

Analysis Questions:

1. Comparing the results of your basic and vectorized implementations at $N=16384$, which code has better performance in terms of MFLOP/s, and by how much? Which code has better memory system utilization, and by how much?

Assuming I am answering this correctly, it seems to be basic for both questions? It is by a fair bit for both questions, too. Especially vectorized.

2. Comparing the results of your basic and OpenMP 8-way parallel implementation at $N=16384$, which code has better performance in terms of MFLOP/s and by how much? Which code has better memory system utilization, and by how much?

Once again, it seems to be basic? The numbers for it are less, so I assume it is faster then, yes? It is better by a fair bit for both questions.

3. Looking at the results of your OpenMP implementation at $N=16384$, what is the speedup of this code going from 1 to 4 threads, from 1 to 16 threads, and from 1 to 64 threads? Use your runtime data to compute these speedup metrics.

Runtime (s)							
Problem size	Blas	Basic	Vectorized	omp-1	omp-4	omp-16	omp-64
1024	0.00015	0.00362	0.00028	0.00101	0.00046	0.00049	0.00045
2048	0.00072	0.01516	0.00139	0.00410	0.00242	0.00182	0.00202
4096	0.00428	0.05870	0.00572	0.01659	0.00519	0.00538	0.00447
8192	0.01857	0.23640	0.02393	0.07297	0.02050	0.01779	0.01667
16384	0.07634	0.98851	0.08755	0.26837	0.07859	0.07037	0.06970

MFLOPs							
Problem size	Blas	Basic	Vectorized	omp-1	omp-4	omp-16	omp-64
1024	14352.06 198	579.7510 4	7603.749 03	2070.852 18	4512.459 47	4253.685 45	4646.058 89

Omar Dajani
 Novemeber 14th, 2023

2048	11667.373 22	553.1900 0	6052.395 46	2043.872 83	3466.859 75	4614.746 64	4159.600 45
4096	7832.367 00	571.6014 1	5868.460 66	2022.634 85	6459.151 31	6241.613 21	7501.026 53
8192	7227.283 78	567.7448 9	5607.901 60	1839.326 56	6548.712 45	7546.208 69	8049.665 27
16384	7032.780 82	543.11278	6131.999 16	2000.464 07	6831.281 42	7629.625 06	7702.666 31

Memory Bandwidth (% memory bandwidth utilized)							
Problem size	Blas	Basic	Vectorize d	omp-1	omp-4	omp-16	omp-64
1024	0.02737	0.00111	0.01450	0.00395	0.00861	0.00811	0.00886
2048	0.01113	0.00053	0.00577	0.00195	0.00331	0.00440	0.00397
4096	0.00373	0.00027	0.00280	0.00096	0.00308	0.00298	0.00358
8192	0.00172	0.00014	0.00280	0.00044	0.00156	0.00180	0.00192
16384	0.00084	0.00006	0.00073	0.00024	0.00081	0.00091	0.00092

Omar Dajani
November 14th, 2023

The screenshot displays the Godbolt compiler explorer interface. On the left, the C++ source code for a function `my_dgemv` is shown. The code includes comments and a loop that performs a dot product of a row of matrix `A` with vector `x` to update an element in vector `y`. The right pane shows the assembly output for `x86-64 gcc 11.2` with optimization flags `-O2 -march=zvnr1 -mprefer-vector-width=256 -f`. The assembly includes instructions for loading the row of `A`, calculating the dot product using `vdotq`, and updating the vector `y`. The interface also shows the compiler explorer's navigation bar with options like 'Output', 'Filters', 'Libraries', 'Overrides', and 'Add new...'. The bottom status bar indicates the output is for `x86-64 gcc 11.2` with 575ms execution time and 678 lines filtered.

The screenshot displays the Godbolt compiler explorer interface. On the left, the C++ source code for a vectorized matrix-vector multiplication is shown. The code defines a function `dgemv` that takes a matrix `A`, a vector `x`, and a scalar `y` as input. It uses a `for` loop to iterate over rows and another `for` loop to iterate over columns, performing the calculation `y[row] += A[offset + column] * x[column];`. The right pane shows the assembly output for x86-64 gcc 11.2, with optimization flags `-O2 -march=znnv1 -mprefer-vector-width=256 -f`. The assembly includes instructions for loading, comparing, and adding values, as well as memory access instructions like `vfmovsd` and `vfmaddsd`. The output is filtered to show only the relevant assembly blocks.