A Case Study of Hardware/Software Partitioning of Traffic Simulation on the Cray XD1

Justin L. Tripp, Member, IEEE, Maya B. Gokhale, Fellow, IEEE, and Anders Å. Hansson, Member, IEEE

Abstract—Scientific application kernels mapped to reconfigurable hardware have been reported to have $10 \times$ to $100 \times$ speedup over equivalent software. These promising results suggest that reconfigurable logic might offer significant speedup on applications in science and engineering. To accurately assess the benefit of hardware acceleration on scientific applications, however, it is necessary to consider the entire application including software components as well as the accelerated kernels. Aspects to be considered include alternative methods of hardware/software partitioning, communications costs, and opportunities for concurrent computation between software and hardware. Analysis of these factors is beyond the scope of current automatic parallelizing compilers.

In this paper, a case study is presented in which a simulation of metropolitan road traffic networks is mapped onto a reconfigurable supercomputer, the Cray XD1. Five different methods are presented for mapping the application onto the combined hardware/software system. An approach for approximating the performance of each method is derived through analytic equations. Our results, both analytically and empirically, show that key predictors of performance (which are often not considered in reported speedup of kernel operations) are not necessarily maximum parallelism, but must account for the fraction of the problem that runs on the reconfigurable logic and the amount data flow between software and hardware.

Index Terms—Simulation, System Integration, HW/SW Co-design

I. INTRODUCTION

RECONFIGURABLE coprocessing puts the extreme performance potential of programmable hardware to work on computationally intensive algorithms in science and engineering. Combining high performance microprocessors, large Field Programmable Gate Arrays (FPGAs), and low latency, high bandwidth interconnect, reconfigurable coprocessors have demonstrated $10$–$100 \times$ acceleration on compute-intensive scientific kernels, e.g. [1], [2]. Leading supercomputer vendors [3]–[5] offer machines that include programmable logic, and new software tools are appearing [6]–[8] that compile high level languages to hardware.

Reconfigurable coprocessors offer the potential to improve computational performance by orders of magnitude. The use of hardware logic yields opportunities for improving both coarse-grained (application-level), medium-grained (instruction-level) and fine-grained (operation-level) parallelism. First, FPGAs provide a large amount of highly parallel, reconfigurable hardware resources, which makes it possible to create structures, such as parallel multiply/add instructions, that greatly accelerate individual operations. Innermost loops can be unrolled to expose additional instruction level parallelism. Loops can also be accelerated through careful scheduling of compute and memory access instructions and by loop pipelining. At a higher level, these parallel structures can be replicated to create additional levels of parallelism, up to the limit of the target device capacity. Exposing and exploiting parallelism is critical to performance as FPGA clock rates are a factor of $10$ slower than high performance microprocessors.

While the hardware offers many opportunities for parallelism, the size and scale of scientific and engineering applications present obstacles to reconfigurable supercomputing. Application programs are often several hundred thousand lines of code. The computationally intensive code segments must be located, and the code partitioned between software and hardware, with the kernel being re-written either in a Hardware Description Language (HDL) or a C dialect that can be compiled to hardware.

Additionally, scientific applications are often dominated by 64-bit floating point computation, which consumes too much area and memory bandwidth on present-day FPGAs to be competitive with dedicated 64-bit floating point units on microprocessors. Computation with gigabyte memory arrays do not usually fit on current FPGA boards, thus requiring the communication of large blocks of data between software and hardware. Often, the kernels that are amenable to hardware acceleration cannot easily be overlapped with software computation, and overall speedup is limited according to Amdahl’s Law [9].

It is usually the case that there are many different ways to partition the application, to implement the kernels in hardware, and to manage communication and synchronization. To exploit the full capacity of FPGA-based computing, it is essential to carefully select the portion(s) of the code to implement in hardware. Ideally, time spent in the code kernel(s) should dominate overall run time, and the code kernel(s) should be able to exploit pipelining and replication, the sorts of spatial parallelism offered by FPGAs. The software portions of the algorithm may also need to be tuned to minimize communication time and to concurrently compute in both software and hardware.

In this paper, we make four significant contributions to the hardware/software partitioning problem. First, we present the mapping of a computationally intense application, metropolitan road traffic simulation, onto a reconfigurable computer, the Cray XD1. The simulation uses the same algorithms and data sets that are used in the TRANSIMS [10] road
network simulator. Second, we demonstrate how to combine judicious implementation with analysis to explore the design space. Third, we quantitatively document methods to optimize communication, which we have demonstrated on the Cray XD1. Finally, we have shown how maximum parallelism does not always equate maximum performance.

There are many different ways that road simulation can be mapped onto hardware/software. In this work, we present five different partitioning schemes and analyze their performance on the XD1. The analysis includes comparing different levels of task parallelism, restructuring of the data, and testing different communication models. This level of detail cannot be done using automatic parallelizing compilers. Four of the approaches presented have been implemented, and performance statistics from these are used to analyze the other alternatives. Our results show that the critical predictors of performance are determined by 1) the fraction of work that runs in hardware, 2) the cost of data transfer and 3) the degree of concurrent execution between software and hardware.

II. CELLULAR AUTOMATON TRAFFIC MODELING

We consider traffic simulation because 1) it is not dominated by 64-bit floating-point computations, 2) its data structures naturally have a high degree of parallelism, and 3) the application is a challenging computation problem due to the large amounts of data needed to describe a realistic urban road network.

