Verifiable Computation with Massively Parallel Interactive Proofs

Justin Thaler*, Mike Roberts*, Michael Mitzenmacher*, and Hanspeter Pfister*
*Harvard University, School of Engineering and Applied Sciences

***Abstract*— As the cloud computing paradigm has gained prominence, the need for *verifiable computation* has grown increasingly urgent. Protocols for verifiable computation enable a weak client to outsource difficult computations to a powerful, but untrusted server, in a way that provides the client with a *guarantee* that the server performed the requested computations correctly. By design, these protocols impose a minimal computational burden on the client, but they require the server to perform a very large amount of extra bookkeeping to enable a client to easily verify the results. Verifiable computation has thus remained a theoretical curiosity, and protocols for it have not been implemented in real cloud computing systems.**

In this paper, we assess the potential of parallel processing to help make practical verification a reality, identifying abundant data parallelism in a state-of-the-art general-purpose protocol for verifiable computation. We implement this protocol on the GPU, obtaining 40-120× server-side speedups relative to a state-of-the-art sequential implementation. For benchmark problems, our implementation thereby reduces the slowdown of the server to within factors of 100-500× relative to the original computations requested by the client. Furthermore, we reduce the already small runtime of the client by 100×. Our results demonstrate the immediate practicality of using GPUs for verifiable computation, and more generally, that protocols for verifiable computation have become sufficiently mature to deploy in real cloud computing systems.

I. Introduction

A potential problem in outsourcing work to commercial cloud computing services is trust. If we store a large dataset with a server, and ask the server to perform a computation on that dataset – for example, to compute the eigenvalues of a large graph – how can we know the computation was performed correctly? We don't want to compute the result ourselves, and we might not even be able to store all the data locally. Despite these constraints, we would like the server to both provide us with the answer and convince us the answer is correct.

Protocols for *verifiable computation* offer a possible solution. The primary goal is to enable the client to obtain results with a guarantee of correctness from the server much more efficiently than performing the computations herself. Another important goal is to enable the server to provide guarantees of correctness *almost* as efficiently as providing results without such guarantees.

Interactive proofs are protocols for establishing guarantees of correctness between a client and server. Although they have been studied in the theory community for decades, there has been no significant effort to implement or deploy such proof systems until very recently. However, a recent line of work substantially advanced the practicality of these techniques. In particular, the work of Cormode, Mitzenmacher, and Thaler [3] demonstrates that: (1) a powerful general-purpose methodology due to Goldwasser, Kalai and Rothblum [5] approaches practicality; and (2) special-purpose protocols for a large class of streaming problems are already practical.

We take things a key step further by leveraging the parallelism offered by GPUs to obtain significant speedups relative to state-of-the-art implementations of [3]. We expect that practical verification protocols will have to be utilized eventually if cloud computing solutions are to be widely adopted for correctness-critical applications, and we expect that parallelization will necessarily be an important part of making this a reality.

Many of the insights of our GPU implementation would also apply to a multi-core CPU implementation. However, here we focus on GPUs because they are increasingly widespread, cost-effective, and power-efficient, and they offer potential speedups beyond those possible with commodity multi-core CPUs.

We obtain server-side speedups ranging from 40-120× for the general-purpose protocol due to Goldwasser *et al.* [5], and 20-50× speedups for related protocols targeted at specific streaming problems. Our general-purpose implementation reduces the server-side cost of providing results with a guarantee of correctness to within factors of 100-500× relative to a sequential algorithm without such guarantees. Similarly, our implementation of the special-purpose protocols reduces the server-side slowdown to within 10-100× relative to a sequential algorithm without such guarantees. Due to space limitations, we only present our general-purpose implementation; the full version of the paper is available on the arxiv [9], and our source code is available at [8].

