GraphStep: A System Architecture for Sparse-Graph Algorithms

Michael deLorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo,
Ian Eslick, Raphael Rubin, Tomás E. Uribe, Thomas F. Knight, Jr., and André DeHon
Dept. of CS, MC 256-80
California Institute of Technology
Pasadena, CA 91125
contact: `<andre@cs.caltech.edu>`

Abstract— **Many important applications are organized around long-lived, irregular sparse graphs (*e.g.*, data and knowledge bases, CAD optimization, numerical problems, simulations). The graph structures are large, and the applications need regular access to a large, data-dependent portion of the graph for each operation (*e.g.*, the algorithm may need to walk the graph, visiting all nodes, or propagate changes through many nodes in the graph). On conventional microprocessors, the graph structures exceed on-chip cache capacities, making main-memory bandwidth and latency the key performance limiters. To avoid this "memory wall," we introduce a concurrent system architecture for sparse graph algorithms that places graph nodes in small distributed memories paired with specialized graph processing nodes interconnected by a lightweight network. This gives us a scalable way to map these applications so that they can exploit the high-bandwidth and low-latency capabilities of embedded memories (*e.g.*, FPGA Block RAMs). On typical spreading-activation queries on the ConceptNet Knowledge Base, a sample application, this translates into an order of magnitude speedup per FPGA compared to a state-of-the-art Pentium processor.**

I. INTRODUCTION

We have long noted that spatial hardware organizations (*e.g.*, FPGAs, reconfigurable architectures) offer computational density superior to conventional, temporal hardware organizations [1], [2]. This conference has reported numerous compute-intensive applications where FPGAs deliver orders of magnitude higher performance than processor-based systems. Nonetheless, many problems are limited by memory speed rather than computation. As processing speed grows faster than memory speed, the effect is exacerbated, leaving many applications limited by memory performance rather than compute performance [3], [4].

Spatial organization of computations turns many memory operations into interconnect [2]. Nonetheless, it often remains infeasible to implement tasks with large data sets in a fully spatial manner (*e.g.*, [5]), leaving a need to use memories for virtualization. To address this need, modern FPGAs integrate increasingly larger quantities of on-chip memory. The aggregate memory bandwidth accessible from the collection of small, distributed memories on modern FPGAs is two orders of magnitude larger than the memory bandwidth available on processors (Section II). This presents a new opportunity for FPGAs to offer superior performance to microprocessors on data-intensive applications.

Algorithms representing data with sparse graphs are a large class of these data-intensive applications. While small graphs can be directly implemented spatially in FPGAs (*e.g.*, [6], [7]), the size of graphs that can be realized with a modest number of FPGAs is extremely limited. Consequently, we introduce a new concurrent system architecture for sparse graph-processing algorithms. The system architecture provides a high-level way to capture a broad range of graph-processing tasks abstracted from the detailed hardware implementation. We can efficiently map tasks in this system architecture to collections of FPGAs with embedded memories, allowing performance to scale with the number of FPGAs used to solve the problem. The new system architecture is complementary to compute-intensive system architectures like SCORE [8], providing a natural way to capture data-intensive applications.

The novel contributions of this work include:

1. Highlighting the raw, memory-bound performance potential of FPGA hardware
2. Introducing a data-centric system architecture for sparse-graph applications
3. Mapping this new system architecture to FPGAs with a collection of small distributed on-chip memories
4. Identifying applications which could benefit from the new system architecture
5. Demonstrating the performance benefit on a sample application

II. RAW MEMORY PERFORMANCE

Table I summarizes the raw, aggregate memory bandwidth available on processors and FPGAs to both on- and off-chip memory. In each case, this is computed in the most simplistic and direct way. For the processors, on-chip bandwidth is the bandwidth available from L1 memory. For the FPGAs, on-chip bandwidth assumes that the specified RAMs (Block RAMs, M4K's) operate concurrently at their dual-port operating speed (given by the memory clock speed) and transferring data on both ports at the highest rated data width. For the FPGA off-chip bandwidth, we assume that the off-chip pins are dedicated SDRAM interfaces (twelve 32b SDRAM interfaces operating at 200MHz for Virtex-2, twelve 32b SDRAM interfaces operating at 300MHz for Virtex 4, eight 16b SDRAM interfaces operating at 300MHz on two edges for Stratix 2).

TABLE I

RAW MEMORY BANDWIDTH AVAILABLE FROM FPGAS AND PROCESSORS

Family	Pentium-4	Virtex-2	Virtex-4	Stratix-2
Chip	Pentium-4 550	XC2V6000	XC4VLX200-12	EP2S180
Technology	90 nm	150 nm	90 nm	90nm
Memory Clock	3.4 GHz	260 MHz	500 MHz	475 MHz
On-chip Memory BW	0.2 Tb/s	1.2 Tb/s	5.4 Tb/s	12 Tb/s
from	L1 D-Cache	144 BRAMs	336 BRAMs	768 M4Ks
On-chip Memory Capacity				
at speed quoted	16 KB	288 KB	688 KB	192 KB
total	1 MB	0.29 MB	0.69 MB	1.1 MB
Off-chip Memory BW	51 Gb/s	77 Gb/s	110 Gb/s	77 Gb/s
Reference	[9]	[10]	[11]	[12]

We can make several important observations from this data:

- A single FPGA can offer higher on-chip memory bandwidth than the most advanced microprocessors—one to two orders of magnitude at comparable technology generations.
- For the FPGA, the on-chip bandwidth is one to two orders of magnitude higher than off-chip bandwidth; further, we expect on-chip capacities and hence potential bandwidth to increase more rapidly than off-chip bandwidth, widening the on-chip vs. off-chip bandwidth gap.
- Assuming we can exploit the parallelism, we can scale bandwidth in large systems by tiling FPGAs; similarly, vendors scale the on-chip bandwidth along with compute capacity by scaling the number of independent, on-chip memory banks.

