Tatum: Parallel Timing Analysis for Faster Design Cycles and Improved Optimization

Kevin E. Murray
Department of Electrical and Computer Engineering
University of Toronto, Ontario, Canada
kmurray@eecg.utoronto.ca

Vaughn Betz
Department of Electrical and Computer Engineering
University of Toronto, Ontario, Canada
vaughn@eecg.utoronto.ca

***Abstract*—Static Timing Analysis (STA) is used to evaluate the correctness and performance of a digital circuit implementation. In addition to final sign-off checks, STA is called numerous times during placement and routing to guide optimization. As a result, STA consumes a significant fraction of the time required for design implementation; to make progress reducing FPGA compile times we need faster STA. We evaluate the suitability of both GPU and multi-core CPU platforms for accelerating STA. On core STA algorithms our GPU kernel achieves a 6.2 times kernel speed-up but data transfer overhead reduces this to 0.9 times. Our best CPU implementation achieves a 9.2 times parallel speed-up on 32 cores, yielding a 15.2 times overall speed-up compared to the VPR analyzer, and a 6.9 times larger parallel speed-up than a recent parallel ASIC timing analyzer. We then show how reducing the run-time cost of STA can be leveraged to improve optimization quality, reducing critical path delay by 4%.**

Keywords*—*Static Timing Analysis (STA), Parallel Algorithms, GPU, multi-core CPU, FPGA, Computer-Aided Design (CAD)

I. Introduction

A common goal when designing a digital circuit is maximizing its performance; as a result a circuit is analyzed repeatedly during design to determine its operating frequency.

The most common approach to determine a digital circuit's speed is Static Timing Analysis (STA). While STA is significantly faster than other approaches such as timing simulation, it is still time-consuming. Consequently designers and optimization tools often opt to sacrifice accuracy, by performing STA only 'occasionally' during the design process, to minimize design iteration times. Despite this a placement tool like VPR will call STA hundreds of times during optimization [1]. However this still means design decisions are made using stale (old and possibly now incorrect) timing information. This leads designers and optimization algorithms (whose decisions can benefit from accurate timing information) to assume unnecessarily pessimistic design conditions – resulting in costly over-design.

Furthermore, design sizes continue to increase rapidly [2], while improvements in single-threaded CPU performance have slowed [3]. Additionally, the number of timing analyses required to fully characterize a design is also increasing due to the proliferation of timing corners [4], and the growing number of clock domains [5]. As a result, in commercial FPGA place and route tools STA typically takes 25% of total run-time, but may *dominate the optimization algorithms* when designs have multiple clocks and timing constraints [6]. Moreover, modern FPGAs have performance-driven architectural features such as pulsed latches [7] and interconnect registers [8], which exacerbate hold-time issues. This requires additional minimum-delay timing analyses to evaluate, and design tools to explicitly optimize for hold-time; requiring numerous rapid calls to STA [9]. Finally, a variety of performant parallel algorithms have been proposed for FPGA placement [3], [10], [11], [12] and routing [13], [14], [15], [16]. As these parallel approaches speed-up the core optimization algorithms, timing analysis becomes an increasingly dominant portion of run-time – limiting the achievable speed-up.[1] These factors all make the development of fast and scalable timing analysis algorithms, which can exploit the parallelism available from modern computing systems, key to reducing FPGA design times.

In this paper we focus on the problem of developing efficient and scalable parallel algorithms for block-based STA. Our contributions include:

- Memory layout optimizations for STA data structures,
- Parallel algorithms for block-based STA and evaluations on GPUs and multi-core CPUs,
- Techniques to efficiently perform multi-corner and multi-clock STA,
- Tatum, an open-source reusable STA library built on our best algorithms,
- Analysis and comparison of Tatum with existing serial and parallel STA tools, and
- An evaluation of Tatum's performance when integrated in an existing placement tool (VPR) to guide its optimization.

Section II discusses background and related work. Sections III to V present our GPU and CPU parallel algorithms and evaluates them within a simplified STA formulation. Sections VI and VII describe our methods for speeding-up multi-corner and multi-clock STA, and Section VIII describes and evaluate our best algorithms within a full-featured STA formulation. Section IX illustrates how fast STA can be exploited to improve optimization quality. The conclusion and future work are presented in Section X.

II. Background & Related Work

Related work can be divided into two components: STA and its related extensions, and parallel algorithms.

A. STA Background

Timing analysis checks whether a digital circuit will operate correctly, given a set of timing constraints. A typical timing constraint is the clock period, which requires signals to be stable before the active clock edge to avoid latching stale data or inducing meta-stability in data storage elements.

[1] Most works on parallel placement and routing do not account for time spent on STA when reporting speed-ups. However STA's impact is significant. For instance, at 25% of total run-time STA limits the best-case speed-up of any parallel placement or routing algorithm to only $4\times$ (Ahmdal's law).

STA [17] is the conventional approach for timing analysis. STA operates on a timing graph, an abstract representation of a digital circuit where nodes represent the pins of circuit elements and uni-directional edges represent the timing dependencies between them. We denotes the number of nodes in the timing graph as N, and the number of levels after topological sorting as L. Two classes of STA algorithms have been proposed: path-based and block-based. Path-based algorithms perform a detailed analysis of every path in a circuit – offering high accuracy, but with worst-case exponential run-time. As a result block-based algorithms, whose computation time grows linearly with the size of the circuit, are usually used despite their more pessimistic analysis.

A more recent development has been Statistical STA (SSTA) [18]. Rather than calculating scalar delays, SSTA calculates delay probability distributions to capture the delay impact of manufacturing process variation. SSTA can be applied with either path-based or block-based algorithms, and calculated either analytically, or by Monte Carlo methods.

Due to their lower computational complexity many industrial design flows, and most optimization tools, use block-based algorithms. We focus on block-based STA, but our techniques can be directly extended to block-based SSTA.

B. Parallel Timing Analysis

Several approaches have been taken to parallelize timing analysis on CPUs. Distributed approaches partition the circuit into pieces which can be processed by multiple machines [19], [20], using message passing for synchronization and communication. Scalability is typically limited by the partitioning overhead [20]. In [21] an enhanced sampling method for Monte-Carlo based SSTA is presented, which is trivially parallelized. OpenTimer [22] presents a static timing analyzer using pipeline parallelism, and is evaluated in Section VIII-A.

Several works have looked at accelerating timing analysis on GPUs. A path-based STA engine, formulated as a Sparse-Matrix Vector Product problem is presented briefly in [23], however it requires $O(N^2)$ memory for a circuit of size N, which limits scalability. [24] describes a GPU accelerated path-based SSTA implementation using the Monte Carlo approach. However, path-based Monte Carlo SSTA is very computationally expensive making it impractical in many industrial design flows, and too expensive to call repeatedly in an optimizer. [25] describes a GPU SSTA algorithm using a dynamic batch construction algorithm. CASTA [26] presents a GPU-based STA implementation and is compared to our approach in Section IV-B.

