GPUを用いたハッシュ関数Keccak の高速化に関する研究

防衛大学校理工学研究科後期課程

電子情報工学系専攻・情報知能メディア学教育研究分野

グエン　ダット　トゥオン

令和 2 年 3 月

目　次

頁

第 1 章

序 論

1.1 研究の背景

ネットワークを介して通信するデータにおいては，個人情報や秘匿性の高い情報を扱う機会が増加している．これらの情報を保護するためには，暗号化技術やハッシュ関数を利用する必要がある．

パスワードの処理等に用いられるハッシュ関数は，同じ入力値からは必ず同じ値が得られる一方，少しでも異なる入力値からはまったく違う値が得られるという特徴がある．不可逆な一方向関数を含むため，ハッシュ値から入力値を割り出すことは簡単には出来ない．しかし，全数探索を行えば入力値を得ることが可能であるため，コンピュータの処理速度の向上により一部のハッシュ関数を用いたパスワードの安全性が低下してきている．例えば，2012 年にハッカーが LinkedIn に侵入し，650 万人分の暗号化パスワードを盗み，ロシアのハッカーフォーラムに掲載した [1][2]．このデータセットを分析した結果によると，約 90%のパスワードが 72 時間以内に解読可能であった [3]．

ハッシュ関数の種類によって，ハッシュ値のビット長が異なるが，ビット長が長いほど，ハッシュ値のとり得る範囲も広くなる．しかし，任意の入力に対して，ハッシュ値のとり得る範囲が限られているため，同じハッシュ値となる別の入力値が必ず存在する．これは衝突（collision）と呼ばれる．ハッシュ関数 MD4[4] や MD5[5] の解析では，少ない計算量，短い時間で衝突が見つけられることを Wang らが発表した [6][7]．この事実により，MD5 の安全性は低下し，使用されている多くの MD5 は SHA-1，SHA-2[8] に移行することになった．また，与えられたハッ

シュ値に対し，時間をかけて全ての入力候補をハッシュ化すれば，入力値を求めることができる．アルゴリズムやハッシュ長を考慮すれば，MD5 と SHA-1 に対する総当たり攻撃の効果が高いと予測できる．現在は未だ広く使用されてはいないが，MD5 や SHA-1 に対する攻撃の研究の進展に対応したものに SHA-3[9] があり，SHA-3 の原案となったものは Keccak[10] である．このハッシュはビット長が 224 から 512 ビットまで，または可変なハッシュ長を出力できるため，128 ビット長の MD5 や 160 ビット長の SHA-1 より安全である．

総当たり攻撃は全数探索であり，時間をかけて行えば必ずパスワードは見つかるものの，莫大な計算時間を要する場合，現実にはパスワードクラックは不可能と考えても良い．しかしコンピュータの処理速度が向上すれば，ハッシュ関数を用いたパスワードの安全性が低下する．現状では，手頃な値段で誰でも購入可能なグラフィック・カードでも GPGPU として使用可能となり，数値演算の処理速度が向上しているため，全数探索がほぼ不可能な状態から実行可能な状態へと少しずつ近づいているアルゴリズムもある．

認証や電子署名等，様々な応用においては，セキュリティレベルが高いものの，高速に計算可能なハッシュ関数が求められるが，その一方，ハッシュ関数を用いたパスワード管理の場合，高速化実装により計算時間が短縮でき，全数探索が可能となると攻撃者が有利になってしまう．

1.2　研究の目的

本研究では，ハッシュ関数 Keccak の一種である，512 ビットのハッシュ値を出力とする Keccak-512 を CUDA[11] を用いて GPU へ高速化実装を行い，それらの処理速度を測定した上でパスワード管理における安全性を総当たり攻撃の可能性と対策について議論する．

また，パスワードクラッキングに有用であるレインボーテーブル攻撃では，事前にレインボーテーブルの準備が必要である．本研究では，Keccak-512 に対応

するレインボーテーブルの生成を高速化実装し，生成されたレインボーテーブルを評価する．この結果を用いて攻撃の可能性と対策について議論を行う．

1.3 論文の構成

本論文の構成は次のとおりである．まず，第 2 章にて暗号学的ハッシュ関数の概要，そして研究対象であるハッシュ関数 Keccak-512 のアルゴリズムを紹介する．さらに，ハッシュ関数に対する攻撃方法や過去の分析データ等，パスワードクラッキングの概要についてもここで説明する．第 2 章の最後に，レインボーテーブル攻撃の概要について紹介する．第 3 章では GPU のアーキテクチャとそれを利用するための開発環境である CUDA の概要，特に CUDA プログラムに必要な構造及び各種メモリを中心にして説明する．第 4 章では，ハッシュ値をより高速に計算するための CUDA を用いた GPU への高速化実装法を提案し，実装・評価結果を先行研究と関連実装と比較を行う．総当たり攻撃への対策として知られる複数回ハッシュの効果についてもここで述べる．第 5 章では，レインボーテーブルの生成の高速化手法，還元関数の改良法，そして生成されたテーブルの評価，攻撃効果の予測・議論を行う．最後に第 6 章では，本研究の内容をまとめ，結論を述べる．

第 2 章

ハッシュ関数

本章では，ハッシュ関数の概要, 安全性低下の状況について説明する．そして，実装対象としたハッシュ関数 Keccak の特徴，アルゴリズムを示す．

2.1　概要

ハッシュ関数は，任意の長さの入力メッセージに対し，固定のビット数のメッセージダイジェスト，またはハッシュ値を出力する．同じハッシュ値となる 2 つの入力メッセージを作成すること，または，あらかじめ指定されたハッシュ値となる入力メッセージを作成することは困難である．この特徴により，ハッシュ関数は，パスワードの管理や認証，電子署名等に適用されている．

ハッシュ関数の安全性については，原像計算困難性，第 2 原像計算困難性及び衝突困難性の 3 つの特徴に依拠している．このうち，原像計算困難性とは，与えられたハッシュ値に対して，そのハッシュ値を出力するようなハッシュ関数への入力を求めることが困難である特徴を示してる．すなわち，与えられたハッシュ値 H を出力とするメッセージ M を見つけることが計算量的に困難であることに対応する．第 2 原像計算困難性とは，与えられた入力値に対して，その入力値をハッシュ関数へ入力したときのハッシュ値と同じハッシュ値を出力する入力値を求めることが困難である特徴を示してる．すなわち，ある既知のメッセージ M と M に対するハッシュ値が与えられたとき，同じハッシュ値を出力するメッセージ M' を見つけることが計算量的に困難であることに対応する．衝突困難性とは，同じハッシュ値を与える二つの入力値 M と M' を求めることが計算量的に困難であること特徴を示してる．

2.2 ハッシュ関数の利用

ハッシュ関数は，メッセージダイジェスト（ハッシュ値）を計算するという目的で開発された．他に，暗号スキームまたは暗号アルゴリズムの構成要素として利用されることが多い．ハッシュ関数の用途としては下記のようなものが代表的である．

- デジタル署名 (ほぼ全てのアルゴリズム)
- 公開鍵暗号 (例: RSA-OAEP[12], RSAES-PKCS1-v1_5 [13] などのスキーム)
- 擬似乱数生成器 (例: FIPS 186-2[14])
- メッセージ認証コード (例: HMAC[15])
- ブロック暗号 (例: SHACAL-2[16], BEAR, LION[17])
- ストリーム暗号 (例: SEAL[18][19])

2.3 代表的なハッシュ関数

ハッシュ関数は様々なものが提案されており，代表的なハッシュ関数 MD4, MD5, RIPEMD[20], SHA-1, SHA-2, SHA-3 について，それらの概要を表 2.1 ににまとめる．

2.4 ハッシュ関数 Keccak

2.4.1 概要

米国の国立標準技術研究所 (NIST) は，2012 年 10 月 2 日に次世代の暗号学的ハッシュ関数の標準を決める SHA-3 候補から，Keccak を選定した．Keccak は，STMicroelectronics の Guido Bertoni，Joan Daemen 及び Gilles Van Assche と NXP Semiconductor の Michael　Peeters が設計したスポンジ構造を有するハッ

表 **2.1** 代表的なハッシュ関数の概要.

名称	ハッシュ長 (bits)	概要
MD4	128	・Rivest が 1990 年に考案 ・2004 年にハッシュ衝突が発見された
MD5	128	・Rivest が 1991 年に考案（MD4 の安全性向上） ・安全性の観点から推奨暗号リストから外された
RIPEMD	128 160	・Dobbertin が 1996 年に考案 ・RIPEMD-160 は最も広く用いられている
SHA-1	160	・NIST によって 1995 年に考案，標準化された ・不正や解読のリスクから SHA-2 へ移行
SHA-2	224 256 384 512	・NIST によって 2001 年に考案，標準化された ・現在広く用いられている
SHA-3	224 256 384 512 可変	・Bertoni らによって 2008 年に考案 ・2012 年に Keccak がコンペティションの勝者として選ばれ，2015 年に正式版が FIPS PUB 202 として公表された

シュ関数である．また，Keccak は MD5 や SHA-1 に対する攻撃の研究進展に対応したものである．

2.4.2 スポンジ構造

スポンジ構造は，固定長の permutation と padding に基づいた利用モードの一種である．このスポンジ構造を図 2.1 に示す [21].

スポンジ構造は，absorbing と squeezing の 2 つのフェーズに分けることがで

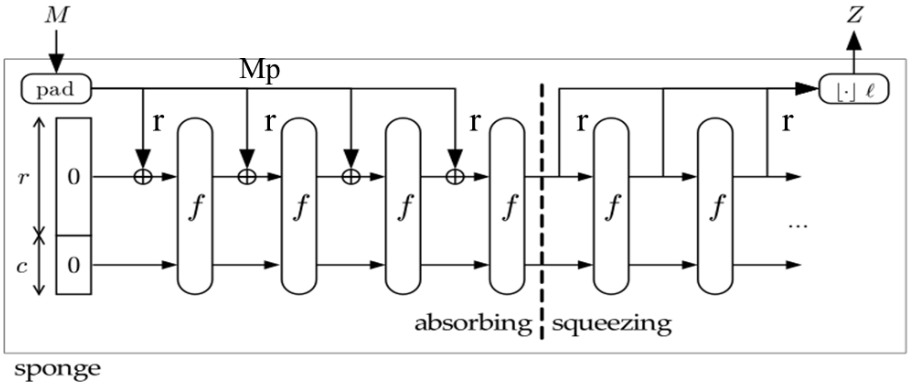


図 2.1　スポンジ構造 [21].

きる．absorbing では，メッセージ M に対し，パディング処理 (pad) を行い，パディング後のメッセージデータ Mp を r[bit] ごとのデータに分割し，内部状態の r[bit] のデータとの XOR 演算の後に Keccak-f 置換関数に入力する．squeezing では，必要な長さ l まで（求めたいハッシュ値 Z のビット長，SHA3-512 の場合は 512 ビット）Keccak-f 置換関数を繰り返し実行させ，逐次その実行結果から r [bit] を取り出す．[22]

ただし，r はビットレートであり，c はメッセージが持つ特徴を外部に漏らさない度合い，すなわちキャパシティである．図 2.1 のように，ここで r と c の初期値は 0 とする．

2.4.3　Keccak-f 置換関数

Keccak の基本となる撹拌関数は，7 つの Keccak-f[b]（b ∈ 25, 50, 100, 200, 400, 800, 1600）で表される Keccak-f 関数の集合から選ばれる．ビット数 b は撹拌幅であり，その関数が保持する内部状態の大きさに対応する．SHA3-512 の Keccak で使用する置換関数は Keccak-f[1600] であり，24 ラウンドの処理を行う．[?]

図 2.2 に示すように，Keccak-f[1600] 置換関数の内部状態は 3 次元で表され，5 × 5 × 64 の配列（state）から構成される．x，y，z 軸はそれぞれ row（行），column（列），lane（レーン）に対応する．x-y 平面を slice（スライス），x-z 平面を plane（プレーン），y-z 平面を sheet（シート）とそれぞれ呼ぶ．Keccak-f[1600] の各

lane は 64 ビットで構成され，64 ビットプロセッサで実装されたとき，64 ビットの CPU レジスタに格納することができる．

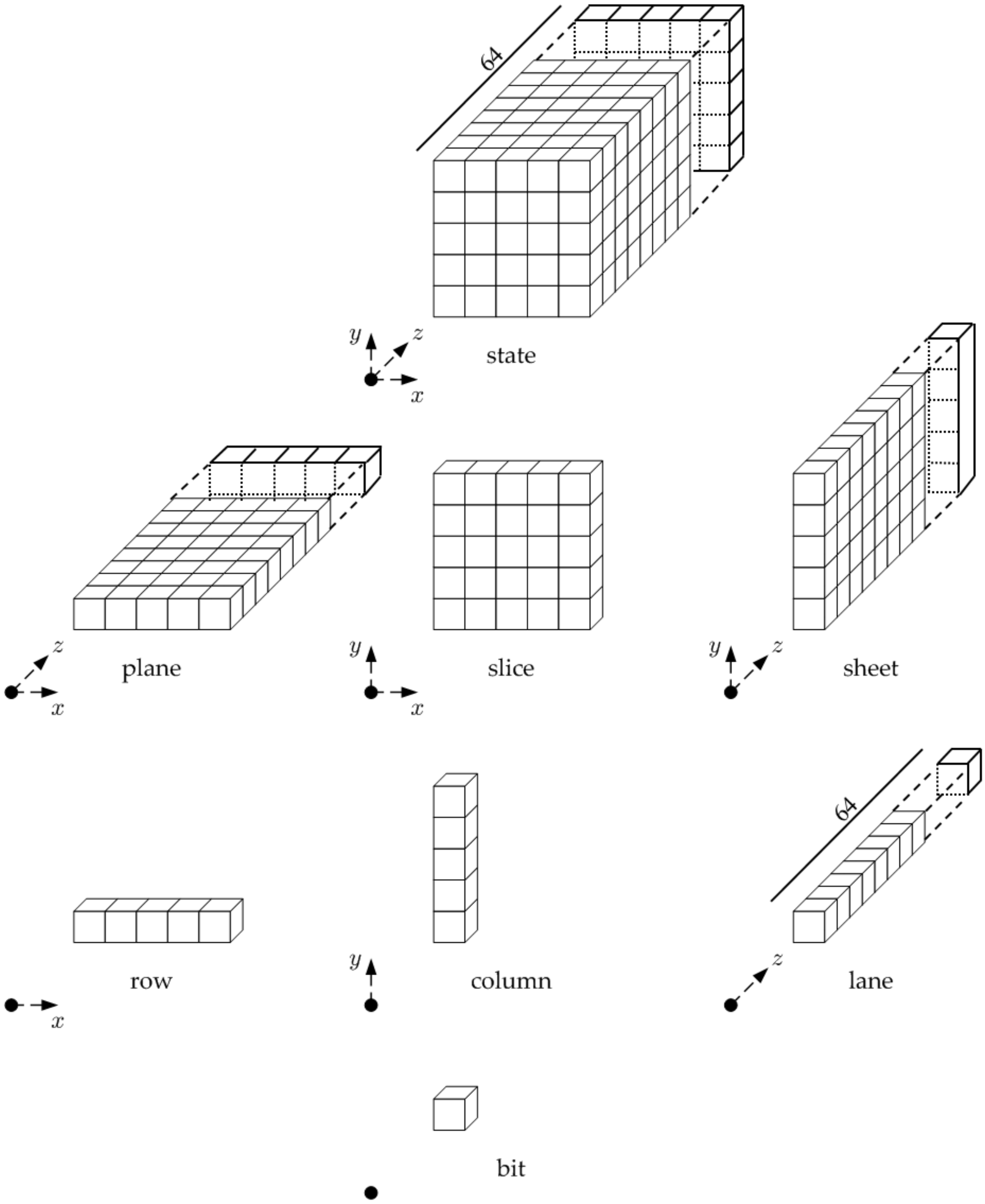


図 2.2　Keccak-f[1600] の内部状態（[23] を元とする）．

Keccak-f 置換関数は θ，ρ，π，χ の 4 つのステップとラウンド定数との XOR 処理を行う ι ステップにより，3 次元の state を計算する．

図 2.3 に示すように，θ ステップでは位置をずらした 2 本の column の 5bit を XOR 演算で足し（$\sum$），それを目的の bit に XOR で足し込む．一般には，全ての column y の row x にあるビットに対し，次の処理を行うものである．

$$C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4] \qquad x, y : \{0,1,2,3,4\} \quad (2.4.1)$$

$$D[x] = C[x-1] \oplus rot(C[x+1],1) \qquad x, y : \{0,1,2,3,4\} \quad (2.4.2)$$

$$A[x,y] = A[x,y] \oplus D[x] \qquad x, y : \{0,1,2,3,4\} \quad (2.4.3)$$

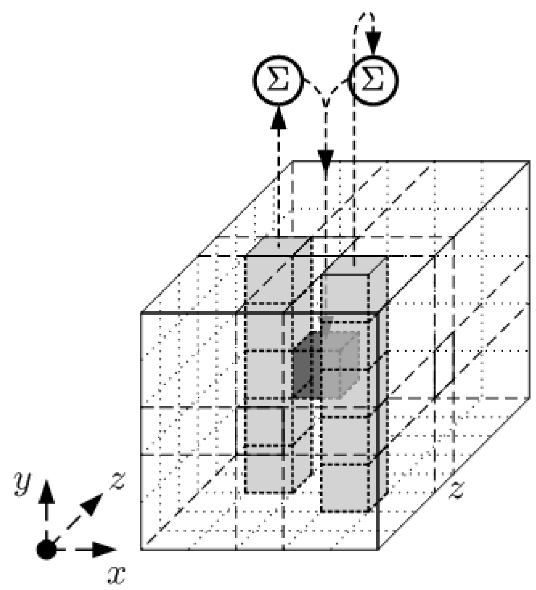


図 2.3　θ ステップ [23].

