the variable *overall_offset* to denote the first position after this array. If there is any array left, go to 1; otherwise, go to 5.
   b) Otherwise, apply the intra-padding, and go to 2.

2) Update array's offset based on the current value of *overall_offset*;
3) Looking for the optimal offset for this array, which is the dummy variable size need to be inserted before it;
4) Update the value of *overall_offset* (including intra-padded size of current array and dummy variable size inserted for this array). If there is any array left, go to 1; otherwise, go to 5.
5) Terminate.

When generating intra-padding option list for each array, if the conflict number of the current padding size is larger than the previous one, it is ignored; otherwise, the padding size and the conflict number are stored by pending it to the end of the option list. This strategy guarantees that the option list has following two features:

1) The conflict number decreases while node index increase;
2) The padding size increases while node index increase.

These features can be used to reduce the workload of intra-padding size optimization. Following two methods use of this information to find optimal or sub-optimal intra-padding solution for all arrays from their option lists.

Method One

Figure 6.8 is an example illustrating the first method. In this figure, there are three arrays: A, B, and C. for each array there are multiple padding options saved in a list. The first step is to find the array that has maximum number of options, and save this option number as *max_option_num*. For other arrays, by repeating the last (optimal) option, extend their list to have *max_option_num* elements. In figure 6.9, the horizontal direction is the option index. For each index *i*, there is a column in which there is one option for each array. Here we denote this column as a set G. If any option in G moves to the left, then the total conflict of G increase, and the total pad size decrease; if any options in G move to the right, the conflict number of G decrease and the total pad size increase.

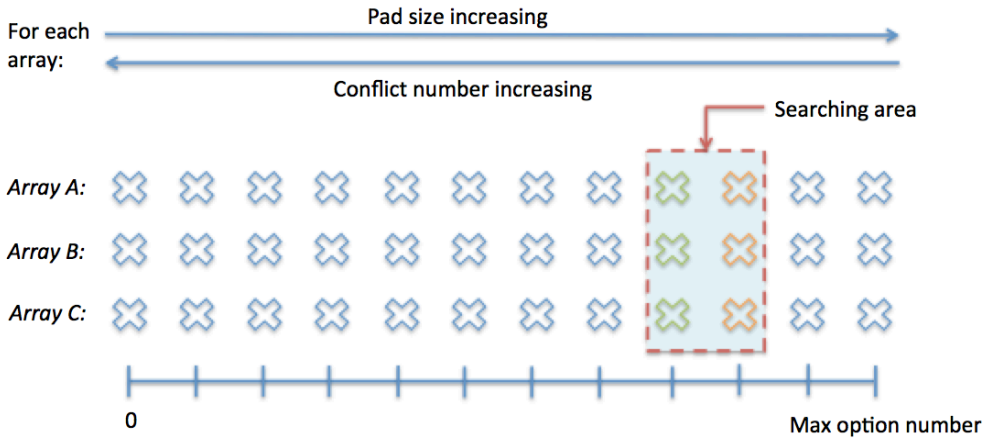


Figure 6.8 Option one: Strategy to reduce workload for intra-padding optimization

Based on this observation, we start from $i = \max\_option\_num - 1$ to $i = 0$ to find the first column G (column in green in Figure 6.8) that can be satisfied by available memory size. Then, denote the next column on the right as G_b (column in orange in Figure 6.8), which is the last column that needs memory space larger than the free memory size.

- Option one: Between G_a and G_b, there are some candidates that can be used as the final solution. An example is shown in Figure 6.9. In this figure, there are 3 arrays. Between G_a and G_b and including G_a, there are 7 options.
- Option two: Start from padding options in G_b, always chooses the pad that is acceptable (not exceed the available memory size) and can reduce conflict number most.

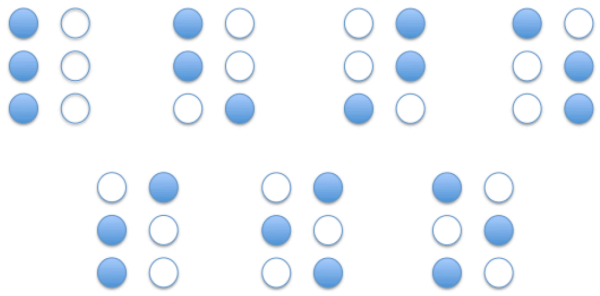


Figure 6.9 Candidate solutions in area of two columns

For both options, by including more neighbor columns in the candidates' area, the number of solution candidates increases. It helps to avoid missing the solutions in which the options are far away from each other in horizontal direction. We need to consider the

balance between choosing the width of candidates' area and the execution time. The simplest solution is to directly take G_a as the final solution. It is the fastest, and good for the case that intra-paddings for all arrays can be accepted. However, for other cases, it has higher risk of missing better solutions.

Method Two

The second option is a greedy method. Figure 6.10 is an example illustrating the procedure. It starts from the left most column and take it as the front edge of optimization. The options in the column are called the nodes on the front edge. At the beginning, in the first column, it looks for the array that can reduce maximum number of conflict, and accept it. For this array, move the front edge node one step ahead. Then for this updated front edge, repeat the same action, until either the conflict numbers of all arrays are zero, or the free memory space is used up. This procedure also can be refined: for current front edge, after find the best step, it can hold to see whether there is any option combination that is better than this option. "Better" means that it uses less memory space but reduces more conflicts. For the example in Figure 6.10, after step 0, the next option for array A can reduce conflict by "7", which is the maximum number among "7", "5", and "4". However, since the combination of options for array B and C is "9" which is better than A's, the front edge nodes for B and C will be moved ahead instead of taking the option for A. Here the combination size (array number in one combination) is the key fact that impacts computing time. If more combinations are considered, then the workload increases dramatically.

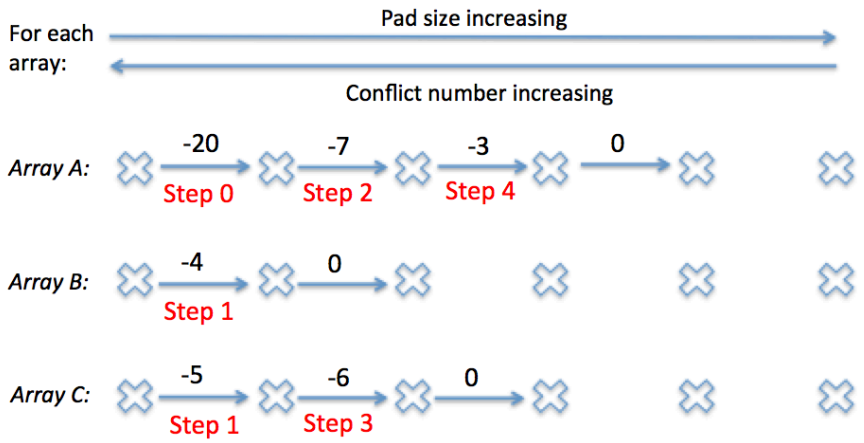


Figure 6.10 Option two: Strategy to reduce workload for intra-padding optimization

Optimization by Progressive Strategy

As mentioned in this section, the outer most loop iterates over all bank access bit-width. This could be a fact that hurt the intra-padding size optimization efficiency. An example is the application 3DFD (This application is described in section 7.1). Using the introduced padding size optimization method, we found the first option with zero-conflict is $< N = 4, pad = 24 >$. It means that 24 different padding sizes are tested to get this result. The second option is $< N = 8, pad = 0 >$, which the first option is when bit-width is 8B. Between these two candidates, the second one is normally more preferred than the first one because:

- It is the first option for $N = 8$ which is easy to find when N equals to 8,
- It needs no extra memory.

To reduce the workload, one optimization is to divide the padding size lookup scope into pieces $[range_0 ... range_t]$, and progressively looks for the solution. Firstly, for different N, calculate the conflict for each padding size in $range_0$. If the final intra-padding solution for all arrays can be found, then the parameter optimization procedure ends. Otherwise, calculate conflict for each padding size in $range_1$ and look for solution in $range_0$ and $range_1$. If no solution is found, extend to next padding range until the solution is found or the padding exceeds the valid range. Since normally the padding size is small, then the $padding = [0,1,2]$ can be used as the first range.

6.5 Summary

This chapter describes the parameter optimization procedure. Different bank access bit-widths are enumerated one by one in the outer most loop of this procedure. Intra-padding size is optimized inside a valid padding size range, and multiple candidates are stored a list for each array. From these lists final intra-padding solution for the whole kernel is derived. After that, inter-padding is used for further conflict optimization. Intra-padding size optimization is realized by two strategies. The first one looks for a potential candidate's area and select final solution from it; the second one reduces conflict step by step by always choosing the best-known option. Finally, the valid intra-padding range is divided into pieces. By looking for the solution progressively, it can found small intra-padding solutions more quickly.

CHAPTER 7 APPLICATION STUDY

7.1 3DFD

In this kernel, there is a 2D array allocated in shared memory. This buffer is used to save intermediate calculation results. The code structure is as in Figure 7.1. The code accesses array data in 2D rectangular access pattern, and warp accesses a 2D data block. The different array accesses in the code have different offsets in X and Y direction. Figure 7.2 is an example of this access pattern.

