Spartan Parallel

Kunming Jiang

Sep 21, 2023

1 Introduction

This document serves as a record for bookkeeping for the implementation of `spartan_parallel`. Its contents stand closer to the Rust code of the repository than to the theory discussed in Srinath's paper.

The original Spartan describes a SNARK on R1CS, which is consisted of three matrices A, B, C of size $X \times Y$ and a length-Y variable (inputs + witnesses) vector z. The goal is to prove $Az \cdot Bz = Cz$.

The data-parallel version of the problem is consisted of P R1CS instances of the form $\{A_i, B_i, C_i\}$, where each A_i, B_i, C_i are $X \times Y$ matrices, X represents the number of constraints and Y represents the number of variables (inputs + witnesses). For each instance $\{A_i, B_i, C_i\}$, there are Q_i (input, witnesses) vectors of length Y that claim to be a correct execution of the instance. We call such vector a **proof**, denoted as $z_{i,0}, \dots z_{i,Q_i}$. The goal is to prove and verify the correctness all proofs and all instances using SNARK: specifically, a modified version of Spartan that can handle data-parallelism.

For simplicity, Spartan makes the following assumption:

1. All matrices A_i, B_i, C_i are square matrices. This means that for every instance, $X = Y$. However, we can relax this assumption to $X = O(Y)$.

   This assumption is used to simplify analysis of time and space complexity. There is no mechanical reason to it.

2. The number of inputs / outputs in any proof is exactly one less than the number of witnesses (we can always set unused inputs to be 0). This means that any z vector can be separated into two halves, where the first half contains the inputs / outputs plus the constant 1, and the second half contains all the witnesses.

   One can see that such setup leads to inefficiencies. This is especially wasteful in our context, as one will see later.

3. Number of constraints and variables (X and Y) are all powers of 2. Again, one can always append dummies.

We add to the list our own set of assumptions:

5. Number of instances and proofs (P and Q_i) are also powers of 2. Specifically, values of Q_i's can differ, but all of them must be powers of 2.

6. Number of proofs for each instance is in decreasing order, i.e. $\forall i \in [0, P), Q_i \geq Q_{i+1}$.

Define $Q_{\texttt{max}} \leftarrow \max_i Q_i$ and $Q_{\texttt{sum}} \leftarrow \sum_i Q_i$. We assume $Q_{\texttt{sum}} = O(Q_{\texttt{max}})$. Let $p = \log P$, $q_i = \log Q_i$, $q_{\texttt{max}} = \log Q_{\texttt{max}}$, $x = \log X$, and $y = \log Y$. We want the total runtime of the Prover ($\mathcal{P}$) to be $O(Q_{\texttt{sum}} \cdot X)$, and that of the Verifier ($\mathcal{V}$) to be $O(\log(P \cdot Q_{\texttt{max}} \cdot Y))$.

2 Spartan as a Non-Interactive Argument System

In contrast to how Spartan is presented in the paper, we here illustrate the Spartan procedure as a SNARK to highlight its correspondence to the code. In the following sections, the Prover $\mathcal{P}$ is the party that produces the proof (which simulates both the prover $\mathcal{P}_i$ and the verifier $\mathcal{V}_i$ of the interactive proof) and the Verifier $\mathcal{V}$ is the party verifying that proof. Furthermore, unless specified otherwise, all "polynomials" in the document refers to multilinear polynomials, and are represented as evaluations on a boolean hypercube of all variables. The **dense** format represents a polynomial of n variables as a length-2^n vector which include every point on the boolean hypercube, while the **sparse** format only records down the non-zero entries and their indexes. Polynomials are represented in dense form by default.

Spartan as a SNARK can be divided into three stages, each with several steps:

2.1 Stage 1: Commitment

This is the one-time setup of stage Spartan with only one step:

1. A third (trusted?) party converts the matrices A, B, C to multilinear polynomials in sparse format. It then commits to these polynomials.

