LA-UR- 07-0085

Approved for public release; distribution is unlimited.

Title: On the Acceleration of Shortest Path Calculations in Transportation Networks

Author(s): Zachary K. Baker
Maya B. Gokhale

Intended for: IEEE Symposium on Field-Programmable Custom Computing Machines

On the Acceleration of Shortest Path Calculations in Transportation Networks[1]

Zachary K. Baker and Maya Gokhale
Los Alamos National Lab
Los Alamos, NM 87545
Email: {zbaker,maya}@lanl.gov

Abstract

Shortest path algorithms are a key element of many graph problems. They are used in such applications as online direction finding and navigation, as well as modeling of traffic for large scale simulations of major metropolitan areas. As the shortest path algorithms are an execution bottleneck, it is beneficial to move their exectution to parallel hardware such as Field-Programmable Gate Arrays (FPGAs).

Hardware implementation is accomplished through the use of a small A* core replicated on the order of 20 times on an FPGA device. The objective is to maximize the use of on-board random-access memory bandwidth through the use of multi-threaded latency tolerance. Each shortest path core is responsible for one shortest path calculation, and when it is finished it outputs its result and requests the next source from a queue.

One of the innovations of this approach is the use of a small bubble sort core to produce the *extract-min* function. While bubble sort is not usually considered an appropriate algorithm for any non-trivial usage, it is appropriate in this case as it can produce a single minimum out of the list in $O(n)$ cycles, where n is the number of elements in the vertex list. The cost of this *min* operation does not impact the running time of the architecture, because the queue depth for fetching the next set of edges from memory is roughly equivalent to the number of cores in the system.

Additionally, this work provides a collection of simulation results that model the behavior of the node queue in hardware. The results show that a hardware queue, implementing a small bubble-type minimum function, need only be on the order of 16 elements to provide both correct and optimal paths.

Because the graph database size is measured in the hundreds of megabytes, the Cray SRAM memory is insufficient. In addition to the A* cores, we have developed a memory management system allowing round-robin servicing of the nodes as well as virtual memory managed over the Hypertransport bus.

With support for a DRAM graph store with SRAM-based caching on the FPGA, the system provides a speedup of roughly 8.9x over the CPU-based implementatation.

1 Introduction

Transportation infrastructure is a highly complex system, where hundreds of thousands of automobiles traverse vast road networks. The ability to understand and predict responses of the network to perturbations of the traffic flow and damage to the roads is of great value to planners. Exploring the effects of various changes to the infrastructure provides understanding of what to expect when similar changes happen in reality.

Part of the TranSIMs data is based on long-form census reports. These reports provide detailed information about the behaviors of large segments of a population. Given data on where a person lives, works, goes after school, etc., we can know where in the city grid they will be at any given time. As they have to get from one place to another, we also know roughly what routes they will travel. Thus, we can model the transportation network at any given point of the day, given certain assumptions, namely, the choice of routing.

The current routing algorithm we are using is a basic shortest path approach. This does not take into account loading on the freeways, but is less data intensive and easier to calculate in parallel. However, it is sufficient to model many types of travel plans. Some of the questions that can be answered are:

- Given a bridge closure, what will be the effect on the mid-day traffic?
- Given necessary freeway repairs, what is the best time of the day to divert traffic onto side roads?
- Given a natural disaster, what is the best way to evacuate a city?

Road and epidimological graph data is highly sparse, providing a challenging problem to hardware designers. Sparse graphs have very little non-zero data, allowing for highly efficient compressed representations. However, determining shortest paths from a sparse graph is difficult because of the irregular data access patterns of algorithms and the tendency of some solution techniques to fill (desparsify) the connection matrix.

The sparseness of the graphs in question ranges from an average connectivity of 2.75 in 560,000 nodes for the Los Angeles road data to 1500 in 1.6 million for the Portland human network.

The transportation network can be modeled at any given point of the day with certain assumptions, namely, the choice of routing algorithm.

