Spatial Hardware Implementation for Sparse Graph Algorithms in GraphStep

MICHAEL DELORIMIER, NACHIKET KAPRE, NIKIL MEHTA, and ANDRÉ DEHON,
University of Pennsylvania

How do we develop programs that are easy to express, easy to reason about, and able to achieve high performance on massively parallel machines? To address this problem, we introduce GraphStep, a domain-specific compute model that captures algorithms that act on static, irregular, sparse graphs. In GraphStep, algorithms are expressed directly without requiring the programmer to explicitly manage parallel synchronization, operation ordering, placement, or scheduling details. Problems in the sparse graph domain are usually highly concurrent and communicate along graph edges. Exposing concurrency and communication structure allows scheduling of parallel operations and management of communication that is necessary for performance on a spatial computer. We study the performance of a semantic network application, a shortest-path application, and a max-flow/min-cut application. We introduce a language syntax for GraphStep applications. The total speedup over sequential versions of the applications studied ranges from a factor of 19 to a factor of 15,000. Spatially-aware graph optimizations (e.g., node decomposition, placement and route scheduling) delivered speedups from 3 to 30 times over a spatially-oblivious mapping.

Categories and Subject Descriptors: D.2.11 [**Software Architecture**]: Domain-specific architectures; D.1.3 [**Concurrent Programming**]: Parallel programming; B.7.1 [**Types and Design Styles**]: VLSI (very large scale integration)

General Terms: Languages, Algorithms, Performance

Additional Key Words and Phrases: Spatial computing, compute model, parallel programming, graph algorithm, graphStep

ACM Reference Format:
Delorimier, M., Kapre, N., Mehta, N., and Dehon, A. 2011. Spatial hardware implementation for sparse graph algorithms in GraphStep. ACM Trans. Autonom. Adapt. Syst. 6, 3, Article 17 (September 2011), 20 pages.
DOI = 10.1145/2019583.2019584 http://doi.acm.org/10.1145/2019583.2019584

1. INTRODUCTION

Managing spatial locality is essential to extracting high performance from modern and future integrated circuits. Technology scaling is giving us more transistors, higher cross-chip communication latency relative to operation latency, and fewer cross-chip wires relative to transistors. The first effect means we have more parallelism to exploit. The second two mean that communication optimizations are primary and are essential concerns that must be addressed to exploit the potential parallelism. Communication latency can dominate the critical path of the computation and interconnect throughput can be the performance bottleneck. By carefully selecting the location of operators and

Authors' addresses: M. Delorimier (corresponding author), N. Kapre, N. Mehta, and A. Dehon, Department of Electrical and System Engineering, University of Pennsylvania, Room 203 Moore Building, 200 South 33rd St., Philadelphia, PA 19104; email: Michael@delorimier.org.

DOI 10.1145/2019583.2019584 http://doi.acm.org/10.1145/2019583.2019584

data in space, we can exploit parallelism effectively by minimizing signal latency and message traffic volume.

We introduce the GraphStep compute model [deLorimier et al. 2006]. The set of applications that GraphStep captures are those that are centered on large, static, sparse graphs. Typically the application iterates over steps where operations are performed on graph nodes and data is sent along edges. Often it is convenient and efficient to perform graph-wide reductions and broadcasts. We draw applications from domains such as semantic networks, CAD optimization, numerical computations, and physical simulations. The applications we test are queries on ConceptNet, a semantic network, circuit retiming, which uses a shortest-path algorithm, and the max-flow/min-cut kernel for vision tasks.

By using the domain-specific GraphStep model we can map high-level, machine-independent programs to spatially optimized implementations. In this domain, the graph structure captures the communication and computation structure that allows us to optimize for spatial locality. The domain-specific model abstracts out race conditions, synchronization details, operation and data placement, and operation scheduling.

We model the performance of GraphStep applications mapped to FPGA logic. FPGA hardware provides a highly parallel, spatial computing platform with the flexibility to support high communication bandwidth, high memory bandwidth, and low synchronization overhead. The logic architecture placed on the FPGAs is a collection of Processing Elements (PEs) interconnected with a Fat-Tree Network [Leiserson 1985] (Figure 4). Each PE has its own memory and compute logic.

The contributions of this work include the following.

(1) We define a concrete programming language for GraphStep (Section 2.3), illustrate it (Figure 2), and give its formal semantics (Electronic Appendix accessible in the ACM Digital Library).
(2) We quantify the benefit of spatially aware optimizations enabled by the GraphStep model, which are graph node decomposition, placement for locality, and static computation and communication scheduling (Section 5).
(3) We quantify the benefit of a spatial implementation compared to a sequential implementation (Section 6).

In Section 2 we explain the GraphStep model and compare it to other parallel compute models. Section 3 gives example GraphStep applications. Section 4 describes the hardware implementation. Section 5 describes and evaluates the optimizations performed on our example applications. Section 6 compares our applications' performances in the GraphStep model to equivalent algorithms implemented sequentially. Section 7 discusses future work. Section 8 concludes. The Electronic Appendix, accessible in the ACM Digital Library, summarizes the formal semantics for GraphStep.

2. GRAPHSTEP MODEL

GraphStep is designed to express algorithms that work on sparse graphs. The computation structure follows the graph structure, so changes made to node state propagate changes along edges to neighboring nodes. This parallel activity is sequenced into steps, with one set of synchronous node updates and propagations per step. In general, a subset of nodes are updated in each step. A subset of the updated nodes then propagate changes to their neighbors. A sequential process initiates and controls parallel activity on the graph. It broadcasts to nodes and receives global reductions from nodes.

An operation on a node or edge generates messages that trigger operations on neighboring nodes and edges. The static graph structure is a directed multigraph, so nodes send messages to their outgoing edges, and edges send messages to their destination nodes. The atomic action of an operation is to: (1) input incoming message state along

with the object state, (2) update object state, and (3) output new messages. The invocation of an operation is called an *operation firing*.

To match the common iterative structure of graph algorithms, this message passing activity is divided into *graph steps*. A graph step consists of three phases.

(1) *Reduce*. Each node performs a reduction on the incoming messages received along its input edges. The reduction should be associative and commutative.
(2) *Update and send*. Each node with a reduction result updates its state and outputs messages to its output edges. The update operation may also output a message to a global reduction.
(3) *Edge*. Each edge with an input message processes it, possibly updates its state, and outputs a message to its destination node to be processed in the next graph step.

At most one operation occurs on each edge and at most one update operation occurs on each node. A global barrier before each node update guarantees that each update sees a consistent set of messages regardless of the number of PEs in the machine or implementation delays. When multiple messages are pending to a node, they are processed as a single reduction operation that is managed by a single locus of control at the graph node. The programmer is responsible for making the reduction operator associative and commutative so their order of reduction is unimportant. Together, this means there is no dependence on the order of messages, race conditions cannot occur, and the result of each graph step is fully deterministic. As a result, the programmer need not reason about relative timing of operations.

A sequential controller broadcasts messages to nodes to initiate the iteration. The broadcast value is fed to a node update operator at each receiving node. Graph steps may continue until the computation has quiesced and no messages are generated. For example, a graph relaxation usually only generates messages upon changes, so upon convergence there are no more messages (e.g., Bellman-Ford [Cormen et al. 1990]). Alternatively, the sequential component of the algorithm may decide when to end the iteration. For example, Conjugate Gradients [Hestenes and Stiefel 1952] uses a global reduce to decide when the error is small enough to stop.

