CS 211 High Performance Computing

Project 1

Name: Kushal Shah

SID: 861186346

Part #1 & Part #2 dgemm0, dgemm1 & dgemm2

Clock frequency of Computer = 2GHz

Additional Delay = Delay to access the memory = 100 cycles / memory accesses

Matrix Size = n = 1000

So, Cycle time = $1/(2*10^9)$ = 0.5 ns

dgemm0:

This algorithm has three load operations (one each for A, B, C matrices) from memory and one store operation to the memory (for matrix C).

Here this 4 load and store operations occurs n^3 times.

Hence, Total memory access time = $(4n^3)$*(Cycle Time)*(Additional Delay)

= $4*1000*1000*1000*0.5*(10^{-9})*100$

= 200 seconds

Each iteration has 2 floating point iteration (one each for multiplication and summation) and innermost loop iterates n^3 times.

Also in each cycle CPU performs 4 double floating-point operations.

Hence, Total floating point operations time = $2*(n^3)$ *(Cycle Time)/4

= $2*1000*1000*1000*0.5*(10^{-9})/4$

= 0.25 seconds

Time taken to complete the **dgemm0** is,

= 200 + 0.25 = **200.25 seconds**

Time wasted in accessing operands that are not in register

= (Total time – Total floating point operations time) / Total Time

= (200.25 - 0.25) / 200.25 = 0.9988

= 99.88% time is wasted

dgemm1:

This algorithm has two load operations (one each for A, B matrices) in innermost loop which iterates n^3 times and two store operation to the memory (for matrix C) which iterates n^2 times.

Hence, Total memory access time = $(2n^3 + 2n^2)$*(Cycle Time)*(Additional Delay)

= $(2*1000*1000 + 2*1000*1000)*0.5*(10^{-9})*100$

= 100.1 seconds

Each iteration has 2 floating point iteration (one each for multiplication and summation) and innermost loop iterates n^3 times.

Also in each cycle CPU performs 4 double floating-point operations.

Hence, Total floating point operations time = $2*(n^3)$ *(Cycle Time)/4

= $2*1000*1000*1000*0.5*(10^{-9})/4$

= 0.25 seconds

Time taken to complete the **dgemm1** is,

= 100.1 + 0.25 = **100.35 seconds**

Time wasted in accessing operands that are not in register

= (Total time – Total floating point operations time) / Total Time

= (100.35 - 0.25) / 100.35 = 0.9975

= 99.75% time is wasted

Execution Time:

Matrix Size (n)	dgemm0 (in milliseconds)	dgemm1 (in milliseconds)	dgemm2 (in milliseconds)
64	7.128216	4.931031	1.803537
128	39.969206	25.759388	8.4104
256	325.499361	166.953458	54.454552
512	3049.546593	2179.83658	783.193455
1024	31205.88699	23135.24844	8132.992738
2048	420966.9449	285941.5489	151813.1982

Performance:

Matrix Size (n)	dgemm0 (in GFLOPS)	dgemm1 (in GFLOPS)	dgemm2 (in GFLOPS)
64	0.073551082	0.106324215	0.290699886
128	0.85059372	0.162826229	0.498704461

256	0.103086015	0.200980755	0.616191499
512	0.08802471	0.123144762	0.342744764
1024	0.068816619	0.092823021	0.264045932
2048	0.040810494	0.060081752	0.11316453

Correctness:

The maximum difference of all matrix elements between the results obtained from the three algorithms:

Matrix Size	dgemm0 & dgemm1	dgemm0 & dgemm2
64	0.000000	0.000000
128	0.000000	0.000000
256	0.000000	0.000000
512	0.000000	0.000000
1024	0.000000	0.000000
2048	0.000000	0.000000

dgemm0 & dgemm1: If we use the register for matrix C then we remove unnecessary load and store in the inner most loop. Which reduces the execution time of algorithm and increases the performance as seen from the Execution time and Performance table.

Here we can see that the results obtained from both algorithms are same which proves its correctness of implementation.

Now in **dgemm2** we further expanded our approaches to reuse the every element loaded from A and B matrices in dgemm1 during each iteration. Here we iterates to 2x2 sized block in each iteration. So, every elements loaded from A and B matrices were used twice inside each iteration of k loop and every element of C matrix is used n times in the k-loop.

So performance of dgemm2 is superior to the dgemm0 and dgemm1.

Part#3: dgemm3

Here we can optimize the matrix multiplication if in dgemm2 we can use wider tile size. In dgemm2 I have used 2x2 sized block. Now If we can use the 3x3 tile size then it can be faster.

The constraint to increase block size beyond 3x3 is number of floating point register available. The dgemm3 algorithm I have implemented uses the 15 floating point registers, available with each core.

Note: The another care should be taken is “the boundary condition” when matrix size is not multiple of 3. This can be achieved by appending zero to the matrix to make it in multiple of 3. I haven’t take care of boundary condition. Instead I have used that matrix size that can work for all 4 implementation and compare their results.

Execution Time:

Matrix Size (n)	dgemm0 (in milliseconds)	dgemm1 (in milliseconds)	dgemm2 (in milliseconds)	dgemm2 (in milliseconds)
66	6.621722	4.816092	1.693514	0.759508
132	34.817608	25.360189	8.634621	6.476681
258	321.320101	198.58523	67.098006	45.724277
516	2348.836613	1585.794235	612.967972	435.972731
1026	18372.79847	13365.86185	4980.412193	2743.350511
2052	219459.9502	123244.001	101395.2948	39827.554831

Performance:

Matrix Size (n)	dgemm0 (in GFLOPS)	dgemm1 (in GFLOPS)	dgemm2 (in GFLOPS)	dgemm2 (in GFLOPS)
66	0.08683421	0.119389746	0.33952598	0.757058517
132	0.132115222	0.181384137	0.532731662	0.710230441
258	0.106893481	0.172958603	0.511893364	0.751176973
516	0.116983953	0.173273547	0.448271695	0.630260043
1026	0.117570067	0.16161256	0.433717345	0.787391601
2052	0.078742063	0.140215581	0.170429301	0.433888781

Correctness:

The maximum difference of all matrix elements between the results obtained from the three algorithms:

Matrix Size	dgemm0 & dgemm1	dgemm0 & dgemm2	dgemm0 & dgemm3
66	0.000000	0.000000	0.000000
132	0.000000	0.000000	0.000000
258	0.000000	0.000000	0.000000
516	0.000000	0.000000	0.000000
1026	0.000000	0.000000	0.000000
2052	0.000000	0.000000	0.000000

How to Compile & Run:

gcc –lrt assignment1v3.c –o assignment1v3

qsub scriptv1.sub

What is in scriptv1.sub file?

#!/bin/sh
#PBS -l nodes=1:ppn=1,walltime=03:00:00

```
#PBS -N Assignment1
#PBS -M kshah016@ucr.edu
#PBS -m abe

module purge
module load gcc-4.6.2
module load mvapich2-1.8/gnu-4.6.2

cd $PBS_O_WORKDIR

./assignment1v3 64 > assignment1v3.txt
./assignment1v3 128 >> assignment1v3.txt
./assignment1v3 256 >> assignment1v3.txt
./assignment1v3 512 >> assignment1v3.txt
./assignment1v3 1024 >> assignment1v3.txt
./assignment1v3 2048 >> assignment1v3.txt
```

Where to find output:

In file: assignment1v3.txt

Note: Running for matrix size not in multiple of 3 may give non-zero answer in correctness results.