Processor Assisted Worklist Scheduling for FPGA Accelerated Graph Processing on a Shared-Memory Platform

Yu Wang, James C. Hoe
Department of Electrical and Computer Engineering
Carnegie Mellon University
$\{yuw, jhoe\}$ $@ece.cmu.edu$

Eriko Nurvitadhi
Intel Labs
eriko.nurvitadhi@intel.com

***Abstract*—FPGA-based processing has gained much attention for accelerating graph analytics because of the demand in performance and energy efficiency. However, while priority scheduling has been shown to be an effective optimization for improving performance for worklist-based graph computations, it is rarely used in accelerator designs due to its implementation complexity and memory-access overhead.**

In this paper, we present a heterogeneous processing approach for priority scheduling on a shared-memory CPU-FPGA platform. By exploiting the closely-coupled integration of the host processor and the FPGA accelerator, our system dynamically offloads the task of scheduling to a software scheduler on the processor for its programmability, high-capacity cache and low memory latency, while the FPGA graph processing accelerator enjoys the scheduling benefit and delivers higher performance at excellent energy efficiency. To understand the effectiveness of our solution, we compared it with FPGA-only solutions for two scheduling schemes: the well-known Dijkstra scheduling for Single Source Shortest Path and a new scheduling optimization we developed for improving the data locality of Breadth First Search. Whereas the FPGA-only solution requires an impractical amount of on-chip storage to implement a priority queue, the proposed processor-assisted scheduling that moves the task of scheduling to the processor consumes a negligible load on the processor and retains most of the performance benefit from priority scheduling.

I. Introduction

Worklist-Based Graph Algorithms. Graph analytics are tools for determining the relationships among connected components in a graph. The wide variety of applications, including item recommendation [1], social network analysis [2], computer-vision [3], and so forth, mark their increasing importance. While differing in function, many graph analytic algorithms can be abstracted as *worklists* and iterative processing routines[4]. The worklist is a data structure that stores a set of pending tasks, often implemented as a queue; the processing routine iteratively dequeues and completes those tasks until the worklist is empty. With the freedom to define the task and the processing routine, this worklist-based model can be generally applied to represent different computations. As a graph computation progresses, new tasks are generated in an order determined by the topology of the input graph and the execution phases, which tend to be highly irregular. While this irregularity imposes a great challenge for extracting parallelism and exploiting data locality, it presents an opportunity for *worklist scheduling*, as the naturally-occurring insertion order of tasks is often not the ideal processing order.

Worklist Scheduling. For many graph algorithms, the tasks stored in the worklist can be scheduled to be processed in any order without affecting its functional correctness. Examples include *Breadth First Search* (*BFS*) and *Single Source Shortest Path* (*SSSP*). Even though FIFO is frequently used as a worklist for in-order processing for its low implementation complexity, there exist better processing schedules that can improve performance by exploiting application-specific information.

A classic example is the *Dijkstra* SSSP algorithm[5]. SSSP determines the shortest paths from a source vertex to the other vertices in a graph by iteratively expanding the frontier of search. By prioritizing the task vertex with the shortest path for frontier to skip the long paths that are less likely to be parts of the final solution, the Dijkstra scheduling drastically reduces the processing time. Another example uses scheduling to improve cache performance [6]. In fact, this type of priority scheduling is very common in software-based graph computations. State-of-the-art graph processing frameworks, such as Galois[4], GraphMat[7] and GunRock[8], all provide built-in support for task prioritization.

Problems with Worklist Scheduling for FPGA-Accelerated Graph Processing. Graph computations are fundamentally memory-bounded, which imposes a challenge to fully utilize the large, fast-clocked cores in modern processors. On the other hand, this provides an opportunity for FPGAs, as they can be configured to target only the needed operations for achieving great efficiency and performance. However, FPGA-based accelerators rarely exploit the optimization of worklist scheduling. To understand why, we conducted two case studies for worklist-based graph computations, including: 1) the aforementioned Dijkstra scheduling for SSSP and 2) a new scheduling optimization for BFS that we proposed for improving on-chip data reuse by prioritizing the processing of topologically close vertices. We first established competitive non scheduling FPGA baselines for both algorithms and

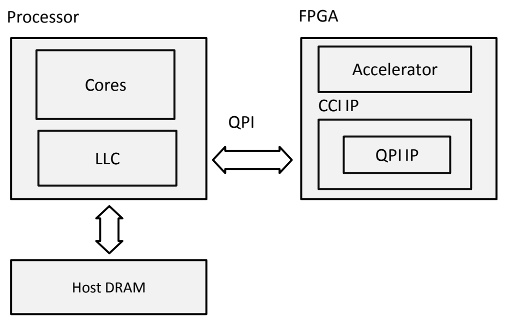


Fig. 1. A QPI-based shared-memory processor-FPGA platform.

discovered that, in addition to the implementation complexity, adding hardware support for priority scheduling consumes a large amount of on-chip BRAM. As the baseline accelerator also needs BRAM for caching vertex data this results in a serious conflict for the limited FPGA resources.

Our Processor-Assisted Solution. To tackle this problem, we propose **Processor Assisted Scheduling (PAS)** on a shared-memory FPGA platform. A shared-memory system, such as *Intel Heterogeneous Accelerator Research Platform (HARP)*[9] shown in Figure 1, offers coherence accesses to a shared off-chip memory for the FPGA and the host processor through a high-performance interconnect, like Intel's *Quick-Path Interconnect (QPI)*. This enables dynamic, fine-grain data-sharing and communication between the CPU and the FPGA. PAS harnesses the power of a shared-memory system by using FPGA for the standard graph processing routine and dynamically offloading the scheduling task to the host processor. The software scheduler exploits the sophisticated cache system and lower memory latency of the processor to deliver fast scheduling service, which helps the accelerator to obtain the benefit of worklist scheduling without consuming precious BRAM resources.

