HAGAR: Efficient Multi-context Graph Processors

Oskar Mencer, Zhining Huang(*), Lorenz Huelsbergen

Bell Labs, Murray Hill
NJ 07974, USA.
{mencer,lorenz}@research.bell-labs.com

(*) Department of Electrical Engineering
Princeton, NJ 08544, USA.
znhuang@ee.princeton.edu

Abstract. Graph algorithms, such as vertex reachability, transitive closure, and shortest path, are fundamental in many computing applications. We address the question of how to utilize the bit-level parallelism available in hardware, and specifically in FPGAs, to implement such graph algorithms for speedup relative to their software counterparts.

This paper generalizes the idea of a data-structure residing in reconfigurable hardware that, along with support logic and software in a microprocessor, accelerates a core algorithm. We give two examples of this idea. First, we draw parallels to content addressable memories. Second, we show how to extend the idea of mapping the adjacency matrix representation of a graph to a HArdware Graph ARray (HAGAR). We describe HAGAR implementations for graph reachability and shortest path. Reachability is a building block that can further be used to implement transitive closure, connected components, and other high-level graph algorithms. To handle large graphs where such an approach can excel relative to software, we develop a methodology, using FPGA internal small RAM blocks, to store and switch between multiple contexts of a regular architecture. The proposed circuits are implemented within the PAM-Blox module generation environment using Compaq's PamDC, and run on an FPGA accelerator card.

1. Introduction

Graph algorithms are fundamental to many applications in computing: routing layouts in networks and VLSI CAD, computer graphics, and scientific programs, are just a few examples. Software implementations of graph algorithms usually involve "walking" the graph by following chains of pointers, or by repeatedly indexing into a 2D array containing the graph's adjacency matrix. The limiting factor for the performance of such implementations is memory latency, which is recognized by the computer architecture community to be a major bottleneck that will become ever more severe as processor speeds increase faster than memory performance [WALL95]. The non-local and irregular memory accesses implied by pointer chasing further diminish the effectiveness of the caches introduced to mitigate this processor/memory speed imbalance.

Custom (or reconfigurable) computing suggests using an FPGA in conjunction with a general-purpose processor to accelerate performance limited applications. Most work in this area (e.g. [SONIC00]) focuses on algorithm's computational parts that, for example, require more arithmetic power than is available from a microprocessor. Such applications are often called *compute-bound*. A complementary set of applications, limited by memory performance, consists of *memory-bound* applications. It is primarily this latter set of applications that our data-structure+algorithm approach addresses.

We propose building circuits that incorporate a data structure—potentially quite complex—and support logic for algorithms specific to this data structure. Some simple data structures and algorithms such as LIFO/FIFO for example, are already common hardware building blocks. We believe that hardware structures for tasks typically relegated to software are feasible and that FPGAs coupled with microprocessors are an ideal vehicle for their realization. The algorithm itself may be distributed among the reconfigurable and microprocessor pieces or may reside almost completely in one or the other.

Typically, a single access to the circuit on the FPGA initiates an operation on the data structure within the FPGA. Implemented operations depend on the particular data structure under consideration. For graphs (which we consider in this paper), the operations insert or delete graph vertices or edges. We present accompanying graph algorithms for shortest unit[1] path and reachability. Using reachability, higher level algorithms such as transitive closure and connected components can also be implemented [HUELS00].

Before venturing into our implementation of graph algorithms, let us take a look at a well known hardware implementation of a non-trivial algorithm and data structure: the Content Addressable Memory (CAM)[FLYNN61]. The CAM extends the idea of SRAM memory to an associative memory. A CAM implicitly implements a search through all elements; i.e. a search in a CAM takes one clock cycle, only slightly longer than an access to conventional SRAM memory, and at a cost of four additional transistors within each memory cell. CAMs are used for applications such as processor caches[HP90], internet routers [PEI91], and simple compression algorithms[KOM93].

