Full accounting for verifiable outsourcing

Riad S. Wahby[⋆] rsw@cs.stanford.edu
Ye Ji[◦] yj682@nyu.edu
Andrew J. Blumberg[†] blumberg@math.utexas.edu
abhi shelat[‡] abhi@neu.edu
Justin Thaler[△] justin.thaler@georgetown.edu
Michael Walfish[◦] mwalfish@cs.nyu.edu
Thomas Wies[◦] wies@cs.nyu.edu

[⋆]Stanford [◦]NYU [†]UT Austin [‡]Northeastern [△]Georgetown

Abstract. Systems for verifiable outsourcing incur costs for a *prover*, a *verifier*, and precomputation; outsourcing makes sense when these costs are cheaper than not outsourcing. Yet, prover costs are generally ignored. The only exception is Verifiable ASICs (VA), wherein the prover is a custom chip; however, the only prior VA system ignores the cost of precomputation.

This paper describes a new VA system, called *Giraffe*; charges Giraffe for all three costs; and identifies regimes where outsourcing is worthwhile. Giraffe's base is an interactive proof geared to data parallel computation. Giraffe makes this protocol *asymptotically optimal* for the prover, which is of independent interest. Giraffe also develops a *design template* that produces hardware designs automatically for a wide range of parameters, introduces hardware primitives molded to the protocol's data flows, and incorporates program analyses that expand applicability. Giraffe wins even when outsourcing several tens of sub-computations, scales to 500× larger computations than prior work, and can profitably outsource *parts* of programs that are not worthwhile to outsource in full.

1 Introduction

In probabilistic proofs—Interactive Proofs (IPs) [12, 42, 43, 49, 65], arguments [26, 44, 45, 53], SNARGs [41], SNARKs [24, 40] and PCPs [9, 10]—a *prover* convinces a *verifier* of an assertion by supplying a proof (possibly interactively); a false assertion is highly likely to cause rejection. These protocols are foundational in complexity theory and cryptography. There has also been substantial progress in implementations over the last six years [14, 15, 17–19, 21, 22, 28, 29, 31–35, 38, 40, 47, 55, 56, 61–64, 66, 68–71] (for a survey, see [72]), based on theoretical refinements and systems work.

A central application example is *verifiable outsourcing*. The verifier specifies a computation and input; the prover returns the (purported) output and proves the assertion that "the returned output equals the computation applied to the input." The essential property here is that the verifier's probabilistic checks are asymptotically less expensive than executing the computation; as a result, outsourcing can be worthwhile for the verifier. This picture motivated a lot of the original theory [13, 39, 42, 53]; today, stories about cloud computing rehearse that motivation. To validate these stories in the context of real implementations, there are three cost sources to consider:

- *Prover overhead.* Even in the best general-purpose probabilistic proof protocols, the prover has enormous overhead in running the protocol versus simply executing the underlying computation: the ratio between these is typically at least 10^7 [72, Fig. 5].
- *Precomputation.* Many of the implemented protocols require a setup phase, performed by the verifier or a party the verifier trusts. This phase is required for each computation and can be reused over different input-output instances. Its costs are usually proportional to the time to run the computation. (Precomputation can be asymptotically suppressed or even eliminated, but at vastly higher concrete cost [17, 21, 22, 29]; see §10.)
- *Verifier overhead.* Separate from precomputation, there are inherent protocol costs that the verifier incurs for each input-output instance. These costs are at least linear in the input and output lengths.

More or less tacitly, "practical" work in this area has bundled in assumptions about the regimes in which these costs are reasonable for the operator of the verifier.[1] For example, one way to tame the costs is not to charge the operator for precomputation. This is the approach taken in Pinocchio, which focuses on per-instance verifier overhead [56, 57].[2] This choice can be justified if there is a trusted third party with extremely inexpensive cycles.

Another possibility is to target data parallel computations, meaning identical sub-computations on different inputs. Here, one can charge the operator of the verifier for the precomputation (which amortizes) and then identify *cross-over points* where the verifier saves work from outsourcing [28, 32, 56, 62–64, 69, 71].

In both of these cases, prover overhead is measured but in some sense ignored (when considering whether outsourcing is worthwhile). This would make sense if the prover's cycles were vastly cheaper than the verifier's (the required ratio is approximately the prover's overhead: $10^7\times$), or if the outsourced computation could not be executed in any other way.

[1] A variant of this story, exploiting an exciting property of some probabilistic proofs [24, 40], involves "zero knowledge" applications where the proof can incorporate input hidden from the verifier [18, 33, 55, 56]. Here, one is often more concerned about the prover's overhead. Nevertheless, the same efforts to identify regimes where overhead is reasonable arise. We do not discuss this in detail, but see §10.

[2] Pinocchio certainly considers precomputation [56, §5.3], but its emphasized comparison is between native execution and verifier overhead.

Recently, Zebra [70] used a different justification by observing that one can gain high-assurance execution of custom chips (ASICs) by using trusted slow chips to verify the outputs of untrusted fast chips. In this *Verifiable ASICs* (VA) domain (§2.3), one can charge the operator for both verifier and prover and still identify regimes where their combination outperforms a baseline of simply executing the given functionality in a trusted slow chip. However, Zebra does not charge for precomputation (and worse, introduces a preposterous assumption about the daily delivery of hard drives to handle the problem).

The work of this paper is to create a system, Giraffe, to *charge the operator for all three costs*, and to seek out regimes where this combined cost is superior to the baseline. Giraffe builds on Zebra and likewise targets the VA setting. However, some of Giraffe's results and techniques apply to verifiable outsourcing more generally.

Giraffe has two high-level aspects. The first is a new probabilistic proof implementation built on a protocol that we call *T13* [66, §7]. As with all systems in the literature, T13 requires computations to be expressed as *arithmetic circuits*, or ACs (§2.1). T13 has three key advantages: (a) T13 is a variant of CMT [31, 42] (Zebra's base), and thus promises amenability to hardware implementation; (b) in the VA context, T13 can in principle pay for precomputation and break even, because it is geared to the data parallel model mentioned earlier: precomputation is proportional to one sub-computation, and amortizes over N sub-computations; and (c) T13 ought to permit breaking even for small N: CMT has low overhead compared to alternatives [72]. From this starting point, Giraffe makes the following contributions (§3):

- Giraffe improves T13. Most significantly, Giraffe makes the prover asymptotically time-optimal: for sufficiently large N, the prover's work is now only a multiple ($\approx 10\times$) of executing the AC (§3.1). This can save an order of magnitude or more for any implementation of T13 in any context, and is of independent interest.
- Giraffe develops a *design template* that automatically instantiates physically realizable, efficient, high-throughput ASIC designs for the prover and verifier, based on a designer's parameters. Consistent with our search for applicable regimes, the parameter ranges are wide (small and large N, different hardware substrates, etc.), which creates a challenge: the optimal architectures are diverse. For example, large ACs (large sub-computations and/or large N) must iteratively reuse the underlying hardware whereas small ACs call for high parallelism. Giraffe meets this challenge with a small set of custom hardware structures that, when applied to the data flows in T13, run efficiently in serial execution *and* parallelize easily.

The second aspect of Giraffe is motivated by our search for applicable regimes. In existing systems, protocol overhead limits the maximum size of a computation that can be outsourced. Worse, outsourcing really makes sense only if the computation is naturally expressed as an AC; otherwise, the asymptotic savings do not apply until program sizes are well beyond the aforementioned maximum. While these systems differ in the particulars, their restrictions are qualitatively similar—and there has been no fundamental progress on the expressivity issue over the last six years. As a consequence, it seems imperative to adapt to this situation. Two possible approaches are to handle these constraints by outsourcing amenable pieces of a given computation and to apply program transformations to increase the range of suitable computations.

We study techniques for each of these approaches (§4). Giraffe employs *slicing*, which takes as input a cost model and a program, automatically identifies amenable subregions of the program, and generates glue code to sew the outsourced pieces into the rest of the program. Slicing is a very general technique that can work with all probabilistic proof implementations. Giraffe also uses *squashing*, which transforms sequential ACs into parallel ACs, and adjusts the verifier to link these computations; this is relevant to CMT and T13, which require parallel ACs.

Giraffe's implementation (§5) applies the above transformations to C programs to produce a high-level representation suitable for T13. Another compiler takes this representation and several design parameters (Fig. 14, Apdx. C) and automatically generates a hardware design, built in SystemVerilog, that can be used for cycle-accurate simulation or synthesized (that is, compiled to a chip).

We evaluate using detailed simulation and modeling of these generated hardware designs. Accounting for all costs (prover, precomputation, verifier), Giraffe saves compared to native execution across a wide range of computation sizes and hardware substrates (§6.2). In our example applications (§8), Giraffe breaks even on operating costs for $N \approx 30$ parallel sub-computations. Compared to prior work in the VA setting, Giraffe scales to $500\times$ larger computation sizes, holding other parameters constant (§8.1). Finally, we demonstrate slicing against an image-matching application that Giraffe could not otherwise handle (§8.2).

Ultimately, Giraffe's significance rests in adopting the most stringent cost regime considered in the verifiable outsourcing literature and (to our knowledge) being the only system that *can* profitably outsource under this accounting. Nevertheless, Giraffe has plenty of limitations: breaking even requires data parallel computations, the absolute cost of verifiability is still very high (as in every system in the research area), the applicability is still far narrower than we would like, and the program transformation techniques have taken only a small first step.

2 Background

2.1 Probabilistic proofs for verifiability

The description below is intended to give necessary terminology; it does not cover all variations in the literature.

Systems for verifiable outsourcing enable the following. A *verifier* $\mathcal{V}$ specifies a computation Ψ (often expressed in a high-level language) to a *prover* $\mathcal{P}$. $\mathcal{V}$ determines input x; $\mathcal{P}$ returns y, which is purportedly $\Psi(x)$. A protocol between $\mathcal{V}$ and $\mathcal{P}$ allows $\mathcal{V}$ to check whether $y = \Psi(x)$ but without executing Ψ. There are few (and sometimes no) assumptions about the scope of $\mathcal{P}$'s misbehavior.

These systems typically have a *front-end* and a *back-end*. The interface between them is an *arithmetic circuit* (AC). In an AC, the domain is a finite field $\mathbb{F}$, usually $\mathbb{F}_p$ (the integers mod a prime p); "gates" are field operations (add or multiply), and "wires" are field elements.

The front-end transforms Ψ from its original expression to an AC, denoted $\mathcal{C}$; this step often uses a compiler [27, 28, 32, 38, 56, 62, 64, 69, 71], though is sometimes done manually [18, 31, 66]. The back-end is a probabilistic proof protocol, targeting the assertion "$y = \mathcal{C}(x)$"; this step incorporates tools from complexity theory and sometimes cryptography.

2.2 Starting point for Giraffe's back-end: T13

Giraffe's back-end builds on a line of interactive proofs [12, 42, 43, 49, 65]: GKR [42], as refined and implemented by CMT [31], Allspice [69], Thaler [66], and Zebra [70]. Our description below sometimes borrows from [69, 70].

In these works, the AC $\mathcal{C}$ must be *layered*: the gates are partitioned, and there are wires only between adjacent partitions (layers). Giraffe's specific base is *T13* [66, §7], with an optimization [67]. T13 requires data parallelism: $\mathcal{C}$ must have N identical sub-circuit copies, each with its own inputs and outputs (x and y now denote the aggregate inputs and outputs). We call each copy a *sub-AC*. Each sub-AC has d layers. For simplicity, we assume that every sub-AC layer has the same width, G (this implies that $|x| = |y| = N \cdot G$). The properties of T13 are given below; probabilities are over $\mathcal{V}$'s random choices (Apdx. A justifies these properties, by proof and reference to the literature):

- **Completeness**. If $y = \mathcal{C}(x)$, and if $\mathcal{P}$ follows the protocol, then $\Pr\{\mathcal{V} \text{ accepts}\} = 1$.
- **Soundness**. If $y \neq \mathcal{C}(x)$, then $\Pr\{\mathcal{V} \text{ accepts}\} < \varepsilon$, where $\varepsilon = (\lceil \log |y| \rceil + 6d \log(G \cdot N))/|\mathbb{F}|$. This holds unconditionally (no assumptions about $\mathcal{P}$). Typically, $|\mathbb{F}|$ is astronomical, making this error probability tiny.
- **Verifier's running time**. $\mathcal{V}$ requires precomputation that is proportional to executing one sub-AC: $O(d \cdot G)$. Then, to validate all inputs and outputs, $\mathcal{V}$ incurs cost $O(d \cdot \log(N \cdot G) + |x| + |y|)$ (which, under our "same-size-layer assumption", is $O(d \cdot \log(N \cdot G) + N \cdot G)$). Notice that the total cost to verify $\mathcal{C}$, $O(d \cdot G + d \cdot \log N + N \cdot G)$, is less than the cost to execute $\mathcal{C}$ directly, which is $O(d \cdot G \cdot N)$.
- **Prover's running time**. $\mathcal{P}$'s running time is $O(d \cdot G \cdot N \cdot \log G)$; we improve this later (§3.1).

Details. Within a layer of $\mathcal{C}$, each gate is labeled with a pair $(n, g) \in \{0,1\}^{b_N} \times \{0,1\}^{b_G}$, where $b_N \triangleq \log N$ and $b_G \triangleq \log G$. (We assume for simplicity that N and G are powers of 2.) We also view labels numerically, as elements in $\{0, \ldots, N-1\} \times \{0, \ldots, G-1\}$. In either case, n (a gate label's upper bits) selects a sub-AC, and g (a gate label's lower bits) indexes a gate within the sub-AC.

Each layer i has an *evaluator function* $V_i \colon \{0,1\}^{b_N} \times \{0,1\}^{b_G} \to \mathbb{F}$ that maps a gate's label to the output of that gate;[3] implicitly, V_i depends on the input x. By convention, the layers are numbered in reverse execution order. Thus, V_0 refers to the output layer, and V_d refers to the inputs. For example, $V_0(n, j_1)$ is the correct j_1th output in sub-AC n; likewise, $V_d(n, j_2)$ is the j_2th input in sub-AC n.

Notice that $\mathcal{V}$ wants to be convinced that y, the purported outputs, matches the correct outputs, as given by V_0. However, $\mathcal{V}$ cannot check this directly: evaluating V_0 would require re-executing $\mathcal{C}$. Instead, $\mathcal{P}$ combines all $V_0(\cdot)$ values into a digest. Then, the protocol reduces this digest to another digest, this one (purportedly) corresponding to all of the values $V_1(\cdot)$. The protocol proceeds in this fashion, layer by layer, until $\mathcal{V}$ is left with a purported digest of the input x, which $\mathcal{V}$ can then check itself.

Instantiating the preceding sketch requires some machinery. A key element is the *sum-check protocol* [49], which we will return to later (§3.1). For now, let $P \colon \mathbb{F}^m \to \mathbb{F}$ be an m-variate polynomial. In a sum-check *invocation*, $\mathcal{P}$ interactively establishes for $\mathcal{V}$ a claim about the sum of the evaluations of P over the Boolean hypercube $\{0,1\}^m$; the number of protocol rounds is m.

Another key element is *extensions*. Technically, an extension $\tilde{f}$ of a function f is a polynomial that is defined over a domain that encloses the domain of f and equals f at all points where f is defined. Informally, one can think of $\tilde{f}$ as encoding the function table of f. In this paper, extensions will always be *multilinear extensions*: the polynomial has degree at most one in each of its variables. We notate multilinear extensions with tildes.

Based on the earlier sketch, we are motivated to express $\tilde{V}_{i-1}$ in terms of $\tilde{V}_i$. To that end, we define several predicates. The functions add$(\cdot)$ and mult$(\cdot)$ are *wiring predicates*; they have signatures $\{0,1\}^{3b_G} \to \{0,1\}$, and implicitly describe the structure of a sub-AC. add$_i(g, h_0, h_1)$ returns 1 iff (a) within a sub-circuit, gate g at layer $i-1$ is an add gate and (b) the left and right inputs of g are, respec-

[3] This definition of V_i transposes the domain relative to [66, §7].

tively, h_0 and h_1 at layer i. mult_i is defined analogously. Note that these predicates ignore the "top bits" (the n component) because all sub-ACs are identical. We also define the *equality predicate* $\text{eq}: \{0,1\}^{2b_N} \to \{0,1\}$ with $\text{eq}(a,b) = 1$ iff a equals b. Notice that these predicates admit extensions: $\tilde{\text{add}}, \tilde{\text{mult}}: \mathbb{F}^{3b_G} \to \mathbb{F}$ and $\tilde{\text{eq}}: \mathbb{F}^{2b_N} \to \mathbb{F}$. (We give explicit expressions in Apdx. A.)

