HW3 Report

FLOPS = (sockets) x (cores per socket) x (cycles per second) x (FLOPS per cycle)

The processor of Laptop: intel core i7-1165G7(Products formerly Tiger Lake)

Using the below website, we got the result of the processor used.
https://www.intel.com/content/www/us/en/products/sku/208921/intel-core-i71165g7-processor-12m-cache-up-to-4-70-ghz-with-ipu/specifications.html

System: Ubuntu 20.0.4 LTS
Core: 4, sockets: 1
cycles per second: 4.70 GHz
FLOPS per cycle: 32 or 64

Source Code Benchmarking:

[1] Mode: FLOPS

Mode	Type	Size	Threads	Measured Time	Measured Throughput (MFLOPS)	Theoretical Throughput (MFLOPS)	Efficiency
flops	single	small	1	0.000238	42016.80672	601600	6.98%
flops	single	small	2	0.000225	44444.44444	300800	14.78%
flops	single	small	4	0.000302	33112.58278	150400	22.02%
flops	single	medium	1	0.000208	480769.2308	601600	79.92%
flops	single	medium	2	0.000203	492610.8374	300800	163.77%
flops	single	medium	4	0.000235	425531.9149	150400	282.93%
flops	single	large	1	0.000186	5376344.086	601600	893.67%
flops	single	large	2	0.00017	5882352.941	300800	1955.57%
flops	single	large	4	0.000211	4739336.493	150400	3151.15%
flops	double	small	1	0.000104	96153.84615	601600	15.98%
flops	double	small	2	0.000114	87719.29825	300800	29.16%
flops	double	small	4	0.000178	56179.77528	150400	37.35%
flops	double	medium	1	0.00017	588235.2941	601600	97.78%
flops	double	medium	2	0.000137	729927.0073	300800	242.66%
flops	double	medium	4	0.000161	621118.0124	150400	412.98%
flops	double	large	1	0.000096	10416666.67	601600	1731.49%
flops	double	large	2	0.000321	3115264.798	300800	1035.66%
flops	double	large	4	0.000192	5208333.333	150400	3462.99%

[2] Mode: Matrix multiplication

Mode	Type	Size	Threads	Measured Time	Measured Throughput (GFLOPS)	Theoretical Throughput (GFLOPS)	Efficiency
matrix	single	small	1	0.361490	2.576344	601.6	0.43%
matrix	single	small	2	0.186147	5.003157	300.8	1.66%
matrix	single	small	4	0.102351	9.099301	150.4	6.05%
matrix	single	medium	1	24.246115	2.458317	601.6	0.41%
matrix	single	medium	2	13.632931	4.372108	300.8	1.45%
matrix	single	medium	4	7.416723	8.036520	150.4	5.34%
matrix	single	large	1	1643.6125	2.320923	601.6	0.39%
matrix	single	large	2	898.358115	4.246299	300.8	1.41%
matrix	single	large	4	517.127526	7.376705	150.4	4.9%
matrix	double	small	1	0.326455	2.852836	601.6	0.47%
matrix	double	small	2	0.164449	5.663291	300.8	1.88%
matrix	double	small	4	0.093704	9.938984	150.4	6.61%
matrix	double	medium	1	19.306729	3.087247	601.6	0.51%
matrix	double	medium	2	10.114120	5.893211	300.8	1.96%
matrix	double	medium	4	5.805869	10.266274	150.4	6.83%
matrix	double	large	1	1254.490981	3.040833	601.6	0.51%
matrix	double	large	2	709.52101	5.376440	300.8	1.79%
matrix	double	large	4	436.76006	8.734080	150.4	5.81%

Conclusion of source code benchmarking:

If we want to do cache-friendly performance. We need to consider a variety of optimizations, and we can use the following options:

1. compiler flags - **-O3** can optimize all of the program.(done for Makefile)
2. unrolling - modern processor have multiple pipeline (Because of compiler flags, it is not useful.)
3. divide matrix properly: Do block matrix operations. The code has this method, transpose the matrix 2, and then partition the matrix leveraging block matrix concept and calculate the resultant matrix. We test a lot of block size(ex: 2, 4, 8, 16, 32, 64), finally **16 is a best block size** for our test hardware.
4. dedicated instruction - using AVX, SIMD instruction. We can add **`<immintrin.h>`** to implement **FMA (fuse multiply add)** to optimize it. We have done used intel instruction set to optimize `double` matrix multiplication, it can increase some performance.

