R07922141
張緣彩
31/10/2018

Data Mining Hw2 Report

I use my own code from homework 1.

Part I. Setting up CUDA environment

```
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 1080 Ti"
  CUDA Driver Version / Runtime Version          9.1 / 9.0
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 11176 MBytes (11719409664 bytes)
  (28) Multiprocessors, (128) CUDA Cores/MP:     3584 CUDA Cores
  GPU Max Clock rate:                            1683 MHz (1.68 GHz)
  Memory Clock rate:                             5505 Mhz
  Memory Bus Width:                              352-bit
  L2 Cache Size:                                 2883584 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 9.1, CUDA Runtime Version = 9.0, NumDevs = 1
Result = PASS
itsmystyle@Frigga:~/test_cuda/NVIDIA_CUDA-9.0_Samples/1_Utilities/deviceQuery$
```

Part II. Frequent Itemset Mining with GPGPU

Implementation

Since the Eclat algorithm has not change much compare to my homework 1, I will not explain much in Eclat algorithm. The only different is the summation function, I have changed it to GPU summation. For more information about Eclat algorithm, please refer to my homework 1 report. I will mainly focus on explaining my PyCuda kernel function.

First, I change the np.sum() to my gpusum(). Which input arguments are data, Data size, Block number and Thread number.

For my parallel kernel, I implemented the parallel summation with reduction techniques according to the hint. But there is a bit problem in the hint sample code. If data size is larger than block number * thread number * 2, we must add multiple rounds to sdata, but I found that there is synchronize problem when performing save data into sdata. So I decided to add the input data and save it to a temporary memory, then move the temporary into sdata at the end of while loop.

```
mod = SourceModule("""
    __device__ void warpReduce(volatile unsigned int *sdata, int tid, int blockSize) {
        if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
        if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
        if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
        if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
        if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
        if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
    }

    __global__ void sum(unsigned short *g_idata, unsigned int *g_odata, int *DATA_SIZE) {
        extern __shared__ unsigned int sdata[];

        const int tid = threadIdx.x;
        unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
        unsigned int tmp = 0;
        unsigned int gridSize = blockDim.x*2*gridDim.x;

        while(i < *DATA_SIZE){
            tmp += (g_idata[i] + g_idata[i + blockDim.x]);
            i += gridSize;
        }
        sdata[tid] = tmp;
        __syncthreads();

        if (blockDim.x >= 512) {
            if (tid < 256) { sdata[tid] += sdata[tid + 256]; }
            __syncthreads();
        }
        if (blockDim.x >= 256) {
            if (tid < 128) { sdata[tid] += sdata[tid + 128]; }
            __syncthreads();
        }
        if (blockDim.x >= 128) {
            if (tid < 64) { sdata[tid] += sdata[tid + 64]; }
            __syncthreads();
        }

        if(tid < 32) warpReduce(sdata, tid, blockDim.x);

        if(tid == 0) g_odata[blockIdx.x] = sdata[0];
    }
""")
```

Since the input data is a bitvector which is a list of 1 and 0, so I decided to make the cpu-gpu IO faster with change integer to unsigned short. I remove the template and change the BlockSize to blockDim.x.

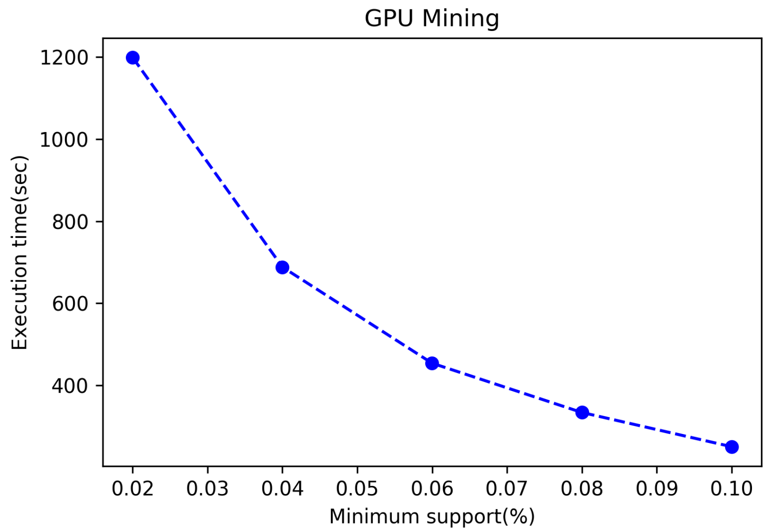
```
sum_ = mod.get_function('sum')
sum_(drv.In(data), drv.Out(result), drv.In(DATA_SIZE), block=(THREAD_NUM, 1, 1), grid=(BLOCK_NUM, 1), shared=THREAD_NUM*4)

return np.sum(result)
```

The finally return the result over numpy summation.