Unlike previous work, we develop multiple parallel CPU algorithms, account for data transfer when evaluating GPUs, consider simultaneous multi-corner and multi-clock analysis, and evaluate our algorithms in an optimization tool on a variety of large benchmark circuits.

III. Baseline Implementation

It is a large effort to develop a full featured timing analysis engine for each parallel algorithm and compute platform. Hence we first consider a simplified timing analysis formulation which captures the key algorithmic characteristics of STA. We develop several STA engine variants to compare a wide variety of parallel approaches. These simplified engines calculate node arrival/required times for single-clock circuits using a pre-calculated delay model. In Section VIII we create and evaluate a full featured parallel timing engine (Tatum) built on the most promising approach, which supports multiple-clocks and additional timing constraints like false and multicycle paths.

TABLE I. Performance of Memory Optimizations

Data Layout	Speed-Up
AoS	1.00
SoA	1.44
SoA & Re-Order Edges	1.61
SoA & Re-Order Nodes	3.98
SoA & Re-Order Edges + Nodes	5.40

Geomean across `neuron`, `openCV`, `denoise`, and `gaussianblur` benchmarks.

The baseline CPU algorithm is shown in Algorithm 1.

Algorithm 1 Baseline Serial Algorithm

```
Require: G levelized timing graph to analyze
1: function SerialWalk(G)
2:   L ← NumLevels(G)
3:   for ℓ ∈ (0 ··· L − 2) do                    ▷ Forward traversal
4:     for node ∈ LevelNodes(G, ℓ) do
5:       FwdTraverseNodePush(node)
6:   for ℓ ∈ (L − 2 ··· 0) do                    ▷ Backward traversal
7:     for node ∈ LevelNodes(G, ℓ) do
8:       BwdTraverseNodePull(node)
```

The timing graph (levelized by a topological sort) is first traversed in the forward direction one level at a time (Line 3), ensuring predecessor nodes have valid arrival times. FwdTraverseNodePush, which walks the out-going edges to update the arrival time of downstream nodes, is then called on each node in the level (Line 5). The graph is traversed similarly in the backward direction to calculate required times (Lines 6 to 8).

A. Memory Layout Optimizations

The graph traversals in Algorithm 1 have a relatively small amount of computation (arrival/required time calculation) compared to data access (graph nodes and edges). As a result, memory access patterns and caching behaviour are key to achieving high performance. We implemented several optimizations to improve this behaviour:

Struct-of-Arrays We change the memory layout from Array-of-Structs (AoS) (e.g. where a node's arrival time, required time and out-going edges are located in adjacent memory) to Struct-of-Arrays (SoA) (e.g. where the arrival times of all nodes are located in adjacent memory).

Re-order Edges We re-order timing graph edges to match the expected traversal order based on the levelization.

Re-order Nodes We similarly re-order timing graph nodes.

Table I shows the impact of these optimizations, which improve spatial and temporal locality of the timing graph. Individually, SoA layout and re-ordering edges provide moderate improvements, while re-ordering nodes causes a significant improvement. These optimizations combine to yield an overall $5.4\times$ speed-up.[2] All following comparisons are made with this optimized implementation unless otherwise noted.

IV. Parallel STA on GPUs

GPUs are a popular platform for compute acceleration, due to their potential high performance if many threads can be utilized effectively. GPU accelerated FPGA placement [12] and routing [16] techniques have been proposed, so evaluating their ability to accelerate STA is warranted. A GPU consists of a

[2]Re-ordering nodes and edges is done once based on the timing graph structure, and takes less time than a full timing analysis. Therefore the re-ordering run-time overhead is trivial in CAD tools like VPR which perform hundreds of timing analyses.

large number of simple compute units, which execute a unique thread but usually share a common control flow [27]. Subsets of the compute units can communicate quickly using a shared local memory, while compute units outside the subset must communicate through slower global memory [27].

A. GPU Algorithm

To evaluate the suitability of GPUs, we focus on accelerating the timing graph traversals. These traversals are the most time-consuming part of STA, accounting for $\sim 90\%$ of the run-time in the optimized CPU implementation. We accelerate the forward traversal (the backward traversal is similar).

The timing graph is processed one level at-a-time using two kernels as shown in Figure 1. In the segmented additive map kernel, each thread is assigned a single edge and calculates the edge arrival time by adding the source node arrival time (Level $i-1$) and the edge delay. Next, in the segmented max reduction kernel, each thread is assigned a sink node (at Level i) and performs a maximum reduction over all incoming edge arrival times.

The key benefit of this approach is the exposure of a large amount of parallelism to the GPU when calculating edge arrival times. This results in better load-balancing compared to processing the timing graph in a strictly node-at-a-time manner, since the work done per edge is effectively constant. This also improves performance by reducing code divergence (divergent branching), which is inefficient on GPUs [27]. The edge and node data is re-ordered as in Section III-A to ensure coalesced memory accesses, and speculatively loaded into local memories for fast access.

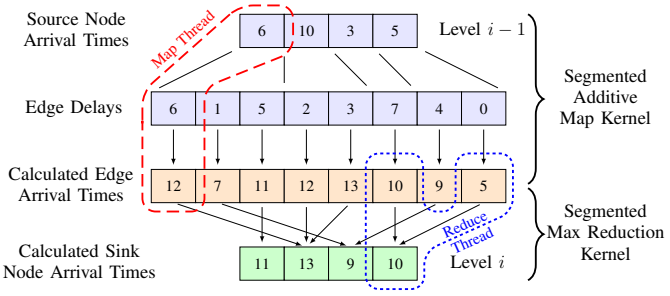

Fig. 1. GPU calculation of level i arrival times.

B. GPU Experimental Results

1) Experimental Methodology: Both the CPU and GPU implementation are evaluated using a selection of large benchmarks from the Titan FPGA benchmark suite [28] run on an Intel Q9550 (45nm 'Yorkfield') processor, and NVIDIA GTX780 (28nm 'GK110') GPU. The GPU algorithm is implemented using OpenCL, and minimizes wall-clock time by overlapping computation and data transfer. Reported run-times are the sum over all kernel invocations required to evaluate the timing graph, and are averaged over 100 runs.

2) Experimental Results: Table II shows the performance results. Comparing first the CPU and GPU Kernel times, we observe the GPU significantly outperforms the CPU by 6.2×. However, taking into account the time spent on data transfer between the host CPU and GPU, the GPU implementation is slower, taking 12% longer to complete. The data transfer overhead is significant and dominates overall run-time, on average accounting for 85% of the GPU's wall-clock time.

