Acceleration of Frequent Itemset Mining on FPGA Using SDAccel and Vivado HLS

Vinh Dang and Kevin Skadron
Department of Computer Science, University of Virginia, Charlottesville, VA 22904, USA
vqd8a@virginia.edu, skadron@virginia.edu

***Abstract*—Frequent itemset mining (FIM) is a widely-used data-mining technique for discovering sets of frequently-occurring items in large databases. However, FIM is highly time-consuming when datasets grow in size. FPGAs have shown great promise for accelerating computationally-intensive algorithms, but they are hard to use with traditional HDL-based design methods. The recent introduction of Xilinx SDAccel development environment for the C/C++/OpenCL languages allows developers to utilize FPGA's potential without long development periods and extensive hardware knowledge. This paper presents an optimized implementation of an FIM algorithm on FPGA using SDAccel and Vivado HLS. Performance and power consumption are measured with various datasets. When compared to state-of-the-art solutions, this implementation offers up to 3.2x speedup over a 6-core CPU, and has a better energy efficiency as compared with a GPU. Our preliminary results on the new XCKU115 FPGA are even more promising: they demonstrate a comparable performance with a state-of-the-art HDL FPGA implementation and better performance compared to the GPU.**

Keywords—Frequent itemset mining; field-programmable gate array (FPGA); hardware acceleration; hardware description language (HDL); high-level synthesis (HLS)

I. INTRODUCTION

Frequent itemset mining (FIM) is a computationally-intensive data-mining technique that was originally developed for market basket analysis [1]. It aims to derive the rules underlying such phenomena as shopping behavior of customers in supermarkets or online shops, or relatedness of events. Specifically, it finds frequently-occurring subsets from a database of transactions. The frequency of a subset is measured by support ratio, which is the number of transactions containing the subset divided by the total number of transactions. Among the best-known FIM algorithms are Apriori [2], Eclat [3], and FP-growth [4]. Apriori iteratively performs three stages, namely candidate generation, support counting, and candidate pruning. It utilizes a breadth-first search to traverse the itemset candidate space. Eclat also uses a candidate generation strategy but uses a depth-first search to traverse the candidate space recursively. Eclat is more efficient in memory than Apriori but makes it hard to parallelize. FP-growth is another popular FIM algorithm introduced in [4]. By using a data structure called a Frequent-Pattern tree that contains all the information from the input database, FP-growth requires only two scans of the database. FP-growth generally has better performance than Apriori and Eclat, but its high memory requirement prevents its use for very large datasets.

Recently, a number of hardware-accelerated solutions for FIM algorithms have been developed [5], [6]. In [5], a state-of-the-art, to our knowledge, FPGA implementation was proposed to accelerate the Eclat algorithm on a four-FPGA board with a binary representation of itemsets. In [6], a GPU-accelerated algorithm, called Frontier Expansion (FE), was proposed. The FE algorithm uses both breadth-first search and depth-first search to provide more parallelism. The parallel paradigm is generalized by a producer-consumer model that makes the implementation applicable to a heterogeneous environment consisting of CPUs and GPUs. To the best of our knowledge, the FE algorithm [6] is the fastest parallel FIM implementation.

Over the past few years, FPGAs have proven their potential in variety of applications with high performance and energy efficiency compared to other computing platforms. This makes FPGAs compelling choices for datacenters, where energy efficiency and power provisioning are both critical factors. For example, Microsoft has designed a customized FPGA board Catapult and deployed it in its datacenters [7], which improved the ranking throughput of the Bing search engine by 2x. However, FPGA designs are often implemented in low-level HDLs such as Verilog and VHDL, which can be time-consuming, and requires a long learning curve on both programming and performance optimizations [8]. Recent advances in high-level synthesis (HLS) allow developers to specify computationally-intensive algorithms in conventional high-level languages such as C/C++ and OpenCL, [9], [10]. The HLS tools can automatically transform the algorithms to HDL implementations and compile them into FPGA hardware binaries. This process does not require extensive knowledge of FPGA hardware or memory interfaces, which reduces development time and cost. The Xilinx SDAccel development environment is built on the existing Vivado HLS capabilities with an optimized compiler that comprehends not only C/C++ but also OpenCL. Internally, it uses Vivado place-and-route engine. The environment currently focuses on x86 CPU-based systems with PCIe interfaces to FPGA-based add-in boards.