The TRANSIMS road network simulator, which is based on Nagel-Schreckenberg [11]–[13] model for freeway traffic, can best be described as a cellular automaton (CA) computation on a semi-regular grid or cell network: The representation of the city road network is split into nodes and links. Nodes correspond to locations where there is a change in the road network such as an intersection or a lane merging point. Nodes are connected by links that consist of one or more unidirectional lanes. A lane is divided into road cells each of which are 7.5 meters long. One cell can hold at most one car, and a car can travel with velocity $v \in \{0, 1, 2, 3, 4, 5\}$ cells per iteration step. The positions of the cars are updated once every iteration step using a synchronous update, and each iteration step advances the global time by one second. The basic driving rules for multi-lane traffic in TRANSIMS can be described by a four-step algorithm. In each step we consider a single cell $i$ in a given lane and link. Note that our model allows passing on the left and right. To avoid cars merging into the same lane, cars may only change lanes to the left on odd time steps and to the right on even time steps. This convention, along with the four algorithm steps described below, produces realistic traffic flows as demonstrated by TRANSIMS.

Local Driving Rules: The micro-simulator has four basic driving rules. We let $\Delta(i)$ and $\delta(i)$ denote the cell gap in front of cell $i$ and behind cell $i$, respectively, $v$ is the velocity of the car in cell $i$ and $\upsilon_{\text{max}}(i)$ is the maximum velocity for this particular road cell, which may be lower than the global $\upsilon_{\text{max}}$ (e.g., a local speed limit).

1) Lane Change Decision: Odd time step $t$: If cell $i$ has a car and a left lane change is desirable (car can go faster in target lane) and permissible (there is space for a safe lane change), flag the car/cell for a left lane change. The case of even numbered time steps is analogous. If the cell is empty nothing is done. The lane change decision is also stochastically modified by the parameter $p_{\text{lane, change}}$.

2) Lane Change: Odd time step $t$: If there is a car in cell $i$, and this car is flagged for a left lane change then clear cell $i$. Otherwise, if there is no car in cell $i$ and if the right neighbor of cell $i$ is flagged for a left lane change then move the car from the neighbor cell to cell $i$. The case of even time steps $t$ is analogous.

3) Velocity Update: Each cell $i$ that has a car updates that car’s velocity using the two-step sequence:

- $v_{\text{next}} := \min(v + 1, \upsilon_{\text{max}}(i), \Delta(i))$ (acceleration)
- If $[\text{UniformRandom()} < p_{\text{brake}}]$ and $[v > 0]$ then $v_{\text{next}} := v - 1$ (stochastic deceleration).

4) Position Update: If there is a car in cell $i$ with velocity $v = 0$, do nothing. If cell $i$ has a car with $v > 0$ then clear cell $i$. Else, if there is a car $\delta(i) + 1$ cells behind cell $i$ and the velocity of this car is $\delta(i) + 1$ then move this car to cell $i$. The nature of the previous velocity update pass guarantees that there will be no collisions.

All cells in a road network are updated simultaneously. The steps 1–4 are performed for each road cell in the sequence they appear. Each step above is thus a classical cellular automaton $\Phi_i$. The whole combined update pass is a product CA, $\Phi = \Phi_4 \circ \Phi_3 \circ \Phi_2 \circ \Phi_1$, i.e., a functional composition of classical CAs. Note that the CAs used for the lane change and the velocity update are stochastic CAs. The rationale for having stochastic braking is that it produces more realistic traffic. The fact that lane changes are done with a certain probability avoids slamming behavior where whole rows of cars change lanes in complete synchrony.

Intersections and Global Behavior: The four basic rules handle the case of straight roadways. TRANSIMS uses travel routes to generate realistic traffic from a global point of view. Each traveler or car is assigned a route that he/she has to follow. Routes mainly affect the dynamics near turn-lanes and before intersections as cars need to get into a lane that will allow them to perform the desired turns. To incorporate routes the road links need to have IDs assigned to them. Moreover, to keep computations as local as possible, cells must store information about the IDs of upcoming left and right turns. The details of the intersection rules are explained in [14].

III. DESCRIPTION OF THE CRAY XD1

A single chassis of the Cray XD1, consists of 12 AMD Opteron 200 series processors, with up to 8 Gigabytes of memory per processor. The processors are paired in SMP processor modules. Each processor has 1 or 2 Cray RapidArray links that connect to a fully non-blocking Cray RapidArray fabric switch. The switch provides either 48 GB/s or 96 GB/s total bandwidth between the SMP pairs. The RapidArray fabric switch is able to achieve 1.7 $\mu$s MPI latency between SMPs.

As shown in Figure 1, one Xilinx Virtex-II Pro (30 or 50) is available for each processor module from the RapidArray fabric. An FPGA has 3.2 GB/s link to the RapidArray fabric,
which connects to the local processors or to other processors on the fabric. The FPGA also has dedicated 2 GB/s RocketIO links to the neighboring SMP module in the same chassis. Four QDR SRAMs are connected to each FPGA providing 3.2 GB/s of bandwidth at 200 MHz [3].

