# BANK CONFLICTS IN SHARED MEMORY IN CUDA | SHARED MEMORY IN CUDA IN DETAIL | SHARED MEMORY AND BANK CONFLICT ION IN CUDA

POSTED BY NITIN GUPTA AT 05:00 | 9 COMMENTS

We have learned about shared memory in CUDA in my previous articles. We also saw shared memory in context of synchronization. In this article I'll let you know detail description about Shared memory in CUDA. Since shared memory is limited in terms of size so, some limitation came out.
This article specifically dedicated to shared memory bank conflict, if you don't know what shared memory is and how to use in CUDA, please follow this article.

Let's start our discussion on bank conflict in shared memory in CUDA.
CUDA is the parallel programming model to write general purpose parallel programs that will be executed on the GPU. **Bank conflicts** in GPUs are specific to shared memory and it is one of the many reasons to slow down the GPU kernel. **Bank conflicts arise because of some specific access pattern of data in shared memory**. It also depends on the hardware. For example, a bank conflict on a GPU device with compute capability 1.x may not be a bank conflict on a device with compute capability 2.x.

For better understanding about the bank conflict in shared memory, one should very clear about concept of Wrap. If you feel that you have little doubt over wrap, I refer you to go through this article.

***Shared memory Banks***

TAGS

CUDA ADVANCE

CUDA PROGRAMMING CONCEPT

OPTIMIZATION IN CUDA

RELATED POSTS

- CUDA complete | Complete reference on CUDA
- Multi GPU programming using texture memory in CUDA | CUDA multi-gpu textures
- CUDA Array in CUDA | How to use CUDA Array in CUDA | CUDA ARRAY
- cudaChannelFormatDesc () in CUDA | How to use cudaChannelFormatDesc in CUDA | CUDA Channels
- Texture Memory in CUDA | What is Texture Memory in CUDA programming

SHARE THIS

RECEN POPUL RANDOM

TEXTURE OBJECT IN CUDA | Bindless Texture in CUDA
01/04/2013 - 0 comments

How to specify architecture while compiling CUDA program in Visual profiler
How to change command line flag in CUDA
24/03/2013 - 0 comments

Using Shared Memory in CUDA C/C++
11/03/2013 - 0 comments

Optimization in histogram CUDA code: When number of Bins not equal to max threads in a block
08/03/2013 - 9 comments

GOOGLE+ FOLLOWERS

Nitin Gupta

Add to circles

8 have me in circles

View all

## Background

Since fast shared memory access is restricted to threads in a block. The shared memory is divided into multiple banks (similar to banks in DRAM modules). Each bank can service only one request at a time. The shared memory is therefore interleaved to increase the throughput. If the shared memory is interleaved by 32 bits, then the bandwidth of each bank is 32 bits or one float data type. The total number of banks is fixed**. It is 16 on older GPUs (with** compute capability **1.x ) and 32 on modern GPUs (with** compute capability **2.x).**

Because it is on-chip, shared memory is much faster than local and global memory. **Shared memory latency is roughly 100x lower than global memory latency.**

## Memory bank

To achieve high memory bandwidth for concurrent accesses, shared memory is divided into **equally sized memory modules, called banks that can be accessed simultaneously**. Therefore, any memory load or store of **n** addresses that spans **n** distinct memory banks can be serviced simultaneously, yielding an effective bandwidth that is **n** times as high as the bandwidth of a single bank.

## What's wrong? Bank Conflict

If multiple addresses of a memory request map to the same memory bank, the accesses are serialized. The hardware splits a memory request that has bank conflicts into as many separate conflict-free requests as necessary**, decreasing the effective bandwidth by a factor equal to the number of separate memory requests.**

## Organization of Shared memory Banks in CUDA

Shared memory banks are organized such that successive 32-bit words are assigned to successive banks and each bank has a bandwidth of 32 bits per clock cycle. **The bandwidth of shared memory is 32 bits per bank per clock cycle**. For devices of compute capability 1.x, the warp size is 32 threads and the **number of banks is 16**.

## How the request to shared memory works in a Wrap

A shared memory request for a warp is split into one request for the first half of the warp and one request for the second half of the warp. Note that no bank conflict occurs if only one memory location per bank is accessed by a half warp of threads.

