UNCOALESCED GLOBAL ACCESSES SAMPLE

v2023.3.1 | December 2023

TABLE OF CONTENTS

Chapter 1. INTRODUCTION

This sample profiles a memory-bound CUDA kernel which does a simple computation on an array of double3 data type in global memory using the Nsight Compute profiler. The profiler is used to analyze and identify the memory accesses which are uncoalesced and result in inefficient DRAM accesses.

Global memory accesses on a GPU

Global memory resides in device memory and device memory is accessed via 32, 64, or 128-byte memory transactions.

When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the data accessed by each thread and the distribution of the memory addresses across the threads. If global memory accesses of the threads within a warp cannot be combined into the same memory transaction then we refer to these as uncoalesced global memory accesses. In general, the more transactions are necessary, the more unused bytes are transferred in addition to the bytes accessed by the threads, reducing the instruction throughput accordingly. For example, if a 32-byte memory transaction is generated for each thread's 4-byte access, throughput is divided by 8.

Chapter 2. APPLICATION

The sample CUDA application adds a floating point constant to an input array of 1,048,576 (1024*1024) double3 elements in global memory and generates an output array of double3 in global memory of the same size. double3 is a 24-byte built-in vector type which is a structure containing 3 double precision floating point values:

```
struct
{
    double x, y, z;
};
```

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

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 A4500 (Ampere GA102) GPU
- Nsight Compute version: 2023.3.1

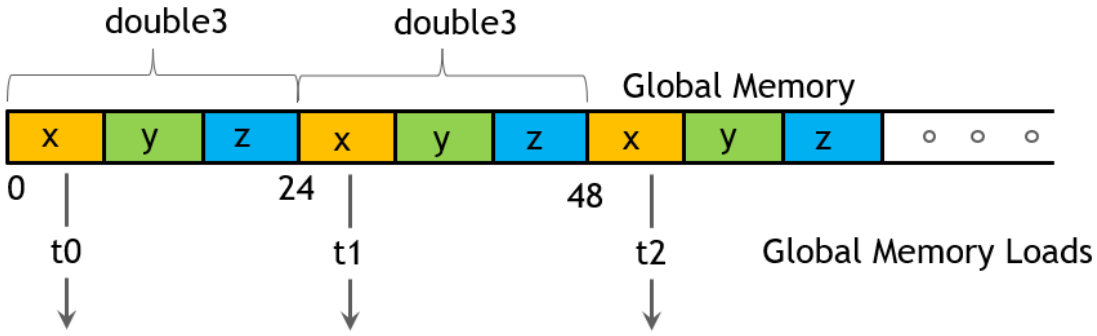
The Nsight Compute UI screen shots in the document are taken by opening the profiling reports on a Windows 10 system.

Chapter 4.
INITIAL VERSION OF THE KERNEL

The initial version of the sample code provides a naive implementation for the kernel which adds a floating point constant to an input array of double3.

```
__global__ void addConstDouble3(int numElements, double3 *d_in, double k,
 double3 *d_out)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < numElements)
    {
        double3 a = d_in[index];
        a.x += k;
        a.y += k;
        a.z += k;
        d_out[index] = a;
    }
}
```

The instruction **`a = d_in[index]`** in the kernel code results in each thread in a warp accessing global memory 24-bytes apart. In the first step all threads request a load for **`d_in[index].x`** as shown in the following diagram. In the second step a load for **`d_in[index].y`** and in the third step a load for **`d_in[index].z`** is made by all threads.



The instruction **`d_out[index] = a;`** has a similar multistep storage pattern.

