2021 年度

早稲田大学大学院基幹理工学研究科

情報理工・情報通信専攻　修士論文

準同型暗号処理で多用される Trace-Type Function の AVX512 による高速化

提出日：2022 年 1 月 24 日

指導：山名 早人 教授

研究指導名：並列・分散アーキテクチャ

学籍番号：5120F007

井上　紘太朗

概要

クラウドサービスの普及により，企業が収集したデータの分析を外部に委託するという事例が増えつつある．こうしたデータを情報漏洩やプライバシー侵害のリスクから保護しつつ利活用する技術として，準同型暗号が挙げられる．準同型暗号は，データを暗号化した状態で計算を行うことができる暗号方式である．特に，近年実用化が進められている Ring Learning With Errors（RLWE）ベースの暗号方式では，ベクトル全体を一つの平文とみなすパッキングという手法により，ベクトルの各要素（スロット）ごとに並列に計算を行うことが可能である．RLWE ベースの暗号方式が行える演算として，加算（準同型加算）や乗算（準同型乗算）に加えて，スロットをシフトする rotation などが存在する．データ分析などのアプリケーションでは，ベクトルに対して rotation と準同型加算を組み合わせて行う trace-type function と呼ばれる演算が用いられる．例えば，ベクトルの総和を求める演算がこれにあたる．しかし，trace-type function の評価には，準同型加算や準同型乗算に比べて時間がかかる．この trace-type function の高速化手法として loop-unrolling を適用する手法が提案されている．本研究では，loop-unrolling の手法に加え，AVX512 を活用した SIMD 演算による高速化を行う．提案手法により，従来手法と比較して 1.05-2.30 倍の計算速度向上を達成した．

目次

第 1 章

はじめに

インターネットの普及に伴い，市場に流通するデータの量は爆発的に増大している．IT 調査会社 IDC の報告によると，2020 年に全世界で 64.2ZB のデータが作成または複製された [1]．また，クラウドサービスの普及に伴い，大量のデータを収集・分析し，活用するまでを一貫して行うことが容易になった．近年では，保有するデータを活用したい企業や個人が，分析のノウハウを持つ第三者へ分析業務を委託するという事例が増えてきている．扱われるデータの中には，遺伝子情報などの個人情報や，企業の機密情報なども存在する．こうしたデータを，プライバシー侵害や情報漏洩のリスクから保護しつつ，安全に利活用する手法について，近年盛んに研究が進められている．その要素技術の一つとして，準同型暗号が挙げられる．

準同型暗号（Homomorphic Encryption:HE）は，データを暗号化した状態で計算を行うことができる暗号方式である．とくに，2009 年に Gentry が初めて構成に成功した方式 [2] は完全準同型暗号（Fully HE:FHE）と呼ばれ，暗号化した状態での加算（準同型加算）と乗算（準同型乗算）が任意回数実行可能な暗号方式である．Gentry による提案以降，扱える値や演算において異なる特徴を持つ様々な暗号方式が提案されている．準同型暗号は，安全性を保証する基盤として数学的な問題を採用している．その問題の一つである Ring Learning With Errors（RLWE）において，暗号文は多項式で表現されている．RLWE ベースの暗号方式では，ベクトル全体を一つの平文とみなすパッキング [3, 4] という手法が存在する．パッキングを用いることで，SIMD（Single Instruction Multiple Data）の要領でベクトルの各要素（スロット）ごとに並列で演算を行うことが可能である．また，準同型暗号上でスロットをシフトする演算を rotation と呼ぶ．実用的なアプリケーションの中には，rotation と準同型加算を連続して適用する演算が用いられることが多く，trace-type function と呼ばれている [5]．例えば，ベクトルの総和をとる演算がこれにあたる．しかし，準同型暗号上の演算は，通常の演算と比較して時間・空間計算量が大きいという問題を抱えている．準同型加算（準同型乗算）は，通常の加算（乗算）と比較し 27,721 倍（168 倍）の実行時間を要する [6]．また，rotation は準同型乗算と比較して 7.15〜8.88 倍の実行時間を要する [7]．とくに，trace-type function は $\mathcal{O}(\log N)$ の rotation を必要とするため，準同型乗算と比較して 85.83〜124.26 倍の実行時間を要する．

準同型暗号の実用化に向けて，計算量の問題は大きな障害となる．そのため，性能向上に向けて理論と実装の双方から盛んに研究が行われている．準同型暗号の性能向上においては，並列実行可能な部分をハードウェアの支援によって並列化していくという方法が研究されている．とくに，CPU（Central Processing Unit）の SIMD 拡張命令セットを用いた手法は，GPU（Graphics Processing Unit）や FPGA（Field Programmable Gate Array）といった特別なハードウェアを必要としないため，よく用いられる．近年，CPU の SIMD 拡張命令セットが扱えるレジスタ長は増えつつあり，より CPU 上で効率良く処理を行えるようになった．その一つが Intel AVX512（Advanced Vector eXtensions）である．AVX512 は，AVX2 の後継として Intel が x86 命令セットに実装した SIMD 拡張命令であり，そのレジスタ長は 512bit である．この AVX512 を活用し，準同型暗号で多用される剰余演算を高速化したものが Intel HEXL である [8]．しかし，AVX512 を用いた trace-type function の演算最適化と性能評価に関する報告はこれまでに無く，AVX512 を用いた trace-type function の演算最適化への有効性を示す

ことは本分野の実用化において喫緊の課題である．

そこで，本研究では trace-type function の演算に焦点を絞り，AVX512 を活用した SIMD 化による最適化を行い, その効果を確認することを目的とする．とくに, trace-type function の最適化手法の一つである loop-unrolling と AVX512 による SIMD 演算の親和性について，シングルスレッドとマルチスレッドの双方の側面から考察を行う．

1.1 論文の構成

第 2 章では，本研究の内容理解に必要な要素技術について述べる．第 3 章では，関連研究を紹介する．第 4 章では，本研究の提案手法について説明する．第 5 章では，本研究で行った実験と結果，得られた結果についての考察を行い，第 6 章でまとめる．

第 2 章

関連技術

本章では，本研究の内容理解に必要な要素技術について述べる． まず，本稿で用いる記号の定義を行う．次に，準同型暗号について述べる．とくに，本研究で用いる CKKS（Cheon-Kim-Kim-Song）方式 [3, 9] について詳しく述べる．最後に，本研究で扱う trace-type function について述べる．

2.1 記号の定義

$\mathbb{N}, \mathbb{Z}, \mathbb{R}, \mathbb{C}, \mathbb{Q}$ を，それぞれ自然数，整数，実数，複素数，有理数を表す集合とする．N を 2 べきの整数とし，$K = \mathbb{Q}[X]/(X^N + 1)$ を $2N$-円分体，およびその整数環を $R = \mathbb{Z}/(X^N + 1)$ とする．すなわち，R 上の任意の元は，たかだか $(N-1)$ 次の多項式の係数列である．R の元を $a(X)$ のように表す．また，整数 q を法（modulus）とする R 上の剰余環 R/qR を R_q と表記する．ベクトルは特に断りのない限り行ベクトルとし，ベクトル $\mathbf{a}$ の i 番目の要素を $\mathbf{a}[i]$ と表記する．整数 q について，$\mathbb{Z} \cap [-q/2, q/2)$ を $\mathbb{Z}_q$ と表記する．ただし，実装上は $\mathbb{Z}_q$ を $\mathbb{Z} \cap [0, q)$ としている．q を法として，整数 a の剰余をとる演算を $[a]_q$ と表記し，$[a]_q = 0$ が成立することを $q|a$ と表記する．また，正の整数 a について，$[0, 1, ..., a-1] \cap \mathbb{Z}$ の範囲を $[a]$ と表記し，$\mathbb{Z}^*_{2N} = 2k + 1_{k \in [N]}$ を $2N$ より小さい正の整数の集合とする．ベクトル $\mathbf{a}, \mathbf{b}$ の要素ごとの積を $\mathbf{a} \circ \mathbf{b}$，内積を $\langle \mathbf{a}, \mathbf{b} \rangle$ と表記する．集合 S から一様にランダムな要素をサンプリングすることを $\xleftarrow{u} S$，ある分布 X からサンプリングすることを $\leftarrow X$ と表記する．また，$|\cdot|$ は配列や集合の要素数を表す．$\oplus$ と $\otimes$ は，それぞれ準同型加算と準同型乗算を表す．

