ハッシュ関数 Keccak の GPU 実装

グエン　ダット　トゥオン[†1]　岩井　啓輔[†1]　黒川　恭一[†1]

概要：次世代ハッシュ関数 SHA-3 の候補であった Keccak は 2012 年 10 月 2 日のコンペティションの勝者として選定され，2015 年 8 月 5 日に正式版が FIPS PUB 202 として公表された．Keccak は，スポンジ構造に基づくハッシュ関数であり，MD5 や SHA-1 に対する攻撃の研究の進展に対応したものであった．
本研究では，GPU 向けの統合開発環境 CUDA を用いて，ハッシュ関数 Keccak（SHA3-512）の高速化実装を行った．Tesla K20Xm を用いて Keccak を実行した結果，ブロック数が 2,048 個，一つのブロックあたりのスレッド数が 32 個の場合，一秒あたり約 23.7M ハッシュを処理することが確認できた．

キーワード：ハッシュ関数 Keccak，SHA-3，GPU，CUDA，実装

Implementation of the hash function Keccak on GPU

THUONG NGUYENDAT[†1], KEISUKE IWAI[†1], and
TAKAKAZU KUROKAWA[†1]

Abstract: Keccak was selected as the winner of the competition on October 2, 2012 for the next-generation hash function SHA-3. The official version has been published as FIPS PUB 202 on August 5, 2015. A hash function Keccak is bases on the sponge structure, and is corresponding to the progress of research on the attack against MD5 and SHA-1.
In this paper, we are using the integrated development environment CUDA for GPU, aiming at the speed up of the implementation of Keccak (SHA3-512). Implementation result of Keccak on Tesla K20Xm confirmed that the process approximately 23.7M Hash per second with 2048 blocks and 32x2048 threads.

Keywords: Hash Function Keccak, SHA-3, GPU, CUDA, Implementation

1. はじめに

ネットワーク上で通信するデータにおいて，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．

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の安全性は低下してきている．

次世代ハッシュ関数 SHA-3 の候補であった Keccak は 2012 年 10 月 2 日のコンペティションの勝者として選定され，2015 年 8 月 5 日に正式版が FIPS PUB 202[1]として公表された．Keccak は，スポンジ構造に基づくハッシュ関数であり, MD5 や SHA-1 に対する攻撃の研究の進展に対応したものであった．

Guillaume Sevestre らの公開されている先行研究[2]では，Tree 構造による Keccak の GPU への実装を行った. GeForce GTS 250 グラフィック・カードに実装した結果，1,183MB/s のスループットとなったことが示されている．

本稿では GPU 向けの統合開発環境 CUDA を用いてハッシュ関数 Keccak（SHA3-512）の実装を行い，処理速度を測定する．その結果から，パスワード管理における Keccak の安全性について検討する．

2. GPU と CUDA プログラミング

2.1 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 ProcessingUnit（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．その特徴により，数値演算に GPU を使った GPGPU（GeneralPurpose Computation on Graphics Processing Unit）の研究が盛んになっている．[3][4]

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ため，GPGPU のプログラミングは容易ではなかった．しかし，2006 年 11 月に NVIDIA 社が C 言語を用いた開発環境である GPU コンピューティング環境 CUDA（Compute Unied Device Architecture）[5]をリリースし，GPGPU のプログラミングが容易になり，広く普及した．さらに，ビデオカードの機能を目的とせずに，数値計算を高速化する GPGPU 専用の

†1 防衛大学校
National Defense Academy

ⓒ2016 Information Processing Society of Japan

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．

2.2 CUDA のプログラム階層・GPU の構造

CUDA のプログラム階層構造は，図 1 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はブロックとなる．さらに，ブロックをまとめたものはグリッドである．

GPU は，複数のストリーミングマルチプロセッサとメモリなどの周辺回路がブロック状に並んだ構成になっている．1 つのブロックは 1 つのストリーミングマルチプロセッサに割り当てられる．

それぞれのストリーミングマルチプロセッサごとに 32 個（Kepler アーキテクチャでは，192 個）の「ストリーミングプロセッサ」が搭載されており，このストリーミングプロセッサによって並列処理が行われる．CUDA のプログラム構造では，１スレッドは１つのストリーミングプロセッサによって計算される．

また，各ストリーミングマルチプロセッサ内には，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である．

そして，全てのストリーミングプロセッサ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．[5]