Profile the initial version of the kernel

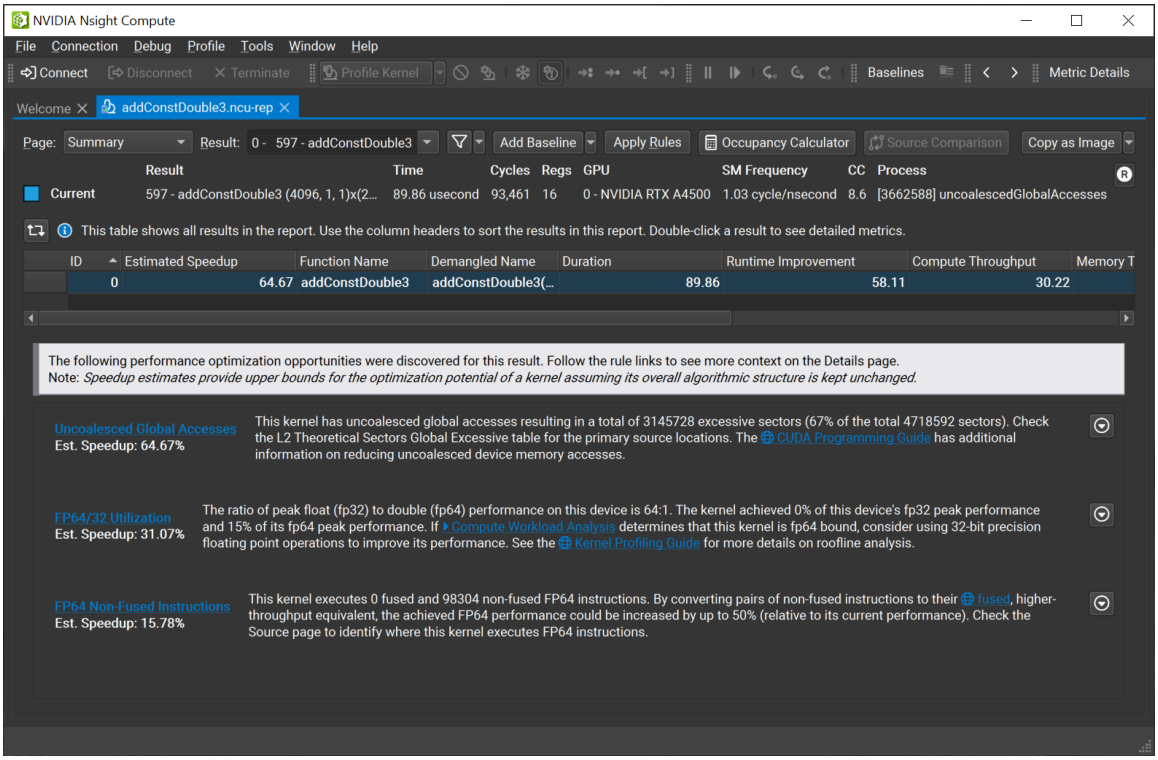
There are multiple ways to profile kernels with Nsight Compute. For full details see the Nsight Compute Documentation. One example workflow to follow for this sample:

- 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
- Set a report name and then click on **Launch**

Summary page

The **Summary** page lists the kernels profiled and provides some key metrics for each profiled kernel. It also lists the performance opportunities and estimated speedup for each. In this sample we have only one kernel launch.