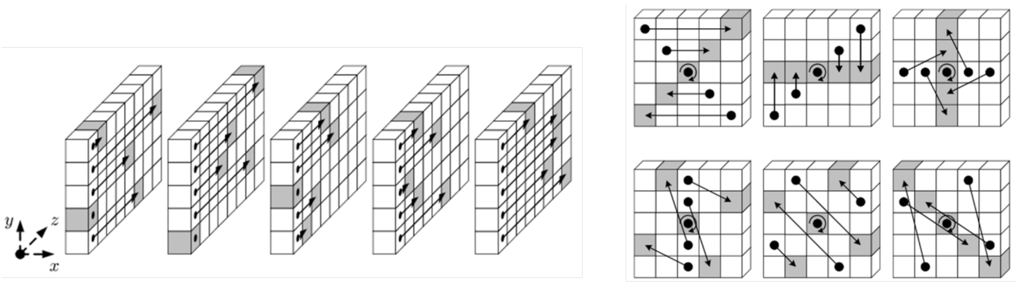
ただし，$A[x,y]$ はその状態における特定の lane を示し，$C[x]$，$D[x]$ は中間的な変数である．$rot(C[i],r)$ は lane サイズを法として，位置 i のビットを位置 $i+r$ に移動する右巡回シフト演算である．また，インデックスを持つ全ての演算は 5 を法として行われる．

次に，ρ ステップでは，sheet ごとに lane の方向のビット移動を行い，π ステッ

プでは slice ごとにビット移動を行う．図 2.4 に示すように，すべての sheet，slice に対し移動を行う．実装プログラムでは，ρ と π ステップを合わせて次の演算を行うものである．

$$B[y, 2x+3y] = rot(A[x,y], r[x,y]) \qquad\qquad x, y : \{0,1,2,3,4\} \tag{2.4.4}$$

ただし，$B[x,y]$ は，$C[x]$，$D[x]$ と同様，中間的な変数であり，$r[x,y]$ はローテーションのオフセット値であり，表 2.2 に示すように与えられる．



ρ ステップ　　　　π ステップ

図 2.4　ρ と π ステップ [23].

表 2.2　ローテーションのオフセット値 $r[x,y]$.

y \ x	0	1	2	3	4
0	0	1	62	28	27
1	36	44	6	55	20
2	3	10	43	25	39
3	41	45	15	21	8
4	18	2	61	56	14

χ ステップでは，各行ごとに，図 2.5 に示す論理演算を行う次のような処理で

ある．

$$A[x,y] = B[x,y] \oplus (\overline{B[x+1,y]} \cdot B[x+2,y]) \qquad x,y : \{0,1,2,3,4\} \quad (2.4.5)$$

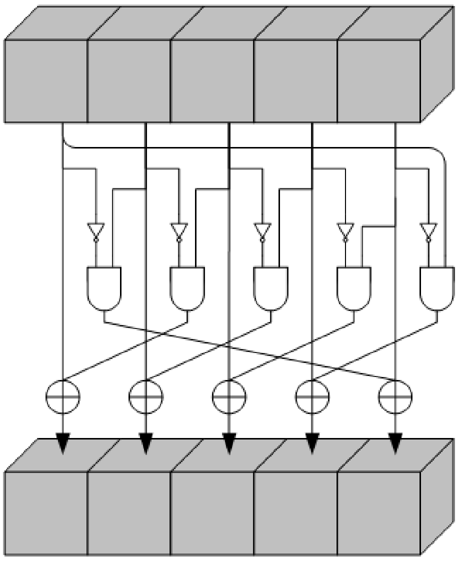


図 2.5 χ ステップ [23].

最後に ι ステップでは，ラウンド定数 $RC[i]$ を用いて，ラウンド定数と state 全体のビットの XOR をとる次のような処理である．

$$A[0,0] = A[0,0] \oplus RC[i] \qquad (2.4.6)$$

ここでラウンド定数 $RC[i]$ は，以下の表 2.3 に示すように与えられる [?][23].

2.5 Keccak と SHA-3 の関係

SHA-3 は Keccak とは違い，パディング処理を実行する前に入力メッセージの末尾に「01」の 2 ビットを追加する．そのため，同じ入力メッセージに対し Keccak と SHA-3 の出力ハッシュ値は同じものではない．

ここで Keccak と SHA-3 の違いについて説明する．Keccak は前に示したよう

表 **2.3**　ラウンド定数 $RC[i]$.

$RC[0]$	$0x0000000000000001$	$RC[12]$	$0x000000008000808B$
$RC[1]$	$0x0000000000008082$	$RC[13]$	$0x800000000000008B$
$RC[2]$	$0x800000000000808A$	$RC[14]$	$0x8000000000008089$
$RC[3]$	$0x8000000080008000$	$RC[15]$	$0x8000000000008003$
$RC[4]$	$0x000000000000808B$	$RC[16]$	$0x8000000000008002$
$RC[5]$	$0x0000000080000001$	$RC[17]$	$0x8000000000000080$
$RC[6]$	$0x8000000080008081$	$RC[18]$	$0x000000000000800A$
$RC[7]$	$0x8000000000008009$	$RC[19]$	$0x800000008000000A$
$RC[8]$	$0x000000000000008A$	$RC[20]$	$0x8000000080008081$
$RC[9]$	$0x0000000000000088$	$RC[21]$	$0x8000000000008080$
$RC[10]$	$0x0000000080008009$	$RC[22]$	$0x0000000080000001$
$RC[11]$	$0x000000008000000A$	$RC[23]$	$0x8000000080008008$

に，Bertoni らによって 2008 年に考案され，2012 年にコンペティションの勝者として選ばれ，ハッシュ関数 SHA-3 の原案であった．また，SHA-3 の正式版は 2015 年に FIPS PUB 202 [24] として公表された．これらは同一なものではなく，Keccak に少し変更を加えたものが SHA-3 である．

例えば，SHA3-512 では，入力メッセージ M に対し，KECCAK[c] 関数を用いて，次の式のように計算を行う：

SHA3-512(M) = KECCAK[1024](M||01, 512).

2.6　ハッシュ関数に対する攻撃

2.6.1　概要

第 1 章で述べたように，秘密情報を守るためにハッシュ関数や様々な暗号化技術が必要である．そのため，ハッシュ関数の安全性についての研究も盛んである．ハッシュ関数における脆弱性や攻撃方法として既に知られているのは，衝突（コ

リジョン）攻撃，差分攻撃，サイドチャネル攻撃等である．特にハッシュ関数を利用したパスワード管理の場合は，辞書攻撃，誕生日攻撃，総当たり攻撃（ブルートフォースアタック）やレインボーテーブルを用いた攻撃などが存在する．

2004 年以前の研究には，MD5 の脆弱性について数件の報告はあったが，MD5 への攻撃の報告はなかった．しかし，2004 年 8 月に MD5 への攻撃成功の速報が発表され [25]，2005 年に Wang らが MD5 や MD ベース型ハッシュ関数への衝突攻撃の詳細について公表した [26]．それから，MD5 やその他のハッシュ関数の衝突攻撃について，多数の改良論文が発表されている．特に，2015 年に Karpman らの研究報告 [27] では，SHA-1 に対し，Free-Start 衝突攻撃の条件で 76 段を約 5 日で攻撃できることを示した [28]．Free-Start とは仕様で固定とされている初期ベクターを可変とすることで難度を下げた攻撃法である．これは SHA-1 の衝突発見に直接つながるものではないが，SHA-1 の衝突発見に至るまでの節目となる出来事の 1 つであり，近い将来に SHA-1 の衝突が発見されるという予測を強く裏付けるものとされている [29]．渡辺ら [30] の研究報告では，暗号危殆化の問題と関連して，共通鍵暗号における安全性評価の最新動向と，暗号技術の脆弱性が発表された．

2.6.2　ハッシュ関数に対する汎用の攻撃

ハッシュ関数 H のそれぞれの攻撃に対する強度には上限が存在し，その攻撃計算量の上限はハッシュ長 n にのみ依存する．それぞれの攻撃方法とその計算量は以下のようになる．

- 第 1 原像探索攻撃 (Pre-image Attack)

  未知のメッセージ M に対するハッシュ値が与えられた時，ハッシュ値が一致する，すなわち $H(M) = H(M')$ を満たすようなメッセージ M' を探索する攻撃のことである．n ビットデータに対する全数探索の計算量は $\Omega(2^n)$ となる．

- 第 2 原像探索攻撃 (Second Pre-image Attack)

  既知のメッセージ M と M に対するハッシュ値が与えられた時，ハッシュ値が一致する，すなわち $H(M) = H(M')$ を満たすような別のメッセージ M' を探索する攻撃のことである．n ビットデータに対する全数探索の計算量 $\Omega(2^n)$ となる．

- 衝突攻撃 (Collision Attack)

  ハッシュ値が一致する，すなわち $H(M) = H(M')$ を満たすような異なる 2 つのメッセージ M と M' を探索する攻撃のことである．n ビットデータに対する衝突攻撃の計算量は $\Omega(2^{n/2})$ となる．

2.6.3 パスワードクラッキング

ハッシュ関数 H への攻撃のうち，特にパスワードを標的としたものをパスワードクラッキングと呼ぶ．主に類推攻撃，総当たり攻撃，辞書攻撃，及びレインボーテーブルを用いた攻撃が存在する．

類推攻撃とは，個人情報に関する知識からパスワードを類推する攻撃である．例えば自分や友人，身内の出身地，誕生日等の情報をパスワードとして使用する．類推攻撃は，誕生日攻撃として知られることが多い．総当たり攻撃（ブルートフォースアタック）は，全てのパスワード候補を試す攻撃方法である．辞書攻撃では，良く使われるパスワード候補を辞書的にファイルに登録し，その登録したファイルを用いて攻撃を行う．レインボーテーブルを用いた攻撃は，ハッシュ値から平文を得るために使われるテクニックであり，特殊なテーブルを使用して表引きを繰り返し行うことで，時間と空間のトレードオフを実現する技術である．

それぞれの攻撃方法で使用するメモリ量，計算量，探索時間の比較を表 2.4 に示す．

表 2.4　各攻撃方法の比較．

攻撃方法	メモリ量	計算量	探索時間
類推攻撃	×	△	○
総当たり攻撃	○	×	×
辞書攻撃	×	△	○
レインボーテーブル	△	△	○

2.6.4　レインボーテーブルの概要

レインボーテーブルを作成，または使用するにあたって，必ず対応するハッシュ関数 H，還元関数 R，そしてそのレインボーテーブルで対象とするパスワード候補の情報が存在する．レインボーテーブルのイメージを図 2.6 に示す．

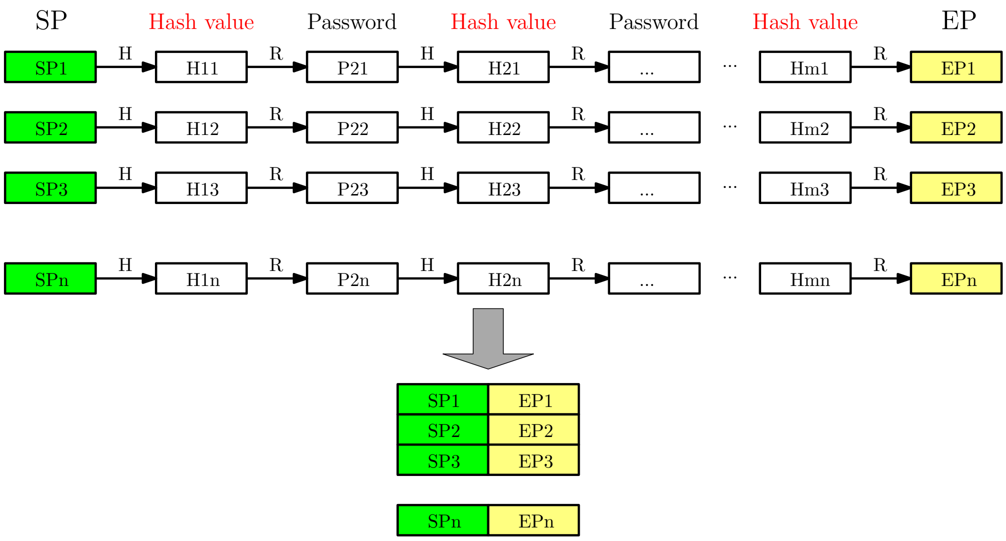


図 2.6　レインボーテーブルのイメージ．

ここで，例えば SP1 から EP1 までの SP1，H11，P21，H21，...，Hm1，EP1 が 1 つのチェーンになる．各 SP，またはパスワード候補である平文をハッシュ関数の入力として，ハッシュ計算を行い，ハッシュ値を取得する．その得られた

ハッシュ値と還元関数を用いて次のパスワード候補の平文が生成される．最初と最後の情報だけがあれば，元のチェーンを復元できるため，それら 2 つの情報を保存することでチェーン全体を保存するのに必要なメモリ量を減少することが可能である．レインボーテーブルの詳細について，以下に説明する．

(1)　ハッシュチェーン

パスワードハッシュ関数 H とパスワード P の集合（有限セット）があると想定する．目標は，ハッシュ関数の出力 h が与えられたときに，$H(p) = h$ となるパスワード p を見つけるか，P の集合にそのようなパスワード p がないことを確認することである．これを行う最も簡単な方法は，P の全ての p に対し，$H(p)$ を計算することであるが，この全ての計算結果のテーブルを格納するには $n(H\_len+p\_len)$ ビットのスペースが必要になる．ここで，n はパスワード P の数，H_len は出力ハッシュ長，p_len は p のビット長である．ハッシュチェーンは，このスペース要件を減らすための手法として開発された．そのアイデアとは，ハッシュ値を P の値にマップする還元関数 R を定義することであり，ハッシュ関数と還元関数とを交互に適用することにより，パスワードとハッシュ値が交互に現れるチェーンが形成される．例えば，P が 6 文字のパスワードのセットで，ハッシュ関数が Keccak-512 の場合，チェーンの 1 つは次のように生成される（ハッシュ値の一部は省略）．

$\texttt{XB4S4l} \xrightarrow{Keccak} \texttt{db43601ec3df...2ccf} \xrightarrow{\text{還元関数}} \texttt{fIEjLY} \xrightarrow{Keccak} \texttt{76893b6bfa08...0723} \xrightarrow{\text{還元関数}} \texttt{7TciaV} \xrightarrow{Keccak} \texttt{fe5809892548...0836} \xrightarrow{\text{還元関数}} \texttt{nLkF3T} \xrightarrow{Keccak} \texttt{be7fa8aa4cfa...3e55} \xrightarrow{\text{還元関数}} \texttt{ocX7UX}$

ここでの還元関数は，ハッシュ値から新しいパスワード候補を生成する関数であり，2 つのパスワード候補･ハッシュ値のペアを結び付ける役になる．チェーンの最初のパスワード候補は Starting Point （SP）と呼ばれ，最後のパスワード候補は Endpoint（EP）と呼ばれる．上記の例では，“XB4S4l” が SP であり，“ocX7UX” が EP となる．SP，ハッシュ関数 H，及び還元関数からチェーンの全てのパスワー

ド候補を計算できるため，チェーンでは，SP と EP のみを保存し，他のパスワードとハッシュ値は保存する必要がない．この例では，チェーンの長さが 4 であり，チェーンが長いほど，より多くのメモリを節約できる．

(2)　レインボーテーブル

レインボーテーブルは，事前に計算された多くのハッシュチェーンから構成され，2003 年に Philippe らの論文 “Making a Faster Cryptanalytic Time-Memory Trade-Off” [31] により提案された．可能性のあるすべてのパスワード（パスワード候補）のハッシュを構築するために必要なメモリと比較して，ハッシュチェーンはメモリを削減できる代わりに，パスワードを取得するのにより多くの時間を必要とする．

表 2.5　レインボーテーブルの例．

SP	EP
XB4S4l	ocX7UX
Vw09eq	fwEk7g
2a2XX3	cbNPTG
1itPhr	VdGUio
5c9H18	kdmipJ
...	...