In our evaluation, PAS allows SSSP and BFS to obtain over 90% and 80% of the theoretical scheduling benefit on an Intel HARP, respectively. Additionally, as the software scheduler spends most of the processing cycles waiting for new tasks to be produced by the accelerator or the priority-queue accesses to complete, this lightweight routine places a negligible load on the CPU and interferes minimally with other concurrent applications, making PAS ideal for servers. By allowing the processor and the accelerator to work on what they are good at, the collaborative processing of PAS achieves better performance and energy efficiency than each can individually deliver.

Our Contributions.

- We developed, implemented and evaluated PAS on an Intel HARP. The evaluation results suggest that PAS allows the accelerator to obtain the majority of the performance benefit from worklist scheduling without consuming extra FPGA resources. In addition, we observed that PAS incurs negligible interference to other applications running in parallel on the CPU.
- As a use case of PAS, we developed a new worklist scheduling for BFS, which combines graph-preprocessing and dynamic scheduling to prioritize the processing of vertices that are likely in near topological neighborhood to improve the utilization of cache and overall performance.

II. Priority Scheduling for Worklist-Based Graph Algorithms

In this section, we will present two examples showing how priority scheduling can be applied to improve the performance for graph algorithms. The first example is the previously mentioned Dijkstra-based SSSP. The second example is a new locality optimization that we developed for this work, which combines graph pre-processing and priority scheduling to improve the data locality for BFS.

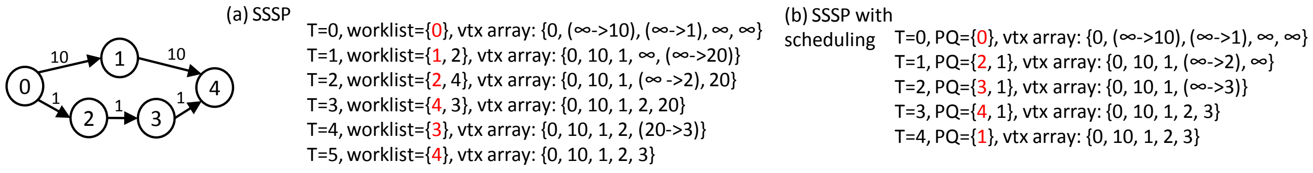
A. Reducing Tasks for SSSP

The Inefficiency of In-Order Processing. A worklist-based implementation of SSSP supporting graphs in *Compressed Sparse Row (CSR)* format is shown in Algorithm 1, which iteratively labels each vertex with its currently-known shortest path. During initialization, the labels of all vertices are set to infinity, and then the worklist enqueues the source. SSSP then enters the main loop for processing the stored tasks, or vertices with newly discovered shortest paths, from the worklist by attempting to ”relax” their neighbors (lines 4-16). Relaxation is the process of exploring the path spanning form the source to the task vertex and then to its neighbor. If this path is shorter than the previous path assigned to the neighbor, the relaxation succeeds, and a new shorter path is found. Finally, successfully-relaxed vertices are inserted to the worklist to be processed later.

Algorithm 1 Baseline SSSP algorithm

```
 1: FIFO_worklist.enqueue(source);
 2: for(int i=0;i¡num_vtx;++i)vtx_array[i].dist=INF;
 3: while !FIFO_worklist.empty() do
 4:     vtx curr_task=FIFO_worklist.dequeue();
 5:     int task_idx=curr_task.idx;
 6:     int task_dist=curr_task.dist;
 7:     int curr_edge_offset=vtx_array[task_idx].edge_offset;
 8:     int num_edge=vtx_array[task_idx].num_edge;
 9:     for num_edge to 0 do
10:         int curr_nbr=edge_array[curr_edge_offset].dst;
11:         int edge_length=edge_array[curr_edge_offset].length;
12:         int curr_dist=edge_length+task_dist;
13:         vtx nbr_vtx=vtx_array[curr_nbr];
14:         if vtx_array[nbr_vtx].dist¿curr_dist then
15:             nbr_vtx.dist=curr_dist;
16:             vtx_array[nbr_vtx.idx]=nbr_vtx;
17:             FIFO_worklist.enqueue(nbr_vtx);
18:         end if
19:     end for
20: end while
```

There are two important observations to be made here. First, while this implementation assumes an in-order FIFO for the worklist, the stored task vertices can actually be processed in



```
T=0, worklist={0}, vtx array: {0, (∞->10), (∞->1), ∞, ∞}
T=1, worklist={1, 2}, vtx array: {0, 10, 1, ∞, (∞->20)}
T=2, worklist={2, 4}, vtx array: {0, 10, 1, (∞ ->2), 20}
T=3, worklist={4, 3}, vtx array: {0, 10, 1, 2, 20}
T=4, worklist={3}, vtx array: {0, 10, 1, 2, (20->3)}
T=5, worklist={4}, vtx array: {0, 10, 1, 2, 3}
```

```
T=0, PQ={0}, vtx array: {0, (∞->10), (∞->1), ∞, ∞}
T=1, PQ={2, 1}, vtx array: {0, 10, 1, (∞->2), ∞}
T=2, PQ={3, 1}, vtx array: {0, 10, 1, (∞->3)}
T=3, PQ={4, 1}, vtx array: {0, 10, 1, 2, 3}
T=4, PQ={1}, vtx array: {0, 10, 1, 2, 3}
```

Fig. 2. SSSP scheduling to reduce number of tasks/relaxations from 6 in (a) to 5 in (b). The dequeued task of each iteration is highlighted in red.

an arbitrary order without violating the algorithm's correctness. Second, later relaxations, which hop over shorter edges, can lead to a shorter paths compared to an earlier relaxation, which hop over fewer long edges. The example in Figure 2.a illustrates this scenario, in which the upper path with two hops actually is actually longer than the lower path with three hops. This phenomenon creates an inefficiency because all vertex updates before the final relaxation are later overwritten and should be avoided in the first place, as they incur unnecessary but expensive writes to memory.

