GPU Acceleration for Boyar-Peralta Algorithm in High Dimensional Linear Layers

Xiao Huang[1], Chao Sun[2], Ge Wu[1] and Liquan Chen[1]

Southeast University, Nanjing, China
220245548@seu.edu.cn, gewu@seu.edu.cn, Lqchen@seu.edu.cn
East China Normal University, Shanghai, China
sunchaopku12345@gmail.com

Abstract. The design of lightweight cryptographic schemes necessitates efficient circuit implementations of linear layers. A commonly used metric for evaluating circuit efficiency is the number of XOR gates required. For a given linear transformation over $\mathbb{F}_2$, finding its circuit implementation with the minimum number of XOR gates is referred to as the Shortest Linear Program (SLP) problem. As a prominent heuristic, the Boyar-Peralta (BP) algorithm provides competitive results, and numerous subsequent studies have introduced further optimizations. However, the inefficient implementations of the BP-like algorithms significantly limit their applicability to high-dimensional and dense matrices, and effective solutions to these performance bottlenecks remain scarce.

In this paper, we address the performance bottlenecks of the BP algorithm and its variants (e.g., A1, A2 and BP-LSL) both theoretically and practically. We introduce the meet-in-the-middle (MITM) approach to reduce their theoretical time complexity. Practically, we re-design the implementation framework of these algorithms to fully leverage GPU parallelism, which significantly reduces the time spent on computing multiple XOR sums of bases, performing binary search, and updating distances. Meanwhile, we also propose a candidate pruning mechanism to reduce the computational burden on the GPU side and further improve the overall efficiency. Compared with the original implementations, we significantly accelerate the solving process for both normal and depth-constrained scenarios while maintaining the same solution quality.

Our experimental results demonstrate that, for a 26-dimensional matrix with density 0.5, our implementation requires only 9.44 seconds on an entry-level GTX 1650 GPU, achieving a speedup of over 6,200× compared with the current recursive, exhaustive, and one-threaded implementation. Using a high-performance RTX 4090 GPU, we extend the solvable boundary to 64-dimensional sparse matrices and 42-dimensional matrices with density 0.381. For 32-dimensional dense matrices, our accelerated A1 algorithm successfully establishes some new g-XOR records, including a 172-step g-XOR implementation for Whirlwind M0 and a 170-step implementation for Whirlwind M1. Besides, we also provide a 157-step g-XOR sequence for AES-InvMixColumn matrix with depth 5 through our BP-LSL implementation. Based on our accelerated BP implementation, we construct some MDS matrices with dimension 40 or more. For 4×4 blockwise circulant matrices of dimensions 40, 48, and 64, the minimum g-XOR sequence lengths we found are 110, 132, and 184.

Keywords: Shortest Linear Program, Boyar-Peralta algorithm, GPU acceleration, Meet-in-the-middle.

1 Introduction

In recent years, the design of lightweight symmetric cryptography has drawn widespread interest to meet the growing encryption demand for resource-constrained IoT devices. These algorithms must strike an optimal balance between security, hardware area, execution speed, and power consumption. Driven by the standardized AES [JV02], most lightweight schemes employ the Substitution-Permutation Network (SPN) architecture. Thus, a central design challenge for these schemes lies in constructing lightweight components, especially for optimizing S-boxes [LP07, Saa12, ZBRL15, JPST17, JBKK25] and linear layers (or P-boxes) [SKOP15, BKL16, KLSW17, ZWS18, LSL+19, HDRM23].

Regarding the design of linear layers, implementation complexity is commonly evaluated by the number of XOR gates required in the corresponding circuit [KPPY14], as fewer gates directly lead to a smaller circuit area. Beyond direct measures based on the matrices' Hamming weights, more refined metrics have been proposed, namely s-XOR and g-XOR. The s-XOR metric restricts operations to sequential updates of the form $\mathbf{x}_i \leftarrow \mathbf{x}_i \oplus \mathbf{x}_j$, whereas g-XOR allows the introduction and reuse of an unlimited number of intermediate bases. Under the g-XOR metric, finding a circuit implementation with the minimum number of XOR gates is formulated as the Shortest Linear Program (SLP) problem [BMP08].

However, for a specific linear transformation over $\mathbb{F}_2$, the SLP problem has been proven to be NP-hard [BMP08], and exact SAT-based solvers [FSK10, ZH23] are often computationally infeasible for large-scale matrices. Therefore, some heuristic algorithms have been proposed to provide near-optimal results. The most representative among them are the Paar algorithm [Paa97] and the Boyar-Peralta (BP) algorithm [BMP08, BP10, BMP13]. Compared with the Paar algorithm, the BP algorithm usually yields shorter g-XOR sequences via introducing the "distance" evaluation metric, and it has been further optimized in follow-up studies [VSP17, BFI19, ME19, TP19, LXZZ21, YWS+24, SYL25]. Furthermore, in low-latency scenarios, circuit depth becomes another key constraint. This leads to the depth-constrained SLP (SLPD) problem, where the depth of each base is strictly restricted by a predefined bound. To tackle SLPD, algorithms like BP-LSL [LSL+19] extend the BP framework to optimize g-XOR sequences under depth limits.

While numerous improvements and variants of the BP algorithm have been proposed, research on its efficiency optimization has long been insufficient. The original BP algorithm implementation proposed in [BMP08] serves as the basis of many works, including [KLSW17, LSL+19, TP19, LXZZ21]. However, this implementation relies on recursive stepwise exhaustive enumeration for distance evaluation and updating, which results in a severe combinatorial explosion as the matrix dimension and Hamming weight increase. For 32-dimensional dense matrices such as the AES-InvMixColumn and Whirlwind matrices, the achievable g-XOR sequences still have substantial potential for optimization in both normal and depth-constrained scenarios. As reported in [LXZZ21], solving SLP instances for specific 32-dimensional matrices with density around 0.5 using Algorithms A1 and A2 (improved BP variants proposed in [TP19]) remains computationally expensive on modern single-threaded CPUs, and current low-latency g-XOR implementations seldom mention these matrices. Although a recent linear-combination-based framework [SYL25] theoretically mitigates this combinatorial explosion by replacing the recursive search with an MILP model, its optimization does not seem to consider extensions to depth-constrained scenarios.

Alongside theoretical advancements, developing a universal acceleration framework for BP and its variants (A1, A2, BP-LSL) by refining the baseline implementation is also essential. Leveraging GPU parallel processing to accelerate these algorithms significantly extends their solvable range. This provides a promising approach to evaluate higher-dimensional linear layers and optimize more challenging SLP (or SLPD) instances, as well as the construction of high-dimensional MDS matrices.

1.1 Our Contributions

Algorithmic Improvement with Meet-in-the-middle Method. In current implementations of the BP algorithm, a critical computational bottleneck lies in the distance update procedure. To determine whether a target $\mathbf{y}$ can be represented as a sum of $d-1$ existing bases in $X = (\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{m-1})$, the algorithm enumerates all $\binom{m}{d-1}$ combinations, which induces high computational complexity in time. To mitigate this efficiency bottleneck, we applied the meet-in-the-middle (MITM) strategy to achieve an optimal time-space trade-off. By halving the enumeration depth, our approach significantly reduces the theoretical time complexity of the BP algorithm from $O(\binom{m}{d-1})$ to $O(\binom{m}{\lceil (d-1)/2 \rceil} \log_2 \binom{m}{\lceil (d-1)/2 \rceil})$. For instances with $m = 48, d = 15$, the proposed strategy delivers a theoretical speedup of two orders of magnitude.

GPU-Based Parallel Acceleration for Distance Evaluation. In addition to our MITM-based theoretical optimization and taking advantage of the massive parallelism of the GPU, we provide a high-performance framework of the BP algorithm to boost key components in it, including the parallelized methods for computing base combinations, performing multiple binary searches, and aggregating distance statistics. These components are designed with simple arithmetic operations to meet the requirements of large-scale parallelization on GPU architectures. Furthermore, we implement a space-time trade-off strategy that pre-computes and sorts all base combinations within a single iteration. By eliminating redundant computation and repeated sorting processes, this approach significantly reduces execution time and extends the applicability of the BP algorithm to higher-dimensional linear layers[1].

Candidate Pruning for Workload Optimization. To further boost our GPU-accelerated implementation, we introduce the candidate pruning mechanism to eliminate useless candidates before large-scale computation on the GPU side. For the A1 and A2 algorithms, we deploy a lightweight pre-testing strategy to discard candidates that cannot update the minimum non-zero entries in the distance vector. This drastically reduces the number of redundant GPU threads and significantly improves performance, particularly for high-dimensional matrices. Regarding the BP algorithm, we propose the "exEasymove" optimization: when the total distance decreases by only 1 in the previous iteration, it focuses on the decrease of the corresponding target until its distance drops to 0. This optimization further boosts the algorithm by about 5 times in the late stage when $n > 40$.

Experimental Validation of Performance Acceleration. We conducted comprehensive experiments to validate the efficiency and practicality of our optimized BP algorithm. On an entry-level GTX 1650 GPU, our accelerated BP implementation achieves a 6,200× speedup for a 26-dimensional dense matrix (0.5 density) over the CPU baseline in [KLSW17], and a 7,900× speedup for depth-5 g-XOR searches compared to [LSL+19]. It also enables the A1 and A2 solutions for 32-dimensional dense matrices (density 0.5), thereby effectively bridging the research gap identified in [LXZZ21]. Furthermore, when switching to a high-performance RTX 4090 (48 GB), our GPU-accelerated BP implementation expands the solvable boundary to 64-dimensional sparse matrices and 42-dimensional matrices (density 0.381) within 1.5 hours.

New g-XOR Results and Newly Found High-Dimensional MDS Matrices. With our framework, we successfully improved the g-XOR results on existing 32-dimensional matrices using our accelerated A1 implementation. For the Whirlwind M0 and M1 matri-

[1] All of our implementations and experimental results are available at https://github.com/huangx607087/bp-gpu-slp

ces [BNN+10], we improve upon existing results by reducing the g-XOR sequence lengths to 172 and 170. For other MDS matrices over $\mathrm{GL}_8(\mathbb{F}_2^4)$ reported in prior work [SKOP15, SS16, KLSW17], our GPU-accelerated implementation discovers improved g-XOR results within minutes, surpassing those reported in [TP19, LXZZ21]. We also evaluated these matrices under a depth constraint and obtained a 157-step solution for AES-InvMixColumn with depth 5, thereby extending the results of [LSL+19]. Furthermore, leveraging our GPU-accelerated BP and A1 algorithms, we constructed several circulant and Hadamard MDS matrices of 40 and higher dimensions. Specifically, these include 8-bit blockwise circulant matrices of dimensions 40, 48, and 56, as well as 4×4 block-circulant and Hadamard matrices of dimensions 40, 48, and 64.

1.2 Related Work

The Paar algorithm [Paa97] is one of the basic heuristic algorithms for minimizing the number of XOR gates in the implementation of linear layers. It merges the most frequent XOR operations in a circuit to eliminate redundancy. As an elimination-free algorithm, it cannot perform the cancellation law over a binary field to further reduce g-XOR operations. To overcome this limitation, the Boyar–Peralta (BP) algorithm was subsequently introduced in [BMP08, BP10] and improved in [BMP13]. By introducing the concept of "distance", it dynamically tracks the minimum number of existing bases required to derive each target, which typically reduces the required XOR operations by approximately 10% to 20%.

The BP algorithm has been further improved for better SLP results in later research. For high-density linear layers, [VSP17] introduced an enhanced heuristic combining cancellation and a "common path" strategy. Higher-quality SLP constructions were proposed in [BFI19] by applying random permutation matrices to the input matrices to introduce more randomness. In 2019, [TP19] proposed the A1 and A2 algorithms. These two algorithms prioritize updating the minimum nonzero entry in the distance vector to improve solution quality. Later, [LXZZ21] combined the Paar, BP, A1, and A2 algorithms and integrated 11 reduction rules into existing heuristic algorithms. They further optimized the iterative search mechanism, which led to better SLP heuristic solutions. Additionally, a recent study [SYL25] proposed a theoretical improvement to the BP algorithm. By generating combinations of the targets and introducing an MILP model, they removed the recursive structure to accelerate the BP algorithm and achieved an 89-step g-XOR solution for the AES MixColumn.

