CUDA-based Parallel Programming

2015 Fall computer architecture project II on parallel computing

Shen Jiyuan
5130309194
sheryalyuan@126.com

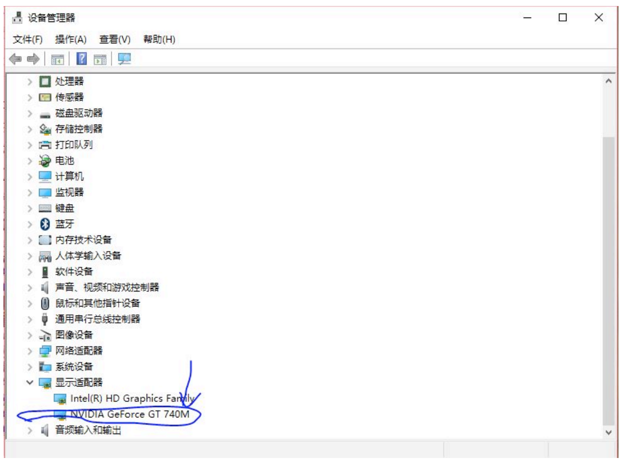
***Abstract*—The string searching problem is implemented here with the acceleration with GPU for which we program on the CUDA platform where parallel computing is focused on.**

***Index Terms*—cuda, string search, kmp, string match, parallel.**

I. Introduction of CUDA

CUDA, an abbreviation for Compute Unified Device Architecture, is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use a CUDA-enabled GPU for general purpose processing - an approach known as GPGPU. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements.

The platform is designed to work with programming languages such as C, C++ and Fortran. This accessibility makes it easier for specialists in parallel programming to utilize GPU resources, as opposed to previous API solutions like Direct3D and OpenGL, which required advanced skills in graphics programming. Also, CUDA supports programming frameworks such as OpenACC and OpenCL.



Different GPUs and cards support different computing capability, as well as the version of CUDA. Thus you may first check the link listed if your machine satisfy the NVIDIA requirement (https://developer.nvidia.com/cuda-gpus).

II. Environment Setting and Preparations

Following installations and implementations are based on operating system - Windows; architecture x86_64; version 10; installation type exe(local). Next you will experience the installing and setting process in the sequence as : Visual Studio 2013; NVIDIA Driver; CUDA 7.5 Toolkit.

A. Installation of Visual Studio 2013

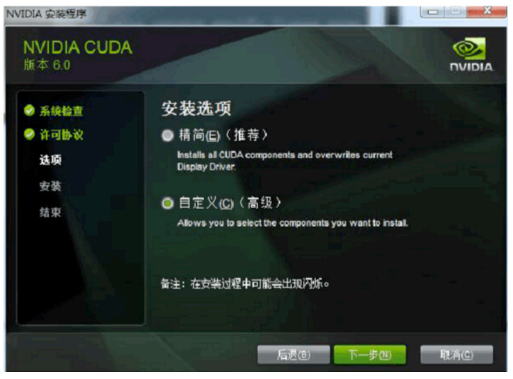
1) Download the local executable file from the website: https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx
2) Double click the local executable file, and follow the installation tips. Then you will get the tool installed.

B. Installation of CUDA Driver

1) Download the local executable file from the website: http://www.nvidia.cn/Download/index.aspx?lang=cn
   Here according to my machine, I chose GeForce—GeForce 700M Series(Notebooks)—GeForce GT 740M—Windows 10 64-bit—Chinese(simplified)
2) Double click the executable file, and simply follow the Installation tips. Again the process is perpetual.

C. Installation of CUDA 7.5 Toolkit

1) Download the local executable file from the website: https://developer.nvidia.com/cuda-downloads/
2) Double click the local executable file, and once again please follow the tips. And you need to pay attention to this step (CHOOSE THE SECOND ONE for SDK!) :



安装选项

精简(E)（推荐）
Installs all CUDA components and overwrites current Display Driver

自定义(C)（高级）
Allows you to select the components you want to install.

