Optimizing Data Transfers on Xilinx Versal AI Engines

Guilherme Prado Alves, Mehmet E. Belviranli*
{galves, belviranli}@mines.edu
Colorado School of Mines
Golden, CO, USA

Marco Minutoli, Antonino Tumeo
{marco.minutoli, antonino.tumeo}@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

1 Introduction

Graph data structure is a natural representation for problems in Discrete Math and in Computer Science. One algorithm that is of special interest is the Breadth-First Search (BFS), as demonstrated by its inclusion in the Graph500 benchmark [6]. One reason that BFS is of such interest is that it can be used as a basis on which to build other graph algorithms. However, scaling the BFS algorithm can be a challenge for a few reasons, namely: memory requirements can easily exceed the capacity of single nodes, so expensive external synchronization across nodes may be required. Further, graph algorithms exhibit low memory locality, both spatially and temporally [6].

High-Performance BFS is often bounded by a sparse memory access pattern. An alternative representation for graph algorithms is through Linear Algebra operations in various semirings [5]. Implementation of graph algorithms as a Linear Algebra operation allows for an easier mapping of the algorithm to vector units, as well as easier partitioning of memory for use across multiple nodes.

Modern computing architectures are rapidly changing. One example is Xilinx's Versal platform, which integrates a traditional Field-Programmable Gate Array (FPGA) platform, an array of very long instruction word (VLIW) single instruction multiple data (SIMD) cores named AI Engines, and a small number of ARM cores. The FPGA and AI Engines are connected by an array of AXI interfaces with aggregate bandwidth higher than DDR4 bandwidth [8]. One challenge in such architectures is in effectively routing data from memory through the available regions.

This work aims to collaboratively utilize the heterogeneous compute resources on the Versal platform to enable High-Performance, scalable, and energy-efficient BFS. In particular, we perform Sparse Matrix-Sparse Vector multiplication in the Boolean Semiring on the AI Engine tiles and use the re-configurable logic, i.e., FPGA, to orchestrate efficient data transfers.

2 Background

The implementation of BFS through Linear Algebra is examined by Kepner in [4]. Given a graph's Adjacency Matrix $\mathbf{A}$ and a current frontier $\vec{x}$, the expression $\mathbf{A}^T\vec{x}$ produces the next frontier when this multiplication is performed on the Boolean Semiring. Matrix Multiplication in the Boolean Semiring is a construct where (1) Boolean values replace real numbers in the matrix and vector, (2) scalar multiplication is replaced by the logical AND operator, and (3) scalar addition is replaced by the logical OR operator. This approach lends itself to parallelization and is suitable for FPGAs as demonstrated by Umuroglu et al. [7].

3 Architectural Considerations

An FPGA platform typically allows for very fine-grained operations to take place. Thus, this platform is ideal for defining a circuit of logical operations (ANDs, ORs, NOTs, etc.) on some input signals. In contrast, coarse grained architectures are better suited to perform operations on larger inputs, such as integers or floating-point numbers. This allows for the use of less area and lower power requirements, at the cost of less control over how these operations are implemented. The Xilinx Versal platform implements both architectures, as well as many channels of communication allowing a programmer to choose which architecture best suits each task required for an application. Tasks well suited to coarse-grained implementation can be implemented on the array of AI Engines, and can benefit from the available vector registers and instructions. At the same time, tasks requiring fine-grained operation or optimizations can be implemented in the Programmable Logic.

AI Engines work by streaming data in and out of small memory banks adjacent to the cores. That is, the kernel begins execution of its program once all of its inputs become available. The result of its calculation is then streamed out. Data can be streamed one value at a time, or in chunks called windows. Thus, windows of data are read from memory by the Programmable Logic and sent to the AI Engine array through one of the available AXI interfaces.

By pulling chunks of data from memory, it is possible to make better use of wide memory interfaces. At each iteration, several adjacent values are pulled and sent to the AI Engine array for processing. Thus, memory accesses are coalesced, better utilizing available resources. Further, the number of values sent through the AXI bus are known before-hand, therefore, these can be sent through a single packet, minimizing protocol overhead.

4 Optimizations

4.1 Input Sparsity

The Matrix-Vector multiplication approach used performs 'Pull' BFS [3]. Algorithm 1 describes this parallel implementation. At each frontier expansion, there are multiple units reading the graph from memory concurrently. Together, these check whether each node is visited in that expansion. Once the graph is fully read, each signals a new kernel that generates the next frontier (or stops execution) and signals the process to start again. Input is sparse because after each round of expansion, less of the graph must be read from memory.