Beyond shortest g-XOR implementations, shortest s-XOR sequences and low-latency implementations have likewise been widely studied. For s-XOR sequences, [JPST17] proposed a graph-based search framework with matrix decomposition to minimize s-XOR counts. Later, [XZL+20] introduced multiplication and commutativity rules for further optimization. Building on this work, [YWS+24] exploited equivalent sequences and graph isomorphism to further reduce s-XOR counts and circuit depth. For low-latency implementations, [LSL+19] introduced depth-awareness into the BP algorithm to constrain the circuit depth. Their approach, BP-LSL, was further improved in [BFI21] by introducing randomness. In [LWF+22], the authors proposed a backward search framework to improve the solution quality. By combining these techniques, some further improvements are reported in [SFX23, LZW23].

As a common heuristic for SLP solutions, the BP algorithm has also been widely adopted to evaluate the implementation cost of MDS matrices. In [KLSW17], g-XOR complexity was employed to evaluate the lightweight efficiency of linear layers. Compared to methods based solely on the matrix Hamming weight [SKOP15, LS16, SS17, SS16], it offers a more accurate estimate of implementation cost and facilitates more efficient designs. Subsequently, [ZWS18] proposed five structured matrix constructions together with an efficient search algorithm for lightweight MDS matrices over matrix polynomial

residue rings. Later, [WLTZ23] reformulated MDS construction via elementary block-matrix decompositions and path-based search, obtaining additional lightweight 4×4 MDS matrices with optimal s-XOR complexity and improved involutory constructions.

While the BP algorithm has undergone substantial improvements in terms of solution quality, the efficiency of its practical implementation remains insufficiently investigated. The distance evaluation constitutes a major bottleneck to efficiency and thereby limits its applicability to high-dimensional linear layers. In this work, we revisit the implementation of the BP algorithm from both theoretical and practical perspectives, including meet-in-the-middle (MITM) acceleration and GPU parallelization. Consequently, we have substantially accelerated this algorithm and surpassed several existing g-XOR records, both with and without depth constraints, for 32-dimensional dense matrices reported in [LXZZ21]. Additionally, we have heuristically constructed MDS matrices in dimensions 40 to 64 and evaluated them using our implementation.

2 Preliminaries

2.1 Notations

Throughout this paper, we adopt the following notational conventions: scalars are denoted by lowercase letters (e.g., a), vectors by bold lowercase letters (e.g., $\mathbf{a}$), and matrices by bold uppercase letters (e.g., $\mathbf{A}$). We use $\mathrm{GL}_n(F)$ to represent all the $n \times n$ matrices over the field F. Unless otherwise specified, all vectors (except the distance vector) and matrices are defined over $\mathbb{F}_2$, where vectors over $\mathbb{F}_2$ can be referred to as "bases".

We mainly focus on bijective linear transformations. Such a transformation is represented by an invertible square matrix $\mathbf{A} \in \mathrm{GL}_n(\mathbb{F}_2)$, where n denotes the dimension of the linear layer. The initial input vectors $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}$ correspond to the rows of the $n \times n$ identity matrix $\mathbf{I}_n$. The target output vectors $\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{n-1}$ are the rows of the transformation matrix $\mathbf{A}$, and are collectively denoted by $\mathbf{Y} = (\mathbf{y}_0, \ldots, \mathbf{y}_{n-1})^\top$.

During the g-XOR evaluation process, we denote $X = (\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{m-1})$ as the sequence of currently available bases, where $\mathbf{x}_k$ with $k \geq n$ denotes an intermediate base. We use m to denote the current number of bases, initializing it as n and incrementing it whenever a new base is added to X. In the g-XOR sequence, we use a quadruple (u, a, b, t) to represent the operation $\mathbf{x}_u \leftarrow \mathbf{x}_a \oplus \mathbf{x}_b$, where $t = -1$ indicates that $\mathbf{x}_u$ is an intermediate base, and $t \neq -1$ indicates that $\mathbf{x}_u$ corresponds to the t-th target value $\mathbf{y}_t$.

Any n-dimensional binary vector $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$ and any polynomial over $\mathbb{F}_2[x]$ can be uniquely mapped to an integer between 0 and $2^n - 1$. For compact representation, binary vectors are denoted using typewriter-style hexadecimal notation with the suffix `h`, while polynomials are represented using sans-serif hexadecimal notation prefixed by `0x`. The Hamming weight of a binary vector $\mathbf{a}$ is denoted by $\mathrm{wt}(\mathbf{a})$. For example, the vector $\mathbf{a} = (1, 0, 1, 1, 1, 0)$ is represented as `1Dh`, whereas the polynomial $x^4 + x + 1$ is written as `0x13`. For brevity, prefixes or suffixes may be omitted for simple values such as `1`, `2`, `4`, and $\mathtt{2}^{-1}$.

In this work, we mainly focus on two types of matrices over $\mathbb{F}_2^n$: the circulant matrices and the Hadamard matrices. For these two types of matrices, we represent a circulant matrix by $\mathrm{Circ}(a_1, a_2, a_3, a_4)$ and a Hadamard matrix by $\mathrm{Had}(a_1, a_2, a_3, a_4)$, where all a_i are the elements in $\mathbb{F}_2^n$, with each matrix specified by the elements in its first row. Besides their smaller search space, we consider circulant matrices because they are used in the diffusion layer of AES for their compact representation and efficient implementation. For Hadamard matrices, on the other hand, they can be used for constructing involutory matrices, as long as the sum of the elements in their first row equals `0x1` [GR13].

2.2 Boyar-Peralta Algorithm

As a heuristic algorithm for the SLP, the Boyar-Peralta (BP) algorithm maintains a distance vector $\mathbf{d} = (d_0, d_1, \ldots, d_{n-1})$ during the execution. For each target $\mathbf{y}_i$, its distance d_i indicates that $\mathbf{y}_i$ can be represented as the XOR sum of at most $d_i + 1$ bases from the current base list $X = (\mathbf{x}_0, \ldots, \mathbf{x}_{m-1})$. In each iteration, the BP algorithm evaluates all pairwise XOR combinations $\mathbf{x}_i \oplus \mathbf{x}_j$ (each denoted as a candidate) from the base list X and observes the resulting changes in $\mathbf{d}$. The algorithm selects the candidate that minimizes $\|\mathbf{d}\|_1 = \sum_{i=0}^{n-1} d_i$. If multiple candidates yield the same minimum $\|\mathbf{d}\|_1$, the one that maximizes $\|\mathbf{d}\|_2 = \sqrt{\sum_{i=0}^{n-1} d_i^2}$ will be chosen.

Specifically, when $1 \in \mathbf{d}$ with $d_k = 1$, the "easymove" mechanism is introduced into the BP algorithm. In this case, the target $\mathbf{y}_k$ can be expressed as the XOR of two bases in the current list X. Consequently, the algorithm enumerates all pairwise combinations of the current vectors to identify the pair $(\mathbf{x}_i, \mathbf{x}_j)$ such that $\mathbf{y}_k = \mathbf{x}_i \oplus \mathbf{x}_j$. Then the operation $\mathbf{x}_m \leftarrow \mathbf{x}_i \oplus \mathbf{x}_j$ is added to the g-XOR sequence, which reduces $\binom{m}{2} - 1$ distance evaluations. This optimization preserves the correctness and solution quality as proved in [BMP13]. The complete BP procedure, including the "easymove" strategy, is summarized in Algorithm 1.

In 2019, Tan et al. [TP19] proposed the A1 and A2 algorithms, which enforce updating the minimum non-zero entry in $\mathbf{d}$ at each step. The A1 algorithm then compares the resulting $\|\mathbf{d}\|_1$ and $\|\mathbf{d}\|_2$ to determine the candidate to insert, as BP does, while the A2 algorithm does not consider $\|\mathbf{d}\|_2$ to introduce more randomness. Compared with the BP algorithm, these two algorithms provide better SLP solution quality but require more execution time, as $\max d_i$ decreases more slowly than in BP.

In addition, the BP algorithm can also be applied to search for low-latency circuits. By introducing the depth-aware strategy, the BP-LSL algorithm was proposed [LSL+19]. By imposing a depth constraint, the search prunes candidate bases that would exceed the depth limit, which effectively reduces the overall depth of the resulting circuit at the cost of more XOR gates. As the A1, A2, and BP-LSL algorithms introduce only simple additional candidate-selection logic compared to the BP algorithm, our optimization strategies for BP proposed in this paper can be directly extended to improve the performance of these algorithms.

3 Comprehensive Optimization for Boyar-Peralta Algorithm

3.1 Theoretical Complexity Improvement via Meet-in-the-middle

As illustrated in Algorithm 1, the BP algorithm has to compute the updated distance vector for a candidate $\mathbf{x}_i \oplus \mathbf{x}_j$ to determine whether it shortens the distance of each target. In most implementations, including [KLSW17, TP19, LXZZ21], this process contains two functions: one is `NewDistance`, which evaluates the distance for $\mathbf{y}_u$ when the candidate $\mathbf{c}$ is inserted into the base sequence X. The other one is `reachable`, which checks whether the distance can be updated. These two functions are described in Algorithms 2 and 3.

The `reachable` function has three main arguments: $\mathbf{t}$, K, and S. It determines whether the target $\mathbf{t}$ can be represented as the XOR sum of K elements starting from index S. It uses a very naive recursive algorithm that has to check every possible combination via exhaustive enumeration. Consequently, its time complexity is at the combinatorial level: for a set X containing m bases, determining whether a target $\mathbf{y}_i$ is reachable within d_i steps necessitates an enumeration of $\binom{m}{d_i}$ cases. Considering all n targets and the $\binom{m}{2}$ candidate combinations, a single iteration requires approximately $O(nm^2)$ calls to the `reachable` function. Let $d_{\max}$ denote the maximum value in $\mathbf{d}$. The complexity in each iteration then reaches $O\left(nm^2 \cdot \binom{m}{d_{\max}-1}\right)$, which has the same order as $O\left(nm^{1+d_{\max}}\right)$.

Algorithm 1 Pseudocode of (Randomized) BP Algorithm

Require: Target bases $\mathbf{Y} = (\mathbf{y}_0, \mathbf{y}_1, ..., \mathbf{y}_{n-1})$
Ensure: Constructing a g-XOR sequence for all target bases in $\mathbf{Y}$.

Initialize $\mathbf{x}_i$ as the i-th unit vector for $i \in \{0, 1, \ldots, n-1\}$.
Set $X \leftarrow (\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_{n-1})$ as a list of vectors.
$m \leftarrow n$
$\mathbf{d} \leftarrow (\mathrm{wt}(\mathbf{x}_0) - 1, \mathrm{wt}(\mathbf{x}_1) - 1, ..., \mathrm{wt}(\mathbf{x}_{n-1}) - 1)$
seq $\leftarrow$ Emptylist
while True **do**
  sumDis $\leftarrow \|\mathbf{d}\|_1$
  if sumDis $= 0$ **then**
    break
  end if
  if $1 \in \mathbf{d}$ with $\mathbf{d}_a = 1$ **then**
    for $\{\mathbf{x}_i, \mathbf{x}_j\} \in X$ **do**
      if $\mathbf{x}_i \oplus \mathbf{x}_j = \mathbf{y}_a$ **then**
        seq.append$((m, i, j, a))$
        X.append$(\mathbf{x}_i \oplus \mathbf{x}_j)$
        $m \leftarrow m + 1$
        $\mathbf{d} \leftarrow$ CalculateDist$(\mathbf{Y}, X, \mathbf{x}_i \oplus \mathbf{x}_j, \mathbf{d})$.
        break
      end if
    end for
    continue
  end if
  minDis, maxNorm $\leftarrow (\|\mathbf{d}\|_1, \|\mathbf{d}\|_2)$
  for $\{\mathbf{x}_i, \mathbf{x}_j\} \in X$ **do**
    $\mathbf{d}' \leftarrow$ CalculateDist$(\mathbf{Y}, X, \mathbf{x}_i \oplus \mathbf{x}_j, \mathbf{d})$.
    (tmpDis, tmpNorm) $\leftarrow (\|\mathbf{d}'\|_1, \|\mathbf{d}'\|_2)$
    if (tmpDis $<$ minDis) **or** (tmpDis $=$ minDis **and** tmpNorm $>$ maxNorm) **then**
      (minDis, maxNorm) $\leftarrow$ (tmpDis, tmpNorm)
      CandidateSet $\leftarrow \{(i, j)\}$
    end if
    if tmpDis $=$ minDis **and** tmpNorm $=$ maxNorm **then**
      CandidateSet $\leftarrow$ CandidateSet $\cup \{(i, j)\}$
    end if
  end for
  $(i^*, j^*) \leftarrow$ RandomChoice(CandidateSet)
  $\mathbf{d} \leftarrow$ CalculateDist$(\mathbf{Y}, X, \mathbf{x}_{i^*} \oplus \mathbf{x}_{j^*}, \mathbf{d})$.
  seq.append$((m, i^*, j^*, -1))$
  X.append$(\mathbf{x}_{i^*} \oplus \mathbf{x}_{j^*})$
  $m \leftarrow m + 1$
end while
return seq

Although $d_{\max}$ may decrease in later iterations, and [TP19] introduces a pruning strategy that immediately returns `true` whenever $\mathrm{wt}(\mathbf{t}) < K$, the increasing number of available bases m keeps the computational cost substantial throughout the algorithm.