We take an idea that is conceptually similar to CAMs and apply it to graphs. Let us define as usual a directed graph $G(V,E)$ as a set of N vertices (or nodes) $V=\{v0,v1,v2,...,vN-1\}$, and a set of directed edges (or arcs) E between the vertices. We construct an adjacency matrix that fully describes a particular instance of such a graph. Matrix element $a(i,j)=1$ *iff* there is an edge from vertex v_i to v_j; otherwise $a(i,j)=0$. The matrix has $N=|V|$ rows and columns and thus the matrix has a size of $O(N^2)$.

This paper describes how efficient multi-context circuits can compute with graphs represented in their adjacency matrix form—each cell of the resulting two-dimensional hardware array corresponds to one entry in the adjacency matrix. This idea, called Dynamic Graph Processors (DGP), is first developed in Huelsbergen

[1] All edges have uniform weight of one.

[HUELS00]. Here we give three extensions to this prior work: (1) a streamlined implementation via tri-state logic that can increase density and therefore graph size, (2) a scheme for scaling the hardware arrays to larger graphs by splitting them into multiple contexts, and (3) an embedding of the graph accelerator circuits in a module generation environment.

An early approach to implementing graphs on FPGAs is part of the RAW project [RAW96]. In contrast to our solution, RAW's graphs are stored in the FPGA by creating a circuit that directly resembles the graph—edges are routes connected to logic representing vertices. As a consequence, changing nodes and edges requires completely rerouting and reconfiguring the entire FPGA circuit, which is extremely costly in time. In our approach, a circuit represents a graph of a particular size, while edges are state bits that can be changed quickly at runtime. Another prior effort eliminates graph specific circuits by synthesizing a technology dependent design "based on the specific domain [algorithm,target]" [DAND99]. Unlike this prior work, our approach admits graph modifications at runtime and performs them in a single write (clock cycle) to the FPGA.

2. Representing Graphs in Reconfigurable Hardware

The approach shown here adapts ideas from [HUELS00] (and originally implemented on the Xilinx XC6200) to the Xilinx XC4000 and Virtex families of FPGAs. In particular, we extend the earlier work to use internal tri-state buffers. This more compactly implements the desired functionality—a compact implementation can hold larger graphs and operate more quickly. Figure 1 shows the general idea of the hardware graph accelerator using tri-state buffers. The tri-state buffer is a controlled switch. The flip-flop connected to its control input stores an entry of the adjacency matrix and decides whether or not the row signal is forwarded to the column—i.e., if the row node has an edge to the column node.

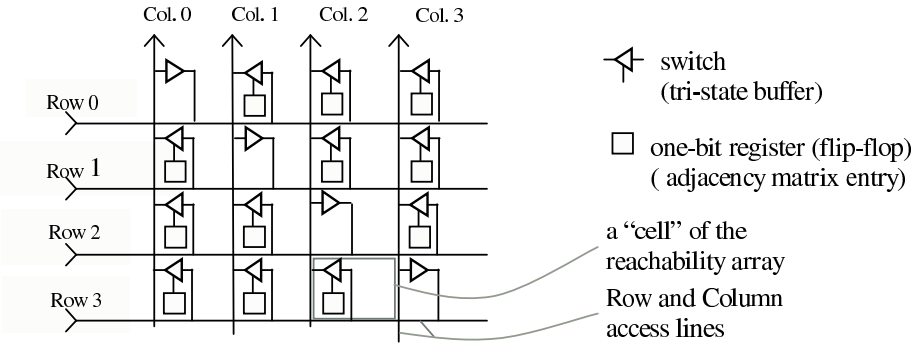

Fig. 1. Buses and tri-state switches implement the graph adjacency matrix in hardware.

It is apparent from the figure that an FPGA of a fixed size can only hold graphs up to a fixed number of nodes. Although the FPGA size places an upper bound on

the number of graph nodes, the number and location of the graph's edges is unbounded in this approach. Inserting an edge into the graph consists of writing a '1' to the register at position (source, destination) in the array. Notice that the cells along the diagonal differ from the other cells in the array. This difference in cell structure implements the propagation of a vertex's value onto its out edges and is crucial to how the array evaluates a graph algorithm such as reachability, described next.