レインボーテーブルの一例を表 2.5 に示す．ここで 1 つの SP-EP のペアが 1 つのチェーンを意味する．SP と EP の間に隠れたハッシュ値の数がチェーンの長さであり，SP または EP の数がチェーンの数である．

レインボーテーブルを用いて開発されたパスワードクラッキングツールは，Rainbow crack [32]，rcracki_mt [33]，Ophcrack [34]，Elcmsoft [35]，L0phtCrack [36] などが存在する． その中でも，Rainbow Crack は最も多く引用されるツールである．このソフトウェアでは，LM，NTLM，MD5，SHA-1，及び SHA256

ハッシュ関数のレインボーテーブルを作成できる．Keccak ハッシュ関数に対応するレインボーテーブルの生成は，現時点ではまだ見つかっていない．

2.6.5 レインボーテーブルを用いたパスワードクラッキング

事前に準備されたレインボーテーブルを用いてパスワードを探索する一例について説明する．

ハッシュ値が "fe5809"，ハッシュ関数 H，還元関数 R，レインボーテーブルの 1 チェーンが次のように生成できると想定する．

$\texttt{XB4S} \xrightarrow{H} \texttt{db4360} \xrightarrow{R} \texttt{fIEj} \xrightarrow{H} \texttt{76893b} \xrightarrow{R} \texttt{7Tci} \xrightarrow{H} \texttt{fe5809} \xrightarrow{R} \texttt{nLkF} \xrightarrow{H} \texttt{be7fa8} \xrightarrow{R} \texttt{ocX7}$

このレインボーテーブルのチェーンを使用したパスワードクラッキング手順は，次の通り行う．

(i) 与えられたハッシュ値 "fe5809" から還元関数 R を用いて，新しいパスワード候補 "nLkF" を計算する．

(ii) そのパスワード候補 "nLkF" とチェーンの EP である "ocX7" とを比較する．両者の値は一致しないため，探索を続ける．

(iii) "nLkF" をハッシュ関数 H の入力メッセージとして，ハッシュ値 "be7fa8" を計算した後，還元関数 R を用いて，新しい パスワード候補 "ocX7" を計算する．

(iv) パスワード候補 "ocX7" がチェーンの EP と一致するため，このチェーンでパスワードをクラッキング可と考えられる．

(v) チェーンの SP である "XB4S" を用いて，与えられたハッシュ値 "fe5809" と比較しながらチェーンを復元する．この例では，パスワード "7Tci" を発見できる．

全てのチェーンを検索してもパスワードが見つからない場合は，与えられた

ハッシュ値に対応するパスワードがそのレインボーテーブルに存在しないことを意味する．

2.7　先行研究

Cayrel ら [37] の先行研究では，GPGPU を用いて Keccak 関数のソフトウェア実装を行った．同時に複数の入力ファイルに対した処理がバッチモードであり，1 回に付き 1 つの大きいファイルに対するハッシュ化処理が Tree モードであることを示した．Keccak-f[1600] の GTX 295 を用いた実行結果を表 2.6 に示す．この結果は Tree モードでの木の高さ H の変化によるスループットへの影響を示している．Cayrel らはバッチモードを実装せず，Tree モードのみの結果を発表した．

表 2.6　[37] の Tree モードのスループット．

File size[bytes]	H=0[GB/s]	H=1[GB/s]	H=2[GB/s]	H=3[GB/s]	H=4[GB/s]
1,050,112	0.0025	0.0101	0.0525	0.0750	0.0553
10,500,096	0.0026	0.0106	0.0729	0.1522	0.1667
25,200,000	0.0026	0.0106	0.0759	0.1669	0.1953
50,400,000	0.0026	0.0106	0.0769	0.1732	0.2533

また，Guillaume Sevestre らの研究 [38] では，Tree 構造による Keccak の GPU への実装を行った．GeForce GTS 250 に実装した結果，1,183MB/s のスループットとなったことが示されている．Lowden らの研究 [39] では，Tree 構造による Keccak の GPU への実装を行い，最大スループットは 3GB/s であった．

上記の 3 つの先行研究は，全てのハッシュ長の Keccak，かつサイズの大きいファイルに対しての，ハッシュ処理の高速化であった．パスワードクラッキングツール Hashcat[40] や仮想通貨のマイニングツール CCMiner Alexis[41] では，その目的から，数多くの入力メッセージが対象となっている．また，Hashcat と CCMiner Alexis を GeForce GTX 1080 の環境で実行した結果，それぞれの最大スループッ

トは770MH/と860MH/sであった．

崎山ら[42]はSHA-3に対するハードウェア実装について，調査報告書にまとめた．Baldwin[43] は，KeccakをVirtex-5に実装し6.3Gbpsのスループット性能を得た．Matsuoら[44]は，Keccakの提案者らが提供しているサンプル・コードを用いて，Virtex-5上のハードウェア性能の評価を行い，ハードウェアの性能評価の結果は 1.0 Gbpsであった．Guoら[45][46]はVirtex-5を用いて，UMC 180nmでKeccakのハードウェア実装を行い，合成結果として42.5 Kgates の回路規模で 10.7 Gbps のスループット性能を得ている．

他にも Keccak の高速ハードウェアが実装が多く報告されている [47] [48] [49] [50] [51]．デバイスの微細化などによる性能向上があり，FPGA 実装では 10～20 Gbps程度，ASIC 実装では20 Gbpsを超える実装結果が得られている．

Gravesらの研究[52]では，CUDAを用いてNTLMハッシュに対応したレインボーテーブルの生成を行った．実験した結果，長さ100,000の100チェーンを持つレインボーテーブルをGPUを用いて18分50秒で生成できた．Gómezらの研究[53]では，MPI (Message Passing Interface)及びCUDAのそれぞれでハッシュ関数MD5，SHA-1，NTLMに対する総当たり攻撃とレインボーテーブルの生成を行った．レインボーテーブルの生成ではMPIが有利であることを発表した．また，兼松ら[54]はCUDAを用いてcrypt(3)のDESに対応したレインボーテーブルの生成を行った．実験した結果，レインボーテーブルの生成時間はCPUのみによる逐次処理と比較して最大で約9.7倍速くなった．Keccakハッシュ関数に対応するレインボーテーブルの生成は，現時点ではまだ見つかっていない．

第3章

GPUとCUDAプログラミング

本章では，GPUとGPGPUの概要を述べた後，CUDA環境とCUDAプログラミングの詳細に関して述べる．

3.1 GPUとGPGPUの概要

コンピュータで演算機能を担うのはCPUである．しかし，近年ではグラフィック処理専用に開発されたGraphics Processing Unit（GPU）の利用が進んでいる．CPUとは違い，GPUには数千ものコアが搭載され，高い演算機能を持っている．その特徴を活用して，数値演算にGPUを使ったGPGPU（General Purpose Computation on Graphics Processing Unit）を用いた研究が盛んになっている．

GPGPUは，当初，OpenGLやDirect XなどのグラフィックスAPI（Application Programing Interface）とシェーダ言語を用いてプログラミングされていた．そのため，GPUの内部構造を熟知している必要があり，プログラミングは容易ではなかった．だが，2006年11月にNVIDIA社がGPUコンピューティング環境CUDA（Compute Unified Device Architecture）を無償でリリースしたことにより，GPGPUの状況を大きく変えた[55][56]．

その後，グラフィックカードの機能を目的とせずに，数値計算を高速化するためのGPGPU専用のアクセラレータボードTeslaが開発され，多くのスーパーコンピュータに採用され，現在では高速数値計算の一翼を担っている．

3.2　CUDA プログラミング

CUDA[11] は，NVIDIA が提供する GPU 向けの C 言語の統合開発環境であり，コンパイラ (nvcc) やライブラリなどから構成されている．CUDA のプログラムは，図 3.1 に示すように，CPU 側 (ホスト) と GPU 側 (デバイス) に分けることができる．GPU で実行されるカーネル関数はホスト側で起動する．

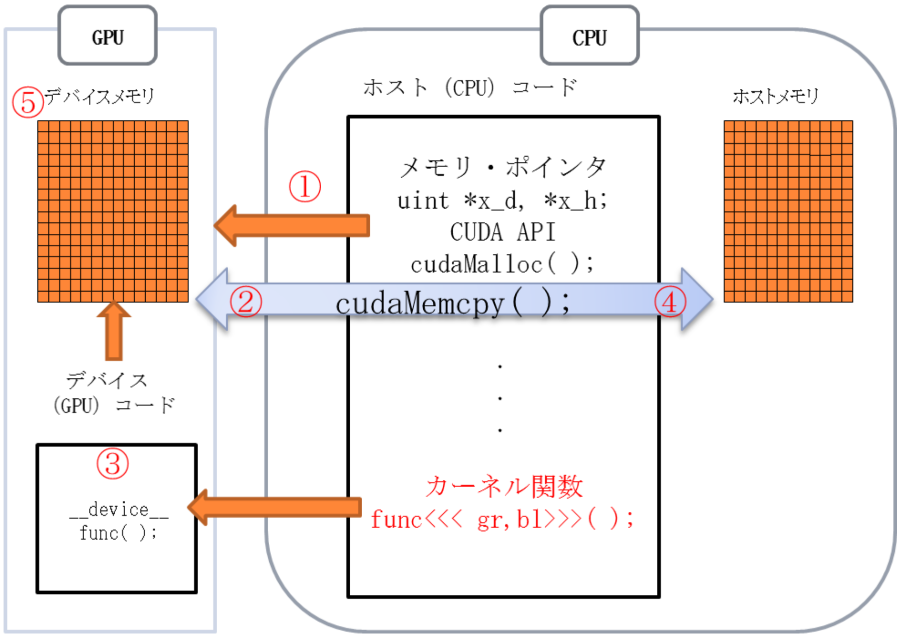


図 **3.1**　CUDA のプログラム構成のイメージ．

CUDA プログラムの処理の流れは次の 5 つのステップを用いて行う．

1. デバイス側のメモリを宣言し，確保する．
2. ホスト側からデバイス側にデータを転送する．
3. ホスト側からカーネル関数を呼び出し，デバイス側でカーネル関数を実行する．
4. デバイス側の実行結果をホスト側に転送する．
5. デバイス側のメモリを解放し，プログラムを終了する．

ステップ 2 とステップ 4 では，cudaMemcpy を用いて，データの転送を行う．

ステップ3では，<<<gr,bl>>>を用いて，グリッド内のスレッド数及び1ブロック当たりのスレッド数を指定することができる．

また，複数のGPU（デバイス）を持つ環境においては，それぞれのGPUの番号が存在し，使用する前にcudaSetDevice(“番号”)を用いて，使用するGPUを指定することができる．

3.2.1 GPUの構造とCUDAのプログラミング階層

CUDAプログラミングの階層構造は，図3.2に示すように，スレッド・ブロック及びグリッドから構成される．スレッドはプログラムを実行する最小単位であり，複数のスレッドをまとめたものがブロックとなる．さらに，ブロックをまとめたものがグリッドである．

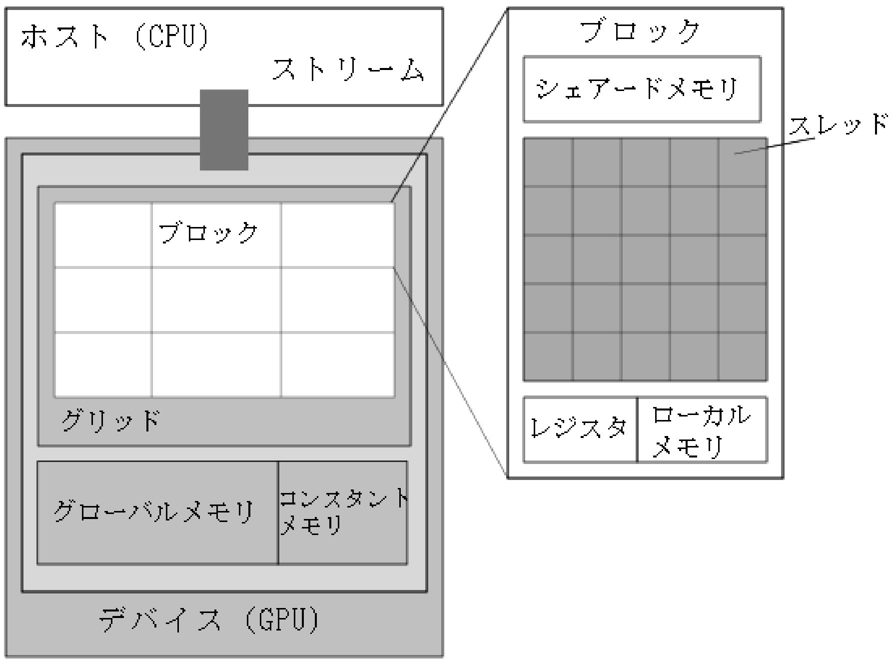


図 3.2　CUDAのプログラム階層．

スレッドは，ホスト側から起動される．多くの処理をスレッドとして並列に動作させることがCUDAプログラミングで重要となる．しかし，処理はすべて非同期であるため，プログラム上で__syncthreads( )関数を呼び出すことによ

り，プログラムの同期をとる必要がある．

例えば，GeForce GTX 1080 は，Pascal アーキテクチャを採用し，GP104 コアを用いて，Graphics Processing Clusters（GPC），ストリーミングマルチプロセッサ（SM），及びメモリコントローラーなど，様々な要素から構成される．GeForce GTX 1080 のブロック図を図 3.3 に示す [57]．GeForce GTX 1080 は，4

図 3.3　GeForce GTX 1080 のブロック図（[57] より引用）．

つの GPC，20 の PascalSM，及び 8 つのメモリコントローラーで構成されている．各 GPC に専用のラスターエンジンと 5 つの SM が付属している．各 SM には，128 個の CUDA コア，256 KB のレジスタ，96 KB の共有メモリ，48 KB の L1 キャッ