Shared memory enables cooperation between threads in a block. When multiple threads in a block use the same data from global memory, shared memory can be used to access the data from global memory only once. Shared memory can also be used to avoid **un-coalesced memory** (discussed below) accesses by loading and storing data in a **coalesced pattern** (discussed below) from global memory and then reordering it in shared memory**. Aside from memory bank conflicts, there is no penalty for non-sequential or unaligned accesses by a half warp in shared memory.**

For better understanding we should know about what is Coalesced memory in CUDA, that I have discussed below.

## Coalesced memory in CUDA

A coalesced memory transaction is one in which all of the threads in a half-warp access global memory at the same time. This is over simple, but the correct way to do it is just have consecutive threads access consecutive memory addresses.

So, if threads 0, 1, 2, and 3 read global memory 0x0, 0x4, 0x8, and 0xc, it should be a coalesced read

**Example**

Let's say we have a matrix which isreside linearly in memory. You can do this however you want, and your memory access should reflect how your matrix is laid out. So, the 3x4 matrix below

```
0 1 2 3
4 5 6 7
8 9 a b
```

Could be done row after row, like this, so that (r,c) maps to memory (r*4 + c)

```
0 1 2 3 4 5 6 7 8 9 a b
```

Suppose you need to access element once, and say you have four threads. Which threads will be used for which element? Probably either

```
thread 0:  0, 1, 2
thread 1:  3, 4, 5
thread 2:  6, 7, 8
thread 3:  9, a, b
or

thread 0:  0, 4, 8
thread 1:  1, 5, 9
thread 2:  2, 6, a
thread 3:  3, 7, b
```

**Which is better? Which will result in coalesced reads, and which will not?**

Either way, each thread makes three accesses. Let's look at the first access and see if the threads access memory consecutively. In the first option, the first access is 0, 3, 6, 9. Not consecutive, not coalesced. The second option, it's 0, 1, 2, 3. Consecutive! Coalesced! Yay!

The best way is probably to write your kernel and then profile it to see if you have non-coalesced global loads and stores.

Bonus: The number of coalesced and un-coalesced memory transactions in GPU

**I'll describe in more detail in my future article.**
**Now back to the picture.**
So we have learned that a bank conflict arises if any of the threads in a half warp access different words in the same bank. When a bank conflict happens the access to the data is serialized.

## So, when bank conflict happens or when not, how we recognize that?
## Let's go though some example to understand.

## *No Bank Conflict*

### Liner addressing

Let say, I have a **shared memory of 32 floats** inside the kernel which look like this.

```
__shared__ float shared[32];
```

We have declared an array of shared memory of float type. Now we access data though shared memory such as,

```
float data = shared[ BaseIndex + S*tid];
```

where "S" is known as Stribe

Memory accessing pattern is show in below figure, with S = 1 and S = 3
Fig 1

CUDA Array in CUDA | How to use CUDA Array in CUDA...

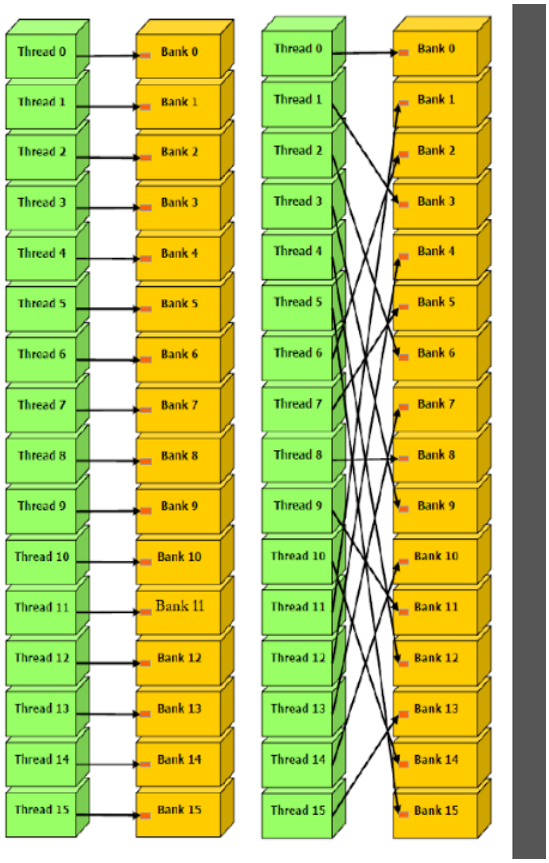
cudaChannelFormatDesc () in CUDA | How to use cuda...