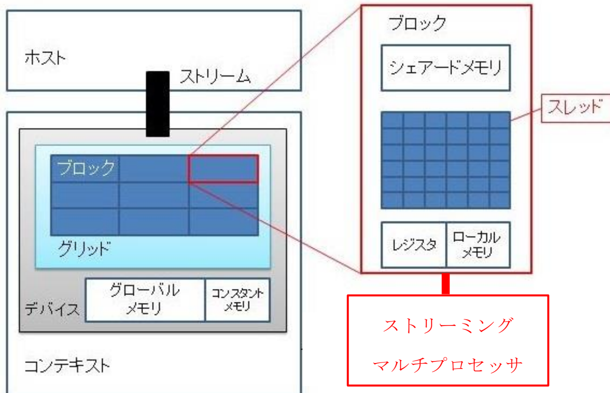

図１．CUDA プログラムの階層構造

2.3 CUDA

CUDA（Compute Unified Device Architecture）[4]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 2 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プログラムはホスト側で起動する．[5]

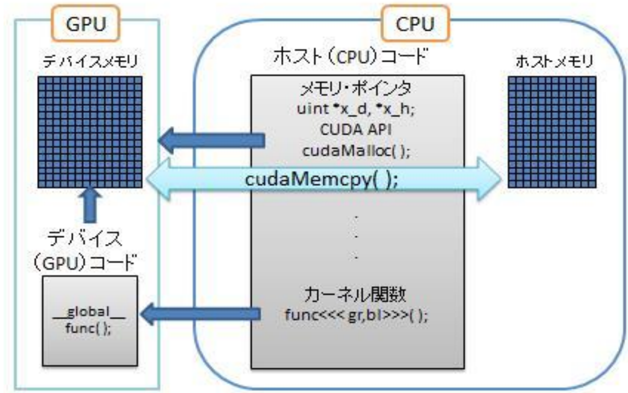

図 2．CUDA プログラム構成のイメージ

3. ハッシュ関数 Keccak の仕様

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シュ関数である．

3.1 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 3 に示す．

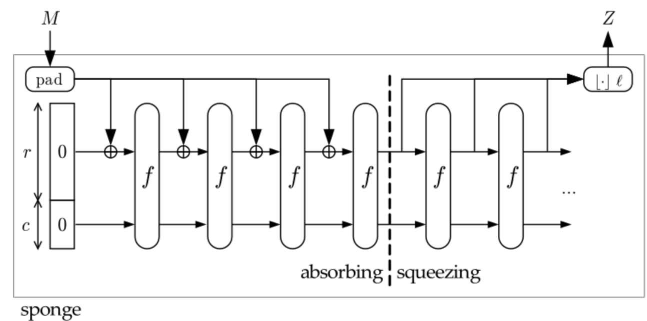

図 3．スポンジ構造 (The sponge construction Z=SPONGE[f,pad,r](N,d)) [5]より引用

スポンジ構造は，absorbing と squeezing の大きく 2 つのフェーズに分かれている．absorbing では，パディング後のメッセージデータ Mp を r [bit] のデータに分割し，ステートの r [bit] のデータとの XOR 演算の後に関数 f に入力する．squeezing では，f 関数とこの取り出しを行う．

3.2 デュプレックス構造

Keccak には，前述のスポンジ構造において absorbing と squeezing を交互に行うデュプレックス構造（図 4）も用意されている．

ⓒ2016 Information Processing Society of Japan

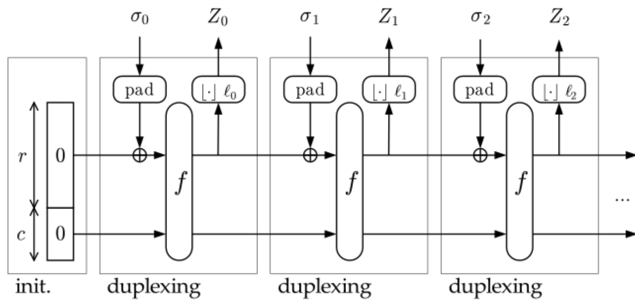

図 4. デュプレックス構造(The duplex construction) [5]より引用

3.3 Keccak-f 置換関数

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 次元のステートを計算する．本研究の対象は Keccak-512 であるため，ラウンド数が 24 であった．それぞれのラウンドでは，次の処理が行われる．

