Processing Posting Lists Using OpenCL

By

Radha Kotipalli

Spring 2015

Project Advisor : Dr. Chris Pollett

Department of Computer Science

San José State University

Contents

Introduction

The main goal of this project is to create a GPU-based, parallel reader of posting lists for Yioop. Yioop is an open source search engine designed and developed in PHP by Dr. Chris Pollett. We will be using OpenCL to code for the GPU. We will then do experiments comparing our GPU posting list handler with the existing mechanism in Yioop.

Search engines use an inverted index to look up for a term all the documents containing that term. This list of documents is called a posting list. Posting lists are typically stored in a compressed binary format.

OpenCL (Open Computing Language) provides a platform for parallel programming and GPUs have hundreds or thousands of processing units rather than one to eight for a typical CPU. OpenCL provides an effective way to program GPUs and to send and receive data from the CPU in a system and a GPU. Many posting list operations can be done in parallel. So implementing posting lists algorithm using OpenCL will likely improve the performance of search engine.

In order to prepare for the actual implementation, I performed few code experiments during this semester. There were four different deliverables in my CS297 which will help me understand the requirements involved in the actual implementation of the project, that will be done in CS298.

The first deliverable was to understand different algorithms for index compressions from the book, " Information retrieval - Implementing and Evaluating Search Engines," by Stefan, B., Clarke , C., & Cormack, G. I also wrote a C-program to implement the Huffman encoder and decoder. The second deliverable was to write a C-program to encode/decode posting lists using the Simple-9 compression algorithm by taking a word and its index positions. The third deliverable was to install C++ bindings for Microsoft Visual studio and write a OpenCL program to compute the squares of elements of an array. Finally the fourth deliverable was to install PHP extensions for Windows and write a simple PHP extensions for the hello world program from the book, "Extending and Embedding PHP," by Golemon, Sara. I will explain those deliverables in detail in the later sections of this paper.

Overview of Deliverables

Deliverable 1: Index Compression and Huffman encoder/decoder

This deliverable was to understand the purpose of compression inverted index and various compression algorithms.

Inverted index consists of two principal components, dictionary and posting lists. Uncompressed inverted index for a given document can be very large, sometimes even greater than the actual source itself. Compressed inverted index can provide advantages such as less storage space requirement, fast query retrieval time, and can accommodate large collections.

Data compression algorithm takes the data and converts into another data which requires fewer bits to store and transfer. It contains an encoder and a decoder. Encoder converts original data A into B. Decoders are two types. First one is lossy which takes B and converts into C, where C can be approximation of A. Examples are JPEG, Mp3. Second one is lossless, which takes B and converts back to A.

One of the compression algorithms is Huffman coding algorithm. Huffman algorithm calculates probability of frequency of characters and using that to find the code for each character. It uses prefix property, no code word is an initial substring of any other code word. Example a-0, b-11, c-100, d-101.

