Matrix Multiplication using Memory Coalescing Technique

Submitted by: Goli Uma Sankar
Regd. No. : 18560

Introduction :

1) Matrix multiplication is a very frequent and at the same time very slow mathematical operation, since its time complexity is close to cubic.

2) Matrix multiplication may be carried out in several ways, also by using block sub-matrix multiplication.

3) This variant of the algorithm is suitable for the testing of the use of the device's shared memory that is much faster than the global one.

4) The implementation speed of matrix multiplication measured in three different ways:
   a) the implementation of ordinary multiplication on a single-processor host,
   b) the implementation of ordinary multiplication on CUDA device with the use of global memory, and
   c) the implementation of modified block multiplication on CUDA device with the use of shared memory.

Ordinary Multiplication on the Host :

For ordinary matrix multiplication on the host, mat_mul_ord function was used in which we applied a known formula for the calculation of (i, j)-element of product matrix.

C[i][j] = SUM of A[i][k]*B[k][j] for all k = 0 to n
where 'n' is the size of the matrix.

// Function to find C = A*B on the host and returns C.

```
 int *mat_mul_ord(int *A, int *B, int *C)
{
        for(int i = 0; i < n; i++)
                for(int j = 0; j < n; j++)
                {
                        int sum = 0;

                        for(int k = 0; k < n; k++)
                                sum += A[i*n+k] * B[k*n+j];

                        C[i*n+j] = sum;
                }

        return C;
}
```

Three nested for loops directly imply the cubic time complexity of this implementation.

<u>Ordinary Multiplication on the Device</u> :

1) Matrices may be multiplied on the device, so that one kernel is charged for the calculation of each product matrix element.

2) This will perform its task by multiplying the corresponding row of the first and column of the second matrix.

3) Ordinary matrix multiplication on the device was carried out with mat_mul_dev kernel.

4) The data on both matrices (matrix A and B) were read directly from the global memory; where to the result (matrix C) was also recorded.

5) The calculation of one product matrix element on the device (similar to that on the host) is performed with the help of one for loop.

6) Kernels were organized in blocks of size BLOCK_SIZE × BLOCK_SIZE, and the number of blocks was modified according to the size of matrices.

7) To avoid troubles with calculating border elements at arbitrary matrix dimensions we limited our tests only to the matrices of ``correct" dimensions (a multiple of BLOCK_SIZE parameter).

8) Invoking the matrixMulDG kernel with the mentioned number of threads and blocks :

```
dim3 block(BLOCK_SIZE, BLOCK_SIZE);
     dim3 grid(n/BLOCK_SIZE, n/BLOCK_SIZE);

mat_mul_dev<<<grid, block>>>(A_d, B_d, C_d);
```

//Function to find C = A*B on the device

```
__global__ void mat_mul_dev(int *A, int *B, int *C)
{

     int x=threadIdx.y+blockIdx.y*blockDim.y;
     int y=threadIdx.x+blockIdx.x*blockDim.x;
     int sum=0;

     if((x<n)&&(y<n))
          for (int k=0;k<n;k++)
               sum += A[x*n+k]*B[y*n+k];

     C[x*n+y]=sum;

}
```

Modified block Multiplication on the Device using Memory Coalesced Technique :

1) The largest deficiency of ordinary matrix multiplication on the device is the use of slow global memory. So, in this method we use shared memory.

2) The condition needed for successful use of shared memory, namely that the threads of the same block use the same data, is not met in classic matrix multiplication.
i.e., namely, here the (i, j)-thread requires the i-row of matrix A and the j-row of matrix B; no other thread requires these two pieces of information.

3) The multiplication procedure is to be modified, so that the threads within the same block will require a smaller quantity of the same data.

4) This is done by using block multiplication of sub matrices of dimension b×b (‘b’ is dimension of submatrices; Here ‘b’ is taken as BLOCK_SIZE).

5) To calculate product A[i,j]B[j,k] of dimension b×b we need b^2 elements of matrix A and just as many elements of matrix B.

6) This multiplication sub-operation is carried out in b^2 threads, namely so that each thread first loads “its own” elements of matrices A and B in the shared memory and then by using local data calculates its own piece of the result.

7) The outer for loop takes care of the “walk” along the block row and column, whereas the inner one for the calculation of one element of product A[i,j]B[j,k].

