Project 1

FEBRUARY 6

High Performance Computing

Authored by: Kayhan Gavahi
Instructor: Dr. Dingwen Tao

THE UNIVERSITY OF ALABAMA

Contents

Tables and Figures

Part #1.

Assume your computer is able to complete 4 double floating-point operations per cycle when operands are in registers and it takes an additional delay of 100 cycles to access any operands that are not in registers. The clock frequency of your computer is 2 Ghz. How long it will take for your computer to finish the following algorithm ***dgemm0*** and ***dgemm1*** respectively for n= 1000? How much time is wasted on accessing operands that are not in registers? Implement the algorithm ***dgemm0*** and ***dgemm1*** and test them on your machine with n= 64, 128, 256, 512, 1024, 2048. Measure the time spend in the triple loop for each algorithm. Calculate the performance (in Gflops) of each algorithm. Performance is often defined as the number of floating-point operations performed per second. A performance of 1 GFLOPS means 1 billion of floating-point operations per second. You must use the system default compiler to compile your program. Your test matrices have to be 64-bit double floating point random numbers. Report the maximum difference of all matrix elements between the two results obtained from the two algorithms. This maximum difference can be used as a way to check the correctness of your implementation.

```
 /*dgemm0: simple ijk version triple loop algorithm*/
for (i=0; i<n; i++)
          for (j=0; j<n; j++)
                    for (k=0; k<n; k++)
                              c[i*n+j] += a[i*n+k] * b[k*n+j];

/*dgemm1: simple ijk version triple loop algorithm with register reuse*/
for (i=0; i<n; i++)
          for (j=0; j<n; j++) {
                    register double r = c[i*n+j] ;
                    for (k=0; k<n; k++)
                              r += a[i*n+k] * b[k*n+j];
                    c[i*n+j] = r;
}
```

Solution:

For ***dgemm0*** we have: operands a, b and c are not in the register, the code needs to load these from the memory and after that store c into the memory again. Therefore, there will be 4 load/stores and hence 4 delays. Two floating-point operation are executed in the innermost loop (K-loop) and since our PC complete 4 operations per cycle, each loop will take half a cycle. Thus:

$$t_0 = \frac{\left(4\times100+\frac{2}{4}\right)\times n^3}{clock\ frequency} = \frac{\left(4\times100+\frac{2}{4}\right)\times1000^3}{2\times10^9} = 200.25\ sec$$

And the wasting time is:

$$\frac{(4\times100)\times1000^3}{2\times10^9} = 200\ sec$$

$$Percent\ wasted = \frac{200}{200.25}\times100 = 99.8752\ \%$$

This means 98.8752 percent of the execution time has been due to the loading/storing from memory. This shows the importance of the register usage in order to reduce delays.

For dgemm1, we already have c in the register. So, there will be two delays in the k-loop and two in the j-loop, therefore:

$$t_1 = \frac{\left(2\times100+\frac{2}{4}\right)\times n^3 + (2\times100)\times n\text{^}2}{clock\ frequency}$$
$$= \frac{200.5\times1000^3 + 200\times1000\text{^}2}{2\times10^9} = 100.35\ seconds$$

And the wasting time is:

$$\frac{(2\times100)\times1000^3 + (2\times100)\times1000^2}{2\times10^9} = 100.1\ sec$$

$$Percent\ wasted = \frac{100.1}{100.35}\times100 = 99.7509\ \%$$

To calculate the elapsed time for each algorithm two method have been used in this project. In the method 1, the elapsed time is determined using the *clock()* command, which has the ability to gives us the execution time in milliseconds. Method 2, on the other hand, uses the chrono library, which uses the steady clock. Therefore, choro has higher precision and gives us the execution time in nanoseconds.