The Cray XD1 architecture differs from other reconfigurable coprocessors in that the FPGA module communicates directly with an interconnect module associated with an SMP module. Thus, each FPGA module is tied to a specific SMP module, in contrast to the SGI [4], in which FPGA resources are not associated with any particular CPU. The Cray uses Opteron processors and has a proprietary interconnection network based on hypertransport (HT). FPGAs can be aggregated either directly using the on-chip Rapid I/O interconnect or through the RapidArray hypertransport interconnect. The latter requires that the CPUs associated with FPGAs manage the communication, adding latency to the hypertransport interconnect path. As shown in Figure 1, the FPGA module on the XD1 board connects directly to the RapidArray link of one of the dual socket Opterons.

IV. MAPPING TRAFFIC SIMULATION TO A RECONFIGURABLE COPROCESSOR

FPGAs have previously been applied to the traffic simulation problem. The earliest system, by George Milne [15], [16], simulated road networks by directly implementing their behavior in hardware. Milne’s direct implementation uses Algotronix’s CAL FPGAs to create a long single-lane road of traffic. The cars can be placed on the road and their behavior with respect to each other simulated. Results of the simulation are read-back from each of the chips used in the simulation. Cars were able to have two speeds (go/stop) and their behavior was determined based on the presence of their nearest neighbor. The direct implementation approach has a very high degree of concurrency that is limited by the amount of hardware available and the level of data visibility required by the simulation.

Although to date, firmware and software to support this feature are not available.

A more recent system, by Marc Bumble [17], implements a generalized system for parallel event-driven simulation. His system consists of an event generator, an event queue, a scheduler, and a unifying communications network in each processing element. The traffic simulation is calculated by streaming data into processing elements. Each processing element is capable of simulating one source, intersection, or destination node with the associated outbound roads of traffic. This system does not scale due to a high FPGA requirements (>200,000). The author does not explain how how data is transferred in and out of the system or how to achieve visibility of the simulated traffic. Also, his road models are limited to single-lanes with simple four-way intersections. This approach does not lend itself to the simulation of metropolitan areas.

The work presented here differs from previous approaches in three ways. First, we are using simulation models which are currently in production use. TRANSIMS models include acceleration, stochastic slow-down, different velocities and cars with routes. Second, we extend our simulation to entire metropolitan areas rather than a specialized configuration with a small number of roads and intersections. Previously, the cost of metropolitan scale traffic simulations solely on FPGAs was too expensive, so as a third point we will examine the cost of partitioning the simulation between the microprocessors and FPGAs. All of these differences help determine the utility of FPGAs in the context of large-scale simulations, such as TRANSIMS.

In order to explore FPGA-based hardware acceleration of the TRANSIMS micro-simulator, we first profiled the execution time of a single-processor, pure software implementation. The original TRANSIMS lacked a single-processor implementation and thus it was necessary to develop a new microsimulator, which implemented the same traffic interaction rules and was compatible with TRANSIMS data files. Targeting the Cray XD1 supercomputer, we profiled our executable on an AMD Opteron 200 series processor with 8 Gigabytes of memory. The results are listed in the first column of Table I below.

**TABLE I**

<table>
<thead>
<tr>
<th>Activity</th>
<th>Fraction of Overall time</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lane change update (T_{lane})</td>
<td>20.0%</td>
<td>.11</td>
</tr>
<tr>
<td>Velocity update (T_v)</td>
<td>34.6%</td>
<td>.19</td>
</tr>
<tr>
<td>Position update (T_{pos})</td>
<td>32.7%</td>
<td>.18</td>
</tr>
<tr>
<td>Intersection update (T_{inter})</td>
<td>12.6%</td>
<td>.07</td>
</tr>
</tbody>
</table>

From the profile, we understand that one could potentially achieve 8× speedup by processing (i) lane change updates, (ii) velocity updates, and (iii) position updates in configurable logic, such as FPGAs. A complicated set of rules controls intersections and therefore this task is not as suitable for FPGA implementation. 8× is a rough upper bound that assumes all lane change updates, velocity updated, and position updates would be done in hardware and does not account for communication costs, but it is nonetheless a number that can serve as motivation for our approach.
To perform the acceleration, computation must be partitioned between the work which will be performed in software and the work will be performed by the accelerator. We include an analysis of the percent of time being spent in the accelerator and software during simulation. Since this analysis is applied to several different approaches, we will next discuss the equations used as the basis for this analysis.

After initialization of the road network, the traffic simulation iterates for the desired number of simulation steps. In each simulation step, it performs the road cell update described in Section II for each road cell in the network. We will only consider the time required for a single simulation step,

\[
T_{iter} = T_{sw_a} + T_{sw_b}.
\]

\(T_{sw_a}\) represents the time it takes to execute the software functions not related to hardware acceleration. \(T_{sw_b}\) is the time it takes to execute the software code related to hardware acceleration. It includes the time to communicate data to the FPGA and the time spent by the FPGA processing that data. The goal of acceleration is to reduce the overall time for a single iteration \(T_{iter}\) by decreasing \(T_{sw_a}\) and increasing the amount of work done in a smaller \(T_{sw_b}\).

