An Updated Model of Computation for VLSI and Applications to FPGA Implementation

Nathaniel Young

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-108
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-108.html

May 13, 2022

An Updated Model of Computation for VLSI and Applications to FPGA Implementation

by Nathaniel Young

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor John Wawrzynek
Research Advisor

May 13, 2022

(Date)

* * * * * * *

Professor Satish Rao
Second Reader

May 13, 2022

(Date)

Acknowledgements

I would like to thank my excellent advisor, Professor John Wawrzynek, for providing me with both the opportunity to study hardware through a theoretical lens and plenty of indispensable guidance and support with which to do so. I would like to thank Professor Satish Rao as well, for supporting me similarly and helping me to understand a lot of difficult concepts – all with great patience. I would also like to thank the other students in Prof. Wawrzynek's group and beyond, most notably Arya Reais-Parsi and Grace Dinh, for many valuable conversations and collaborations. I want to thank my parents for supporting me unfalteringly in everything, and finally I want to thank the rest of my family and friends, especially my brothers Benjamin and Jonathan, who kept me sane during the pandemic.

Abstract

In this report, we present a theoretical model for VLSI computation, with assumptions updated for modern technology, and a number of asymptotic lower bounds in this model. Among other facts, we show unconditionally that all n-input computations of a single output require time $\Omega(\sqrt[3]{n})$, that dense matrix multiplication requires time $\Omega(n)$ for $n \times n$ matrices, and that sparse-matrix-times-dense-vector multiplication (SpMV) requires time $\Omega(\sqrt{n/\log n})$ for some matrices. We also show that implementation of the Bellman-Ford shortest paths algorithm requires time $\Omega(n^{4/3})$ for some graphs.

Additionally, we develop bounds on placement quality for FPGA designs, and algorithms for applying them. We present comparisons between our bounds and the quality of an actual placement for several benchmark designs.

1 Introduction

In 1979, shortly after the practical development of Very Large-Scale Integrated circuits (VLSI), Clark Thompson developed a theoretical model of VLSI chips and several asymptotic lower bounds to go with it. His results sparked more than a decade of interest among theoreticians in applying graph theory, communication complexity, and similar fields to problems in VLSI circuit layout and to the problem of mapping computation to VLSI chips in general.

Today, far more advanced VLSI technology is the backbone of all computation, but theoretical interest in it has waned, and most research into hardware design takes practical approaches to practical problems. However, with exponential scaling of single-core processor performance dead, and Moore's law faltering, there is renewed interest in building task-specific hardware and in approaching the physical limits of parallel computation. In order to understand those limits, it may be time for a resurgence of theoretical study of VLSI models and higher-level hardware models.

In the first part of this report, we provide modernized assumptions for VLSI models and several asymptotic lower bounds using them. Some are updated from lower bounds which were already known using the VLSI models of the 1980s, and some are entirely new.

In the second part, we provide a theoretical framework and several explicit algorithmic tools – mainly lower bounds on the length of the longest path after placement – for the analysis of digital designs, with a focus on implementation on field-programmable gate arrays (FPGAs). Some of these tools take inspiration directly from the VLSI lower bounds of the first part.

Though they have connections to each other, the two parts of this report can be considered independent; readers interested only in VLSI theory can skip the second part, and readers interested only in FPGA implementation can skip the first.

2 VLSI lower bounds

In this section, we will present lower bounds on the chip area and execution time of VLSI implementations of several algorithms and classes of algorithms.

The first task will be to develop the assumptions that make up a model of VLSI computations. VLSI technology has changed significantly since the 1980s, and which models are reasonable have changed with it.

We will show some simple, very general lower bounds using these assumptions, and then how to use them in more sophisticated arguments and how to generalize them.

The result of most practical interest is probably theorem 2.13, which provides a bound on the time needed to multiply a sparse matrix by a dense vector; however, the result of most theoretical interest may be theorem 2.4, which provides a non-obvious lower bound on the VLSI execution time of the Bellman-Ford shortest paths algorithm by using algorithm structure to 'amplify' a simple unconditional lower bound through iteration.

2.1 Preliminaries

The bounds in this section concern implementation of certain computation graphs on VLSI chips. Before we present them, we should discuss what our terms mean.

A *computation graph* is a directed acyclic graph (DAG) which represents all the work taking place in some computation. Each node in the graph represents a *value* in the computation, and edges represent dependencies between them. Sources and sinks in the computation graph correspond to inputs and outputs of the computation respectively. Two computation graphs representing computations to solve the same problem will therefore have identical sets of sources and sinks, but the structure of the internal nodes of the computation may differ, as the two algorithms may compute different intermediate values. Note that a computation graph is a *nonuniform* model; that is, it is stated for a fixed size of input and output, and no notion of how to generate computation graphs for arbitrary input sizes from a fixed general representation is given.

A value b in a computation is said to *depend* on another value a if it is impossible to accurately compute b without first knowing a; that is, if there is some setting to the inputs of the computation such that a change in the setting to a induces a change in the correct setting to b. Note that if a computation graph G contains a and b, with b depending on a, then there must be a path in G from a to b. Two values are said to *have communication* with each other if one depends on the other.

A value in a computation graph is said to be *useful* if at least some output depends on it. We will generally assume that all values in a computation graph are useful.

A VLSI chip is said to *implement* a computation graph if the entire computation represented by the graph is performed by the chip: all values are computed (or in the case of inputs, are received) somewhere on the chip at some time, and all communications necessary to perform these computations have happened physically in the wires on the chip.

2.2 Assumptions

For our lower bounds in the VLSI model, we will use four main assumptions:

1. No two distinct values produced or used by the computation may appear at the same place on the chip at the same time (each value requires its own $\Omega(1)$ space).

2. Across any boundary of length ℓ in the chip, only $O(\ell)$ values may pass at any one time.

   This assumption was the primary one used in 1980s VLSI theory. Since a chip of area A can be divided into two equal halves with a boundary of length $\sqrt{A}$, finding that at least b values must cross any such boundary gives an area-time tradeoff: $\sqrt{A}T \geq \Omega(b)$. Squaring this relationship (to make it look nicer) gives the primary object of study of that line of work (after which it is commonly named): an AT^2 bound. This approach was introduced by Thompson [17][18], who used it in conjunction with communication complexity to find unconditional lower bounds on AT^2 for several problems. Later authors writing similar bounds often used other methods for finding b, but almost all bounds were stated unconditionally for particular problems, independent of the algorithm being implemented to solve them, and so no such bound (without the use of other assumptions) is larger than $AT^2 \geq \Omega(n^2)$ for a problem with n bits of input and output.

3. Communicating a value a distance d across the chip requires $T = \Omega(d)$ time.

   In the classic VLSI theory literature from the 1980s, this assumption made little appearance. It was used by Chazelle and Monier, who justified it using speed-of-light delay [2], but at the time, it was controversial. On-chip communications in the 1980s were performed at nowhere near the speed of light, and a communication which was running too slowly could in many cases be sped up by increasing the size of the transistors in the driver circuitry for the wire along which the communication took place. A theoretical study of minimizing wirelength from the time [12] stated that a reasonable assumption appeared to be $T = \Omega(\sqrt{d})$. In modern chips, however, long communications are done through a linear number of constant-length buffered wire segments to avoid introducing a quadratic RC delay, and the speed of the very fastest communications along wires is affected by transmission-line delay, which is linear in distance [13]. So, we now regard $T = \Omega(d)$ as not only a technically correct assumption due to speed-of-light delay, but also a practical assumption relevant for actual modern chips.

4. All inputs to the computation must enter the chip at a single place and time, and all outputs from the computation must leave the chip at a single place and time.

   Note that this means all values used by the computation are assigned a single spot on the chip where they originate, and computation using that value must pay the time and area cost to communicate with that spot.

   This assumption is quite reasonable from the perspective of chip design, but it is a little less obvious for general computation – is it really not allowed to copy the input to the computation multiple times before putting those copies on the chip? However, this assumption will be important for a few of our bounds, and it does make sense. A computation which wishes to replicate an input in many places should pay the communication cost to do so on the chip.

   Some prior work in VLSI theory assumed that inputs to the chip must appear on the chip perimeter. Chazelle and Monier, for instance, relied on all the inputs to the computation passing through some convex boundary for most of their lower bounds [2]. However, modern "3D packaging" allows for this assumption to be broken and for communication onto and off of the chip to happen at arbitrary points on its surface, so we no longer regard that stronger assumption as reasonable. In addition, allowing inputs to appear at any place and time on the chip means that they do not strictly have to be input to the chip then – the input values to the computation could be the outputs of an earlier computation performed on the same chip, and bounds using these assumptions will still hold.

2.3 Basic Lower Bounds

We will present a few simple lower bounds here, which we will expand upon later.

Theorem 2.1 ($AT \geq \Omega(N)$)**.** *Performing a computation of N values on a VLSI chip, using minimal bounding box area A and time T, requires $AT \geq \Omega(N)$.*

Proof. This follows immediately from the first assumption; since all N values require their own constant amount of area and time, and no two overlap, we can sum their area-time requirements together, for $AT \geq N \cdot \Omega(1) = \Omega(N)$. □

This fact also appears in [15]. Figure 1 shows a useful picture to keep in mind: the area of a chip, extended through a third dimension representing time to produce an "area-time volume." With this intuition, theorem 2.1 states that the area-time volume must be $\Omega(N)$, because each value computed requires a constant area-time volume of its own.

Theorem 2.2 ($T \geq \Omega(\sqrt{A})$)**.** *Performing a one-output computation where all values are useful on a VLSI chip, using time T and a minimal bounding box of area A, requires $T \geq \Omega(\sqrt{A})$.*

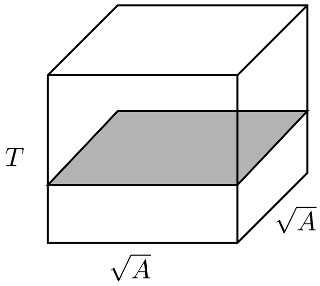


Figure 1: An area-time volume for a square chip, with a single 'time slice' (the chip at a single timestep) indicated.

Proof. The intuition for this result is that all values must be communicated to the output, so at least some communication must cross at least half the bounding box. We will now formalize this somewhat.