These are, of course, peak memory numbers. Neither architecture is likely to achieve them. Processors can seldom run with their data contained exclusively in L1 memory and practical caching schemes fail to exploit the potential bandwidth available (*e.g.*, [13]). Nonetheless, these observations do point to real performance ceilings and raw potential that we may be able to exploit.

Further, traditional ways of organizing computations result in very significant deviations from these peaks when the dataset is large. That is, traditional processor applications will fetch data and stall execution until the data is returned (allowing multiple outstanding memory references helps, but does not completely compensate for this strategy). Consequently, when the dataset is large and cannot fit in the on-chip memory, bandwidth is limited by the off-chip access latency rather than the on- or off-chip bandwidth. This effect may easily drop effective memory bandwidth by another order of magnitude.

III. IDEA

If we could arrange for all of our data to reside in distributed on-chip memory (*e.g.*, FPGA Block RAMs), and arrange to perform parallel operations and hence parallel access to the data, we could exploit this raw potential (Section II) and achieve orders of magnitude improvement in net memory bandwidth and hence performance on data-centric processing tasks. To handle large tasks, we assemble multiple-FPGA collections to contain the data. This gives us two additional wins:

1) We scale bandwidth and processing with the dataset.
2) We keep all data within a constant (small) latency of the active processing.

Of course, we get less memory capacity per die (per λ^2 or per cm^2 of silicon) using memory in an FPGA than we get using off-chip DRAMs. This is a deliberate trade-off to get higher performance on these tasks. If performance is limiting the application, then this gives us a way to trade area for higher performance.

We can also engineer FPGAs with a different memory/logic balance or with embedded DRAMs (*e.g.*, [14], [15]) that would provide an architectural point between these extremes. These architectures might trade only a factor of 2 to 3 in net memory density for orders of magnitude improvement in usable memory bandwidth.

IV. GRAPH APPLICATIONS

Many applications are naturally represented by sparse graph data structures and can exploit the opportunity identified in the previous section. In these problems:

- The graph is sparse and irregular, meaning nodes have a bounded [$O(1)$] number of edges, but are not necessarily connected in nearest-neighbor fashion in any number of dimensions. Because of the irregular connectivity and data access, it is not possible to localize processing to a small subset of the graph; *i.e.*, traditional *spatial locality* exploited in cache-line blocking and virtual memory pages is not adequate to hide the long delay to off-chip memory on processors.
- Algorithms require that the whole graph (or large fractions of it) be traversed as part of an iteration.
- Algorithms admit to parallelism across the graph.

To be concrete, consider the following kernels and applications:

- **Iterative Sparse Matrix-Vector Multiply** – Here we must complete each sparse matrix-vector multiply (SMVM) before starting the next, and each SMVM requires that we access all the sparse-matrix coefficients. Each entry in the vector result is independent and can be computed in parallel [5].
- **Sparse Neural-Network Evaluation** – This can essentially be the same problem as SMVM above.

- **Shortest Path** – A traditional (*e.g.*, Bellman-Ford [16]) shortest path computation requires that every node update its delay on every cycle. The serialization goes only as the depth (diameter) of the graph, which is typically small compared to the size of the graph for high-speed circuit graphs.
- **Routing** – Routing (*e.g.*, FPGA routing such as Pathfinder [17]) is based on a series of shortest path searches. For nets that cross the entire device, the shortest path search can potentially touch the majority of routing resources in the circuit. When nets are highly localized, it may be possible to perform multiple route searches on different portions of the device in parallel. We already have evidence that this parallelism can lead to substantial speedups in routing [18]–[20].
- **Timing Calculations** – Simple timing analyses (ASAP and ALAP calculations) also perform whole graph traversals in order to update delays and slack.
- **Placement** – Node placement can move a large number of nodes, potentially all of them, and update their costs in parallel [21], [22].
- **Associative Search** – In some applications, we need to check every graph node for some property.
- **Transitive Closure** – Transitive closure is a reachability search that can be seen as a simplified version of the shortest path problem.
- **Marker Passing** – Many knowledge-base queries, inferences, and classification tasks can be supported by algorithms that propagate binary data along neighboring links and perform local and global binary state operations [23], [24].

In general, any application that needs to walk the entire graph will fit the properties noted above, particularly when the operations at each node can be cast as one of the following:
- perform local operation at a node (data parallel)
- accumulate information from nodes (associative reduce)
- propagate information to neighboring nodes

V. GRAPHSTEP SYSTEM ARCHITECTURE

To exploit the idea introduced above, we have developed the GraphStep concurrent system architecture. We call this a concurrent system architecture in the spirit of "Software Architectures" [25], and, in fact, GraphStep is closely related to an Object-Oriented or Repository software architecture. As a concurrent system architecture, GraphStep gives a gross organization for conceiving the task and managing the parallelism in the task.

