SAMPLE

Bitonic Sort

1 Overview

1.1 Location

$<*AMDAPPSDKSamplesInstallPath*>\samples\opencl\cl\1.x

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
$<*AMDAPPSDKSamplesInstallPath*>\samples\opencl\bin\x86\ for 32-bit builds, and
$<*AMDAPPSDKSamplesInstallPath*>\samples\opencl\bin\x86_64\ for 64-bit builds.

Type the following command(s).

1. BitonicSort
   This sorts an array of 64 randomly generated numbers.
2. BitonicSort -h
   This 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 meaning.
	--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.
	--dump	Dump binary image for all devices.
	--load	Load binary image and execute on device.
	--flags	Specify compiler flags to build the kernel.
-p	--platformId	Select the platformId to be used (0 to N-1, where N is the number of available platforms).
-d	--deviceId	Select deviceId to be used (0 to N-1, where N is the number of available devices).
-v	--version	AMD APP SDK version string.
-x	--length	Length of the input array.
-s	--sort	Sort in descending/ascending order.
-i	--iterations	Number of iterations for kernel execution.

2 Introduction

This sample sorts an arbitrary sequence of numbers using a bitonic sorting algorithm. It was chosen because Bitonic Sort is well-suited for parallel architectures. It works by creating bitonic sub-sequences in the original array, starting with sequences of size 4 and continuously merging the sub-sequences to generate bigger bitonic subsequences. Finally, when the size of this subsequence is the size of the array, it means the entire array is a bitonic sequence. Repeating the steps makes the array a part of a bitonic subsequence that is twice the size of the array. Thus, this part of the sub-sequence, the full array, is monotonic (sorted).

It has a constant computational complexity of $O(N * \log_2(N) * \log_2(N))$.

3 Implementation Details

Notes:

- For a detailed description of the algorithm, see references [1] and [2]. at the end of this document.
- This implementation is slightly different from the algorithm described in [1].
- Note: For brevity, "increasing" denotes "non-decreasing," and "decreasing" denotes "non-increasing."
- This algorithm works only for arrays with length of a power of 2. We assume the length of the array is 2^N.

For an array of length 2^N, the sorting is done in N stages. The first stage has one pass; the second has two passes; the third stage has three passes, and so on.

Every pass does length/2 comparisons; that is, $2^{(N-1)}$. Let us call this value *numComparisons*. The kernel compares once and writes out two values. So, for every pass there are numComparisons invocations of the kernel; that is, numComparisons is the number of threads to be invoked per pass.

3.1 Stages

The first stage converts the unsorted array into *segments* of length 2. The first pair of adjacent segments form a bitonic sequence. The next pair (the third and fourth segments) form a bitonic sequence, and, similarly, every pair of odd and successive even segments form a bitonic sequence.

The second stage converts the sequence resulting from the first stage into a sequence where the segments are of length 4. This means, the first pair of 4-element segments forms a bitonic sequence, the second pair (third and fourth 4-element segments) form a bitonic sequence, and so on. The size of the segment now is doubled. Every segment is monotonic; that is, either uniformly increasing or uniformly decrasing.

Similarly, the result of the third stage is pairs of adjacent 8-element segments forming a bitonic sequence. If the first stage is treated as converting length 1 segments into length 2 segments, then every stage doubles the size of the segment. When the segment size is equal to the length of the array, the result is a sorted array, because the other half of the bitonic sequence is outside the bounds of the array, and the first half is a monotonic sequence. This doubling of segment size is done in multiple steps (passes) in each stage.

3.2 Detailing One Pass

Consider stage 2 of Table 2.

The first pass compares elements at a distance of 4 and swaps them depending on the sorting direction for that *block* (eight successive elements in this pass). If the sort direction for that block is increasing, the lesser value is kept to the left. For the next pass, the same thing is done, but the block width now is 4. This is repeated until the block width in a pass is 2. At the end of every pass, every block is made such that every element of the left half of a block is less than the corresponding element of the right half of the same block (or greater than it, if the sort direction for that block is decreasing order). The distance between the elements being compared is half the width of the block. This is known as *pairDistance*.

