Spring 5-20-2016

Processing Posting Lists Using OpenCL

Radha Kotipalli
San Jose State University

Processing Posting Lists Using OpenCL

A Project Report

Presented to

The Faculty of Department of Computer Science

San Jose State University

In Partial fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

By

Radha Kotipalli

Spring 2016

SAN JOSE STATE UNIVERSITY

The Undersigned Project Committee Approves the Project Titled

Processing Posting Lists Using OpenCL

By

Radha Kotipalli

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

_____

Dr. Chris Pollett, Department of Computer Science Date

_____

Dr. Sami Khuri, Department of Computer Science Date

_____

Dr. Thomas Austin, Department of Computer Science Date

APPROVED FOR THE UNIVERSITY

_____

Associate Dean Office of Graduate Studies and Research Date

ABSTRACT

Processing Posting Lists Using OpenCL

One of the main requirements of internet search engines is the ability to retrieve relevant results with faster response times. Yioop is an open source search engine designed and developed in PHP by Dr. Chris Pollett. The goal of this project is to explore the possibilities of enhancing the performance of Yioop by substituting resource-intensive existing PHP functions with C based native PHP extensions and the parallel data processing technology OpenCL. OpenCL leverages the Graphical Processing Unit (GPU) of a computer system for performance improvements.

Some of the critical functions in search engines are resource-intensive in terms of processing power, memory, and I/O usage. The processing times vary based on the complexity and magnitude of data involved. This project involves different phases such as identifying critical resource intensive functions, initially replacing such methods with PHP Extensions, and eventually experimenting with OpenCL code. We also ran performance tests to measure the reduction in processing times. From our results, we concluded that PHP Extensions and OpenCL processing resulted in performance improvements.

ACKNOWLEDGEMENTS

I would like to express sincere thanks to my project advisor Dr. Chris Pollett, for his guidance and encouragement through every step of this project. His mentoring helped me learn a lot of best practices and gain familiarity with many concepts. I would also extend my thanks to my project committee members, Dr. Sami Khuri and Dr. Thomas Austin, for their guidance and their time.

I also would like to thank my family and my friends, especially my husband and my daughter for the co-operation and motivation that they lent me throughout my Master's program at San Jose State University.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CHARTS

CHAPTER 1

INTRODUCTION

Search engines use an inverted index to look for a word in all the documents that contain that word. This list of documents is called a posting list. Posting lists are typically stored in a compressed binary format. Yioop is a PHP-based open source search engine designed and developed by Dr. Chris Pollett. The objective of this project is to improve the performance of Yioop by using PHP extensions as well as a GPU-based parallelization.

Search engines are among the most used applications on the internet. The processing demands needed to create inverted indexes and to handle even more queries have steadily grown alongside the internet. Search engines rely on index compression as an important tool to be able to handle the increasing amounts of data on the internet. A compressed index enables the caching of more data in memory. The compression also saves bandwidth and reduces the time required to transfer data from memory to the CPU cache. The correct selection of a compression scheme also can play a critical role in improving query performance. Performance can be doubled by using a well-implemented byte-wise compression scheme, as compared to a bit-wise scheme [7].

Decoding a compressed posting list can be made faster when working with whole bytes as opposed to operating on individual bits in that byte. Similarly, operating on whole machine word (16, 32, or 64 bit) is more efficient than accessing the bytes individually [4]. Currently, the Yioop search engine uses a word aligned compression method. Yioop is completely written in PHP.

PHP is an easy-to-use interpreted application programming language. PHP by default provides a rich set of standard function libraries. However, application developers do not have to restrict themselves to the out-of-box PHP functionality. Developers can leverage their preferred programming language to develop PHP extensions. Developers may lose the ease of coding but PHP extensions will provide them with greater flexibility as they get more control over resource allocation and functionality. The application speed can further be improved by leveraging parallelization provided by GPU-based application development frameworks such as OpenCL or CUDA.

According to the research document, "A Performance Study of General Purpose Applications on Graphics Processors", performance gains of up to 40 times can be obtained by switching workloads to graphical processors (CUDA) for several computationally demanding applications including Traffic Simulation, Thermal Simulation, and K-Means[8]. The experimentation part of this project is very computationally intensive. Many modern commodity computer systems are equipped with advanced graphics processing units. Part of this project measures the performance gains obtained through GPU power using OpenCL-based implementation.

OpenCL (Open Computing Language) provides a platform for parallel programming. A typical CPU may have one to eight cores, but GPUs normally contain hundreds or thousands of processing units. Some of the existing posting list processing of Yioop can be parallelized. Therefore, implementing a posting lists algorithm using OpenCL will improve the performance of the Yioop search engine. Encoding and decoding are two major algorithms used in Yioop.

Encoding and decoding algorithms provides important functionality in a search engine. The inverted indexes are created by crawling through various websites and obtaining data. An

encoding algorithm is used to compress the inverted indexes. Similarly, the decoding algorithm allows search engines to retrieve results based on a search word.

This document is divided into a total of six chapters. Chapter 2 describes the concepts of inverted indexes and Yioop's encoding and decoding process flow. Chapter 3 provides background information on PHP extensions and OpenCL. Chapter 4 lists code implementations for this project. Chapter 5 contains the performance tests and results. Chapter 6 consists of observations and the conclusion and. Lastly, Appendix 1 provides the steps needed to set the testing environment and Appendix 2 lists the test machine configurations.

CHAPTER 2

Background of Inverted Index and Yioop's Encoding and Decoding

This chapter discusses inverted index data structures and various compression algorithms, such as γ-codes, Golomb/Rice codes, vByte codes, and Simple-9 codes often used in this construction. The second section of this chapter explains the encoding and decoding process of posting lists in Yioop.

Inverted Index

An inverted index consists of two principal components: the dictionary and the posting lists. The uncompressed inverted index for a given document can be very large, sometimes even greater than the actual source itself. A compressed inverted index can provide advantages including lesser storage space requirements, faster query retrieval time, and the ability to accommodate large collections.

A data compression algorithm converts data represented in one format into another format that requires fewer bits to store and transfer. It contains an encoder and a decoder. An encoder converts original data A into B. The decoder converts the output of an encoder back to the original data. There are two types of decoders. The first one is lossy, which takes B and converts into C, where C can be an approximation of A. Some of the examples of this type of decoder include JPEG and Mp3 files. The second type is lossless, which takes B and converts back to A. Lossless decoders are necessary for search engines.

One of the most important compression algorithms in use is the Huffman coding algorithm. The Huffman algorithm calculates the probability of frequency of characters and uses that

information to find the code word for each character. It uses prefix property; no code word is an initial substring of any other code word-for example a-0, b-11, c-100, d-101.

Posting lists occupy more space than the dictionary, so it is important to have an efficient compression algorithm to minimize the storage space as well as to retrieve the information quickly and without any loss. Therefore, it is necessary to have an efficient algorithm for encoding and decoding.

Posting lists may contain very large number of elements and each number may occur only once, so using standard compression algorithms like Huffman Coding is not feasible. Posting lists contain only monotonically increasing index positions. Replacing index positions with Δ-values, an equivalent sequence of difference between consecutive elements, would be advantageous since elements can be smaller and can be encoded using fewer bits. The algorithms that are available to compress posting lists can be categorized as parametric and nonparametric codes.

Nonparametric codes do not consider the Δ-values in a given posting list while encoding that posting list. They consider that all posting lists share some common features-for example, smaller Δ-values are usually more common than longer ones. Elias's γ-code is one of the examples of a nonparametric gap compression algorithm for positive integers. Γ-codeword for a positive integer contains two components. The first component is a selector that specifies the length of the second component, and the second component is body, the binary representation of the positive integer [4].

Table 1 Elias's γ-codeword

Positive Integer (k)	selector	body	**γ-codeword**
1	1	1	1
5	001	101	00101
7	001	111	00111
16	00001	10000	000010000

A positive integer k consists of $\lfloor \log2(k) \rfloor + 1$ bits in its binary representation. The length of its code word is $|\gamma(k)|=2\lfloor \log2(k) \rfloor+1$ bits.

Encoding: γ-codeword for a positive integer (k): $\lfloor \log2(k) \rfloor$ number of 0's followed by a 1 and then remaining binary bits of k.

Decoding: Count number of zero's until the first 1and consider it as N and 1 is the first bit of the integer with value 2^N and read remaining N bits of the integer.

Parametric codes consider the specific characteristics of the list to be compressed. Golomb/Rice code is an example of a parametric gap compression method. To compress a list whose Δ-values follow a geometric distribution, i.e. $Pr[\Delta=k]=(1-p)^{k-1}p$ for some constant p between 0 and 1. Group the Δ-values according to their bit length and compute the range, which most of the Δ-values lies under.

To encode the list using Golomb/Rice code:

- Choose an integer M, the modulus (M is a power of 2, then Rice code, arbitrary modulus M, Golomb code)

- Split each Δ-value into two components, a quotient q(k) and a remainder r(k) where:

  $q(k) = \lfloor \frac{k-1}{M} \rfloor, \quad r(k) = (k-1) \bmod M$

Encode k by writing $q(k)+1$ in unary followed by $r(k)$ as a $\lfloor \log(M) \rfloor$ bit or $\lceil \log(M) \rceil$ bit number. Golomb code gives better compression rates than Rice code, but Rice decoders run between 0.2 and 0.4 times faster than Golomb decoders [4].

Table 2 Golomb/Rice codes

Integer	Golomb M=3	Codes M=6	Rice M=4	Codes M=8
1	1 0	1 00	1 00	1 000
2	1 10	1 01	1 01	1 001
3	1 11	1 100	1 10	1 010
4	10 0	1 101	1 11	1 011
5	01 10	1 110	01 00	1 100

To improve both encoding and decoding processing times, it is better to look at codes so that the split between code words falls on byte or word boundaries. There are two such methods: byte-aligned and word-aligned codes. One of the simplest examples of a byte-aligned method is vByte (variable-byte coding). It splits the binary representation of each Δ-value into 7-bit chunk + 1 bit continuation flag [4].

Example: **L = (1624, 1650, 1876, 1972, 2350 ...)**

Δ (L) = (1624, 26, 226, 96, 384 ...)

1 1011000 **0** 0001100 **0** 0011010 **1** 1100010 **0** 0000001 **0** 1100000 **1** 0000000 **0** 0000011...

0 at the beginning of the chunk indicates the end of the current code word. ($88 + 12 \times 2^7 = 1624$).

For faster and more efficient decoding purposes, it's better to process the entire machine words such as 16-bit, 32-bit, or 64-bit. But encoding each Δ-value as a 32-bit integer might defeat the intent of compression. This can be solved by using a word-aligned encoding method, where the algorithm inspects the postings list's Δ-values and tries to insert as many consecutive Δ-values as possible into a 32-bit machine word. The simplest example of a word-aligned method is Simple-9.

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 these 32 bits, 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 them into chunks of equal size.

Table 3 Simple-9

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/word	0	0	1	0	3	0	1	0	0

Example: L = (1624, 1650, 1876, 1972, 2350 ...)

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 and Decoding in Yioop

Encoding:

The following flow chart shows the basic flow of the encoding process in the Yioop search engine.

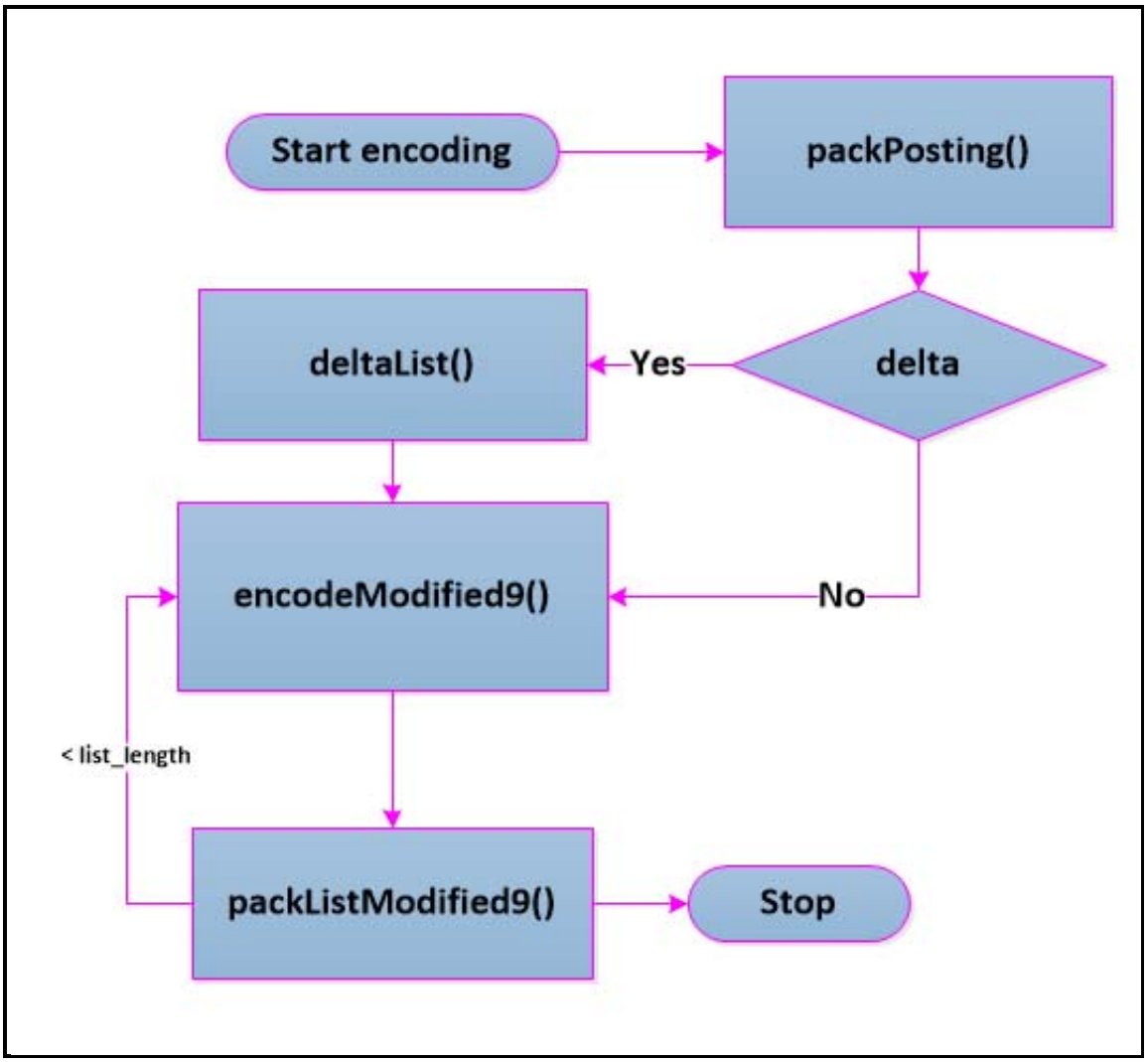

Figure 1 Flow diagram for Encoding

To encode the posting lists, Yioop uses an algorithm called Modified9, which is similar to the Simple-9 algorithm. The Simple-9 algorithm is an example of a word-aligned code compression algorithm that allows for the storage of several consecutive values as a single 32-bit machine word (4 bytes).