For loop

Read data to shared memory

Do calculations in shared memory

End for loop

Figure 7.1 Kernel structure of 3DFD

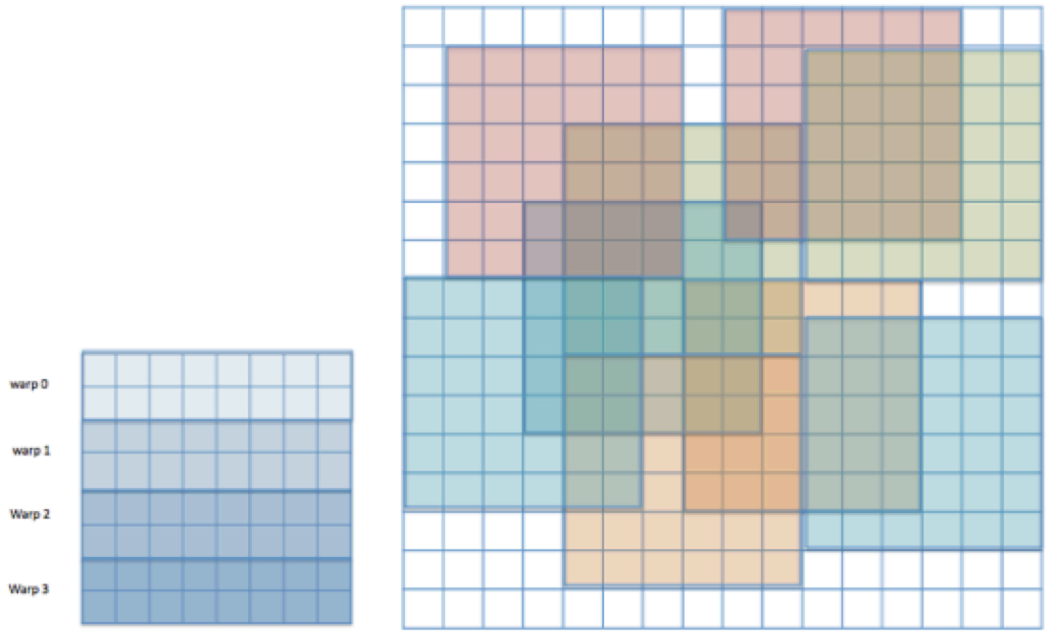


Figure 7.2 Memory access pattern of 3DFD

7.2 ConvolutionSeperable: convolutionRowsKernel

This kernel allocates a 2D rectangular array in shared memory. Firstly, through a "for" loops data are read from global memory to shared memory. After thread synchronization, a 2-level nested loop computes the results. In the loop body, the array in shared memory is used as input for the calculation. The code structure is shown in Figure 7.3, and the access pattern is shown in Figure 7.4. The loop in this application is different from the one in 3DFD: the iteration variable is used in array sub-index function. For different iterations, the access offsets are different. As mentioned in section 5.4, conflict numbers of the P distinct iteration cases are calculated, then the finally conflict number are calculated through a reduction.

```
For loop
    Read data to shared memory
End for
……
For loop
    Read data to shared memory
End for
For loop
    For loop
        Read input data from shared memory and compute.
    End for loop
End for loop
```

Figure 7.3 Kernel structure of convolutionRowKernel

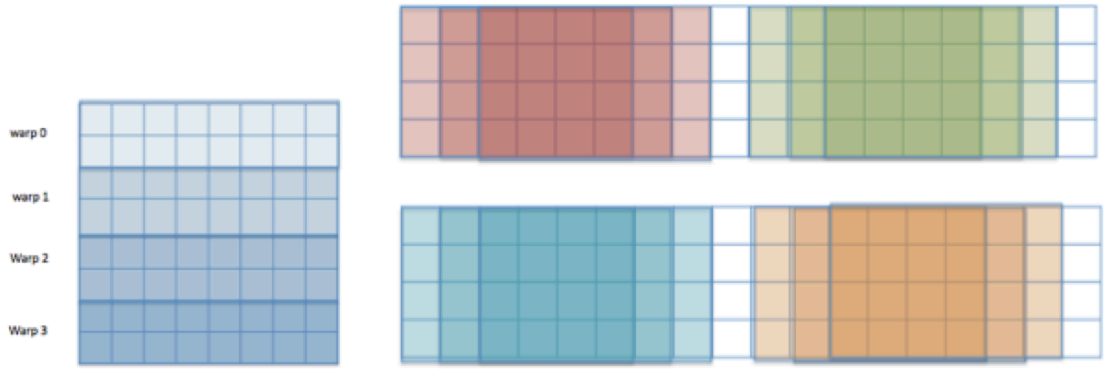


Figure 7.4 Memory access pattern of convolutionRowKernel

7.3 ConvolutionSeperable: convolutionColKernel

Similar to the kernel in 7.2, this kernel allocates a 2D array in shared memory and uses 2D warps to access the array data (Figure 7.5). The code structure is also similar to the kernel in section 7.2. The main difference lies in the nested computation loop: the column-major access pattern is used to get data.

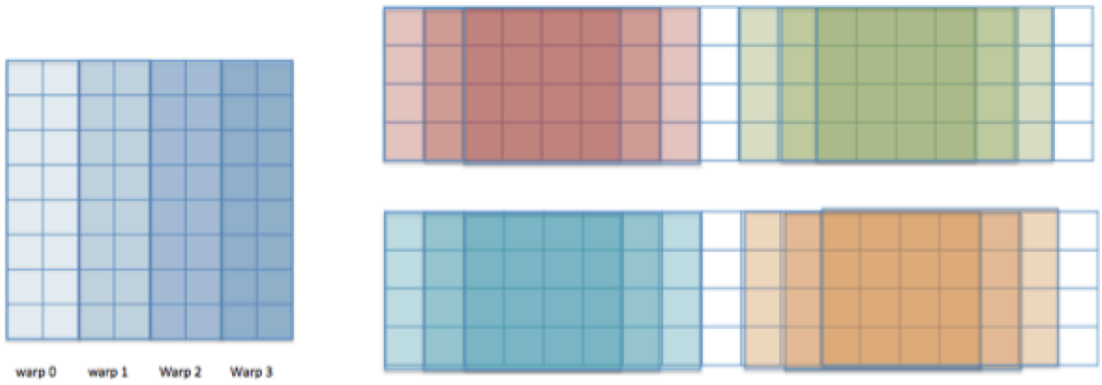


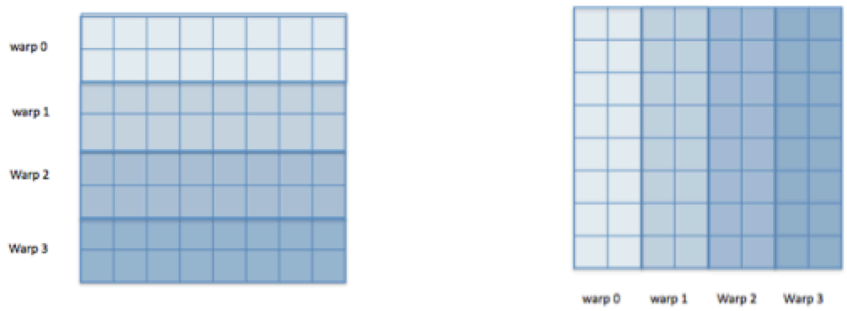
Figure 7.5 Memory access pattern of convolutionColKernel

7.4 Transpose: TransposeCoalesed, TansposeDiagonal, TransposeFineGrained

In these kernels, a 2D array is allocated in shared memory. This array is read and written in a 2-level nested loop. The iteration variable of the outer loop has no impact on array access addresses. For the inner loops, the iteration variable is used in array sub-indexing expression. The 2D array is mainly used to avoid the penalty of un-coalesced global memory access. For the first inner loop, the 2D warp reads 2D block of data in row-major pattern, which causes no conflict. For the second inner loop, the array is read in column-major direction, and it causes conflicts. The code structure is shown in Figure 7.6 the main features of these kernels.

```
For loop
    For loop
        Read data from device memory to shared memory
    End for loop
    For Loop
        Read data from shared memory to device memory
    End for loop
End for loop
```

Figure 7.6 Kernel structure of TransposeCoalesed



(a) First inner loop (2) Second inner loop

Figure 7.7 Memory access pattern of TransposeCoalesed

7.5 Transpose: TransposeCoalesedGrained

As same as kernels in section 7.4, a 2D array is read and written inside a 2-level nested loop. This array has a pad which size is 1. The outer loop's iteration variable has no impact of array access; for the inner loop, the iteration variable is used in array sub-indexing expression to change offset. For both the first and the second inner loop, the 2D warp read 2D block of data in row-major direction.

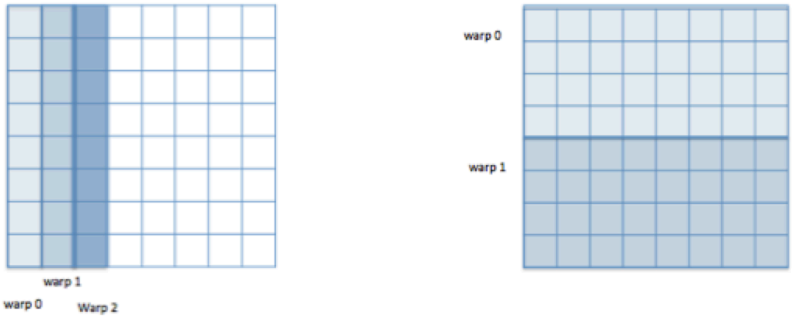
7.6 shfl_scan: shfl_vertical_shfl

In this kernel, a 2D array is allocated in shared memory. This array is read and written inside a loop body, the iteration variable has no impact of array access. There are

multiple access patterns used to visit the array elements (Figure 7.9). 1D warps access data in column-major direction (pattern A), and 2D warps access data in row-major direction (pattern B). The code structure is shown in Figure 7.8.

```
Write to shared memory in pattern A;
For loop
    Write to shared memory in pattern A;
    Read from shared memory in pattern B;
    Do computing;
    Write to shared memory in pattern B;
    if (threadIdx.y > 0)
        Read shared memory in pattern A;
    Read from shared memory in pattern A.
End for loop
```

Figure 7.8 Kernel structure shfl_vertical_shfl



(a) Pattern A (2) Pattern B

Figure 7.9 Memory access pattern shfl_vertical_shfl

7.7 lud: lud_diagonal

In this kernel, a 1D thread block visits the columns of a 2D array in shared memory. Memory operations are warped by “for” loops, and “if” condition statement. For the “for” loops, their iteration variable could have or not have impact on sub-indexing functions. The “if” statement impacts the conflict number by allowing different set of threads to access the data in shared memory. The code structure is shown in Figure 7.10 lists the main features of this kernel.

```
For loop
    Write in row-major direction.
End for loop
For loop
    If thread ID larger than S
        For loop
            Do calculation: read and write data in column-major direction
    Sync;
    If thread ID larger than S
        For loop
            Do calculation: read and write array in row-major direction
End for loop
For loop
    Read in row-major direction.
End for loop
```