2.1. Enabled Optimizations

In order to take advantage of GraphStep on a spatial implementation (e.g., FPGA or multicore processors) we must minimize communication work and latency and load balance memory, computation, and communication to fit into small, distributed processing elements. To do this we use the exposed graph communication structure and exploit the associativity and commutativity properties of reduce operations.

2.1.1. Node Decomposition. Load-balancing nodes into PEs must be performed to minimize the memory area per PE and minimize the computational work per PE. Decreasing PE memory area decreases cross-chip communication latency by decreasing the latency of communication across each PE. Often nodes with large numbers of neighbors prevent the computational load from being spread evenly across large numbers of PEs. Associativity and commutativity of reduction operations allows us to decompose a large node and distribute it across multiple processing elements (Figure 1). Node decomposition transforms a node with a large input-arity to a fanin tree of reduce operators followed by the state-holding root node. A node with a large number of outputs is decomposed into a root with a fanout tree to fanout messages. Note that knowledge of the structure of the graph is required to make connections from fanout tree leaves to fanin tree leaves.

2.1.2. Placement for Locality. The static graph structure is used to maximize the locality of neighboring nodes. The number of neighboring nodes placed into the same PE is

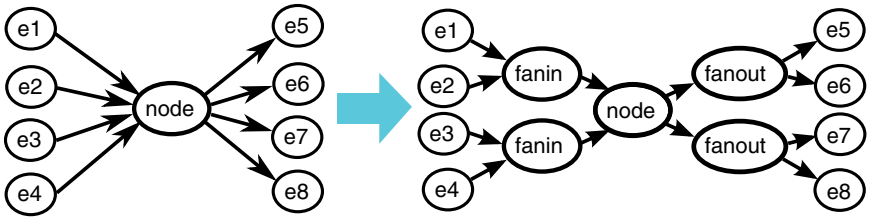


Fig. 1. Node decomposition.

maximized, and the distance between neighboring nodes in different PEs is minimized. This minimizes the volume of message traffic between processing elements. Since local communication is fast compared to cross-chip communication, it also minimizes each graph step's critical path latency due to message passing.

2.1.3. Static Scheduling. The static graph structure is used to preschedule operation firings and message routes. When the graph is loaded, a static schedule is computed for a single graph step where each node and edge is active. Each PE and network switch inputs the predetermined firing and routing choices from a dedicated memory. These time-multiplexed memories store a data-independent, VLIW instruction sequence that is evaluated once per graph step.

Without knowledge of the static graph, a dynamic schedule must be computed online by: (1) a packet-switched network to route messages and (2) extra PE control logic to fire operations. We find that static scheduling is typically more efficient than dynamic scheduling in terms of hardware area and time (Section 5). The dynamically scheduled case uses more hardware area than the statically scheduled case due to the high cost of packet-switched interconnect switches. Further, the static router performs offline, global routing to minimize network congestion. The static scheduler also combines the compute and communicate phases of each GraphStep.

2.1.4. Hardware. We can also specialize hardware to the GraphStep model. Node and edge operators and node and edge memories can be pipelined so each operator fires at the rate of one edge per cycle. To feed the operator pipeline, messages must be input and output at the rate of one per cycle. Lightweight message handling is enabled by the GraphStep model since logic need not perform message ordering or buffer resizing.

Global broadcasts and reduces could be a significant source of latency since they must cross the entire machine. We map them to dedicated binary tree interconnect to eliminate potential congestion with other messages and eliminate latency due to interconnect switches.

2.2. Compute Model Comparison

This section explains how GraphStep differs from related concurrency models. GraphStep is high level in its domain, which reduces the detail the programmer must specify and manage. The compiler and runtime use the exposed communication structure to optimize for a spatial implementation.

Actors. In actors languages (e.g., Act1 [Lieberman 1987] and ACTORS [Agha 1998]), computation is performed with concurrently active objects that communicate via message passing. All computation is reduced to atomic operations that mutate local object state and are triggered by and produce messages. Similar to actors, GraphStep has the aforesaid restrictions. The primary difference is that actors programs are low-level descriptions of any concurrent computation pattern on objects, rather than a high-level description of a particular domain. The communication structure is, in general, dynamic and hence not visible to the compiler.

Streaming. Streaming, persistent data flow languages have a static or mostly static graph of operators that are connected by streams (e.g., Kahn Networks [Kahn 1974], SCORE [Caspi et al. 2000], Ptolemy [Lee 2005], Synchronous Data Flow [Lee and Messerschmitt 1987], Brook [Brook Project 2004], Click [Shah et al. 2004]). These are used for high-performance applications such as packet switching and filtering, signal processing, and real-time control. Like GraphStep, streaming data flow languages are often high level, domain specific, and the spatial structure of a program can be used by a compiler. The primary difference is that in streaming computations the persistent nodes are operators given by the program, whereas in GraphStep the persistent nodes are data objects given by input to the program. For GraphStep the global graph steps free the implementation from the need to track an unbounded length sequence of tokens on each channel.

Data Parallel. Data parallelism [Blelloch et al. 1993; Koelbel et al. 1994; Hillis 1985; Dean and Ghemawat 2004] is a simple way to orchestrate parallel activity. A thread applies an operation in parallel to the elements of a collection. The operation may be applied to each element independently (map). It may also be a reduction or parallel-prefix operation (reduce) [Hillis and Steele 1986; Dean and Ghemawat 2004].

Machines that are entirely SIMD or have SIMD leaves [Hillis 1985; Habata et al. 2003; Lindholm et al. 2008] are an important target for data-parallel languages. Like GraphStep, data-parallel programs can be very efficient since they map well to SIMD hardware. However, they do not typically describe operations on irregular data structures efficiently and do not expose the communication structure of the application to the compiler.

Bulk Synchronous Parallel. BSP is an abstract model of parallel computers [Valiant 1990]. Programs written with a BSP library or language use barriers to synchronize between processors. Processors input messages from the last barrier-synchronized step and output messages to the next barrier-synchronized step. Unlike GraphStep, BSP programs do not expose the communication structure to the compiler.

2.3. GraphStep Syntax

We have developed a high-level language for expressing GraphStep computations. This high-level language does not automatically compile to hardware yet. Our algorithms are currently expressed in a slightly lower-level language that we expect will be an easy mapping target from this high-level language. Figure 2 shows Bellman-Ford in our syntax. The syntax was chosen to be similar to Java [Microsystems 1995] when possible.

GraphStep distinguishes three kinds of classes for the graph processing domain.

—A `node` class describes the data and operations located at a graph node.
—An `edge` class describes the same for graph edges.
—The `glob` class describes the global controller's data and procedures. There is only one global controller object at runtime so there is only one `glob` class.

In the example, there is one `node` class named `N` and one `edge` class named `E`.

Atomic data types include {`boolean`, `int`, `unsigned`, `float`, `double`}. Additionally, a tuple type may be made of previously defined data types. Tuples allow finitely nested types. Method arguments, return values, variables, and data fields all have data types.

