CONTACT NAME
Fang Liu: liufang@sccas.cn

Fast Computing of Linkage Disequilibrium on GPU

Fang Liu[1] Jue Wang[1] Xian-Yu Lang[1] Chi-Xue Bin[1] Hai-Nan Zhao[2] Jin-Sheng Lai[2]

SuperComputing Center of Chinese Academy of Sciences[1] China Agricultural University[2]

Introduction

Linkage disequilibrium is the non-random associations of alleles between two loci in a population, which is a fundamental concept to analyze the pattern of genetic diversity and map disease-associated genes[1,2]. Consider two loci (A and B), each segregating for two alleles (A_1, A_2, B_1 and B_2). Four haplotypes (A_1B_1, A_1B_2, A_2B_1 and A_2B_2) are expected to present in the population, with frequencies of x_{11}, x_{12}, x_{11}, x_{22}, respectively. The corresponding frequency of each allele could be expressed as: $p_1 = x_{11} + x_{12}$, $p_2 = x_{21} + x_{22}$, $q_1 = x_{11} + x_{21}$, and $q_2 = x_{12} + x_{22}$. Suppose two alleles A_1 and B_1 at a pair of loci are totally independent from each other, then the observed frequencies above should satisfy the below equation: $x_{11} = p_1 * q_1$. Heuristically, the linkage disequilibrium can be measured using the commonly used correlation coefficient [3]:

$$r^2 = \frac{D^2}{p_1 q_1 p_2 q_2}, where\ D = x_{11} - p_1 * q_1$$

The computation of LD is time-consuming because the time complexity rises with the square of number of alleles. This work presents a fast algorithm to compute LD on GPU using CUDA[4], which gains around a thousand times speedup than its serial counterparts on CPU.

Our Algorithm

For a pair of allele each with M samples, there are four values to be counted during the loop over the samples:

- Number of samples valid (non-missing) on both loci: N ($N \le M$)
- Number of valid samples which contains A_1 on the first locus and B_1 on the second locus: n_{11} ($n_{11} = x_{11} * N$)
- Number of valid samples which contains A_1 on the first locus: n_1 ($n_1 = p_1 * N$, and $p_2 = 1 - p_1$)
- Number of valid samples which contains B_1 on the second locus: n_2 ($n_2 = q_2 * N$, and $q_2 = 1 - q_1$)

Then the correlation coefficient could be equivalently rewritten as:

$$r^2 = \frac{(n_{11} * N - n_1 * n_2)^2}{n_1(N - n_1) * n_2(N - n_2)}$$

The above values are traditionally counted using boolean operations. It can be converted to bitwise operations as shown above, thus can benefit from a specially designed instruction '*__popc*' on NVIDIA GPU devices. The instruction can return the number of set bits in a 32-bit integer parameter, which is mapped to a single instruction for devices of compute capability 2.0 or higher. In this way, the input data can be pre-processed and each allele can be converted from a 8-bit '*char*' to two bits, one is the *data* bit shows to which kind the allele belongs, and the other is the *flag* bit indicating whether the data is valid. For a pair of allele each with M samples, the *data* bits and *flag* bits of all samples can be packed into *K=(M+31)/32* 32-bit integers. The redundant bits at tail are padded using '0' if necessary. While looping over the integers on a certain locus, suppose the k^{th} pair of data integers are denoted as $data_1^k$, $data_2^k$, and flag integers as $flag_1^k$, $flag_2^k$, the number of samples valid on both loci N can be accumulated as:

$$N = \sum_{k=0}^{K-1} \_\_popc(flag_1^k \,\&\, flag_2^k)$$

The valid samples which contains A_1 on the first locus will be represented by '1' on the corresponding bit in $data_1^k$. So the value n_1 can be accumulated simply as below, and n_2 and n_{11} can be computed similarly:

$$n_1 = \sum_{k=0}^{K-1} \_\_popc(data_1^k \,\&\, flag_1^k \,\&\, flag_2^k)$$

$$n_2 = \sum_{k=0}^{K-1} \_\_popc(data_2^k \,\&\, flag_1^k \,\&\, flag_2^k)$$

$$n_{11} = \sum_{k=0}^{K-1} \_\_popc(data_1^k \,\&\, data_2^k \,\&\, flag_1^k \,\&\, flag_2^k)$$

After the loop, the four values needed have been accumulated to compute the final LD value. The above algorithm processes 32 samples simultaneously using only several bitwise instructions, and reduces the input data of each allele to 1/4 from a 8-bit '*char*' to two bits. So after the bitwise re-construction, our algorithm will gain at least 32 times speedup theoretically on Fermi/Kepler GPUs than its traditional serial counterparts. Other techniques such as data reorganization and atomic instructions are also applied in the implementation to reduce the memory latency and the memory consumption. The pseudo-code of our algorithm is summarized in Fig. 1 for reference.

However, the memory on a single GUP might exhaust for quite large input data or relative low threshold for output. In that case, the input data can be cut into pieces with moderate sizes and processed pair by pair. They can also be processed in parallel by several nodes on a GPU cluster to further reduce the computing time.

