SHARED MEMORY BANK CONFLICTS SAMPLE

v2023.1.0 | May 2023

TABLE OF CONTENTS

Chapter 1. INTRODUCTION

This sample profiles a CUDA kernel which transposes an N x N square matrix of float elements in global memory using the Nsight Compute profiler. To avoid uncoalesced global memory accesses this kernel reads the data into shared memory. The profiler is used to analyze and identify the shared memory bank conflicts which result in inefficient shared memory accesses.

Shared memory accesses on a GPU

Shared memory is located on-chip, so it has much higher bandwidth and much lower latency than either local or global memory. Shared memory can be shared across a compute Cooperative Thread Array (CTA). In CUDA, CTAs are referred to as Thread Blocks. Compute CTAs attempting to share data across threads via shared memory must use synchronization operations (such as **`__syncthreads()`**) between stores and loads to ensure data written by any one thread is visible to other threads in the CTA.

Shared memory has 32 banks that are organized such that successive 32-bit words map to successive banks that can be accessed simultaneously. Any 32-bit memory read or write request made of 32 addresses that fall in 32 distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is 32 times as high as the bandwidth of a single request. However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The exception to this rule is when all threads read the same shared memory address, which results in a broadcast where the data at that address is sent to all threads in one transaction.

To get maximum performance, it is therefore important to understand how memory addresses map to memory banks in order to schedule the memory requests so as to minimize bank conflicts.

Chapter 2. APPLICATION

The sample CUDA application transposes a matrix of floats. The input and output matrices are at separate memory locations. For simplicity it only handles square matrices whose dimensions are integral multiples of 32, the tile size.

The sharedBankConflicts sample is available with Nsight Compute under <nsight-compute-install-directory>/extras/samples/sharedBankConflicts.

Chapter 3. CONFIGURATION

The profiling results included in this document were collected on the following configuration:

- Target system: Linux (x86_64) with a NVIDIA RTX A2000 (Ampere GA106) GPU
- Nsight Compute version: 2023.2.0

The Nsight Compute UI screen shots in the document are taken by opening the profiling reports on a Windows 10 system.

Chapter 4.
INITIAL VERSION OF THE KERNEL

The initial version of the kernel transposeCoalesced uses shared memory to ensure that global memory accesses for loading data from the input matrix **idata** and storing data in the output matrix **odata** are coalesced. The matrix is sub-divided into tiles of size 32 x 32. The tile size is defined as:

```
#define TILE_DIM   32
```

For simplicity the code only handles square matrices whose dimensions are integral multiples of 32, the tile size. Each block transposes a tile of 32 x 32 elements. Each thread in the block transposes TILE_DIM/BLOCK_ROWS i.e. 4 elements, where BLOCK_ROWS is defined as:

```
#define BLOCK_ROWS   8
```

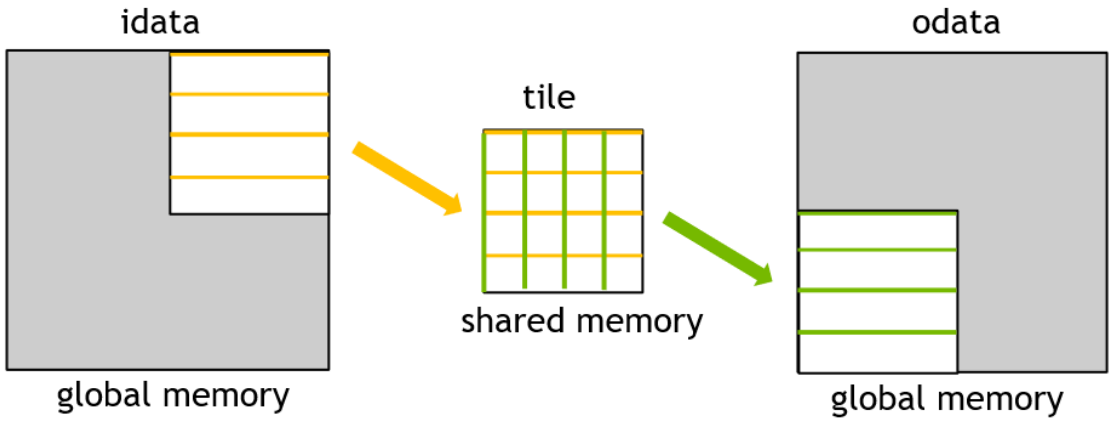
TILE_DIM must be an integral multiple of BLOCK_ROWS.