Table 1. Execution time of ***dgemm0*** *and* ***dgemm1*** *algorithms.*

	n=66	n=126	n=252	n=510	n=1020	n=2052	n=4050
Dgemm1 (ms)	0.000	5.002	41.111	386.027	4529.083	55,208.935	No time
Dgemm0(ms)	1.0300	7.018	64.170	586.562	7602.259	91,518.561	No time

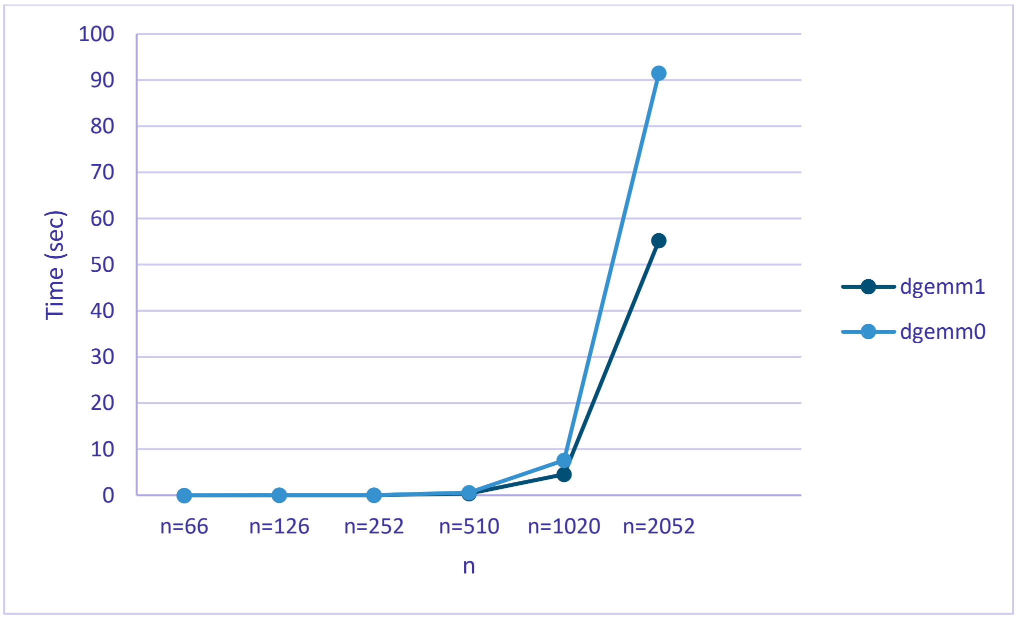


Figure 1. Execution Time for dgemm1 and dgemm0 algorithms.

Both algorithms perform $2 \times n^3 = 2 \times 10^9$ floating-point operations in total. To calculate the Flops, this number should be divided by the execution time of each algorithm provided in Table 1.

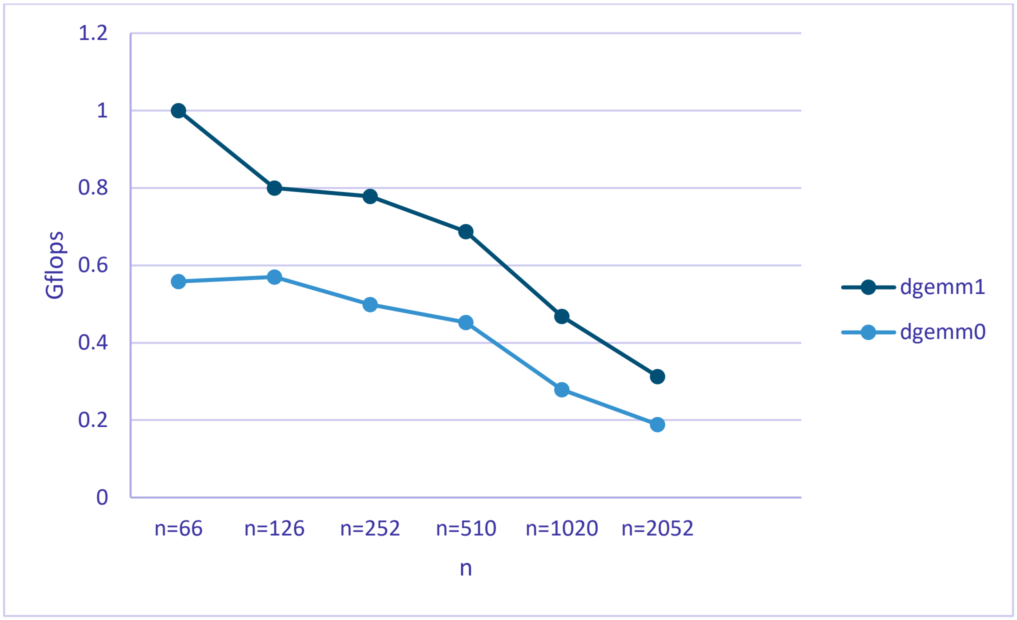


Figure 2. GFlops of dgeem1 and dgemm0 algorithms.

