James Hahn
CS1645 High Performance Computing
Dr. Jaber Hasbestan
Homework #4

Algorithm presentation

As an example, let's say the integral is being calculated on the interval from 0 to 1. With 8 blocks, each block computes the integral on some interval of length 1 / 8 = .125 . So, block 0 would compute the integral on the interval [0, .125], block 1 computes [.125, .250], and so on until block 7 computes [.875, 1]; each interval is referred to as a chunk. Additionally, the user passes in a value **n** to determine the how granular each chunk will be. For example, if there are 8 chunks, but the user specifies **n = 32**, then each chunk will contain 32 / 8 = 4 parts. So, each block will further break down their interval. For example, block 0 will compute the integral over intervals [0, .03125], [.03125, .0625], [.0625, .09375], and [.09375, .125], further increasing the precision of our integral value. Furthermore, each block has a user-specified number of threads per block. In our above example, with 8 blocks, if the user specifies 4 threads per block, then each thread will compute the integral value for one part. The values of these parts undergo a global addition reduction to compute a chunk's integral value. Then, the chunks' integral values undergo an additional global addition reduction to compute the final integral value.

Please note there are maximums to the number of blocks and threads per block specified by the user. On the CRC, a GTX1080 GPU has a maximum number of 28 blocks and 1024 threads per block. As such, if a user over specifies these numbers, the program will exit due to lack of resources.

The program was compiled with the following command: *nvcc -arch=sm_60 integration.cu* .

Results

All results were computed for the same value of points, N, resulting in higher accuracy in the computation of pi. A higher number of blocks and threads per block decreases the execution time of computation while a higher value of N increases the accuracy of the integral.

All times were gathered using the CUDA events discussed in class. Furthermore, all time recordings were measured using an interactive session on CRC with the following command: *crc-interactive.py -g -t 1 -n 1 -p gtx1080 -c 1*. To gather specific times of execution, the program was ran with the command *nvprof ./a.out* and times were written down for the *calc_chunk* function (this is what the below times were measured from). Additionally, at the end of execution, using the above CUDA events, the time is output in milliseconds. All times were averaged over five simulations.

N = 2,000,000,000 points		
# of blocks	**# of total threads**	**Time (seconds)**
1	256	4.9591
1	512	4.7515
1	1024	4.7095
4	256	4.2012
4	512	2.0939
4	1024	1.2438
8	256	4.1170
8	512	2.1191
8	1024	1.0498
16	256	4.1111
16	512	2.0942
16	1024	1.0311

The results are satisfactory. As the number of blocks increases, the execution time decreases until we reach the difference between 8 and 16 blocks. At this point, the execution times remain fairly stagnant. This happens since the total number of threads never changes, but the number of threads per block does change. For example, with 8 blocks and 1024 threads, there are 128 threads assigned per block. With 16 blocks and 1024 threads, there are 64 threads assigned per block. I believe this is the tradeoff point in terms of blocks and threads per block. After this, execution times level off as the main concern slowing down execution is the amount of inter-block and inter-thread communication.

Additionally, the execution time decreases significantly as the number of total threads increases while the number of blocks remains constant. This makes sense since that's the whole point of multithreading; work is being split up on a more granular level.

I also tested the program with different numbers of threads per block. The results can be seen below:

N = 2,000,000,000 points		
# of blocks	**threads per block**	**Time (seconds)**
4	250	1.2786
4	500	1.2537
4	1000	1.2257
8	250	.64734
8	500	.63764
8	1000	.61105
16	250	.33888
16	500	.31547
16	1000	.30573

Clearly, if the number of threads per block increases for a given number of blocks, the execution time is slightly reduced. Additionally, we see the true power of increasing the number of blocks. There is a clear separation in the total execution time as the number of blocks increases while the number of threads per block remains stagnant.

Finally, as the number of points (n) increases, the accuracy of the integral increases. This is shown in the below table. The values converge toward pi with a decimal precision of 9 digits.

Number of points (n)	**Integral value**
1	3.00000000
2	3.10000000
4	3.13117647
8	3.13898849
16	3.14094161
32	3.14142989
64	3.14155196
128	3.14158248
256	3.14159011