2.2 Stage 2: Proof Generation

In this stage, a prover $\mathcal{P}$ is given the instances and the variable list, and needs to generate a proof that the inputs and witnesses satisfy the constraints. It does so by simulating an interactive proof between a prover $\mathcal{P}_i$ and verifier $\mathcal{V}_i$, and needs to perform the following steps:

2. $\mathcal{P}$ converts the witnesses into a multilinear polynomial $w(y)$ in dense form and commits.

3. $\mathcal{P}$ converts z into a multilinear polynomial in dense form.

4. $\mathcal{P}$ simulates $\mathcal{V}_i$ and produces a length-x random vector τ. It computes the polynomial $\widetilde{\mathrm{eq}}_\tau(x)$ in dense form.

5. $\mathcal{P}$ computes polynomials $Az(x)$, $Bz(x)$, $Cz(x)$ and express them in dense format.

6. $\mathcal{P}$ simulates sum-check 1 between $\mathcal{P}_i$ and $\mathcal{V}_i$. The goal is to prove

$$0 = \sum_x \widetilde{\mathrm{eq}}_\tau(x) \cdot (Az(x) \cdot Bz(x) - Cz(x))$$

   At the end of the sum-check, $\mathcal{P}$ obtains a length-x random vector r_x, as well as $Az(r_x)$, $Bz(r_x)$, $Cz(r_x)$, $\widetilde{\mathrm{eq}}_\tau(r_x)$, and

$$e_x = \widetilde{\mathrm{eq}}_\tau(r_x) \cdot (Az(r_x) \cdot Bz(r_x) - Cz(r_x))$$

7. $\mathcal{P}$ generates a proof that e_x is indeed $\widetilde{\mathrm{eq}}_\tau(r_x) \cdot (Az(r_x) \cdot Bz(r_x) - Cz(r_x))$, in zero-knowledge if necessary.

8. $\mathcal{P}$ simulates $\mathcal{V}_i$, produces 3 random values r_A, r_B, and r_C, and computes

$$T = r_A \cdot Az(r_x) + r_B \cdot Bz(r_x) + r_C \cdot Cz(r_x)$$

9. $\mathcal{P}$ computes polynomial $ABC(y)$ in dense form, defined as
$$ABC(y) = r_A \cdot A(r_x, y) + r_B \cdot B(r_x, y) + r_C \cdot C(r_x, y)$$

10. $\mathcal{P}$ also converts z into a polynomial $Z(y)$ in dense form.

11. $\mathcal{P}$ simulates sum-check 2 between $\mathcal{P}_i$ and $\mathcal{V}_i$. The goal is to prove
$$T = \sum_y ABC(y) \cdot Z(y)$$
At the end of the sum-check, $\mathcal{P}$ obtains a length-y random vector r_y, as well as $ABC(r_y)$, $Z(r_y)$, and
$$e_y = ABC(r_y) \cdot Z(r_y)$$
Note: As stated by assumption 2, the first half of Z are inputs and the second half of Z are witnesses, so $Z(r_y) = (1 - r_y[0]) \cdot w(r_y[1..]) + r_y[0] \cdot \widetilde{(io,\ 1)}(r_y[1..])$

12. $\mathcal{P}$ computes $w_{r_y} = w(r_y[1..])$ and produces a proof of the computation, presumably in zero-knowledge.

13. $\mathcal{P}$ generates a proof that e_y is indeed $ABC(r_y) \cdot Z(r_y)$, presumably in zero-knowledge.

Finally, $\mathcal{P}$ reveals the commitment for w, e_x, e_y, w_{r_y}, as well as proofs in step 7, 12, and 13.

2.3 Stage 3: Proof Verification

In this stage, a verifier $\mathcal{V}$ wants to verify the correctness of the proof generated by $\mathcal{P}$, which is consisted of the following steps:

14. $\mathcal{V}$ follows $\mathcal{P}$'s transcript and reproduces the length-x random vector τ.

15. $\mathcal{V}$ verifies the correctness of the procedure for sumcheck 1. During the verification, $\mathcal{V}$ reproduces r_x.

16. $\mathcal{V}$ computes $\widetilde{\text{eq}}_\tau(r_x)$ and verifies $e_x = \widetilde{\text{eq}}_\tau(r_x) \cdot (Az(r_x) \cdot Bz(r_x) - Cz(r_x))$, potentially in zero-knowledge.