The use of a parallel single-source shortest path approach to solving the all-pairs or many-pairs shortest path problem can leverage a sparse representation of a graph. Because each shortest path problem is only concerned with one set of edges, it is does not damage the sparsity of the graph during the solution. For completeness' sake, we remind the reader of the A* strategy [2]:

```
while Q is not an empty set
   u := Extract-Min(Q)
   S := S union {u}
   for each edge (u,v) outgoing from u
      if w(u,v) < d[v] --unvisited
        d[v] :=  w(u,v)
        previous[v] := u
```

In this formulation, $w(u, v)$ is the weight of the edge from u to v. In the A* algorithm, a "heuristic estimation" is used instead of the weight of the edge. In this case, $w(u, v)$ is usually the cost from u to the objective. In our case, the objective could be the closest exit point from the hierarchical block (be it a neighborhood or city). This can be estimated using a manhattan or euclidian distance. The interest in using A* over Dijkstra's is that it can utilize the knowledge of the graph we already have – that is, latitude and longitude, to predict it's distance from an objective point. In a general graph $G(v, e, w)$ this sort of short-cutting is impossible, but we have a map graph $(G(v1, e, w, lat, long))$, which can speed things up. The difference in implementation is trivial.

In order for this approach to work, several assumptions are required:

total number of paths > 1000
extract-min should require $\approx$
*#units * time for memory fetch*

The cost of *extract-min*, in cycles, is roughly the number of paths in the set Q of possible paths. This number can increase in an exponential manner (depending on the graph) during the execution of the shortest path.

Equivalently, *#units* $\approx$ *extract-min / time for memory fetch*, meaning that if the number of units able to fit on the device is too small, the memory bandwidth will not be used to its maximum capacity. The approximate value of these terms will be determined by simulation (below).

2 Related Work

Some other work has been done in exploring some of the hardware components used in this work. One, [3], deals with a fast heap implementation, and an implementation of Dijkstra's that uses a full hardware priority encoder [4]. In both situations, the priority encoder requires a large percentage of the slices available. This prevents the designers from using a larger number of units to process shortest paths in parallel. Because we are feasibly prior-

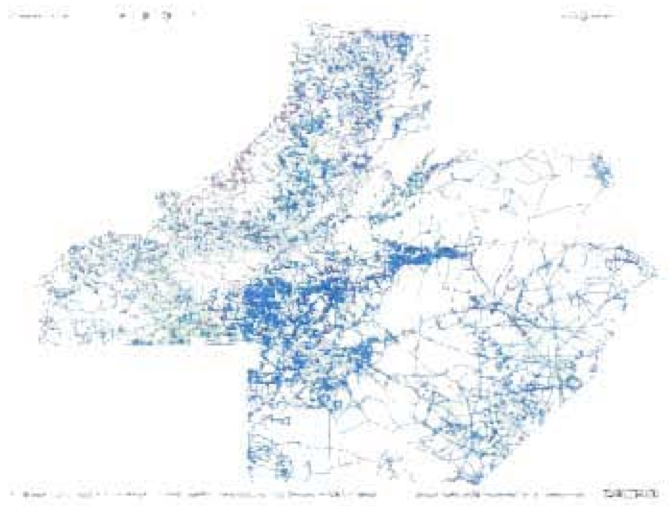


Figure 1: Example routing for the Los Angeles Graph

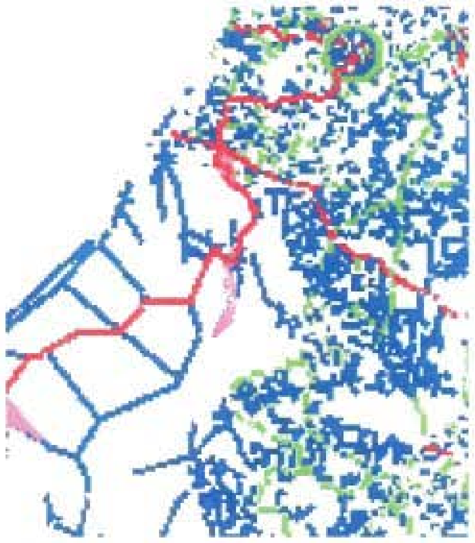


Figure 2: Zoomed in section showing where queue is required after greedy selection of routes causes A* algorithm to get stuck in a dead-end