*Table 2. GFlops of **dgemm0** and **dgemm1** algorithms.*

	n=66	n=126	n=252	n=510	n=1020	n=2052	n=4050
Dgemm1	inf	0.7998	0.7785	0.6872	0.4686	0.3130	No time
Dgemm0	0.5582	0.5700	0.4987	0.4523	0.2791	0.1888	No time

**all the values are in GFO/s*

Considering the GFlops values in Table 2., GFlops is decreasing by increasing n (why?)

To check the correctness of each algorithm, two 3×3 matrices with random values were multiplied and the resulted matrix c was compared with the results of ***dgemm0*** and ***dgemm1.*** They matched completely. Also, the maximum difference of all matrix elements between the two results obtained from the ***dgemm0*** and ***dgemm1*** for n=64 is:

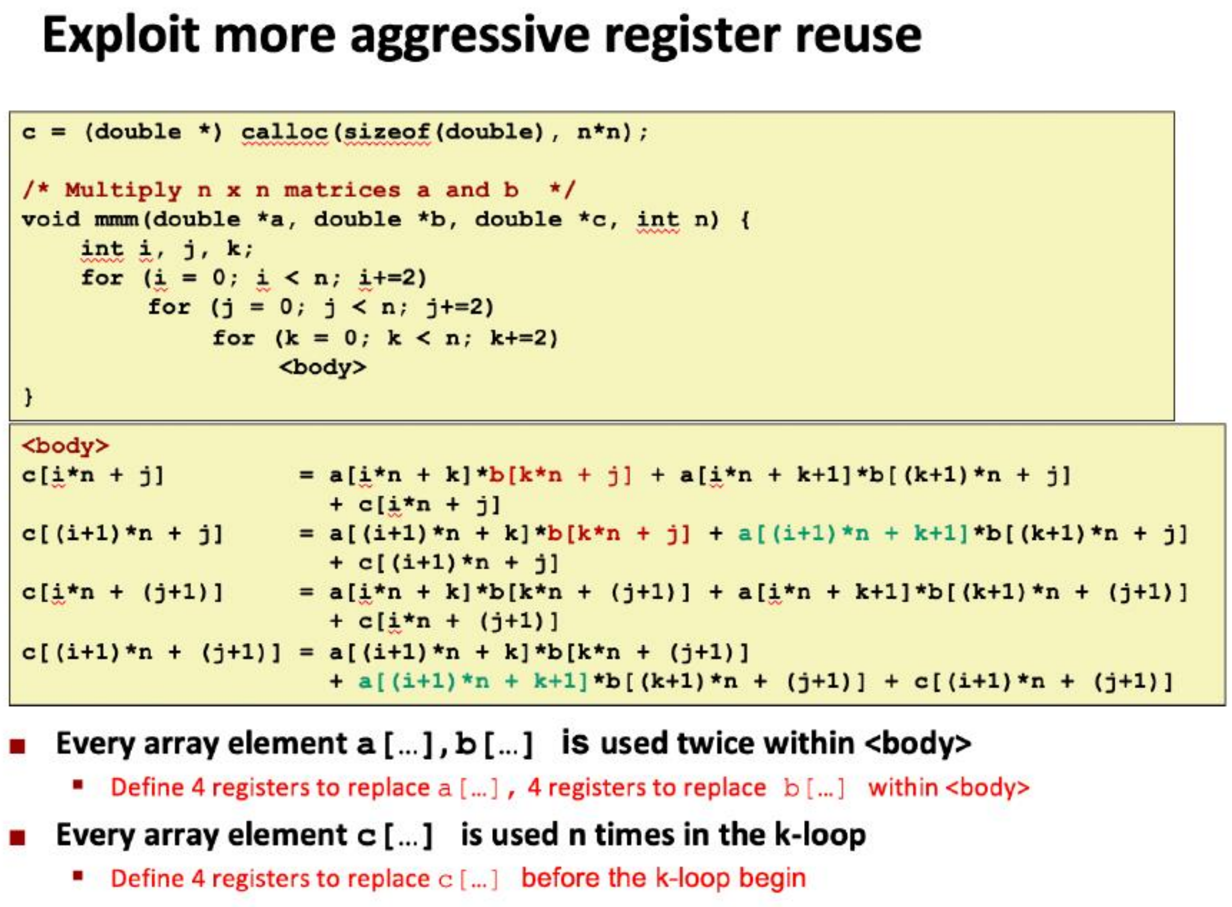
$$\text{maximum difference} = 0.000000000000000$$

