Galois for FPGA
An Efficient Microarchitecture and Compiler for Accelerating Amorphous Data Parallel Graph Applications

A Dissertation Proposal by

Dan Zhang

20 April 2015

Submitted to the graduate faculty of the

Department of Electrical and Computer Engineering

at The University of Texas at Austin

in partial fulfillment of the requirements

for the Dissertation Proposal and

subsequent Ph.D. in Computer Architecture and Embedded Processors

Outline

Abstract

Although the behavior of irregular algorithms can be very complex, many of them have a generalized form of data parallelism known as *amorphous data parallelism.* Through the use of unordered atomic transactions and dynamic worklist-based scheduling, the Galois system has been demonstrated to be a practical approach to exploiting this form of parallelism. I propose Galois for FPGA, an efficient custom microarchitecture and compiler to run Galois programs at higher performance and power efficiency than traditional general-purpose architectures.

1 Introduction

The pervasiveness of multicore processors have shifted the burden of improving program execution speed from chip manufacturers to software developers. Data parallelism often occurs in *regular* programs, which manipulate dense matrices and arrays. Many language constructs have been proposed to enable programmers to easily express regular data-parallel application in their apps, such as parallel-for in OpenMP [8], thread blocks in CUDA [15], and work-groups in OpenCL [31].

However, *irregular* programs are much harder to parallelize. Irregular programs use pointer-based data structures such as lists, trees, and graphs, and are common outside of computational science. Examples can be found in domains like n-body simulation [2], data mining [32], maxflow [6], social networks [12], and meshing [30]. Recent research efforts have made considerable headway, demonstrating that data-driven parallel algorithms are algorithmically efficient and support higher performance than topology-driven implementations on both general purpose processors [21] [33] as well as GPUs [4] [20].

One system for data-driven applications is Galois [27], which comprises the Galois worklist-based data-driven programming model, the unordered transactional **foreach** iterator, a set of concurrent library components, and the Galois software runtime system. These applications can benefit significantly from dedicated worklist accelerators on both general purpose multicore processors [17] and GPUs [14]. However, as fixed logic, these accelerators cannot be easily adapted to the different scheduling requirements of each algorithm, potentially resulting in lost performance [11].

1.1 Thesis Proposal

For my thesis, I propose the Galois system for FPGA, consisting of a microarchitecture designed to accelerate Galois applications, a set of hardware accelerators matching the Galois software library components, and a compiler to emit a custom pipelined engine from Galois software code. By automatically generating custom hardware from a Galois software specification, Galois for FPGA can achieve both higher performance and energy efficiency compared to Galois running on general purpose hardware.

1.2 The Galois System

In [27], Pingali et al. introduce the notion of operator formulation, in which an algorithm is viewed in terms of its action on data structures. The authors demonstrate that operator formulation is both general [26] and effectively exposes the fine-grained amorphous data-parallelism present in irregular

```
Graph graph = { /* init */}; // Initialize graph contents
WorkList<GraphNode> workQ();
workQ.enq(init); // worklist initialized with the starting node
foreach(uint nid : workQ) {
    uint nid = workQ.deq();
    GraphNode node = graph.getNode(nid);
    for(int i = 0; i < node.numEdges; i++) {
        Edge e = graph.getEdge(node.edgePtr + i);
        uint newDist = node.dist + edge.weight;
        if(newDist < graph.getNode(edge.dest).dist) {
            node.dist = newDist;
            workQ.enq(edge.dest);
        }
    }
}
```

Figure 1: Galois pseudocode for SSSP

algorithms [22]. They propose the Galois programming model, which is a sequential, object-oriented model that is augmented with the Galois set operator **foreach(e in Set S) {B(e)}**, in which the loop body *B(e)* is executed for each element *e* of set *S*. Set *S* may be either partially ordered or unordered. However, unlike a regular **for** loop, the loop ordering is treated as a hint simply to improve performance. It is the programmer's responsibility to write Galois code in a way that ordering does not affect algorithm correctness. Each loop iteration may conflict with one another. The Galois runtime automatically generates locking mechanisms to ensure atomicity is not violated. When an iteration executes, it may add additional elements to set *S*. In Galois, set *S* is referred to as the ***worklist***.

The Galois system provides a concurrent data structure library, including worklists and graphs. Each data structure has several variants, each with several parameters; it's up to the programmer to choose the best variant and parameters to fit the algorithm and input. However, with a properly designed algorithm, the variant chosen only affects performance, not correctness.

1.2.1 SSSP on Galois

As a simple example, consider Single-Source Shortest Path (SSSP), the problem of finding the shortest path from a single source node to all connected nodes. SSSP is a well-known and long-studied problem with many practical applications on a wide spectrum of graph types. The traditional serial approach to SSSP is Dijkstra's Algorithm [16], which uses a priority queue to process the node with the shortest cumulative distance from the single source node first. While this is an efficient serial algorithm *(O(v log v + e))*, it is poorly suited for multithreading. Bellman-Ford [9] trades off work efficiency *(O(ev))* for parallelism by processing every edge connection in the graph until the distances converge. A more flexible solution is Delta-stepping [19], which is a hybrid of Dijkstra's and Bellman-Ford. Rather than considering the vertex with the lowest distance like Dijkstra's, delta-stepping groups vertices into buckets. Within each bucket, vertices are unordered and thus can be executed in parallel. If the bucket size is 1, then delta-stepping is identical to Dijkstra's algorithm. If the bucket size is infinite, then delta-stepping is the same as Bellman-Ford. Thus, by choosing a proper bucket size, or delta, we can make the proper tradeoff between available parallelism and work efficiency. Note that the appropriate bucket size not only changes based on the graph input but also on the host architecture running the algorithm. Galois source code for SSSP is shown in Figure 1. If the worklist *workQ* is using strict priority order, then the source code is an implementation of Dijkstra's Algorithm. Else, the source code depicts delta-stepping.

