Derek Alyne (dalyne2)
Joshua Sanchez (jsanch84)
Howard Shan (howards2)
ECE 408
Professor Sanjay Patel
8 March 2019

Report

I. Milestone 1

A. Include a list of all kernels that collectively consume more than 90% of the program time.

- `[CUDA memcpy HtoD]`
- `void cudnn::detail::implicit_convolve_sgemm<float, float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>(int, int, int, float const *, int, float*, cudnn::detail::implicit_convolve_sgemm<float, float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>*, kernel_conv_params, int, float, float, int, float, float, int, int)`
- `volta_cgemm_64x32_tn`
- `void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7, cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*, cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float, dimArray, reducedDivisorArray)`
- `void fft2d_c2r_32x32<float, bool=0, bool=0, unsigned int=1, bool=0, bool=0>(float*, float2 const *, int, int, int, int, int, int, int, int, int, float, float, cudnn::reduced_divisor, bool, float*, float*, int2, int, int)`
- `Volta_sgemm_128x128_tn`
- `void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>(cudnnTensorStruct, float const *, cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>, cudnnTensorStruct*, cudnnPoolingStruct, float, cudnnPoolingStruct, int, cudnn::reduced_divisor, float)`

- `void fft2d_r2c_32x32<float, bool=0, unsigned int=0, bool=0>(float2*, float const *, int, int, int, int, int, int, int, int, int, cudnn::reduced_divisor, bool, int2, int, int)`

B. Report: Include a list of all CUDA API calls that collectively consume more than 90% of the program time.

- `cudaStreamCreateWithFlags`
- `cudaMemGetInfo`
- `cudaFree`

C. Report: Include an explanation of the difference between kernels and API calls

- API calls are functions defined by the CUDA library, such as `cudaMalloc` and `cudaMemcpy`, while kernels are functions that the programmer (or other libraries) defines to run on the gpu. They are a minimal set of extensions to the C language and a runtime library to help the programmer interface with the gpu. Kernel functions are typically run a large number of times in parallel, using multiple blocks and threads. According to the CUDA documentation, “A kernel is defined using the __global__ declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<...>>>execution configuration syntax.”

D. Report: Show output of rai running MXNet on the CPU

- `✱ Running /usr/bin/time python m1.1.py`
- `Loading fashion-mnist data... done`
- `Loading model... done`
- `New Inference`
- `EvalMetric: {'accuracy': 0.8236}`
- `8.91user 3.64system 0:05.11elapsed 245%CPU (0avgtext+0avgdata 2470716maxresident)k`
- `0inp`
- `uts+2824outputs (0major+666444minor)pagefaults 0swaps`

E. Report: List program run time

- 5.11 seconds

F. Report: Show output of rai running MXNet on the GPU

- `✱ Running /usr/bin/time python m1.2.py`
- `Loading fashion-mnist data... done`
- `Loading model... done`
- `New Inference`
- `EvalMetric: {'accuracy': 0.8236}`
- `4.43user 3.37system 0:04.33elapsed 180%CPU (0avgtext+0av`
- `gdata 2841612maxresident)k`
- `8inputs+1728outputs (0major+660933minor)pagefaults 0swaps`