The encoder takes a list of posting lists that consists of the index positions of the word occurrences in the document and a document index and returns a packed integer string. First, index positions in the posting lists are replaced with their Δ-values and the document id is attached at the front of the list. Then, Modified9 inspects the Δ-values in a posting sequence and tries to squeeze as many Δ-values as possible into a 32 bit (4 bytes) machine word. In this 32 bit, the high order 2 bits of a given word indicate whether or not to look at the next word.

The first 2-bit codes are as follows:

- 11 start of encoded string
- 10 continue four more bytes
- 01 end of encoded
- 00 indicates the whole sequence encoded in one word

After the first 2 bits, the next most significant bits can be up to either 2, 4, 5 or 6 bits. These most significant bits are called selector, which indicates the format of the current word, i.e. how many Δ-values of equal size have been inserted in the remaining bits. There are nine different possibilities.

Table 4 Modified9 Algorithm

Selector	00	01	10	1100	1101	1110	11110	111110	111111
Number of Δ's	1	2	3	4	5	6	7	12	24
Bits per Δ	28	14	9	6	5	4	3	2	1
Unused bits	0	0	1	0	1	2	4	0	0

A typical posting list consists of a doc_index and the list of position occurrences of the word, i.e [doc_index, Δ-values].

Example: Postings List: [25, [1624 1650 1876 1972 ...]] (doc_index: 25)

Δ-values: [1624, 26, 226, 96, ...] [Δ-value: [1650 -1624] = 26, ...]

doc_index and first index position are incremented by 1 so that they are not equal to zero, which is the requirement for the Yioop's compression scheme, Modified9.

- The above indexes can be saved as 26 and 1625 together as two 14-bits each in one 4-byte word
- 26, 226, and 96 together as three 9-bits each, and one unused bit in one 4-byte word

The final encoded string:

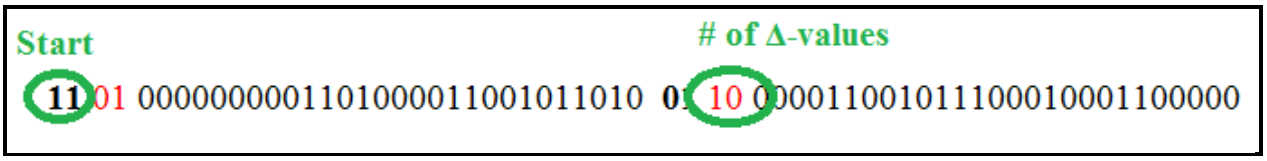

Figure 2 Encoded String

Hex String: D0 06 86 5A A0 65 C4 60

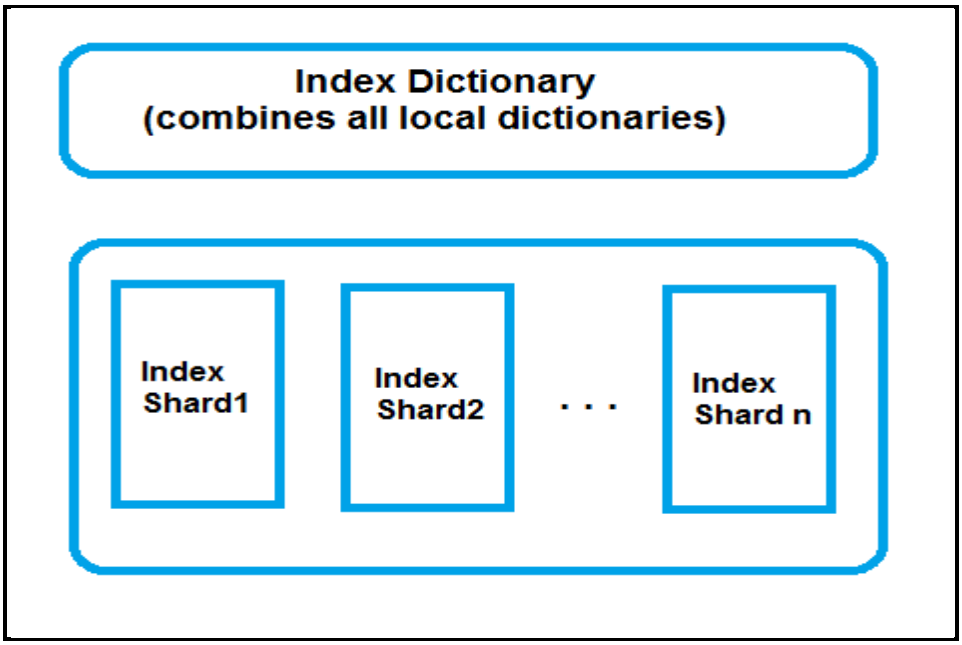

Figure 3 Index storage in Yioop

The above figure represents the storage system of inverted indexes inside the Yioop database. It contains a global index dictionary, the combination of all local dictionaries and several index shards. Each index shard contains a local dictionary and the posting lists.

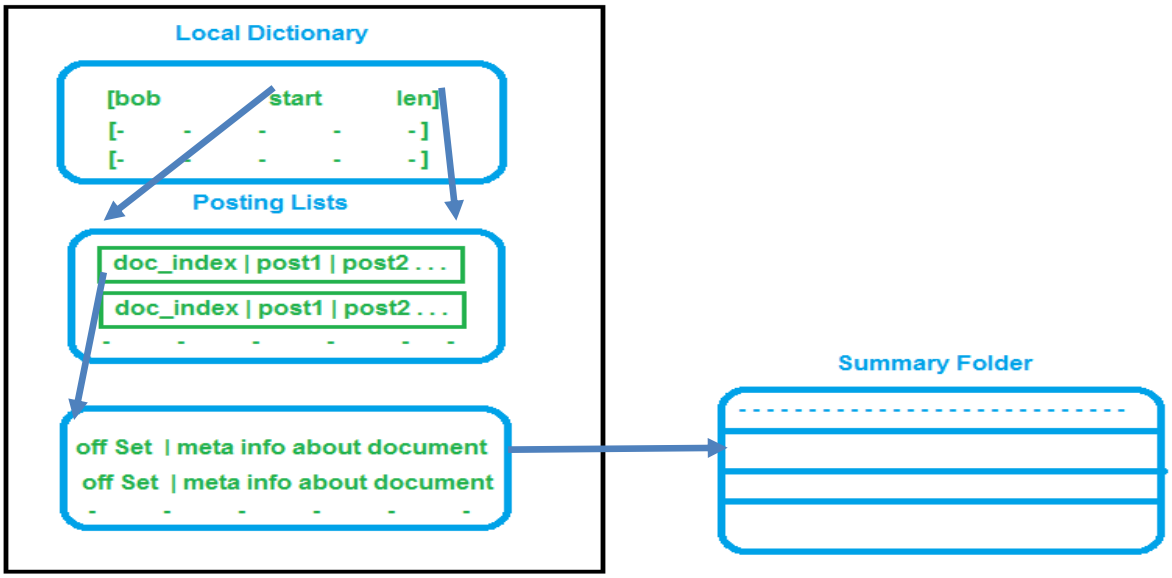

Figure 4 Index Shard

Decoding:

The following flow chart shows the basic flow of the decoding process in the Yioop search engine.

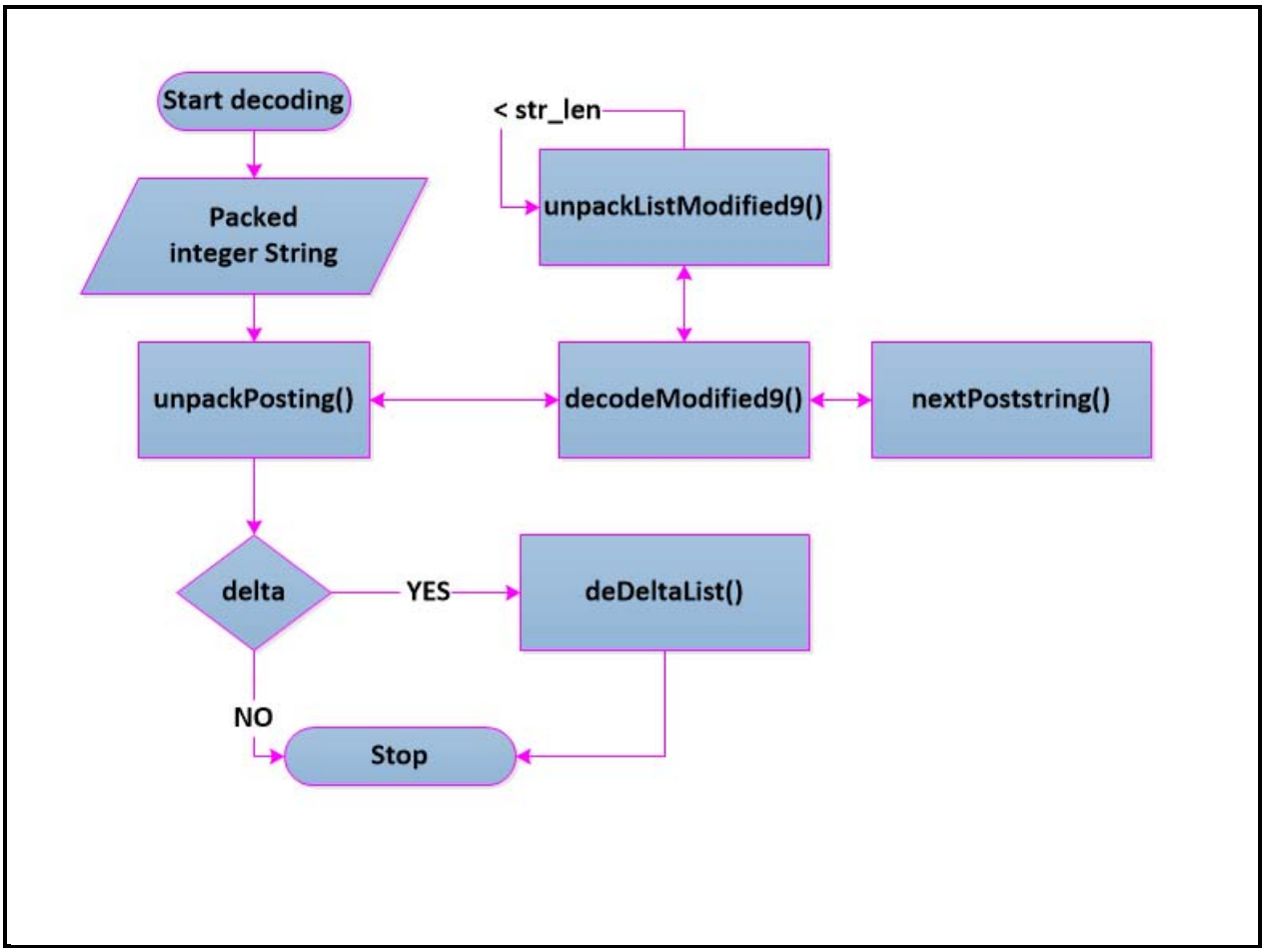

Figure 5 Flow diagram for Decoding

The decoder takes a given packed integer string (encoded using the Modified-9 algorithm), uses the top three bytes to calculate a document index of a document in the shard, and uses the low order bytes to compute the number of occurrences of a word in that document.

At first, decoder algorithm identifies the complete posting string from the given packed integer string of a posting list by checking first two Most Significant Bits of each 4-byte string ("11" for start and "01" for the end). Then takes off first two MSB bits from each 4-byte string and then observes the next bits to identify the number of Δ-values in that 4-byte string, using that

information to decode the string back into its Δ-values according to Modified9. The algorithm repeats the process until the end of the complete posting string to get back all the Δ-values in that posting list. After getting all the Δ-values, the function, *deDeltaList()*, converts the values back into the original index positions. Finally, the decoder attaches the document index at the front and returns an array consisting of the document index and a sub-array consisting of all integer positions of a given word in the document.

CHAPTER 3

Background of PHP Extensions and OpenCL

This chapter reviews the details of PHP extensions and OpenCL, which are used to enhance the performance of the Yioop search engine as part of this project. The PHP extensions section provides an introduction to PHP extensions and the reasons why developers choose to embed in PHP source code. The second part of this chapter provides an introduction to OpenCL and its components by showing the basic OpenCL program flow.

PHP extensions:

Programming languages and compilers provide an extensive set of function libraries. Oftentimes, the standard functionality is enough for general purpose applications. However, there may be a need to alter standard behavior or add additional functionality for complex applications. In such circumstances, customizations like extensions come in handy. PHP extensions are a way to customize or extend the default functionality of PHP. PHP implementation provides many widely used extensions, known as standard extensions called modules, as part of the PHP interpreter. PHP also provides extensions such as session, SPL, PCRE, MySQL, and sockets that can be disabled or enabled through configuration settings. Some of these extensions can also be built through the phpize tool. PHP extensions can be written in many languages, such as Java, Perl, C, C++. The following figure shows the entry point for the PHP extensions and provides the startup and shutdown methods [6]. For this project, the original encoding and decoding functions of Yioop that were written in PHP are replaced with PHP extensions.

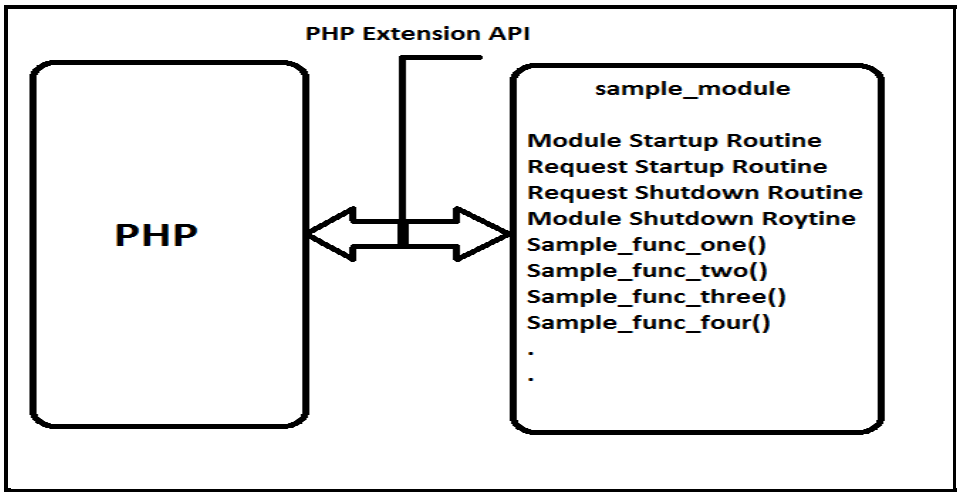

Figure 6 PHP Extension Entry Point

Benefits of PHP extensions:

PHP is a widely used language for developing websites. It's an interpreted language, and it may not always be suitable for applications where speed of processing is the main factor. One such application is web search engine, where results have to be retrieved quickly. PHP extensions provide a way to improve performance while still leveraging the ease of use of coding in PHP. Slow performing code can be moved to other languages such as C or C++ and embedded within PHP functions by using PHP extension methodology [6].