1.2.2 Galois vs. other programming models

Galois contains a high-level programming model; thus it can be implemented within any parallel framework such as CUDA and OpenCL. The Galois authors have implemented Galois on both Java and C++, and are working towards implementing Galois for GPU using CUDA [4]. Davidson et al. [7] demonstrated the potential value of Galois for GPU by implementing a SSSP delta-stepping within CUDA using worklists, claiming 14x to 340x higher performance compared to a Bellman-Ford implementation depending on the input. However, the authors had to construct custom Galois-like worklists and synchronization mechanisms on top of CUDA. As this was a custom solution designed to run only SSSP, these modules must be reimplemented to effectively target a different application. Considerable work must be done to build up the synchronization, scheduling, work-balancing, and work spilling features of a Galois worklist.

There are many other programming frameworks targeting graph applications, such as PowerGraph [10], Ligra [29], and GraphGen [24]. These frameworks are all *vertex-centric*, meaning that computation is expressed in terms of each vertex and its neighbors. The schedule is simple: iterate across all vertices in the graph until the algorithm converges. Although this programming model is useful and easy to understand, vertex-centric models can be inefficent since even nodes without any work must be executed every cycle. In addition, since a vertex only has information about its immediate neighbors, information is propogated slowly, one hop at a time. For example, Tian et al. [33] demonstrated that a graph-centric model of a connected components algorithm ran 63x faster and used 204x fewer network messages than its vertex-centric counterpart. Its important to note that its possible to write vertex-centric programs within Galois [22]; however, Galois provides the user the flexibility to design more efficient graph-centric algorithms.

1.3 Task Scheduling

One common parallel programming approach is to decompose a program into parallel tasks and allow an underlying software layer to schedule these tasks to different threads. Examples of popular multithreaded task-based runtime systems include Cilk [3], OpenMP [8], and CUDA [15]. These systems all implement sophisticated forms of load balancing, including techniques such as work stealing and chunking. However, researchers have demonstrated that many algorithms also benefit significantly proper task prioritization due to faster convergence, leading to a drastic net reduction in work [7] [28]. State-of-the-art runtime systems like that in Galois are able to take advantage of this prioritization to achieve massive speedups [11].

Despite software tasked-based runtime systems already achieving a high degree of speedup, there is still room for even more improvement through hardware acceleration. Traditional multicore processors are not well suited towards executing parallel programs with abundant fine-grained parallelism expressed as many small threads [18]. These programs have small tasks for which software task schedulers achieve only limited parallel speedups due to the increased overhead relative to the amount of work done per task [17]. Researchers have proposed augmenting processors with dedicated hardware to aid in dynamic load distribution and balancing. Rigel is a 1024 core architecture designed for task-level parallelism through its task-centric memory model [13]. StarT [25] implemented dedicated hardware queues for message passing with high priority levels for kernel code. Carbon [17] proposes adding specialized instructions and dedicated hardware queues with fixed LIFO priority to handle task queue management. IsoNet [18] proposes a dedicated network of load balancing engines scalable to thousands of cores. Dedicated hardware queues have even been proposed for GPUs [14]. However, these techniques do not consider the potential performance improvements gained by a more optimal priority schedule. Although [28] proposes asynchronous direct messages to accelerate custom software schedulers, the authors only consider simple schedules such as FIFO and LIFO rather than true priority schedules.

Task scheduling is a first-order concern in the realm of internet core routers [34]. State-of-the-art routers have the ability to dynamically schedule between tens of thousands of queues at hundreds of millions of scheduling decisions per second per chip [35]. While there are differences between networking and fine-grained task scheduling, such as the size of the packets, latency timescales, and various other area and performance tradeoffs, some networking routing, prioritization, and scheduling algorithms may be relevant to my thesis. For my thesis, aside from developing my own worklist scheduling algorithms, I will consider scheduling techniques from networking to determine what concepts can be directly applied to improve my own scheduling techniques.

2 Proposed Work

2.1 Approach

The use of a high-level language like Galois with dynamic work generation and dynamic dependencies means that static scheduling approaches found in traditional high-level synthesis no longer apply. The Galois programming model semantics coupled with the Galois concurrent library components means that the user does not deal with issues such as atomicity, scheduling, work balancing, worklist spilling, prefetching, and more. In addition, the Galois HLS compiler is responsible for the generation of custom pipelines. These pipelines are multithreaded, in which each pipeline stage is working on a different thread. Unlike pipelines generated by traditional HLS tools such as Vivado, instructions flowing through the pipeline are issued dynamically as work items arrive. Pipeline stages may stall based on the control flow and variable latency accelerators. The compiler is responsible for generating the proper stall and forwarding logic to preserve correctness.

Due to pointer-chasing memory patterns and large memory footprint of graph applications, traditional memory hierarchies found in general purpose processors tend to perform poorly. This is where FPGAs can be much more effective. By tailoring the Galois microarchitecture to be able to saturate the available memory bandwidth of the system, an FPGA platform can outperform a general purpose CPU platform with similar bandwidth since the CPU often cannot effecitvely exploit all the available bandwidth in the system.

While the Galois for FPGA framework requires a substantial amount of work implementing transactional memory semantics, this is outside the scope of my thesis. My labmate Xiaoyu Ma will be implementing this as part of his thesis.

2.2 Proposed User Workflow

Using Galois for FPGA is similar to using Galois for software. The user writes a Galois program using the Galois library modes and the **foreach** Galois construct. This Galois program will have different syntax to the current Galois SW implementation in C++, but that's because the authors of Galois did not wish to implement a compiler, instead implementing Galois as several libraries within C++. In addition, instead of using C++, the Galois HLS compiler will use C and have the same restrictions as a standard HLS tool such as the lack of dynamic memory allocation outside of the Galois constructs, as I plan on implementing Galois by augmenting an existing HLS tool. Pragmas to provide high-level microarchitectural hints may be required, but will be avoided if possible.