G. Report: List program run time

- 4.33 seconds

II. Milestone 2

A. Whole Program Execution Time

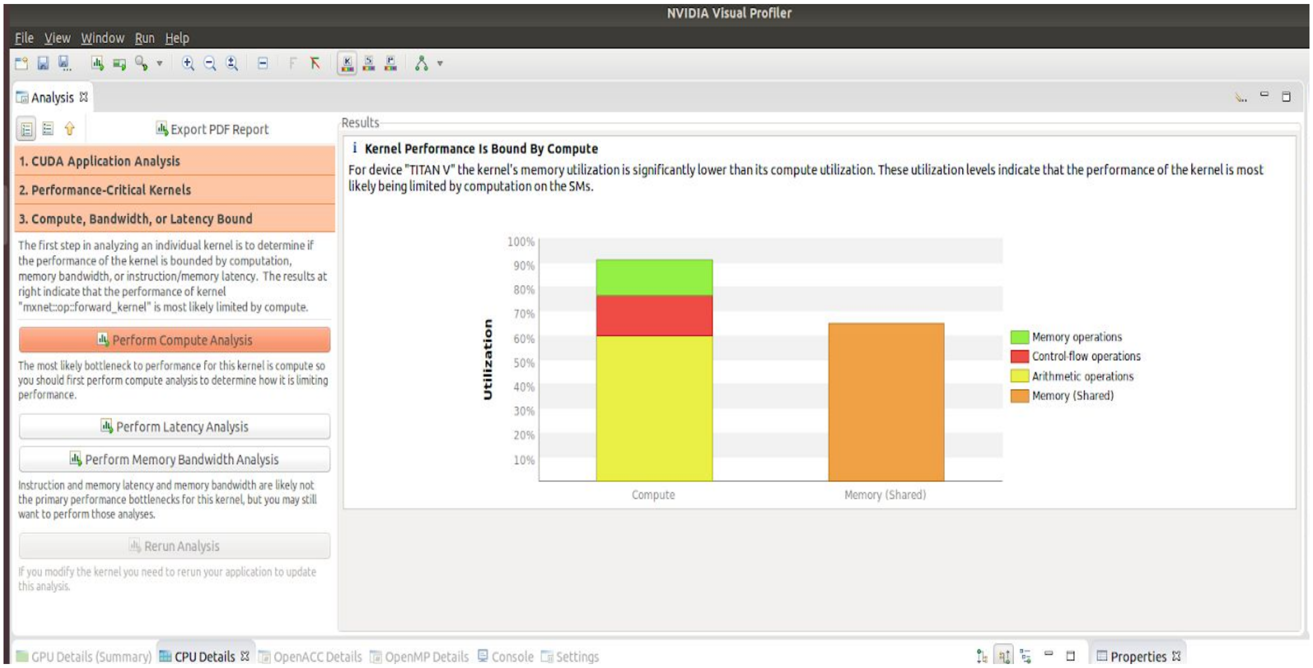
- 12.13 seconds

B. OpTimes:

- 2.583861 seconds
- 7.785734 seconds

III. Milestone 3

The following images are the result of loading analysis.nvvp into the Visual Profiler:



The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "mxnet::op::forward_kernel" is most likely limited by compute.

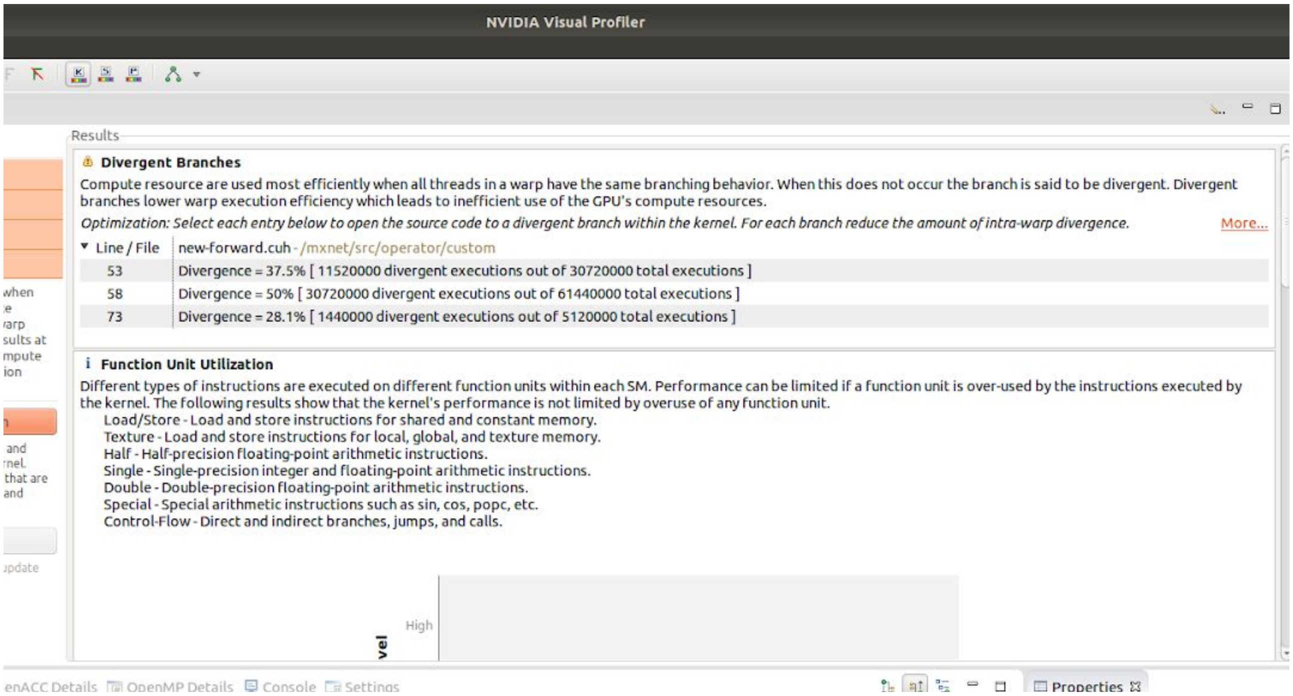
The most likely bottleneck to performance for this kernel is compute so you should first perform compute analysis to determine how it is limiting performance.