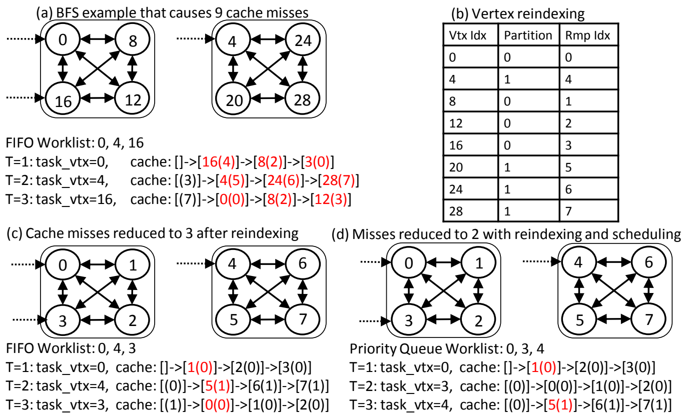
Dijkstra-Based Scheduling. To tackle this problem, the worklist scheduler can prioritize the processing of the task vertices with short path labels. As the paths to their neighbors are likely also short, this helps short paths that are likely part of the final solutions to be discovered early, and it reduces the relaxations for longer paths and total numbers of tasks. This technique is the core of the Dijkstra algorithm, which can be implemented by simply replacing the queue-based worklist in Algorithm 1 (*FIFO_worklist* in line 3 and 17) with an MIN priority queue. Figure 2.b shows an example of how priority scheduling reduces tasks.

B. Improving Cache Performance for BFS

Algorithm 2 Baseline BFS algorithm

```
 1: FIFO_worklist.enqueue(source);
 2: for(int i=0;i¡num_vtx;++i)vtx_array[i].predecessor=-1;
 3: while !FIFO_worklist.empty() do
 4:     int task_vtx=FIFO_worklist.dequeue();
 5:     int curr_edge_offset=vtx_array[task_vtx].edge_offset;
 6:     int num_edge=vtx_array[task_vtx].num_edge;
 7:     for num_edge to 0 do
 8:         int curr_nbr=edge_array[curr_edge_offset];
 9:         int nbr_vtx=vtx_array[curr_nbr];
10:         if vtx_array[nbr_vtx].predecessor==-1 then
11:             vtx_array[nbr_vtx].predecessor=task_vtx;
12:             FIFO_worklist.enqueue(nbr_vtx);
13:         end if
14:     end for
15: end while
```

A worklist-based implementation of BFS is shown in Algorithm 2. Like SSSP, BFS is another graph traversal algorithm for finding the shortest paths. However, BFS operates on non-weighted graphs with constant-length edges; every node is updated exactly once regardless of the visiting order. Thus, the task reduction scheduling from the Dijkstra SSSP cannot be applied. Although the Dijkstra scheduling is not applicable, worklist scheduling can be used to improve data locality and


(a) BFS example that causes 9 cache misses
0
8
16
12
4
24
20
28
(b) Vertex reindexing
(c) Cache misses reduced to 3 after reindexing
0
1
3
2
4
6
5
7
(d) Misses reduced to 2 with reindexing and scheduling
0
1
3
2
4
6
5
7

Vtx Idx	Partition	Rmp Idx
0	0	0
4	1	4
8	0	1
12	0	2
16	0	3
20	1	5
24	1	6
28	1	7

```
FIFO Worklist: 0, 4, 16
T=1: task_vtx=0,    cache: []->[16(4)]->[8(2)]->[3(0)]
T=2: task_vtx=4,    cache: [(3)]->[4(5)]->[24(6)]->[28(7)]
T=3: task_vtx=16,   cache: [(7)]->[0(0)]->[8(2)]->[12(3)]
```

```
FIFO Worklist: 0, 4, 3
T=1: task_vtx=0,  cache: []->[1(0)]->[2(0)]->[3(0)]
T=2: task_vtx=4,  cache: [(0)]->[5(1)]->[6(1)]->[7(1)]
T=3: task_vtx=3,  cache: [(1)]->[0(0)]->[1(0)]->[2(0)]
```

```
Priority Queue Worklist: 0, 3, 4
T=1: task_vtx=0,  cache: []->[1(0)]->[2(0)]->[3(0)]
T=2: task_vtx=3,  cache: [(0)]->[0(0)]->[1(0)]->[2(0)]
T=3: task_vtx=4,  cache: [(0)]->[5(1)]->[6(1)]->[7(1)]
```

Fig. 3. An example of locality-aware scheduling for BFS, assuming the cache stores only one cacheline, and each cacheline stores four vertex labels. Cache misses are highlighted in red.

cache performance by prioritizing the tasks that are likely to reuse cached data during their relaxation.

Locality-Aware Scheduling. To introduce locality-awareness to the worklist scheduler, we need an efficient heuristic to identify tasks that are likely to reuse cache data. One possible approach is to prioritize the task vertices with the largest number of cached neighbors, which was proposed in [6]. While this is an intuitive method, the scheduler requires dynamic access to the content of the cache, which requires significant hardware changes to the cache architecture and introduces extra design and silicon area overhead.

To avoid the above-mentioned issue, an alternative approach is to exploit the locality within a "community" by dispatching the tasks from the same community in consecutive order. A community is a sub-graph that is densely connected internally. During the process of relaxation, due the clustering nature of a community, processing task vertices of the same community consecutively likely results in better temporal locality. As many real world networks have been shown to contain community structures [10], this community-based scheduling can be generally applied.

Prior works[11], [12], [13] have explored static graph preprocessing that reindexes vertices based on graph partitioning results to improve the likelihood that adjacent vertices are from the same communities, and this technique can be exploited by our scheduler to dynamically prioritize the task vertices with near indices to improve cache performance. Similar to the SSSP scheduling, this can be done simply by replacing the

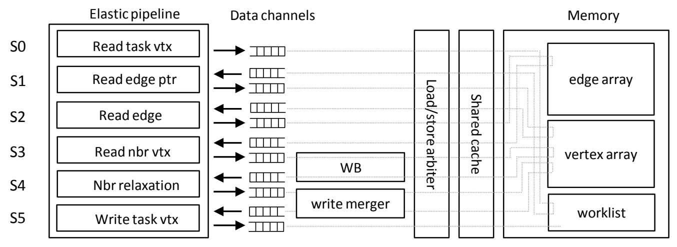


Fig. 4. Proposed baseline accelerator.