On a per-benchmark basis it is clear that the GPU implementation's performance varies widely. This is largely tied to the size of the benchmark and the number of levels in the timing graph. The GPU implementation performs better on larger benchmarks, with the `bitcoin_miner` benchmark achieving the largest speed-up (2.0× overall, 24.5× kernel only). Compared to the other benchmarks `bitcoin_miner` has substantially fewer (and wider) levels in the timing graph. This increases the work per kernel invocation, improving GPU utilization and providing more opportunity to overlap computation and data transfer.

TABLE II. GPU PERFORMANCE

Benchmark	CPU Opt. (P=1)	GPU Kernel		GPU Total		Transf. Frac.
neuron	0.014	0.006	(2.33×)	0.031	(0.45×)	0.81
openCV	0.046	0.014	(3.30×)	0.077	(0.60×)	0.82
bitcoin_miner	0.179	0.007	(24.46×)	0.090	(1.97×)	0.92
sparcT1_chip2	0.124	0.013	(9.52×)	0.107	(1.16×)	0.88
LU230	0.100	0.020	(5.02×)	0.108	(0.92×)	0.82
GEOMEAN	0.068	0.011	(6.17×)	0.076	(0.89×)	0.85

Time in seconds, speed-up relative to optimized CPU baseline in brackets.

TABLE III. GPU SPEED-UP COMPARISON

	GPU Kernel vs CPU Baseline (P=1)	GPU Kernel vs CPU Opt. (P=1)	GPU Total vs CPU Opt. (P=1)
This Work	33.3×	6.2×	0.9×
CASTA [26]	12.9×		

Blank values unreported.

Data transfer dominates the compute time, even though the GPU implementation overlaps computation and data transfer. The small amount of computation and large data transfer time means there is relatively little opportunity for overlap to occur. The largest amount of overlap occurs on `bitcoin_miner` where overlapping data transfer reduced wall clock time by 11%.

Table III summarizes the achieved GPU speed-ups and compares them with [26]. Comparing only GPU Kernel execution time, our method achieved a larger speed-up (33.3× vs 12.9×) than [26] when compared to baseline CPU implementations. However using an optimized CPU implementation (Section III-A) our achieved speed-up shrinks to 6.2×. This is a smaller speed-up than [26] but is against a more realistic baseline.[3] Including data-transfer times (GPU Total) further decreases our speed-up to 0.9×. Data-transfer times are important, since they dominate the Kernel execution time, but are not discussed in [26].

3) Conclusion: While our results show that GPUs are amenable to accelerating the core kernels of block-based STA, the data transfer overhead dominates the computation resulting in a net slow down. Fundamentally, unlike many popular algorithms accelerated by GPUs, block-based STA is a linear-time algorithm – requiring a linear amount of data to be transferred. This limits (in the Ahmdal's law sense) the overall performance of the GPU.[4]

Given that our GPU implementation considered only a simplified STA kernel, a more full-featured STA engine (supporting multiple-clocks, more complex delay models and timing exceptions) would likely perform worse, due to increased code-divergence, increased load-imbalance between threads and additional data transfer overhead.

[3]The CPU baseline in [26] contained GPU specific modifications, and was not optimized for CPU execution.

[4]Even if the GPU kernel run-time was reduced to zero the average speed-up would be only 1.05× for the benchmarks in Table II.

V. Parallel STA on CPUs

As shown in Section IV, while GPUs show promising performance on the core computational kernels, the overhead of marshalling data between the CPU and GPU is prohibitively expensive. We now focus on parallel STA algorithms targeting multi-core CPUs. By parallelizing on a CPU we avoid the data-transfer overhead associated with an off-chip accelerator. This also allows easier integration with existing CPU-based data structures and optimization algorithms, which can call STA a very large number of times.

A. CPU Algorithm A: Levelized Locks

The first parallel algorithm is a parallel extension of the serial baseline (Algorithm 1). The timing graph is still processed one level at a time, but the nodes within each level are processed in parallel. To avoid race-conditions while updating arrival times (when a node pushes arrival time updates to its successors), we synchronize access using fine-grained per-node locking. There is no race-condition when updating node required times as the uni-directional timing graph ensures only one thread updates a node's required time (by pulling the required time updates from its predecessors).

B. CPU Algorithm B: Levelized No-Locks

The observation that the parallel backward traversal can calculate required times without explicit locking synchronization, suggests the same can be done while calculating arrival times during the forward traversal. This would reduce synchronization costs and improve scalability. However, to do this efficiently, we must modify the timing graph to be *bi-directional*; each edge in the graph must store both its source and sink nodes. The memory costs associated with this are small and evaluated in Section V-D.

The improved levelized walk algorithm is shown in Algorithm 2. During the forward traversal the levels are processed

Algorithm 2 Levelized walk with no locks

```
Require: G levelized timing graph to analyze
1: function ParallelLevelizedWalkNoLocks(G)
2:   L ← NumLevels(G)
3:   for ℓ ∈ (1 ··· L − 1) do                    ▷ Forward traversal
4:     parallel_for node ∈ LevelNodes(G, ℓ) do
5:       FwdTraverseNodePull(node)
6:   for ℓ ∈ (L − 2 ··· 0) do                    ▷ Backward traversal
7:     parallel_for node ∈ LevelNodes(G, ℓ) do
8:       BwdTraverseNodePull(node)
```

serially (Line 3), but nodes within the level are processed in parallel (Line 4) using a *pull* based arrival time calculation.

Algorithm 3, updates the arrival time of a node. It iterates through a node's input edges (Line 2) and uses the bi-directional timing graph to identify the source node driving the edge (Line 3). It then calculates the arrival time from that edge (Line 4) and updates the arrival time for the current node (Line 5).[5] Since only a single thread updates a node no locks are required for synchronization.

C. CPU Algorithm C: Dynamic

While the levelized algorithm presented in Section V-B eliminates explicit locking, it still relies on implicit barriers after each **parallel_for** in Algorithm 2 to synchronize across levels.

To avoid this per-level synchronization we developed a dynamic graph walk algorithm. Instead of walking the graph in a strictly levelized manner, each node is processed only when its dependencies are satisfied. Once a node's dependencies are satisfied, it is added to a queue of nodes to be processed. Worker threads then pop nodes off the queue to process.[6] Rather than use high-overhead locks we opt for a lock-free approach, using an atomic compare-and-swap operation to avoid race conditions when updating dependencies.

[5]For SSTA, statistical addition/max operators would be used.