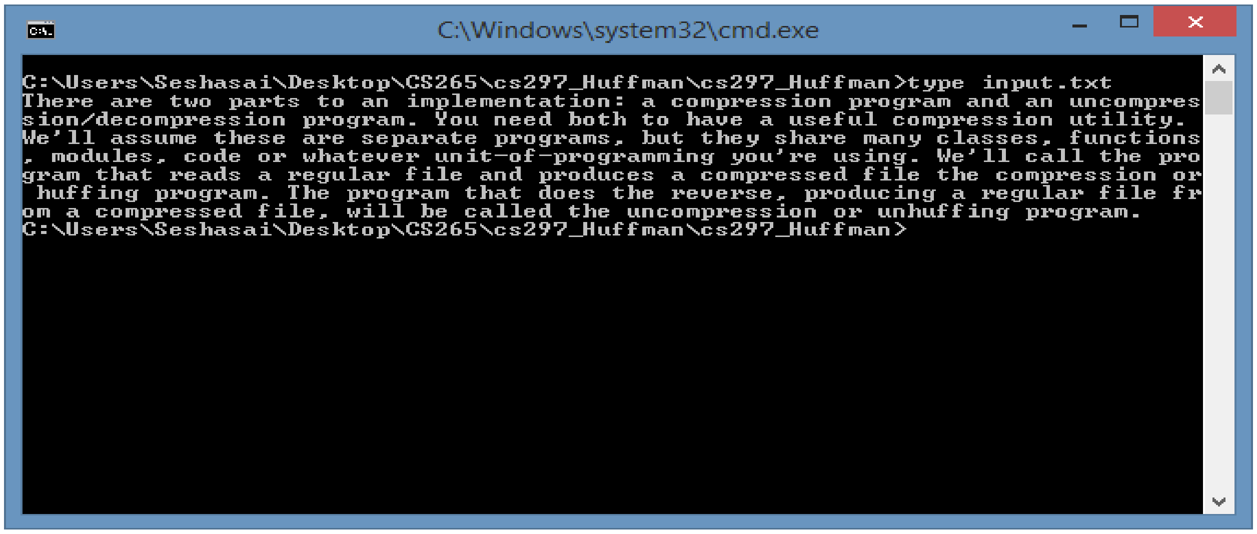
Huffman Coding Algorithm to build a tree:

- Generate probability of occurrences of each character
- Create each individual node with the probability
- Find two minimum nodes and combined them into one node with sum of their probabilities

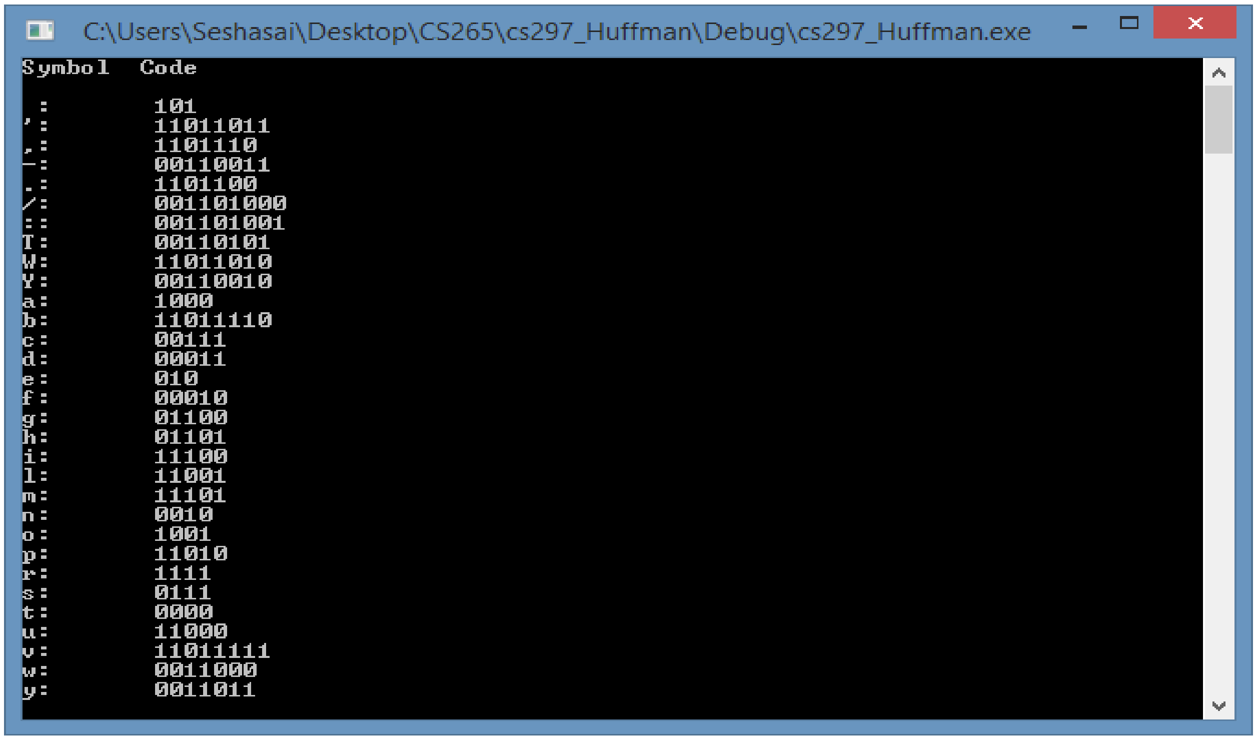
- Two minimum nodes become left and right nodes.
- Repeat it until ends with a single node