There are many different reasons to utilize PHP extensions. PHP may not provide ways to directly call specific third party or custom libraries. Sometimes developers may need to improve application response times and throughput. PHP extensions can also help reduce memory footprint, as developers have more control through custom extension code. When developers distribute their PHP code, native extensions can be compiled and shared. This will allow the hiding of proprietary source code, protecting intellectual property. Also, organizations can reuse their existing code written in C or C++ through PHP extensions without having to rewrite the same functionality in PHP [2].

OpenCL (Open Computing Language):

OpenCL is a framework built specifically for parallel processing over heterogeneous systems. OpenCL allows developers to write parallel programs in C-language and can exploit the power of GPU threads. The greatest feature associated with OpenCL is its portability across multiple platforms [1]. It works with AMD, NVidia, Intel, IBM, and other GPU vendors. It can also run on integrated graphics, which allows for the use of cached memory for quicker data reading and writing. Integrated graphics card is part of system board and uses part of system memory instead having its own memory.

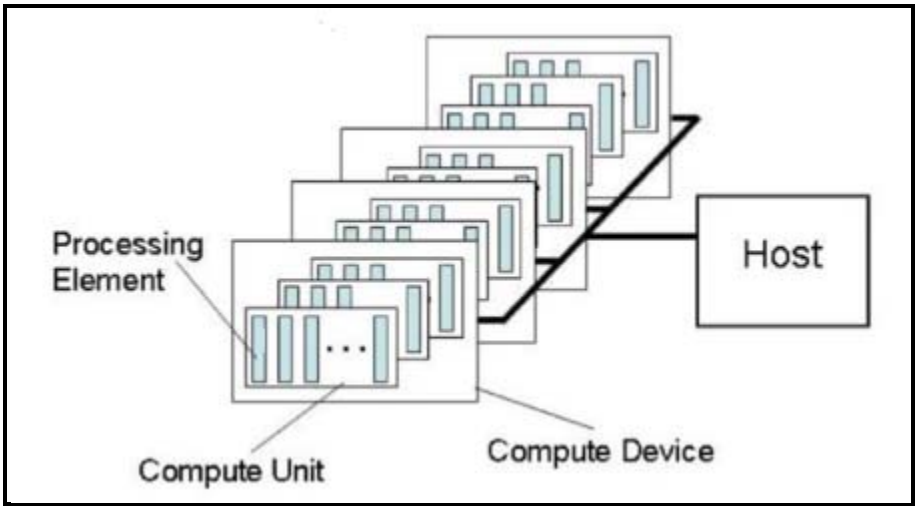

Figure 7 OpenCL Platform Model

Both the host (CPU) and device (GPU) of a given computer system act as computing devices, each of which has several parallel computing units. Each of these computing units is similar to a core or thread and can execute the code in parallel. Host code is typically written in C and, executes in the CPU, whereas device code is written in OpenCL and executes in the GPU. Host code sends the instructions to transfer data between the memories of the host and the devices or to execute device code. Devices will perform these actions and return the results back to the host [3].

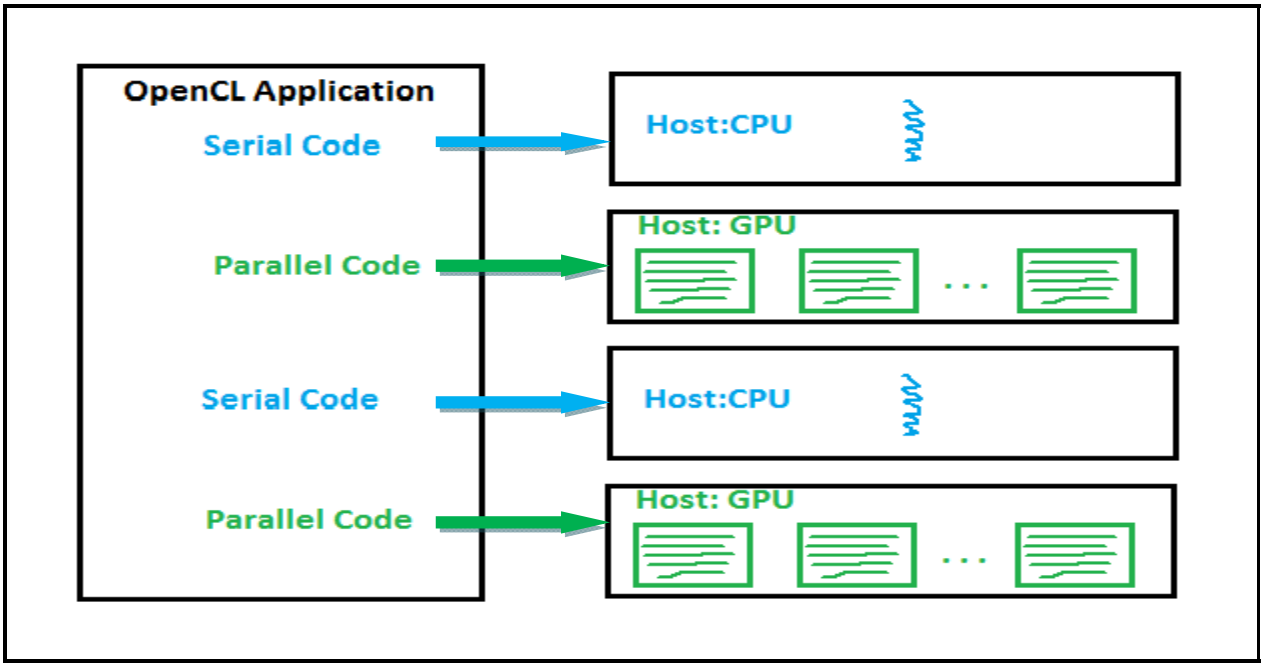

Figure 8 OpenCL Application

All the serial code executes in a host (CPU), and the parallel code executes in many device (GPU) threads across multiple processing elements.

OpenCL Program Flow:

A typical OpenCL program contains the following steps.

1. Organize resources, Create command queue
2. Compile Kernel
3. Transfer data from host to GPU memory
4. Launch threads running kernels on GPU, Perform main computation
5. Transfer data back to host memory from GPU
6. Free allocated memory [5]

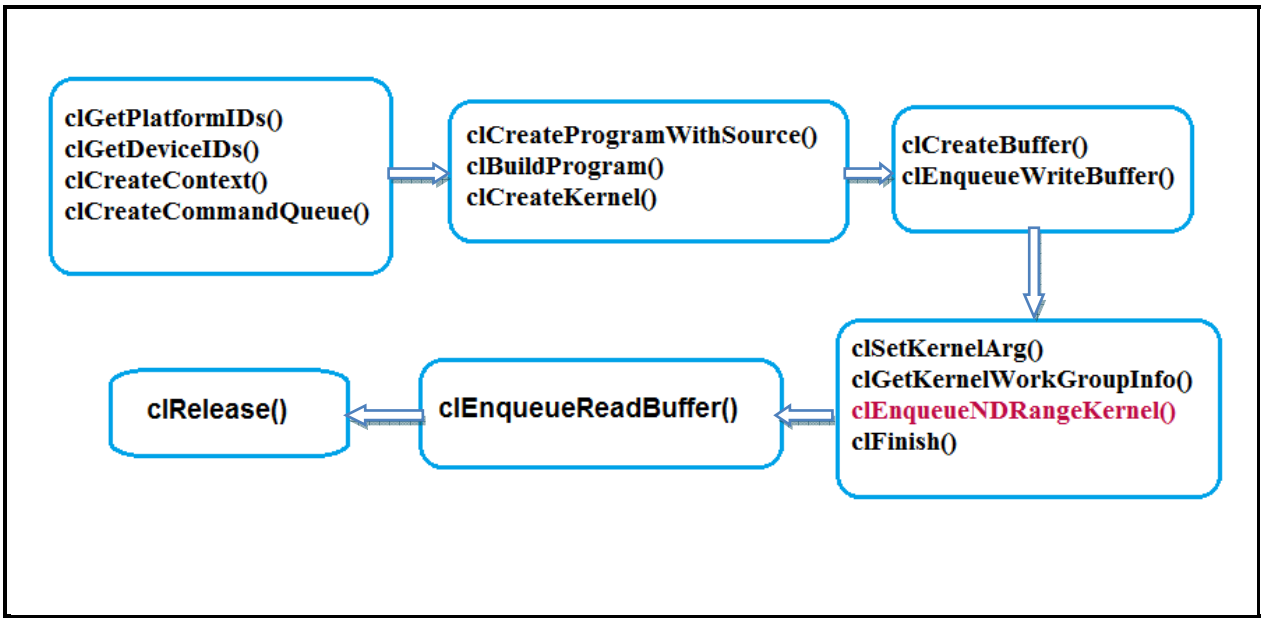

Figure 9 OpenCL Program flow

CHAPTER 4

Code Implementation

This chapter compares the original PHP source snippet and the corresponding C and OpenCL replacement code where applicable.

Encoding:

PHP Function:

The *packPosting()* method is starting point where the encoding of posting lists starts. The following is a snippet of an existing Yioop PHP code. It is located in the Utility.php file. This function takes three input parameters, an integer of the document index, an array of posting lists, and an optional boolean value, which determines whether to use the Δ-values of a posting list. This function encodes the posting list using the Modified9 algorithm and returns a packed integer string of the posting list.

```
function packPosting($doc_index, $position_list, $delta = true)
{
	if ($delta) {
		$delta_list = deltaList($position_list);
	}
	else {
		$delta_list = $position_list;
	}
	if (isset($delta_list[0])){
		$delta_list[0]++;
	}
	if ($doc_index >= (2 << 14) && isset($delta_list[0])
		&& $delta_list[0] < (2 << 9) && $doc_index < (2 << 17)) {
		$delta_list[0] += (((2 << 17) + $doc_index) << 9);
	}
	else {
		// we add 1 to doc_index to make sure not 0 (modified9 needs > 0)
		array_unshift($delta_list, ($doc_index + 1));
	}
	$encoded_list = encodeModified9($delta_list);
	return $encoded_list;
}
```

Code Snippet 1: PHP Encoding

C Function:

The following code snippet is the equivalent C code for the above PHP function. In PHP extensions a regular PHP string is read as a struct of char* and its length, and the structure of an array is a hash table with a label and data. Since this code is embedded into a PHP extension, two struct types, php_string and php_array, were added additionally.

Functions *array_shift()* and *array_unshift()* are built in PHP functions. The *array_shift* function inserts an integer at the front of an array and returns the number of elements in that array. The *array_unshift* function returns the first element of an array and readjusts the remaining elements. But there are no such built-in functions in C to perform these actions. Additional functions *c_shift ()* and *c_unshift()* were written to achieve this functionality.

```
php_string packPosting(int doc_index, php_array  position_list, bool delta)
{
	php_array delta_list;
	if (delta)	{
		delta_list = deltaList(position_list);
	}
	else	{
		delta_list = position_list;
	}
	if (delta_list.arr[0])	{
		delta_list.arr[0] = delta_list.arr[0] + 1;
	}
	if ((doc_index >= (2 << 14) && delta_list.arr[0])
		&& delta_list.arr[0] < (2 << 9) && doc_index < (2 << 17))	{
		delta_list.arr[0] += (((2 << 17) + doc_index) << 9);
	}
	else	{
		// we add 1 to doc_index to make sure not 0 (modified9 needs > 0)
		delta_list = c_unshift(delta_list, (doc_index + 1));
	}
	php_string encoded_list = encodeModified9(delta_list);
	return encoded_list;
}
```

Code Snippet 2: C Encoding

C Extensions:

The below code snippet is an equivalent PHP extension function for the above PHP and C++ functions. Since this function needs to use the same input and output parameters that were used in the original PHP function, an integer was declared to get a *doc_index*, a *zval* * which reads an array of posting list, and *zend_bool* which reads the boolean delta value. The below zend function is used to read input parameters.

```
zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "la|b", &doc_index,
&position_list, &delta)
```

Figure 10 Zend Function to read input

- Letter "l" represents a variable type long which is used to read an integer or long values.
- Letter "a" represents a variable type array and "b" represents a boolean value type.
- Symbol '|' is given in front of the variable type, if the parameter is an optional.

The following is the complete representation of code that incorporates above PHP and C functions. The highlighted code converts the *zval** array structure to regular C array. This will facilitate the reuse of code in inner functions.

```
PHP_FUNCTION(packPosting)
{
	zval* position_list, **data;
	HashTable *arr_hash;
	HashPosition pointer;
	int doc_index;
	zend_bool delta = TRUE;
	php_array delta_list;

	if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "la|b",
		&doc_index, &position_list, &delta) == FAILURE) {
		RETURN_NULL();
	}
	arr_hash = Z_ARRVAL_P(position_list);
	int array_count = zend_hash_num_elements(arr_hash);

	php_array list;
	list.arr = (unsigned int*)malloc(sizeof(unsigned int)*array_count);
	list.arr_len = array_count;
	int i = 0;
	for (zend_hash_internal_pointer_reset_ex(arr_hash, &pointer);
		zend_hash_get_current_data_ex(arr_hash,(void**)&data,
		&pointer)== SUCCESS;
		zend_hash_move_forward_ex(arr_hash, &pointer))	{
		list.arr[i] = Z_LVAL_PP(data);
		i++;
	}
	if (delta)	{
		delta_list = c_deltaList(list);
	}
	else	{
		delta_list = list;
	}
	if (delta_list.arr[0])	{
		delta_list.arr[0] = delta_list.arr[0] + 1;
	}
	if ((doc_index >= (2 << 14) && delta_list.arr[0])
		&& delta_list.arr[0] < (2 << 9) && doc_index < (2 << 17))	{
		delta_list.arr[0] += (((2 << 17) + doc_index) << 9);
	}
	else	{
		// we add 1 to doc_index to make sure not 0 (modified9 needs > 0)
		delta_list = c_unShift(delta_list, (doc_index + 1));
	}

	php_string encoded_string = encodeModified9(delta_list);
	free(delta_list.arr);

	ZVAL_STRINGL(return_value, encoded_string.name, encoded_string.name_len, 1);
	free(encoded_string.name);
}
```

Code Snippet 3: C Extensions Encoding

Decoding:

PHP Function:

The *unpackPosting()* function plays a vital role in retrieving the data from the database when a user searches for a word or a phrase using the Yioop search engine. The input parameters for this function are an *encoded string*, an *offset*, and a *dedelta*. The *encoded string* contains the information about all the posting lists of the specified search. The *offset* is an integer value that contains the starting position of the posting list. The *dedelta* is a boolean value that determines whether the posting list has Δ-values or not.

The *$offset* is sent as a pass-by-reference. Because each encoded string contains several postings, *offset* needs to be updated after the decoding of each posting has been completed.