（詳細は FIPS PUB 202[1]を参照）

(1) θ ステップ（図 5）

x と y が 0…4 で,

C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4]

D[x] = C[x-1] xor rot(C[x+1],1)

A[x,y] = A[x,y] xor D[x] (1)

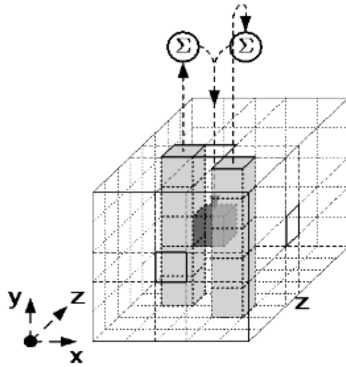

図 5. θ ステップ

(2) ρ と π ステップ（図 6，7）

x と y が 0…4 で,

B[y,2*x+3*y] = rot(A[x,y], r[x,y]) (2)

（rot (W, r) はビット毎の巡回右シフト演算であり，位置 i のビットを位置 i + r に移動する）

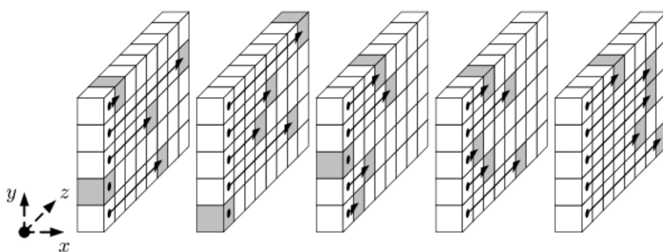

図 6. ρ ステップ

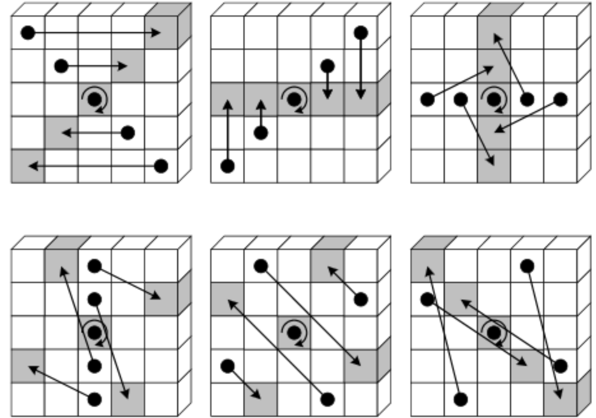

図 7. π ステップ

(3) χ ステップ（図 8）

x と y が 0…4 で,

A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y]) (3)

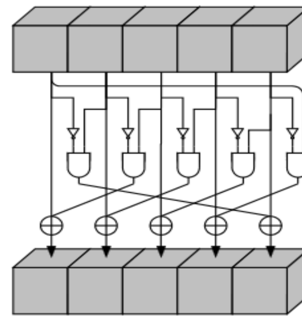

図 8. χ ステップ

(4) ι ステップ

A[0,0] = A[0,0] xor RC (4)

（RC はラウンド定数（round constants）である．）

（図 5～8：[6][7]より引用）

3.4 ハッシュ関数 SHA-3

入力メッセージMに対し，ハッシュ関数 SHA-3 は以下の通り定義されている．[1]

SHA3-224(M) = Keccak[448](M|| 01, 224);

SHA3-256(M) = Keccak[512](M|| 01, 256);

SHA3-384(M) = Keccak[768](M|| 01, 384);

SHA3-512(M) = Keccak[1024](M|| 01, 512); (5)

SHAKE128(M,d) = Keccak[256](M||1111,d);

SHAKE256(M,d) = Keccak[512](M||1111,d); (6)

本研究の対象は，ハッシュ関数 SHA3-512(M)とした．

4. Keccak の GPU 実装

4.1 概要

本研究では, CUDA を用いてハッシュ関数 Keccak の GPU

ⓒ2016 Information Processing Society of Japan

実装を行った．

パスワードを想定した入力メッセージ M に対し，ハッシュ値 SHA3-512(M)，つまり Keccak[1024](M|| 01, 512)を計算した．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，各スレッドがそれに割り当てられた入力メッセージのハッシュ処理を行った．これは総当たり攻撃（brute force attack）のように大量のハッシュ処理を行うプログラムとなる．

