Questions

1. What is control flow divergence?

In CUDA, threads work in warps of 32 and all of the threads in a warp execute the same task (instruction) concurrently. If threads in the same warp have to do different tasks, it is called control flow (warp) divergence. In NVIDIA graphical processing units, there are instructions that are executed if a conditional flag is true. All threads in CUDA execute all conditional branches so the cost doubles, leading to a loss of performance. Because of this, the NVIDIA compiler checks if all threads in a warp take the same conditional branch, and does warp voting. Every warp chooses the most efficient branch and in some cases, at compilation, all of the threads in a warp may go to the same branch. Different threads executing different warps and all threads going to the same branch causes a huge performance decrease.

1. How can we create a dynamic sized shared memory?

Dynamically allocated shared memory is used when the amount of shared memory I not known a priori. To allocate dynamic sized shared memory, an optional third configuration parameter is specified when calling a kernel function that uses dynamically sized shared memory:

someFunction<<<1, n, n\*sizeof(float)>>>(arr, n);

Dynamic sized shared memory is allocated as in the below example:

extern \_\_shared\_\_ int s[];

Here, extern specifies that the array is shared and the size of the array comes from the third configuration parameter, which is a size input that is calculated during compile time.

1. How can we use shared memory to accelerate our code?

Using shared memory is much faster than the global memory that is not cached. This is due to the fact that shared memory is allocated per thread block, so all of the threads in one block utilize the same memory, causing increased locality and controlled data caches, high performance cooperative parallel algorithms such as parallel reductions.

1. Which CUDA operations give us device properties? To answer this question you should write a simple program and query the device properties of the machine you are working with.

Code:

**int** **main**()

{

**int** device\_no;

//get device number

cudaGetDeviceCount(&device\_no);

//for each device find the props

**int** i, driverVersion, runtimeVersion;

**for**(i = 0; i < device\_no; i++)

{

cudaDeviceProp properties;

cudaGetDeviceProperties(&properties, i);

printf("Name of device %d: %s\n", i, properties.name);

cudaDriverGetVersion(&driverVersion);

cudaRuntimeGetVersion(&runtimeVersion);

printf("\tCUDA driver version: %d.%d\n", driverVersion/1000, (driverVersion%100)/10);

printf("\tCUDA runtime Version: %d.%d\n", runtimeVersion/1000, (runtimeVersion%100)/10);

printf("\tCUDA capability version number: %d.%d\n", properties.major, properties.minor);

printf("\tMemory clock rate (KHz): %.0f Mhz\n", properties.memoryClockRate \* 1e-3f);

printf("\tMemory bus width (bits): %d\n", properties.memoryBusWidth);

printf("\tPeak memory bandwidth: (GB/s): %f\n", 2.0\*properties.memoryClockRate\*(properties.memoryBusWidth/8)/1.0e6);

printf("\tTotal constant memory (bytes): %lu\n", properties.totalGlobalMem);

printf("\tTotal global memory: %.0f MBytes (%llu bytes)\n", (**float**)properties.totalGlobalMem/1048576.0f, (**unsigned** **long** **long**) properties.totalGlobalMem);

printf("\tMaximum shared memory available on a thread block (bytes): %lu\n", properties.sharedMemPerBlock);

printf("\tMaximum number of 32-bit registers on a thread block: %d\n", properties.regsPerBlock);

printf("\tWarp size: %d\n", properties.warpSize);

printf("\tMaximum number of threads per block: %d\n", properties.maxThreadsPerBlock);

printf("\tMaximum size of each dimension of a block: %d, %d, %d\n", properties.maxThreadsDim[0], properties.maxThreadsDim[1], properties.maxThreadsDim[2]);

printf("\tMaximum size of each dimension of a grid: %d, %d, %d\n", properties.maxGridSize[0], properties.maxGridSize[1], properties.maxGridSize[2]);

printf("\tClock Rate (KHz): %d\n\n", properties.clockRate);

} }

Code Output:

[gulsum@HPZ820 src]$ /usr/local/cuda-9.1/bin/nvcc deviceProps.cu -o deviceProps

[gulsum@HPZ820 src]$ ./deviceProps

Name of device 0: Tesla K20c

CUDA driver version: 9.1

CUDA runtime Version: 9.1

CUDA capability version number: 3.5

Memory clock rate (KHz): 2600 Mhz

Memory bus width (bits): 320

Peak memory bandwidth: (GB/s): 208.000000

Total constant memory (bytes): 4972937216

Total global memory: 4743 MBytes (4972937216 bytes)

Maximum shared memory available on a thread block (bytes): 49152

Maximum number of 32-bit registers on a thread block: 65536

Warp size: 32

Maximum number of threads per block: 1024

Maximum size of each dimension of a block: 1024, 1024, 64

Maximum size of each dimension of a grid: 2147483647, 65535, 65535

Clock Rate (KHz): 705500

Name of device 1: GeForce GTX 480

CUDA driver version: 9.1

CUDA runtime Version: 9.1

CUDA capability version number: 2.0

Memory clock rate (KHz): 1848 Mhz

Memory bus width (bits): 384

Peak memory bandwidth: (GB/s): 177.408000

Total constant memory (bytes): 1545469952

Total global memory: 1474 MBytes (1545469952 bytes)

Maximum shared memory available on a thread block (bytes): 49152

Maximum number of 32-bit registers on a thread block: 32768

Warp size: 32

Maximum number of threads per block: 1024

Maximum size of each dimension of a block: 1024, 1024, 64

Maximum size of each dimension of a grid: 65535, 65535, 65535

Clock Rate (KHz): 1401000

1. What are the necessary compiler options in order to use atomic operations?

There are several functions that allow atomic operations such as “atomicAdd()”, “atomicSub()”, “atomicMax()” and so on. These methods ensure that during the operation, there is no interference from other threads (so no race conditions happen). However, they do not ensure memory fencing and synchronization. The atomic functions can only be used in device functions. There are memory fence functions provided by CUDA that allow enforcement for the ordering of memory accesses. These functions are independent of the type of the memory (shared, global, peer device). There are also synchronization functions that ensure waiting until all the threads in a thread block or all of the threads have reached the point and makes memory accesses visible to all threads.