We can now express $\tilde{V}_{i-1}$ in terms of a polynomial $P_{q,i}$:

$$P_{q,i}(r_0, r_1, r') \triangleq \tilde{\text{eq}}(q', r') \cdot \left[\tilde{\text{add}}_i(q, r_0, r_1) \cdot \left(\tilde{V}_i(r', r_0) + \tilde{V}_i(r', r_1)\right) + \tilde{\text{mult}}_i(q, r_0, r_1) \cdot \tilde{V}_i(r', r_0) \cdot \tilde{V}_i(r', r_1)\right]. \quad (1)$$

$$\tilde{V}_{i-1}(q', q) = \sum_{h_0, h_1 \in \{0,1\}^{b_G}} \sum_{n \in \{0,1\}^{b_N}} P_{q,i}(h_0, h_1, n). \quad (2)$$

The signatures are $P_{q,i}: \mathbb{F}^{2b_G + b_N} \to \mathbb{F}$ and $\tilde{V}_{i-1}, \tilde{V}_i: \mathbb{F}^{b_N} \times \mathbb{F}^{b_G} \to \mathbb{F}$. Equation (2) follows from an observation of [67] applied to a claim in [66, §7]. For intuition, notice that (i) $P_{q,i}$ is being summed only at points where its variables are 0-1, and (ii) at these points, if (q', q) is a gate label (rather than an arbitrary value in $\mathbb{F}^{b_N} \times \mathbb{F}^{b_G}$), then the extensions of the predicates take on 0-1 values and in particular eliminate all summands except the one that contains the inputs to the gate (q', q).

An excerpt of the protocol appears in Figure 1; the remainder appears in Appendix A. It begins with $\mathscr{V}$ wanting to be convinced that $\tilde{V}_0$ (which is the extension of the correct $\mathscr{C}(x)$) is the same polynomial as $\tilde{V}_y$ (which denotes the extension of the purported output y). $\mathscr{V}$ thus chooses a random point in both polynomials' domain, (q'_0, q_0), and wants to be convinced that $\tilde{V}_0(q'_0, q_0) = \tilde{V}_y(q'_0, q_0) \triangleq a_0$. Notice that (i) $\tilde{V}_0(q'_0, q_0)$ can be expressed as the sum over a Boolean hypercube of the polynomial $P_{q_0,1}$ (Equation (2)), and (ii) $P_{q_0,1}$ itself is expressed in terms of $\tilde{V}_1$ (Equation (1)). Using a sum-check invocation, the protocol exploits these facts to reduce $\tilde{V}_0(q'_0, q_0) = a_0$ to a claim: $\tilde{V}_1(q'_1, q_1) = a_1$. This continues layer by layer until $\mathscr{V}$ obtains the claim: $\tilde{V}_d(q'_d, q_d) = a_d$. $\mathscr{V}$ checks that assertion directly.

T13 incorporates one sum-check invocation—each of which is $2b_G + b_N$ rounds—for each polynomial $P_{q_0,1}, \ldots, P_{q_{d-1},d}$.

2.3 Verifiable ASICs

Giraffe's back-end works in the *Verifiable ASICs* (VA) setting [70]. Giraffe also borrows evaluation metrics and some design elements from [70]; we summarize below.

Consider some *principal* (a government, fabless semiconductor company, etc.) that wants high-assurance execution of a custom chip (known as an ASIC) [70, §1,§2.1]. The ASIC must be manufactured at a trustworthy foundry, for example one that is onshore. However, for many principals, high-assurance manufacture means an orders-of-magnitude sacrifice in price and performance, relative

```
1:  function VERIFY(ArithCircuit c, input x, output y)
2:      (q'_0, q_0) <-R F^{log N} x F^{log G}
3:      a_0 <- V~_y(q'_0, q_0) // V~_y is the multilin. ext. of the output y
4:      SendToProver(q'_0, q_0)
5:      d <- c.depth
6:
7:      for i = 1, ..., d do
8:          // Reduce V~_{i-1}(q'_{i-1}, q_{i-1}) ?= a_{i-1} to P_{q,i}(r_0, r_1, r') ?= e
9:          (e, r', r_0, r_1) <- SUMCHECKV(i, a_{i-1})
10:
11:         // Below, P describes a univariate polynomial H(t),
12:         // of degree log G, claimed to be V~_i(r', (r_1 - r_0)t + r_0)
13:         H <- ReceiveFromProver() // see Line 47 of Figure 13
14:         v_0 <- H(0)
15:         v_1 <- H(1)
16:
17:         // Reduce P_{q,i}(r_0, r_1, r') ?= e to two questions:
18:         // V~_i(r', r_0) ?= v_0 and V~_i(r', r_1) ?= v_1
19:
20:         if e != eq~(q'_{i-1}, r') . [add~_i(q_{i-1}, r_0, r_1) . (v_0 + v_1)
21:                                      + mult~_i(q_{i-1}, r_0, r_1) . v_0 . v_1] then
22:             return reject
23:
24:         // Reduce the two v_0, v_1 questions to V~_i(q'_i, q_i) ?= a_i
25:         tau_i <-R F
26:         a_i <- H(tau_i)
27:         (q'_i, q_i) <- (r', (r_1 - r_0) . tau_i + r_0)
28:
29:         SendToProver(tau_i)
30:
31:     // V~_d(.) is the multilinear extension of the input x
32:     if V~_d(q'_d, q_d) = a_d then
33:         return accept
34:     return reject
```

FIGURE 1—$\mathscr{V}$'s side of T13 [66, §7], with an optimization [67]. $\mathscr{V}$'s side of the sum-check protocol and $\mathscr{P}$'s work are described in Appendix A, Figures 9 and 13.

to an advanced but untrusted foundry. This owes to the economics and scaling behavior of semiconductor technology. In the VA setup, one manufactures a prover in a state-of-the-art but untrusted foundry (we refer to the manufacturing process and hardware substrate as the *untrusted technology node*) and a verifier in a trusted foundry (the *trusted technology node*). A trusted integrator combines the two ASICs. This arrangement makes sense if their combined cost is cheaper than the *native baseline*: an ASIC manufactured in the trusted technology node.

VA is instantiated in a system called Zebra, which implements an optimized variant of CMT [31, 67, 69]. Zebra is evaluated with two metrics [70, §2.3]. The first is *energy* (E, in joules/run), which is a proxy for operating cost. Energy tracks asymptotic (serial) running time: it captures the number of operations and the efficiency of their implementation. The second is *area/throughput* (A/T, in

mm^2/(ops/sec)). Area is a proxy for manufacturing cost; normalizing by throughput reflects cost at a given performance level.

Furthermore, Zebra is designed to respect two physical constraints. The first is a maximum area, to reflect manufacturability (larger chips have more frequent defects and hence lower yields). The second is a maximum power dissipation, to limit heat. The first constraint limits A (and thus the hardware design space) and the second limits the product of E and T.

Zebra's prover architecture consists of a collection of pipelined *sub-provers*, each one doing the execution and proving work for one layer of an AC [70, §3.1–3.2]. Within a sub-prover, there is dedicated hardware for each AC gate in a layer. Zebra's verifier is also organized into layers [70, §3.5]. Giraffe incorporates this overall picture, including some integration details [70, §4]. However, Giraffe requires a different architecture, as we explain next.

3 Protocol and hardware design

Three goals drive Giraffe's hardware back-end:

G1: Scale to large N without sacrificing G. $\mathcal{V}$'s precomputation scales with the size of one sub-AC (§2.2); it needs to amortize this over multiple N. Further, we have an interest in handling large computations (sub-ACs and ACs). This implies that Giraffe's design must reuse underlying hardware modules: for large G and N, requiring a number of modules proportional to $N \cdot G$ is too costly. Zebra's design is not suitable, since it requires logic proportional to the amount of work in an AC layer [70, Fig. 5].

G2: Be efficient. In this context, good efficiency implies lower cross-over points on the metrics of merit (§2.3). This in turn means custom hardware, which is expected in ASIC designs but, for us, is in tension with the next goal.

G3: Produce designs automatically. Ideally, the goal is to produce a compiler that takes as input a high-level description of the computation along with physical parameters (technology nodes, chip area, etc.) and produces synthesizable hardware (§5). This goes beyond convenience: a goal of this work is to understand where (in terms of computations, technology nodes, G, N, etc.) an abstract algorithm (T13) applies; we need to be able to optimize hardware for these parameters. This is challenging because, over the parameter range that we target, different hardware designs make sense. For example, if N and G are small, iteratively reusing hardware might not consume all available chip area; one would prefer to spend this area to gain parallelism and thus increase throughput.

Giraffe manages this by developing a *design template* that, when instantiated with different parameters, produces an optimized hardware design. The template's "primitives" are custom hardware structures that enable efficient reuse (serial execution) when there are few of them, but can be automatically parallelized. The designer chooses template parameters (Fig. 14, Apdx. C; §6.2), and design generation is automatic.

In the rest of the section, we modify T13 to obtain an asymptotic improvement in $\mathcal{P}$'s work (§3.1); this contributes to Giraffe's scalability, and is of independent interest. Second, we describe aspects of the hardware design template for $\mathcal{P}$ (§3.2). Finally, we do the same for $\mathcal{V}$, and also describe optimizations that help offset the cost of precomputation (§3.3). These optimizations are modest, but because $\mathcal{V}$'s costs dominate, they have a direct effect on the bottom-line numbers.

Notation. $[a,b]$ denotes $\{a, a+1, \ldots, b\}$. For a vector u, $u[\ell]$ denotes the ℓth entry, indexed from 1; $u[\ell_1..\ell_2]$ denotes the sub-vector between indices ℓ_1 and ℓ_2, inclusive. Define $\chi_0, \chi_1 : \mathbb{F} \to \mathbb{F}$ as $\chi_1(w) = w$, $\chi_0(w) = 1 - w$. Similarly, if $s \in \{0,1\}^\gamma$ and $u \in \mathbb{F}^\gamma$, $\chi_s(u) \triangleq \prod_{\ell=1}^{\gamma} \chi_{s[\ell]}(u[\ell])$. Notice that when u comprises 0-1 values, $\chi_s(u)$ returns 1 if $u = s$ and 0 otherwise.

3.1 Making $\mathcal{P}$ time-optimal

This section describes an algorithmic refinement that, by restructuring the application of the sum-check protocol, slashes $\mathcal{P}$'s overhead. Specifically, $\mathcal{P}$'s running time drops from $O(d \cdot N \cdot G \cdot \log G)$ to $O(d \cdot (N \cdot G + G \cdot \log G))$. If $N \gg \log G$, $\mathcal{P}$'s new running time is linear in the number of total gates in the AC—that is, the prover has no asymptotic overhead! Prior work [66, §5] achieved time-optimality in special cases (if the AC's structure met an ad hoc and restrictive condition); the present refinement applies in general, whenever there are repeated sub-ACs.

The $O(\log G)$ reduction translates to concrete double digit factors. For example, software provers in this research area [31, 66, 68, 69] typically run with G at least 2^{18}; thus, a software T13 prover's running time improves by at least $18\times$. For a hardware prover, the A/T metric improves by approximately $\log G$, as computation is the main source of area cost (Apdx. C, [70, Fig. 6 and 7]). The gain is less pronounced for the E metric: storage and communication are large energy consumers but are unaffected by the refinement (Apdx. C).

Before describing the refinement, we give some background on sum-check protocols; for details, see [8, §8.3; 42, §2.5; 49; 65]. Consider a polynomial P in m variables and a claim that $\sum_{(t_1,\ldots,t_m)\in\{0,1\}^m} P(t_1,\ldots,t_m) = L$. In round j of the sum-check protocol, $\mathcal{P}$ must describe to $\mathcal{V}$ a degree-α *univariate* polynomial $F_j(t^*)$, where α depends on P and j:

$$F_j(t^*) = \sum_{(t_{j+1},\ldots,t_m)\in\{0,1\}^{m-j}} P(\rho_1,\ldots,\rho_{j-1},t^*,t_{j+1},\ldots,t_m).$$

To discharge this obligation, $\mathcal{P}$ computes evaluations $F_j(k)$, for $\alpha+1$ different values of k. Then, at the end of

round j, $\mathscr{V}$ sends ρ_j, for use in the next round. Notice the abstract pattern: in every round j, $\mathscr{P}$ computes $\alpha+1$ sums over a Boolean hypercube of dimension $m-j$. The number of hypercube vertices shrinks as j increases, because variables that were previously summed become set, or *bound*, to a ρ_j.

Let us map this picture to our context. There is one sum-check run for each layer $i \in [1,d]$; P is the per-layer polynomial $P_{q,i}$ defined in Equation (1); $m = 2b_G + b_N$; the ρ_j are aliases for the components of r_0, r_1, r'; likewise, the t_j alias the components of h_0, h_1, n. Also, α is 2 or 3; this follows from Equation (1), recalling that each multilinear extension ($\widetilde{\text{eq}}$, $\widetilde{\text{add}}$, etc.) by definition has degree one in its variables.

There are now two interrelated questions: In what order should the variables be bound? How does $\mathscr{P}$ compute the $\alpha+1$ sums per round? In T13, the order is h_0, h_1, n, as in Equation (2). This enables $\mathscr{P}$ to compute the needed sums in time $O(N \cdot G \cdot \log G)$ per-layer [66, §7]. $\mathscr{P}$'s total running time is thus $O(d \cdot N \cdot G \cdot \log G)$.

Giraffe's refinement changes the order in which variables are bound, and exploits that order to simplify $\mathscr{P}$'s work. Giraffe's order is n, h_0, h_1. From here on, we write $P_{q,i}(h_0, h_1, n)$ as $P^*_{q,i}(n, h_0, h_1)$; $P_{q,i} \equiv P^*_{q,i}$ except for argument order. Below, we describe the structure of $\mathscr{P}$'s per-round obligations, fixing a layer i. This serves as background for the hardware design (§3.2) and as a sketch of the argument for the claimed running time. A proof, theorem statement, and pseudocode are in Appendix B.

The rounds decompose into two *phases*. Phase 1 is rounds $j \in [1, b_N]$. Observe that in this phase, $\mathscr{P}$'s sums seemingly have the form: $F_j(k) = \sum_{n[j+1..b_N]} \sum_{h_0,h_1} P^*_{q,i}(r'[1..j-1], k, n[j+1..b_N], h_0, h_1)$, where the outer sum is over all $n[j+1..b_N] \in \{0,1\}^{b_N - j}$. However, many (h_0, h_1) combinations cause $P^*_{q,i}(\ldots, h_0, h_1)$ to evaluate to 0.[4] As a result, there is a more convenient form for the inner sum. Define $S_{\text{all},i} \subseteq \{0,1\}^{3b_G}$ as all layer-$(i-1)$ gates with their layer-i neighbors, and OP_g as "+" if g is an addition gate and "$\cdot$" if g is a multiplication gate. Then $F_j(k)$ can be written as:

$$F_j(k) = \sum_{n[j+1..b_N]} \sum_{(g,g_L,g_R) \in S_{\text{all},i}} \text{termP1}_{j,n,k} \cdot \text{termP2}_g \cdot \text{OP}_g(\text{termL}_{j,n,g_L,k}, \text{termR}_{j,n,g_R,k}), \quad (3)$$

where termP1 depends on j, n, k; termP2 depends on g, and so forth; these also depend on values of ρ from prior rounds and prior layers. Section 3.2 makes some of these terms explicit (Apdx. B fully specifies).

Phase 2 is the remaining $2b_G$ rounds. Here, there is only a single sum, over increasingly bound components of h_0, h_1. As with phase 1, it is convenient to express the sum "gatewise". Specifically, for rounds $j \in [b_N+1, b_N+2b_G]$, one can write $F_j(k) = \sum_{(g,g_L,g_R) \in S_{\text{all},i}} \text{termP}_{j,g,k} \cdot \text{OP}_g(\text{termL}_{j,g_L,k}, \text{termR}_{j,g_R,k})$.