This function returns a list containing two elements. The first is an integer value of a *doc_index*, and the second element is a list containing the decoded posting lists.

```
function unpackPosting($posting, &$offset, $dedelta = true)
{
	$delta_list = decodeModified9($posting, $offset);
	$doc_index = array_shift($delta_list);
	if (($doc_index & (2 << 26)) > 0) {
		$delta0 = ($doc_index & ((2 << 9) - 1));
		array_unshift($delta_list, $delta0);
		$doc_index -= $delta0;
		$doc_index -= (2 << 26);
		$doc_index >>= 9;
	}
	else {
		$doc_index--;
	}
	if (isset($delta_list[0])) {
		$delta_list[0]--;
	}
	if ($dedelta) {
		deDeltaList($delta_list);
	}
	return[$doc_index, $delta_list];
}
```

Code Snippet 4: PHP Decoding

PHP C Extensions Function for Decoding:

Since one of the arguments *($offset)* must be passed as pass-by-reference, the argument needs to be declared ahead by using the *ZEND_BEGIN_ARG_INFO_EX()* function.

In order to send one of the arguments as pass-by-reference in PHP extensions, it needs to be declared in arginfo structure. The last argument value of 2 in the signature of function *ZEND_BEGIN_ARG_INFO_EX()* means that a minimum of 2 arguments are required. The code is below.

```
ZEND_BEGIN_ARG_INFO_EX(unpackPosting_arginfo, 0, ZEND_RETURN_VALUE, 2)
       ZEND_ARG_INFO(0, posting) // 0 means "passed by value"
       ZEND_ARG_INFO(1, off_set) // 1 means "passed by reference"
ZEND_END_ARG_INFO();
```

Code Snippet 5: ZEND_ARG_INFO( )

Inside the zend function entry, the function with pass-by-reference needs to be declared, as below.

PHP_FE(unpackPosting, unpackPosting_arginfo)

```
PHP_FUNCTION(unpackPosting) {

        . . .

       int offset = Z_LVAL_P(off_set);

       if (posting_len % 4>0)
              posting_len += posting_len % 4;

       php_array_delta_list = decodeModified9(posting, posting_len, &offset);

       zval_dtor(off_set);
       Z_LVAL_P(off_set) = offset;

       int doc_index = c_shift(&delta_list);

       . . .

       zval * list;
       MAKE_STD_ZVAL(list);
       array_init(list);
       for (int i = 0; i<delta_list.arr_len; i++)       {
              add_next_index_long(list, delta_list.arr[i]);
       }

       array_init(return_value);
       add_next_index_long(return_value, doc_index);
       add_next_index_zval(return_value, list);
}
```

Destroying the original value and updating with new value

Returning [doc_index, [list]]

Code Snippet 6: C Extensions Decoding

OpenCL Code:

deltaList() :

The *deltaList()* function takes a list of integers, computes the differences between two consecutive elements, and returns the list of Δ-values. The following code snippets show the original PHP and C code, with foreach and for loop respectively to iterate through the list.

```
function deltaList($list)
{
	$last = 0;
	$delta_list = [];
	foreach($list as $elt) {
		$delta_list[] = $elt - $last;
		$last = $elt;
	}
	return $delta_list;
}
```

Code Snippet 7: PHP deltaList

```
php_array  c_deltaList(php_array list)
{
	int last = 0;  int elt;
	int i = 0;
	for (i = 0; i < list.arr_len; i++)
	{
		elt = list.arr[i];
		list.arr[i] = elt - last;
		last = elt;
	}
	return list;
}
```

Code Snippet 8: C deltaList

For loop functionality is achieved concurrently in one step using OpenCL's parallelism. The code snippets below show the OpenCL kernel and the host code for the deltaList function.

```
__kernel void deltaList(__global const unsigned int *list,
                        __global unsigned int *delta_list) {

	// Get the index of the current element
	int i = get_global_id(0);

	delta_list[i] = list[i + 1] - list[i];
}
```

Code Snippet 9: OpenCL kernel

The code snippet below is the host (CPU) code for the OpenCL *deltaList()* function. Two memory buffers are created for the device code. One is read only for sending the list of unsigned integers, and the other one is write only, for the result array consists of Δ-values, the difference of consecutive elements. The *global_size* is set as $(list\ length - 1)$, because the first element of the list does not change.

```
php_array opencl_deltaList(php_array list) {

        int len = list.arr_len - 1;

        // Create memory buffers on the device for each vector
        cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY |
        CL_MEM_COPY_HOST_PTR,(len + 1) * sizeof(unsigned int), list.arr, &ret);

        cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY, (len)*
                                          sizeof(unsigned int), NULL, &ret);

        // Create the OpenCL kernel
        cl_kernel kernel = clCreateKernel(program, "deltaList", &ret);

        // Set the arguments of the kernel
        ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
        ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);

        // Execute the OpenCL kernel on the list
        size_t global_item_size = len;                Sets global_id from 0 - (len-1)
        ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
                                     &global_item_size, 0, 0, NULL, NULL);

        // Allocate the memory for outPut
        unsigned int * outPut = (unsigned int*)malloc(sizeof(unsigned int)*
                                                      (list.arr_len));
        outPut[0] = list.arr[0];

        //read result from the device to array outPut
        ret = clEnqueueReadBuffer(command_queue, b_mem_obj, CL_TRUE, 0,
                                  (len)* sizeof(int), &outPut[1], 0, NULL, NULL);

        // Clean up
        ret = clReleaseKernel(kernel);
        ret = clReleaseMemObject(a_mem_obj);
        ret = clReleaseMemObject(b_mem_obj);
        free(list.arr);

        list.arr = outPut;
        return list;
}
```

Code Snippet 10: OpenCL Host Code

unpackListModified():

The code snippet below shows the PHP function where *unpackListModified()* function is called repeatedly. Instead of regular loop, this function uses *call_user_func_array("array_merge", array_map(C\NS_LIB . "unpackListModified9", unpack("N*", $post_string)))*.

The unpack() function returns 32 bits from *$post_string* at a time, which is an input parameter to the function ***C\NS_LIB . "unpackListModified9"***. ***"C\NS_LIB"*** specifies that this function is computed under C land. The functions *array_merge()* and *array_map()* are used to combine all the outputs from the *unpackListModified()* function.

```
function decodeModified9($input_string, &$offset)
{
	$post_string = nextPostString($input_string, $offset);
	return call_user_func_array("array_merge",
		array_map(C\NS_LIB . "unpackListModified9",
		unpack("N*", $post_string)));
}
```

Code Snippet 11: PHP

A regular while loop is used in C Extensions to call *unpackListModified()* function repeatedly to process the entire encoded string 4 bytes at a time.

```
while (len  <  str_len)  {

	temp_string = (char*)memcpy(temp_string, post_string + len, 4);
	temp_array = unpackListModified9(temp_string);
	size += temp_array.arr_len;
	if(decoded_arr.arr_len == 0)
	    decoded_arr.arr = (unsigned int*)malloc(temp_array.arr_len * 4);
	else
	    decoded_arr.arr = (unsigned int*)realloc(decoded_arr.arr,(size * 4));
	    memcpy(decoded_arr.arr + decoded_arr.arr_len, temp_array.arr,
                                                  temp_array.arr_len * 4);
	decoded_arr.arr_len += temp_array.arr_len;
	len += 4;
}
```

Code Snippet 12: C Extensions

The following code snippet shows *opencl_unpackListModified()* function call, the replacement of the above PHP and C loops.

```
php_array decodeModified9(char* input_string, int str_len, int *offset)
{
	char* post_string = nextPostString(input_string, &str_len, offset);
	return opencl_unpackListModified(post_string, str_len);
}
```

Code Snippet 13: OpenCL

The *unpackListModified()* function uses three constant char arrays in its calculations to decode the encoded string. To minimize the data transfer time, these three constant char arrays are declared inside the kernel code as shown below instead of copying these values from CPU land to the GPU land for each function call.

```
// Declaring the constatnts for unpackListModified9
__constant char MOD9_NUM_CODES[9] = { 63, 62, 60, 56, 52, 48, 32, 16, 0 };
__constant char MOD9_NUM_ELTS[9] = { 24, 12, 7, 6, 5, 4, 3, 2, 1 };
__constant char MOD9_NUM_BITS[9] = { 1, 2, 3, 4, 5, 6, 9, 14, 28 };
```

Code Snippet 14: Constants in Kernel

The code snippet below is the device/kernel code for the OpenCL *unpackListModified()* function. Parameters of this function are the *post_string,* which is a whole encoded string and the *decoded_list,* an unsigned integer array of size equal to $strlen(post\_string)/4$. The *decoded_list* will return all the decoded Δ-values back to the host device (CPU).

```
__kernel void unpackListModified9(__global const char *post_string,
                                  __global unsigned int *decoded_list) {

       int num = get_global_id(0); int i;
       int num_bits; int num_elts; int mask; int  shift;
       int pre_elt; int code = 0; int first_char;
       unsigned int encoded_list = (post_string[num * 4] & 0xff) << 0x18;
       encoded_list += (post_string[num * 4 + 1] & 0xff) << 0x10;
       encoded_list += (post_string[num * 4 + 2] & 0xff) << 0x8;
       encoded_list += (post_string[num * 4 + 3] & 0xff);

                      .  .   .

       decoded_list[num * 25] = num_elts;
       for (i = 0; i < num_elts; i++) {
              if ((pre_elt = encoded_list & mask) == 0)
                     break;
              decoded_list[num * 25 + num_elts - i] = pre_elt;
              encoded_list >>= num_bits;
       }
         .  .   .
}
```

Code Snippet 15: Kernel code

The below code snippet is the host code of an OpenCL *unpackListModified()* function. In order to request an OpenCL kernel to open $strlen(post\_string)/4$ threads, *global_size* should be set to $strlen(post\_string)/4$. Each thread computes four bytes of an input string in parallel.

Not knowing the output size ahead is not a problem with PHP and C functions, but with OpenCL, the size needs to be known to allocate enough memory on the device. Since each 4 byte string can only hold a maximum of 24 unsigned integers, an output unsigned integer array is allocated with size equal to to $25 \times strlen(post\_string)/4$. The first element of every 25 element array chunk holds the actual number of values stored in that 4 bytes string. After getting the results back to the CPU land, a while loop is created to iterate according to the value in the first element of every 25 element array chunk and inserts the decoded Δ-values into an output array.

```
php_array opencl_unpackListModified(char* str_encoded_list, int len) {

	// Create memory buffers on the device for each vector
	cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY |
			CL_MEM_COPY_HOST_PTR,(len)* sizeof(char), str_encoded_list, &ret);
	cl_mem e_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
		sizeof(unsigned int)* (len / 4) * 25, NULL, &ret);

		. . .

	// Execute the OpenCL kernel on the list
	size_t global_item_size = len / 4; // Process the entire lists
	ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, 0,
						&global_item_size, 0, 0, 0, 0);

	unsigned int* temp = (unsigned int*)malloc(len * 25);

	//read result from the device to array temp
	ret = clEnqueueReadBuffer(command_queue, e_mem_obj, CL_TRUE, 0,
			sizeof(unsigned int)* (len / 4) * 25, temp, 0, NULL, NULL);

	int count = 0; int size = 0; unsigned int*list; int pre_size = 0;

	while (count < ((len / 4) * 25))	{
		int k = temp[count];
		if (count == 0)		{
			pre_size = size;
			size = k;
			list = (unsigned int *)malloc(size * 4);
		}
		else		{
			pre_size = size;
			size = size + k;
			list = (unsigned int*)realloc(list, size * 4);
		}

		for (int i = 1; i <= k; i++)		{
			list[pre_size + i - 1] = temp[count + i];
		}
		count = count + 25;
	}

	 . . .

	return decoded_list;
}
```

Code Snippet 16: OpenCL unpackListModified

CHAPTER 5

TESTS AND RESULTS

Zipf's law is a commonly used distribution model for the occurrences of terms in a collection. Zipf's law states that the frequency of the ith most frequent term is inversely proportional to its rank (i) as shown in the diagram below, where β is a normalizing constant and its value is 0 < β <1 and α is very close to 1. $\beta \odot f_i$ is collection frequency, the number of occurrences of term t_i in a collection.

$$f_i \sim \frac{1}{i^{\alpha}}$$

$$f_i = \frac{\beta}{i^{\alpha}}$$

Figure 11 Zipf's law

For this performance comparison testing, posting lists are created for two collections- one contains 10,000 documents and the second one contains 100,000. Each document contains 5000 tokens and a total of 10,000 unique words in each collection. Inverted indexes are created using Zipffian distribution by setting β as 0.5 and α as 1.1. Three different ranking words (*i=10, 310, 3000)* from these posting lists are randomly chosen from ranks (1-100), (101 -100), and (1001 - 10,000) such that the first one is the most frequent term, second one is a moderately frequent term, and the third one is a less frequent term according to their frequency of occurrence in the document collection.

The first test scenario compares the performance of the encoding of each word's postings into a single packed integer string. The second scenario compares the performance of decoding the

integer string from the first scenario back to the original postings. These tests are run with three variations: PHP, C Extensions, and OpenCL Extensions code. These tests are also repeated on different CPU and GPU combinations, 32/64 bit application mode, and with different PHP versions. Performance results are represented in terms of the time taken to perform the encoding or decoding respectively for both scenarios.

PHP 5 Encoding test:

The following chart shows the results of encoding test of a posting list with different word frequency ranks of 10,000 documents collection. The test was run on *i5* machine with an Intel HD graphics processor. A performance improvement of up to 3 times was observed with C Extensions compared to the original PHP code for all three chosen term ranks. As the term frequency increases, the OpenCL Extensions results show improvement, but not as good as the C Extensions. This is because of the overhead associated with OpenCL's data transfer between the kernel and CPU.

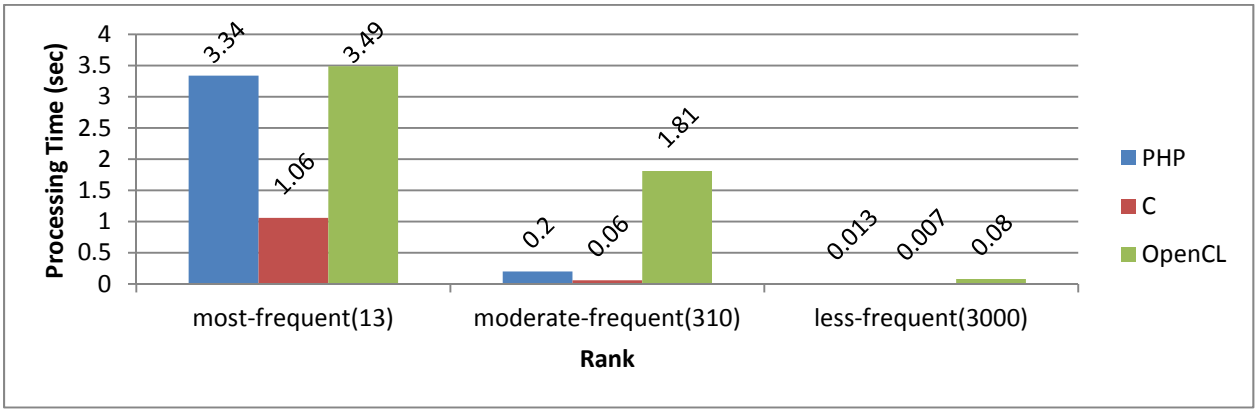

Chart 1 Encoding Test Results for10,000 documents (PHP5, 32 bit, i5+HD GPU)

Chart 2 shows the results of encoding as above, but with 100,000 documents. These test results follow same pattern as above.