3. Reachability for Small Graphs

As a starting point, we take the simple design illustrated in Figure 1 above and implement a graph with a small number of nodes so that the entire adjacency matrix fits into one FPGA configuration. Section 4 shows how to create multi-context versions with identical functionality but for larger graphs.

Reachability is a graph primitive from which many graph algorithms, such as transitive closure and connected components, may be constructed. It takes as input a source and a destination node and decides if there is a path from the source to the destination; in other words, if the destination node is reachable from the source node.

How do we use the structure proposed in the previous section to compute reachability? After the graph adjacency matrix is loaded into the registers in the cells, it suffices to drive a value of '1' onto the source node's row and to observe the row of the destination node. In fact, all rows corresponding to nodes that are reachable from the source node will be driven high ('1') within the time it takes the signal to propagate through the graph circuit. Thus, the performance of computing reachability is converted to the propagation delay of a combinational circuit.

4. Multi-context Arrays for Large Graphs

Since the size of the adjacency matrix grows quadratically with the maximum number of graph vertices, the size of the FPGA limits the absolute size of the graph. HAGARs address this problem by partitioning the graph into several pieces called contexts. During evaluation of the algorithm, HAGARs switch between these contexts in an orchestrated fashion to obtain the desired result. Ideally, context switches should be as fast as possible.

Instead of using FPGA configurations as contexts, we implement a multi-context circuit within the FPGA by using the FPGA's CLBs as context memories, albeit small ones. A four-input lookup table (LUT) has 16 bits of storage and can therefore store up to 16 contexts.

4.1 Multi-context reachability

Consider a HAGAR for N vertices $V=\{v0,v1,v2,...,vN-1\}$. M rows per context gives $C=N/M$ contexts (or partitions). Context 0 consists of the first M rows of the HAGAR, context 1 holds the second set of M rows, etc. Suppose the partitioned blocks have M rows and N columns. We store C contexts in FPGA lookup tables configured as distributed RAMs and switch between them to compute reachability.

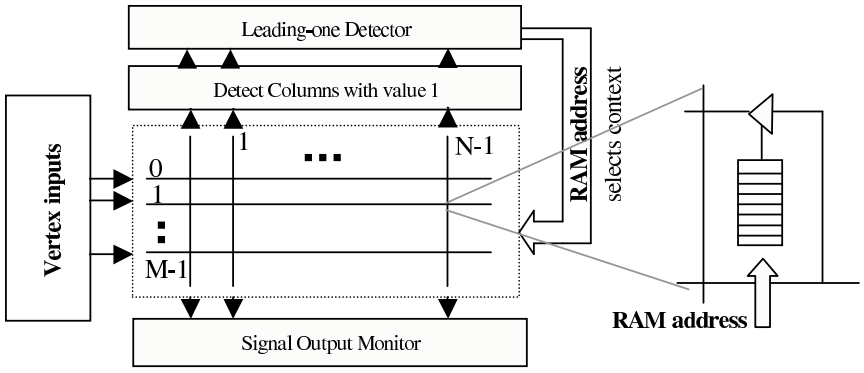

Fig. 2. Diagram of a multi-context HAGAR for reachability, including the multi-context cell architecture.

During a context switch, column values stay unchanged. At every column, a register at the top of the array retains the column value. Once a column receives a signal and becomes high, the column bus remains high during the entire evaluation, which in general consists of multiple context switches.

The algorithm works as follows: Assume we want to compute the reachability of vertex $v0$, and thus we switch to context 0, and drive a '1' on row 0. The '1' signal on row 0 now propagates to the connected columns. Suppose column M+2 and 2M+4 become '1'. At the next clock cycle, the current context will be swapped for another. Candidates for swapping-in are those who have received a '1' signal but have not yet been checked. Vertex $v0$ has signal '1' but has been checked. Vertices $v(M+2)$ and $v(2M+4)$ now become candidates. We simply pick up the first candidate (leading-one detector on top of the array), and switch to context 01. When there are no more unchecked candidates, the computation is done. By monitoring column outputs, the set of vertices reachable from $v0$ consists of the vertices with column values of '1'.

Figure 2 shows the general organization of the HAGAR circuit. Changing the distributed RAM addresses effects a context switch.