2.2 準同型暗号

本節では，準同型暗号（Homomorphic Encryption: HE）の概要と実装上の特徴について述べる．

準同型暗号は，データを暗号化した状態で演算が可能な暗号方式である．例えば，$m1$，$m2$ を平文とし，$\mathsf{Dec}, \mathsf{Enc}$ をそれぞれ復号，暗号化を行うアルゴリズムとしたとき，ある演算 $\cdot$ について以下のような関係が成り立つ．

$$m1 \cdot m2 \approx \mathsf{Dec}(\mathsf{Enc}(m1) \cdot \mathsf{Enc}(m2)) \tag{2.1}$$

2009 年に Gentry が初めて完全準同型暗号（Fully HE:FHE）の構成に成功して以降，様々な準同型暗号の方式が提案されている．これらの方式は，それぞれ平文として扱える値の種類，計算可能な演算の種類や回数などが異なっている．主要な準同型暗号の方式を表 2.1 に示す．本研究で採用した CKKS 方式 [3, 9] は，復号結果に誤差が含まれることを許容することにより，実数や複素数を扱うことのできる準同型暗号の一方式である．CKKS 方式は，機械学習等の実アプリケーションで幅広く利用されている．

表 2.1　主要な準同型暗号の方式

提案年	名称	扱える値
2012	B/FV [10, 11]	整数
2014	BGV [12]	整数 or ブール代数
2018	CGGI(TFHE) [13]	ブール代数
2017	CKKS [3, 9]	複素数 or 実数

2.2.1　剰余数系（Residue Number System）

準同型暗号では，暗号文の法がワードサイズ（x86-64 環境では 64bit）に収まらないほど大きい場合が多く，通常では実装に多倍長整数が必要となる．しかし，多倍長整数を用いた演算は計算コストが大きいため，中国剰余定理（Chinese Remainder Theorem: CRT）を用いて，巨大な法をワードサイズに収まる複数の小さな法に分割する剰余数系（Residue Number System: RNS）と呼ばれる手法がよく用いられる [14]．Q_l を暗号文の法とし，$Q_l = \prod_{i=0}^{l} q_i$ を満たす互いに素な整数 q_i をとる．このとき，q_i はワードサイズに収まるように選択する．すると，$\mathbb{Z}_{Q_l}$ 上の計算は $\mathbb{Z}_{q_i}$ ごとに並列で行うことができる．このように，RNS により多倍長演算にかかる計算コストを削減することができる．本稿では，$x \in R_{Q_l}$ について，$\{x^{(i)}\}_{i\in[l+1]}$ を x の RNS 表現（representation）と呼び，$x^{(i)} = [x]_{q_i}$ を RNS component，$\{q_0, ..., q_l\}$ を RNS 表現の基底（base）と呼ぶ．

RLWE ベースの方式では，多項式環と呼ばれる数学的構造が用いられており，平文や暗号文は多項式の係数列として表現されている．用いる多項式の次数 N について，準同型加算の計算量は係数同士の和で表現されるため $\mathcal{O}(N)$ で計算可能である．対して，準同型乗算は多項式同士の畳み込みを行う必要があるため，計算量は $\mathcal{O}(N^2)$ となる．準同型乗算の計算コストを削減するために用いられる手法が，数論変換（Number Theoretic Transform: NTT）である．本稿では，NTT の逆変換を inverse-NTT（invNTT）と表記する．N を多項式の次数とすると，NTT（および invNTT）の計算量は $\mathcal{O}(N \log_2 N)$ である．NTT を適用した暗号文同士の乗算は係数ごとに $\mathcal{O}(N)$ で行うことが可能となり，全体の計算量は $\mathcal{O}(N \log N)$ となる．ゆえに，多くの準同型暗号の実装では準同型乗算の最適化に NTT が用いられている．とくに，RNS と NTT を組み合わせた手法は，Full-RNS と呼ばれている．本稿で用いる CKKS 方式についても，Full-RNS を適用した実装 [9] を使用する．PALISADE[15] の実装において，通常の状態の暗号文を COEFFICIENT format，NTT を適用した状態の暗号文を EVALUATION format と呼称する．

Full-RNS を用いた暗号方式の実装では，しばしば基底変換（base conversion）[16] と呼ばれる演算が用いられる．Full-RNS 版の CKKS においては，近似基底変換（approximate base conversion）[9] と呼ばれる演算が定義されている．これは，RNS で表現された値を別の基底の RNS 表現へ変換する演算である．$\mathcal{C} = \{q_0, ..., q_{l-1}\}, \mathcal{B} = \{p_0, ..., p_{k-1}\}$ を，それぞれ $Q = \prod_{i=0}^{l-1} q_i, P = \prod_{j=0}^{k-1} p_j$ を満たす RNS の基底とする．$x \in \mathbb{Z}_Q$ としたとき，$\mathcal{C}$ を基底とする RNS 表現 $x_i = [x]_{q_i}$ から，$\mathcal{B}$ を基底とする RNS 表現 $x_j = [x]_{p_j}$ への近似基底変換 $\mathsf{Conv}_{\mathcal{C}\to\mathcal{B}}$ は以下のように定義される．ただし，$e \in \mathbb{Z}, e \le l/2$ としたとき，$x_j = [x + Q \cdot e]_{p_j}$ を満たす．

$$\mathsf{Conv}_{\mathcal{C}\to\mathcal{B}} = \left(\sum_{i=0}^{l-1} [x_i \cdot \frac{q_i}{Q}]_{q_i} \cdot \frac{Q}{q_i} \mod p \right)_{p\in\mathcal{B}} \tag{2.2}$$

2.2.2 パッキング

CKKS 方式をはじめとした RLWE ベースの方式では，複数の平文を一つの平文としてみなすパッキングと呼ばれる手法が用いられる．パッキングを行うことによって，SIMD の要領で複数の値に対して並列で演算を適用することができる．複数の値を一つにまとめる演算を Encode と呼ぶ．また，その逆演算を Decode と呼ぶ．

2.2.3 CKKS 方式

本研究で扱う CKKS 方式は，任意回数の準同型加算と指定した回数の準同型乗算が実行可能な Leveled 準同型暗号と呼ばれる部類に属している．CKKS 方式において，暗号文に対して実行可能な準同型乗算の回数は multiplicative depth と呼ばれるパラメータ L によって決定される．後述の Rescale を暗号文に対して適用すると，暗号文の「レベル」と呼ばれるパラメータが 1 減少する．法が Q_l である暗号文のレベルは l である．レベルの初期値は L であり，レベルが 0 となると Rescale はそれ以上実行できない．

CKKS 方式では，固定小数点を整数係数多項式として表現するため，スケーリングを行っている．計算の精度は scaling factor と呼ばれるパラメータ Δ により決定される．例えば，$\Delta = 2^{16}$ としたとき，2.4 は $2.4 \cdot 2^{16} \approx 157286$ と表現できる．準同型乗算の出力は，入力された暗号文のスケール値の積となる．例えば，スケール値が 2^{16} の 2 つの暗号文を入力とする準同型乗算は，スケール値が 2^{32} である暗号文を出力する．準同型乗算を繰り返し適用すると，2^{64}，2^{128} とスケール値は指数的に増加する．表現可能な桁数には限りがあるため，スケール値の増加によって整数部の表現に必要な桁数が不足してしまう．そこで，Rescale によりスケール値を削減することにより，準同型乗算によるスケール値の指数的な増加を緩和することができる．厳密には，Rescale の適用有無は，スケール値増加の度合いによって準同型暗号の使用者が自由に決定できる．ただし，多くの場合で準同型乗算を行なった後には Rescale を実行するため，事実上レベルは対象の暗号文が実行できる準同型乗算の残数を表しているといえる．

CKKS 方式の演算のうち，本稿に関係するものを以下に一部抜粋した．

- KeyGen: 公開鍵 pk，秘密鍵 sk，後述の準同型乗算 HMul で使用する relinearize key rk，Rotate で使用する rotate key といった鍵を生成する．これらの鍵は後述の計算で使用する．
- SwitchKeyGen: key-switching key（evaluation key）と呼ばれる鍵を生成する．詳細については次節で述べる．
- KeySwitch: 2 つの異なる秘密鍵 sk_1, sk_2 について，sk_1 で暗号化された暗号文 ct を，sk_2 で復号可能な暗号文 ct' へ変換する処理である．
- Encode($\mathbf{m}$): 複数の値 $\mathbf{m}$ を一つの平文 $ptxt$ へパッキングする演算．
- Encrypt(pk, $ptxt$): 平文 $ptxt$ を暗号化する演算．
- Decrypt(sk, ct): 暗号文 ct を復号する演算．
- Decode($ptxt$): パッキングされた平文 $ptxt$ から元の複数の値の近似値を展開する演算．
- HAdd: 準同型加算．演算子としての表記は $\oplus$ を使用する．
- HMul: 準同型乗算．演算子としての表記は $\otimes$ を使用する．
- Rescale: 暗号文のレベルを 1 消費し，HMul によって増加したスケール値を演算前の状態に近い値にする．
- Rotate: 与えられた暗号文のスロットをシフトする．詳細については次節で述べる．