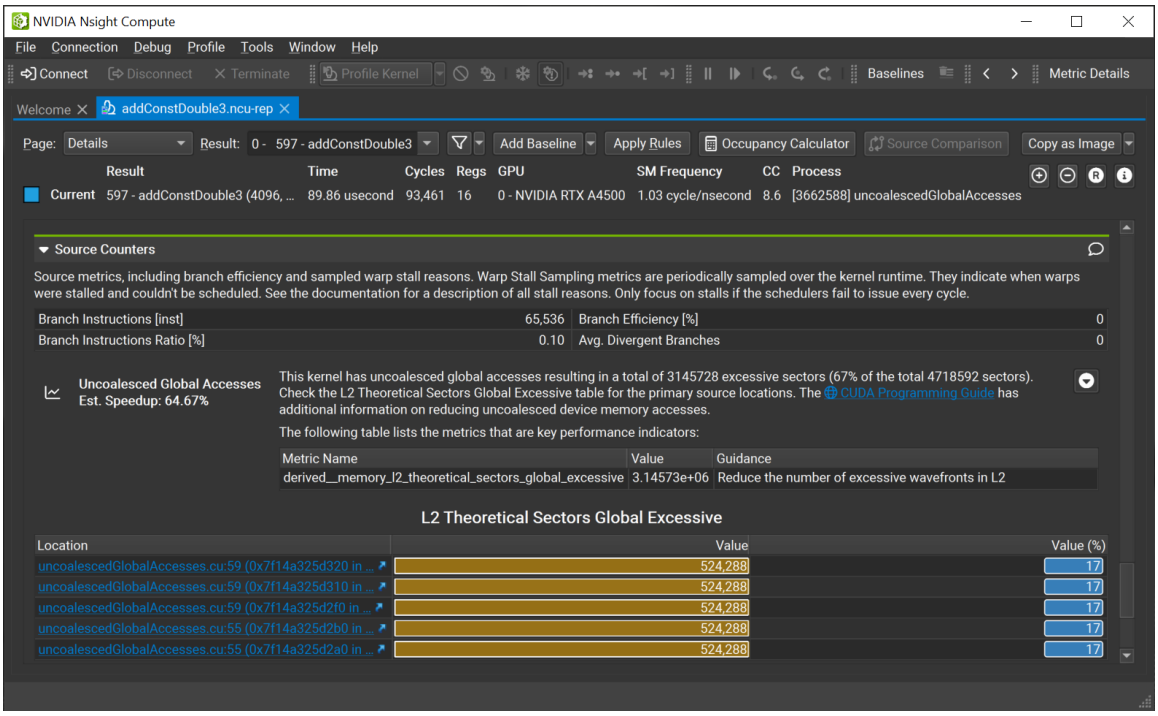
The duration for this initial version of the kernel is 89.86 microseconds and this is used as the baseline for further optimizations.



For this kernel it shows a hint for **Uncoalesced Global Accesses** and suggests checking the **L2 Theoretical Sectors Global Excessive** table for the primary source locations. Click on **Uncoalesced Global Accesses** rule link to see more context on the **Details** page. It opens the **Source Counters** section on the **Details** page.

Details page - Source Counters section

The Source Counters section shows a hint for **`Uncoalesced Global Accesses`**. It explains that the metric **`L2 Theoretical Sectors Global Excessive`** is the indicator for uncoalesced accesses. The table for this metric lists the source lines with the highest value. Click on the **`Apply Rules`** button at the top to apply rules so that the we can also see the hints at the source line level on the source page. Click on one of the source lines to view the kernel source at which the bottleneck occurs.



Source page

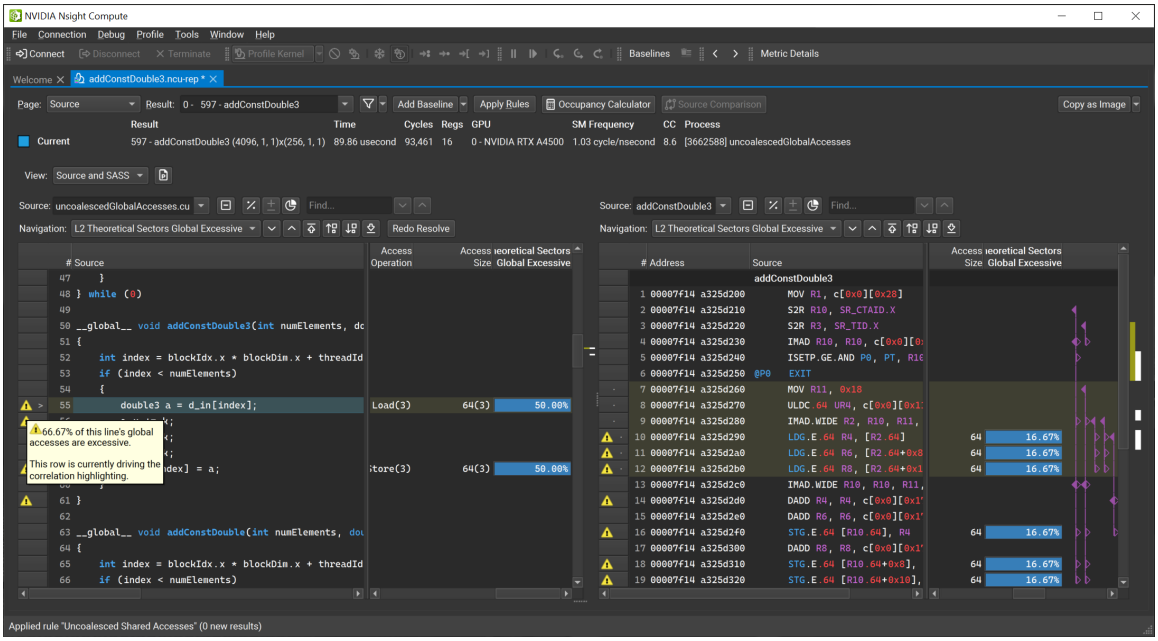
The CUDA source and SASS(GPU Assembly) for the kernel is shown side by side. When opening the Source page from Source Counters section, the Navigation metric is automatically filled in to match, in this case **`L2 Theoretical Sectors Global Excessive`**. You can see this by the bolding in the column header. The source line at which the bottleneck occurs is highlighted.

