

Analysis Report

cuda_DEEPS2D_Stage2(FlowNode2D<double, int=3>*, FlowNodeCore2D<double, int=3>*, unsigned long, unsigned long, unsigned long, double, double, int, double, double, ChemicalReactionsModelData2D*, int, double, double, double, double, double, FlowType, int, double*, int, unsigned int*, double, TurbulenceExtendedModel, SolverMode, int)

Duration	54,674 ms (54 674 145 ns)
Grid Size	[80050,1,1]
Block Size	[16,1,1]
Registers/Thread	114
Shared Memory/Block	0 B
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB
Shared Memory Bank Size	4 B

[1] GeForce GTX TITAN Z

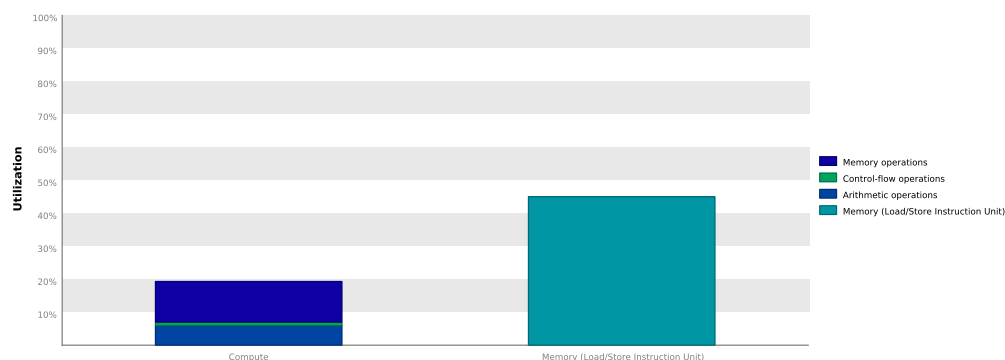
GPU UUID	GPU-bf84f461-f7a8-e964-a98f-3cc8237dbcb1
Compute Capability	3.5
Max. Threads per Block	1024
Max. Threads per Multiprocessor	2048
Max. Shared Memory per Block	48 KiB
Max. Shared Memory per Multiprocessor	48 KiB
Max. Registers per Block	65536
Max. Registers per Multiprocessor	65536
Max. Grid Dimensions	[2147483647, 65535, 65535]
Max. Block Dimensions	[1024, 1024, 64]
Max. Warps per Multiprocessor	64
Max. Blocks per Multiprocessor	16
Single Precision FLOP/s	5,043 TeraFLOP/s
Double Precision FLOP/s	1,681 TeraFLOP/s
Number of Multiprocessors	15
Multiprocessor Clock Rate	875,5 MHz
Concurrent Kernel	true
Max IPC	7
Threads per Warp	32
Global Memory Bandwidth	336,48 GB/s
Global Memory Size	5,941 GiB
Constant Memory Size	64 KiB
L2 Cache Size	1,5 MiB
Memcpy Engines	1
PCIe Generation	3
PCIe Link Rate	8 Gbit/s
PCIe Link Width	16

1. Compute, Bandwidth, or Latency Bound

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 below indicate that the performance of kernel "cuda_DEEPS2D_Stage2" is most likely limited by instruction and memory latency. You should first examine the information in the "Instruction And Memory Latency" section to determine how it is limiting performance.

1.1. Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GTX TITAN Z". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.



2. Instruction and Memory Latency

Instruction and memory latency limit the performance of a kernel when the GPU does not have enough work to keep busy. The performance of latency-limited kernels can often be improved by increasing occupancy. Occupancy is a measure of how many warps the kernel has active on the GPU, relative to the maximum number of warps supported by the GPU. Theoretical occupancy provides an upper bound while achieved occupancy indicates the kernel's actual occupancy. The results below indicate that occupancy can be improved by reducing the number of registers used by the kernel.

