Data Mining HW2 Report

電機碩二　r06921048　李友岐

Part 1.

```
./deviceQuery Starting...

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

Detected 2 CUDA Capable device(s)

Device 0: "GeForce GTX 1080 Ti"
  CUDA Driver Version / Runtime Version          9.0 / 8.0
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 11166 MBytes (11707875328 bytes)
  (28) Multiprocessors, (128) CUDA Cores/MP:     3584 CUDA Cores
  GPU Max Clock rate:                            1709 MHz (1.71 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:                     Yes
  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
  Device PCI Domain ID / Bus ID / location ID:   0 / 2 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

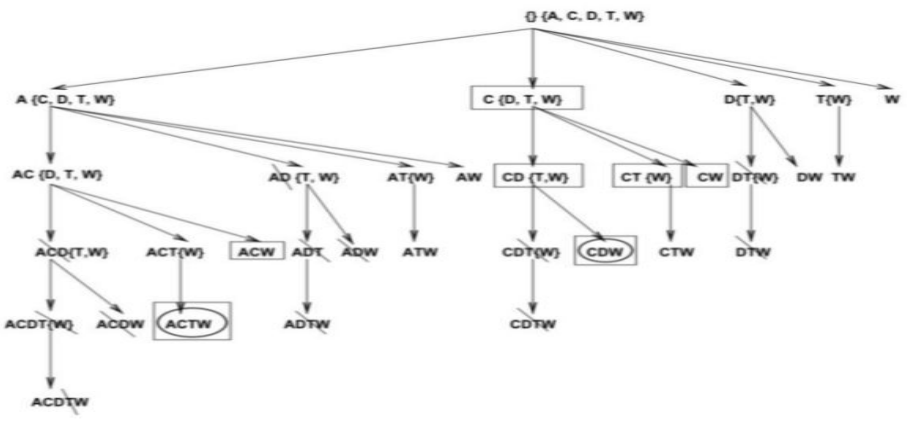
```
Device 1: "GeForce GTX 1080 Ti"
  CUDA Driver Version / Runtime Version          9.0 / 8.0
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 11172 MBytes (11715084288 bytes)
  (28) Multiprocessors, (128) CUDA Cores/MP:     3584 CUDA Cores
  GPU Max Clock rate:                            1709 MHz (1.71 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
  Device PCI Domain ID / Bus ID / location ID:   0 / 129 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
> Peer access from GeForce GTX 1080 Ti (GPU0) -> GeForce GTX 1080 Ti (GPU1) : No
> Peer access from GeForce GTX 1080 Ti (GPU1) -> GeForce GTX 1080 Ti (GPU0) : No