In the most basic case, \(T_{sw_b}\) is defined by

\[
T_{sw_b} = T_{comm_1} + T_{hw} + T_{comm_2} + T_{sw_f}.
\]

\(T_{comm_1}\) is the time it takes to re-format and write data to the accelerator. \(T_{hw}\) is the time required to process the data and produce the results. \(T_{comm_2}\) represents the time required to move the results from the accelerator back to the host machine and the final attribute \(T_{sw_f}\) is the time required to merge the results back into the original software.

In the approaches described, the road network is partitioned in such a way that the most common case, straight road sections, are processed by the hardware, while intersections and merging nodes are updated by a software module. Most importantly, this hybrid hardware/software strategy means that hardware processing is governed by a simple, homogeneous set of traffic rules, while all road plan decisions are handled by software. Coprocessing with hardware and software means that data must be shared between both domains. As with parallel programming, data on the “edges” must be exchanged to keep both domains current.

**TABLE II**

<table>
<thead>
<tr>
<th>City</th>
<th>Number of Lanes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portland</td>
<td>88.6 9.8 1.6 0.5</td>
</tr>
<tr>
<td>Houston</td>
<td>85.9 10.0 1.6</td>
</tr>
<tr>
<td>Chicago</td>
<td>85.9 13.7 0.5</td>
</tr>
</tbody>
</table>

A common approach applied in reconfigurable computing, streams data from the memories to a number of computational units. In the context of traffic simulation, streaming can be achieved through a computation engine that processes a stream of road data and subsequently outputs a stream of updated data. Using this approach, the number of road cells are not limited to available FPGA area. A streaming hardware design is scalable and can handle large-scale road networks.

The data representing straight lanes is fed to the hardware update engine against the flow of traffic, starting from the end of each lane. However, due to the partitioning of the road network, the cars in the last \(v_{max}\) cells of each lane cannot be updated since the engine lacks knowledge about the network topology and the road plans. For this reason we define an overlap region \(I\) (see Figure 2(a)), which is the last \(v_{max}\) cells of each lane, and because of the processing direction of the computation engine, these cells are processed first. Although cars that are inside an overlap region at the beginning of the hardware computation cannot be updated by the engine, it is important to note that the engine can move other cars into these first cells during its computational pass. Naturally, the software module needs to update the position and velocity of cars inside the overlap regions at the end of each hardware update pass.

![Fig. 2. Traffic Acceleration Road Link Structure and the Position Update Hardware](image)

The software must also write information to the first \(v_{max}\) cells of each lane, which corresponds to new cars moving into a lane (arriving from other lanes, either through an intersection or by merging). However, computing velocities and positions of these new cars requires complete knowledge of the first \(v_{max}\) cells of each lane. Therefore, the first \(v_{max}\) cells of each lane constitute another overlap region (see overlap region II in Figure 2(a)) whose state information must be known in software.

In consequence of the hardware/software partitioning, there is a need for synchronizing the status of road cells in the overlap regions. In order to minimize the cost of memory synchronization, it is desirable to process only single-lane traffic in hardware (and let software process multiple-lane traffic). The reason is that multiple-lane traffic would require the synchronization of more than \(v_{max}\) cells since lane change permissibility assumes knowledge of preceding cells.\(^2\)

Restricting the hardware computation to single-lane traffic may sound like a severe limitation. However, the road lane

\(^2\)There is an exception: If the cells were empty at the beginning of the update pass, the software module does not need to read back the updated status since hardware can only move cars from the first cells.
distribution for three U.S. cities, shown in Table II, reveals that close to 90% of all roads are one-lane roads. This means that an overwhelming majority of all road cells will in fact be updated in hardware, and significant hardware acceleration remains a possibility.

The streaming engine calculates a car’s position by shifting the car one cell every clock cycle (see Figure 2(b)) until its newly calculated velocity matches the distance from the end of a shift register, which is \( v_{max} + 1 \) cells long. At the point when there is a match, the streaming engine exports all car information to the destination road cell. This pipelining design makes it possible for the computation engine to read and write one word of road data every clock cycle. If we have access to \( N \) concurrent data streams, it would be advantageous to instantiate \( N \) parallel replicas of the compute engine.

In the following, we will describe several different hardware and software partitioning strategies and designs to map road network simulation onto the Cray XD1. The approaches are divided into two major groups: Maximize Coprocessor Parallelism and Maximize Performance. The two groups explore two different extremes in the design space. The analysis demonstrates that the design space exploration must be performed to quantify trade-offs, and understand that parallelism may not be equivalent to performance.

V. MAXIMIZE COPROCESSOR PARALLELISM

Our initial approach were driven by the desire to maximize the available parallelism in the FPGA. When accelerating the common case, it is best to reduce the time required to complete the computation. This can be done by performing as many operations as possible in parallel on the co-processor.

A. Direct Approach

With CAs, a straightforward way to take advantage of the concurrency is to build the CA directly in hardware. The direct implementation of the traffic simulation CA instantiates a separate road cell for each road cell in the traffic network. The road cell provides its current state to its neighbors so that all the cells in that local neighborhood can calculate their next state.

This approach provides the highest level of parallelism available in programmable hardware. The entire traffic network operates concurrently and executes a single simulation second at megahertz speeds. A simple stop intersection and multi-lane designs were implemented using this approach. Multi-lane designs are able to achieve 65MHz clock speeds, where six clock cycles were required for each iteration step. However, it would take approximately 12,400 FPGAs to implement the entire Portland road network, not including the cost for implementing several different kinds of intersections. Furthermore, each FPGA would have a different design, implying that the 12,400 different designs would have to be put through synthesis, place and route. Due to the large amount of parallelism, the direct implementation is prohibitively expensive.