FIFO-based worklist Algorithm 2 with a priority queue to sort based on the indices of the stored tasks. Figure 3.d illustrates how priority scheduling can be applied to further improve performance of BFS with a reindexed graph. In this work, we employed a modified version of the two-level hierarchical reindexing proposed in [11].

C. Worklist Scheduling for Other Applications

The application of worklist scheduling is not limited to the above-mentioned examples. Although explained in the context of BFS, it has been shown that scheduling can be applied to improve the locality for other applications, including Page Rank and Collaborative Filtering [6]. Another use case of priority scheduling is time-stamp sorting. Simulation applications, such as Discrete Event Simulation[14] and Asynchronous Variational Integrators[15] from the Galois benchmarks, use priority queues to efficiently find tasks with the smallest time-stamp instead of manually managing the ordering of task processing. Furthermore, the fact that major graph-processing frameworks such as GraphLab[16] and Galois all feature native support for task prioritization explains its importance and general applicability.

III. Baseline FPGA-Only Accelerators for Graph Algorithms

In this section, we will present a high-quality traditional FPGA-only accelerator for SSSP and BFS that represents a reasonable non scheduling baseline. We will first explain the design in the context of SSSP. To tackle the irregular data access, the main bottleneck for graph algorithms, our baseline FPGA-only accelerator is designed with emphasis on two principles: 1) exploiting available MLP, and 2) maximizing on-chip data reuse. Figure 4 shows the architecture diagram of our FPGA-only SSSP accelerator based on Algorithm 1.

MLP Extraction. This accelerator uses an elastic pipeline architecture, in which the first four pipeline stages are responsible for issuing read requests for different types of data, including: 1) task vertices and their distance labels from the worklist, 2) the edge pointers of task vertices, 3) outgoing edges from the task vertices, and 4) the neighbor vertices pointed by the edges. The last two stages are responsible for performing memory writes for vertex relaxation and worklist updates, respectively. To address the emphasis on MLP, each pipeline stage can independently issue memory requests, allowing for four loads and two stores to be issued in parallel at each clock period in the ideal case when there is no stall.

This approach is fundamentally different from most software implementations, which rely mainly on multi-threading to parallelize across the outer loop iteration (line 3 in Algorithm 1) as it is difficult to extract MLP within the loop body. In a multi-threaded software, locks or other synchronization mechanism are often needed to guarantee atomicity when different threads try to access the same vertex, which incurs a large overhead. In comparison, this issue does not concern this accelerator as MLP is extracted across pipeline stages for different data type and requires no locking.

Improve On-Chip Data Reuse. To capture data reuse, we provide isolated on-chip buffers for the worklist, edge, and vertex data to minimize conflict misses. For the worklist item and edge data, since there exists a good amount of spatial locality, we allocated a small 64-byte register to buffer the loaded cacheline, which often contains multiple useful words and is repeatedly accessed. For the vertex data, we use a dedicated 512KB two-way associative read-write cache to capture the temporal locality among accesses. In addition, this accelerator has a bandwidth reduction feature that coalesces the stores to the same memory address. Overall, the design closely resembles the accelerator proposed in [11].

Support BFS. This accelerator design can also support BFS by changing how vertex labels are interpreted. Specifically, the guarding condition for relaxation in stage S4 in Figure 4 changes from a path-length comparison to an equal check to see if the vertex has been relaxed before, and the adder for computing new path lengths for a new relaxation is removed as BFS labels represent predecessors instead of distances.

IV. FPGA-Only Performance Baseline with and without Scheduling

In this section, we will perform three sets of studies for our FPGA-only baseline. After explaining the evaluation setups, in section IV.B, we demonstrate that our baseline is a reasonable reflection of standard FPGA-only based graph processing by comparing it with existing works. In section IV.C, we assess the theoretical benefit of worklist scheduling by assuming a zero-overhead scheduler in simulation. In section IV.D, we study the practicality of adding an FPGA-based scheduler next to the accelerator.

A. Evaluation Setup

In this paper, the experiments are conducted on top of a physical shared-memory FPGA platform and a hardware simulator. Except for the studies in section IV.C and IV.D, the evaluations were based on RTL implementations of the proposed accelerator for an Intel HARP.

Intel HARP features a cache-coherent integration of a Stratix V FPGA and a Xeon E5-2680 v2 processor at 2.8GHz. The FPGA has a maximum memory bandwidth of 7GB/second (bounded by QPI)[13] and 5MB of on-chip dual-ported BRAM. The Xeon has a peak memory bandwidth of 49 GB/second and 25 MB of three-level on-chip cache. We

name	vertices	edges	type	description
Rand_cluster	512K	8M	Directed, unweighted	Synthetic graph with 16 communities. Each edge has a 25% probability of connecting to a random vertex from a different community and a 75% probability of connecting internally.
RMAT_sparse	512K	1M	Directed, unweighted	Sparse RMAT graph [17] with default distribution parameters.
RMAT_dense	512K	8M	Directed, unweighted	Dense RMAT graph [17] with default distribution parameters.
circuit_4	80k	307k	Directed, unweighted	Circuit simulation graph from UF Sparse Matrix Collection [18].
us_roads	129k	165k	Directed, unweighted	Road network in USA from [18].
rgg_n_2_19	524k	6M	Undirected, unweighted	Synthetic RGG graph from [18].
cond-mat	40k	351k	Undirected, weighted	Collaboration network on the condensed matter archive [18].
road_FLA	1m	2.7M	Directed, weighted	Road network in FLA from [18]
Linux_call	324k	1.2M	Directed, weighted	Call graph of the linux kernel from [18]
webbase-1M	1M	3.1M	Directed, weighted	Web connectivity matrix from a Nvidia technical report [18]
Patent_main	240K	560K	Directed, weighted	Citation network for NBER patents [18]

Fig. 5. Benchmark graphs.