シュ，及び 8 つのテクスチャユニットが含まれる．SM は，マルチプロセッサーであり，SM 内の CUDA コア及びその他の実行ユニットへのワープ（32 スレッドのグループ）をスケジュールする，GPU 内で最も重要な部分である．GeForce GTX 1080 には 20 個の SM が搭載され，合計 2,560 個の CUDA コアと 160 個のテクスチャユニットが付属している．

3.3　メモリ階層

3.3.1　メモリの種類

各 SM 内には，それぞれシェアードメモリ (共有メモリ) とレジスタが搭載されている．これらのメモリは容量が小さくアクセスが高速という特徴がある．また，全ての SM からアクセスできるグローバルメモリが存在する．このメモリは，シェアードメモリやレジスタなどに比べるとアクセス速度は遅いが，容量が大きい．

CUDA のメモリ階層のイメージを図 3.4 に示す．

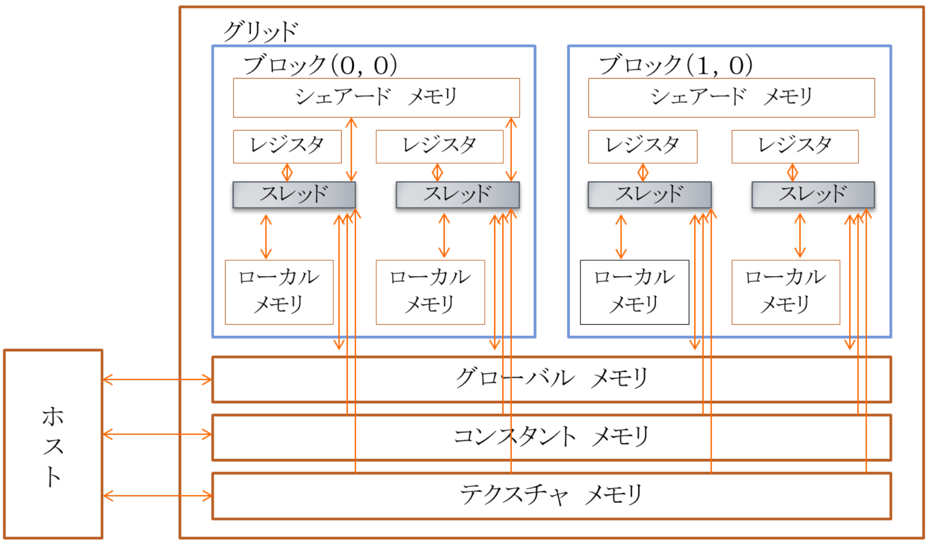


図 **3.4**　CUDA のメモリ階層．

物理的には，CUDA のメモリは GPU チップ内にあるオンチップメモリと GPU

チップ外にあるオフチップメモリに分けることができる．それぞれのメモリの種類を表 3.1 及び表 3.2 にそれぞれ示す．

表 3.1　オンチップメモリ [55].

	レジスタ	シェアードメモリ	テクスチャキャッシュ	コンスタントキャッシュ
容量	小	小	小	小
速度	高	高	高	高
ホストとのアクセス	不可	不可	不可	不可
デバイスとのアクセス	読み書き可（スレッドのみ）	同一ブロック内のスレッドから読み書き可	同一ブロック内のスレッドから読み書き可	同一ブロック内のスレッドから読み書き可

表 3.2　オフチップメモリ [55].

	ローカルメモリ	グローバルメモリ	テクスチャメモリ	サーフェスメモリ	コンスタントメモリ
容量	小	大	大	大	小
速度	低	低	高	高	高
ホストとのアクセス	不可	読み書き可	読み書き可	読み書き可	読み書き可
デバイスとのアクセス	読み書き可（スレッドのみ）	全てのスレッドから読み書き可	読み可	読み可	読み可

オンチップメモリは小容量である一方，アクセス速度は非常に速い．それに対し，オフチップメモリは低速アクセスだが容量は大きい．また，ローカルメモリ以外のオフチップメモリは CPU から直接アクセス可能であるが，オンチップメモリは GPU のみアクセス可能である．

3.3.2　高速化に関わる各種メモリ

(1)　グローバルメモリ

グローバルメモリはホストとデバイス両方から読み書きできるため，CUDA プログラミングでは，必ず利用される．GPU チップ外にあるオフチップメモリの一種類であるため，グローバルメモリへのアクセス速度は遅いという特徴がある．

CUDA プログラムでは，ある程度のサイズにまとめたデータでグローバルメモリとの通信を行うため，少ないデータのアクセスは効率が悪くなる．逆に，データのサイズを適切に合わせると効率の良いアクセスになる．

(2)　コンスタントメモリ

コンスタントメモリの容量は小さいが，キャッシュが効くため，グローバルメモリよりも高速にアクセスできる．ただし，書き込みはホスト側からのみ可能で，デバイス側からは読み込みしかできない. デバイスの全てのブロックからコンスタントメモリにアクセスできる．

(3)　シェアードメモリ

カーネル関数内で，`__shared__`を用いて変数を定義することでシェアードメモリを使用することができる．GPU チップ内にあるため，グローバルメモリと比較して 10 倍以上高速にアクセスできる共有メモリであり，計算速度を向上させるためには重要である．

シェアードメモリへのアクセスは，その対応したブロックの全てのスレッドから可能である．また，シェアードメモリは，バンクと呼ばれるユニットの集まりになっている．メモリ空間は 16 バンク（1 バンク 32 ビット）に分割されており，16 スレッドが各バンクに競合無しアクセスすると，並列アクセスが発生する．一方，16 スレッドがシェアードメモリの同じバンクにアクセスした場合，そのアクセスは 16 回のアクセスにシリアライズされ，並列にアクセスできなくなる [58]．Pascal アーキテクチャでは，SM ごとに 96KB のシェアードメモリを持っている．

(4)　レジスタ

レジスタは，各スレッド内からのみアクセス可能であり，容量は小さいが高速に読み書きできるメモリである．スレッド内の一時的な変数などは，このレジスタを使用して実行される．Pascal アーキテクチャでは，スレッドごとに最大 255 レジスタを利用できる．

3.3.3　各種メモリの使用方法

それぞれのメモリ容量の上限やアクセス速度も大きく異なっているため，高速化実装において，各種メモリの使用法を変更すれば，プログラムの効率も大きく変化するため，適切なメモリの使用が必要である．

まずは，ホスト側（CPU) から GPU のレジスタ，シェアードメモリやローカルメモリへのアクセスが不可能であるため，ホストとデバイス間のデータ転送には，グローバルメモリを使用しなければならない．それぞれのスレッドでの処理における変数は，高速，かつスレッドのみ読み書き可能であるレジスタを使用する．このレジスタの容量が小さいため，できるだけ少ないレジスタを使用するようにプログラムを書く必要がある．

対象となるアルゴリズムに固有の定数がある場合，全てのスレッドで使用する定数は同じものとなる．各スレッドで行う処理は SIMD のため,，コンスタントメモリやシェアードメモリに定数を保存することにより，プログラムの効率を向上できる. しかし，全てのブロックからコンスタントメモリにアクセスできる一方，シェアードメモリは同一ブロック内のスレッドからのみ読み書き可能である．そのため，数多くのスレッドが同じ共有メモリに競合アクセスすると，処理速度が落ち，プログラムの効率も低下する．特に同時に同じバンクへのアクセスを要求すると，競合アクセスが多く発生し，アクセス速度が大きく落ちてしまう．

3.3.4　GPU キャッシュ

GPU は SM ごとに L1 キャッシュメモリを持っている．例えば，GeForce GTX 1080 の場合，SM ごとの L1 キャッシュメモリの容量は 48KB である．また，GeForce

GTX 1080ではGP104チップを使用し，2,048KBのL2キャッシュを持っている．GP104では，L1キャッシュとテクスチャキャッシュの機能が組み合わせられ，メモリアクセスのバッファとして機能するユニファイドL1/テクスチャキャッシュに統合する．グローバルロードがL1にキャッシュするデータアクセスユニットは32Bに制限される．また，スレッドローカルメモリはL1キャッシュにキャッシュすることになるため，最高のパフォーマンスを確保するには，メモリ占有率を確認する必要がある．[59]

数多くのスレッドからグローバルメモリに乗せている同一アドレスのデータへアクセスするとき，キャッシュを通してアクセスすることになる．GPUでプログラムの処理速度に関わるキャッシュメモリはL1キャッシュである．シェアードメモリとL1キャッシュメモリは同じハードウェアを使用し，シェアードメモリとキャッシュ容量の割り合を表3.3に示すオプションで指定できる[60]．これは，下に示すようにCUDAプログラムで指定できる．

cudaFuncSetCacheConfig(Keccak,[オプション])

これは，カーネル関数の直前で呼び出さなければならない．

表 3.3 シェアードメモリとL1キャッシュの割合．

オプション	割合
cudaFuncCachePreferNone	割合をコンパイラに任せる（デフォルト）
cudaFuncCachePreferShared	シェアードメモリの方が多くのメモリを使用
cudaFuncCachePreferL1	L1キャッシュの方が多くのメモリを使用
cudaFuncCachePreferEqual	シェアードメモリとL1キャッシュの容量が同じ

本研究では，シェアードメモリとキャッシュ容量の割り合をコンパイラに任せる設定とした．

3.4　実装環境

表 3.4 に実装環境を示す．使用した GPU は GeForce GTX 1080 であり，CPU より高い演算処理能力を持っている．

表 3.4　実装環境.

OS	Ubuntu 16.04.2 LTS
CPU	Intel Xeon E5-1620 v4 (3.60GHz)
GPU	GeForce GTX 1080
CUDA Ver.	10.0
Compiler	gcc ver 7.3.0; nvcc ver 10.0 (CUDA)
Compiler Option	-O3

GPU を用いた実装において，処理速度等の実装結果は使用するデバイスに依存するが実装方法，原理は同じである．本研究で使用する GPU は GeForce GTX 1080 であり，2020 年現在，最新ではないものの性能が高く，多くのシステムが採用してる．性能の高いデバイスで実装した場合，より良い結果が得られる見込みである．スペックの高い最近の NVIDIA 社の GPU デバイスの一部を表 3.5 に示す．参考価格は 2020 年 1 月の価格.com や Amazon の平均価格を参考にしている．また，実際の性能を比較するためにベンチマークツール 3DMark[61] の結果を使用する．

GPU のチップメーカーは主に NVIDIA 社と AMD 社が存在し，スペックに関する規定値が設けられているため，どこのメーカー品を使用しても性能に差が出ることはありません．CUDA は NVIDIA 社が開発・提供してるため，全社の製品の方が CUDA プログラミングに適してる．

表 **3.5**　最近の NVIDIA の GPU デバイス.

GPU 名称	3DMark	TDP	メモリ	CUDA コア数	参考価格
Titan RTX	37,472	280W	24GB GDDR6	4,608 基	330,000
GeForce RTX 2080 Ti	33,500	250W	11GB GDDR6	4,352 基	130,000
GeForce RTX 2080	27,900	215W	8GB GDDR6	2,944 基	80,000
GeForce GTX 1080 Ti	27,800	250W	11GB GDDR5X	3,584 基	100,000
GeForce GTX 1080	22,000	180W	8GB GDDR5X	2,560 基	65,000

第4章

ハッシュ関数Keccak-512のGPUへの高速化

本研究で想定しているハッシュ関数の入力は，パスワードに対応する平文のメッセージである．8 文字のパスワードの場合，入力メッセージ長は 64 ビットである．本研究では，一つのパスワードにつき，GPU 上の 1 スレッドを用いてハッシュ化の処理を行う．実装においては，複数の入力メッセージ（パスワードを想定した平文）を一つの 2 次元配列にまとめて GPU 上のメモリにコピーする．GPU では，図 4.1 に示すように，各スレッドがそれぞれに割り当てられた入力メッセージのハッシュ化処理を行った．

なお，総当たり攻撃に応用する場合，本研究では未実装であるものの，GPU でハッシュ化処理した値と与えられたハッシュ値とを照合するプロセスが必要となる．

GPU プログラムの効率を向上させるために，以下の 4.1〜4.4 に示す 4 つの手法による改良を加えて実装し，それぞれの実装結果を示していく．なお，ここでは Keccak の提案者が示したプログラムコード [62] をリファレンスコードとし，このリファレンスコードを基本として高速化を行っていく．

4.1 ルックアップテーブルの再構成

Keccak の計算時に，ステップ ρ と π では巡回シフトの定数を 2 次元配列で保存する必要があり，実装では事前にテーブル#1 とテーブル#2 でこれらの定数を格納する．第 2 章で説明した Keccak のアルゴリズムにおける式 (2.4.4) で，2 次元配列の要素 $B[y, 2x+3y]$ にアクセスする必要がある. ここで,`Index2X3YM5[y][x]` として事前に計算した $(2x+3y)\ mod\ 5$ の結果をテーブル#1 に格納させる．ま

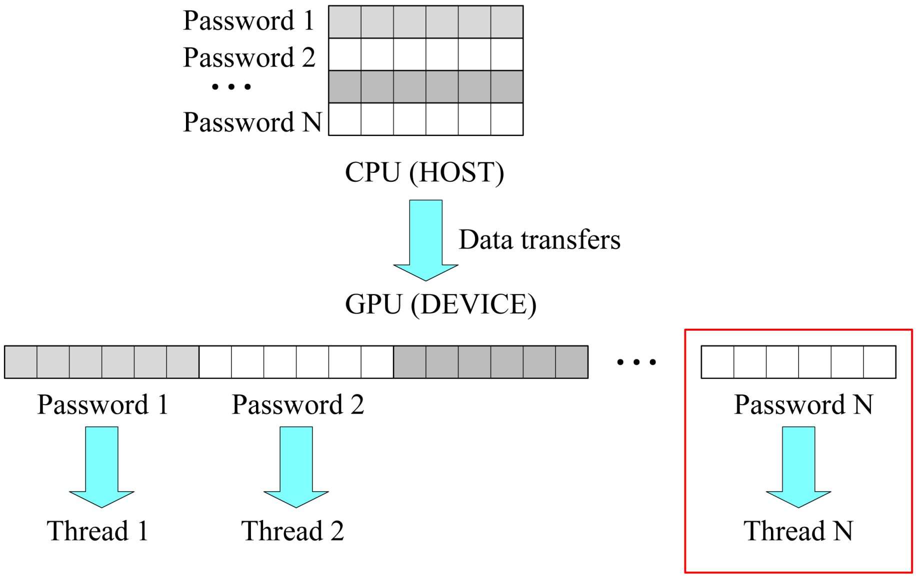


図 4.1　ハッシュ化処理の並列.

た, テーブル#2 にはラウンドオフセットの値 $r[y][x]$ を格納させる．この場合，レファレンスコードでは 2 次元配列を利用する 2 つのテーブルを準備することになる．これらの事前計算テーブルは 5 ∗ 5 のサイズで 25 要素の 2 次元配列を持っている．ここで 24 要素の 1 次元配列に再構成することによって，レジスタの使用量を減らし，プログラムの速度を向上させる．

レファレンスコードで使用するプログラムは以下の Listing 4.1 に示す.