// Function to find C = A*B on device using shared memory (Memory Coalescing Technique).

```
__global__ void matrixMul(int *A, int *B, int *C)
{

       // Declaration of the shared memory arrays As and Bs used to store the sub-matrices of A and B respectively
       __shared__ int As[BLOCK_SIZE][BLOCK_SIZE];
       __shared__ int Bs[BLOCK_SIZE][BLOCK_SIZE];

       int w = BLOCK_SIZE;

       // Block Index
       int bx = blockIdx.x;
       int by = blockIdx.y;

       // Thread Index
       int tx = threadIdx.x;
       int ty = threadIdx.y;


        // Row 'row' and Column 'col' of matrix A or B
       int col = bx*w + tx;
       int row = by*w + ty;

       // Cv is used to store the element of the block sub-matrix that is computed by the thread
```

```
    int Cv = 0;

    // Loop over all the sub-matrices of A and B required to compute the block sub-matrix
    for(int k = 0; k < n/w; k++)
    {
        // Load the matrices from device memory to shared memory; each thread loads one element of each matrix
        As[ty][tx] = A[row*n + (k*w + tx)];
        Bs[ty][tx] = B[(k*w + ty)*n + col];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together; each thread computes one element of the block sub-matrix
        for(int l = 0; l < w; l++)
            Cv += As[ty][l] * Bs[l][tx];

        // Write the block sub-matrix to device memory; each thread writes one element
        C[row*n + col] = Cv;
    }

}
```

Correctness :

1) This is done by comparing the values of ordinary matrix multiplication on Host with multiplication on Device and modified block multiplication.

2) The following function ‘compare’ is used for this purpose :

```
//Function for comparing 2 matrices elements
void compare(int N, double *wref, double *w)
{
    double maxdiff,this_diff;
    int numdiffs;
    int i,j;
    numdiffs = 0;          maxdiff = 0;

    for(i=0;i<N;i++)
        for(j=0;j<N;j++)
        {
            this_diff = wref[i*N+j]-w[i*N+j];

            if(this_diff < 0)
                this_diff = -1.0*this_diff;

            if(this_diff>threshold)
            {
                numdiffs++;

                if(this_diff > maxdiff)
```

```
                    maxdiff=this_diff;
                }
            }

        if(numdiffs > 0)
            printf("%d Diffs found over threshold %f; Max Diff = %f\n", numdiffs, threshold, maxdiff);

        else
            printf("No differences found between reference and test versions\n");
}
```

The Speed Tests :

1) To measure the speed of different implementations, we took ‘n’ values as 128, 256, 512, 1024, 2048, 4096 and 8192.

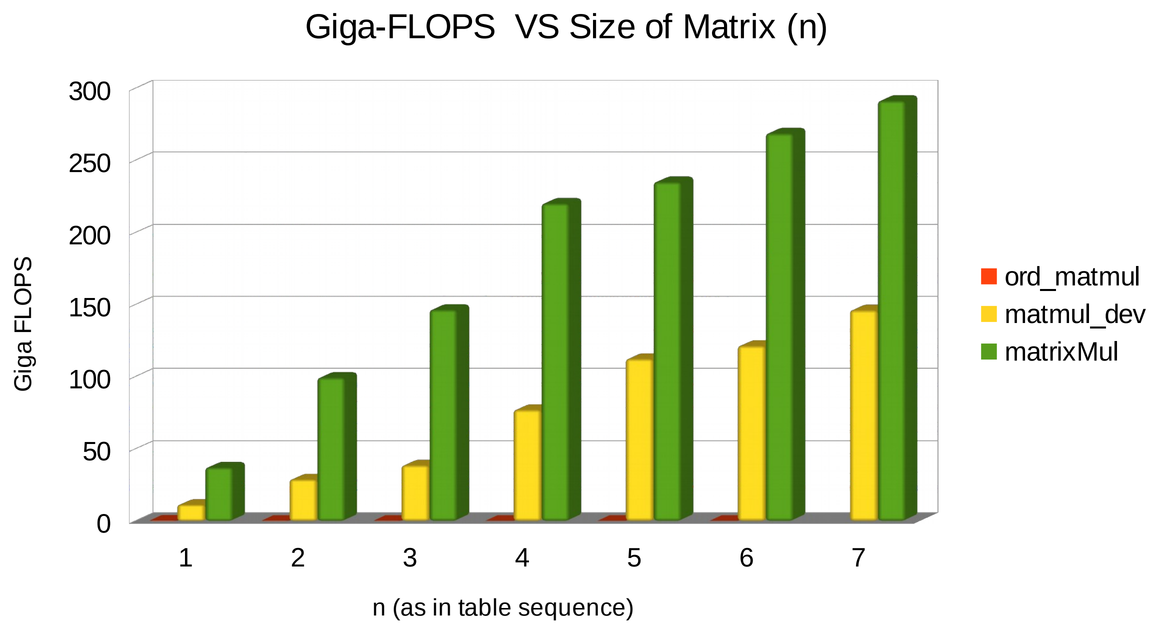
2) We have two graphs representing the speed tests :
   (a) Giga-FLOPS ***VS*** Values of ‘n’
   (b) Time(in secs) ***VS*** Values of ‘n’

3) We can clearly see that ordinary multiplication on the host is slower than multiplication on the device with the use of global memory.