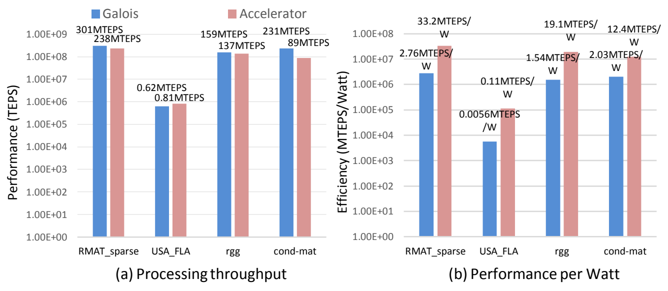


Fig. 6. Comparison of the baseline BFS accelerator with the Galois software BFS on HARP

implemented the baseline accelerator on the Stratix FPGA. The BRAM utilization is limited to 3 MB, and off-chip memory utilization is limited translating 128 8-MB pages, as the timing analysis fails when compiled with larger memory allocations. We used a set of synthetic[17] and standard networks[18] as the experiment inputs, which are summarized in Figure 5.

For the limit study presented in section IV.C and an analysis for integrating FPGA-based scheduling with the accelerator in section IV.D, which are difficult to realize on a physical platform, we used an in-house C-simulator of the system. The simulator features cycle-accurate modeling of the accelerator's pipeline stages and off-chip memory system, including caches and DRAM.

B. Performance of the Baseline FPGA-Only Accelerator

To ensure that our baseline FPGA-only accelerator delivers competitive performance, we evaluated the proposed accelerator using the FPGA of a physical Intel HARP. As we found it more difficult to obtain published data of competitive software baselines for SSSP, we will focus on the BFS accelerator in

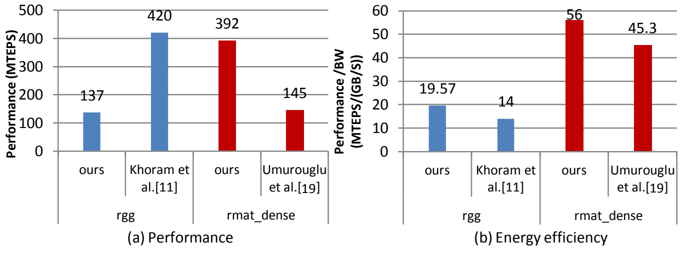


Fig. 7. Comparison of the baseline BFS accelerator with prior FPGA works.

this section. However, due to their algorithmic similarity, we expect the outcome to be similar for SSSP. For a set of selected graphs, we measured the computation throughput in the metric of *Traversed-Edges-per-Second (TEPS)* and compared the results with Galois multi-threaded BFS implementation on the Xeon E5-2680 v2. The results are shown in Figure 6.a. Overall, the accelerator delivers performance comparable to Galois except for Cond-Mat, for which Galois outperforms our baseline by roughly 40% as the accelerator experienced a relatively large number of conflict misses in its vertex cache.

In terms of energy efficiency, the accelerator is much more efficient across all inputs. The performance-per-watt measurement is shown in Figure 6.b. While the CPU consumes, on average, 108 watts, our accelerator delivers mostly comparable performance but consumes merely 7 watts, allowing the accelerator to outperform Galois by roughly 10.2x for MTEPS-per-watt. We compared our baseline accelerators with previously published BFS implementations for FPGAs [12], [19] with similar types of input graphs. Due to the platform differences, there exist gaps in the raw performance. After normalizing the performance measurements by the memory bandwidths of the respective platforms, the results are roughly comparable, as shown in Figure7.

Discussion. Despite the Xeon having a higher-bandwidth system compared to the FPGA, the software BFS implementation achieves only roughly the same performance as the accelerator, which implies memory resource under-utilization for the processor. Upon closer examination, we discovered that the processor fails to extract parallel memory reads from the instructions, as it issued only one load every 7.8 clock cycle on average. This implies severe underutilization of the CPU's memory interface, which can theoretically accept two loads per clocks cycle in each core.

The problem of memory interface underutilization is not unique to Galois, as we also observed the same problem in another BFS implementation [7]. One probable cause is the CPU's inability to parallelize loads across neighbor relaxations in the inner loop (lines 7 to 14 in Algorithm 2) due to the dependent instructions within the loop body.

C. Potential Benefit of Worklist Scheduling

To understand the theoretical ceiling of performance improvement from worklist scheduling, we conducted a simulation-based limit study for the locality-aware scheduling for BFS and work reduction scheduling for SSSP using our

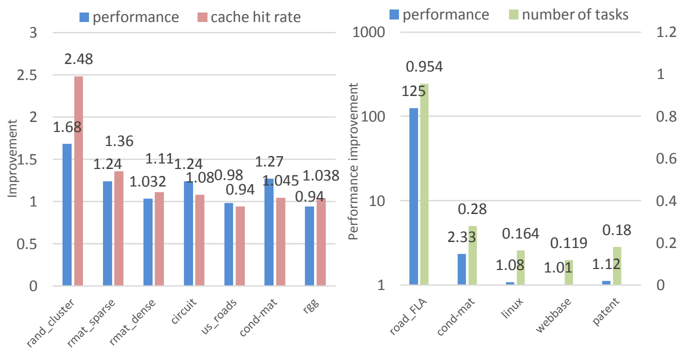


Fig. 8. Limit studies for the potential performance improvement of worklist scheduling.

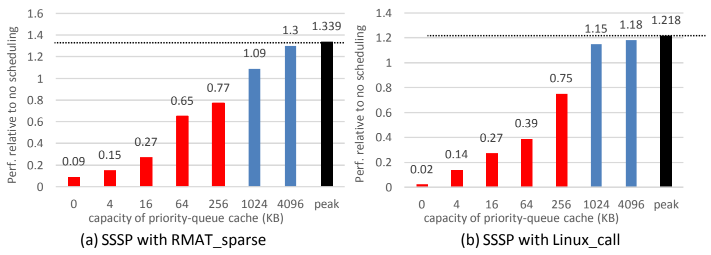


Fig. 9. Effect of caching to priority-queue on performance for two graphs. For BFS and SSSP, builds with cache smaller than 1MB results in performances lower than the non-scheduling baselines (highlighted in red).

simulator that models our baseline accelerator. We swapped the default FIFO-based worklist with a **zero-overhead priority queue** that supports **instantaneous item insertion and extraction**. By ignoring priority scheduling overheads such as the latency of insertion, the performance improvement brought by scheduling represents its maximum limit.