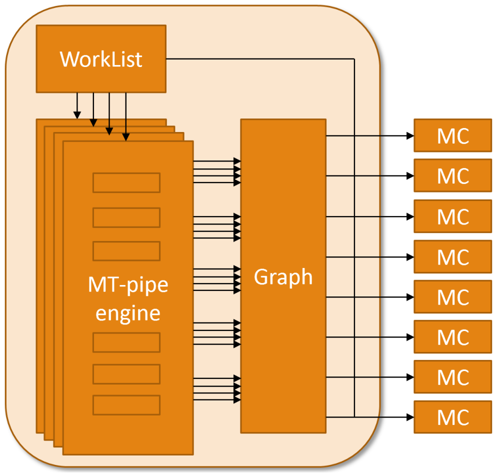

Figure 2: The Galois microarchitecture

After writing the Galois program, the user calls the Galois HLS compiler. The compiler generates the custom multithreaded pipelined engine code executing the **foreach** block and connects the engines to the Galois microarchitecture implemented in Bluespec. Parameters are passed to the Galois microarchitecture by the compiler with possible user input. Either the user or the compiler will tweak the parameters until the design meets timing and routing FPGA constraints.

2.3 Galois Microarchitecture

The Galois microarchitecture is a high-throughput packet processor that interfaces with custom engine code generated by the Galois HLS compiler as seen in Figure 2. It consists of four major components: the custom engine code, the instantiated worklist, the instantiated graph module, and the surrounding infrastructure logic. The worklist and graph module variant and parameters are chosen by the user through explicit instantiation in the user code. Some other parameters may be set by the Galois HLS compiler based on the number of engines generated and the Galois library calls made by the engines. The surrounding infrastructure logic provides glue logic for the components and is custom to the target FPGA platform.

The Galois microarchitecture bears many similarities to Tera [1]. Like Tera, the Galois microarchitecture is a heavily multithreaded throughput system without data caches. However, there are several key differences. Since the Galois engines are custom-generated for the target program, they can achieve a high throughput per engine at low power. Depending on the application and available bandwidth in the FPGA platform, instead of needing 256 cores like Tera, the Galois microarchitecture only needs tens of engines or even fewer to fully saturate memory bandwidth. In addition,

Method	Description
void enq(WorkItem work)	Enqueue work item into worklist
void enq(Priority p, WorkItem work)	Enqueue work item into worklist with priority p
WorkItem deq()	Dequeue and return work item from worklist

Table 1: Worklist interface

Galois supports a dedicated hardware priority scheduler and transactional memory framework.

2.3.1 Worklist

The worklist library component performs work scheduling within the Galois microarchitecture. In the Galois programming model, the user writes a Galois kernel consisting of a foreach loop, wherein each loop iteration dequeues a work item from the worklist and may enqueue any number of additional work items to the worklist. The worklist therefore presents a simple FIFO interface is exposed to the programmer shown in Table 1, but behind the curtains it must perform dynamic work scheduling across tens to hundreds of threads on multiple engines across multiple FPGAs. In addition, the size of the worklist is unbounded, so the worklist must support dynamic fills and spills to memory. Proper care must be taken to ensure that work is balanced evenly using optimizations like work stealing, with optimizations such as streaming and double buffering to ensure memory spills and fills do not bottleneck the engines.

The standard worklist supports a simple FIFO schedule. However, many algorithms such as SSSP converge much faster given a priority schedule [11]. Current state-of-the-art for software worklists require the user to specify a static priority algorithm, such as the bucket size for SSSP. Various priorities have been proposed for different algorithms, optimizing not only for convergence speed but also cache hit rate. Unfortunately, the best priority not only depends on the algorithm but also on the host architecture and shape of the input graph. In addition, I strongly suspect that the optimal priority may change as the amount of parallelism in the system dynamically grows and shrinks during execution.

2.3.2 Worklist priority scheduling

To further improve work efficiency, I propose the use of dynamic priority scheduling. Dynamic priority scheduling will utilize microarchitectural performance counters to optimize for convergence speed when possible. For example, in SSSP delta-stepping, the size of the bucket should be as small as possible while still providing enough work to avoid backpressuring the engines. By monitoring the amount of work in each bucket and the rate of engine work consumption, the worklist can

Method	**Description**
void startTransaction()	Start atomic transaction
void endTransaction()	End atomic transaction
Node readNode(NodeID id)	Read node from graph
Edge readEdge(EdgeID id)	Read edge from graph
EdgeIterator readEdges(Node node)	Galois iterator for each edge in graph node
Bool cas(Payload& cmpVal, Payload swapVal)	Compare-and-swap, returns success and old val
void writeNodePayload(Payload p)	Write node payload
void writeEdgePayload(Payload p)	Write edge payload

Table 2: Graph interface

dynamically adjust bucket size to provide near-optimal performance.

Other algorithms will require different types of scheduling to maximize the rate of convergence. Ideally, choosing the correct schedule should be determined by the microarchitecture. I will explore the possibility of determining the right scheduling algorithm to use and its parameters to maximize for performance, which is a function of the convergence rate and rate of work item completion.

2.3.3 Graph

Compiler-emitted Galois kernel engines do not interface directly with memory. Instead, they interface with the Graph concurrent library module, which handles preserving atomicity, interfacing with the underlying host memory fabric, and handling multiple memory operations in flight across multiple memory channels. The graph interface is shown in Table 2. Note that the user will not have to explicitly call startTransaction() and endTransaction(). Instead, the Galois HLS compiler will emit these operations in place of the user. However, if the user wishes to perform low-level optimizations, he or she may choose to forgo the use of transactional memory and instead manually call the provided compare-and-swap atomic operation.

Note that this graph library module provided is a local computation graph, i.e. the graph structure does not change. The user is only able to modify the node payload contents, i.e. the local data associated with each node. This limitation still supports large number of graph algorithms, including SSSP, breadth-first search, preflow push, PageRank, and many more. I will investigate supporting mutable graphs using a simple hardware memory allocator and garbage collector.