Figure 7.10 Kernel structure of lud_diagonal

7.8 lud: lud_perimeter

In this kernel, a 1D warp is used to visit rows or columns of a 2D array allocated in shared memory. The thread block has only one warp: an "if" condition statement divides it into a group of the first 16 threads and a group of the remaining 16 threads. Each group visits array rows or columns. The code structure is shown in Figure 7.11 lists the main features of this kernel.

```
If thread Id less than D
    For loop
        Write in row-major direction.
    End for loop
    For loop
        Write in row-major direction.
    End for loop
Else
    For loop
        Write in row-major direction.
    End for loop
    For loop
        Write in row-major direction.
    End for loop
End if

If thread Id less than D
    For loop
        For loop
            Read and write in row-major direction.
    End for loop
Else
    For loop
        For loop
            Read and write in column-major direction.
    End for loop
End if
……
```

Figure 7.11 Kernel structure of lud_perimeter

7.9 NW

In this kernel, multiple 2D arrays are allocated in shared memory. Shared memory operations are warped by “for” loop, “if” condition statement, and “for-if” combination. For the “for-if” wrapped cases, the data is accessed in diagonal directions (Figure 7.13). The code structure is shown in Figure 7.12 lists the main related features of this kernel.

```
If thread ID equals to 0
    Write array in column-major direction.
End if

For loop
    Write array in row-major way. Iteration variable impact offset.
End for

Write array in column-major way.
Write array in row-major way.

For loop
    If thread ID less or equal to iteration variable
        Calculate array x and y sub-index based on thread ID and iteration variable.
        Access array according to these sub-index.
    Endif
End for

For loop
    If thread ID less or equal to iteration variable
        Calculate array x and y sub-index based on thread ID and iteration variable.
        Access array according to these sub-index.
    Endif
End for

For loop
    Write array in row-major way. Iteration variable impact offset.
End for
```

Figure 7.12 Kernel structure of nw

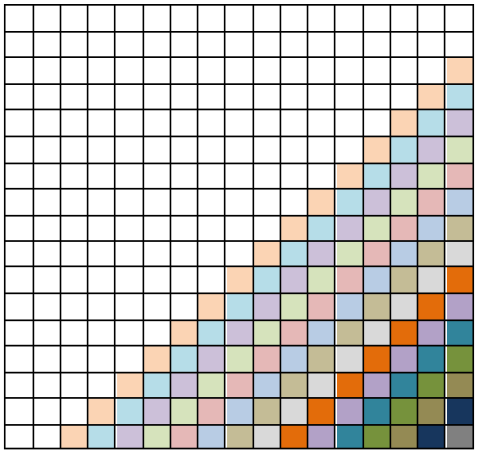


Figure 7.13 Memory access pattern of nw

7.10 Summary

This chapter introduces the applications that are used to test the proposed optimization tool. These applications are selected from RODINIA and NVIDIA CUDA SDK. Some of them are commonly used benchmarks that are helpful for understanding typical computation workload and testing devices such as GPUs. The performances of these kernels are suffered from shared memory bank conflict penalty. Basically, these kernels perform 1D/2D accesses to arrays, and some kernels have multiple arrays. They also include cases that uses “for” loop, “if” condition thread filters, and “for-if” combination to control the memory accesses.

CHAPTER 8 PERFORMANCE EXPERIMENTS

In this chapter we test the performance and efficiency of proposed analysis tool. Section 8.1 includes three basic experiments that test the execution time of conflict analysis modules. Section 8.2 presents the optimization results of 13 kernels by using proposed analysis tool. The platform info is as following:

- GPU device: Tesla K20c,
  - Shared memory:
    - 49152B per SM;
    - Bank number is 32;
    - W is 8B.
  - Warp size: 32 threads.
  - Compute capability: 3.5
- Programming model: CUDA 5.0
- Profiler: NVIDIA NVPROF, release 5.0

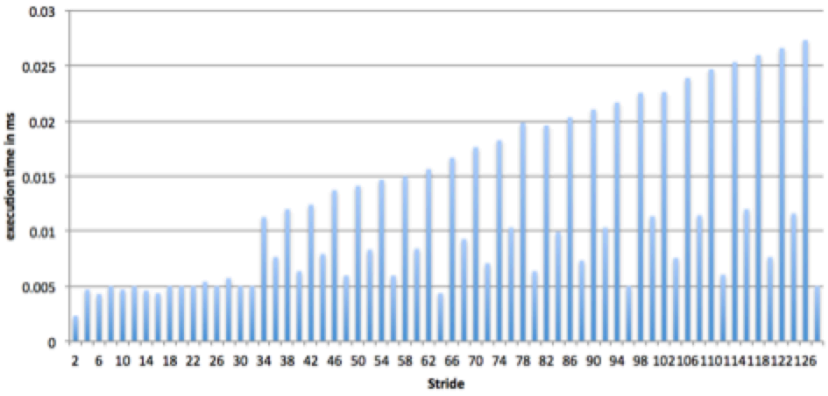
8.1 Conflict Analysis Time Experiments

In GPU programming model, many threads execute in parallel according to one GPU kernel. When memory bank conflict is the bottleneck, it would be helpful to have a static bank conflict analysis tool that can find an optimization solution within a limited period. In this section, three experiments are used to test the conflict analysis time.

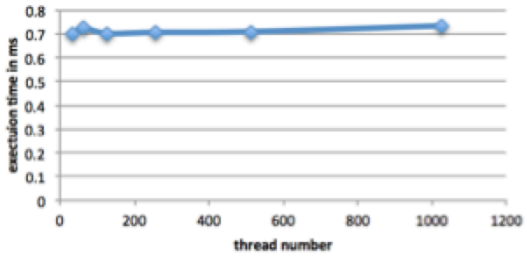
The first experiment exams the analysis execution time of multi-warp memory access. Figure 8.1 (a) is single warp 1D stride analysis execution time. The x-axis is the stride value, and the y-axis is the execution time. As it shows, the analysis time is related to stride value. For existing GPU devices that support dynamic bank access width (W=8B), when multiple warps share same 1D even stride, they have the same in layer offset. It means that the overall conflict number can be obtained by $conflict_0 \times warp\_num$ ($conflict_0$ is the conflict number of warp 0). Then we don't need

to test 1D stride multi-warp analysis time. Figure 8.1 (b) is an experiment of 2D stride analysis efficiency. The x-axis is the thread number (the increasing step is 32, which is the thread number per warp); the y-axis is the execution time. For 2D stride cases, each warp might have different in-layer offsets. The distinct case number P needs to be calculated and then the final conflict number is obtained. As it shows, the execution time remains relatively constant. The reason is that the value of P remains the same no matter how warp number increases.

Normally GPU kernels are executed by many thread blocks. Each block has its own shared memory space and usually uses shared memory in similar way. For such kernels, the proposed tool only needs to analysis one block, and other blocks can share the solution to improve performance.



(a). 1D stride single warp analysis execution time.



(b). Execution time of 2D strides multiple warp analysis

Warp is 2 × 16 and warp number is 8; stride is *<1, 16, 24, 16>*

Figure 8.1 analysis module execution time.

The second experiment tests the analysis efficiency of "for" loop wrapped memory accesses (To simplify the experiment, we use strides that are power-of-two. For other

even stride and 2D strides, the loop analysis routine works in same way.). Two loop examples are shown in Figure 8.2. In the first example memory accesses are warped by a "for" loop and the iteration variable has no impact on memory addressing. For this case, the bank conflict analysis is performed once for the first iteration and then the overall bank conflict number can be calculated.

```
For i in {100, 200, 300, ... , 1000}
        Array[threadIdx.x *stride]  =Array[threadIdx.x*stride+C1]*C2
Endfor
```

(a) Base case: iteration variable has no impact on array access address

```
For i in {100, 200, 300, ... , 1000}
        Array[threadIdx.x *stride+i] =Array[threadIdx.x*stride+i+C1]*C2
Endfor
```

(b) Iteration variable has impacts on array access address by adding offset

Figure 8.2 Loops used to test conflict estimation tool

In the second example, the iteration variable impacts the memory access address by adding an offset which depends on the iteration variable. For example, for iteration variable i, the extra offset could be $a \times i + b$. For this case, since different in-layer offset could make conflict number different, the tool use the function *lcm()* to calculate the number of iterations each of which has distinct in-layer offset. Then the overall conflict number is calculated without enumerating all iterations.

The test result is shown in Figure 8.6, Figure 8.4, Figure 8.5, and Figure 8.6. The chart in Figure 8.3 is the original program execution time. The x-axis is the loop iteration number; the y-axis is execution time in *ms*, the execution time increases linearly to iteration number. Charts in Figure 8.4 are the performance of conflict estimation reference code, which calculates conflict number by calculating bank index and layer index of every single memory access. As expected, the time consumed is linear to iteration number, and it is up to 60*ms* when iteration number is 1000. Charts in Figure 8.5 are similar to charts in Figure 8.4 except that the proposed conflict analysis method is used to analysis each iteration. Compares to Figure 8.4, the execution time is obviously shorter. However, since it still goes through all iterations one by one, its execution time is linear to iteration number. Charts in Figure 8.6 are the performance of final solution used

in proposed tool. As it shows, the execution time is relatively constant when the iteration number increase. The reason is that the distinct case number is constant and it is up bounded by $bank\_num \times \frac{W}{M}$. The time consumed in the final proposed solution is less than $0.05ms$, which is much more efficient than two previous bank conflict estimation methods, and also comparable to execution time of the original program.