Algorithm 3 Forward node traverse with no locks

```
Require: n_snk the timing graph node to process
1: function FwdTraverseNodePull(n_snk)
2:   for e_in ∈ InEdges(n_snk) do
3:     n_src ← Source(e_in)                      ▷ Bidir. edge to predecessor
4:     α ← Arr(n_src) + Delay(e_in)
5:     Arr(n_snk) ← Max(Arr(n_snk), α)           ▷ Single writer
```

D. CPU Experimental Results

1) Experimental Methodology: The various STA algorithms are evaluated on the same benchmarks used in Section IV-B. All algorithms are implemented in C++, and run on an Intel Xeon E5-1620 (32nm 'Sandy Bridge', 4 cores). Results are the average across 100 runs.

It should be noted that the simplified timing analysis formulation evaluated here performs limited work per-node. Therefore the synchronization overhead is high, limiting the absolute parallel speed-ups. A more full-featured timing analyzer which performs more computation per node will scale better (as shown in Sections VI and VIII). However, this pessimistic scenario allows us to differentiate and evaluate the relative scalability of these algorithms in a high stress setting.

2) Results: The performance of the parallel algorithms are shown in Table IV.

The 'Levelized Locks' algorithm (Section V-A) achieves only limited speed-up (1.13×). Closer examination showed the forward traversal achieved no parallel speed-up – the synchronization enforced by locking effectively serialized the traversal.

By avoiding locks with a bi-directional timing graph, 'Levelized No-Locks' (Section V-B) produces better results, achieving an average speed-up of 1.92×. Avoiding the synchronization cost of locks (by ensuring only a single writer per-node) improves scalability.

The 'Dynamic algorithm' (Section V-C) performs significantly worse, running 4× *slower* than the serial algorithm. The cost of tracking when a node's dependencies are satisfied, and enqueuing nodes to be processed is substantial and dominates the computation.

To determine how close these algorithms come to the best-case performance we implemented an additional 'No Dependencies' algorithm. This algorithm performs the same amount of work as the other algorithms, but ignores the dependencies between timing graph nodes.[7]

The 'No Dependencies' algorithm achieves an average speed-up of 2.32×. This speed-up is less than linear, indicating that memory (rather than computation and synchronization) is the remaining bottleneck. The 'Levelized No-Locks' algorithm achieves a speed-up only 21% less than 'No Dependencies', showing the levelized approach can extract nearly all of the parallelism available. The remaining 21% difference consists

[6]This is performed efficiently using Cilk Plus task spawning [29], which uses per-thread queues and work-stealing to avoid load imbalance.

[7]While 'No Dependencies' does not produce a correct analysis (due to race-conditions), it produces an upper-bound on the achievable performance since no synchronization is performed.

TABLE IV. SPEED-UP OF CPU PARALLEL ALGORITHMS

Benchmark	Levelized Locks (P=4)	Levelized No-Locks (P=4)	Dynamic (P=4)	No Dep. (P=4)
neuron	0.98	1.70	0.26	2.21
openCV	1.09	1.87	0.28	2.43
bitcoin_miner	1.22	2.04	0.27	2.37
sparcT2_chip2	1.25	1.94	0.27	2.31
LU230	1.13	2.06	0.18	2.31
GEOMEAN	1.13	1.92	0.25	2.32

Speed-up relative to (serial) optimized CPU baseline.
P is the number of processor cores.

TABLE V. TIMING GRAPH MEMORY USAGE

Benchmark	Uni-Directional Edges		Bi-Directional Edges	
neuron	63.4	(0.72%)	103.8	(1.19%)
opencv	147.7	(0.71%)	237.9	(1.14%)
denoise	125.4	(0.87%)	204.0	(1.41%)
gaussianblur	645.1	(0.78%)	1,071.6	(1.30%)

Memory in MB. Percentage of VPR's memory in brackets.

of both the effect of the timing graph's dependency structure which can not be parallelized, and the overhead of using implicit barriers for synchronization.

The memory required to store bi-directional and uni-directional edges in the timing graph is shown in Table V. While storing bi-directional edges increases the timing graph size by 63% on average, the timing graph is a small fraction of a CAD tool's memory usage. For instance in VPR this corresponds to an increase of less than 0.5%. Combined with the improved speed-ups achieved with 'Levelized No-Locks' this represents a good trade-off.

VI. MULTI-CORNER TIMING ANALYSIS

An additional challenge facing digital design is the rapid increase in the number of timing corners to analyze [4], both for sign-off and to drive optimization in multi-corner aware tools [30]. From a computational perspective analyzing multiple corners simultaneously should be beneficial, since it increases the amount of work performed per node, better amortizing the data access and synchronization costs. Results are again collected on a 4-core Intel Xeon E5-1620 processor.

A. Parallel

Figure 2 illustrates the impact of increasing the number of timing corners (T) on run-time.

All parallel algorithms exhibit similar slopes which are substantially lower than those of the 'Serial' algorithm, meaning their speed-ups improve as T (work per node) increases. For instance the 'Levelized No-Locks' algorithm improves its speed-up from $1.7\times$ at $T = 1$ to $3.8\times$ at $T = 32$ (a nearly linear speed-up with 4 cores).

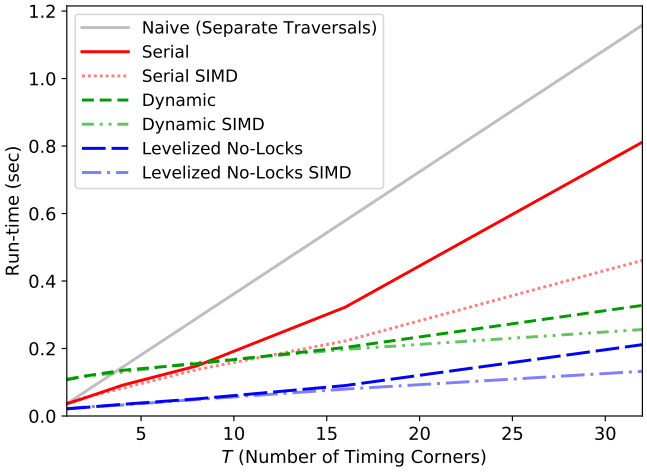

Fig. 2. Multi-corner STA performance for `openCV` at $P = 4$.

The 'Dynamic' algorithm has substantial synchronization overhead (Section V-D), and is always dominated by 'Levelized No-Locks' The per-node synchronization performed by the 'Dynamic' algorithm substantially outweighs the benefit of avoiding the per-level barriers used by 'Levelized No-Locks'.

It is interesting to note that performing a combined multi-corner analysis is inherently more efficient than the 'Naive' approach of traversing the timing graph for each corner analyzed. For example, the 'Naive' method runs $32\times$ longer at $T = 32$ compared to $T = 1$, while the 'Serial' algorithm at $T = 32$ runs only $22.4\times$ longer (since it traversed the timing graph only once). The difference is more substantial for the parallel algorithms, since their scalability improves as the amount of work increases. For instance, 'Levelized No-Locks' at $T = 32$ runs $8.7\times$ faster than 'Naive'.