17. $\mathcal{V}$ reproduces r_A, r_B, r_C and verifies the correctness of the procedure for sumcheck 2. During the verification, $\mathcal{V}$ reproduces r_y.

18. $\mathcal{V}$ verifies $w_{r_y} = w(r_y)$ (in zero-knowledge).

19. $\mathcal{V}$ converts the input into a multilinear polynomial $\widetilde{(\text{io, 1})}$ in sparse form and evaluates it on $r_y[1..]$.

20. $\mathcal{V}$ uses w_{r_y} and $\widetilde{(\text{io, 1})}(r_y[1..])$ to compute $Z(r_y)$, which is then used to verify the correctness of e_y.

Note that verifications of sum-checks and computations take $O(\log X)$ time (assume $Y = O(X)$). Thus, runtime of $\mathcal{V}$ largely depends on time to open commitment for w (step 18) and to list the number of non-zero inputs (step 19). When implementing data-parallelism to Spartan, these two factors become crucial.

3 Implement Spartan Using Data-Parallelism

We begin by modifying Spartan to support data-parallelism. The data-paralleled version very much resembles the original Spartan. We first present a naive implementation, then reason about how to improve the time and space compexity.

The naive `spartan_parallel` protocol is shown below, again in three stages.

3.1 Stage 1: Commitment

This is the one-time setup of stage Spartan with only one step:

1. A third (trusted?) party converts the matrices A_i, B_i, C_i to $P \times Q_{\texttt{max}} \times X \times Y$ multilinear polynomials in sparse format. It then commits to these polynomials.

3.2 Stage 2: Proof Generation

In this stage, a prover $\mathcal{P}$ is given the instances and the variable list, and needs to generate a proof that the inputs and witnesses satisfy the constraints. It does so by simulating an interactive proof between a prover $\mathcal{P}_i$ and verifier $\mathcal{V}_i$, and needs to perform the following steps:

2. $\mathcal{P}$ converts the witnesses into a multilinear polynomial $w(p, q, y)$ in dense form and commits.
3. $\mathcal{P}$ converts z_i into a multilinear polynomial $Z(p, q, y)$ in dense form.
4. $\mathcal{P}$ simulates $\mathcal{V}_i$ and produces a length-$(p+q+x)$ random vector τ. It computes the polynomial $\widetilde{\text{eq}}_\tau(p, q, x)$ in dense form.
5. $\mathcal{P}$ computes polynomials $Az(p, q, x)$, $Bz(p, q, x)$, $Cz(p, q, x)$ and express them in dense format.
6. $\mathcal{P}$ simulates sum-check 1 between $\mathcal{P}_i$ and $\mathcal{V}_i$. The goal is to prove

$$0 = \sum_{p,q,x} \widetilde{\text{eq}}_\tau(p, q, x) \cdot (Az(p, q, x) \cdot Bz(p, q, x) - Cz(p, q, x))$$

   At the end of the sum-check, $\mathcal{P}$ obtains a length-$(p+q+x)$ random vector (r_p, r_q, r_x), as well as $Az(r_p, r_q, r_x)$, $Bz(r_p, r_q, r_x)$, $Cz(r_p, r_q, r_x)$, $\widetilde{\text{eq}}_\tau(r_p, r_q, r_x)$, and

$$e_x = \widetilde{\text{eq}}_\tau(r_p, r_q, r_x) \cdot (Az(r_p, r_q, r_x) \cdot Bz(r_p, r_q, r_x) - Cz(r_p, r_q, r_x))$$

7. $\mathcal{P}$ generates a proof that e_x is indeed $\widetilde{\text{eq}}_\tau(r_p, r_q, r_x) \cdot (Az(r_p, r_q, r_x) \cdot Bz(r_p, r_q, r_x) - Cz(r_p, r_q, r_x))$, in zero-knowledge if necessary.
8. $\mathcal{P}$ simulates $\mathcal{V}_i$, produces 3 random values r_A, r_B, and r_C, and computes

$$T = r_A \cdot Az(r_p, r_q, r_x) + r_B \cdot Bz(r_p, r_q, r_x) + r_C \cdot Cz(r_p, r_q, r_x)$$