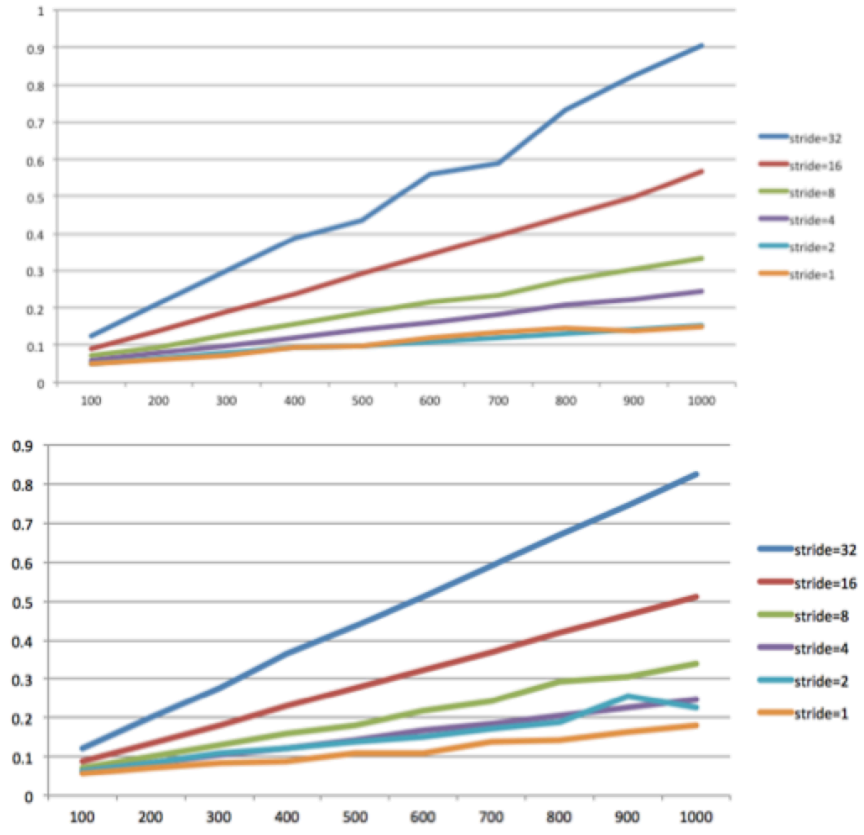


Figure 8.3 Original program execution time

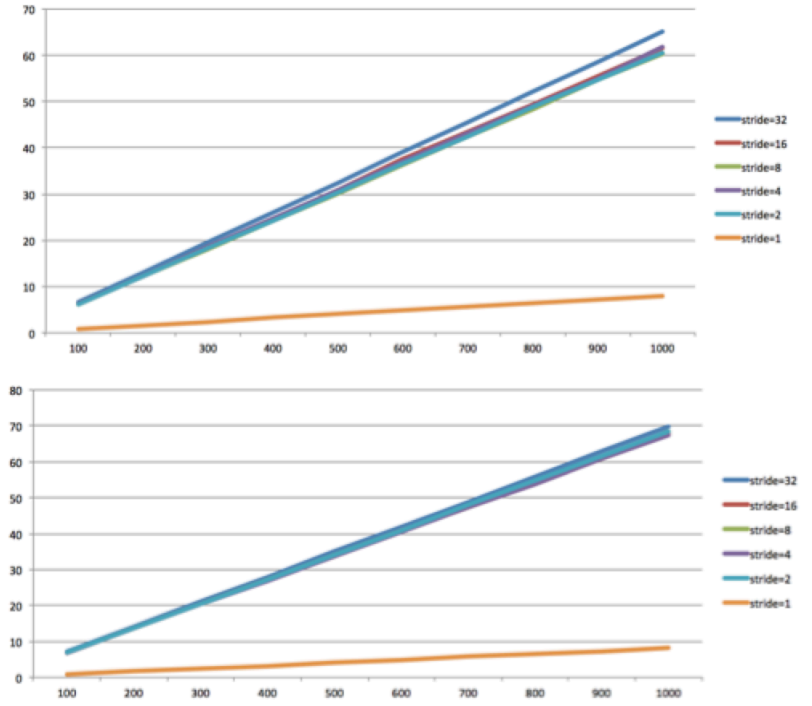


Figure 8.4 Basic analysis method: enumerate all access and compute conflict number

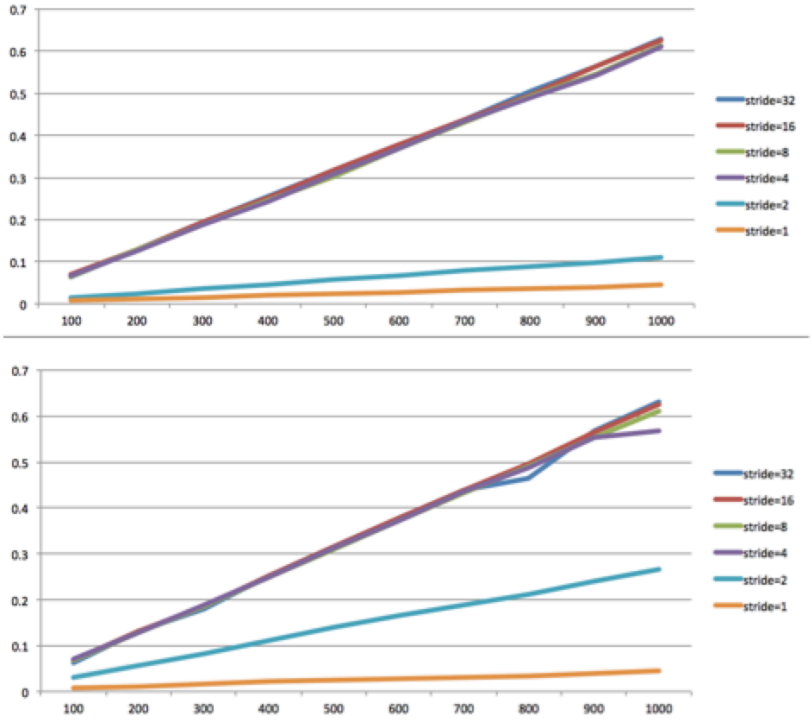


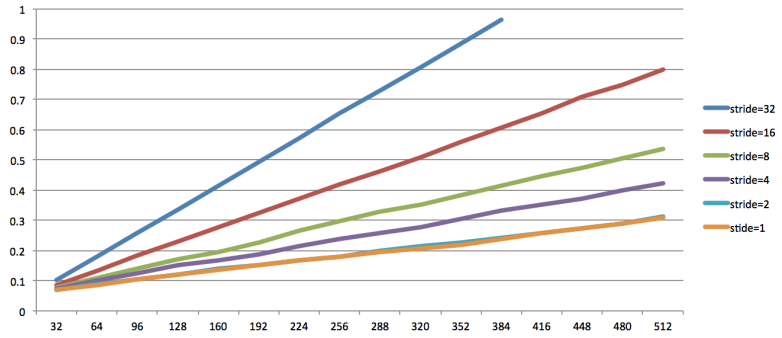
Figure 8.5 Analysis with no "for" loop optimization

Figure 8.6 Proposed conflict analysis tool execution time

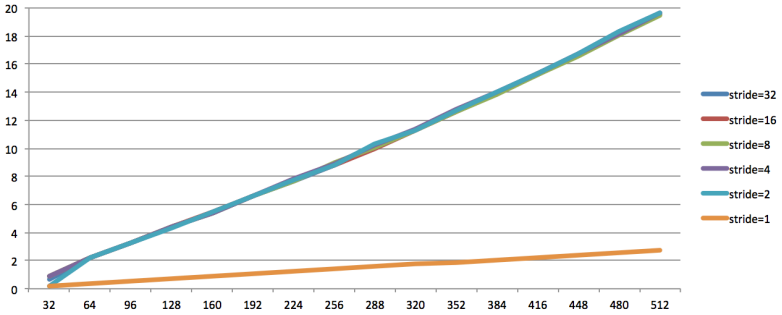
The third experiment is for memory accesses with "for-if" wrapper. The example code in use is shown in Figure 8.7. As introduced in chapter 5, for such cases, the memory access expression is warped by a "for" loop, and the iteration variable is used to filter the threads that are allowed to access the memory.

```
For i in {1, …, 256}
    If (threadIdx.x < i)
        Array[threadIdx.x *stride] =Array[threadIdx.x*stride+C1]*C2
Endfor
```

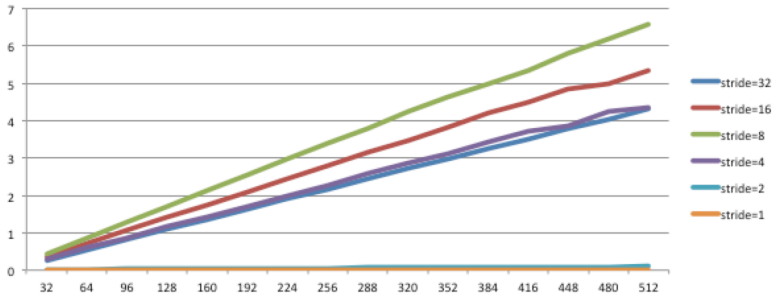
Figure 8.7 Loop used to test conflict estimation tool



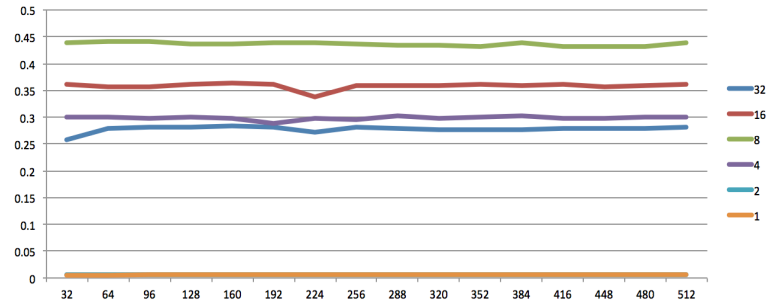
(a) Original program execution time.



(b) Basic analysis method: enumerate all access and check conflict



(c) Analysis with no “for-if” optimization



(d) Proposed conflict analysis tool

Figure 8.8 execution time comparison for “for-if” case

The executing time result is shown in Figure 8.8. The chart in (a) shows the original program execution time. The x-axis is the thread number; the y-axis is execution time in ms. According to the code in Figure 8.7, the workload ratio between $n-1$ and n is $\frac{(n-1)\times(n-2)}{n\times(n-1)} = \frac{n^2-n}{n^2-3n+2}$. It becomes "1" when n is large enough. Charts in (b) is the performance of the conflict estimation reference code which calculates conflict by calculating bank index and layer index of all single vector memory access. As expected, the time consumed is linear to iteration number, and it is up to 20ms when iteration number is 512. The chart in (c) is similar to the one in (b) except that the basic single expression conflict analysis module is used to analyze each iteration. Compares to (b), the execution time is obviously reduced. However, since it still goes through all iterations one after another, its execution time is linear to the iteration number. Chart in (d) is the performance of final solution used in proposed tool. As it shows, the execution time is relatively constant when the iteration number increases. The reason is that the distinct case number is constant and it is up bounded by $bank\_num \times \frac{W}{M}$. The time consumed in the final proposed solution is less than 0.5ms, which is much more efficient than two other methods.