Instruction and memory latency and memory bandwidth are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

If you modify the kernel you need to rerun your application to update this analysis.

i **Kernel Performance Is Bound By Compute**

For device "TITAN V" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.



Divergent Branches

Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.

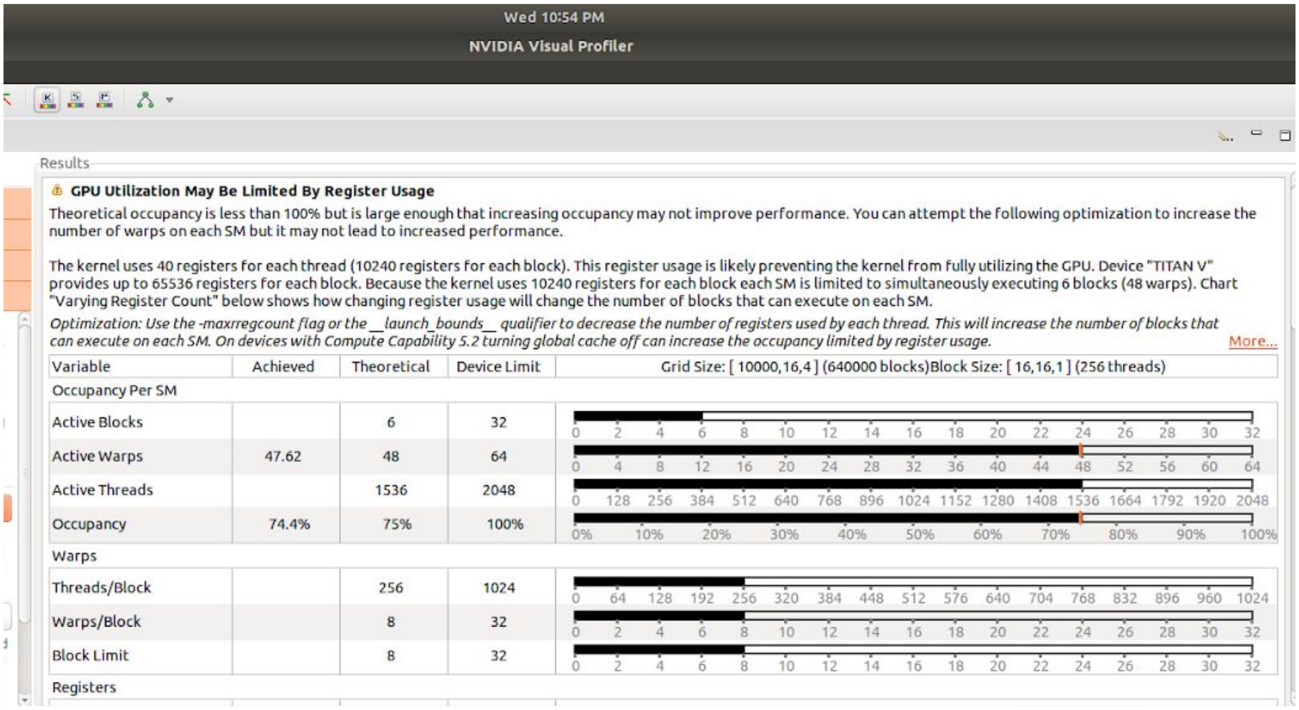
Optimization: Select each entry below to open the source code to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence. More...

▼ Line / File	new-forward.cuh - /mxnet/src/operator/custom
53	Divergence = 37.5% [11520000 divergent executions out of 30720000 total executions]
58	Divergence = 50% [30720000 divergent executions out of 61440000 total executions]
73	Divergence = 28.1% [1440000 divergent executions out of 5120000 total executions]

Function Unit Utilization

Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is not limited by overuse of any function unit.

Load/Store - Load and store instructions for shared and constant memory.
Texture - Load and store instructions for local, global, and texture memory.
Half - Half-precision floating-point arithmetic instructions.
Single - Single-precision integer and floating-point arithmetic instructions.
Double - Double-precision floating-point arithmetic instructions.
Special - Special arithmetic instructions such as sin, cos, popc, etc.
Control-Flow - Direct and indirect branches, jumps, and calls.