BFS. We first evaluated locality-aware scheduling for BFS with the non-scheduling accelerator as the baseline. We compared their cache hit rate and computation throughput, and the results are shown in Figure 8.a . We observed a peak 2.73x improvement in cache hit rate and 2.09x improvement in performance. The minimum improvement was 4% with 2% for cache hit rate and throughput, respectively. This wide difference is caused by the topological differences of the graphs. The graphs with clearly-separable, densely-interconnected community structures are more likely to benefit from the scheduling. On average, the cache hit-rate is improved by 35%, and the execution time is improved by 28.6%.

SSSP. For SSSP, the scheduler prioritizes task vertices with smaller distance labels. The reduction for the number of active vertices pushed to the worklist is shown in Figure 8.b. The work reduction shows a wide variation from 13% to over 99%, depending on the distribution of edge lengths and graph connectivity. In general, the graphs with wide variation in edge lengths and high vertex degrees tend to have larger opportunities. As the lowering of worklist inserts implies shorter processing time, the performance improvement is strongly correlated to the number of reduced works.

D. Adding Hardware Priority Queue to the Baseline FPGA-Only Accelerator

The limit study in section IV.B reveals promising potential in worklist scheduling for improving performance, if the priority-queue worklist has no performance and resource overhead. To understand this overhead for a hardware priority-queue implementation, we modified our simulator to model the BRAM tree, a binary-heap-based priority queue architecture from [20].

The original BRAM tree requires the entire priority-queue to be implemented using on-chip BRAM, which limits the capacity and prevents it from supporting the processing of large graphs. To resolve this issue, we modified the design to support the spilling of higher levels of the binary heap to off-chip DRAM when the tree size exceeds BRAM's storage capacity. This design essentially uses BRAM as a cache to buffer the items at the lower levels of a tree; thus, its performance would be dependent on the size of the BRAM allocated for caching, as spilling to off-chip DRAM incurs long latency.

We evaluated this design with different priority-queue cache sizes using the RMAT_sparse dataset [17] for BFS. The result is shown in Figure 9.a. The key observation is that a large amount of BRAM (enough to buffer 50% of the vertices) would be needed as the heap cache to offset the DRAM spilling overhead and break even in performance with the baseline BFS accelerator without scheduling. With less than a 1 MB heap cache, the scheduling overhead of accessing the priority-queue in memory actually lowers performance compared to the baseline. Similar results were observed when running SSSP with the Linux_call graph, as shown in Figure 9.b. This creates a problem in resource contention as the baseline accelerator also needs large amounts of on-chip storage to deliver competitive performance. More BRAM used for heap buffering implies less on-chip storage for caching graph vertices. This can impact the performance of the accelerator negatively, as the vertex cache hit rate lowers due to smaller capacity.

V. PROCESSOR-ASSISTED SCHEDULING ON SHARED-MEMORY FPGA PLATFORMS

Recently, platforms like the Intel HARP which offer coherent memory access for its the host processor and FPGA fabric have become available. The close integration of the two devices enables the possibility of fine-grain, heterogeneous collaboration. In this work, to get around the problem of BRAM contention for integrating worklist scheduling, we exploit this opportunity of collaboration for worklist scheduling. Specifically, our proposed processing paradigm, Processor-Assisted Scheduling (PAS), dynamically offloads the task of worklist scheduling to the host processor, which establishes a new use case for the shared-memory platforms.

A. Motivations for PAS

There have been a few prior attempts to exploit the close integration of the processor and the FPGA for collaborative

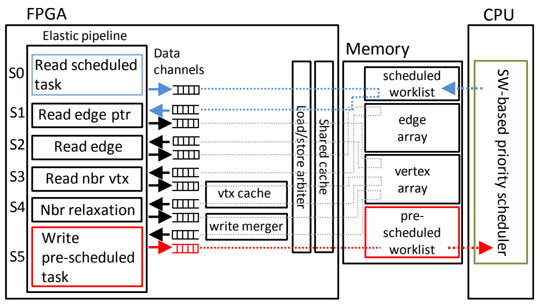


Fig. 10. Block diagram of Processor-Assisted Scheduling. Changes to the baseline accelerator are highlighted in colors.

graph processing, (e.g., [21], [19]). In both works, the main graph processing routines remained on the processor. In contrast, our proposed PAS solution maps the main routine of relaxation to an efficient FPGA accelerator, which delivers performance comparable to the processor while consuming much-lower power, as shown in the previous section, and offloads priority scheduling to a lightweight software scheduler that consumes a negligible load on the processor. By assigning the processor and the FPGA the tasks at which they perform best, PAS allows them to deliver better performance together than each can achieve alone.

To summarize, the benefits of PAS are as follows:

1) **No BRAM contention.** PAS does not integrate the priority-queue on the FPGA, which takes away large amounts of on-chip buffering of BRAM. Thus, the accelerator has full control over the utilization of BRAM for graph-data caching.
2) **Lower memory latency.** Modern processors have large caches with advanced features for minimizing misses. In addition, in the case of an Intel HARP, the host processor has significantly lower memory latency compared to its FPGA, as the memory requests do not have to traverse through the QPI interconnect like the FPGA does [13]. These features for low-latency accesses provide better priority-queue performance when traversing the non cached levels of the tree.
3) **Negligible processor load.** The worklist scheduling routine often stays idle, as the insertion of new tasks happens only once in a while. If we were to implement the scheduler using software, this property implies minimal interference with the other applications. We will demonstrate this property, which makes PAS an ideal technique for servers, later in section VI.
4) **Ease of implementation.** Compared to adding a hardware scheduler to the accelerator, the hardware changes needed for PAS are minimal, and implementing the scheduling routine as software is fundamentally simpler in comparison to designing it in RTL. In addition to the specific scheduling we implemented, it becomes possible to efficiently adopt from software other equally lightweight scheduling alternative [22].

