Multi-stream 2D frequency table computation on dataflow architecture

P. Škoda*, V. Sruk**, and B. Medved Rogina*

* Ruđer Bošković Institute, Zagreb, Croatia
** Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia
pskoda@irb.hr

Abstract – Frequency table computation is a common procedure used in variety of machine learning algorithms. In this paper we present a parallelized kernel for computing frequency tables. The kernel is targeted for dataflow architecture implemented on field programmable gate array (FPGA). Its performance was evaluated against a parallelized software implementation running on a 6-core CPU. The kernel with six concurrent input data streams running on 300 MHz achieved speedup of up to 6.26x, compared to 6 threaded software implementation running on 3.2 GHz CPU.

Keywords – frequency table, dataflow, field programmable gate arrays, decision tree learning

I. INTRODUCTION

The 2D frequency table is a two-dimensional extension of histogram. In machine learning context, the frequency table is often used to provide information on how frequent specific combinations of attribute and class values are in a given dataset. This specific use is a key part of the attribute selection process in decision tree learning algorithms [1]. Applications of decision tree learning, as well as the field of data mining in general, are faced with continuing increase in dataset sizes. This drives efforts to develop new faster and more efficient algorithms, as well as developing new hardware platforms that provide high computational power [2].

One hardware platform that is gaining more ground in computing is the field programmable gate array (FPGA). FPGAs are digital integrated circuits designed to be user-configured after manufacturing [3], [4]. They enable fast development and deployment of custom digital hardware, and thus allow implementation of custom computational architectures that can be reconfigured on demand.

The most suitable computational mode for FPGA is the dataflow model. Most computer systems implement a control-flow model, in which the computation is described by a sequence of operations that are executed in specified order. In dataflow model, computation is described by chain of transformations applied to a stream of data [5]. The dataflow is described in form of a graph. Nodes of the graph represent operations, and edges represent data links between the nodes. The dataflow graph maps naturally to the hardware implementation. Nodes/operations are translated to functional hardware blocks, while edges are translated to data buses and signals. By dividing the graph into stages separated by registers, it’s transformed into a pipelined structure suitable for implementation on FPGA.

In this paper we present a novel implementation of 2D frequency table computation targeted for dataflow architecture realized on FPGA. The kernel is developed for integration with C4.5 decision tree learning program [6], [7]. Its result is used to calculate dataset split quality, using an entropy based measure, which is then used for attribute test selection.

II. RELATED WORK

2D frequency table computation implemented on FPGA has been previously published as a part of efforts in implementing decision tree learning algorithms. FPGAs have been proven to be suitable of implementing data mining algorithms, though so far little attention has been directed to decision tree learning algorithms. Only a few attempts of using FPGAs to accelerate execution of decision tree learning have been reported [8], [9].

Chrysoς et al. [9] implement frequency table computation on FPGA as part of HC-CART system for learning decision trees. Frequency table computation is implemented in Frequency Counting module, which receives attribute-class label pairs. The pairs are used to generate addresses for frequency table locations which contents are then incremented by one. In their implementation, all attribute-class label pairs received by the kernel have to be prepared by program running on CPU, and then preloaded to memory attached to FPGA. Input data is transferred from CPU to FPGA memory many times in the course of program execution.

In our previous work [10], we implemented a 2D frequency table computation using FPGAs. The ComputeFreq kernel receives two input streams – one for address and one for class labels – and counts occurrences of pairs in FPGA’s internal memory. The dataset is held in FPGA attached SDRAM. The dataset is transferred to the FPGA’s SDRAM only once, and only data transfer between the kernel and the host system is the transfer of computed frequency tables. We build upon this work by expanding it to multiple attribute streams which allow computation of several tables simultaneously, and by implementing efficient data transfer for the computed tables.
III. DATAFLOW ENGINE ARCHITECTURE

The accelerator is implemented in form of a dataflow engine (DFE) which consists of at least one kernel and a manager. The computation is implemented by kernels, while the role of manager is to organize data movement within the DFE. Kernels are instantiated within the manager, and data links between the kernels, external RAM, and CPU are defined.

The frequency table computation was implemented using the Maxeler platform [11]. The platform includes an FPGA board, drivers, and API. The DFE is coded in Java, using the API which provides objects and methods that are translated to hardware units by the compiler. The compiler translates the Java code into VHDL, and generates software interface for CPU code [12]. The DFE was implemented on Maxeler Vectis-Lite board, which contains a single FPGA, and on-board SDRAM. The board is connected to the host PC workstation via PCIe bus. Basic information on the board is shown in Table I.