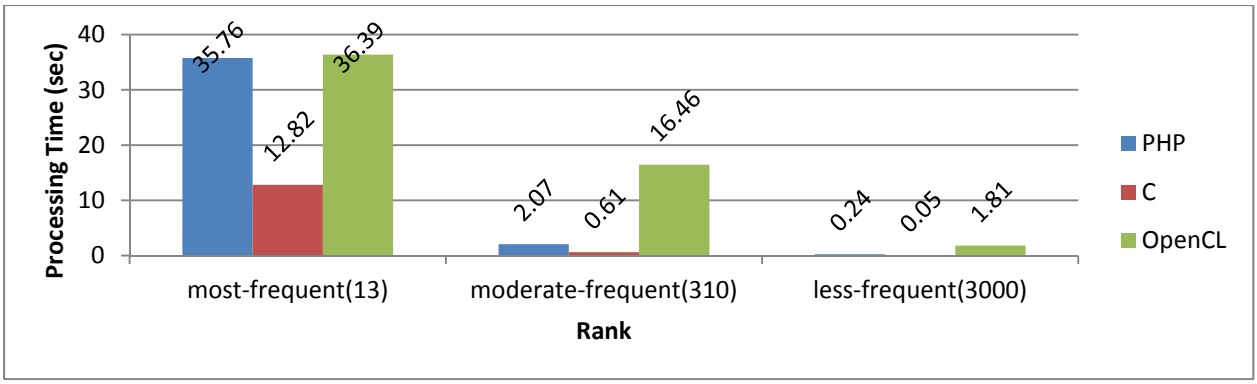

Chart 2 Encoding Test Results for 100,000 documents (PHP5, 32 bit, i5+HD Graphics)

The following chart 3 shows the results of the encoding test of a posting list with different word frequency ranks of 10,000 documents collection run on an *i7* machine with an Nvidia graphics processor. Similar to the *i5* processor, a performance improvement of up to 3 times was observed with C Extensions compared to the original PHP code for all three chosen term ranks with an *i7* processor. For the most frequent search term (rank 13), an improvement of about 0.4 times was observed with OpenCL extensions using an Nvidia GPU when compared relatively with the original PHP implementation.

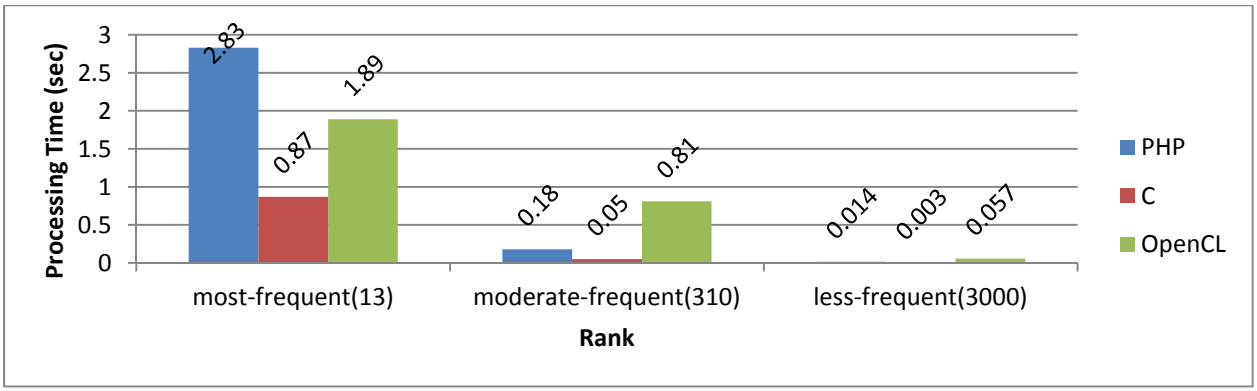

Chart 3 Encoding Test Results for 10,000 documents (PHP5, 32 bit, i7+Nvidia Graphics)

Chart 4 shows the results of the encoding test as above, but with 100,000 documents. These test results follow the same pattern as above with OpenCL and C Extensions.

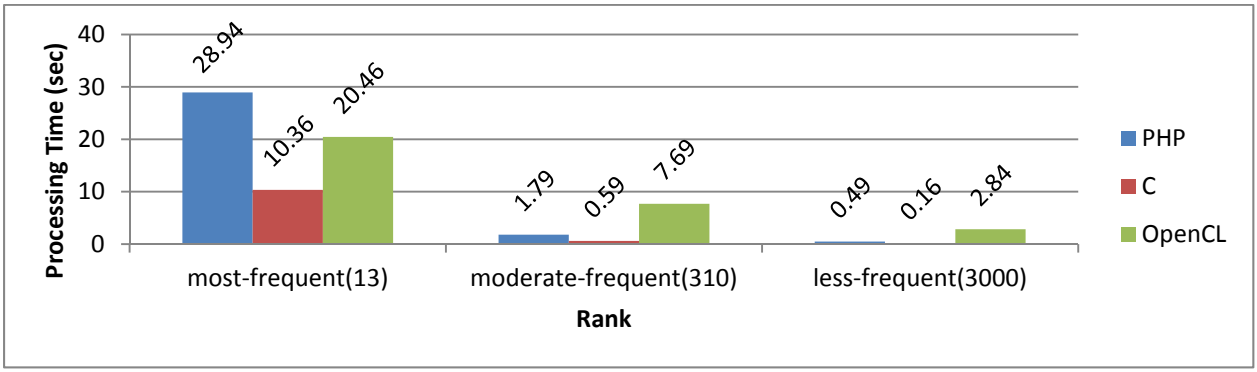

Chart 4 Encoding Test Results for 100,000 documents (PHP 5, 32 bit, i7+Nvidia Graphics)

PHP 7 Encoding test:

The following chart shows the results of repeating the encoding test of a posting list with different word frequency ranks of 10,000 documents collection, run on an *i5* machine with an Intel HD graphics processor with PHP version 7. This test compares the processing time required for PHP and C Extensions. An approximately 300% improvement is observed with C Extensions for all variations of tests relative to PHP code.

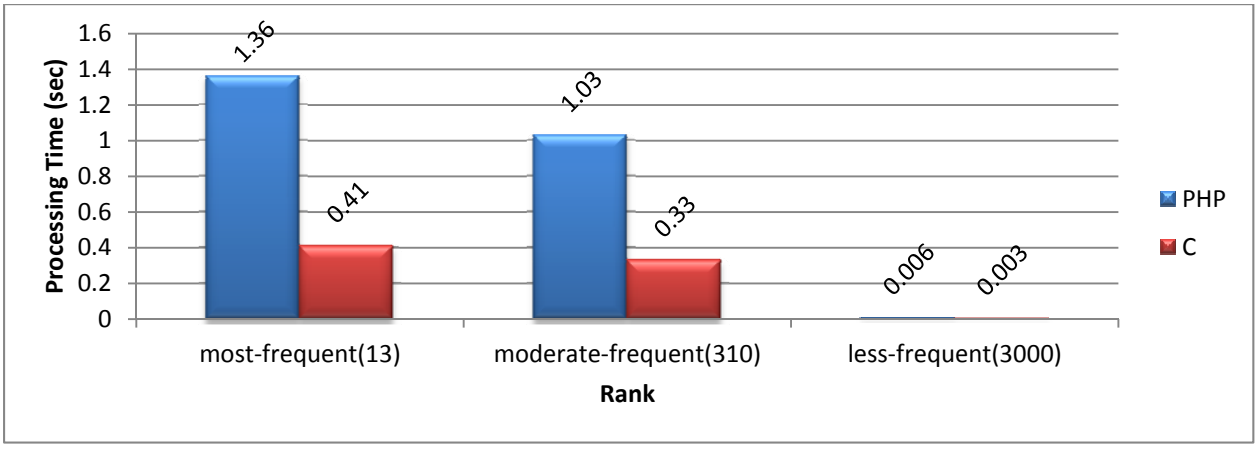

Chart 5 Encoding Test Results for 10,000 documents (PHP7, 32 bit, i5)

Chart 6 shows the results of the encoding test as above, but with 100,000 documents. These test results follow the same pattern as above with PHP 7 C Extensions.

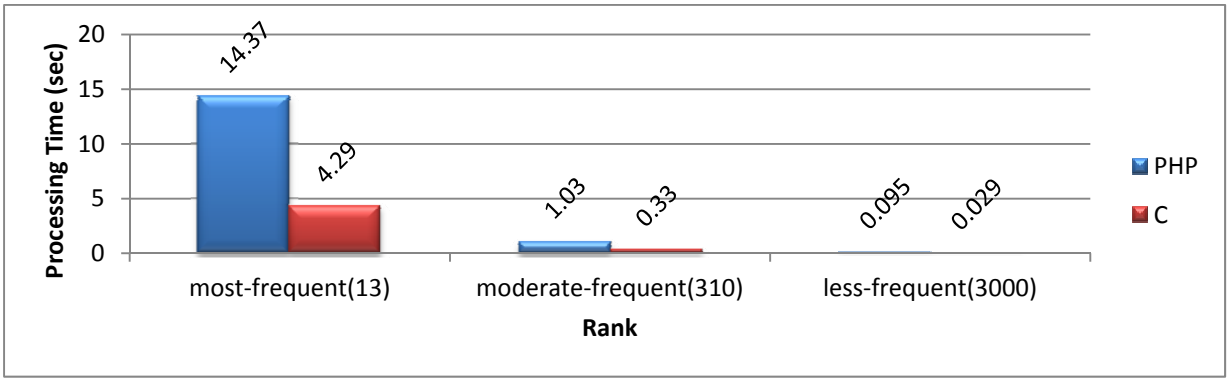

Chart 6 Encoding Test Results for 100,000 documents (PHP7, 32 bit, i5)

The following chart 7 shows the results of the encoding test of a posting list with different word frequency ranks of 10,000 documents collection, run on an *i7* machine with an Nvidia graphics processor. Similar to the *i5* processor, a performance improvement of up to 3 times was observed with C Extensions compared to original PHP code for all three chosen term ranks with an *i7* processor.

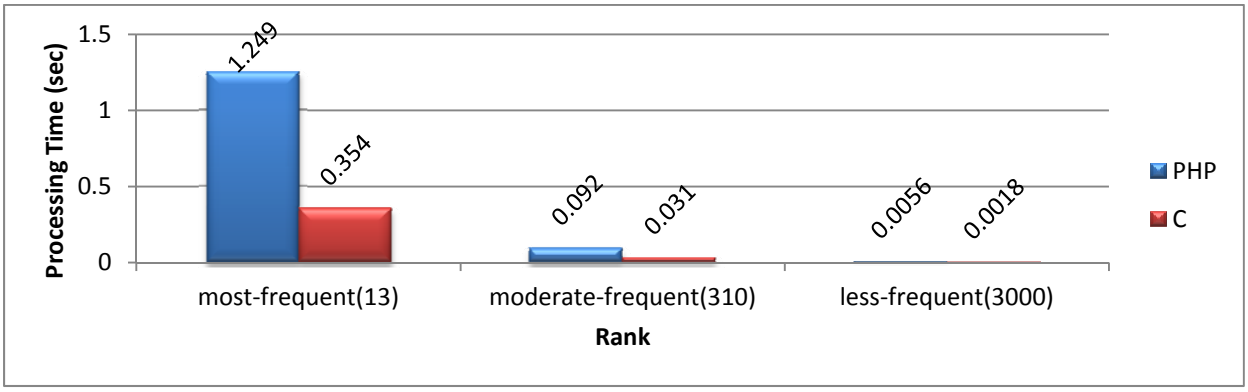

Chart 7 Encoding Test Results for 10,000 documents (PHP7, 32 bit, i7)

Chart 8 shows the results of the encoding test as above, but with 100,000 documents. These test results follow the same pattern as above with C Extensions.

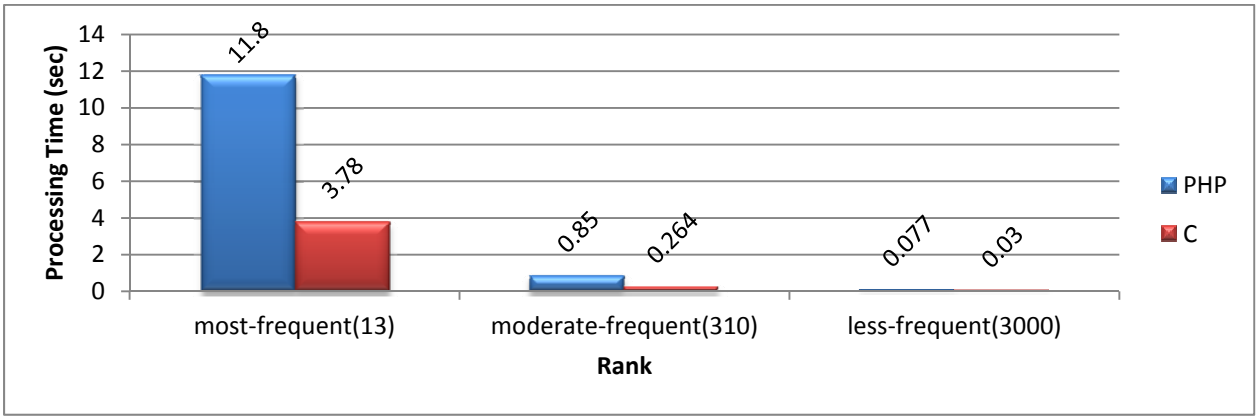

Chart 8 Encoding Test Results for 100,000 documents (PHP7, 32 bit, i7)

PHP 5 Decoding test:

The following chart shows the decoding test results of 10,000 documents run on an *i5* machine with Intel HD graphics processors with PHP version 5. There is an approximate performance improvement of 6 times with C extensions and about 4 times with OpenCL for the test case involving the most frequently occurring term (rank=13).