Algorithm 3 Software priority scheduler in PAS

```
1: while accelerator is still running do
2:     sleep(PROBE_PERIOD);
3:     WL_header header=presch_WL[0];
4:     if header.valid then
5:         insert(presch_WL[1], header.numTasks, PQ);
6:         presch_WL=presch_WL+CLINE_SIZE;
7:     end if
8:     if (header.acc_idle() or (sch_timer = SCH_PERIOD)) and
       (!PQ.empty()) then
9:         insert(PQ, MIN(PQ.size(), CLINE_SIZE-1), sch_WL);
10:        sch_WL=sch_WL+CLINE_SIZE;
11:        sch_timer=0;
12:    else
13:        sch_timer=sch_timer+1;
14:    end if
15: end while
```

B. *Implementation of PAS*

To offload the task of worklist scheduling to the processor, we made several changes to the worklist writing (S5) and reading (S0) stages of the base BFS and SSSP accelerators. The new system architecture is shown in Figure 10. We split the worklist into two data structures: a **Pre-Scheduled Queue** and a **Scheduled Queue**; the software priority scheduler is shown in Algorithm 3.

The Pre-Scheduled Queue is a circular FIFO, with the accelerator being the producer and the software scheduler being the consumer. When new works are generated due to successful relaxation, they are inserted to the head position of the Pre-Scheduled Queue sequentially, just like the original worklist. The software scheduler periodically probes into the tail position of the queue to ascertain whether new works have arrived by checking a valid flag tagged to each cacheline in the Pre-Scheduled Queue. This tag is labeled by the accelerator when writing the new works to memory. When valid work is received, the software scheduler inserts those new tasks to a priority queue, which allows the efficient identification of high-priority tasks.

The Scheduled Queue is another circular FIFO with the software scheduler being the producer and the accelerator being the consumer. When the scheduler predicts that the accelerator needs new work to keep its pipeline busy, the high priority works will be extracted from the priority queue and then inserted into the Scheduled Queue, awaiting processing by the accelerator.

The timing of writing to the Scheduled Queue is important for performance. Writing too frequently causes constant draining of the priority queue without letting high-priority tasks to “bypass” to the front, which defeats the purpose of scheduling. Writing too infrequently can potentially leave the accelerator in starvation for tasks when the pipeline is idle. To tackle this problem, we introduce awareness to the idleness of the accelerator on top of a simple periodic scheduling protocol. In this protocol, the software schedules new tasks either periodically or when the accelerator signals its idleness. In the case of Intel HARP, which writes to memory at 64-byte

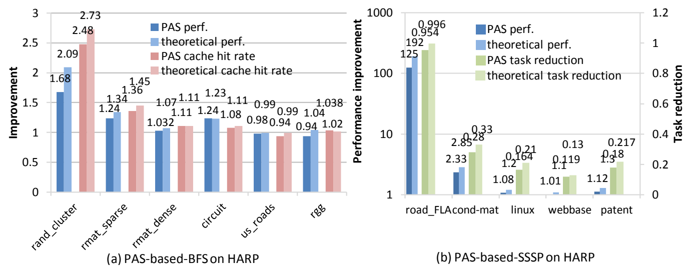


Fig. 11. Results for evaluation of PAS.

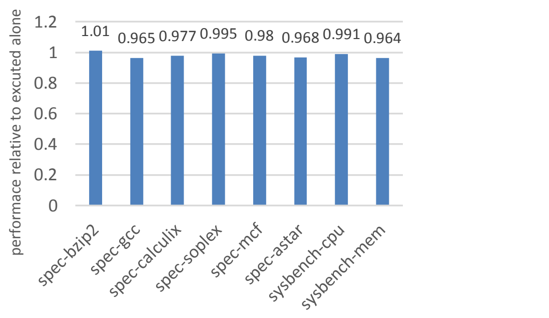


Fig. 12. Performance of CPU benchmarks running in parallel with PAS relative to running alone.

granularity, an idleness flag is tagged to every non-scheduled work cacheline by the accelerator. The 1-bit idleness flag is set when the edge pointer fetching stage (S2) for active vertices remains idle for multiple cycles due to lack of work, implying that it needs more scheduled tasks from the software scheduler.

VI. Evaluation

Performance. We evaluated PAS on a physical Intel HARP for both BFS and SSSP (not simulation). As PAS introduces extra memory access overhead for heterogeneous communication between the processor and the accelerator, we decided to first assess whether PAS remains practical on HARP by comparing it with the result in the no-overhead ideal case obtained using the simulation depicted in section IV.C. For BFS, this comparison is shown in Figure 11.a. On average, PAS improves the cache hit-rate by 29% and throughput improvement by 20%. In comparison to the limit study without scheduling overhead, the differences of the improvements are only around 5%, implying that PAS is capable of retaining most of the theoretical benefit of scheduling, even with the overhead from a physical platform.

For SSSP, the result is similar to BFS, as shown in Figure 11.b. We observed the overhead for SSSP to be slightly higher; this is likely due to the slightly more complex scheduling routine, which is based on the distance labels of vertices instead of just the indices of vertices like BFS. Overall, the difference between the limit study and the physical implementation of PAS-based SSSP is around 15%.

While at a glance, the scheduling overhead might be surprisingly small for the large amounts of memory probing, there is a simple explanation. The QPI interconnect of Intel HARP maintains coherency at the CPU's last-level cache (LLC). This means that when the accelerator updates the Pre-Scheduled Queue, the new tasks would be brought to the last-level cache directly for the processor to retrieve. This eliminates the long-latency off-chip memory accesses during task probing by exploiting a special form of locality between the accelerator and the scheduler.