9. $\mathcal{P}$ computes polynomial $ABC_{r_x}(p, y)$ in dense form, defined as

$$ABC_{r_x}(p, y) = r_A \cdot A_{r_x}(p, y) + r_B \cdot B_{r_x}(p, y) + r_C \cdot C_{r_x}(p, y)$$

10. $\mathcal{P}$ also converts z into a polynomial $Z_{r_q}(p, y)$ in dense form.

11. Note that $f(y) = ABC_{r_x}(r_p, y) \cdot z_{r_q}(r_p, y)$ is not a multilinear polynomial, since it is quadratic to r_p. Instead, $\mathcal{P}$ computes $\widetilde{\text{eq}}_{r_p}(p)$ in dense form.

12. $\mathcal{P}$ simulates sum-check 2 between $\mathcal{P}_i$ and $\mathcal{V}_i$. The goal is to prove

$$T = \sum_{p^*, y} ABC_{r_x}(p^*, y) \cdot Z_{r_q}(p^*, y) \cdot \widetilde{\text{eq}}_{r_p}(p^*)$$

At the end of the sum-check, $\mathcal{P}$ obtains a length-$(p + y)$ random vector (r_p^*, r_y), as well as $ABC_{r_x}(r_p^*, r_y)$, $Z_{r_q}(r_p^*, r_y)$, $\widetilde{\text{eq}}_{r_p}(r_p^*)$, and

$$e_y = ABC_{r_x}(r_p^*, r_y) \cdot Z_{r_q}(r_p^*, r_y) \cdot \widetilde{\text{eq}}_{r_p}(r_p^*)$$

Note: Similary, as stated by assumption 2, the first half of z_i are inputs and the second half of z_i are witnesses, so $Z_{r_q}(r_p^*, r_y) = (1 - r_y[0]) \cdot w_{r_q}(r_p^*, r_y[1..]) + r_y[0] \cdot \widetilde{(io,\ 1)}_{r_q}(r_p^*, r_y[1..])$

13. $\mathcal{P}$ computes $w_{r_y} = w_{r_q}(r_p^*, r_y[1..])$ and produces a proof of the computation, presumably in zero-knowledge.

14. $\mathcal{P}$ generates a proof that e_y is indeed $ABC_{r_x}(r_p^*, r_y) \cdot Z_{r_q}(r_p^*, r_y)$, presumably in zero-knowledge.

Finally, $\mathcal{P}$ reveals the commitment for w, e_x, e_y, w_{r_y}, as well as proofs in step 7, 12, and 13.

3.3 Stage 3: Proof Verification

In this stage, a verifier $\mathcal{V}$ wants to verify the correctness of the proof generated by $\mathcal{P}$, which is consisted of the following steps:

14. $\mathcal{V}$ follows $\mathcal{P}$'s transcript and reproduces the length-$(p + q + x)$ random vector τ.

15. $\mathcal{V}$ verifies the correctness of the procedure for sumcheck 1. During the verification, $\mathcal{V}$ reproduces r_p, r_q, r_x.

16. $\mathcal{V}$ computes $\widetilde{\text{eq}}_\tau(r_p, r_q, r_x)$ and verifies $e_x = \widetilde{\text{eq}}_\tau(r_p, r_q, r_x) \cdot (Az(r_p, r_q, r_x) \cdot Bz(r_p, r_q, r_x) - Cz(r_p, r_q, r_x))$, potentially in zero-knowledge.

17. $\mathcal{V}$ reproduces r_A, r_B, r_C and verifies the correctness of the procedure for sumcheck 2. During the verification, $\mathcal{V}$ reproduces r_p^*, r_y.

18. $\mathcal{V}$ computes $\widetilde{\text{eq}}_{r_p}(r_p^*)$

19. $\mathcal{V}$ verifies $w_{r_y} = w(r_p^*, r_q, r_y)$ (in zero-knowledge).