In both phases, $\mathscr{P}$ can compute each sum over $S_{\text{all},i}$ with $O(G)$ work. Thus, per-layer, the running time for phase 1 is $O(G \cdot N/2) + O(G \cdot N/4) + \cdots + O(G) = O(G \cdot N)$, and for phase 2 it is $O(G \cdot \log G)$, yielding the earlier claim of $O(d \cdot (N \cdot G + G \cdot \log G))$.

```
1:  // initialize W: G vectors of N values
2:  for h = 0,...,G-1 and σ = 0,...,N-1 do
3:      W[h][σ] ← V_i(σ, h)
4:
5:  function EvalTermLR(Array-of-vectors W)
6:      for j = 1,...,b_N do
7:          look up all termL, termR in W (see text)
8:
9:          r'[j] ← Receive from 𝒱      // see Figure 12, line 19
10:
11:         for h = 0,...,G-1 do
12:             Collapse(W[h], N/2^{j-1}, r'[j])
13:
14: function Collapse(Array A, size len, r ∈ 𝔽)
15:     for σ = 0,...,len/2-1 do
16:         A[σ] ← (1-r)·A[2σ] + r·A[2σ+1]
```

FIGURE 2—EvalTermLR: a dynamic programming algorithm for computing termL, termR for all rounds j. EvalTermLR adapts a prior technique [66, §5.4; 70, §3.3] [1–3].

3.2 Design of $\mathscr{P}$

Consider $\mathscr{P}$'s obligations in layer i, summarized at the end of the previous section. Notice that $\mathscr{P}$'s phase-2 obligations are independent of N. This is a consequence of Section 3.1; there is no such independence in the original variable order [66, §7]. In fact, $\mathscr{P}$'s phase-2 obligations are almost isomorphic to those of the Zebra prover; accordingly, Giraffe incorporates that design as a module.

$\mathscr{P}$'s principal work is in phase 1. Within that phase, the heaviest work item is computing termL, termR in each round. The rest of this section describes the obligation, the algorithm by which $\mathscr{P}$ discharges it, and the hardware design that computes the algorithm. $\mathscr{P}$'s other obligations (computing $\text{termP1}_{j,n,k}$, etc.) and algorithms for discharging them are described in Appendix B.

Algorithm for computing termL,termR. Fixing a layer i, in round j, termL and termR are:

$$\begin{aligned} \text{termL}_{j,n,g_L,k} &\triangleq \tilde{V}_i\left(r'[1..j-1], k, n[j+1..b_N], g_L\right) \\ \text{termR}_{j,n,g_R,k} &\triangleq \tilde{V}_i\left(r'[1..j-1], k, n[j+1..b_N], g_R\right) \end{aligned} \quad (4)$$

Notice that for each k, Equation (4) refers to $G \cdot N/2^j$ values of $\tilde{V}(\cdot)$.

Figure 2 depicts an algorithm, EvalTermLR, that computes these values in time $O(G \cdot N/2^j)$ for round j, by

[4] In particular, if there is no gate at layer $i-1$ whose left and right inputs are h_0 and h_1, then $P^*_{q,i}(\ldots, h_0, h_1) = 0$. This is a consequence of Equation (1) in §2.2, and Equations (7) and (8) in Appendix A.

RWSR specification

- Power-of-two storage locations, K
- Only locations 0 and 1 can be read
- The only write operation is $\overset{s}{\leftarrow}$. It is specified below. Informally, it updates one location, and causes all the "even" locations to behave like a distinct shift register (location 6 shifts to 4, etc.), and likewise with all of the "odd" locations.

```
1: operator RWSR[a] ←s v is
2:     // Note that all updates happen simultaneously
3:     RWSR[a] ← v
4:     for ℓ < K, ℓ ≠ a do
5:         RWSR[ℓ] ← RWSR[ℓ + 2]
6:
7: function RWSRCollapse(RWSR R, size len, r ∈ F)
8:     for σ = 0, ..., len/2 − 1 do
9:         R[len − 2 − σ] ←s (1 − r) · R[0] + r · R[1]
```

FIGURE 3—RWSR specification (§3.2) and RWSR-based Collapse implementation (Fig. 2).

adapting a prior technique [66, §5.4; 70, §3.3] (see also [1–3]). EvalTermLR is oriented around a recurrence. Let h be a bottom-bit gate label at layer i. Then for all $\sigma \in \{0,1\}^{b_N - j}$, the following holds (derived in Apdx. B.1):

$$\tilde{V}_i\left(r'[1..j], \sigma, h\right) = \left(1 - r'[j]\right) \cdot \tilde{V}_i\left(r'[1..j-1], 0, \sigma, h\right) + r'[j] \cdot \tilde{V}_i\left(r'[1..j-1], 1, \sigma, h\right). \quad (5)$$

EvalTermLR relies on a two-dimensional array W, and maintains the following invariant, justified shortly: *at the beginning of every round j, $W[h][\sigma]$ stores $\tilde{V}_i(r'[1..j-1], \sigma, h)$, for $h \in [0, G-1]$ and $\sigma \in [0, N/2^{j-1}-1]$.*

Given this invariant, $\mathscr{P}$ obtains all of the termL, termR values from W (in line 7), as follows. We focus on termL. Write $n[j+1..b_N]$ as n_{j+1}. Then, for $k = \{0,1\}$, $\text{termL}_{j,n,g_L,k}$ is $W[g_L][k + 2 \cdot n_{j+1}]$; this follows from Equation (4) plus the invariant. Meanwhile, for $k = -1$, $\text{termL}_{j,n,g_L,-1} = 2 \cdot \text{termL}_{j,n,g_L,0} + (-1) \cdot \text{termL}_{j,n,g_L,1}$. This follows from Equations (4) and (5); $k = 2$ is similar. termR is the same, except g_R replaces g_L. The total time cost is $O(G \cdot N/2^j)$ in round j: Collapse performs $(N/2^{j-1})/2$ iterations, and there are G calls to Collapse.

The invariant holds for $j = 1$ because $\tilde{V}_i(r'[1..j-1], \sigma, h) = \tilde{V}_i(\sigma, h) = V_i(\sigma, h)$, which initializes $W[h][\sigma]$ (line 3); the latter equality holds because functions equal their extensions when evaluated on bit vectors. Now, at the end of j, line 16 applies Equation (5) to all $\sigma \in [0, N/2^j - 1]$, thereby setting $W[h][\sigma]$ to $\tilde{V}_i(r[1..j], \sigma, h)$. This is the required invariant at the start of round $j+1$.

Computing EvalTermLR in hardware. To produce a design template for $\mathscr{P}$ consistent with Giraffe's goals, we must answer three questions. First, what breakdown of $\mathscr{P}$'s work makes sense: which portions are parallelized, and what hardware is iteratively reused in a round (G1)? Second, for iterative parts of the computation, how does $\mathscr{P}$ load operands and store results (G2)? Finally, how can this design be adapted to a range of parameters (G3)?

A convenient top-level breakdown is already implied by the prior formulation of W: since Collapse operates on each $W[h]$ vector independently, it is natural to parallelize work across these vectors. Giraffe allocates separate storage structures and logic implementing Collapse for each $W[h]$ vector (and, of course, reuses this hardware from round to round for each vector). We therefore focus on the design of one of these modules.

To answer the second question, we first consider two straw men. The first is to imitate a software design: instantiate one module for field arithmetic and a RAM to store the $W[h]$ vector, then iterate through the σ loop sequentially, loading needed values, computing over them, and storing the results. In practice, however, VLSI designs often avoid RAM, for several reasons: generality has a price (e.g., address decoding imposes overheads in area and energy), RAM often creates a throughput bottleneck, and RAM is a frequent cause of manufacturability issues.

The second straw man is essentially the opposite: instantiate a bank of registers to hold values in $W[h]$, along with two field multipliers and one adder per pair of adjacent registers, then create a wiring pattern such that the adder for registers 2σ and $2\sigma+1$ connects to the input of register σ. This arrangement computes the entire σ loop in parallel. This is similar to prior work [70, §3.3], but in Giraffe $O(NG)$ multipliers is extremely expensive when N and G are large. It is also inflexible: in this design, the number of multipliers is fixed after selecting N and G.

Giraffe's solution is a hybrid of these approaches; we first explain a serial version, then describe how to parallelize. Giraffe instantiates two multipliers and one adder that together compute one step of the σ loop. The remaining challenge is to get operands to the multipliers and store the result from the adder. Giraffe does so using a custom hardware structure that is tailored to the access pattern of the $W[h]$ arrays: for each $A = W[h]$, read two values, write one value, read two values, and so on. Giraffe uses *RWSRs*, ("random-write shift registers"), one for each $W[h]$. Figure 3 specifies the RWSR and shows its use for Collapse.

Compared to a standard shift register (which is inexpensive to implement), an RWSR pays a small overhead to connect every storage location to the input source v (Fig. 3). But RWSRs are significantly more efficient than RAMs, in part because of the restriction that only two locations may be read. And although the $\overset{s}{\leftarrow}$ operation allows writes to arbitrary locations (which might require address decoding in the general case), RWSRCollapse makes it possible to optimize away most address logic because σ takes a predictable sequence of values.

The remaining question is how this design can be efficiently and *automatically* parallelized. Notice that the

loop over σ is serialized (because RWSRs allow only one write at a time); but what if a designer has extra chip area and is willing to use four multipliers for $W[h]$ instead of two? In other words, how can Giraffe's design template automatically improve RWSRCollapse's throughput by using more chip area?

To demonstrate the approach, we refer to the pseudocode of Figure 2. First, split each $W[h]$ array into two arrays, $W1[h]$ and $W2[h]$. In place of the Collapse invocation (line 12), run two parallel invocations on $W1[h]$ and $W2[h]$, each of half the length. Notice that each array has increasing "empty" space as the rounds go on. In round j, the "live values" are the first $N/2^j$ elements in each of $W1[h]$ and $W2[h]$; regard $W[h]$ as their concatenation.

To see why this gives the correct result, notice that each Collapse invocation combines neighboring values of its input array. We can thus regard the values of $W[h]$ as the leaves of a binary tree, and Collapse as reducing the height of the tree by one, combining leaves into their parents. In this view, $W1[h]$ and $W2[h]$ represent the left and right subtrees corresponding to $W[h]$. As a result, in round $j = b_N$, $W1[h]$ and $W2[h]$ each have one value; to obtain the final value of the Collapse operation, compute $(1-r) \cdot W1[h][0] + r \cdot W2[h][0]$.

To implement this picture in hardware, Giraffe instantiates two RWSRs, each of half the size. For even more parallelism, observe that each RWSR corresponds to a subtree of the full computation, and thus its work can be recursively split into two even smaller RWSRs, each handling a correspondingly smaller subtree. Because of this structure, different choices of parallelism do not require the designer to do any additional design work (§5).

3.3 Scaling and optimizing $\mathscr{V}$

In this section, we explain how $\mathscr{V}$ meets the starting design goals of scalability, efficiency, and automation. We do so by walking through three main costs for $\mathscr{V}$, and how Giraffe handles them. Some of the optimizations apply to any CMT-based back-end [31, 66, 68–70].

Multilinear extensions of I/O. $\mathscr{V}$'s principal bottleneck is computing the multilinear extension of its input x and output y (Figure 1, lines 3 and 32). Recall (§2.2) that $|x| = |y| = N \cdot G$; $\mathscr{V}$'s computation has at least this cost. When N and G are large, this is expensive and must be broken into parallel and serial portions. We show below that this work has a similar form to $\mathscr{P}$'s (termL, termR; §3.2). This insight lets $\mathscr{V}$ reuse $\mathscr{P}$'s hardware design.

Consider the input x and $\tilde{V}_d$ (y and $\tilde{V}_y$ are similar). $\mathscr{V}$ must compute $\tilde{V}_d(q'_d, q_d)$. For $\sigma \in [0, N \cdot G - 1]$, $\tilde{V}_d(\sigma) = V_d(\sigma)$, the σth component of the input (§2.2). For $\sigma \in \{0,1\}^{b_N + b_G - \ell}$, we have

$$\tilde{V}_d(r[1..\ell], \sigma) = (1 - r[\ell]) \cdot \tilde{V}_d(r[1..\ell-1], 0, \sigma) + r[\ell] \cdot \tilde{V}_d(r[1..\ell-1], 1, \sigma).$$

This form is very close to Equation (5); the derivation is similar (Apdx. B.1). It follows that $\mathscr{V}$ can use $\mathscr{P}$'s EvalTermLR to evaluate $\tilde{V}_d(q'_d, q_d)$: $\mathscr{V}$ initializes an array A, setting $A[\sigma]$ to the σth input value, for $\sigma \in [0, N \cdot G - 1]$ (cf. line 3, Fig. 2). $\mathscr{V}$ then invokes Collapse inside a non-blocking loop, in each iteration setting r to the next element of (q'_d, q_d). At the end, $A[0]$ holds the result.

This approach applies to related systems, and improves on their constant factors. Allspice's approach to this computation has leading constant 4 [69, §5.1]. Zebra [70] reduces the constant to 3 using a custom hardware structure; this does not meet Giraffe's goal of producing designs automatically for a range of parameters. Giraffe's approach reduces the leading constant to 2. To see how, note that the initial size of A is $N \cdot G$. When $j = 1$, Collapse costs $N \cdot G$ multiplications; in each successive invocation, the number of multiplications is reduced by half. Summing gives $2 \cdot N \cdot G - 1$ multiplications. Although the reduction appears modest, in practice this computation dominates $\mathscr{V}$'s costs and the improvement is thus significant.

Polynomial evaluation. The protocol requires $\mathscr{V}$ to evaluate polynomials (specified by $\mathscr{P}$) at randomly chosen points (specified by $\mathscr{V}$). This occurs after the sum-check invocation (Fig. 1, line 26) and in each round of the sum-check protocol (Apdx. B; Fig. 9, line 21). Our description here focuses on the former: the degree-b_G polynomial H, evaluated at τ. Giraffe applies the same technique to the latter, namely computing $F(r_j)$, but those polynomials are degree-2 or 3, and thus the savings are less pronounced.

In the baseline approach [31, 66, 69, 70] to computing $H(\tau)$, $\mathscr{P}$ sends *evaluations* (meaning $H(0), \ldots, H(b_G)$), and $\mathscr{V}$ uses Lagrange interpolation. (Lagrange interpolation expresses $H(\tau)$ as $\sum_{j=0}^{b_G} H(j) \cdot f_j(\tau)$; the $\{f_j(\cdot)\}$ are basis polynomials.) But interpolation costs $O(b_G^2)$ [46] for each polynomial (one per layer), making it $O(d \log^2 G)$ overall. Prior work [69, 70] cut this to $O(d \log G)$, by precomputing $\{f_j(\tau)\}$, and not charging for that.

Giraffe observes that the protocol works the same if $\mathscr{P}$ describes H in terms of its *coefficients*; this is because coefficients and evaluations are informationally equivalent. Thus, in Giraffe, $\mathscr{P}$ recovers the coefficients by interpolating the evaluations of H, incurring cost $O(d \log^2 G)$. $\mathscr{V}$ uses the coefficients to evaluate $H(\tau)$ via Horner's rule [46]. The cost to $\mathscr{V}$ is now $O(b_G)$ per layer, or $O(d \log G)$ in total, *without* relying on precomputation.

Summarizing, $\mathscr{V}$ shifts its burden to $\mathscr{P}$, and in return saves a factor $\log G$. This refinement is sensible if the same operation at $\mathscr{P}$ is substantially cheaper (by at least a $\log G$ factor) than at $\mathscr{V}$. This easily holds in the VA context. But it also holds in other contexts in which one would use a CMT-based back-end: if cycles at $\mathscr{P}$ were not substantially cheaper than at $\mathscr{V}$, the latter would not be outsourcing to the former in the first place.

Precomputation. $\mathscr{V}$ must compute $P^*_{q,i}(r', r_0, r_1,)$, given claimed $\tilde{V}_i(r', r_0)$ and $\tilde{V}_i(r', r_1)$: Figure 1, lines 20–21. The main costs are computing $\tilde{\text{add}}_i(q, r_0, r_1)$, $\tilde{\text{mult}}_i(q, r_0, r_1)$, and $\tilde{\text{eq}}(q', r')$. This costs $O(G)$ per layer [69], and hence $O(d \cdot G)$ overall. (Apdx. A describes the approach.) This is the "precomputation" in our context, and what was not charged in prior work in the VA setting [70, §4]. We note that this is not *pre*computation per se—it's done alongside the rest of the protocol—but we retain the vocabulary because of the cost profile: the work is proportional to executing one sub-AC, and is incurred once per sum-check invocation, thereby amortizing over all N sub-ACs.