Texture Memory in CUDA | What is Texture Memory in...

► January (41)

► 2012 (15)

@cudaprogramming. Powered by Blogger.




Clearly in memory access pattern, it is clearer that each thread in a warp access a different bank like **thread 0 access bank 0 or thread 15 access bank 15 and so on, on the other fig. thread 0 access bank 0, thread 1 access bank 3 and so on.** So, clearly each request made concurrently (In parallels). **SO, No Bank Conflict.** But why there is no bank conflict? Think!!!

**this is only bank-conflict-free if S shares no common factors with the number of ad banks.**

**Example: 16 banks on G80, so S must be odd**

**Example Scenario**

Let's say we have an array of size 256 of integer type in global memory and we have 256 threads in a single Block, and we want to copy the array to shared memory. Therefore every thread copies one element.

```
shared_a[threadIdx.x] = global_a[threadIdx.x];
```

So, what u think, does it trap into bank conflict? (Before reading answer, think first)

Ok Ok!!

First let's assume your arrays are say for example of the type int (*a 32-bit word*). Your code saves these ints into shared memory, across **any half warp the Kth thread is saving to the Kth memory bank.** So for example thread 0 of the first half warp will save to shared_a[0] which is in the first memory bank, thread 1 will save to shared_a[1], each half warp has 16 threads these map to the 16 4byte banks. In the next half warp, the first thread will now save its value into shared_a[16] which is in the first memory bank again. So if you use a 4byte word such int, float etc, then this example will not result in a bank conflict.

**Be Cautions**

If you use a 1 byte word such as char, in the first half warp threads

0, 1, 2 and 3 will all save their values to the first bank of shared memory which will cause a bank conflict known **as 4-way bank conflict** (we'll discussed later) .

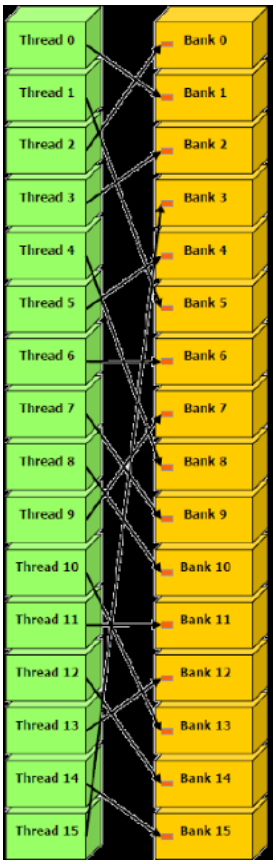
Question time!!! So, tell me is there any bank conflict in the following statement?

```
foo = shared[baseIndex + threadIdx.x]
```

A> If data is type of 32-bits; 4 Byte?
B> If data is type of 8-bits; 1 Byte?
You can answer these questions as comment to this post. ☺

## Random addressing

In Random addressing itself there is no bank conflict. Shown in below figure,
Fig 2




## *Bank Conflicts*

### *2 way bank conflict*

Let say, I have a **shared memory of 32 doubles/shorts** inside the kernel which look like this.

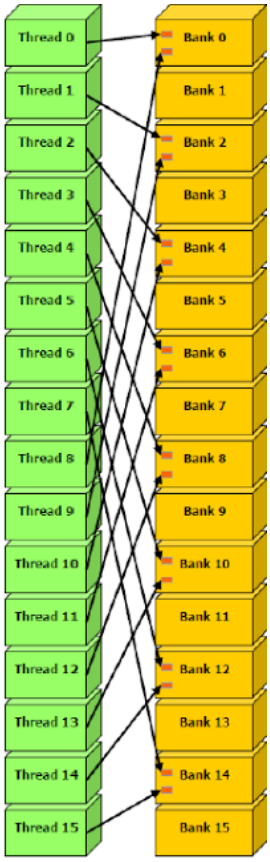
```
__shared__double shared[32];
Or
__shared__short shared[32];
```

We have declared an array of shared memory of double/short type. Now we access data though shared memory such as,

```
double data = shared[ BaseIndex + S*tid];
or
short data = shared[ BaseIndex + S*tid];
```

where "S" is known as Stribe.