During the distance evaluation of the `reachable` function, it only determines whether $\mathbf{t}$ can be represented as the XOR of K elements from X, without identifying the specific elements involved in the representation. As we mostly consider matrices with density no

Algorithm 2 Implementation of `NewDistance` function in [KLSW17]

Require: Index u, new base element $\mathbf{c}$, target bases $\mathbf{Y} = (\mathbf{y}_0, \mathbf{y}_1, ..., \mathbf{y}_{n-1})$, current distances $\mathbf{d} = (d_0, d_1, ..., d_{n-1})$ and current bases list $X = (\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_{m-1})$.
Ensure: Updated distance for the u-th target base.
if $\mathbf{y}_u \in X$ **or** $\mathbf{c} = \mathbf{y}_u$ **then**
  return 0
end if
$\mathbf{t} \leftarrow \mathbf{y}_u \oplus \mathbf{c}$
$K \leftarrow d_u - 1$
if `reachable`$(\mathbf{t}, K, 0)$ **then**
  return K
else
  return d_u
end if

Algorithm 3 Implementation of `reachable` function in [KLSW17]

Require: Target $\mathbf{t}$ (argument 1), number of elements K (argument 2), start index S (argument 3), current base set $X = (\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_{m-1})$.
Ensure: **true** if $\mathbf{t}$ is a sum of K elements in X; **false** otherwise.
if $(m - S) < K$ **then**
  return false
end if
if $K = 1$ **then**
  for $i \leftarrow S$ **to** $m - 1$ **do**
    if $\mathbf{t} = \mathbf{x}_i$ **then**
      return true
    end if
  end for
  return false
end if
if reachable$(\mathbf{t} \oplus \mathbf{x}_S, K - 1, S + 1)$ **then**
  return true
end if
if reachable$(\mathbf{t}, K, S + 1)$ **then**
  return true
end if
return false

more than 0.5, if the search depth is halved, the combination number $\binom{m}{\lceil (d_i-1)/2 \rceil}$ will be substantially smaller than $\binom{m}{d_i-1}$, thereby potentially mitigating this computational bottleneck.

Therefore, motivated by this observation, we employ the MITM strategy to reduce the theoretical complexity via the space-time trade-off. As illustrated in Figure 1, for a target $\mathbf{y}_i$ and current candidate $\mathbf{c}$ (formed by the XOR sum of two vectors in X), we define $K_1 = \lceil (d_i-1)/2 \rceil$ and $K_2 = \lfloor (d_i-1)/2 \rfloor$. The optimization proceeds by first computing the XOR sums of all possible combinations of K_1 distinct vectors from X and storing them in a sorted array V_1. For each XOR sum $\mathbf{v}$ generated from combinations of K_2 vectors in array V_2, we use binary search to determine whether $\mathbf{y}_i \oplus \mathbf{c} \oplus \mathbf{v}$ exists in V_1. Consequently, the total time complexity is reduced to $O\left(\left(\binom{m}{K_1} + \binom{m}{K_2}\right) \cdot \log_2 \binom{m}{K_1}\right) \sim O\left(2 \cdot \binom{m}{K_1} \cdot \log_2 \binom{m}{K_1}\right)$, which represents a significant improvement over the original exhaustive search. In the case

where $m = 48$ and $d_{\max} = 15$, the MITM optimization achieves a theoretical speedup of $\frac{\binom{48}{14}}{2\binom{48}{7}\log_2\binom{48}{7}} \approx 125.33\times$ in the first iteration.

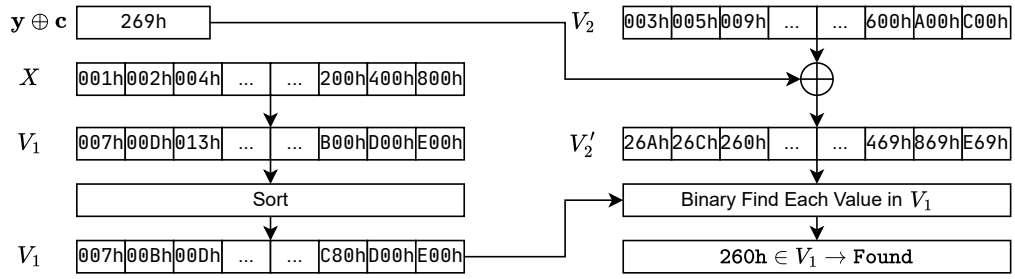

Figure 1: Schematic representation of the MITM optimization for the `reachable` function

3.2 Redundancy Elimination and Space-Time Trade-off

Although the MITM approach achieves a theoretical optimization of time complexity, redundant computations remain in the execution. Specifically, for $\mathbf{d} = (14, 13, 14, 13)$, the XOR sums of any 6 distinct bases in X will be computed $6 \times \binom{m}{2}$ times, and the XOR sums of any 7 distinct bases in X will be computed $2 \times \binom{m}{2}$ times. However, as X remains unchanged during this process, another space-time trade-off can be employed: we can precompute the XOR sums of any 1 to $\lceil (d_{\max} - 1)/2 \rceil$ elements in X and store them in arrays W_1 to $W_{\lceil (d_{\max}-1)/2 \rceil}$, where W_i has length $\binom{m}{i}$. For each update of d_i, we compute $K_1 = \lceil (d_i - 1)/2 \rceil$ and $K_2 = \lfloor (d_i - 1)/2 \rfloor$, followed by a parallel search in W_{K_1} for every $\mathbf{c} \oplus \mathbf{v}$ where $\mathbf{v} \in W_{K_2}$ and $\mathbf{c} \in \mathsf{CandidateSet}$. Reusing precomputed values in W within the same iteration further improves efficiency. For example, in the case of $n = 32$, if the distance vector $\mathbf{d}$ contains 16 entries of 15, 8 entries of 16, and 8 entries of 14, such reuse yields an effective speedup of $\frac{\binom{32}{2}(8\binom{32}{8}+48\binom{32}{7}+8\binom{32}{6})}{\sum_{i=1}^{8}\binom{32}{i}(1+\log_2\binom{32}{i})} \approx 353$ times.

3.3 GPU Parallelization for Acceleration

Building upon the aforementioned optimizations, we theoretically reduced the time complexity of the BP algorithm through MITM strategies and eliminated redundant computation. These two optimizations both successfully yield speedups of over two orders of magnitude in specific scenarios. However, for dense matrices or high-dimensional matrices, the reduced computational burden remains substantial in practice, especially in single-threaded implementations.

To further accelerate the execution of the BP algorithm, we take advantage of the high parallelism and throughput of GPUs. Our GPU acceleration mainly focuses on two key optimizations: the first acceleration lies in precomputing the arrays $W_1, W_2, \ldots, W_{\lceil (d_{\max}-1)/2 \rceil}$ for the XOR sums of any 1 to $\lceil (d_{\max} - 1)/2 \rceil$ elements in X. To construct W_k, $\binom{m}{k}$ GPU threads are executed in parallel, where the i-th thread evaluates the corresponding k-combination $(c_0, c_1, \ldots, c_{k-1})$ over the current base list X. The XOR sum $\bigoplus_{j=0}^{k-1} \mathbf{x}_{c_j}$ is then computed sequentially and stored in $W_k[i]$.

The second acceleration targets the binary search process for the MITM optimization, which performs the same function as the `reachable` function in current implementations. We serially handle each distinct target value $\mathbf{y}_i$ and concurrently process all $\binom{m}{2}$ candidates. Each candidate corresponds to $\binom{m}{\lfloor (d_i-1)/2 \rfloor}$ threads, and each thread performs a binary search on one element of $W_{\lceil (d_i-1)/2 \rceil}$. For an instance with $n = 32$, $d_{\max} = 16$, this parallel

binary search reduces the first iteration time cost from approximately 6,100 seconds to only 12 seconds on a GTX 1650 GPU. The implementation details for our GPU acceleration will be presented in Section 4.

3.4 Candidate Pruning and the ExEasyMove Strategy

Although GPU parallelization significantly accelerates the execution, a notable performance bottleneck still remains in the BP algorithm. In the late stage, most iterations reduce $\|\mathbf{d}\|_1$ by only 1, yet it still has to evaluate all $\binom{m}{2}$ candidates. For some instances with $n \geq 48$, m sometimes exceeds 300. Even though $d_{\max}$ has decreased to between 7 and 8, the required number of threads may still exceed that of the early stage when $d_{\max} = 12$ (i.e., $\binom{60}{2} \times \binom{60}{6} \approx 8.86 \times 10^{10}$, whereas $\binom{300}{2} \times \binom{300}{3} \approx 2.00 \times 10^{11}$). Moreover, for the A1 and A2 algorithms, the minimum non-zero entry in $\mathbf{d}$ must decrease by 1 in each iteration, and in BP-LSL, the depth of each candidate should also be taken into consideration. Thus, directly evaluating all candidates leads to a substantial waste of computational resources.

For the A1 and A2 algorithms, all candidates are first pre-tested in parallel to determine whether they can reduce one of the minimum entries in $\mathbf{d}$. Candidates satisfying this condition are marked as valid. After all distances corresponding to the minimum non-zero target value have been examined, unmarked candidates are discarded from the candidate set of the current iteration. Consequently, subsequent large-scale distance computations are performed only on valid candidates, thereby reducing GPU workload and improving overall execution efficiency. Similarly, for BP-LSL, if a non-target candidate reaches the predefined maximum depth, it can also be pruned early during the distance update phase. This strategy likewise eliminates redundant computations, effectively reducing GPU overhead and streamlining parallel processing.

For the BP algorithm, we introduce the "exEasyMove" strategy: if a BP iteration reduces $\|\mathbf{d}\|_1$ by only 1 through decreasing d_{i^*}, the algorithm continues reducing d_{i^*} until it reaches zero. To justify this strategy, we revisit the BP execution process. In each iteration, BP selects the candidate that minimizes $\|\mathbf{d}\|_1$. If ties occur, it breaks them by choosing the candidate that maximizes $\|\mathbf{d}\|_2$. Since the previous iteration decreased $\|\mathbf{d}\|_1$ by 1 for entry d_{i^*}, d_{i^*} must have been *one of* the minimum non-zero entries in $\mathbf{d}$. After this reduction, d_{i^*} becomes the *unique* minimum non-zero entry. Following the core principle of A1 and A2, which prioritizes reducing the minimum entry, we restrict the search space solely to candidates that further decrease d_{i^*}, bypassing other candidates until d_{i^*} drops to zero. This candidate pruning strategy substantially reduces the GPU computational burden, achieving a 5× to 7× speedup in late-stage execution while preserving solution quality.