The way to avoid uncoalesced global memory access is to read the data into shared memory, and have each warp access noncontiguous locations in shared memory in order to write contiguous data to **odata**. The above procedure requires that each element in a tile be accessed by different threads, so a **__syncthreads()** call is required to ensure

that all reads from **idata** to shared memory have completed before writes from shared memory to **odata** commence.

```
__global__ void transposeCoalesced(float* odata, float* idata, int width, int
 height)
{
   __shared__ float tile[TILE_DIM][TILE_DIM];

   int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
   int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
   int indexIn = xIndex + yIndex*width;

   xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
   yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
   int indexOut = xIndex + yIndex*height;

   for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS)
   {
       tile[threadIdx.y + i][threadIdx.x] = idata[indexIn + i * width];
   }

   __syncthreads();

   for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS)
   {
       odata[indexOut + i * height] = tile[threadIdx.x][threadIdx.y + i];
   }
}
```

A depiction of the data flow of a warp in the coalesced transpose kernel is given below. The warp writes four rows of the **idata** matrix tile to the shared memory 32x32 array "tile" indicated by the yellow line segments. After a **__syncthreads()** call to ensure all writes to tile are completed, the warp writes four columns of tile to four rows of an **odata** matrix tile, indicated by the green line segments.



Profile the initial version of the kernel

There are multiple ways to profile kernels with Nsight Compute. For full details see the Nsight Compute Documentation. One example workflow to follow for this sample:

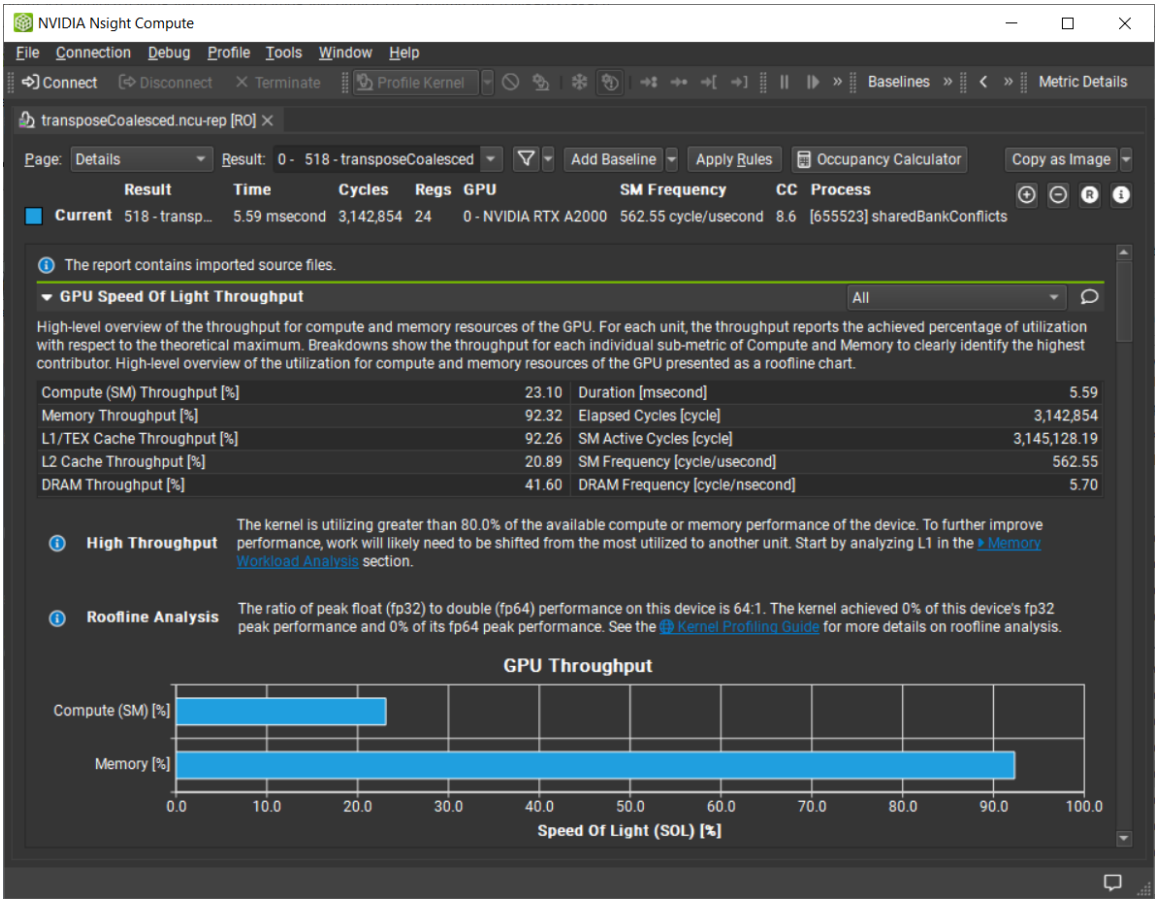
- Refer to the README distributed with the sample on how to build the application
- Run ncu-ui on the host system