备注：在安装过程中可能会出现闪烁。

D. Environment Configuration

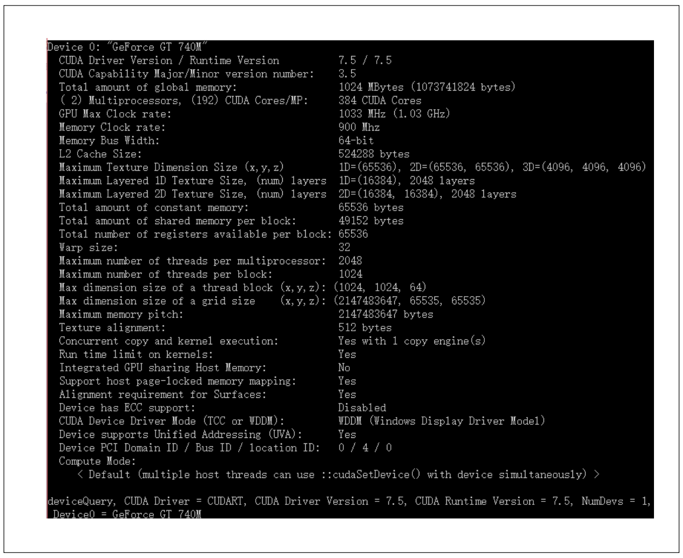
1) Configure the system variables:
   1. The installation above has already configured two system variables:
      CUDA_PATH ; CUDA_PATH_V7_5
   2. We need to add five system variables:
      CUDA_BIN_PATH : %CUDA_PATH%\bin
      CUDA_LIB_PATH : %CUDA_PATH%\lib\x64
      CUDA_SDK_BIN_PATH : %CUDA_SDK_PATH%\bin\win64
      CUDA_SDK_LIB_PATH : %CUDA_SDK_PATH%\common\lib\x64
      CUDA_SDK_PATH : C:\Program Files\NVIDIA Corporation\Installer2\CUDASamples_7.5
   3. Add the follow string to the 'path' system variable:
      ;%CUDA_LIB_PATH%;%CUDA_BIN_PATH%
      ;%CUDA_SDK_LIB_PATH%
      ;%CUDA_SDK_BIN_PATH%
2) If you would like to configure a more cuda-friendly programming environment, you can refer to the website given next(the steps are really clear):
   http://ju.outofmemory.cn/entry/97435

E. Test for your Installation

There are lots of sample programs in the SDK. Run the samples for testing the installation.

III. Parameters of My Device

Run the sample program in "1_Utilities" named "device Query" . And the device parameters are shown:



```
Device 0: "GeForce GT 740M"
  CUDA Driver Version / Runtime Version          7.5 / 7.5
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 1024 MBytes (1073741824 bytes)
  ( 2) Multiprocessors, (192) CUDA Cores/MP:     384 CUDA Cores
  GPU Max Clock rate:                            1033 MHz (1.03 GHz)
  Memory Clock rate:                             900 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 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 1 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
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 4 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 7.5, CUDA Runtime Version = 7.5, NumDevs = 1,
 Device0 = GeForce GT 740M
```

(If it not clear in the picture, main parameters we focus on in this project are listed)

1. Max dimension size of a thread block:
   (x,y,z): (1024,1024,64);
2. Max dimension size of a grid size:
   (x,y,z):(2147483647,65535,65535);

IV. String Search Problem

In computer science, string search algorithms, sometimes called string match algorithms, are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text.

Here we use CUDA to accelerate a simple string searching problem. Given two strings S and P, we need to find out the positions where pattern string P appears in S. For example, S="ababa", P="aba", the position would be 0 and 2. Note that here we allow overlapping for pattern strings in S.

We assume that each character in S has an ASCII value in the range [37,126], and is a printable character. And we assure that the character '$' does not appear in the input. M denotes the length of S, which is smaller than 40,000,000. And N denotes the length of P which is smaller than 4,000. We assume that M>>N. The input file is "input.txt", which contains two lines. The first line is string S and the second line is string P. The answer would be stored in "output.txt". Each line contains a number, and the numbers are in ascending order. If P does not appear in S, then output "Not found".

V. String Search Algorithms overview

String Search Algorithms are proverbially explored in many areas, and here we have a very short overview and summary of some well-known ones.

A. Brute Force (bf)

It is the most intuitive string match method, and no preprocess is included here. A simple but inefficient way to see where one string occurs inside another is to check each place it could be, one by one, to see if it's there. In this case, the time complexity is Θ((n-m+1) m); n for text length; m for pattern length.