4 Implementation Details of Our GPU-Accelerated Design

4.1 Parallel Design of the MITM Search Kernel

In our implementation, we focus on bijective linear layer matrices of size up to 64×64 and a row Hamming weight of at most 21. Each base $\mathbf{x}_i$ is compressed into a 64-bit integer b_i and stored in array `B`, while pre-computed binomial coefficients are stored in array `C`. The distance vector $\mathbf{d}$ and search targets $\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{n-1}$ are stored in arrays `D` and `Y`. Most data arrays are maintained concurrently on both the CPU and GPU. However, to support parallel computation of base combinations, the large arrays storing the intermediate XOR sums reside exclusively on the GPU. Arrays `B` and `D` are synchronized after each iteration. The overall architecture and data flow are illustrated in Figure 2, where GPU-side variables are prefixed with `d_` for clarity.

Our design requires calculating the XOR sums of up to $\lceil (d_{\max} - 1)/2 \rceil$ distinct bases

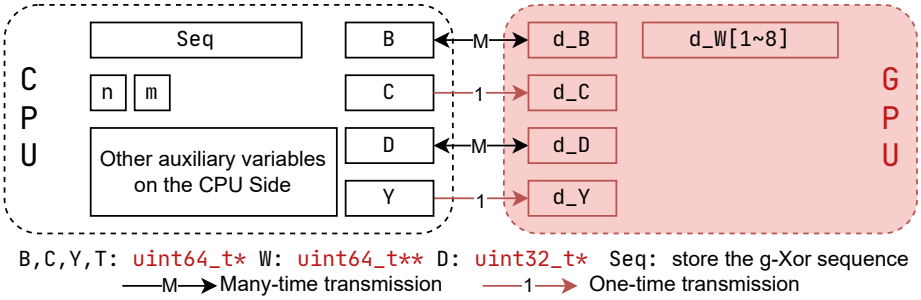

Figure 2: Overview of our basic implementation architecture

from B. When a new iteration begins, we define a maximum combination depth $K_{\max} = \lceil (d_{\max} - 1)/2 \rceil$, which is bounded by 10 within our parameter scope. We precompute the XOR sums for all combinations of k distinct bases ($1 \le k \le K_{\max}$) and store these values in allocated GPU arrays d_W[1] through d_W[$K_{\max}$]. Concurrently, candidate values generated by pairwise XOR operations over the elements in B are stored sequentially in the GPU-side array d_NextB. Leveraging the massive parallelism of the GPU, each individual d_W array is processed and subsequently sorted using the thrust library to prepare for the on-demand distance calculation.

In the parallel binary search, for a target $\mathbf{y}_{i^*}$ with distance d_{i^*} and candidate vector $\mathbf{c}$, we define the exact split search depths as $K_1 = \lceil (d_{i^*} - 1)/2 \rceil$ and $K_2 = \lfloor (d_{i^*} - 1)/2 \rfloor$. The GPU launches $\binom{m}{K_2}$ threads in parallel, where each thread computes a dedicated search target $\mathbf{t} = \mathbf{y}_{i^*} \oplus \mathbf{c} \oplus \mathtt{W}[K_2][i]$ and performs a binary search within the pre-sorted array d_W[K_1].

Similar to the reachable function in current implementations, we only check whether a target can be represented by a valid XOR combination of $d_{i^*} - 1$ elements in B, instead of recovering the specific combination. Thus, we define a variable d_R on the GPU side and initialize it to 0. Once a thread successfully finds the XOR sum $\mathbf{v} \oplus \mathbf{c}$ (with $\mathbf{v} \in W_{K_2}$ and $\mathbf{c}$ being a candidate), it directly sets d_R to 1. This process should be implemented as "write-only" and necessitates a direct assignment rather than an increment (++) or a bitwise (|= 1) operation, as reading d_R at this highly parallelized stage might cause memory synchronization issues and incur the overhead of atomic read-modify-write instructions. After this process, the existence of a valid combination is determined by reading the value of d_R. A value of 1 indicates that the current candidate $\mathbf{c}$ can further reduce d_{i^*}, whereas a value of 0 indicates that $\mathbf{c}$ does not lead to an update of d_{i^*}.

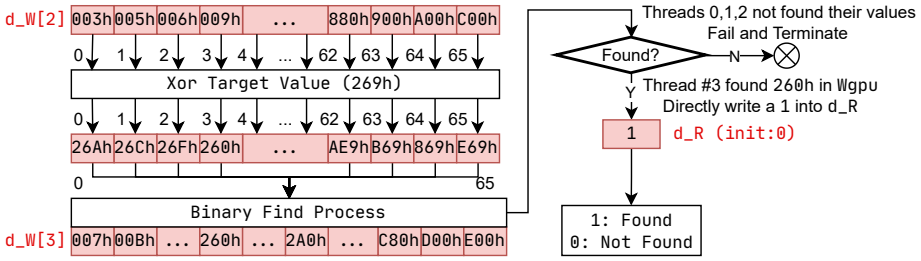

Figure 3: GPU-side parallel binary search diagram ($m = 12, K_1 = 3, K_2 = 2$)

4.2 High-Throughput Reachability Judge for Multiple Candidates

While the GPU parallelization illustrated above significantly accelerates the BP algorithm practically, there still exists a new performance bottleneck in its later stages: As the algorithm progresses, the number of variables m in array `B` (or variable list X) often grows to over 250 or even 300 when $n \geq 48$. Though $d_{\max}$ also decreases, it often remains substantial (e.g., from above 13 to around 9). If the GPU kernel checks only one candidate at a time, the serial scheduling complexity escalates to $O(nm^2)$. For instance, with $(n, m) = (48, 250)$ in the mid-to-late stages, the GPU kernel will be launched nearly 1.5 million times ($\binom{250}{2} \times 48 = 1,494,000$). Such frequent kernel invocations and CPU-GPU synchronizations reduce GPU utilization from over 95% initially to roughly 37%, resulting in substantial GPU underutilization and reduced effectiveness of large-scale parallel execution.

To address the issue of low GPU utilization, we optimized the framework based on our initial architecture, resulting in the enhanced version shown in Figure 4. While the n target values are processed serially in an outer loop, the reachability evaluation for all $\binom{m}{2}$ candidate pairs is fully parallelized for each target. We accomplish this by extending the single-result variable `d_R` into an array of length $\binom{m}{2}$. Consequently, the number of concurrent GPU threads increases from $\binom{m}{K_2}$ to $\binom{m}{2} \times \binom{m}{K_2}$. Because this thread count may exceed the GPU grid-size limit of $2^{31} - 1$, we partition the computation into multiple batches. Since each batch still provides massive thread-level parallelism, this partitioning has a negligible impact on GPU utilization in practice.

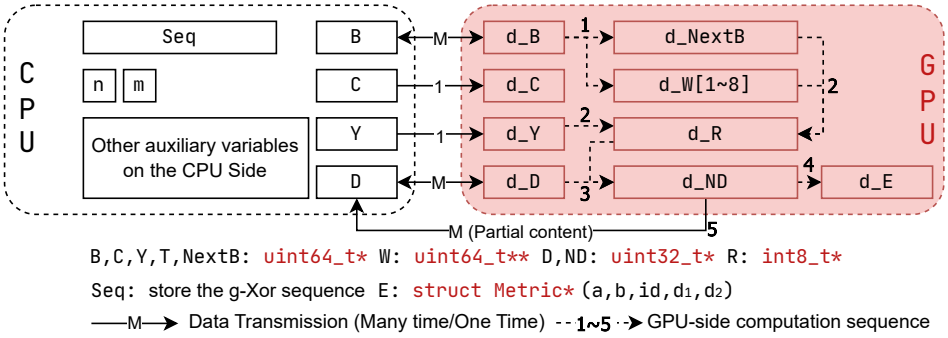

Figure 4: Overview of our complemented implementation architecture

Figure 5 illustrates the execution flow of our highly parallelized `reachable` function for updating the distance of the target `Y[i]` under evaluation. We launch $\binom{m}{2} \times \binom{m}{K_2}$ threads at the GPU side. For a thread with its thread ID k, its corresponding candidate index (`pairID`) and combination index (`combID`) can be extracted through a simple division-modulo process. It then searches the XOR sum of the `pairID`-th candidate and the `combID`-th combination in W_{K_2}, (e.g., `NextB[pairID]` $\oplus$ `W[K2][combID]`) within `d_W[K1]` and writes to `d_R[pairID]` if the binary search succeeds. As each candidate `d_NextB[pairID]` corresponds to $\binom{m}{K_2}$ threads, the non-atomic direct assignment strategy described in Section 4.1 is adopted here to prevent potential serialization for memory read-write locking.

After all $\binom{m}{2}$ search results are collected into `d_R`, we process the new distance evaluation for the current target `Y[i]` on the GPU. By combining `d_R` and `Y[i]`'s distance `d_D[i]`, the i-th component of the updated distance vector corresponding to each candidate insertion is computed and written into `d_ND`. As shown in Figure 6, the k-th thread computes the difference between `d_D[i]` and the previously obtained binary search result `d_R[k]`, producing an updated distance value, which is stored in `d_ND`. The array `d_ND`

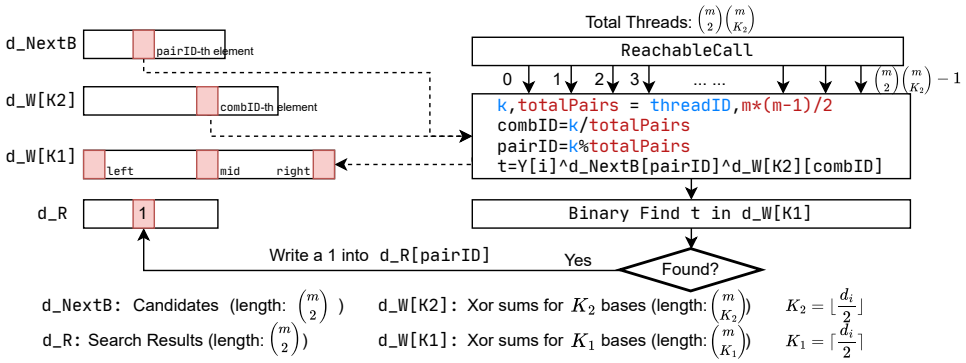

Figure 5: Execution flow for our highly parallelized `reachable` function

has length $\binom{m}{2} \times n$, with each group of n consecutive entries representing the distances after inserting a specific candidate. Accordingly, the updated distance vector for the k-th candidate is located between `d_ND[k*n]` and `d_ND[k*n+n-1]`. Therefore, for target `Y[i]`, the k-th thread writes the new distance value at index $kn + i$ of `d_ND`.

To avoid extensive data transfer between GPU and host memory, a structure `Metric` with five elements $(\mathtt{a}, \mathtt{b}, \mathtt{id}, \mathtt{d}_1, \mathtt{d}_2)$ is defined to aggregate the distance data in `d_ND`. It represents the insertion of the `id`-th candidate, which is derived from the XOR sum of `B[a]` and `B[b]`, and will update the distance vector $\|\mathbf{d}\|$ with $\|\mathbf{d}\|_1 = \mathtt{d}_1$ and $\|\mathbf{d}\|_2^2 = \mathtt{d}_2$. By maintaining the `Metric` array `d_E` on the GPU for the aggregation of `d_ND`, the data transmission volume is reduced to $5/n$ of that of directly transferring `d_ND`. As the length of `d_E` cannot exceed 10^5 in our tests, this data transfer incurs negligible latency. We then perform a sort process and binary search on `E` to determine the number of tied optimal solutions and randomly select one (with $\mathtt{id} = \mathtt{id}^*$) to update the base array `B` and the g-XOR sequence, along with updating the distance vector `D` to the segment between $\mathtt{d_ND}[\mathtt{id}^* \cdot n]$ and $\mathtt{d_ND}[\mathtt{id}^* \cdot n + n - 1]$.