8.2 Application Optimization

We select 6 applications (13 kernels) from RODINIA benchmark [73] and NVDIA CUDA SDK. These six benchmarks has bottleneck of shared memory bank conflict, and Figure 8.9 shows the instruction replay overhead caused bank conflict. These kernels can be optimized manually by changing bank access width, and array padding. The detail information of these kernels can be found in chapter 7. Table 8.1is a summary of their feature.

Table 8.1 Information of application CUDA kernels

Application Name	Array Name	Array Offset in Byte	Element Type Size	Array Size	Thread Block Dimension <x, y>	Instruction execution times for one thread block	Stride Dimension	If statement related to thread identification	Loop statement
3DFD	s_data	0	4B	24x24	16,16	8	2D	Y	N
							2D	N	N
							2D	N	N
NW	temp		4B	17x17	16, 1	1/2	1D	N	N
							1D	N	N
							1D	Y	Y
							1D	Y	Y
	ref		4B	16x16	16, 1	1/2	1D	Y	Y
							1D	Y	Y
TP-CSM	tile	0	4B	16x16	16,16	8	2D	N	N
TP-TC	tile	0	4B	16x16	16,16	8	2D	N	Y
TP-TNBC	tile	0	4B	16x17	16,16	8	2D	N	Y
Lud-DIA	shadow		4B	16x16	16, 1	1/2	1D	N	Y
							1D	Y	Y
							1D	Y	Y
							1D	Y	Y
Lud-PERI	dia	0	4B	16x16	32, 1	1	1D	Y	Y
							1D	Y	Y
	peri_row	1024	4B	16x16	32, 1	1	1D	Y	Y
	peri_col	2048	4B	16x16	32, 1	1	1D	Y	Y
CS-CRK	s_Data	0	4B	4x160	16, 4	2	2D	N	Y
CS-CCK	s_data	0	4B	16x81	16, 8	4	2D	N	Y
Shfl_scan	sums	0	4B	32x8	32, 8	8	2D	Y	N
							2D	Y	N
							2D	Y	N

* Note: TP-TCG and TC-TFG are similar to other TP kernels.

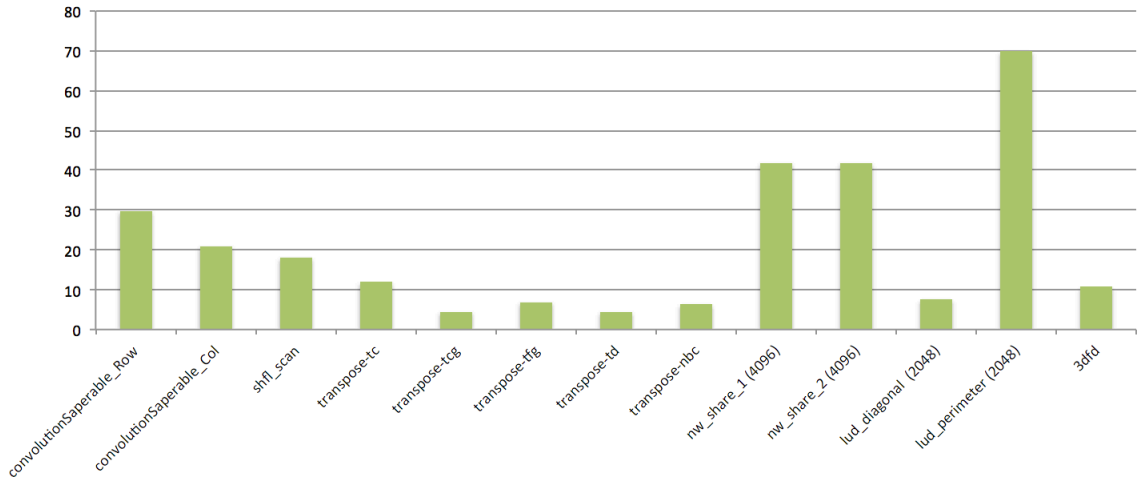
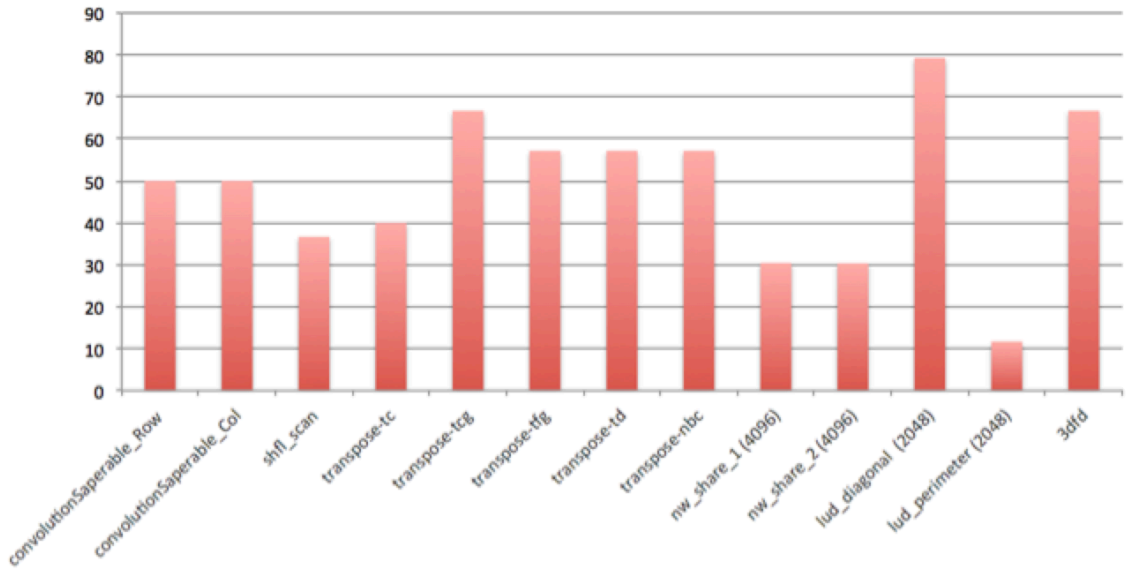
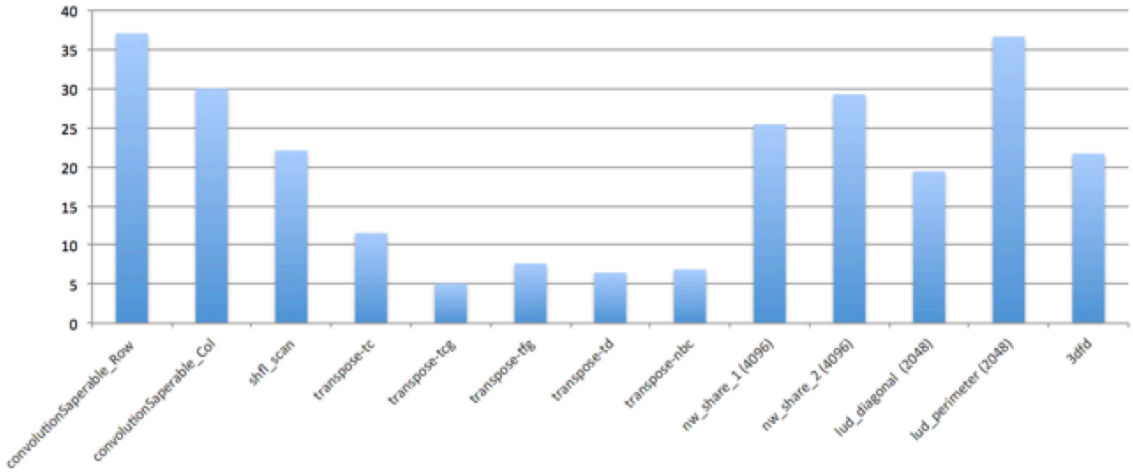


Figure 8.9 Percentage of bank access replay among total executed instructions

Figure 8.10 is the performance improvement of these kernels. The chart of Figure 8.10 (a) is the rate of bank access instruction number before and after optimization. The smaller the rate, the better the optimization effect on reducing access instruction replay. Figure 8.10 (b) is the speedup after optimization. Comparing Figure 8.9 and Figure 8.10 (b), for these 13 kernels, the kernels that have higher instruction percentage of replay get better improvement of execution time.



(a). Replay improvement: $rate = \frac{access\_operation\_after}{access\_operation\_before} \times 100\%$



(b). Execution time speedup

Figure 8.10 Performance experiment of 13 kernels

8.3 Summary

This chapter presents experiment results of this analysis tool. Section 8.1 includes experiments test the conflict analysis execution time. It shows that the proposed static analysis module is a practical solution in that its execution time is not related to warp number and for loop iteration number. Section 8.2 exams applications' performance improvement after accepts the solutions provided by proposed tool. As it shows, for applications that have bottleneck of shared memory bank conflict, this tool can help to improve efficiency by providing a solution which causes less or zero conflict number.

CHAPTER 9 CONCLUSION AND FUTURE WORK

In this dissertation, we explore how to improve GPU processing efficiency by reducing shared memory bank conflicts. We analyzed conflict patterns, then developed algorithms to perform inter and intra padding as well as configuring the shared memory bank bit width. Using this approach, we obtain an average 19% improvement for a set of benchmark applications.

The contributions of this work include analysis of shared memory bank conflicts, followed by techniques for selecting memory bank bit widths and applying inter and intra padding to optimize access patterns. This work can impact a broad spectrum of applications targeting GPUs.

We also developed the GPU Accelerated Scalable Parallel Random Number Generator (GASPRNG) library [74, 75] based on the previous SPRNG [76] and HASPRNG [77, 78] work.

For future work, the techniques from this dissertation could be integrated into a GPU compiler suite. Additionally, one could explore detailed modeling of GPU performance that includes the bank conflict analysis developed here [79-82].