Without loss of generality, assume the bounding box for the computation is at least as wide as it is tall; since the area is A, this means the width of the box is at least $\sqrt{A}$. Now, again without loss of generality, assume the output of the computation is in the right half of the box (or exactly in the middle horizontally); this means that the distance from the left edge of the bounding box to the output is at least $\sqrt{A}/2$. Since the bounding box is minimal, there must be some value in the computation which is on the left edge; call this value x_L. Then, considering the path from x_L to the output, we have

$$\sum_{(u,v)\in(x_L\to o)} d_x(u,v) \geq \sqrt{A}/2$$

where $d_x(u,v)$ is the horizontal distance between u and v and the sum is over all edges in the path from x_L to the output (this is by the triangle inequality, applied to the distances between values). But for any edge (u,v), the time difference $T(u,v)$ between when u is produced and when v is produced is at least $\Omega(d(u,v)) \geq \Omega(d_x(u,v))$ by assumption 3. So, summing the delays incurred by the path, we have that

$$T(x_L,o) = \sum_{(u,v)\in x_L\to o} T(u,v) \geq \Omega(\sqrt{A})$$

which gives us $T \geq \Omega(\sqrt{A})$. □

In later theorems, we will not bother with the above argument that a path incurs communication time at least as large as the distance between its endpoints; we will allow ourselves to assume that as a direct consequence of assumption 3.

Corollary 2.3 ($T \geq \Omega(\sqrt[3]{N})$)**.** *Performing a one-output computation of N values, all useful, on a VLSI chip in time T requires $T \geq \Omega(\sqrt[3]{N})$.*

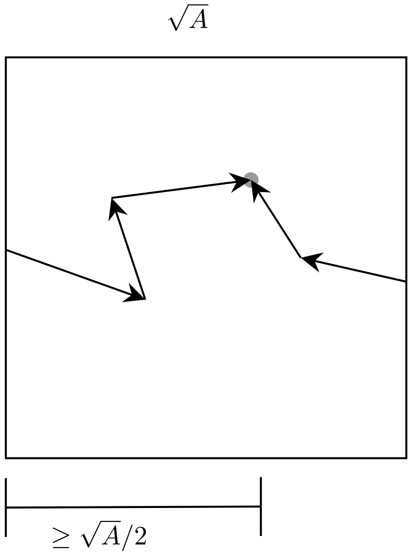


Figure 2: A picture for the proof of theorem 2.2

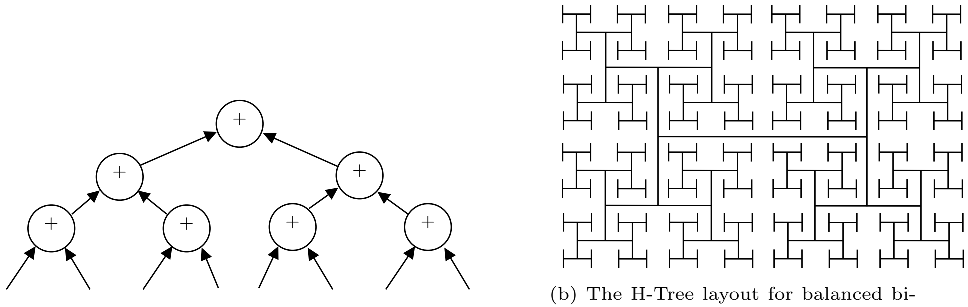
Proof. This follows directly from the above two theorems. Consider the minimal bounding box area A used for this computation. We have that $T^2 \geq \Omega(A)$ and $AT \geq \Omega(N)$, which gives us $T^3 \geq \Omega(AT) \geq \Omega(N)$, or $T \geq \Omega(\sqrt[3]{N})$. □

This result is interesting in that it provides an unconditional bound on the effect of parallelization on a chip – no computation of a single output can be parallelized to run faster than the cube root of the serial runtime. Note also that any computation which uses n inputs and depends on all of them has $T \geq \Omega(\sqrt[3]{n})$.

Chazelle and Monier [2] found a lower bound $T \geq \Omega(\sqrt{n})$ on n-input computations, which is larger, but as noted in the assumptions above, they assumed that all inputs to the computation had to pass through some convex boundary on the chip, which we do not regard as reasonable for modern chips.

To see the relevance of this result to an actual parallel computation, let's consider computing the sum of an array of n numbers. The classic way to approach parallelizing this computation is a binary tree as in figure 3a; this computation graph has depth $O(\log n)$, and so in communication-ignoring models, will take $O(\log n)$ parallel time. On a chip, however, the communication dominates the asymptotic runtime. One way to implement the binary reduction tree on a chip is the famous H-tree layout (figure 3b); this incurs $\Theta(\sqrt{A}) = \Theta(\sqrt{n})$ communication time (the longest individual wires alone in an H-tree layout have length $\sqrt{A}/4$) [8].

In order to match the lower bound of corollary 2.3, we will first notice that theorem 2.2 implies that we should not use an area of greater than $A = \Theta(n^{2/3})$. This is enough area to store and accumulate $n^{2/3}$ independent partial sums of



(a) A binary sum-reduction tree

(b) The H-Tree layout for balanced binary trees

Figure 3: Classic binary reduction trees

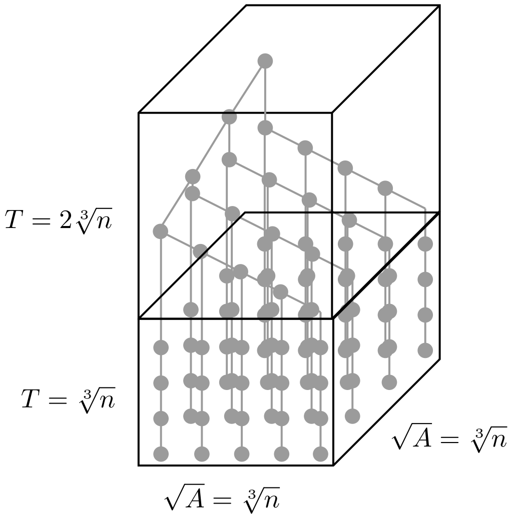
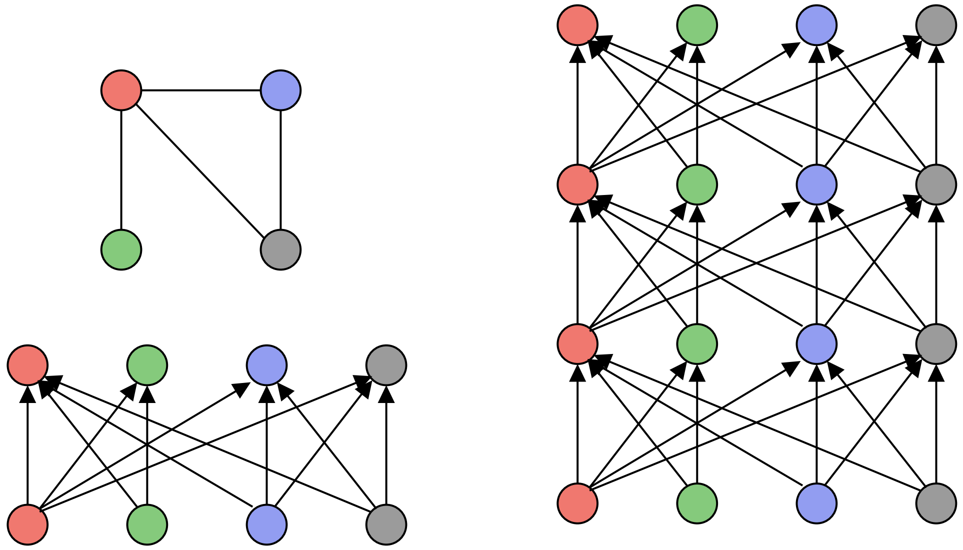


Figure 4: An $O(\sqrt[3]{n})$-time scheduling of an n-element reduction

$\sqrt[3]{n}$ elements each by bringing $n^{2/3}$ different elements onto the chip at every time step. Once these have been accumulated, they can be summed to produce a single output using any $\sqrt{A}$ time reduction tree. This approach takes $\sqrt[3]{n}$ time to bring in the inputs and accumulate the $n^{2/3}$ sums, plus a further $\sqrt[3]{n}$ time to sum these $n^{2/3}$ values together, for $O(\sqrt[3]{n})$ time total. A diagram of this process (using a simple $2\sqrt[3]{n}$-depth, $2\sqrt[3]{n}$-time reduction tree) is provided in figure 4.

2.4 A first nontrivial application

In this section, we will see a way to apply corollary 2.3 iteratively, and obtain an interesting lower bound on the parallel implementation time of a practical algorithm.



(a) A 4-node undirected graph and its "bipartite version"

(b) 3 iterations of the Bellman-Ford computation graph for the graph in figure 5a

Figure 5: Bellman-Ford graphs

Theorem 2.4 (Bellman-Ford takes $n^{4/3}/d$ time)**.** *Performing the Bellman-Ford algorithm for Single Source Shortest Paths for an undirected graph of n nodes and unweighted diameter d, on a chip, requires time $T \geq \Omega(n^{4/3}/d)$.*

Proof. First, we will show that performing d iterations of Bellman-Ford on a graph of diameter d requires $\Omega(\sqrt[3]{n})$ time (between the time the first input is presented and the time the first output is produced); then the overall result will follow by repetition of this result $\frac{n-1}{d}$ times, for the $n-1$ iterations that must be performed in the worst case.

Recall that a single 'iteration' of the Bellman-Ford algorithm involves, for each node v in the graph, checking each of its neighbors u to determine whether the shortest path length to v computed at the previous iteration can be extended to u to construct a shorter path from the source to u than was found at the previous iteration. The computation graph for a single iteration, then, is essentially the 'bipartite version' of the input graph (see figure 5a for an example). To see that any d iterations must take at least $\sqrt[3]{n}$ time, we appeal to corollary 2.3. Consider the computation graph C_G which is the DAG of those d iterations (see figure 5b). Since the input graph has diameter d, there is a path in the computation graph of those d iterations from any input to any output. Accordingly, each output depends on all n inputs, meaning that none can complete before $\sqrt[3]{n}$ time after the first input arrives. □

2.5 More general bounds

In this section, we will see a few more bounds on computation time, with an application to two important tasks: dense matrix-matrix multiplication, and sparse matrix-vector multiplication.