A. Kernel architecture

Architecture of the ComputeFreq-MS kernel is shown in Fig. 1. The kernel has two scalar inputs for parameters: one for number of items to process (items), and one for stream length in items (strmLen). Stream length sets the number of items to read from the on-board SDRAM, which requires all accesses to be made in 96 byte blocks, i.e. 24 item blocks (4 bytes per item). The kernel processes the first items elements from the stream, while the rest are read from memory and ignored.

The kernel consists of six identical frequency counter structures. Each frequency counter receives two input streams – one for attribute and one for class values. Both streams carry 32 bit unsigned integers. The input streams are sliced so the low \( N_t \) bits retained from the attribute, and \( N_c \) bits are retained from the class values. The retained bits are concatenated to form the frequency table address. In the table address, the attribute value is the high, and the class value is the low part of the address word.

The kernel has a total of seven stream inputs, six for attribute (\( att_0 – att_5 \)), and one for class (\( class \)) value streams. Each attribute value stream is connected to a single frequency counter. The class value stream fans out to all frequency counters, so that they all receive identical class value streams.

The frequency tables are stored in on-chip memory – block RAM (BRAM). Each frequency counter has one BRAM which is addressed by the address formed from the low bits of attribute and class streams. The addressed location is read, its value incremented by one, and written back to the same location. The BRAM is configured as single port RAM with read-first synchronization. Due to access latencies of the BRAM itself, and the registers inserted by the compiler, there are latencies inside the loop. To ensure correct counting, the kernel’s input streams are throttled to allow input of one element each \( m \) clock cycles, where \( m \) is the internal latency of the loop.

After all elements of interest are streamed in, the contents of the BRAMs are streamed out to the host main memory through stream output \( s \). The output stream \( s \) is made by joining output streams from all frequency counters (\( s_0 – s_5 \)) into a one vector stream. The output vector is padded to 8 elements, i.e. two additional dummy elements are added to it. The padding enables efficient conversion of the output stream to word width of 128 bits which is used in communication over PCIe bus.

As BRAMs are read, they are at the same time reset to zero. During the read phase, the BRAMs are addressed by a counter, and their write ports are switched from increment-by-one values to constant zero. In this case there are no loop dependencies, and the output stream is not throttled – it outputs one element every clock cycle. At the same time, any remaining elements in the input stream are read un-throttled (one per clock) and ignored.

B. Manager

A single ComputeFreq-MS kernel is instantiated in manager. Input streams \( attClass \) and \( att_0 – att_5 \) are linked to on-board SDRAM. All input streams use linear memory access pattern. Output stream \( s \) is linked to

TABLE I. BASIC INFORMATION ON MAXELER VECTIS-LITE FPGA BOARD

<table>
<thead>
<tr>
<th>Resource</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td>Xilinx Virtex-6 XC7VSX475T</td>
</tr>
<tr>
<td>Off-chip RAM (LMem)</td>
<td>6 Gb (6x 1 Gb) DDR3-800 SDRAM</td>
</tr>
<tr>
<td>On-chip RAM (FMem)</td>
<td>~ 4 MiB Block RAM</td>
</tr>
<tr>
<td>CPU ↔ FPGA bandwidth</td>
<td>2 GB/s</td>
</tr>
<tr>
<td>LMem ↔ FPGA bandwidth</td>
<td>38.4 GB/s max.</td>
</tr>
</tbody>
</table>

TABLE II. FPGA RESOURCE USAGE BY THE DFE

<table>
<thead>
<tr>
<th>Resource</th>
<th>Used</th>
<th>Total available</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs</td>
<td>34,724</td>
<td>297,600</td>
<td>11.67 %</td>
</tr>
<tr>
<td>Flip-flops</td>
<td>50,064</td>
<td>297,600</td>
<td>16.82 %</td>
</tr>
<tr>
<td>DSP blocks</td>
<td>0</td>
<td>2,016</td>
<td>0.00 %</td>
</tr>
<tr>
<td>Block RAM</td>
<td>226</td>
<td>2,128</td>
<td>10.62 %</td>
</tr>
</tbody>
</table>
computer main memory. Source addresses for input streams \( att_0 – atts, \) \( attClass, \) values for scalar inputs \( items \) and \( strmLen, \) and destination address for output stream \( s \) are defined through the generated DFE interface.

The kernel parameters were set for up to 64 unique attribute and class values \((N_A = 64, N_C = 64)\). Each frequency table holds 4096 32-bit words. Due to the frequency counter’s loop latency, it can process one item \((\text{attribute} - \text{class} \text{pair})\) every five clock cycles. Kernel clock frequency is set to 300 MHz. With six attribute streams, the DFE can process up to \( 360 \times 10^6 \) elements per second. FPGA resource usage for the DFE is given in Table II.

