



INSTRUCTION MIX SAMPLE

v2023.1.0 | September 2023



TABLE OF CONTENTS

Chapter 1. Introduction.....	1
Chapter 2. Application.....	2
Chapter 3. Configuration.....	3
Chapter 4. Initial version of the kernel.....	4
Chapter 5. Updated version of the kernel.....	9
Chapter 6. Resources.....	11

Chapter 1.

INTRODUCTION

This sample profiles a CUDA kernel which applies a simple sobel edge detection filter to a image in global memory using the Nsight Compute profiler. The profiler is used to analyze and identify the performance bottleneck due to an imbalanced instruction mix.

Chapter 2. APPLICATION

This sample CUDA application applies a simple sobel edge detection filter to an image in global memory. The input and output images are at separate memory locations. For simplicity it only handles image sizes which are an integral multiple of block size. (**BLOCK_SIZE** - defined in the source file "**instructionMix.cu**")

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

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: 2023.3.0

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 Sobel operator performs a 2-D spatial gradient measurement on an image and so emphasizes regions of high spatial frequency that correspond to edges. Typically it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image. Each thread applies the Sobel operator to one pixel of the input image and generates one pixel of the output image. The operator uses two 3x3 kernels which are convolved with the original image to calculate approximations of the derivatives - one

for horizontal changes, and one for vertical. The **Sobel** kernel is defined as a function template that can be used as a generic function for different floating point precisions.

```
template<typename FLOAT_T>
__global__ void Sobel(
    uchar4* pOut,
    uchar4* pImg,
    const int imgWidth,
    const int imgHeight)
{
    const int tx = blockIdx.x * blockDim.x + threadIdx.x;
    const int ty = blockIdx.y * blockDim.y + threadIdx.y;
    const int outIdx = ty * imgWidth + tx;

    const int SX[] = {1, 2, 1, 0, 0, 0, -1, -2, -1};
    const int SY[] = {1, 0, -1, 2, 0, -2, 1, 0, -1};

    FLOAT_T sumX = 0.;
    FLOAT_T sumY = 0.;
    for (int j = -1; j <= 1; ++j)
    {
        for (int i = -1; i <= 1; ++i)
        {
            const auto idx = (j + 1) * 3 + (i + 1);
            const auto sx = SX[idx];
            const auto sy = SY[idx];

            const auto luminance = GetPixel(pImg, tx + i, ty + j, imgWidth,
imgHeight);
            sumX += (FLOAT_T)luminance * (FLOAT_T)sx;
            sumY += (FLOAT_T)luminance * (FLOAT_T)sy;
        }
    }

    sumX /= (FLOAT_T)9.;
    sumY /= (FLOAT_T)9.;

    const FLOAT_T threshold = 24.;
    if (sumX > threshold || sumY > threshold)
    {
        pOut[outIdx] = make_uchar4(0, 255, 255, 0);
    }
}
```

The initial version of the kernel **Sobel** executes the math operations on the grayscale values in double precision floating point accuracy.

```
Sobel<double><<<grid, block>>>( pDstImage, pSrcImage, imgWidth,
imgHeight);
```

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 is to perform the following steps:

- ▶ 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.

The screenshot shows the NVIDIA Nsight Compute interface. The main window displays the 'Summary' page for a kernel named 'Sobel'. The kernel's result is shown as '632 - Z5SobellidEvP6uchar4S1_ji (64, 64, 1)x(16, 16, 1)' with a duration of 627.87 usecond, 659,451 cycles, and 34 registers. The GPU used is 0 - NVIDIA RTX A4500, with an SM frequency of 1.05 cycle/nsecond and 8.6 [749038] instruction mix.

ID	Estimated Speedup	Function Name	Demangled Name	Duration	Runtime Improvement	Compute Throu
0	80.70	Sobel	void Sobel<double>(uchar4 *, uchar4 *, int, int)	627.87		506.71

Below the table, there are three performance optimization opportunities:

- FP64/32 Utilization**: The ratio of peak float (fp32) to double (fp64) performance on this device is 64.1. The kernel achieved close to 0% of this device's fp32 peak performance and 53% of its fp64 peak performance. If [Compute Workload Analysis](#) determines that this kernel is fp64 bound, consider using 32-bit precision floating point operations to improve its performance. See the [Kernel Profiling Guide](#) for more details on roffline analysis. **Est. Speedup: 80.70%**
- Tex Throttle**: On average, each warp of this kernel spends 207.0 cycles being stalled waiting for the L1 instruction queue for texture operations to be not full. This stall reason is high in cases of extreme utilization of the L1TEX pipeline. Try issuing fewer texture fetches, surface loads, surface stores, or decoupled math operations. If applicable, consider combining multiple lower-width memory operations into fewer wider memory operations and try interleaving memory operations and math instructions. Consider converting texture lookups or surface loads into global memory lookups. Texture can accept four threads' requests per cycle, whereas global accepts 32 threads. This stall type represents about 73.9% of the total average of 280.3 cycles between issuing two instructions. **Est. Speedup: 73.87%**
- FP64 Non-Fused Instructions**: This kernel executes 884736 fused and 655360 non-fused FP64 instructions. By converting pairs of non-fused instructions to their [fused](#), higher-throughput equivalent, the achieved FP64 performance could be increased by up to 21% (relative to its current performance). Check the Source page to identify where this kernel executes FP64 instructions. **Est. Speedup: 18.39%**

For this kernel it shows a hint for **FP64/32 Utilization** and suggests using 32-bit precision floating point operations to improve performance. Click on **FP64/32 Utilization** rule link to see more context on the **Details** page. It opens the **GPU Speed of Light Throughput** section on the **Details** page.

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 initial version of the kernel has a duration of 628.87 microseconds and this is used as the baseline for further optimizations.

The screenshot shows the NVIDIA Nsight Compute interface. The main window displays the following summary table:

Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	632 - _Z5Sob...	627.87 usecond	659,451	34	0 - NVIDIA RTX A4500	1.05 cycle/nsecond	8.6 [749038] instructionMix

Below the summary table, there are several informational sections:

- GPU Speed Of Light Throughput:** A high-level overview of the throughput for compute and memory resources of the GPU. It includes a table with the following data:

Metric	Value	Metric	Value
Compute (SM) Throughput [%]	85.19	Duration [usecond]	627.87
Memory Throughput [%]	1.97	Elapsed Cycles [cycle]	659,451
L1/TEX Cache Throughput [%]	2.00	SM Active Cycles [cycle]	650,032.12
L2 Cache Throughput [%]	1.10	SM Frequency [cycle/nsecond]	1.05
DRAM Throughput [%]	1.12	DRAM Frequency [cycle/nsecond]	7.60
- High Throughput:** A hint indicating that the kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. It suggests looking at the [Compute Workload Analysis](#) section.
- FP64/32 Utilization:** A hint indicating that the ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The kernel achieved close to 0% of this device's fp32 peak performance and 53% of its fp64 peak performance. It suggests looking at the [Compute Workload Analysis](#) section.

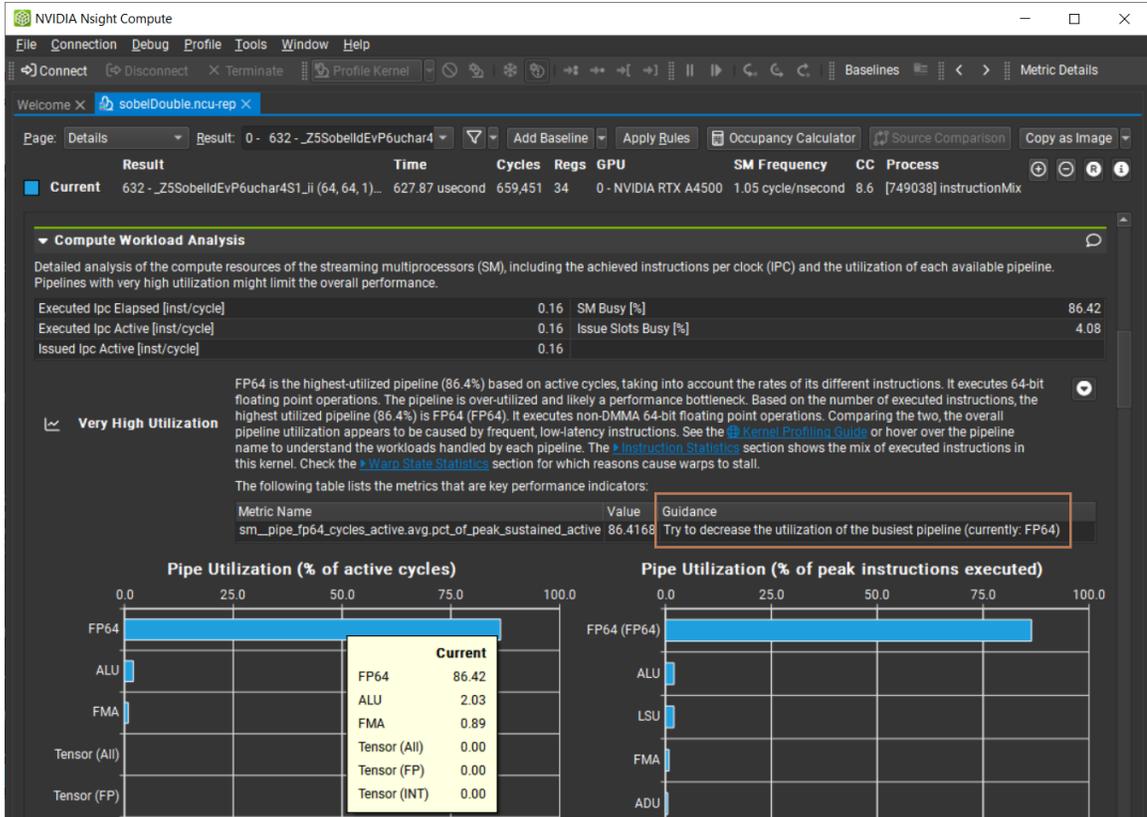
At the bottom of the interface, there are two detailed tables:

Compute Throughput Breakdown		Memory Throughput Breakdown	
SM: Pipe Fp64 Cycles Active [%]	85.19	L1: Lsuin Requests [%]	1.97
SM: Mio Pq Write Cycles Active [%]	20.33	L1: Data Pipe Lsu Wavefronts [%]	1.82
SM: Mio Inst Issued [%]	4.38	DRAM: Cycles Active [%]	1.12
SM: Issue Active [%]	4.02	L2: T Sectors [%]	1.10
SM: Inst Executed [%]	4.01	L1: Lsu Writeback Active [%]	1.09

For this kernel it shows a hint for **High Throughput** and **FP64/32 Utilization** and suggests looking at the **Compute Workload Analysis** section. Also we can see the **GPU Throughput Breakdown** tables at the bottom for Compute Throughput and Memory Throughput. The Compute Throughput Breakdown table shows that the SM FP64 pipe throughput is high (85.19%). Click on **Compute Workload Analysis** to analyze workloads.

Details page - Compute Workload Analysis section

The **Compute Workload Analysis** section shows a hint for **Very High Utilization**. It shows that FP64 is the highest-utilized pipeline (86.4%). The FP64 pipeline executes 64-bit floating point operations. It mentions that the pipeline is over-utilized and likely a performance bottleneck. The guidance provided is to try and decrease the utilization of the FP64 pipeline.



Chapter 5.

