HPC Hw4 (Zhe Chen)

Machine configuration

Results are obtained by running on CIMS's gpu machines cuda{1-5}.cims.nyu.edu. Configurations of these servers can be found on https://cims.nyu.edu/webapps/content/systems/resources/computeservers.

Modules environment is gcc-4.9.2, cuda-9.3.

Notice that one must use cuda 9 or higher verision to support __syncwarp in P1.

P1: Matrix-vector operations on a GPU

a. Inner Product of Vectors

See inner_product.cu.

To do inner product between two very long vectors, we use gpu parallel computing here, i.e. cuda. The same blocking trick as reduction.cu is implemented here. Vectors are divided into blocks, where size of a block is BLOCK_SIZE=1024. And then we do sum of inner products of all blocks by the same blocking method and circularly until we get one summation. We use __syncwrap(), which required cuda 9.0 or higher, to synchronize among threads in same wraps. This solves the wrap divergence problem. Also, we use shared memery of vectors on devices, which is shared by the same block. This trick accelerate memory reading speed greatly since it's really slow between devices and hosts, compared with reading directly from gpu's memory. We also compared it with the cpu parrallel omp version code we wrote before.

We set length of vectors n=2^{24} and results from different gpus, cuda1-3, are shown here. Error between cpu and gpu results is 0.0.

cuda	1	2	3
cpu bandwidth(GB/s)	26.312661	12.729974	9.762167
gpu bandwidth(GB/s)	125.298597	260.949331	501.127507

b. Matrix vector product

Here we implemented matrix-vector product in matrix_vector.cu.

Consider a M-by-N matrix multiplied by a N-by-1 vector. Generally, it's the same trick as vector innner product. We regard it as M inner products of each row of matrix and the N-by-1 vector. Blocking and shared memory are also used here in each innner product. BLOCK_SIZE=1024 and M=N=2^{12} since we want to make M*N the same as N=2^{24} in problem a to make this comparison meaningful.

Bandwidth of gpu cuda code dramatically decreases here because we don't paralel the each row's inner-product.

cuda	1	2	3
cpu bandwidth(GB/s)	34.307205	26.616298	17.013255
gpu bandwidth(GB/s)	1.334414	1.688865	1.750776

P2: 2D Jacobi and Gauss-Siedel on GPUs

a. 2D Jacobi

2D Jacobi method is paralleled on GPUs in jacobi_cuda.cu. It's compared with cpu parallel codes we wrote before using omp.

We use the same trick as figure filtering which is discussed on class. A matrix of (N+2)-by-(N+2) is to store the solution and we update the matrix with Jacobo method at each time. Here we also use blocking method and shared memory. On each block, there's BLOCK_SIZE=32 big matrix, which is shared within a block, and we update the innner (BLOCK_SIZE-2) big matrix. It's similar as we use a 3-by-3 kernel to filter a matrix at each step.

We set N=128 and compare it on cuda1-3. It's set to run until maxium iteration so that the comparison is fair. Maxium iteration is 10000 and after MaxIter, the remaining relative residual is 0.00306506. For cpu omp code, we use 4 threads.

cuda	1	2	3
cpu bandwidth(GB/s)	10.731210	5.211650	9.801787
gpu bandwidth(GB/s)	7.942732	5.889139	3.806528

b. 2D Gauss-Seidel (EXTRA CREDIT)

2D GS method is implemented as GPU parallel version in gs2D-omp.cu. It's generally similar with Jacobi method. We also use blocking and shared memory in blocks. Main difference is that GS method need red-black trick to make it parallizable. Moreover, GS does not need a temporary matrix to store the new solution at each step because red points are completely independently from the black points, while Jacobi method does.

As the same, we set BLOCK_SIZE=32, N=128. It's set to run until maxium iteration so that the comparison is fair. Maxium iteration is 10000 and after MaxIter, the remaining relative residual is 0.00306506. For cpu omp code, we use 4 threads.

cuda	1	2	3
cpu bandwidth(GB/s)	10.532527	5.399020	10.986437
gpu bandwidth(GB/s)	7.888837	4.661180	3.587161

P3: Final Project Detail

Fast parallel cuda code for Ewald Summation for Skokes potential

I, with Guanchun Li, will together take the Ewald Summation problem for our final prokect for HPC class. Here's detail of our plan.

Green function of Stokes flow is shown in Oseen-Burgers tensor, or Stokeslet. Thus, convolution of green function is the key to compute velocity field. In free space, the main difficulty is the slow decay, 1/r, of the kernel, which requires big N and O(N$^{2)}$ complexity. Fortunately, Ewald Summation method does great jobs in this problem. Generally, it split the convolution into real space, which can be adjusted by Ewald parameter ξ, and k-space, which can be computed fast by FFT based method. By this mean, Ewald method change convolution of this problem from O(N$^{2)}$ to O(N logN) with spectral accuracy.

The fact that Ewald method is FFT based makes it possible for us to do parallel computing. We plan to write cuda gpu-parallel code to improve Ewald method. First, we want to implemented parallel FFT method by our own or using cuFFT library, which can be plugged into solution to k-space of Ewald decomposition. Also, there's an integral step in algorithm of Ewald Summation, which is described in http://dx.doi.org/10.1016/j.jcp.2010.08.026. Since it's periodic, we can simply use trapezoidal integral to get spectral accuracy. Moreover, the trapezoidal integral can also be paralleled since it's a summation of very long vector.

The goal of this project is to develop a CUDA-based gpu-parallel libary to implement Ewald summation fast. Hopefully, we could use my CFD final project, which is about particles in a stokes flow above a wall, as an exemple to discuss performance of this method.