Matrix-Matrix Multiplication using CUDA and OpenMP

Hirokatsu (Hiro) Suzuki

Problem

- Perform matrix-matrix multiplication.
- Matrices are not assumed to be square.
- The entries of matrices should have floating point.
- Print resultant matrix (commented out).
- Compute the speedup when parallelism is on and off.
- Use two technologies to create hybrid program.

Solution

- I used CUDA and OpenMP to create a hybrid program that computes matrix-matrix multiplication.
- This code has both matrix-matrix multiplication on GPU and CPU.
- It was tested on rectangular and square matrices.
- Parallelism (CUDA+OpenMP) was used to compute the speedup with and without it.
- Total of three cores were used and each GPU was dedicated to 4 threads.
- Block dimension in the GPU was modified to test the speedup.
- Compiled with “nvcc <cuda file> -std=c++11 -Xcompiler -fopenmp”
- Pseudo-code is shown below.

```
void fill_matrix(int *A, float fac, int n, int m) {

        // loop through matrix of size nxm to fill in float entries.
        ……
}

void print_matrix(int *A, int m, int n) {

        // loop through resultant matrix and print result
        …..
}

void perform_operation(int *A, int *B, int *C, int m, int n) {

        // CPU version of matrix-matrix multiplication
        …..
}

__global__ void perform_operation_cuda(int *A, int *B, int *C, int m, int n) {
         int i = blockIdx.x * blockDim.x + threadIdx.x;
         int j = blockIdx.y * blockDim.y + threadIdx.y;
         if (i<m)
         {
          if (j<n)
          {
            C[i*n+j] = A[i*n+j]*B[i*n+j];
          }
         }
}

struct aux_device {
         int gpuid; //gpu id
         int *A_d; // device array A on the gpu gpuid
};

int main (void) {

         // initialize variables
         …..

         // define the grid and block dimensions
         dim3 dimGrid(…..);
         dim3 dimBlock(…..);

         // initialize gpu, cpu, threads number and ids
         …..

         // fill in matrices – resultant C will be zeros
         fill_matrix(A,…..);
         fill_matrix(B,…..);
         fill_matrix(C,…..);

         // initialize the GPU numbers and threads number
         …..

         // Set the threads to each GPUs - OpenMP
        #pragma omp parallel shared(num_gpus, dev_mem) private(cpu_thread_id, gpuid)
        {
        #pragma omp critical
         {
         cpu_thread_id = omp_get_thread_num();
         cudaSetDevice(cpu_thread_id % num_gpus);
         cudaGetDevice(&gpuid);
         dev_mem[gpuid].gpuid = cpu_thread_id;
```

```
 }
}

  //allocate and copy the array A ones by GPU - OpenMP
#pragma omp parallel shared(A, dev_mem, num_gpus) private(f, cpu_thread_id, gpuid)
{
        #pragma omp critical
        {
          cpu_thread_id = omp_get_thread_num();
          for (f = 0;f<num_gpus;f++)
          {
            if (cpu_thread_id == dev_mem[f].gpuid)
            {
              cudaMalloc( (void **)&dev_mem[f].A_d, sizeof(int) * N*M);
              cudaMemcpy( dev_mem[f].A_d, A, sizeof(int) * N*M,
        cudaMemcpyHostToDevice);
              cudaGetDevice(&gpuid);
              // printf("CPU thread %d uses CUDA device %d\n", cpu_thread_id, gpuid);
              break;
            }
          }
         }
}

  // matrix-matrix multiplication on GPU
  sum = 0;
#pragma omp parallel shared(dimGrid, dimBlock, nf, sum, dev_mem) private(f, B, B_d,
C_d, C, gpuid, cpu_thread_id)
{
         // allocate memory for B and C
         cudaMalloc( (void **)&B_d, sizeof(int) * N*M);
         cudaMalloc( (void **)&C_d, sizeof(int) * N*M);


         cudaGetDevice(&gpuid);
         #pragma omp for reduction(+:sum)
         for (f=0; f<nf; f++)
         {
           // fill in matrix B and allocate memory
           …..

           // begin time
           …..
```

```
                    // call function
                    perform_operation_cuda<<<dimGrid, dimBlock>>>(dev_mem[gpuid].A_d,
                  B_d, C_d, N, M);

                    // end time
                    …..

                    // print C
                    …..

                    // copy data to host
                    cudaMemcpy( C, C_d, sizeof(int) * N*N, cudaMemcpyDeviceToHost);
                   }

                   // free memory
                   cudaFree(B_d);
                   cudaFree(C_d);
          }

           for (f = 0;f<num_gpus;f++)
           {
             cudaFree(dev_mem[f].A_d);
           }

           // reset cuda
           cudaDeviceReset();

           // begin time
           …..

           // call function - CPU
           perform_operation(A, B, C, N, M);

           // end time
           …..

           return 0;
}
```

Result

Block Dimension	GPU1 (sec)	GPU2 (sec)	GPU3 (sec)	Elapsed (sec)	CPU (sec)	Speedup
(1,1)	0.000015	0.000016	0.000014	0.000045	0.000208	4.62222222
(5,5)	0.000015	0.000014	0.000015	0.000044	0.000221	5.02272727
(20,20)	0.000015	0.000016	0.000013	0.000044	0.000191	4.34090909
(25,25)	0.000016	0.000013	0.000013	0.000042	0.000186	4.42857143
(50,50)	0.000002	0.000002	0.000002	0.000006	0.000203	33.8333333
(100,100)	0.000002	0.000002	0.000002	0.000006	0.000215	35.8333333
(200,200)	0.000003	0.000001	0.000002	0.000006	0.000221	36.8333333

Table.1 Matrix multiplication of 100x512 and 100x512 matrices. 4 threads are used in each GPU. Grid dimension was (4,4).

Block Dimension	GPU1 (sec)	GPU2 (sec)	GPU3 (sec)	Elapsed (sec)	CPU (sec)	Speedup
(1,1)	0.000017	0.000016	0.000015	0.000048	0.000342	7.125
(5,5)	0.000017	0.000018	0.000016	0.000051	0.000333	6.52941176
(20,20)	0.000016	0.000015	0.000018	0.000049	0.000532	10.8571429
(25,25)	0.000033	0.000034	0.000028	0.000095	0.000493	5.18947368
(50,50)	0.000002	0.000002	0.000003	0.000007	0.000379	54.1428571
(100,100)	0.000003	0.000003	0.000004	0.00001	0.000487	48.7
(200,200)	0.000003	0.000002	0.000002	0.000007	0.00043	61.4285714

Table.2 Matrix multiplication of two 300x300 matrices. 4 threads are used in each GPU. Grid dimension was (4,4).

As shown above, there are indeed speedups between GPU and CPU using different block dimensions used for matrix-matrix multiplications. With block dimension less than (25, 25), the speed up was consistent in the rectangular multiplications and little fluctuations in square multiplications. Once the block dimension was increased to (50,50), the speedup drastically increased. More studies of this hybrid program can be extended by using more GPU cores, threads, grid dimension, etc.