LIST OF REFERENCE

1. NVIDIA, *OpenCL Programming Guide for CUDA Architecture 3.1*. 2011.
2. Nvidia, C., *Nvidia cuda c programming guide.* NVIDIA Corporation, 2011. **120**.
3. Cecilia, J.M., J.M. García, and M. Ujaldón, *Cuda 2D stencil computations for the Jacobi method*, in *Applied Parallel and Scientific Computing*. 2012, Springer. p. 173-183.
4. Harris, M., S. Sengupta, and J.D. Owens, *Parallel prefix sum (scan) with CUDA.* GPU gems, 2007. **3**(39): p. 851-876.
5. Ruetsch, G. and P. Micikevicius, *Optimize matrix transpose in CUDA*. 2009, Nvidia.
6. Flynn, M.J., *Computer Architecture Pipelined and Parallel Processor Design, 1995*. Jones and Bartlett Publishers, Inc. p. 360.
7. Fetterman, M., et al., *Dynamic bank mode addressing for memory access*. 2012, Google Patents.
8. Harper III, D.T. and D.A. Linebarger, *Conflict-free vector access using a dynamic storage scheme.* Computers, IEEE Transactions on, 1991. **40**(3): p. 276-283.
9. Russell, R.M., *The CRAY-1 computer system.* Communications of the ACM, 1978. **21**(1): p. 63-72.
10. Budnik, P. and D.J. Kuck, *The Organization and Use of Parallel Memories.* Computers, IEEE Transactions on, 1971. **C-20**(12): p. 1566-1569.
11. Park, J.W., *Multiaccess memory system for attached SIMD computer.* Computers, IEEE Transactions on, 2004. **53**(4): p. 439-452.
12. Oed, W. and O. Lange, *On the effective bandwidth of interleaved memories in vector processor systems.* Computers, IEEE Transactions on, 1985. **100**(10): p. 949-957.
13. Frailong, J.M. *XOR-Schemes: A flexible data organization in parallel memories*. in *1985 International Conference on Parallel Processing*. 1985.
14. Rau, B.R. *Pseudo-randomly interleaved memory*. in *ACM SIGARCH Computer Architecture News*. 1991. ACM.
15. Raghavan, R. and J.P. Hayes, *Reducing interference among vector accesses in interleaved memories.* Computers, IEEE Transactions on, 1993. **42**(4): p. 471-483.
16. Valero, M., et al., *Conflict-free access for streams in multimodule memories.* Computers, IEEE Transactions on, 1995. **44**(5): p. 634-646.
17. Ryoo, S., et al. *Optimization principles and application performance evaluation of a multithreaded GPU using CUDA*. in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. 2008. ACM.
18. Baskaran, M.M., et al. *A compiler framework for optimization of affine loop nests for GPGPUs*. in *Proceedings of the 22nd annual international conference on Supercomputing*. 2008. ACM.
19. Yang, Y., et al. *A GPGPU compiler for memory optimization and parallelism management*. in *ACM Sigplan Notices*. 2010. ACM.
20. Rivera, G. and C.-W. Tseng. *Data transformations for eliminating conflict misses*. in *ACM SIGPLAN Notices*. 1998. ACM.
21. Watanabe, T., *Architecture and performance of NEC supercomputer SX system.* Parallel Computing, 1987. **5**(1): p. 247-255.

22. Coorporation, I., *Intel 64 and IA-32 architectures optimization reference manual*. 2009, May.
23. Rau, B.R., M.S. Schlansker, and D.W. Yen. *The Cydram 5 Stride-Insensitive Memory System*. in *ICPP (1)*. 1989. Citeseer.
24. Kuzmanov, G., G. Gaydadjiev, and S. Vassiliadis, *Multimedia rectangularly addressable memory.* Multimedia, IEEE Transactions on, 2006. **8**(2): p. 315-322.
25. Vitkovski, A., G. Kuzmanov, and G. Gaydadjiev. *Memory organization with multi-pattern parallel accesses*. in *Proceedings of the conference on Design, automation and test in Europe*. 2008. ACM.
26. Shan, H. and E. Strohmaier. *Performance characteristics of the Cray X1 and their implications for application performance tuning*. in *Proceedings of the 18th annual international conference on Supercomputing*. 2004. ACM.
27. Micikevicius, P. *GPU Performance Analysis and Optimization*. in *GPU Technology Conference, http://developer. download. nvidia. com/GTC/PDF/GTC2012/PresentationPDF/S0514-GTC2012-GPU-Performance-Analysis. pdf*. 2012.
28. Harper, D. and Y. Costa, *Analytical estimation of vector access performance in parallel memory architectures.* Computers, IEEE Transactions on, 1993. **42**(5): p. 616-624.
29. Shapiro, H.D., *Theoretical limitations on the efficient use of parallel memories.* Computers, IEEE Transactions on, 1978. **100**(5): p. 421-428.
30. Sohi, G.S., *High-bandwidth interleaved memories for vector processors-a simulation study.* Computers, IEEE Transactions on, 1993. **42**(1): p. 34-44.
31. Harper, D.T., *Block, multistride vector, and FFT accesses in parallel memory systems.* Parallel and Distributed Systems, IEEE Transactions on, 1991. **2**(1): p. 43-51.
32. Chunyang, G., *Customizable Memory Schemes for Data Parallel Accelerators.* 2011.
33. Harper, D., *Increased memory performance during vector accesses through the use of linear address transformations.* Computers, IEEE Transactions on, 1992. **41**(2): p. 227-230.
34. Lawrie, D.H., *Access and alignment of data in an array processor.* Computers, IEEE Transactions on, 1975. **100**(12): p. 1145-1155.
35. Lawrie, D.H. and C.R. Vora, *The prime memory system for array access.* IEEE transactions on Computers, 1982. **31**(5): p. 435-442.
36. Wijshoff, H.A. and J. Van Leeuwen, *The structure of periodic storage schemes for parallel memories.* Computers, IEEE Transactions on, 1985. **100**(6): p. 501-505.
37. Harper, D.T. and J.R. Jump, *Vector access performance in parallel memories using a skewed storage scheme.* Computers, IEEE Transactions on, 1987. **100**(12): p. 1440-1449.
38. Aho, E., J. Vanne, and T. Hamalainen. *Parallel memory architecture for arbitrary stride accesses*. in *Design and Diagnostics of Electronic Circuits and systems, 2006 IEEE*. 2006. IEEE.

39. Trenas, M.A., et al. *A memory system supporting the efficient SIMD computation of the two dimensional DWT*. in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*. 1998. IEEE.
40. Kaufman, A. and D. Cohen-Or, *A 3D skewing and de-skewing scheme for conflict-free access to rays in volume rendering.* IEEE Transactions on Computers, 1995. **44**(5): p. 707-710.
41. Liu, C., X. Yan, and X. Qin. *An optimized linear skewing interleave scheme for on-chip multi-access memory systems*. in *Proceedings of the 17th ACM Great Lakes symposium on VLSI*. 2007. ACM.
42. Lentaris, G. and D. Reisis, *A graphics parallel memory organization exploiting request correlations.* Computers, IEEE Transactions on, 2010. **59**(6): p. 762-775.
43. Sung, I.-J., J.A. Stratton, and W.-M.W. Hwu. *Data layout transformation exploiting memory-level parallelism in structured grid many-core applications*. in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. 2010. ACM.
44. Pop, S., et al. *GRAPHITE: Polyhedral analyses and optimizations for GCC*. in *Proceedings of the 2006 GCC Developers Summit*. 2006. Citeseer.
45. Mowry, T.C., M.S. Lam, and A. Gupta. *Design and evaluation of a compiler algorithm for prefetching*. in *ACM Sigplan Notices*. 1992. ACM.
46. Vanderwiel, S.P. and D.J. Lilja, *Data prefetch mechanisms.* ACM Computing Surveys (CSUR), 2000. **32**(2): p. 174-199.
47. Wang, Y., et al. *Memory partitioning for multidimensional arrays in high-level synthesis*. in *Proceedings of the 50th Annual Design Automation Conference*. 2013. ACM.
48. Lin, H. and W. Wolf. *Co-design of interleaved memory systems*. in *Proceedings of the eighth international workshop on Hardware/software codesign*. 2000. ACM.
49. Zhang, Q., et al. *Reducing memory bank conflict for embedded multimedia systems*. in *Multimedia and Expo, 2004. ICME'04. 2004 IEEE International Conference on*. 2004. IEEE.
50. Jaeger, J. and D. Barthou. *Automatic efficient data layout for multithreaded stencil codes on CPU sand GPUs*. in *High Performance Computing (HiPC), 2012 19th International Conference on*. 2012. IEEE.
51. Weiß, C., et al. *Memory characteristics of iterative methods*. in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*. 1999. ACM.
52. Lee, V.W., et al. *Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU*. in *ACM SIGARCH Computer Architecture News*. 2010. ACM.
53. Che, S., J.W. Sheaffer, and K. Skadron. *Dymaxion: optimizing memory access patterns for heterogeneous systems*. in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011. ACM.
54. Jang, B., et al., *Exploiting memory access patterns to improve memory performance in data-parallel architectures.* Parallel and Distributed Systems, IEEE Transactions on, 2011. **22**(1): p. 105-118.