4 Software design

Giraffe incorporates two program transformation techniques that broaden the scope of computations that are amenable to outsourcing:

- *Slicing* extends applicability to computations that are too large to be outsourced as a whole or contain parts that cannot be profitably outsourced.
- *Squashing* rearranges tall, narrow ACs to produce shallow and wide ones, reducing cost for CMT-based back-ends [31, 69, 70]. It can also turn some sequential computations into data-parallel ACs, making them amenable to outsourcing with a T13-derived back-end like the one presented in Section 3 (see also §2.2).

Slicing. One approach to handling larger outsourced computations is to outsource individual *slices*, with the verifier handling the non-outsourced pieces locally.

This approach works as follows: a compiler converts each sliced subcomputation into an AC. The program state just prior to entering the code in the slice yields the input of the corresponding AC. Likewise, the output of the AC provides the program state after execution of the slice. The threading of the inputs and outputs between the individual runs of the back-end, the intervening local computations, and the orchestration of the runs is all performed by glue code that the verifier executes locally. We refer to this code as the *manifest* of the sliced computation.

We now describe the slicing algorithm used in Giraffe. It takes as a parameter a cost model for the target back-end. Its input is a C program with the following restrictions (commonly imposed by the most efficient front-ends [32, 56, 71, 72]): loop bounds are statically computable, no recursive functions, and no function pointers.

The algorithm first inlines all function calls. It then considers for outsourcing consecutive subsequences of the top-level program statements. For each subsequence, the algorithm transforms the program statements into an AC and then uses the back-end cost model to determine the associated cost. Next, the algorithm uses a greedy heuristic to choose for outsourcing a set of non-overlapping subsequences that maximizes savings. Finally, the algorithm further analyzes parts of the program not in any of the outsourced subsequences. It adds atomic statements like assignments to the manifest for local execution. For non-atomic statements (for example, each branch of an if-else statement), the algorithm recursively invokes itself to identify additional outsourcing opportunities.

Giraffe uses the same back-end for all sliced subcomputations, but its approach generalizes to considering multiple back-ends simultaneously [38, 69].

Squashing. In back-ends descended from CMT (including T13), shallow and wide ACs are more cost-effective than ACs that are narrow but deep (§2.2). Squashing turns a deep but narrow computation (for example, a loop) into a shallow but wide one by laying chunks of the computation (e.g., the loop iterations) side by side. The result is a *squashed AC*. The intermediate values at the output of each chunk in the original computation become additional inputs and outputs of the squashed AC. $\mathscr{P}$ communicates these to $\mathscr{V}$, which uses them to construct the input and output vectors for the squashed AC; this binds $\mathscr{P}$ to the claimed values and the computation's inputs to its purported outputs. This technique also generalizes to the case of code "between" the chunks.

Squashing for T13. For T13-derived back-ends, the squasher takes a C program as input and identifies opportunities for converting the program's looping constructs to data-parallel computations using simple heuristics. These heuristics suffice in many cases because loops naturally express repeated subcomputations; more sophisticated analyses exist (e.g., automatic parallelization [25]).

Specifically, the squashing analysis assumes that chunks start and end at loop boundaries and comprise one or more loop iterations. Consider a loop with I dependent iterations of a computation F, where F corresponds to an AC of depth d and uniform width G. The squasher chooses N such that each chunk contains I/N unrolled iterations, and generates a sub-AC of width G and depth $d' = I \cdot d/N$, subject to a supplied cost model.

Squashing for CMT. For CMT-derived back-ends, the squasher takes as input an arbitrary AC. It searches for a value ℓ such that splitting the AC into chunks of ℓ layers minimizes cost subject to a supplied model. Unlike in T13, the boundaries of these chunks can be at any layer of the original AC because data parallelism is not required.

5 Implementation

Front-end. The front-end produces an executable manifest in Python plus a high-level arithmetic circuit description (similar to the one used by Allspice [69] and Zebra) for each outsourced sub-computation. Outsourced subcomputations in the manifest execute using the simulation framework (below). The front-end comprises about 6100 lines of Scala and 300 lines of miscellaneous glue.

Back-end. Giraffe's back-end has two components. The first is a compiler that takes the front-end's high-level AC descriptions and design parameters for $\mathscr{P}$ and $\mathscr{V}$ parallelization (Fig. 14, Apdx. C) and automatically produces $\mathscr{P}$ and $\mathscr{V}$ designs in fully synthesizable SystemVerilog. The second is a cycle-accurate simulation framework built on Icarus Verilog [73]. The back-end comprises 14 600 lines of SystemVerilog, 6800 lines of C/C++, 3300 lines of Python, and 600 lines of miscellaneous glue; parts of the SystemVerilog and C/C++ borrow from Zebra [4].

We will release all of Giraffe's code in the near future.

6 Back-end evaluation

We evaluate Giraffe's back-end by answering:

1. When does Giraffe beat "native" (§2.3)?
2. What is the largest computation Giraffe supports?
3. How does Giraffe's performance vary with computation and physical parameters?

6.1 Cross-over and scaling

Method. We consider a generic computation in the form of an arithmetic circuit $\mathscr{C}$ with depth d, sub-AC width G, number of parallel copies N, and fraction of multipliers δ. The baseline is direct evaluation of $\mathscr{C}$ on the same technology node as $\mathscr{V}$. We measure the energy cost for the baseline by summing the total cost of field operations plus the energy associated with receiving inputs and transmitting outputs of the computation.

For Giraffe's energy costs, we use a combination of simulation and modeling. The simulations are cycle-accurate Verilog simulations of Giraffe's execution. From these simulations we extract a cost model (a simplified model is given in Fig. 14, Apdx. C), and we spot check with additional simulations to ensure that this model is correct. Practical considerations demand this approach: simulating Giraffe over a broad range of parameters would be prohibitively time consuming.

We account for all costs for both $\mathscr{V}$ and $\mathscr{P}$: protocol execution, $\mathscr{V}$-$\mathscr{P}$ communication, storage, random number generation, and the cost to receive inputs and transmit outputs. We simplify the accounting of the protocol execution's energy cost by counting just the energy consumed by field operations. This approximation neglects the energy consumed by control logic and miscellaneous circuitry associated with protocol execution. As in prior work [70, §7.2], we expect these costs to be negligible; confirming this is future work. Computations in this section are over $\mathbb{F}_p, p = 2^{61} - 1$. We use concrete costs for arithmetic, communication, storage, random number generation, and I/O circuits from prior work [70, Figs. 6–7].

Results. Figure 4 compares the cost of Giraffe with the baseline. Giraffe's total cost is dominated by $\mathscr{V}$; $\mathscr{P}$'s cost is at most a few percent of the total. For small N, $\mathscr{V}$'s

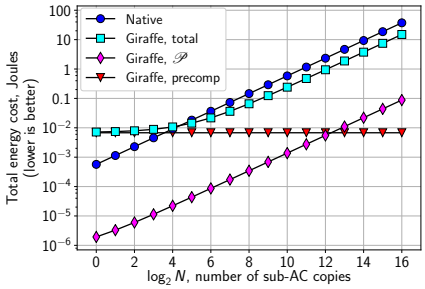


FIGURE 4—Evaluation of Giraffe's back-end. We compare Giraffe's costs to the native baseline, varying N. Giraffe beats native for $N \approx 30$. Fixed $\mathscr{C}$ parameters are: depth $d = 20$; width of sub-AC $G = 2^8$; fraction of multipliers $\delta = 0.5$; trusted technology = 350 nm; untrusted technology = 7 nm; maximum chip area $A_{\max} = 200$ mm^2. In Section 6.2 we consider manufacturing costs; there, Giraffe is less competitive with native.

precomputation (§3.3) dominates. As N increases, $\mathscr{V}$'s multilinear extension computation (§3.3) dominates. The cross-over point for savings versus native is roughly 30 copies. The cross-over point is insensitive to G because precomputation cost and per-sub-AC savings are both proportional to G, and they offset.

For the concrete costs we consider here, Giraffe can handle about 2^{16} parallel executions of a sub-AC with $G = 2^8, d = 20$; in total this is about 80 million gates. For a given hardware substrate, the maximum $N \cdot G$ product is nearly fixed. $\mathscr{P}$'s costs increase with d (Fig. 14, Apdx. C), so maximum size shrinks as d increases.

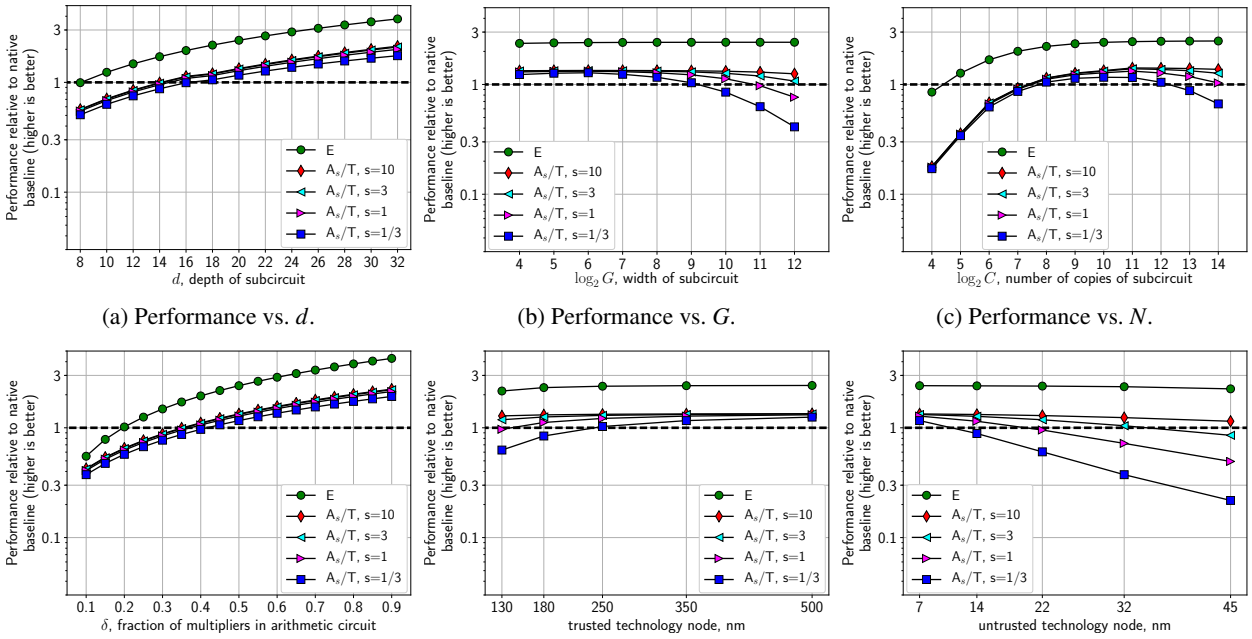
6.2 Parameter variation

Method. In addition to energy, we now consider manufacturing cost for a given performance level. Our metric is A_s/T [70]. T is throughput. $A_s = A_{\mathscr{V}} + A_{\mathscr{P}}/s$, a weighted sum of $\mathscr{V}$'s and $\mathscr{P}$'s chip area; s accounts for the difference between untrusted and trusted manufacturing costs.

We use the same simulations and detailed cost modeling as in Section 6.1 to compute costs for Giraffe. As a proxy for chip area dedicated to protocol execution, we use the area occupied by field adder and multiplier circuits. This neglects area dedicated to control logic and miscellaneous circuitry associated with protocol execution, but as in prior work [70, §7.2] we expect these costs to be negligible; confirming this is future work.

For throughput, we use cycle-accurate Verilog simulations to measure the delay of each stage of the execution and proving pipeline (Apdx. C). End-to-end throughput is given by the inverse of the maximum delay in any stage of the computation. Concrete costs are the same as in Section 6.1. For each experiment we vary one parameter and fix the others; fixed parameters are $d = 20$, $G = 2^8$, $N = 2^{10}$, $\delta = 0.5$, trusted technology node = 350 nm, and untrusted technology node = 7 nm.

For the native baseline, we optimize A/T given $A_{\max}$ subject to the arithmetic circuit's layering constraints.



(a) Performance vs. d.

(b) Performance vs. G.

(c) Performance vs. N.

(d) Performance vs. δ.

(e) Performance vs. trusted technology node.

(f) Performance vs. untrusted technology node.

FIGURE 5—Giraffe's overall performance ($\mathscr{V}$ and $\mathscr{P}$ costs) compared to native baseline on E and A_s/T metrics (§6.2), varying $\mathscr{C}$ parameters and technology nodes. In each case, we vary one parameter and fix the rest. Fixed parameters are: depth of $\mathscr{C}$, $d = 20$; width of subcircuit $G = 2^8$; number of sub-AC copies $N = 2^{10}$; fraction of multipliers $\delta = 0.5$; trusted technology node = 350 nm; untrusted technology node = 7 nm; maximum chip area $A_{\max} = 200$ mm^2.

Choosing parameters for Giraffe. We optimize Giraffe's A_s/T by choosing the design template parameters (Apdx. C). First, we fix $\mathscr{V}$'s area equal to native baseline, which is no more than $A_{\max}$. We also limit $\mathscr{P}$'s area to no more than max $A_{\max}$ and fix $n_{\mathscr{P},\text{pl}} = d$. Then we optimize $n_{\mathscr{V},\text{io}}$ and $n_{\mathscr{V},\text{sc}}$ based on available area and relative delay of sumcheck computations and multilinear extensions of inputs and outputs. Finally, given $\mathscr{V}$'s optimal delay value, we search for settings of $n_{\mathscr{P},\text{ea}}$, $n_{\mathscr{P},\tilde{\text{V}}}$, and $n_{\mathscr{P},\text{sc}}$ that optimize overall A_s/T.

Results. Figure 5 summarizes results. Giraffe's operating cost (i.e., energy consumption) beats the baseline's over a wide range of AC parameters and hardware substrates.

As in Section 6.1, energy cost is dominated by $\mathscr{V}$. Savings increase with d (Fig. 5a) because $\mathscr{V}$'s per-layer work is much less than the native baseline's. Similarly, as δ increases (Fig. 5d), the native baseline's costs increase but $\mathscr{V}$'s do not. $\mathscr{V}$'s savings are insensitive to G (Fig. 5b): the cost of multilinear extensions of I/O scales with G, balancing the increased savings in per-layer work.

Manufacturing costs are often dominated by $\mathscr{P}$. As G increases (Fig. 5b), $\mathscr{P}$'s area also increases (§3.2). As N increases (Fig. 5c), $\mathscr{P}$'s storage costs increase (Fig. 14, Apdx. C). In these cases, even if Giraffe's operating costs are better than the native baseline's, its manufacturing costs at a given performance level may be worse.

Finally, as the gap between the trusted and untrusted technology nodes shrinks (Figs. 5e and 5f), $\mathscr{P}$'s energy cost increases relative to $\mathscr{V}$'s, reducing overall performance versus the native baseline. As the trusted technology node gets more advanced (i.e., smaller, Fig. 5f), $\mathscr{V}$'s throughput increases and thus $\mathscr{P}$'s size must increase to avoid becoming a bottleneck. As the untrusted technology node gets less advanced (i.e., bigger, Fig. 5e), $\mathscr{P}$'s area grows and throughput decreases, making A_s/T worse.

7 Front-end evaluation

This section answers the following questions:

1. How does slicing result in savings compared to full outsourcing and native execution?
2. For deep loops with dependent iterations, how effective is squashing at extracting parallelism?

Setup and method. We create a sequence of programs written in C, each containing two generic blocks, F1 and F2, consisting of purely arithmetic computations. Among the programs, these blocks vary in the fraction δ_1 and δ_2 of multipliers, width of computation (G_1, G_2 respectively), the depth of the computation (d_1, d_2 respectively), and number of parallel instances N. Unless specified, we fix $N = 2^{10}$, $G_1 = G_2 = 2^8$, $d_1 = d_2 = 20$, $\delta_2 = 0.05$.