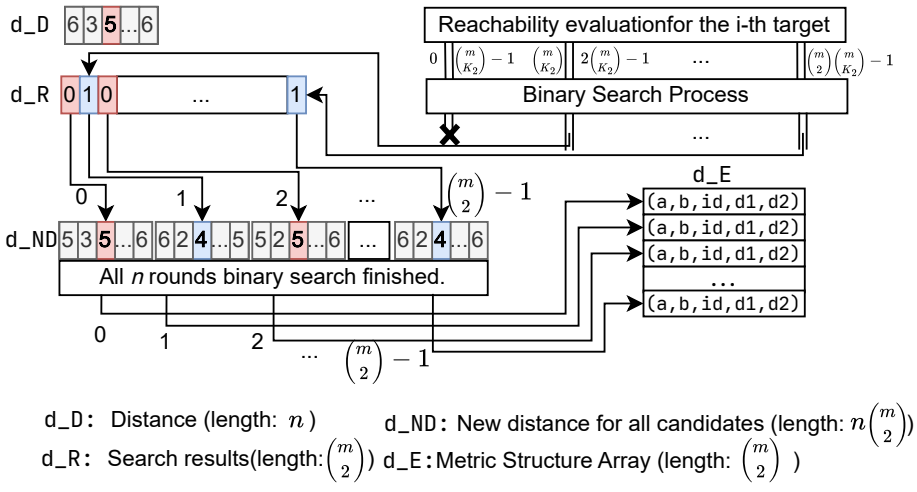

Figure 6: Diagram of processing `d_ND` and `d_E`

4.3 Filtering out Useless Candidates

In Section 4.2, we transitioned the computation of distance vectors $\mathbf{d}$ following the insertion of each candidate in `d_NextB` from a serially executed nested loop with layered recursion to a highly parallel architecture on the GPU. However, managing $\binom{m}{2} \times \binom{m}{K_2}$ threads also poses challenges to the GPU device due to its parallelism limit. This causes congestion and substantial latency on the GPU side. Thus, for the BP, A1, and A2 algorithms, effective candidate pruning evidently reduces the number of parallel threads and improves execution efficiency.

For the BP algorithm, we apply the "exEasymove" mechanism mentioned in Section 3.4. When $\|\mathbf{d}\|_1$ drops by only 1 in the last iteration with d_{i^*} decreased, an additional array `d_OK` is applied to determine whether a candidate can further reduce the distance component d_{i^*}. For the u-th candidate `d_NextB[u]` $=$ `B[a]` $\oplus$ `B[b]`, if it further reduces d_{i^*}, we set `d_OK[u]` to 2^{16}`a` $+$ `b` since `a` and `b` do not exceed 65,535. Otherwise, `d_OK[u]` is set to the maximum value of a 32-bit unsigned integer, i.e., `0xFFFFFFFF`. Following this procedure, we sort the `d_OK` array and count the number of elements that are not equal to `0xFFFFFFFF`, which corresponds to the number of valid candidates. This allows us to determine the total number of valid candidates and retrieve the indices of their corresponding elements from the original set `B`. This detailed process is illustrated in Figure 7.

In regard to the A1 and A2 algorithms, a similar approach is adopted to check whether a candidate reduces one of the minimum non-zero entries in $\mathbf{d}$. Compared with the "exEasymove" strategy for BP, the only modification is to replace the candidate-checking criterion: instead of testing whether a candidate further decreases the distance of the target responsible for the one-step reduction in $\|\mathbf{d}\|_1$, we test whether it can further decrease the distance of one of the targets corresponding to the minimum non-zero entries in the current $\mathbf{d}$. The processing of `d_OK` follows a similar principle, as we mark the candidate only when it can further decrease one of the minimum non-zero entries in $\mathbf{d}$. Meanwhile, for BP-LSL, its depth-aware process can be utilized to enable GPU threads to return early, thereby avoiding the binary search process. Additionally, in the non-easymove phase with a depth constraint D, candidates with depth $D-1$ can be pruned in advance to further reduce the number of evaluated GPU threads.

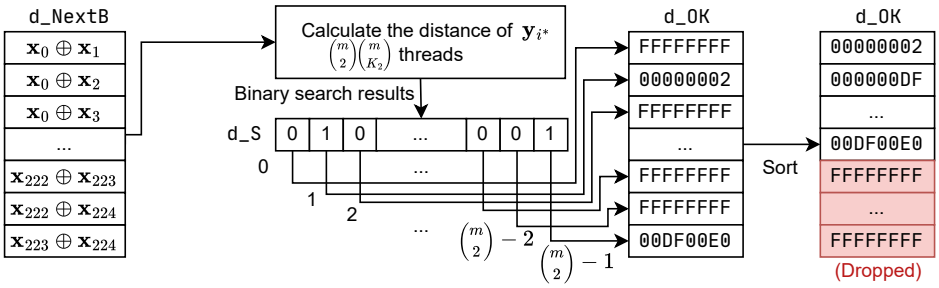

Figure 7: Diagram of the proposed candidate filtering process on GPU ($m = 225$)

5 Experimental Evaluation and Results

In our experiments, we evaluated our GPU-accelerated implementations based on the following three aspects: the performance on high-dimensional random linear layers, new g-XOR records for 32-dimensional dense cryptographic linear layers, and constructions of MDS matrices with 40 or more dimensions. To systematically quantify the impact

of matrix dimension n and density w in our initial benchmarks, we developed a custom matrix generation utility that constructs invertible matrices over $\mathbb{F}_2$ based on dimension n, row Hamming weight w, and random seed s for random matrix evaluations. All random matrices generated from it contain $\lfloor n/4 \rfloor$ rows with a Hamming weight of $w+1$, $\lfloor n/4 \rfloor$ rows with a Hamming weight of $w-1$, and all remaining rows with a Hamming weight of w.

5.1 Performance and Solution Quality on Random Matrices

We first evaluated the performance and solution quality of our GPU-accelerated BP algorithm against the baseline implementation from [KLSW17] across matrix dimensions $n \in [16, 26]$. All test matrices were generated over $\mathbb{F}_2$ with a fixed seed $s = 1$, and the experiments in this subsection were conducted on a laptop equipped with a GTX 1650 (4 GB) GPU. As presented in Table 1, our GPU-accelerated implementation achieves substantial execution speedups while maintaining nearly identical solution quality (XOR count). Notably, for the most computationally intensive instance $(n, w) = (26, 13)$, the implementation in [KLSW17] requires over 16 hours, whereas our GPU approach completes the search within only 9.44 seconds, representing a speedup of over $6,200\times$.

Table 1: Performance comparison between original and our GPU-accelerated BP algorithm

n	Method	Time consumption (s) / XOR count						
		$w=6$	$w=8$	$w=9$	$w=10$	$w=11$	$w=12$	$w=13$
16	[KLSW17]	0.37/49	0.86/51	–	–	–	–	–
	Ours	0.01/47	0.01/51	–	–	–	–	–
18	[KLSW17]	1.10/55	3.51/60	7.27/62	–	–	–	–
	Ours	0.01/53	0.01/61	0.01/62	–	–	–	–
20	[KLSW17]	3.12/64	17.9/77	49.2/78	192/82	–	–	–
	Ours	0.01/62	0.02/75	0.05/78	0.18/81	–	–	–
22	[KLSW17]	8.09/69	57.8/87	129/90	549/93	600/88	–	–
	Ours	0.01/67	0.04/87	0.09/87	0.27/91	0.42/86	–	–
24	[KLSW17]	11.8/81	143/95	703/102	967/102	1479/104	5087/110	–
	Ours	0.01/79	0.08/94	0.31/101	0.38/101	0.67/105	1.49/106	–
26	[KLSW17]	52.8/87	681/100	4452/113	7921/116	9594/125	22828/122	59079/127
	Ours	0.02/86	0.17/100	1.12/111	1.26/116	3.19/121	4.24/122	9.44/125

We also evaluated our implementation of the BP-LSL algorithm under depth constraints. In the original implementation [LSL+19], inserting new intermediate bases does not strictly guarantee a reduction in $\|\mathbf{d}\|_1$, which often leads to search stagnation or prohibitive execution time. Therefore, we set the depth limit to the theoretical minimum depth plus one ($d_{\min} + 1$), resulting in a depth limit of 4 for $w = 6$ and 5 for $w \geq 8$. Additionally, we incorporated the tie-breaking rule from [SFX23] for scenarios with depth constraints: when multiple candidates yield identical minimal $\|\mathbf{d}\|_1$, the candidate that *minimizes* $\|\mathbf{d}\|_2$ is selected. The evaluation results are summarized in Table 2, and it can be observed that our GPU-accelerated implementation consistently delivers significant speedups over [LSL+19] while maintaining comparable solution quality.

5.2 Performance Evaluation on High-Dimensional Random Matrices

5.2.1 Results on the BP Algorithm

We evaluated the computational efficiency and memory overhead of our GPU implementation across large-scale matrices with dimensions $n \in [32, 64]$ and row Hamming weights

Table 2: Performance comparison between the original and our GPU-accelerated BP-LSL algorithm

n	Method	Time consumption (s) / XOR count						
		$w=6$	$w=8$	$w=9$	$w=10$	$w=11$	$w=12$	$w=13$
16	[LSL+19]	0.15/51	0.84/56	–	–	–	–	–
	Ours	0.02/50	0.02/57	–	–	–	–	–
18	[LSL+19]	0.29/55	5.05/65	9.47/69	–	–	–	–
	Ours	0.03/55	0.03/62	0.05/68	–	–	–	–
20	[LSL+19]	0.60/63	12.3/78	42.6/80	117/87	–	–	–
	Ours	0.03/64	0.05/77	0.07/83	0.13/88	–	–	–
22	[LSL+19]	1.56/73	30.3/89	103/91	278/99	534/99	–	–
	Ours	0.04/73	0.07/90	0.14/95	0.17/97	0.35/98	–	–
24	[LSL+19]	2.84/82	72.3/99	339/109	862/108	2522/113	5365/125	–
	Ours	0.04/81	0.10/99	0.20/105	0.35/109	0.88/111	1.31/129	–
26	[LSL+19]	9.00/88	246/107	1614/115	2662/125	6440/129	21264/131	42260/141
	Ours	0.06/88	0.13/105	0.42/118	0.69/126	1.26/125	2.87/134	5.29/142

$w \in \{8, 10, 12, 14, 16\}$ on an RTX 4090 GPU (48 GB). To ensure consistency across this parameter space, all test instances were generated using a fixed random seed $s = 1$.

As illustrated in Table 3, for sparse instances ($w \in \{8, 10\}$), our BP implementation solves matrices up to $n = 64$ within a few minutes to an hour while maintaining a very low memory footprint (≤ 2.2 GB). Although higher row density w increases computational complexity, our approach significantly expands the solvable boundary of the BP algorithm within 1.5 hours. Specifically, we successfully process dense matrices up to $n = 54$ for $w = 12$, $n = 48$ for $w = 14$, and $n = 42$ for $w = 16$, which were computationally infeasible with the implementation in [KLSW17].

Table 3: Performance of our BP algorithm implementation on high-dimensional matrices

n	$w=8$			$w=10$			$w=12$			$w=14$			$w=16$		
	Time	Mem	ΔPaar	Time	Mem	ΔPaar	Time	Mem	ΔPaar	Time	Mem	ΔPaar	Time	Mem	ΔPaar
32	2.0	1.1	-5	0.7	1.1	-13	2.8	1.1	-24	18.4	1.3	-19	33.3	1.7	-32
34	0.2	1.1	-7	1.5	1.1	-11	7.2	1.1	-20	20.6	1.2	-28	66.3	2.0	-30
36	0.6	1.1	-8	1.6	1.1	-14	11.8	1.1	-20	46.5	1.4	-26	504.1	4.8	-28
38	0.3	1.1	-9	2.6	1.2	-15	20.0	1.2	-23	124.3	1.7	-26	292.3	2.6	-39
40	0.8	1.1	-2	3.8	1.1	-14	27.9	1.3	-23	505.3	3.5	-30	1247.1	7.3	-35
42	0.9	1.1	-7	6.9	1.1	-16	44.7	1.6	-19	375.8	1.9	-32	1048.5	11.6	-30
44	1.8	1.1	-6	9.7	1.1	-12	196.0	2.0	-22	336.0	7.0	-20	–	–	–
46	3.9	1.1	-9	14.1	1.4	-16	441.0	2.4	-16	2530.8	7.3	-30	–	–	–
48	2.3	1.1	-10	21.0	1.2	-11	184.8	1.9	-18	1879.5	7.9	-31	–	–	–
50	7.2	1.2	-6	30.3	1.2	-15	1147.8	2.4	-11	–	–	–	–	–	–
52	12.1	1.2	-6	53.4	1.4	-12	2140.7	3.4	-17	–	–	–	–	–	–
54	11.2	1.2	-3	209.7	1.3	-15	3163.7	9.5	-18	–	–	–	–	–	–
56	15.7	1.2	-9	414.2	1.7	-14	–	–	–	–	–	–	–	–	–
58	15.1	1.2	-8	971.5	1.3	-12	–	–	–	–	–	–	–	–	–
60	26.8	1.1	-8	631.2	1.4	-14	–	–	–	–	–	–	–	–	–
62	27.2	1.2	-7	960.4	2.2	-9	–	–	–	–	–	–	–	–	–
64	32.6	1.2	-11	4183.9	1.8	-11	–	–	–	–	–	–	–	–	–

5.2.2 Results on the A1 and A2 Algorithms

By enforcing a strict decrease in the minimum non-zero entry of $\mathbf{d}$ during each iteration, the A1 and A2 algorithms typically yield shorter g-XOR sequences than BP. However, this constraint slows the decay rate of $d_{\max}$, sustaining a high per-iteration complexity of $O(m^{\lfloor d_{\max}/2 \rfloor})$ as m grows. Consequently, memory overhead can exhaust the 48 GB GPU

memory capacity in dense, high-dimensional instances (Table 4, where runtime is reported in seconds and memory consumption in GB). Nevertheless, our candidate pruning scheme effectively eliminates redundant computations, ensuring high execution efficiency with most solvable instances completing within 15 minutes.