2.1. Instruction Latencies

Instruction stall reasons indicate the condition that prevents warps from executing on any given cycle. The following chart shows the break-down of stalls reasons averaged over the entire execution of the kernel. The kernel has low theoretical or achieved occupancy. Therefore, it is likely that the instruction stall reasons described below are not the primary limiters of performance and so should not be considered until any occupancy issues are resolved.

Texture - The texture sub-system is fully utilized or has too many outstanding requests.

Instruction Fetch - The next assembly instruction has not yet been fetched.

Memory Throttle - Large number of pending memory operations prevent further forward progress. These can be reduced by combining several memory transactions into one.

Not Selected - Warp was ready to issue, but some other warp issued instead. You may be able to sacrifice occupancy without impacting latency hiding and doing so may help improve cache hit rates.

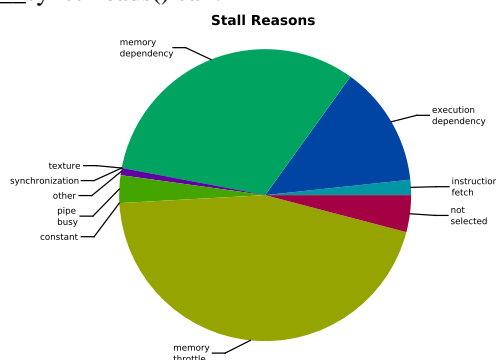
Execution Dependency - An input required by the instruction is not yet available. Execution dependency stalls can potentially be reduced by increasing instruction-level parallelism.

Memory Dependency - A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Data request stalls can potentially be reduced by optimizing memory alignment and access patterns.

Constant - A constant load is blocked due to a miss in the constants cache.

Pipeline Busy - The compute resource(s) required by the instruction is not yet available.

Synchronization - The warp is blocked at a `__syncthreads()` call.



2.2. GPU Utilization Is Limited By Register Usage

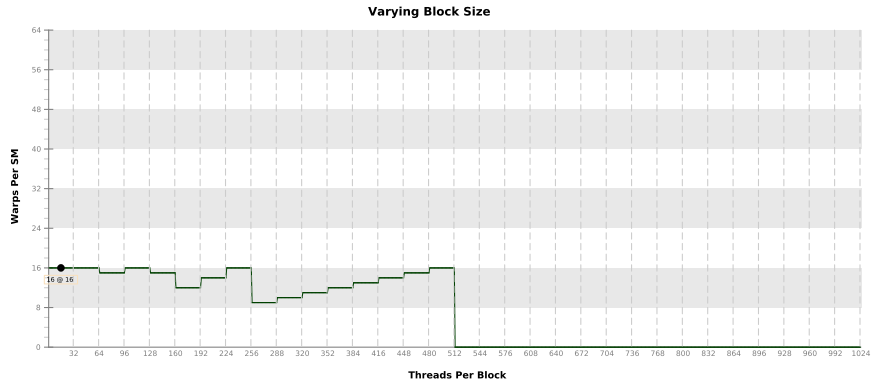
The kernel uses 114 registers for each thread (1824 registers for each block). This register usage is likely preventing the kernel from fully utilizing the GPU. Device "GeForce GTX TITAN Z" provides up to 65536 registers for each block. Because the kernel uses 1824 registers for each block each SM is limited to simultaneously executing 16 blocks (16 warps). Chart "Varying Register Count" below shows how changing register usage will change the number of blocks that can execute on each SM.