In this work, we adopt the Frontier Expansion framework and implement it using SDAccel targeting an Alpha Data ADM-PCIE-KU3 board equipped with a Kintex UltraScale XCKU060 FPGA. We choose to develop our kernel in C/C++ language to leverage Vivado HLS for efficient architectural optimizations. One advantage of this framework is that it can easily be extended further for a heterogeneous platform of FPGAs, GPUs and CPUs. In this paper, performance and power consumption are measured with various datasets. We compare the performance of the FPGA implementation over a range of minimum support values against the implementations in [6] on both multi-core CPU and GPU. We also present preliminary results using the Kintex UltraScale XCKU115

978-1-5090-4825-0/17/$31.00 ©2017 IEEE

FPGA, which has larger resources and higher memory bandwidth, and compare our implementation with the HDL FPGA implementation of the Eclat algorithm in [5].

II. Implementation of Frontier Expansion Algorithm

Details of the Frontier Expansion (FE) algorithm can be found in [6]. In this section, we provide a brief overview to help our discussion on its FPGA implementation. The FE algorithm uses a bitvector representation, where each item is represented by a binary sequence of transactions. Each bit in the sequence denotes a transaction and is set to one if the item is contained in the transaction. It is noted that the bitvector length is fixed for all items. With the bitvector representation, support counting can be performed using bitwise logical operations. The candidates are stored in a data structure called the "frontier stack," managed by the CPU. Each stack element contains one candidate and a reference to its bitvector in the FPGA's DRAM. The algorithm repeatedly expands the stack by generating new candidates from old candidates with the common prefix on the top of the stack, and deleting the infrequent candidates from the stack. Specifically, the new candidates are generated by intersecting the old candidates and the support counts of the new candidates are computed on the FPGA. In order to avoid the overhead of repeated memory allocations and deallocations during the expansion, the largest possible contiguous block of memory (dubbed memory pool) in the FPGA's DRAM is allocated for the largest possible datasets (i.e. maximal number of bitvectors). A CPU-based runtime manager is used to issue and revoke the free addresses in the stack.

A. Candidate Generation on CPU

The frontier stack is organized by clustering candidates into equivalent classes, which contains a set of candidates with the same size, sharing the common prefix. The initial contents of the stack are 1-itemsets, which are all in the same equivalence class. During the candidate generation (as shown in Fig. 1), equivalent classes are popped from the stack and their candidates are self-joined to generate new equivalent classes, containing new candidates. The process continues until the size of the new candidate list is larger than a predefined `threshold`. The supports of these new candidates are then counted by the FPGA kernel. Those new candidates that meet the minimum support threshold are pushed back into the stack. Note that the expansion procedure is repeated, which implies the FPGA kernel is called repeatedly, until the stack is empty. The larger the `threshold`, the more candidates generated, which leads to fewer number of FPGA kernel calls, but requires a larger memory space. Hence, the value of `threshold` is chosen as a tradeoff between performance and memory usage. In this work, we set `threshold` equal to 8192.

B. Support-Counting on FPGA

After new candidates are generated on the CPU, the addresses of the bitvectors of the candidates (old and new) are given to the FPGA support-counting kernel. The FPGA kernel (its pseudo-code is shown in Fig. 2) computes bitvector intersections and population counts (i.e. support values) for each new bitvector (i.e. new candidate). The kernel receives an

```
Algorithm: Frontier Expansion
Input: frontier_stack, threshold, min_sup, frequent_itemset
1   while frontier_stack is not empty
2       expansion_size = 0
3       candidate_list = ∅
4       while expansion_size < threshold
5           pop equivalent class prev_eqv from frontier_stack
6           new_eqv = expand(prev_eqv)
7           update candidate_list from new_eqv
8           expansion_size += size(new_eqv)
8       support_counting(candidate_list)
9       remove infrequent candidates from new_eqvs
10      add new_eqvs to frequent_itemset
11      push new_eqvs to frontier_stack
12  return
```

Fig. 1. Pseudo-code of the Frontier Expansion algorithm.