A. System Architecture Description

In the GraphStep architecture, the computation is organized as a graph of nodes connected by edges.

Nodes: Each node is an object or actor [26]. It has
- local state, typically in typed data fields
- edges to other graph node objects along which it can send messages or method invocations

- a set of methods through which the object data is accessed and modified

It can be useful to think of each object as having its own locus (thread) of control and acting concurrently with all other objects. The program counter is part of its local state. As explained below, the objects synchronize in "steps", so it is alternately possible to simply think of the objects being invoked in a data-parallel, concurrent manner and performing operations that depend on their state.

Methods: In strict, object-oriented fashion, the object can be accessed only through its methods. Most methods are invoked through messages from edges (connected objects), although methods can also be self-invoked or invoked globally (typical in broadcast operations). Methods are of bounded length and atomic. Self-invoked methods may be used to perform recursive operations on a single node. In response to a method invocation, an object may change its state and send a message (*i.e.*, method invocation) along each of its edges or may produce a message into a global reduce operation.

Graph Operations: The graph evaluates as a series of synchronized steps. The evaluation model is a Receive-Update-Send sequence:
1. Graph nodes receive input messages.
2. Graph nodes wait for a barrier synchronization to proceed.
3. Graph nodes perform an update operation.
4. Graph nodes send output messages.

This evaluation sequence is the basis of semantic correctness and scaling. Graph node operations appear concurrent in that all nodes perform their update and exchange messages between synchronization events regardless of how they are sequentialized onto physical processing engines. Deterministic computation is guaranteed by forcing a step's set of messages to be received before performing each update. The GraphStep name was selected to emphasize this step-by-step operation.

Global Operations: A central controller can perform global broadcast and reduce operations on the graph or an activated subset of the nodes in the graph. The broadcast operations are effectively a designated method invocation on every node.

B. Relation to Other Concurrent System Architectures

The GraphStep architecture can be seen as a stylized restriction of the Bulk-Synchronous Parallel (BSP) model [27]. Like BSP, its semantics are based on a series of steps synchronized across the entire machine. The GraphStep architecture is more stylized in that it restricts the computational tasks to method updates on an object graph and emphasizes communication along object links, whereas BSP takes no stand on how communication occurs.

GraphStep can also be seen as a Data Parallel model in that operations are performed on a set of concurrent objects. The operations are not necessarily homogeneous actions applied to data because
- Nodes may be of different object types.
- Operations performed depend on the methods invoked, which may differ within a single operational step.
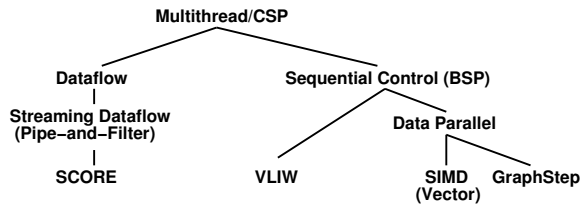
Fig. 1. Portion of Concurrent System Architecture Taxonomy Placing GraphStep

The SCORE architecture [28] also organizes computation as a graph of nodes. However, there is a fundamental difference between the semantics of the SCORE model and the GraphStep model in that SCORE is based on dataflow semantics, while GraphStep is based on lock-step sequential semantics. That is, SCORE nodes (operators or "filter" using the "pipe-and-filter" terms) synchronize only on the presence of data on their inputs, allowing some nodes to run ahead of other nodes as long as they have present data. In GraphStep all nodes are allowed to evaluate each step. In SCORE, a computation may wait for a set of inputs to occur, whereas in GraphStep, the node processes all the edges that have arrived on a cycle, even when this is only a subset of the potential inputs. One consequence of the dataflow semantics is that SCORE allows unbounded FIFOs on the edges (streams, pipes) between nodes, whereas GraphStep demands that all messages be delivered and consumed synchronously.

Philosophically, GraphStep is a data-centric concurrent system architecture and consequently takes a very different stand on how computation progresses than either SCORE or traditional, multithreaded computations. In GraphStep, the data contained in the graph nodes remains in a fixed location, and the computations are sent to the data. In SCORE, the graph is the computation and data is streamed through the graph. In a traditional processor organization, the computation runs on a processor and data is fetched from memory (possibly remote) in order for computation to proceed. Consequently, multithreaded, processor-oriented computations always involve a round-trip message pair to acquire data. Without careful latency-hiding hardware (*e.g.*, [29], [30]), the round-trip latency for data fetches can end up limiting exploitable data bandwidth and computational throughput. In contrast, GraphStep operations have a Continuation Passing Style (CPS) (*e.g.*, [31]) with execution always moving to the data.

Figure 1 shows a piece of the concurrent system architecture taxonomy, illustrating how GraphStep is related to the other architectures discussed in this section.

C. Possible Realizations

The concurrent system architecture defines the way the computation should be organized and expressed, as well as its semantics. While preserving the semantics, the architecture admits to a wide range of implementations. For example:

- **Fully Spatial** – The entire graph can be implemented spatially, with each node getting its own processing engine and with dedicated links between graph nodes.

The graph may be configured on top of one or more FPGAs (*e.g.*, [6], [7], [19]).

- **Sequential Processor** – The entire graph could be processed by a single processor, which picks up each node and executes it in turn. In this case, during data propagation steps, when no global operations are performed, the implementation may keep an active node set so it can avoid visiting nodes that have received no messages during the previous GraphStep send operation.