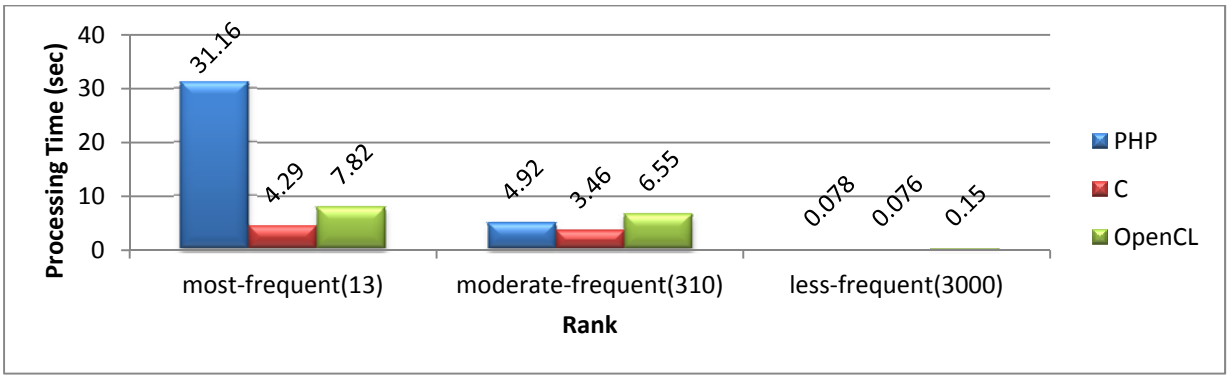

Chart 9 Decoding Test Results for 10,000 documents (PHP5, 32 bit, i5+HD Graphics)

Chart 10 shows the results of the decoding test as above, but with 100,000 documents. These test results show even more improvement than the previous test. There is an approximate performance improvement of about 40 times with C extensions and 30 times with OpenCL for the test case involving the most frequently occurring term (rank=13).

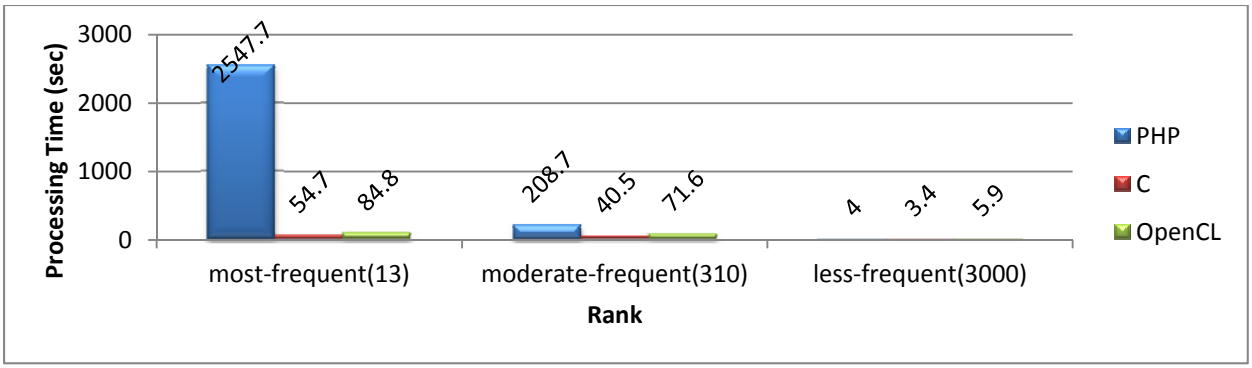

Chart 10 Decoding Test Results for 100,000 documents (PHP5, 32 bit, i5+HD Graphics)

The chart below shows the test results of the decoding test of an encoded posting list with different word frequency ranks of 10,000 documents collection run on *i7* machine with an Nvidia graphics processor. A performance improvement of 8 times with C Extensions and 5 times with OpenCL was observed with the most frequently occurring term (rank=13).

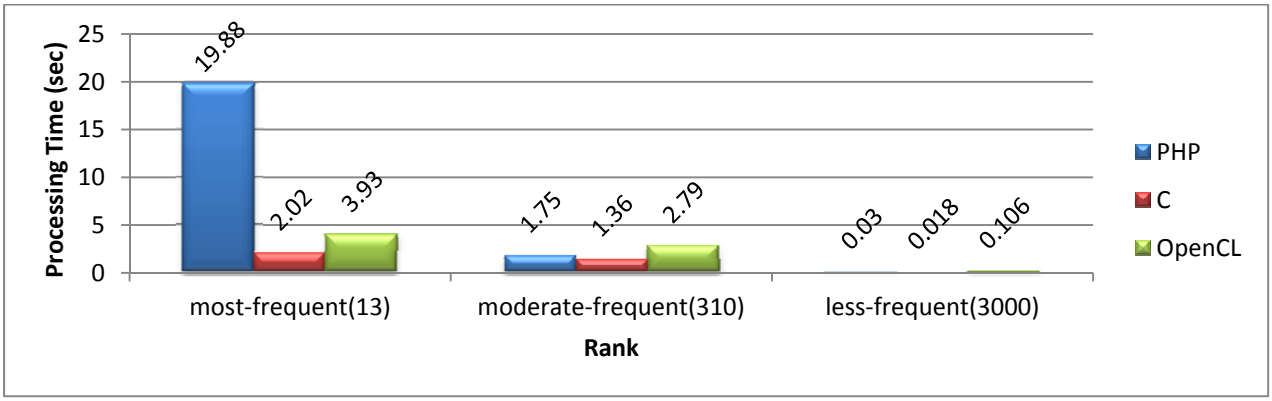

Chart 11 Decoding Test Results for 10,000 documents (PHP5, 32 bit, i7+Nvidia Graphics)

Chart 12 shows the results of decoding test as above, but with 100,000 documents. There was an approximate performance improvement of about 80 times with C extensions and 50 times with OpenCL for the test case involving the most frequently occurring term (rank=13). Original PHP code took approximately the same time on both *i7* and *i5* machines, but C and OpenCL extensions took only half the time on *i7* compared to *i5*.

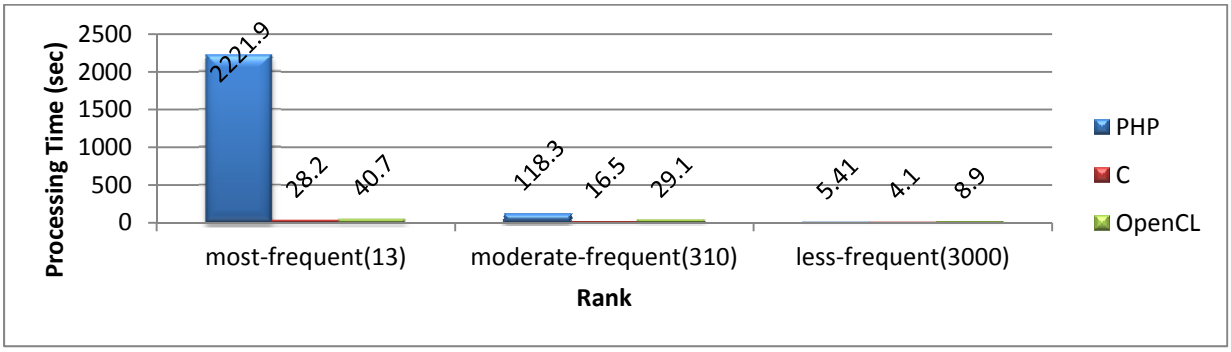

Chart 12 Decoding Test Results for 100,000 documents (PHP5, 32 bit, i7+Nvidia Graphics)

PHP 7 Decoding test:

The following test scenarios are run under PHP 7 environment. Charts 13 and 14 represent the testing results of decoding on an *i5* machine for 10,000 and 100,000 documents, respectively. Charts 15 and 16 represent the testing results of decoding on an *i7* machine for 10,000 and 100,000 documents, respectively. PHP 7 has significant processing improvements when compared to PHP 5. Beyond that, C extensions offer additional processing speed improvements of 10 to 50%.

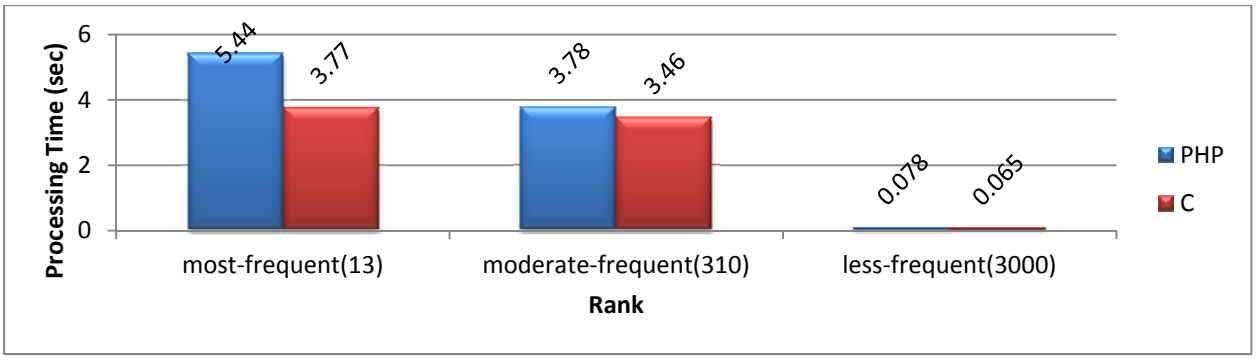

Chart 13 Decoding Test Results for 10,000 documents (PHP7, 32 bit, i5)

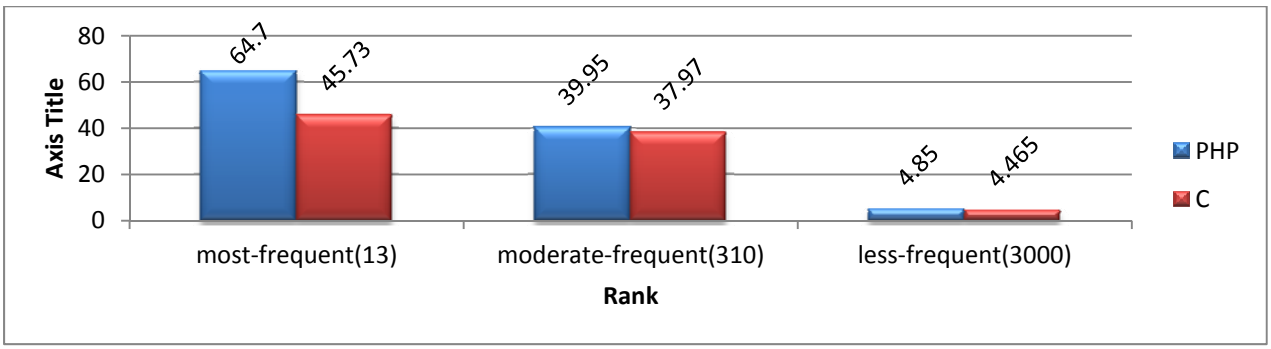

Chart 14 Decoding Test Results for 100,000 documents (PHP7, 32 bit, i5)

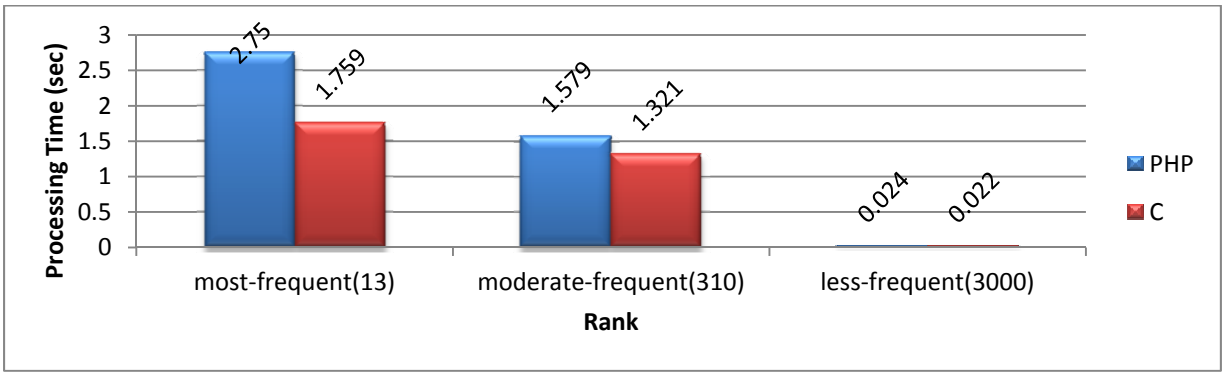

Chart 15 Decoding Test Results for 10,000 documents (PHP 7, 32 bit, i7)

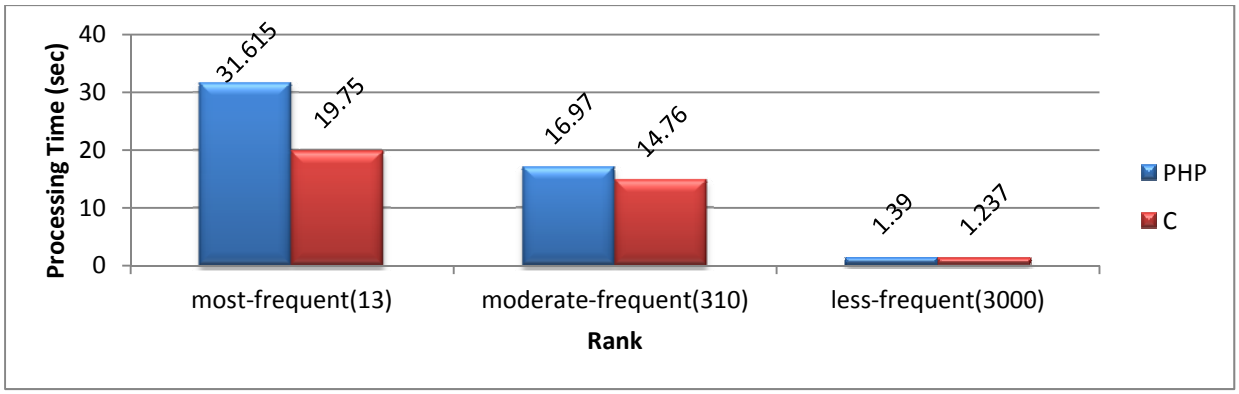

Chart 16 Decoding Test Results for 100,000 documents (PHP7, 32 bit, i7)

PHP 5 Vs PHP 7:

The objective of this test case is to measure the performance improvements obtained by simply using PHP 7 instead of PHP5. The test also compares results with C extensions. The PHP7 encoding test shows a 2.5 times performance improvement relative to PHP5. Even though PHP7 already offers better performance, switching to a C extension further improves processing speed up to 3 times.

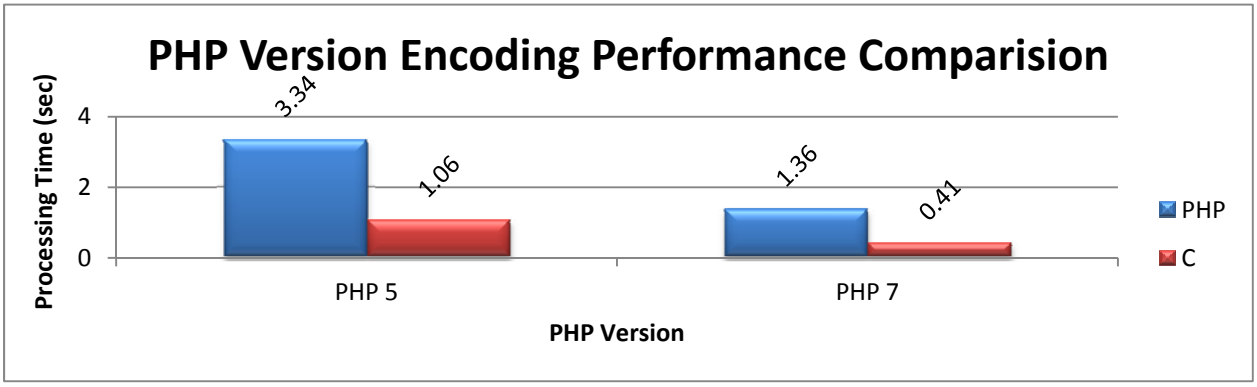

Chart 17 PHP5 Vs PHP7 Encoding Performance Comparison

The following chart shows the decoding test results with different PHP versions. Switching to PHP7 improves performance by about 6 times. C extensions further yield about a 0.8 times improvement.