B. SIMD

Processing multiple timing corners in parallel also exposes potential new data-level parallelism, since each basic operation (addition, max etc.) can be applied element-wise to a vector of values. This can be performed efficiently using the Single Instruction Multiple Data (SIMD) instructions found on many modern processors [27]. The Intel Xeon E5-1620 processor targeted supports 8-way SIMD on 32-bit values.

Using SIMD further improves scalability, flattening out the slopes of all algorithms in Figure 2. This results in larger speed-ups compared to 'Serial', with the 'Dynamic SIMD' and 'Levelized No-Locks SIMD' algorithms achieving speed-ups of $3.2\times$ and $6.2\times$ respectively at $T = 32$ using 4 cores.

C. Analysis within Fixed CAD Run-Time Budgets

In practise STA is often performed with a fixed CAD run-time budget. This could be a fixed portion of run-time dedicated to STA within an optimization tool, or a design engineer's constraint to evaluate a proposed design change (e.g. overight). In both cases the thoroughness of the analysis is limited by what can fit within the run-time budget.

Since the scalability of the parallel techniques improve as work-load increases this changes the trade-offs. For instance, in the same fixed-time budget required by 'Naive' to analyze 2.2 corners, 'Serial' analyzes 7.1 corners, 'Levelized No-Locks' analyzes 24.6 corners, and 'Levelized No-Locks SIMD' analyzes 32 corners. This means with 4 cores, 'Levelized No-Locks' and 'Levelized No-Locks SIMD' can respectively analyze $11.2\times$ and $14.5\times$ more corners than 'Naive', enabling designers and optimization tools to perform a much more detailed analysis than would otherwise be possible.

VII. MULTI-CLOCK TIMING ANALYSIS

So far, we have only considered timing analysis with a single clock. We now consider a full featured timing analyzer which supports advanced features such as multi-clock timing analysis and timing exceptions. Modern designs make use of a large number of clock domains, for instance, FPGA designs can use tens to hundreds of clock domains [5]. As a result, performing efficient timing analysis in the presence of multiple clock domains is important [6].

The *multi-traversal approach* to multi-clock timing analysis traverses the timing graph for each pair of launch and capture clock domains. However, as discussed in Section III-A, the memory accesses to traverse the timing graph are expensive.

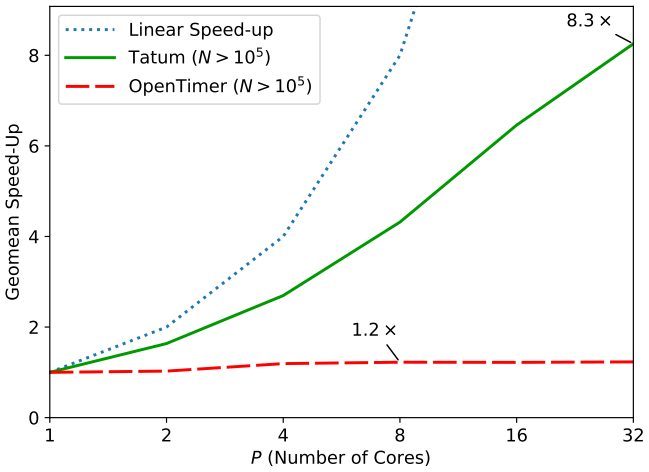

Fig. 3. Self-relative parallel speed-up on Tau'15.

A better method is the *tagged approach* where the timing graph is traversed only once (reducing memory access costs), with all clock domains handled simultaneously. This requires tagging each arrival/required time with the associated clock domain, and propagating them through the timing graph.

While the tagged approach has the same asymptotic computational complexity as the multi-traversal approach, it is more efficient by a constant factor and performs better in practise. Furthermore, the tag-based approach makes it straight forward to efficiently analyze clocks localized to only a portion of the timing graph (avoiding useless work), and to handle timing exceptions (such as inter-domain false paths). We evaluate the performance of the tag-based approach in Section VIII-B.

VIII. Tatum

Guided by the preceeding results, we developed Tatum, an open-source C++ library suitable for integration with CAD tools. Tatum provides facilities for application defined timing graphs and delay calculation.[8] These capabilities facilitate the creation of timing-aware CAD tools and the development of timing-aware optimization algorithms. Tatum supports multi-clock timing analysis and timing exceptions.

Tatum, uses the most performant CPU parallel algorithm ('Levelized No-Locks' from Section V-B), uses the tagged approach (Section VII) to perform multi-clock timing analysis, and uses Cilk Plus [29] for parallelization.

A. Comparison to OpenTimer

To evaluate the efficacy of Tatum's parallelization methods we compare its scalability with OpenTimer [22], an ASIC oriented tool which to the best of our knowledge is the only other open-source parallel static timing analyzer.

We evaluate Tatum and OpenTimer on the Tau 2015 ASIC Timing Contest benchmarks [31]. The smallest benchmarks with fewer than 10^5 timing nodes ($N \leq 10^5$) are excluded, as they are too small to benefit from parallelization. We report the self-relative speed-up achieved when fully recalculating the timing graph, averaged over 100 runs. Run-time results were collected with two Intel E5-2683 v4 CPUs (14nm 'Broadwell', 16 cores), for a total of 32 cores.

As Figure 3 shows, OpenTimer achieves a maximum speed-up of $1.2\times$, and does not scale beyond 8 cores. OpenTimer's speed-up is limited by its parallelization method, a fixed task-based pipeline [22], which yields at most an $O(1)$ constant-factor speed-up as the number of cores increases [32].

[8]Tatum's source code is available at: https://kmurray.github.io/tatum.

TABLE VI. VPR Multi-clock STA Run-Time During Placement

Benchmark	Timing Graph Nodes	Clocks	Classic VPR	Tatum $P = 1$		Tatum $P = 32$	
stereo_vision	416,747	4	93.7	39.6	(2.37×)	4.8	(19.7×)
sparcT1_core	513,974	3	78.9	39.6	(1.99×)	4.8	(16.5×)
minres	1,278,986	3	268.1	127.9	(2.10×)	10.5	(25.6×)
sparcT2_core	1,566,943	3	402.6	132.7	(3.03×)	11.6	(34.6×)
gsm_switch	2,666,350	5	979.2	273.7	(3.58×)	18.6	(52.6×)
LU230	3,038,632	3	1,218.8	416.4	(2.93×)	40.7	(30.0×)
mes_noc	3,261,440	10	1,855.9	383.4	(4.84×)	29.7	(62.5×)
LU_Network	3,355,076	21	1,494.0	427.5	(3.49×)	28.7	(52.0×)
sparcT1_chip2	3,765,406	3	1,099.4	446.4	(2.46×)	35.7	(30.8×)
bitcoin_miner	4,513,409	3	1,891.4	943.6	(2.00×)	66.1	(28.6×)
directrf	4,902,870	3	2,365.5	1,066.5	(2.22×)	86.2	(27.4×)
GEOMEAN	2,081,093	4	673.7	248.5	(2.71×)	21.1	(32.0×)

Time in seconds; Speed-up relative to 'Classic VPR' in brackets.
P = number of cores used for STA; $P = 32$ uses two sockets.