2.3 Key-Switching と Rotation

本節では，trace-type function で内部的に使用されている rotation と key-switching について詳しく述べる．key-switching は，CKKS 方式においては KeySwitch という演算が担っている．これは，$s, s' \in R$ とした

とき，2 つの異なる秘密鍵 $sk = (1, s), sk' = (1, s')$ について，sk で暗号化された暗号文 ct を sk' で復号可能な暗号文 ct' へ変換する処理である．key-switching には，SwitchKeyGen という演算により生成される key-switching key（evaluation key）と呼ばれる鍵 **swk** と，暗号文の法 Q_l とは別に用意された特殊な法（special modulus）$P = \prod_{i=0}^{k-1} p_i$ が必要となる [17]．ここで，k は整数とし，p_i は互いに素な整数とする．χ を離散ガウス分布とし，$a_i \xleftarrow{u} R_{PQ_l}, e_i \leftarrow \chi$ としたとき，s から s' への key-switching を行う evaluation key は $\mathbf{swk}_{s \to s'} = \{(e - as' + PB_i s, a)\}_{i \in [d]} \in R_{PQ_l}^{d \times 2}$ で定義される．

rotation は，CKKS 方式においては Rotate という演算が担っている．厳密には，Rotate は automorphism と呼ばれる演算によって構成されている．$t \in \mathbb{Z}_{2N}^*$ としたとき，automorphism は写像 $\psi_\kappa : R \to R, a(X) \mapsto a(X^\kappa) \bmod \Phi_{2N}(X)$ として定義される．ただし，$\Phi_{2N}(X)$ は 1 の原始 2N 乗根に対する円分多項式とする．このとき，κ を automorphism index と呼ぶ．暗号文に対して automorphism を適用すると，秘密鍵が s から $\psi_\kappa(s)$ に変わり，副作用としてスロットが特定の数だけシフトされる [18]．秘密鍵が異なる暗号文は計算を行うことができないため，automorphism の適用後には key-switching を行う必要がある．本稿では automorphism と key-switching は rotation を構成する一連の演算としてみなす．

2.4 Trace-Type Function

本節では，trace-type function について述べる．trace-type function とは，rotation と準同型加算（rotate-and-sum）の組を再帰的に適用した形をとる関数を指す [5]．trace-type function の例としては，total-sums が挙げられる．

total-sums は，全スロットの総和を計算し，その結果が全てのスロットに格納されるものである．Halevi ら [19] により初めて構成され，rotate-and-sum を再帰的に適用することにより実現されている．また，total-sums の亜種として，部分和をとるものも存在する．total-sums は，しばしば秘匿 DB 検索 [20] や秘匿文書分類 [21] などの準同型暗号を用いたアプリケーション内部で使用されている．

trace-type function のアルゴリズムをアルゴリズム 2.4.4 と 2.4.5 に示す．ここでは，rotate-and-sum を M 回（$M \leq \log_2 N$）適用することを想定する．CKKS 方式において，$M = \log_2 N$ のときは total sums に，$M \leq \log_2$ のときは部分和に対応する．なお，多項式の次数を N，k は key-switching 用の特殊な法の数とし，$d = (L+1)/k$，$\alpha = (L+1)/d$，$\beta = \lceil (l+1)/\alpha \rceil$，$\hat{Q}_j = \{\prod_{i=0}^{\alpha-1} q_{j\alpha+i}\}_{j \in [d]}$，$Q'_j = Q_L/\hat{Q}_j$ とする．ただし，$d|(L+1)$ とする．また，$\mathcal{C}_l = \{q_0, q_1, ..., q_l\}$，$\mathcal{B} = \{p_0, p_1, ..., p_{k-1}\}$，$\mathcal{C}'_j = \{q_{j\alpha}, ..., q_{(j+1)\alpha-1}\}$ を，それぞれ Q_l ，P，$\hat{Q}_j$ に対応する基底とし，$\mathcal{D}_i = \{\cup_{j \in [i]} \mathcal{C}'_j\} \cup \mathcal{B}$，$\mathcal{C}''_j = \mathcal{D}_\beta \setminus \mathcal{C}'_j$ とする．アルゴリズム 2.4.5 は，$\mathcal{O}(\log N)$ の automorphism および準同型加算を含む．

アルゴリズム 2.4.1 $\mathsf{ModUp}_{\mathcal{C}'_j \to \mathcal{D}_\beta}$, Algorithm 7 in [5]

Input: $\hat{d}_j \in R_{\hat{Q}_l}$ in EVALUATION format

Output: $\tilde{d} \in R^{\beta}_{PQ_l}$ in EVALUATION format

$d' \leftarrow \hat{d}_j$

$(d'^{(0)}, ..., d'^{(\alpha-1)}) \leftarrow d'$ ▷ Parse RNS components

$(d''^{(0)}, ..., d''^{(\alpha-1)}) \leftarrow (0, 0, ..., 0)$

for $i = 0$ to $\alpha - 1$ **do**

    $d''^{(i)} \leftarrow \mathsf{invNTT}(d'^{(i)})$

end for

$(\tilde{d}^{(i)}, ..., \tilde{d}^{(l)}) \leftarrow \mathsf{Conv}_{\mathcal{C}'_j \to \mathcal{C}''_j}(d''^{(0)}, ..., d''^{(\alpha-1)})$

for $i = 0$ to l **do**

    $\tilde{d}^{(i)} \leftarrow \mathsf{NTT}(\tilde{d}^{(i)})$

end for

$\tilde{d} \leftarrow \mathsf{Align}((d''^{(0)}, ..., d''^{(\alpha-1)}), (\tilde{d}^{(0)}, ..., \tilde{d}^{(l)}))$ ▷ R_{q_i} and R_{p_i} parts are aligned accordingly, forming an element of R_{PQ_l}

return d

アルゴリズム 2.4.2 $\mathsf{RNSModDown}$, Algorithm 8 in [5]

Input: $(c^{(i)})_{i \in [l+k+1]} \in R_{PQ_l}$ in EVALUATION format

Output: $(cl^{(i)})_{i \in [l+1]} \in R_{PQ_l}$ in EVALUATION format

$c' = (c'^{(0)}, ..., c'^{(l+k)}) \leftarrow (c^{(0)}, ..., c^{(l+k)})$

for $j = 0$ to $l + k$ **do**

    $c'^{(j)} \leftarrow \mathsf{invNTT}(c'^{(j)})$

end for

$(c''^{(0)}, ..., c''^{(l)}) \leftarrow \mathsf{Conv}_{\mathcal{B} \to \mathcal{C}_l}(c'^{(0)}, ..., c'^{(k-1)})$

for $i = 0$ to l **do**

    $c'^{(i)} \leftarrow (c'^{(i+k)} - c''^{(i)}) \cdot [P^{-1}]_{q_i}$

    $c'^{(i)} \leftarrow \mathsf{NTT}(c'^{(i)})$

end for

return $(c'^{(0)}, ..., c'^{(l)})$

アルゴリズム 2.4.3 RNSDecompModUp, Algorithm 6 in [5]

Input: $c = (c^{(i)})_{i \in [l+1]} \in R_{Q_l}$ in EVALUATION format

Output: $\mathbf{d} \in R^{\beta}_{PQ_l}$ in EVALUATION format

$c' = (c'^{(0)}, ..., c'^{(l)}) \leftarrow (c^{(0)}, ..., c^{(l)})$
for $j = 0$ to $\beta - 1$ **do**
    $\hat{d}_j \leftarrow [c']_{\hat{Q}_j}$
end for
$\mathbf{d} \leftarrow []$
for $j = 0$ to $\beta - 1$ **do**
    for $i = 0$ to $\alpha - 1$ **do**
        if $j \cdot \alpha + i \leq l$ **then**
            $\hat{d}_{j^{(i)}} \leftarrow [\hat{d}_j]_{q_{j\alpha+i}} \cdot [Q'_{j^{-1}}]_{q_{j\alpha+i}}$
        end if
    end for
    $\mathbf{d}[j] \leftarrow \mathsf{ModUp}_{\mathcal{C}'_j \to \mathcal{D}_\beta}(\hat{d}_j)$
end for
return d

アルゴリズム 2.4.4 Automorphism and Key-Switching (AKS), Algorithm 1 in [5]

Input: κ is an automorphism index, $c = (c_0, c_1) \in R^2_{Q_l}$ is a ciphertext in EVALUATION format, $\mathbf{swk}_{\psi_\kappa(s) \to s} \in (R^{\beta \times 2}_{PQ_l})$ is an evaluation key in EVALUATION format.