`threshold`×3 array. The array consists of three sub-arrays, namely `src_list1`, `src_list2`, and `dst_list`, which store the addresses referencing to two source bitvectors and one destination bitvector. The kernel first reads the source bitvector src_bitvec1 indexed by `src_list1`, and stores it in BRAM. Then, it executes the intersection and counting operations while loading src_bitvec2 indexed by `src_list2` at the same time. The intersection results is stored in BRAM and then written into the destination bitvector, dst_bitvec, referenced by `dst_list`. It should be noted that src_bitvec1, src_bitvec2, and dst_bitvec are all stored in a memory pool (i.e. "`base`" array in Fig. 2) in FPGA's DRAM as discussed before.

Several optimization techniques are applied in the FPGA kernel implementation, including:

(1) loop pipelining for the loop at line 8 in Fig. 2;

(2) data packing the `base` buffer for a wider bitwidth (512 bits) of the memory interface to increase the global memory bandwidth;

(3) partitioning `buf_src1` and `buf_dst` into 16 physical BRAMs instead of one single BRAM for each to improve the pipelining in line 8. Each buffer can sustain 16 concurrent transactions;

(4) partially unrolling the loop at line 8 by a factor of 16 which allows 16 samples to be processed in parallel;

(5) reducing the number of data transfers from DRAM to BRAM (line 6) by reusing bitvector in `buf_src1` (BRAM) as much as possible;

```
Algorithm: Support Counting
Input: src_list1, src_list2, dst_list, base, list_siz, bitvec_siz
1   bitvec_len = bitvec_siz/32;
2   uint buf_src1[bitvec_len];//BRAM
3   uint buf_dst [bitvec_len];//BRAM
4
5   for(j=0; j<list_siz; j++){
6       memcpy(buf_src1, base[src_list1], bitvec_siz);
7       popcnt_ = 0;
8       for(i=0; i<bitvec_len; i++){
9           tmp_var = buf_src1[i] & base[src_list2 + i];
10          buf_dst[i]= tmp_var;
11          popcnt +=population_count(tmp_var);
12      }
13      memcpy(base[dst_list], buf_dst, bitvec_siz);
14  }
```

Fig. 2. Pseudo-code of support counting on FPGA.

(6) reducing the number of data transfers from BRAM to DRAM (line 13) by only transferring new candidates if their supports are greater than the minimum support ratio.

C. Producer-Consumer Model

The support-counting kernel is synthesized on one compute unit (CU) (defined as the element in the FPGA device onto which the kernel is executed [10]). Each topmost branch of the search tree is processed sequentially. However, each branch can be expanded independently with its own allocated memory pool on DRAM. If multiple CUs are synthesized on the device, these expansions can be done in parallel, as long as there are enough resources on the FPGA device and on the DRAM. We use a producer-consumer model, [6], to scale the FE algorithm to a multi-CU architecture. After 1-itemsets are initialized in the frontier stack, the producer thread splits topmost branches (i.e. equivalent classes) and inserts them into a shared buffer. The consumer threads process one class at a time from that buffer. When the buffer is full, the producer stops until one class is consumed by a consumer thread. The producer thread finishes when all the equivalent classes are generated and inserted into the buffer. In our FPGA implementation, each consumer thread is assigned to a CU on the FPGA. This model can also be realized on a multi-CU multi-FPGA architecture.

III. Results

We compare the FPGA implementation with the single-core CPU (1CPU), the 6-core CPU (6CPU) and the GPU-accelerated (1GPU) implementations of FE. Because the FPGA only accelerates the support-counting operation, we show the performance results of both the counting operation (FPGA only) and the total computation (FPGA+CPU) in this section. The two following systems are utilized for experiments:

• CPU and GPU: the system is equipped with an Intel 6-Core i7-5820K (3.3GHz), 32GB DDR4 RAM, and an Nvidia Kepler K80 (12GB GDDR5 memory). The system uses CUDA 8.0.

• FPGA: the system is populated with an Intel 4-Core i7-4820K (3.7GHz), 32GB DDR3 RAM, and an ADM-PCIE-KU3 board featuring Kintex UltraScale XCKU060 FPGA and 16 GB of DDR3 memory. The system uses SDAccel 2016.2.

For all implementations, we compare the performance over a range of relative minimum support values. The relative minimum support number is defined as the ratio of minimum support number to the total number of transactions.