In contrast, Figure 3 shows Tatum scales to at least 32 cores, achieving a maximum speed-up of $8.3\times$. Tatum's data-parallel approach scales as $O(N/L)$ as the number of cores increases. This means Tatum has a desirable property: its speed-up improves as design size (N) increases.

While Tatum is more scalable than OpenTimer, it scales sub-linearly with the number of cores due to:

- the inherently unparallelizable work (i.e. dependencies between nodes in different levels, L), and
- the communication and synchronization costs inherent to real computing systems.

B. Evaluation in VPR

We next integrated Tatum into the VPR FPGA CAD tool, which repeatedly calls STA to guide optimization [33]. We compare Tatum's performance (and verify correctness) against the classic VPR timing analyzer.[9]

We report both the run-time spent on STA and total run-time while placing the Titan FPGA benchmarks [28], which are large designs ranging in size from 90K to 1.8M netlist primitives. The results were again collected on a system with dual Intel E5-2683 v4 CPUs (32 cores).

1) STA Run-Time: Table VI shows the STA run-time results on the multi-clock subset of Titan benchmarks. With a single core ($P = 1$) Tatum runs $2.7\times$ faster than Classic VPR, showing the tagged approach (Section VII) performs better on multi-clock circuits. With 32 cores Tatum runs $32\times$ faster.

Figure 4 shows how Tatum's performance scales with the number of cores. Tatum achieves an average speed-up of $15.2\times$ on the full Titan benchmark set with 32 cores, and an average speed-up of $7.7\times$ on the single-clock subset of the Titan benchmarks.

2) Total Run-Time: Table VII shows the results for total placement run-time. With VPR's classic timing analyzer STA accounts for 10% of total run-time on average, with multi-clock circuits spending up to 22% of run-time on STA. When VPR is run using Tatum's parallel timing analysis total placement run-time is reduced by 11%.

IX. Improving Optimization

CAD tools like VPR have been tuned to minimize the amount of run-time spent on STA [1]; instead making optimization choices using stale timing information.

Since Tatum reduces the run-time overhead of maintaining up-to-date timing information it is worth considering whether this can improve optimzation quality. To this end we modified VPR's placement algorithm to exploit more accurate timing information.

[9]VPR's classic analyzer performs a serial levelized graph traversal (Algorithm 1) and uses the multi-traversal approach to multi-clock analysis.

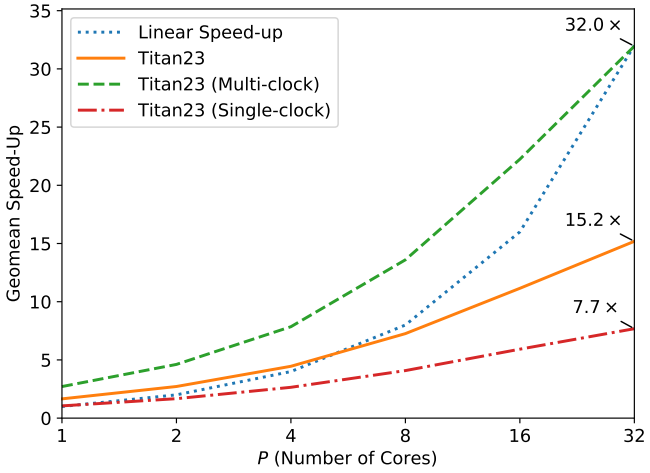

Fig. 4. Tatum Performance Scaling over Classic VPR STA.

TABLE VII. VPR PLACEMENT TOTAL RUN-TIME

Benchmark	Timing Graph Nodes	Clocks	STA Updates	VPR Classic		VPR Tatum ($P = 32$)		VPR Tatum Speed-Up
stereo_vision	416,747	4	143	7.1	(22.0%)	5.6	(1.9%)	1.26×
neuron	451,916	1	143	6.7	(10.7%)	6.1	(1.9%)	1.10×
sparcT1_core	513,974	3	125	8.7	(15.1%)	7.5	(1.4%)	1.16×
cholesky_mc	615,925	1	148	10.4	(11.6%)	9.4	(2.3%)	1.10×
SLAM_spheric	756,307	1	170	21.3	(8.7%)	19.7	(1.5%)	1.08×
des90	779,404	1	140	14.4	(11.4%)	13.0	(1.7%)	1.11×
segmentation	980,309	1	193	31.3	(9.7%)	30.0	(5.8%)	1.04×
dart	1,137,708	1	126	20.9	(10.3%)	19.0	(1.2%)	1.10×
stap_qrd	1,248,502	1	224	60.2	(7.8%)	56.2	(1.0%)	1.07×
minres	1,278,986	3	135	32.1	(13.9%)	27.9	(1.0%)	1.15×
openCV	1,281,693	1	146	25.6	(11.2%)	23.0	(1.3%)	1.11×
cholesky_bdti	1,392,239	1	185	40.1	(9.4%)	36.8	(1.4%)	1.09×
bitonic_mesh	1,409,553	1	127	32.6	(9.6%)	29.8	(1.0%)	1.09×
sparcT2_core	1,566,943	3	141	58.7	(11.4%)	52.2	(0.5%)	1.12×
denoise	2,068,769	1	245	115.5	(5.7%)	112.0	(2.8%)	1.03×
gsm_switch	2,666,350	5	142	109.8	(14.9%)	94.0	(0.5%)	1.17×
LU230	3,038,632	3	176	182.7	(11.1%)	163.4	(0.6%)	1.12×
mes_noc	3,261,440	10	142	183.9	(16.8%)	153.7	(0.5%)	1.20×
LU_Network	3,355,076	21	166	202.7	(12.3%)	178.6	(0.5%)	1.13×
sparcT1_chip2	3,765,406	3	143	203.9	(9.0%)	186.3	(0.4%)	1.09×
bitcoin_miner	4,513,409	3	272	355.8	(8.9%)	325.8	(0.5%)	1.09×
directrf	4,902,870	3	254	489.4	(8.1%)	452.0	(0.5%)	1.08×
gaussianblur	11,554,254	1	150	1,143.5	(1.6%)	1,127.3	(0.2%)	1.01×
GEOMEAN	1,591,104	2	162	55.2	(10.0%)	49.8	(1.0%)	1.11×

Time in minutes; Percentage of time spent on STA in brackets.
P = number of cores used for STA; $P = 32$ uses two sockets.

