Balakrishnan A
CS20B012

CS2610 Lab 5 Report

Performance profiling with perf

Contents

Summary of the blocking algorithm

The cache blocking algorithm for matrix multiplication takes advantage of locality principles by multiplying blocks of matrices instead of the matrices as a whole. The matrices are split into blocks of different orders, which are powers of 2 (say, 4x4, 16x16, etc., or generally *b*x*b*), so we will have, for each matrix, $(n/b)^2$ blocks. The algorithm can be described in two layers: block multiplication and element multiplication. The block multiplication layer (the outer 3 loops in the C++ code) selects the blocks from matrices **a** and **b** to multiply, simultaneously deciding which block in **c** to store the values to. This is effectively the same as regular matrix multiplication, if we treat our blocks as elements of a regular matrix. The element multiplication layer does the actual element-to-element multiplication across these blocks. For every pair of multipliable blocks we choose from **a** and **b**, we pick multipliable pairs of elements (in the three internal loops) and multiply them and increment the appropriate element of **c** by the obtained value. This procedure exploits locality within the L1 cache as long as each currently-used block from **a**, **b**, **c** remains in the cache throughout the block-wise multiplication. Each block occupies $4b^2$ bytes of memory, hence we require $12b^2$ to be less than the L1 cache size (32 KB for the machine on which the code was executed). Hence we can expect *b* upto 32 or 64 to show visible improvement.

Observations

Upon analyzing the output obtained using perf (included in a text file within the ZIP file, summarized in 3 tables following the observations), the following were observed:

For matrix size 128, the total number of cache references and cache misses remained nearly the same (with a pretty large increase in misses for block size 64), but the L1-dcache miss rate reduced significantly using the blocked algorithm, reaching its minimum for block sizes of 16 and 32 (local miss rate of 0.16%).

For matrix sizes 256 and 512, a good deal of improvement is visible with lower block sizes (<=16) wrt L1-dcache miss rate, and furthermore, a huge reduction in cache references is also seen in the blocked algorithm against the naive one! This detail showing up here but not in 128-sized matrices could be attributed to the fact that the matrices **a**, **b**, **c** can almost be accommodated entirely in the L1 cache when the size of the matrices is 128 (memory required for each matrix = $(2^7)^2.4$ = 64 KB, whereas L1 cache size is 32 KB), which would prevent the naive algorithm from performing too badly since it can implement a good deal of locality by itself.

Considering L1-dcache misses as our benchmarking standard, ideal block size turns out to be any size <= 16 (though 16 doesn't perform as well as lower powers of 2 for 512, it is still a reasonable improvement over the naive algorithm), which matches closely with the prediction made at the end of the algorithm description.

Tables (values)

Matrix size: 128

Block size & Metric	0	4	8	16	32	64
Cache misses	58,689	42,833	51,650	66,966	56,112	89,941
Cache references	1,95,518	1,98,958	1,80,558	1,97,219	1,98,599	2,15,002
Cache miss rate	30.017%	21.529%	28.606%	33.955%	28.254%	41.833%
L1-dcache load misses	21,55,370	1,86,548	1,00,565	85,262	83,575	3,99,334
L1-dcache loads	4,86,72,059	5,61,88,622	5,30,77,639	5,17,99,811	5,12,18,394	5,09,46,732
L1-dcache stores	80,25,210	93,68,908	86,79,772	83,07,071	79,23,726	81,01,450
L1-dcache miss rate	4.43%	0.33%	0.19%	0.16%	0.16%	0.78%
L1-icache load misses	3,07,996	2,61,663	4,10,344	3,16,519	3,25,820	3,41,295

Matrix size: 256

Block size & Metric	0	4	8	16	32	64
Cache misses	1,15,680	1,18,507	80,955	1,09,994	89,003	66,634
Cache references	2,97,57,967	23,18,023	10,84,441	7,66,427	8,19,260	9,90,884
Cache miss rate	0.389%	5.112%	7.465%	14.352%	10.864%	6.725%
L1-dcache load misses	1,59,00,096	11,78,658	4,78,142	4,61,623	32,61,743	1,57,62,228
L1-dcache loads	34,86,53,939	40,24,79,679	37,25,63,672	36,42,87,031	36,01,53,918	36,36,21,853
L1-dcache stores	4,78,70,891	6,17,62,511	5,98,86,066	5,75,65,762	5,59,34,170	5,67,94,334
L1-dcache miss rate	4.56%	0.29%	0.13%	0.13%	0.91%	4.33%
L1-icache load misses	13,225	3,17,885	4,11,734	4,31,633	4,23,145	3,82,885

Matrix size: 512

Block size & Metric	0	4	8	16	32	64
Cache misses	4,13,693	4,36,642	4,19,514	2,43,635	3,70,614	3,66,016
Cache references	56,74,84,586	3,63,90,960	1,21,55,628	2,11,57,968	1,66,54,505	1,94,19,816
Cache miss rate	0.073%	1.200%	3.451%	1.152%	2.225%	1.885%
L1-dcache load misses	13,28,31,216	89,98,476	55,83,040	4,02,33,488	12,61,67,610	12,67,01,121
L1-dcache loads	2,67,08,29,600	3,11,00,23,247	2,90,15,26,102	2,81,29,50,177	2,81,79,86,947	2,77,10,44,535
L1-dcache stores	36,45,31,492	43,86,84,441	37,98,75,779	39,12,92,899	35,46,63,886	36,62,76,647
L1-dcache miss rate	4.97%	0.29%	0.19%	1.43%	4.48%	4.57%
L1-icache load misses	7,11,275	2,73,861	3,92,580	9,78,098	5,08,268	5,43,849