2.3.4 Galois Infrastructure

The Galois infrastructure consists of glue logic, buffering, and flow control for the engines, worklist, and graph. Graph and worklist operations must be converted from generic memory operations to the interface required for running on that particular FPGA platform. Arbitration is required between the worklist and graph, as well as with the parallel workers within the worklist and graphs themselves. The infrastructure must balance requests between the available memory channels on the host and guarantee forward progress, as well as performance optimizations such as memory request balancing (either static or dynamic) across the memory channels. As each FPGA platform has its own unique memory interface, a new Galois infrastructure must be designed for every FPGA platform. The current infrastructure is designed to exploit the Convey MX-100 platform's atomic memory operations and multi-FPGA environment, which many platforms do not support.

2.4 Galois HLS compiler

For my thesis, I plan on augmenting an existing HLS compiler to support the **foreach** Galois construct as well as the Galois concurrent worklist and graph library modules. A good candidate is LegUp [5], an HLS compiler built on top of LLVM. The Galois HLS compiler should detect the **foreach** construct and call specialized code routines to perform the appropriate pipelining. As there are loops in Galois code, e.g. to iterate over a node's edges, the compiler must generate the appropriate code to handle the flow control and deadlock avoidance. The compiler will only generate the engine, i.e. the **foreach** loop body converted to hardware. After engine generation, the compiler instantiates and connects the Galois microarchitecture surrounding the engine from its set of Galois library components.

The focus of this proposal has been on the microarchitecture. I have hand-designed a fully pipelined engine to handle SSSP and integrates with the Galois infrastructure, including the worklist and graph library components. The goal is for the compiler to generate comparable code.

Method	Description
CPU	1x Intel Xeon E5-2643
CPU Memory	256GB DDR3
FPGA	4x Virtex-6 HX-565T
FPGA Memory	128GB SG-DIMMs
FPGA Memory Bandwidth	128GB/s

Table 3: Worklist interface

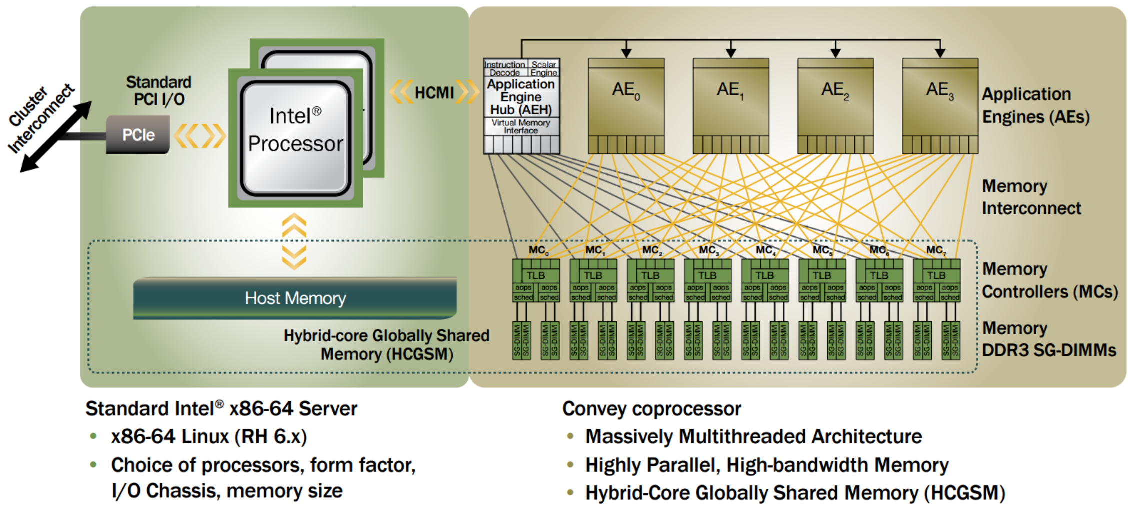
Cluster Interconnect
Standard PCI I/O
PCIe
Intel® Processor
HCMI
Host Memory
Hybrid-core Globally Shared Memory (HCGSM)
Instruction Decode
Scalar Engine
Application Engine Hub (AEH)
Virtual Memory Interface
AE0
AE1
AE2
AE3
Application Engines (AEs)
Memory Interconnect
Memory Controllers (MCs)
Memory DDR3 SG-DIMMs
TLB
Standard Intel® x86-64 Server
x86-64 Linux (RH 6.x)
Choice of processors, form factor, I/O Chassis, memory size
Convey coprocessor
Massively Multithreaded Architecture
Highly Parallel, High-bandwidth Memory
Hybrid-Core Globally Shared Memory (HCGSM)

Figure 3: Convey MX-100 architectural overview

3 Work Completed

As part of the preliminary evaluation process, I am in the process of implementing the Galois microarchitecture depicted in Figure 2 with hand-written rather than compiler-generated engines for the delta-stepping SSSP algorithm. All hardware modules, including both Galois library modules as well as the SSSP engines, are written in Bluespec System Verilog [23], a high-level hardware description language. Ideally, the use of Bluespec should enable much tighter control of the generated Galois microarchitecture compared to HLS languages while greatly increasing productivity over hardware description languages like SystemVerilog.

3.1 The Convey MX-100

I am currently targeting the Convey MX-100 as my FPGA platform. As shown in Table 3, the MX-100 is a bleeding-edge machine coupling fast general-purpose processors with multiple FPGAs with dedicated high-bandwidth scatter-gather memory optimized for 64-bit operations. Figure 3 depicts the overall platform architecture. The four FPGAs, known as Application Engines, are connected with a crossbar to 8 memory controllers, each connecting to two channels of scatter-gather

```
Graph graph = { /* init */}; // Initialize graph contents
WorkList<GraphNode> workQ();
workQ.enq(init); // worklist initialized with the starting node
foreach(uint nodeId : workQ) {
    GraphNode node = graph.readNode(nodeId);
    foreach(Edge edge : graph.readEdges(node)) {
        bool retry = true;
        while(retry) {
            retry = false;
            uint newDist = node.dist + edge.weight;
            if(newDist < graph.readNode(edge.dest).dist) {
                if(graph.cas(node.dist, newDist)) {
                    workQ.enq(edge.dest);
                }
                else {
                    retry = true;
                }
            }
        }
    }
}
```

Figure 4: Modified Galois target code for SSSP

memory optimized for 64-bit random access, for a total of 16 memory channels. To design a Convey hybrid-core program, the engineer first designs a custom Personality, consisting of the FPGA bit-file containing a custom instruction set designed to accelerate individual algorithms within applications. The Personality source code must interface with Convey's Personality Development Kit (PDK), which presents an interface to Convey's abstraction layer, including its memory interface. Convey users must explicitly transfer data to the FPGA memory before calling the Personality instruction, and transfer the results back to CPU memory upon instruction completion. To run on Convey, the Galois microarchitecture resides within a Personality. The Galois HLS tool generates the hybrid-core program along with the required data transfers to and from FPGA memory.