Since objects are not first class, a class's fields are separated into pointer fields and data fields. A pointer field (e.g., `edges` in class `N`) will point to a fixed object, if labeled with the attribute `single`, and to a static set of objects if labeled with `set`. Pointer fields declare the class of the object pointed to. Data fields (e.g., `distance` in class `N`) are read and written by methods and declare the data type.

```
glob Graph {

  set N nodes; // nodes is a set of objects of type N
  single N source; // source is a pointer to one object of type N

  boolean bellman_ford() {
    unsigned nnodes = nodes.size();
    source.min_edge(0);
    // iterate until convergence (no nodes updated distance
    //                            in the last iteration)
    //            or iter==nodes.size() in which case there
    //                                  is a negative cycle
    unsigned iter;
    boolean active = true;
    for (iter = 0 ; active && iter < nnodes ; iter++) {
      // step is a primitive function which initiates one graph step and
      // returns true iff there are pending operation fires
      active = step();
    }
    // return true iff there is no negative cycle
    return iter < nnodes;
  }
}

node N {
  set E edges;
  float distance;

  // operate on two messages, bound to distance1 and distance2 respectively
  reduce tree min_edge (float distance1) (float distance2) {
    if (distance1 < distance2) return distance1;
    else return distance2;
  }
  update min_edge (float newdist) {
    if (newdist < distance) {
      distance = newdist;
      edges.propagate(distance);
    }
  }
}

edge E {
  single N to;
  float length;
  fwd propagate (float distance) {
    to.min_edge(distance + length);
  }
}
```

Fig. 2. Bellman-Ford code.

Designated method kinds in the `node` and `edge` classes support the send-receive-update phases described earlier.

—A `fwd` method in an `edge` class is invoked during the send phase. Each `fwd` method (e.g., `propagate` in `E`) receives a message from an `update` method in its source `node`, performs local read and write operations, then sends a message to a `reduce` or

`update` method in its sink `node`. The message send is a dispatch on a pointer field and method name.

—A `reduce tree` method in a `node` class (e.g., `reduce tree min_edge`) is used to reduce all the incoming messages to a node into a single message following the barrier at the end of the send phase. A `reduce tree` method literally describes a binary reduction which is composed in a binary tree to reduce all incoming messages to one message. A `reduce tree` method may not mutate any of the node state.

—An `update` method in a `node` class (e.g., `update min_edge`) either inputs the result of the reduction of the same name or inputs a singular message from the global controller or an `edge`. An `update` that does not have a corresponding reduce declared receives singular messages. The `update` method may mutate local data fields and send messages.

Nodes are constrained to send to edges or the `glob` object, and each edge is constrained to send to one node. These constraints on the pointer structure are enforced by the compiler.

Control is performed by sequential procedures defined in the `glob` class, for example `bellman_ford` in `Graph`. `glob` procedures may contain branch and loop statements. They may call other `glob` procedures, possibly recursively, or issue commands to orchestrate parallel activity. To initiate parallel activity, a `glob` procedure broadcasts a message to a pointer field's nodes via a message dispatch (e.g., `source.min_edge(0)`). The built-in `step` command instructs the parallel computation to advance on a graph step. `step` returns `true` if-and-only-if there are currently messages pending. `reduce tree` methods in the `glob` class are used for global reductions. The built-in `step_reduce` control operator is used to advance a graph step and return the result of the global reduce.

3. APPLICATIONS

GraphStep is designed to conveniently support a large variety of graph algorithms. In this section, we review several broad classes of graph algorithms and highlight how GraphStep supports their structure. We also call out prior work spatial implementations of these graph algorithms and the three algorithms used in this article.

3.1. Iterative Numerical Methods

These are frequently used for solving linear equations, finding eigenvalues, and numerical optimization. In the examples Conjugate Gradients, Lanczos, and Gauss-Jacobi, a common compressed sparse row representation for the matrix uses one node to represent a row of the matrix and one edge to represent a nonzero of the matrix. Each matrix-vector multiply is performed by one graph step, and each global dot-product is performed by a global reduce. The spatial implementation of sparse matrix-vector multiply from deLorimier and DeHon [2005] achieved a speedup of an order of magnitude over highly tuned sequential implementations. Similarly, a spatial implementation of a direct matrix solver for SPICE achieved an order of magnitude speedup over sequential implementations [Kapre and DeHon 2009].

3.2. Graph Relaxation Algorithms

Algorithms in this subdomain are composed of relaxation operations on directed edges. A relaxation operation on an edge updates its destination node's state based on its source node's state. If the destination node's state computed by the relaxation is different than its current state then its state changes. Every time the state of a node changes its out edges must relax. When there are no remaining relaxations, node states have reached a fixed point and the algorithm is finished.

If relaxations are ordered improperly then there could be an exponential number of relaxation operations compared to the optimal timing. Synchronizing relaxations into

graph steps results in the same number of graph steps as the maximum length of the chain of edge relaxation dependencies in the optimal ordering. Problems which may use relaxation algorithms include shortest-path searches, depth-first search tree construction, strongly connected component identification [Chandy and Misra 1982], global optimizations on program graphs [Kildall 1973], max-flow/min-cut calculations, and constraint propagation in combinatorial problems like CNF SAT [Logemann et al. 1962].

3.2.1. Bellman-Ford. Bellman-Ford (Figure 2) is a shortest-paths algorithm [Cormen et al. 1990]. Each edge in the graph has a length, possibly negative. It finds the shortest path from a designated source node to all nodes, or detects the presence of a negative cycle. Each iteration takes a set of nodes whose distance from the source was updated on the previous iteration. Each out edge from the updated nodes is relaxed, which means that the shortest path so-far through the updated node is checked against the shortest previously computed path to its destination node. If the new path is shorter, then the node updates its distance. The iteration continues until it quiesces, or until it detects a negative cycle.

We test Bellman-Ford as the kernel of register retiming of a circuit to find the minimum cycle time [Leiserson et al. 1983].

3.2.2. Preflow-Push. Preflow-Push (sometimes called Push-Relabel) uses interactions between neighboring nodes to find the maximum flow and minimum cut on a graph from a single source to a single sink [Cormen et al. 1990]. Preflow-Push is a relatively more complex relaxation algorithm that propagates updates through a graph. Unlike Bellman-Ford it always converges to a solution. It uses two basic operation types on nodes and whether an operation can be applied to a node depends on its neighbors' states. The GraphStep algorithm cycles through eight types of graph steps with different operations on nodes and edges in each. We optimized our implementation by using fractional flows. This optimization increases the fraction of useful work performed in each graph step which results in fewer total graph steps.

We test Preflow-Push as the kernel of stereo vision problems [Boykov et al. 1998; Kolmogorov and Zabih 2001].

3.3. CAD Algorithms

CAD algorithms typically perform NP-hard optimizations on a circuit graph. Multilevel partitioning algorithms cluster nodes and iteratively reassociate them with partitions [Karypis and Kumar 1999]. Iterative placement algorithms move nodes to reduce cost functions with a random element to avoid local minima [Wrighton and DeHon 2003]. Parallel routing may perform shortest-path reachability searches on the circuit graph [DeHon et al. 2006]. The just mentioned placer and router are spatial implementations that act directly on the circuit graph and show orders of magnitude speedup over state-of-the-art sequential processor implementations. These were designed and implemented by hand, whereas GraphStep versions would automate much of the implementation work. Furthermore, the router can use Bellman-Ford as its shortest-path kernel.

3.4. Semantic Networks, Knowledge Bases and Databases

When these are represented as graphs, knowledge-base queries and inferences take the form of parallel graph algorithms, including marker passing [Fahlman 1979; Kim and Moldovan 1993], subgraph isomorphism, subgraph replacement, and spreading activation (e.g., ConceptNet [Liu and Singh 2004]).

3.4.1. ConceptNet. ConceptNet is a knowledge base for common-sense reasoning compiled from a Web-based, collaborative effort to collect common-sense knowledge [Liu and Singh 2004]. Nodes are concepts and edges are relations between concepts, each

labeled with a relation-type. Spreading activation is a key operation for ConceptNet. Edges are given weights depending on their relation type. An initial set of nodes is chosen and each is given an activity of 1.0. Activities are propagated through the network, stimulating related concepts. After a fixed number of iterations, nodes with high activities are identified as the most relevant to the query.