Output: $\mathbf{c}' \in R^2_{Q_l}$ is a ciphertext in EVALUATION format

$(c'_0, c'_1) \leftarrow \mathbf{c}$
$(d_0, d_1) \leftarrow (\psi_\kappa(c'_0), \psi_\kappa(c'_1))$
$\hat{\mathbf{d}} \leftarrow \mathsf{RNSDecompModUp}(d_1)$
$\tilde{c_0} \leftarrow \psi_{k_i}(\langle \hat{\mathbf{d}}, \mathbf{swk}_{\psi_\kappa(s) \to s}[0] \rangle)$
$\tilde{c_1} \leftarrow \psi_{k_i}(\langle \hat{\mathbf{d}}, \mathbf{swk}_{\psi_\kappa(s) \to s}[1] \rangle)$
$(c''_0, c''_1) \leftarrow (\mathsf{RNSModDown}(\tilde{c_0}), \mathsf{RNSModDown}(\tilde{c_1}))$
$(c'_0, c'_1) \leftarrow (d_0, 0) \oplus (c''_0, c''_1)$
return c'

アルゴリズム 2.4.5 Homomorphic Trace-Type Function Algorithm 2 in [5]

Input: $c = (c_0, c_1)$ is a ciphertext in EVALUATION format, $\mathbf{k} = (k_0, k_1, ..., k_{|\mathbf{k}|-1})$ is an array of automorphism indices such that $|\mathbf{k}| = M (\leq \log_2 N)$

Output: c' is a ciphertext in EVALUATION format

$c' \leftarrow c$
for $i = 0$ to $M - 1$ **do**
    $c' \leftarrow c' \oplus \mathsf{AKS}(\mathbf{k}[i], c')$
end for
return c'

2.5 Intel Advanced Vector Extensions 512 (Intel AVX512)

Intel AVX512(Intel Advanced Vector Extensions 512) は，Intel 製の CPU に搭載されている SIMD 拡張命令セットの一つである．AVX2 の後継として開発され，SIMD レジスタ長が 512bit であることが特徴である．AVX512 を活用することで，64bit 整数の 8-way SIMD 演算（または 32bit 整数の 16-way SIMD 演算）が可能となる．Intel AVX512 の命令セットを利用するには，コンパイラの自動ベクトル化を活用する以外に，intrinsic 関数を用いて利用する命令とタイミングを直接指定する方法が存在する．本研究では，intrinsic 関数を用いて AVX512 を利用する．AVX512 には，以下のような intrinsic 関数が存在する．[22]

- `__m512i _mm512_mullo_epi64(__m512i x, __m512i y)`: パックされた符号なし 64bit 整数 x，y について，128bit の積 xy の下位 64bit を返却する．
- `__m512i _mm512_add_epi64(__m512i x, __m512i y)`: パックされた符号なし 64bit 整数 x，y について，$x + y$ を返却する．
- `__m512i _mm512_sub_epi64(__m512i x, __m512i y)`: パックされた符号なし 64bit 整数 x，y について，$x - y$ を返却する．

また，AVX512 を準同型暗号の高速化に応用する試みとして，Intel HEXL[8] と呼ばれるライブラリが開発されている．これは，準同型暗号で多用される剰余演算の実装を提供するライブラリであり，内部で AVX512 を使用している．本研究の実装では，AVX512 を用いた実装を補助するライブラリとして，Intel HEXL の実装を使用する．Intel HEXL は，以下のような AVX512 の補助関数を実装している．ただし，`__m512i` は 8 つの 64bit 整数がパックされた 512bit 整数型である．

- `__m512i _mm512_hexl_mullo_epi<64>(__m512i x, __m512i y)`: パックされた符号なし 64bit 整数 x，y について，128bit の積 xy の下位 64bit を返却する．内部では`_mm512_mullo_epi64` を直接呼んでいる．
- `__m512i _mm512_hexl_mulhi_epi<64>(__m512i x, __m512i y)`: パックされた符号なし 64bit 整数 x，y について，128bit の積 xy の上位 64bit を返却する．
- `__m512i _mm512_hexl_shrdi_epi64<bit_shift>(__m512i x, __m512i y)`: パックされた符号なし 64bit 整数 x，y について，x と y を結合して 128 ビットの中間結果を生成する．中間結果を `bit_shift` ビットだけ右シフトし，下位 64 ビットを返却する．
- `__m512i _mm512_hexl_small_mod_epi64<k>(__m512i x, __m512i q, __m512i* q_times_2, __mm512i* q_times_4))`: $k \in \{1, 2, 4, 8\}$ としたとき，パックされた符号なし 64bit 整数 x，q の各要素が $0 \leq x[i] \leq k \cdot q[i]$ を満たすとする．このとき，$x \mod q$ を計算し返却する．$k = 2$ のとき，$x < 2q$ であることを利用して，次のように計算が行われる．

  $$x \mod q = \min(x - q, x)$$

  `q_times_2` $= 2q$，`q_times_4` $= 4q$ は，それぞれ $k = 4$，$k = 8$ のときに必要となる入力である．$k = 4$ と $k = 8$ のときは，それぞれ前述の計算が再帰的に $\log_2 k$ 回実行される．例えば，$k = 4$ のときは，$x \mod q = \min(\min(x - 2q, x) - q, \min(x - 2q, x))$ となる．
- `__m512i _mm512_hexl_mulhi_approx_epi<64>(__m512i x, __m512i y)`: パックされた符号なし 64bit 整数 x，y について，128bit の積 xy を計算し 128bit の中間結果を得る．中間結果の上位 64bit を返却する．ただし結果はたかだか 1 の誤差を含む．

第 3 章

関連研究

本章では，関連研究を紹介する．まず，SIMD を活用した準同型暗号の高速化についての関連研究を紹介し，続いて trace-type function の高速化についての関連研究を紹介する．

3.1 SIMD を用いた準同型暗号の高速化

本節では，SIMD を用いた準同型暗号の高速化手法について述べる．準同型暗号に限らず，SSE（Streaming SIMD Extensions）や AVX2（Advanced Vector eXtensions）といった Intel 系 CPU の SIMD 拡張命令セットを用いた高速化は広く用いられている．NTL[*1]や NFLlib[23] といった数論ライブラリについても，SSE や AVX2 を使用した実装が組み込まれている．PALISADE[15][*2]や HElib[*3]，HEAAN[*4]等の準同型暗号ライブラリは NTL に，FV-NFLlib[*5]は NFLlib にそれぞれ依存しているため，間接的に SIMD 拡張命令セットによる高速化の恩恵を受けることができる．

近年は，AVX512 を活用した実装についても盛んに研究されている．Jung ら [6] は，AVX512 を用いて準同型乗算を実装し，従来実装の HEAAN と比較して 2.06 倍の計算速度向上を実現した．Boemer ら [8] は，AVX512-IFMA52（Integer Fused Multiply Add）を用いてベクトル同士の剰余演算と NTT/invNTT を実装，これらを用いて PALISADE v1.11.3 を対象に準同型乗算を実装し，従来実装の 2.59 倍の計算速度向上を実現した．

3.2 準同型暗号における Trace-Type Function

本節では，準同型暗号における trace-type function の高速化についての関連研究を紹介する．

Halevi ら [19] は，total-sums を初めて構成すると共に，rotate-and-sum を再帰的に適用する repeated doubling（rotate-and-sums とも呼ばれる）を提案した．Halevi ら [24] は，baby-step/giant-step（BS/GS）を用いて automorphism の適用回数を削減する手法，および，hoisting を用いて automorphism 自体の実行コストを削減する手法を提案した．なお，hoisting とは，同じ暗号文に対して複数行う処理を処理順序の入れ替えと事前計算によって 1 回に削減する手法を指す．Bossuat ら [25] は，Halevi らの hoisting の手法を改善し，鍵生成の際にあらかじめ事前計算を行うことで automorphism のコストをさらに削減する手法を提案した．Ishimaki ら [5] は，これらの手法に加え，loop-unrolling と lazy modulus-down を用いて，ベースラインである repeated doubling と比較して 1.32-2.12 倍の性能向上を実現した．