The costs of obtaining correctness guarantees with techniques in this paper may already be considered modest in some correctness-critical applications. Consider e.g. Assured Cloud Computing for military contexts: a user may need integrity guarantees when computing in the presence of cyber attacks, or when coordinating

critical computations across a mixture of secure and insecure networks [1]. As another example, a hospital that outsources the processing of patients' electronic medical records may require guarantees that none of the records are dropped or corrupted. Even if every computation is not explicitly checked, the mere ability to check the computation could mitigate trust issues and stimulate users to adopt cloud computing solutions.

II. Background

What are interactive proofs? Interactive proofs (IPs) were introduced within the computer science theory community more than a quarter century ago. In any IP, there are two parties: a *prover* $\mathcal{P}$, and a *verifier* $\mathcal{V}$. $\mathcal{P}$ is typically considered to be computationally powerful, while $\mathcal{V}$ is considered to be computationally weak.

In an IP, $\mathcal{P}$ solves a problem using her (possibly vast) computational resources, and tells $\mathcal{V}$ the answer. $\mathcal{P}$ and $\mathcal{V}$ then have a conversation, which involves engaging in a randomized protocol involving the exchange of one or more messages. During this conversation, $\mathcal{P}$'s goal is to convince $\mathcal{V}$ that her answer is correct.

IPs naturally model the problem of a client (whom we model as $\mathcal{V}$) outsourcing computation to an untrusted server (whom we model as $\mathcal{P}$). That is, IPs provide a way for a client to hire a cloud computing service to store and process data, and to efficiently *check* the integrity of the results returned by the server. This is useful whenever the server is not a trusted entity, either because the server is deliberately deceptive, or is simply buggy or inept. We therefore interchange the terms server and prover where appropriate, and similarly for client and verifier.

Any IP must satisfy two properties. The first is that if $\mathcal{P}$ answers correctly and follows the prescribed protocol, then $\mathcal{P}$ will convince $\mathcal{V}$ to accept the provided answer. The second is a security guarantee, which says that if $\mathcal{P}$ is lying, then $\mathcal{V}$ must catch $\mathcal{P}$ and reject the provided answer with high probability. A trivial way to satisfy this property is to have $\mathcal{V}$ compute the answer to the problem herself, and accept only if her answer matches $\mathcal{P}$'s. But this defeats the purpose of having a prover. The goal of an interactive proof system is to allow $\mathcal{V}$ to check $\mathcal{P}$'s answer using resources considerably smaller than those required to solve the problem from scratch.

At first blush, this may appear difficult or even impossible to achieve. However, IPs have turned out to be surprisingly powerful. We direct the interested reader to [2, Chapter 8] for an excellent overview of this area.

How do interactive proofs work? At the highest level, many interactive proof methods work as follows. Suppose the goal is to compute a function f of input x.

First, the verifier makes a single streaming pass over the input x, during which she extracts a short *secret* s. This secret is actually a single (randomly chosen) symbol of an error-corrected encoding $\text{Enc}(x)$ of the input. To be clear, the secret does *not* depend on the problem being solved; for our protocol, it is not necessary that the problem be determined until *after* the secret is extracted.

Next, $\mathcal{P}$ and $\mathcal{V}$ engage in an extended conversation, during which $\mathcal{V}$ sends $\mathcal{P}$ various challenges, and $\mathcal{P}$ responds to the challenges. The challenges are all related to each other, and the verifier checks that the prover's responses to all challenges are *consistent*.

The challenges are chosen so that the prover's response to the first challenge must include a (claimed) value for the function of interest. Similarly, the prover's response to the last challenge must include a claim about what the value of the verifier's secret s should be. If all of $\mathcal{P}$'s responses are consistent, and the claimed value of s matches the true value of s, then the verifier is convinced that $\mathcal{P}$ followed the prescribed protocol and accepts. Otherwise, the verifier *knows* that $\mathcal{P}$ deviated at some point, and rejects. From this point of view, the purpose of all intermediate challenges is to guide the prover from a claim about $f(x)$ to a claim about the secret s, while maintaining $\mathcal{V}$'s control over $\mathcal{P}$.