We consider two baselines: *native execution* and *full outsourcing*. The cost of native execution is defined as in

```
// x1 and x2 are inputs
// y1 and y2 are outputs
y1 = F1(x1);
y2 = F2(x2);
```

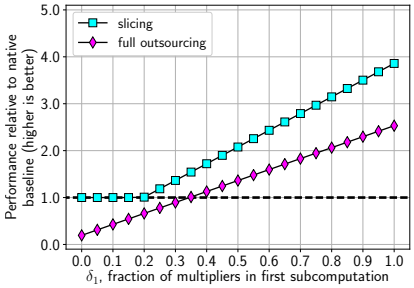
(a) Slicing: a simple computation.

```
// x1 and x2 are inputs
// y is output
if (pred) y = F1(x1);
else      y = F2(x2);
```

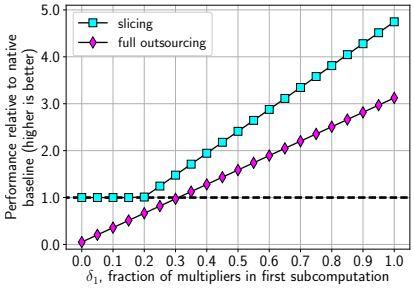
(c) Slicing: conditionals.

```
// y is output, x is input
y = x;
for (i = 0; i < I; i++)
    y = F1(y);
```

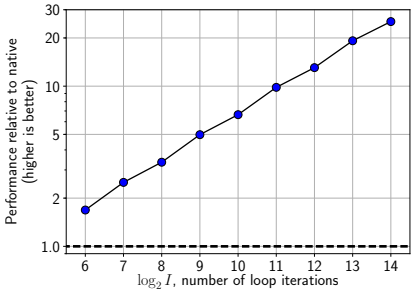
(e) Squashing: dependent iterations.



(b) Simple slicing vs. δ_1.



(d) Conditional slicing vs. δ_1.



(f) Squashing vs. number of iterations.

FIGURE 6—Evaluation of Giraffe's front end. Higher is better. F1 and F2 are computations corresponding to arithmetic circuits with $N = 2^{10}$, $G = 2^8$, $d = 20$. δ_1 and δ_2 are the fraction of multipliers in F1 and F2, respectively; we fix $\delta_2 = 0.05$. Figures 6a and 6c show inputs to Giraffe's slicing transformation. In Figures 6b and 6d, we vary δ_1, which changes whether F1 is amenable to outsourcing. We compare the efficacy of outsourcing the full computation and of first applying the slicing transform; when outsourcing would not result in savings, Giraffe executes the computation natively. Figure 6e is a deep loop with dependent iterations. Giraffe converts this to a data-parallel computation that can be outsourced, saving compared to native execution.

the prior section: the cost of computation in the same technology node as $\mathscr{V}$. We estimate the cost of full outsourcing by applying Giraffe's back-end to the raw program, *without* Giraffe's front-end transformations (§4).

To compute costs for Giraffe, we apply the selected transformation to produce a manifest (§4), then evaluate the total cost of execution, as dictated by that manifest. We use the model of Section 6 to determine the cost of the outsourced portions of the manifest. For local computations, we sum the cost of all field operations, as in the native baseline.

Slicing. We begin with a simple slicing example and then consider slicing for conditionals.

Warmup. Consider the computation of Figure 6a. We vary δ_1 from 0 to 1. For each subcomputation, we allow the front-end to decide either to outsource or to execute locally. Note that F1's amenability to outsourcing depends on δ_1: native execution cost increases with δ_1 (multiplies are more expensive than adds) while $\mathscr{V}$'s protocol costs depend only on AC *size*. Because $\delta_2 = 0.05$, F2 is not amenable to outsourcing; it native execution cost is less than the cost to outsource. For full outsourcing we generate a sub-AC that combines F1 and F2, which is conservative because it saves on precomputation.

Figure 6b plots the performance of executing the slicer's manifest and of outsourcing the entire computation, normalized to native execution. Giraffe's front-end never outsources F2 because native execution is cheaper. F1 is amenable to outsourcing when $\delta_1 > 0.2$. In contrast, full outsourcing pays extra costs for F2 compared to native execution. Thus, slicing always beats full outsourcing.

Conditionals. In Figure 6c we consider a similar setup, but with a conditional. We assume that `pred` evaluates to true, so F1 is the desired branch. Naively converting this program to an AC results in a computation that materializes both F1 and F2, and selects the result based on the value of `pred`. In essence, part of the work is useless.

Figure 6d plots the performance of executing the slicer's manifest and the performance of outsourcing the entire computation, normalized to the performance of native execution. Giraffe's front-end never outsources F2 because its branch is never taken. When $\delta_1 > 0.2$, F1 is amenable to outsourcing and Giraffe's performance is better than native. Full outsourcing, meanwhile, evaluates an AC that incurs the cost for both branches. However, for large enough δ_1, the savings from F1 offsets the useless work, allowing full outsourcing to beat native.

Squashing. We also experiment with a loop comprising I iterations of F1 (Fig. 6e). Parameters are as above, $\delta_1 = 0.5$, and we vary I. This is deep ($I \cdot d_1$) and narrow (G_1), and not data parallel. The squasher (§4) chooses N. Effective depth is $d' = I \cdot d/N$ for each chunk, balancing $\mathscr{V}$'s I/O cost against the per-layer cost. This happens when depth and $|x| + |y|$ are within a constant factor, i.e., $N \cdot G = d' = \mathrm{O}(\sqrt{I})$ (overall cost is the sum). Figure 6f shows the results: as I increases from 2^{11} to 2^{14}, performance improves by $\approx 3\times$.

8 Applications

8.1 Curve25519

Curve25519 is a high-performance elliptic curve used in cryptographic protocols [5, 23]. This section compares

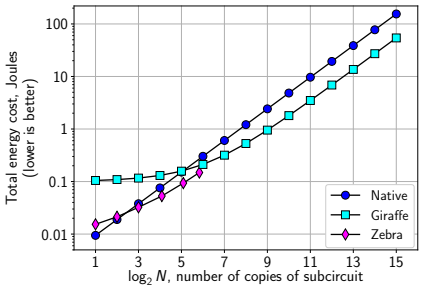


FIGURE 7—Energy cost of Giraffe, native baseline (§8.1), and Zebra [70, §8.2] versus number of copies of Curve25519 subcircuit. Each subcircuit computes 20 parallel evaluations of five sequential double-and-add steps. Untrusted technology node = 350 nm; trusted technology node = 7 nm; $A_{\max} = 200$ mm^2. Zebra's scaling is limited to about 1150 parallel evaluations. Giraffe scales to more than 500× more parallel computations for the same chip area. Because of Giraffe's refinements (§3), its improvement compared to native is greater than Zebra's. But Giraffe must amortize precomputation, so it needs more subcomputations than Zebra to break even.

three implementations of the *point multiplication* operation on this curve: a baseline, Zebra, and Giraffe. This operation takes as inputs a 255-bit scalar value v and a curve point Q, and computes the point $R = [v]Q$ via 255 *double-and-add* steps [11], one for each bit of v. Our algorithm employs a Montgomery ladder, as is standard [11, 23, 54]. Double-and-add is naturally expressed as an arithmetic circuit over $\mathbb{F}_p$, $p = 2^{255} - 19$, with $d = 7$ and $G \approx 8$.

Zebra. This implementation [70, §8.2] groups 5 Montgomery ladder steps into a block and requires 51 (= $^{255}/_{5}$) iterations of this block per operation. Zebra uses a special *mux* gate for efficiency, requiring all double-and-add operations in a protocol run to use the same scalar input v. The authors argue that this restriction is acceptable, with reference to specific applications.

Baseline implementation. Consistent with published hardware implementations of point multiplication on Curve25519 [59, 60] and the implementation from Zebra, our baseline directly executes 5 Montgomery ladder steps.

Giraffe. In Giraffe there are two degrees of freedom: L, the number of parallel double-and-add steps in a sub-AC (which determines G); and N. Each copy of the sub-AC uses the same L scalars, $\{v_1, \ldots, v_L\}$; this is because wiring predicates are reused across the N sub-ACs. In our experiment, we fix $L = 20$, and vary N; larger values of L are also possible.

Results. We compute energy for Giraffe and the native baseline as in Section 6.1. For Zebra, we use published results [70, §8.2]. We set the untrusted technology node = 350 nm, the trusted technology node = 7 nm, and $A_{\max} = 200$ mm^2, the same as in Zebra.

Figure 7 shows the results. Giraffe breaks even when $N \approx 30$, or at about 600 parallel double-and-add operations. In contrast, Zebra breaks even for about 100 parallel operations. This is because Giraffe pays the cost of precomputation while Zebra does not. However, the Zebra system handles at most 1150 parallel copies for the given chip area, whereas Giraffe can accommodate roughly 32,000 parallel operations corresponding to roughly 100M arithmetic gates, about 500× more than Zebra, for the same technology node parameters.

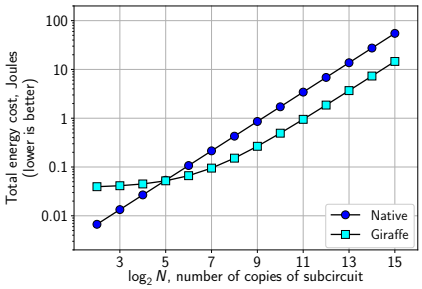


FIGURE 8—Energy cost of Giraffe and native baseline (§8.2) versus number of parallel image pyramid matching evaluations. Each evaluation matches 16-word needles against 128-word haystacks for a two-level image pyramid (§8.2). Words are represented as elements in $\mathbb{F}_p, p = 2^{61} + 2^{19} + 1$. Untrusted technology node = 350 nm; trusted technology node = 7 nm; $A_{\max} = 200$ mm^2. Giraffe breaks even for ≈ 30 parallel searches.

8.2 Image pyramid

An *image pyramid* is a data structure for image processing [6] that holds multiple copies of an image at different resolutions. The "base" of the pyramid is a full resolution image and higher "layers" summarize the image at progressively lower resolutions. One application of an image pyramid is *fast pattern matching*. In the first step, a coarse pattern is matched against the coarsest layer (top) of the pyramid. Guided by the results, a finer pattern is matched against a small part of the next layer until eventually a portion of the full resolution image is matched against the finest pattern.

We use a convolution-based matching algorithm [30] that allows the pattern to contain "don't care" symbols that match any input. If the text is $t = t_0 t_1 \ldots t_n$ and the pattern is $p = p_0 p_1 \ldots p_m$, then the matching algorithm uses convolutions to compute $c_i = \sum_{j=0}^{m} p_j (p_j - t_{i+j})^2$ for each $i \in \{1, \ldots, n\}$ and reports a match at i if $c_i = 0$.

In our implementation, the input consists of a two-layer image pyramid, a coarse pattern, and a fine pattern. The bottom layer of the pyramid has $2^7 \times 2^7$ words, and the top layer has 1×2^7 words. Both patterns comprise 2^4 words. Words are represented over $\mathbb{F}_p, p = 2^{61} + 2^{19} + 1$, and we implement convolution using the number theoretic transform over $\mathbb{F}_p$. The entire application processes N instances in parallel; each instance specifies its own input and pattern. The application is written as a C program.

Baseline implementation. In our baseline implementation, each convolution is implemented using the direct iterative implementation of the number theoretic trans-

form (NTT) and its inverse. Energy costs are accounted as in the baseline of Section 6.

Giraffe. We apply Giraffe's front-end to process our C program into a manifest; the local computation selects the needed portion of the next layer. We compute energy consumption of the resulting manifest as in Section 7. We report the results of fully automated compilation on a realistic application: no hand optimization was applied.

Results. Figure 8 compares the cost of executing the manifest to the cost of the native baseline. Trusted and untrusted technology nodes and $A_{\max}$ are as in Section 8.1. Giraffe needs roughly 30 parallel evaluations to break even, after which it uses 5× less energy than the baseline. Giraffe can scale to handle 32,000 parallel instances within the area constraint, or about 100 million AC gates.

9 Discussion and limitations

To understand Giraffe's results, it is useful to provide context about overhead and break-even points from other implemented systems. For example, in the SNARK (§10) literature, careful examination indicates that verifier overhead is so high that enormous computations are required to break even: millions of AC gates [72; 21; 56, §5.3; 71, §2]. Yet, the maximum size that implementations can handle is around 20 million AC gates (due to memory limits). And even on a best-case problem (matrix multiplication), Pinocchio [56] requires more than 6,000 instances, and BCTV [20, 22] requires more than 90,000 instances to break even [72, Fig.4]. (Note that we have not even discussed keeping track of prover overhead; even for small ACs, these provers takes minutes on stock hardware.)

In contrast, Giraffe's performance (keeping track of prover costs) has only a weak dependence on computation size, even for ACs of only a few hundred gates (Figs. 5a and 5b, §6.2). Moreover, the number of parallel copies required to amortize is small, ≈ 30 (§6, §8). The maximum instance size for a Giraffe sub-AC is around 1.5 million gates; this is largely a function of the constraints imposed by hardware. These numbers are very encouraging (although note that [69] achieves similar instance numbers). Of course, SNARKs have distinct advantages: precomputation amortizes indefinitely in the non-interactive setting, and a broader class of computations can be handled (subject to the usual AC requirement).

Since Giraffe is largely focused on the hardware setting, it is also worthwhile to contrast with Zebra [70]. On the one hand, Zebra does not impose the requirement for data parallel computations (to amortize precomputation). But on the other, Zebra is limited to approximately 500,000 AC gates total; as Giraffe supports 1.5 million per sub-AC and N scales to 50 in this case, Giraffe is two orders of magnitude better than Zebra in total size. More importantly, Giraffe can break even despite paying for precomputation, while Zebra's costs—let alone whether it breaks even—are reasonable only under a fanciful assumption about daily delivery of trusted precomputations [70, §4].

Since it is hard to imagine a near-future system that relaxes the size constraints, Giraffe uses program transformation to extend its applicability to the wide range of large programs that have portions suitable for outsourcing. As the image pyramid example (§8.2) demonstrates, Giraffe can be practical in situations where Giraffe simply cannot outsource the entire computation.

To be sure, Giraffe has serious limitations. The price of verification remains high and evaluation shows that the overall win is not that substantial (Fig. 4, §6). Given the prover overhead, Giraffe still requires a large technology gap between the $\mathscr{P}$ and $\mathscr{V}$ technology nodes to be practical (§6.2). And finally, the regime of applicability is fundamentally narrow (as noted in the introduction).

10 Related work

Probabilistic proofs. Giraffe relates to the extensive body of recent work on verifiable outsourced computation [14, 15, 17–19, 21, 22, 28, 29, 31–35, 38, 40, 47, 55, 56, 61–64, 66, 68–71]; see [72] for a comparatively recent survey. Specifically, Giraffe descends from the GKR [42] interactive proof line [31, 66, 68–70]. This line imposes certain limitations (a more restricted class of computations, limitations on auxiliary prover input). Another line of work uses argument protocols, both interactive [62–64] and non-interactive [14, 21, 24, 40, 47, 56] (the latter are known as SNARKs). However, these protocols seem largely incompatible with hardware implementation (see [70, §9] for discussion of the issues), impose more stringent cryptographic assumptions (particularly in the non-interactive setting), and tend to have higher precomputation costs. (These costs can be asymptotically limited but at very high concrete cost [21, 22, 29]—for example, the prover is two [22] to six [21, 29] orders of magnitude worse.) On the other hand, non-interactive arguments can support zero knowledge (zkSNARK) protocols; this enables applications that are not possible otherwise.

Much of the work in the area fits into the cost framework outlined in the introduction: precomputation, verifier overhead, and prover overhead, with native execution as a sensible baseline. There are a few exceptions. In the zkSNARK setting, the cost assessment depends on the premium that one is willing to pay for otherwise unachievable functionality [14, 18, 33]. Also, two works in the verifiable outsourcing setting do not require precomputation. The first is CMT [31, 68] (and [66]) when applied to highly regular wiring patterns; however, such wiring patterns substantially limit applicability. The second is SCI [17], which aims to be general purpose. SCI is, roughly speaking, an argument protocol built atop "short PCPs", and is an exciting development. However,

inspection of SCI makes clear that the costs are orders of magnitude higher than in other works in the area, and that the built system does not currently scale beyond very small problem sizes.

PL techniques in cryptographic protocols. Squashing (§4) is related to but distinct from Geppetto's [32] optimizations for loops. At a high level the goals are similar (use loop transformations to adapt a computation to a protocol), but they differ in particulars because each technique leverages features of its respective back-end. In settings where they are both relevant, we believe the two approaches are complementary. (Giraffe pursues automatic inference for this optimization, which is discussed but not explored in [32].)

Our work on slicing is in the tradition of a great deal of work adapting PL techniques to generating code that implements cryptographic protocols and manages their interaction with external programs. In the verifiable outsourcing literature, there are a handful of examples (e.g., Buffet [71] uses sophisticated loop unrolling techniques to optimize loop handling, and Geppetto analyzes conditionals to minimize evaluation of "dead code").