One commonly-used real-world dataset, Accidents, obtained from the Frequent Itemset Mining Dataset Repository [11], and one synthetic dataset, T80I20N0_3D2000K, obtained from the IBM Market-Basket Synthetic Data Generator are tested. The third dataset, T60I20N0_5D500K, is used for comparison with the Eclat FPGA implementation in [5]. The details of these datasets are shown in Table 1.

TABLE I. Real-World and Synthetic Datasets

Dataset	Trans#	Item#	Size(MB)
Accidents (real-world)	340183	468	34
T80I20N0_3D2000K (synthetic)	2M	300	516
T60I20N0_5D500K (synthetic)	500K	500	103

A. Impact of Optimization Techniques

It is important to study the impacts of individual optimization techniques (Section II.B) in our FPGA implementation, since the performance can be significantly improved with proper optimizations. In Fig. 3, we show speedups, as optimizations applied, with respect to the FPGA implementation with no optimization, dubbed Non-Opt. Note that since optimization (2) through (4) are related to throughput improvement, we group them together with optimization (1) into Opt1_to_4 in Fig. 3. The All-Opt in the figure denotes the performance when all optimizations are applied. As shown, through loop unrolling, we can allocate more resources to allow more concurrency in the FPGA kernel. Along with data packing and BRAM partitioning, we can achieve significant performance speedup (increased from 3x with Opt1 to ~30x with Opt1_to_4). When all optimizations are utilized, the performance speedups with respect to the Non-Opt reach 43.6x and 78.2x for Accidents and T80I20N0_3D2000K, respectively. Due to this significant improvement, we only present the results of FPGA implementations with all optimizations applied in the following sections.

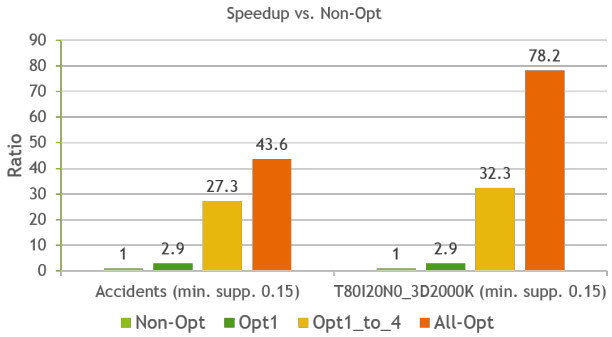


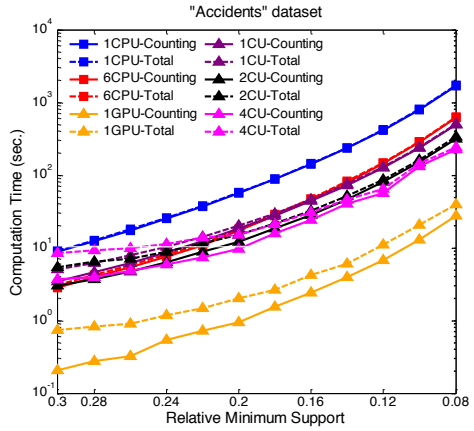
Fig. 3. Performance of optimization techniques on the FPGA.

B. Performance

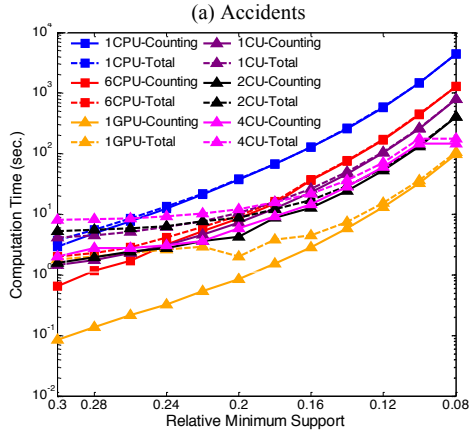
Fig. 4 shows the runtimes of the CPU implementations (1CPU and 6CPU), the GPU implementation and the FPGA implementations (using 1 CU, 2 CUs, and 4 CUs) on two datasets. It is observed that the runtimes of all implementations grow exponentially as minimum support number decreases. However, GPU and FPGA implementations show less runtime and slower growth of runtimes as minimum support number decreases. We also show speedup results of the FPGA and the GPU versus the 6-core CPU, in Fig. 5. For the sake of brevity, we do not show speedup results with respect to the 1-core CPU. However, speedup ratios can be easily calculated based on the runtimes in Fig. 4. The GPU implementation outperforms the FPGA implementations, even when multiple CUs are used. To explain this, we revisit some details of the GPU implementation in [6]. During the FE process, the CUDA kernel reads bitvectors from the global memory, performs computations, and saves the intermediate bitvectors to the global memory. Each bitvector intersection is computed by one block. Threads within a block collectively process intersections of the two bitvectors. In other words, each iteration in the FOR loop at line 5 in Fig. 2 is corresponding to a block and these massively parallel blocks can run concurrently performing load/store operations to GPU global memory. However, this is not the case in our FPGA implementations. In FPGA systems,