- Use a local connection if the GPU is on the host system. If the GPU is on a remote system, set up a remote connection to the target system
- Use the "Profile" activity to profile the sample application
- Choose the "full" section set
- Use defaults for all other options

Details page - GPU Speed Of Light Throughput

The details page "GPU Speed Of Light Throughput" section provides a high-level overview of the throughput for compute and memory resources of the GPU used by the kernel.

The duration for this initial version of the kernel is 5.59 milliseconds and this is used as the baseline for further optimizations.

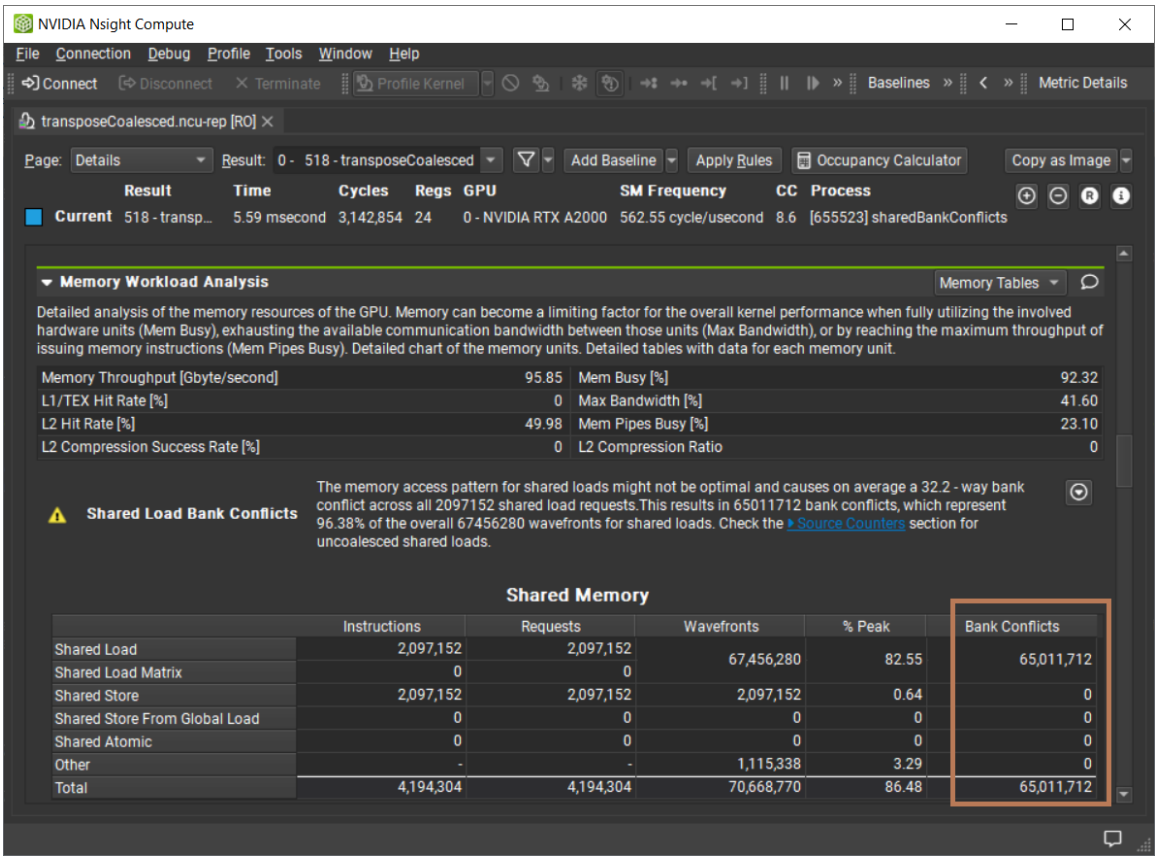


For this kernel it shows a hint for High Throughput and suggests looking at the memory workload analysis section. Click on Memory Workload Analysis.

Details page - Memory Workload Analysis section

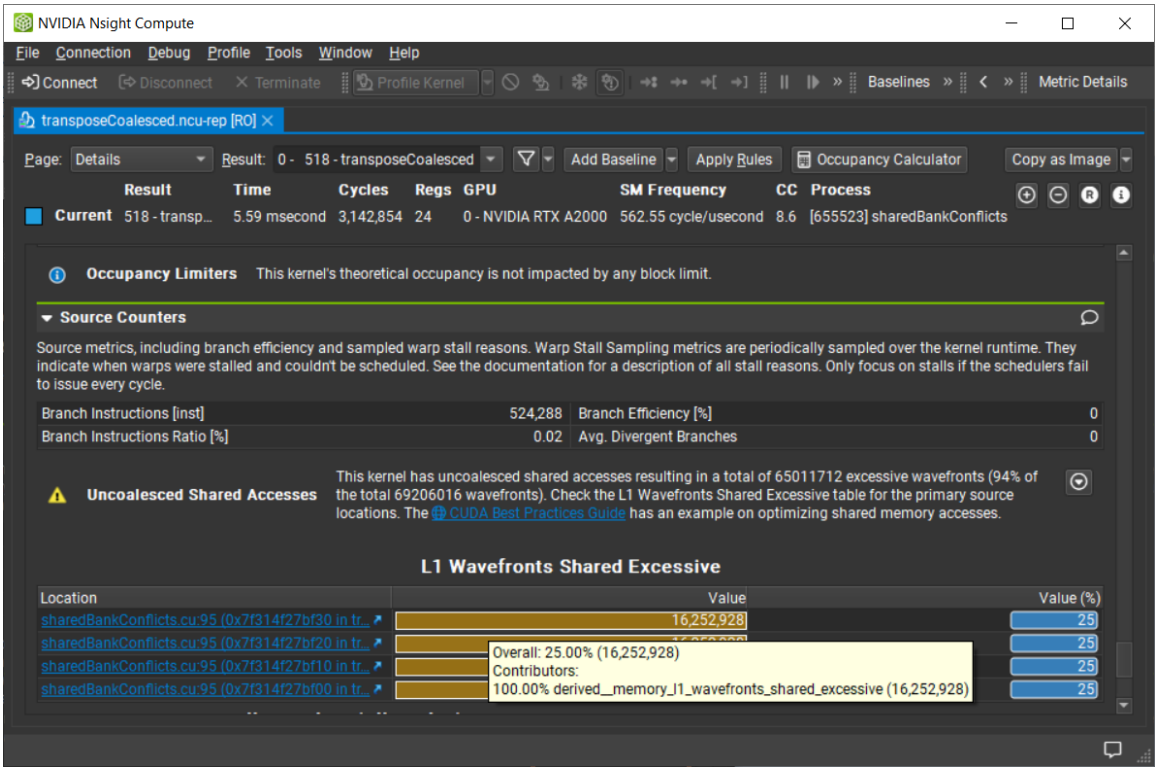
The Memory Workload Analysis section shows a hint for Shared Load Bank Conflicts and suggests looking at the Source Counters section for uncoalesced shared loads. The

Shared Memory table shows a high count of bank conflicts. Click on Source Counters to identify the source line resulting in bank conflicts.