4. IMPLEMENTATION

On a modern FPGA, the Virtex 6, we can perform a memory operation in less than 3ns, a 16-bit add in less than 3ns, and send a signal across the distance of 2 processing elements (PEs) in the same 3ns cycle. However, we can place 512 PEs on today's largest Virtex 6, meaning it takes over an order of magnitude longer (24 times) to communicate across the chip than to perform a local operation. Further Moore's Law scaling will allow us to place more PEs on a chip while maintaining fairly comparable relative delays such that cross-chip communication will easily be two or more orders of magnitude greater delay than a local operation. These ratios of compute and communicate latency mean the location of computations matter.

We use specific area and timing costs from the Virtex 6, but this general trend where cross-chip communication latencies exceed local computation costs by orders of magnitude will be true of all high-performance silicon computations. To generate logic for the Virtex 6 we use Synplify Pro 9.6.1 for synthesis and Xilinx ISE 10.1 for placement and routing.

4.1. Processing Element

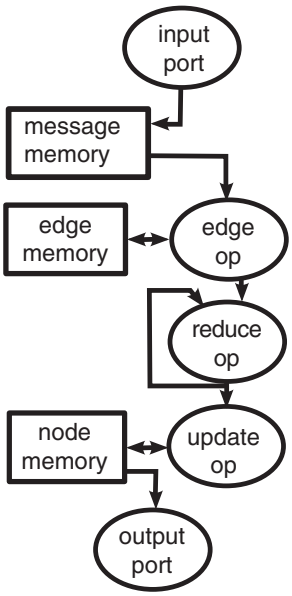
PEs are designed to achieve high throughput data transfer between operator logic and application memory and between the PE and the interconnect. In a spatial implementation, memory is local to logic to enable high memory bandwidth and low latency. The communication-centric approach requires a high message input and output rate. Typically a graph has many more edges than nodes so we provide dedicated node, edge, and message memories that allow the PE pipeline to read and write one edge per cycle (Figure 3(a)). The PE pipeline steps required for each edge are:

(1) output a message from the source PE;
(2) input the message at the destination PE;
(3) perform the edge operation; and
(4) perform one binary reduce operation.

Operators are pipelined to perform one operation per 3ns cycle. Memories are dual ported to remove structural hazards between operations. On the output side, the node memory has one read per output edge. On the input side, the message memory has one read and one write per input edge and the edge memory has one read and one write per input edge. The node update operator fires only once per node so we allow it to share a memory port with edge sends.

The specialized datapath can be contrasted to a PE using an instruction set processor and a single local memory. Figure 3(b) shows the program for such a PE. We assume the sequential PE can also perform the edge, reduce, and node operator operations in a single cycle. Each edge requires 11 sequential cycles as indicated in the figure.

For an efficient implementation, the number of PEs must be large enough so that memory area is comparable to logic area. We report area measured in terms of Virtex 6 slices. Each Virtex 6 slice contains four 6-input Lookup Tables (6-LUTs). Application memories are implemented with BlockRAMs. In the Virtex 6, each BlockRAM provides 18Kb of memory. There is one BlockRAM for every 83 slices. Table II shows the area used due to network components, PE logic, and memory for the ConceptNet-default application (Table I). PE area includes operators and controller logic, memory for the



(a) Spatial Processing Element Datapath

```
// body executed per node
for i = 0 to nnodes
  emin = edgeoff[i]
  emax = edgeoff[i+1]
  r = reduce_id[i]
  j = emin
  // loop performs edge op, reduce op
1 while j < emax
2   m = input_message[j]
3   e = edge[j]
4   (f, n) = edge_op(e, m)
5   edge[j] = f
6   r = reduce_op(r, n)
7   j++
  // node update op
  a = node[i]
  (b, z) = update_op(a, r)
  node[i] = b
  j = emin
  // fanout output to out edges
8 while j < emax
9   sa = send_addr[j]
10  output_message[sa] = z
11  j++
```

(b) A sequential PE program that processes all input messages on one graph step and produces all output messages for the next graph step. The 11 instructions required per edge are numbered.

Fig. 3. PE logic operation.

Table I. Characteristics for Benchmark Graphs Used with Sample Applications

Application	Input	Nodes	Edges	Max In Arity	Max Out Arity
ConceptNet	small	14556	27275	226	2538
	default	224876	553836	16175	36562
Bellman-Ford	tseng	1048	3760	122	445
	ex5p	1065	4002	63	721
	pdc	4576	17193	40	1499
	s38584.1	6448	20840	304	2989
	s38417	6407	21344	106	661
	clma	8384	30462	82	5453
Preflow-Push	BVZ-tsukuba10_8	45273	143592	5408	5408
	BVZ-tsukuba10_4	90055	285220	9961	9961
	BVZ-tsukuba10_2	185388	591552	24425	24425

application, and memory for the static schedule. Since BlockRAMs and slices are separate hardware, the total PE area in slices is

$$PE_{area} = \max(83 \times N_{BlockRAMs}, N_{slices}). \quad (1)$$

Including interconnect, the area is 611 slices per PE. At this size, a signal can cross 2 PEs per 3ns cycle.

4.2. Interconnect

The interconnect topology is designed to fit a two-dimensional spatial layout. PEs are laid out in a grid, and connected with a Butterfly Fat-Tree (BFT) interconnect topology [Leiserson 1985]. Although a mesh topology would also correspond to two

Table II. Area Model for ConceptNet with the Default Graph Using Static Hardware with 2048 PEs

Component	Slices Each	Number	Slices
Total			1250K
Network total			411K
switch logic	28	992	28K
switch context memory	342	992	339K
channels			44K
PE total	410	2048	840K
logic	183	2048	375K
application memory	410	2048	840K
context memory	208	2048	426K

Area is measured in terms of Virtex 6 slices (see text).

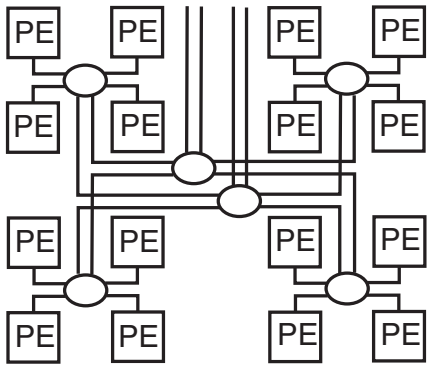


Fig. 4. Two stages of the BFT with 16 PEs and 4 channels up to the next stage.

dimensions, the BFT is simpler to route. The BFT is constructed recursively, where the top-level switches of a tree with n PEs connect the top-level switches of four $n/4$ PE subtrees (Figure 4). The number of switches connecting two subtrees increases with the level of the tree in order to accommodate a larger number of cut graph edges between the two subtrees. The Rent parameter of the BFT, p, relates the number of PEs, n, in a subtree to the number of channels out of the subtree: $io\_channels = n^p$ [Landman and Russo 1971]. To fit the two-dimensional hardware, we set $p = 0.5$, so the number of channels out of an area scales with its perimeter. When $p = 0.5$, switches take a constant fraction of total area, with one switch for every two PEs [DeHon 2000]. Furthermore, the maximum number of PEs a signal crosses in a subtree with n PEs is $8\sqrt{n}$ PEs, which is proportional to the diameter of the subtree. The number of switches in the path is $\log_2 n$. Table II shows the area due to interconnect along with the area due to switch logic, memory for the static schedule, and the channels connecting switches.