the load/store operations from/to global memory will compete for the on-board global memory bandwidth. Moreover, the FPGA's global memory system lacks a dedicated cache hierarchy like in the GPU, which causes the global memory transactions to be less efficient than that of the GPU. Thus, in the FPGA implementations, the FOR loop at line 5 in Fig. 2 is sequential. The parallelization is only applied to the inner loop at line 8 through pipelining and unrolling. Moreover, since source bitvectors and resulting bitvectors are stored in the same global memory space, which prevents the loop at line 8 from being fully pipelined, it costs extra memory transfers between DRAM and BRAM. These make the FPGA implementations slower than the GPU implementation. Hence, a GPU-like global memory system is desirable for future FPGAs.

Minimum support threshold is used by support counting to filter infrequent candidates from new candidate sets. This threshold affects not only the number of new itemsets but also the search depth of the algorithm. Our analysis demonstrates that FPGA implementations achieve better performance over CPU implementations for low minimum support values. It is apparent that lower support ratios result in more computation for support counting, and the FPGA kernel is parallelized better than CPU implementations for this purpose. As shown in Fig. 5, FPGA speedup ratios with respect to a 6-core CPU can be achieved up to 1.27 (1-CU) and 2.65 (4-CU) for Accidents and 1.63 (1-CU) and 3.16 (2-CU) for T80I20N0_3D2000K, at the relative minimum support of 0.08.

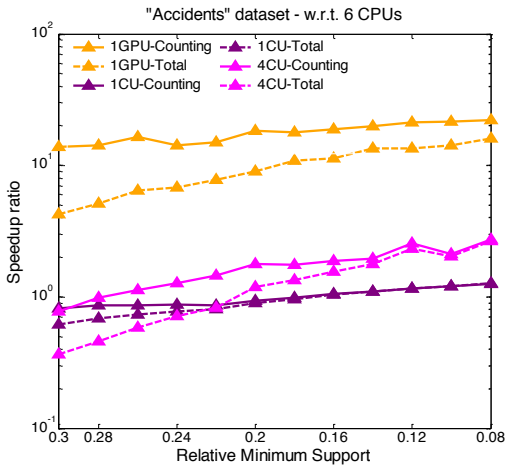


(a) Accidents

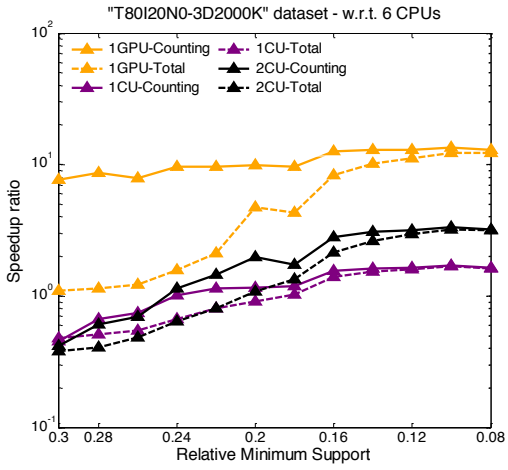


(b) T80I20N0_3D2000K

Fig. 4. Performance results of CPU, GPU and FPGA implementations.



(a) Accidents



(b) T80I20N0_3D2000K

Fig. 5. Speedups over 6-core CPU implementation.