4.2 Multi-context shortest path

A HAGAR implementation of shortest path algorithm is slightly more complex than reachability. The shortest path algorithm also takes two input nodes, a source and a destination. From the set of paths from source to destination, a shortest such

path is the result of the algorithm; otherwise, the algorithm indicates that no path exists from source to destination.

In this section we explain how to find the shortest path between two vertices using a hardware array. To find the shortest path, we use a special partition of the adjacency matrix, with contexts being single rows (*M=1*) of the original square HAGAR circuit. Contexts still contain *N* columns where *N* is the number of vertices in the graph. There are therefore *C=N* contexts stored in the distributed RAMs, named context 0 to context (*N*-1). We start with the source vertex, say vertex *v0* and context 0. The computation ends when we reach the destination vertex, say vertex *v7*. If column 7 changes from '0' to '1', which means vertex *v7* is reached, the shortest path is available in the register file.

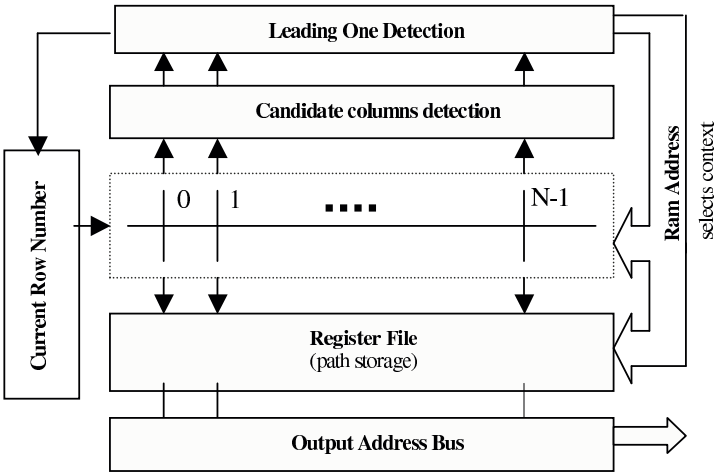

Fig. 3. HAGAR circuit for a multi-context implementation of shortest path.

The general organization of the HAGAR circuit for finding the shortest path is shown in Figure 3. The register file holds vertices in the traversed path. For each column *i* (where *i* is between *0* and *N-1*) the register file keeps the log(*N*) bits that identify the vertex that immediately precedes column *i* on the shortest paths. For example, if column 8 goes from '0' to '1' during a cycle, this means that vertex *v8* is reached. If the current context is 11, vertex *v8* is reached via vertex *v11*.

We keep a distance counter in the circuit to count how many steps we are away from the source vertex. At each step *k* there is a set of vertices that needs to be checked. This set of vertices contains the vertices that can be reached from the source vertex in a minimum of *k* steps. After all vertices have been checked, the distance counter is increased by one.

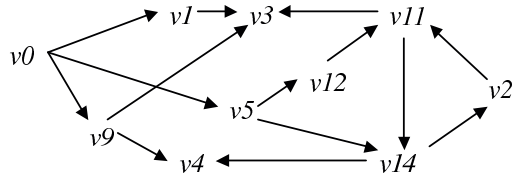

Fig. 4. Graph for shortest path example (*v0* → *v5* → *v12* → *v11*).

Consider for example the graph of Figure 4 where the vertex *v0* is taken as the source and has edges to *v1*, *v5*, and *v9*. The vertex set {*v1*,*v5*,*v9*} is therefore checked during step (distance) one. From *v1* we can reach *v3*; from *v5* we can reach *v12* and *v14*; from *v9* we can reach *v3* and *v4*. Vertices *v3*, *v4*, *v12*, and *v14* now form the set that needs to be checked in step two. This process continues until the destination vertex *v11* is reached, or no progress is made. The registers in each column record the context number as the column goes from '0' to '1'. Context numbers correspond to vertex numbers. In our example, the registers at columns 1, 5, and 9 record zero while the register at column 3 records a one. When the destination vertex is reached, we retrieve the shortest path from the register file as follows. The recorded values form a linked list. In our example with destination vertex *v11*, we first read out the value 12 from the register at column 11. At column 12 we read the value 5. At column 5 we read the value 0 (the source), and thus have a shortest path: *v0* → *v5*→*v12*→*v11*.