5. IMPACT OF SPATIALLY AWARE OPTIMIZATIONS

This section uses our spatial hardware model to evaluate the benefit of the optimizations highlighted in Section 2.1. We evaluate the benefit of node decomposition and placement for locality and compare the static scheduling option to dynamic scheduling.

5.1. Optimization Types

The baseline implementation places nodes of the original graph into PEs with the objective of maximizing the load balance between PEs. The load balancer takes the weight of a node to be the maximum of its input-arity and output-arity; since the PE

processes one edge per cycle (Section 4.1), this is the approximate number of cycles required to evaluate the node for each graph step.

The first optimization (Section 2.1.1) decomposes graph nodes with large input-arity or output-arity (Figure 1). Decomposition reduces the size of the largest nodes to allow scaling to a large number of small PEs. Fanin-tree and fanout-tree topology is chosen to minimize depth while bounding the arity to 64.

The second optimization (Section 2.1.2) uses the static graph structure to place neighboring nodes in the same PE or in nearby PEs in order to minimize total message volume and minimize the critical path of a graph step. At the top level of the BFT, graph nodes are partitioned into the two subtrees. The bipartition is chosen to minimize the number of cut edges between the two partitions while satisfying a load-balance constraint. Bipartitioning is applied recursively until nodes are placed into the leaf PEs. We use the UMpack's multilevel partitioner available from UCLA's MLPart5.2.14 [Caldwell et al. 2000].

The third optimization (Section 2.1.3) preschedules operation firings and message routes. Prescheduling removes the need for complex hardware and improves the quality of the schedule. At graph load time the static scheduler first computes the schedule then loads the schedule into switch and PE context memories. The unoptimized, dynamically scheduled implementation uses a packet-switched network to route messages as they are generated.

5.2. Optimization Results

Figure 5 shows the cycles used relative to the baseline implementation for the example applications with just decomposition applied (decomposed), decomposition and placement for locality applied (local), and decomposition, placement for locality, and static scheduling applied (static). The number of PEs used for each application and each optimization is chosen to minimize the total number of cycles with a maximum of 2048 PEs. Figure 6 shows an example of this selection for Bellman-Ford-clma. Table III reports the number of PEs chosen for each application.

Figure 5 shows that the best combination of all three optimizations achieves speedups between 3 times (BVZ-tsukuba10_8) and 30 times (ConceptNet-default). Decomposition alone achieves a speedup of 15 times for ConceptNet-default. Placement for locality provides an additional speedup of 2 for BVZ-tsukuba10_2. Static scheduling provides an additional speedup of 2.7 times for Bellman-Ford-clma.

5.2.1. Node Decomposition. Large nodes can prevent us from balancing the computational load on the PEs due to fragmentation. The computation time for each node is proportional to its input-arity plus output-arity. The largest node imposes a lower bound on the cycles required for a graph step. Figure 7 shows the result of decomposition on the distribution of node arities for the ConceptNet-default graph (Table I). Before decomposition, the largest node has an arity of 52,737, where the sum over all node arities is twice the number of edges: 1,107,672. Since the largest node in this case is about 1/20th the weight of all nodes, speedup is limited when we scale above 20 PEs (see Figure 6). After decomposition the largest node has an input- and output-arity of 64.

The benefit of decomposition increases as graph size increases because large decomposed graphs can efficiently utilize more PEs than small decomposed graphs. The benchmark graphs in Figure 5 are ordered by edge count from left to right for each application. For example, the speedup from decomposition for Bellman-Ford increases from 1.6 times for the tseng to 10 times for clma.

Decomposition is often required to keep PEs small. The memory required for each PE must be large enough to satisfy the memory requirement for the largest node. This, in turn, sets the lower bound on the area per PE (see Eq. (1)). A larger area per

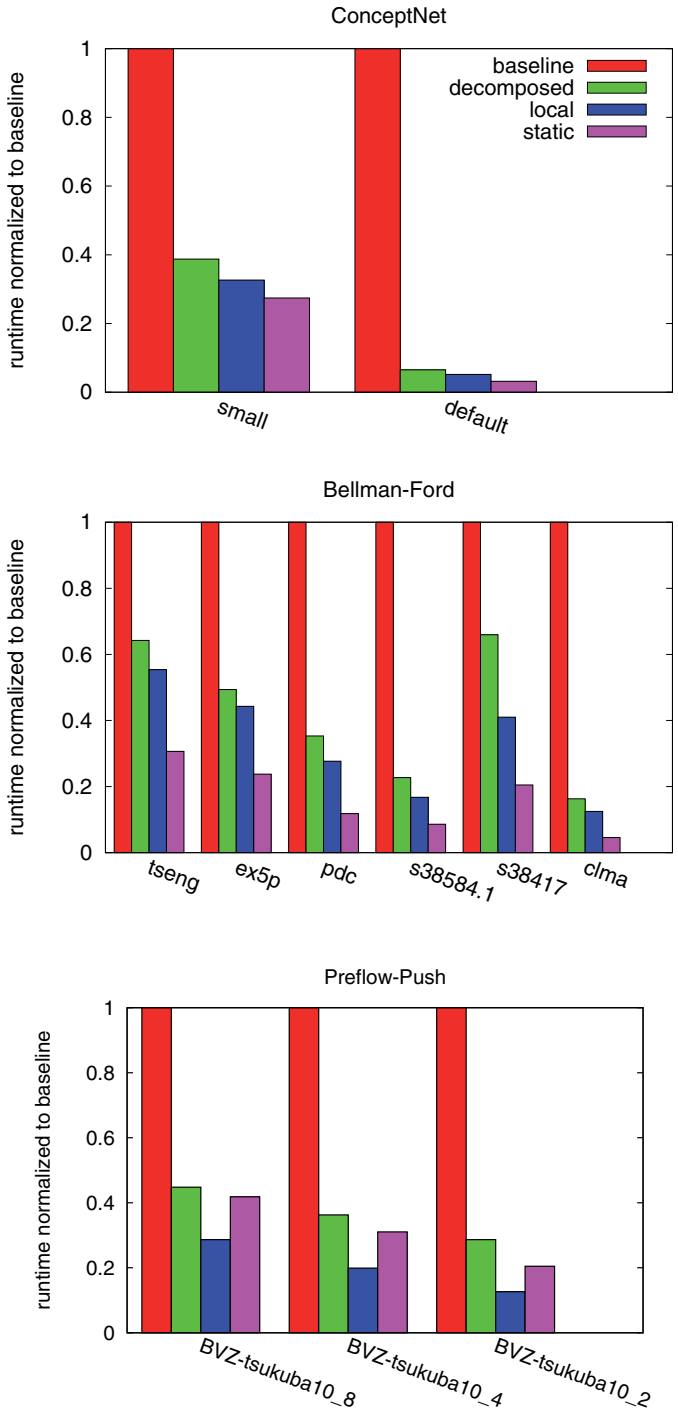


Fig. 5. Time of optimized implementations relative to the baseline for each application, benchmark graph, and optimization.