The MX-100 is so far the only machine designed by Convey to support atomic memory operations, including atomic add, subtract, and compare-and-swap (CAS) operations. I chose the MX-100 platform because these atomic operations are essential for supporting high-performance synchronization across multiple FPGAs. In addition, the specialized Convey scatter-gather memory is perfect for supporting the pointer-chasing memory access patterns present in many Galois programs such as SSSP.

```
Graph graph = { /* init */}; // Initialize graph contents
WorkList<GraphNode> workQ();
workQ.enq(init); // worklist initialized with the starting node
foreach(uint nodeId : workQ) {
    graph.readNodeReq(nodeId);
    GraphNode node = graph.readNodeResp();
        graph.readEdgesReq(node.edgePtr + i);
        foreach(Edge edge : graph.readEdgesResp());
        bool retry = true;
        while(retry) {
            retry = false;
            uint newDist = node.dist + edge.weight;
            graph.readNodeReq(edge.dest);
            if(newDist < graph.readNodeResp().dist) {
                graph.casReq(node.dist, newDist);
                if(graph.casResp()) {
                    workQ.enq(edge.dest);
                }
                else {
                    retry = true;
                }
            }
        }
    }
}
```

Figure 5: Transformed Galois code for SSSP

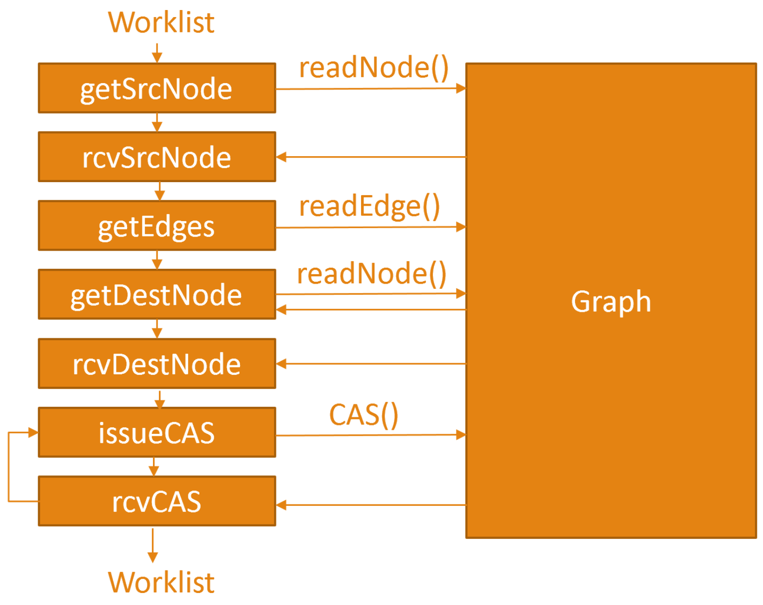

Figure 6: Scheduled SSSP Pipeline

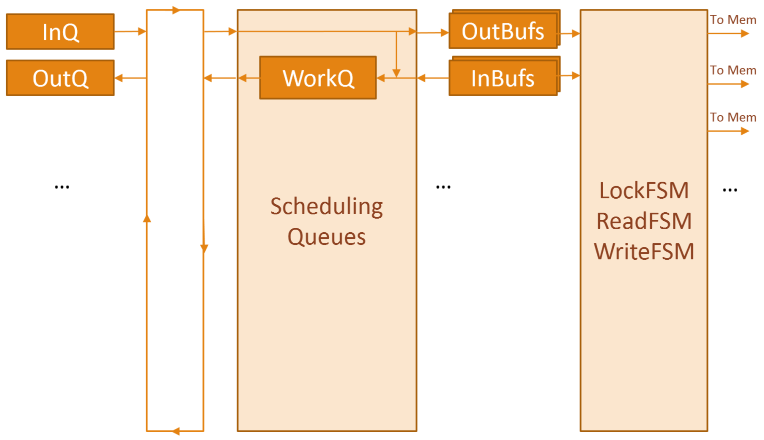

Figure 7: Worklist Microarchitecture

3.2 Implemented Custom SSSP Engine

To generate the engine for SSSP, I manually performed the role of the Galois HLS compiler by scheduling the source code into the optimal pipeline stages and generating the requisite scaffolding. Since transactional memory support is in the process of being developed by my labmate Xiaoyu Ma, I chose to use a modified version of SSSP with a non-atomic **foreach** with explicit synchronization through the use of the atomic compare-and-swap operation. To maximize throughput, the compiler must support multiple Graph method calls in flight; to do this, a simple mechanical code transformation is necessary, shown in Figure 5. By decoupling the method request and response calls, the compiler can now schedule the request and response calls into different pipeline stages and generate the appropriate context buffering to support multiple calls in flight. The final schedule is shown in Figure 6, along with the decoupled Graph request and response calls.

Note that since there are loops in the pipeline, special care must be taken to prevent deadlock, which can occur when a thread needs to loop back but the context buffer is full. Since looping back is performed by enqueueing a thread context from the end of the pipeline to an earlier stage, this causes deadlock as that thread prevents other threads from making forward progress. I solve the deadlock issue through a credit-based system. All threads attempting to enter the loop must first reserve space in the context buffer. If no space exists, then the thread stalls until a thread has finished looping.