Table 4: Performance of our A1 algorithm implementation on high-dimensional matrices

n	$w=8$			$w=10$			$w=12$			$w=14$			$w=16$		
	Time	Mem	ΔPaar	Time	Mem	ΔPaar	Time	Mem	ΔPaar	Time	Mem	ΔPaar	Time	Mem	ΔPaar
32	2.2	1.1	-4	0.3	1.1	-12	0.9	1.2	-23	4.3	5.0	-31	6.8	2.6	-36
34	0.2	1.1	-9	0.5	1.1	-10	4.5	8.4	-23	3.8	2.7	-36	31.4	20.0	-41
36	0.3	1.1	-7	0.9	1.4	-15	2.9	2.7	-23	12.6	11.4	-35	Fail	>48	–
38	0.4	1.1	-6	1.8	2.2	-16	4.5	4.8	-26	22.8	12.9	-39	111.5	36.5	-39
40	0.5	1.1	-4	2.2	1.2	-15	7.4	3.4	-27	50.8	46.5	-34	Fail	>48	–
42	0.8	1.1	-8	2.8	1.3	-14	8.5	6.5	-22	Fail	>48	–	–	–	–
44	1.4	1.4	-8	5.4	1.9	-8	37.0	10.0	-27	104.2	28.9	-31	–	–	–
46	1.7	1.3	-8	6.6	2.8	-15	97.6	19.5	-25	–	–	–	–	–	–
48	2.5	1.4	-10	11.0	3.8	-15	286.4	30.7	-18	–	–	–	–	–	–
50	2.6	1.5	-5	14.3	5.4	-13	625.7	42.0	-14	–	–	–	–	–	–
52	2.9	1.5	-6	41.2	16.7	-9	Fail	>48	–	–	–	–	–	–	–
54	4.7	2.1	-5	43.2	6.8	-18	Fail	>48	–	–	–	–	–	–	–
56	4.7	1.1	-9	104.5	13.4	-13	Fail	>48	–	–	–	–	–	–	–
58	5.8	2.2	-10	44.6	18.5	-16	–	–	–	–	–	–	–	–	–
60	17.2	6.4	-7	Fail	>48	–	–	–	–	–	–	–	–	–	–
62	9.3	1.6	-3	Fail	>48	–	–	–	–	–	–	–	–	–	–
64	11.4	2.0	-7	137.5	37.6	-13	–	–	–	–	–	–	–	–	–

To evaluate the solvable boundary of our GPU-accelerated A1 and A2 algorithms under 48 GB GPU memory, we conducted robustness tests on 15 distinct sample matrices with random seeds ranging from $s = 12$ to 26. The results are summarized in Figure 8. Here, annotated numbers indicate the count of successfully solved instances, while unannotated dark and light cells represent regions with high and low solving success probabilities. It should be noted that this test provides a qualitative characterization of the practical solvability boundary, rather than a formal theoretical analysis.

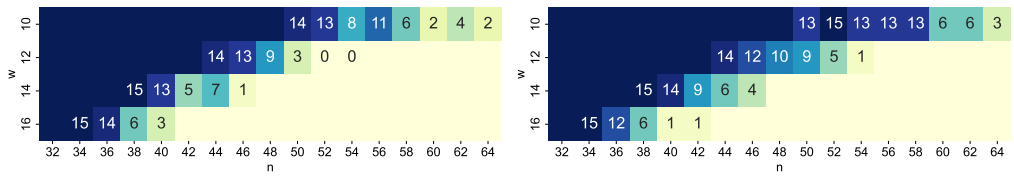

Figure 8: Robustness test over 15 samples under 48 GB memory limit for A1 (left) and A2 (right)

In our experiments, we observed temporary GPU memory spikes where usage abruptly doubled. Specifically, although total memory consumption generally increases with m for a fixed $d_{\max}$, temporary buffer allocations during GPU-side `thrust` parallel sorting frequently trigger momentary memory doubling before reverting to baseline levels. To mitigate such memory spikes, we also implemented a shared-memory version utilizing CUDA unified memory (`cudaMallocManaged`). By providing a unified virtual memory address space between host and device, this mechanism dynamically migrates pages on demand. Compared with the original explicit memory management, this unified memory approach makes full use of both host and device memory, avoiding out-of-memory failures and improving algorithmic robustness, but it introduces slight runtime overhead because of page migration.

5.3 New g-XOR Results on 32-Dimensional Known Dense Matrices

5.3.1 Shorter g-XOR Results Produced with GPU-Accelerated A1

Building upon our evaluations with random matrices, we tested nine 32-dimensional dense matrices using our GPU-accelerated A1 algorithm on an RTX 5080m GPU in this subsection. The experimental results are displayed in Table 5. Among the nine matrices, we obtained improved XOR counts for seven of them, which correspond to the bold values in the table. Although two of the XOR counts produced by our GPU-accelerated A1 do not surpass modern records obtained via matrix-decomposition-based s-XOR approaches [XZL+20, YWS+24], they still remain the best g-XOR results among previously reported ones.

Compared with [LXZZ21], we successfully applied the A1 algorithm to these dense matrices, providing experimental results for cases that remained unsolved in its Tables 5 and 7. Moreover, whereas previous works [TP19, LXZZ21] required up to five days to process 32-dimensional matrices, our GPU-accelerated A1 implementation completes a single run of each instance within 40 minutes. For all nine matrices considered, we allocated a total runtime budget of at most 12 hours per matrix, allowing multiple independent runs and retaining the best result obtained.

Table 5: Comparison of g-XOR count for 32-Dimensional Dense Matrices

Linear Layer	Existing g-XOR Results				Existing s-XOR Results		Our GPU-accelerated A1	
	[KLSW17]	[BFI19]	[TP19]	[LXZZ21]	[XZL+20]	[YWS+24]	Result	Time(s)
AES-InvMixColumn [JV02]	–	–	152	–	92	–	99	123
Whirlwind M0 [BNN+10]	212	210	–	183	183	173	**172**	2281
Whirlwind M1 [BNN+10]	235	234	–	180	190	181	**170**	1859
TwoFish [SKW+98]	129	125	122	121	111	110	117	20
$GL_8(\mathbb{F}_2^4)$ in [SKOP15]	209	194	–	169	170	169	**167**	468
$GL_8(\mathbb{F}_2^4)$ in [SS17]	201	203	–	177	183	183	**168**	179
$GL_8(\mathbb{F}_2^4)$ Involutory in [SKOP15]	217	212	–	172	185	182	**166**	2089
$GL_8(\mathbb{F}_2^4)$ Cauchy in [KLSW17]	196	–	–	–	–	–	**173**	525
$GL_8(\mathbb{F}_2^4)$ Vandermonde in [KLSW17]	212	–	–	–	–	–	**167**	649

5.3.2 New g-XOR Results under Depth Constraints with GPU-Accelerated BP-LSL

Beyond finding shorter g-XOR sequences for these 32-dimensional dense matrices, we further optimized their implementations under depth constraints using our GPU-accelerated BP-LSL algorithm. To the best of our knowledge, prior studies (e.g., [LSL+19, LWF+22, SFX23, LZW23]) have rarely evaluated 32-dimensional dense matrices with Hamming weights exceeding 350. To address this limitation, we conducted depth-bounded searches with a maximum base limit of 8,000. As detailed in Table 6, our GPU-accelerated BP-LSL implementation processes each dense matrix in under 3 minutes, demonstrating its high efficiency in depth-constrained optimization and providing competitive benchmark references for minimal-depth realizations of 32-dimensional dense matrices.

Table 6: New g-XOR results on 32-dimensional dense matrices under minimal-depth constraints

Matrix	Hamming Weight		Minimal g-XOR		Minimal Depth g-XOR		
	Total	RowMax	Depth	g-XOR	Depth	g-XOR	Time (s)
AES-InvMixColumn [JV02]	472	19	19	99	5	157	27
Whirlwind M0 [BNN+10]	520	19	22	172	5	218	74
Whirlwind M1 [BNN+10]	568	19	30	170	5	228	167
TwoFish [SKW+98]	359	14	16	117	4	148	4
$GL_8(\mathbb{F}_2^4)$ in [SKOP15]	464	15	19	167	4	194	47
$GL_8(\mathbb{F}_2^4)$ in [SS17]	426	16	20	168	4	207	27
$GL_8(\mathbb{F}_2^4)$ Involutory in [SKOP15]	544	19	31	166	5	213	89
$GL_8(\mathbb{F}_2^4)$ Cauchy in [KLSW17]	488	17	23	173	5	200	21
$GL_8(\mathbb{F}_2^4)$ Vandermonde in [KLSW17]	572	21	32	167	5	219	174

5.4 Results on High-Dimensional MDS Matrix Constructions

Based on our accelerated BP and A1 implementations, we constructed high-dimensional MDS matrices of dimension 40 or more in both circulant and Hadamard forms. Inspired by the criterion in [KLSW17], where the density of non-zero entries directly correlates with the final g-XOR count, our search prioritizes lightweight block candidates, specifically including 0x2, 0x3, $0x2^{\pm k}$, and $0x2^{\pm k}\pm1$ over $\mathbb{F}_2[x]/p(x)$, where $p(x)$ denotes an irreducible polynomial.

Specifically, for $n \in \{40, 48, 56\}$, we construct circulant matrices over $\mathrm{GL}_{n/8}(\mathbb{F}_2^8)$. For $n \in \{40, 48\}$, we additionally explore circulant and Hadamard matrices over $\mathrm{GL}_4(\mathbb{F}_2^{n/4})$, while for $n = 64$, we restrict our search to constructions over $\mathrm{GL}_4(\mathbb{F}_2^{16})$. To comprehensively evaluate each matrix class, we test constructions over different finite fields using multiple irreducible moduli: two distinct irreducible pentanomials are chosen for $\mathbb{F}_2^8$ and $\mathbb{F}_2^{16}$, whereas one irreducible trinomial and one irreducible pentanomial are selected for $\mathbb{F}_2^{10}$ and $\mathbb{F}_2^{12}$. For each configuration, Tables 7–10 present the top five constructions ranked by their g-XOR sequence lengths.