The mechanics for this implementation were inspired by Attia et al. [2]. A CSC sparse matrix is represented using 3 arrays. The Values contains all non-zero values of the matrix. The Row Indices Array has one entry for each entry in the Values array, specifying which column that value belongs to. The Column Indices Array separates the above arrays into groups, where each group represents the entirety of the column. In this implementation, the Values array is removed, and every index in the Row Indices Array implicitly refers to a *True* in the matrix. In this work, the Row Indices Array is an array of 64-bit values, containing multiple values packed

*This work was performed when Guilherme Prado Alves was affiliated with PNNL.

```
input  : Adjacency Matrix in CSC Format, M
input  : Initial Frontier, f⃗
output : Vector o⃗, Each Element is Step Visited
while Frontier f⃗ is not empty do
    i ← 1
    forall Nodes n in Matrix M in parallel do
        if Node n has not yet been visited then
            if Any of Node n's parents are in frontier f⃗ then
                o⃗[n] ← i
            end
        end
    end
    Clear f⃗
    forall Indexes n in o⃗ do
        if n = i then
            Insert n into f⃗
        end
    end
    i ← i + 1
end
```

Algorithm 1: Parallel BFS

into each index, with a layout dependent on a 1-bit marker in the Least Significant Bit. A node that has not yet been visited holds the column's start index in its upper 32 bits, the length of that column in its next 31 bits, and a 0 in the least significant bit. A node that has already been visited holds the step number in its upper 63 bits, and a 1 in the least significant bit. This approach eliminates the need for an extra memory lookup to check if each node has already been visited. See Table 1 for a visual description of this format.

4.2 Window Splitting

Each AI Engine tile begins execution once all input windows become available. In this work, each window represents a portion of a node's adjacency list. If this list is smaller than the window size, then it must be padded out. If it is larger, then it must be split into multiple windows. This will generate the same partitioning on every frontier expansion. Therefore, instead of partitioning and padding columns as needed, this work is done by the Host program before data is transfered to the device.

Future work is necessary to determine optimal window size. A large window could result in wasted space if a large proportion of the graph's nodes are low-degree. However, a window that is too small would cause high-degree nodes to require many kernel calls, possibly causing unnecessary work to be done. Furthermore, it might be beneficial to have different window sizes for the column and the vector.

4.3 Short Circuiting

Given a sparse matrix and a sparse vector, multiplication in the Boolean Semiring becomes equivalent to stepping through both

Table 1: Data Format for Values in Row Indices Array

32 Bits	31 Bits	1 Bit
Column Start Index	Row Length	0
Visit Depth		1

Table 2: Runtimes for Synthetic Kronecker Graphs on Emulated Alveo U200

Number of Nodes	Emulated Time (ms)
430	0.7105
1200	4.4794
5100	18.5222

(sorted) arrays. Let R_i, V_i represent the values from each row or vector respectively at index i. These indicate the location of true values in the adjacency matrix and vector. If the value of both is equal, then the dot product of the row and this vector must be true, so there is no need to continue the operation on this row. Otherwise, the index of the smaller value should be increased, and the test retried. Because both are in increasing order, it is guaranteed that if the same value appears on both lists (indicating a positive conjunction), the algorithm will return true and skip over unnecessary work. Intuitively, this is because if a $True$ exists in an index of a particular column of the matrix, and in the same location in the vector, then that column is guaranteed to be visited during an expansion step.

5 Results

Graphs were obtained from MIT's Graph Challenge website [1] in .tsv format. A small program was written to convert from this format to a CSC format that could be used by the BFS kernels. Table 2 contains the emulated runtime of this approach. All data was gathered with 2 units performing BFS concurrently, though this could be increased as resources allow. This emulated time increased linearly with respect to the number of nodes in the graph, with $R^2 = 0.9977$.

The low computational density of the algorithm as described above means that this algorithm is bounded by memory access. Currently, data gathering is not possible, so preliminary results were gathered by emulating all kernels for an Alveo U200 platform. The timing data can be found in Table 2.

6 Conclusions and Future Work

This work presents BFS on the Versal Platform. This is accomplished by utilizing the heterogeneous resources made available in this platform. Memory is handled by kernels implemented in Fine-Grained Programmable Logic, while the Coarse-Grained AI Engines can efficiently handle Sparse Matrix-Vector multiplication. Preliminary results indicate that latency scales linearly as the number of nodes increases. The Versal Platform provides two architectures that excel at different tasks, and combining both could open the door for exciting future developments.

This work performs BFS in the 'Pull' configuration only. As demonstrated by Beamer et al. [3], Breadth-First Search can better utilize memory by first using a 'Push' configuration to expand the frontier by examining edges originating in frontier nodes, and switching to the more memory-intensive 'Pull' configuration (checking all nodes' incoming edges) once the frontier is sufficiently large. In a 'Pull'-only configuration, the first few expansions must stream the entire graph, but the frontier is still so small that most of this work will not produce new frontier nodes. Therefore, incorporating an initial 'Push' step could provide a performance benefit.

References

[1] [n.d.]. Data Sets | GraphChallenge. https://graphchallenge.mit.edu/data-sets. (Accessed on 08/22/2021).

[2] Osama G. Attia, Tyler Johnson, Kevin Townsend, Philip Jones, and Joseph Zambreno. 2014. CyGraph: A Reconfigurable Architecture for Parallel Breadth-First Search. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*. 228–235. https://doi.org/10.1109/IPDPSW.2014.30

[3] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–10.

[4] Jeremy Kepner. [n.d.]. *1. Graphs and Matrices*. 3–12. https://doi.org/10.1137/1.9780898719918.ch1 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898719918.ch1

[5] Jeremy Kepner and John Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, USA.

[6] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.

[7] Yaman Umuroglu, Donn Morrison, and Magnus Jahre. 2015. Hybrid breadth-first search on a single-chip FPGA-CPU heterogeneous platform. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.1109/FPL.2015.7293939

[8] Xilinx 2021. *Versal ACAP AI Engine Architecture Manual*. Xilinx.