Memory accessing pattern is show in below figure, with S = 1
Fig 3




Clearly in memory access pattern, it is clearer that two threads in a warp access a same bank memory location like **thread 0 access bank 0 or thread 8 access bank 0 and so on.** So, clearly these requests become serialize, Known as 2-Way bank conflict.

For example, refer last example the only changes is instead of int's make them double/short.

***4 way bank conflict***

Let say, I have a **shared memory of 32 char** inside the kernel which look like this.

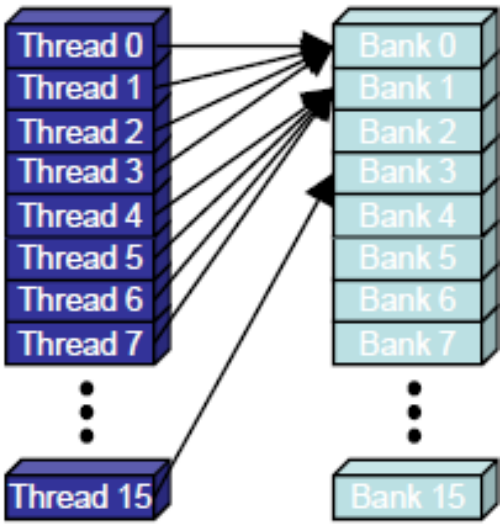
```
__shared__double shared[32];
Or
__shared__short shared[32];
```

We have declared an array of shared memory of char type. Now we access data though shared memory such as,

```
double data = shared[ BaseIndex + S*tid];
or
short data = shared[ BaseIndex + S*tid];
```

where "S" is known as Stribe.

Memory accessing pattern is show in below figure, with S = 1
Fig 4




Clearly in memory access pattern, it is clearer that four threads in a warp access a same bank memory location like **thread 0,1,2,3 access bank 0 and so on.** So, clearly these requests become serialize, Known as 4-Way bank conflict.
For example, refer last example the only changes is instead of int's make them char.

***Structure and Bank Conflict | Bank Conflict in Structure in CUDA***

Now let us discuss bank conflict in structures. In structures let say of floats or chars or ints or doubles or even combination of these, we need to take special care about it. So, let us first understand how the bank conflict arises in structure itself.
I'll explain it by examples !!!

Let us say we have;

```
struct vector { float x, y, z; };
__shared__ struct vector vectors[64];
```

```
struct myType2 {float f; int c; };
__shared__ struct myType2 myTypes2[64];
```

```
struct myType8 {float x,y,z,w; };
__shared__ struct myType8 myTypes8[64];
```

So, **according to you which have bank conflict and which don't have?**

Ok Ok!!

**Example 1:** This has no bank conflicts for vector; **struct size is 3 words**
**3 accesses per thread, contiguous banks (no common factor with 16)**

```
struct vector v = vectors [ baseIndex + threadIdx.x];
```

**Example 2:** This has **2 way bank** conflicts for myType2;
(2 accesses per thread) **(Have common factor with 16; 2 )**

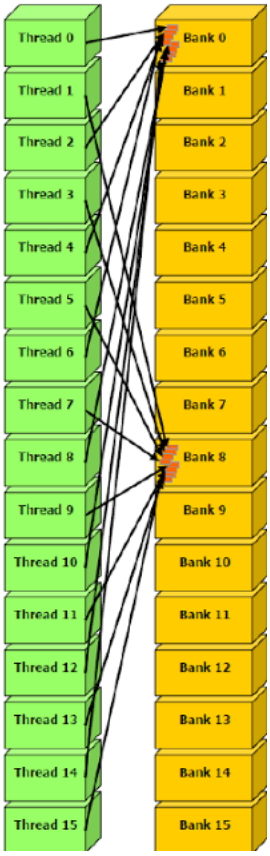
```
struct myType2 m = myTypes4[baseIndex +
```

```
threadIdx.x];
```