Listing 4.1 レファレンスコードのプログラム（抜粋）.

```
for(y=0; y<5; y++){
  for(x=0; x<5; x++){
    B[y, Index2X3YM5[y][x]] = ROTL(A[x,y], r[y][x]);
  }
}
```

ここで ROTL(A[i],r) は右巡回シフト計算であり，A(i) の位置を A(i + r) に

変更する処理になる．テーブル#1 とテーブル #2 を `index[i]` と `r[i]` の 1 次元配列に再構成した．再構成したプログラムは，次の Listing 4.1 に示す．

Listing 4.2 再構成したプログラム．

```
temp = state[1];
for (i = 0; i < 24; i++) {
  j = index[i];
  C[0] = A[j];
  A[j] = ROTL(temp, r[i]);
  temp = C[0];
}
```

この提案手法の実装結果を表 4.1 に示す．ここでは，レファレンスコードを RFC として，提案手法とのスループットを比較している．また，入力メッセージの数を 262,144 に固定し，使用するスレッド総数が入力メッセージ数に一致するようにスレッド・ブロックを変化させている．スループットの単位は GigaBytes per second(GB/s) で表示される．提案手法を用いた実装は，通常の実装より 1.15〜1.21 倍高速であることが確認できた．

表 4.1 スループットの比較．

ブロックの数	スレッド数	RFC [GB/s]	提案手法 [GB/s]	倍率
512	512	24.109	28.209	1.17
1,024	256	32.837	38.197	1.16
2,048	128	33.104	38.545	1.16
4,096	64	32.601	37.884	1.16
8,192	32	32.923	38.041	1.15
16,384	16	31.084	37.650	1.21

今回，高速化できた理由は配列への参照時の処理だと考えられる．ここで1次元配列 A[M*N] と2次元配列 B[M][N] について考察する．どちらの配列でも M*N 個の要素を持っているが， [i][j] 位置の要素にアクセスするときの手順が異なる．2元配列 B[M][N] において B[i][j] にアクセスする場合，毎回 i*N+j の計算を終えてからアクセスするが，本実装では予め1次元配列に再構成することによって，1回のアクセスだけで済むため，全体のスループットを向上することができた．

4.2 定数テーブルのメモリ配置

第3章で説明したように，グローバルメモリは全てのスレッドからアクセスできるが，そのアクセス速度が遅いため，定数については，コンスタントメモリやシェアードメモリを積極的に使う必要がある．コンスタントメモリを使用することにより，各スレッドから高速に定数を呼び出すことが可能になる．しかし，巨大な数のスレッドから同じコンスタントメモリの異なるアドレスにアクセスの要求命令を出すと，アクセス競合が発生し，全体の処理速度が低下する．そのため，本研究では全スレッドで共有するコンスタントメモリの他に，各ブロックごとに共有するシェアードメモリも使用することとした．

本研究の GPU 実装では，定数を3つの事前計算テーブルに格納する．前項で説明したステップ ρ とステップ π で使用するインデックス数及びラウンドオフセットをそれぞれテーブル#1とテーブル#2に保存することとした．その他に，ステップ ι で使用するラウンド定数 $RC[i]$ をテーブル#3に格納することとした．テーブル#1〜#3を GPU デバイスにあるグローバルメモリ，コンスタントメモリ，及びシェアードメモリに実装する組み合わせ総数は27通り有り，そのうち有効な9通りを実装し，スループットを測定した．表4.2に512ブロックを用意し，1ブロックあたり512スレッドを使用した場合の測定結果を示す．ここで，G，C，そしてSはそれぞれグローバルメモリ，コンスタントメモリ，シェアードメモリ

を意味している．

表 **4.2** テーブルのメモリ配置によるスループットへの影響．

メモリの種類			スループット [GB/s]
テーブル#1	テーブル#2	テーブル#3	
G	G	G	3.514
C	C	C	41.203
C	C	S	43.056
C	S	C	35.201
C	S	S	35.157
S	C	C	3.511
S	C	S	3.474
S	S	C	3.463
S	S	S	3.491

表 4.2 に示したように，テーブル#1 をコンスタントメモリに配置した場合（2 行目〜5 行目）は，グローバルメモリ（1 行目）やシェアードメモリを使用する場合（6 行目〜9 行目）に比べていずれも約 10 倍高速であった．2 行目及び 3 行目のように，テーブル#1 と#2 をコンスタンメモリに載せるとスループットが大きくなった．この理由は，各種メモリへのアクセス状況とそれに対するアクセス速度の関係にある．第 3 章で説明したように，コンスタントメモリとシェアードメモリは，どちらもグローバルメモリに比較して 10 倍以上高速にアクセスできる．しかし，テーブル#1 と#2 はランダムアクセスという特徴がある．ランダムアクセスで異なるアドレスへアクセスした場合，シェアードメモリへの速度低下が非常に大きいことを 6 行目〜9 行目のように確認できた．

ここで，テーブル#1, テーブル#2, 及びテーブル#3 は 24 要素を持っている．Keccak-512 で使用する Keccak-f[1600] 関数は 24 ラウンドの処理によって計算を

行う．各ラウンドでは，テーブル#1, テーブル#2 の全ての要素にランダムに 25 回アクセスするが，テーブル#3 は，ラウンドごとに同じ 1 要素のみ 1 回アクセスする．GPU プログラムでは，決まったアドレス先にアクセスする場合，多くのスレッドからでもコンスタントメモリとシェアードメモリへのアクセス速度が速くなる．しかし，ランダムなアクセスが頻繁に発生すると，シェアードメモリへのアクセスがグローバルメモリの速度程度まで低下した．実際の測定結果により，本研究ではランダムアクセスを高速化するために，テーブル#1 とテーブル#2 をコンスタントメモリに載せることにした．テーブル#3 へのアクセスは各ラウンドに決まった要素のみにアクセスするため，コンスタンメモリとシェアードメモリどちらに載せてもほぼ同様のスループットを得られる．今回の実装では，テーブル#3 をシェアードメモリに格納したときに，最大のスループットを得ることができた．

4.3 占有率とブロック・スレッドの最適な構成

GPU はブロック数・スレッド数の組み合わせを設定することによって，スレッド総数が同じでも処理速度が変化することが知られている．本実装では，これらの数を変化させ，スループットへの影響を検討した．また，その結果から処理速度やスループットを向上させるためのブロック数・スレッド数の条件を考察した．ここで，ブロック数は 1 グリッド内のブロックの数のことであり，グリッドのサイズを現す．例えば，グリッドが `grid[128,1,1]` のように指定した場合，そのグリッドには 128 ブロックが含まれることを意味する．また，スレッド数は 1 ブロック内のスレッドの数であり，ブロックのサイズを現す．

CUDA プログラムでは，ストリーミングマルチプロセッサ（SM）ごとに同時実行されるスレッドの数を占有率（occupancy rate）とし，プログラムの効率を評価することができる．その占有率は，ブロックごとのスレッド数，スレッドごとのレジスタ，及びブロックごとの共有メモリの使用量から計算される [63]．本

実装では，各スレッドが 1 つのメッセージをハッシュするため，スレッドごとのレジスタ数とブロックごとの共有メモリサイズは固定している．GeForce GTX 1080 及び CUDA 10.0 環境では，スレッドごとに最大 255 レジスタを使用でき，最大のブロックサイズが 1024 である．また，カーネル関数はスレッドごとに 76 個のレジスタ，ブロックごとに 192 バイトの共有メモリを使用している．スレッドごとにレジスタを配することにより，占有率は高くはならない．ブロックサイズに応じた占有率の変化を表 4.3 に示す．

表 4.3 ブロックサイズと占有率との関係.

スレッド数	8	16	32	64	128	256	512
占有率	37.5%	37.5%	37.5%	37.5%	37.5%	37.5%	25.0%

CUDA の占有率は，ブロックごとのスレッド数，スレッドごとのレジスタ量，及びブロックごとの共有メモリサイズに依存し， ブロック数（グリッドサイズ）には依存しないものの，同じ占有率であっても，ブロック数が変更されるとスループットも変化してしまう．最大のパフォーマンスを実現するために，占有率を 100%にする必要はない．表 4.3 の結果から，スレッド数が 8，16，32，64，128，または 256 に設定した場合，占有率はほぼ同様で 37.5%に達し，ブロックごとに 512 スレッドを使用した場合よりも高かった．また，占有率が同じ 37.5%で最大スループットが得られることを示しているが，それぞれのブロックあたりのスレッド数は異なっている．

前章及び前節 4.2 でも説明した通り，各ブロック毎にシェアードメモリが存在し，ブロック内の全てのスレッドで共有してる．また，各ブロックは同じコンスタントメモリを共有して利用する．そのため，GPU で実行されているカーネル関数の処理時間は，ブロック・スレッドの数に依存する．ブロック・スレッドの最適な構成を見つけるために，入力メッセージ総数，ブロック数，ブロックごと

のスレッドの数を変更し，GPU 上の Keccak の最大スループットを測定した結果を表 4.4 に示す．

表 4.4 ブロック・スレッド数による最大スループット．

メッセージの総	ブロックの数	スレッド数	最大スループット [GB/s]
256	8	32	2.346
512	16	32	4.591
1,024	16	64	8.936
2,048	16	128	16.783
4,096	256	16	24.097
8,192	512	16	31.558
16,384	1,024	16	40.221
32,768	1,024	32	44.526
65,536	1,024	64	48.562
131,072	1,024	128	50.428
262,144	2,048	128	51.477
524,288	4,096	128	59.931
1,048,576	8,192	128	60.357
2,097,152	16,384	128	60.586
4,194,304	16,384	256	60.555
8,388,608	16,384	512	50.078

本実装では，GeForce GTX 1080 に実装し，1 マルチプロセッサ（MP）あたり 128CUDA コアを備えた 20 個の MP が搭載されるため，ブロックあたり 128 スレッドを使用すると高いスループットが得られたと考えられる．実装した結果，最大スループットは 60.586 GB/s であった．

4.4 CUDA ストリームとオーバーラッピング

CUDA ストリームとは，GPU 内の処理のスケジュール管理の単位，または処理キューのようなものである [64]．データ転送やカーネル実行命令を CPU から GPU に出すとき，GPU の処理完了を待つか，次の動作に移るか決めるスケジューリングになる．通常のプログラムでは，ストリームを設定しない Null Stream では，順次処理と同じように，GPU とのデータ転送やカーネル関数の実行が終わらない限り，CPU は待ち状態になるため，次の動作が実行されない．複数のストリームを使用して複数のカーネルを起動することにより，グリッドレベルの同時実行を実現し，データ転送を隠すことが可能となる．ストリームの宣言，作成，破棄，及び起動させる命令を次の Listing 4.3 に示す．

Listing 4.3 ストリームを使用するプログラム [64]．

```
cudaStream_t stream1;
cudaError_t result;
result = cudaStreamCreate(&stream1);
result = cudaStreamDestroy(stream1);
.....
kernel<<<grid, block, sharedMemsize, stream>>>(argument list);
```

実装で 1 ストリーム及び 3 ストリームを使用する場合のタイムラインを次の図 4.2 に示す．ここで,「make pwd」はホスト（CPU）で実行されるパスワード作成のプロセスである．ホストからデバイスへのデータ転送（H2D）は，生成されたパスワードを CPU から GPU にコピーするプロセスである．GPU でカーネルを実行すると，全てのパスワードに対しハッシュ化し，ハッシュ値が得られる．これらのハッシュ値をデバイスからホストへのデータ転送（D2H）によって CPU 側に返される．

この場合，同じストリームのデータ転送やカーネル同士は順次実行されるが，別のストリームでのデータ転送とカーネルはオーバーラップさせることで処理時

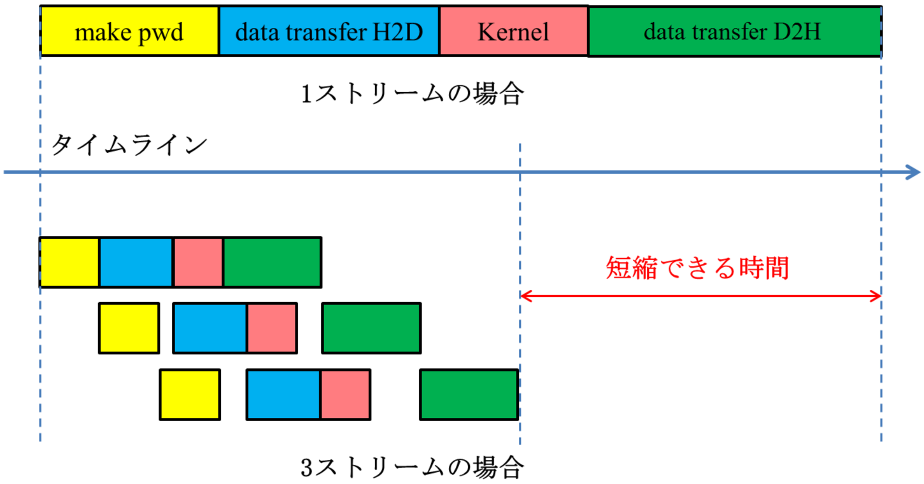


図 **4.2**　1ストリームと3ストリームのタイムライン比較．

間が短縮できることがある．パスワード候補の生成，データ転送，及びカーネル実行時間を配慮した結果，本研究では3ストリームを使用することにした．その実装・評価結果を表4.5に示す．ここで，ストリームを使用することにより，スループットを約1.07倍向上できた．

4.5　先行研究との比較

Sevestreらの研究で公開されているTree構造によるKeccak-512の高速化のソースコード[65]を本研究の実装環境で実行した結果，そのスループットは4.6 GB/sであったのに対し，本研究の最大スループットは64.582 GB/sまで達することができた．

また，Hashcatを実行した結果，同じ環境で769.6 MH/sの処理速度を確認でき，CCMiner Alexisの処理速度は860.0 MH/sであった．本研究の最大スループットは64.582 GB/sであり，同じ単位に変換すると999.6 MH/sになる．

表 4.5　CUDA ストリームによるスループットへの効果.

メッセージの数	1 ストリーム [GB/s]	3 ストリーム [GB/s]	倍率
8,192	31.558	33.830	1.072
16,384	40.221	43.037	1.070
32,768	44.526	47.687	1.071
65,536	48.562	51.864	1.068
131,072	50.428	53.908	1.069
262,144	51.477	55.131	1.071
524,288	59.931	64.066	1.069
1,048,576	60.357	64.582	1.070

4.6　パスワードクラッキングへの対策

パスワードクラッキングの事例は年々増加しているため，システム管理者側とシステム利用者側，どちらも対策を準備する必要がある．2019 年 8 月に IPA より公開された「情報セキュリティ10 大脅威 2019」[66][67] においても直接または間接的にパスワードクラッキングが関係している脅威が多数ランクインしている．

システム利用者の観点から，最も簡単な対策としては，パスワードを設定するときに，意味を持たず，できるだけ長いパスワードにすることである．パスワードクラッキングは，総当たりや推測によってパスワードを割り出そうとするものが多く，短いパスワードや意味のある文字列では攻撃を受けるリスクが高まる．もう一つの対策としては，複数のサービスで同じパスワードを使いまわさないことである．複数のサービスで同じパスワードを使いまわした場合，そのうちの一つのサービスからパスワードが漏洩した際に，パスワードリスト攻撃によって他のサービスにまで不正利用の被害が及んでしまう．また，各システム，サービスを利用する際，常にセキュリティ面に意識を向ける必要がある．[68]

システム管理者側の対策としては，ログインやパスワード設定時の対策及びパスワード管理時の対策に分けることができる．例えば，アカウントごとのログイン試行回数に上限に設けることで，特定のアカウントに対する総当たり攻撃を防ぐことが可能である．また，短すぎる，文字種が少ない，設定されやすい文字列などというパスワードが設定された場合，警告を表示して再設定を促す機能を実装することで，危険なパスワードの利用を防ぐことが可能である．[69]

パスワード管理において，攻撃者からサーバなどに保存されたID・パスワードが窃取された際，平文のまま保存されていた場合は直ちに不正ログインの被害へと繋がる．その対策としては，ハッシュ化として平文の特定を困難にするという手法が一般にとられている．また，レインボーテーブルへの対策として，ハッシュ化前の平文に，ユーザごとに異なるソルトと呼ばれる文字列を付加する方法や，ハッシュ化に求めた値を更にハッシュ化する処理を複数回繰り返す実装がある．例えば，MySQLのパスワード管理アルゴリズムでは,「key=SHA1(SHA1(password, salt))」のように2回ハッシュ化処理を行っており，また，ビットコインのアドレスを生成する処理においては，SHA-256を3回ハッシュ化する処理を行っている[70]．パスワードを管理するサービスであるLastPass[71]，ビットコインウォレットの一種であるBlockchain[72]やその他のPBKDF2（Password-Based Key Derivation Function2）[73][74]を利用するシステムでは，そのハッシュ化の反復回数が非常に大きく，1,000回から10,000回程度を指定することもできる．