It shows uncoalesced global memory load accesses at line #55:

```
        double3 a = d_in[index];
```

It shows uncoalesced global memory store accesses at line #59:

```
        d_out[index] = a;
```

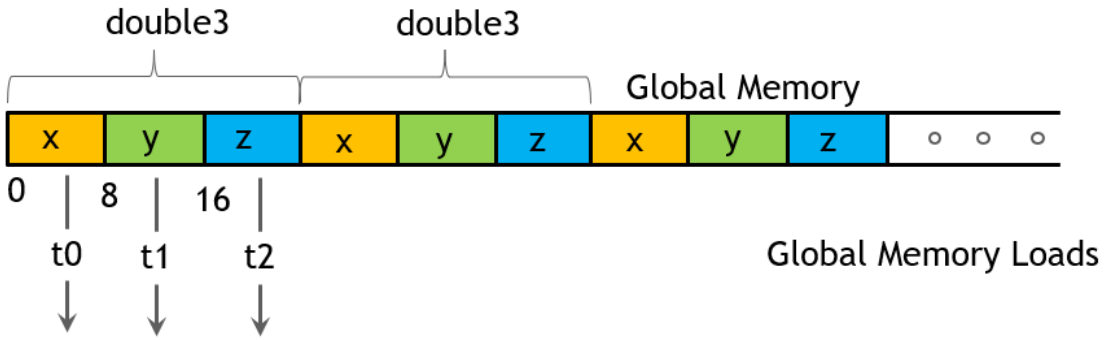


The source page shows notification as Source Markers in the left header of both the source and SASS code. By hovering the mouse on a marker it shows details in a pop-up window for the specific source line.

Chapter 5.
UPDATED VERSION OF THE KERNEL

Considering the uncoalesced accesses reported by the profiler we analyze the global load access pattern. Each thread executes 3 reads for the three double values in double3.

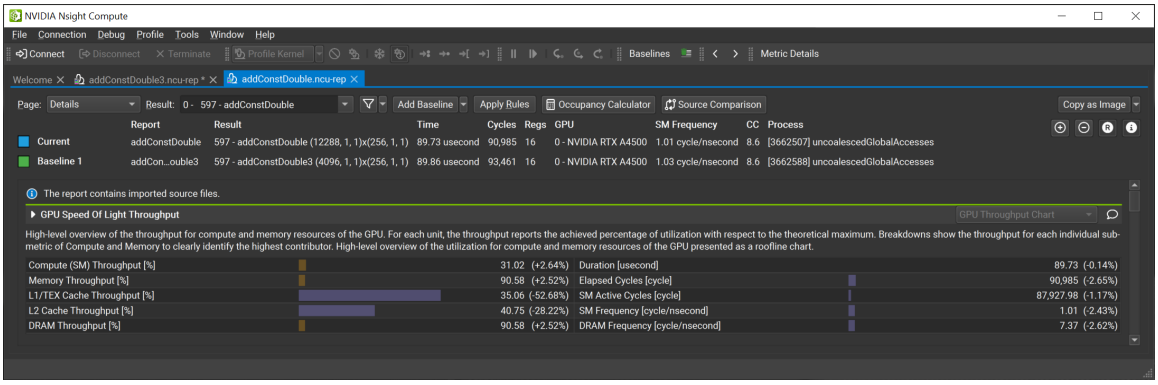
We can treat the double3 array as a double array and each thread can process one double instead of one double3. With this change threads in a warp access consecutive double values and both loads and stores are coalesced.