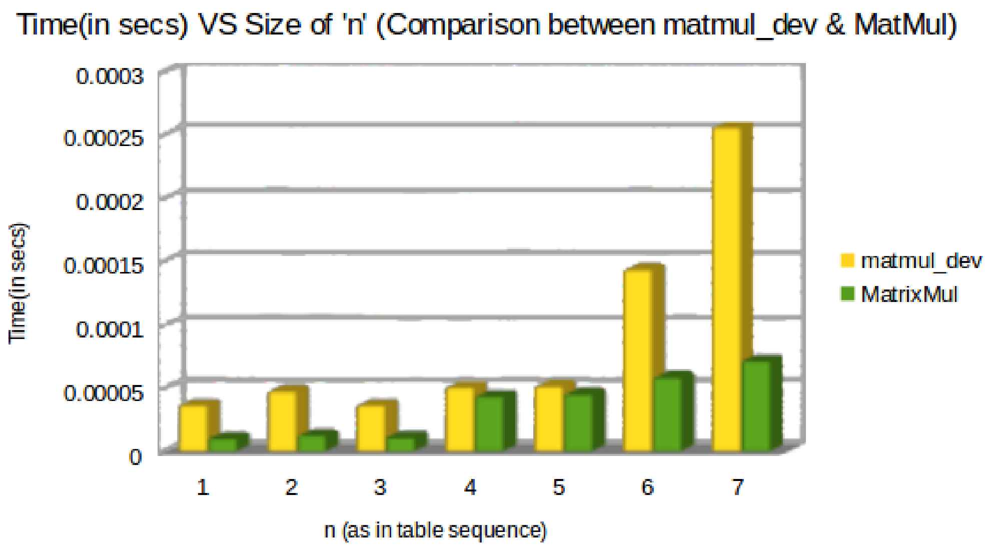
4) In turn, multiplication on the device with the use of global memory is slower than multiplication on the device with the use of shared memory.

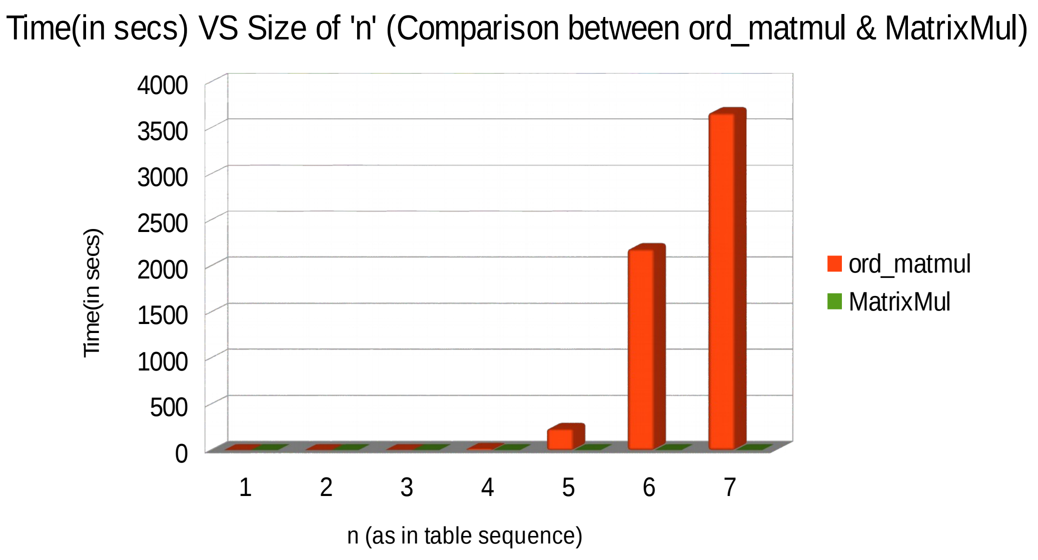
5) Related tables and graphs for these are given below (for ***BLOCK_SIZE = 32***):

Size(n)	**Gflops for ‘ord_matmul’**	**Gflops for ‘matmul_dev’**	**Gflops for ‘matrixMul’**
128	0.3	11.3	37.2
256	0.2	28.9	99.5
512	0.2	38.7	146.6
1024	0.1	77.2	220.3
2048	0.1	112.5	235.2
4096	0.1	121.6	268.9
8192	0.3	146.3	291.5



Size(n)	Time(in sec) for 'ord_matmul'	Time(in sec) for 'matmul_dev'	Time(in sec) for 'matrixMul'
128	0.015359	0.000037	0.000011
256	0.159795	0.000048	0.000013
512	1.243145	0.000037	0.000012
1024	15.437091	0.000051	0.000044
2048	232.114766	0.000052	0.000046
4096	2185.184887	0.000144	0.000059
8192	3663.930842	0.000256	0.000072





Conclusion :

1) Here, we implemented three implementations of algorithms solving the problem of matrix multiplication – a classical CPU implementation and two GPU implementations, one using global and the other using shared memory.

2) The results show that GPU computation has a lot of advantages, especially in speeding up the execution time of our programs while on the same time the computer's main processor is released and thus able to perform other tasks.

3) When designing the GPU implementation, a special attention should be paid to the usage of device's shared memory.