For multi-CU FPGA implementation, each CU should be assigned to a separate bank for optimum memory access. More than one CU sharing a memory bank can lead to performance degradation, especially for large dataset, because of memory access contention. Note that the KU3 FPGA board has two DDR3 banks. Taking into account this limiting factor and the hardware resource of the FPGA device, we can synthesize up to four CUs on this board, where two CUs share one DDR3 bank. The 4-CU design gives the best performance for Accident dataset while it cannot deliver this for larger dataset. For T80I20N0_3D2000K dataset, the 2-CU design shows better performance than the 4-CU. Provisioning more memory banks in future generations of FPGA boards can solve the issue of performance degradation with higher number of CUs.

C. *Performance Projection*

In the multi-CU FPGA implementation, since the KU3 FPGA board only has two memory banks, multiple CUs might have to share a memory bank, which causes performance degradation. Hence, in order to investigate the performance scalability with increasing the number of CUs, we project the FPGA results into larger FPGAs with more memory banks. We assume up to 12 memory banks and 12 CUs can be used.

Based on the runtimes of each equivalent class expansion in the 1-CU design, the producer-consumer model is utilized to project runtimes onto larger number of CUs. Fig. 6 shows the projected speedups with respect to 1-CPU and 6-CPU implementations at the minimum support value of 0.1 for T80I20N0_3D2000K dataset. The dashed curves denote the projected results while the solid curves indicate the measured results of 1-CU, 2-CU and 4-CU designs. As shown in Fig. 6, we can achieve speedups up to 28.3x (w.r.t 1CPU) and 6.5x (w.r.t 6CPU) using 12 CUs for T80I20N0_3D2000K. Similarly, for Accidents, these numbers are 10.5x and 4.0x with respect to 1-core CPU and 6-core CPU, respectively. Although the multi-CU implementation shows relatively good projected speedups, it does not show good scalability, as can be seen in Fig. 6. The main reason is the imbalance of the search trees, which may have some extremely long branches. CUs responsible for these branches will dominate and execute longer than other CUs. Hence, it is essential to have a better distribution scheme that can equally assign the workload to the required CUs.

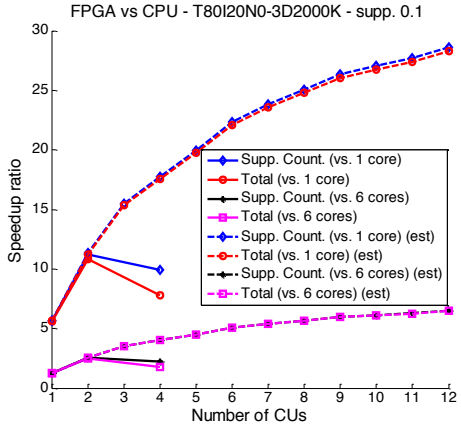


Fig. 6. Projected speedups w.r.t CPUs of the FPGA implementation.

D. *Power Analysis and Energy Efficiency*

Since SDAccel has not supported on-board power measurement, and the GPU and the FPGA are located on different systems, we decide to measure dynamic power. Power utilization of the entire system is recorded at one second intervals with a Watts up? PRO power meter. The procedure is as follows: (1) measuring the power usage of the idle system; (2) measuring the power usage of the system while it is performing the FE algorithm; (3) dynamic power is calculated as the difference between active power and idle power. The power results of the CPU, the GPU and the FPGA for Accidents and T80I20N0_3D2000K at the minimum support value of 0.1 are shown in Fig. 7.

To investigate the energy efficiency, we look at energy-delay products. Fig. 8 shows energy efficiency of the GPU and the FPGA against CPU for both datasets. Speedups are given in parentheses for convenient comparison. Actual measurements are represented in solid bars (6CPU, 1GPU, 1CU, 2CU, 4CU, respectively). FPGA implementations are faster and more energy efficient than 6-core CPU implementation while they are slower than GPU. However, when energy is considered, the gap between GPU and FPGA becomes narrower. For example,

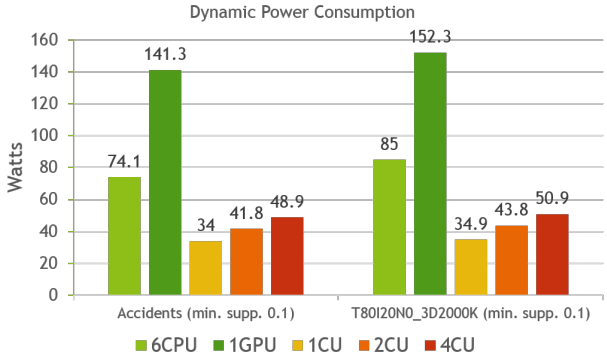


Fig. 7. Dynamic power consumption of CPU, GPU, and FPGA.

the energy efficiency of the 2-CU FPGA implementation on T80I20N0_3D2000K dataset is comparable to that of the GPU (6.2x vs. 6.9x).