itizing thousands of distance measures, it is infeasible to implement this in an $O(1)$ full-parallel hardware solution. By spreading the *extract-min* operation over $O(n)$ cycles with $O(1)$ hardware, we take advantage of the round-robin memory scheduler's request latency while simultaneously fitting far more hardware on the device.

The Floyd-Warshall architectures, such as [1], are only appropriate for dense matrices. Because the all-pairs problem demands a solution from every node to every other node, the benefits of the sparse matrix are lost as the algorithm runs. The dynamic programming solution fills in the entire matrix, as follows:

$$\forall k, i, j \quad D^{k+1}_{(i,j)} = min\{D^k_{(i,j)}, D^k_{(i,k)} + D^k_{(k,j)}\}$$

Because of the matrix-filling property of Floyd-Warshall's, it is pointless to use a sparse-matrix data structure. While the matrix starts out sparse, it becomes completely filled, unless the graph has specific properties (namely, a set of disjoint connected subgraphs would remain sparse, but it would be better to handle them as independent closures). The sparse structure has high overheads in terms of random-access to elements. Thus, the combination of an All-pairs solution and a sparse matrix is not ideal. The parallel single-source problem, run over all nodes (or some subset thereof) seems to be able to leverage the sparse graph more effectively.

3 Introduction to Our Approach

Dermining the next edge to explore is a key element of the A* algorithm, and the running time of the algorithm is largely determined by the data structure used to provide this information. Software implementations use a Fibonacci Heap to provide $O(log(n))$ average performance. However, this is an amortized performance achieved through a complicated data structure that would be difficult to implement well in hardware. There have been implementations of priority queues [3] but they are of a size that would require far too many device resources.

The Fibonacci heap is the key to an efficient software implementation. However, based on our simulations it seemed that the Fibonacci heap might be providing more that is needed, at a cost that is undesirable. The main observation is that the heap can grow without bound during execution of the A* algorithm. Edges are potentially added whenever a new node is explored, but queue elements are only removed when they are explored. This leads to a queue that can grow into the thousands of elements for a large graph. However, after simulation, we noticed that the size of the queue – within limits – does not affect the correctness of results.

For any route, given a start, destination, and graph of the road network, there is a certain size of queue that is required. The purpose of the queue, going back to the A* algorithm, is to provide some ability to backtrack. The A* algorithm repeatedly performs the extract-min() operation on the heap, allowing the best greedy route to be followed. This is fine for a regular grid, but in a real city, with complications such as housing developments, rivers, and freeways, the best greedy route is often not able to be completed. For instance, see Figure 2, which is a closeup of the example route shown in Figure 1. The best greedy route leads the algorithm into a dead-end. The heap is then consulted for the next best option. This may mean that all of the nodes leading into the dead-end will be explored. Because the algorithm only knows about the nodes in the queue, it cannot backtrack if the queue is not big enough. A subdivision, for instance, with one entry point but many internal streets, can cause the algorithm to not complete. Our tests of the Los Angeles network, over 150 randomly selected start/destination pairs, show that a queue size of 16 elements is sufficient for most cases. Figure 3 shows the number of completions and accuracy versus an infinite buffer as the queue size is reduced. Even while the accuracy is of the returned route is reduced, the algorithm still completes the majority of the cases until the queue size is reduced to 8, which is the equivalent of 3-4 intersections worth of streets.