This calculation is being done at the end of the source code ***all_in_one.cpp.*** At the same time, the maximum difference within elements of each matrix c is also reported in ***all_in_one.cpp***. If the results from different algorithms match completely, these numbers should also be the same and buy running ***all_in_one.cpp*** we will see that they are.

All the algorithms in this project have been provided in ***all_in_one.cpp*** source code in which each algorithm has been implemented as a function. ***all_in_one.cpp*** makes the performance comparison and correctness checking visually easier.

Part #2.

Let's use ***dgemm2*** to denote the algorithm in the following ppt slide from our class. Implement ***dgemm2*** and test it on your machine with n= 64, 128, 256, 512, 1024, 2048. Measure the time spend in the algorithm. Calculate the performance (in GFLOPS) of the algorithm. You must report the CPU and compiler information that you test your program. Your test matrices have to be 64-bit double floating-point random numbers. Do not forget to check the correctness of your computation results.



Exploit more aggressive register reuse

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=2)
         for (j = 0; j < n; j+=2)
              for (k = 0; k < n; k+=2)
                   <body>
}
```

```
<body>
c[i*n + j]           = a[i*n + k]*b[k*n + j] + a[i*n + k+1]*b[(k+1)*n + j]
                       + c[i*n + j]
c[(i+1)*n + j]       = a[(i+1)*n + k]*b[k*n + j] + a[(i+1)*n + k+1]*b[(k+1)*n + j]
                       + c[(i+1)*n + j]
c[i*n + (j+1)]       = a[i*n + k]*b[k*n + (j+1)] + a[i*n + k+1]*b[(k+1)*n + (j+1)]
                       + c[i*n + (j+1)]
c[(i+1)*n + (j+1)] = a[(i+1)*n + k]*b[k*n + (j+1)]
                       + a[(i+1)*n + k+1]*b[(k+1)*n + (j+1)] + c[(i+1)*n + (j+1)]
```