4.2 高速化 CUDA プログラムの流れ

今回の高速化実装プログラムの大まかな流れを逐次的に列挙すると，次のようになる．

① cudaSetDevice(GPU 番号）を用いて，使用する GPU（デバイス）を選択する．

② cudaMalloc で GPU 側メモリを宣言し，確保する．

③ CPU 側で入力メッセージを生成し，2 次元配列に格納する．その後，cudaMemcpyHostToDevice を用いてその入力メッセージの情報を GPU 側に転送する．

④ __global__ void kernel() で定義したカーネル関数を kernel<<<ブロック，スレッド>>>() のように呼び出し，GPU の各スレッドでハッシュ処理を行う．

⑤ 時間を測定し，その結果を表示する．

⑥ 使用したメモリを解放する．

4.3 高速化実装の提案手法

GPU でのプログラム実行の効率を向上させるために今回用いた高速化手法を次に示す．

(1) 条件分岐の回避

GPU では，条件分岐命令によって性能が低下する．今回の実装では，入力メッセージはパスワードを想定したものであるため，パディング後の長さを固定することになり，ブロックに対するループを除去できた．これにより，カーネル関数内で条件分岐命令を減少させた．

(2) GPU 上のメモリ階層の有効利用

本実装では，ハッシュ処理において各スレッドで共有できる定数をコンスタントメモリに設定した．コンスタントメモリは，スレッドで共通の変数を置くところであり，各スレッドから高速にアクセスできる．しかし，使用するスレッド総数が膨大な数に増えると，同時にコンスタントメモリへのアクセスが集中するため速度が低下する．そこでさらにシェアードメモリを利用し，より高速にアクセスすることにした．

より効率的にカーネル関数を実行できるように，キャッシュの設定を行い，カーネル関数の直前で次のように呼び出した：

cudaFuncSetCacheConfig(Keccak,cudaFuncCachePreferL1);[8]

GPU でレジスタのアクセス速度は速いがレジスタの数が限られている．そのため，レジスタを多く取り過ぎないように処理に影響しない変数を再使用した．

(3) ブロック数・スレッド数の構成変更

GPU はスレッド数･ブロック数の組み合わせ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，これらの数の組み合わせを変化させ，最適な組み合わせを調べた．

5. 実装結果

5.1 実装環境

表 1 と表 2 を用いて，実装環境を示す．

表 1．使用した GPU のスペック

メーカ	Tesla K20Xm
グローバルメモリ	5 GB
CUDA コア数	2688 (14*192)
Warp サイズ	32

表 2．実装環境とコンパイルオプション

OS	CentOS release 6.2 (kernel ver 2.6.32)
CPU	Intel Xeon E5-2620 (2.00Ghz, 6 Core)
GPU	Tesla K20Xm (2688 CUDA Core)
CUDA	Ver. 6.0
Compiler	gcc ver 4.4.7; nvcc ver 6.0 (CUDA)
Compiler option	CPU: -O3; GPU: -O3 (最適化オプション)

5.2 GPU 上のメモリ階層の有効利用

CUDA プログラムで提案手法による処理速度の向上を検討するために，提案手法を使わないプログラム及び改良したプログラム両方のハッシュ処理を測定して比較した．

4．3 の(1)，(2)で述べた提案手法を使わない，改良前のプログラムでは，長さ任意の入力メッセージを対応し，適量のレジスタに改良しない，処理に必要な定数をコンスタントメモリに格納すると共にデフォルトのキャッシュ設定で実行した．

改良後のプログラムは固定長の入力メッセージ（70 文字まで対応可能)をハッシュ処理し，レジスタ数を最適化し，一部のコンスタントメモリを使った値をシェアードメモリに移した，キャッシュ設定も設定して実行した．

キャッシュ設定のオプションで cudaFuncCachePreferL1（L1 キャッシュを 48kB，シェアードメモリを 16kB）を使用する場合に一番効果が得られた．

１ブロックあたり 128 個のスレッドを使用した場合の実行結果は表 3 に示す．

表 3. 改良前・後の処理スループットの比較「単位 : MH/s」

入力メッセージ数	改良前	改良後
8192	6.255	15.181
16384	7.606	17.248
32768	8.448	19.710
65536	8.755	20.995
131072	8.889	22.028
262144	8.954	22.503
524288	8.988	22.620
1048576	9.009	22.748