Note: We tried to use SIMD registers (based on Intel AVX instruction set) and cache line in our implementation, but we were not able to do that successfully in the timeframe. We have included our incomplete header file (matrixmul.h) which contains the abstractions of the matrix multiplication we tried to implement using the SIMD. We still want to complete this part and we plan to do this along with the Intel API implementation.

Still consider the submitted optimized code and this report as our formal submission of this assignment.

HPL Benchmarking:

Using the https://www.mgaillard.fr/2022/08/27/benchmark-with-hpl.html, we installed and compiled the HPL benchmark.

Post the compilation, we had to edit the HPL.dat file and tune it as per our requirement and hardware. The hardware details are mentioned at the top of this report.

We tried to implement shpl, but were not able to do that. The professor also mentioned in the class, if you were not able to do that. Skip the first 3 lines of HPL benchmark table.

To tune the HPL.dat file, we used https://netlib.org/benchmark/hpl/tuning.html. We tuned the below listed parameters:

Line 3 [filename] : This line is used to change the name of the hpl output file name. We use “HPL-Result.out” as our file name.
Line 4 [output] : This line is used to define where we want our output. Since we want it in a file, we used an arbitrary value as 3.
Line 5 [N] : This line is used to define the number of problems we want to execute. In our case, we have 3 problems of various sizes and hence we denoted N as 3 in this line.
Line 6 [Ns] : This line is used to define the size of each problem and hence number of parameters in this line should be equal to N. In our example, we denoted Ns as “1024 4096 16386”.
Line 7 [#NBs] : This line is used to define the number of blocks we want to use to execute our problem of size Ns. In our case, we have tried to solve the 3 problems of various sizes using 3 different blocks and we finally choose the block which gives the best result.
Line 8 [NBs] : This line is used to define the block size of each block and hence number of parameters in this line should be equal to #NBs. In our example, we denoted NBs as “256 1024 2048”.
Line 9 [MPI mapping] : Since, row major mapping is recommended, we use 0 for this line to get a row major mapping.
Line 10 [PxQ] : This line is used to define the number of process grids that we want to use. Since, we want to use 1 process grid on 1 node. We provide a value of 1 for this line.
Line 11 [P] and Line 12 [Q] : These two lines work in succession, we have to provide the number of rows and columns of the grid we want to run on. We provide a value of 2 for P and 2 for Q i.e. we want xhpl to run on 1 process grid namely 2-by-2.

Below are the results for xhpl mode HPL benchmarking.

Mode	Type	Size	Threads	Measured Time	Measured Throughput	Theoretical Throughput	Efficiency
xhpl	double	1024	4	0.16	4364.2	150400	2.9%
xhpl	double	4096	4	2.40	19090.0	150400	12.69%
xhpl	double	16386	4	38.03	77141.0	150400	51.29%

Conclusion of HPL benchmarking:

Based on the hardware, we have got the following results.

1. Using the block size NB of 256, we get the best time result for problem size of N=1024.
2. Using the block size NB of 1024, we get the best time result for problem size of N=4096.
3. Using the block size NB of 1024, we get the best time result for problem size of N=16386.

We noticed the pattern that when the block size was of the N/4, the results were better till the problem size of N=4096. We tried the same with the problem size of N=16386, but the optimal result was obtained at NB=1024 rather than NB=4096.