In case of a cycle in the graph, or multiple paths to the same node, the algorithm records the first time the node is reached. Any subsequent visit to the same node is ignored since it can not indicate a path shorter than the one through which the node was initially visited.

5. Implementation and Results

We implemented our HAGARs using the PAM-Blox II module generation environment[PBLOX02], built on top of the register transfer level (RTL) FPGA design library for C++, Compaq PamDC[PAM92]. The circuits were implemented on a Xilinx Virtex 300E FPGA (speedgrade –6) using Xilinx place-and-route tools (with effort level 4). The results below include the HAGAR and associated I/O circuitry for a PCI-based FPGA accelerator card.

Figure 5 shows the increase of cycle time for multi-context reachability. The performance of the HAGARs for reachability and shortest path—i.e. the worst case time to compute the reachability for a particular input is $O(|V|)$ while the worst case for shortest path is $O(|E|)$. The speedups are not obtained by improving the underlying complexity of the algorithm, but by reducing the time of their central computational primitive by a large factor [HUELS00].

We expect average performance to increase with increasing context size, especially with regard to shortest path computations. Average execution time depends strongly on the shape and size of the graph. Due to the larger contexts of reachability, average execution time for reachability should be much shorter than average execution time for shortest path. While the cycle time does not vary much with the number of contexts, the largest number of nodes (88 nodes on an XCV300E) can only be handled by *8* contexts, suggesting a tradeoff between context overhead and HAGAR size. We observe that the cycle time for shortest path is less sensitive to an increase in the number of nodes due to the single row context, and the bus-based architecture, as shown in Figure 6.

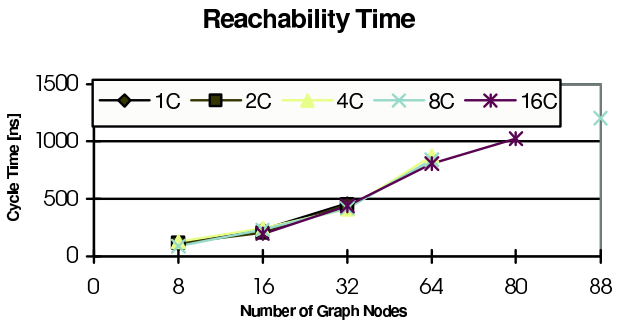

Fig. 5. Maximal delay or minimal clock cycle time, in [ns] for the multi-context reachability circuit, as a function of the number of graph nodes. Results show 1-16 contexts (1C-16C).

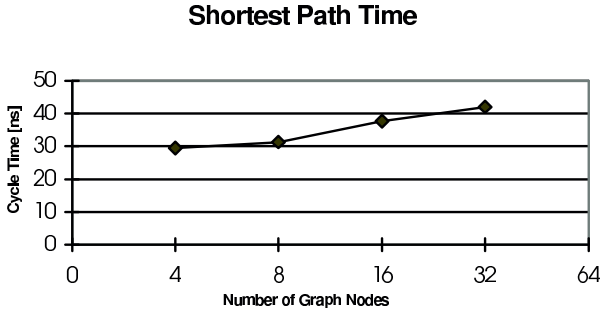

Fig. 6. Maximal delay, or minimal clock cycle time, in [ns] for the multi-context shortest path circuit, as a function of the number of graph nodes.

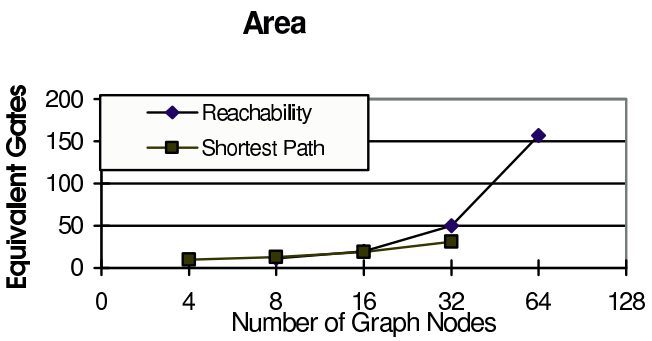

Fig. 7. Area in equivalent gates for the shortest path circuits, and reachability circuits with 4-context circuits.