A. Algorithmic Enhancements

We focus on critical path optimizations during VPR's final placement optimization stage: the 'quench' phase of the anneal, where only modifications which improve quality are accepted. All other placement steps use the standard VPR formulation.

During the quench, we call STA after each move (swap of blocks) to ensure the optimizer has a fully accurate view of the move's timing impact. Interestingly, with VPR's default move generator and cost formulation this resulted in no net improvement in critical path delay.

To improve efficiency we biased the move generator to pick blocks which are connected to highly critical connections (criticality > 0.7). We also set a high criticality exponent (50) and timing trade off (0.99) to focus the optimization on improving the critical path, and used a small region limit (1) to generate small incremental modifications to the circuit. We observed that VPR's connection-based criticality timing cost formulation [1] fails to provide accurate guidance to reduce the critical path in this final optimization, as its connection oriented approach does not directly evaluate the critical path delay. At this final stage of fine-tuning, focusing on specific paths is key to improving results, and we modified the timing cost function during the quench to directly optimize the circuit's critical path delay. Finally, we exit the quench when no moves have resulted in cost improvements for a significant period of time (indicating the system has frozen out and reached a minima), or the standard VPR move limit is reached.

TABLE VIII. IMPACT OF STA FREQUENCY DURING QUENCH

Benchmark	Per-Move STA				Single STA		
	WL	CPD	Place Time ($P = 1$)	Place Time ($P = 24$)	WL	CPD	Place Time
tseng	1.00	0.97	3.80	2.75	1.00	1.08	0.77
dsip	1.00	0.93	2.97	1.39	1.00	1.07	0.77
diffeq	1.00	1.00	4.15	2.33	1.00	1.02	0.90
bigkey	1.00	0.99	3.21	1.43	1.00	0.99	0.82
s298	1.00	0.95	6.16	3.38	1.00	1.02	0.88
frisc	1.00	0.96	9.79	3.38	1.00	1.14	0.90
elliptic	1.00	0.92	11.36	3.23	1.00	1.08	1.01
s38584.1	1.00	0.98	11.07	2.44	1.00	1.05	1.04
s38417	1.00	0.97	12.11	2.80	1.00	1.04	1.01
clma	1.00	0.94	12.75	3.02	1.00	0.99	1.01
ex5p	1.00	0.97	4.45	3.32	1.00	1.02	1.00
apex4	1.00	0.96	4.78	2.98	1.00	1.09	1.05
misex3	1.00	0.95	4.02	2.10	1.00	1.03	0.95
alu4	1.00	0.97	3.20	1.62	1.00	1.06	0.78
des	1.00	0.99	4.27	1.84	1.00	1.04	0.85
seq	1.00	0.98	3.82	1.80	1.00	1.07	0.84
apex2	1.00	0.97	5.23	2.17	1.00	1.02	0.97
spla	1.00	0.98	9.41	2.63	1.00	1.07	1.06
pdc	1.00	0.95	9.87	2.57	1.00	1.05	0.91
ex1010	1.00	0.97	8.51	2.34	1.00	1.05	0.90
GEOMEAN	1.00	0.96	5.96	2.39	1.00	1.05	0.92

Values normalized to VPR default.
P = number of cores used for STA; $P = 24$ uses two sockets.

B. Results

Table VIII compares the impact of using stale and up-to-date timing information while directly optimizing the critical path during the quench. Results were collected on the MCNC20 benchmarks [34] targeting a $K = 4$, $N = 1$ architecture with length one wires, and were run on a machine with two Intel Xeon Gold 6146 (14nm 'Skylake', 12 cores), for a total of 24 cores. Reported values are the average over 5 placement seeds.

Performing STA during the evaluation of each proposed move (Per-Move STA) improves Critical Path Delay (CPD) by 4% on average, and by up to 8% on some circuits (e.g. `elliptic`) compared to standard VPR. These CPD improvements are obtained with no degredation in wirelength (WL).

To show that these improvements are in fact derived from frequent timing analysis (rather than the algorithmic modifications listed above), we also report results where only a single STA is preformed at the start of the quench (Single STA). In this scenario, the optimizer is forced to make decisions using stale timing information. This leads it to over-optimize paths which originally appeared critical, degrading the actual critical path by 5% compared to standard VPR.

While we've shown that frequent timing analysis can improve optimization quality, it comes with a run-time overhead – increasing placement run-time by $5.96\times$. However, this overhead is dominated by STA and can be reduced to $2.39\times$ when Tatum uses 24 cores. This makes the approach suitable for late stages of the design process (such as final timing closure), where increasing tool effort is acceptable in return for improved quality.

We expect these results to improve on heavily pipelined designs (which are increasingly common), where up-to-date timing information has also been shown to improve timing [35].

X. CONCLUSION & FUTURE WORK

Static Timing Analysis is a key element in FPGA design flows and accounts for a significant fraction of design implementation time. This fraction threatens to increase further in the face of growing design sizes and increasingly complex timing conditions. Furthermore, as other algorithms such as placement and routing are parallelized serial STA becomes the dominant run-time component. Therefore speeding-up STA is fundamental to reducing FPGA design cycle times.

To address these issues we have investigated a variety of techniques to speed-up Static Timing Analysis, including data layout optimizations and parallel algorithms on both GPUs and CPUs. While GPUs were shown to offer good kernel speed-ups (average $6.2\times$), the overhead of data transfer to/from the host CPU negated the performance improvement. Of the CPU parallel algorithms, a level-parallel approach performed best, particularly when synchronization costs are reduced by avoiding expensive locks and atomic operations. This approach achieved an average self-relative speed-up of $1.9\times$ with 4 cores.

Analyzing multiple timing corners increases the amount of work per memory access, improving the speed-up to $3.8\times$ with 4 cores. Exploiting SIMD techniques further improves the speed-up to $6.2\times$. The combination of these methods allows $14.5\times$ more timing corners to be analyzed on 4 cores within a fixed CAD run-time budget.

Focusing on advanced STA features, we described how tagged arrival/required times improve performance on multi-clock circuits by reducing the number of timing graph traversals.

We introduced the open-source Tatum STA library, supporting efficient parallel timing analysis of complex designs with multiple clocks and timing constraints. Tatum reduces the barrier to producing timing-aware CAD tools, and has been adopted as the timing analysis engine in CGRA-ME [36] and version 8.0 of VTR [37]. We compared Tatum's performance with OpenTimer, showing that OpenTimer's parallel approach achieves a maximum $1.2\times$ speed-up at 8 cores, while Tatum's speed-up increases to $8.3\times$ at 32 cores.