More generally, the secure multi-party computation literature has seen a great deal of work using program analysis and transformation techniques to produce optimized protocols, starting with Fairplay [50] and notably including the line of work represented by [74]. There has also been relevant work in the Oblivious RAM community, for example [48] using PL techniques to partition variables to ensure obliviousness. Another area in which these techniques are used is in automatic compilation for secure distributed programming [37]. Perhaps most similar to our slicing protocol are the various compilers for zero knowledge proofs of knowledge [7, 16, 52], most notably ZQL and Z0 [36, 38]. The latter weaves together explicitly annotated zero knowledge regions with ordinary code, and does automatic inference for assigning functionality to tiers in client-server applications (see also [51] for automatic tier partitioning). Giraffe is distinguished by performing automatic inference for slicing based on a cost model without explicit annotation.

Acknowledgments

We thank Fraser Brown and Keith Winstein for helpful comments. The authors were supported by NSF grants CNS-1423249, CNS-1514422, and CNS-1646671; AFOSR grant FA9550-15-1-0302; ONR grant N00014-16-1-2154; DARPA grant HR0011-15-2-0047; and a Google Research Award.

Giraffe's source code is available at:
`http://www.pepper-project.org/`

References

[1] `https://github.com/pepper-project/releases/blob/master/ginger-allspice.tar.gz`.
[2] `http://people.cs.georgetown.edu/jthaler/code/code.htm`.
[3] `http://people.cs.georgetown.edu/jthaler/TRMPcode.htm`.
[4] `https://github.com/pepper-project`.
[5] Things that use Curve25519. `https://ianix.com/pub/curve25519-deployment.html`.
[6] E. H. Adelson, C. H. Anderson, J. R. Bergen, P. J. Burt, and J. M. Ogden. Pyramid method in image processing. *RCA Engineer*, 29(6):33–41, Nov. 1984.
[7] J. B. Almeida, M. Barbosa, E. Bangerter, G. Barthe, S. Krenn, and S. Z. Béguelin. Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols. In *ACM CCS*, 2012.
[8] S. Arora and B. Barak. *Computational Complexity: A modern approach*. Cambridge University Press, 2009.
[9] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, May 1998.
[10] S. Arora and S. Safra. Probabilistic checking of proofs: a new characterization of NP. *J. ACM*, 45(1):70–122, Jan. 1998.
[11] R. M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2005.
[12] L. Babai. Trading group theory for randomness. In *STOC*, May 1985.
[13] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *STOC*, May 1991.
[14] M. Backes, M. Barbosa, D. Fiore, and R. M. Reischuk. ADSNARK: Nearly practical and privacy-preserving proofs on authenticated data. In *IEEE S&P*, May 2015.
[15] M. Backes, D. Fiore, and R. M. Reischuk. Verifiable delegation of computation on outsourced data. In *ACM CCS*, Nov. 2013.
[16] E. Bangerter, J. Camenisch, S. Krenn, A. Sadeghi, and T. Schneider. Automatic generation of sound zero-knowledge protocols. *IACR Cryptology ePrint Archive*, 2008. http://eprint.iacr.org/2008/471.
[17] E. Ben-Sasson, I. Ben-Tov, A. Chiesa, A. Gabizon, D. Genkin, M. Hamilis, E. Pergament, M. Riabzev, M. Silberstein, E. Tromer, and M. Virza. Computational integrity with a public random string from quasi-linear PCPs. In *EUROCRYPT*, 2017.
[18] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Decentralized anonymous payments from Bitcoin. In *IEEE S&P*, May 2014.
[19] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO*, Aug. 2013.
[20] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. TinyRAM architecture specification, v0.991. `http://www.scipr-lab.org/system/files/TinyRAM-spec-0.991.pdf`, 2013.
[21] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO*, Aug. 2014.
[22] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security*, Aug. 2014.
[23] D. J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *PKC*, Apr. 2006.
[24] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, Jan. 2012.
[25] P. Boulet, A. Darte, G. Silber, and F. Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 24(3-4):421–444, 1998.
[26] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *J. of Comp. and Sys. Sciences*,

37(2):156–189, Oct. 1988.
[27] B. Braun. Compiling computations to constraints for verified computation. UT Austin Honors thesis HR-12-10, Dec. 2012.
[28] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*, Nov. 2013.
[29] A. Chiesa, E. Tromer, and M. Virza. Cluster computing in zero knowledge. In *EUROCRYPT*, Apr. 2015.
[30] P. Clifford and R. Clifford. Simple deterministic wildcard matching. *Information Processing Letters*, 101(2):53 – 54, 2007.
[31] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, Jan. 2012.
[32] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. In *IEEE S&P*, May 2015.
[33] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno. Cinderella: Turning shabby X.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *IEEE S&P*, May 2016.
[34] D. Fiore, C. Fournet, E. Ghosh, M. Kohlweiss, O. Ohrimenko, and B. Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *ACM CCS*, Oct. 2016.
[35] D. Fiore, R. Gennaro, and V. Pastro. Efficiently verifiable computation on encrypted data. In *ACM CCS*, Nov. 2014.
[36] C. Fournet, M. Kohlweiss, G. Danezis, and Z. Luo. ZQL: A compiler for privacy-preserving data processing. In *USENIX Security*, Aug. 2013.
[37] C. Fournet, G. Le Guernic, and T. Rezk. A security-preserving compiler for distributed programs: From information-flow policies to cryptographic mechanisms. In *ACM CCS*, 2009.
[38] M. Fredrikson and B. Livshits. ZØ: An optimizing distributing zero-knowledge compiler. In *USENIX Security*, Aug. 2014.
[39] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, Aug. 2010.
[40] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, 2013.
[41] C. Gentry and D. Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *STOC*, June 2011.
[42] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. *J. ACM*, 62(4):27:1–27:64, Aug. 2015. Prelim version STOC 2008.
[43] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. on Comp.*, 18(1):186–208, 1989.
[44] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *IEEE CCC*, June 2007.
[45] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC*, May 1992.
[46] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*, chapter 4.6.4. Addison-Wesley, third edition, 1997.
[47] A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos. TRUESET: Faster verifiable set computations. In *USENIX Security*, Aug. 2014.
[48] C. Liu, M. Hicks, and E. Shi. Memory trace oblivious program execution. In *Computer Security Foundations Symposium (CSF)*, June 2013.
[49] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39(4):859–868, Oct. 1992.
[50] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *USENIX Security*, Aug. 2004.
[51] D. Manolescu, B. Beckman, and V. B. Livshits. Volta: Developing distributed applications by recompiling. *IEEE Software*, 2008.
[52] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. ZKPDL: a language-based system for efficient zero-knowledge proofs and electronic cash. In *USENIX Security*, 2010.
[53] S. Micali. Computationally sound proofs. *SIAM J. on Comp.*, 30(4):1253–1298, 2000.
[54] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Math. of Computation*, 48(177):243–264, Jan. 1987.
[55] A. Naveh and E. Tromer. PhotoProof: Cryptographic image authentication for any set of permissible transformations. In *IEEE S&P*, May 2016.
[56] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE S&P*, May 2013.
[57] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Communications of the ACM*, volume 59, pages 103–112, Feb. 2016.
[58] G. N. Rothblum. *Delegating Computation Reliably: Paradigms and Constructions*. PhD thesis, Massachusetts Institute of Technology, 2009.
[59] P. Sasdrich and T. Güneysu. Efficient elliptic-curve cryptography using Curve25519 on reconfigurable devices. In *ARC*, Apr. 2014.
[60] P. Sasdrich and T. Güneysu. Implementing Curve25519 for side-channel–protected elliptic curve cryptography. *ACM TRETS*, 9(1):3:1–3:15, Nov. 2015.
[61] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *HotOS*, May 2011.
[62] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*, Apr. 2013.
[63] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, Feb. 2012.
[64] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, Aug. 2012.
[65] A. Shamir. IP = PSPACE. *J. ACM*, 39(4):869–877, Oct. 1992.
[66] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO*, Aug. 2013. Citations refer to full version: `https://arxiv.org/abs/1304.3812`.
[67] J. Thaler. A note on the GKR protocol. http://people.seas.harvard.edu/ jthaler/GKRNote.pdf, 2015.
[68] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud Workshop*, June 2012.
[69] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE S&P*, May 2013.
[70] R. S. Wahby, M. Howald, S. Garg, abhi shelat, and M. Walfish. Verifiable ASICs. In *IEEE S&P*, May 2016.
[71] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, Feb. 2015.
[72] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near practicality. *Communications of the ACM*, 58(2):74–84, Feb. 2015.
[73] S. Williams. Icarus Verilog. `http://iverilog.icarus.com`.
[74] S. Zahur and D. Evans. Circuit structures for improved efficiency of security and privacy tools. In *IEEE S&P*, pages 493–507, May 2013.

```
1: function SUMCHECKV(layer i, a_{i-1})
2:     e ← a_{i-1}
3:
4:     (r', r_0, r_1) ←R F^{b_N} × F^{b_G} × F^{b_G}
5:     r ← (r', r_0, r_1) // variable order is from §3.1
6:     // For the protocol of Theorem A.1 (variable order of §2.2)
7:     // replace the above line with r ← (r_0, r_1, r')
8:
9:     for j = 1, 2, ..., (b_N + 2b_G) do
10:
11:        // F_j is a degree-2 or degree-3 polynomial
12:        F_j ← ReceiveFromProver() // see lines 18,46 of Fig. 12
13:        // For the protocol of Theorem A.1,
14:        // see lines 22 and 41 of Figure 13
15:
16:        if F_j(0) + F_j(1) ≠ e then
17:            return reject
18:
19:        SendToProver(r[j])
20:
21:        e ← F_j(r[j])
22:
23:    return (e, r', r_0, r_1)
```

FIGURE 9—Part of Giraffe's and T13's $\mathscr{V}$ pseudocode. Reduces the claim that a_i equals the sum $\sum_{n,h_0,h_1} P^*_{q,i}(n,h_0,h_1)$ (this sum equals $\tilde{V}_{i-1}(q'_{i-1},q_{i-1})$, per Equation (2)) to the claim $e = P^*_{q,i}(r',r_0,r_1)$. The depiction here follows Section 3.1: r' comes before r_0, r_1 in the variable order, and the polynomial is $P^*_{q,i}(n,h_0,h_1)$, not $P_{q,i}(h_0,h_1,n)$.

```
1: function PROVE(ArithCircuit c, input x)
2:     (q'_0, q_0) ← ReceiveFromVerifier()    // see line 4
3:     d ← c.depth
4:
5:     // each circuit layer induces one sumcheck invocation
6:     for i = 1, ..., d do
7:         r', r_0, r_1 ← SUMCHECKP(c, i, (q'_{i-1}, q_{i-1}))
8:         τ_i ← ReceiveFromVerifier()    // see line 29 of Figure 1
9:         (q'_i, q_i) ← (r', (r_1 − r_0) · τ_i + r_0)
```

FIGURE 10—Part of Giraffe's $\mathscr{P}$ pseudocode. (The corresponding pseudocode for the protocol of Theorem A.1 is identical, though it refers to the SUMCHECKP pseudocode in Figure 13 rather than Figure 12.)

A Details of T13 (with an optimization)

Recall from §2.2 that the starting point of Giraffe's backend is *T13* [66, §7], with an optimization [67]. A complete description of the verifier's work in this protocol can be found in Figures 1 and 9. A complete description of the prover's work can be found in Figures 10 and 13.

The following theorem restates the relevant properties of this protocol (§2.2): completeness, soundness, $\mathscr{V}$'s runtime, and $\mathscr{P}$'s runtime. The proof of this theorem is omitted for brevity; it essentially follows the analysis of [66, §7], as the only difference between the protocol of [66, §7] and the protocol of this section is the inclusion of the optimization of [67]. We do, however, provide a detailed proof of the claim about $\mathscr{V}$'s runtime, as Giraffe's verifier is implemented in a similar manner.

Theorem A.1. *Consider the protocol with verifier described in Figures 1 and 9, and prover described in Figure 13. When applied to a circuit $\mathscr{C}$ as in Section 2.2, the protocol satisfies completeness, and satisfies soundness with $\varepsilon = (\lceil \log |y| \rceil + 6d \log(G \cdot N))/|\mathbb{F}|$. $\mathscr{V}$ requires precomputation that is $O(d \cdot G)$. Then, to validate all inputs and outputs, $\mathscr{V}$ incurs cost $O(d \cdot \log(N \cdot G) + |x| + |y|)$. $\mathscr{P}$'s running time is $O(d \cdot G \cdot N \cdot \log G)$.*

It will be convenient to have the following expression for the multilinear extension. For a function $f\colon \{0,1\}^\gamma \to \mathbb{F}$, the multilinear extension $\tilde{f}$ of f is given by:

$$\tilde{f} = \sum_{s \in \{0,1\}^\gamma} f(s) \cdot \chi_s. \tag{6}$$

This follows because both sides of the equality are multilinear polynomials that agree at all Boolean inputs, and hence must be equal as formal polynomials.

Remark. $\tilde{f}$ can be viewed as an encoding of a table of f's values. Specifically, let us view $f(\cdot)$ as a function table with 2^γ entries, where each $s \in \{0,1\}^\gamma$ is an index into that table. Notice that every point in the domain of $\tilde{f}$ is a linear combination of all 2^γ entries in this table.

$\mathscr{V}$'s Precomputation. $\mathscr{V}$'s precomputation evaluates $\tilde{\text{add}}_i(q_{i-1},r_0,r_1)$ and $\tilde{\text{mult}}_i(q_{i-1},r_0,r_1)$ for each $i = 1,\ldots,d$, and all points $(q_{i-1},r_0,r_1) \in \mathbb{F}^{\log G} \times \mathbb{F}^{\log G} \times \mathbb{F}^{\log G}$ encountered in Lines 20 and 21 of Figure 1 over the course of the protocol execution.[5]

Hence, to show that $\mathscr{V}$'s precomputation work is $O(d \cdot G)$, it suffices to show that for each i, $\tilde{\text{add}}_i(q_{i-1},r_0,r_1)$ and $\tilde{\text{mult}}_i(q_{i-1},r_0,r_1)$ can be evaluated in $O(G)$ time. An algorithm for achieving this was claimed by Vu et al. [69]; we present the details of such an algorithm.

Let $S_{\text{add},i} \subseteq S_{\text{all},i} \subseteq \{0,1\}^{3b_G}$ denote the set of all addition gates at layer-$(i-1)$, with their layer-i neighbors, and similarly for $S_{\text{mult},i}$. By Equation (6),

$$\begin{aligned}\tilde{\text{add}}_i &= \sum_{u \in \{0,1\}^{3b_G}} \text{add}_i(u) \cdot \chi_u = \sum_{u \in S_{\text{add},i}} \chi_u \\ &= \sum_{(g,g_L,g_R) \in S_{\text{add},i}} \chi_g \cdot \chi_{g_L} \cdot \chi_{g_R}\end{aligned} \tag{7}$$

Likewise,

$$\text{mult}_i = \sum_{(g,g_L,g_R) \in S_{\text{mult},i}} \chi_g \cdot \chi_{g_L} \cdot \chi_{g_R} \tag{8}$$

[5] Figure 1 states that the (q_{i-1},r_0,r_1) values are only determined over the course of the protocol execution, but in fact they can be determined in precomputation, as they only depend on $\mathscr{V}$'s randomness.

1: // $t[\ell] \in \mathbb{F}$ are elements of vector t,
2: // which is indexed $1,\ldots,b_G$ from LSB to MSB
3: // A is a vector of length G
4:
5: $A[0] \leftarrow 1 - t[b_G]$
6: $A[1] \leftarrow t[b_G]$
7: **for** $\ell = b_G - 1, b_G - 2, \ldots, 1$ **do**
8: **for** $k = 2^{b_G-\ell} - 1, 2^{b_G-\ell} - 2, \ldots, 0$ **do**
9: $A[2k] \leftarrow (1 - t[\ell]) \cdot A[k]$
10: $A[2k+1] \leftarrow t[\ell] \cdot A[k]$

FIGURE 11—Pseudocode for computing $A_t = \{\chi_0(t), \ldots, \chi_{G-1}(t)\}$ in time $O(G)$. $\mathcal{V}$ needs to compute A_q, A_{r_0}, and A_{r_1}. Each of q, r_0, r_1 is a member of $\mathbb{F}^{b_G}$.