I have implemented this algorithm by assuming that the document contains only the characters whose ASCII values are from 32 -126 i.e from space to '~'. For a given input text file my program creates a Huffman-tree, a code for each symbol/character and copy them to a output file. The first 8 bits in the output file contains the starting address of the actual encoded documentation, followed by the Huffman-tree and then the encoded input file. This format makes decoding easy because the output file contains the codes of each symbol/character that has been used in encoding so the output file itself is sufficient to decode without using any auxiliary code.

Example: input.txt contains following text



```
C:\Users\Seshasai\Desktop\CS265\cs297_Huffman\cs297_Huffman>type input.txt
There are two parts to an implementation: a compression program and an uncompres
sion/decompression program. You need both to have a useful compression utility.
We'll assume these are separate programs, but they share many classes, functions
, modules, code or whatever unit-of-programming you're using. We'll call the pro
gram that reads a regular file and produces a compressed file the compression or
 huffing program. The program that does the reverse, producing a regular file fr
om a compressed file, will be called the uncompression or unhuffing program.
C:\Users\Seshasai\Desktop\CS265\cs297_Huffman\cs297_Huffman>
```


C:\Users\Seshasai\Desktop\CS265\cs297_Huffman\Debug\cs297_Huffman.exe

```
Symbol   Code

 :        101
':        11011011
,:        1101110
-:        00110011
.:        1101100
/:        001101000
::        001101001
T:        00110101
W:        11011010
Y:        00110010
a:        1000
b:        11011110
c:        00111
d:        00011
e:        010
f:        00010
g:        01100
h:        01101
i:        11100
l:        11001
m:        11101
n:        0010
o:        1001
p:        11010
r:        1111
s:        0111
t:        0000
u:        11000
v:        11011111
w:        0011000
y:        0011011
```

Posting lists can be very large and each element occurs only single time. So standard compression methods like Huffman coding is not feasible. Several other algorithms are available to compress the posting lists such as γ-codes, Golomb/Rice codes, byte-aligned codes, and word-aligned codes. The index positions in the posting lists can be replaced with Δ-values, transformed into an equivalent sequence of difference between consecutive elements, so that elements can be smaller and can be encoded using fewer bits.

Deliverable 2: Compressing Posting List Using Simple-9 Algorithm

The purpose of this deliverable was to write a C-program to encode/decode posting lists using Simple-9 compression algorithm, which is almost same algorithm used in Yioop to create its posting lists.

Generally compressed posting list is stored using an integral number 16-bit, 32-bit, or 64-bit machine words. Instead of storing each Δ-value in a separate 32-bit, word-aligned code compression algorithm allows you to store several consecutive values as a single 32-bit. Simple-9 is one of the examples of word-aligned code compression algorithm,

Simple-9 inspects the Δ-values in a posting sequence and tries to squeeze as many of them as possible into a 32 bit machine word. In this 32 bit, 4 bits are reserved for a selector, which tells how many Δ-values of equal size have been inserted in the remaining 28 bits. There are nine different ways of dividing then into chunks of equal size.