We evaluated Tatum's performance in VPR, showing that it runs on average $15.2\times$ faster than VPR's classic timing analyzer with 32 cores, and $32\times$ faster on more complex multi-clock circuits; reducing total placement time of the Titan benchmarks by 11%.

Finally, we showed how faster timing analysis can be used to improve optimization results, showing that more frequent STA can improve critical path delay by 4%.

As Tatum is an open-source re-usable library, we believe it can be leveraged by other researchers to facilitate the creation of advanced timing-aware CAD tools. Such tools enable new CAD algorithm explorations, and a wider variety of more accurate architectural studies.

ACKNOWLEDGEMENTS

We would like to thank Tom Spyrou, Chris Wysocki, and Paul Leventis for valuable discussions on the STA problem, and Cristiana Amza and Andreas Moshovos for additional feedback. This work was supported by an NSERC CGS-D scholarship, the NSERC/Intel Industrial Research Chair in Programmable Silicon, and Huawei. This research was also enabled in part by Compute Canada (www.computecanada.ca).

REFERENCES

[1] A. Marquardt, V. Betz, and J. Rose, "Timing-driven placement for fpgas," in *FPGA*. New York, NY, USA: ACM, 2000, pp. 203–213.

[2] S. M. Trimberger, "Three ages of fpgas: A retrospective on the first thirty years of fpga technology," *Proceedings of the IEEE*, vol. 103, no. 3, 2015.

[3] A. Ludwin and V. Betz, "Efficient and deterministic parallel placement for fpgas," *ACM TODAES*, vol. 16, no. 3, pp. 22:1–22:23, 2011.

[4] A. B. Kahng, "New game, new goal posts: A recent history of timing closure," in *DAC*, 2015, pp. 4:1–4:6.

[5] C. Ebeling *et al.*, "Stratix^{TM}10 high performance routable clock networks," in *FPGA*, 2016, pp. 64–73.

[6] M. Hutton *et al.*, "Efficient static timing analysis and applications using edge masks," in *FPGA*, 2005, pp. 174–183.

[7] D. Lewis *et al.*, "Architectural enhancements in stratix v^{TM}," in *FPGA*, 2013, pp. 147–156.

[8] D. Lewis, G. Chiu *et al.*, "The stratix^{TM}10 highly pipelined fpga architecture," in *FPGA*, 2016, pp. 159–168.

[9] R. Fung, V. Betz, and W. Chow, "Slack allocation and routing to improve fpga timing while repairing short-path violations," *IEEE TCAD*, vol. 27, no. 4, pp. 686–697, April 2008.

[10] J. B. Goeders, G. G. F. Lemieux, and S. J. E. Wilton, "Deterministic timing-driven parallel placement by simulated annealing using half-box window decomposition," in *ReConFig*, Nov 2011, pp. 41–48.

[11] M. An, J. G. Steffan, and V. Betz, "Speeding up fpga placement: Parallel algorithms and methods," in *FCCM*, May 2014, pp. 178–185.

[12] C. Fobel, G. Grewal, and D. Stacey, "A scalable, serially-equivalent, high-quality parallel placement methodology suitable for modern multicore and gpu architectures," in *FPL*, Sept 2014, pp. 1–8.

[13] M. Gort and J. H. Anderson, "Deterministic multi-core parallel routing for fpgas," in *FPT*, Dec 2010, pp. 78–86.

[14] C. H. Hoo, A. Kumar, and Y. Ha, "Paralar: A parallel fpga router based on lagrangian relaxation," in *FPL*, Sept 2015, pp. 1–6.

[15] M. Stojilovi, "Parallel fpga routing: Survey and challenges," in *FPL*, Sept 2017, pp. 1–8.

[16] M. Shen and G. Luo, "Corolla: Gpu-accelerated fpga routing based on subgraph dynamic expansion," in *FPGA*, 2017, pp. 105–114.

[17] J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*. Springer, 2009.

[18] D. Blaauw *et al.*, "Statistical timing analysis: From basic principles to state of the art," *IEEE TCAD*, vol. 27, no. 4, pp. 589–607, 2008.

[19] W. Donath and D. Hathaway, "Distributed static timing analysis. us patent 6,557,151." Patent, 1998.

[20] A. Holder *et al.*, "Prototype for a large-scale static timing analyzer running on an ibm blue gene," in *IEEE Int. Sym. on Paral. Distr. Proc.*, April 2010, pp. 1–8.

[21] V. Veetil *et al.*, "Efficient monte carlo based incremental statistical timing analysis," in *DAC*, June 2008, pp. 676–681.

[22] T. W. Huang and M. D. F. Wong, "Opentimer: A high-performance timing analysis tool," in *ICCAD*, Nov 2015, pp. 895–902.

[23] Y. Deng, B. D. Wang, and S. Mu, "Taming irregular EDA applications on GPUs," in *ICCAD*, Nov. 2009, p. 539.

[24] K. Gulati *et al.*, "Accelerating statistical static timing analysis using graphics processing units," in *ASP-DAC*, Jan. 2009, pp. 260–265.

[25] Y. Shen and J. Hu, "Gpu acceleration for pca-based statistical static timing analysis," in *ICCAD*, Oct 2015, pp. 674–679.

[26] H. Wang *et al.*, "Casta: Cuda-accelerated static timing analysis for vlsi designs," in *Int. Conf. on Paral. Proc.*, 2014, pp. 192–200.

[27] J. L. Hennessy and D. A. Patterson, "Data-level parallelism in vector, simd, and gpu architectures," in *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2011.

[28] K. E. Murray *et al.*, "Timing-Driven Titan: Enabling Large Benchmarks and Exploring the Gap Between Academic and Commercial CAD," *ACM TRETS*, vol. 8, no. 2, p. 18, 2015.

[29] Intel Corporation, www.cilkplus.org, 2017.

[30] R. Fung and V. Betz, "Optimizing long-path and short-path timing and accounting for manufacturing and operating condition variability," Patent 7 254 789, August, 2007.

[31] "Tau 2015 Contest Resources," https://sites.google.com/site/taucontest2015, 2015, accessed: April 12, 2018.

[32] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012.

[33] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *FPL*, 1997, pp. 213–222.

[34] S. Yang, "Logic Synthesis and Optimization Benchmarks User Guide 3.0," MCNC, Tech. Rep., 1991.

[35] K. Eguro and S. Hauck, "Enhancing Timing-driven FPGA Placement for Pipelined Netlists," in *DAC*, 2008, pp. 34–37.

[36] K. P. Niu and J. H. Anderson, "Compact Area and Performance Modelling for CGRA Architecture Evaluation," in *FPT*, 2018.

[37] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu *et al.*, "VTR 7.0: Next Generation Architecture and CAD System for FPGAs," *ACM TRETS*, vol. 7, no. 2, pp. 6:1–6:30, June 2014.