3.3 Implemented Worklist

For my initial worklist implementation shown in Figure 7, I focused on designing a simple microarchitecture capable of scaling to a large ($\tilde{1}6$) number of engines. The worklist is comprised of two

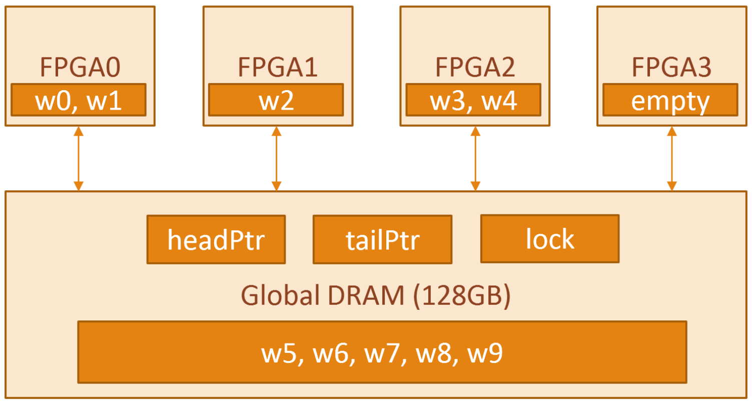

Figure 8: Worklist Data Organization Example

components, the front-end and back-end. The front-end is responsible for work balancing across engines and implementing any partial priority ordering schemes specified by the user. The worklist presents a simple packet enqueue/dequeue interface to each engine, where each packet currently consists of a single worklist work item. The back-end handles worklist spills and fills to main memory.

The worklist front-end consists of N lanes, where each lane is connected directly with the engine. Currently, only FIFO work ordering is supported. Each lane has a dedicated large-capacity queue (currently 2048 entries). As work is generated dynamically under the Galois programming model, I have implemented a work stealing ring network optimized for FPGA routability and fast work distribution. If an engine wishes to dequeue a worklist packet but none are available, the worklist sends a work steal request to its neighbor. The next time the neighbor performs a worklist enqueue, the packet is delivered to the requestor's work stealing queue.

When a worklist front-end lane attempts to perform an enqueue but its work queue is full, it sends the packet to the back-end. The back-end contains dedicated per-lane input and output double buffers to stream work items to and from global memory. When an output buffer is filled, the data write FSM is triggered. As shown in Figure 8, global memory contains a unified work queue shared among all FPGAs in the system. The unified work queue handles work distribution between FPGAs, enables unbounded work generation, and enables each FPGA to store only a small percentage of work items in local BRAM. The FSM therefore locks the global unified work queue before writing to the queue. To minimize locking overhead, the FSM performs a streaming write operation consisting of all packets in a lane's output buffer (currently 1024).

The worklist back-end read FSM is triggered when any lane's input buffers are empty and the global work queue is not empty. As the FPGA's worklist back-end contains a stale copy of the global worklist head and tail pointers, the readFSM will poll the worklist to determine if new work has

Module	Slices	% of total	Slice Reg	LUTs	BRAM
XC6VHX565T	88,560	100%	708,480 (100%)	354,240 (100%)	912 (100%)
Convey Top	67,586	76%	198,903 (28%)	182,824 (51%)	146 (12%)
* BSV Wrapper	33,979	38%	88,601	68,060	96
** SSSP Top	20,831	24%	50,816	46,617	80
*** SSSP Engines	3,036	3.4%	8,796	7,916	24
*** Graph	5,576	6.3%	15,862	13,262	24
*** Worklist	4,491	5.1%	9,420	10,291	32
**** Worklist Backend	3,629	4.1%	7,650	8,334	24

Table 4: SSSP FPGA Resource Usage (4 engines)

FPGA Count	4
Engine Count per FPGA	4
Clock Speed	125MHz
Graph Mem Ops in Flight per Port	128
Worklist Entries per Engine	8192
Engine Double Buffer Size	1024

Table 5: SSSP uArch Tested Configuration

been added. Like the write FSM, the read FSM locks the global work queue before removing entries from the queue.

3.4 Implemented Graph

As my labmate Xiaoyu Ma is in charge of implementing transactional memory, for the purposes of evaluating SSSP I have only implemented the readNode, readEdges, and compare-and-swap operations. Each operation is implemented as a pipeline supporting many concurrent operations in flight. Right now, as shown in Figure 6, the compiler instantiates a dedicated graph operation pipeline for each engine graph call. The compiler must then properly bind the pipeline memory requests to memory channels. With 16 engines and 16 memory channels, the compiler simply needs to map each engine's pipeline requests to a single memory channel. However, with 8 engines or 4 engines, the compiler must map more than one memory channel to a single engine's graph requests. Proper care must be taken to balance memory channel utilization to achieve high performance while satisfying FPGA routing constraints.

3.5 SSSP Simulated Results

I was able to successfully synthesize a 4-engine SSSP microarchitecture. The FPGA bitfile is programmed to all four Convey Application FPGAs, resulting in 16 engines in total. The details of my configuration are shown in Table 5. With up to 128 memory operations in flight per memory