このような矛式に対応するため，GPU上での2，3回ハッシュの実装を行い，速度を測定した結果を表4.6及び図4.3に示す．

表 4.6　1，2，3 回ハッシュの最大スループットの比較．

ブロックの数	スレッド数	1 回 [MH/s]	2 回 [MH/s]	3 回 [MH/s]
512	512	584.590	336.319	243.181
1,024	256	725.023	365.359	260.196
2,048	128	719.553	366.273	252.329
4,096	64	718.710	365.285	250.417
8,192	32	717.652	364.148	247.447
16,384	16	425.648	252.591	183.941

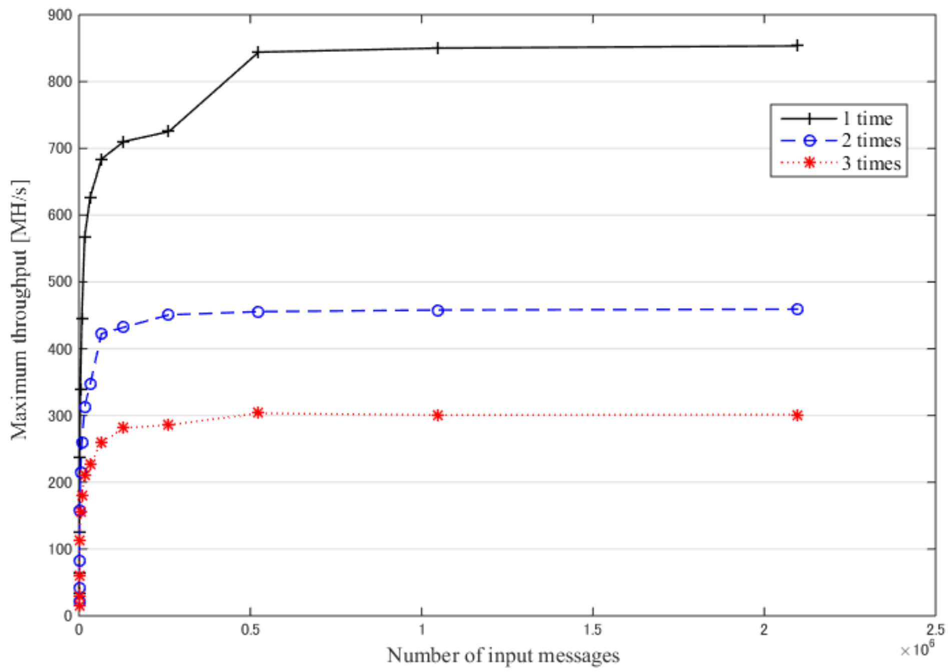


図 4.3　1，2，3 回ハッシュのスループットの最大スループットの比較．

2 回ハッシュの処理速度は 1 回ハッシュより約 11.8%遅くなり，3 回ハッシュの処理速度は 1 回ハッシュより約 26.6%遅くなった．この結果は，2，3 回程度のハッ

シュ化を行うことで処理速度を落とさせて，全数探索である総当たり攻撃の難易度を上げるには効果が少ないことを示している．

本研究では一秒あたり 999.6 メガハッシュを処理できることを示してる．英文字の小文字、大文字及び数字を使った 8 文字パスワードであれば 62^8（約 218 兆）通りの組合せが存在する．つまり本研究の実装と同じ環境であれば，約 60.7 時間（2 日半程度）で総当たり攻撃が可能となる．9 文字のパスワードの場合，所要時間は約 3,762 時間（156.7 日）である．また，レインボーテーブルを使用することにより，さらにパスワードクラッキングの時間を短縮することができる．従って，単なる Keccak-512 を用いてパスワードを 1 回や数回のハッシュ化処理を行い，そのハッシュ値でパスワードを管理することは非常に危険である．Keccak-512 も含め，ハッシュ関数を利用したパスワード管理システムでは，パスワードの後にソルトを付加することや 1,000 回以上ハッシュ化を反復する PBKDF2 のような関数を採用することで，パスワードクラッキングからパスワードを保護することが必要である．

第 5 章

GPU を用いたレインボーテーブル生成の高速化

5.1 GPU によるレインボーテーブル生成の高速化

5.1.1 Keccak-512 に対応したレインボーテーブル生成の提案

パスワードの長さが長いほど，チェーンの数またはチェーンの長さが長くなり，レインボーテーブルの計算量が膨大になる．ここでは，各チェーンを生成する処理は同じであり，入力として異なる SP を持つ．全てのチェーンは独立しており，同じ操作を実行するため，GPU では SIMD 処理で効率的にその計算を並列化できる．つまり，チェーン生成を並列化することにより，レインボーテーブル全体の生成時間を短縮することが可能である．本研究では，GPU の 1 つのスレッドを用いて 1 つのチェーンを生成する．その手順を次に示す．あわせて図 5.1 にも示す．

① CPU 側でランダムにチェーンの数と同数の SP（パスワード候補）を生成する．

② 生成された SP の集合を配列に格納し，GPU のメモリに転送する．

③ それぞれの SP を GPU の各スレッドに割り当て，1 スレッドで 1 チェーンを生成し，SP に対応する EP を計算する．GPU で実行するカーネル関数の出力は EP である．例えば，スレッド i に SPi が割り当てられると想定する．ハッシュ関数 H を用いて SPi に対するハッシュ値 H1 を計算した後，還元関数 R を用いて H1 からパスワード候補 P2 を計算する．その処理を繰り

返すことにより最終的に EPi のパスワード候補をスレッド i が計算結果として出力する．

④ 計算された EP の集合を CPU 側に転送し，別の配列に格納する．

⑤ 全ての SP-EP のペアをレインボーテーブルとして保存する．

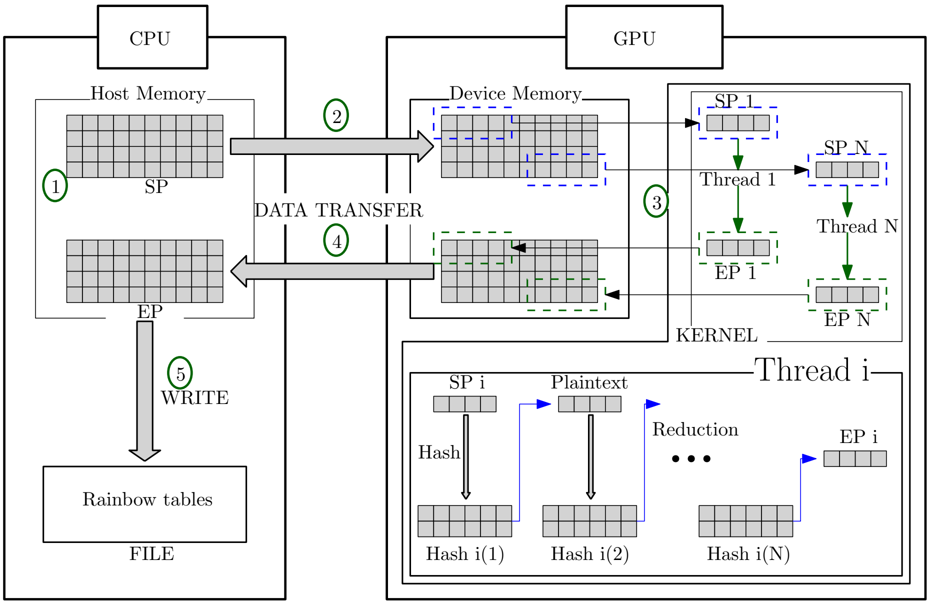


図 **5.1**　チェーン生成処理の流れ．

ハッシュ値の計算処理は何度も実行され，チェーン生成処理のほとんどの時間を占める．Keccak ハッシュ関数に対応するレインボーテーブルの生成は，現時点ではまだ見つかっていないため，本研究では，第 4 章に示した Keccak-512 の提案手法を活用し，ハッシュ値の計算処理を高速化し，チェーン生成時間の減少を図る．

5.1.2 還元関数の改良

(1) パスワード候補の衝突

還元関数とは，ハッシュ値から次のパスワード候補を生成する関数のことであり，特に生成法が決定されるものではなく，レインボーテーブルごとに異なる．例えば，Rainbow Crack の還元関数は次のように定義される．

$$R = H \ mod \ N \tag{5.1.1}$$

ここで，R は還元関数，H はハッシュ値，N はパスワード候補の総数である．

計算時間の問題により，レインボーテーブルの還元関数はあまり複雑な関数ではなく，単純な計算から構成される．しかし，単純な関数を使用すると，多くの場合，パスワード候補の衝突と呼ばれる異なるハッシュ値から同じパスワード候補が生成される問題が生じる．例えば 1 つのチェーンが次のように生成される．

$\texttt{XB4S} \xrightarrow{H} \texttt{db4360} \xrightarrow{R} \texttt{fIEj} \xrightarrow{H} \texttt{76893b} \xrightarrow{R} \texttt{7Tci} \xrightarrow{H} \texttt{fe5809} \xrightarrow{R} \texttt{nLkF} \xrightarrow{H} \texttt{be7fa8} \xrightarrow{R} \texttt{7Tci} \xrightarrow{H} \texttt{fe5809} \xrightarrow{R} \texttt{nLkF} \xrightarrow{H} \texttt{be7fa8} \xrightarrow{R} \ldots$

この場合，同じパスワード `7Tci` が異なるハッシュ値 `76893b` と `be7fa8` から生成されたため，チェーンの一部が重複し，無駄な計算が発生してしまう．

一方，複雑な機能を使用すると衝突を減らすことができるが，計算時間が長くなる．例えば，`BASE62` を用いれば，異なるハッシュ値に対して必ず異なるテキストを得ることがでる．しかし，この処理は場合によっては時間がかかることがある．そのため，多くのレインボーテーブルでは，単純な計算のみから構成された還元関数を使用している．

(2) 還元関数の改良

本研究では，剰余算を用いた還元関数を使用することとした．対象ハッシュ関数が `Keccak-512` であるため，512 ビットのハッシュ値を計算する．パスワード候補で使用される文字は，数字，大文字の `A`～`Z`，及び小文字の `a`～`z` で，合計 62 種類となる． 512 ビットのハッシュ値は 64 個の 1 バイトデータに分割され，1 バイトのパスワード候補はこれらの 1 バイトのハッシュ値から次のように計算さ

れる．

$$c = charset[H\ mod\ 62] \tag{5.1.2}$$

ここで，`c` は還元関数によって計算されたパスワード候補の文字，`H` は 1 バイトのハッシュ値，`charset` は数字，大文字，小文字を格納した配列である．

パスワードの衝突を減らすために，本研究では，還元関数のパラメータにチェーン上のパスワードの位置情報を追加することにした．pos を位置情報として，改良した還元関数は次のように定義される．

$$c[i] = charset[H[(i + pos)\ mod\ 64]\ mod\ 62] \tag{5.1.3}$$

この位置情報を追加することで，チェーン内や別のチェーンでのパスワード候補の重複が発生しても，次のチェーンの部分で重複されないようなり，異なるチェーンの異なる位置におけるチェーンの重複を避けることができる．ここで，別のチェーンの同じ位置で衝突が発生しない限り，衝突があっても次は別のパスワード候補が生成され，繰り返し衝突が生じることはない．

5.2 実装・評価結果

5.2.1 還元関数の改良

衝突による重複の問題があるため，レインボーテーブル内に含まれるパスワード候補の数は全パスワード候補の総数よりも多く生成する必要がある．ここで，レインボーテーブル内に含まれるパスワード候補の数は，テーブル内のユニークなパスワードの数と定義する．レインボーテーブルの効果を評価するために最も簡単な方法としては，そのテーブルに含まれるパスワード候補の数と全パスワード候補の総数との割合を計算する方法がある．それはレインボーテーブルのパスワード網羅率と呼び，次の式のように計算できる．

$$\text{網羅率} = \frac{\text{テーブル内のパスワード候補の数}}{\text{全パスワード候補の総数}} * 100(\%) \tag{5.2.1}$$

パスワード網羅率が高いほど，レインボーテーブルによる攻撃の成功率が高くなる．

還元関数に位置情報を追加することの効果を確認するために，全てのチェーンで生成された全部のパスワード候補をファイルに保存し，それらのパスワードの網羅率を確認した．位置情報が追加された場合と位置情報を使わない場合の網羅率の比較を表 5.1 及び図 5.2 に示す．ここでは，チェーンの長さが 50 であり，4 文字のパスワードを生成するレインボーテーブルとする．

表 5.1 位置情報による網羅率への効果.

チェーンの数	網羅率 [%] (位置情報あり)	網羅率 [%] (位置情報なし)
524,288	71.88	24.12
1,048,576	86.72	32.51
2,097,152	94.86	43.94
4,194,304	98.30	57.45
4,598,517	98.55	57.50

比較の結果，位置情報なしの場合の網羅率が 50%以下であるのに対し，位置情報ありの場合の網羅率が 98.55%まで上げられた．単純な位置情報を追加することで約 2〜3 倍程度の網羅率を向上することができた．また，レインボーテーブルの網羅率はチェーンの数に比例するが，ある程度の網羅率に到達すれば，チェーンの数を大幅に増やしてもレインボーテーブルの網羅率は微増に留まることが確認できた．

関連研究 [54] では，チェーン長が 50 で，チェーン数が 4,598,517 の場合，生成されたレインボーテーブルの網羅率が 99.09%と示されてる．今回の実装での網羅率は 98.55%であった．この結果の差異の理由は，対象ハッシュ関数の性質の違いによる．[54] のハッシュ値が 104 ビットであったのに対して，本研究のハッ

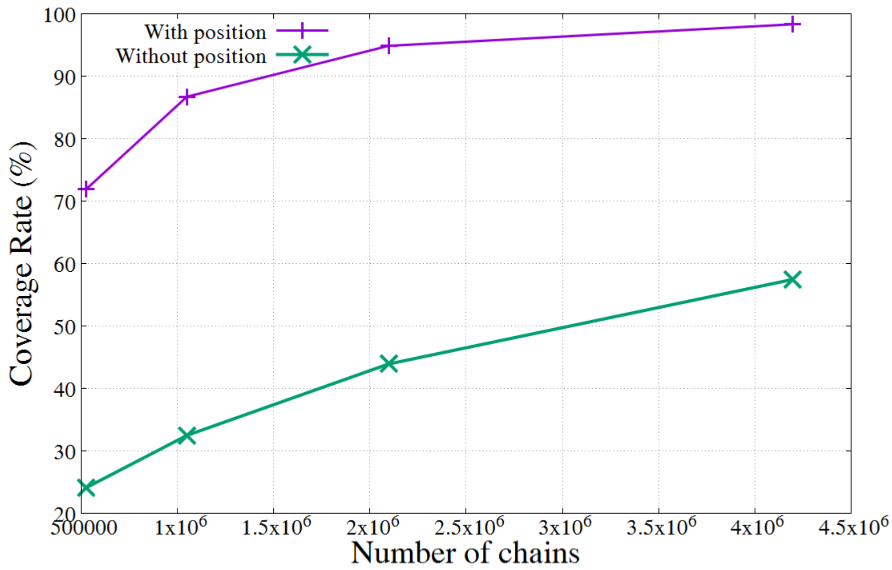


図 5.2 位置情報による網羅率への効果．

シュ値が 512 ビットである．還元関数の入力はハッシュ値であるため，[54] での還元関数はよりユニークなパスワード候補を生成することができる．さらに，SP が CPU によってランダムに生成されるため，SP でも重複が発生してしまってるためである．

5.2.2 チェーンの数による GPU 実装の効果

チェーンの数を変化させたときのレインボーテーブルの生成時間を表 5.2 と図 5.3 に示す．ここでは，GPU と CPU での生成時間を比較するために，チェーンの長さを 50 に固定し，チェーンの数を変更して，実験を行った．また，GPU ではスレッドブロックごとに 128 スレッドの 4,096 スレッドブロックを使用した．

実装結果では，GPU を使用することでレインボーテーブルの生成時間は，チェーンの数が多いほど，GPU による高速化率が高くなる．CPU で生成する場合と比較して約 70 倍高速になったことが確認できた．兼松らの関連研究 [54] では，9 倍の高速化を実現している．この関連研究 [54] では，レインボーテーブルの EP として 104 ビットのハッシュ値を保存していたのに対し，本研究では 16 ビットの

表 5.2　チェーンの数による GPU 実装の効果.