Figure 7 shows the results for the area requirements of the proposed circuits for reachability and shortest path. The results for shortest path are limited to 32 nodes because the current implementation uses only a single small (32 bit) RAM for contexts. Larger graphs could be implemented by using multiple RAMs.

6. Conclusions and Future Work

This paper describes multi-context FPGA solutions for graph reachability and shortest path algorithms. It is possible to extend these ideas to other graph algorithms including transitive closure and connected components as described by [HUELS00], as well as to graph coloring, minimum spanning trees, etc. Implementation details of such additional algorithms are future work.

An additional improvement over the proposed method could be obtained by using multiple small RAMs, or larger block RAMs, for keeping contexts and/or storing temporary results. Such block RAMs are available in most current FPGAs. Furthermore, a finer balance between the usage of tri-state buffers and lookup tables could further improve the efficiency of the resulting circuits. However, even without these improvements we expect our HAGAR generators to scale to hundreds of nodes on the largest currently available FPGAs and to achieve speedups of orders of magnitude over general purpose microprocessor implementations.

Our work has the potential to help with one of the main problems in computer architecture: the growing gap between processor speed and time to access memory. Instead of software data structures constructed from pointers residing in a microprocessor's memory, complex data-structure+algorithm combinations can, as we have shown, be implemented very efficiently as HAGAR FPGA circuits. Thus, HAGARs have the potential to mitigate the memory bottleneck of certain algorithms and applications by representing the central data structure as a reconfigurable circuit.

Acknowledgements

Thanks to Rae McLellan, Rob Pike, Miron Abramovici, and Wayne Luk for helpful discussions and comments on this work.

References

[WALL95] W.A. Wulf, S.A. McKee, "Hitting the Memory Wall: Implications of the Obvious," Computer Architecture News, 23(1):20-24, March 1995.

[SONIC00] S.D. Haynes, J. Stone, P.Y.K. Cheung, W. Luk, "Video Image Processing with the Sonic Architecture," IEEE Computer, April 2000.

[FLYNN61] M. Flynn, "Operations in an Associative Memory," PhD Thesis, EE Dept., Purdue Univ., June 1961.

[HP90] J. Hennessy, D. Patterson, "Computer Architecture: A Quantitative Approach," Morgan Kaufmann, 1990.

[PEI91] T.B. Pei, C. Zukowski, "Routing Tables: Tries and CAMs," Proc. Infocom 1991.

[KOM93] E. Komoto, T. Homma, T. Nakamura, "A High-Speed and Compact-Size JPEG Huffman Decoder using CAM," Symposium on VLSI Circuits, Kyoto, Digest of Technical Papers ch. 61 v pp. 37-X3, 1993.

[HUELS00] L. Huelsbergen, "A Representation for Dynamic Graphs in Reconfigurable Hardware and its Application to Fundamental Graph Algorithms," Proc. ACM Int. Symposium on Field Programmable Gate Arrays (FPGA 2000), Monterey, Feb. 2000

[RAW96] J. Babb, M. Frank, A. Agarwal, "Solving graph problems with dynamic computation structures," High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic, Proc. SPIE 2914, 1996.

[DAND99] A. Dandalis, A. Mei, V.K. Prasanna, "Domain Specific Mapping for Solving Graph Problems on Reconfigurable Devices," 6th IEEE Reconfigurable Architectures Workshop (RAW), April 1999.

[PBLOX02] O. Mencer, "PAM-Blox II: Design and Evaluation of C++ Module Generation for Computing with FPGAs," IEEE Symposium on Field Programmable Custom Computing Machines, FCCM, Napa Valley, CA, 2002.

[PAM92] P. Bertin, D. Roncin, J. Vuillemin, "Programmable Active Memories: A Performance Assessment," ACM FPGA Conference, Monterey, Feb. 1992.