Intuitively, what gives the verifier surprising power to detect deviations is the error-correcting properties of $\text{Enc}(x)$. Any good error-correcting code satisfies the property that if two strings x and x' differ in even one location, then $\text{Enc}(x)$ and $\text{Enc}(x')$ differ in many locations. In the same way, interactive proofs ensure that if $\mathcal{P}$ flips even a single bit of a single message, then $\mathcal{P}$ either has to make an inconsistent claim at some later point, or else has to lie almost everywhere in her final claim about the value of the secret s. Thus, if the prover deviates from the prescribed protocol *even once*, the verifier will detect this with high probability.

Previous work. Despite their power, IPs have had very little influence on real systems where integrity guarantees on outsourced computation would be useful. There appears to have been a folklore belief that these methods are impractical [7]. Nonetheless, a recent line of work has made substantial progress in advancing the practicality of these techniques. Goldwasser *et al.* [5] described a powerful general-purpose protocol (henceforth referred to as the GKR protocol) that achieves a polynomial-time prover and nearly linear-time verifier for a large class of computations. Very recently, Cormode, Mitzenmacher, and Thaler [3] showed how to significantly speed up the prover in the GKR protocol [5], and demonstrated experimentally that their implementation approaches practicality. Even with their optimizations, the bottleneck in the implementation of [3] is the prover's runtime, with all other costs (such as verifier space and runtime) being extremely low.

A related line of work has looked at protocols for specific *streaming* problems. Here, the goal is not just to

save the verifier time, but also to save the verifier *space*. This is motivated by settings where the client does not even have space to store a local copy of the input, and thus uses the cloud to both store and process the data. We omit further discussion of this direction due to space constraints, but stress that all protocols we consider work even in this restricted setting.

Also relevant is work by Setty *et al.* [7], who implemented a protocol for verifiable computation due to Ishai *et al.* [6]. While [7] represents a clear advance in the practicality of verification protocols, the implementation of the asymptotically more efficient GKR protocol described in both this paper and in [3] has several advantages in comparison. For example, the GKR implementation saves space and time for the verifier even when outsourcing a single computation, while [7] saves time for the verifier only when batching together several dozen computations and amortizing the verifier's cost over the batch. Our results also indicate that that the prover in the sequential implementation of [3] based on the GKR protocol runs significantly faster than the prover in the implementation of [7], at least for the benchmark problem of matrix multiplication.

III. Parallelizing the GKR protocol

In this section, we give an overview of the methods implemented in this paper. Due to their highly technical nature, we seek only to convey a high-level description of the protocols relevant to this paper, and deliberately avoid rigorous definitions or theorems. We direct the interested reader to prior work for further details [3].

Overview of GKR protocol. $\mathcal{P}$ and $\mathcal{V}$ first agree on a layered arithmetic circuit of fan-in two over a finite field $\mathbb{F}$ computing the function of interest. An arithmetic circuit is just like a boolean circuit, except the inputs are elements of $\mathbb{F}$ rather than boolean values, and the gates perform addition and multiplication over the field $\mathbb{F}$, rather than computing AND, OR, and NOT operations.

Suppose the output layer of the circuit is layer d, and the input layer is layer 0. The protocol of [5] proceeds in iterations, with one iteration for each layer of the circuit. The first iteration follows the general outline described in Section II, with $\mathcal{V}$ guiding $\mathcal{P}$ from a claim about the output of the circuit to a claim about a secret s, via a sequence of challenges and responses. The challenges sent by $\mathcal{V}$ are simply random coins, which are interpreted as random points in the finite field $\mathbb{F}$. The prescribed responses of $\mathcal{P}$ are polynomials, where each prescribed polynomial depends on the preceding challenge.