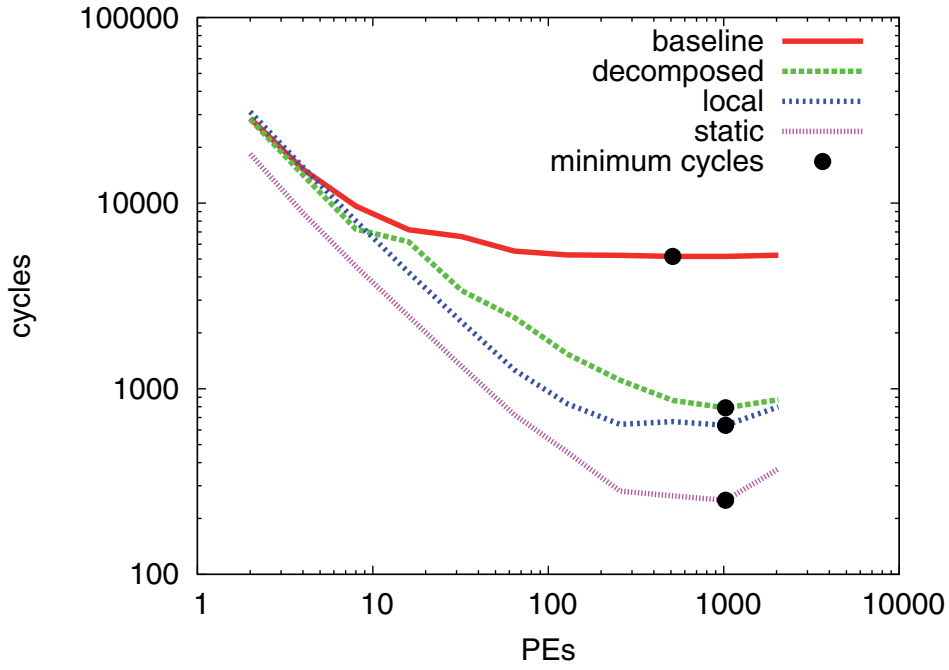


Fig. 6. Number of PEs and cycles for Bellman-Ford-clma.

Table III. Number of PEs Used by Each Application, Graph and Optimization

Application	Graph	Baseline	Decomposed	Local	Static
ConceptNet	small	128	256	256	512
	default	256	2048	2048	2048
Bellman-Ford	tseng	256	128	128	128
	ex5p	256	256	128	256
	pdc	1024	1024	1024	512
	s38584.1	1024	1024	512	256
	s38417	1024	1024	512	2048
	clma	512	1024	1024	1024
Preflow-Push	BVZ-tsukuba10_8	2048	1024	2048	2048
	BVZ-tsukuba10_4	2048	1024	2048	2048
	BVZ-tsukuba10_2	1024	2048	2048	2048

PE increases computation time due to a higher chip-crossing latency. For ConceptNet-default decomposition reduces slices per PE by a factor of 4.5. Since chip-crossing latency scales with the square root of the area per PE this decreases message latency by about a factor of two for ConceptNet-default.

5.2.2. Placement for Locality. The primary effect of placement for locality is to decrease the message traffic. This can be seen in Figure 8 which compares for ConceptNet-default the lower bound imposed by message traffic, the lower bound imposed by chip-crossing latency, and the total cycles required for a graph step. The load balanced case which ignores locality is labeled *not local* and placement for locality is labeled *local*. Decomposition and static scheduling are performed for both placement types. Whereas Figure 5 compares the benefit of locality for the dynamically scheduled case, static scheduling is used here so we can calculate reasonable lower bounds. Here we see that the locality-placed design significantly reduces the minimum cycles required

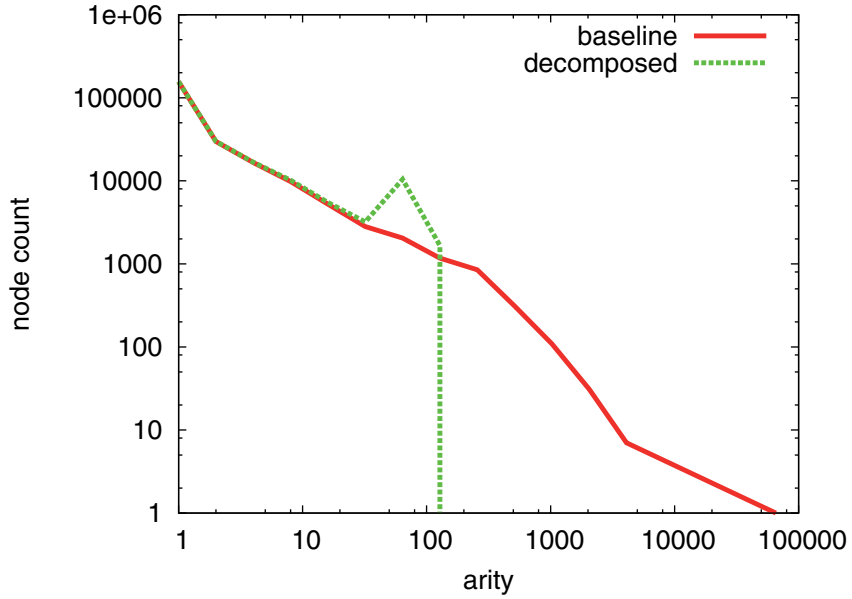


Fig. 7. Distribution of node arities (input-arity plus output-arity) for the baseline case and the decomposed case for ConceptNet-default.

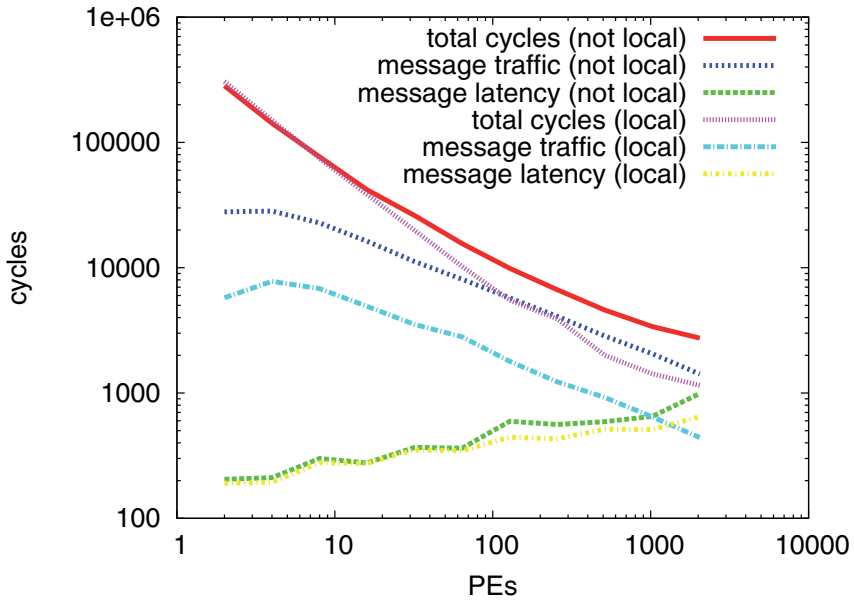


Fig. 8. Comparison between placement for locality and load balancing ignoring locality for ConceptNet-default. Both cases use the static and decomposition optimizations.

Table IV. Area Model for ConceptNet with the Default Graph with Dynamic Hardware with 2048 PEs

Component	Slices Each	Number	Slices
Total			2199K
Network total			1109K
logic-switch	974	992	966K
channels			143K
PE total	532	2048	1090K
logic	183	2048	375K
app mem	532	2048	1090K

Area is measured in terms of Virtex 6 slices (see Section 4).

Table V. Activity Factors for Each Application and Graph

Application	Graph	Activity
ConceptNet	small	0.11
	default	0.25
Bellman-Ford	tseng	0.77
	ex5p	0.83
	pdc	0.81
	s38584.1	0.82
	s38417	0.71
	clma	0.83
Preflow-Push	BVZ-tsukuba10_8	0.16
	BVZ-tsukuba10_4	0.16
	BVZ-tsukuba10_2	0.18

for communications by avoiding the message traffic bottleneck to get a speedup of over 2 times at 2048 PEs. Figure 8 further shows that performance at high PE counts is limited by communication latency, and this latency is improved by 1.5 times by placement for locality. As noted in Section 7, we believe this latency can be reduced further, allowing an additional locality speedup of 2 times.

