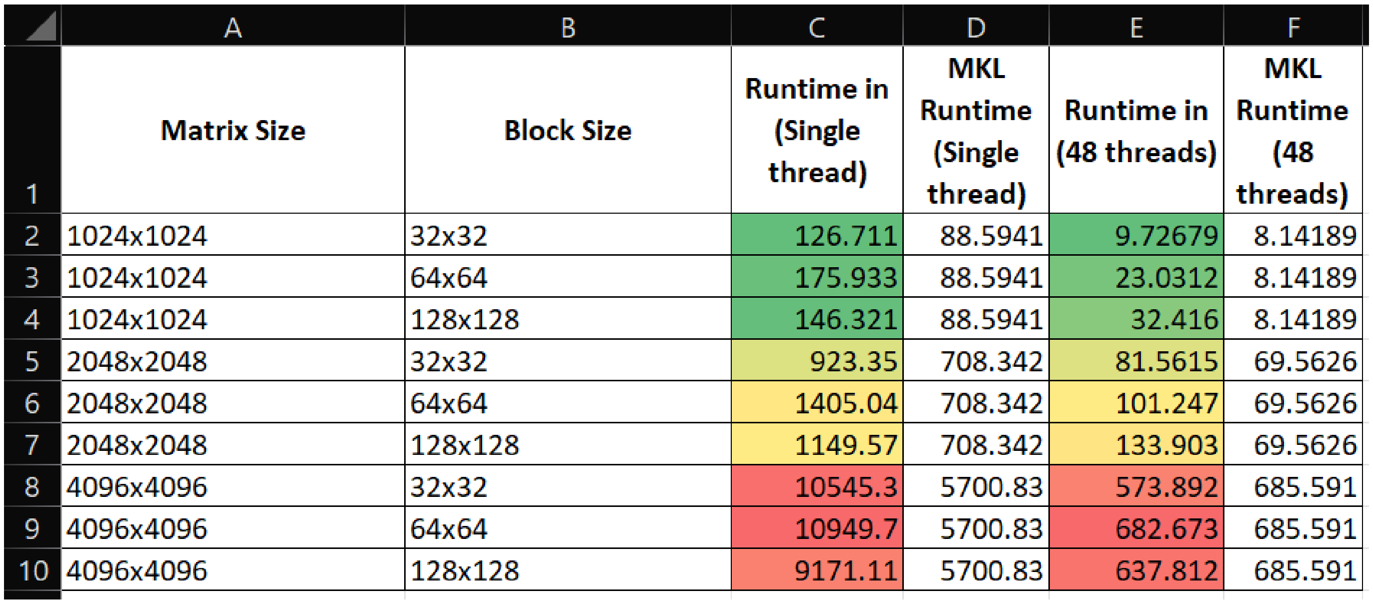
CS639: Homework 4 (git) Rajesh Shashi Kumar

1 Machine Configuration

Attribute	**Value**
Hostname	barolo.cs.wisc.edu
OS	Ubuntu 20.04.5 LTS
Compiler	g++ (gcc version 9.4.0), OpenMP version 4.5
Compile command	*(Makefile included in root directory of zip file)* *icc *.cpp -Wall -O3 -o conjugate_gradients_mkl -qopenmp -mkl*
CPU	AMD EPYC 7451 24-Core Processor
Cache configuration	L1-3MiB, L2-24MiB, L3-128MiB
Memory bandwidth	~150 GiB/s (Source,2). The machine has 256GB of memory spread across (8 out of 16) slots each housing 32GB stick. This information was retrieved using *sudo lshw -class memory*
Number of threads	24 cores * 2 threads/core = 48 threads
Dependencies	GCC, OpenMP, ICC (source /s/intelcompilers-2019/bin/iccvars.sh intel64)



	A	B	C	D	E	F
1	Matrix Size	Block Size	Runtime in (Single thread)	MKL Runtime (Single thread)	Runtime in (48 threads)	MKL Runtime (48 threads)
2	1024x1024	32x32	126.711	88.5941	9.72679	8.14189
3	1024x1024	64x64	175.933	88.5941	23.0312	8.14189
4	1024x1024	128x128	146.321	88.5941	32.416	8.14189
5	2048x2048	32x32	923.35	708.342	81.5615	69.5626
6	2048x2048	64x64	1405.04	708.342	101.247	69.5626
7	2048x2048	128x128	1149.57	708.342	133.903	69.5626
8	4096x4096	32x32	10545.3	5700.83	573.892	685.591
9	4096x4096	64x64	10949.7	5700.83	682.673	685.591
10	4096x4096	128x128	9171.11	5700.83	637.812	685.591