- **Multiprocessor** – The graph nodes can be distributed among the processors in a multiprocessor. Each processor is responsible for evaluating its nodes in sequence. This could even be realized using multiprocessor chips with local memory such as MIT's RAW [32] or IBM's Cell [33]. Processor-In-Memory (PIM) message passing processors would also allow us to exploit a close coupling of on-chip memory and data (*e.g.*, [34]–[36]).

- **Specialized Graph Processor** – It may be useful to build specialized processors designed to handle the typical operations involved in handling graph node messages. This could include integrated message handling (*e.g.*, [37], [38]).

- **Reconfigurable with Embedded Memories** – the graph nodes can be distributed among specialized graph processing engines configured on top of an FPGA with the nodes associated with each graph processing engine stored in on-chip, embedded memories (*e.g.*, Block RAMs; see Section VI-D).

- **Object-Specialized Graph Processing Engines** – When implementing the processing engines on an FPGA, we can assign graph nodes to processing engines by object type and specialize each processing engine to handle a single type of node object.

In practice, the fully spatial case is unlikely to be ideal when supporting graphs with thousands of nodes. In particular, the GraphStep model demands that we complete communication between phases. That means we must wait for the worst-case communication latency between nodes in the graph. If this latency is large (*e.g.*, hundreds of cycles) compared to the processing of a single message or node update (*e.g.*, 1–10 cycles), then a fully spatial implementation will spend all of its time waiting for messages to be routed. Consequently, sharing a processing engine among a modest number of graph nodes will better balance out the computation and communication latency. Effectively, this allows us to use substantially less hardware without increasing execution time; since the worst-case communication distance shrinks with the size of the physical hardware, up to a point, this may yield a net reduction in the time required for each GraphStep. Ultimately, node serialization will dominate communication latency and further serialization comes at the expense of slower computation.

VI. EXAMPLE: CONCEPTNET

As a concrete example, we consider an FPGA implementation of spreading activation on the ConceptNet Knowledge

Base [39] and compare this to a C-coded, sequential Pentium implementation.

A. Knowledge Base

ConceptNet is a knowledge base for commonsense reasoning compiled from a Web-based, collaborative effort to collect commonsense knowledge [39]. Nodes in the ConceptNet knowledge base are nouns and verb-noun pairs (*e.g.*, "run marathon"). Edges are distinguished by type to denote specific semantic relationships (*e.g.*, "effect of", "used for"). The knowledge base is used in natural language processing and commonsense reasoning tasks. Specific applications have included identifying contextual neighborhoods, topic gisting, analogy generation, predictions from sensor data, semantic prediction (projections), disambiguation, and affect sensing.

A "small" version of the ConceptNet knowledge base contains more than 14K nodes and 27K edges. The default ConceptNet knowledge base contains 220K nodes and 550K edges. There are 25 types of semantic relationships.

B. Spreading Activation

A key operation on the ConceptNet knowledge base is spreading activation. First, an initial set of graph nodes is chosen; these may be keywords, or portions of a natural language text. Depending on the application, each edge is given a weight coefficient based on its type. Starting with an activation potential of 1.0 for the initial nodes, activities are propagated through the network, stimulating related concepts. After a series of propagation steps, each node in the network will have an updated activity factor. Typically, nodes with the highest activity factors are then identified as being most relevant to the initial query. This calculation is similar to neural-network simulation; in spreading activation, the link weights vary based on the application in which ConceptNet is used and the specific query being performed.

Figure 2 sketches the spreading activation calculation. For actual implementation, this can be optimized while achieving the same semantics. Sequential implementations can take care to visit only nodes that receive at least one input message in a step. Since the update operation is associative, an implementation can directly sum the message into *step-activity* without waiting for the update phase; this avoids the need to make a full pass over the inputs during the update phase and avoids the need for space to store the full set of input activities in a step. To avoid buffering all the incoming messages, the send and receive phases can be overlapped.

C. Sequential Implementation

For baseline comparison, we implemented a streamlined version of spreading activation in C to run on standard microprocessors. The default ConceptNet graph requires >30MB to represent and, consequently, will not fit in the 1MB on-chip cache on Pentium processors. Even the smallest ConceptNet graph requires 1.5MB to represent.

To optimize the sequential implementation, we use an active graph node queue so that we need to visit only the nodes that

```
AUPDATE(v1,v2)
    tmax = max(v1,v2)
    tmin = min(v1,v2)
    return(tmax+(1-tmax)×tmin)
```

```
SPREADINGACTIVATION
//start with activities of non-initial nodes set to zero
foreach step
    foreach graph node g
        // receive
        foreach incoming message m
            g.edges[m.edge].activity←m.activity
        wait for step synchronization
        // update
        g.step-activity←0
        foreach input edge e to g
            g.step-activity ← AUpdate(g.step-activity,
                                       e.activity)
            g.node-activity ← AUpdate(g.node-activity,
                                       e.activity)
        // send
        foreach output edge e from g
            if (g.step-activity>THRESHOLD)
                send to e.sink with
                    activity=g.step-activity×g.discount
                            ×weight[e.type]
        // reset
        foreach input edge e to g
            e.activity←0
```

Fig. 2. Basic Computation for Spreading Activation

have new activity on each graph step. We also use an efficient radix sort data structure (similar to the one used in [40]) so we can extract the highest-activity nodes without walking the entire graph or paying $O(N \log(N))$ to perform the sort. Both insertion into the activity queue and replacement in the sort are $O(1)$ operations.