20. $\mathcal{V}$ converts the input into a multilinear polynomial $\widetilde{(\text{io, } 1)}$ in sparse form and evaluates it on $(r_p^*, r_q, r_y[1..])$.

21. $\mathcal{V}$ uses w_{r_y} and $\widetilde{(\text{io, } 1)}(r_p^*, r_q, r_y[1..])$ to compute $Z_{r_q}(r_y)$, which is then used to verify the correctness of e_y.

Note that the runtime of the entire protocol is dominated by steps related to sum-check 1. This will be the focus of improvements later.

The proof of this protocol largely follows that of Spartan.

4 Identifying Runtime Overhead

Ideally, we want the time and space complexity for $\mathcal{P}$ to be $O(Q_{\texttt{sum}} \cdot X)$, where $Q_{\texttt{sum}} = \sum_i Q_i$ and $Y = O(X)$ by assumption 1. The time and space complexity for $\mathcal{V}$ should be on a logarithmic scale. We relax the constraints slightly so that $\mathcal{V}$ can run in $O(P \cdot Q_{\texttt{max}} \cdot X)$.

However, the naive protocol above fails to achieve the target runtime for $\mathcal{P}$. It also does not satisfy our ideal runtime for $\mathcal{V}$, albeit for a different reason. Below we analyze in detail overheads in the naive protocol that needs to be overcome.

4.1 Verifier Cost

The protocol largely achieves the verifier runtime of $O(\log(P \cdot Q_{\texttt{max}} \cdot Y))$. However, the two factors in Spartan are still in play here:

- In step 19: in Spartan, verifying the witness commitment takes $O(\sqrt{X})$. In `spartan_parallel`, this cost is $O(\sqrt{P \cdot Q_{\texttt{max}} \cdot X})$, which is considerably worse than our expectation. We later show how to improve this to $O(Q_{\texttt{sum}})$.

- In step 20: in Spartan, cost to compute $\widetilde{(\text{io}, 1)}$ is linear to the number non-zero inputs. This means that in `spartan_parallel`, even if every proof has $O(1)$ non-zero inputs, verifier cost will be at least $O(Q_{\texttt{sum}})$. However, this complexity is by design since $\mathcal{V}$ needs to be able to process every input anyways. We later show improve for the special case of `circ_blocks`.

4.2 Prover Cost

The prover's cost for naive `spartan_parallel` is $O(P \cdot Q_{\texttt{max}} \cdot X)$, and we want to reduce it to $O(Q_{\texttt{sum}} \cdot X)$. Here's all the steps that exceeds the target complexity:

- In step 2: the prover converts the witnesses into a polynomial whose dense format is represented as a $P \cdot Q_{\texttt{max}} \cdot X$ array. This exceeds the target time and space complexity. However, only $Q_{\texttt{sum}} \cdot X$ entries are non-zero, which provides ground for improvement.

- In step 4: producing $\widetilde{\text{eq}}_\tau$ also takes $O(P \cdot Q_{\texttt{max}} \cdot X)$ time and space. However, note that: a) $\widetilde{\text{eq}}_\tau$ is not sparse at all, although the Az, Bz, and Cz are, and b) any modification that breaks the pattern of $\widetilde{\text{eq}}$ polynomials (i.e. inserting or replacing zero-entries) would result in $\mathcal{V}$ unable to verify it in logarithmic time.

- In step 5: Az, Bz, Cz are $P \cdot Q_{\texttt{max}} \cdot X$ sparse polynomials.

- In step 6: Sum-check 1 takes $p + q_{\texttt{max}} + x$ rounds, which naively also requires $O(P \cdot Q_{\texttt{max}} \cdot X)$ runtime for $\mathcal{P}$.

- In step 10: compute Z_{r_q} requires computing Z and then binding it to r_q. Z is a $P \cdot Q_{\texttt{max}} \cdot X$ sparse polynomial and naive binding takes $O(P \cdot Q_{\texttt{max}} \cdot X)$ time.