For a single iteration the direct implementation approach uses the following equation:

\[
T_{iter} = T_{hw}
\]

No software is necessary and \( T_{sw} \) becomes \( T_{hw} \). The time for an iteration of the simulation is the time for the most complex hardware logic to complete. Due to the infeasibility of this approach, we will not consider it further.

Despite the high cost, this approach represents the extreme end of the parallel spectrum. If new methods for realizing hardware (e.g., nanomaterials) allow for millions of concurrent state machines and their related logic, then it will be possible to take advantage of the greater amount of parallelism.

B. Partitioning by External Memory

The Cray XD1 combines high performance microprocessors, a high speed, low latency interconnect network, and reconfigurable computing elements. This provides a system where data transfer latencies and bandwidth associated with I/O buses is greatly reduced. Normally, reconfigurable systems need to process large amounts of data to amortize the cost of transferring the data to the FPGA. However, with tight integration of processors and reconfigurable computing, a computation problem can be split between the CPU and FPGA with close synchronization and fast communication between software and hardware.

Since the direct approach is not feasible, the greatest bandwidth is available between the SRAMs and the FPGA. This provides up to 3.2 Gigabytes per second of bandwidth per memory port. The host will need to send data to the memories as well as receive and send updates to the simulation. After the FPGA’s SRAMs are loaded with road network data, the processor will only exchange updates for shared data.

As described in the XD1 FPGA Development manual [18], there are asymmetric costs associated with reads and writes between the host and the FPGA’s QDR SRAMs. Writes can take advantage of write combining in the Linux kernel and are non-blocking. On the other hand, reads are blocking and cannot be combined. This creates an asymmetric cost which must be overcome.

For a single iteration the direct implementation approach uses the following equation:

\[
T_{iter} = T_{hw}
\]

No software is necessary and \( T_{sw} \) becomes \( T_{hw} \). The time for an iteration of the simulation is the time for the most complex hardware logic to complete. Due to the infeasibility of this approach, we will not consider it further.

Despite the high cost, this approach represents the extreme end of the parallel spectrum. If new methods for realizing hardware (e.g., nanomaterials) allow for millions of concurrent state machines and their related logic, then it will be possible to take advantage of the greater amount of parallelism.

B. Partitioning by External Memory

The Cray XD1 combines high performance microprocessors, a high speed, low latency interconnect network, and reconfigurable computing elements. This provides a system where data transfer latencies and bandwidth associated with I/O buses is greatly reduced. Normally, reconfigurable systems need to process large amounts of data to amortize the cost of transferring the data to the FPGA. However, with tight integration of processors and reconfigurable computing, a computation problem can be split between the CPU and FPGA with close synchronization and fast communication between software and hardware.

Since the direct approach is not feasible, the greatest bandwidth is available between the SRAMs and the FPGA. This provides up to 3.2 Gigabytes per second of bandwidth per memory port. The host will need to send data to the memories as well as receive and send updates to the simulation. After the FPGA’s SRAMs are loaded with road network data, the processor will only exchange updates for shared data.

As described in the XD1 FPGA Development manual [18], there are asymmetric costs associated with reads and writes between the host and the FPGA’s QDR SRAMs. Writes can take advantage of write combining in the Linux kernel and are non-blocking. On the other hand, reads are blocking and cannot be combined. This creates an asymmetric cost which must be overcome.

For a single iteration the direct implementation approach uses the following equation:

\[
T_{iter} = T_{hw}
\]

No software is necessary and \( T_{sw} \) becomes \( T_{hw} \). The time for an iteration of the simulation is the time for the most complex hardware logic to complete. Due to the infeasibility of this approach, we will not consider it further.

Despite the high cost, this approach represents the extreme end of the parallel spectrum. If new methods for realizing hardware (e.g., nanomaterials) allow for millions of concurrent state machines and their related logic, then it will be possible to take advantage of the greater amount of parallelism.

B. Partitioning by External Memory

The Cray XD1 combines high performance microprocessors, a high speed, low latency interconnect network, and reconfigurable computing elements. This provides a system where data transfer latencies and bandwidth associated with I/O buses is greatly reduced. Normally, reconfigurable systems need to process large amounts of data to amortize the cost of transferring the data to the FPGA. However, with tight integration of processors and reconfigurable computing, a computation problem can be split between the CPU and FPGA with close synchronization and fast communication between software and hardware.

Since the direct approach is not feasible, the greatest bandwidth is available between the SRAMs and the FPGA. This provides up to 3.2 Gigabytes per second of bandwidth per memory port. The host will need to send data to the memories as well as receive and send updates to the simulation. After the FPGA’s SRAMs are loaded with road network data, the processor will only exchange updates for shared data.

As described in the XD1 FPGA Development manual [18], there are asymmetric costs associated with reads and writes between the host and the FPGA’s QDR SRAMs. Writes can take advantage of write combining in the Linux kernel and are non-blocking. On the other hand, reads are blocking and cannot be combined. This creates an asymmetric cost which must be overcome.
and the traffic engine to write to the SRAMs. Reads do not require a mux for the data, but both read and write require muxes for the address lines (not shown). Using this setup we benchmarked reads and writes from the host to the FPGA.