Selector	0	1	2	3	4	5	6	7	9
Number of Δ's	1	2	3	4	5	7	9	14	28
Bits per Δ	28	14	9	7	5	4	3	2	1
Unused bits per word	0	0	1	0	3	0	1	0	0

Example: Input Word: "The" 1624 1650 1876 1972 2356

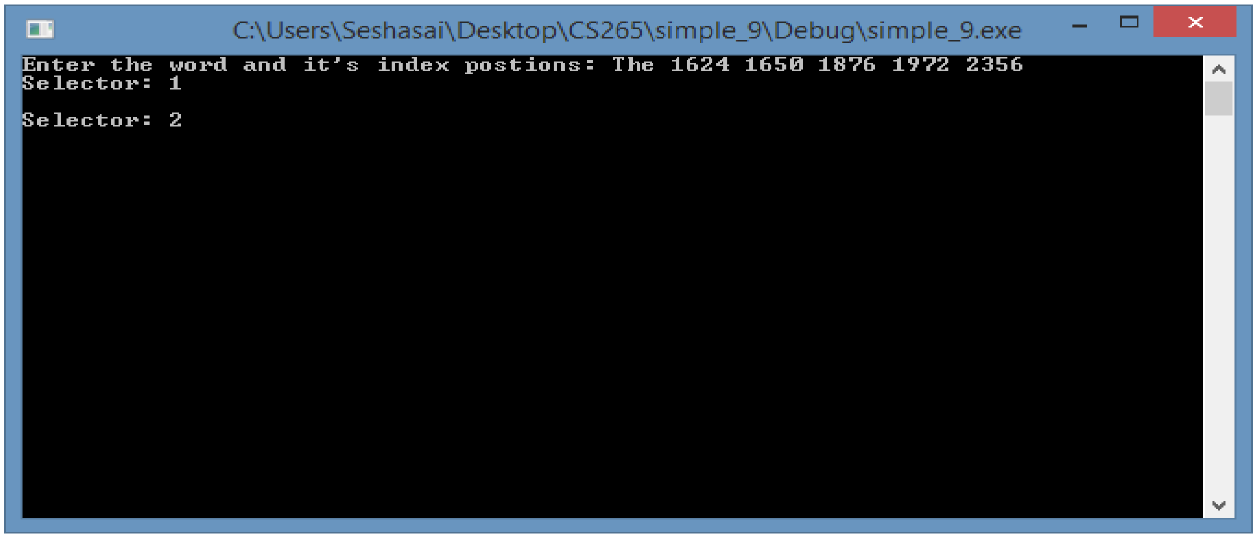
Δ-values : 1624 25 225 95 383 [Δ-value: [1650 -1624 -1] = 25, ...]

The above indexes can be saved as 1624 and 25 together as two 14-bits each; 225, 95, and 383 together as three 9-bits each, and one unused bit at the end.

Encoding: In my program I have written a method which takes a maximum of 28 Δ-values, Since we can insert a maximum of 28 different Δ-values (if they are 1 bit each), at a time and returns its appropriate selector value.

```
/*
 This method finds the selector that can be used to store the bits
*/
int selector(int linkCount, int cnt[])
  {
        //Number of possible inserted value in 28-bits
        int counter[8]={28,14,9,7,5,4,3,2};
        //Maximum number of bits
        int check[8]={1,2,3,4,5,7,9,14};
        int flag=0,i;int simple=0;
        switch(linkCount)
        {
          case 28:i=0;break;
          case 14:i=1;break;
          case 9: i=2;break;
          case 7: i=3;break;
          case 5: i=4;break;
          case 4: i=5;break;
          case 3: i=6;break;
          case 2: i=7;break;
        }
        for( i; i<8;i++)
       {
          flag=0;
          for(int j=0;j<counter[i];j++)
         {
             if(cnt[j]>check[i])
            {
               flag=-1; break;
             }
          }
         if(flag!=-1)
        {
            switch(counter[i])
             {
                 case 28: simple=9;break;
                 case 14: simple=7;break;
                 case 9: simple=6;break;
                 case 7: simple=5;break;
```

```
                        case 5: simple=4;break;
                        case 4: simple=3;break;
                        case 3: simple=2;break;
                        case 2: simple=1;break;
                     }
                    return simple;
                  }
              }
     return simple;
    }
```