Results

Data Name	Loci Number	Sample Number	Related Number	CPU Time		GPU Time			Output Time	Total CPU Time	Total GPU Time	Speedup
				Load	Count	Load	Count	Sort				
1.map	45099	514	78828	0.8s	3171.7s	1.0s	1.3s	0.02s	1.2s	3173.7s	3.5s	906.7
2.map	44458	514	81871	0.8s	3008.5s	1.1s	1.4s	0.01s	0.1s	3009.4s	2.6s	1157.5
3.map	131489	514	1916593	2.4s	26087.7s	12.2s	3.1s	0.3s	2.1s	26092.2s	17.7s	1474.1
4.map	91014	514	1090031	1.6s	12717.4s	2.0s	5.8s	0.2s	1.1s	12720.1s	9.1s	1397.8
5.map	244128	514	8310368	4.4s	94856.1s	7.7s	42.0s	1.2s	8.8s	94869.3s	59.7s	1589.1
6.map	256878	514	8545398	4.7s	102940.9s	6.0s	47.6s	1.8s	9.3s	102954.9s	64.7s	1591.2
7.map	279743	514	11251627	5.1s	118919.4s	6.4s	56.7s	1.7s	11.1	118935.6s	75.9s	1567.0
8.map	276850	514	12059072	5.1s	115760.4s	6.3s	55.4s	1.8s	13.1s	115778.6s	76.6s	1511.5
9.map	275798	514	11684379	5.0s	117498.6s	7.9s	55.2s	1.7s	12.1s	117515.7s	76.9s	1504.8
10.map	271244	514	11631394	5.4s	117625.8s	8.4s	53.1s	2.4s	13.8s	117645s	77.7s	1514.1
11.map	272686	514	10412867	5.3s	112128.4s	7.6s	53.8s	1.5s	10.4s	112144.1s	73.3s	1529.9
12.map	279171	514	11929101	5.6s	122863.7s	7.7s	56.4s	2.4s	12.7s	122882.0s	79.2s	1551.5
13.map	65576	514	1121128	1.5s	6436.1s	1.7s	3.0s	0.2s	1.1s	6438.7s	6.0s	1073.1

Table 1: Performance of our algorithm. The threshold for the output is set to be 0.5, and the size of the array for output is set to be 150M.

We tested our algorithm on the Tesla GPU Cluster in SuperComputing Center of Chinese Academy of Sciences. Each computing node is equipped with 2 NVIDIA C2075 cards with 448 CUDA cores and 3GB memory each, and a quad-core Intel Xeon5410 CPU with 8GB memory. The test results of our algorithm on a typical dataset are shown in Table 1 in comparison to a traditional serial implementation of the LD computation. We only compute the LD values of the alleles within each piece. The comparisons between different pieces can be tested in a similar way. As shown in the table, the speedup of our algorithm reaches up to around 1000 times for the test data with moderate sizes.

Pseudo-code of the CUDA Kernel

```
1:int objectAlleleIdx = blockIdx.x * gridDim.y + blockIdx.y;
2:int targetAlleleIdx = blockIdx.z * blockDim.x + threadIdx.x;
3:if( targetAlleleIdx >= objectAlleleIdx || targetAlleleIdx >= alleleNum
4:    || objectAlleleIdx >= alleleNum ) return;
5:for(int k = 0;k < K; k++){
6:    uint data1 = inputData[k*alleleNum+objectAlleleIdx];
7:    uint data2 = inputData[k*alleleNum+targetAlleleIdx];
8:    uint flag1 = inputFlag[k*alleleNum+objectAlleleIdx];
9:    uint flag2 = inputFlag[k*alleleNum+targetAlleleIdx];
10:   uint validFlag = flag1 & flag2;
11:   N += __popc( validFlag);//Number of valid samples
12:   n1 += __popc( data1 & validFlag ); //n1=p1*N
13:   uint ftFlag = data2 & validFlag;
14:   n2 += __popc( ftFlag );//n2=p2*N
15:   n11 += __popc( data1 & ftFlag );//n11=x11*N
16:}
17:if( !n1 && !n2 && n1 != N && n2 != N){
18:   D = n11*N - n1*n2;
19:   r2 = D*D/(n1*(N - n1)* n2*(N - n2));
20:   if(r2 >= THRESHOLD ){
21:      int index = atomicInc(&counter[0],MAX_SIZE-1);
22:      objectNames[index] = objectAlleleIdx;
23:      targetNames[index] = targetAlleleIdx;
24:      resultLD[index] = r2;
25:   }
26:}
```

Figure 1: Pseudo-code of our algorithm

Future Work

Our algorithm successfully gains great speedup for LD computation on modern GPUs. However, it can only deal with homozygous species up to now. In future we are going to extend it to heterozygous species in a similar way and apply our method to genome-wide association studies.

References

[1] Kristin G.A., Leonid K., and Mark S. 2002. Patterns of linkage disequilibrium in the human genome, Nature Reviews Genetics 3, 299-309.

[2] Lynn B. J. 1995. Linkage disequilibrium as a gene-mapping tool. Am J Hum Genet. 56(1): 11–14.

[3] Hill W.G, and Robertson A. 1968. Linkage disequilibrium in finite populations. Theor Appl Genet, 38, 226-231.

[4] NVIDIA. 2012. NVIDIA CUDA: Compute unified device architecture. NVIDIA Corporation.