AMD **Accelerated** Parallel Processing TECHNOLOGY

SAMPLE

String Search

1 Overview

1.1 Location

`$(AMDAPPSDKSAMPLESROOT)\samples\opencl\cl\app`

1.2 How to Run

See the *Getting Started* guide for how to build samples. You first must compile the sample.

Use the command line to change to the directory where the executable is located. The pre-compiled sample executable is at `$(AMDAPPSDKSAMPLESROOT)\samples\opencl\bin\x86\` for 32-bit builds, and `$(AMDAPPSDKSAMPLESROOT)\samples\opencl\bin\x86_64\` for 64-bit builds.

Type the following command(s).

1. `StringSearch`
   This command generates the sharpened image of the input image file.
2. `StringSearch -h`
   This command prints the help file.

1.3 Command Line Options

Table 1 lists, and briefly describes, the command line options.

Table 1 Command Line Options

Short Form	Long Form	Description
`-h`	`--help`	Shows all command options and their respective meanings.
	`--device`	Devices on which the program is to be run. Acceptable values are `cpu` or `gpu`.
`-q`	`--quiet`	Quiet mode. Suppresses all text output.
`-e`	`--verify`	Verify results against reference implementation.
`-t`	`--timing`	Print timing-related statistics.
`-i`	`--iterations`	Number of iterations.
`-f`	`--file`	Input file where the pattern will be searched
`-s`	`--substr`	Sub-string to search for (multi-word substrings must be specified within double quotes).
`-c`	`--sensitive`	Case Sensitive (default value is Case Insensitive).
`-v`	`--version`	AMD APP SDK version string.

2 Introduction

The String Search sample demonstrates a workload rebalancing technique in which work-items within a workgroup make use of a local queue to overcome the problem of thread divergence. GPU performance diminishes whenever there is an imbalance in the workload among work-items within a workgroup. This sample provides an approach to overcome this problem by using a work queue created on local memory that feeds work to all the work-items in the workgroup. The approach used is based on the workload rebalancing technique presented by Alexander Lyashevsky[1].

The sample has a naïve implementation of string search that has an inherent thread divergence problem. An optimized version of the string search is also implemented that demonstrates the workload rebalancing technique using a local work queue.

3 Implementation

The sample implements two versions of string search:

1. Naive
2. Load-balanced

3.1 Naive implementation

As the name says, this is the basic version of string search. Each workgroup works on a sub-block of source text and each work-item within the workgroup begins its search from particular position of the source text. An outer loop iterates for the length on the sub-block. In the inner loop, every work-item loops through the pattern length and does a character by character comparison with source text. Some work-items might finish early if the character match fails while others continue if the character match succeeds. This results in thread divergence and workload imbalance within the workgroup.

The pseudo-code for naive string matches is as follows:

```
text // input text
pattern // pattern to search
length = length (text) - length (pattern) + 1
list // list for match positions

foreach (index in length(text)) //outer loop
        for each (l in length (pattern)) //inner loop
        {
   if (text(index+l) != pattern (l)) //char by char comparison
   {
//break from the inner loop if comparison fails. This may happen in in
//one or more work-items causing thread divergence
break
}

//if there is complete pattern match, store the corresponding index
   push (list, index)
         }
```

3.2 Load-balanced implementation

In the load balanced version, a work queue is created on the local memory that is populated with a set of index positions that could potentially match with the pattern. The length of this local queue is such that it can feed work to all work items in the workgroup thereby minimizing workload imbalance.

Load-Balanced String search is performed in three phases:

1. Quick filter on bad positions
2. Filter on partial sub-string match positions
3. Filter all bad positions and get the good positions.

In Phase 1, string matching starts from last two bytes of each sub-string in the text. If last two bytes match we say this is a good position and store the position in a work queue created on local memory. This position will be used in next phase. Here the idea is to make a quick filter on the bad positions by check on 2-bytes and fill the queue sufficient enough so that each work-item has at least one position to continue search further in next phase.

In Phase 2, string matching continues from the matched position obtained from phase-1. This is an intermediate level of string search for large sub-strings or patterns. Here we filter out more positions by continuing the comparison on the next 8/16 bytes. The good positions are saved in the queue for the final phase of string match.

In Phase 3, which is the final phase, string comparison continues from positions as saved in previous phase and matches till the last byte.

The pseudo-code for load-balanced string matches is as follows:

```
text // input text
pattern //pattern to search
pos = 0
length = length (text) - length (pattern) + 1
list // list for match positions

while (true)
        // level-1: quick filter on 2-byte match
        if ((text [pos] == pattern [0]) && (text [pos+1] == pattern [1])))
//store the probable matching index position in a queue
   push (queue1, pos)
        increment (pos)

        if ((queue1 not full) && (pos < length))
        {
   //Load the queue until sufficient work can be fed to all work-items
             continue;
        }

        // level-2: filter on next 8-byte match
        foreach q_pos in queue1
   if (compare (text+q_pos+2, pattern+2, 8))
          //store the probable matching index position in a 2nd queue
           push (queue2, q_pos)
        if ((queue2 not full) && (pos < length)) continue;
```

```
        // level-3: filter on remaining byte match
        foreach q_pos in queue2
   if (compare (text+q_pos+10, pattern+10))
              //if there is complete pattern match, store the corresponding
index
           push (list, q_pos)

        if (pos > length) break
```

4 References

1. http://amddevcentral.com/afds/assets/presentations/2911_1_final.pdf

Contact

Advanced Micro Devices, Inc.
One AMD Place
P.O. Box 3453
Sunnyvale, CA, 94088-3453
Phone: +1.408.749.4000

For AMD Accelerated Parallel Processing:
URL: developer.amd.com/appsdk
Developing: developer.amd.com/
Support: developer.amd.com/appsdksupport
Forum: developer.amd.com/openclforum