Then : O(0) ; O(N*M)

B. Boyer-Moore (bm)

The algorithm preprocesses the string being searched for (the pattern), but not the string being searched in (the text). It is thus well-suited for applications in which the pattern is much shorter than the text or where it persists across multiple searches. It uses bad-character-rule and good-suffix-rule to get a preprocessing result. And do the matching by skip unnecessary positions.

Then : O(N+M^2) ; O(N)

C. Knuth-Morris-Pratt (kmp)

When a mismatch occurs, the pattern itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters. The preprocessing stage deals with the pattern for mismatch situations.

Then : O(M) ; O(N)

D. Rabin-Karp

It uses hashing to find any one of a set of pattern strings in a text. Then : O(0) ; O(N*M)

E. Indeed-simple summary of algorithms

The table lists the time complexity of standard implements of some popular string match algorithms:

Algorithm	Preprocess_time	Match_time
Brute Force	O(0)	Θ((n-m+1)*m)
Boyer-Moore	Θ(m+k)	Ω(n/m);O(n)
Knuth-Morris-Pratt	Θ(m)	Θ(n)
Rabin-Karp	Θ(m)	Ω(n+m);O((n-m)m)
Finite-state automaton	Θ(mk)	Θ(n)
Bitap	Θ(m+k)	O(mn)

TABLE I. Comparison of Algorithms

I choose KMP algorithm to implement on CUDA platform after the comparison from the complexity table.

VI. Deep discussion on NON-parallel KMP

A. Preprocessing

The preprocessing of KMP deals with the pattern and it will get a partial match table (in the project I call it '*next' in the host). The table aims at allowing the algorithm not to match any character of text more than once.

The key observation about the nature of a linear search that allows this to happen is that in having checked some segment of the main string against an initial segment of the pattern, we know exactly at which places a new potential match could continue to the current position or could begin prior to the current position. In other words, the partial match table denotes all possible fallback positions that bypass maximum of hopeless characters while not sacrificing any potential matches in doing so. My CPU-based codes for preprocessing is listed:

```
void getNext(char *pattern, int pattern_len, int *next)
{
    int len = 0, i;
    next[0] = 0; i = 1;
    while (i < pattern_len)
    {   if (pattern[i] == pattern[len])
        {
         len++;
         next[i] = len;
         i++;
        }
        else // (pat[i] != pat[len])
        {    if (len != 0)
             {
               len = next[len - 1];
             }
             else
             {
               next[i] = 0;
               i++;}}
    }
}
```

B. Matching

The matching process is feasible to understood. Each time we care about two characters. If they match, both the label index in text and pattern will increase; otherwise, check the partial match table to get the next match position and start matching again.

My CPU-based codes for matching is listed:

```
void KMP(char *pattern, int pattern_len, char *text, int
text_len, int *answer, int *next)
{
            int j = 0;//j as index for pattern
            int i = 0;//i as index for text
            while (i < text_len)
            {
                        if (pattern[j] == text[i])
                        {
                                    j++;
                                    i++;
                        }
                        if (j == pattern_len)
                        {
                                    answer[i - j] = 1;
                                    j = next[j - 1];
                        }
                        else if (pattern[j] != text[i])
                        {
                                    if (j != 0)
                                         j = next[j - 1];
                                    else
                                         i= i+ 1;
                        }
            }
}
```

C. Analysis of complexity

For preprocessing: The key factor is the while loop whose time complexity is O(pattern_length). And within each loop, 'len' only be changed once which means the number of change of 'len' equals to the time complexity of getNext funcion. (Matched) Because 'i++' executes at most (pattern_length-1), 'len++' too. (Unmatched) Only when 'len=next[len];' the variable 'len' will decrease. And because 'len' increases at most (pattern_length-1), it decreases at most (pattern_length-1). Therefore, the time complexity of getNext is O(pattern_length).

For matching, it processes from the first character to the last one. And each move will be forward. Thus, the time complexity is O(text_length).

In summary, (the whole time complexity) = (the time complexity of preprocessing + the time complexity of matching) = (O(pattern_length) + O(text_length))

VII. Parallel implementation of KMP

The parallel implements of KMP is based on the non-parallel one which means the main idea will not change. We will still have both getNext and KMP functions. And what we should pay attention to are - the block and thread setting; the threads allocation method for computing.