These simulation tests have led us to a novel modification to the A* algorithm that makes it ideal for hardware implementation. Rather than an unnecesarily complex hardware Fibonacci heap implementation, we have developed a simple bubble sort core that provides one extract-min() operation every k cycles, where k is the depth of the queue. Bubble sort would not normally be an efficient method for providing min-heap functionality. The average behavior is worse than a randomized sorting method, like quick-sort. However, the hardware resources required to implement it are very low, consisting of a single register and comparator.

The algorithm only requires one minimum element required per exploration time step. Another option would be to just remove the minimum element from the queue, which would not require any write-back to the queue. While this would be somewhat simpler, as the "bubbling" action does require a write-back port on the memory, it would cause gaps in the queue where elements have been removed. Additionally, the bubbling action has the additional benefit of pushing large elements toward the tail of the queue. As large elements are less likely to be used, there is little problem if they are overwritten as the queue loops.

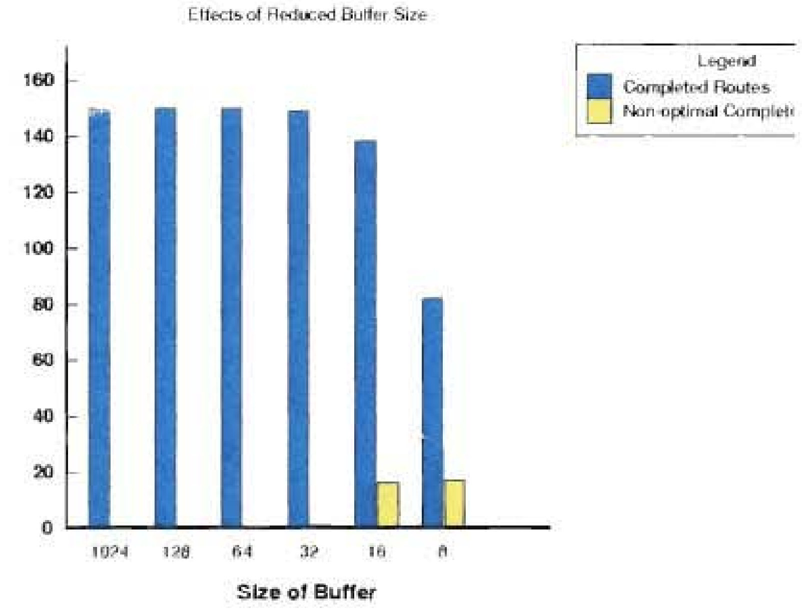
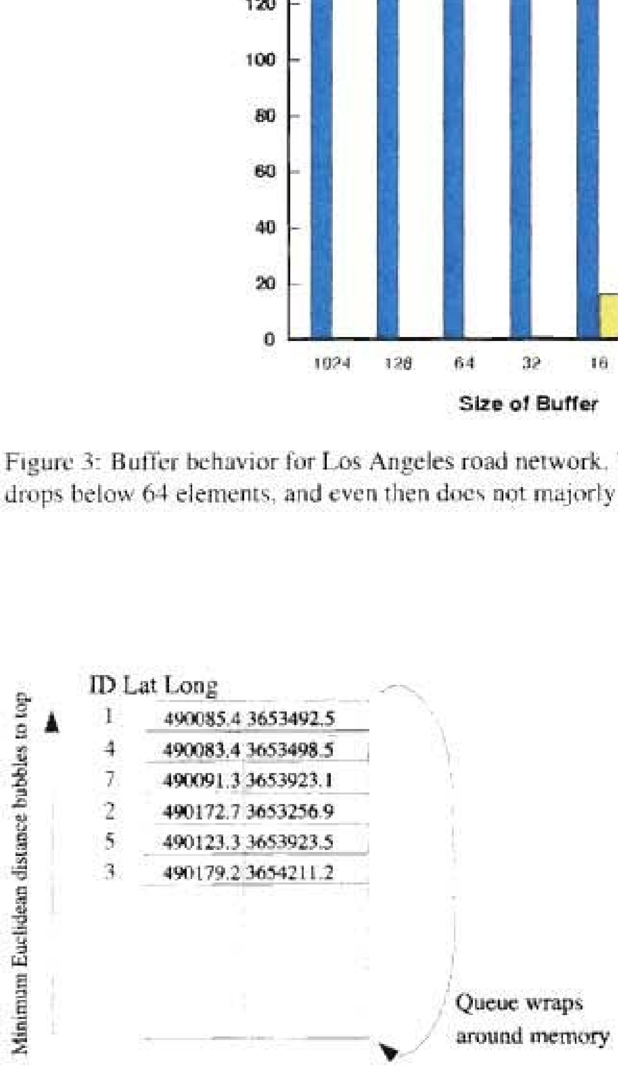