One of the main factors that causes inferior performance of FPGA compared to GPU is data transfers between global memory and BRAM. Storing intermediate bitvectors on chip in a scratchpad can potentially improve FPGA performance. Since the actual BRAM capacity is limited, we use a scratchpad simulator to simulate the FPGA kernel operation with varying scratchpad sizes. Exploiting the SDAccel timeline trace collection feature, we measure the actual runtimes of individual loops in the FPGA kernel (i.e. loops in line 6, line 8, and line 13 in Fig. 2). Using these runtimes and the scratchpad simulator, we can estimate the FPGA kernel runtime based on the required scratchpad accesses. However, we observe that the use of scratchpad only shows improved performance with very large scratchpad size. Hence, we only include the results of ideal cases in which the scratchpad size is assumed to be very large to keep all necessary bitvectors in Fig. 8 (bars with *spm* in their names). As demonstrated in Fig. 8, the simulated FPGA implementation can achieve up to 4.5x (T80I20N0_3D2000K dataset) speedup over the 6-core CPU. In terms of energy efficiency, FPGA can obtain 7.5x more efficient than the 6-core CPU while the GPU achieves 6.9x.

These results show that despite the modest XCKU060 FPGA, with proper optimization, we can achieve better energy efficiency compared to GPUs and multiple times better compared to multi-core CPUs. With the latest FPGAs and more optimized SDAccel environment, we expect FPGAs to be more competitive against GPUs.

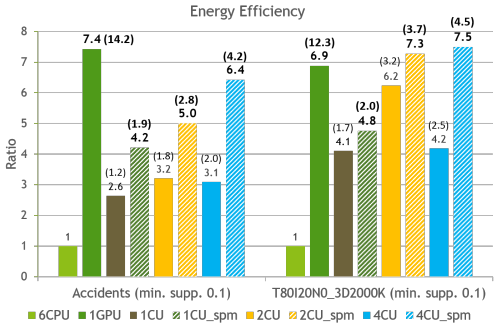


Fig. 8. Energy efficiency of GPU and FPGA, normalized to CPU results.

E. Preliminary Results with Nimbix Cloud

We also test our FPGA designs with the new Xilinx XIL-ACCEL-RD-KU115 board, equipped with a Kintex UltraScale XCKU115 FPGA and four DDR4 banks (16GB total). The KU115 board enables synthesizing kernels at 300MHz clock compared to 250MHz clock (KU3 board used in previous experiments). The KU115 board is accessed through Nimbix Cloud. The system that hosts the KU115 board is populated with an Intel Xeon 8-Core E5-2640 v3 (2.6GHz), 128GB DDR4 RAM, and SDAccel 2016.4. Since these experiments were carried out on the cloud, where is hard to acquire power utilization, we only present the performance results in this section.

In Fig. 9, we show the runtimes performed for the T80I20N0_3D2000K dataset (min. sup. 0.1) on the KU115 board along with the results of the CPU, the GPU and the FPGA on KU3 board. We also extend the projection to the 12-CU design for KU115 board use the projection approach in Section III.C. With KU115 board, we can achieve ~2x better speedup compared to KU3 board. The runtime of the 12-CU design for T80I20N0_3D2000K dataset is comparable to the runtime of GPU (35.4s vs. 36.3s), so we project this would also be significantly more energy efficient, since FPGAs typically use much less power than GPUs.

In Fig. 10, we also compare our FPGA implementation with the state-of-the-art FPGA implementation of the Eclat algorithm [5]. The design in [5] was developed using VHDL and tested on GiDEL PROCStar III PCIe board, containing 4 Altera Stratix III 260 FPGAs, with each FPGA connected to three DRAM memory banks. Up to 12 processing elements can run on the entire board at 200MHz (denoted as 12PE_ref in Fig. 10). To have a fair comparison, the experiment is conducted with the smaller dataset T60I20N0_5D500K (min. sup. 0.02). It is observed that the 4-CU design on KU115 board is close to 12PE_ref while the 12-CU design are ~2x better than 12PE_ref. With this small dataset, the 12-CU design on KU115 board even shows better performance than GPU.