On a typical, modest query ("boy" "play" "park") on the default ConceptNet database, we allow activation to spread for three steps and visit 539,819 edges. Each edge visit takes about 700 cycles (around 200 ns) including one cache miss to main memory that accounts for roughly 300 of the 700 cycles. On average, this includes 12 L1 cache misses that are serviced by the L2 cache at 20 cycles apiece. All told, the query takes over 386,841,905 cycles, or about 113 ms. This query starts with three graph nodes activated, so the first few graph steps have moderate activity as activation spreads out from the initial nodes. Queries that start with many initial terms or high fanout nodes, as is typical in document processing tasks, will start with more of the graph active and consequently visit more nodes and require greater runtime (*e.g.*, the NYT query in Table II).

To collect data for the sequential implementation, we compiled the code with GCC 3.4.1 using the -O3 option and ran it on a 3.4 GHz Pentium-4 Xeon machine. We used the

TABLE II

COMPARISON OF QUERY EXECUTION TIMES ON SMALL CONCEPTNET DATABASE

Small ConceptNet		P4-3.4 GHz			XC2V6000							
					12 FPGAs (128 PEs)				64 FPGAs (512 PEs)			
Query	initial nodes	edges visited	% active	query time	edges visited	query time	speed up		edges visited	query time	speed up	
							total	per FPGA			total	per FPGA
"run marathon"	1	370	0.45	75 μs	81K	15 μs	5	0.42	81K	6.8 μs	11	0.27
"boy" "play" "park"	3	4600	5.6	0.99 ms	81K	15 μs	66	5.5	81K	6.8 μs	146	2.3
"person" "play" "dog" "park"	4	22K	28	3.4 ms	81K	15 μs	230	19	81K	6.8 μs	500	7.8
NYT-Abramoff article	109	23K	28	3.5 ms	81K	15 μs	230	19	81K	6.8 μs	510	8.0

TABLE III

COMPARISON OF QUERY EXECUTION TIMES ON DEFAULT CONCEPTNET DATABASE

Default ConceptNet		P4-3.4 GHz			XC2V6000 288 FPGAs (2048 PEs)			
							speed up	
Query	initial nodes	edges visited	% active	query time	edges visited	query time	total	per FPGA
"run marathon"	1	450K	27	94 ms	1.6M	33 μs	2800	9.9
"boy" "play" "park"	3	540K	32	110 ms	1.6M	33 μs	3300	12
"person" "play" "dog" "park"	4	920K	55	190 ms	1.6M	33 μs	5800	20
NYT-Abramoff article	109	930K	56	190 ms	1.6M	33 μs	5800	20

Pentium cycle counters to capture complete runtime. Separate non-timing runs were used to collect basic statistics on edges visited. Cache statistics were captured with the Pentium event counters using PAPI-3.2.1 [41], [42].

Tables II and III summarize the results from several, typical ConceptNet queries.

D. FPGA Implementation

For the FPGA implementation, we place graph nodes into Block RAMs and build a specialized processing engine for ConceptNet spreading activation, which is pipelined to handle one edge operation per cycle. Each such processing engine requires 320 Virtex-2 slices. We exploit the dual-port capabilities of the Block RAM to perform a read of the current graph node state, compute an activity update, and write back the graph node state in the edge-update pipeline. We connect graph-processing engines together with a packet-switched or time-multiplexed overlay network (*i.e.* Network-on-a-Chip—see [43]). The processing engine and network operate at 166 MHz (XC2V6000-4). To avoid serial bottlenecks on node processing, we decompose large nodes, those with high fanin or fanout, into a set of edge-limited nodes using fanin and fanout trees to preserve the original graph connectivity. To minimize network contention, we place graph nodes onto memory blocks to maximize locality using an efficient partitioner (UMpack's multi-level partitioner, `UCLA_MLPart5.2.14` [44]) similar to [5].

In the simplest case, we use a time-multiplexed network and process every graph node and every edge on every graph step. That is, we do not exploit activity sparseness. Note that since each edge update occurs in pipelined fashion, we spend two cycles processing each edge (one sending and one

receiving) for a total of 12 ns (XC2V6000-4) compared to the 200 ns per edge for the processor. Further, we get multiple processing engines per FPGA (*e.g.*, 32 on an XC2V6000), so we obtain two to three orders of magnitude higher edge-processing throughput on the FPGA than on the processor. Since the FPGA implementation processes every edge, it processes an order of magnitude more edges than the processor in modest queries like ("boy" "play" "park"); however, it takes no more time to process compound queries that start with more initial terms (see Tables II and III).

Each ConceptNet edge can be represented in 32b. Assuming we group together Block RAMs into sets which are powers of two, we use 128 of the 144 Block RAMS on the XC2V6000. This gives us $128 \times 512 = 64$K edges per XC2V6000. Consequently, it will take at least 16 leaf FPGAs to hold the default ConceptNet knowledge base.