Definition 2.5 (Path Diameter)**.** We say that a computation graph has path diameter δ when the undirected diameter of its transitive closure has diameter δ; that is, the path diameter is the minimum value δ where for any two values in the computation x and y there is a series of at most δ paths which together connect x to y. Formally, this means that there is a series of $\delta - 1$ values $v_1, v_2, \ldots, v_{\delta-1}$ where x has communication with v_1, v_1 has communication with v_2, and so on, with $v_{\delta-1}$ having communication with both $v_{\delta-2}$ and y.

Another way of thinking about path diameter is that it is the maximum number of times one needs to 'change direction' to get from any node to any other (using only forward edges followed by only backward edges followed by only forward edges, etc).

Theorem 2.6 ($T \geq \Omega(\sqrt{A}/\delta)$)**.** *Implementing a computation graph of path diameter δ on a VLSI chip, using time T and a minimal bounding box of area A, requires $T \geq \Omega(\sqrt{A}/\delta)$.*

Proof. We can prove this using a generalization of the strategy taken for theorem 2.2. We will show that there must exist two values in the computation graph which have communication with each other and which are at least a distance $\sqrt{A}/\delta$ apart; this will imply that $\Omega(\sqrt{A}/\delta)$ time is needed to perform this communication.

Suppose (towards a contradiction) that any two values in the computation graph which have communication with each other (and so are connected by a single path) are strictly less than $\sqrt{A}/\delta$ apart. Once again, assume without loss of generality that the bounding box is at least as wide as it is tall (and so has width at least $\sqrt{A}$). Consider the two values x_L and x_R in the computation which define the left and right edges of the bounding box. Since the graph has path diameter δ, consider the $\delta - 1$ values $v_1 \ldots v_{\delta-1}$ that connect x_L to x_R. By assumption, we have that $d(x_L, v_1) < \sqrt{A}/\delta$, that $d(v_1, v_2) < \sqrt{A}/\delta$, and so on, with $d(v_{\delta-1}, x_R) < \sqrt{A}/\delta$. Thus, by the triangle inequality, we have:

$$d(x_L, x_R) \leq d(x_L, v_1) + \sum_{i \in [\delta-2]} d(v_i, v_{i+1}) + d(v_{\delta-1}, x_R) < \delta(\sqrt{A}/\delta) = \sqrt{A}.$$

However, x_L and x_R are on opposite ends on the chip, meaning they are a distance of exactly $\sqrt{A}$ apart from each other in the horizontal direction (and so at least $\sqrt{A}$ total); this is a contradiction.

So there must be some pair of values in the computation graph which have a single path between them and are at least $\sqrt{A}/\delta$ apart on the chip. By assumption 3 (and the argument made in theorem 2.2), this path – and thus the computation as a whole – must take at least $\Omega(\sqrt{A}/\delta)$ time total. □

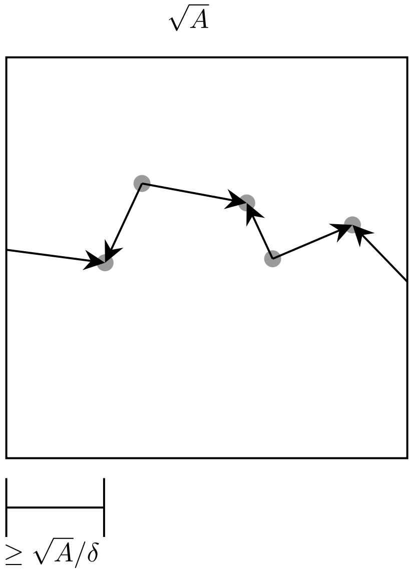


Figure 6: The path diameter bound of theorem 2.6

The reader may notice that in fact, theorem 2.2 can be viewed as a special case of theorem 2.6, as any computation graph implementing a one-output computation in which all values are useful has path diameter at most 2, due to communication with the output. We proved them separately in order to provide better intuition, and because it is not clear how to state theorem 2.6 in the same unconditional language as theorem 2.2, without appealing to the specific computation graph being implemented.

Corollary 2.7 ($T \geq \Omega(\sqrt[3]{N/\delta^2})$)**.** *Implementing a computation graph of path diameter δ on a VLSI chip in time T requires $T \geq \Omega\left(\sqrt[3]{N/\delta^2}\right)$.*

Proof. This, like corollary 2.3, follows directly from theorems 2.6 and 2.1: $T^2 \geq \Omega(A/\delta^2)$, which gives us $T^3 \geq \Omega(N/\delta^2)$ or equivalently $T \geq \Omega(\sqrt[3]{N/\delta^2})$. □

Lemma 2.8 (Matrix Multiplication has constant δ)**.** *Any computation graph implementing dense matrix multiplication in which all values are useful has path diameter at most 6.*

Proof. This fact is best illustrated with a short sequence of symbols:

$$v_1 \rightarrow C_{ij} \leftarrow A_{i0} \rightarrow C_{in} \leftarrow B_{0n} \rightarrow C_{mn} \leftarrow v_2.$$

Given any two values v_1 and v_2, consider the output values they contribute to; call them C_{ij} and C_{mn} respectively. These can be linked by a sequence of 4 paths: C_{ij} depends on row i of A, and C_{mn} depends on column n of B; C_{in} depends on both of these. This is 6 paths total, linking our two arbitrary values v_1 and v_2. □

Theorem 2.9 (Matrix Multiplication takes $T \geq \Omega(n)$)**.** *Implementing multiplication of two dense $n \times n$ matrices on a chip using time T requires $T \geq \Omega(n)$.*

Proof. This follows from the early VLSI theory results of [14], showing that $AT^2 \geq \Omega(n^4)$, combined with theorem 2.6 and lemma 2.8: $T^4 \geq \Omega(n^4/36)$, meaning $T \geq \Omega(n)$. □

This result was also found by Chazelle and Monier [2], but again using the convex-boundary assumption; we regard our contribution here as showing that this result still holds under more modern assumptions.

We will not give the details here, but it is well-known that this time bound is achievable, for instance by keeping the output matrix in place on the chip and computing it incrementally as a sum of outer products.

Definition 2.10 (Minimum balanced hypergraph cut)**.** Consider the 'hypergraph version' of a directed graph G which is obtained by collecting all edges directed out of each node v into a single hyperedge h_v. The *minimum balanced hypergraph cut* of G is then the minimum number of hyperedges cut by any balanced partition of its hypergraph version.

Note that given a partition of a computation graph, the number of hyperedges it cuts is exactly the number of values which are communicated between one side of the partition and the other. When minimized over all balanced partitions, this quantity is the same as the "minimal information cross-flow" of the computation, but is a more tangible quantity for fixed computation graphs, which is why we use minimum balanced hypergraph cut instead of minimal information cross-flow in the statement of the following theorem.

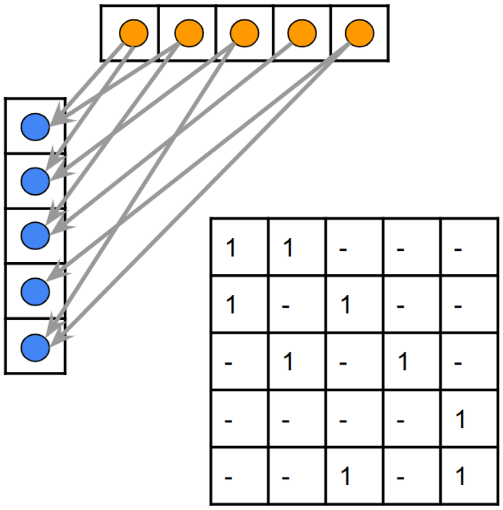
Theorem 2.11 ($AT^2 \geq \Omega(b^2)$)**.** *Implementing a computation graph with a minimum balanced hypergraph cut of size b (or any computation which requires communicating at least b bits across any balanced cut of the inputs and outputs), on a chip of area A and time T, requires $AT^2 \geq \Omega(b^2)$.*

As noted in the explanation of assumption 2, this "A-T-squared" style of bound was the main object of study in 1980s VLSI theory. We will not reproduce a proof here; instead, we refer the reader to [14] for further discussion, and [18] for the original use of the approach and proof.

Corollary 2.12 ($T \geq \Omega(\sqrt{b/\delta})$)**.** *Implementing a computation graph of path diameter δ and minimum balanced hypergraph cut b on a VLSI chip in time T requires $T \geq \Omega\left(\sqrt{b/\delta}\right)$.*

Proof. This follows directly from theorems 2.6 and 2.11: $T^2 \geq \Omega(A/\delta^2)$, which gives us $T^4 \geq \Omega(b^2/\delta^2)$ or equivalently $T \geq \Omega(\sqrt{b/\delta})$. □

Theorem 2.13 (SpMV takes time $T \geq \Omega(\sqrt{b/d})$ and $T \geq \Omega(\sqrt[3]{n/d^2})$)**.** *Multiplying a sparse $n \times n$ matrix A by a dense n-element vector v on a chip in time T requires $T \geq \Omega\left(\sqrt{b/d}\right)$ and, separately, $T \geq \Omega\left(\sqrt[3]{n/d^2}\right)$, where b*



1	1	-	-	-
1	-	1	-	-
-	1	-	1	-
-	-	-	-	1
-	-	1	-	1

Figure 7: A sparse matrix and the bipartite computation graph which its pattern of nonzeros induces

is the minimum hypergraph cut, and d is the diameter, of the bipartite graph whose adjacency matrix has the same pattern of nonzeros as A. Furthermore, this bound applies even when the matrix is known far in advance, and arbitrary precomputation and chip organization can be performed using it.

Proof. This result will follow easily after noting that the dependency graph between values of the input and output vector is exactly the bipartite graph whose adjacency matrix is the matrix by which the input vector is being multiplied: v_i influences $(Av)_j$ if and only if A_{ji} is nonzero. An example of this is shown in figure 7.

So, if this bipartite graph has minimum hypergraph cut b and diameter d, we have $T \geq \Omega\left(\sqrt{b/d}\right)$ by corollary 2.12, and $T \geq \Omega\left(\sqrt[3]{n/d^2}\right)$ by corollary 2.7.