Note: when calculating runtime complexity, what counts as an operation matters. These are the ways to define an operation: a change in iterator, any multiplication operation, or any array access. In general, the three measurements match, but we will in certain cases reason about each individual measurement.

5 A Compact Polynomial Representation

We start our improvement by redesigning the dense representation of multilinear polynomials. As described earlier, the dense form expresses a polynomial as its evaluation on the boolean hypercube of all its variables. In `spartan_parallel`, however, such form is excessive:

- As discussed earlier, polynomials like w, Az, Bz, Cz, and Z contains $p + q_{\texttt{max}} + x$ variables. The dense form takes in an array of size $P \cdot Q_{\texttt{max}} \cdot X$ but only $Q_{\texttt{sum}} \cdot X$ non-zero entries.
- Furthermore, the non-zero entries are concentrated together. In particular, for any given instance i, all entries between proof Q_i to $Q_{\texttt{max}}$ are going to be zero. This makes the sparse representation, which records down indices of non-zero entries, also undesirable.

The solution is to introduce the *compact form* to represent multilinear polynomial. Compact form is similar to dense form, but instead of listing the evaluations on the boolean hypercube in a single array, we store the values in a three-dimensional array with pointer redirection, with each dimension representing P, Q_i, and X. Pointer redirection allows us to have varying length across the same dimension, so we only need to allocate $Q_{\texttt{sum}} \cdot X$ space.

5.1 Operations on Compact Polynomial

Matrix multiplication on compact polynomial is simple: since every Q_i is known, they can be used to skip the zero parts of the dot product. Using this method, one can easily compute Az, Bz, and Cz in $O(Q_{\texttt{sum}} \cdot X)$ time.

Binding and summing over points on the boolean hypercube also follow the same idea, and can be performed in $O(Q_{\texttt{sum}} \cdot X)$ time.

5.2 Modifying Compact Polynomials

While evaluations on the boolean hypercube can is easy, without cares, any modification to the polynomial can result in $\mathcal{P}$ losing the sparsity information: during the sum-check, binding any of the P variables to a value r_p other than 0 or 1 would land $\mathcal{P}$ in an evaluation where Q_i's are no longer applicable: as a result, would force any future operations back to $O(P \cdot Q_{\texttt{max}} \cdot X)$ time.

To solve this problem, $\mathcal{P}$ always binds the X variables first in sum-check. This enables it, at least in the first x rounds, to sum over the polynomial on points where the P and Q variables are within the boolean hypercube. Thus, $\mathcal{P}$ can skip the zero section of the polynomial for the first x rounds, reducing the summing cost of Sum-Check 1 to the $O(\max(P \cdot Q_{\texttt{max}}, Q_{\texttt{sum}} \cdot X))$.

This idea can and needs to be further expanded. The motivation is that to produce Z_{r_q}, $\mathcal{P}$ needs to only bind the Q variables of the compact polynomial Z. If $\mathcal{P}$ starts binding from the beginning (most significant bits) of Q, then information regarding Q_i's are lost after the first round. However, if $\mathcal{P}$ starts from the end of (least significant bits) Q, then the information is preserved (if $Z(1,0,0)$ and $Z(1,0,1)$ are both 0, then $Z(1,0,r)$ is also 0).

To further optimize caching, we let the compact form stores the q variables *in reverse*, i.e. the cell $Z[p][q][x]$ stores $Z(p, q_{\texttt{rev}}, x)$, where $q_{\texttt{rev}}$ is q with bits reversed. This allows us to perform evaluation and binding of polynomials in the sum-check in the "right" order (i.e. from left to right), but generates the r_q list in reverse order. We call this reversed list $r_{q_{\texttt{rev}}}$. Note that as long as every polynomial (Az, Bz, Cz, Z, w, $\widetilde{(\text{io}, 1)}$) stores $q_{\texttt{rev}}$ instead of q, nothing needs to be changed in the protocol to accomodate $r_{q_{\texttt{rev}}}$.