Scheduler Load. To understand the software scheduler's processor-resource consumption, we characterized it using Intel PCM [23] and observed an extremely low 5% core utilization and near-idle 15.7-watt power consumption. We then assessed the scheduler's potential interference with other concurrent programs on the processor by dispatching PAS in parallel with a set of selected SPEC2006 [24] and Sysbench[25] benchmarks. All test cases are configured to 20 threads to fully exploit HARP's 10-core hyper-threading-enabled Xeon. The results, shown in Figure 12, indicate that the PAS scheduler causes minimal interference with the benchmarks. This characteristic implies that PAS can be a good processing paradigm for servers, as the light-weight scheduler is unlikely to impact the services to the other server clients.

VII. Conclusion

This paper presented our exploration of the worklist-scheduling optimization for graph algorithms on a shared-memory FPGA platform. We first developed an high-quality non-scheduling accelerator for BFS and SSSP. With this as the baseline, we analyzed the effectiveness of integrating priority scheduling on FPGA and discovered, besides adding logic complexity, its heavy consumption of BRAM and potential resource contention with the baseline accelerator.

To solve this problem, we developed Processor-Assisted Scheduling (PAS), which exploits the close integration of the processor and the accelerator on a shared-memory platform by offloading the task of worklist scheduling to the processor. In our evaluation, we case-studied a locality optimization for BFS and the task reduction scheduling of the Dijkstra-based SSSP. Benefitting from the processor's superior memory sub-system, the software scheduler delivers timely scheduling service for the accelerator, that is able to obtain over 90% and 80% of the theoretical scheduling performance benefit for BFS and SSSP, respectively. Additionally, the light-weight scheduler consumes minimal processor resources and does not interfere other concurrent applications. This approach allows the two CPU and the FPGA to focus on the tasks at which they perform best and collaboratively achieve better performance than each can alone deliver.

VIII. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1618014 and by an Intel Strategic Research Alliance (ISRA) award. We thank Intel for donation of the FPGA hardware and tools used in this research.

References

[1] B. J. Mirza, B. J. Keller, and N. Ramakrishnan, "Studying recommendation algorithms by graph analysis," *J. Intell. Inf. Syst.*, vol. 20, no. 2, pp. 131–160, Mar. 2003. [Online]. Available: https://doi.org/10.1023/A:1021819901281

[2] T. Wang, Y. Chen, Z. Zhang, T. Xu, L. Jin, P. Hui, B. Deng, and X. Li, "Understanding graph sampling algorithms for social network analysis," in *2011 31st International Conference on Distributed Computing Systems Workshops*, June 2011, pp. 123–128.

[3] A. Shokoufandeh and S. Dickinson, "Theoretical aspects of computer science," G. B. Khosrovshahi, A. Shokoufandeh, and A. Shokrollahi, Eds. New York, NY, USA: Springer-Verlag New York, Inc., 2002, ch. Graph-theoretical Methods in Computer Vision, pp. 148–174. [Online]. Available: http://dl.acm.org/citation.cfm?id=644608.644614

[4] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," *SIGPLAN Not.*, vol. 46, no. 6, pp. 12–25, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/1993316.1993501

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959. [Online]. Available: http://dx.doi.org/10.1007/BF01386390

[6] N. Beckmann and D. Sanchez, "Cache-guided scheduling exploiting caches to maximize locality in graph processing," in *1st International Workshop on Architectures for Graph Processing*, 2017.

[7] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1214–1225, Jul. 2015. [Online]. Available: http://dx.doi.org/10.14778/2809974.2809983

[8] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," *SIGPLAN Not.*, vol. 51, no. 8, pp. 11:1–11:12, Feb. 2016. [Online]. Available: http://doi.acm.org/10.1145/3016078.2851145

[9] P. Gupta, "Xeon+fpga platform for the data center," June 2015. [Online]. Available: https://www.archive.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf

[10] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *CoRR*, vol. abs/0810.1355, 2008. [Online]. Available: http://arxiv.org/abs/0810.1355

[11] S. Khoram, J. Zhang, M. Strange, and J. Li, "Accelerating graph analytics by co-optimizing storage and access on an fpga-hmc platform," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: ACM, 2018, pp. 239–248. [Online]. Available: http://doi.acm.org/10.1145/3174243.3174260

[12] J. Zhang and J. Li, "Degree-aware hybrid graph traversal on fpga-hmc platform," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: ACM, 2018, pp. 229–238. [Online]. Available: http://doi.acm.org/10.1145/3174243.3174245

[13] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern cpu-fpga platforms," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 109:1–109:6. [Online]. Available: http://doi.acm.org/10.1145/2897937.2897972

[14] ISS Universtiy of Texas at Austin, "Descrete event simulator," (Date last accessed 2018-10-15). [Online]. Available: http://iss.ices.utexas.edu/?p=projects/galois/benchmarks/discrete_event_simulation

[15] ——, "Asynchronous variational integrator," (Date last accessed 2018-10-15). [Online]. Available: http://iss.ices.utexas.edu/?p=projects/galois/benchmarks/asynchronous_variational_integratorsm

[16] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," *CoRR*, vol. abs/1006.4990, 2010. [Online]. Available: http://arxiv.org/abs/1006.4990

[17] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," vol. 6, 04 2004.

[18] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

[19] Y. Umuroglu, D. Morrison, and M. Jahre, "Hybrid breadth-first search on a single-chip fpga-cpu heterogeneous platform," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2015, pp. 1–8.

[20] M. Huang, K. Lim, and J. Cong, "A scalable, high-performance customized priority queue," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2014, pp. 1–4.

[21] S. Zhou and V. K. Prasanna, "Accelerating graph analytics on cpu-fpga heterogeneous platform," in *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct 2017, pp. 137–144.

[22] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *The 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[23] T. Willhalm, R. Dementiev, and P. Fay, "Intel performance counter monitor - a better way to measure cpu utilization," (Date last accessed 2018-10-15). [Online]. Available: https://software.intel.com/en-us/articles/intel-performance-counter-monitor

[24] Standard Performance Evaluation Corporation, "Spec cpu 2006," (Date last accessed 2018-10-15). [Online]. Available: https://www.spec.org/cpu2006/

[25] "Sysbench," (Date last accessed 2018-10-15). [Online]. Available: https://wiki.gentoo.org/wiki/Sysbench