Table III shows the bandwidth results for reads and writes accomplished using four different approaches. The FPGA SRAMs are mapped into the host’s local memory and can be accessed using arrays with indexes, using pointers, or by using the `memcpy` function call. Array accesses were found to be slightly slower, but the bandwidth is roughly equivalent. “Pairs” measures the cost for reading and writing non-adjacent memory locations. Hypertransport data packets contain eight adjacent 64 bit words. Writing words that are not adjacent reduces the efficiency since each packet holds a single 64 bit word.

The difference in cost between reads and writes is a factor of 200. Due to the difference in reads versus writes in the XD1, the described approach results in an overall slowdown—.565s with hardware processing compared to .550s for software alone. A “write-only” approach is required to achieve a speed up.

### C. Write-Only Architecture

To overcome the asymmetric cost between reads and writes a write-only architecture is necessary. The host must write data to the FPGA memory and the FPGA must write the data that the host needs to read. To further reduce communication, the FPGA and CPU will filter their communication by only sending the necessary changes in the shared data.

Using push back and keeping all the data in the available memories, the time spent in each iteration can be described as

\[
T_{iter} = T_{sw_{a1}} + \max(T_{sw_{a2}}, T_{sw_{f}})
\]

With this approach, the software on the processor is split between the work that must be done serially, \(T_{sw_{a1}}\), and the work, \(T_{sw_{a2}}\), which can be done concurrently with work in hardware, \(T_{sw_{f}}\). Due to memory limitations on the Cray XD1, 69% of the total work is done in software, while the remaining 31% is done in hardware. This is because only 31% of the 64-bit road cells can be stored in the external memory of the XD1.

The serial software work is described by

\[
T_{sw_{a1}} = T_{comm_{1}} + T_{int} + T_{lane} + T_{sw_{f}}
\]

where \(T_{comm_{1}}, T_{int}, T_{lane}, \) and \(T_{sw_{f}}\), are the amount of time required to send the data to the FPGA hardware, update the intersections, change lanes and merge the results from the FPGA hardware back into the software simulation. For the Portland network, the measured values for \(T_{int}, T_{lane}, \) and \(T_{sw_{f}}\) are .07s, .11s and .002s. \(T_{comm_{1}}\) is estimated to be .009s based on the average number of cars in overlap regions.

\(T_{sw_{a2}}\) is the amount of time spent doing work in software concurrently with work in hardware \(T_{sw_{f}}. T_{sw_{a2}} = .69 \times (T_v + T_{pos}) = .255s\), where \(T_{pos}\) and \(T_v\) are the measured time to update position and velocity of all road cells in software.

For the design described above, with four engines, the work in the hardware is described as \(T_{sw_{f}} = T_{hw} + T_{comm_{2}}. \) The FPGA can process the design using four engines at 130 MHz, \(T_{hw} = .004s. T_{comm_{2}}\) captures the amount of time to transfer the remaining results out of the FIFO buffers, but is too small to be considered significant. Using Equation 4 to add everything up, the overall predicted speed is .446 seconds. The actual measured speed is .459s (see Section VII).

### D. Data Compression

The original four engine design described above is only able to handle about 2.1 million road cells, which is only 31% of the total number of cells that need to be calculated. This requires 69% of the work to be calculated in software, which significantly decreases the impact of hardware acceleration on the overall application run time. As stated in [19], there are several ways to reduce the size of the data represented on the FPGAs. We compressed the data to 32 bits by removing redundant bits.

Using 32 bits to describe a road cell allowed the amount of data in the external memories to be doubled, which doubles the amount of work that can be completed on the FPGA. The number of engines were doubled, since the engines do not require much area and eight engines can processes twice as much data concurrently.

The eight engine streaming implementation is described by the same equations as the four engine implementation above. With 62% of the work in hardware, \(T_{sw_{a2}} = .38 \times (T_v + T_{pos}) = .14s\), the parallel software time is reduced by almost half. \(T_{hw}\) does not change since the number of traffic engines was doubled as well. \(T_{comm_{2}}\) is doubled to .018s, due to

![Fig. 4](image-url) - The data push process is added to make the design write-only.

![Diagram](image-url) - Rapid Array Fabric and Traffic Engines.
the increase of shared overlap regions, but is still very small compared to the software time. All these values combined in Equation 4 result in a predicted overall execution time of 34s. Measured time for this implementation was 309s.

With the available space on the FPGA an estimated 25 traffic compute engines could be placed in the hardware. Since there are still more FPGA resource to be utilized other data compression techniques could be used to reduce the overall cost of transferring data to the FPGA. However, data compression will require CPU processing of the data. One such compression, could be to send only the cars and their positions instead of sending empty cells. This would allow the data to be expanded in the FPGA and allow for high effective rates of data transfer. Unlike the reduced bit representation, this would be highly data dependent and more difficult to model.

VI. MAXIMIZE OVERALL PERFORMANCE

Despite achieving a high degree of parallelism with the data compression and external memories, the impact of the overall performance is reduced by the percentage of work that can be applied to the coprocessor. The coprocessor processes data faster than a standard microprocessor, but must be fed enough data to take advantage its speed. Thus a new approach was devised, where the parallelism is reduced, but the total amount of work completed by the coprocessor is increased.