IV. EXPERIMENTAL RESULTS

A. Test environment

The kernel was benchmarked using code form C4.5 Release 8 decision tree learning program [13]. Parts needed to load the datasets, and computing the frequency tables were extracted from the program and used for benchmarking. The frequency table computation was parallelized using OpenMP [14] by distributing attributes of the dataset over the threads. This included replicating the frequency table data structure to accommodate multi-threaded execution. The original \textit{ComputeFrequencies} function was modified to use the replicated data structure. Since the attributes were distributed over threads, multiple invocations of \textit{ComputeFrequencies} function were necessary if number of attributes exceeded the number of threads.

Functions for transforming the dataset to the appropriate format, and loading it to the DFE were added to the benchmark program. For execution on DFE, a new function was created by modifying the original \textit{ComputeFrequencies} function. Modifications involve removing the computation loop and replacing it with function calls to the DFE. Received results are transformed into the frequency table data structure used by the parallelized software implementation. Since the DFE processes six attributes in parallel, the \textit{ComputeFrequencies} function is invoked once for every group of six attributes.

Time measurement was added to the \textit{ComputeFrequencies} function. Entire run time of the function is measured. The function’s throughput is calculated from the measured times using the formula:

\[
T_{a,i} = \frac{a \cdot i}{t_{a,i}}
\]  
(1)

where \( a \) is number of attributes, \( i \) is number of items, \( t_{a,i} \) is function execution time, and \( T_{a,i} \) is the calculated function throughput for the dataset with \( a \) attributes, and \( i \) items.

A set of 102 synthetic datasets were used in the benchmark. Number of attributes ranged from 6 to 1,536, in a geometric sequence \( 6 \times 2^i \), with \( i = 0, 2,..., 7 \). Number of items ranged from \( 2048 \) to \( 4 \times 2^{20} \), in a geometric sequence \( 2^i \), with \( i = 11, 12,..., 22 \). The dataset size was limited to 3 Gb, i.e. to \( 768 \times 2^{10} \) elements. Datasets were generated randomly, with uniformly distributed values in range of 1-63 for attribute values, and 0-64 for class values.

Benchmark program was compiled using gcc compiler, version 4.4.7. Benchmarks were run on Intel Xeon E5-1600 workstation, with 16 GiB DDR3-1600 RAM, under Centos 6.5 Linux OS.

B. CPU benchmark results

For the baseline CPU performance, the measurements were conducted on a six-threaded parallelized implementation of \textit{ComputeFrequencies} function. Measured execution times are given in Table III, and are shown in Fig. 2. For datasets with over \( 128 \times 2^{10} \) items, execution time scales approximately linearly with number of items. There is a larger increase in execution in area of \( 16 \times 2^{10} – 32 \times 2^{10} \) items, for datasets with 96 or more attributes, and \( 32 \times 2^{10} – 256 \times 2^{10} \) for datasets with less than 96 attributes.

This increase in execution is more obviously visible in function throughput, shown in Fig. 3, as a sharp drop of the throughput. From the figure it’s visible that for datasets with fewer items, the throughput increases with increasing number of items. Once the dataset exceeds a certain number of items (dependent on number of attributes), throughput drops, and quickly stabilizes to an approximately constant value. As number of attributes increases, function throughput decreases, assuming equal...
number of items. The number of items at which peak throughput is recorded, also decreases with increasing number of attributes, as well as the peak throughput value itself. For datasets with 384 or more attributes, there is no increase of throughput. Throughput just drops from one approximately constant value to another.

Maximum measured constant (invariant to number of items) throughput is $617 \times 10^6$ elements/s, achieved on dataset with 12 attributes. Minimal measured constant throughput is $52.2 \times 10^6$ elements/s, measured on dataset with 1,536 attributes.

The drop in function throughput is most likely a consequence of interaction of CPUs cache system and main SDRAM. Smaller datasets have a smaller likelihood of cache miss, which translates into higher throughput for smaller datasets. Another important factor is the data structure for storing $ht$ dataset. The dataset is stored in attribute major order, which leads to memory accesses with stride. With more attributes, the stride is larger, and consequentially, so is the likelihood of cache miss.