提案手法によって，GPU でハッシュ関数 Keccak の処理スループットを 2.5 倍上げることができた．

5.3 GPU のブロック・スレッド数の変化

ブロック数及び 1 ブロックあたりのスレッド数を変更しつつ，ハッシュ処理のスループットを測定した結果を表 4 に示す．

この結果のグラフを図 9 に示す．ブロック数が約 256 個以上になると，スループットが向上することが確認できた．また，1 ブロックあたりのスレッド数が 32 個の場合，最大のスループットが得られた．

表 4. ブロック・スレッド構成によるスループットの変化

	16	32	64	128	256
8	0.747	1.437	2.921	5.670	9.023
16	1.440	2.881	5.638	8.835	9.996
32	2.944	5.796	11.054	13.624	13.347
64	5.581	10.762	14.431	15.181	16.478
128	7.966	14.414	15.606	17.248	17.121
256	8.851	15.578	18.389	19.710	18.249
512	11.040	18.258	19.797	20.995	18.852
1024	12.850	20.858	21.952	22.028	19.032
2048	13.922	22.366	22.289	22.503	19.157
4096	14.463	23.173	22.859	22.620	19.232
8192	14.717	23.505	23.001	22.748	19.259
16384	14.947	23.760	23.018	22.799	19.273

＊単位：MH/s
＊＊一列目：ブロック数,
一行目：一ブロックあたりのスレッド数

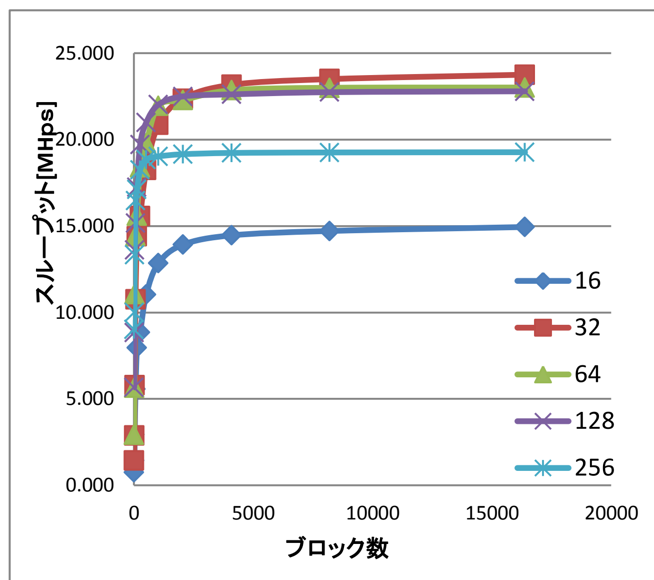

図 9. ブロック・スレッド構成によるスループットの変化

5.4 CPU と GPU の処理時間比較

CPU と GPU での処理速度を比較するために，同じアルゴリズムで実装を行い，それぞれの実行時間を測定した．入力メッセージの数を変更し，ハッシュの処理時間を表 5 に示す．

GPU では，一ブロック当たりのスレッド数を 128 個と設定した．スレッド総数が入力メッセージと一致するようにブロック数を増やして測定を行った．

表 5. Keccak の処理時間の比較「単位：ミリ秒」

メッセージ数	CPU	GPU
8192	10.000	0.499
16384	20.000	0.945
32768	40.000	1.681
65536	80.000	3.132
131072	140.000	5.950
262144	260.000	11.649
524288	510.000	23.178
1048576	980.000	46.096

この結果を図 10 で確認すると，圧倒的に GPU の処理能力が高いことがわかった．入力メッセージ数が 131,072 メッセージ（512*128）の場合，GPU の処理は CPU より 25.5 倍の速度となることが確認できた．