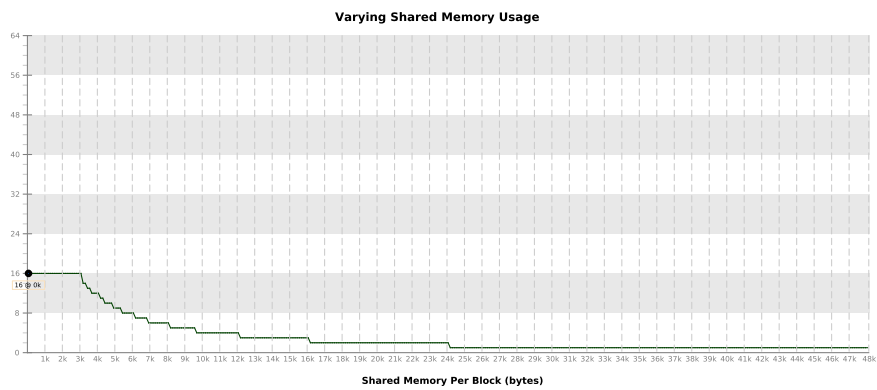
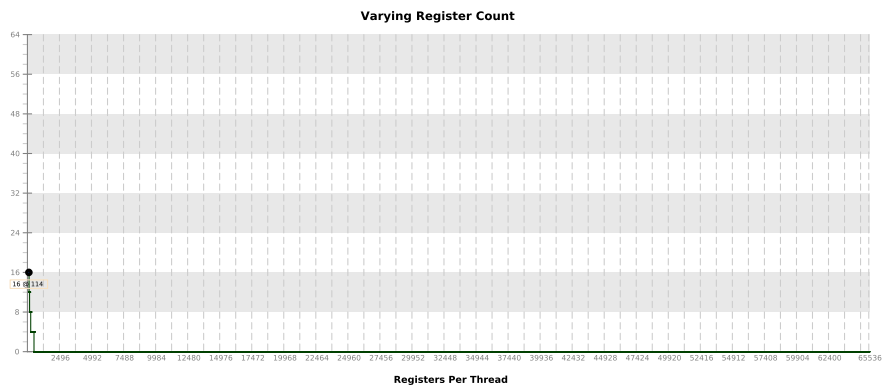
Optimization: Use the `-maxrregcount` flag or the `__launch_bounds__` qualifier to decrease the number of registers used by each thread. This will increase the number of blocks that can execute on each SM. On devices with Compute Capability 5.2 turning global cache off can increase the occupancy limited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

Variable	Achieved	Theoretical	Device Limit	Grid Size: [80050,1,1] (80050 blocks) Block Size: [16,1
Occupancy Per SM				
Active Blocks		16	16	
Active Warps	15,79	16	64	
Active Threads		512	2048	
Occupancy	24,7%	25%	100%	
Warps				
Threads/Block		16	1024	
Warps/Block		1	32	
Block Limit		64	16	
Registers				
Registers/Thread		114	65536	
Registers/Block		3840	65536	
Block Limit		16	16	
Shared Memory				
Shared Memory/Block		0	49152	
Block Limit			16	

2.3. Occupancy Charts

The following charts show how varying different components of the kernel will impact theoretical occupancy.





3. Compute Resources

GPU compute resources limit the performance of a kernel when those resources are insufficient or poorly utilized. Compute resources are used most efficiently when all threads in a warp have the same branching and predication behavior. The results below indicate that a significant fraction of the available compute performance is being wasted because branch and predication behavior is differing for threads within a warp.

3.1. Low Warp Execution Efficiency

Warp execution efficiency is the average percentage of active threads in each executed warp. Increasing warp execution efficiency will increase utilization of the GPU's compute resources. The kernel's maximum warp execution efficiency is 50% because the number of threads per block is not a multiple of the warp size.

Optimization: Reduce the amount of intra-warp divergence and predication in the kernel.

3.2. 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: Each entry below points to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence.