Our FPGA performance numbers are calculated from a mapped implementation for the key elements (processing engine and network switches) and a cycle-accurate schedule of a graph step. We mapped our processing engine and network switches to an XC2V6000-4 and validated 166MHz operation. Datapaths are 16b wide to accommodate the activity value. On one XC2V6000, we get 32 processing engines using a Butterfly Fat Tree (BFT) interconnect structure (see Table IV). At the root of the leaf FPGAs, we have 8 input and 8 output channels. We use dedicated route FPGAs with 4 input and output downlinks and 2 input and output uplinks to continue to connect the leaf FPGAs up into a $p \approx 0.5$ BFT (see Figure 3 and Table V). Based on timing from this implementation (*e.g.*, cycles per switch, pipeline stages in the processing engine), we completely schedule computation and communication in a

TABLE IV

TABLE IV

BREAKDOWN OF LOGIC IN CONCEPTNET LEAF FPGA WITH 32 PES (XC2V6000)

Component	#	Slices Each	Total Slices	% Area
Processing Engines	32	320	10240	30%
Node Address Memory	460	12/node	5520	16%
	max graph nodes/PE			
BFT Switches			1920	6%
L1	16 π	48	768	
L2	16 T	24	384	
L3	8 π	48	384	
L4	8 T	24	192	
L5	4 π	48	192	
TM Memory	2112	7.5/cycle	15840	47%
	max cycles supported			
Total			33520	99%

TABLE V

MULTICHIP BFT COMPOSITION

| | | FPGAs | |
Total PEs	Compute Leaves	Tree Interconnect	Total
128	4	8	12
512	16	4×8+16=24	64
2048	64	4×48+32=224	228

single graph step for a given number of processors and network organization [43].

E. Discussion

As shown in Tables II and III, the reconfigurable implementation gets an order of magnitude speedup per FPGA compared to the processor solution for modest queries. We normalize speedup to the number of FPGAs to demonstrate that we get both parallelism speedup from using multiple components and speedup per component from the greater bandwidth identified in Section II. This shows that the FPGA solution has excellent scaling to tens and hundreds of FPGAs, whereas the processor version will not scale as nicely. For compound queries, the advantage per FPGA increases. For the simple queries with low activity, it may be possible to also exploit sparse activity using packet-switched interconnect to further reduce the FPGA runtime (see [43]).

VII. VARIATIONS AND FUTURE WORK

The applications outlined so far have all worked on static graphs. That is, we know the graph before the computation starts and the graph does not change during the computation. Further, since the graphs are known, we can place the tasks offline for spatial locality. Note that placement and routing are graph algorithms, so we expect to be able to use the same machine for placement and routing of the graph as we use to run the graph algorithms.

One generalization for future work is to efficiently support algorithms where the graph changes during the computation, that is, allow nodes and edges to be added and removed. In
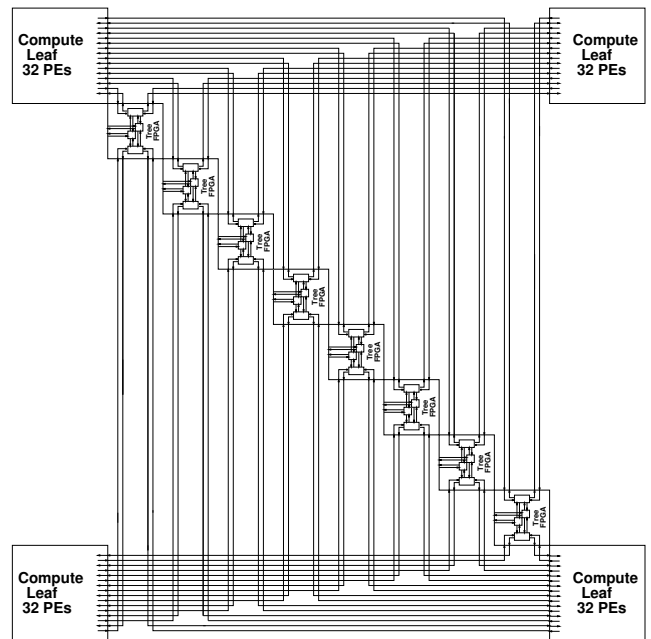

Fig. 3. BFT Network with 128 PEs in 12 FPGAs

addition to support for the new nodes, this will demand online placement of the new nodes and routing of the new links.

Many applications have mostly static graphs. That is, the graph may be large (millions of nodes and edges), but only a few edges are changed at a time. One example is a large knowledge base that filters out facts and adds new facts (nodes and edges) as it identifies them. Another example is a learning-based SAT solver (*e.g.*, [45]). In these SAT solvers, the learned clause database becomes large (hundreds of thousands to millions of entries); however, there will be many graph operations per conflict and each conflict adds only a few clauses to the database. Consequently, we are changing only a tiny fraction (maybe 0.001%) of the graph at a time.

As noted in Section VI, our primary comparison is to a static, time-multiplexed GraphStep implementation. For low activities, a dynamic version might be more efficient. Further, low activities and evolving graphs might motivate adaptive techniques for graph node placement, such as moving nodes based on dynamic activity to enhance locality and parallelism.

VIII. RELATED WORK

The idea of integrating computing with memory certainly is not new [15], [34]–[36], [46]–[48]. What is new is a suitable concurrent system architecture that organizes applications to exploit the parallelism and high memory bandwidth of these hardware architectures. As already noted in Section V-C many existing or proposed multiprocessor and PIM architectures could be useful implementation targets.