Details page - Source Counters section

The Source Counters section table for the metric "L1 Wavefronts Shared Excessive" which is the indicator for shared memory bank conflicts lists the source lines with the highest value. Click on one of the source lines to view the kernel source at which the bottleneck occurs.

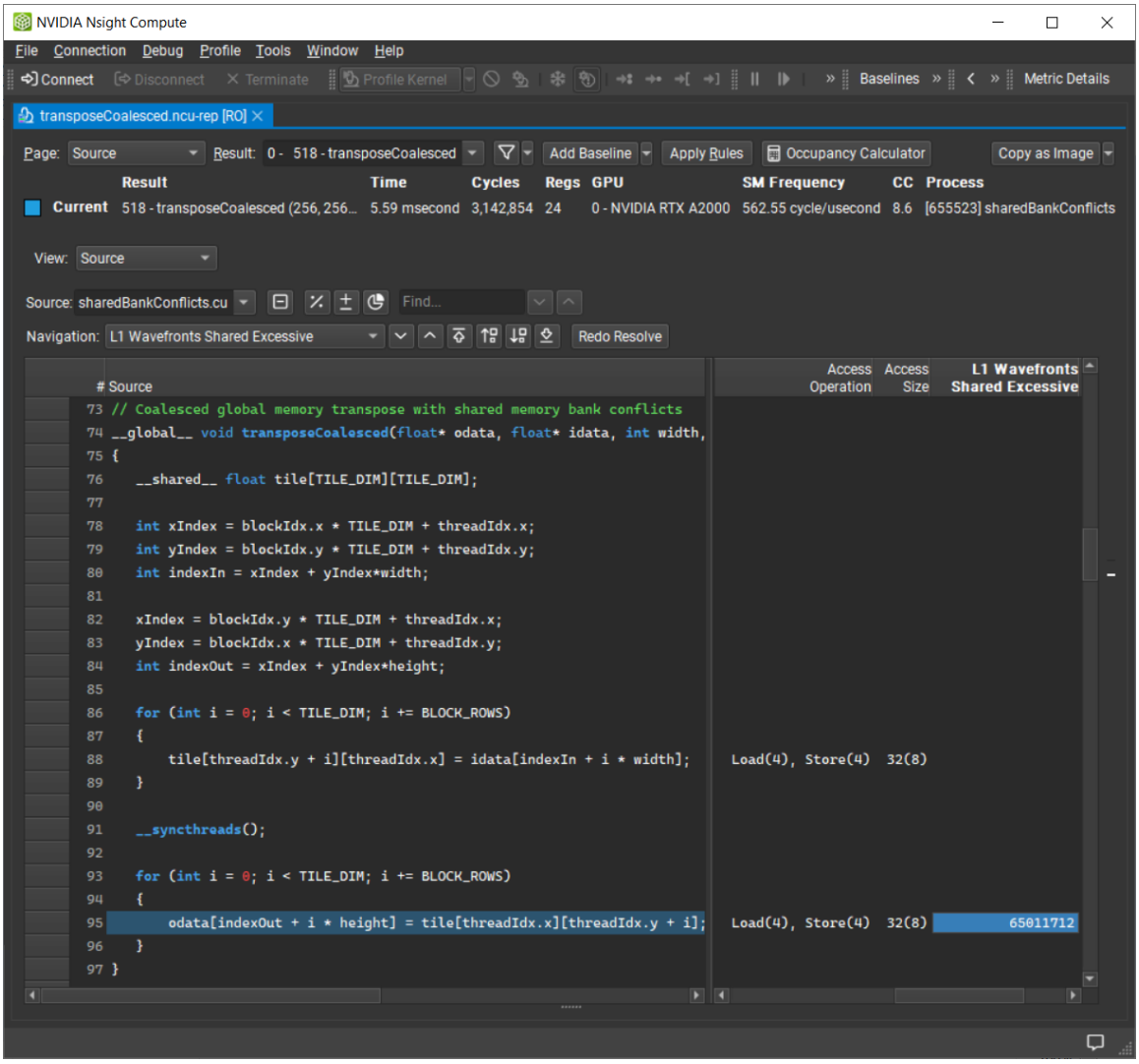


Source page

The CUDA source for the kernel is shown. When opening the Source page from Source Counters section, the Navigation metric is automatically filled in to match, in this case the "L1 Wavefronts Shared Excessive" metric. You can see this by the bolding in the column header. The source line at which the bottleneck occurs is highlighted.