Note that we have not made any appeal to the values input to the computation as part of the sparse matrix itself. We have bounded the execution time not of the computation graph *using* the sparse matrix, but just the computation graph *induced* by it, which means that our bound does not include the time for the matrix to be input and organized, and our argument does not assume anything about when the matrix is learned or how it is stored or used. □

Since some bipartite graphs (specifically, bipartite expander graphs) have minimum balanced hypergraph cut $\Omega(n)$ and diameter $O(\log n)$, for the corresponding matrices we have $T \geq \Omega\left(\sqrt{n/\log n}\right)$.

We will not give any thorough or rigorous analysis of upper bounds for SpMV, but on a $\sqrt{n} \times \sqrt{n}$ grid of values (such as a VLSI chip holding many input values), it appears that all values can be broadcast to all grid points in $\sqrt{n}$ time: the maximum distance traveled is $\sqrt{n}$, and the average congestion is also $\sqrt{n}$. This communication structure, applied to the elements of an input vector, suffices to perform any multiplication by a matrix.

Note that if the matrix A is taken to define a directed graph G on n nodes, instead of an undirected bipartite graph on $2n$ nodes, the computation graph above is almost the same as that for performing an iteration of Bellman-Ford on G. This does not immediately imply a lower bound of $\Omega\left(n\sqrt{b/d}\right)$ for Bellman-Ford, however, as the lower bound of corollary 2.12 is from the first input to the last output, rather than the first output, and so cannot be iterated without some extra arguments or assumptions about synchronization between successive iterations.

2.6 Other models and quantities

An interesting bit of intuition about the AT^2 bounds of the 1980s is that they appear to be *comparing the minimum bisection of the computation to that of the architecture*; that is, a computation graph with a minimum balanced hypergraph cut of b requires $\sqrt{A}T \geq \Omega(b)$ because the chip can be divided into two pieces of equal area, with a communication bandwidth of only $\sqrt{A}$ bits between them. For this reason, the same bound applies equally to specific interconnect structures, 3d volumes, etc: on an architecture with minimum balanced cut c, $cT \geq \Omega(b)$. Note that for a general 3d volume V, $c = V^{2/3}$.

By the same token, it appears that theorem 2.6 above compares the *diameter* of the computation graph (really, the path diameter) to that of the architecture. A computation graph with a path diameter of δ requires $T \geq \sqrt{A}/\delta$ because there are two points on the chip which require $\sqrt{A}$ time to communicate between them; in a sense, the chip has "time diameter" $\sqrt{A}$. Once again, this bound applies elsewhere: on an architecture with time diameter d, $T \geq d/\delta$. Note that for a general 3d volume V, $d = \sqrt[3]{V}$.

Some of the particular approaches we used above which we consider new to VLSI complexity have been used previously for other models; notably, the focus on particular algorithms (like Bellman-Ford) rather than the problems they solve (like shortest paths) is important to bounds on communication between processors and memory [3][7].

In addition to other architecture models, it is useful to consider bounds on other resources in VLSI implementation. In this section, we presented bounds mainly on the longest communication distance required by the algorithm because we were interested in computation latency, but it would be interesting to also see bounds on total communication distance (which is a bound on total energy consumption), on off-chip memory usage and communication, and so forth. We leave these considerations to future work.

3 Digital Designs

In this section, we will present a particular simplified model of synchronous digital designs, which we call "digital design graphs," motivated by timing analysis and timing-driven placement. We will then show how a few standard analysis algorithms fit into this model, and present several lower bounds on placement quality, along with algorithms which apply them. Most of the lower bounds for placement quality are closely analogous to the time lower bounds from the VLSI theory section, but they differ qualitatively and quantitatively because an entire digital design must be mapped to the chip at once, whereas we imposed no such restriction on mapping general computation graphs to chips. We end the section with empirical results from these algorithms on benchmark designs, as well as a few insights into how looking at digital design graphs may inform placement algorithms.

3.1 Digital Design Graphs and Dagification

Definition 3.1 (Digital Design Graphs)**.** A *digital design graph* is a directed graph $G = (V, E)$ along with a set $S \subseteq V$ of 'synchronous' nodes, where all sinks and sources of G are in S and all directed cycles in G contain at least one element of S.

In a design synthesized for a homogeneous FPGA, one can think of the elements of S as flip-flops, latches, and I/Os, while the elements of $V \setminus S$ are LUTs. In a design synthesized for an ASIC, $V \setminus S$ represents all combinational elements.

An edge (u, v) is present in a digital design graph when element v is a sink for the net which has its source at u; that is, when v is a consumer of the value produced by u.

The assumption that all sinks and sources of G are contained in S corresponds to the assumption that the inputs and outputs of the digital design are synchronous; the assumption that all cycles contain at least one element of S corresponds to the assumption that the design contains no combinational loops.

For simplicity, we will assume all elements are single-output, but most algorithms and theorems presented in this section generalize easily beyond this case. For designs with multiple-output components, the digital design graph is replaced by a directed multigraph (that is, a graph with parallel edges allowed), since it is possible that some single component v will receive more than one of the values produced by u, and so there will need to be several different edges (u, v).

A digital design graph is almost exactly the same thing as a netlist, but we redefine it explicitly here anyway because we will want to be careful about exactly which aspects of the design we analyze. Since we will be organizing most of our effort around analyzing and minimizing delay, rather than wire area or routability, it makes sense for us to break up the nets (which are hyperedges) into many individual edges (replace each net with a star graph) so that each

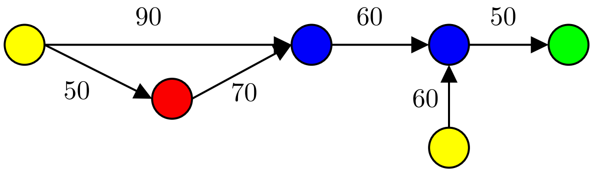


Figure 8: An imagined digital design graph, with imagined delays as edge weights

In this and all pictures of digital designs in this section, yellow nodes indicate inputs to the design, red nodes indicate synchronous internal nodes (e.g. flip-flops), blue nodes indicate logic gates, and green nodes indicate outputs to the design.

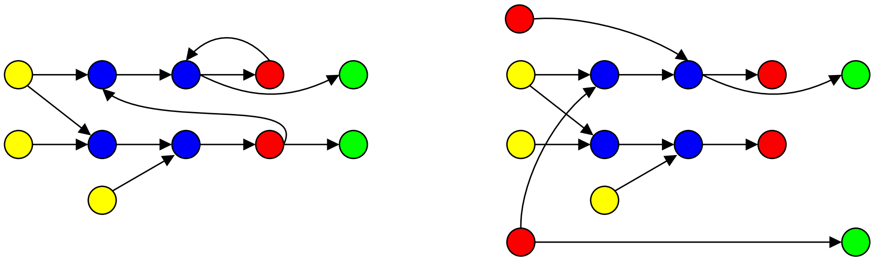


Figure 9: An imagined digital design graph, and its dagified version

Note that the dagification procedure has broken both internal synchronous nodes into an 'input' side and an 'output' side.

edge can be assigned a separate delay without confusion.

Suppose each edge (u, v) of a graph G is weighted by the delay between the elements u and v; in particular, the delay between the value being produced at u and it being received at v, plus the logic delay of v measured from the input port on which it receives from u. For a synchronous element s, we will add the setup time of s to the delay of the edge (x, s) on which s receives its input, and add the clock-to-Q delay of s (if it has one) to the delays of all edges (s, y) on which it produces its output.

We will now see some algorithms for analyzing digital design graphs. Many of the algorithms in this section will depend on a procedure we call "dagification."

Definition 3.2 (The dagification procedure)**.** To "dagify" a digital design graph G with synchronous elements S, split each synchronous element s into two copies s_o and s_i. Instead of edges (x, s), include edges (x, s_i); instead of edges (s, y), include edges (s_o, y); do not connect s_i to s_o. Leave the rest of the graph unchanged.

Since we assumed that the design does not contain any combinational loops, and therefore that all directed cycles in the graph contain a synchronous element, the dagification process will break all cycles in the graph, thus producing a directed acyclic graph or DAG (hence "dagification").

The usefulness of the dagification procedure comes mainly from facts like the following:

Theorem 3.3 (Critical Path Length)**.** *The critical path length in a delay-weighted digital design is equal to the weighted length of the longest path in the dagified version of the digital design graph.*

Proof. The critical path of the digital design is the path which starts at a synchronous element, ends at a synchronous element, contains no synchronous elements besides, and whose total delay is maximum. We can drop the explicit requirement that the path begins and ends at synchronous elements without affecting this definition, since delays are positive and so the paths which include edges up to synchronous elements at the start and end have larger delay than their subpaths which do not; then the critical path is just "the largest delay path which does not pass *through* any synchronous elements" (these paths are exactly the combinational paths in the design).

Note that *no* paths in the dagified version of a digital design pass through a synchronous element, and that by construction there is a one-to-one correspondence, preserving weights, between such paths in the digital design graph and paths in the dagified version. So the longest combinational path in the design exactly corresponds to the longest weighted path in the dagified graph. □

Together with the classic linear-time dynamic programming algorithm for finding the longest path in a DAG, the fact above shows that the dagification procedure gives us a linear time algorithm for finding the critical path in a digital design.

Most timing analysis algorithms (and indeed perhaps most algorithms for analyzing digital designs in any way) almost certainly use something like dagification implicitly; it is not original. The timing engine in nextpnr, for instance, does not construct the DAG explicitly, but it does maintain a topological order of its nodes [16]. Despite these uses, we are not aware of a reference which describes dagification explicitly in the context of digital design analysis. After all, there are much more difficult concepts to be dealt with in actual static timing analysis for ASIC flows (extracting the net and gate delays themselves, crosstalk, clock skew, false and multicycle paths, clock domains, etc). However, we do know that similar concepts have been used in scheduling digital signal processing computations, which are often described in a form similar to that of digital designs.[1]

Note a particularly attractive intuitive property of dagification: the dagified version of a digital design is exactly the computation graph performed by the design in a single cycle. This intuition will be useful for some of the ideas described in the next two subsections.

[1] Thanks for this insight goes to Christopher Yarp, who uses and describes a procedure very much like dagification in his PhD thesis.

3.2 Analysis of Digital Designs

In this section, we will use the notions of digital design graphs and dagification in several algorithms useful for static analysis of digital designs.