**Example 3:** This has **8 way bank** conflicts for myType8;
(8 accesses per thread) **(Have common factor with 16; 8 )**

```
struct myType8 m = myTypes8[baseIndex +
threadIdx.x];
```

Fig 5




Are you still having confusion? if Yes, you may not catch the Eagle, let me do for you, the all movie is end up with this statement;
**This is only bank-conflict-free if S shares no common factors with the number of ad banks.**
**Example: 16 banks on G80, so S must be odd, if S is even then there is a bank conflict.**

## How to avoid bank conflict?

•The old fashion method: (don't use it)

```
__shared__ intshared_lo[32];
__shared__ intshared_hi[32];

double dataIn;
shared_lo[BaseIndex+tid]= __double2loint(dataIn);

shared_hi[BaseIndex+tid]= __double2hiint(dataIn);

double dataOut =__hiloint2double(
shared_hi[BaseIndex+tid], shared_lo[BaseIndex+tid] );
```

•For array of structures, bank conflict can be reduced by changing it to structure of array. "Memory padding"(we'll discuss later)

Now time for exercise and some real application examples.

## Common Array Bank Conflict Patterns 1D

Start with this, each thread loads 2 elements into shared memory:

```
int tid = threadIdx.x;
shared[2*tid] = global[2*tid];
shared[2*tid + 1 ] = global[2*tid + 1];
```

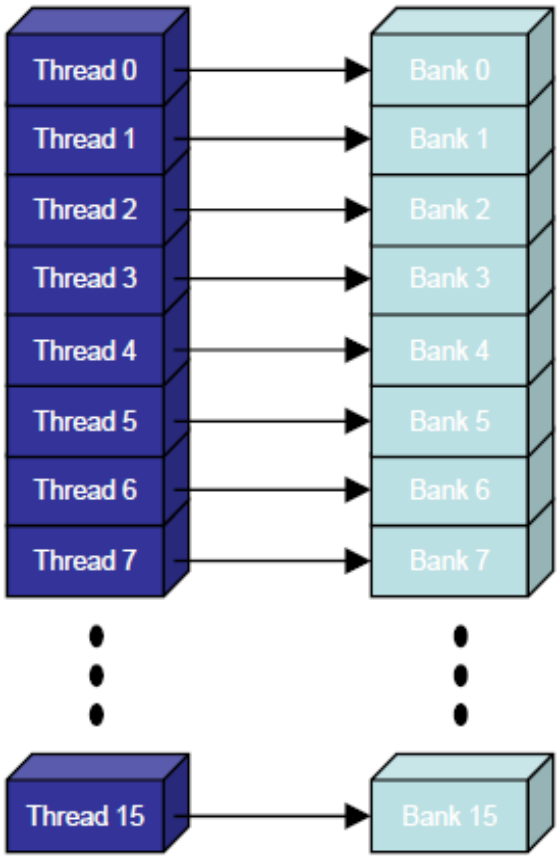
Does it have bank conflict?
Did you say yes and 2-Way bank conflict? OH!!, yes it has 2-way bank conflict. Since, 2-way-interleaved loads result in 2-way bank conflicts. Great Job!!!

**Note:**
This makes sense for traditional CPU thread, locality in cache line usage and reduce sharing traffic but not in **shared memory usage where there is no cache line effects but banking effects. ;)**

**So, how you'll solve this problem ? do you have any solution for this?**
**I have!!**
A Better Array Access Pattern: Each thread loads one element in every consecutive group of blockDim elements, like this;

```
shared[tid] = global[tid];
shared[tid + blockDim.x] = global[tid + blockDim.x];
```