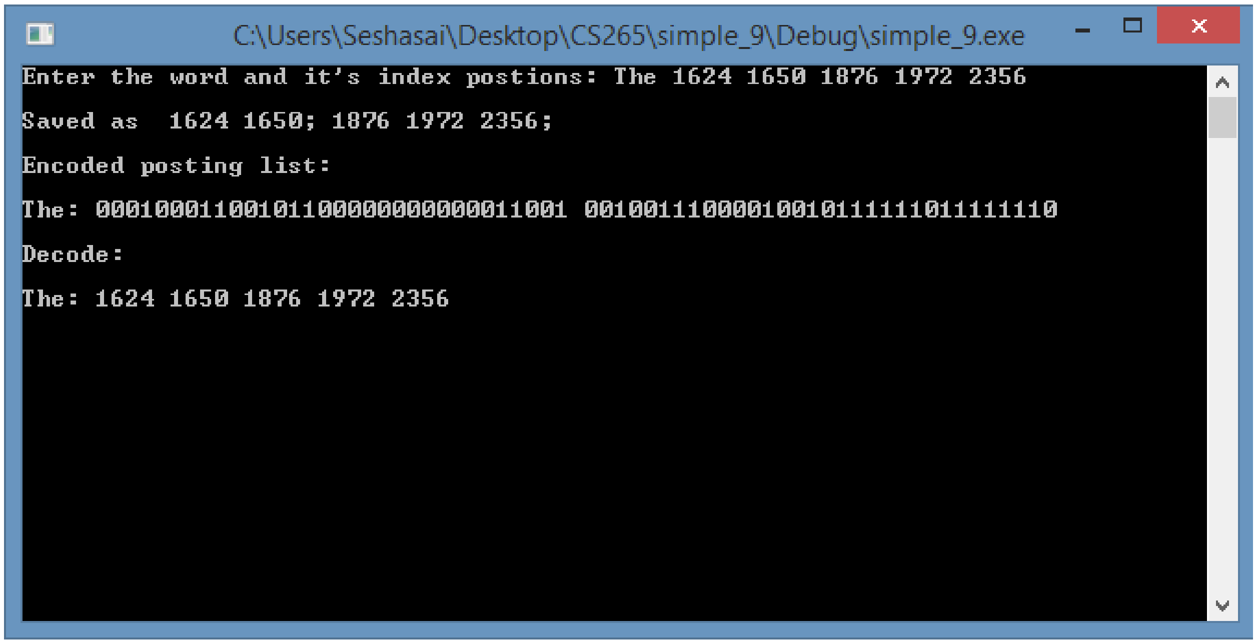


```
Enter the word and it's index postions: The 1624 1650 1876 1972 2356
Selector: 1

Selector: 2
```

Using the above selector value to convert the appropriate number of Δ-values into 28 bit chunks and repeat the process till the end of the list.

Decoding: I have taken the encoded linked list of 32 bit chunks, found the selector by reading the first 4 bits and used that selector value to split the remaining 28 bits into equal sizes and converted it back to the original index positions.



```
Enter the word and it's index postions: The 1624 1650 1876 1972 2356
Saved as  1624 1650; 1876 1972 2356;
Encoded posting list:
The: 00010001100101100000000000011001 00100111000010010111111011111110
Decode:
The: 1624 1650 1876 1972 2356
```

