CUDA Notes

Eric Andrews

April 8, 2018

CUDA Implementation

1 General Structure

The entire matrices to be multiplied are sent to the GPU, and a result matrix is created on the GPU. For each block, pointers to the start of the sub-matrices are sent to the GPU; each thread uses its thread id to compute its location in the matrix, then iterates over the blocks for its row and column, accumulating the sum of multiplications into a variable. This is then added to the result matrix. When the entire matrix multiplication is done, the result matrix is copied from the GPU back to the processor.

2 Known Issues

- Some fixed sizes of grids lead to componentwise errors; this suggests that there may be minor arithmetic errors in the code. However, with the variables as set, the program runs successfully up to 769×769 matrices.

3 Performance

3.1 Baseline performance

When run with no CUDA whatsoever (simply using a serial blocked implementation), the program obtains the following results:

Size	**Percentage**
31	5.56
32	6.18
127	5.59
128	5.01
511	4.25
512	2.07
Average	4.72

3.2 1 Block, 2d Block Layout

16 $\times$ Blocks, 16 $\times$ 16 Cuda Blocks

Size	**Percentage**
31	.83
32	.86
127	1.45
128	1.45
511	1.49
512	1.51
Average	1.39

32 $\times$ 32 Blocks, 32 $\times$ 32 Cuda Blocks

Size	**Percentage**
31	.94
32	1.05
127	1.62
128	1.62
511	1.69
512	1.69
Average	1.58

3.3 Multiple Blocks, 2d Block Layout

16 × 16 Blocks, 16 × 16 Cuda Blocks

Size	**Percentage**
31	.85
32	.99
127	1.46
128	1.49
511	1.46
512	1.52
Average	1.41

32 × 32 Blocks, 32 × 32 Cuda Blocks

Size	**Percentage**
31	.89
32	.99
127	1.61
128	1.63
511	1.69
512	1.69
Average	1.58

64 × 64 Blocks, 16 × 16 Cuda Blocks

Size	**Percentage**
31	1.37
32	1.91
127	14.77
128	14.94
511	22.47
512	22.94
Average	16.94

3.4 Fixed Block, Grid Sizes; 2d Layout

512 × 512 Blocks, 16 × 16 Cuda Blocks in a 32 × 32 Grid

Size	**Percentage**
31	1.49
32	1.82
127	17.95
128	18.44
511	38.95
512	39.02
Average	27.13

3.5 Performance Summary

The fastest runtimes were achieved by fixing as many variables as possible ahead of time with compiler directives; this is likely due to that reducing the amount of computation required per loop. Additionally, increasing the size of the outer layer dgemm blocks led to considerable speedup up to 512 × 512.

To achieve faster speeds, a shared memory approach could be utilized.

OpenACC Implementation

4 General Structure

The structure is basically the same as the naive matrix multiply with the addition of compiler directives to accelerate the outer two loops.

5 Performance

5.1 Baseline Performance

Naive Matrix Multiply

Size	**Percentage**
31	3.61
32	4.30
127	2.58
128	2.60
511	2.80
512	1.83
Average	2.733

Simple Acceleration

Size	**Percentage**
31	.93
32	1.06
127	11.60
128	11.31
511	17.33
512	19.93
Average	14.11

Simple Acceleration with Restricted A, B, C

Size	**Percentage**
31	1.12
32	1.17
127	19.54
128	19.12
511	42.53
512	42.46
Average	31.13

Simple Acceleration with Restricted A, B, C, and Tuned Vectors, Gangs

Size	**Percentage**
31	1.11
32	1.28
127	19.73
128	18.93
511	42.33
512	44.44
Average	30.83

5.2 Performance Summary

The best performance came with a vector length of 64 and 1,024 gangs, using the restricted keyword on matrices A, B, and C. Use of a reduction on the innermost loop saw no increase in performance.

Higher performance could possibly be achieved using a proper blocked implementation with optimized memory usage.