Figure 3: Buffer behavior for Los Angeles road network. The size of the queue does not affect results until queue size drops below 64 elements, and even then does not majorly affect results until the size falls to 8 elements.


Minimum Euclidean distance bubbles to top
Queue wraps around memory

ID	Lat	Long
1	490085.4	3653492.5
4	490083.4	3653498.5
7	490091.3	3653923.1
2	490172.7	3653256.9
5	490123.3	3653923.5
3	490179.2	3654211.2

Figure 4: Small, area-efficient bubble sort core provides sorted entries (heap) at low cost

3.1 Simulation Results

In tests of the Los Angeles dataset using A*, the maximum heap size required was 4,926 over a set of randomly selected origin/destination pairs. With a 20 bit street identifier (there are 560,000 some street segments in Los Angeles. There may be lots of "slip" roads that can be removed via optimization) and a 16 bit distance, we will require about 10 block RAMs.

This results means we can fit between at least 20 shortest path units on a chip, depending on the device.

4 Analysis

The objective of this work is to maximize the possible throughput ($\frac{\#pathscompleted}{time}$) by targeting a large number of small shortest-path units at the highest frequency, thus

$$\frac{\#pathscompleted}{time} = \frac{freq * \#units}{\#cyclesperpath}$$

The objective is for this to maximize the usage of the available random access bandwidth as well.

$$\frac{\#pathscompleted}{time} = bandwidth(lookups/sec) * \#units * lookups/path$$

In processing the Los Angeles street graph, we have found that the average connectivity is 2.75 links over the 560,000 individual street end points. Thus, when running the shortest path algorithm, each lookup should retrieve 2.75 nodes on average. The total number of lookups is related to the diameter of the graph and is highly dependent on the particular pair of endpoints selected.

Because Dijkstra's and A* are only concerned with the new nodes, and the structure of the graph determines how new nodes present themselves, we must simulate to develop an understanding of the memory access patterns of the system. For instance, a tree that is explored using Dijkstra's from the root will always present new nodes. A lattice (as in a regularly laid-out road network) is easily predictable in terms of the A* traversal. At each node, the choice of the next node to explore is generally the edge that leads in the direction of the destination. However, few cities are that simple. Freeways, bridges, rivers, and undeveloped areas can make route finding, and the data access patterns associated with them, more complex.

5 Architecture

The efficient bubble sorting technique allows for more units per device. The delay of the min-bubble operation is mitigated through the same multi-threading technique that allows the SRAM (and, occasionally, the virtual memory manager) latency to be ignored. By having several dozen A* units working in parallel, an SRAM request for one unit is handled while another unit is busy sorting its queue. This allows the SRAM bandwidth to be maximixed – the round-robin memory controller can provide a new data request on every SRAM data cycle, because the A* units immediately sort their queue as soon as a data request is completed and then wait for their next access graph memory.

The Los Angeles road graph has roughly 65,000 edges. In our data structure representation, this occupies about 48 MB. Thus, the 16 MB QDRII SRAM banks on the XD-1 will not be able to hold the entire graph at once. Whenever the SRAM does not have the particular graph block required, it starts the software-controlled virtual memory system. This is illustrated in Figure 5.

