E0 243: DMM performance optimization

Name: Paras Lohani, S.R.No: 18226

1 Task

Write CUDA version of the optimized multithreaded code of CPU.

2 System configuration

GPU Server provided by IISc is used to test the changes.

3 Part B

3.1 Concepts

There are two categories in CUDA programming: Host(CPU and its memory) and Device(GPU and its memory). Device part runs parallel and Host part runs serial. I have used some CUDA APIs like $cudaMalloc, cudaMemcpy, cudaFree$ which is equivalent to $malloc, memcpy, free$ of C. $\_\_global\_\_$ is the keyword which indicates that the function will run on device. This function can be called from Host part. $Function\_name <<< Blocks, Threads >>> (Parameters)$ is a way to call a function. $ThreadIdx.x$ and $BlocksIdx.x$ to find which block and thread is running the cuda function. This will represent for one dimension. It can also be modified for two dimension. I have worked with one dimension block only.

3.2 Idea

I have used the idea that was implemented in PartA. The processor has a line size of 64B and we are given matrix of integers i.e. each are 4B. Therefore the block size is of dimension 16x16. Since the matrix size is 128x128, therefore the number of blocks are 64. In CUDA, I am using 64 blocks with 1 thread in it (was commented in main code (single thread)). Therefore 1 block of CUDA is assigned 1 block of matrix. Every block runs parallely in CUDA and each block is computed by single thread.

3.3 Optimizations

I am using cache structure for optimally dividing the whole matrix into the cache blocks to maximize cache hit. Each such blocks are assigned to the corresponding CUDA block so that it run parallely. This will reduce the latency and will be a better optimized implementation. We can also increase the thread count per block to increase the parallelism inside the block also.

3.4 Implementation

The processor has a line size of 64 bytes and we are given the matrix of integers which is of 4 bytes. This helps to conclude that the block size will be $64/4 = 16$. Therefore the matrix will be divided into blocks of size 16. Each block is then computed by each CUDA block.

To decrease the execution time further, the concept of loop unrolling is used where use of loop is reduced and is replaced by separate instructions. It will expose the instruction level parallelism.

```
__global__ void DMMMul(int N, int *matA, int *matB, int *output) {

    int i = blocksize * (blockIdx.x / (N / blocksize));
    int j = blocksize * (blockIdx.x % (N / blocksize));
    int ind = blockIdx.x * (2 * N - 1);

    for(int k = i; k < i + blocksize; k++) {
        #pragma unroll(4)
        for (int l = j; l < j + blocksize; l++) {
            output[ind + (k + l)] += matA[k * N + l] * matB[l * N + N - 1 - k];
        }
    }
    __syncthreads();
}
```

Figure 1: DMM CUDA device function

3.5 Results

I added the clock time code around function calls to collect the running time and found the below result:

3.5.1 Input size 128

Reference execution time	=	0.263 ms
Device function(DMMul) execution time	=	0.025 ms
GPU thread execution time	=	336.649 ms
Speedup (Device function considered)	=	10.52

For calculating speedup, I considered Device function because the DMM logic is implemented in that. Overall GPU thread execution time increased a lot as we are doing heavy computation like copying host variable values to device variables i.e. cudaMemcpy and other CUDA APIs which is a type of overhead.

3.6 Figures

```
[paraslohani@cl-gpusrv1 PartB]$ make run_server
./diag_mult_server data/input_128.in
Input matrix of size 128
Reference execution time: 0.263 ms
Device function execution time: 0.025ms
gpu thread execution time: 336.649 ms
[paraslohani@cl-gpusrv1 PartB]$
```

Figure 2: Execution Time Output