- Every array element a[…], b[…] is used twice within <body>
  - Define 4 registers to replace a[…], 4 registers to replace b[…] within <body>
- Every array element c[…] is used n times in the k-loop
  - Define 4 registers to replace c[…] before the k-loop begin

Solution:

The implementation of the algorithms is provided in ***dgemm2*** and ***dgemm2v2*** functions in the ***all_in_one.cpp*** source code file. In ***dgemm2*** function***,*** the algorithm has been implemented without using any register reuse. In ***dgemm2v2***, on the other hand, 12 registers have been defined to replace a, b and c matrices in the body of the algorithm. In this algorithm the number of floating-point operations is again: $4 \times 4 \times \left(\frac{n}{2}\right)^3 = 2n^3$. This number and the execution times provided in Table 3., are then used to calculate GFlops which are presented in Table 4.

*Table 3. Execution times of **dgemm2** and **dgemm2v2** algorithms.*

	n=66	n=126	n=252	n=510	n=1020	n=2052	n=4050
dgemm2v2 (ms)	0.000	2.006	21.087	181.482	2139.721	17,754.243	No time
dgemm2 (ms)	1.003	5.041	42.113	342.912	3671.798	35,628.821	No time

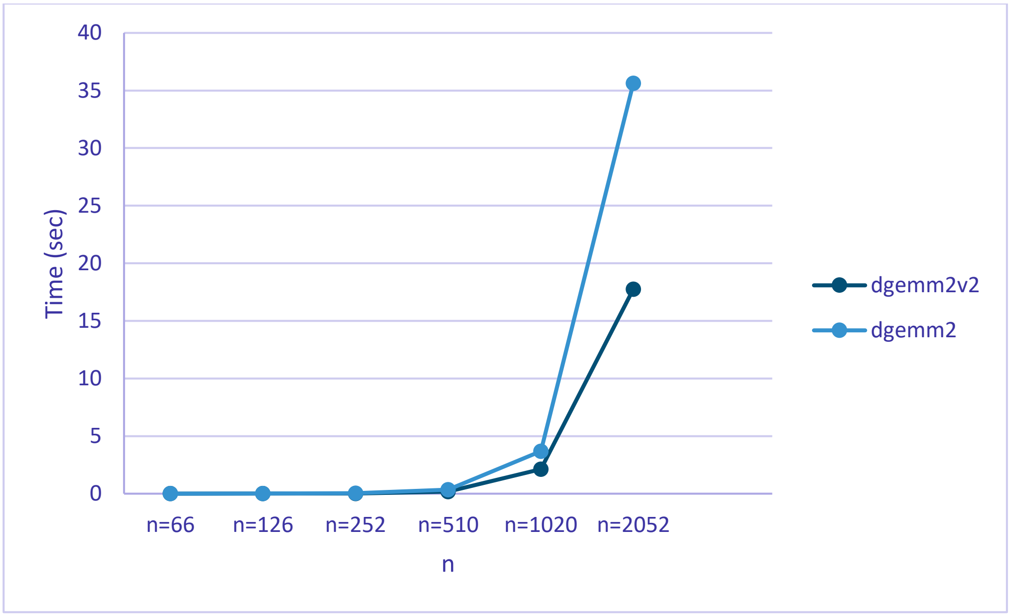


Figure 3. Execution Time for dgemm2v2 and dgemm2 algorithms.

*Table 4. GFlops of **dgemm2** and **dgemm2v2** algorithms.*

	n=66	n=126	n=252	n=510	n=1020	n=2052	n=4050
dgemm2v2	inf	1.9943	1.5178	1.4618	0.9919	0.9733	No time
dgemm2	0.5732	0.7936	0.7600	0.7736	0.5780	0.4850	No time

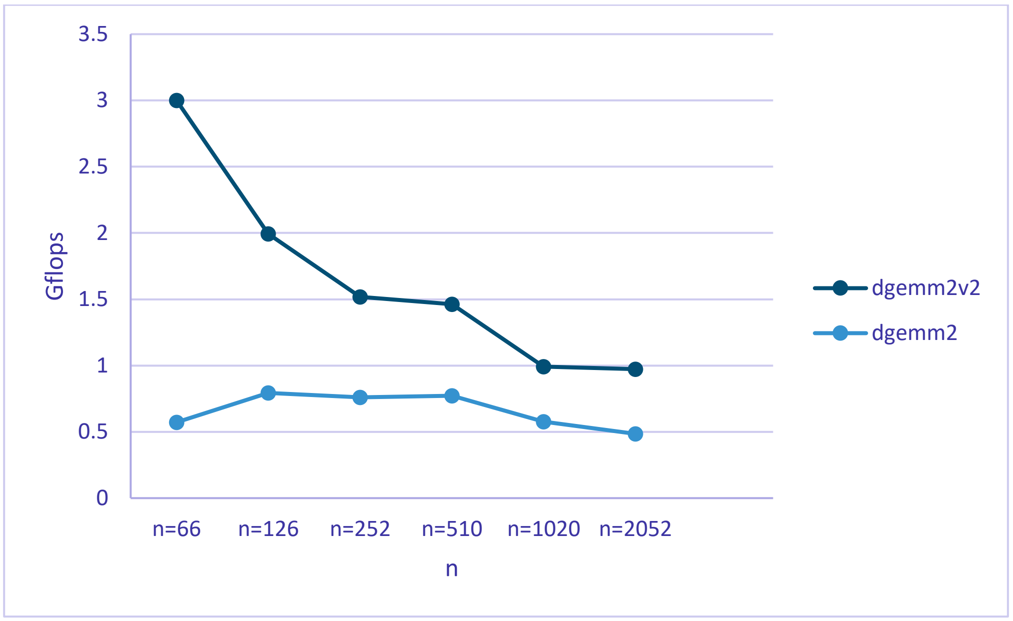
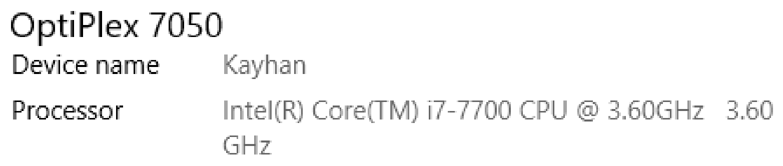


Figure 4. GFlops of dgeem2v2 and dgemm2 algorithms.

CPU information:



OptiPlex 7050

Device name	Kayhan
Processor	Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz 3.60 GHz

Compiler Name: GCC (using "*g++ -O0 all_in_one.cpp*" command in the command prompt of windows)