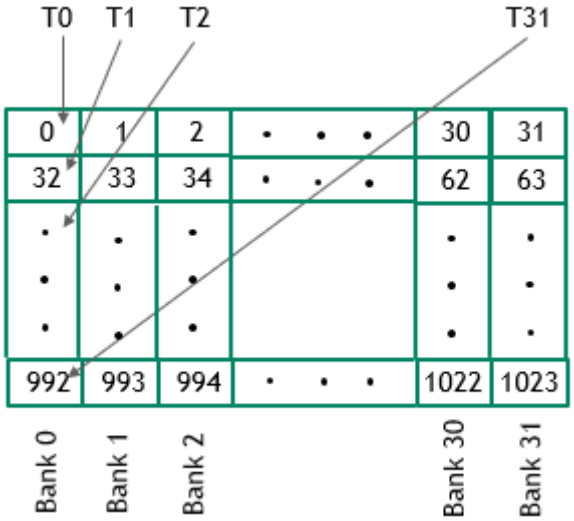
It shows shared memory bank conflicts at line #68:

```
        odata[indexOut + i * height] = tile[threadIdx.x][threadIdx.y + i];
```


NVIDIA Nsight Compute
File Connection Debug Profile Tools Window Help
Connect Disconnect Terminate Profile Kernel Baselines Metric Details
transposeCoalesced.ncu-rep [RO]
Page: Source Result: 0 - 518 - transposeCoalesced Add Baseline Apply Rules Occupancy Calculator Copy as Image
Result Time Cycles Regs GPU SM Frequency CC Process
Current 518 - transposeCoalesced (256, 256... 5.59 msecond 3,142,854 24 0 - NVIDIA RTX A2000 562.55 cycle/usecond 8.6 [655523] sharedBankConflicts
View: Source
Source: sharedBankConflicts.cu Find...
Navigation: L1 Wavefronts Shared Excessive Redo Resolve
Source
Access Operation
Access Size
L1 Wavefronts Shared Excessive
73 // Coalesced global memory transpose with shared memory bank conflicts
74 __global__ void transposeCoalesced(float* odata, float* idata, int width,
75 {
76 __shared__ float tile[TILE_DIM][TILE_DIM];
77
78 int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
79 int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
80 int indexIn = xIndex + yIndex*width;
81
82 xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
83 yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
84 int indexOut = xIndex + yIndex*height;
85
86 for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS)
87 {
88 tile[threadIdx.y + i][threadIdx.x] = idata[indexIn + i * width];
Load(4), Store(4) 32(8)
89 }
90
91 __syncthreads();
92
93 for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS)
94 {
95 odata[indexOut + i * height] = tile[threadIdx.x][threadIdx.y + i];
Load(4), Store(4) 32(8) 65011712
96 }
97 }

Chapter 5. UPDATED VERSION OF THE KERNEL

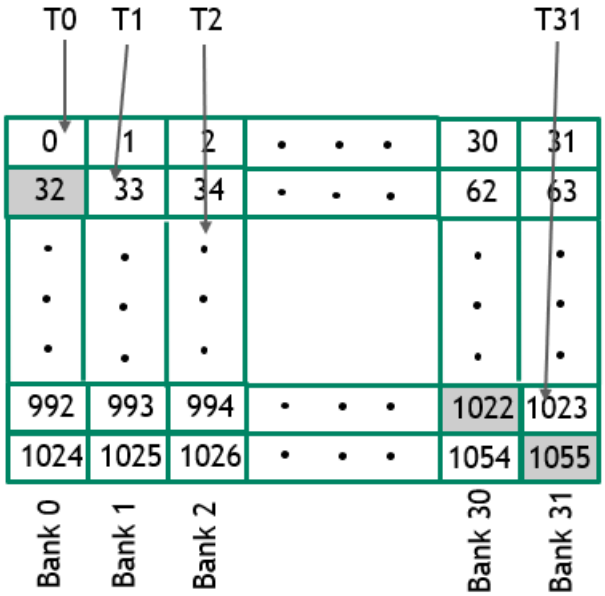
Considering the shared memory bank conflicts reported by the profiler we analyze the shared memory access pattern. The coalesced transpose uses a 32x32 shared memory array of floats. For this sized array, all data in each column is mapped to the same shared memory bank. As a result, when writing columns from the tile in shared memory to rows in **odata** the warp experiences a 32-way bank conflict and serializes the request.