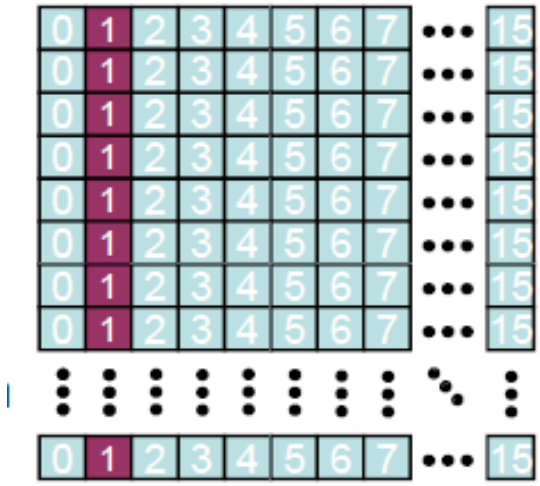
**Fig 6**




## Common Bank Conflict Patterns (2D)

Suppose you are operating on 2D array of floats in shared memory like when we do image processing. Let say our block

size is 16x16 (don't bother about grid size). So our scenario is that each thread processes a row, so threads in a block access the element in each column simultaneously(example: row 1 with in purple color fig. 7)
Fig 7



So, we suffer here with 16-way bank conflict, since all row start with 0. ☹

## Solution

Fortunately, we have solutions.

1> Memory padding; Add one float to end of each row.
2> Transpose the matrix before processing,
3> Change the address pattern.

Unfortunately, second one may have bank conflict, so for this article I'll concern about only memory padding and in the future article I'll let you know the solution with second option. For time being concentrate on first solution.

Memory padding like as shown in fig.

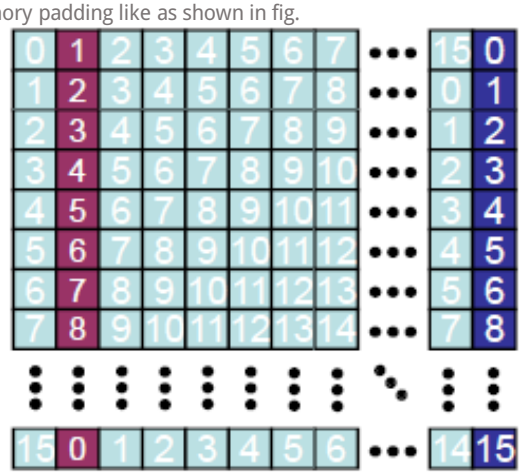


Fig 8

Solutions is very simple with Memory padding. Instead of creating shared memory as

```
__shared__ int shared[TILE_WIDHT][TILE_HEIGHT] ;
```

Create it as

```
__shared__ int shared[TILE_WIDHT][TILE_HEIGHT + 1 ] ;
```

And fill the unused space with 0's. Since C/C++ is row major, the leading dimension is the row-width (number of elements in a row). ☺

**I'll describe you memory padding with pitch that is the correct way to pad memory, in future article.** [since it is itself a long article ☺]

I hope you must like this article and have learned Shared memory *bank conflict* and how to avoid them in CUDA.

Got Questions?
Feel free to ask me any question because I'd be happy to walk you through step by step!
Want to Contact us? Click here

9 comments:

**Anonymous** 14 February 2013 11:46

Can you please explain on how to deal with bank conflicts for the case where we deal 2d array with a 2d thread block. eg: 16x16 threads working on 2d 16x16 data in shared memory

Reply

**Nitin Gupta** 16 February 2013 10:23

Author

Hello
Well, I have already described bank conflict in 2D array under "Common Bank Conflict Patterns (2D)" ;
here is bit more explanation but before explaining it i want to clear you there is no effect made by 2D blocks on bank conflict :) ;
Let say (as your eg.) we have 16x16 threads working in a block;
for example ;
__shared__ int Shared [16][16] ;

now consider this scenario
__shared__ int shared[16][16];

```
// map from threadIdx/BlockIdx to data position
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;

// calculate the global id into the one dimensional array
int tid = x + y * numCols;

// load shared memory
shData[threadIdx.x][threadIdx.y] = gIn[gid];
__syncthreads();

// write data back to global memory
gOut[gid] = shData[threadIdx.x][threadIdx.y];
```

now you can clearly understand here bank conflict occur since we are reading and writing data from/into shared memory in column major since, C (CUDA) organize 2D array in Row major order;
like this
bank0 .... bank15
row 0 [ 0 .... 15 ]
1 [ 16 .... 31 ]
2 [ 32 .... 47 ]
3 [ 4 .... 63 ]
4 [ 64 .... 79 ]
5 [ 80 .... 95 ]
6 [ 96 .... 111 ]
7 [ 112 .... 127 ]
8 [ 128 .... 143 ]
9 [ 144 .... 159 ]
10 [ 160 .... 175 ]
11 [ 176 .... 191 ]
12 [ 192 .... 207 ]
13 [ 208 .... 223 ]

14 [ 224 .... 239 ]
15 [ 240 .... 255 ]
col 0 .... col 15

So here is 16 way bank conflict, since each thread access bank of other warp; (see in fig. above)
Solution: read in row major order; jst change shData[threadIdx.y][threadIdx.x]; now each thread access in row major order and each thread access single 32bit bank of its warp .