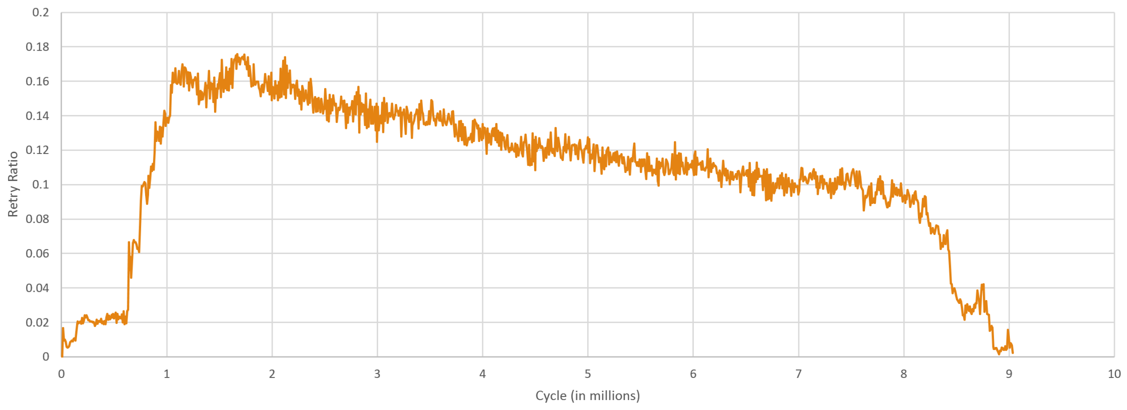

Figure 9: Compare-and-Swap Retry Ratio

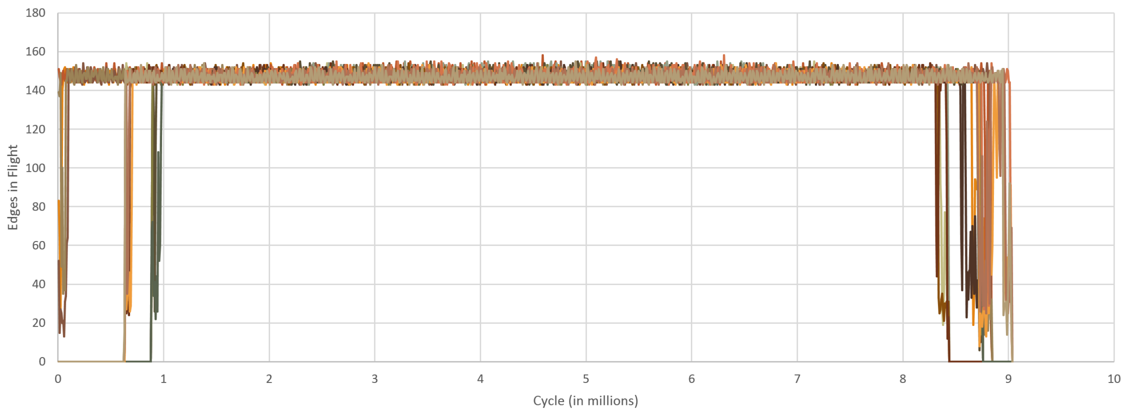

Figure 10: Edges in Flight per Engine

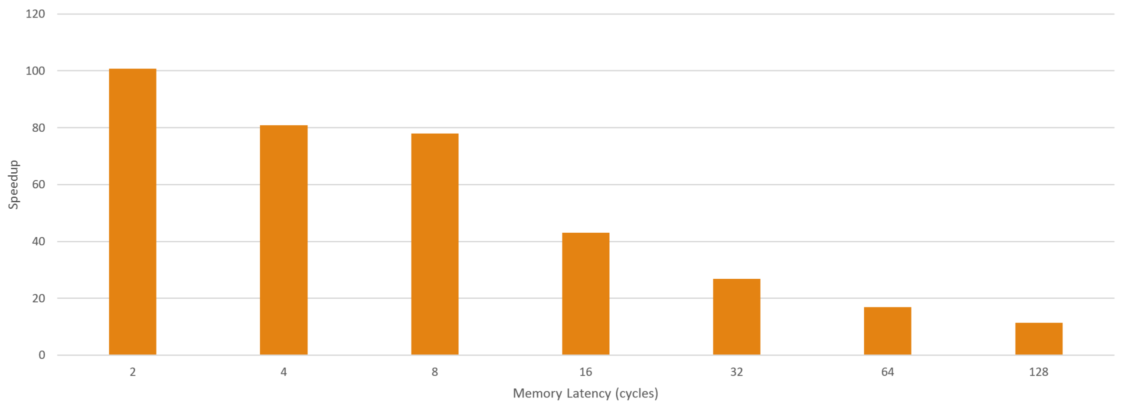

Figure 11: Edges Committed per Second Speedup vs. Software

port, the design should in theory be able to send a memory request to each channel every cycle, resulting in up to 16 memory requests per cycle. FPGA resource utilization is shown in Table 4. Convey overheads are high: the Galois microarchitecture is only 32% of the size of the entire design, consuming 24% of total slices in the FPGA. The Bluespec wrapper itself uses 14% of available FPGA slices; additional engineering effort will be required to minimize this. Unfortunately, I am currently unable to meet timing at 150MHz. Instead, the Galois microarchitecture is downclocked to 125MHz while the rest of the Convey infrastructure runs at 150MHz.

Although I was able to synthesize my design, I have so far been unable to run on hardware due to some minor technical issues. However, I am able to run reasonably-sized graphs to completion in the Bluesim simulation framework. The SSSP design is functionally correct compared to a software golden model. For my tests, I used USA-road-NY, an undirected graph consisting of all roads in the state of New York, containing 264K nodes and 730K edges. However, this graph is too small to obtain any meaningful result running in software. As a rough approximation to overall performance, I measured the rate at which the SSSP engines were able to commit SSSP destination nodes. A commit occurs when a shorter path has been discovered, resulting in a distance update to the destination node and the destination node being added to the worklist. I then measured the average edges committed each second by a single-threaded Galois software implementation running a large workload (USA-road-USA) on a 4-socket Xeon E7540 server.

The results in Figure 11 shows the potential of the Galois microarchitecture, but also its current limitations. Although I can achieve 80x to 100x speedup with 2-8 cycles of memory latency, performance drops significantly as memory latency is swept from 16 to 128. This is due to the microarchitecture being immature; I believe that the issue lies with the number of memory round-trips required to synchronize and begin a worklist spill or fill operation. The current worklist spills to a global shared queue; a simple optimization would be to split the global shared queue into private queues and perform dynamic work balancing.

To preserve atomicity, SSSP node distance updates are made using the atomic compare-and-swap operation. Figure 9 shows the number of CAS retry operations as a function of time across the entire span of program execution. Retries start out low, but quickly ramp up as multiple engines are assigned work by the scheduler. The retry ratio peaks after all engines are warmed up and executing, and gradually decreases as each engine starts processing nodes further away from other engines. The small spikes near the end of execution are a result of better paths being discovered, resulting in idle engines being woken up and causing more conflicts.