Graph

- **Different minimum support. (Thread number 128, Block number 128)**

Min sup.	0.1%	0.08%	0.06%	0.04%	0.02%
Time(s)	250.151	333.610	453.652	688.114	1199.028

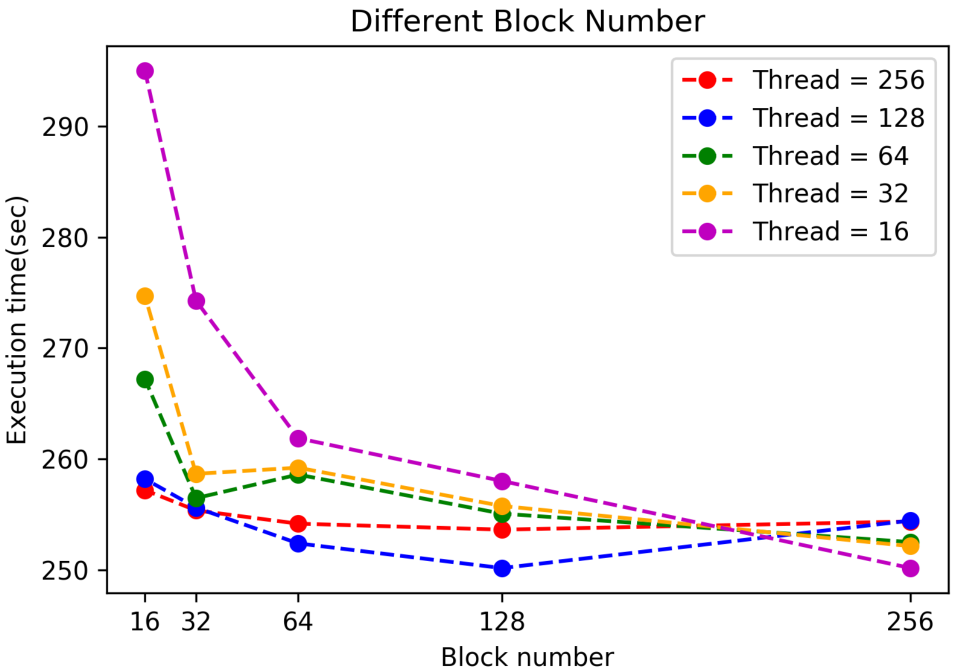


The figure above showed us the execution time with different minimum support. The execution time and minimum support have positive correlation. This is because the smaller the minimum support, the more itemset that GPU has to operate.

- **Different block and thread number. (Minimum support 0.1%)**

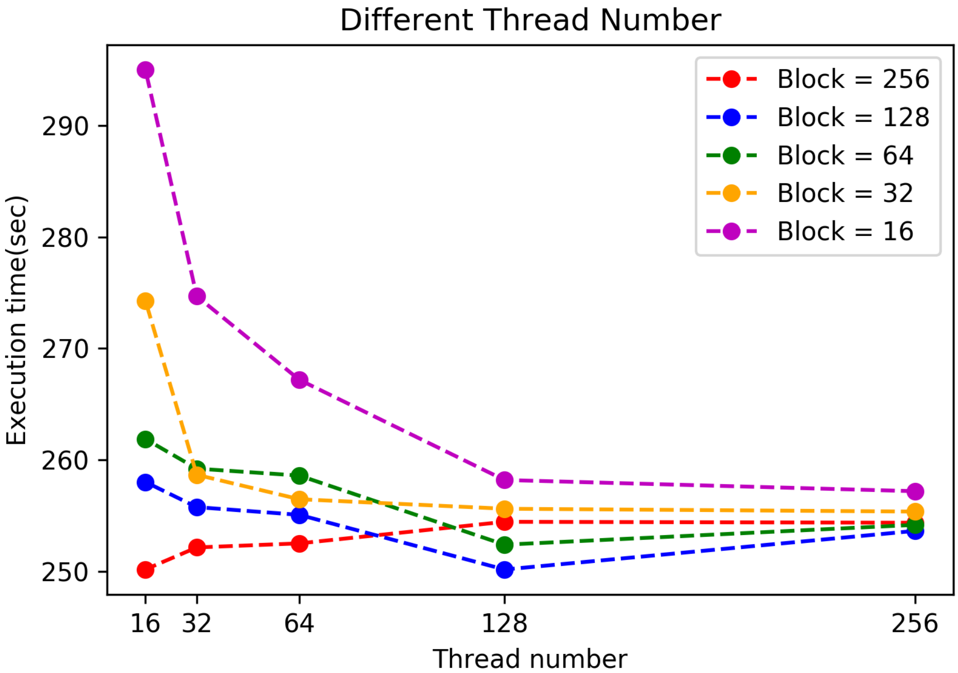
T/B	**256**	**128**	**64**	**32**	**16**
256	254.351	253.618	254.158	255.354	257.182
128	254.434	250.151	252.379	255.607	258.187
64	252.498	255.059	258.585	256.461	267.181
32	252.133	255.752	259.207	258.654	274.687
16	250.162	258.007	261.863	274.272	295.005

  - **Different block number.**



The figure above described the execution time of different block number. With all five lines, we can conclude that the execution time decreased over block number increment. Especially the purple line (thread = 16), the execution time decrease rapidly according to block number because the many the block are, the many the threads can be operating in the same time.

- **Different thread number.**



The figure above described the execution time of different thread number. We can conclude that the execution time decreased over thread number increments. Especially for the purple line (block = 16), the execution time decrease rapidly according to thread number because the many the thread are, the more the operation power can be operate in the same time.