Other concurrent system architectures have explored logic and DRAM integration. Active Pages [48] was designed to support a data-parallel model that specifically did not efficiently handle interconnect between pages. Vector IRAM [49] supported a vector model, making it suitable for dense

applications, but not necessarily efficient for irregular, sparse-graph applications.

The GraphStep system architecture follows the vision of Hillis' Connection Machine (CM) [50]. The CM was an early herald of the data-parallel system architecture [51], and the first Connection Machines were SIMD implementations. As Figure 1 suggests, GraphStep is a refinement and restriction on the data parallel system architecture to more directly and efficiently support parallel graph algorithms.

IX. CONCLUSIONS

The high bandwidth and low latency available from the small, distributed, on-chip memories in modern FPGAs provide another opportunity for delivering high performance with field-programmable custom computing machines. This opens up the opportunity for these machines to accelerate a distinct and complementary class of applications to those which traditionally exploited the high computational throughput of FPGAs and reconfigurable architectures. We can capture many of these data-intensive applications with a sparse, graph-oriented concurrent system architecture. We show how we can use the GraphStep system architecture to exploit the high memory performance of FPGA-based machines to deliver performance that is orders of magnitude better on these memory-bound applications than that of microprocessors.

REFERENCES

[1] A. DeHon, "The Density Advantage of Configurable Computing," *IEEE Computer*, vol. 33, no. 4, pp. 41–49, April 2000.

[2] ——, "Reconfigurable Architectures for General-Purpose Computing," MIT Artificial Intelligence Laboratory, 545 Technology Sq., Cambridge, MA 02139, AI Technical Report 1586, October 1996. [Online]. Available: http://www.cs.caltech.edu/~andre/abstracts/dehon_phd.html

[3] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[4] S. A. McKee, "Reflections on the memory wall," in *Proceedings of Computing Frontiers*, April 2004.

[5] M. deLorimier and A. DeHon, "Floating-Point Sparse Matrix-Vector Multiply for FPGAs," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 2005, pp. 75–85.

[6] J. Babb, M. Frank, and A. Agarwal, "Solving graph problems with dynamic computational structures," in *Proceedings of SPIE: High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, vol. 2914, November 1996, pp. 225–236.

[7] O. Mencer, Z. Huang, and L. Huelsbergen, "Hagar: Efficient multicontext graph processors," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2002, pp. 915–924.

[8] E. Caspi, M. Chu, R. Huang, N. Weaver, J. Yeh, J. Wawrzynek, and A. DeHon, "Stream computations organized for reconfigurable execution (SCORE): Extended abstract," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, ser. LNCS. Springer-Verlag, August 28–30 2000, pp. 605–614.

[9] Intel Corporation, "Intel Pentium 4 processor product briefs," http://www.intel.com/design/Pentium4/prodbref/, December 2005.

[10] *Xilinx Virtex-II Platform FPGAs Data Sheet*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, October 2003, dS031 <http://direct.xilinx.com/bvdocs/publications/ds031.pdf> .

[11] *Xilinx Virtex-4 Family Ovreview*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, June 2005, dS112 <http://direct.xilinx.com/bvdocs/publications/ds112.pdf> .

[12] *Stratix II Device Handbook*, 4th ed., Altera Corporation, 2610 Orchard Parkway, San Jose, CA 95134-2020, December 2005.

[13] A. S. Huang and J. P. Shen, "A limit study of local memory requirements using value reuse profiles," in *Proceedings of MICRO-28*, December 1995, pp. 71–91.

[14] M. Motomura, Y. Aimoto, A. Shibayama, Y. Yabe, and M. Yamashina, "An embedded DRAM-FPGA chip with instantaneous logic reconfiguration," in *Digest of Technical Papers Symposium on VLSI Circuits*, 1997, pp. 55–56.

[15] S. Perissakis, Y. Joo, J. Ahn, A. DeHon, and J. Wawrzynek, "Embedded DRAM for a Reconfigurable Array," in *Proceedings of the 1999 Symposium on VLSI Circuits*, June 1999.

[16] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[17] L. McMurchie and C. Ebling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. ACM, February 1995, pp. 111–117.

[18] A. DeHon, R. Huang, and J. Wawrzynek, "Hardware-Assisted Fast Routing," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002, pp. 205–215.

[19] R. Huang, J. Wawrzynek, and A. DeHon, "Stochastic, Spatial Routing for Hypergraphs, Trees, and Meshes," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 2003, pp. 78–87.

[20] R. R.-F. Huang, "Hardware-Assisted Fast Routing for Runtime Reconfigurable Computing," Ph.D. dissertation, University of California at Berkeley, 2004.

[21] M. Wrighton and A. DeHon, "Hardware-Assisted Simulated Annealing with Application for Fast FPGA Placement," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 2003, pp. 33–42.

[22] M. Wrighton, "A Spatial Approach to FPGA Cell Placement by Simulated Annealing," Master's thesis, California Institute of Technology, June 2003. [Online]. Available: http://www.cs.caltech.edu/~wrighton/ms_thesis.doc

[23] S. E. Fahlman, *NETL: A System for Representing and Using Real-World Knowledge*. MIT Press, 1979.

[24] J.-T. Kim and D. I. Moldovan, "Classification and retrieval of knowledge on a parallel marker-passing architecture," *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, no. 5, pp. 753–761, October 1993.

[25] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[26] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the International Joint Conference on AI*, 1973.