5.2.3. Static Scheduling. Figure 5 shows that static scheduling improves performance for ConceptNet and Bellman-Ford [Kapre et al. 2006]. This benefit comes from: (1) the static scheduler can compute a higher-quality route than the dynamic scheduler given the same set of messages, (2) the static scheduler can combine the compute and communicate phases of each graph step eliminating the latency of one global barrier, and (3) static hardware typically has lower area which decreases the chip-crossing latency.

Table IV reports areas for the dynamically scheduled hardware components for the ConceptNet-default application. It shows that the primary difference between dynamic and static hardware areas (Table II) is due to interconnect switch size. Figure 9 shows the Virtex 6 slices per PE for each application and optimization. The statically scheduled hardware area is about half the dynamic area. This area savings is particularly important when cross-chip latency is limiting performance as illustrated in Figure 8. As before, the number of PEs was chosen to minimize the total number of cycles.

However, static scheduling decreases performance for the Preflow-Push graphs tested. Table V shows the average fraction of edges activated over graph steps. For Preflow-Push the low activation allows the dynamically scheduled implementation to perform 16% to 18% of the operations the statically scheduled implementation performs. For the Virtex 6 target, we can select between the statically scheduled and packet-switched implementations on a per-application basis.

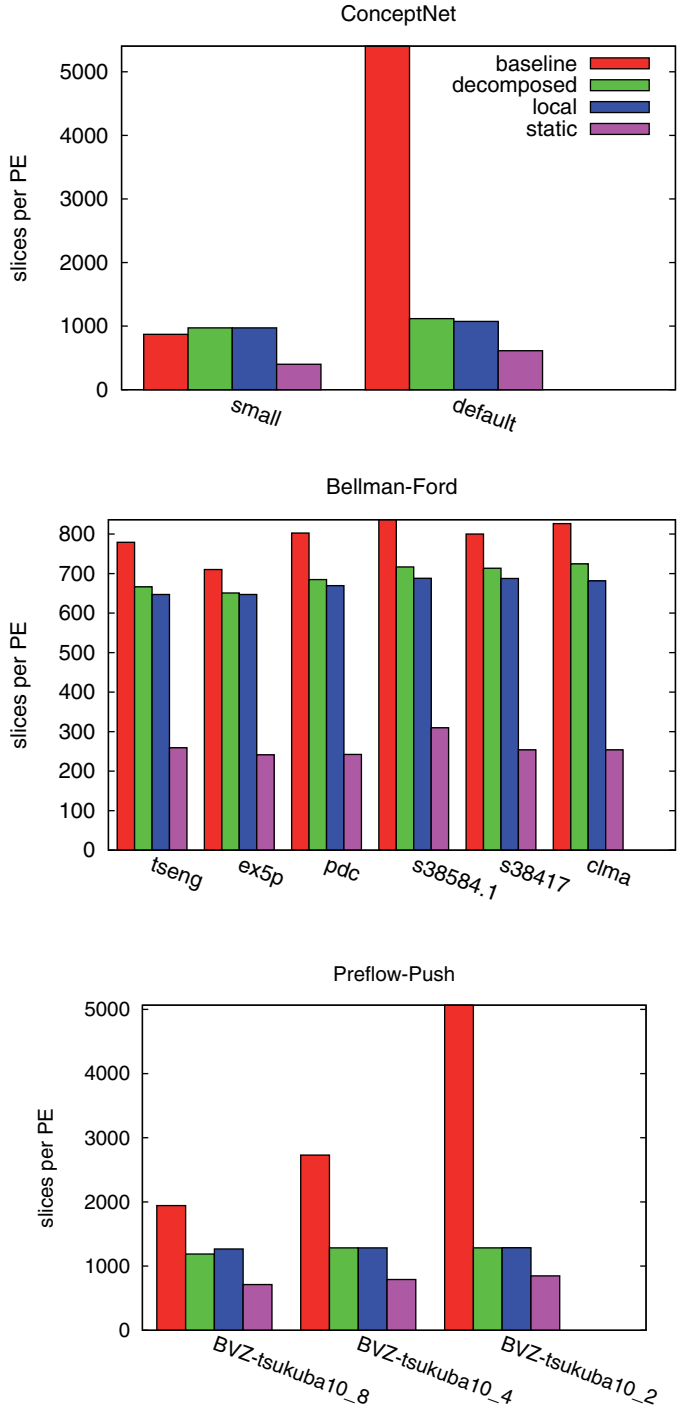


Fig. 9. Area in Virtex 6 slices of a PE for each application, benchmark graph, and optimization.

Table VI. Sequential and GraphStep Runtimes for Each Application and Graph

Application	Graph	Sequential Time	GraphStep Time	Speedup
ConceptNet	small	10ms	7.2μs	1389
	default	490ms	31μs	15806
Bellman-Ford	tseng	72ms	3.8ms	19
	ex5p	68ms	3.3ms	21
	pdc	1.6s	13ms	123
	s38584.1	2.8s	29ms	97
	s38417	3.2s	21ms	152
	clma	6.9s	44ms	157
Preflow-Push	BVZ-tsukuba10_8	20s	0.19s	105
	BVZ-tsukuba10_4	100s	0.45s	222
	BVZ-tsukuba10_2	623s	1.9s	328

6. COMPARISON TO SEQUENTIAL PERFORMANCE

To evaluate the benefit of using GraphStep on spatial hardware, we compare its runtime to sequential implementations of the GraphStep algorithms. Table VI shows the results of the total runtime for the best GraphStep implementation and a sequential implementation of each application studied. The Preflow-Push GraphStep implementation used here is dynamically scheduled. ConceptNet-default performs the most favorably with a speedup over 15,000. The larger Bellman-Ford graphs reach a two orders of magnitude speedup. The largest Preflow-Push graph also reaches a two orders of magnitude speedup.

The sequential programs were run on a 3GHz Xeon. ConceptNet is implemented in C and compiled with gcc 4.3.2 using the `-O3` option. It uses an active node queue to perform only the necessary updates on each iteration. Register retiming and its Bellman-Ford kernel are implemented in Ocaml and compiled with ocamlopt 3.10.2 using the `-unsafe` and `-inline 2` options. Since activity for our Bellman-Ford graphs is close to 1 (Table V), the implementation iterates over all nodes in the graph on each step. Preflow-Push is implemented in Ocaml and compiled with ocamlopt 3.10.2 using the `-unsafe` and `-inline 4` options. Each step of the outer iteration is analogous to a graph step and uses two queues to keep track of the active nodes. It uses efficient array-based queues with O(1) time per push or pop operation.

7. FUTURE WORK

For many of our applications, scaling is limited by critical path latency. That is, the latency of a chain of messages from a source root node, through a fanout tree, through a fanin tree, to a sink root node may impose a lower bound on cycles per graph step (see Figure 8). Our BFT and partitioning-based placer are not fully exploiting spatial locality. A placer that directly attempts to minimize the critical-path may help reduce critical path latency. Using a mesh or BFT with shortcuts may also help reduce latency. On a mesh, the distance a signal must travel from one PE to another is the Manhattan distance between the PEs. Our BFT often requires an integer factor more latency than the Manhattan distance between PEs. Further, our placement of fanin and fanout nodes results in a critical path with 2 to 4 routes through the top level of the BFT. Avoiding the need for multiple crossover routes should reduce the latency by, at least, a factor of two.