To check the correctness of the algorithm, the maximum difference between the results obtained from ***dgemm2v2***, ***dgemm2***, ***dgemm1*** and ***dgemm0*** was checked:

For $n = 64$ and ***dgemm2***, **dgemm0**:

$$\text{maximum difference} = 0$$

For $n = 64$ and ***dgemm2v2***, **dgemm0**:

$$\text{maximum difference} = 0$$

For $n = 64$ and ***dgemm2v2***, **dgemm2**:

$$\text{maximum difference} = 0$$

For $n = 64$ and ***dgemm2v2***, **dgemm1**:

$$\text{maximum difference} = 0$$

For $n = 64$ and ***dgemm2***, **dgemm1**:

$$\text{maximum difference} = 0$$

Part #3.

Assume you have 16 registers to use, please maximize the register reuse in your code (call this version code **dgemm3**) and compare your performance with ***dgemm0, dgemm1,*** and ***dgemm2***.

Solution:

For this part, instead of using 2*2 blocks as in ***dgemm2*** algorithm, 3*3 blocks were used. Doing this the number of equations in the body part of the ***dgemm2*** will increase to: $3^2 = 9$. Also, i, j and k will jump by three instead of two. The code is presented in the next page. Using a 3*3 block, will gives us 9 a[…], 9 b[..] and 9 c[…] which in total will be 27. If possible the optimum thing to do would be using 27 register reuses. But only 16 is allowed in this part. 9 registers must be used for c. This will leave us with 7 more registers. The most optimal approach will be using three registers for the first row of matrix *a* and 3 for the first column of matrix *b*. After performing the first row*column operation, replace those 6 register with the second row and the second column of *a* and *b* matrices, respectively, and so on. After each replacement in the innermost loop, the next term of the equations *c[…]+=a[…]b[...]+a[…]b[…]…* will be calculated and will added to the register that has replaced the c in the first place.