[*1] https://libntl.org/

[*2] 厳密にはデフォルトで NTL に依存していないが，コンパイル時にオプションを与えることで NTL を使用した実装が有効になる．

[*3] https://github.com/homenc/HElib

[*4] https://github.com/snucrypto/HEAAN

[*5] https://github.com/CryptoExperts/FV-NFLlib

3.2.1 loop-unrolling を用いた Trace-Type Function の高速化

本稿では Ishimaki らの手法 [5] をベースとするため，Ishimaki らの loop-unrolling の手法について詳しく述べる．Ishimaki らの手法を，アルゴリズム 3.2.1 と 3.2.2 に示す．以降，Ishimaki らの loop-unrolling の手法を適用した trace-type Function を unrolled trace-type function と呼称する．アルゴリズム 2.4.4 で必要なイテレーション数を $M(\leq \log_2 N)$ とし，loop-unrolling を行なった後のイテレーション数を $h(\leq M)$ とする．[5] によると，アルゴリズム 3.2.2 は $\mathcal{O}(h\sqrt[h]{N})$ の rotate-and-sum を含む．

アルゴリズム 3.2.1 HoistKS, Algorithm 4 in [5]

Input: $c = (c_0, c_1)$ is a ciphertext in EVALUATION format, $\mathbf{k} = (k_0, k_1, ..., k_{|\mathbf{k}|-1})$ is an array of automorphism indices, and $\mathbf{ek} = \{\mathbf{swk}_{s\to s(X^{(k[i])^{-1}})}\}_{i\in[|\mathbf{k}|]} \in (R_{PQ_l}^{\beta\times 2})^{|\mathbf{k}|}$ is an precomputed value from evaluation key in EVALUATION format.

Output: c' is a ciphertext in EVALUATION format

1: $(c'_0, c'_1) \leftarrow c$
2: $\hat{\mathbf{d}} \leftarrow \mathsf{RNSDecompModUp}(c'_1)$ ▷ $\hat{\mathbf{d}} \in R_{PQ_l}^{\beta}$
3: $\mathbf{v} \leftarrow []$ ▷ Empty array of R_{Q_l} of length $|\mathbf{k}|$
4: $\tilde{\mathbf{v_0}}, \tilde{\mathbf{v_1}} \leftarrow []$ ▷ Empty array of R_{PQ_l} of length $|\mathbf{k}|$
5: **for** $i = 0$ to $|\mathsf{k}| - 1$ **do**
6: $\quad \mathbf{v}[i] \leftarrow \psi_{k_i}(c'0)$
7: $\quad \tilde{\mathbf{v_0}} \leftarrow \psi_{k_i}(\langle \hat{\mathbf{d}}, \mathbf{ek}[i][0] \rangle)$
8: $\quad \tilde{\mathbf{v_1}} \leftarrow \psi_{k_i}(\langle \hat{\mathbf{d}}, \mathbf{ek}[i][1] \rangle)$
9: **end for**
10: $d_0 \leftarrow \mathbf{v}[0]$
11: $\tilde{c_0} \leftarrow \tilde{\mathbf{v_0}}[0]$
12: $\tilde{c_1} \leftarrow \tilde{\mathbf{v_1}}[0]$
13: **for** $i = 0$ to $|\mathsf{k}| - 1$ **do**
14: $\quad d_0 \leftarrow d_0 + \mathbf{v}[i]$
15: $\quad \tilde{c_0} \leftarrow \tilde{c_0} + \tilde{\mathbf{v_0}}[i]$
16: $\quad \tilde{c_1} \leftarrow \tilde{c_1} + \tilde{\mathbf{v_1}}[i]$
17: **end for**
18: $(c''_0, c''_1) \leftarrow (\mathsf{RNSModDown}(\tilde{c_0}), \mathsf{RNSModDown}(\tilde{c_1}))$
19: $(c'_0, c'_1) \leftarrow (d_0, 0) \oplus (c''_0, c''_1)$ ▷ EVALUATION format
20: **return** (c'_0, c'_1)

アルゴリズム 3.2.2 Unrolled version of Trace-Type Function, Algorithm 5 in [5]

Input: $c = (c_0, c_1)$ is a ciphertext in EVALUATION format, h is a number of iterations after unrolling, $\mathbf{K} = (\mathbf{k}_0, ..., \mathbf{k}_{h-1})$ is an array of vectors of automorphism indices per iteration, and $\mathbf{E} = (\mathbf{ek}'_0, ..., \mathbf{ek}'_{h-1})$ is an array of vectors of evaluation keys in EVALUATION format for each of the automorphisms, where $\mathbf{ek}'_i = \{\mathbf{swk}_{s \to s(X^{(k[i])^{-1}})}\}_{j \in [|\mathbf{k}_i|]} \in (R_{PQ_l}^{\beta \times 2})^{|\mathbf{k}_i|}$ for each $i \in [h]$.

Output: c' is a ciphertext in EVALUATION format

1: $c' \leftarrow c$
2: **for** $i = 0$ to $h - 1$ **do**
3: $\quad c' \leftarrow c' \oplus \mathsf{HoistKS}(c', \mathbf{k}_i, \mathbf{ek}'_i)$
4: **end for**
5: **return** c'

第 4 章

提案手法

本章では，unrolled trace-type function をベースとし，AVX512 を組み合わせた実装を提案する．AVX512 を用いた実装を行うにあたり，ライブラリとして Intel HEXL を使用した．さらに，新たにベクトル同士の融合積和演算（FMA:fused multiply-add）を Intel HEXL へ追加し，Trace-Type Function へ適用した．

4.1　ベクトル同士の融合積和演算

本節では，新たに Intel HEXL に追加したベクトル同士の融合積和演算（FMA:fused multiply-add）について述べる．Intel HEXL には，ベクトル対スカラーの融合積和演算が既に存在する．しかし，ベクトル同士の融合積和演算については未実装であり，別々の関数として定義されている剰余加算と剰余乗算を順次適用する必要がある．

今回新たに実装したベクトル同士の融合積和演算は，剰余加算と剰余乗算を内部的に順次適用するものである．その際，剰余乗算と剰余加算の結果出力時で合計 2 回必要な剰余演算を，関数出力時の 1 回のみに集約するよう実装を行なった．新たに実装したベクトル同士の融合積和演算の API 定義をソースコード 4.1，内部実装の擬似コードを，実際に使用した実装をソースコード 4.2 に示す．

アルゴリズム 4.1.1 EltwiseVectorVectorFMAMod, based on Algorithm 3 in [8]

Input: q is a 512-bit packed value contains $q < 2^{62}$ modulus in all 8-lane, X,Y,Z is a 512-bit packed value contains $0 < X, Y, Z < q < 2^{63}$, and barr_lo is a 512-bit packed value contains $\lfloor 2^L/q \rfloor$ across all 8-lane.

Output: X′ is a 512-bit packed value contains $0 < X' < q$.

1: prod_hi ← _mm512_hexl_mulhi_epi<64>(X,Y)
2: prod_lo ← _mm512_hexl_mullo_epi<64>(X, Y)
3: c1 ← _mm512_hexl_shrdi_epi64<L-1>(prod_lo, prod_hi)
4: c3 ← _mm512_hexl_mulhi_epi<64>(c1, barr_lo)
5: c4 ← _mm512_hexl_mullo_epi<64>(c3, q)
6: c4 ← _mm512_sub_epi64(prod_lo, c4)
7: c5 ← _mm512_add_epi64(c4, Z)
8: q_times_2 ← 2 · q
9: q_times_4 ← 4 · q
10: X′ ← _mm512_hexl_small_mod_epu64<8>(c5, q, q_times_2, q_times_4)
11: **return** X′

ソースコード 4.1　新たに実装したベクトル同士の融合積和演算 EltwiseFMAMod の API 定義

```
1 namespace intel {
2 namespace hexl {
```

```
3
4  void EltwiseFMAMod(uint64_t* result,
5      const uint64_t* arg1, // vector to multiply
6      const uint64_t* arg2, // vector to multiply
7      const uint64_t* arg3, // vector to add
8      uint64_t n, // # of elements in each vector
9      uint64_t modulus, // modulus
10     uint64_t input_mod_factor // must be 1
11 );
12
13 }  // namespace hexl
14 }  // namespace intel
```