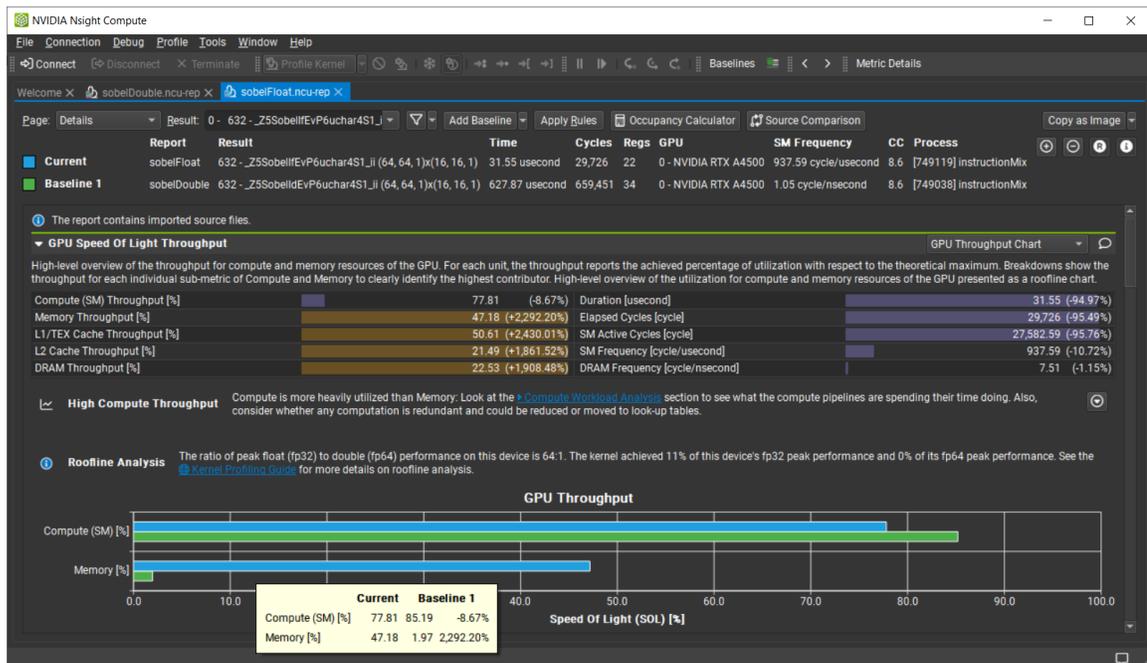
UPDATED VERSION OF THE KERNEL

Based on the profiler hint of high FP64 pipeline utilization, we modify the code to use single precision floating point instead of double precision. Since our input image has a very limited value range and the Sobel operator is not receptive to minor differences in precision, switching the computations from double to single precision has no negative impact on its functionality.

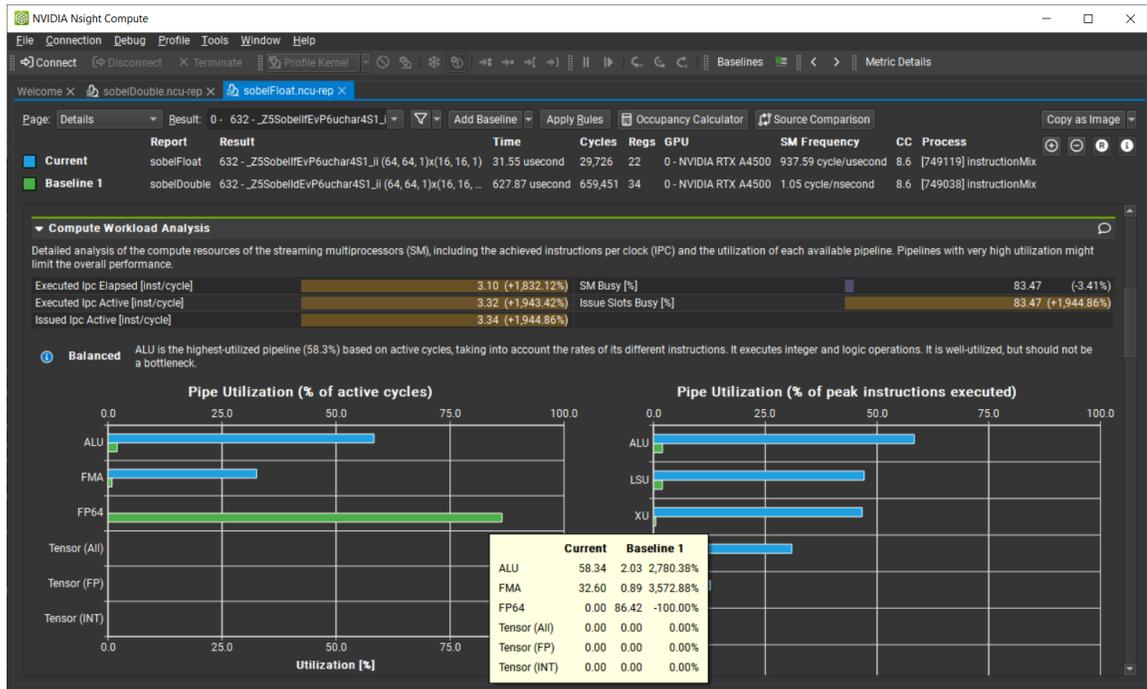
```
Sobel<float><<<grid, block>>>( pDstImage, pSrcImage, imgWidth, imgHeight);
```

Profile the updated kernel

The kernel duration has reduced from 627.87 microseconds to 31.55 microseconds. We can set a baseline to the initial version of the kernel and compare the profiling results.



We can confirm from the **Compute Workload Analysis** section that no pipeline has a high utilization.



It shows a message that the pipe utilization is balanced and now the ALU is the highest-utilized pipeline (58.3%). From the pipeline utilization chart we see that the FP64 pipeline utilization is reduced from 86.42% to 0% and the single precision FMA pipeline utilization has increased from 0.89% to 32.60%.

Chapter 6.

RESOURCES

- ▶ [Instruction Optimization section of the CUDA C++ Best Practices Guide](#)
- ▶ [Nsight Compute Documentation](#)

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2023-2023 NVIDIA Corporation and affiliates. All rights reserved.

This product includes software developed by the Syncro Soft SRL (<http://www.sync.ro/>).