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 9.0, CUDA Runtime Version = 8.0, NumDevs = 2, Device0 = GeForce GTX 1080 Ti,
 Device1 = GeForce GTX 1080 Ti
Result = PASS
```

Part 2. (I use my own code)

(1) Design and Implementation:

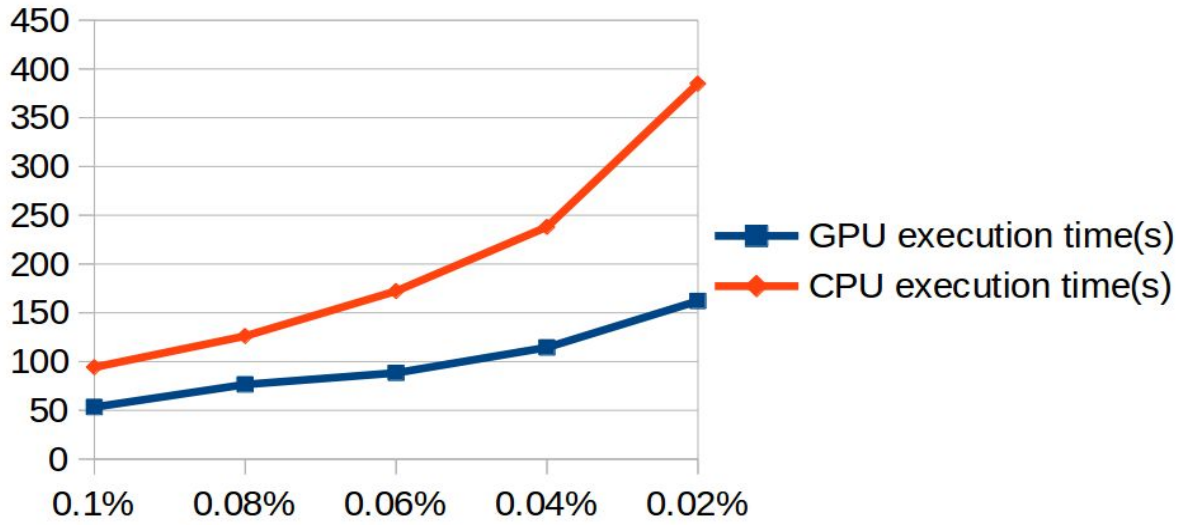
The algorithm is implemented in Python and Python's built-in container set, list and numpy.array are the data structures we used. In Eclat algorithm, we store the information with vertical data layout. Each item has a data set to record which rows contain it. For example, if a number shows up in row 2, row 3 and row 5, then its data set is equal to {2, 3, 5}. If we want to check whether the new itemset generated from these two items is frequent, we can just use the intersection of the data set of these two items. The support number of new itemset is the length of its data set, which is the intersection of the data set of these two items. For example, if item 1 and item 2 are included in {1, 2, 3} and {2, 3, 4}, respectively. Then itemset [1, 2] is included in the intersection of {1, 2, 3} and {2, 3, 4}, which is {2, 3}. Therefore, the support number of itemset [1, 2] is the element length of {2, 3}, which is 2. To expand the itemset, we define a function taking current itemset and item index as parameters. If the new itemset is frequent, then we keep calling this function to expand the itemset until it becomes not frequent. If the new itemset is not frequent, then there is no need to expand it. To speed up the process, we can compute the different parts independent with each other in parallel. During the process, we use a for loop to check whether the current itemset can be expanded with each item. In CPU mining, the computation in a for loop can be done one after one only. In GPU mining, however, the computation can be done in parallel. Take the figure in hw1 as example, for A {C, D, T, W}, we can compute the data set of AC, AD, AT and AW at the same time with GPU. In CPU version, we can only compute them one after one with a for loop. In GPU version, we store the data set of A into a numpy.array and store the data set of {C, D, T, W} into another numpy.array. Next, we pass them to PyCuda to compute the intersection of each combination in parallel and pass back the array with result. Finally, we check whether it is frequent for each combination. If this combination is frequent, then we keep expanding it by calling the same function with different item index. (To expand A, we pass the index of C and therefore we start from C. To expand AC, we pass the index of D and therefore we start from D.)



(2) Plot:

1. Execution time versus Minimum support

Minimum Support	GPU execution time(s)	CPU execution time(s)
0.1%	53.29	94.16
0.08%	76.50	126.08
0.06%	88.37	172.23
0.04%	114.40	238.05
0.02%	162.15	384.86

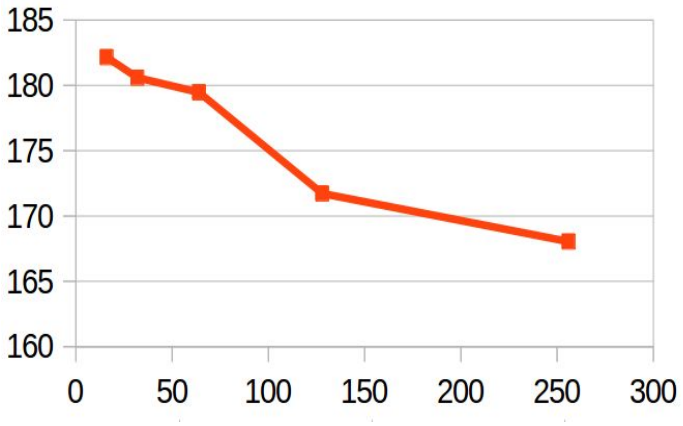


When the minimum support becomes smaller, the number of the items we need to compute increases. Therefore, the execution time also increases. Since GPU version has parallel design, it is extremely faster than the origin CPU version.

2. Execution time versus Block number

(Set minimum support =0.0002, Thread number=256)

Block Number	Execution Time (s)
16	182.15
32	180.58
64	179.46
128	171.72
256	168.04

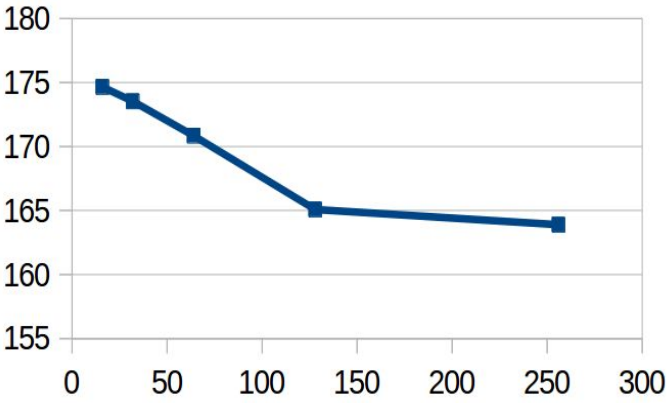


For each block number, we run 20 times and then compute the average.
Note that the execution time of CPU version under minimum support=0.0002 is 384.86s.
If we have more blocks, then typically we can compute more data in parallel. As a result, the execution time reduces as the block number increases.

3. Execution time versus Thread number

(Set minimum support =0.0002, Block number=256)

Thread Number	Execution Time (s)
16	174.65
32	173.53
64	170.84
128	165.08
256	163.89



For each thread number, we run 20 times and then compute the average.
Note that the execution time of CPU version under minimum support=0.0002 is 384.86s.
If we have more threads, then typically we can compute more data in parallel. As a result, the execution time reduces as the thread number increases.