ソースコード 4.2　新たに実装したベクトル同士の融合積和演算 EltwiseFMAMod の内部実装

```
1  void EltwiseFMAModAVX512(uint64_t* result, const uint64_t* arg1, const uint64_t*
     arg2, const uint64_t* arg3, uint64_t n, uint64_t modulus) {
2    // Port from EltwiseMultModAVX512DQInt
3    // Algorithm 2 from
4    // https://homes.esat.kuleuven.be/~fvercaut/papers/bar_mont.pdf
5    constexpr int64_t beta = -2;
6    HEXL_CHECK(beta <= -2, "beta must be <= -2 for correctness");
7    constexpr int64_t alpha = 62;  // ensures alpha - beta = 64
8    uint64_t gamma = Log2(1);
9    HEXL_UNUSED(gamma);
10   HEXL_CHECK(alpha >= gamma + 1, "alpha must be >= gamma + 1 for correctness");
11
12   const uint64_t ceil_log_mod = Log2(modulus) + 1;  // "n" from Algorithm 2
13   uint64_t prod_right_shift = ceil_log_mod + beta;
14
15   // Barrett factor "mu"
16   HEXL_CHECK(ceil_log_mod + alpha >= 64, "ceil_log_mod + alpha < 64");
17   uint64_t barr_lo = MultiplyFactor(uint64_t(1) << (ceil_log_mod + alpha - 64), 64,
       modulus).BarrettFactor();
18
19   __m512i v_barr_lo = _mm512_set1_epi64(static_cast<int64_t>(barr_lo));
20   __m512i v_modulus = _mm512_set1_epi64(static_cast<int64_t>(modulus));
21   __m512i v_twice_mod = _mm512_set1_epi64(static_cast<int64_t>(2 * modulus));
22   __m512i v_quad_mod = _mm512_set1_epi64(static_cast<int64_t>(4 * modulus));
23   const __m512i* vp_arg1 = reinterpret_cast<const __m512i*>(arg1);
24   const __m512i* vp_arg2 = reinterpret_cast<const __m512i*>(arg2);
25   __m512i* vp_result = reinterpret_cast<__m512i*>(result);
26
27   const __m512i* vp_arg3 = reinterpret_cast<const __m512i*>(arg3);
28   HEXL_LOOP_UNROLL_4
29   for (size_t i = n / 8; i > 0; --i) {
30     // ----------------
31     // <MULT>----------
32     __m512i v_op1 = _mm512_loadu_si512(vp_arg1);
33     __m512i v_op2 = _mm512_loadu_si512(vp_arg2);
34
35     v_op1 = _mm512_hexl_small_mod_epu64<1>(v_op1, v_modulus, &v_twice_mod);
36     v_op2 = _mm512_hexl_small_mod_epu64<1>(v_op2, v_modulus, &v_twice_mod);
37
```

```
38      __m512i v_prod_hi = _mm512_hexl_mulhi_epi<64>(v_op1, v_op2);
39      __m512i v_prod_lo = _mm512_hexl_mullo_epi<64>(v_op1, v_op2);
40
41      // c1 = floor(U / 2^{n + beta})
42      __m512i c1 = _mm512_hexl_shrdi_epi64(
43          v_prod_lo, v_prod_hi, static_cast<unsigned int>(prod_right_shift));
44
45      // alpha - beta == 64, so we only need high 64 bits
46      // Perform approximate computation of high bits, as described on page
47      // 7 of https://arxiv.org/pdf/2003.04510.pdf
48      __m512i q_hat = _mm512_hexl_mulhi_approx_epi<64>(c1, v_barr_lo);
49      __m512i v_result = _mm512_hexl_mullo_epi<64>(q_hat, v_modulus);
50      // Computes result in [0, 4q)
51      v_result = _mm512_sub_epi64(v_prod_lo, v_result);
52      // </MULT>---------
53      // ----------------
54
55      // ----------------
56      // <ADD>-----------
57      __m512i v_op3 = _mm512_loadu_si512(vp_arg3);
58      // Computes result in [0, 5q)
59      v_result = _mm512_add_epi64(v_result, v_op3);
60      // Reduce result to [0, q)
61      v_result = _mm512_hexl_small_mod_epu64<8>(v_result, v_modulus, &v_twice_mod, &
          v_quad_mod);
62      _mm512_storeu_si512(vp_result, v_result);
63      // </ADD>----------
64      // ----------------
65
66      ++vp_arg1;
67      ++vp_arg2;
68      ++vp_arg3;
69      ++vp_result;
70    }
71
72    HEXL_CHECK_BOUNDS(result, n, modulus, "result exceeds bound " << modulus);
73  }
```

4.2 Unrolled Trace-Type Function に対する AVX512 の適用

本節では，前述の融合積和演算を含めた AVX512 を用いた実装を，Unrolled Trace-Type Function へ適用する．具体的には，準同型暗号ライブラリに定義されているベクトル同士の剰余演算に対して Intel HEXL を適用することにより，Unrolled Trace-Type Function の実行速度向上を試みる．本研究では，準同型暗号のライブラリ実装として PALISADE v1.9.2[15] をベースとし，剰余演算の実装に対して Intel HEXL を適用した．ただし，PALISADE v1.11.5[26] にすでに実装されているコード（剰余乗算，NTT/invNTT，modulus switching）については新たに実装を行わず，v1.9.2 で動作するように改修の上，移植を行なった．加えて，新たに剰余加算，剰余減算，剰余融合積和の実装を行なった．

PALISADE は C++ の演算子オーバーロードを用いて剰余演算が定義されているため，基本的には通常の演算子（例：+，-，*）を直接用いることができる．ゆえに，PALISADE 上の剰余演算実装を AVX512 での実装で置き換えることにより，Unrolled Trace-Type Function の内部で使用している剰余演算が AVX512 に自動的に対応

する．しかし，剰余融合積和については C++ に演算子が存在しないため，別途関数を定義することによってユーザ側から利用可能とした．剰余融合積和が利用可能な箇所としては，アルゴリズム 3.2.1 の行 5 から 17 が挙げられる．