A. Setting and Discussion

In my coding, the preprocessing function for the partial match table is exactly the same as the non-parallel one. Because the time complexity is O(pattern_length), I think compared to execute the process every time in a matching, the partial table could be just calculated once in the host ‘main’. And by sacrificing the space for delivering arrays in size pattern_length, we can avoid thread_number*O(pattern_length) computing!
My CUDA-based codes for block_thread and KMP is listed:

```
//======device KMP search function=====//
__device__ void KMP(char *pattern, int pattern_len, char
*array, int array_len, int *answer, int *next, int cursor, int
end)
{   //cursor as index for array
    int j = 0;//j as index for pattern
    while (cursor < end)
    {
        if (pattern[j] == array[cursor])
        {
             j++; cursor++;
        }
        if (j == pattern_len)
        {
             answer[cursor - j] = 1;
             j = next[j - 1];
        }
      else if (pattern[j] != array[cursor])
           {
             if (j != 0) j = next[j - 1];
             else cursor = cursor + 1;
           }
      }
}
//========global kernel function=========//
__global__ void kmp_kernel(char *arrayIn, char
*patternIn, int *answerIn, int *next, int array_len, int
pattern_len)
{
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    int offset = 2 * pattern_len, cursor, end;

    if (id < 0.5*(array_len / pattern_len))
    {
         cursor = id*offset;
         end = id*offset + offset;
    }
    else
   { //辅助线程(aid threads)
         cursor = (id % ((array_len / pattern_len) /
2))*offset + offset - pattern_len;
         end = (id % ((array_len / pattern_len) / 2))*offset
+ offset + pattern_len;
   }

   KMP(patternIn, pattern_len, arrayIn, array_len,
answerIn, next, cursor, end);
}
//===================================//
```

My setting for threads is that each thread computes one pattern_length string. According to a paper ‘An acceleration with GPU’, I set the ‘thread_per_block’ be 1024 (while the maximum limit is 2048). As for each time I set one pattern_length_size set of characters to be computed in one thread, which requires 2*pattern_length characters read by a KMP search function in one thread.

B. Complexity Analysis

Time Complexity

Here simple arithmetic calculations are considered constant.
1. Input file processing:
    Loop for two array sizes: O(m+n)
    Loop for two array initialization: O(m+n)
    Answer array initialization: O(n)
2. Preprocessing:
    (deep analysis is the same as the non-parallel one)
    O(pattern_length)=O(m)
3. Memory copy from host to device:
    Two arrays: O(m+n)
4. Kernel function:
    The threads are computed parallel.
    (deep analysis as the non-parallel one, just here the text_ length is instead by 2*pattern_length)
    Each thread: O(2*m)
5. Memory copy from host to device:
    One array: O(m)
6. Output file processing:
    Output each result to the file by a loop: O(n)

In conclusion, we can get the total time complexity by adding them up = O(7*m+5*n) = O(m+n), when m,n>>0
= O(n) , when n>>m

Space Complexity

Here the temp variables defined for input use, output use or index is not considered for they could be emit in space.
1. Host array allocation: O(2*n+m+k)
    Two text_length arrays
    One pattern_length array
    k stands for variables in constant operations
2. Device global array allocation: O(2*n+2*m+k)
    Two text_length arrays
    Two pattern_length arrays
    k stands for variables in constant operations
3. Kernel allocation: O(m/n+k)
    Thread for each block: 1024
    Block number: O((m/n)/1024)
    k stands for variables in constant operations

In conclusion, the space complexity = O(m+n) , when m,n>>0
= O(n) , when m>>n
Notice that minimal unit is ‘char’, namely ‘1-byte’.

VIII. Tips On TestFiles

Notice that the input file should be named ‘input.txt’. And there are two lines in the file. You can refer to the toy codes in C++ listed here to generate your test file if you like. However, I have to clarify that in this way it would be likely to generate many ‘not found’ cases.

```
int main()
{
        ofstream fout;
        fout.open("input.txt");
        char *str =
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZa
bcdefghijklmnopqrstuvwxyz";
        srand(time(NULL));
        for (int temp = 0; temp < 38000000; temp++)
        {
                fout<<str[rand() % 62];
        }
        fout << '\n';
        fout << 'a';fout << 'a'; fout << 'b';
        fout.close();
        return 0;
}
```