A simple way to avoid this conflict is to pad the shared memory array by one column:

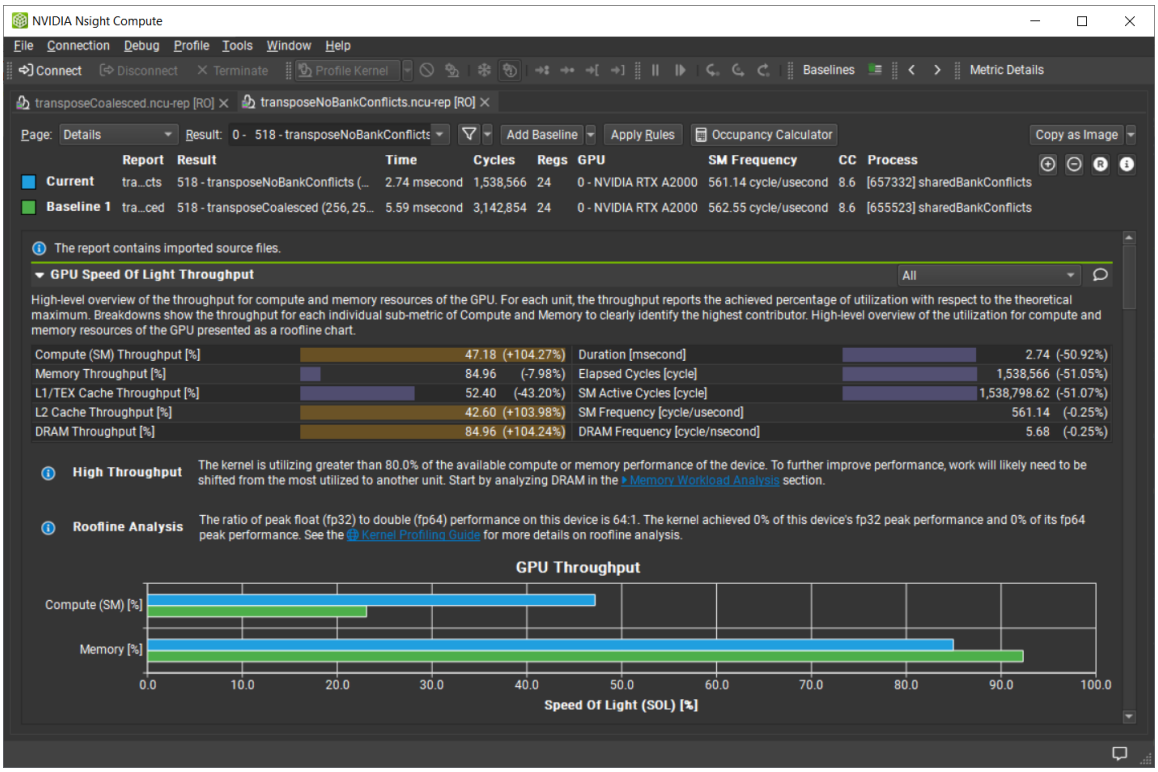
```
shared float tile[TILE_DIM][TILE_DIM+1];
```

The padding does not affect shared memory bank access pattern when a warp writes data to shared memory, which remains conflict free. But by adding a single column now the access of data in a column from shared memory by a warp is also conflict free. In the diagram below the elements in the extra column added for padding are shown with grey background.