3.2.1 Criticality

Theorem 3.4 (Criticality)**.** *There is a linear-time algorithm which computes, for all edges in a (weighted) digital design, the (weighted) length of the longest combinational path which contains that edge.*

Proof. The algorithm proceeds as follows: first, dagify the digital design graph and compute the topological order of the dagified graph. Then, for each node in the dagified graph, compute the (weighted) length ℓ_s of the longest path which starts at that node, and the (weighted) length ℓ_t of the longest path which ends at that node. Then, for each edge (u, v) in the graph, the length of the longest path which contains it is

$$\ell_t(u) + w_{u,v} + \ell_s(v)$$

where $w_{u,v}$ is the weight of the edge (u, v) (or 1 if the graph is unweighted).

The ℓ_s and ℓ_t values can be computed in linear time total through the dynamic programming algorithm for longest path: $\ell_t(s) = 0$ for all sources s of the DAG, then for all other vertices v, $\ell_t(v) = \max_u \ell_t(u) + w_{u,v}$. All of these values can be computed without redundant computation by processing vertices in topological order. ℓ_s values can be computed similarly, in reverse topological order.

Given the above, it is easy to see that the algorithm as a whole is correct, since any path containing an edge (u, v) consists of a (possibly empty) path to u, followed by (u, v), followed by a (possibly empty) path from v. It is also easy to see that the algorithm runs in linear time, since dagification, topological sort, and aggregating values for every edge all take linear time. □

In the weighted case, the criticality notion above is useful for determining which wires should be prioritized for timing optimization during placement (nextpnr appears to use a similar type of criticality for its timing-driven analytical placer).

In the unweighted case (before placement), however, it only gives a crude measure of edge importance, as the criticality of each edge does not take into account the criticality of other edges in the same path. An example of this is given in figure 10. In order to improve the usefulness of our analysis in the unweighted case, we propose a way to "fairly" assign timing budgets to each edge: assign each node v a "time proportion" value $p(v) = \ell_t(v)/(\ell_s(v) + \ell_t(v))$, representing what proportion of the cycle time we expect to have elapsed by the time the node is reached, and then assign each edge a budget equal to the difference in time proportions of its endpoints: (u, v) gets budget $p(v) - p(u)$.

This second notion of criticality is not truly 'fair' either, but it is better than the first.

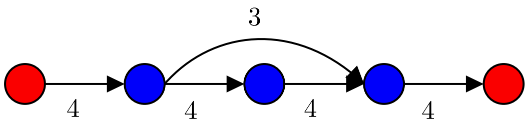


Figure 10: An example where criticality does not provide a good measure of edge importance

Each edge in the graph is labeled with its 'unweighted criticality' (in the sense of theorem 3.4). The edge with criticality 3 does not need to complete before both of the edges below it complete, so it should be considered as having half their timing importance, but it actually has three-quarters their criticality.

3.2.2 Bounds

The criticality analyses above are potentially useful, but they do not incorporate any of the complications arising from the fact that the real performance of digital designs is only determined after they are placed and routed; properties of the design which make it impossible to place without long wires can outweigh the 'post-synthesis' criticality and timing considerations above. In an attempt to address this shortcoming, we now present a few bounds on the possible implementation quality of digital designs, computable from the structure of the digital design graph alone.

Note that these bounds are not asymptotic – they hold, including constant factors and additive constants, for all parameter ranges. As such, they are stated in exact terms, and we will not present examples of designs or tasks for which application of these bounds by hand will yield interesting results. Instead, we will present algorithms in the next subsection which can be used to automatically apply these bounds to input digital design graphs.

Before we present bounds on placement quality, we will present a general bound on the longest edge length in the layout of a graph in two dimensions. This will provide intuition for (and be used in) the later arguments. The bound will rely on the following lemma:

Lemma 3.5 (Number of grid points in two-dimensional ℓ_1 ball)**.** *The number of elements of $\mathbb{Z}^2$ (integer points in 2d space) within ℓ_1 (Manhattan) distance at most k from the origin is at most $2(k+2)^2$ for any nonnegative integer k.*

Proof. We will first count the number of such grid points within the nonnegative quadrant (and find it to be $\binom{k+2}{2}$, including points on the axes); then, since each point is in at least one quadrant and there are 4 quadrants, the total number of points is at most $4\binom{k+2}{2} = 2(k+2)(k+1) \leq 2(k+2)^2$.

To count the number of nonnegative grid points, note that each is uniquely identified by its coordinates (x, y) where $x \geq 0$, $y \geq 0$, and $x + y \leq k$ (and every such pair of coordinates corresponds to a unique point). Equivalently, we can identify points with triples (x, y, z) where $x, y, z \geq 0$ and $x + y + z = k$. The number of choices for these points is exactly the number of ways to distribute k identical balls among 3 bins (think of x as the number of balls in the first bin, y

as the number of balls in the second, and z as the number of balls in the third). By a standard counting argument, this is $\binom{k+2}{2}$. $\square$

Now we can prove the bound itself.

Theorem 3.6 (Maximum distance in placement on grids)**.** *Suppose we have a set S of $|S| = n$ distinct grid points. Then, for any point $a \in S$, there must be some point $b \in S$ such that the ℓ_1 distance $|a_x - b_x| + |a_y - b_y|$ between a and b is at least $\left(\sqrt{\frac{n}{2}} - 2\right)$.*

Proof. Suppose (towards a contradiction) that this is not true: all points are within strictly less than $(\sqrt{\frac{n}{2}} - 2)$ distance of some a. Without loss of generality, we will assume a is the origin (if not, shift all points in the set so that a is at the origin, without changing any distances). However, the number of distinct grid points within distance less than $\sqrt{\frac{n}{2}} - 2$ of the origin is less than n by lemma 3.5. This is a contradiction, since there are n distinct elements in S. $\square$

We will view results such as theorem 3.6 about placement on grids as applying directly to placement on a homogeneous FPGA, given the design to be placed as a graph of (virtual) configurable logic blocks (CLBs). Note that this means we will count wirelength in units of "number of CLBs" (with a wire connecting two adjacent CLBs having length 1, a wire which passes one CLB to connect the two CLBs on either side of it having length 2, etc). The first application of this view is in the following corollary to theorem 3.6.

Corollary 3.7 (Descendant counting)**.** *Let $G = (V, E)$ be a digital design graph where the nodes are CLBs. Suppose there is a vertex $v \in V$ which has k descendants in the dagified version of G; then, in any placement of G there must be a combinational path (in particular, a combinational path starting with v) which has total wirelength at least $\sqrt{\frac{k}{2}} - 2$.*

Proof. It is easy to see by theorem 3.6 that in any placement of G, some descendant u of v must have ℓ_1 distance at least $\sqrt{\frac{k}{2}} - 2$ from v. Therefore, the total length of the path between v and u must be at least $\sqrt{\frac{k}{2}} - 2$ by the triangle inequality. $\square$

This bound can be seen as analogous to corollary 2.3 from the VLSI theory section, and like corollary 2.3, it can be improved by iteration. Suppose we find a set D of $|D| = k_1$ nodes, all of which are descended from v, and all of which have at least k_2 descendants of their own (these sets of descendants need not be disjoint). Then, in any layout of the graph, there must be a path beginning at v of total wirelength at least $\sqrt{\frac{k_1}{2}} + \sqrt{\frac{k_2}{2}} - 4$, which is sometimes larger than the corresponding bound without iteration of $\sqrt{\frac{k_1+k_2}{2}} - 2$. This approach can be continued for an arbitrary number of iterations, finding a node which has k_1 descendants, each of which has k_2 descendants, each of which has k_3

descendants, etc. up to k_m, for a total path length bound of $\sum_{i=1}^{m}\left(\sqrt{\frac{k_i}{2}}-2\right)$. This iteration procedure is very similar to that used for theorem 2.4.

During empirical evaluations, the iterated bound found mainly nodes with large numbers of descendants in the dagified digital design, and reported bounds equal to or only slightly larger than those which would be obtained by applying corollary 3.7 directly. This suggests that the corollary could provide a useful bound without iteration if the largest number of descendants of any node in the graph can be computed more quickly than the potentially stronger iterated bound. It appears that, if we are content with an approximate answer, it can (see theorem 3.13).

Theorem 3.8 (Radius bound for placement on grids)**.** *Let $G = (V, E)$ be an undirected, unweighted graph with n nodes, and let each vertex v of the graph be assigned a unique pair of integers (v_x, v_y) (i.e., let G be placed on a two-dimensional grid without collisions between vertices). Suppose some vertex $c \in V$ has eccentricity at most r (that is, every vertex in V can be connected to c by a path of length at most r). Then, there is some edge $(u, v) \in E$ such that the ℓ_1 distance $|u_x - v_x| + |u_y - v_y|$ between its endpoints is at least $\left(\sqrt{\frac{n}{2}}-2\right)/r$.*

Proof. Suppose (towards a contradiction) that all edges connect endpoints that are placed strictly less than $(\sqrt{\frac{n}{2}}-2)/r$ apart in ℓ_1 distance. Then every vertex in the graph is placed within less than ℓ_1 distance $\sqrt{\frac{n}{2}}-2$ of c. However, the number of other grid points within distance less than $\sqrt{\frac{n}{2}}-2$ of the origin is less than n by lemma 3.5. This is a contradiction, since G has n vertices. $\square$

Definition 3.9 (Path-Condensed digital design)**.** Given a digital design graph $G = (V, E)$ with synchronous elements $S \subseteq V$, the path-condensed version of G is an undirected graph on the same set of vertices V as G and an edge (u, v) whenever there is a combinational path (that is, a path which does not pass through any synchronous elements) from u to v (or vice versa) in G.

Definition 3.10 (Combinational path radius in digital designs)**.** The combinational path radius of a digital design graph (or a subgraph of a digital design graph) G is the undirected radius of the path-condensed version of G (or if G is a subgraph of a larger digital design graph, of the subgraph induced on the nodes of G of the path-condensed version of the larger graph). Equivalently, it is the minimum number r for which there is some node c from which all nodes in G can be reached using a sequence of at most r combinational paths (some potentially backwards).