In the dynamically scheduled case, concurrently activated computational work may be unbalanced across the PEs. Neighboring nodes may be simultaneously activated while more distant nodes are idle. Consequently, a locality-enhancing placement could produce a poor dynamic load balance. For example, a wavefront of activity can impact a small subset of PEs at any one point in time. In future work, it would be good to quantify the impact of dynamic computation load imbalance and, if it is significant,

develop placement algorithms that can minimize this imbalance. This may demand dynamic reoptimization and placement.

One direction to extend the applicability of GraphStep is to support graph algorithms that add and remove nodes and edges dynamically but otherwise fit the GraphStep model. Example dynamic applications include finite element methods with mesh refinement [Castanos and Savage 2000] and SAT with clause learning [Marques-Silva and Sakallah 1999]. These evolving graphs will also demand more dynamic support for node placement.

8. CONCLUSION

To continue to turn the additional transistors provided by technology scaling into performance, we must exploit parallelism. Effective exploitation of this parallelism demands careful management of the location of computations so that fragmentation, communication latency, and bandwidth requirements do not undermine the benefits of parallelism. Knowing the communication structure of a computation, we can automatically select the location of computations to minimize these costs and achieve efficient spatial implementations. We demonstrate automated, spatially aware optimizations that improve performance up to 30 times. These spatial implementations can be orders of magnitude faster than sequential implementations. Our GraphStep model captures this domain and exposes the communication structure to enable spatial optimizations without placing the burden of locality management on the programmer.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

REFERENCES

AGHA, G. 1998. *ACTORS: A model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA.

BLELLOCH, G. E., CHATTERJEE, S., HARDWICK, J. C., SIPELSTEIN, J., AND ZAGHA, M. 1993. Implementation of a portable nested data-parallel language. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 102–111.

BOYKOV, Y., VEKSLER, O., AND ZABIH, R. 1998. Markov random fields with efficient approximations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 648–655.

BROOK PROJECT. 2004. Brook project web page. http://brook.sourceforge.net.

CALDWELL, A., KAHNG, A., AND MARKOV, I. 2000. Improved algorithms for hypergraph bipartitioning. In *Proceedings of the Asia and South Pacific Design Automation Conference*. 661–666.

CASPI, E., CHU, M., HUANG, R., WEAVER, N., YEH, J., WAWRZYNEK, J., AND DEHON, A. 2000. Stream computations organized for reconfigurable execution (SCORE): Extended abstract. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*. Lecture Notes in Computer Science. Springer, 605–614.

CASTANOS, J. AND SAVAGE, J. 2000. Repartitioning unstructured adaptive meshes. In *Proceedings of the Parallel and Distributed Processing Symposium*. IEEE, 823–832.

CHANDY, K. M. AND MISRA, J. 1982. Distributed computation on graphs: Shortest path algorithms. *Comm. ACM 25,* 11, 833–837.

CORMEN, T., LEISERSON, C., AND RIVEST, R. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

DEAN, J. AND GHEMAWAT, S. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Symposium on Operating System Design and Implementation*. 137–150.

DEHON, A. 2000. Compact, multilayer layout for butterfly fat-tree. In *Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA'00)*. ACM, 206–215.

DEHON, A., HUANG, R., AND WAWRZYNEK, J. 2006. Stochastic spatial routing for reconfigurable networks. *J. Microprocess. Microsyst. 30,* 6, 301–318.

DELORIMIER, M. AND DEHON, A. 2005. Floating-point sparse matrix-vector multiply for FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 75–85.

deLorimier, M., Kapre, N., Mehta, N., Rizzo, D., Eslick, I., Rubin, R., Uribe, T. E., Knight, Jr., T. F., and DeHon, A. 2006. GraphStep: A system architecture for sparse-graph algorithms. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 143–151.

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

Habata, S., Yokokawa, M., and Kitawaki, S. 2003. The earth simulator system. *NEC Res. & Develop. 44,* 1, 21–26.

Hestenes, M. R. and Stiefel, E. 1952. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand. 49,* 6, 409–436.

Hillis, W. D. 1985. *The Connection Machine*. MIT Press, Cambridge, MA.

Hillis, W. D. and Steele, G. L. 1986. Data parallel algorithms. *Comm. ACM 29,* 12, 1170–1183.

Kahn, G. 1974. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP CONGRESS 74*. North-Holland Publishing Company, 471–475.

Kapre, N. and DeHon, A. 2009. Parallelizing sparse matrix solve for SPICE circuit simulation using FPGAs. In *Proceedings of the International Conference on Field-Programmable Technology*. IEEE, 190–198.

Kapre, N., Mehta, N., deLorimier, M., Rubin, R., Barnor, H., Wilson, M. J., Wrighton, M., and DeHon, A. 2006. Packet-Switched vs. time-multiplexed FPGA overlay networks. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 205–213.

Karypis, G. and Kumar, V. 1999. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20,* 1.

Kildall, G. A. 1973. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'73)*. ACM Press, New York, 194–206.

Kim, J.-T. and Moldovan, D. I. 1993. Classification and retrieval of knowledge on a parallel marker-passing architecture. *IEEE Trans. Knowl. Data Engin. 5,* 5, 753–761.

Koelbel, C. H., Loveman, D. B., Schreiber, R. S., Guy L. Steele, J., and Zosel, M. E. 1994. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA.

Kolmogorov, V. and Zabih, R. 2001. Computing visual correspondence with occlusions using graph cuts. In *Proceedings of the IEEE International Conference on Computer Vision*. Vol. 2. 508–515.

Landman, B. S. and Russo, R. L. 1971. On pin versus block relationship for partitions of logic circuits. *IEEE Trans. Comput. 20*, 1469–1479.

Lee, E. 2005. UC Berkley ptolemy project. http://www.ptolemy.eecs.berkeley.edu/.

Lee, E. A. and Messerschmitt, D. G. 1987. Synchronous data flow. *Proc. IEEE 75,* 9, 1235–1245.

Leiserson, C., Rose, F., and Saxe, J. 1983. Optimizing synchronous circuitry by retiming. In *Proceedings of the 3rd Caltech Conference On VLSI*.

Leiserson, C. E. 1985. Fat-Trees: Universal networks for hardware efficient supercomputing. *IEEE Trans. Comput. C-34,* 10, 892–901.

Lieberman, H. 1987. *Concurrent Object-Oriented Programming in Act 1*. MIT Press, Cambridge, MA.

Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. 2008. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro 28,* 2, 39–55.

Liu, H. and Singh, P. 2004. Conceptnet – A practical commonsense reasoning tool-kit. *BT Tech. J. 22,* 4, 211.

Logemann, G., Loveland, D., and Davis, M. 1962. A machine program for theorem proving. *Comm. ACM 5,* 7, 394–397.

Marques-Silva, J. P. and Sakallah, K. A. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Comput. 48,* 5, 506–521.

Microsystems, S. 1995. The java language environment. White paper. http://java.sun.com/docs/white/langenv/.

Pierce, B. C. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA.

Shah, N., Plishker, W., Ravindran, K., and Keutzer, K. 2004. NP-Click: A productive software development approach for network processors. *IEEE Micro 24,* 5, 45–54.

Valiant, L. G. 1990. A bridging model for parallel computation. *Comm. ACM 33,* 8, 103–111.

Wrighton, M. and DeHon, A. 2003. Hardware-assisted simulated annealing with application for fast FPGA placement. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 33–42.

Received August 2009; revised April 2010; accepted June 2010