C. DFE benchmark results

DFE was benchmarked under the same condition as the six-threaded software implementation. Execution times measured on DFE are given in Table IV, and shown in Fig 4. Execution time on DFE scales approximately linearly with number of items in the dataset, on datasets with $512 \times 2^{10}$ or more items. On datasets with fewer items, the execution times asymptotically approach a certain minimal value, dependent on number of attributes. For a fixed number of items, the execution times scale linearly with number of attributes.

The kernel throughput graph, shown in Fig 5, shows that curves for all number of attributes form a tight group. The curves for different number of attributes cannot be distinguished one from another. On datasets with $512 \times 2^{10}$ items, the throughput is approximately constant at $340 \times 10^6$ elements/s. The measured throughput is close to the theoretical maximum of $360 \times 10^6$ elements/s.

The DFE performs better with datasets with larger number of items, close to theoretical maximum when number of items is $512 \times 2^{10}$ or more. This is a consequence of communication and control overheads between the DFE and the host computer. The overhead is independent from number of items, and its influence diminishes as the number of items increases. As can be seen from Table IV, execution times are approximately constant for datasets with 8,192 or less items. The minimum DFE execution time was calculated from these measurements, amounting to 801.1 $\mu$s.

D. Comparison of the results

DFE and CPU results were compared by computing speedup, i.e. ratio of execution times on CPU and on DFE. The speedup is shown in Fig 6. For most of the test dataset sizes, execution on DFE is slower than on CPU. For datasets with fewer than 48 attributes, the DFE is slower no matter what the data set size is. For 48 attributes the speedup exceeds unity value for numbers of items $128 \times 2^{10}$ or more. The maximum speedup is achieved on datasets with 384 and more attributes. Speedups on these datasets are close in value, suggesting that by further increasing number of attributes will result in negligible increase in speedup. These values can then be used as approximation of the upper bound on speedup. The maximum speedup measured is $6.26 \times$, achieved on dataset with 1,546 attributes and $512 \times 2^{10}$ items.

The total performance of the DFE was estimated by calculating the weighted average of speedups, weighted by number of elements in the dataset.
TABLE III. MEASURED EXECUTION TIMES ON CPU

<table>
<thead>
<tr>
<th>Number of items</th>
<th>6</th>
<th>12</th>
<th>24</th>
<th>48</th>
<th>96</th>
<th>192</th>
<th>384</th>
<th>768</th>
<th>1,536</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,048</td>
<td>50.59 µs</td>
<td>105.3 µs</td>
<td>215.0 µs</td>
<td>434.9 µs</td>
<td>916.9 µs</td>
<td>1,769 ms</td>
<td>6,049 ms</td>
<td>8,887 ms</td>
<td>21.32 ms</td>
</tr>
<tr>
<td>4,096</td>
<td>63.31 µs</td>
<td>135.5 µs</td>
<td>280.4 µs</td>
<td>571.0 µs</td>
<td>1,237 ms</td>
<td>3,096 ms</td>
<td>10,59 ms</td>
<td>14,37 ms</td>
<td>42.51 ms</td>
</tr>
<tr>
<td>8,192</td>
<td>88.21 µs</td>
<td>190.1 µs</td>
<td>403.6 µs</td>
<td>847.8 µs</td>
<td>2,226 ms</td>
<td>5,027 ms</td>
<td>19.56 ms</td>
<td>37.46 ms</td>
<td>89.34 ms</td>
</tr>
<tr>
<td>16,384</td>
<td>139.2 µs</td>
<td>302.8 µs</td>
<td>668.8 µs</td>
<td>1,557 ms</td>
<td>3,321 ms</td>
<td>13,08 ms</td>
<td>41.85 ms</td>
<td>91.15 ms</td>
<td>195.6 ms</td>
</tr>
<tr>
<td>32,768</td>
<td>245.6 µs</td>
<td>538.0 µs</td>
<td>1,219 ms</td>
<td>2,867 ms</td>
<td>13,996 ms</td>
<td>46,18 ms</td>
<td>156.3 ms</td>
<td>360.6 ms</td>
<td>803.1 ms</td>
</tr>
<tr>
<td>65,536</td>
<td>462.1 µs</td>
<td>1,014 ms</td>
<td>2,296 ms</td>
<td>10,92 ms</td>
<td>35,06 ms</td>
<td>97,45 ms</td>
<td>375.2 ms</td>
<td>808.1 ms</td>
<td>1,860 s</td>
</tr>
<tr>
<td>131,072</td>
<td>899.0 µs</td>
<td>2,066 ms</td>
<td>6,949 ms</td>
<td>25,08 ms</td>
<td>62,94 ms</td>
<td>191.5 ms</td>
<td>706.2 ms</td>
<td>1,702 s</td>
<td>3,795 s</td>
</tr>
<tr>
<td>262,144</td>
<td>2,056 ms</td>
<td>5,184 ms</td>
<td>14,48 ms</td>
<td>49,73 ms</td>
<td>132,1 ms</td>
<td>382.5 ms</td>
<td>1,518 s</td>
<td>3,467 s</td>
<td>7,658 s</td>
</tr>
<tr>
<td>524,288</td>
<td>4,215 ms</td>
<td>10,35 ms</td>
<td>28,61 ms</td>
<td>98,80 ms</td>
<td>267,1 ms</td>
<td>766.7 ms</td>
<td>3,079 s</td>
<td>6,987 s</td>
<td>15,44 s</td>
</tr>
<tr>
<td>1,048,576</td>
<td>8,365 ms</td>
<td>20,48 ms</td>
<td>56,80 ms</td>
<td>197,3 ms</td>
<td>537,8 ms</td>
<td>1,553 s</td>
<td>6,359 s</td>
<td>14,01 s</td>
<td>–</td>
</tr>
<tr>
<td>2,097,152</td>
<td>16,70 ms</td>
<td>40,80 ms</td>
<td>113,3 ms</td>
<td>394,1 ms</td>
<td>1,071 s</td>
<td>3,176 s</td>
<td>13,21 s</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>4,194,304</td>
<td>33,49 ms</td>
<td>81,27 ms</td>
<td>226,3 ms</td>
<td>787,0 ms</td>
<td>2,139 s</td>
<td>6,303 s</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