また，アルゴリズム 3.2.1 において，行 5 と 13 の 2 つの for ループについてはそれぞれ並列化を行った．

以上の実装を行なった実際のソースコードを，ソースコード 4.3 に示す．ソースコード 4.3 は，アルゴリズム 3.2.1 を実装したものである．

ソースコード 4.3　HoistKS のソースコード

```
1  Ciphertext<DCRTPoly> EvalHoistedAutomorph(
2      ConstCiphertext<DCRTPoly> ciphertext,  // original ciphertext to  rotate
3      const vector<usint>& autoids,
4      map<usint, std::pair<std::vector<DCRTPoly>, std::vector<DCRTPoly>>>&
5          inv_evks) {
6    Ciphertext<DCRTPoly> newCiphertext = ciphertext->CloneEmpty();
7
8    const shared_ptr<vector<DCRTPoly>> expandedCiphertext =
9        ciphertext->GetCryptoContext()->EvalFastRotationPrecompute(ciphertext);
10
11   // 1. Retrieve the automorphism key that corresponds to the auto index.
12   std::vector<DCRTPoly> c(2);
13
14   c[0] = ciphertext->GetElements()[0].Clone();
15   const shared_ptr<LPCryptoParametersCKKS<DCRTPoly>> cryptoParamsLWE =
16       std::dynamic_pointer_cast<LPCryptoParametersCKKS<DCRTPoly>>(
17           ciphertext->GetCryptoParameters());
18
19   const shared_ptr<typename DCRTPoly::Params> paramsQ =
20       ciphertext->GetElements()[0].GetParams();
21   const shared_ptr<typename DCRTPoly::Params> paramsP =
22       cryptoParamsLWE->GetAuxElementParams();
23   const shared_ptr<typename DCRTPoly::Params> paramsPQ =
24       (*expandedCiphertext)[0].GetParams();
25   size_t cipherTowers = paramsQ->GetParams().size();
26   size_t towersToSkip =
27       cryptoParamsLWE->GetElementParams()->GetParams().size() - cipherTowers;
28
29   size_t n = autoids.size();
30   vector<DCRTPoly> c_vec(n);
31   vector<DCRTPoly> cTilda0_vec(n);
32   vector<DCRTPoly> cTilda1_vec(n);
33 #pragma omp parallel for
34   for (size_t ii = 0; ii < n; ++ii) {
35     const usint autoIndex = autoids[ii];
36
37     // Retrieve the automorphism key that corresponds to the auto index.
38     auto evalKey = ciphertext->GetCryptoContext()
39                        ->GetEvalAutomorphismKeyMap(ciphertext->GetKeyTag())
40                        .find(autoIndex)
41                        ->second;
42
43     const std::vector<DCRTPoly>& b = inv_evks[autoIndex].first;
44     const std::vector<DCRTPoly>& a = inv_evks[autoIndex].second;
```

```
45
46      DCRTPoly ct0;
47      DCRTPoly ct1;
48
49      // Inner Prod with Evk
50      DCRTPoly cTilda0(paramsPQ, Format::EVALUATION, true);
51      DCRTPoly cTilda1(paramsPQ, Format::EVALUATION, true);
52      for (uint32_t j = 0; j < expandedCiphertext->size(); j++) {
53        for (usint i = 0; i < (*expandedCiphertext)[j].GetNumOfElements(); i++) {
54          usint idx = (i < cipherTowers) ? i : i + towersToSkip;
55          cTilda0.SetElementAtIndex(
56              i, b[j].GetElementAtIndex(idx).FMA(
57                     (*expandedCiphertext)[j].GetElementAtIndex(i),
58                     cTilda0.GetElementAtIndex(i)));
59          cTilda1.SetElementAtIndex(
60              i, a[j].GetElementAtIndex(idx).FMA(
61                     (*expandedCiphertext)[j].GetElementAtIndex(i),
62                     cTilda1.GetElementAtIndex(i)));
63        }
64      }
65      vector<usint> perm;
66      GenAutomorphTable(cTilda0.GetRingDimension(), autoIndex, perm);
67      c_vec[ii] = std::move(ciphertext->GetElements()[0].Permute(perm));
68      cTilda0_vec[ii] = std::move(cTilda0.Permute(perm));
69      cTilda1_vec[ii] = std::move(cTilda1.Permute(perm));
70    }
71
72    DCRTPoly c_sum(paramsQ, Format::EVALUATION, true);
73    DCRTPoly cTilda0_sum(paramsPQ, Format::EVALUATION, true);
74    DCRTPoly cTilda1_sum(paramsPQ, Format::EVALUATION, true);
75
76    for (size_t ii = 0; ii < n; ++ii) {
77      c_sum += c_vec[ii];
78      cTilda0_sum += cTilda0_vec[ii];
79      cTilda1_sum += cTilda1_vec[ii];
80    }
81
82    cTilda0_sum.SetFormat(Format::COEFFICIENT);
83    cTilda1_sum.SetFormat(Format::COEFFICIENT);
84
85    DCRTPoly cHat0 = cTilda0_sum.ApproxModDown(
86        paramsQ, paramsP,
87        cryptoParamsLWE->GetPInvModQTable(),        // P^{-1} mod q_i
88        cryptoParamsLWE->GetPInvModQPreconTable(),  // Barrett Const to multiply
89                                                    // with P^{-1} mod q_i
90        cryptoParamsLWE->GetPHatInvModPTable(),
91        cryptoParamsLWE->GetPHatInvModPPreconTable(),
92        cryptoParamsLWE->GetPHatModQTable(),
93        cryptoParamsLWE->GetModBarretPreconQTable());
94
95    DCRTPoly cHat1 = cTilda1_sum.ApproxModDown(
96        paramsQ, paramsP,
97        cryptoParamsLWE->GetPInvModQTable(),        // P^{-1} mod q_i
```

```
98          cryptoParamsLWE->GetPInvModQPreconTable(),  // Barrett Const to multiply
99                                                      // with P^{-1} mod q_i
100         cryptoParamsLWE->GetPHatInvModPTable(),
101         cryptoParamsLWE->GetPHatInvModPPreconTable(),
102         cryptoParamsLWE->GetPHatModQTable(),
103         cryptoParamsLWE->GetModBarretPreconQTable());
104
105   cHat0.SetFormat(Format::EVALUATION);
106   cHat1.SetFormat(Format::EVALUATION);
107
108   DCRTPoly ct0;
109   DCRTPoly ct1;
110   ct0 = c_sum + cHat0;
111   ct1 = cHat1;
112
113   newCiphertext->SetElements({ct0, ct1});
114   newCiphertext->SetDepth(ciphertext->GetDepth());
115   newCiphertext->SetLevel(ciphertext->GetLevel());
116   newCiphertext->SetScalingFactor(ciphertext->GetScalingFactor());
117
118   return newCiphertext;
119 }
```

第 5 章

評価実験と考察

本章では，提案手法の評価を行う．まず，本研究で新たに追加したベクトル同士の融合積和演算を対象に実行時間の評価を行う．次に，AVX512 を適用した trace-type function を対象に実行時間の評価を行う．

測定に用いた実験環境を表 5.1 に示す．また，本実験で使用した CPU 構成の概要図を図 5.1 示す．実験時は，numactl コマンドを用いて NUMA1 の CPU と RAM が使用されるように設定した．なお，ベンチマークの安定性を考慮し，Turbo Boost と Hyper-Threading を無効化した．さらに，動作周波数を定格周波数の 3.60GHz へ固定した．

実験に用いるプログラムは，全て C++17 で記述した．コンパイルオプションとして，`-march=native -O3 -fopenmp -Wall -NDEBUG` を指定した．なお，並列化ライブラリとして OpenMP[*1]4.5 を使用した．

本研究では，準同型暗号のライブラリ実装として，PALISADE v1.9.2[15] を使用する．実験では，PALISADE のネイティブ実装をベースとし，剰余演算の実装を AVX512 を利用するよう修正を行なったものを使用した．このパッチには，PALISADE v1.11.5 にすでに実装されているコード（剰余乗算，NTT/invNTT，modulus switching）を移植したものが含まれている．加えて，新たに剰余加算，剰余減算，剰余融合積和の実装を行なった．なお，今回使用した PALISADE は，NTL 等の数論ライブラリには依存していない．比較のため，実験条件は Ishimaki ら [5] と同じものとする．準同型暗号の方式としては CKKS 方式を採用し，scaling factor を 2^{40} とした．また，multiplicative depth を可変とするために APPROXRESCALE[*2]モードに設定する．その他のパラメータは，標準委員会が定める白書 [27] をもとに，128bit セキュリティを満たすように設定した．また，AVX512 による実装を補助するライブラリとして，Intel HEXL v1.2.3[8] を使用した．

[*1] https://www.openmp.org
[*2] Rescale を手動で行うモードである．

表 5.1　実験環境

項目		値
CPU	型番	3rd Gen Intel(R) Xeon(R) Gold 6334
	動作周波数（定格） [GHz]	3.60
	L1 キャッシュ (データ/命令） [KiB/Core]	48 / 32
	L2 キャッシュ [KiB/Core]	1280
	L3 キャッシュ [MiB/CPU]	18
	コア数	8
	ソケット数	2
RAM	サイズ [GB]	128
	OS	Ubuntu 20.04 LTS
	g++(GCC) version	9.3.0
	OpenMP version	4.5
	PALISADE version	1.9.2
	Intel HEXL version	1.2.3
	googlebench version	f730846