However, unlike in Section II, the secret s is not a symbol in an error-corrected encoding of *the input*, but rather a symbol in an error-corrected encoding of the *gate values* at layer $d-1$. Unfortunately, $\mathcal{V}$ cannot compute this secret s on her own. Doing so would

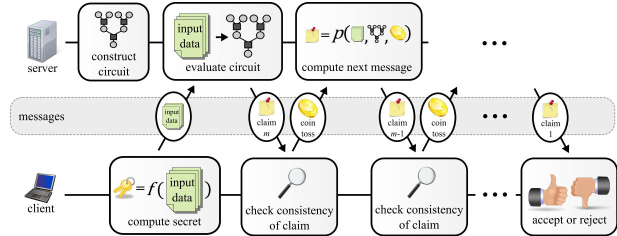

Fig. 1: High-level depiction of the GKR protocol.

require evaluating all previous layers of the circuit, and the whole point of outsourcing is to avoid this. So $\mathcal{V}$ has $\mathcal{P}$ *tell her* what s should be. But now $\mathcal{V}$ has to make sure that $\mathcal{P}$ is not lying about s.

This is what the second iteration accomplishes, with $\mathcal{V}$ guiding $\mathcal{P}$ from a claim about s, to the claim about a new secret s', which is a symbol in an encoding of the gate values at layer $d-2$. This continues until we get to the input layer. At this point, the secret is actually a symbol in an error-corrected encoding of *the input*, and $\mathcal{V}$ can compute this secret in advance from the input easily on her own. Figure 1 illustrates the entirety of the GKR protocol at a very high level.

Parallelizing $\mathcal{P}$'s computation. In every one of $\mathcal{P}$'s responses in the GKR protocol, the prescribed message from $\mathcal{P}$ is defined via a large sum over roughly S^3 terms, where S is the size of the circuit, and so computing this sum naively would take $\Omega(S^3)$ time. Roughly speaking, Cormode *et al.* in [3] observe that each gate of the circuit contributes to only a single term of this sum, and thus this sum can be computed via a single pass over the relevant gates. The contribution of each gate to the sum can be computed in constant time, and each gate contributes to logarithmically many messages over the course of the protocol. Using these observations carefully reduces $\mathcal{P}$'s runtime from $\Omega(S^3)$, to $O(S \log S)$.

The same observation reveals that $\mathcal{P}$'s computation can be parallelized: each gate contributes *independently* to the sum in $\mathcal{P}$'s prescribed response. Hence $\mathcal{P}$ can compute the contribution of many gates in parallel, save them in a temporary array, and use a parallel reduction to sum the results. Figure 2 illustrates this process.

Parallelizing $\mathcal{V}$'s computation. The bulk of $\mathcal{V}$'s computation (by far) consists of computing her secret s from the error-corrected encoding of the input x. As observed in prior work [4], each symbol of the input contributes *independently* to s. Thus, similarly to $\mathcal{P}$, $\mathcal{V}$ can compute the contribution of many input symbols in parallel, and sum the results via a parallel reduction. Although $\mathcal{V}$ runs extremely quickly even in the sequential implementation of [3], parallelizing $\mathcal{V}$'s computation is an appealing goal, especially as GPUs and multi-cores become more common on personal computers and mobile devices.

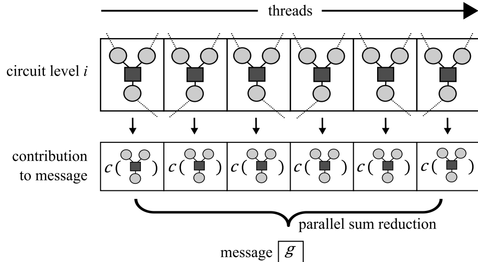

Fig. 2: Illustration of parallel computation of the server's message to the client in the GKR protocol.

Implementation Challenges. The primary bottleneck in the scalability of our parallel implementation is the memory-intensive nature of the GKR protocol: for a circuit of size S, the prover in the GKR protocol has to store all S gates explicitly, because she needs the values of these gates to compute her prescribed messages. This means that even "small" circuits with tens of millions of gates cannot fit in device memory.

We succeeded in mitigating this issue by keeping the circuit in host memory, and only copying information to the device when it is needed. This is possible because the GKR protocol satisfies the following critical property: *any iteration of the protocol involves only two layers of the circuit at a time.* In the ith iteration, the verifier guides the prover from a claim about gate values at layer $d-i$ to a claim about gate values at layer $d-i-1$. Gates at higher or lower layers do not affect the prescribed responses within iteration i.