ⓒ2016 Information Processing Society of Japan

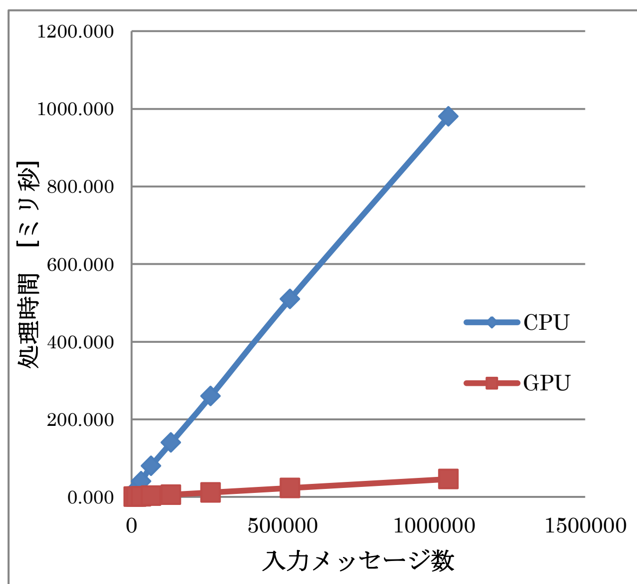

図 10. CPU と GPU の処理時間の比較

6. 考察

CUDA を用いて Keccak の GPU への高速化実装を行った結果，1 ブロックあたり 32 個のスレッドの場合に最大のスループットが出ていることが確認できた．しかし，パスワードクラックの前提を考えると入力メッセージが多くなり，全体の処理時間も考慮すると，1 ブロックあたり 256 個のスレッドが効果を出せると考えられる．

1 ブロックあたりのスレッド数が 32 個の場合，1 秒間に最大 23.7 メガハッシュを処理できた．本研究では，パスワードを想定した入力メッセージであったため，パディングの処理は単純になり，70 文字までの入力メッセージまで対応できる一方，70 文字までパスワードの文字数を増やしても処理速度が低下せず，安定したスループットを出せることができる．本研究の対象であった Keccak-512 のハッシュ値は 64 文字になるので，さらにこのハッシュ値を入力メッセージとした場合にも対応できる．

先行研究である Guillaume Sevestre らの Tree 構造による Keccak の GPU への実装が公開されているソースコード[9]を同じマシンで実行した結果，GPU の処理速度は 1,117,550 kB/s であった．本研究で 70 文字の入力メッセージで考える場合，最大 1,663,177 kB/s の処理速度となった．Tree 構造は，大きいなブロックの入力に対するハッシュ処理が非常に優れていたが，本研究の対象入力はパスワードを想定した入力メッセージであったため，効果が少ないと考えられる．

7. おわりに

ハッシュ関数 Keccak SHA3-512 に対し CUDA を用いて GPU への高速化実装を行った．処理速度を比較するため，同じプログラムを CPU 及び GPU に実装し，処理時間を測定した．Tesla K20m の GPU は Xeon E5-2620 の CPU より最大 25.5 倍の速度でハッシュ処理を行った．

GPU のパフォーマンスを上げるために，カーネル関数内の条件分岐の回避，効率的なメモリの利用及びブロック数，スレッド数の組み合わせを検討し実験を行った．GPU では 1 ブロックあたり 32 個の場合，最大のスループットを出せることができた．1 秒当たり約 23.7 メガハッシュを処理できた結果から考えると，約 2.2e14 の組合せが存在する英文字の小文字，大文字及び数字を使った 8 文字のパスワードに対し，全体の検索に必要な時間は 2,572 時間（3 か月以上）となる．MD5，SHA-1 に比べるとより安全性が持っていることが確認できた．

今後の課題として，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検証し，さらなるパスワード管理における Keccak の可能性を考察する．

参考文献

[1] “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions”. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf, (2015-8).
[2] “Keccak Tree Gpu Project documentation“. http://sites. google. com/site/keccaktreegpu/GPU_Keccak_doc. pdf.
[3] 青木尊之，額田彰． はじめての CUDA プログラミング． 工学社, 2009.
[4] 伊藤智義． GPU プログラミング入門 CUDA5 による実装． 講談社，2013.
[5] “CUDA Zone “． developer. nvidia. com/category/zone/cuda-zone/.
[6] G． Bertoni, J． Daemen, M． Peeters, and G． Van Assche． “The Sponge Functions Corner“． http://sponge.noekeon.org/.
[7] “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”. http://cryptrec.go.jp/estimation/techrep_id2301.pdf.
[8] “CUDA Toolkit Documentation”. http://docs． nvidia. com/cuda/cuda-runtime-api/group__CUDART__EXECUTION. html.
[9] “Keccak Tree Gpu Project sources“. http://sites.google.com/site/keccaktreegpu/KeccakTreeGpu.zip.

ⓒ2016 Information Processing Society of Japan