The combinational path radius of digital design graphs presented here and the path diameter of the VLSI theory section (definition 2.5) are similar, but note the two main differences between them: first, we use radius instead of diameter (so that we can apply theorem 3.8), and second, we use the fact that some of the sources of the dagified graph are in fact the same nodes as some of its sinks, by allowing paths to connect through synchronous elements. Indeed, the path diameter of the dagified graph is an upper bound for the combinational path diameter of the graph (if it is defined analogously to combinational path radius), and the path-condensed version of a digital design graph can be found by dagifying it, taking the transitive closure, and then "undagifying" by merging each pair of s_i and s_o nodes back together. We will use combinational path radius in a bound analogous to that of corollary 2.7.

Theorem 3.11 (Combinational path radius bound)**.** *Suppose a digital design graph G has a subgraph of size m with combinational path radius r; then in any placement of G there must be a combinational path which has ℓ_1 length at least $\left(\sqrt{\frac{m}{2}} - 2\right)/r$.*

Proof. Applying theorem 3.8 immediately gives us that in any placement of G, one of the edges from the path-condensed version of G has length at least $(\sqrt{\frac{m}{2}} - 2)/r$. By construction of the path-condensed version, the endpoints of this edge are connected by a combinational path in G. This path must have total length at least $(\sqrt{\frac{m}{2}} - 2)/r$ by the triangle inequality. □

Theorem 3.12 (Weighted radius bound)**.** *Let $G = (V, E)$ be a (weighted) digital design graph (or a subgraph of one) with m nodes and undirected weighted radius r. Then in any placement of G, there must be some edge $e \in E$ with ℓ_1 length $w_e \cdot \left(\sqrt{\frac{m}{2}} - 2\right)/r$.*

Proof. This fact can be seen through a similar argument to that made for theorem 3.8. By lemma 3.5, some node must be at least $(\sqrt{\frac{m}{2}} - 2)$ away from the central element c in ℓ_1 distance. This element is connected to c by a path p of weighted length $\sum_{e \in p} w_e \leq r$, so by an averaging argument, at least one edge e on this path must have ℓ_1 length at least $w_e \cdot (\sqrt{\frac{m}{2}} - 2)/r$ (otherwise the total length of the path would be strictly less than $\sum_{e \in p}(w_e \cdot (\sqrt{\frac{m}{2}} - 2)/r) \leq (\sqrt{\frac{m}{2}} - 2)$, which is a contradiction). □

Note that we have had to be careful about which quantities we bound in order to make the bounds useful. Applying theorem 3.8 to a digital design directly would lead only to a bound on the maximum wirelength in the placement; while this is useful in that it does provide a lower bound on the length of the longest combinational path, it is a fairly weak bound. In fact, even bounds on total wirelength would not tell us much – if we know that any placement of the graph must have several long wires, we care very much whether it is also the case that those wires *must be in the same combinational path.*

The bounds above represent a few different ways of handling this subtlety. The basic descendant-counting bound works only with distances between nodes which are both members of at least one common combinational path. The path radius bound finds distances between nodes which can be connected through k combinational paths, but then divides by k to get a bound on the distance between two nodes in the same path. Note that both of these suffer from the same problem – they provide a bound only on the distance between two nodes in a path, and they take this distance as the bound on the path wirelength. If all placements of a particular digital design require that some path "zig-zags" heavily, making its length much larger than the distance between any two of its nodes, the basic descendant-counting bound and the path radius bound will not capture this behavior. The iterated descendant-counting bound, on the other hand, appears to have no such restriction, and could perhaps find larger bounds in cases where "zig-zags" are necessary.

The weighted radius bound is qualitatively different from the others – rather than providing an unconditional bound on the length of the longest edge, it finds a bound which is relative to provided weights. If we think of w_e as a timing budget (really, a wirelength budget) for edge e, then the bound implies that it is impossible to hit all budgets simultaneously whenever there is a subgraph for which $(\sqrt{\frac{m}{2}} - 2)/r$ is greater than 1.

3.2.3 Algorithms computing bounds

Most of the bounds presented above suggest fairly natural, simple algorithms for computing them given a digital design. For the combinational path radius bound, compute the path-condensed version P of the design, and run breadth-first search from every node; this will enumerate, for all nodes c and all integers r, the largest subgraph of P with radius r centered at c. Whichever of these subgraphs maximizes $(\sqrt{\frac{m}{2}} - 2)/r$ defines the bound. Similarly, the weighted radius bound can be computed by running a shortest-paths-tree algorithm from every node in the undirected version of the digital design, and the descendants bound can be computed simply by finding the number of descendants of every node in the dagified graph. We will present an explicit algorithm for computing the iterated descendants bound, and we will also present a nearly-linear-time algorithm for counting descendants (the naive algorithm requires quadratic time).

Algorithm 1 Computing the bound from iterated descendant counting

```
procedure IteratedBound(Dagified digital design graph $D$)
    $b(s) \leftarrow 0$ for all sinks $s$ of $D$
    for each node $u$ of $D$, in reverse topological order do
        $b(u) \leftarrow 0$
        $i \leftarrow 0$
        for each descendant $v$ of $u$, sorted by decreasing $b(v)$ do
            $i \leftarrow i + 1$
            $b(u) \leftarrow \max(b(u), \sqrt{\frac{i}{2}} - 2 + b(v))$
        end for
    end for
end procedure
```

We will not provide a formal analysis of algorithm 1, but note that it can be run in $O(n^2 \log n)$ time. Also, note that if the optimal bound on node u counts a set of descendants, the lowest bound among which is $b(v)$, then we lose nothing by including all descendants of u with bounds at least $b(v)$; so the "go through descendants in order of bound size and count how many have been considered so far" approach used by algorithm 1 is valid and optimal.

As an aside, algorithm 1 can actually be modified to include logic delay in the bound, by adding an appropriate constant to $b(u)$ at the end of each iteration of the outer loop. This makes it potentially more attractive for practical analysis, but we do not include logic delay in our empirical results.

Theorem 3.13 (Nearly Linear-Time Descendant Counting)**.** *There is a randomized algorithm which runs in time $O(n \log^2 n)$ on a DAG of n nodes and $O(n)$ edges and approximates for every node v the number of descendants of v within 10% accuracy with probability at least 99%.*

Proof. The main idea of this algorithm will be to use a sketching data structure, so that for each node, we can represent its set of descendants with only $O(\log^2 n)$ bits.

In particular, we will use the datastructure from the KMV (k-minimum values) algorithm [1][11]. The main purpose of the KMV datastructure is to return the number of distinct elements in a stream of n elements, without storing all distinct elements (in fact, using only $O(\log n)$ bits). We refer the reader to [11] (or the original paper, [1]) for a discussion of how it does this and the performance it achieves. We will only need the following properties from it:

1. $\texttt{KMV}_{\varepsilon,\delta,n}()$ (for $\varepsilon, \delta > 0$ and n positive integer) instantiates a copy of the KMV datastructure using $\Theta(\varepsilon^{-2} \log(1/\delta) \log(n))$ bits, in time $O(\varepsilon^{-2} \log(1/\delta) \log(n))$. At instantiation, the datastructure represents an empty set.

2. $a.\texttt{update}(m)$, for a $\texttt{KMV}_{\varepsilon,\delta,n}$ datastructure a and element $m \in [n]$, adds m to the set of elements represented by a in time $O(\varepsilon^{-2} \log(1/\delta) \log(n))$.

3. $a.\texttt{merge}(b)$, for two $\texttt{KMV}_{\varepsilon,\delta,n}$ datastructures a and b, updates a to represent the *union* of the sets represented by a and b, in time $O(\varepsilon^{-2} \log(1/\delta) \log(n))$.

4. $a.\texttt{count}$, for a $\texttt{KMV}_{\varepsilon,\delta,n}$ datastructure a, returns an estimate $\hat{t}$ of the total number of elements in the set represented by a in time $O(\varepsilon^{-2} \log(1/\delta) \log(n))$. This estimate is very likely to be nearly correct in that $|\hat{t} - t| > \varepsilon t$ for the true count t (that is, it differs from the true count by more than 100ε percent) with probability at most δ.

Our algorithm is as follows:

Algorithm 2 Nearly Linear-Time Descendant Counting

```
procedure DescCount(Dagified digital design graph $D$, with $n$ nodes identified by integers in $[n]$)
    Initialize an empty map KMVSets, from nodes to KMV datastructures
    for each node $u$ of $D$, in reverse topological order do
        KMVSets[$u$] ← KMV$_{1/10,1/(100n),n}$()
        KMVSets[$u$].update($u$)
        for each out-neighbor $v$ of $u$ do
            KMVSets[$u$].merge(KMVSets[$v$])
        end for
    end for
    return KMVSets[$u$].count, for all nodes $u$
end procedure
```

It is easy to see that algorithm 2 runs in time $O(n \log^2 n)$; it performs one $\texttt{KMV}_{1/10,1/(100n),n}$ intialization and one UPDATE for each node in the graph, along with one MERGE for each edge; each of these takes

$$O(\varepsilon^{-2} \log(1/\delta) \log(n)) = (100 \log(100n) \log(n)) = O(\log^2 n) \text{ time.}$$

In order to see the correctness of the algorithm, note first that it would be exactly correct if the KMV datastructure represented sets without error. For each node u, the set of descendants of u is u itself, plus the union of the sets of descendants of u's immediate children.

Now, note that by the bound given in [1], for each individual u,

$$\Pr[|\texttt{KMVSets}[u]\texttt{.count} - desc(u)| \geq desc(u)/10] \leq \delta = 1/(100n).$$

We are actually interested in the probability that *any* node errors by more than 10%. To find this, we can simply take a union bound over all nodes:

$$\Pr[\text{any error}] \leq \sum_{u \in [n]} 1/(100n) = 0.01$$

□

In the algorithm above, we have glossed over one subtlety: instantiating the KMV datastructure involves picking some random hash functions, and in order to `merge` two instances, they will need to use the same hash functions. We ignored this for simplicity in the theoretical algorithm above; our practical implementation does not require considering it.

We have presented the result above in full for convenience and because we discovered it independently, but it is not new; a similar algorithm is described in [10], and a similar result with a different framework is given by [4].