IV. Conclusions

In this paper, we present an FPGA implementation of the FE algorithm using SDAccel and Vivado HLS. Performance and energy efficiency are compared with multi-core CPU, GPU and state-of-the-art FPGA implementations. An empirical model used to project the FPGA runtime on a larger FPGA with more memory banks is presented.

With the KU3 FPGA board, although we could not beat the GPU performance, we could achieve better energy efficiency than GPU, and could beat multi-core CPU in terms of performance and power efficiency. Our preliminary results on the new KU115 FPGA board demonstrate a comparable performance with the state-of-the-art FPGA implementation and even a better performance compared to GPU. It is highly promising that such performance and power efficiency can be achieved on FPGAs with a high-level language such as C/C++. An investigation of the FE algorithm on a heterogeneous platform will be targeted in our future work.

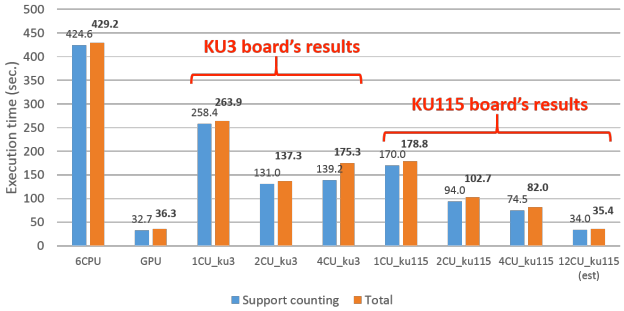


Fig. 9. Runtimes of CPU, GPU, FPGA (KU3) and FPGA (KU115) (T80I20N0_3D2000K, min. sup. 0.1).

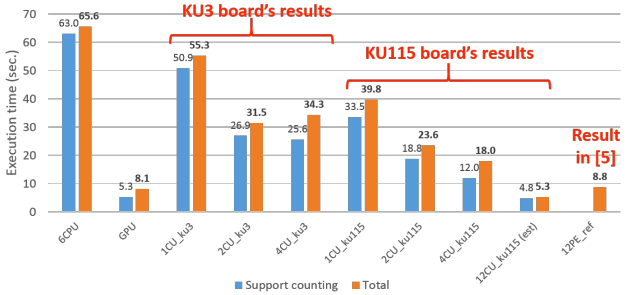


Fig. 10. Runtimes of CPU, GPU, FPGA (KU3) and FPGA (KU115) (T60I20N0_5D500K, min. sup. 0.02).

Acknowledgment

We thank Xilinx for providing us generous support on Xilinx FPGA boards, as well as remote access to SDAccel Development Environment in the Nimbix Cloud.

References

[1] K. Wang, Y. Qi, J. J. Fox, et al., "Association rule mining with the Micron Automata Processor," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, pp. 689-699, May 25-29, 2015.

[2] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. of 20th Intl. Conf. on VLDB*, pp. 487-499, 1994.

[3] M. J. Zaki, "Scalable algorithms for association mining," *IEEE Trans. on Knowl. and Data Eng.*, vol. 12, no. 3, pp. 372–390, 2000.

[4] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *Proc. of SIGMOD'00*, 2000.

[5] Y. Zhang, F. Zhang, Z. Jin, et al., "An FPGA-based accelerator for frequent itemset mining," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 6, no. 1, May 2013.

[6] F. Zhang, Y. Zhang, and J. D. Bakos, "Accelerating frequent itemset mining on graphics processing units," *J. Supercomput.*, vol. 66, no. 1, pp. 94–117, 2013.

[7] Y.-K. Choi, J. Cong, Z. Fang, et al., "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms," *53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, Austin, TX, 2016, pp. 1-6.

[8] Z. Wang, S. Zhang, B. He, and W. Zhang, "Melia: A MapReduce framework on OpenCL-based FPGAs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 12, pp. 3547-3560, December 2016.

[9] Altera Corporation. Altera SDK for OpenCL. [Online]. Available: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html

[10] Xilinc, Inc. SDAccel development environment. [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[11] "Frequent itemset mining dataset repository," http://fimi.ua.ac.be/data/.