Table 7: Results on construction of circulant MDS matrices over $\mathrm{GL}_{n/8}(\mathbb{F}_2^8)$ with $n \in \{40, 48, 56\}$

Modulus	$n = 40$		$n = 48$		$n = 56$	
	Matrix	g-XOR	Matrix	g-XOR	Matrix	g-XOR
0x169	Circ(1,1,0xb4,0x04,0x58)	162	Circ(1,1,0x03,2,4,0x5a)	239	Circ(1,1,2^{-1},1,4,0x03,0xb0)	303
	Circ(1,1,0x02,0x5a,0xb0)	164	Circ(1,1,0x58,2,4,0xb5)	242	Circ(1,1,2,1,4^{-1},0xb5,0x09)	331
	Circ(1,1,0x02,0xa2,0xb0)	165	Circ(1,1,0x50,2,4,0x2d)	245	Circ(1,1,2^{-1},1,4,0xb5,0x08)	332
	Circ(1,1,0x02,0x05,0x08)	167	Circ(1,1,0x58,2,4,0x5a)	248	Circ(1,1,2^{-1},1,4,0xb5,0x09)	332
	Circ(1,1,0x02,0x03,0xb5)	167	Circ(1,1,0x2c,2,4,0xb5)	248	Circ(1,1,2^{-1},1,4,0x03,0x58)	336
0x11b	Circ(1,1,0x02,0xcb,0x8c)	165	Circ(1,1,0x03,2,4,0xcb)	232	Circ(1,1,2^{-1},1,4,0x8c,0x05)	310
	Circ(1,1,0x02,0xcb,0x46)	165	Circ(1,1,0x22,2,4,0x8c)	242	Circ(1,1,2^{-1},1,4,0x03,0x46)	332
	Circ(1,1,0x8d,0x04,0x23)	166	Circ(1,1,0xe8,2,4,0x46)	250	Circ(1,1,2^{-1},1,4,0x8c,0x10)	339
	Circ(1,1,0x02,0xcb,0x85)	166	Circ(1,1,0x8c,2,4,0xcb)	252	Circ(1,1,2,1,4^{-1},0x8c,0xc3)	347
	Circ(1,1,0x8d,0x46,0xca)	167	Circ(1,1,0x46,2,4,0xcb)	257	Circ(1,1,2^{-1},1,4,0x46,0x10)	351

Table 8: Results on construction of MDS matrices over $\mathrm{GL}_4(\mathbb{F}_2^{10})$

Modulus	Circulant		Hadamard-Normal		Hadamard-Involutory	
	Matrix	g-XOR	Matrix	g-XOR	Matrix	g-XOR
0x409	Circ(1,1,0x081,0x080)	110	Had(1,0x081,0x020,0x024)	97	Had(1,0x008,0x081,0x089)	116
	Circ(1,1,0x008,0x020)	110	Had(1,0x102,0x100,0x205)	108	Had(1,0x008,0x040,0x048)	118
	Circ(1,1,0x020,0x120)	112	Had(1,0x204,0x003,0x200)	109	Had(1,0x002,0x204,0x206)	119
	Circ(1,1,0x100,0x200)	113	Had(1,0x004,0x102,0x008)	110	Had(1,0x102,0x020,0x122)	120
	Circ(1,1,0x204,0x020)	113	Had(1,0x081,0x091,0x090)	110	Had(1,0x002,0x102,0x100)	121
0x48b	Circ(1,1,0x245,0x244)	116	Had(1,0x002,0x004,0x005)	123	Had(1,0x002,0x245,0x247)	128
	Circ(1,1,0x002,0x003)	121	Had(1,0x004,0x003,0x005)	125	Had(1,0x245,0x367,0x122)	128
	Circ(1,1,0x008,0x009)	127	Had(1,0x245,0x003,0x241)	127	Had(1,0x245,0x004,0x241)	129
	Circ(1,1,0x002,0x244)	129	Had(1,0x004,0x003,0x241)	127	Had(1,0x002,0x367,0x365)	135
	Circ(1,1,0x367,0x366)	129	Had(1,0x002,0x245,0x244)	131	Had(1,0x002,0x010,0x012)	136

6 Conclusion and Future Work

In this paper, we present both theoretical and practical optimizations for the BP algorithm and its variants (A1 and A2). We also extend these optimizations to the BP-LSL algorithm, enabling efficient search for minimum g-XOR implementations under explicit depth constraints. For 32-dimensional dense matrices (density near 0.5), our approach fills the long-standing research gap in [LSL+19, LXZZ21] by providing the first practical evaluations of A1 and generating low-latency g-XOR sequences with BP-LSL for such instances. For higher-dimensional matrices, our GPU-accelerated framework significantly expands the solvable boundary of the BP algorithm family at a reasonable time cost.

Table 9: Results on construction of MDS matrices over $\mathrm{GL}_4(\mathbb{F}_2^{12})$

Modulus	Circulant		Hadamard-Normal		Hadamard-Involutory	
	Matrix	g-XOR	Matrix	g-XOR	Matrix	g-XOR
0x1021	Circ(1,1,0x081,0x080)	132	Had(1,0x081,0x040,0x042)	118	Had(1,0x002,0x810,0x812)	139
	Circ(1,1,0x020,0x040)	133	Had(1,0x204,0x200,0x811)	129	Had(1,0x810,0x040,0x850)	144
	Circ(1,1,0x002,0x850)	136	Had(1,0x810,0x003,0x814)	131	Had(1,0x002,0x102,0x100)	145
	Circ(1,1,0x810,0x811)	136	Had(1,0x002,0x811,0x40a)	131	Had(1,0x004,0x408,0x40c)	145
	Circ(1,1,0x810,0x850)	136	Had(1,0x408,0x003,0x400)	133	Had(1,0x004,0x204,0x200)	146
0x1183	Circ(1,1,0x8c1,0x8c0)	138	Had(1,0x002,0x8c1,0x8c0)	154	Had(1,0x002,0x8c1,0x8c3)	147
	Circ(1,1,0x002,0x003)	141	Had(1,0x002,0x8c1,0x003)	158	Had(1,0x8c1,0xca1,0x460)	152
	Circ(1,1,0x8c0,0x003)	151	Had(1,0x8c1,0x010,0x011)	160	Had(1,0x002,0x004,0x006)	152
	Circ(1,1,0xca1,0xca0)	151	Had(1,0x002,0x8c1,0x460)	162	Had(1,0xca1,0xe91,0x230)	155
	Circ(1,1,0x004,0x005)	151	Had(1,0x8c1,0x008,0x8c0)	162	Had(1,0x002,0x022,0x020)	158

Table 10: Results on construction of MDS matrices over $\mathrm{GL}_4(\mathbb{F}_2^{16})$

Modulus	Circulant		Hadamard-Normal		Hadamard-Involutory	
	Matrix	g-XOR	Matrix	g-XOR	Matrix	g-XOR
0x1002d	Circ(1,1,0x8016,0x8017)	184	Had(1,0x400b,0xa013,0x2005)	187	Had(1,0x0002,0x8016,0x8014)	197
	Circ(1,1,0x0002,0x0003)	186	Had(1,0x0002,0x0004,0x0005)	195	Had(1,0x0002,0x0004,0x0006)	200
	Circ(1,1,0x400b,0x400a)	188	Had(1,0x8016,0x400b,0x400a)	196	Had(1,0x8016,0x400b,0xc01d)	205
	Circ(1,1,0x0002,0x8014)	195	Had(1,0x0002,0x8016,0x0003)	199	Had(1,0x8016,0xa013,0x2005)	206
	Circ(1,1,0x0003,0x8017)	196	Had(1,0x8016,0x8017,0x2005)	203	Had(1,0x0002,0x400b,0x4009)	208
0x10129	Circ(1,1,0x0002,0x0003)	186	Had(1,0x404a,0x8095,0x4048)	187	Had(1,0x0002,0x8094,0x8096)	197
	Circ(1,1,0x8094,0x8095)	189	Had(1,0x8094,0x0003,0x8090)	190	Had(1,0x0002,0x0004,0x0006)	203
	Circ(1,1,0x404a,0x404b)	195	Had(1,0x0002,0x8095,0x4048)	191	Had(1,0x8094,0x404a,0xc0de)	206
	Circ(1,1,0x2025,0x2024)	195	Had(1,0x8094,0x404a,0x404b)	196	Had(1,0x0002,0x404a,0x4048)	207
	Circ(1,1,0x0004,0x0005)	198	Had(1,0x0002,0x0004,0x0005)	199	Had(1,0x8094,0x0004,0x8090)	209

Our GPU implementation combines algorithmic refinements with parallel acceleration. By using a meet-in-the-middle (MITM) strategy and eliminating redundant computation, we reduce theoretical complexity while exploiting GPU parallelism to accelerate k-XOR sum enumeration and binary search. These optimizations significantly improve the performance of BP, A1, A2, and BP-LSL while maintaining solution quality, enabling previously infeasible 32-dimensional dense instances from [LSL+19, LXZZ21] to be solved within hours. Furthermore, we conduct heuristic searches for MDS matrices in dimension 40 or more and report corresponding experimental results on these constructions.

For future work, we identify two possible directions. One direction is to integrate our implementation into existing optimization frameworks, such as [LXZZ21] and [SYL25], to further improve the solution quality and yield shorter g-XOR sequences for higher-dimensional linear layers. Second, we plan to investigate advanced memory optimization techniques. While our meet-in-the-middle (MITM) strategy significantly accelerates the search process, it naturally introduces additional memory overhead. Developing efficient candidate pruning and memory management strategies will help reduce GPU memory consumption, thereby improving scalability and extending the practical solvable range of BP-like heuristic algorithms for SLP or SLPD.

References

[BFI19] Subhadeep Banik, Yuki Funabiki, and Takanori Isobe. More results on shortest linear programs. In Nuttapong Attrapadung and Takeshi Yagi, editors, *IWSEC 19*, volume 11689 of *LNCS*, pages 109–128. Springer, Cham, August 2019.

[BFI21] Subhadeep Banik, Yuki Funabiki, and Takanori Isobe. Further results on efficient implementations of block cipher linear layers. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 104(1):213–225, 2021.

[BKL16] Christof Beierle, Thorsten Kranz, and Gregor Leander. Lightweight multiplication in GF(2^n) with applications to MDS matrices. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 625–653. Springer, Berlin, Heidelberg, August 2016.

[BMP08] Joan Boyar, Philip Matthews, and René Peralta. On the shortest linear straight-line program for computing linear forms. In Edward Ochmanski and Jerzy Tyszkiewicz, editors, *International Symposium on Mathematical Foundations of Computer Science*, volume 5162 of *Lecture Notes in Computer Science*, pages 168–179. Springer, 2008.

[BMP13] Joan Boyar, Philip Matthews, and René Peralta. Logic minimization techniques with applications to cryptology. *Journal of Cryptology*, 26(2):280–312, April 2013.

[BNN+10] Paulo S. L. M. Barreto, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Elmar Tischhauser. Whirlwind: a new cryptographic hash function. *DCC*, 56(2-3):141–162, 2010.

[BP10] Joan Boyar and René Peralta. A new combinational logic minimization technique with applications to cryptology. In Paola Festa, editor, *International Symposium on Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science*, pages 178–189. Springer, 2010.

[FSK10] Carsten Fuhs and Peter Schneider-Kamp. Synthesizing shortest linear straight-line programs over gf(2) using sat. In Ofer Strichman and Stefan Szeider, editors, *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing*, volume 6175 of *Lecture Notes in Computer Science*, page 71–84, Berlin, Heidelberg, 2010. Springer-Verlag.

[GR13] Kishan Chand Gupta and Indranil Ghosh Ray. On constructions of involutory MDS matrices. In Amr Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien, editors, *AFRICACRYPT 13*, volume 7918 of *LNCS*, pages 43–60. Springer, Berlin, Heidelberg, June 2013.

[HDRM23] Solane El Hirch, Joan Daemen, Raghvendra Rohit, and Rusydi H. Makarim. Twin column parity mixers and gaston - A new mixing layer and permutation. volume 14083 of *Lecture Notes in Computer Science*, pages 475–506. Springer, 2023.

[JBKK25] Yongjin Jeon, Seungjun Baek, Giyoon Kim, and Jongsung Kim. A framework for generating S-box circuits with boyar-peralta algorithm-based heuristics, and its applications to AES, SNOW3G, and saturnin. *IACR TCHES*, 2025(1):586–631, 2025.

[JPST17] Jérémy Jean, Thomas Peyrin, Siang Meng Sim, and Jade Tourteaux. Optimizing implementations of lightweight building blocks. *IACR Trans. Symm. Cryptol.*, 2017(4):130–168, 2017.

[JV02] Daemen Joan and Rijmen Vincent. The design of rijndael: Aes-the advanced encryption standard. *Information Security and Cryptography*, 196:1–253, 2002.