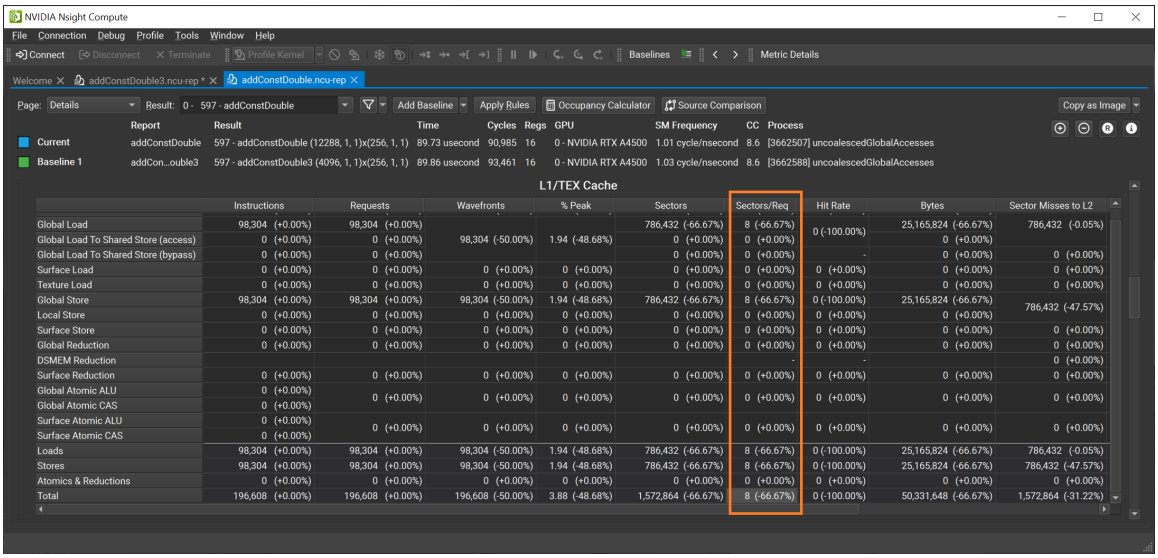

```
__global__ void addConstDouble(int numElements, double *d_in, double k, double
 *d_out)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < numElements)
    {
        d_out[index] = d_in[index] + k;
    }
}
```

