Optimizing CUDA Shared Memory Usage

Shuang Gao
EECS, University of Tennessee at Knoxville
Knoxville, USA
sgao3@utk.edu

Gregory D. Peterson
EECS, University of Tennessee at Knoxville
Knoxville, USA
gdp@utk.edu

***Abstract*— CUDA shared memory is fast, on-chip storage. However, the bank conflict issue could cause a performance bottleneck. Current NVIDIA Tesla GPUs support memory bank accesses with configurable bit-widths. While this feature provides an efficient bank mapping scheme for 32-bit and 64-bit data types, it becomes trickier to solve the bank conflict problem through manual code tuning. This paper presents a framework for automatic bank conflict analysis and optimization. Given static array access information, we calculate the conflict degree, and then provide optimized data access patterns. Basically, by searching among different combinations of inter- and intra-array padding, along with bank access bit-width configurations, we can efficiently reduce or eliminate bank conflicts. From RODINIA and the CUDA SDK we selected 13 kernels with bottlenecks due to shared memory bank conflicts. After using our approach, these benchmarks achieve 5%-35% improvement in runtime.**

Keywords— shared memory; CUDA; bank conflict

I. INTRODUCTION

CUDA shared memory is low-latency, on-chip storage. It is commonly used as cache to reduce memory access overhead, and as a shared space to enable efficient thread communication. Bank conflict is a primary issue when using shared memory. Normally programmers tune their code to reduce conflicts [1]. For earlier NVIDIA GPUs that have low-order interleaving banks, the ***GCD*** (Greatest Common Divisor) function can be used to calculate the conflict degree [2]. However *GCD* is no longer sufficient by itself since dynamic bit-width bank access was adopted in the NVIDIA Kepler GPU. This feature provides better support for 4-byte and 8-byte data types [3]. This work addresses reducing conflicts for configurable bank widths, addressing the broader optimization problem. Consider a 3DFD code as an example. A 2D array of 4-byte elements is defined using shared memory storage. By simply changing the bank access width to 8-bytes, without any padding the conflict can be eliminated. This paper introduces an approach using a heuristic optimization method to reduce or eliminate conflicts. The optimization solution is found with a combination of inter-padding, intra-padding, and bank access bit-width configuration.

II. CONFLICT ANALYSIS

As shown in figure 1, given an array ***A*** of 4-byte elements, by setting the bank access width as 4-bytes or 8-bytes, the array data are mapped horizontally or vertically inside one layer of all banks. In this paper, we call them ***row-major bank mapping*** and ***column-major bank mapping***. The proposed approach optimizes bank access efficiency by inter-padding, intra-padding, and bank bit-width configuration. Figure 2 describes how these three schemes impact the conflict degree. Figure 2 (a) shows the default row-major data mapping with 2-way conflict. (b), (c), and (d) transform data layout through different means to reduce conflicts. Since manually exploring the large potential solution search space is tedious and time consuming, our approach supports automated optimization.

A[0]	A[1]	A[2]	A[3]
A[4]	A[5]	A[6]	A[7]

A[0]	A[2]	A[4]	A[6]
A[1]	A[3]	A[5]	A[7]

Fig. 1. Array data bank mapping and bank access bit-width

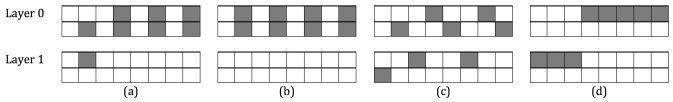

Fig. 2. Using inter-padding, intra-padding, and changing bank access bitwidth to eliminate bank conflicts: (a) original problem has 2-way conflict, (b) change access offset, (c) change access stride, and (d) use 8-byte bank access bit-width.

A. Single warp conflict analysis

Row-major bank mapping has no conflict when the stride is odd. For even strides, we divide them into two categories: (1) stride is power-of-two, and (2) other even valued strides.

When the stride is power-of-two, we use the *GCD* result and the warp access offset to calculate the conflict degree. The ***warp access offset*** is the offset of the first visited site from the beginning of a layer. Given *stride* = 2^s, *bank_num* = 2^b, and r is the number of rows per layer:

- When *stride* $\geq 2^{b+r}$, all visit sites lie in the same bank. The bank conflict degree is the warp size.
- When $2^b \leq$ *stride* $< 2^{b+r}$, all visit sites lie in the same bank, and the bank conflict degree is *ceil(GCD x* 2^{s-b}*/r)*.
- When *stride* $< 2^b$, the *bank_num can* be divided by stride, and the conflict degree is *ceil(GCD/r)*..

When *stride* is some other even number, we can write it as $\sigma \times 2^e$, where σ is an odd number larger than 3, and ***e*** is a natural number. The warp visited sites can be divided into 2^e groups, where each group occupies σ rows. For the i^{th} row of all groups, they visit the same banks. So there must be conflict if not all of them lie in the same layer. Inside each group, there is no conflict possibility. Based on this observation, the task becomes to check the conflict among the i^{th} rows of all groups.