Hence, $\mathcal{V}$'s algorithm for evaluating $\tilde{\text{add}}_i(q_{i-1}, r_0, r_1)$ and $\tilde{\text{mult}}_i(q_{i-1}, r_0, r_1)$ first constructs three zero-indexed arrays, each with G elements:

$$A_q = \{\chi_0(q), \ldots, \chi_{G-1}(q)\}$$
$$A_{r_0} = \{\chi_0(r_0), \ldots, \chi_{G-1}(r_0)\}$$
$$A_{r_1} = \{\chi_0(r_1), \ldots, \chi_{G-1}(r_1)\}.$$

To construct each array, consider the algorithm in Figure 11. This algorithm uses dynamic programming to avoid recomputing suffixes. For example, notice that for all even $h \in [0, G-1]$, $\chi_h(q) = (1 - q[1]) \cdot L$ and $\chi_{h+1}(q) = q[1] \cdot L$, where $L = \prod_{\ell=2}^{b_G} \chi_{h[\ell]}(q[\ell])$; the algorithm computes L only once. Constructing an array takes $O(G)$ time because for each iteration of the outer loop, the number of iterations in the inner loop ascends as $2^1, 2^2, \ldots, 2^{b_G-1}$, making the total number of inner loop iterations $\sum_{i=1}^{b_G-1} 2^i < 2^{b_G} = G$. Moreover, each inner loop iteration requires 2 field multiplications, so constructing all 3 arrays requires at most $6 \cdot G$ multiplications.

Once the three arrays are computed, $\mathcal{V}$ computes the right hand sides of Equations (7) and Equation (8) by iterating over each gate $s = (g, g_L, g_R)$; looking up the three quantities $A_q[g]$, $A_{r_0}[g_L]$, $A_{r_1}[g_R]$; multiplying them; and adding this product to a running sum for $\tilde{\text{add}}_i(q, r_0, r_1)$ or $\tilde{\text{mult}}_i(q, r_0, r_1)$, depending on whether the gate is an addition or multiplication gate. This requires an additional $2G$ multiplications, and G additions.

In summary, the above shows that both $\tilde{\text{add}}_i(q_{i-1}, r_0, r_1)$ $\tilde{\text{mult}}_i(q_{i-1}, r_0, r_1)$ can be computed in $O(G)$ time, with at most $8G$ field multiplications in total.

Finally, recall that prior work has the prover specify the univariate polynomial H appearing in Figure 1 by specifying its evaluations at $b_G + 1$ inputs (§3.3). Some works have $\mathcal{V}$ evaluate Lagrange basis polynomials at various points in precomputation, in $O(d \log^2 G)$ time [69, 70]. This ensures that $\mathcal{V}$ can later evaluate $H(\tau_i)$ (line 26, Figure 1) in $O(\log G)$ time per evaluation.

$\mathcal{V}$'s remaining costs. Given the results of $\mathcal{V}$'s precomputation, inspection of Figures 1 and 9 indicates that $\mathcal{V}$ runs in $O(d \cdot \log(N \cdot G) + |x| + |y|)$ time, provided that $\mathcal{V}$ can accomplish the following tasks in the following time bounds:

- For any point $(q'_d, q_d) \in \mathbb{F}^{b_N} \times \mathbb{F}^{b_G}$, evaluate $\tilde{V}_0(q'_0, q_0)$ in time $O(|y|)$.
- For any point $(q'_{i-1}, r') \in \mathbb{F}^{b_N} \times \mathbb{F}^{b_N}$, evaluate $\widetilde{\text{eq}}(q'_{i-1}, r')$ in time $O(b_N)$.
- For any point $(q'_d, q_d) \in \mathbb{F}^{b_N} \times \mathbb{F}^{b_G}$, evaluate $\tilde{V}_d(q'_d, q_d)$ in time $O(|x|)$.

The first and third bullets are handled as in Section 3.3 (cf. the paragraph on multilinear extensions of I/O). To establish the second bullet, note that $\widetilde{\text{eq}}\colon \mathbb{F}^{2b_N} \to \mathbb{F}$ has the following form [58, Prop. 3.2.1] (see also [69, Apdx. A.1]):[6]

$$\widetilde{\text{eq}}(q', r') = \prod_{\ell=1}^{b_N} \left(q'[\ell] \cdot r'[\ell] + (1 - q'[\ell]) \cdot (1 - r'[\ell])\right) \quad (9)$$

Each term simplifies to $2q'[\ell] \cdot r'[\ell] + 1 - (q'[\ell] + r'[\ell])$, which can be computed with one multiplication and four additions. Thus the whole computation requires $4b_N$ additions and $2b_N - 1$ multiplications.

B Details of Giraffe's back-end

Recall from §3.1 that Giraffe's back-end differs from the protocol of Appendix A in that it changes the order in which variables are bound within each invocation of the sum-check protocol, and exploits that order to simplify $\mathcal{P}$'s work.

A complete description of the verifier's work in this protocol can be found in Figure 1 and Figure 9. A complete description of the prover's work can be found in Figures 10 and 12. The following theorem restates the relevant properties of this protocol.

Theorem B.1. *Consider the protocol with verifier described in Figures 1 and 9, and prover described in Figure 12. When applied to a circuit $\mathcal{C}$ as in Section 2.2, the protocol satisfies completeness, and satisfies soundness with $\varepsilon = (\lceil \log |y| \rceil + 6d \log(G \cdot N))/|\mathbb{F}|$. $\mathcal{V}$ requires precomputation that is $O(d \cdot G)$. Then, to validate all inputs and outputs, $\mathcal{V}$ incurs cost $O(d \cdot \log(N \cdot G) + |x| + |y|)$. $\mathcal{P}$'s running time is $O(d \cdot (G \cdot N + G \cdot \log G))$.*

The conclusion of Theorem B.1 is identical to that of Theorem A.1, except for the improvement in $\mathcal{P}$'s runtime.

Proof. The proof of completeness, soundness, and the bound on $\mathcal{V}$'s runtime is essentially identical to that of

[6] The validity of this equation can be seen by observing that the right hand side is a multilinear polynomial in the components of q' and r', and agrees with the function eq whenever q' and r' are in $\{0,1\}^{b_N}$

function SUMCHECKP(ArithCircuit c, layer i, q'_{i-1}, q_{i-1})
  for $j = 1, \ldots, b_N$ **do**
    // Prover sends degree-3 polynomial F_j. Does this by computing $F_j(-1), F_j(0), F_j(1), F_j(2)$ and then interpolating.

    for all $\sigma \in \{0,1\}^{b_N - j}$ and $g \in \{0,1\}^{b_G}$ and $k \in \{-1,0,1,2\}$ **do**
      $s \leftarrow (g, g_L, g_R)$ // g_L, g_R are labels of g's layer-i inputs in sub-circuit.

      termP $\leftarrow \widetilde{\text{eq}}(q'_{i-1}, r'[1], \ldots, r'[j-1], k, \sigma[1], \ldots, \sigma[b_N - j]) \cdot \chi_g(q_{i-1})$
      termL $\leftarrow \tilde{V}_i(r'[1], \ldots, r'[j-1], k, \sigma[1], \ldots, \sigma[b_N - j], g_L)$
      termR $\leftarrow \tilde{V}_i(r'[1], \ldots, r'[j-1], k, \sigma[1], \ldots, \sigma[b_N - j], g_R)$

      if g is an add gate **then** $F_j[\sigma, g][k] \leftarrow$ termP $\cdot$ (termL + termR)
      else if g is a mult gate **then** $F_j[\sigma, g][k] \leftarrow$ termP $\cdot$ termL $\cdot$ termR

    for $k \in \{-1,0,1,2\}$ **do**
      $F_j[k] \leftarrow \sum_{\sigma \in \{0,1\}^{b_N - j}} \sum_{g \in \{0,1\}^{b_G}} F_j[\sigma, g][k]$
    // Use Lagrange interpolation to compute coefficients of F_j and send them to $\mathcal{V}$
    SendToVerifier(F_j, 3)
    $r'[j] \leftarrow$ ReceiveFromVerifier() // see line 19 of Figure 9

  $r' \leftarrow (r'[1], \ldots, r'[b_N])$ // notation

  for $j = 1, \ldots, 2b_G$ **do**
    // In these rounds, prover sends degree-2 polynomial $F_{b_N + j}$.
    for all gates $g \in \{0,1\}^{b_G}$ and $k \in \{-1,0,1\}$ **do**

      $s \leftarrow (g, g_L, g_R)$ // g_L, g_R are labels of g's layer-i inputs in subcircuit
      $u_k \leftarrow (q_{i-1}[1], \ldots, q_{i-1}[b_G], r[1], \ldots, r[j-1], k)$
      termP $\leftarrow \widetilde{\text{eq}}(q'_{i-1}, r') \cdot \prod_{\ell=1}^{b_G + j} \chi_{s[\ell]}(u_k[\ell])$

      if $j \leq b_G$ **then**
        termL $\leftarrow \tilde{V}_i(r', r[1], \ldots, r[j-1], k, g_L[j+1], \ldots, g_L[b_G])$
        termR $\leftarrow \tilde{V}_i(r', g_R)$
      else // $b_G < j \leq 2b_G$
        termL $\leftarrow \tilde{V}_i(r', r[1], \ldots, r[b_G])$
        termR $\leftarrow \tilde{V}_i(r', r[b_G+1], \ldots, r[j-1], k, g_R[j-b_G+1], \ldots, g_R[b_G])$

      if g is an add gate **then**
        $F_{b_N + j}[g][k] \leftarrow$ termP $\cdot$ (termL + termR)
      else if g is a mult gate **then**
        $F_{b_N + j}[g][k] \leftarrow$ termP $\cdot$ termL $\cdot$ termR

    for $k \in \{-1,0,1\}$ **do**
      $F_{b_N + j}[k] \leftarrow \sum_{g \in \{0,1\}^{b_G}} F_{b_N + j}[g][k]$
    // Use Lagrange interpolation to compute coefficients of $F_{b_N + j}$ and send them to verifier
    SendToVerifier($F_{b_N + j}$, 2)
    $r[j] \leftarrow$ ReceiveFromVerifier() // see line 19 of Figure 9

  $r_0 \leftarrow (r[1], \ldots, r[b_G])$ // notation
  $r_1 \leftarrow (r[b_G+1], \ldots, r[2b_G])$ // notation

  for $\ell = \{0, \ldots, b_G\}$, $H[\ell] \leftarrow \tilde{V}_i(r', (r_1 - r_0) \cdot \ell + r_0)$
  // Use Lagrange interpolation to compute coefficients of H and send them to $\mathcal{V}$
  SendToVerifier(H, b_G)

  return (r', r_0, r_1)

FIGURE 12—$\mathcal{P}$ pseudocode in Giraffe for the layer-i sum-check invocation.

1: **function** SUMCHECKP(ArithCircuit c, layer i, q'_{i-1}, q_{i-1})
2: **for** $j = 1, \ldots, 2b_G$ **do**
3: // In these rounds, prover sends degree-2 polynomial F_j. Does this by computing and sending $F_j(0), F_j(1), F_j(2)$.
4:
5: **for** all $\sigma \in \{0,1\}^{b_N}$ and all $g \in \{0,1\}^{b_G}$ and $k \in \{0,1,2\}$ **do**
6: $s \leftarrow (g, g_L, g_R)$ // g_L, g_R are labels of g's layer-i inputs in subcircuit
7: $u_k \leftarrow (q_{i-1}[1], \ldots, q_{i-1}[b_G], r[1], \ldots, r[j-1], k)$
8: termP $\leftarrow \widetilde{\text{eq}}(q'_{i-1}, \sigma) \cdot \prod_{\ell=1}^{b_G+j} \chi_{s[\ell]}(u_k[\ell])$
9:
10: **if** $j \leq b_G$ **then**
11: termL $\leftarrow \tilde{V}_i(\sigma, r[1], \ldots, r[j-1], k, g_L[j+1], \ldots, g_L[b_G])$
12: termR $\leftarrow V_i(\sigma, g_R)$ // $V_i = \tilde{V}_i$ on gate labels
13: **else** // $b_G < j \leq 2b_G$
14: termL $\leftarrow \tilde{V}_i(\sigma, r[1], \ldots, r[b_G])$
15: termR $\leftarrow \tilde{V}_i(\sigma, r[b_G+1], \ldots, r[j-1], k, g_R[j-b_G+1], \ldots, g_R[b_G])$
16:
17: **if** g is an add gate **then** $F_j[\sigma, g][k] \leftarrow \text{termP} \cdot (\text{termL} + \text{termR})$
18: **else if** g is a mult gate **then** $F_j[\sigma, g][k] \leftarrow \text{termP} \cdot \text{termL} \cdot \text{termR}$
19:
20: **for** $k \in \{0,1,2\}$ **do**
21: $F_j[k] \leftarrow \sum_{\sigma \in \{0,1\}^{b_N}} \sum_{g \in \{0,1\}^{b_G}} F_j[\sigma, g][k]$
22: SendToVerifier(F_j, 2)
23: $r[j] \leftarrow$ ReceiveFromVerifier() // see line 19 of Figure 9
24:
25: $r_0 \leftarrow (r[1], \ldots, r[b_G])$ // notation
26: $r_1 \leftarrow (r[b_G+1], \ldots, r[2b_G])$ // notation
27:
28: **for** $j = 1, \ldots, b_N$ **do**
29: // In these rounds, prover sends degree-3 polynomial F_{2b_G+j}, so computes $F_{2b_G+j}(0), \ldots, F_{2b_G+j}(3)$
30:
31: **for** all $\sigma \in \{0,1\}^{b_N-j}$ and $k \in \{0,1,2,3\}$ **do**
32: termP $\leftarrow \widetilde{\text{eq}}(q'_{i-1}, r'[1], \ldots, r'[j-1], k, \sigma[1], \ldots, \sigma[b_N - j])$
33: termL $\leftarrow \tilde{V}_i(r'[1], \ldots, r'[j-1], k, \sigma[1], \ldots, \sigma[b_N - j], r_0)$
34: termR $\leftarrow \tilde{V}_i(r'[1], \ldots, r'[j-1], k, \sigma[1], \ldots, \sigma[b_N - j], r_1)$
35:
36: // See text for computation of $\tilde{\text{add}}(q_{i-1}, r_0, r_1)$ and $\tilde{\text{mult}}(q_{i-1}, r_0, r_1)$
37: $F_{2b_G+j}[\sigma][k] \leftarrow \text{termP} \cdot \big(\tilde{\text{add}}(q_{i-1}, r_0, r_1) \cdot (\text{termL} + \text{termR}) + \tilde{\text{mult}}(q_{i-1}, r_0, r_1) \cdot \text{termL} \cdot \text{termR}\big)$
38:
39: **for** $k \in \{0,1,2,3\}$ **do**
40: $F_{2b_G+j}[k] \leftarrow \sum_{\sigma \in \{0,1\}^{b_N-j}} F_{2b_G+j}[\sigma][k]$
41: SendToVerifier(F_{2b_G+j}, 3)
42: $r'[j] \leftarrow$ ReceiveFromVerifier() // see line 19 of Figure 9
43:
44: $r' \leftarrow (r'[1], \ldots, r'[b_N])$ // notation
45:
46: for $\ell = \{0, \ldots, b_G\}$, $H[\ell] \leftarrow \tilde{V}_i(r', (r_1 - r_0) \cdot \ell + r_0)$
47: SendToVerifier(H, b_G)
48:
49: **return** (r', r_0, r_1)

FIGURE 13—$\mathscr{P}$ pseudocode in T13 [66, §7] (with optimization [67]) for the layer-i sum-check invocation.

Theorem A.1. This is because the principal difference between the two protocols is the order in which variables are bound, and this change does not affect completeness, soundness, or $\mathscr{V}$'s runtime. Hence, the remainder of the proof is devoted to bounding $\mathscr{P}$'s runtime.

From inspection of Figure 10, the claim about $\mathscr{P}$'s runtime is true as long as each of the d calls to SUMCHECKP (cf. line 7 of Figure 10) can be implemented in time $O(G \cdot N + G \cdot \log G)$.

To show this, we begin by explaining how the first for loop of the SUMCHECKP function (lines 2–19 in Figure 12) can be implemented to run in time $O(G \cdot N)$. As in Section 3.1, we call this part of the protocol "phase 1".

We begin with the inner for loop of phase 1 (lines 5-13 in Figure 12). This loop has $4G \cdot N/2^j$ iterations. Lines 12 and 13 each take constant time per iteration, leading to a contribution of $O(\sum_{j=1}^{b_N} G \cdot N/2^j) = O(G \cdot N)$. Next, consider the computation of termL and termR in lines 9 and 10. Section 3.2 (see the "algorithm" paragraph) explained how to compute, in iteration j, all required values of termL and termR (across σ, g, k) in total time $O(G \cdot N/2^j)$, leading to another contribution of $O(\sum_{j=1}^{b_N} G \cdot N/2^j) = O(G \cdot N)$.