```
int i,j,k;
for (i=0 ;i<n ;i+=3 )
    for (j=0; j<n; j+=3){

        register double rc1= c[(i+0)*n+(j+0)];
        register double rc2= c[(i+1)*n+(j+0)];
        register double rc3= c[(i+2)*n+(j+0)];

        register double rc4= c[(i+0)*n+(j+1)];
        register double rc5= c[(i+1)*n+(j+1)];
        register double rc6= c[(i+2)*n+(j+1)];

        register double rc7 =c[(i+0)*n+(j+2)];
        register double rc8 =c[(i+1)*n+(j+2)];
        register double rc9 =c[(i+2)*n+(j+2)];



        for (k=0 ; k<n; k+=3){

            register double a00=a[(i+0)*n+(k+0)];
            register double a10=a[(i+1)*n+(k+0)];
            register double a20=a[(i+2)*n+(k+0)];

            register double b00=b[(k+0)*n+(j+0)];
            register double b01=b[(k+0)*n+(j+1)];
            register double b02=b[(k+0)*n+(j+2)];

            rc1 += a00*b00;rc2 += a10*b00;rc3 += a20*b00;

            /*.........................................................*/

            rc4 += a00*b01; rc5 += a10*b01; rc6 += a20*b01;


            /*.........................................................*/

            rc7 += a00*b02; rc8 += a10*b02; rc9 += a20*b02;

            a00=a[(i+0)*n+(k+1)];
            a10=a[(i+1)*n+(k+1)];
            a20=a[(i+2)*n+(k+1)];

            b00=b[(k+1)*n+(j+0)];
            b01=b[(k+1)*n+(j+1)];
            b02=b[(k+1)*n+(j+2)];

            rc1 += a00*b00;rc2 += a10*b00;rc3 += a20*b00;


            /*.........................................................*/

            rc4 += a00*b01;rc5 += a10*b01;rc6 += a20*b01;


            /*.........................................................*/

            rc7 += a00*b02;rc8 += a10*b02;rc9 += a20*b02;

            a00=a[(i+0)*n+(k+2)];
            a10=a[(i+1)*n+(k+2)];
            a20=a[(i+2)*n+(k+2)];

            b00=b[(k+2)*n+(j+0)];
            b01=b[(k+2)*n+(j+1)];
            b02=b[(k+2)*n+(j+2)];

            rc1 += a00*b00;rc2 += a10*b00;rc3 += a20*b00;


            /*.........................................................*/

            rc4 += a00*b01;rc5 += a10*b01;rc6 += a20*b01;


            /*.........................................................*/

            rc7 += a00*b02;rc8 += a10*b02;rc9 += a20*b02;




        }
        c[(i+0)*n+(j+0)]=rc1;
        c[(i+1)*n+(j+0)]=rc2;
        c[(i+2)*n+(j+0)]=rc3;


        c[(i+0)*n+(j+1)]=rc4;
        c[(i+1)*n+(j+1)]=rc5;
        c[(i+2)*n+(j+1)]=rc6;


        c[(i+0)*n+(j+1)]=rc4;
        c[(i+1)*n+(j+1)]=rc5;
        c[(i+2)*n+(j+1)]=rc6;


        c[(i+0)*n+(j+2)]=rc7;
        c[(i+1)*n+(j+2)]=rc8;
        c[(i+2)*n+(j+2)]=rc9;