Thus, only two layers of the circuit need to reside in device memory at a time. Although host-to-device copies are expensive, this approach is viable because copying only needs to happen once per layer. Since there are several dozen messages to be computed for each layer, this ensures that the copying from host to device can be amortized highly effectively over many messages.

IV. Evaluation

In this section we describe our results for two benchmark problems: matrix multiplication (denoted MatMult) and computing the sample variance of a data stream (denoted F_2). These two problems are representative of two important yet qualitatively distinct kinds of computation. F_2 is a well-studied data aggregation problem in the streaming literature; due to its low complexity, we could evaluate our implementation on rather large inputs. In contrast, we chose matrix multiplication as a benchmark application requiring superlinear time to solve. The arxiv version considers additional problems.

With one exception, we performed our experiments on an Intel Xeon 3 GHz workstation with 16 GB of host memory, and an NVIDIA GeForce GTX 480 GPU with 1.5 GB of device memory. By running on a NVIDIA Tesla C2070 GPU with 6 GBs of device memory, we were able to push to 256×256 matrices for MatMult, as reported in Table I. We implemented all our GPU code in CUDA and Thrust with all compiler optimizations turned on. We stress that all reported costs do count the time taken to copy data between host and device, and that all reported speedups are relative to the *sequential* (i.e. single-threaded) implementation of [3].

Figure 3 demonstrates the performance of our GPU-based implementation of the GKR protocol. Table I also gives a succinct summary of our results, showing the costs for the largest instance of each problem we ran on. We consider the main takeaways of our experiments to be the following.

Server-side speedup obtained by GPU computing. Compared to the sequential implementation of [3], our GPU-based server implementation ran over $100\times$ faster for the F_2 circuit and about $40\times$ faster for MatMult. For F_2, we need to look at large inputs to see the asymptotic behavior of the parallel prover's runtime. Due to the log-log scale in Figure 3, the curves for both the sequential and parallel implementations are asymptotically linear, and the $100\times$ speedup obtained by our GPU-based implementation manifests as an additive gap between the two curves. The explanation for this is simple: there is considerable overhead relative to the total computation time in parallelizing the computation at small inputs, but this overhead is more effectively amortized as the input size grows.

In contrast, notice that for MatMult the slope of the curve for the parallel prover remains significantly smaller than that of the sequential prover. This is because our GPU-based implementation ran out of device memory well before the overhead in parallelizing the prover's computation became negligible. We believe the speedup for MatMult would be higher than the $40\times$ speedup observed if we were able to run on larger inputs.

Could a parallel verifiable program be faster than a sequential unverifiable one? The first step of the prover's computation in the GKR protocol is to evaluate the circuit. In theory this can be done efficiently in parallel, but in practice we observe that the time it takes to copy the circuit to the device exceeds the time it takes to evaluate the circuit sequentially. This observation suggests that on the current generation of GPUs, no prover could run *faster* than a sequential *unverifiable* algorithm. This applies not just to the GKR protocol, but to any protocol that uses a circuit representation of the computation (which is a standard technique in the theory literature [6], [7]). Nonetheless, we can certainly hope to obtain a GPU-based implementation that is *competitive* with sequential unverifiable algorithms.

Server-side slowdown relative to unverifiable sequen-

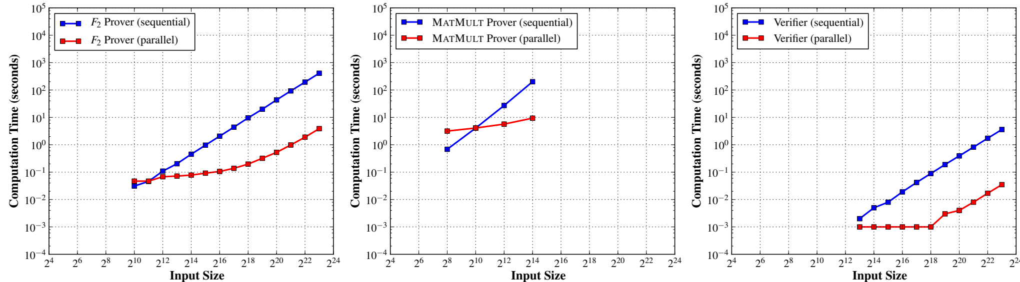

Fig. 3: Comparison of $\mathcal{P}$ and $\mathcal{V}$ runtimes for a sequential implementation of GKR [3] and our GPU implementation.