55. Bacon, D.F., S.L. Graham, and O.J. Sharp, *Compiler transformations for high-performance computing.* ACM Computing Surveys (CSUR), 1994. **26**(4): p. 345-420.
56. Kowarschik, M. and C. Weiß, *An overview of cache optimization techniques and cache-aware numerical algorithms*, in *Algorithms for Memory Hierarchies*. 2003, Springer. p. 213-232.
57. Ghosh, S., M. Martonosi, and S. Malik. *Precise miss analysis for program transformations with caches of arbitrary associativity*. in *ACM SIGPLAN Notices*. 1998. ACM.
58. Ghosh, S., M. Martonosi, and S. Malik, *Cache miss equations: a compiler framework for analyzing and tuning memory behavior.* ACM Transactions on Programming Languages and Systems (TOPLAS), 1999. **21**(4): p. 703-746.
59. Ishizaka, K., M. Obata, and H. Kasahara, *Cache optimization for coarse grain task parallel processing using inter-array padding*, in *Languages and Compilers for Parallel Computing*. 2004, Springer. p. 64-76.
60. Bacon, D.F., et al. *A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness*. in *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*. 1994. IBM Press.
61. Torrellas, J., M.S. Lam, and J.L. Hennessy. *Share Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates*. in *ICPP (2)*. 1990.
62. Bailey, D.H., *Unfavorable strides in cache memory systems (RNR Technical Report RNR-92-015).* Scientific Programming, 1995. **4**(2): p. 53-58.
63. Jeremiassen, T.E. and S.J. Eggers, *Reducing false sharing on shared memory multiprocessors through compile time data transformations*. Vol. 30. 1995: ACM.
64. Bolosky, W., R. Fitzgerald, and M. Scott. *Simple but effective techniques for NUMA memory management*. in *ACM SIGOPS Operating Systems Review*. 1989. ACM.
65. Rivera, G. and C.-W. Tseng. *Tiling optimizations for 3D scientific computations*. in *Supercomputing, ACM/IEEE 2000 Conference*. 2000. IEEE.
66. Hsu, C.-h. and U. Kremer, *A quantitative analysis of tile size selection algorithms.* The Journal of Supercomputing, 2004. **27**(3): p. 279-294.
67. Qasem, A., *Automatic tuning of scientific applications*. 2007, Rice University.
68. Rivera, G. and C.-W. Tseng. *A comparison of compiler tiling algorithms*. in *Compiler Construction*. 1999. Springer.
69. Fursin, G., M.F. O'Boyle, and P.M. Knijnenburg, *Evaluating iterative compilation*, in *Languages and Compilers for Parallel Computing*. 2005, Springer. p. 362-376.
70. Badawy, A.-H.A., et al. *Evaluating the impact of memory system performance on software prefetching and locality optimizations*. in *Proceedings of the 15th international conference on Supercomputing*. 2001. ACM.
71. Rivera, G. and C.-W. Tseng. *Eliminating conflict misses for high performance architectures*. in *Proceedings of the 12th international conference on Supercomputing*. 1998. ACM.
72. Cano, A., J.M. Luna, and S. Ventura, *High performance evaluation of evolutionary-mined association rules on GPUs.* The Journal of Supercomputing, 2013. **66**(3): p. 1438-1461.

73. Cano, A., A. Zafra, and S. Ventura, *Speeding up the evaluation phase of GP classification algorithms on GPUs.* Soft Computing, 2012. **16**(2): p. 187-202.
74. Gao, S. and G.D. Peterson, *GASPRNG: GPU accelerated scalable parallel random number generator library.* Computer Physics Communications, 2013. **184**(4): p. 1241-1249.
75. Gao, S. and G.D. Peterson. *GPU accelerated scalable parallel random number generators*. in *Proc. 2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC'10)*. 2010.
76. *Scalable Parallel Pseudo Random Number Generators Library*. Available from: http://sprng.fsu.edu
77. Lee, J., G.D. Peterson, and R.J. Harrison, *Hardware accelerated scalable parallel random number generators.* Proceedings of the 3rd Annual Reconfigurable Systems Summer Institute, 2007.
78. Lee, J., et al., *Implementation of hardware-accelerated scalable parallel random number generators.* VLSI Design, 2010. **2010**: p. 12.
79. Hong, S. and H. Kim. *An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness*. in *ACM SIGARCH Computer Architecture News*. 2009. ACM.
80. Smith, M.C. and G.D. Peterson, *Optimization of shared high-performance reconfigurable computing resources.* ACM Transactions on Embedded Computing Systems (TECS), 2012. **11**(2): p. 36.
81. Peterson, G.D. and R.D. Chamberlain, *Parallel application performance in a shared resource environment.* Distributed Systems Engineering, 1996. **3**(1): p. 9.
82. Peterson, G.D. and R.D. Chamberlain, *Beyond execution time: Expanding the use of performance models.* Parallel & Distributed Technology: Systems & Applications, IEEE, 1994. **2**(2): p. 37-49.

APPENDIX

Appendix A-1: 1D Single Warp Analysis for Column-major Bank Mapping

Figure A-1 is an example of column-major bank mapping when $R = 4$ and $stride = 11$. When stride is smaller than R, there is no potential bank conflict. The reason is that the bank scope of the vector access is less than the total bank number. So here we only consider the strides that is larger than R.

0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	.....
1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61	65	69	73	77	81	.....
2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62	66	70	74	78	82	.....
3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63	67	71	75	79	83	.....

Figure A-1: An example of column-major data mapping. The data are layout in column major direction. Each column is one layer of one bank; the whole grid is one layer of all banks. Blue blocks are elements accessed when stride=11

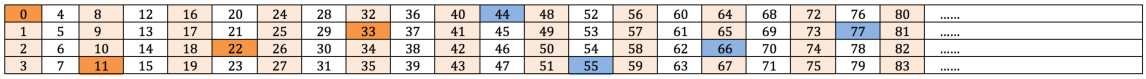
ODD STRIDE ANALYSIS

Firstly, we consider the visited sites in ith rows of all visited layers. When stride is odd, the visited elements are uniformly distributed in these rows. Each row has $\frac{vec\_length}{R}$ visited sites, and the distance of any two consecutive elements is as same as the stride. As the example in Figure A-1, the blue blocks are evenly distributed into four rows, and in each row the distance between two neighbor blue blocks are 11, which is the value of the stride.

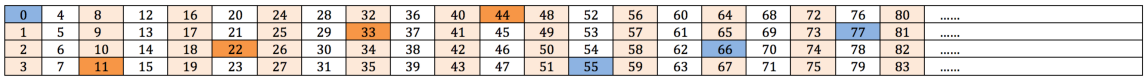
To analyze bank conflict, we study the distance between the visited sites that cause conflict. For the example in Figure A-1, figure A-2 shows the four cases with different offset. In Figure A-2 (a) the current vector visit starts from the first row; the first four visited elements are in $<0^{th}$ *row,* 0^{th} *col*$>$, $<3^{rd}$ *row,* 2^{nd} *col*$>$, $<2^{nd}$ *row,* 5^{th} *col*$>$ and $<1^{st}$ *row,* 8^{th} *col*$>$. Figure A-2 (b), (c), (d) have start points in 3rd row, 2nd row and 1st row. To locate first R visited elements, array *imm_col_offset* and array *row_offset* are calculated as following:

$$\begin{cases} imm\_col\_offset[i] = \left\lfloor \dfrac{row\_idx + stride \times i}{R} \right\rfloor \\ row\_offset[i] = \mathrm{mod}(row\_idx + stride \times i, R) \end{cases} \quad \text{(A.1.1)-a,b}$$

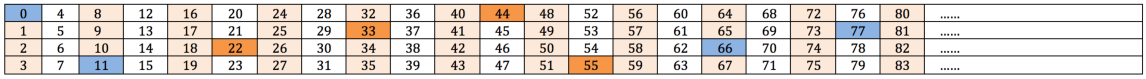
row_idx is the row index of the first visited elements. For example, in Figure A-2 (a), *row_idx=0*. Variable i denotes the i[th] visited elements and $0 \le i < R$ (the four orange elements in Figure A-2 (a)). Table A-1 lists the locations of the orange elements in Figure A-2 (a) to (d).



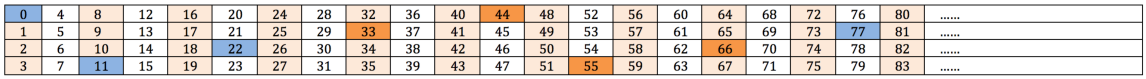
(a) The first element visited by current vector access lies in the 0[th] row. row_idx = 0



(b) The first element visited by current vector access lies in the 1[st] row. row_idx = 3



(c) The first element visited by current vector access lies in the 2nd row. row_idx = 2



(d) The first element visited by current vector access lies in the 3[rd] row. row_idx = 1

Figure A-2 a vector access could start from different rows.

Table A-1 Location of the visited elements shown in figure A-2. *stride=11. row_num=4*

row_idx of the 1st element	***1st elem (i=0) <imm_col_offset, row_offset>***	***2nd elem (i=1) <imm_col_offset, row_offset>***	***3rd elem (i=2) <imm_col_offset, row_offset>***	***4th elem (i=3) <imm_col_offset, row_offset>***
0	<0, 0>	<2, 3>	<5, 2>	<8, 1>
1	<0, 1>	<3, 0>	<5, 3>	<8, 2>
2	<0, 2>	<3, 1>	<6, 0>	<8, 3>
3	<0, 3>	<3, 2>	<6, 1>	<9, 0>

Given a certain value of *row_idx*, based on *elem_per_row*, *imm_col_offset* and *row_offset*, we can define *row_scale_num* as following:

$$row\_scale\_num[i] = \left\lceil \frac{imm\_col\_offset[i] + (elem\_per\_row - 1) \times stride + 1}{bank\_num} \right\rceil \quad \text{(A.1.2)}$$

In arrays of *row_scale_num*, *imm_col_offset* and *row_offset*, the ith element is the info of ith visited site. We use following functions to transform these arrays; the i[th] element becomes the info of the visited site in ith row. Then we set *row_offset[i] = i* for $0 \le i < R$.

$$\begin{cases} i' = row\_offset[i] \\ row\_scale\_num[i'] = row\_scale\_num[i] \\ imm\_col\_offset[i'] = imm\_col\_offset[i] \end{cases} \quad \text{(A.1.3)}$$

For odd strides, the visited sites in ith row of all layers cannot cause any conflict due to the reason that $\gcd(stride, bank\_num) = 1$ for odd strides. However, visited sites lie in different rows of different layers might cause conflict. For an interleaved memory that has R rows in each layer, there are $R^2 - R$ conflict possibilities between different rows (Table A-2). For the example in figure A-1, the potential conflict row index pair are listed in table A-2.

Table A-1-2 Possibilities of conflicts between rows from different layers

	Row 0	***Row 1***	***Row 2***	***Row 3***
Row 0	X	V	V	V
Row 1	V	X	V	V
Row 2	V	V	X	V
Row 3	V	V	V	X

When the x[th] element in row *i* of layer m conflict with the y[th] element in row *j* of layer *n*, if $i < j$ and $m \ge n$, we have:

$$\begin{cases} dist = (imm\_col\_offset[i] + stride \times x) - (imm\_col\_offset[j] + stride \times y) \\ dist = layer\_scale \times bank\_num \\ layer\_scale \times R \le (row\_scale\_num[i] - 1) \times R \end{cases}$$

(A.1.4) – a,b,c

with $layer\_scale = m - n$. *dist* is used to denote the difference between the offsets of conflict elements in its own row. It can be calculated through equation A.1.4 (a), and it also need to meet the requirement of equation A.1.4 (b). Table A-3 is an example used to show the meaning of dist and layer_scale. In this example, *bank_num=32*, *row_num=4*,

n=0, *m={0,1,2}*, *i=1*, and *j=3*. Green block is xth element in row *i* of layer *m*; red block is y[th] element in row *j* of layer *n*.