Finally, a clarification to the previous paragraph: $Z[p][q][x]$ actually stores $Z(p, q_{\texttt{rev}} \cdot s, x)$, where $s = Q_{\texttt{max}}/Q_p$. This is because non-zero entries in $q_{\texttt{rev}}$ are no longer together, but rather always s

apart. Take an example of $Q_{\texttt{max}} = 16, Q_i = 4$. Non-zero entres of q are $(0000_2, 0001_2, 0010_2, 0011_2)$, but non-zero entries of $q_{\texttt{rev}}$ are $(0000_2, 1000_2 = 8, 0100_2 = 4, 1100_2 = 12)$. To store them compactly, we put $q_{\texttt{rev}} = 4$ in slot 1, $q_{\texttt{rev}} = 8$ in slot 2, etc. This mostly does not affect any operation, especially not binding.

Finally, we can derive the cost analysis for summing and binding a polynomial Z of compact form with $q_{\texttt{rev}}$:

1. Since every valid $Z[p][q_{\texttt{rev}}]$ vector is not sparse, binding one x variable before any p and $q_{\texttt{rev}}$ always cut the number of non-zero entries by half.

2. If we bind one $q_{\texttt{rev}}$ variable before any p variables (regardless of whether x variables are binded or not), then for any instance i, as long as there are more than one proof, the number of non-zero entries are cut by half. This means binding all $q_{\texttt{rev}}$ variables takes at least $O(P \cdot \log Q_{\texttt{max}})$ (or $O(P \cdot \log Q_{\texttt{max}} \cdot X)$) time. Since we assume $Q_{\texttt{sum}} = O(Q_{\texttt{max}})$, this is acceptable.

3. If we bind any p variable before finishing all $q_{\texttt{rev}}$ variables, compactness is lost. Fortunately none of the steps requires this process.

Using these properties, we can revise the Prover Cost:

- In step 5: Az, Bz, Cz are now $Q_{\texttt{sum}} \cdot X$ compact polynomials and computing them take $O(Q_{\texttt{sum}} \cdot X)$ time.

- In step 6: Sum-check 1 still takes $p + q_{\texttt{max}} + x$ rounds, but summing over polynomials now takes in total $O(Q_{\texttt{sum}} \cdot X)$ time. Since $\mathcal{P}$ binds in the order of $x \to q_{\texttt{rev}} \to p$, their respective binding costs are:
  - $O(Q_{\texttt{sum}} \cdot X)$ for the first x rounds
  - $O(P \cdot \log Q_{\texttt{max}})$ for the next q rounds
  - $O(P)$ for the last p rounds

  As a result, cost for Sum-check 1 excluding $\widetilde{\mathrm{eq}}_\tau$ is $O(Q_{\texttt{sum}} \cdot X)$.

- In step 10: compute Z_{r_q} requires computing Z and then binding it to $r_{q_{\texttt{rev}}}$. Z can be stored in compact form, taking $O(Q_{\texttt{sum}} \cdot X)$ space. Binding, as explained earlier, takes $O(P \cdot \log Q_{\texttt{max}} \cdot X)$ time.

This leaves us with the problem of witness commitment and $\widetilde{\mathrm{eq}}_\tau$ creation.

5.3 Committing Compact Polynomials

For polynomial commitment, Spartan expresses a polynomial expression $M(r)$ as

$$M(r_x, r_y) = \sum_{i,j::M(i,j)\neq 0} M(i,j) \cdot \widetilde{\mathrm{eq}}(i, r_x) \cdot \widetilde{\mathrm{eq}}(j, r_y)$$

where r_x is the first half of r and r_y is the second.

The evaluation $M(r)$ can then be treated as a SNARK of the dot product, thus $\mathcal{V}$, which is also the verifier of the commitment, only needs to pay the cost of dot product on one particular (r_x, r_y), which is $O(\sqrt{n})$, where n is the number of non-zero entries in M.

We first note that if M is in sparse polynomial form, then $\mathcal{P}$'s work is linear to the number of non-zero entries. This applies also to `spartan_parallel`, which means nothing needs to be altered to commit A, B, C matrices.

6 Other Improvements

6.1 Converting Inputs into Witnesses

6.2 Better Encoding of $\widetilde{\mathbf{eq}}_\tau$