`/home/ss/Work/OpenHyperFLOW2D-GitHub/OpenHyperFLOW2D/branches/OpenHyperFLOW2D-CUDA-v2.01/libDEEPS2D/./
libOpenHyperFLOW2D/hyper_flow_node.hpp`

Line 486	Divergence = 2% [1600 divergent executions out of 80000 total executions]
Line 491	Divergence = 4% [3200 divergent executions out of 80000 total executions]
Line 523	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 533	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 577	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 582	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 597	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 683	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1092	Divergence = 0% [0 divergent executions out of 5306 total executions]
Line 1092	Divergence = 0% [0 divergent executions out of 4463 total executions]
Line 1092	Divergence = 0% [0 divergent executions out of 4463 total executions]
Line 1092	Divergence = 0% [0 divergent executions out of 4463 total executions]
Line 1092	Divergence = 0% [0 divergent executions out of 5306 total executions]
Line 1092	Divergence = 0% [0 divergent executions out of 5306 total executions]

`/home/ss/Work/OpenHyperFLOW2D-GitHub/OpenHyperFLOW2D/branches/OpenHyperFLOW2D-CUDA-v2.01/libDEEPS2D/
cuda_deep2d_kernels.cu`

Line 64	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 64	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 64	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 64	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 64	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 64	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 64	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 64	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 64	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 64	Divergence = 0% [0 divergent executions out of 80000 total executions]

```
/home/ss/Work/OpenHyperFLOW2D-GitHub/OpenHyperFLOW2D/branches/OpenHyperFLOW2D-CUDA-v2.01/libDEEPS2D/
cuda_deeps2d_kernels.cu
```

[illegible]

/home/ss/Work/OpenHyperFLOW2D-GitHub/OpenHyperFLOW2D/branches/OpenHyperFLOW2D-CUDA-v2.01/libDEEPS2D/
cuda_deeps2d_kernels.cu

Line 745	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 746	Divergence = 0% [0 divergent executions out of 320000 total executions]
Line 747	Divergence = 0% [0 divergent executions out of 240000 total executions]
Line 777	Divergence = 0% [0 divergent executions out of 320000 total executions]
Line 782	Divergence = 0% [0 divergent executions out of 320000 total executions]
Line 782	Divergence = 0% [0 divergent executions out of 320000 total executions]
Line 811	Divergence = 0% [0 divergent executions out of 560000 total executions]
Line 824	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 857	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 880	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 898	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 898	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 900	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 900	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 905	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 918	Divergence = 0% [0 divergent executions out of 50 total executions]

/usr/local/cuda-8.0/include/device_functions.hpp

Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1478	Divergence = 1,4% [4463 divergent executions out of 320000 total executions]
Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1478	Divergence = 0% [0 divergent executions out of 320000 total executions]
Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]

Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1478	Divergence = 0% [0 divergent executions out of 80000 total executions]
Line 1769	Divergence = 0% [0 divergent executions out of 80000 total executions]

3.3. Function Unit Utilization

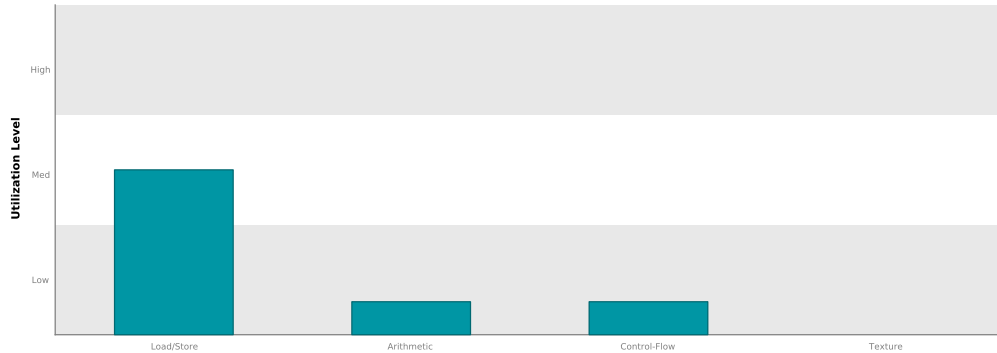
Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is not limited by overuse of any function unit.

Load/Store - Load and store instructions for local, shared, global, constant, etc. memory.