The bulk of our attention on the inner loop is on computing all required values of termP in line 8 in $O(G+N)$ time across all iterations $j = 1, \ldots, b_N$. This decomposes into (a) computing $\chi_g(q_{i-1})$ for all $g \in \{0,1\}^{b_G}$, and (b) computing $\widetilde{\text{eq}}(q'_{i-1}, r'[1], \ldots, r'[j-1], k, \sigma[1], \ldots, \sigma[b_N - j])$, where j ranges from 1 up to b_N, σ ranges over $\{0,1\}^{b_N - j}$, and k ranges over $\{-1, 0, 1, 2\}$. For (a), Apdx A (the "precomputation" paragraph) explained precisely how to compute the $\chi_g(q_{i-1})$ in time $O(G)$. To achieve (b) in time $O(N)$, consider the function $Z\colon \{0,1\}^{b_N} \to \mathbb{F}$ given by $Z(\cdot) = \widetilde{\text{eq}}(q'_{i-1}, \cdot)$. Observe that $\tilde{Z}(\cdot)$ and $\widetilde{\text{eq}}(q'_{i-1}, \cdot)$ are equal as formal polynomials, because they are both multilinear and agree at all Boolean inputs. Hence, task (b) is equivalent to evaluating $\tilde{Z}$ at all points of the form $(r'[1], \ldots, r'[j-1], k, \sigma[1], \ldots, \sigma[b_N - j])$.

$\mathscr{P}$ can achieve this in two steps. In the first step, prior to round $j = 1$, $\mathscr{P}$ evaluates Z on all Boolean inputs as follows. Observe that for any $\sigma \in \{0,1\}^{b_N}$, Equation (9) implies that $Z(\sigma) = \widetilde{\text{eq}}(q'_{i-1}, \sigma) = \chi_\sigma(q'_{i-1})$. $\mathscr{P}$ can again use the technique from Apdx A, this time to build an array containing $\chi_\sigma(q'_{i-1})$ for all $\sigma \in \{0,1\}^{b_N}$ in time $O(N)$.

In the second step, $\mathscr{P}$ evaluates $\tilde{Z}$ at all of the necessary points using the following method. Notice that when explaining how to efficiently compute termL and termR (§3.2), we more generally showed that the following is true. For any b-variate function $f\colon \{0,1\}^b \to \mathbb{F}$ and any $k \in \mathbb{F}$, given f's values on all Boolean inputs, one can, in total time $O(2^b)$, evaluate $\tilde{f}$ at all points of the form $(r'[1], \ldots, r'[j-1], k, \sigma[1], \ldots, \sigma[b-j])$, where j ranges from 1 up to b, and σ ranges over $\{0,1\}^{b-j}$. Hence, once $\mathscr{P}$ has evaluated Z on all Boolean inputs, $\mathscr{P}$ can apply the aforementioned result to $f = Z$ in order to evaluate $\tilde{Z}$ at the necessary points in time $O(2^{b_N}) = O(N)$.

In total, both steps of task (b) are dispatched in $O(N)$ time.

By inspection, lines 15-19 of Figure 12 can be dispatched in $\sum_{j=1}^{b_N} O(G \cdot N/2^j) = O(G \cdot N)$ time. Hence, phase 1 of the protocol can be dispatched in $O(G \cdot N)$ time in total.

The next cost to $\mathscr{P}$ to account for is the for loop with $2b_G$ iterations (cf. line 23); as in Section 3.1, we refer to this as "phase 2". This for loop can be dispatched in $O(G)$ time per iteration (hence, $O(G \log G)$ time in total), in a manner analogous to the prover implementation of CMT [31] (indeed, the pseudocode of Figure 12 already incorporates key insights from [31]).

In more detail, it is enough to show that for each iteration $j \in [1, 2b_G]$ of this for loop, all $3G$ iterations of the inner for loop in line 25 of Figure 12 can be dispatched in $O(G)$ total time, as this will yield a time bound of $O(G \cdot b_G) = O(G \log G)$. The dominant cost of these iterations is in computing the termP, termL, and termR values. The termL and termR values are handled via essentially the same method as in phase 1, requiring $O(\sum_{j=b_N+1}^{b_N+b_G} G/2^j) = O(G)$ time in total (across all $2b_G$ iterations j).

The bottleneck for phase 2 is the time required to compute termP (cf. line 29). The prover stores, at all iterations $j \in [1, 2b_G]$ of the outer loop, and for each gate $g \in \{0,1\}^{b_G}$ and $k = 0$, the value $U[g] = \widetilde{\text{eq}}(q'_{i-1}, r') \cdot \prod_{\ell=1}^{b_G+j-1} \chi_{s[\ell]}(u_k[\ell])$ (see lines 27 and 28 for the definition of s and u). Given these $U[g]$ quantities, in each iteration j of the outer loop, $\mathscr{P}$ can compute each value of termP and update $U[g]$ in constant time. This means that for each iteration j, all $O(G)$ values of termP can be computed in $O(G)$ total time, resulting in the claimed $O(G \log G)$ time bound across all $2b_G$ iterations of the outer loop.

The final cost to account for in $\mathscr{P}$'s work is evaluating the degree-b_G univariate polynomial $H = \tilde{V}_i(r', (r_1 - r_0) \cdot \ell + r_0)$ at $b_G + 1$ values of ℓ (see lines 52 and 53 of Figure 12). Consider the function $Q\colon \{0,1\}^{b_G} \to \mathbb{F}$, defined as $Q(\cdot) = \tilde{V}_i(r', \cdot)$. Then $\tilde{Q}(\cdot)$ and $\tilde{V}_i(r', \cdot)$ are equal as formal polynomials, because the right- and left-hand sides are multilinear polynomials that agree at all Boolean inputs.

Hence, $\mathscr{P}$ must compute $\tilde{Q}((r_1 - r_0) \cdot \ell + r_0)$ for $\ell \in \{0, \ldots, b_G\}$. For this purpose, we use the following result, which is a variant of the one given earlier (in reference to task (b)): given the evaluations of a b_G-variate function Q on all Boolean inputs, one can evaluate $\tilde{Q}$ at any point in time $O(2^{b_G}) = O(G)$. This follows from again applying the technique from Section 3.2 used to compute all of the termL and termR values (and is described in Section 3.3, the paragraph on multilinear extensions of I/O).

To get the evaluations of $Q(\cdot)$ on all Boolean inputs, we need $\tilde{V}_i(r', \sigma)$ for $\sigma \in \{0,1\}^{b_G}$. These evaluations can be computed in time $O(N \cdot G)$, again using the Section 3.2 technique. Then, we apply the previous paragraph to the $b_G + 1$ points $\{(r_1 - r_0) \cdot \ell + r_0 \mid \ell = 0, \ldots, b_G\}$, yielding additional computational cost of $O(G \cdot b_G)$.

In summary, lines 52 and 53 of Figure 12 together can be dispatched in time $O(N \cdot G + G \cdot \log G)$. □

B.1 Recursive expression for $\tilde{V}_i$

The Equation (5) recurrence in Section 3.2 is derived as follows:

$$\begin{aligned}
\tilde{V}_i\left(r'[1..j], \sigma, h\right) &= \sum_{s \in \{0,1\}^{b_N + b_G}} V_i(s) \cdot \chi_s\left(r'[1..j], \sigma, h\right) \\
&= \sum_{s \in \{0,1\}^{b_N + b_G} : s_j = 0} V_i(s) \cdot \chi_s\left(r'[1..j], \sigma, h\right) \\
&\quad + \sum_{s \in \{0,1\}^{b_N + b_G} : s_j = 1} V_i(s) \cdot \chi_s\left(r'[1..j], \sigma, h\right) \\
&= \left(1 - r'[j]\right) \cdot \tilde{V}_i\left(r'[1..j-1], 0, \sigma, h\right) \\
&\quad + r'[j] \cdot \tilde{V}_i\left(r'[1..j-1], 1, \sigma, h\right).
\end{aligned}$$

The first and last equalities apply Equation (6).

B.2 Other implementation considerations

The choice of values $k \in \{-1, 0, 1, 2\}$. In phase 1, Giraffe's $\mathscr{P}$ evaluates $F_j(k), k \in \{-1, 0, 1, 2\}$. This is a small optimization compared to, e.g., $k \in \{0, 1, 2, 3\}$. Recall from Section 3.2 that for $k = -1$, $\text{termL}_{j,n,g_L,-1} = 2 \cdot \text{termL}_{j,n,g_L,0} + (-1) \cdot \text{termL}_{j,n,g_L,1}$. Multiplication by 2 and by -1 can both be implemented as an addition rather than a multiplication, while $k = 3$ requires either two additions or a multiplication. A further slight optimization arises in $\mathscr{P}$'s work interpolating F_j: interpolating a third-degree polynomial for evaluations at the chosen points allows a few more multiplications to be traded for additions. In phase 2, Giraffe uses $k \in \{-1, 0, 1\}$ (Fig. 12) for the same reason.

$\mathscr{V}$'s precomputation hardware. $\mathscr{V}$ implements the dynamic programming algorithm of Figure 11 using an approach similar to the one described in Section 3.2. In brief, the access pattern of the algorithm is read one, write two, read one, and so forth. $\mathscr{V}$ instantiates two multipliers, one for each of the products in the innermost loop of Figure 11, and uses a variant of the RWSR design to store operands and results.

C Cost model

Figure 14 presents a simplified cost model for Giraffe's operating cost (energy), manufacturing cost (chip area), and performance (delay, i.e., inverse throughput). Roughly speaking, energy captures the number of operations executed, area corresponds to parallelism, and throughput represents the time spent on the critical path of execution.

Both $\mathscr{P}$ and $\mathscr{V}$ are designed to allow the designer to trade chip area for throughput. Section 3.2 describes one such tradeoff; Giraffe applies similar techniques in other parts of both $\mathscr{P}$ and $\mathscr{V}$. In addition, Giraffe's protocol requires computations expressed as layered arithmetic circuits (§2), and as with prior work [70, §3.2], Giraffe can take advantage of this requirement using pipelining. In this arrangement, $\mathscr{P}$ and $\mathscr{V}$ comprise a number of submodules, all running in parallel and executing different instances of the proof protocol.

To control area and throughput, Giraffe's $\mathscr{P}$ and $\mathscr{V}$ designs each have several parameters. For $\mathscr{V}$, the parameters are $n_{\mathscr{V},\text{io}}$, the chip area dedicated to computing the multilinear extension of inputs and outputs; and $n_{\mathscr{V},\text{sc}}$, the number of sumcheck instances $\mathscr{V}$ executes simultaneously. For $\mathscr{P}$, the parameters are $n_{\mathscr{P},\text{pl}}$, the number of in-flight computations in the pipeline; $n_{\mathscr{P},\text{sc}}$, the number of sumcheck instances $\mathscr{P}$ executes simultaneously; $n_{\mathscr{P},\text{ea}}$, $\mathscr{P}$'s parallelism in the early rounds of the sumcheck (§3.2); and $n_{\mathscr{P},\tilde{\text{V}}}$, $\mathscr{P}$'s parallelism in the final $\tilde{V}$ evaluation (Fig. 12, line 52).

cost	verifier	prover
energy		
compute	$G(6d+2C)E_{\text{mul,t}} + G(d+2C)E_{\text{add,t}}$	$d\left[G(6C+8\log G)E_{\text{mul,u}} + G(11N+8\log G)E_{\text{add,u}} + 4G(N+\log G)\left\langle E_{\text{g,u}}\right\rangle\right]$
$\mathscr{V}$-$\mathscr{P}$ *tx*	$(d(2\log G+\log N)+GN)E_{\text{tx,t}}$	$(d(7\log G+4\log N)+NG)E_{\text{tx,u}}$
store	—	$dNG\cdot E_{\text{sto,u}}$
PRNG	$d(2\log G+\log N)E_{\text{prng,t}}$	—
$\mathscr{V}$ *I/O*	$2NG\cdot E_{\text{io,t}}$	—
area		
compute	$n_{\mathscr{V},\text{sc}}\left(4A_{\text{mul,t}}+3A_{\text{add,t}}\right)+2n_{\mathscr{V},\text{io}}\left(2A_{\text{mul,t}}+A_{\text{add,t}}\right)$	$n_{\mathscr{P},\text{sc}}\left[\left(4Gn_{\mathscr{P},\text{ea}}+\frac{N}{2}+2n_{\mathscr{P},\tilde{\text{V}}}\log G\right)A_{\text{mul,u}}+\left(4Gn_{\mathscr{P},\text{ea}}+\frac{N}{2}+n_{\mathscr{P},\tilde{\text{V}}}\log G\right)A_{\text{add,u}}\right]$
$\mathscr{V}$-$\mathscr{P}$ *tx*	$(d(2\log G+\log N)+GN)A_{\text{tx,t}}$	$(d(7\log G+4\log N)+NG)A_{\text{tx,u}}$
store	—	$dNGn_{\mathscr{P},\text{pl}}\cdot A_{\text{sto,u}}$
PRNG	$d(2\log G+\log N)A_{\text{prng,t}}$	—
$\mathscr{V}$ *I/O*	$2NG\cdot A_{\text{io,t}}$	—
delay: Giraffe's overall throughput is $1/\max(\mathscr{V}\text{ delay}, \mathscr{P}\text{ delay})$; the expressions for $\mathscr{V}$ and $\mathscr{P}$ delay are given immediately below:	$\max\left(\frac{dG}{n_{\mathscr{V},\text{sc}}}\left(3\lambda_{\text{mul,t}}+\lambda_{\text{add,t}}\right),\ \frac{CG}{n_{\mathscr{V},\text{io}}}\left(\lambda_{\text{mul,t}}+\lambda_{\text{add,t}}\right)\right)$	$\frac{d}{n_{\mathscr{P},\text{sc}}}\left[\left(\frac{3C}{n_{\mathscr{P},\text{ea}}}+\frac{G}{n_{\mathscr{P},\tilde{\text{V}}}}\right)\left(\lambda_{\text{mul,u}}+\lambda_{\text{add,u}}\right)\right]$

$n_{\mathscr{V},\text{io}}$: $\mathscr{V}$ parameter; trades area vs I/O delay
$n_{\mathscr{V},\text{sc}}$: $\mathscr{V}$ parameter; trades area vs sumcheck delay
$\left\langle E_{\text{g,u}}\right\rangle$: mean per-gate energy of $\mathscr{C}$, untrusted
$n_{\mathscr{P},\text{pl}}$: $\mathscr{P}$ parameter; # in-flight runs
$n_{\mathscr{P},\text{sc}}$: $\mathscr{P}$ parameter; trades area vs delay
d, G, N: depth, width, and number of copies of arithmetic circuit $\mathscr{C}$
$n_{\mathscr{P},\text{ea}}$: $\mathscr{P}$ parameter; parallelism in early rounds (§3.2)
$n_{\mathscr{P},\tilde{\text{V}}}$: $\mathscr{P}$ parameter; parallelism for final $\tilde{V}(z_3,\cdot)$ eval
$E_{\{\text{add,mul,tx,sto,prng,io}\},\{\text{t,u}\}}$: energy cost in {trusted, untrusted} technology node for {+, ×, $\mathscr{V}$-$\mathscr{P}$ interaction, store, PRNG, $\mathscr{V}$ I/O}
$A_{\{\text{add,mul,tx,sto,prng,io}\},\{\text{t,u}\}}$: area cost in {trusted, untrusted} technology node for {+, ×, $\mathscr{V}$-$\mathscr{P}$ interaction, store, PRNG, $\mathscr{V}$ I/O}
$\lambda_{\{\text{add,mul}\},\{\text{t,u}\}}$: delay in {trusted, untrusted} technology node for {+, ×}

FIGURE 14—$\mathscr{V}$ and $\mathscr{P}$ costs as a function of $\mathscr{C}$ parameters and technology nodes (simplified model; low-order terms discarded). We assume $|x| = |y| = N \cdot G$. Energy and area constants for interaction, store, PRNG, and I/O indicate costs for a single element of $\mathbb{F}_p$. $\mathscr{V}$-$\mathscr{P}$ tx is the cost of interaction between $\mathscr{V}$ and $\mathscr{P}$; $\mathscr{V}$ I/O is the cost for the operator to communicate with Giraffe. For $\mathscr{P}$, store is the cost of buffering pipelined computations. Transmit, store, and PRNG occur in parallel with execution, so their delay is not included under the assumption that the corresponding circuits execute quickly enough.