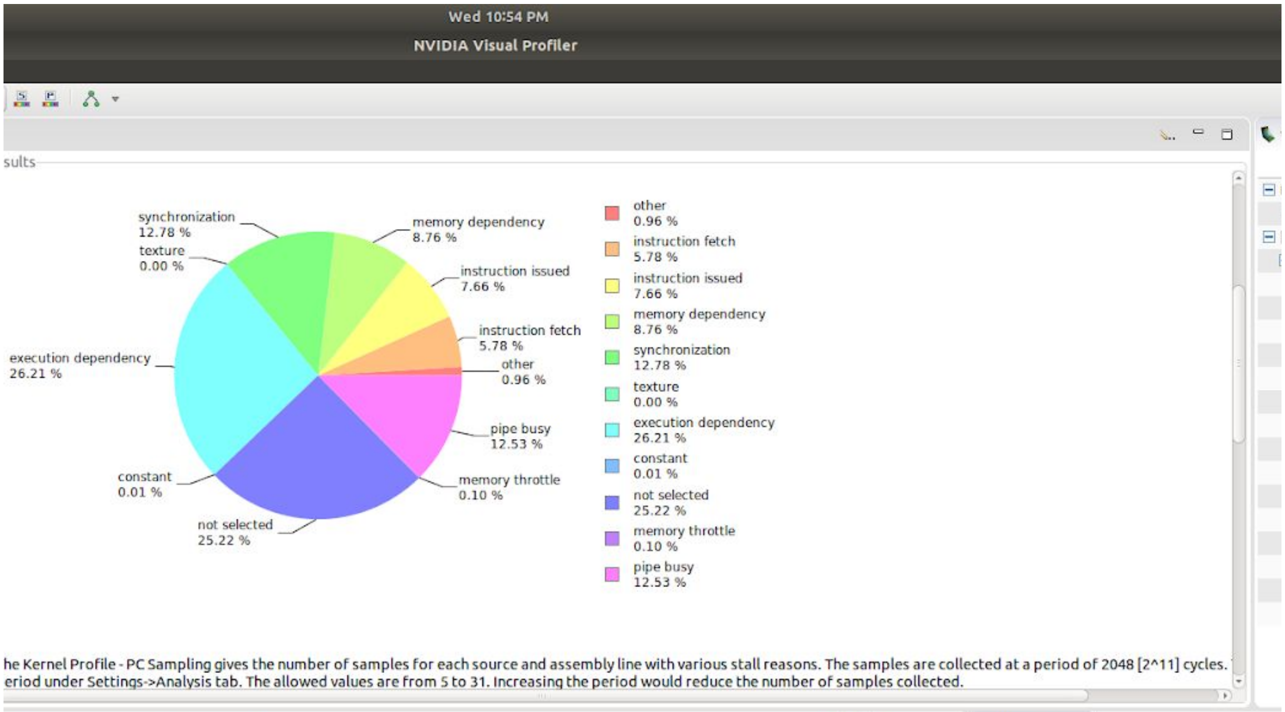
GPU Utilization May Be Limited By Register Usage

Theoretical occupancy is less than 100% but is large enough that increasing occupancy may not improve performance. You can attempt the following optimization to increase the number of warps on each SM but it may not lead to increased performance.

The kernel uses 40 registers for each thread (10240 registers for each block). This register usage is likely preventing the kernel from fully utilizing the GPU. Device "TITAN V" provides up to 65536 registers for each block. Because the kernel uses 10240 registers for each block each SM is limited to simultaneously executing 6 blocks (48 warps). Chart "Varying Register Count" below shows how changing register usage will change the number of blocks that can execute on each SM.

Optimization: Use the -maxrregcount flag or the __launch_bounds__ qualifier to decrease the number of registers used by each thread. This will increase the number of blocks that can execute on each SM. On devices with Compute Capability 5.2 turning global cache off can increase the occupancy limited by register usage. More...

Variable	Achieved	Theoretical	Device Limit	Grid Size: [10000,16,4] (640000 blocks)Block Size: [16,16,1] (256 threads)
Occupancy Per SM				
Active Blocks		6	32	0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
Active Warps	47.62	48	64	0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64
Active Threads		1536	2048	0 128 256 384 512 640 768 896 1024 1152 1280 1408 1536 1664 1792 1920 2048
Occupancy	74.4%	75%	100%	0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Warps				
Threads/Block		256	1024	0 64 128 192 256 320 384 448 512 576 640 704 768 832 896 960 1024
Warps/Block		8	32	0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
Block Limit		8	32	0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
Registers				



he Kernel Profile - PC Sampling gives the number of samples for each source and assembly line with various stall reasons. The samples are collected at a period of 2048 [2^11] cycles.
eriod under Settings->Analysis tab. The allowed values are from 5 to 31. Increasing the period would reduce the number of samples collected.

IV. Milestone 4