One note about the lower bounds here is that because they can be found for input graphs in polynomial time, we should not expect them to be close to the actual best placement quality for all graphs (just as we should not expect a polynomial-time placement algorithm to find a placement of close to optimal quality for all graphs). A lower bound algorithm finding a bound within a small factor of optimal for all graphs would suffice to solve gapped problems: distinguish whether the optimal placement of a design has maximum path length at most some ℓ, or at least $k\ell$ for some factor k (this is the same setting used for hardness of approximation). We are not aware of an analysis showing hardness of gapped problems for the precise problem of minimizing the maximum path length in a two-dimensional layout, but a similar problem in one dimension, the *graph bandwidth* problem, is known to be hard [5].

3.2.4 Empirical results

We ran the algorithms described above to compute lower bounds on a set of common FPGA benchmarks, taken from the VTR benchmark suite [9]. Since we are focused on simple homogeneous fabrics and we wish to compare our lower bounds to actual placements, we only include those benchmarks which do not contain large memories and which worked easily with our synthesis and placement flow.

All benchmarks listed were synthesized with yosys-abc9, and packed and placed using nextpnr [16]. The target architecture was a homogeneous fabric of CLBs, each consisting of one 4-LUT and one flip-flop (nextpnr's 'generic' architecture). The fabric was a square grid, oversized by 10% in both the horizontal and vertical directions relative to the number of LUTs plus the number of flip-flops in the design. IOs were placed on the edges. The placer used default settings, except for the 'criticality exponent,' which was set to 5 to provide heavily timing-driven placements. For each benchmark, 5 random scripts for the abc tool were generated, and placements were generated for each; only the one which produced the shortest maximum path length was used.

The numbers reported were calculated using a graph read from the netlist after packing by nextpnr (that is, a graph of CLBs, where some use the flip-flop and some do not). We report our results in 3 tables.

Table 1 reports basic information for each benchmark:

1. Total number of nodes (CLBs after packing)
2. Unweighted length of longest path in dagified graph (equivalently, the maximum unweighted criticality in the sense of theorem 3.4)
3. The percentage of nodes in the graph which have at least one connected edge with unweighted criticality within 20% of the maximum (equivalently, which are on at least one path of unweighted length within 20% of the maximum).
4. The (exact) maximum number of descendants that any node has in the dagified graph

5. The maximum number of descendants that any node has in the dagified graph (as estimated by algorithm 2)

For convenience, our practical implementation of algorithm 2 actually used the HyperLogLog++ datastructure [6] as implemented by the datasketch python package [19] instead of the k-minimum-values datastructure of [1]. In addition, we used a fixed parameter ($p = 12$) rather than varying the space complexity with the number of nodes in the design.

Benchmark	Nodes	Max. Crit.	Crit. Frac.	Max Desc. (exact)	Max Desc. (HLL)
`blob_merge`	5396	16	3.06%	369	369.15
`diffeq1`	4233	21	53.6%	2592	2607.79
`diffeq2`	3895	18	54.74%	2138	2163.96
`sha`	1772	11	8.86%	998	999.64
`stereovision0`	13421	5	0.43%	2016	2025.56
`stereovision1`	27227	8	5.21%	2784	2825.15
`stereovision2`	55091	14	12.52%	1526	1518.78

Table 1: Basic numbers for VTR benchmarks

Table 2 reports results of the lower bounds:

1. The bound found by the iterated descendants method (algorithm 1)
2. The bound found using path radius (theorem 3.11)
3. The pseudo-bound found using weighted radius (theorem 3.12), with weights assigned by the 'fair budget assignment' procedure mentioned below theorem 3.4
4. The length (in sum of Manhattan distances) of the longest path in the actual placement found by nextpnr

For all the designs used here, the iterated descendants bound actually matched the non-iterated maximum descendant count bound exactly, and so we do not bother reporting them in two separate columns. We know anecdotally that iteration sometimes provides a higher bound, though.

Benchmark	(Iter.) Desc.	Path	Weighted	nextpnr
`blob_merge`	11.583	19.233	27.174	114
`diffeq1`	34.0	13.37	77.645	302
`diffeq2`	30.696	11.121	54.37	250
`sha`	20.338	12.463	20.338	72
`stereovision0`	29.749	29.749	29.749	280
`stereovision1`	35.31	35.242	40.393	325
`stereovision2`	25.622	26.502	33.325	452

Table 2: Results of lower-bound algorithms on VTR benchmarks

The results in table 2 show that there is often a fairly large discrepancy between the computed lower bounds and the actual placed wirelength. In order to determine whether the bounds have identified some true 'difficult-to-place' part of the design regardless, we also computed the average wirelength in each design, and the average wirelength in the subgraph of the design which formed the bound (the subgraph induced on the nodes which "participated" in the maximum bound). For instance, in the case of the descendants bound, if the node with the most descendants is u, we report the average wirelength of all edges both of whose endpoints are descendants of u. Table 3 reports these results:

1. The average length of edges both of whose endpoints participate in the simple descendants bound
2. The average length of edges both of whose endpoints are among the m nodes used for the $(\sqrt{\frac{m}{2}} - 2)/r$ combinational path radius bound
3. The average length over all edges in the whole design
4. The average weighted length of edges both of whose endpoints are among the m nodes used for the $(\sqrt{\frac{m}{2}} - 2)/r$ weighted radius bound
5. The average weighted length over all edges in the whole design

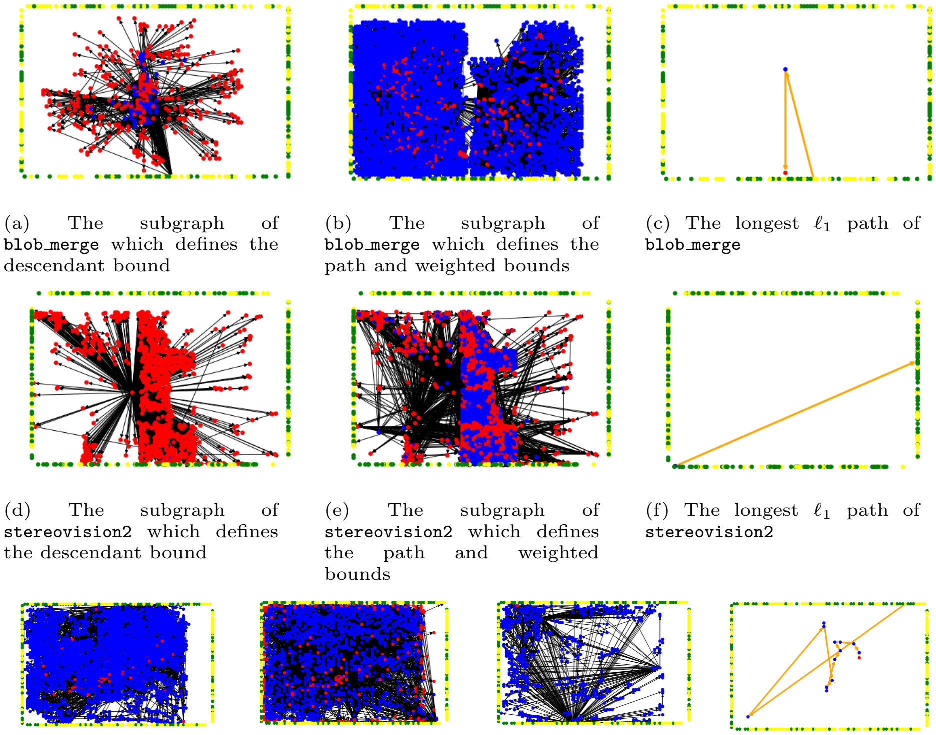
All weights are assigned by the 'fair budget assignment' procedure mentioned below theorem 3.4.

Benchmark	Descendants	Path Radius	Overall	Weighted Radius	Weighted Overall
blob_merge	14.16	23.48	21.52	17.08	16.61
diffeq1	10.21	31.27	20.62	2.61	6.71
diffeq2	7.98	18.35	17.5	1.03	5.2
sha	11.83	10.81	10.78	8.16	7.64
stereovision0	26.59	26.59	20.8	26.59	19.58
stereovision1	35.08	35.23	29.69	18.02	18.84
stereovision2	47.88	28.12	19.31	14.89	13.02

Table 3: Average wirelength among nodes participating in each bound vs entire graph

The average wirelength on the 'bound-defining subgraph' is often much larger than the average wirelength in the design overall, especially for the path radius bound; we see this as encouraging.

Figure 11 shows some of these bound-defining subgraphs, as well as some of the longest-total-wirelength paths, embedded within the placement of the design by nextpnr. For some of the benchmarks, two or more of the bounds are actually defined by the exact same subgraph.



(a) The subgraph of `blob_merge` which defines the descendant bound

(b) The subgraph of `blob_merge` which defines the path and weighted bounds

(c) The longest ℓ_1 path of `blob_merge`

(d) The subgraph of `stereovision2` which defines the descendant bound

(e) The subgraph of `stereovision2` which defines the path and weighted bounds

(f) The longest ℓ_1 path of `stereovision2`

(g) The subgraph of `diffeq1` which defines the descendant bound

(h) The subgraph of `diffeq1` which defines the path radius bound

(i) The subgraph of `diffeq1` which defines the weighted radius bound

(j) The longest ℓ_1 path of `diffeq1`

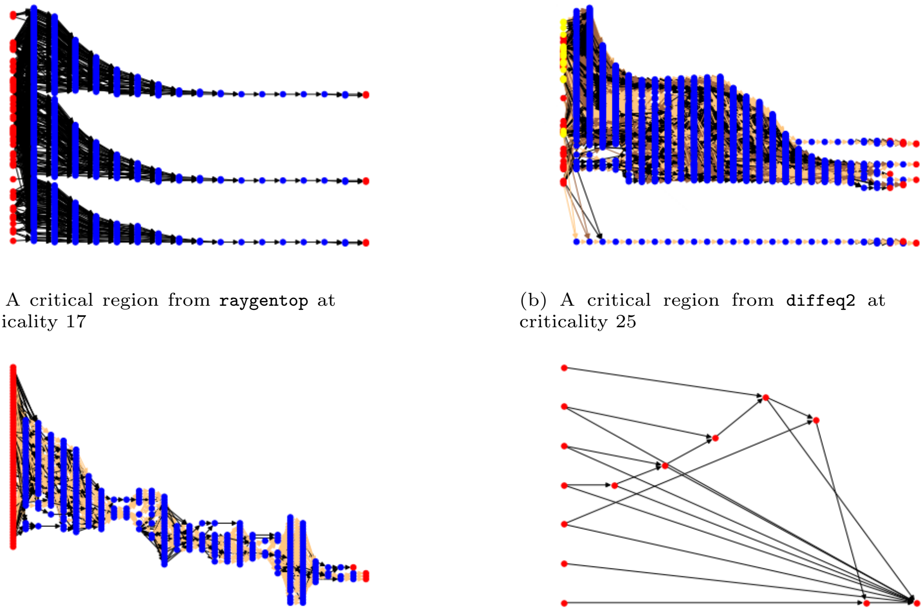
Figure 11: Bound-defining node sets and longest-wirelength paths of some benchmarks, shown with node locations given by the nextpnr placement, plus I/O nodes for context

3.3 Critical Regions

Intuitively, combinational paths with more edges are more likely to become critical after placement; so it makes sense to focus primarily on these paths when designing FPGA architectures and placement algorithms.