\[ S = \sum_{a} \sum_{n} \frac{a n s_{a,n}}{\sum_{a} \sum_{n} a n} \] (1)

where \( S \) is the average speedup, and \( s_{a,n} \) is the speedup on dataset with \( a \) attributes and \( n \) items. The average speedup is 4.1.

For further comparison, execution efficiency was computed for CPU and DFE. The efficiency is defined as:

\[ E_{a,n} = \frac{m f}{F_{a,n}} \] (1)

where \( E_{a,n} \) is efficiency and \( F_{a,n} \) is throughput for dataset with \( a \) attributes and \( n \) items, \( f \) is clock frequency, and \( m \) is number of threads or number of streams on CPU and DFE respectively. Peak throughput values were used to calculate efficiency.

The software implementation (CPU baseline) was executed on six threads, on CPU clocked at 3.2 GHz. Peak throughput was \( 875 \times 10^6 \) elements/s. The DFE uses six streams, and was clocked at 300 MHz. Peak throughput on DFE was \( 355 \times 10^6 \) elements/s. Execution efficiency on CPU is 21.9 clock cycles per element, and on DFE it’s 5.08 clock cycles per element. Of the 5.08 clocks, 4 are consequence of the frequency counter’s internal loop latency.

V. CONCLUSION

In this paper we presented a multi-streamed compute architecture for 2D frequency matrix computation, implemented on FPGA platform. This multi-streamed architecture is an advancement on our previous work [10].

Benchmark results reveal that CPU outperforms DFE for smaller datasets. This is a consequence of control and communication latencies between the FPGA board and the host computer. Minimal time required to execute any action on DFE is approximately 800 µs, regardless of dataset size. Small datasets are easily processed by the CPU in less time. The DFE outperforms CPU for larger datasets, that have at least 48 attributes, \( 32 \times 2^{10} \) items, and \( 6 \times 2^{20} \) elements. Best speedup achieved by DFE is 6.26x. The DFE is more efficient in processing data, requiring only 5.08 clock cycles per dataset element, while CPU requires 21.9, when performing at peak efficiency. Of the DFE’s 5.08 clock cycles, 4 cycles are consequence of frequency counter’s internal loop latencies.

The kernel can be further improved by vectorizing inputs stream in the same manner its output was vectorized. This will allow efficient processing of more...
elements in parallel, and better utilization of available memory bandwidth. Another improvement would be reducing or eliminating the internal loop latencies. This can potentially quadruple its performance assuming that operating frequency of the kernel does not drop due to higher design complexity.

In future work, in addition to the stated improvements, this kernel will be integrated in the C4.5 program as an accelerator unit. However, to achieve significant performance gains the communication latency between host computer and FPGA will have to be compensated for. This will most likely involve adding support for batch processing of a series of small dataset, and require some modifications of the original algorithm. Overall, with additional improvements the kernel is expected to outperform the CPU for a wider range of dataset sizes.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Union’s Seventh Framework Programme (FP7), under grant agreement no. 317882 – E2LP Embedded Computer Engineering Learning Platform.

REFERENCES