The first stage inverts the direction of sorting between every successive pair of elements. In the second stage, this inversion is done after two pairs of elements (four elements). This subsection of the array, in which the direction of comparison is the same, is termed a *sameDirectionBlock*. A sameDirectionBlock is measured in terms of the number of elements pairs. So, the first stage has a sameDirectionBlockSize of 1; the second one has a sameDirectionBlockSize of 2; the third 4, and so on.

Table 2 illustrates the stages, passes and comparisons made in a bitonic sort, in increasing order, of a 16-element array. The stages, passes, threads, and elements are numbered starting from 0. In this table, 0<1 implies:

- The pass compares elements at indices 0 and 1.
- The pass compares to sort these two in increasing order.
- It ensures that it writes to these two locations in that order (swapping them if necessary).

Table 2 Bitonic Sort Stages, Passes, and Comparisons

Stage	Pass of Stage	Thread ID								Pair Distance	Block Width	Same Direction Block Size
		0	1	2	3	4	5	6	7			
0	0	0<1	2>3	4<5	6>7	8<9	10>11	12<13	14>15	1	2	1
1	0	0<2	1<3	4>6	5>7	8<10	9<11	12>14	13>15	2	4	2
1	1	0<1	2<3	4>5	6>7	8<9	10<11	12>13	14>15	1	2	2
2	0	0<4	1<5	2<6	3<7	8>12	9>13	10>14	11>15	4	8	4
2	1	0<2	1<3	4<6	5<7	8>10	9>11	12>14	13>15	2	4	4
2	2	0<1	2<3	4<5	6<7	8>9	10>11	12>13	14>15	1	2	4
3	0	0<8	1<9	2<10	3<11	4<12	5<13	6<14	7<15	8	16	8
3	1	0<4	1<5	2<6	3<7	8<12	9<13	10<14	11<15	4	8	8
3	2	0<2	1<3	4<6	5<7	8<10	9<11	12<14	13<15	2	4	8
3	3	0<1	2<3	4<5	6<7	8<9	10<11	12<13	14<15	1	2	8

PairDistance = $2^{(\text{Stage} - \text{PassOfStage})}$

This is the distance between the pair of elements that are being compared in each invocation/call of the thread.

BlockWidth = $2*2^{(\text{Stage} - \text{PassOfStage})}$

A block is a group of consecutive elements in the array that are being compared. The width of the block is defined by BlockWidth.

SameDirectionBlockSize = 2^{stage}

This is the number of comparisons in a stage that are monotonically increasing or monotonically decreasing. For instance, in Stage 2 the first four comparisons are increasing comparisons; the next four are decreasing comparisons. Thus, the SameDirectionBlockSize is 4 for stage 2.

Leftid = (threadId % pairDistance) + (threadId / pairDistance) * blockwidth
Rightid = Leftid + pariwisedist

Leftid and Rightid are the indices of the numbers to be compared in the invocation of the work thread. The smaller of the two indices in the array is the LeftId; the larger one is the RightId. The number at LeftId is always to the left of the number at RightId in the array.

In our implementation, thread 0 compares the element at index 0 to the corresponding index on its right. The right element is chosen based on pairDistance for that stage and pass. Thread 1 takes the first element that thread 0 does not handle, and uses it as Leftid. As before, Rightid is computed from Leftid. Continuing this pattern, one can see that thread K takes the first element not covered by threads 0 to K-1, and uses it as Leftid. The equations and the table above help show how Leftid can be derived from threadid.

Once a thread has Leftid, Rightid, and sortDirection, it can compare (and swap, if necessary) and write to the same locations.

4 References

1. http://facultyfp.salisbury.edu/taanastasio/COSC490/Fall03/Lectures/Sorting/bitonic.pdf Explanation of Bitonic sort with examples by Thomas Anasosio.
2. http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm

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