Problem	Input Size (number of entries)	Circuit Size (number of gates)	GPU $\mathcal{P}$ Time (s)	Sequential $\mathcal{P}$ Time (s)	Circuit Evaluation Time (s)	GPU $\mathcal{V}$ Time (s)	Sequential $\mathcal{V}$ Time (s)	Unverified Algorithm Time (s)
F_2	8.4 million	25.2 million	3.7	424.6	0.1	0.035	3.600	0.028
MATMULT	65,536	42.3 million	39.6	1,658.0	0.9	0.003	0.045	0.080

TABLE I: Prover and Verifier runtimes in the GKR protocol.

tial algorithms. For F_2, the slowdown for the prover was roughly 130×. We stress that it is likely that we overestimate the slowdown resulting from our protocol, because we did not count the time it takes for the unverifiable implementation to compute the number of occurrences of each item i, that is, to *aggregate* the stream into its frequency vector representation. Instead, we simply generated the vector of frequencies at random (we did not count the generation time).

For MATMULT, our GPU-based server implementation ran roughly 500× slower than naive matrix-multiplication for 256×256 matrices. Moreover, experiments on larger matrices suggest this number is likely inflated due to cache effects from which the naive unverifiable algorithm benefited (running times for the naive algorithm are almost 100 times larger for 512×512 matrices). We therefore expect the slowdown of our implementation would fall to under 100× if we were to scale to larger matrices.

Client-side speedup obtained by GPU computing. The bulk of $\mathcal{V}$'s computation consists of evaluating a single symbol in an error-corrected encoding of the input; this computation is *independent* of the circuit being verified. For reasonably large inputs, our GPU-based client implementation performed this computation 100× faster than the sequential implementation of [3].

Client-side speedup relative to unverifiable sequential algorithms. Our matrix-multiplication results clearly demonstrate that for problems requiring super-linear time to solve, even the sequential implementation of [3] will save the client time compared to doing the computation locally. Indeed, the runtime of the client is dominated by the cost of evaluating a single symbol in an error-corrected encoding of the input, and this cost grows linearly with the input size. Even for relatively small matrices of size 256×256, the client in the implementation of [3] saved time. For matrices with tens of millions of entries, our results demonstrate that the client will still take just a few seconds, while performing the matrix multiplication computation would require orders of magnitude more time. Our results demonstrate that GPU computing can also be used to reduce the verifier's computation time by another 100×.

V. CONCLUSIONS

Practical verification protocols appear key for cloud computing solutions to be widely adopted for correctness-critical applications. Our primary contribution here is demonstrating the power of parallelization, and GPU computing in particular, to obtain robust speedups for some of the most promising protocols in this area. However, substantial work remains to make practical verification a reality. For example, the GKR protocol is rather inefficient for the prover non-arithmetic computations. Developing improved protocols for such problems remains an important direction for future work.

REFERENCES

[1] J. Applequist. New assured cloud computing center to be established at Illinois. May 2011.

[2] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2009.

[3] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *Proc. ITCS*, 2012.

[4] G. Cormode, J. Thaler, and K. Yi. Verifying computations with streaming interactive proofs. In *Proc. VLDB Endowment*, 2011.

[5] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *Proc. STOC*, pages 113–122, 2008.

[6] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *Proc. CCC*, pages 278–291, 2007.

[7] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *Proc. NDSS*, 2012.

[8] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Source code. http://people.seas.harvard.edu/~jthaler/TRMPcode.htm. 2012.

[9] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. *CoRR*, abs/1202.1350, 2012.