チェーンの数	CPU [秒]	GPU [秒]	CPU/GPU 倍率	網羅率 [%]
524,288	34.947	0.528	66.19	71.88
1,048,576	69.831	1.032	67.66	86.71
2,097,152	139.306	1.934	72.03	94.86
4,194,304	278.429	3.834	72.62	98.30

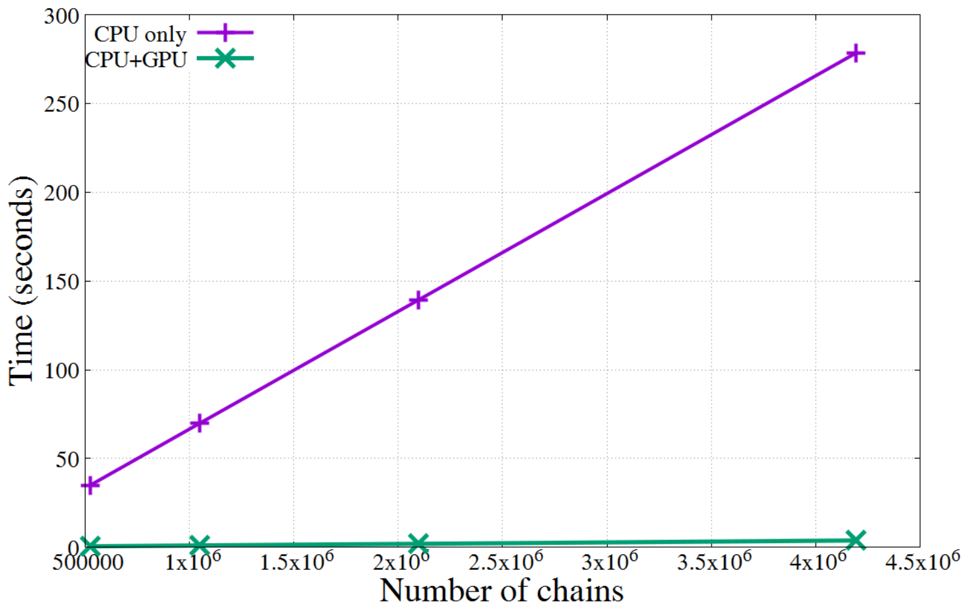


図 5.3　チェーンの数による GPU 実装の効果.

パスワード候補を EP として保存した．EP の書き込み時間は EP のビット長に比例するため，ビット長の短いパスワードを EP として保存する本研究の方が生成時間の効率が良いこととなる．

5.2.3　チェーンの長さによる GPU 実装の効果

チェーンの長さを変化させたときのレインボーテーブルの生成時間を表 5.3 と図 5.4 に示す．ここでは，GPU と CPU での生成時間を比較するために，チェーン

の数を 524,288 に固定し，チェーンの長さを変更しつつ，実験を行った．また，GPU ではスレッドブロックごとに 128 スレッドの 4,096 スレッドブロックを使用した．

表 5.3　チェーンの長さによる GPU 実装の効果．

チェーンの長さ	CPU [秒]	GPU [秒]	CPU/GPU 倍率	網羅率 [%]
10	7.180	0.499	14.38	30.04
30	22.585	0.511	44.19	58.08
50	34.947	0.527	66.19	71.88
100	74.520	0.553	134.86	83.66
200	148.634	0.621	239.42	86.08

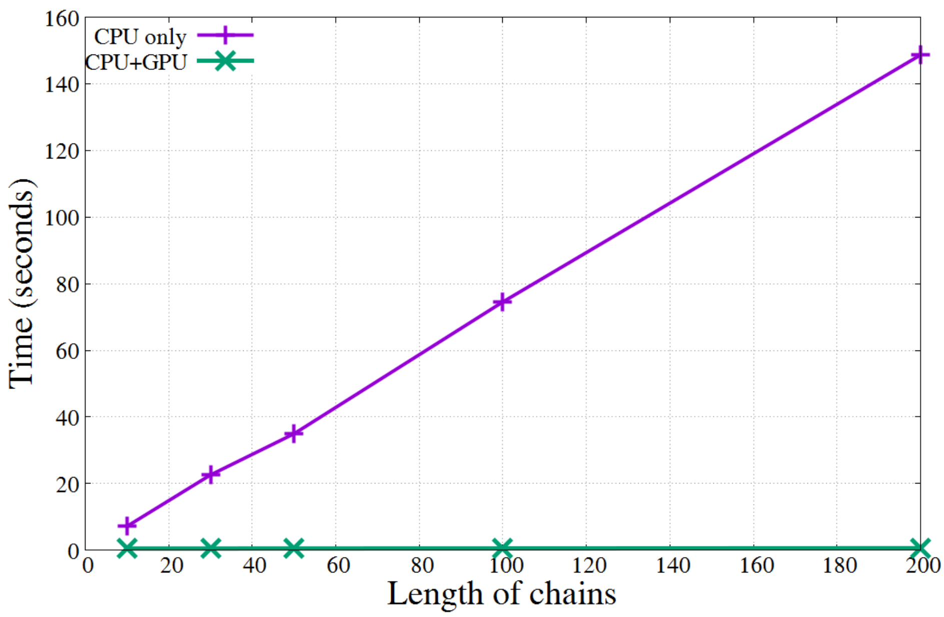


図 5.4　チェーンの長さによる GPU 実装の効果．

GPU で並列化される部分はチェーンの生成処理であったため，チェーンの長さ

が長くなるほど高速化の効果が大きくなった．実装した結果，チェーンの長さが200の場合，レインボーテーブルの生成時間はGPUを使用した場合はCPUを使用する場合よりも約239倍高速であった．

5.2.4　生成時間，メモリ使用量と網羅率の関係

レインボーテーブル攻撃では，ハッシュチェーンによるメモリの削減の代わりに，パスワードを取得するのにより多くの時間を必要とする．レインボーテーブルの生成は実際の攻撃に最も重要な準備であるため，生成されたテーブルのパスワード網羅率は大きく攻撃効果に影響を及ぼす．レインボーテーブルの生成時間，メモリ使用量とパスワード網羅率の関係を表5.4に示す．

表 5.4　生成時間，メモリ使用量と網羅率の関係．

チェーンの長さ	チェーンの数	CPU [秒]	GPU [秒]	CPU/GPU 倍率	メモリ量 [MB]	網羅率 [%]
200	2,097,152	556.980	2.347	237.31	20	97.16
50	1,048,576	69.831	1.032	67.66	10	86.71
	2,097,152	139.306	1.934	72.03	20	94.86
	4,194,304	278.429	3.834	72.62	41	98.30

SP-EPのみを保存するため，レインボーテーブルのメモリ使用量はチェーンの数に比例する．同じメモリ量を使用した場合，チェーンの長さが長いほど，GPUで生成する価値が高くなる．表5.4では，チェーンの長さが50，かつチェーンの数が2,097,152の場合，CPUの生成時間はGPUで生成するより72.03倍であるが，チェーンの長さを200に変更するとその倍率は237.31倍になるが合計生成時間は多少長くなる．この2つの場合では，SP-EPの数が同じであるがGPUで生成した合計のパスワード候補の数は異なるため，チェーンの長さが長い方がパスワード網羅率が高くなる．また，網羅率を上げられるように，チェーンの長さを短くし，チェーンの数を増やす必要があるため，メモリ使用量が大きくなり，GPUの効果が低くなる．

レインボーテーブルの生成において，生成時間とメモリ使用量及びパスワード網羅率は互いに関係を持つ．それぞれの目的でどの要素を重心にするかによって，チェーンの長さとチェーンの数を適切に変更することになる．また，実際に生成したレインボーテーブルを用いた攻撃の効果を検討する必要がある．

5.2.5　カーネル関数の実行時間

GPU を用いたプログラムの実行時間を各データ転送時間と，カーネルの計算時間，データの書き込み時間に分けて分析した結果を表 5.5 及び図 5.5 に示す．ここで，チェーンの数を変化させ，チェーンの長さを 50 に固定し，GPU ではスレッドブロックごとに 128 スレッドの 4,096 スレッドブロックを使用した．

表 5.5　テーブル生成の詳細な実行時間．

チェーンの数	カーネル実行 (GPU) [ms]	データ転送 [ms]	書き込み (CPU) [ms]
524,288	38.646	220.346	269.001
1,048,576	71.989	428.022	532.524
2,097,152	143.998	786.221	1,003.811
4,194,304	287.927	1,561.274	1,988.918

この結果から，カーネル実行時間，データ転送時間，及び SP-EP ペアの書き込み時間は，チェーンの数に比例することがわかった．また，チェーンの数によらず，GPU でのカーネル関数の実行時間は合計実行時間の約 7%しか掛からなかった．この結果から，チェーンの長さを長くし，チェーンの数を減らすことで全体の実行時間に占めるカーネル関数の実行時間を増やし，同じデータ量のレインボーテーブルをより少ない時間で生成することが可能であることが判明した．

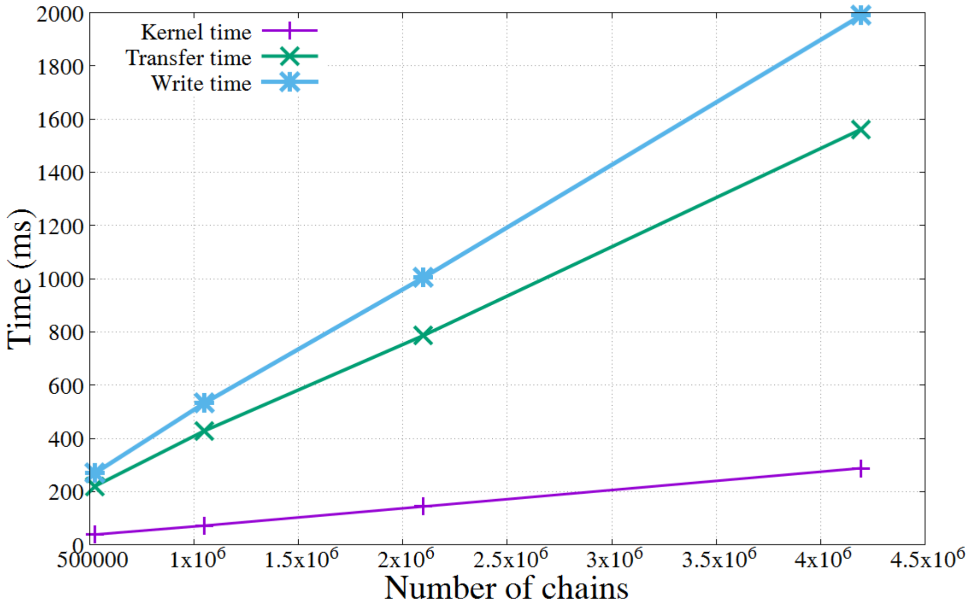


図 5.5　テーブル生成の詳細な実行時間.

第6章

結論

本研究においては，ハッシュ関数Keccak-512のGPUへの高速化実装を行った．ハッシュ関数の実装において，複数の入力ファイルに対した処理がバッチモードであり，1回に付き1つの大きいファイルに対するハッシュ化処理がTreeモードである．本研究の実装はそのバッチモードを実現し，一つのパスワードをGPU上の1スレッドを用いてハッシュ化処理を行ない，同時に多数のスレッドを起動して並列処理した．その実装においては，次の4つの高速化のための方法を提案した．一つ目はルックアップテーブルの再構成，二つ目は定数の適切なメモリへの配置である．三つ目はブロック数・スレッド数の変更，最後はGPUでのストリームを用いて，データ転送と計算のオーバーラップを行うことである．これらの手法を用いて実装した結果，最大64.582GB/sのスループットを実現できた．パスワードクラッキングツールであるHashcatを同じ環境で実装し，実行時間を測定した結果，最大一秒あたり769.6メガハッシュを処理できた．一方，本研究では一秒あたり999.6メガハッシュを処理できることを示してる．本研究の結果を用いてHashcatのようにパスワードに対する総当たり攻撃の性能を向上することが可能であることが明らかになった．

また，パスワードクラッキングの代表的な方法として，総当たり攻撃，辞書攻撃，そしてレインボーテーブルを用いた攻撃等が存在する．前2者の攻撃方法の欠点を改善したものがレインボーテーブルを用いた攻撃であり，辞書攻撃と同じように事前にデータを準備するため，総当たり攻撃よりも時間が掛からず，特殊な計算方法により辞書攻撃よりも少ないメモリで実現できる攻撃方法である．本研究では，レインボーテーブルの一つのチェーンをGPUの1スレッドに割り当て，並列処理を行うことで，生成処理の高速化を図った．具体例には，生成時

間を短縮するために，チェーン内のハッシュ化処理の部分において，ハッシュ関数 Keccak-512 の高速化実装手法を活用した．また，チェーンの生成処理において還元関数の性質により衝突が起きてしまうと，チェーン内のその後のパスワード候補が全て重複してしまい，レインボーテーブルの効率が落ちることになる．チェーン内や別のチェーンでこのパスワード候補の重複が発生しても次のチェーンの部分で重複されないように，還元関数にパスワード候補のチェーン内の位置情報を追加することとした．これにより，異なるチェーンの異なる位置におけるチェーンの重複を避けることができる．この手法で生成されたレインボーテーブルを評価し，攻撃効果の予測・議論を行った．4 文字のパスワードを対象とし，チェーンの長さが 50，かつチェーンの数が 4,598,517 の場合，生成されたレインボーテーブルのパスワード候補の網羅率は 98.55%であった．また，GPU を用いて高速化したレインボーテーブルの生成は CPU のみで生成する場合に比べると，約 239 倍高速化できることが確認された．

本研究では一秒あたり 999.6 メガハッシュを処理できることを示してる．英文字の小文字、大文字及び数字を使った 8 文字パスワードであれば 62^8（約 218 兆）通りの組合せが存在する．つまり本研究の実装と同じ環境であれば，約 60.7 時間（2 日半程度）で総当たり攻撃が可能となる．9 文字のパスワードの場合，所要時間は約 3,762 時間（156.7 日）である．また，レインボーテーブルを使用することにより，さらにパスワードクラッキングの時間を短縮することができる．従って，単なる Keccak-512 を用いてパスワードを 1 回や数回のハッシュ化処理を行い，そのハッシュ値でパスワードを管理することは非常に危険である．Keccak-512 も含め，ハッシュ関数を利用したパスワード管理システムでは，パスワードの後にソルトを付加することや 1,000 回以上ハッシュ化を反復する PBKDF2 のような関数を採用することで，パスワードクラッキングからパスワードを保護することが必要である．

今後の課題として，実際に本研究の実装を用いて総当たり攻撃や生成されたレ

インボーテーブルを用いたレインボーテーブル攻撃の効果を検証する必要がある．また，与えられたハッシュ値からもとのパスワードのクラック処理を行うことでパスワード管理における Keccak の安全性をより明確にすると共にパスワードクラッキングの対策として知られるソルト付加や複数回ハッシュ等の効果と必要性について明らかにする．

謝　　辞

この論文は，防衛大学校理工学研究科後期課程における研究成果として，防衛大学校情報工学科 岩井 啓輔 准教授，松原 隆 准教授及び 黒川 恭一 教授の御指導のもと執筆されたものです．研究の遂行および論文の執筆にあたり，多大な御指導および御鞭撻を賜りましたことに心からの感謝を申し上げます．

　また，研究の遂行に関し，比較的ストレスの多い研究環境の中，様々な面で支えていただいた コンピュータ工学研究室 の学生およびOBの皆様，ならびに防衛大学校入校期間中，関わりのあった全ての方々に対し，心からの感謝を申し上げます．

令和　２年 ３月

参 考 文 献

[1] LinkedIn. https://www.linkedin.com/.