[KLSW17] Thorsten Kranz, Gregor Leander, Ko Stoffelen, and Friedrich Wiemer. Shorter linear straight-line programs for MDS matrices. *IACR Trans. Symm. Cryptol.*, 2017(4):188–211, 2017.

[KPPY14] Khoongming Khoo, Thomas Peyrin, Axel York Poschmann, and Huihui Yap. FOAM: Searching for hardware-optimal SPN structures and components with a fair comparison. In Lejla Batina and Matthew Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 433–450. Springer, Berlin, Heidelberg, September 2014.

[LP07] G. Leander and A. Poschmann. On the classification of 4 bit s-boxes. In Claude Carlet and Berk Sunar, editors, *Arithmetic of Finite Fields*, volume 4547 of *Lecture Notes in Computer Science*, pages 159–176. Springer Berlin Heidelberg, 2007.

[LS16] Meicheng Liu and Siang Meng Sim. Lightweight MDS generalized circulant matrices. In Thomas Peyrin, editor, *FSE 2016*, volume 9783 of *LNCS*, pages 101–120. Springer, Berlin, Heidelberg, March 2016.

[LSL+19] Shun Li, Siwei Sun, Chaoyun Li, Zihao Wei, and Lei Hu. Constructing low-latency involutory MDS matrices with lightweight circuits. *IACR Trans. Symm. Cryptol.*, 2019(1):84–117, 2019.

[LWF+22] Qun Liu, Weijia Wang, Yanhong Fan, Lixuan Wu, Ling Sun, and Meiqin Wang. Towards low-latency implementation of linear layers. *IACR Trans. Symm. Cryptol.*, 2022(1):158–182, 2022.

[LXZZ21] Da Lin, Zejun Xiang, Xiangyong Zeng, and Shasha Zhang. A framework to optimize implementations of matrices. In Kenneth G. Paterson, editor, *CT-RSA 2021*, volume 12704 of *LNCS*, pages 609–632. Springer, Cham, May 2021.

[LZW23] Qun Liu, Zheng Zhao, and Meiqin Wang. Improved heuristics for low-latency implementations of linear layers. In Mike Rosulek, editor, *CT-RSA 2023*, volume 13871 of *LNCS*, pages 524–550. Springer, Cham, April 2023.

[ME19] Alexander Maximov and Patrik Ekdahl. New circuit minimization techniques for smaller and faster AES SBoxes. *IACR TCHES*, 2019(4):91–125, 2019.

[Paa97] Christof Paar. Optimized arithmetic for reed-solomon encoders. In *Proceedings of IEEE international symposium on information theory*, page 250. IEEE, 1997.

[Saa12] Markku-Juhani O. Saarinen. Cryptographic analysis of all 4×4-bit S-boxes. In Ali Miri and Serge Vaudenay, editors, *SAC 2011*, volume 7118 of *LNCS*, pages 118–133. Springer, Berlin, Heidelberg, August 2012.

[SFX23] Haotian Shi, Xiutao Feng, and Shengyuan Xu. A framework with improved heuristics to optimize low-latency implementations of linear layers. *IACR Trans. Symm. Cryptol.*, 2023(4):489–510, 2023.

[SKOP15] Siang Meng Sim, Khoongming Khoo, Frédérique E. Oggier, and Thomas Peyrin. Lightweight MDS involution matrices. In Gregor Leander, editor, *FSE 2015*, volume 9054 of *LNCS*, pages 471–493. Springer, Berlin, Heidelberg, March 2015.

[SKW+98] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Twofish: A 128-bit block cipher. *NIST AES Proposal*, 15(1):23–91, 1998.

[SS16] Sumanta Sarkar and Habeeb Syed. Lightweight diffusion layer: Importance of Toeplitz matrices. *IACR Trans. Symm. Cryptol.*, 2016(1):95–113, 2016.

[SS17] Sumanta Sarkar and Habeeb Syed. Analysis of toeplitz MDS matrices. In Josef Pieprzyk and Suriadi Suriadi, editors, *ACISP 17, Part II*, volume 10343 of *LNCS*, pages 3–18. Springer, Cham, July 2017.

[SYL25] Yao Sun, Runhe Yang, and Ting Li. Revisit the boyar-peralta algorithm to solve the shortest linear program problem. Cryptology ePrint Archive, Paper 2025/1493, 2025.

[TP19] Quan Quan Tan and Thomas Peyrin. Improved heuristics for short linear programs. *IACR TCHES*, 2020(1):203–230, 2019.

[VSP17] Andrea Visconti, Chiara Valentina Schiavo, and René Peralta. Improved upper bounds for the expected circuit complexity of dense systems of linear equations over GF(2). Cryptology ePrint Archive, Report 2017/194, 2017.

[WLTZ23] Shi Wang, Yongqiang Li, Shizhu Tian, and Xiangyong Zeng. Four by four mds matrices with the fewest xor gates based on words. *Advances in Mathematics of Communications*, 17(4):845–872, 2023.

[XZL+20] Zejun Xiang, Xiangyong Zeng, Da Lin, Zhenzhen Bao, and Shasha Zhang. Optimizing implementations of linear layers. *IACR Trans. Symm. Cryptol.*, 2020(2):120–145, 2020.

[YWS+24] Yufei Yuan, Wenling Wu, Tairong Shi, Lei Zhang, and Yu Zhang. A framework to improve the implementations of linear layers. *IACR Trans. Symm. Cryptol.*, 2024(2):322–347, 2024.

[ZBRL15] Wentao Zhang, Zhenzhen Bao, Vincent Rijmen, and Meicheng Liu. A new classification of 4-bit optimal S-boxes and its application to PRESENT, RECTANGLE and SPONGENT. In Gregor Leander, editor, *FSE 2015*, volume 9054 of *LNCS*, pages 494–515. Springer, Berlin, Heidelberg, March 2015.

[ZH23] Fuxin Zhang and Zhenyu Huang. Optimizing S-box implementations using SAT solvers: Revisited. Cryptology ePrint Archive, Report 2023/1721, 2023.

[ZWS18] Lijing Zhou, Licheng Wang, and Yiru Sun. On efficient constructions of lightweight MDS matrices. *IACR Trans. Symm. Cryptol.*, 2018(1):180–200, 2018.

A SLPD Solution of AES-InvMixColumn with 157 Steps

Table 11: Our 157-Step Implementation of AES-InvMixColumn with Depth 5

Step	Operation	Depth	Step	Operation	Depth	Step	Operation	Depth	Step	Operation	Depth
1	(32,5,21,–1)	1	40	(71,37,53,–1)	4	79	(110,30,92,–1)	3	118	(149,112,148,8)	5
2	(33,13,29,–1)	1	41	(72,5,41,–1)	2	80	(111,80,88,–1)	3	119	(150,64,106,–1)	3
3	(34,6,22,–1)	1	42	(73,13,72,–1)	3	81	(112,47,110,–1)	4	120	(151,59,150,25)	5
4	(35,14,30,–1)	1	43	(74,44,52,–1)	4	82	(113,36,64,–1)	3	121	(152,75,120,–1)	4
5	(36,32,33,–1)	2	44	(75,28,54,–1)	3	83	(114,72,81,–1)	3	122	(153,74,152,12)	5
6	(37,34,36,–1)	3	45	(76,2,57,–1)	3	84	(115,63,67,–1)	2	123	(154,26,105,–1)	4
7	(38,11,23,–1)	1	46	(77,63,64,–1)	2	85	(116,45,70,–1)	3	124	(155,95,154,10)	5
8	(39,7,27,–1)	1	47	(78,18,57,–1)	3	86	(117,64,67,–1)	2	125	(156,79,109,–1)	4
9	(40,19,31,–1)	1	48	(79,67,68,–1)	3	87	(118,69,102,–1)	3	126	(157,61,156,2)	5
10	(41,3,15,–1)	1	49	(80,34,49,–1)	2	88	(119,7,96,–1)	3	127	(158,76,118,–1)	4
11	(42,2,35,–1)	2	50	(81,4,46,–1)	2	89	(120,38,80,–1)	3	128	(159,62,158,27)	5
12	(43,9,25,–1)	1	51	(82,37,57,–1)	4	90	(121,36,98,–1)	3	129	(160,23,67,–1)	2
13	(44,1,17,–1)	1	52	(83,35,58,–1)	3	91	(122,53,121,–1)	4	130	(161,82,160,16)	5
14	(45,10,26,–1)	1	53	(84,73,83,–1)	4	92	(123,89,122,29)	5	131	(162,48,125,–1)	4
15	(46,12,28,–1)	1	54	(85,53,84,6)	5	93	(124,51,103,–1)	3	132	(163,71,162,13)	5
16	(47,0,16,–1)	1	55	(86,71,84,22)	5	94	(125,54,99,–1)	3	133	(164,91,107,–1)	3
17	(48,18,42,–1)	3	56	(87,12,20,–1)	1	95	(126,63,66,–1)	2	134	(165,65,164,23)	5
18	(49,4,20,–1)	1	57	(88,4,29,–1)	1	96	(127,51,81,–1)	3	135	(166,37,114,–1)	4
19	(50,8,24,–1)	1	58	(89,48,75,–1)	4	97	(128,70,93,–1)	3	136	(167,89,166,5)	5
20	(51,39,40,–1)	2	59	(90,17,34,–1)	2	98	(129,69,100,–1)	3	137	(168,92,104,–1)	4
21	(52,36,43,–1)	3	60	(91,14,55,–1)	2	99	(130,25,115,–1)	3	138	(169,65,168,15)	5
22	(53,38,51,–1)	3	61	(92,15,56,–1)	2	100	(131,59,130,9)	5	139	(170,42,79,–1)	4
23	(54,40,45,–1)	2	62	(93,11,26,–1)	1	101	(132,37,110,–1)	4	140	(171,97,170,18)	5
24	(55,15,31,–1)	1	63	(94,12,80,–1)	3	102	(133,65,132,31)	5	141	(172,17,126,–1)	3
25	(56,7,23,–1)	1	64	(95,48,77,–1)	4	103	(134,94,116,–1)	4	142	(173,62,172,1)	5
26	(57,47,50,–1)	2	65	(96,10,44,–1)	2	104	(135,74,134,28)	5	143	(174,48,127,–1)	4
27	(58,22,49,–1)	2	66	(97,25,90,–1)	3	105	(136,1,117,–1)	3	144	(175,74,174,20)	5
28	(59,37,44,–1)	4	67	(98,34,88,–1)	2	106	(137,62,136,17)	5	145	(176,76,119,–1)	4
29	(60,46,58,–1)	3	68	(99,13,87,–1)	2	107	(138,78,128,–1)	4	146	(177,71,176,3)	5
30	(61,34,48,–1)	4	69	(100,20,46,–1)	2	108	(139,59,138,19)	5	147	(178,7,66,–1)	2
31	(62,35,52,–1)	4	70	(101,1,2,–1)	1	109	(140,73,111,–1)	4	148	(179,82,178,0)	5
32	(63,8,23,–1)	1	71	(102,19,26,–1)	1	110	(141,89,140,21)	5	149	(180,48,129,–1)	4
33	(64,7,24,–1)	1	72	(103,3,10,–1)	1	111	(142,63,104,–1)	4	150	(181,74,180,4)	5
34	(65,32,60,–1)	4	73	(104,36,91,–1)	3	112	(143,112,142,24)	5	151	(182,96,97,–1)	4
35	(66,0,15,–1)	1	74	(105,9,90,–1)	3	113	(144,78,124,–1)	4	152	(183,95,182,26)	5
36	(67,16,31,–1)	1	75	(106,9,66,–1)	2	114	(145,62,144,11)	5	153	(184,91,108,–1)	3
37	(68,45,66,–1)	2	76	(107,7,35,–1)	2	115	(146,30,114,–1)	4	154	(185,65,184,7)	5
38	(69,38,41,–1)	2	77	(108,23,34,–1)	2	116	(147,122,146,14)	5	155	(186,14,53,–1)	4
39	(70,39,41,–1)	2	78	(109,9,101,–1)	2	117	(148,14,113,–1)	4	156	(187,98,114,–1)	4
									157	(188,186,187,30)	5