Deliverable 3: OpenCL Program

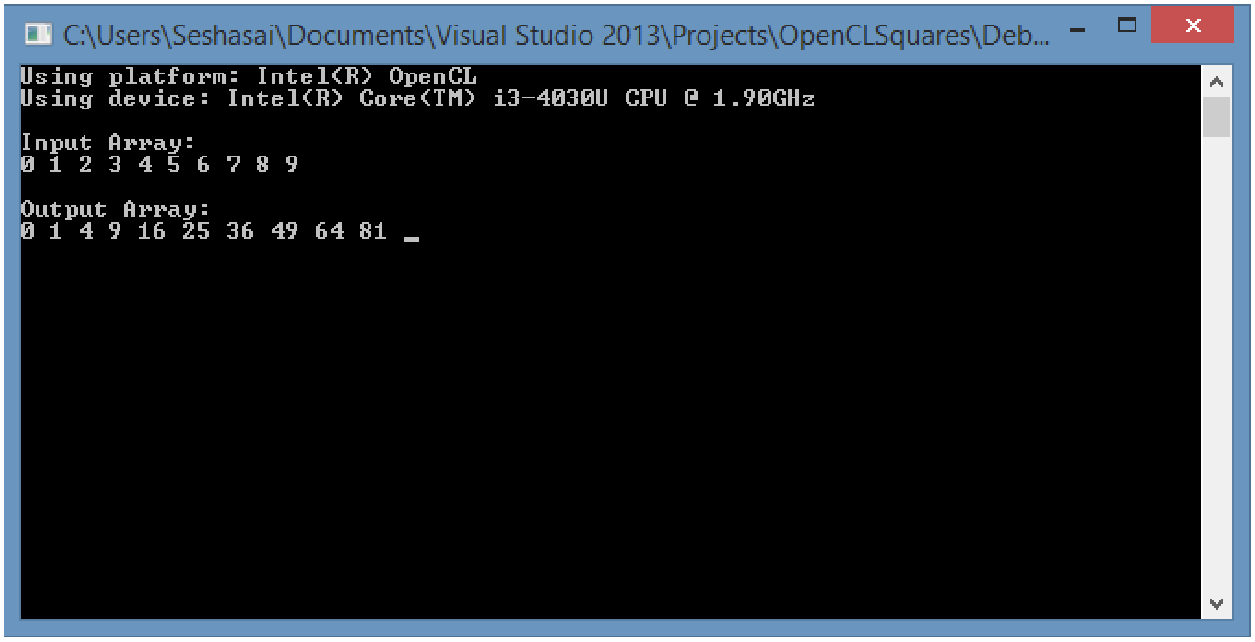
The purpose of this deliverable was to install the necessary C++ bindings for Microsoft Visual studio and write a OpenCL program to compute the squares of each element of an array.

I had installed Intel INDE Starter Edition from https://software.intel.com/en-us/intel-inde. The Starter Edition provides tools and libraries to develop OpenCL kernels for GPU. Then wrote a simple OpenCL program, which takes an array and display the squares of its elements.(Took the sample code from the website http://simpleopencl.blogspot.com/2013/06/tutorial-simple-start-with-opencl-and-c.html and modified according to my program requirements).

The first step is to get the OpenCL platform, in this case it is Intel. Next step is to get device of our platform(CPU or GPU). Next step is to create a context, a handle to device and platform. Next step is the actual OpenCL function simple_square that will calculate the squares of each element in an array.

```
// kernel calculates for each element C=A*A
string kernel_code =
       "   void kernel simple_add(global const int* A, global int* C){       "
       "       C[get_global_id(0)]=A[get_global_id(0)]*A[get_global_id(0)];  "
       "   }
```

Above code calculates square of each element by opening one thread for each element. Here get_global_id() gets the id of current thread, which can go from 0 to (array size -1).


C:\Users\Seshasai\Documents\Visual Studio 2013\Projects\OpenCLSquares\Deb...

```
Using platform: Intel(R) OpenCL
Using device: Intel(R) Core(TM) i3-4030U CPU @ 1.90GHz

Input Array:
0 1 2 3 4 5 6 7 8 9

Output Array:
0 1 4 9 16 25 36 49 64 81 _
```

Deliverable 4: PHP Extensions for a Hello World C-Program

The purpose of this deliverable was to install PHP extensions for windows and write a simple PHP extensions for the hello world program from the book, "Extending and Embedding PHP," by Golemon, Sara.