Fig 1. GEMM benchmarking sensitivity analysis across varying configurations

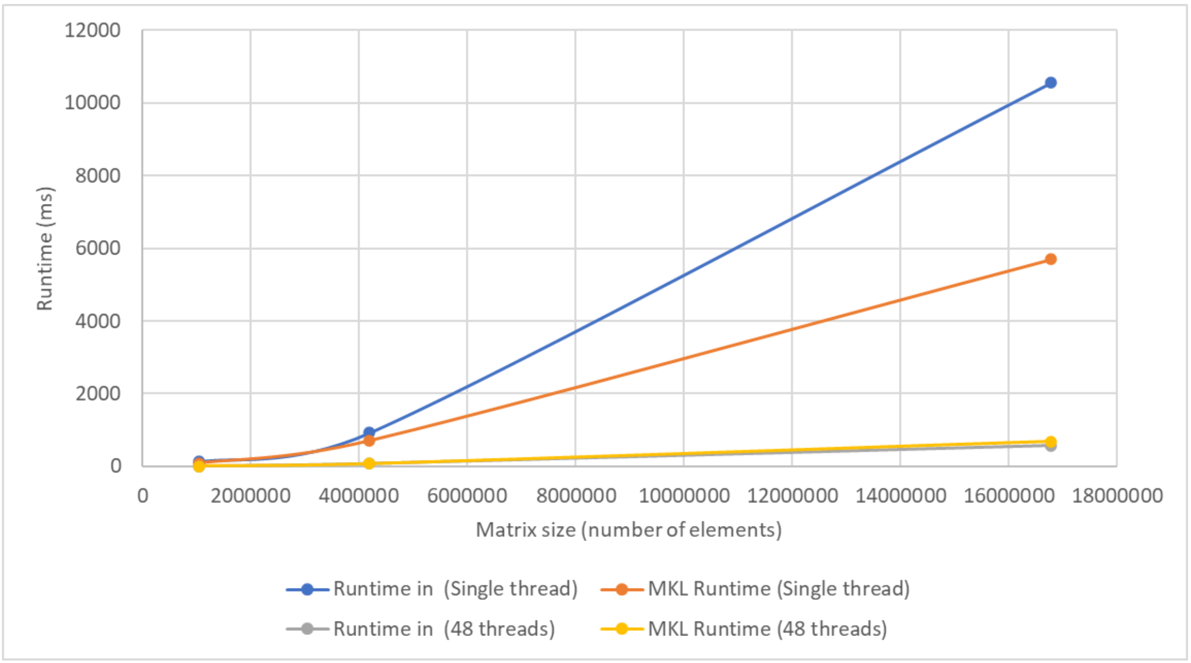


Fig 2. Graph plot showing performance scaling for 32x32 block size

Observations on the performance results:

- There is strong scaling in performance when we increase the number of threads. This indicates that the matrix sizes under observation here are insufficient to saturate the compute throughput of the underlying hardware.
- With increasing matrix sizes, the runtime increases linearly as seen in the graph plot. This is expected since the compute cores would need to fetch from deeper cache hierarchies to get the necessary data for the active blocks.
- Across all analysis, 32x32 seems to be the best performing block size. The approximate number of 4B float elements, ignoring side-effects such as replacement policy, that can be stored in the system I ran the experiments on is equal to :- **L1** -> 3*1024B = 768 elements ~ 26 x 26 block size.
  From the above number, it is evident that 32x32 is likely the best configuration since the

pre-requisite data elements are available close to the compute resources.

- Operating larger and larger matrix sizes seems to diminish the difference between the various block sizes. This is likely due to the fact that the L1 can no longer hold all the elements needed to compute a specific element in the output matrix beyond a certain size. The block size in this case only limits the amount of parallelism available and does not improve memory access latencies.
- As seen in Fig2, for the larger resolution (4096x4096), the 18x speed-up with larger number of threads is not quite proportional compared to the scaling in performance observed for 2048x2048 (11x) and 1024x1024 (14x). I found that it is slightly unclear as to what the reason for this could be from looking at the fraction of runtimes alone. It would need deeper profiling to disseminate the various components (memory and compute) contributing to the difference in performance.