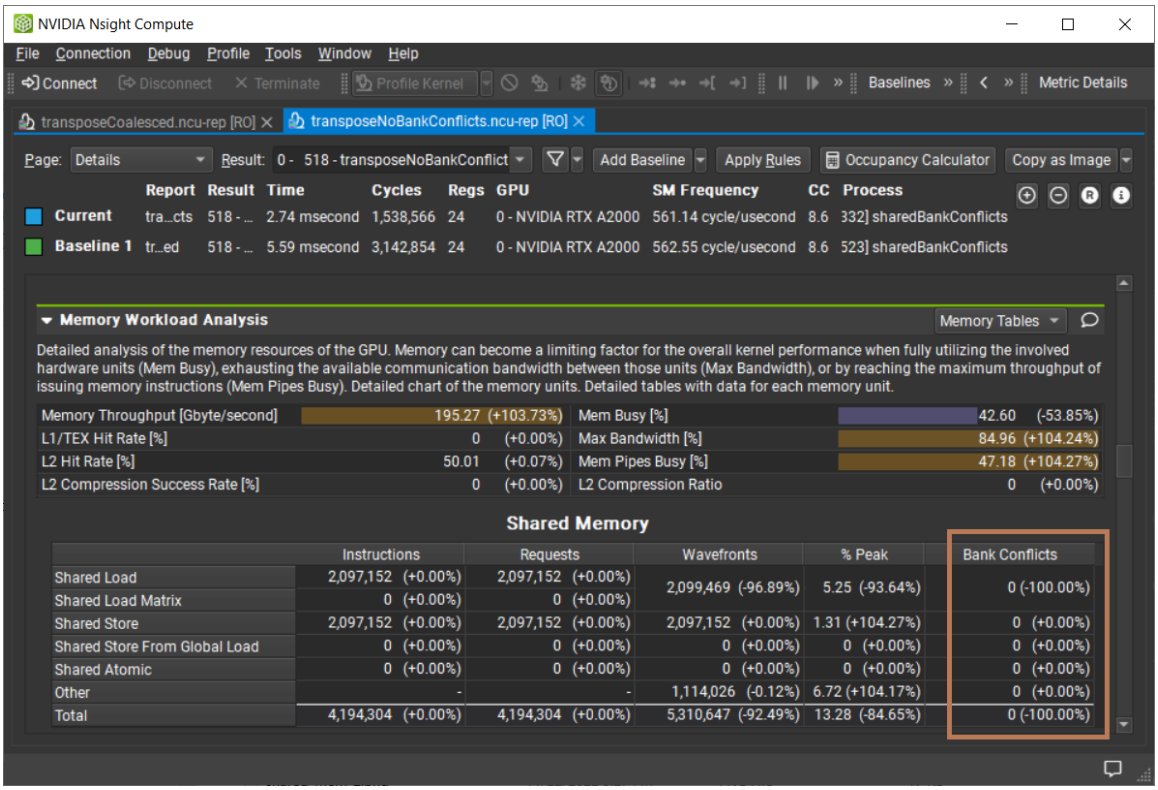


Profile the updated kernel

The kernel duration has reduced from 5.59 milliseconds to 2.74 milliseconds. We can set a baseline to the initial version of the kernel and compare the profiling results.



We can confirm that there are no shared memory bank conflicts by looking at the Shared Memory metrics table under the Memory workload analysis section.



Note that the reported bank conflicts in the shared memory metrics table under the Memory workload analysis section includes:

- (A) conflicts within the warp due to shared memory access pattern for the active threads of the warp; and
- (B) additional conflicts that are caused by multiple clients trying to access the memory banks at the same time, as the L1 Cache and Shared Memory are both backed by the same physical memory banks.

The Source Counters section in the Details page and the Source page only count conflicts of type (A) mentioned above. So in some cases there can be a difference in bank conflict counts between the Memory workload analysis and source counters. Also due to conflicts of type (B) in some cases the bank conflicts can be non-zero for the **`transposeNoBankConflicts`** kernel in the shared memory table.

Chapter 6. RESOURCES

- GPU Technology Conference 2022 talk S41723: How to Understand and Optimize Shared Memory Accesses using Nsight Compute
- NVIDIA CUDA Sample transpose document - Optimizing Matrix Transpose in CUDA https://github.com/NVIDIA/cuda-samples/blob/master/Samples/6_Performance/transpose/doc/MatrixTranspose.pdf
- NVIDIA CUDA Sample transpose source code transpose.cu
- Nsight Compute Documentation

Notice

Trademarks

Copyright