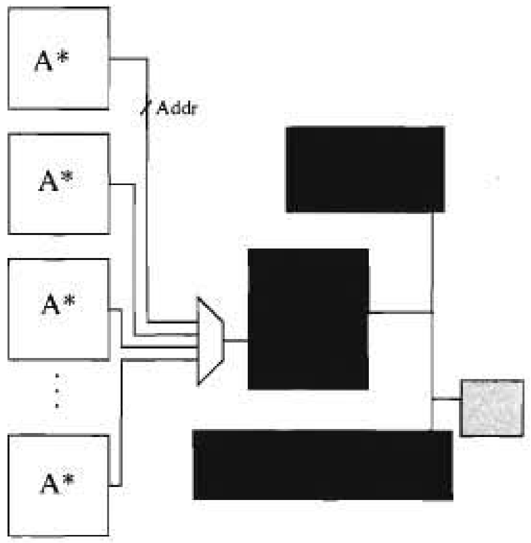


Figure 5: Virtual memory controller

Because the XD-1 is much faster at software-controlled block DRAM transfers to the SRAM than FPGA-initiated DRAM requests, we control the virtual memory system using the CPU. The round-robin memory controller switches to a new A* unit and determines if there is an outstanding request. If a memory request is ready, the memory controller looks at the high-order bits and compares it against a set of CAM entries. This is accomplished in one cycle. If the data is already in memory, the CAM returns the appropriate offset into the SRAM banks and puts the request on the memory address bus. Because the SRAM has a guaranteed latency, the memory controller can signal to the A* unit that it can expect its data on the bus after 20 cycles. Because the latency is guaranteed, the memory controller can go on to process the next unit.

If the data block is not already in SRAM memory, the memory controller switches into the virtual memory request mode. Because there is only one memory bus to the SRAM, the A* units can continue sorting, but outstanding requests will block until the requested data is paged in from DRAM. The controller writes a register detailing that it requires memory service and the block required.

Number of bits	20	20	10	5
Field	Lat	Long	Node Addr	Num of Edges

Table 1: SRAM word layout

The CPU then initiates a DMA transfer from DRAM to the SRAM and tells the memory controller which cache block was written.

As in all systems, this is an expensive operation compared to normal SRAM-based memory fetches. Fortunately, the graph data is laid out in blocks such that there is temporal and spatial locality in the data accesses. This makes the cache behavior more efficient than if the data was laid out randomly in memory.

The Cray XD-1 has a 16 MB of SRAM available to the FPGA. We address this as a unified single bank with many blocks addressed from a single memory manager.

Each 64-bit word coming out of the QDRII SRAM is laid out as illustrated in Table 1:

Each memory word is contains enough information to provide the data inputs to the queue. Latitude and longitude is necessary to provide the queue's sorting ability. The address provides the location in memory for the node in question. At that address, the following memory words are the edges connected to the node.

Table 2 illustrates the effect of various SRAM block sizes on the virtual memory manager. Because the XD-1 only has 16 MB of 64 bit words, there are not a lot of options for the block sizes. The V2Pro 60 device can support about 24 A* processing elements. As the memory manager services the A* units in a round-robin fashion, there must be at least 24 blocks. We tested 32, 64, 128, and 256 blocks, corresponding to block sizes of 64, 32, 16, and 8 words. For each size, we tested two replacement policies, a round robin approach that is very easy to implement, and a last-used policy. Because the A* units move through a block of graph data in several cycles, the round-robin unit is workable. The last-used approach is a more traditional approach; if the block of data isn't currently being used by any of the A* units, it is a sensible block to replace.

After experimentation with a set of random origin/destination pairs, we found that the last-used replacement policy with 32 blocks is the most effective methodology.

	Number physical blocks			
Replacement Policy	32	64	128	256
Round robin	7046	6830	7902	10359
Last used	5875	6030	7404	10157

Table 2: Total number of SRAM memory loads for various sizes of pages

6 Performance Results

Using the Cray XD-1 with a Virtex 2-Pro 50 -7, we placed and routed the design with 24 A* units at 130 MHz. This clock rate is less due to the design of the dot product units and more to the SRAM interfaces and arrangement of the pin constraints for the FPGA board.