[2] 2012 LinkedIn hack.
https://en.wikipedia.org/wiki/2012_LinkedIn_hack.

[3] 117 million LinkedIn emails and passwords
from a 2012 hack just got posted online.
http://techcrunch.com/2016/05/18/117-million-linkedin-emails-and-passwords-from-a-2012-hack-just-got-posted-online/, （参照 2019-12）.

[4] Ronald Linn Rivest. RFC 1320: The MD4 Message-Digest Algorithm. IETF. https://tools.ietf.org/html/rfc1320, 1992,（参照 2019-12）.

[5] Ronald Linn Rivest. RFC 1321: The MD5 Message-Digest Algorithm. IETF. https://tools.ietf.org/html/rfc1321）, 1992,（参照 2019-12）.

[6] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. *IACR Cryptology ePrint Archive*, Vol. 2004, p. 199, 2004.

[7] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In *Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT’05, pp. 19--35, 2005.

[8] Information Technology Laboratory National Institute of

Standards and Technology. FIPS PUB 180-4 secure hash standard (SHS). NIST. http://dx.doi.org/10.6028/NIST.FIPS.180-4, 2015, (参照 2019-12).

[9] Information Technology Laboratory National Institute of Standards and Technology. FIPS PUB 202 sha-3 standard: Permutation-based hash and extendable-output functions. NIST. http://dx.doi.org/10.6028/NIST.FIPS.202, 2015, (参照 2019-12).

[10] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Team Keccak. https://keccak.team, (参照 2019-12).

[11] CUDA Zone. https://developer.nvidia.com/category/zone/cuda-zone/, (参照 2019-12).

[12] Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. IETF. https://tools.ietf.org/html/rfc3447, 2003, (参照 2019-12).

[13] PKCS #1: RSA Encryption Version 1.5. IETF. https://tools.ietf.org/html/rfc2313, 1998.

[14] Information Technology Laboratory National Institute of Standards and Technology. FIPS PUB 186-4: Digital Signature Standard (DSS). NIST. http://dx.doi.org/10.6028/NIST.FIPS.202, 2015, (参照 2019-12).

[15] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. RFC2104: HMAC: Keyed-Hashing for Message Authentication. RFC Editor. https://doi.org/10.17487/RFC2104, 1997, (参照 2019-12).

[16] Jongsung Kim, Guil Kim, Sangjin Lee, Jongin Lim, and Junghwan Song. Related-Key Attacks on Reduced Rounds of SHACAL-2. In *Progress in Cryptology - INDOCRYPT 2004*, pp. 175--190, 2005.

[17] Ross Anderson and Eli Biham. Two practical and provably secure block ciphers: BEAR and LION. In *Fast Software Encryption*, Lecture Notes in Computer Science, Vol. 1039, pp. 113--120. Springer, Berlin, Heidelberg, 1996.

[18] Don Coppersmith and Phillip Rogaway. Software-efficient pseudorandom function and the use thereof for encryption, 1995. US Patent 5,454,039.

[19] Don Coppersmith and Phillip Rogaway. Computer readable device implementing a software-efficient pseudorandom function encryption, 1997. US Patent 5,675,652.

[20] Bart Preneel, Antoon Bosselaers, and Hans Dobbertin. The cryptographic hash function RIPEMD-160. *CryptoBytes 3(2)*, pp. 9--14, 1997.

[21] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. The sponge and duplex constructions. https://keccak.team/sponge_duplex.html, （参照 2019-12）.

[22] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Specifications. https://keccak.team/specifications.html,（参照 2019-12）.

[23] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël

Peeters, Gilles Van Assche, and Ronny Van Keer.
KECCAK sponge function family main document.
https://keccak.team/obsolete/Keccak-main-1.0.pdf, （参照 2019-12）.

[24] Information Technology Laboratory National Institute of Standards and Technology. FIPS PUB 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. NIST. http://dx.doi.org/10.6028/NIST.FIPS.202, 2015,（参照 2019-12）.

[25] 太田和夫. 暗号学的ハッシュ関数の衝突攻撃に対する安全性評価. 電気通信普及財団研究調査報告書, No. 22, pp. 290--297, 2007.

[26] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In *Advances in Cryptology - EUROCRYPT 2005*, Lecture Notes in Computer Science, Vol. 3494, pp. 19--35. Springer, Berlin, Heidelberg, 2005.

[27] Practical Free-Start Collision Attacks on 76-step SHA-1. In *Advances in Cryptology - CRYPTO 2015*, Lecture Notes in Computer Science, Vol. 9215, pp. 623--642. Springer, Berlin, Heidelberg, 2015.

[28] 情報処理推進機構. CRYPTREC Report 2015.
http://www.cryptrec.go.jp/report/c15_eval_web.pdf, 2015, （参照 2019-12）.

[29] 情報処理推進機構. SHA-1 の安全性について.
http://www.cryptrec.go.jp/topics/cryptrec_20151218_sha1_cryptanalysis.html, 2015,（参照 2019-12）.

[30] 渡辺大．共通鍵暗号と暗号危殆化問題．電子情報通信学会大会講演論文集，2012 基礎・境界，pp. 87--88，2012．

[31] Philippe Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. In *Advances in Cryptology - CRYPTO 2003*, Lecture Notes in Computer Science, Vol. 2729, pp. 617--630. Springer, Berlin, Heidelberg, 2003.

[32] RainbowCrack Project. RainbowCrack. http://project-rainbowcrack.com/,（参照 2019-12）．

[33] Distributed Rainbow Table Project. rcracki_mt. https://www.freerainbowtables.com/,（参照 2019-12）．

[34] Objectif Securite. Ophcrack. http://ophcrack.sourceforge.net/,（参照 2019-12）．

[35] Elcmsoft. Advanced Office Password Breaker. https://www.elcomsoft.jp/aopb.html,（参照 2019-12）．

[36] L0phtCrack. L0phtCrack 6. http://www.l0phtcrack.com/,（参照 2019-12）．

[37] Pierre-Louis Cayrel, Gerhard Hoffmann, and Michael Schneider. GPU Implementation of the Keccak Hash Function Family. In *Information Security and Assurance*, Communications in Computer and Information Science, Vol. 200, pp. 33--42. Springer, Berlin, Heidelberg, 2011.

[38] Guillaume Sevestre. Keccak Tree hashing on GPU, using Nvidia Cuda API. http://sites.google.com/site/keccaktreegpu/, 2010,（参照 2019-12）．

[39] Jason Lowden, Marcin Lukowiak, and Sonia Lopez Alarcon. Design and performance analysis of efficient KECCAK tree hashing on GPU architectures. *Journal of Computer Security*, Vol. 23, pp. 541--562, 2015.

[40] hashcat - advance password recovery. https://hashcat.net/hashcat/,（参照 2019-12）.

[41] GitHub-yoyosan/ccminer-alexis-1.0. https://github.com/yoyosan/ccminer-alexis-1.0,（参照 2019-12）.

[42] 崎山一男. ハッシュ関数 SHA-224，SHA-512/224，SHA-512/256 及び SHA-3(Keccak) に関する実装評価. 2013 年度 Cryptrec 外部技術報告. https://www.cryptrec.go.jp/exreport/cryptrec-ex-2301-2013.pdf, 2014,（参照 2019-12）.

[43] Brian Baldwin, Andrew Byrne, Liang Lu, Mark Hamilton, Neil Hanley, Maire O'Neill, and William Peter Marnane. FPGA Implementations of the Round Two SHA-3 Candidates. In *2010 International Conference on Field Programmable Logic and Applications*, pp. 400--407, 2010.

[44] Shin'ichiro Matsuo, Miroslav Knežević, Patrick Schaumont, Ingrid Verbauwhede, Akashi Satoh, Kazuo Sakiyama, and Kazuo Ota. How Can We Conduct "Fair and Consistent" Hardware Evaluation for SHA-3 Candidate? In *The Second SHA-3 Candidate Conference*. NIST Computer Security Resource Center, Accepted Papers. https://csrc.nist.gov/CSRC/media/Events/The-Second-SHA-3-

Candidate-Conference/documents/SHA3_Aug2010_Papers.zip, 2010, (参照 2019-12).

[45] Xu Guo, Sinan Huang, Leyla Nazh, and Patrick Schaumont. Fair and Comprehensive Performance Evaluation of 14 Second Round SHA-3 ASIC Implementations. In *The Second SHA-3 Candidate Conference*. NIST Computer Security Resource Center, Accepted Papers. https://csrc.nist.gov/CSRC/media/Events/The-Second-SHA-3-Candidate-Conference/documents/SHA3_Aug2010_Papers.zip, 2010, (参照 2019-12).

[46] Xu Guo, Meeta Srivastav, Sinan Huang, Dinesh Ganta, Michael Henry, Leyla Nazhandali, and Patrick Schaumont. Silicon Implementation of SHA-3 Finalists: BLAKE, Grøstl, JH, Keccak and Skein. In *ECRYPT II Hash Workshop*. CRYPTO, 2011.

[47] Kashif Latif, Muhammad Tariq, Arshad Aziz, and Athar Mahboob. Efficient Hardware Implementation of Secure Hash Algorithm (SHa-3) Finalist - Skein. In *Frontiers in Computer Education*, Advances in Intelligent and Soft Computing, Vol. 133, pp. 933--940. Springer, Berlin, Heidelberg, 2012.

[48] Miroslav Knežević, Kazuyuki Kobayashi, Jun Ikegami, Shin'ichiro Matsuo, Akashi Satoh, Ünal Kocabas, Junfeng Fan, Toshihiro Katashita, Takeshi Sugawara, Kazuo Sakiyama, Ingrid Verbauwhede, and Kazuo Ohta. Fair and Consistent Hardware Evaluation of Fourteen Round Two SHA3 Candidates. *IEEE*

Transactions on Very Large Scale Integration Systems - VLSI, Vol. 20, pp. 827--840, 2012.

[49] Bernhard Jungk and Marc Stöttinger. Among slow dwarfs and fast giants: A systematic design space exploration of KECCAK. In *ReCoSoC*, pp. 1--8. IEEE, 2013.

[50] Atefeh Gholipour and Sattar Mirzakuchaki. Throughput Optimum Architecture of KECCAK Hash Function. *International Journal of Computer and Electrical Engineering*, Vol. 4, pp. 937--939, 2012.

[51] Fábio Dacêncio Pereira, Edward David Moreno Ordonez, Ivan Daun Sakai, and Allan Mariano de Souza. Exploiting Parallelism on Keccak: FPGA and GPU Comparison. In *Parallel & Cloud Computing*, Vol. 2, pp. 1--6, 2013.

[52] Russell Edward Graves. High performance password cracking by implementing rainbow tables on nVidia graphics cards (IseCrack). Master's thesis, Iowa State. University, 2008.

[53] Julio Gómez, Francisco G. Montoya, R. Benedicto, A. Jimenez, Consolación Gil, and Alfredo Alcayde. Cryptanalysis of hash functions using advanced multiprocessing. In *Distributed Computing and Artificial Intelligence*, pp. 221--228, 2010.

[54] 卓也兼松，寛明桑原，哲太郎上原，義敏國枝．GPGPU によるレインボーテーブル生成の高速化．コンピュータセキュリティシンポジウム 2016 論文集，第 2016 巻，pp. 1260--1267, 2016.

[55] 伊藤智義．GPU プログラミング入門: CUDA5 による実装．講談社，2013.

[56] 青木尊之，額田彰．はじめての CUDA プログラミング： 驚異の開発環境 [GPU+CUDA] を使いこなす！ I/O books．工学社，2009．

[57] NVIDIA GeForce GTX 1080 Whitepaper. https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf,（参照 2019-12）.

[58] 西川尚紀．GPGPU による暗号アルゴリズムの並列化に関する研究．情報数理 (応用システム工学)，修士論文，防衛大学校，2010．

[59] NVIDIA. Tuning CUDA Applications for Pascal. https://docs.nvidia.com/cuda/pdf/Pascal_Tuning_Guide.pdf，（参照 2019-12），2019.

[60] NVIDIA. CUDA Runtime API. https://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf,（参照 2019-12），2019.

[61] 3DMARK BENCHMARK YOUR PC TODAY. https://www.3dmark.com,（参照 2019-12）.

[62] Seth Hoffert Michaël Peeters Gilles Van Assche Guido Bertoni, Joan Daemen and Ronny Van Keer. The Keccak reference. http://keccak.noekeon.org/Keccak-reference-3.0.pdf，（参照 2019-12），2011.

[63] CUDA Occupancy Calculator - Nvidia. https://developer.download.nvidia.com/compute/cu-da/CUDA_Occupancy_calculator.xls,（参照 2019-12）.

[64] NVIDIA Developer Blog. https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/,（参照 2019-12）.

[65] Guillaume Sevestre. Keccak Tree GPU Project sources. http://sites.google.com/site/keccaktreegpu/KeccakTreeGpu.zip,（参照 2019-12），2010.

[66] IPA（独立行政法人情報処理推進機構）． 情報セキュリティー. https://www.ipa.go.jp/security/,（参照 2019-12）．

[67] IPA（独立行政法人情報処理推進機構）． 情報セキュリティ10 大脅威 2019． https://www.ipa.go.jp/security/vuln/10threats2019.html,（参照 2019-12）．

[68] IPA（独立行政法人情報処理推進機構）． 不正ログイン対策特集ページ. https://www.ipa.go.jp/security/anshin/account_security.html,（参照 2019-12）．

[69] JPCERT． 適切なパスワードの設定・管理方法について. https://www.jpcert.or.jp/newsflash/2018040401.html,（参照 2019-12）．

[70] Address - Bitcoin Wiki. https://en.bitcoin.it/wiki/Address,（参照 2019-12）．

[71] LastPass. https://www.lastpass.com/,（参照 2019-12）．

[72] ブロックチェーン - 最も信頼されている仮想通貨企業. https://www.blockchain.com/,（参照 2019-12）．

[73] PKCS #5: Password-Based Cryptography Specification Version 2.0. IETF. https://tools.ietf.org/html/rfc1321, 2000,（参照 2019-12）．

[74] LastPass. パスワードの反復（PBKDF2）. https://helpdesk.lastpass.com/ja/account-settings/general/password-iterations-pbkdf2/,（参照 2019-12）.

研 究 業 績

学術誌

[1] Thuong Nguyen Dat, Keisuke Iwai, Takashi Matsubara, and Takakazu Kurokawa. Implementation of high speed hash function Keccak on GPU. *International Journal of Networking and Computing*, Vol. 9, No. 2, pp. 370--389, 2019.

国際学会

[1] Thuong Nguyen Dat, Keisuke Iwai, and Takakazu Kurokawa. Implementation of High Speed Hash Function Keccak Using CUDA on GTX 1080. In *2017 Fifth International Symposium on Computing and Networking (CANDAR)*, pp. 475--481, 2017.

[2] Thuong Nguyen Dat, Keisuke Iwai, Takashi Matsubara, and Takakazu Kurokawa. Implementation of High Speed Rainbow Table Generation Using Keccak Hashing Algorithm on GPU. In *2019 6th NAFOSTED Conference on Information and Computer Science (NICS)*, pp. 166--171, 2019.

国内学会

[1] グェン ダットトゥオン，岩井啓輔，黒川恭一．ハッシュ関数 Keccak の GPU 実装． 研究報告コンピュータセキュリティ（CSEC），Vol. 2016, No. 7, pp. 1--6, 2016.

[2] グェン ダットトゥオン，岩井啓輔，松原隆，黒川恭一．CUDA を用いたハッシュ関数Keccakに対応するレインボーテーブル生成の高速化．研究報告コンピュータシステム（CPSY），Vol. 119, No. 372, pp. 181--186, 2020.