Table A-1-3 "*dist*" and "*layer_scale*" between conflict elements.

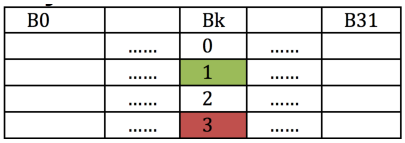
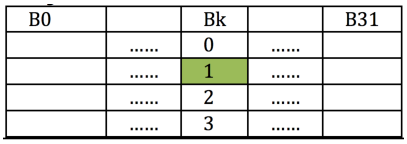
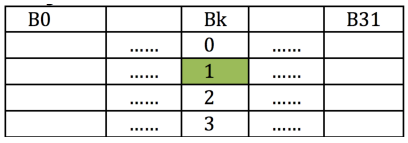
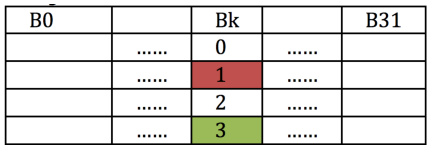
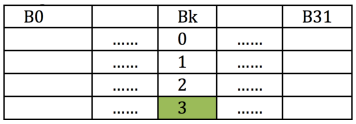
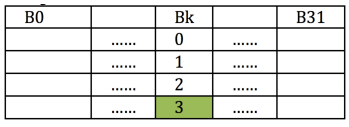
Layer	***Description***
Layer 0: 	In the same layer: the *dist* between red and green is: $dist = 0 \times 32$, *layer_scale=0*
Layer 1: 	In the 2nd layer: the *dist* between red and green is: $dist = 1 \times 32$, *layer_scale=1*
Layer 2: 	In the 3rd layer: the dist between red and green is: $dist = 2 \times 32$, *layer_scale=2*

Table A-1-4 "*dist*" and "*layer_scale*" between conflict elements.

Layer	***Description***
Layer 0: 	In the same layer: the *dist* between red and green is: $dist = 0 \times 32$, *layer_scale=0*
Layer 1: 	In the 2nd layer: the *dist* between red and green is: $dist = 1 \times 32$, *layer_scale=1*
Layer 2: 	In the 3rd layer: the *dist* between red and green is: $dist = 2 \times 32$, *layer_scale=2*

When the x[th] element in row *j* of layer *n* conflict with the y[th] element in row *i* of layer *m*, with $i < j$ and $m \le n$. The dist can be calculated in similar way. Table A-1-4 is an example when *bank_num=32*, *row_num=4*, *m=0*, *n={0, 1, 2}*, *i=1*, and *j=3*. Green block is x[th] element in row *j* of layer *n*; red block is y[th] element in row *i* of layer *m*.

$$\begin{cases} dist = (imm\_col\_offset[j] + stride \times y) - (imm\_col\_offset[i] + stride \times x) \\ dist = layer\_scale \times bank\_num \\ layer\_scale \times R \le (row\_scale\_num[j] - 1) \times R \end{cases}$$

(A.1.5)-a.b.c

The constrain of *x* and *y* is:

$$\left\{ (x,y) \middle| x > 0, x \le \frac{vec\_access\_length}{R}, y > 0, y \le \frac{vec\_access\_length}{R} \right\} \quad \text{(A.1.6)}$$

With equations in (A.1.4), (A.1.5), and (A.1.6), we can find the *(x, y)* pairs that cause conflicts. Particularly, when *W=N=8* and *M=4*, it has $R = 2$, and the following conclusion can be made: When $R = 2$, for odd strides, if there is at least one conflict, then the conflict degree is always 2. The proof can be found in appendix B.

Appendix A-2: 2D access bank conflict analysis

For 2D stride access, denote the base access address of current warp as offset, it has:

$$bank\_conflict\_dgr(offset) = bank\_conflict\_dgr(\mathrm{mod}(offset, bank\_number \times R))$$

This means that the conflict degree for $offset = \{0,1,.....MAX\_VALID\_OFFSET\}$ periodically repeat the conflict degree for $base\_set\{0,1,...,bank\_num \times R - 1\}$. We calculate the bank conflict of each offset in base_set. For other offset values not belong to base_set, we map it to an offset in base_set to get the bank conflict degree. Figure A-4 is an example with 2D stride is <*stride_x=1, repeat_x=2, stride_y=3, repeat_y=4*>

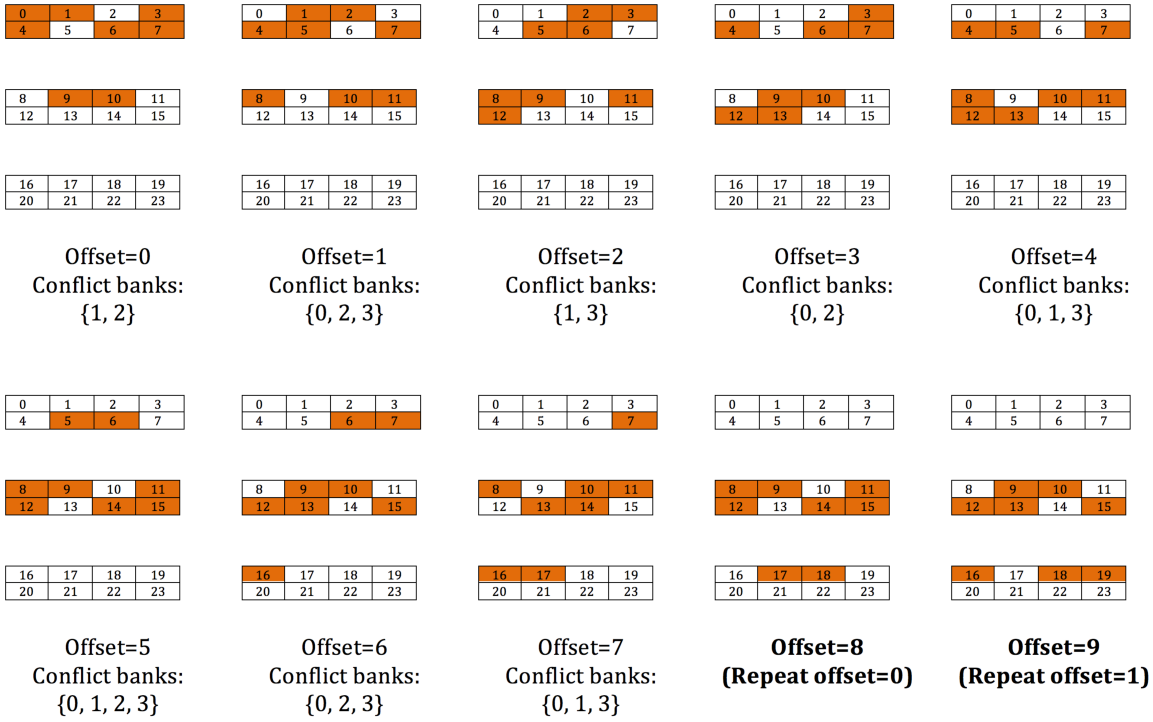


Figure A-4 a 2D stride example with different access offset.

Appendix A-3: Two-way Conflict for Column-major Bank Mapping with *R*=2

Observation: When $R = 2$, for odd strides that have at least one bank conflict, it is always 2-way conflict.

Proof:

For odd stride $stride = 2 \times l + 1$ with $l \geq 1$, the pair of visit sites that cause bank conflict must be from different rows: one from upper row and one from lower row. We describe it as $c =< ur, lr >$, ur is the offset of the visit site in upper row, and lr is the offset of the visit site in lower row.

When it has bank conflict, if there is only one conflict, then it is 2-way conflict since one conflict cannot visit more than 2 layers of same bank.

If there are more than one bank conflicts, for any two of them $c_1 =< ur_1, lr_1 >$ and $c_2 =< ur_2, lr_2 >$, we need to proof :

The distance of between them on the upper row is can always be divided by $stride$.

Reason: For any conflict $c_i =< ur_i, lr_i >$, its visited site in the upper row always has distance of $N \times stride$ ($N \in [1,..., \frac{vec\_length}{2} - 1]$) from other visited sites in the upper rows.

For a conflict c_i that has upper row offset ur_i, its neighbor pairs $c_{i-1} =< ur_{i-1}, lr_{i-1} >$ and $c_{i+1} =< ur_{i+1}, lr_{i+1} >$ must cause conflict as long as $ur_{i-1} = ur_i - stride$, $ur_{i+1} = ur_i + stride$, $lr_{i-1} = lr_i - stride$, $lr_{i-1} = lr_i + stride$ are inside the range of current parallel access.

Reason:

$\because \quad ur_i \equiv lr_i \mod(vec\_length)$

$\therefore \quad ur_i + stride \equiv lr_i + stride \mod(stride)$

$ur_i - stride \equiv lr_i - stride \mod(stride)$

Now we can conclude that when there is M ($M > 1$) conflicts, they can be described as $c_i =< ur_o + i \times stride, lr_o + i \times stride >$ with $i \in [0, M)$. $c_0 =< ur_0, lr_0 >$ is the first conflict which has the smallest value of ur. This means that the conflicts are mapped to banks in the 1D odd stride pattern, and total number of conflict is less or equal to $\frac{vec\_length}{2}$. So, there are no two conflicts that appear in same bank, and bank conflict degree is always 2.

VITA

Shuang GAO got her Degree of Bachelor in Biomedical and Engineering in 2001, and Degree of Master in Computer Science in 2005, both from Zhejiang University, Hangzhou, China. After graduated from Zhejiang University, she worked as a software engineer for 4 year. In 2009, she started her PhD study in University of Tennessee at Knoxville. After 5 years of study, she finished her research and graduated.