For this deliverable I had built PHP from source as per the steps given at https://wiki.php.net/internals/windows/stepbystepbuild

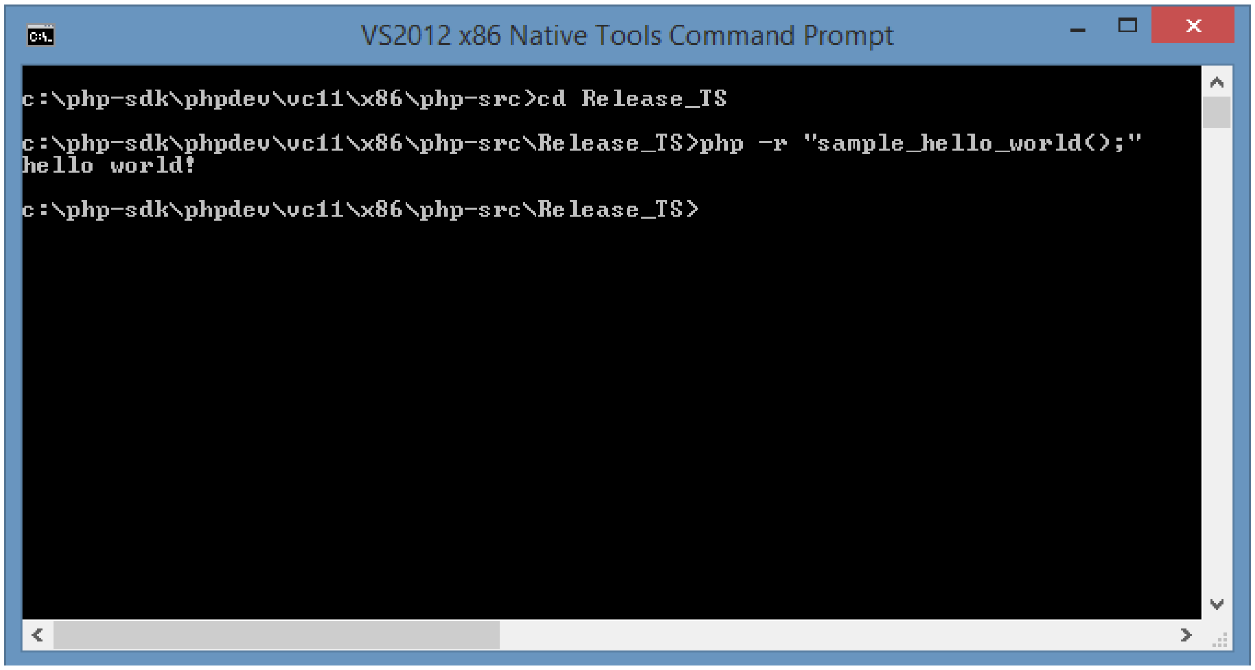
After PHP was built, I wrote a simple C-program. This program contains a function called sample_hello_world(). The code of that function is

```
PHP_FUNCTION(sample_hello_world)
{
        php_printf("hello world !\n");
}
```

Code was compiled into a dll library as per the steps given in Chapter 5, "Your First Extension" from the book "Extending and Embedding PHP". The resulting library was copied to php extension_dir specified in php.ini file. The following entries are added to the php.ini file.

```
extension_dir = "C:/php-sdk/phpdev/vc11/x86/php-src/Release_TS/ext/"
 extension=php_sample.dll
```

The C function can be invoked from PHP as below.



```
c:\php-sdk\phpdev\vc11\x86\php-src>cd Release_TS

c:\php-sdk\phpdev\vc11\x86\php-src\Release_TS>php -r "sample_hello_world();"
hello world!

c:\php-sdk\phpdev\vc11\x86\php-src\Release_TS>
```

Conclusion

Deliverable 1 and 2 helped me understand the inverted index and posting list's Simple-9 compression algorithm. OpenCL allows writing parallel programs in C-language and can exploit the power of threads of GPU. Yioop is a PHP based search engine. Custom PHP extensions will provide a way to invoke from PHP programs written in C. Deliverable 3 covered a simple OpenCL example and Deliverable 4 covered a simple PHP extension. All the four deliverables which were explained in previous sections will help me with my CS298 project implementation.

References:

1. Stefan, B., Clarke , C., Cormack, G. (2010). Information retrieval - Implementing and Evaluating Search Engines . Cambridge, Massachusetts: MIT Press.

2. Golemon, Sara. Extending and Embedding PHP. Indianapolis, Ind.: Sams, 2006. Print.