Table 4 contains the performance results for a CPU implementation, the SRAM-only implementation, and the DRAM-based SRAM cache implementation. The results are based on a small set of randomly selected origin/destination pairs, with roughly 21,600 accesses to the graph memory. The SRAM-based implementation are not valid for real use, as the small QDR memories do not hold entire graphs, and the load time from the disk and DRAM are not considered. Thus, the speedup of 1500 is largely theoretical.

The DRAM-based results are more reflective of reality. This result is the full system implementation, including the memory manager that caches graph memory to the SRAM. The load time of the SRAM blocks is the main bottleneck for this implementation, and largely determines the running time and overall performance. The speedup of 8.9x over the CPU is lower than we had expected at the beginning of the project, but the data access costs were so high, even with the addition of the caching SRAM design, that the overall system performance was severely impacted.

While the architecture could easily be implemented in a custom ASIC – in fact, the simple units that make up the systolic array are designed explicitly for ease of ASIC implementation – the use of FPGA allows the user to utilize parameterized designs which allow for variable numbers of spectral bands as well as optimized memory sizes for a particular problem. As well, FPGAs allow the design to be scaled upward easily as process technology allows for ever-larger gate counts.

Num Units	Area (slices)	Area (%)	Mult (of 232)	BRAM (of 232)
3	5277	22	6	20
8	8562	36	16	40
16	14628	64	32	72
24	18165	76	48	112

Table 3: Some mapping results on the Cray XD-1, with a Xilinx 2vp50 -7 running at 130 MHz

platform	time (sec)	loads	rate/sec	speedup
CPU	0.25	21600 words	86376	1
XD-1 (SRAM)	0.0002	21600 words	130000000	1512
XD-1 (DRAM)	0.028	3443 blocks	6800	8.9

Table 4: Comparison between CPU and XD-1 performance for both SRAM-only graphs and DRAM-based systems with SRAM caching.

7 Conclusion

This paper contributes two main advancements to the field of shortest-path computation. One, the bubble-sorted queue, is a restriction that is only possible because of the limited branching structure of transportation grids. However, it enables a much smaller hardware implementation of the A* algorithm, allowing a much higher level of parallelism. This parallelism allows for the memory bandwidth of the XD-1 to be utilized more effectively than if shortest paths were calculated sequentially.

The second contribution is the virtual memory round-robin manager. This is obviously not particularly novel in terms of computer architecture, but its implementation on the XD-1 does provide a capability that is useful to a many applications where data access is random over a large, sparse database.

The next step in improving the modeling capabilities of this architecture is to add support for congestion. By keeping track of the load on any given road segment during a particular time-step, the system can weight edges such that heavily trafficked arteries are less desireable. A* normally sorts on the distance to the destination, but that can easily be changed to a weighted distance based on the congestion and average rates of a particular road link. This will provide more accurate modeling, as cars will be attracted to freeways until they too become congested, and otherwise route toward fast routes and away from traffic.

References

[1] U. Bondhugula, A. Devulapalli, J. Dinan, J. Fernando, p. Wyckoff, E. Stahlberg, and P. Sadayappan. Hardware/Software Integration for FPGA-based All-Pairs Shortest-Path. In *Proceedings of the IEEE Conference on Field-Customizable Custom Computing Machines*, 2006. Floyd-warshall dense graph.

[2] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[3] A. Ioannou and M. Katevenis. Pipelined Heap(Priority Queue) Management for Advanced Scheduling in High Speed Networks. In *IEEE Transactions on Networking*, 2007.

[4] M. Tommiska and J. Skytta. Dijkstra's Shortest Path Routing Algorithm in Reconfigurable Hardware. In *Proceedings of Field-Programmable Logic (FPL '01)*, 2001.