A. Streaming Data

The approach described in Section V-B attempts to minimize the amount of communication between the processor and the hardware. This uses a common approach in parallel programming, where only the updates are shared between two nodes. However, measured speeds for this bursty communication does not seem to support this approach. Monitoring of the communication indicates that cars are more often in the overlap regions than expected and that it is costly to transfer these regions due to the sparseness of the data. Tests which transferred pairs of 64 bit words which are offset by a single word, (e.g., one and three, two and four, etc.) indicate that the transfer rate is halved with these sparse writes (see Pairs in Table III). In order to achieve the highest data write efficiency, writes should be done as large contiguous blocks.

Also the design should maximize the amount of work that can be performed on the FPGA. If all the single-lane roads that have one complete overlap region are executed on the FPGA, this would account for 89% of the work to be processed. The data could be processed on the FPGA by having the FPGA transfer large, contiguous blocks of data to the FPGA, and having the FPGA process the data and return results back to the CPU. Since the FPGA starts the memory transfer and puts the results back into the memory of the processor, the processor frees CPU cycles which were used to send the data to the FPGA. The processor can then spend all of its time processing data instead of spending time transferring data.

This approach uses the same high-level Equation 1, where $T_{swb}$ is

$$T_{swb} = (T_{comm1} + T_{hw} + T_{comm2}) + T_{swf},$$

$T_{comm1}$ is the cost to send 89% of the cell data down to the FPGA, $T_{hw}$ is the time to process the data and $T_{comm2}$ is the latency for the first result to arrive in CPU memory. Since the transmission of the results can be overlapped with execution, the last result arrives no later than the latency of sending the communication back.

$$T_{swb} = T_{inter} + T_{lane} + \%work \times (T_v + T_{pos})$$

$T_{swb}$ includes all of the remaining work that cannot be executed on the FPGA. In this case, the $\%work$ remaining on the CPU is 12%.

This approach improves on the previous approaches in three ways. First, the percentage of work on the FPGA is higher. This is an important gain, since communication to the FPGA is cheaper than processing on the CPU. Second, this approach avoids software complexity to synchronize overlap regions on the predominant short road segments (see Table IV). Third, it takes advantage of high performance block transfers. This approach still allows concurrent execution of the software alongside the hardware.

### TABLE IV

<table>
<thead>
<tr>
<th># Lanes</th>
<th>Min.</th>
<th>Median</th>
<th>Mean</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>15.0</td>
<td>28.46</td>
<td>3931</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>26.0</td>
<td>44.96</td>
<td>2738</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>53.5</td>
<td>90.29</td>
<td>2092</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>83.5</td>
<td>176.58</td>
<td>2083</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>124.0</td>
<td>211.49</td>
<td>1450</td>
</tr>
<tr>
<td>6</td>
<td>60</td>
<td>374.0</td>
<td>303.20</td>
<td>451</td>
</tr>
</tbody>
</table>

Only a two engine design is necessary to process data with this streaming approach. Most of the time is still spent in software, where Equation 7 becomes $T_{swb} = .182 + .12 \times (.35) = .222s$. The time to process in hardware is dominated by the cost of transferring the data, where $T_{comm} = .0177s$, and $T_{swb} = T_{comm1} + T_{hw} + T_{comm2} + T_{swf} = .0197s$. The estimated $T_{iter}$ is .226s per iteration. Each iteration of the streaming approach was measured to be .241s.

B. Multi-Lane Data

An improvement on the streaming approach is also compute the velocity update and position update for multiple lanes on the FPGA. In the case of multiple lanes, the velocity update and position update do not consider data other than the cells in front of and behind a particular car. It is then possible to compute the lane change step separately and then transfer the updated data down to the FPGA for position and velocity calculations. This will allow 100% of the possible data to be computed on the FPGA.

With this approach the equations are the same as those in the previous section, but instead of 12%, all of the position and velocity update computation can be moved to the FPGA. Equation 7 becomes $T_{swb} = .182 + 0.0 \times (.35) = .182s$. The time to process in hardware is dominated by the cost of transferring the blocks, where $T_{comm} = .0202s$, and $T_{swb} = T_{comm1} + T_{hw} + T_{comm2} + T_{swf} = .0202s$. The estimated $T_{iter}$ is .202s per iteration.
VII. RESULTS AND CONCLUSION

A comparison of the results is shown in Table V. The values shown are the estimated execution for the analyses as well as three measured results. Initial, Four, Eight, Streaming, and AllStream are the results from the initial four-engine, four-engine write-only, eight engine write-only, streaming data, and streaming all the data analyses. Software is the time measured for software and is the basis for the analysis. As more work is moved into hardware with each approach, the amount of time required to calculate each iteration decreases. SWA1, SWA2, SWB and MPI are the times for $T_{swg1}$, $T_{swg2}$, $T_{swb}$ and $T_{commmp1}$. Measured is the time for software and the three engines that were built for the Cray XD1.