    }
}
```

As it can be seen in this code, 9 registers have been defined to replace the *c[…]* matrices. The number of floating-point operations is 6 for each equation and therefore we will have 6×9 floating-point operations in the innermost loop. Also, the number of iterations this time is: $\left(\frac{n}{3}\right)^3$. but we should expect the same number of floating-points operation as previous algorithms, which is exactly the case here as well: $8 \times 16 \times \left(\frac{n}{4}\right)^3 = 2n^3$. The execution time and GFlops of ***dgemm3*** is provided in Table 5 and 6.

Table 5. Execution time of different algorithms for various n values (all values are in milliseconds)

	n=66	n=126	n=252	n=510	n=1020	n=2052	n=4050
dgemm3	0.000	2.005	14.065	127.339	1463.926	12,458.181	No time
dgemm2v2	0.000	2.006	21.087	181.482	2139.721	17,754.243	No time
dgemm2	1.003	5.041	42.113	342.912	3671.798	35,628.821	No time
dgemm1	0.000	5.002	41.111	386.027	4529.083	55,208.935	No time
dgemm0	1.0300	7.018	64.170	586.562	7602.259	91,518.561	No time

Table 6. GFlops of different algorithms for various n values

	n=66	n=126	n=252	n=510	n=1020	n=2052	n=4050
dgemm3	inf	1.9953	2.2755	2.0834	1.4498	1.3870	No time
dgemm2v2	inf	1.9943	1.5178	1.4618	0.9919	0.9733	No time
dgemm2	0.5732	0.7936	0.7600	0.7736	0.5780	0.4850	No time
dgemm1	inf	0.7998	0.7785	0.6872	0.4686	0.3130	No time
dgemm0	0.5582	0.5700	0.4987	0.4523	0.2791	0.1888	No time

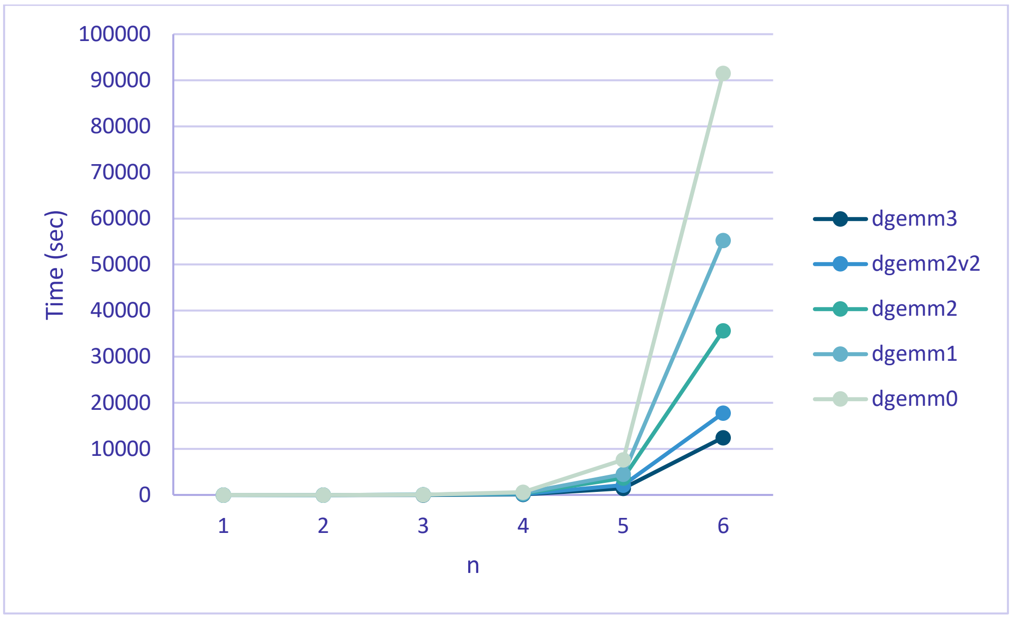


Figure 5. Time comparison between all algorithms

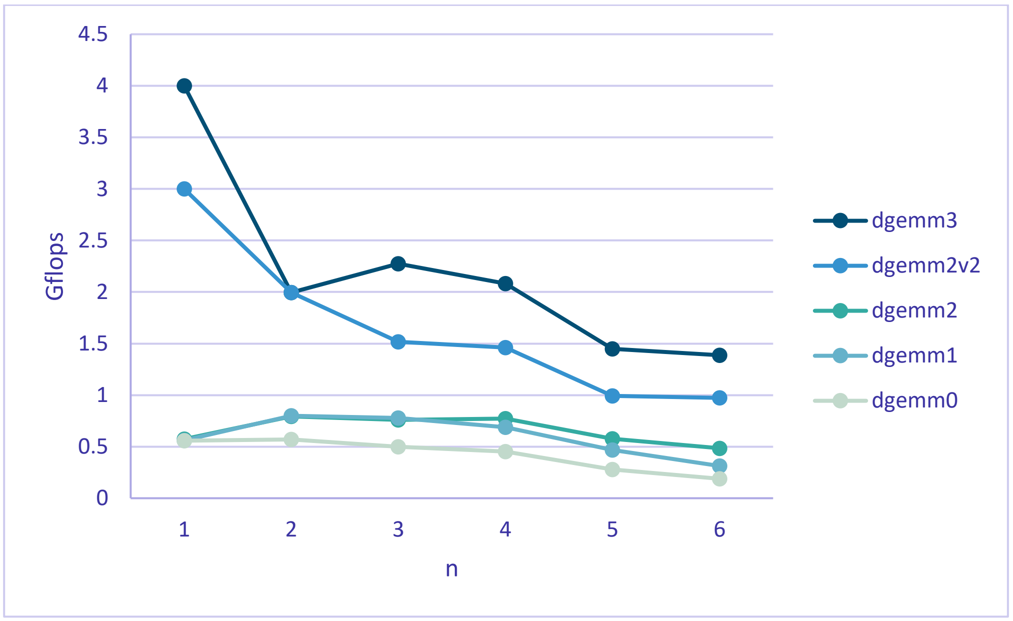


Figure 6. GFlops comparison between all algorithms