Profile the updated kernel

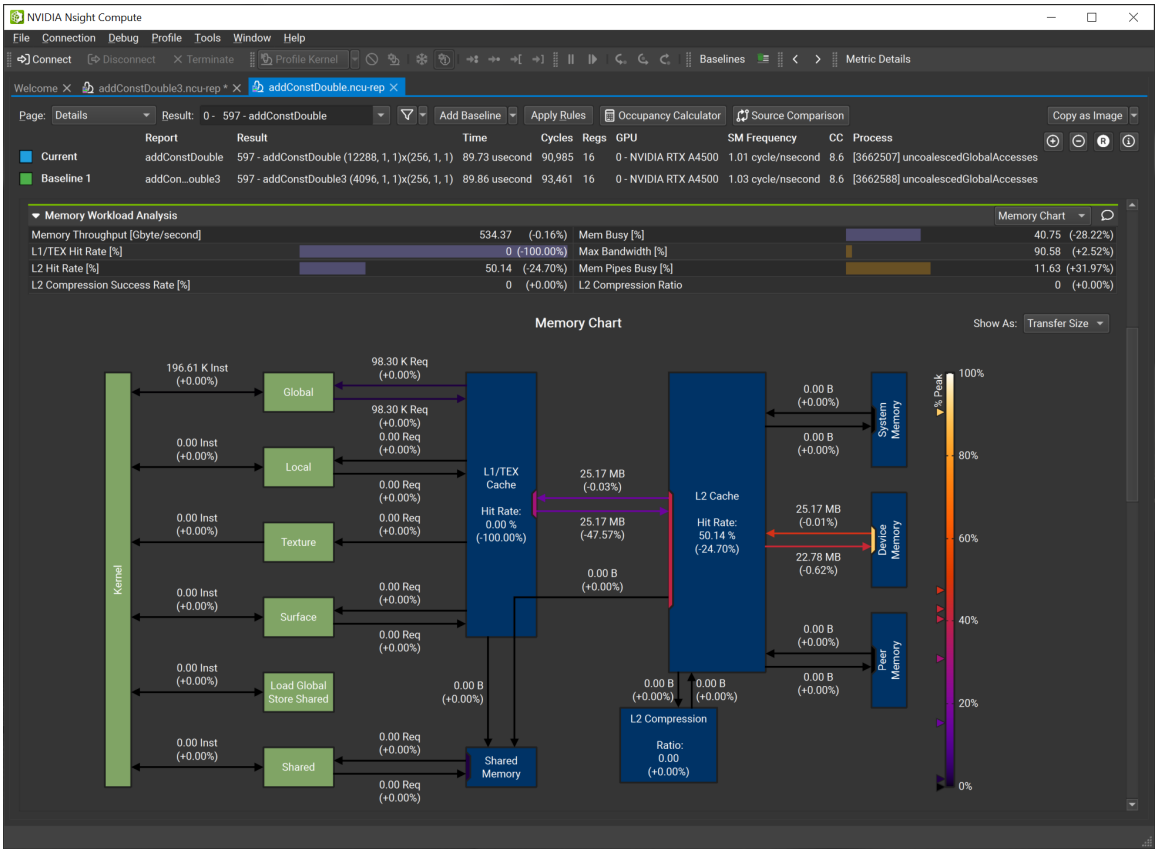
After profiling the updated version, we can set a baseline to the initial version of the kernel and compare the profiling results. The kernel duration has reduced only slightly on this GPU from 89.86 microseconds to 89.73 microseconds. However, the **`Speed Of Light`** section already reveals that the pressure on the L1/TEX and L2 Caches has significantly been reduced.



We can also confirm that the global memory accesses are now coalesced. In the L1/TEX Cache metrics table under the **`Memory Workload Analysis`** section we see that the **`Sectors/Req`** metric value is 8 for both global loads and global stores. Also the **`Source Counters`** section does not show any **`Uncoalesced Global Accesses`**.



To get a better understanding of why the runtime did not decrease for the updated version of the kernel on this device we can navigate to the **`Memory Chart`**. The chart reveals that previously the caches were able to counteract the bad memory accesses pattern, as is apparent from the significant drop in the L2 hit rate and the much reduced data transfer between L1 and L2 caches in the updated version of the kernel. The chart also indicates that the actual bottleneck is the DRAM throughput, which is closest to its peak. Because the amount of data transferred between Device Memory and the L2 Cache did not significantly change between the two versions of the kernel, the runtime did not change either.



Improving the memory access pattern as shown here, and thereby reducing the pressure on the caches, will have a significant impact on the runtime, in particular when using non-streaming memory access patterns, and on device with smaller caches.

Chapter 6. RESOURCES

- GPU Technology Conference 2021 talk S32089: Requests, Wavefronts, Sectors Metrics: Understanding and Optimizing Memory-Bound Kernels with Nsight Compute
- Nsight Compute Documentation

Notice

Trademarks

Copyright