For the column-major mapping, both even and odd strides could cause bank conflict. When *stride* is odd, the visited sites in the i[th] row of all layers have no conflict because *GCD*=1 for odd strides. Then we need to calculate the conflict caused by sites visited in different rows. Figure 3 is an example with *stride*=5, the conflict occurs between row1 of layer0 and row0 of layer1. For even strides, the problem can be transformed either to an odd stride problem, or to a conventional bank access problem which can be solved by *GCD*.

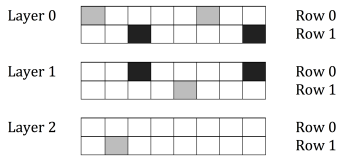

Fig. 3. An example of odd stride access for column-major bank mapping

B. *Single expression conflict analysis*

In CUDA, one shared memory access expression drives multiple warps in a single thread block to access a sequence of data elements in parallel. These warps share the same memory access pattern but have different offsets. The offset of the i[th] warp has the following relation with the offset of the first warp:

$$offset[i] = \mathrm{mod}(offset[0] + i \times C, L)$$

where ***C*** is the offset increment step size, and ***L*** is the number of 4-byte elements that can be saved in one layer. This means that the warp offsets are periodic: when $\mathrm{mod}(i \times C, L) = 0$, *offset[i]* equals to *offset[0]*. To get the overall bank conflict number, firstly, we calculate the number of distinct offsets and the conflict for each of them. Then we calculate the warp numbers that share the same offset. Finally, we compute the total conflict number.

III. Parameter Optimization Strategy

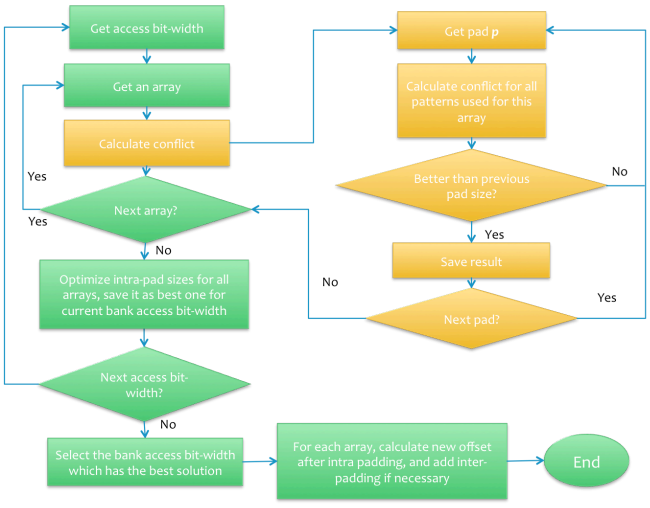

Fig. 4. parameter optimization strategy

Figure 4 illustrates the parameter optimization procedure. The outer most loop iterates over different bank access bit-widths. Inside the loop, for each array, an initial investigation is used to collect information, which will be used for inter-padding and intra-padding size optimization. Then a range of padding sizes are applied to this array, and the corresponding conflict number is calculated and stored in the intra-padding option list of this array. After obtaining the intra-padding option lists for all arrays, we use a greedy algorithm to obtain a solution that meets following requirements: 1) the total memory size used is within the maximum free memory size, and 2) for each array, this solution yields an optimized intra-padding solution. After we obtain the optimization solution for each bank bit-width configuration, we select the one that has the minimum conflict number and uses less memory. Finally, if conflict is not eliminated, we tune the inter-padding size by inserting dummy space between arrays. These steps have been implemented in a prototype tool to automate the optimization.

IV. Experiment Results

6 benchmarks (13 kernels) from RODINIA and the CUDA SDK are used to test the approach. The test platform includes a Telsa 20c with CUDA 5.0. The results in figure 2 show that this approach greatly reduces the bank access replay number and results in 5%-35% execution time improvement for these benchmarks.

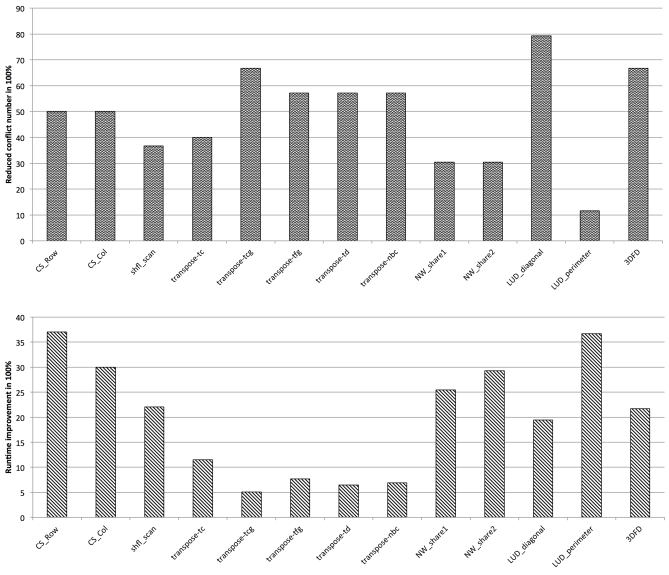

Fig. 5. Reduced bank access replay number in 100% and runtime speedup

References

[1] Nvidia, C., *Nvidia cuda c programming guide.*

[2] 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.

[3] Fetterman, M., et al., *Dynamic bank mode addressing for memory access*. 2012, Google Patents.