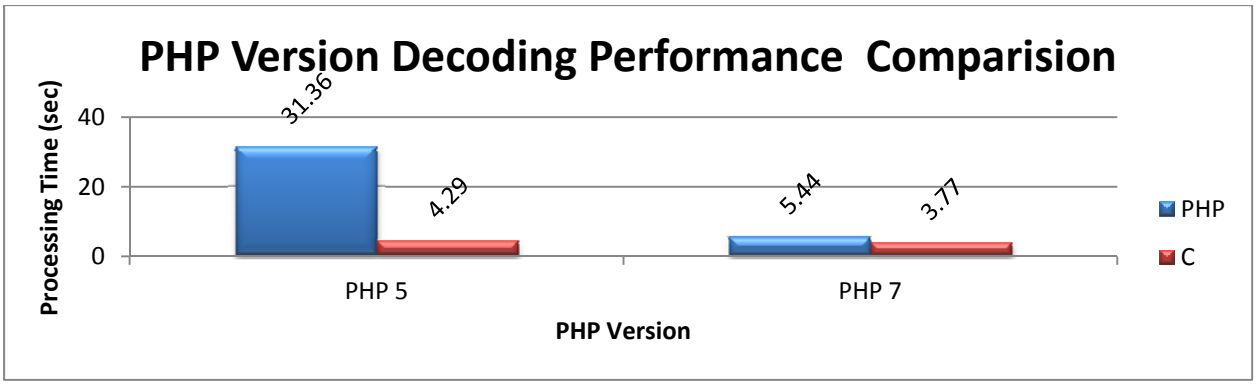

Chart 18 PHP5 Vs PHP7 Decoding Performance Comparison

Browser Testing:

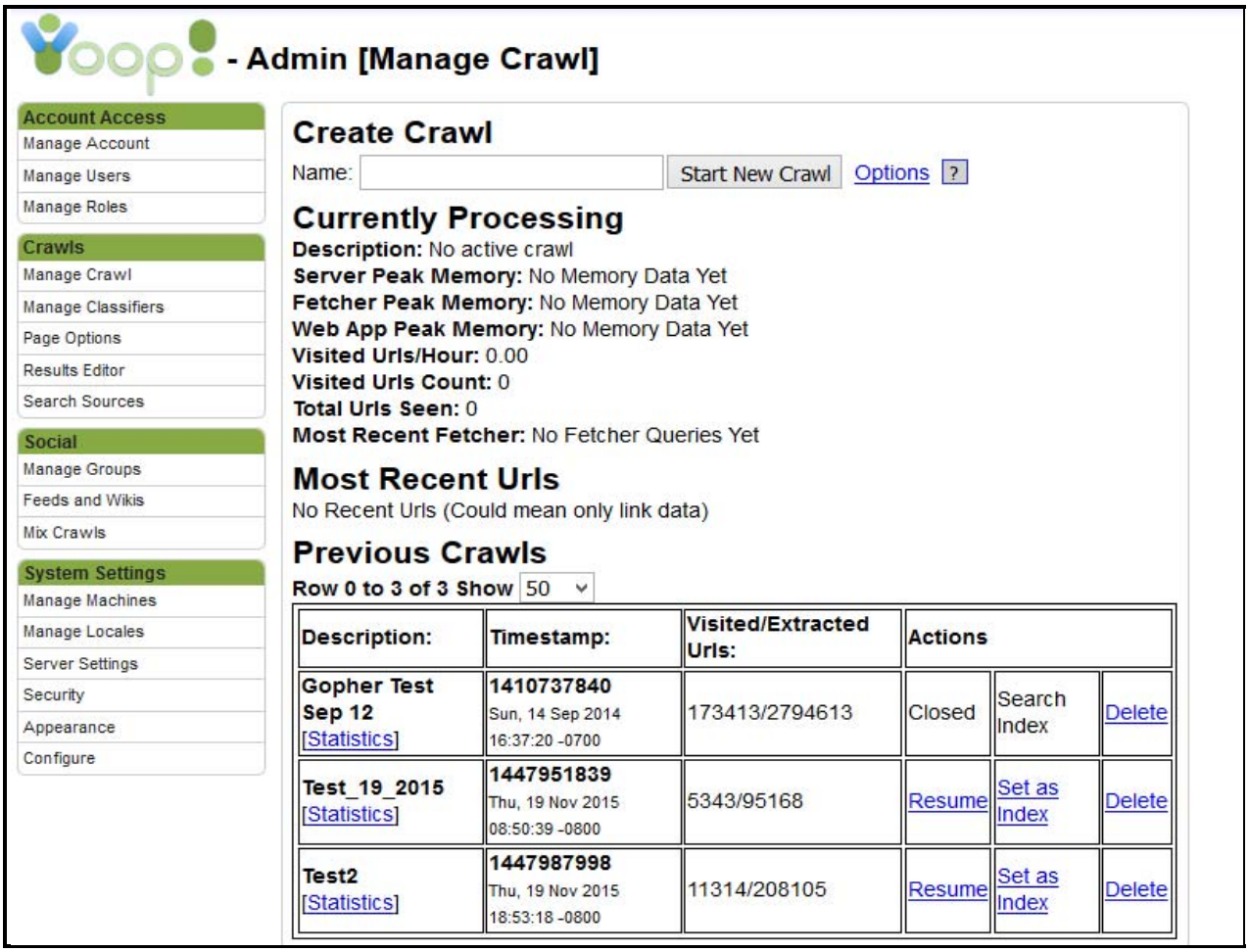

Figure 12 Yioop crawl management screen

PHP:

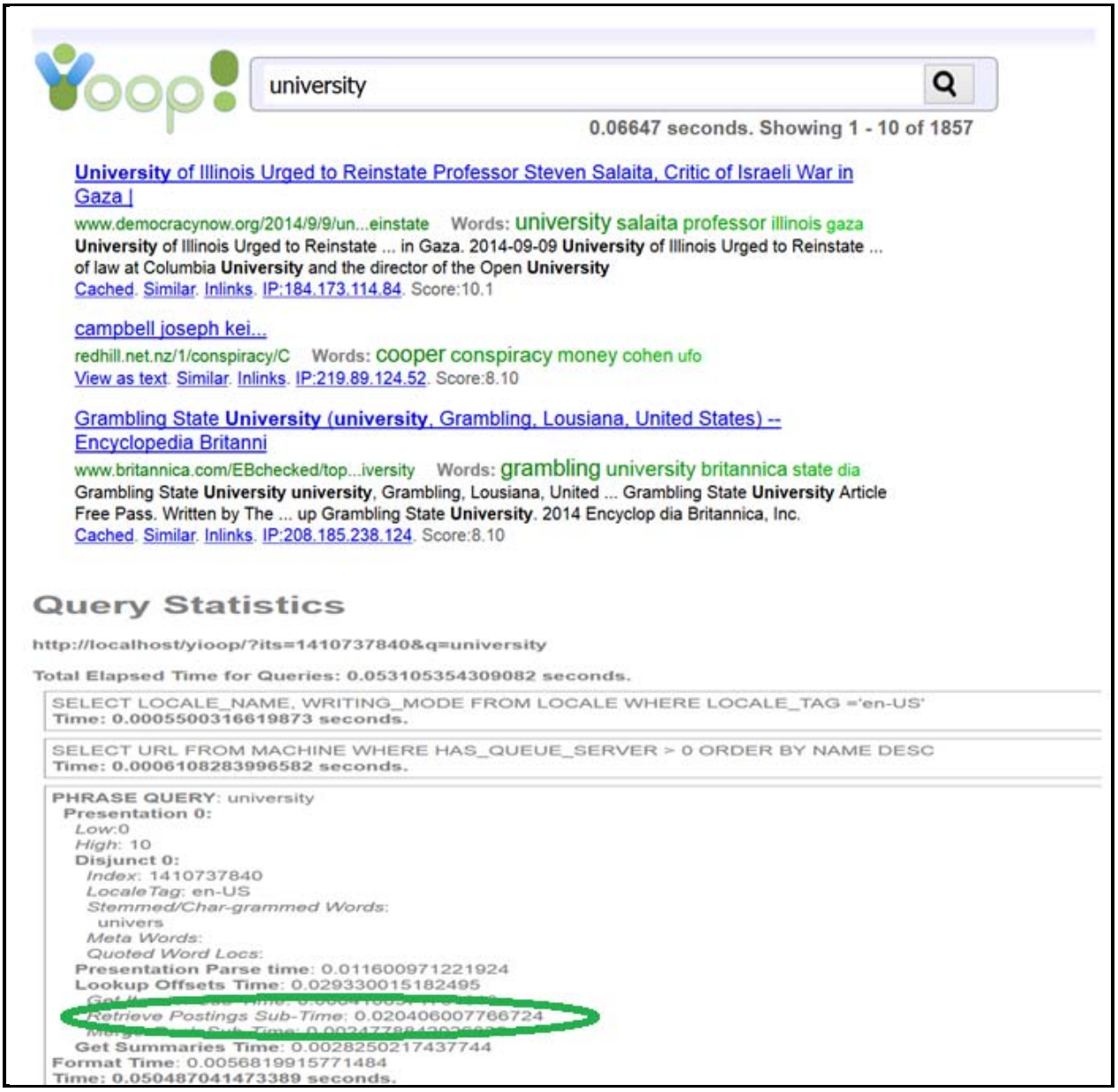

Figure 13 Search results with PHP

Figure 13 above shows the search performance results and corresponding query statistics with the original PHP code. This test was done for a single word for the search criteria.

C Extensions:

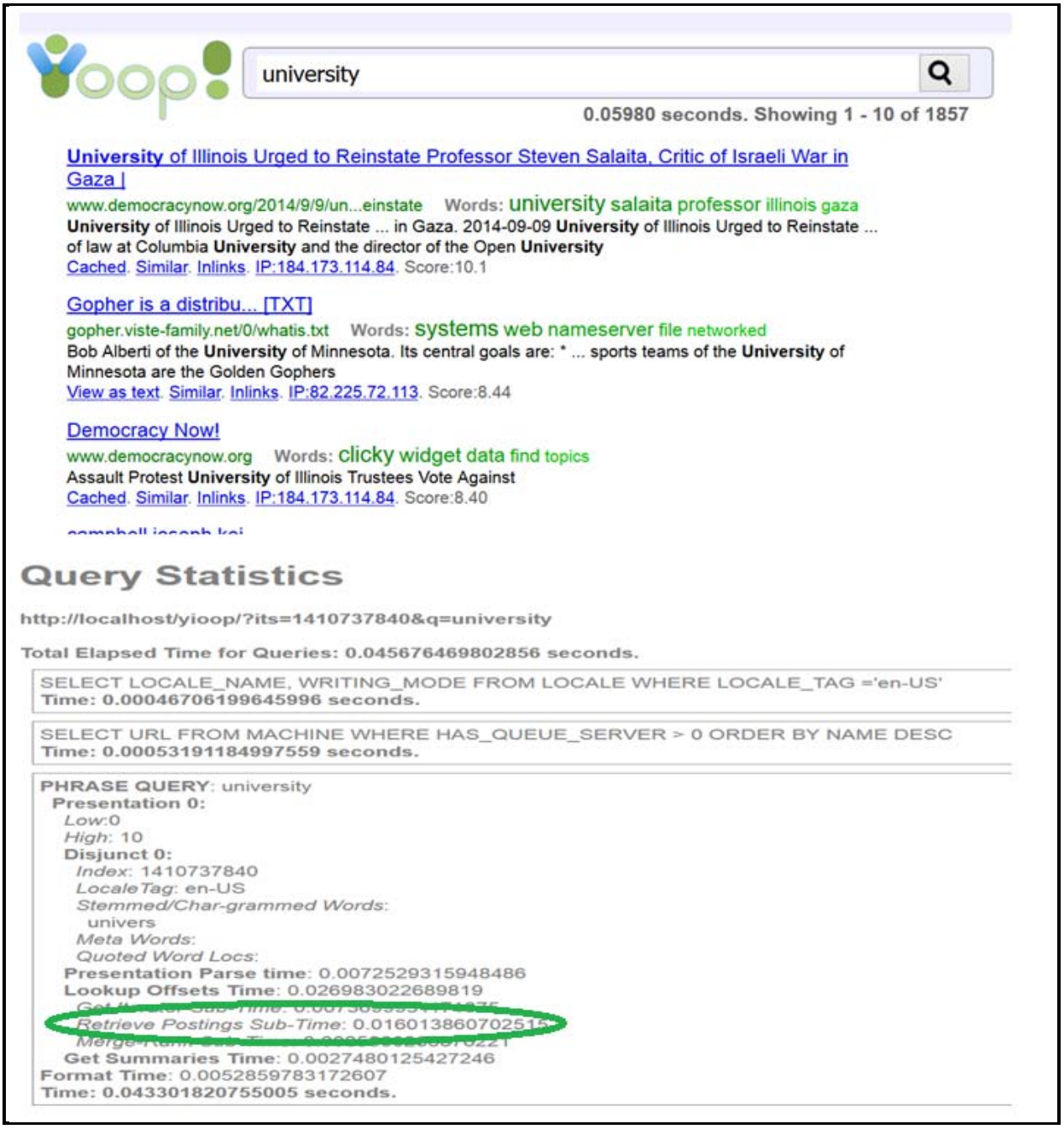

Figure 14 Search results with C Extensions

Figure 14 shows the search performance results and corresponding query statistics with the PHP code replaced with C Extensions. This test was done for the same search word used for PHP search test.

OpenCL Extensions:

Figure 15 Search results with OpenCL Extensions

Figure 15 shows the search performance results and corresponding query statistics with the PHP code replaced with OpenCL Extensions. This test was done for the same search word used for PHP search test.

The following chart shows the browser based query performance test results. This test was done on an *i7* CPU based system with an Nvidia GPU. The test was repeated with three search patterns, a single word, two words, and with three words. The C Extension based test case results have shown about 30% performance improvement as compared to the original PHP code. However, the OpenCL performance did not show improvement because index size is not big enough for OpenCL to perform better. In this case, overhead of OpenCL kernel to host switching is high.