Figure 10 shows the number of edges in flight per engine as a function of time. After each FPGA has been woken up, a steady state of roughly 150 edges in flight is quickly reached and remains there for the entire duration of program execution. This demonstrates that there is sufficient parallelism in the algorithm and input dataset, which enables high performance from the throughput-optimized Galois microarchitecture.

4 Research Roadmap

The preliminary SSSP Galois microarchitecture is already implemented and running within the Convey MX-100 FPGA platform in simulation. The remaining components of the proposed thesis and anticipated time to implement, test, and evaluate them are listed below:

- Accurate Convey MX-100 memory timing simulator to aid evaluation - 1-2 weeks.
- Run and tune SSSP implementation on Convey MX-100 with large data inputs - 2-3 weeks.
- Design, implement, and evaluate worklist scheduling algorithms - 4-5 months.
- Integrate Xiaoyu Ma's transactional memory algorithms within Galois uarch - 2-4 weeks.
- Design, implement, and evaluate HLS algorithms for engine auto-generation - 2-3 months.
- Miscellaneous integration, tuning, testing, and debugging - 1-2 months.
- Writing of dissertation and preparing for thesis defense - 2 months.

In total, I anticipate roughly 12 to 15 months of work remain to complete the proposed work, produce the dissertation, and prepare for the thesis defense.

5 Conclusion

I propose the use of the parallel graph language Galois for FPGA, which presents a high-level transactional programming model, allowing the user to describe the fine-grained task parallelism available in amorphous data-parallel graph algorithms to be more easily extracted by the compiler and Galois runtime. By combining the high-level transactional semantics of Galois with the automatic Galois HLS compiler and Galois microarchitecture that supports dynamic scheduling and synchronization, my tool can simultaneously improve quality of results while decreasing programmer effort. To achieve these goals, I will augment an existing HLS compiler to support the Galois unordered **foreach** loop iterator, and design a set of Galois library components and Galois microarchitectural template. The compiler will emit custom pipelined engines and connect them to my proposed Galois microarchitecture. Given the scope of the project, I will first focus on Galois programs that do not modify the structure of the input graph, but will later experiment with a simple memory allocator for mutable graphs. I will also not be focusing on preserving atomicity across **foreach** loop iterations, as that is the thesis project of my labmate Xiaoyu Ma. Preliminary results for SSSP demonstrate that my proposed Galois microarchitecture is promising and should be effective for accelerating other Galois programs.

Bibliography

[1] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The tera computer system. In *ACM SIGARCH Computer Architecture News*, volume 18, pages 1–6. ACM, 1990.

[2] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. 1986.

[3] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.

[4] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on GPUs. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151, 2012.

[5] Jongsok Choi, S. Brown, and J. Anderson. From software threads to parallel hardware in high-level synthesis for fpgas. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 270–277, Dec 2013.

[6] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[7] A. Davidson, S. Baxter, M. Garland, and J.D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 349–359, May 2014.

[8] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of openmp task scheduling strategies. In *OpenMP in a new era of parallelism*, pages 100–110. Springer, 2008.

[9] LR Ford. Network flow theory. 1956.

[10] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[11] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. Ordered vs unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 3–12, New York, NY, USA, 2011. ACM.

[12] Kirsten Hildrum and Philip S Yu. Focused community discovery. In *Data Mining, Fifth IEEE International Conference on*, pages 4–pp. IEEE, 2005.

[13] Daniel R Johnson, Matthew R Johnson, John H Kelm, William Tuohy, Steven S Lumetta, and Sanjay J Patel. Rigel: A 1,024-core single-chip accelerator architecture. *IEEE Micro*, 31(4):30–41, 2011.

[14] Ji Yun Kim and C. Batten. Accelerating irregular algorithms on gpgpus using fine-grain hardware worklists. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 75–87, Dec 2014.

[15] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[16] Donald E Knuth. A generalization of dijkstra's algorithm. *Information Processing Letters*, 6(1):1–5, 1977.

[17] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 162–173, New York, NY, USA, 2007. ACM.

[18] Junghee Lee, Chrysostomos Nicopoulos, Hyung Gyu Lee, Shreepad Panth, Sung Kyu Lim, and Jongman Kim. Isonet: Hardware-based job queue management for many-core architectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 21(6):1080–1093, 2013.

[19] U. Meyer and P. Sanders. &#x0394;-stepping: A parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, October 2003.

[20] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Data-driven versus topology-driven irregular computations on GPUs. In *IEEE 27th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 463–474, 2013.

[21] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, 2013.

[22] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, 2013.

[23] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, June 2004.

[24] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C. Hoe, Jose F. Martinez, and Carlos Guestrin. Graphgen: An fpga framework for vertex-centric graph computation. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 25–28, May 2014.

[25] G. M. Papadopoulos, G. A. Boughton, R. Greiner, and M. J. Beckerle. *t: Integrated building blocks for parallel computing. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 624–635, New York, NY, USA, 1993. ACM.

[26] Keshav Pingali, Milind Kulkarni, Donald Nguyen, Martin Burtscher, Mario Mendez-Lojo, Dimitrios Prountzos, Xin Sui, and Zifei Zhong. Amorphous data-parallelism in irregular algorithms. regular tech report TR-09-05, The University of Texas at Austin, 2009.

[27] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[28] Daniel Sanchez, Richard M Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *ACM Sigplan Notices*, volume 45, pages 311–322. ACM, 2010.

[29] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[30] Dan A Spielman, Shang-Hua Teng, and Alper Ungor. Parallel delaunay refinement: Algorithms and analyses. *arXiv preprint cs/0207063*, 2002.

[31] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[32] Pang-Ning Tan, Michael Steinbach, Vipin Kumar, et al. *Introduction to data mining*, volume 1. Pearson Addison Wesley Boston, 2006.

[33] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[34] Xipeng Xiao and Lionel M Ni. Internet qos: a big picture. *Network, IEEE*, 13(2):8–18, 1999.

[35] M. Yamada, T. Yazaki, N. Matsuyama, and T. Hayashi. Power efficient approach and performance control for routers. In *Communications Workshops, 2009. ICC Workshops 2009. IEEE International Conference on*, pages 1–5, June 2009.