<table>
<thead>
<tr>
<th>TABLE V</th>
<th>COMPARISON OF APPROACHES IN TIME (MS)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SWA1</td>
</tr>
<tr>
<td>Software</td>
<td></td>
</tr>
<tr>
<td>Initial</td>
<td>190.0</td>
</tr>
<tr>
<td>Four</td>
<td>199.8</td>
</tr>
<tr>
<td>Eight</td>
<td>182.0</td>
</tr>
<tr>
<td>Streaming</td>
<td>182.0</td>
</tr>
<tr>
<td>AllStream</td>
<td></td>
</tr>
</tbody>
</table>

The results show that the best performance will be obtained through data transfer controlled by the FPGA and stream processing. The analysis demonstrated in this work is indicative of the sort of design space exploration that must be done to understand potential speedup of different mapping techniques. Although some aspects of the analysis are specific to the problem, many of the issues are common to coprocessor acceleration. Often descriptions of reconfigurable applications are limited to kernels. Few consider the whole application and the combine measurement with careful analytical analysis that is conducted in this work.

As demonstrated above, the most important design goals for accelerating coprocessing applications are to maximize: 1) the portion of the application that can run in hardware, 2) the portion of the application that can run in software concurrently with the hardware and 3) reduce communication costs. If the hardware (and concurrent software) part of the work accounts for 90% of the time, then, at best, a $10 \times$ speed up is possible. If the hardware (and concurrent software) part of the work is only 50% of the time, the best it can be is $2 \times$. This maximal speed up will be further reduced by the overhead required to send and receive the results if communication cannot be overlapped with computation.

In the case of traffic simulation, several large external memories with separate memory ports can be used to increase the impact of acceleration. Larger memories on the XD1 – 25MB for the Portland data set – would allow faster simulation speeds by permitting more of the road data to stay resident on the FPGA board. However, the data requirements for traffic simulation increase to the GB range as larger areas and larger cities need to be simulated. At some point, the on-board external memory cannot fit the data set, and then the network bandwidth must be larger and faster to balance the computational power of the FPGA.

New FPGA chips and systems must be designed that can better match between the high level of parallelism and its appetite for data. Without this, the parallelism can only be exploited for application in which computation requires many cycles or where there is natural data re-use. 3D stacking technology might be of benefit to break the bandwidth barrier [20].

In this paper, we have addressed the impact of different design decisions. We investigated a cellular automaton model for simulating large-scale urban roadway traffic and used a Cray XD1 as our target platform. Based on the execution profile of a pure software implementation, and based on the problem structure, we chose to partition the simulation such that all straight road sections are processed in reconfigurable hardware, while the intersections are processed by the CPU.

Even with these restrictions, we were left with several different design approaches in both hardware and software. Here we reported on several alternative mappings and discussed tradeoffs, performance bottlenecks, and techniques maximize acceleration. Amdahl’s law helps explain that we need to move the maximum amount of work to the accelerator to gain the maximum benefit. However, increasing parallelism is not the only trade-off that must be done to achieve higher system-level performance. One must also consider the cost of transferring data between the processor and coprocessor as well as the degree of concurrent execution between software and hardware. Complete understanding of the trade-offs of the system are necessary to know how to properly fit the algorithm to the efficient parts of the system. Using a combination of measurement and analysis, we show speed up of more than $2 \times$ over software on a single node, and an order of magnitude improvement of an MPI-parallel hardware-accelerated implementation over a single software node.

REFERENCES


Justin L. Tripp (S’94–M’04) received a B.S., M.S. and Ph.D. from Brigham Young University in 1996, 1999, and 2004, respectively. He is currently a Technical Staff Member on the Advanced Architectures Team in the Computer and Computational Science Division at Los Alamos National Laboratory, Los Alamos, New Mexico. His research interests include reconfigurable logic designs, synthesis, compilers, and parallel computing.

Maya B. Gokhale (M’99,SM’05,F’07) received the PhD degree from the University of Pennsylvania in computer and information sciences in 1983 and has held positions in industry (Unisys, Hewlett-Packard, Sarnoff Corporation), academia (University of Delaware), and federal research labs (IDA, Los Alamos National Laboratory, Lawrence Livermore National Laboratory). She is a computer scientist in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. Her research interests include reconfigurable computing with FPGAs, storage intensive supercomputing, high performance computing, parallel languages, and embeddable architectures. Her book “Reconfigurable Computing: Accelerating Computation with FPGAs” by Gokhale and Graham was published by Springer in 2005. She is on the program committees of the IEEE Field-Programmable Custom Computing Machines (FCCM) Symposium, the Engineering of Reconfigurable Systems and Applications (ERSA) Conference, and High Performance Embedded Computing (HPEC) Workshop.

Anders Å. Hansson (S’98–M’03) was born in Kalmar, Sweden, on April 30, 1972. He received the M.S. degree in electrical engineering from Chalmers University of Technology, Gothenburg, Sweden, in 1996. In 1997, he was working with microolithography technology in Stockholm and Tokyo. He returned to academia and Chalmers in 1998, spent the 2000–2001 academic year as a Visiting Scholar at University of Southern California, Los Angeles, CA, before he received his Ph.D. degree in electrical engineering from Chalmers in 2003, with a dissertation on detection algorithms for mobile radio channels. He continued to follow a nonlinear trajectory to the Information Sciences Group at Los Alamos National Laboratory, where he is currently a Team Leader, conducting research on analysis and modeling of large-scale socio-technical systems. Dr. Hansson is also interested in belief propagation algorithms.