Reply

**Nitin Gupta** 16 February 2013 10:26

I'll describe bank conflict and its solution in subsequent post/tutorial in 2D in Deep

Reply

**Anonymous** 19 February 2013 08:20

thank you,

Reply

Replies

**Nitin Gupta** 20 February 2013 02:11

Most Welcome; Keep visiting us

**Gopal** 28 February 2013 06:58

Hi,
Thanks for sharing this !!

Assume I am using a card which has 32KB of shared memory per SM. And the word size of my program is 64 bytes. It means I can have approximate maximum (32*1024*8 / 64*8 = 512) threads per SM. Hence I am transferring 512 consecutive words to each SM.

If each thread access a word of 64 bytes,

Will it cause bank conflicts, how ?
Is it beneficial to use shared memory in this scenario or not?

Reply

Replies

**Nitin Gupta** 1 March 2013 04:16

Hello Gopal, you are welcome.
since you are accessing per thread a word of 64 bytes and the shared memory is divided into 4 bytes consecutive section either 16 or 32 banks depend of architecture. your one 64 bytes word divided into 16 banks, right ?
it implies that you are accessing 16 bank per thread (4*16 = 64)
so let say you have thread 0, 1 , 2 .... in warp 0, then request is divided in two parts 1 for each half warp (16:16)

thread 0 accessing bank 0-15
thread 1 accessing bank 16-31
thread 2 accessing bank 0-15 (on next banks in memory ) cause no bank conflict !!! and so on
thread 15 accessing bank 16-31 and so on ...

Now if my calculation is not wrong up to far, then we can conclude that we have no bank conflicts in this situations.
Golden Lines always remember
1. "bank conflict happens where in a half warp, any two or more thread access same bank in that memory bank section, not applicable for 16 treads access same bank,called Broadcast".
2. "This is only bank-conflict-free if S(tride) shares no common factors with the number of ad banks."

this is my perception; you can cross check that does in this situation we have bank conflict or not by implementing above idea and use visual profiler for profiling your code, "check warp serialization counter for that" if you need help for that, you can comment back to me.
thanks

**Nitin Gupta** 1 March 2013 04:16

That was the good question @Gopal

Reply

**Anonymous** 17 June 2013 09:51

Hi,

I wrote a CUDA kernel with shared memory in double precision. The code looks doing something like this:

...
double val = shmem[threadIdx.x + offset]; // For all threads in a thread block of one dimension, offset = 8
...

I do observe bank conflict from the part. However, if I configure the size of each bank element to be 8 bytes, it went away:

cudaDeviceSetSharedMemConfig(cudaSharedMemBankSizeEightByte);

Can you explain what happens with/without the configuration? Thank you so much.

Reply

Enter your comment...

Comment as: Google Account

Publish Preview

Help us to improve our quality and become contributor to our blog

Links to this post

Create a Link

Newer Post Home Older Post

Become a contributor to this blog. Click on contact us tab

## LABELS

- Books on CUDA (9)
- C program (2)
- Compilation (3)
- CUDA Advance (25)
- CUDA Basics (31)
- CUDA Function (1)
- CUDA Programming Concept (41)
- CUDA programs Level 1.1 (10)
- CUDA programs Level 1.2 (4)

## LIKE US

## CLOUD

## ADMIN

Nitin Gupta

- CUDA programs Level 2.1 (3)
- Debugging (2)
- Images Processing (6)
- Installation (2)
- Kepler Features (1)
- Matlab Coding (3)
- Optimization in CUDA (17)


08:30:15 am

facebook

Name:
Nitin Gupta
Email:
nitinguptaiit@gmail.com
Status:
Very impressive, Team India!! The Champion of...