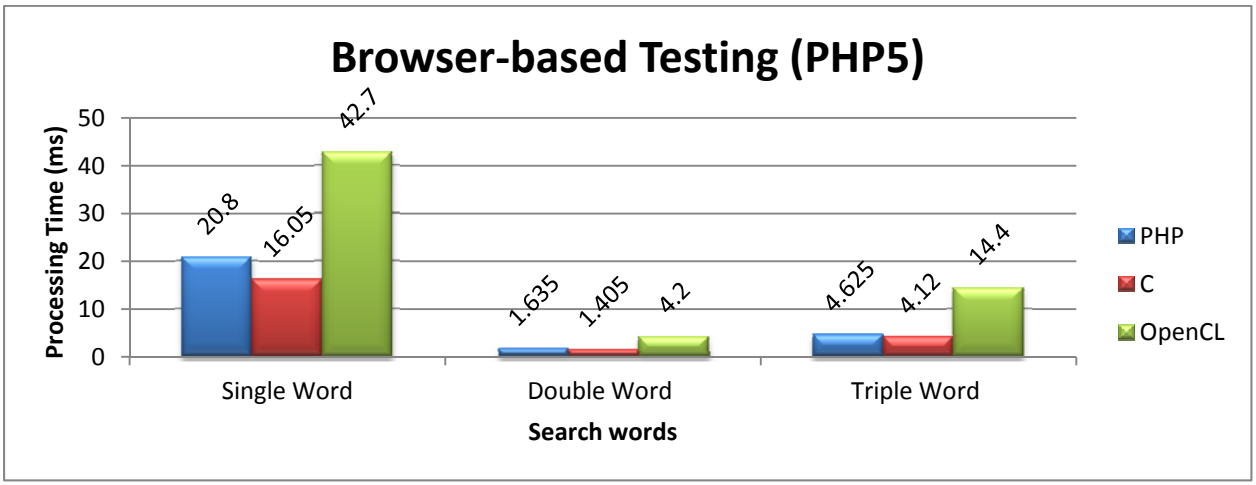

Chart 19 Query Performance from browser (PHP5)

The following chart 20 shows the browser based query performance test results with PHP7. This test was done on *i7* machine for PHP and C Extensions. The C Extension test results have shown about 25% performance improvement compared to the original PHP code.

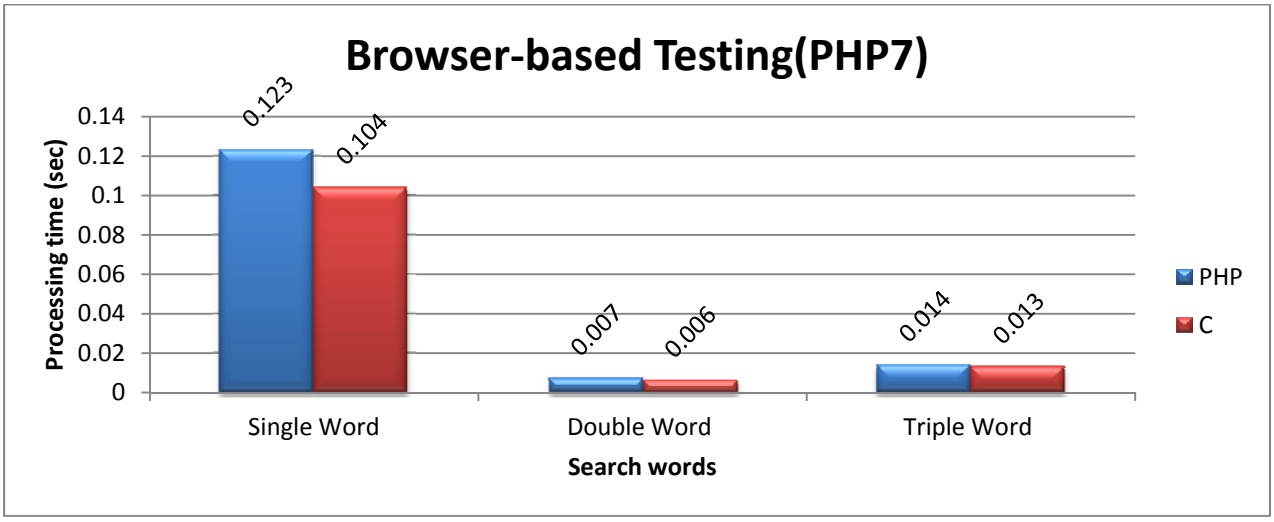

Chart 20 Query Performance from browser (PHP7)

CHAPTER 6

Observations and Conclusions

The goal of improving Yioop's performance has been achieved by experimenting with replacing encoding and decoding functionality with alternatives like C and OpenCL. The following are some of the observations and conclusions arrived at while running the experiments.

C extensions performed 3 times better when compared to the original PHP code for the encoding test case. However, OpenCL has shown only a 0.4 times improvement for the most ranked term on an Nvidia based GPU. The overhead of data transfer between kernel and CPU land outweighs the performance gains achieved through OpenCL parallelism.

C extensions performed 5 times better when compared to the original PHP code for the decoding test case. OpenCL has achieved a 4 times improvement for the most ranked term on an Nvidia based GPU, as well as on an Intel HD graphics GPU. OpenCL performed better when tried with the most ranked terms as it contains a high number of postings. But for other ranked test cases, OpenCL did not yield much improvement, as the overhead of context switching between CPU and GPU is high.

Yioop performed about 2.5 times better for encoding case and about 6 times for the decoding test by simply switching to PHP7. Another 3 times improvement was observed using C Extensions along with PHP7 for the encoding case and about 0.4 times with the decoding test case.

In the interest of empirical rigorousness, performance measurements were done with a 64 bit setup instead of a 32bit setup. There is no performance difference observed with 32 bit versus 64 bit software.

When running tests on an *i7* machine, initial performance numbers were found to be very low, as Windows' defender service was consuming lot of system resources. Stopping the defender service improved the results. When the tests were run on a Windows 10-based system, memory compression service was taking up more system resources than the application itself and skewing the results.

Browser based tests also had shown performance gains when C Extensions were used.

Overall, Yioop's original code is already well optimized and achieving further improvements is not a trivial task. However, these experiments proved that the Yioop search engine's performance can be improved by using a combination of OpenCL and C extensions for most resource and compute intensive functions. The operating system and the right type of GPU and CPU combination will help achieve optimum performance results.

Appendix 1

Environment Setup

The following step by step instructions explain the process of setting the PHP extension development environment on Windows 8.

Prerequisites

1. Microsoft Visual Studio 2013 Professional 32bit
2. Apache 2.4.x 32 bit binary version
3. PHP 5.6.15 Source
4. Intel INDE OpenCL drivers
5. 7Zip Utility

Setting up PHP Extensions for Visual studio 2013

Visual Studio 2013:

- Install Visual Studio 2013 32 bit Professional version.
- Install Visual C++ Redistributable Packages for Visual Studio 2013 from http://www.microsoft.com/en-us/download/confirmation.aspx?id=40784
- Launch Visual studio and make sure it opens fine and its license is fine.
- Exit Visual Studio.

PHP Dev environment:

Ref: The following instructions are based on the steps from https://wiki.php.net/internals/windows/stepbystepbuild.

- Create a directory c:\php-sdk

- Download PHP binary tools from http://windows.php.net/downloads/php-sdk/php-sdk-binary-tools-20110915.zip and extract the contents of the zip to c:\php-sdk. It will create 3 sub folders bin, script and share.
- Open Visual Studio 2013 command window from C:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\Tools\Shortcuts and launch VS2013 x86 Native Tools Command Prompt.
- cd to c:\php-sdk and run the following command

  bin\phpsdk_buildtree.bat phpdev
- Make a copy C:\php-sdk\phpdev\vc9 folder and rename it to C:\php-sdk\phpdev\vc12
- Download a file called deps-5.6-vc11-x86.7z from http://windows.php.net/downloads/php-sdk/. Extract the contents using the 7zip utility and copy the contents of x86 folder to C:\php-sdk\phpdev\vc12\x86
- Download a stable version of PHP source from http://windows.php.net/downloads/releases/php-5.6.15-src.zip and extract the contents to C:\php-sdk\phpdev\vc12\x86\php-5.6.15-src
- From c:\php-sdk run the following commands

  bin\phpsdk_setvars.bat

  cd C:\php-sdk\phpdev\vc12\x86\php-5.6.15-src

  run buildconf

  configure --help

  configure --enable-cli --enable-mbstring --with-curl --with-sqlite3 --with-gd --enable-apache2-4handler --enable-pdo --with-pdo-sqlite --enable-mbregex --with-mcrypt

  nmake

  nmake snap

- Copy php.ini-production from C:\php-sdk\phpdev\vc12\x86\php-5.6.15-src to C:\php-sdk\phpdev\vc12\x86\php-5.6.15-src\Release_TS and rename it php.ini
- Extract ssleay32.dll, libeay32.dll from http://windows.php.net/downloads/releases/php-5.6.15-Win32-VC11-x86.zip) to C:\php-sdk\phpdev\vc12\x86\php-5.6.15-src\Release_TS
- Uncomment the following line in php.ini

  extension=php_curl.dll
- Change ***zend.multibyte = On*** from Off in php.ini

Setting up Apache:

- Download Apache Binary for VC11 compatible 32bit version from http://www.apachelounge.com/download/VC11/
- Extract the zip to c:\apps and Apache will be available under C:\apps\httpd-2.4.17-win32-VC11
- Download PHP Binary http://windows.php.net/downloads/releases/php-5.6.15-Win32-VC11-x86.zip (download thread-safe version only) and extract the zip to a temp folder. Copy libssh2.dll, libeay32.dll and ssleay32.dll from this folder to C:\php-sdk\phpdev\vc12\x86\php-5.6.15-src\Release_TS
- Edit C:\apps\httpd-2.4.7-win64-VC11\Apache24\conf\httpd.conf and add the following lines at the end

  LoadModule php5_module
  "C:/php-sdk/phpdev/vc12/x86/php-5.6.15-src/Release_TS/php5apache2_4.dll"
  AddHandler application/x-httpd-php .php
  PHPIniDir "C:/php-sdk/phpdev/vc12/x86/php-5.6.15-src/Release_TS"

- Also update ServerRoot and DocumentRoot as below in the httpd.conf file

DocumentRoot should look like the following

DocumentRoot "C:/apps/httpd-2.4.17-win32-VC11/Apache24/htdocs/"
<Directory "C:/apps/httpd-2.4.17-win32-VC11/Apache24/htdocs/">

AllowOverride All //Change None to All

- Enable rewrite module by un-commenting the following

  LoadModule rewrite_module modules/mod_rewrite.so

- Start apache from command line
  cd C:\apps\httpd-2.4.17-win32-VC11\Apache24\bin
  httpd.exe

Installing Intel INDE drivers

- Download driver from https://registrationcenter.intel.com/registersninfo.aspx?sn=CPV8-J3767J9J&EmailID=sowmyu94%40yahoo.com&Sequence=1608158&pass=yes

- Copy CL folder from C:\Intel\INDE\code_builder_5.1.0.25\include to C:\php-sdk\phpdev\vc12\x86\php-5.6.15-src

- Copy OpenCL.lib from C:\Intel\INDE\code_builder_5.1.0.25\lib\x86 to C:\php-sdk\phpdev\vc12\x86\deps\lib

Setting and Configuring Yioop

- Send request for access at https://seekquarry.com/download/3.4.0
- Change directory to ***C:\apps\httpd-2.4.17-win32-VC11\Apache24\htdocs***
- Checkout latest version of Yioop from https://seekquarry.com/git/Yioop
- Test Yioop application from http://localhost/Yioop and then follow configuration instructions from https://www.seekquarry.com/p/Documentation to complete the configuration

How to compile and run PHP extension example:

- Create a folder sample under C:\php-sdk\phpdev\vc12\x86\php-5.6.15-src\ext\sample
- Copy conFigurew32, php_sample.h and sample.c code to this folder.
- Copy php.ini-production from C:\php-sdk\phpdev\vc12\x86\php-5.6.15-src to C:\php-sdk\phpdev\vc12\x86\php-5.6.15-src\Release_TS and rename it php.ini
- Uncomment extension_dir and the contents should like

  extension_dir = "C:/php-sdk/phpdev/vc12/x86/php-5.6.15-src/Release_TS/"

  extension=php_sample.dll

- Open Visual Studio command prompt using the shortcut provided above
- cd C:\php-sdk

  bin\phpsdk_setvars.bat

- cd C:\php-sdk\phpdev\vc12\x86\php-5.6.15-src and run

  buildconf

  configure --enable-sample=shared

  nmake php_sample.dll

  cd Release_TS

  php -r "sample_hello_world();"

- You should see output hello world !!!!!!!!!!!

This completes the successful setup of PHP extensions

Note: If nmake fails, stop Apache as it may be locking the dll file and failing to update it.

Appendix 2

Table 5 Machine Configurations

No	Description	Remarks
1	Lenovo Ideapad Intel Core i5-3210M @2.5Ghz Intel HD Graphics 4000N GPU Parallel compute units 16 6GB RAM 420GB HDD Windows 8.1	Used for all i5 based test cases
2	Lenovo W530 Intel Core i7-3840QM @2.8Ghz Nvidia Quadro K2000MN Parallel compute units 2N 12GB RAM 460GB HDD Windows 8.1	Used for all i7 based test cases

References

[1] The open standard for parallel programming of heterogeneous systems. (2016). Retrieved on January 15, 2016 from https://www.khronos.org/opencl/

[2] Extension Writing Part I: Introduction to PHP and Zend. (2015). Retrieved on April 16, 2015 from http://devzone.zend.com/303/extension-writing-part-i-introduction-to-php-and-zend/#Heading2

[3] Woolley, C. (2010). Introduction to OpenCL. Retrieved on November 5, 2015 from http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/06-intro_to_opencl.pdf

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

[5] Benedict, G., Howes, L., Kaeli, D., Mistry, P., Schaa, D. (2011). Heterogeneous Computing with OpenCL. Morgan Kaufmann.

[6] Golemon, Sara. Extending and Embedding PHP. Indianapolis, Ind.: Sams, 2006. Print.

[7] Scholer, F., Williams, H. E., Yiannis, J., and Zobel, J. (2002). *Compression of inverted indexes for fast query evaluation. In Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 222-229. Tampere, Finland.

[8] Che, S., Jiayuan, M., W. Sheaffer, J., Skadron, K., (). A Performance Study of General Purpose Applications on Graphics Processors. Retrieved on September 2, 2015, from https://pdfs.semanticscholar.org/03aa/649535c7e01ac2b3255f2f44131380dc93c7.pdf

[9] Keane, A. (2016). "GPUS ARE ONLY UP TO 14 TIMES FASTER THAN CPUS" SAYS INTEL. Retrieved on March 6, 2016, from https://blogs.nvidia.com/blog/2010/06/23/gpus-are-only-up-to-14-times-faster-than-cpus-says-intel/

[10] Yioop website. Retrieved on September 2, 2015, from http://www.seekquarry.com/