Arithmetic - All arithmetic instructions including integer and floating-point add and multiply, logical and binary operations, etc.

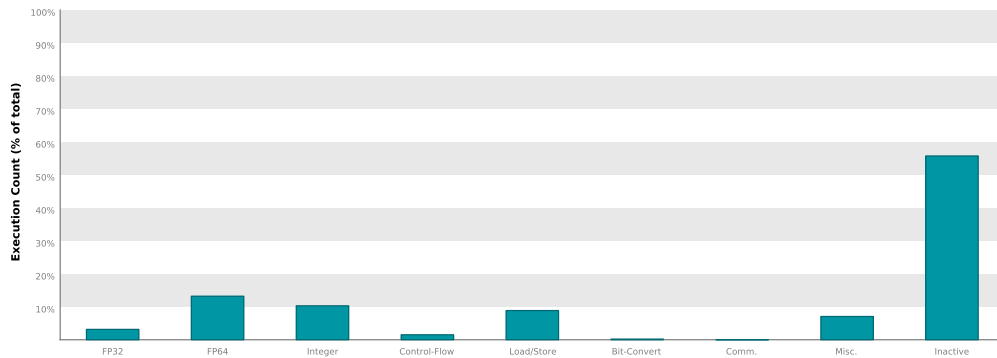
Control-Flow - Direct and indirect branches, jumps, and calls.

Texture - Texture operations.



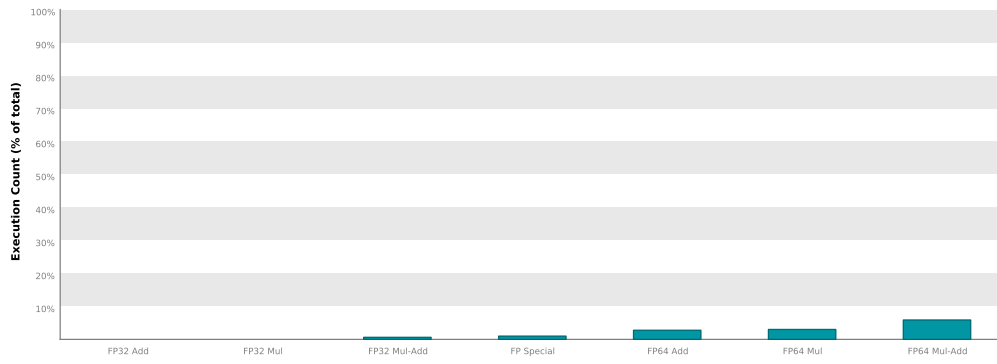
3.4. Instruction Execution Counts

The following chart shows the mix of instructions executed by the kernel. The instructions are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing instructions in that class. The "Inactive" result shows the thread executions that did not execute any instruction because the thread was predicated or inactive due to divergence.



3.5. Floating-Point Operation Counts

The following chart shows the mix of floating-point operations executed by the kernel. The operations are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing operations in that class. The results do not sum to 100% because non-floating-point operations executed by the kernel are not shown in this chart.








4. Memory Bandwidth

Memory bandwidth limits the performance of a kernel when one or more memories in the GPU cannot provide data at the rate requested by the kernel.

4.1. Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory.

Transactions	Bandwidth	Utilization	
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	164392752	95,225 GB/s	
Global Stores	136929600	79,317 GB/s	
Atomic	1340356	1,34 MB/s	
L1/Shared Total	302662708	174,544 GB/s	
L2 Cache			
L1 Reads	164392752	95,225 GB/s	
L1 Writes	136929600	79,317 GB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	2560000	741,446 MB/s	
Total	301322352	174,542 GB/s	
Texture Cache			
Reads	0	0 B/s	
Device Memory			
Reads	42893846	24,846 GB/s	
Writes	100088633	57,977 GB/s	
Total	142982479	82,823 GB/s	
System Memory			
[PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	0	0 B/s	
Writes	1	579 B/s	