[27] L. G. Valliant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, p. 103, August 1990.

[28] E. Caspi, M. Chu, R. Huang, N. Weaver, J. Yeh, J. Wawrzynek, and A. DeHon, "Stream Computations Organized for Reconfigurable Execution (SCORE): Introduction and Tutorial," <http://www.cs.berkeley.edu/projects/brass/documents/score_tutorial.html> , short version appears in FPL'2000 (LNCS 1896), 2000.

[29] Arvind and R. A. Ianucci, "Two fundamental issues in multiprocessing," in *Proceedings of DFVLR Conference on Parallel Processing in Science and Engineering*, West Germany, June 1987, pp. 61–88.

[30] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz, "Multi-processor performance on the tera mta," in *Proceedings of Supercomputing*, November 1998.

[31] A. Appel and T. Jim, "Continuation-passing, closure-passing style," in *Proceedings of the ACM Conference on Principles of Programming Languages*, 1989, pp. 293–302.

[32] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: Raw machines," *IEEE Micro*, vol. 30, no. 9, pp. 86–93, September 1997.

[33] D. C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. M. Harvey, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa, "Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor," *IEEE Journal of Solid State Circuits*, vol. 41, no. 1, pp. 179–196, January 2006.

[34] C. Lutz, S. Rabin, C. Seitz, and D. Speck, "Design of the mosaic element," in *Proceedings, Conference on Advanced Research in VLSI*, P. Penfield, Jr., Ed., Cambridge, MA, January 1984, pp. 1–10.

[35] W. J. Dally, S. J. A. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, P. R. Nuth, R. E. Davison, and G. A. Fyler, "The message-driven processor: A multicomputer processing node with efficient mechanisms," *IEEE Micro*, pp. 23–39, April 1992.

[36] T. Sunaga, H. Miyatake, K. Kitamura, P. M. Kogge, and E. Retter, "A processor in memory chip for massively parallel embedded applications," *IEEE Journal of Solid State Circuits*, vol. 31, no. 10, pp. 1556–1559, October 1996.

[37] D. S. Henry and C. F. Joerg, "A tightly-coupled processor-network interface," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[38] W. S. Lee, W. J. Dally, S. W. Keckler, N. P. Carter, and A. Chang, "An efficient, protected message interface," *IEEE Computer*, vol. 31, no. 11, pp. 69–75, November 1998.

[39] H. Liu and P. Singh, "ConceptNet – A Practical Commonsense Reasoning Tool-Kit," *BT Technical Journal*, vol. 22, no. 4, p. 211, October 2004.

[40] C. M. Fiduccia and R. M. Mattheyses, "A linear time heuristic for improving network partitions," in *Proceedings of the 19th Design Automation Conference*, 1982, pp. 175–181.

[41] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *The International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, 2000.

[42] PAPI Project, "Performance application programming interface," <http://icl.cs.utk.edu/papi/> , January 2006.

[43] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon, "Packet-switched vs. time-multiplexed FPGA overlay networks," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2006.

[44] A. Caldwell, A. Kahng, and I. Markov, "Improved Algorithms for Hypergraph Bipartitioning," in *Proceedings of the Asia and South Pacific Design Automation Conference*, January 2000, pp. 661–666.

[45] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," in *Proceedings of the International Conference on Computer-Aided Design*, 2001, pp. 279–285.

[46] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM: IRAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, Mar/Apr 1997.

[47] N. Margolus, "An FPGA architecture for DRAM-based systolic computations," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1997, pp. 2–11.

[48] M. Oskin, F. T. Chong, and T. Sherwood, "Active pages: a model of computation for intelligent memory," in *Proceedings of the International Symposium on Computer Architecture*, June 1998.

[49] C. Kozyrakis and D. Patterson, "Vector vs superscalar and VLIW architectures for embedded multimedia benchmarks," in *Proceedings of the International Symposium on Microarchitecture*, 2002, pp. 283–293.

[50] W. D. Hillis, *The Connection Machine*. MIT Press, 1985.

[51] W. D. Hillis and G. L. Steele, "Data parallel algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, December 1986.

APPENDIX

This query set had 109 keywords (initially activiated nodes) extracted from a New York Times cover article about Jack Abramoff. Terms: "the", "jack", "count", "today", "end", "testify", "tax", "set", "begin", "use", "business", "colleague", "charge", "include", "effort", "wednesday", "expect", "connection", "purchase", "boat", "line", "talk", "speak", "picture", "associate", "prepare", "reduce", "exchange", "knowledge", "action", "element", "can", "serve", "prison", "time", "night", "follow", "communication", "office", "lawyer", "prosecution", "florida", "come", "schedule", "stand", "edge", "close", "house", "leader", "earn", "dollar", "represent", "interest", "island", "help", "funnel", "friend", "power", "trip", "develop", "work", "suspect", "justice", "leadership", "believe", "take", "meal", "downtown", "restaurant", "press", "secretary", "delay", "agreement", "year", "put", "reach", "hi", "person", "involve", "hear", "cut", "take place", "stage", "witness", "face", "evidence", "focus", "location", "sentence", "place", "week", "trial", "word", "tie", "member", "list", "core", "form", "pressure", "case", "deal", "in prison", "public", "final", "separate", "more", "free", "key", "last", "own". This kind of query is typical of those used to identify the topic of a piece of text.