Definition 3.14 (Critical Region)**.** A *critical region* at criticality k of a (possibly dagified) unweighted digital design graph G is a maximal weakly-connected subgraph of G in which all edges have criticality k or higher (in the sense of theorem 3.4).

We include the condition that they be *maximal* subgraphs satisfying this property; this means for instance that a single critical path does not constitute a full critical region if other critical paths are connected to it (although some of our observations about critical regions will also apply to smaller parts of critical regions).



(a) A critical region from `raygentop` at criticality 17

(b) A critical region from `diffeq2` at criticality 25

(c) A critical region from `bgm` at criticality 27

(d) Connections among several critical regions in `mcml`

Figure 12: A few critical regions taken from VTR benchmarks

Figure 12 shows some examples of critical regions in dagified digital designs from FPGA benchmarks. Looking at those examples, we can notice the first interesting property of critical regions.

Theorem 3.15 (Critical regions are layered)**.** *The nodes of a critical region at criticality k of a dagified digital design graph can be assigned integer 'layers' $\ell(\cdot)$ such that every edge (u, v) in the region satisfies $1 \leq \ell(v) - \ell(u) \leq m - k + 1$, where m is the maximum criticality of any edge in the region.*

Proof. The layer assignment for which this property holds will actually be to set $\ell(v) = \ell_t(v)$ for all nodes v; that is, layer nodes according to the length of the longest combinational path in the design which ends with them.

To see that this layering gives $1 \leq \ell(v) - \ell(u)$ for all edges (u, v), note that there is a path of length $\ell(u)$ ending at u. Adding (u, v) to this path produces a path ending at v of length $\ell(u) + 1$, meaning that the length of the *longest* path ending at v must be at least $\ell(v) \geq \ell(u) + 1$.

To see that it gives $\ell(v) - \ell(u) \leq m - k + 1$, note that all paths from source to sink in the region have length between k and m. This includes the longest path containing v, which has length $\ell_t(v) + \ell_s(v)$. It also includes the path created by joining the longest path ending at u with the edge (u, v) and the longest path starting at v, which has total length $\ell_t(u) + 1 + \ell_s(v)$. So we have

$\ell_t(v) + \ell_s(v) \leq m$ and $-\ell_t(u) - 1 - \ell_s(v) \leq -k$, which when added together give us $\ell_t(v) - \ell_t(u) - 1 \leq m - k$, which gives us our result. □

Note that this fact does not quite apply to critical regions in non-dagified digital designs, as critical paths in one pipeline stage may connect to critical paths in others in non-obvious ways through synchronous elements. Figure 12d shows a particularly nasty example from one synthesis of the `mcml` benchmark from the VTR suite. In that graph, each node represents a collection of flip-flops, and each edge represents a collection of (maximum-length) combinational paths connecting the flip-flops at its endpoints. Even though this part of the design contains no cycles and all edges have the same criticality, the layering property does not apply without modification.

Perhaps the most striking immediate consequence of theorem 3.15 is that single-stage regions where all edges have the maximum criticality can be placed perfectly in layers, with every edge between adjacent layers. This suggests a timing-first approach to FPGA interconnect design and placement strategy, where these "layered regions" are placed on "layered interconnect," with abundant fast routing resources connecting adjacent columns of the fabric and less attention paid to wires which cross many columns or do not cross any. Unfortunately, after much exploration of this idea, it appears that a huge number of these inter-column resources would be needed (unless large improvements are made in finding good orderings of CLBs within columns) and it is not clear how these routing resources could be provided with the quantity and quality necessary to make this idea viable. We leave further exploration to future work.

4 Summary, conclusions, and future work

In this report, we presented an updated theory of VLSI computation, together with several asymptotic lower bounds on execution time in this new model, some of which have not been previously described in any similar model.

We also presented related bounds on the length of the longest path in FPGA placements, and showed how they could be computed using efficient algorithms, yielding tools for analyzing digital designs in the context of timing-driven implementation quality.

We propose that both of these areas should be studied further. We hope that future work will further explore the theoretical implications of the VLSI model presented, and extend our results to other performance measures and quantities of interest, other variations on model assumptions, and other algorithms and tasks. More importantly, we hope this work can be extended to inform actual chip designs and execution schedules.

We also hope that future work will improve our approach to bounding placement quality for digital designs. If tools like the ones presented here are reimagined using stronger bounds on total path length, as well as bounds on other quantities like total wirelength and timing slack, they could become a very useful addition to the techniques used to implement digital designs, both in placement-aware synthesis and in placement strategies themselves.

References

[1] Ziv Bar-Yossef et al. "Counting Distinct Elements in a Data Stream". In: *Randomization and Approximation Techniques in Computer Science*. Ed. by José D. P. Rolim and Salil Vadhan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 1–10. ISBN: 978-3-540-45726-8.

[2] Bernard Chazelle and Louis Monier. "A Model of Computation for VLSI with Related Complexity Results". In: *Journal of the ACM* 32 (1985), pp. 573–588.

[3] Michael Christ et al. *Communication lower bounds and optimal algorithms for programs that reference arrays – Part 1*. 2013. DOI: 10.48550/ARXIV.1308.0068. URL: https://arxiv.org/abs/1308.0068.

[4] Edith Cohen. "Size-Estimation Framework with Applications to Transitive Closure and Reachability". In: *Journal of Computer and System Sciences* 55.3 (1997), pp. 441–453. ISSN: 0022-0000. DOI: https://doi.org/10.1006/jcss.1997.1534. URL: https://www.sciencedirect.com/science/article/pii/S0022000097915348.

[5] Chandan Dubey, Uriel Feige, and Walter Unger. "Hardness Results for Approximating the Bandwidth". In: *J. Comput. Syst. Sci.* 77.1 (Jan. 2011), pp. 62–90. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2010.06.006. URL: https://doi.org/10.1016/j.jcss.2010.06.006.

[6] Stefan Heule, Marc Nunkesser, and Alex Hall. "HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm". In: *Proceedings of the EDBT 2013 Conference*. Genoa, Italy, 2013.

[7] Hong Jia-Wei and H. T. Kung. "I/O Complexity: The Red-Blue Pebble Game". In: *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*. STOC '81. Milwaukee, Wisconsin, USA: Association for Computing Machinery, 1981, pp. 326–333. ISBN: 9781450373920. DOI: 10.1145/800076.802486. URL: https://doi.org/10.1145/800076.802486.

[8] Charles E. Leiserson. "Area-efficient graph layouts". In: *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*. 1980, pp. 270–281. DOI: 10.1109/SFCS.1980.13.

[9] Jason Luu et al. "VTR 7.0: Next Generation Architecture and CAD System for FPGAs". In: *ACM Trans. Reconfigurable Technol. Syst.* 7.2 (July 2014). ISSN: 1936-7406. DOI: 10.1145/2617593. URL: https://doi.org/10.1145/2617593.

[10] Suman Nath et al. “Synopsis Diffusion for Robust Aggregation in Sensor Networks”. In: *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems.* SenSys ’04. Baltimore, MD, USA: Association for Computing Machinery, 2004, pp. 250–262. ISBN: 1581138792. DOI: `10.1145/1031495.1031525`. URL: `https://doi.org/10.1145/1031495.1031525`.

[11] Jelani Nelson. *Sketching Algorithms.* `https://www.sketchingbigdata.org/fall20/lec/notes.pdf`. Dec. 2020.

[12] M. S. Paterson, W. L. Ruzzo, and L. Snyder. “Bounds on Minimax Edge Length for Complete Binary Trees”. In: *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing.* STOC ’81. Milwaukee, Wisconsin, USA: Association for Computing Machinery, 1981, pp. 293–299. ISBN: 9781450373920. DOI: `10.1145/800076.802481`. URL: `https://doi.org/10.1145/800076.802481`.

[13] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital integrated circuits- A design perspective.* 2ed. Prentice Hall, 2004.

[14] John E. Savage. “Area—time tradeoffs for matrix multiplication and related problems in VLSI models”. In: *Journal of Computer and System Sciences* 22.2 (1981), pp. 230–242. ISSN: 0022-0000. DOI: `https://doi.org/10.1016/0022-0000(81)90029-5`. URL: `https://www.sciencedirect.com/science/article/pii/0022000081900295`.

[15] John E. Savage. *Models of Computation: Exploring the Power of Computing.* 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201895390.

[16] David Shah et al. *Yosys+nextpnr: an Open Source Framework from Verilog to Bitstream for Commercial FPGAs.* 2019. DOI: `10.48550/ARXIV.1903.10407`. URL: `https://arxiv.org/abs/1903.10407`.

[17] C. D. Thompson. “A Complexity Theory for VLSI”. PhD thesis. Carnegie-Mellon University, 1980. DOI: `10.1184/R1/6714269.v1`. URL: `https://kilthub.cmu.edu/articles/thesis/A_Complexity_Theory_for_VLSI/6714269`.

[18] C. D. Thompson. “Area-Time Complexity for VLSI”. In: *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing.* STOC ’79. Atlanta, Georgia, USA: Association for Computing Machinery, 1979, pp. 81–88. ISBN: 9781450374385. DOI: `10.1145/800135.804401`. URL: `https://doi.org/10.1145/800135.804401`.

[19] Eric Zhu. *datasketch.* `https://ekzhu.github.io/datasketch`. Version 1.5.7.