Advanced Computer Architecture (ACA)
GPU (CUDA) Vector Reduction

Overview:

Vector Reduction - In simple terms, the vector reduction operator calculates the sum of all components of a vector.

For example, for the following vector [1 5 10 2], the sum is 17. For small arrays, this computation may not seem very amendable to parallelization. However, if the array is large:

[1 5 10 2 2 4 9 4 2 3 5 7 9 7 5 3]

It can be broken down into 4 sums that are performed in parallel, and then the partial sums added to get the global sum:

[1 5 10 2] = 17
[2 4 9 4] = 19
[2 3 5 7] = 17
[9 7 5 3] = 24

Total sum is 77

You are to write the vector reduction that performs the partial sum generation using the GPU. Each “block” of execution on the GPU will represent a partial sum. In the example above, there were 4 blocks of work that divided equally the total amount of work. However, the block size could have been 8 elements creating 2 blocks worth of work. The characteristic of block size is not automatically determined by the number of elements, but instead relates to the performance of the machine. One important note is that if the 4 blocks of 4 elements are computed in parallel, there are still 4 partial sums that need to be added together. These partial sums need to either sent back to the CPU to be added in sequence, or if there are a sufficiently large number of partial sums, another GPU pass could add the numbers in parallel.

Vector reduction is covered in class lecture is great detail.

[Part A] Use the code skeleton to write the code for vector reduction. In order to do this, you must follow the seven steps: allocate GPU memory, allocate CPU memory, transfer data (the In array) between CPU and GPU, run kernel, transfer data (the Out array) back from GPU to CPU, and free the data on GPU and CPU.

Parts of the provided code are provided for vectorReduce.cu

Create a directory (and put the Makefile and vectorReduce.cu file) at:
NVIDIA_CUDA_SDK/C/src/

```
>> cd vectorReduce
change the code
>> make
```

remember that the binary program will be located at: ../../bin/linux/release/vectorReduce

You can run the program on various input sizes, such as 200000, and can also specify the blocksize.

```
vectorReduce -size 200000 –blocksize 32
```

The blocksize and kernel are integrated to allocate the amount of shared memory for each block by the following lines of code:

```
int sharedMemSize = threadsPerBlock * sizeof(float);

VecReduce<<<blocksPerGrid, threadsPerBlock, sharedMemSize>>>(d_In, d_Out, N);
```

The “sharedMemSize” is presented to the Kernel launch to allocate for each block.

[Part B] Use the timing routines to calculate the amount of time to perform the vector reductions (only the GPU calculation) and the memory transfer time. You must also account for any execution time of adding partial sums (performed by the CPU).   You will also experiment with different block sizes for performing the vector reduction: 32, 64, 128, 256. Note, when performing block size of 32, there will be more blocks and thus more partial sums. For block size of 256, it means that 256 elements are being condensed down into 1 partial sum.

Example: if you start with 1024 values, using block size of 32, the GPU will have 32 partial results (32 * 32 = 1024) and your code will need to transfer over the 32 results to be added in a loop. It is recommended that you have each block on the GPU set one output result in the Out array, making the transfer of 32 elements to the host array (h_Out) with one GPU to CPU transfer command. You would then run the code:

```
sum = 0;
for (i=0; i < 32; i++) {
    sum = sum + h_Out[i];
}
```

[Handin – Upload to the electronic system]

You must upload the following to blackboard. Your results will be given points based on clarity and following what is asked.

[Item 1] - For the assignment, submit the .cu file and the results of the following.

[Item 2] - Provide the execution time and memory transfer time for the following data sizes:

Input Size	Blocksize	GPU Execution Time	Memory Transfer Time	CPU Time (to add partial sums)	Overall Execution Time (Memory + GPU Execution)
1000	32				
10000	32				
100000	32				
1000000	32				
2000000	32				
1000	64				
10000	64				
100000	64				
1000000	64				
2000000	64				
1000	128				
10000	128				
100000	128				
1000000	128				
2000000	128				
1000	256				
10000	256				
100000	256				
1000000	256				
2000000	256				

[Item 3] – Graph the result using Excel or any plotting tool, clearly showing the overall execution time versus Input Size and Block Size.

[Item 4] – You must MODIFY the CUDA kernel to perform the following:

Kernel 1: Similar to the regular original CUDA reduction kernel, except each block should use the atomic support operations to add their local_block_sum to the collective total. In this case, only one thread of the block should perform the atomicAdd:

```
atomicAdd(d_Total, local_block_sum);
```

In the case of the reduction, the resulting answer will be a single value and require no CPU computation, but still needs to be transferred back to the CPU. Compare the performance to the previous item (2), include all memory transfer times:

Input Size	Blocksize	Previous total execution (CPU+GPU)	Total execution (atomic support in GPU)	Speedup
1000	32			
10000	32			
100000	32			
1000000	32			
2000000	32			
1000	64			
10000	64			
100000	64			
1000000	64			
2000000	64			
1000	128			
10000	128			
100000	128			
1000000	128			
2000000	128			
1000	256			
10000	256			
100000	256			
1000000	256			
2000000	256			

[Item 5] – You must examine the overhead of performing the runtime check within the kernel of

```
 if (globalid < N) {
… // work
  }
```

at runtime for values that are a multiple of the blocksize. In the previous experiments, you examined values 1000, 10000, etc. For this experiment, you will perform the following : 1024, 4096, 16384, 262144, 1048576. The difference is that these are multiples of the blocksize, so there is no need to include the if check, the work is always execute. The goal of this task is to determine the overhead that the if statement incurs for kernel execution. Note, in the case of N=1000 and blocksize of 32, the if statement overhead is incurred for the first 31 blocks [Block 0-30] even though the computation (work) is always entered, but the if statement matters only to the last block [block 31] where some of the threads of the last block execute and some do not.

Input Size	GPU Overall Execution Time (blocksize=32) with the if statement present	GPU Overall Execution Time (blocksize=32) without the if statement present	Percentage different in performance
1024			
4096			
16384			
262144			
1048576			