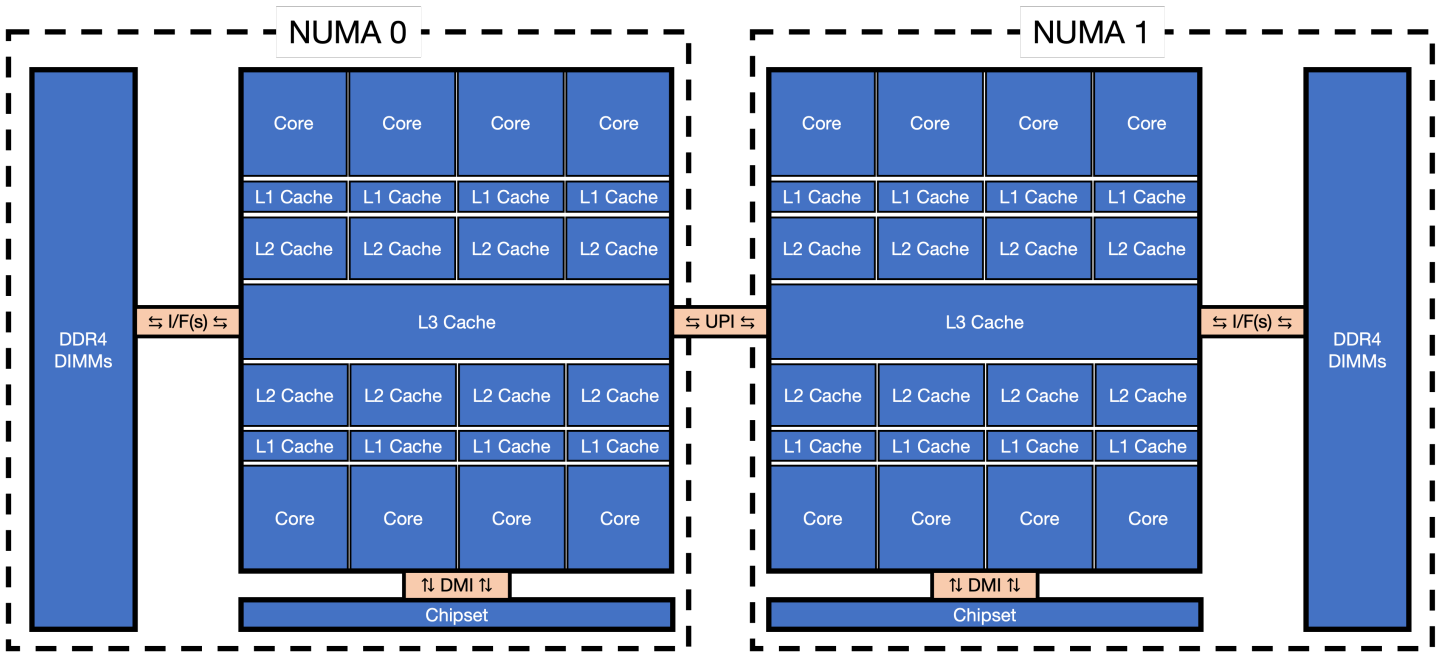

図 5.1　使用した CPU 構成の概要図

5.1　ベクトル同士の融合積和演算

本節では，新たに実装したベクトル同士の融合積和演算について実行速度の評価を行う．

5.1.1 実験方法

本実験では，乗算と加算を順番に適用したもの（以下，MultAdd）と，今回新たに実装した融合積和演算（以下，FMA）との比較を行った．MultAdd は，Intel HEXL が提供している剰余加算・剰余乗算の関数を呼び出す形で実装を行った．ゆえに，MultAdd と FMA はともに AVX512 を使用した実装である．MultAdd と FMA は，多項式の次数 N，l を自然数としたとき，要素数 $N * l$ の 64bit 整数配列 a, b, c，および要素数 l の 64bit 整数配列 m を入力にとり，$i \in [N * l], a[i] = [a[i] * b[i] + c[i]]_{m[\lfloor i/N \rfloor]}$ を計算する．$\log_2 N$ と l のパラメータの組を $(\log_2 N, l) \in \{10, 11, 12, 13, 14, 15\} \times \{1, 2, 3, ..., 14, 15\}$ と定義する．なお，実験には 60-bit modulus を使用する．それぞれのパラメータ対に対して 10,000 回実行し，その実行時間の平均を比較する．また，ベンチマークを記述する際の補助ライブラリとして，コミットハッシュが f730846 の googlebench[*3]を用いた．

ソースコード 5.1 と 5.2 は，それぞれ MultAdd と FMA のベンチマークに使用したソースコードである．googlebench は，`for (auto _ : state){}` のブロック内コードをベンチマーク対象として実行する．ここでは，Iterations に 10,000 を設定しているため，ブロック内のコードが 10,000 回実行される．

ソースコード 5.1　剰余乗算と剰余加算を順次適用（MultAdd）した際のベンチマークプログラム．ベンチマーク補助ライブラリとして googlebench を使用した．N を多項式の次数，l を整数としたとき，パラメータの組は $(\log_2 N, l) \in \{10, 11, 12, 13, 14, 15\} \times \{1, 2, 3, ..., 14, 15\}$ とした．

```
1  static void BM_EltwiseVectorVectorMultAddModAVX512(
2      benchmark::State& state) {          //  NOLINT
3    size_t input_size = state.range(0);  // N: polynomial degree
4    size_t level = state.range(1);       // l: level, # of RNS moduli
5    // Generate 60-bit l primes for RNS moduli
6    std::vector<uint64_t> moduli = GeneratePrimes(level, 60, false, input_size);
7
8    /* Generate random vectors which have N * l elements */
9    auto reserved_size = std::pow(2, std::ceil(std::log2l(input_size * level)));
10   AlignedVector64<uint64_t> input1(reserved_size), input2(reserved_size),
11       input3(reserved_size);
12   for (auto&& modulus : moduli) {
13     AlignedVector64<uint64_t> tmp1 =
14         GenerateInsecureUniformRandomValues(input_size, 0, modulus);
15     AlignedVector64<uint64_t> tmp2 =
16         GenerateInsecureUniformRandomValues(input_size, 0, modulus);
17     AlignedVector64<uint64_t> tmp3 =
18         GenerateInsecureUniformRandomValues(input_size, 0, modulus);
19     input1.insert(input1.end(), tmp1.begin(), tmp1.end());
20     input2.insert(input2.end(), tmp2.begin(), tmp2.end());
21     input3.insert(input3.end(), tmp3.begin(), tmp3.end());
22   }
23
24   for (auto _ : state) {  // BEGIN of benchmark block
25     auto input1_ptr = reinterpret_cast<uint64_t*>(input1.data());
26     auto input2_ptr = reinterpret_cast<uint64_t*>(input2.data());
27     auto input3_ptr = reinterpret_cast<uint64_t*>(input3.data());
28     for (size_t i = 0; i < level; ++i) {
29       // Perform vector-to-vector Mult
30       EltwiseMultMod(input1_ptr, input1_ptr, input2_ptr, input_size, moduli[i],
31                      1);
```

[*3] https://github.com/google/benchmark

```
      // Perform vector-to-vector Add
      EltwiseAddMod(input1_ptr, input1_ptr, input3_ptr, input_size, moduli[i]);
      input1_ptr += input_size;
      input2_ptr += input_size;
      input3_ptr += input_size;
    }
  }  // END of benchmark block
}

BENCHMARK(BM_EltwiseVectorVectorMultAddModAVX512)
    ->Unit(benchmark::kMicrosecond)
    ->Iterations(10000)                                          // # of iterations
    ->ArgsProduct({{static_cast<int64_t>(std::pow(2, 10)),   // logN = 10
                    static_cast<int64_t>(std::pow(2, 11)),   // logN = 11
                    static_cast<int64_t>(std::pow(2, 12)),   // logN = 12
                    static_cast<int64_t>(std::pow(2, 13)),   // logN = 13
                    static_cast<int64_t>(std::pow(2, 14)),   // logN = 14
                    static_cast<int64_t>(std::pow(2, 15))},  // logN = 15
                   {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
                    15}});  // level, from 1 to 15
```

ソースコード 5.2　新たに実装した融合積和演算（FMA）のベンチマークプログラム．ベンチマーク補助ライブラリとして googlebench を使用した．N を多項式の次数，l を整数としたとき，パラメータの組は $(\log_2 N, l) \in \{10, 11, 12, 13, 14, 15\} \times \{1, 2, 3, ..., 14, 15\}$ とした．

```
static void BM_EltwiseVectorVectorFMAModAVX512(
    benchmark::State& state) {          //  NOLINT
  size_t input_size = state.range(0);  // N: polynomial degree
  size_t level = state.range(1);       // l: level, # of RNS moduli
  // Generate 60-bit l primes for RNS moduli
  std::vector<uint64_t> moduli = GeneratePrimes(level, 60, false, input_size);

  /* Generate random vectors which have N * l elements */
  auto reserved_size = std::pow(2, std::ceil(std::log2l(input_size * level)));
  AlignedVector64<uint64_t> input1(reserved_size), input2(reserved_size),
      input3(reserved_size);
  for (auto&& modulus : moduli) {
    AlignedVector64<uint64_t> tmp1 =
        GenerateInsecureUniformRandomValues(input_size, 0, modulus);
    AlignedVector64<uint64_t> tmp2 =
        GenerateInsecureUniformRandomValues(input_size, 0, modulus);
    AlignedVector64<uint64_t> tmp3 =
        GenerateInsecureUniformRandomValues(input_size, 0, modulus);
    input1.insert(input1.end(), tmp1.begin(), tmp1.end());
    input2.insert(input2.end(), tmp2.begin(), tmp2.end());
    input3.insert(input3.end(), tmp3.begin(), tmp3.end());
  }

  for (auto _ : state) {  // BEGIN of benchmark block
    auto input1_ptr = reinterpret_cast<uint64_t*>(input1.data());
    auto input2_ptr = reinterpret_cast<uint64_t*>(input2.data());
    auto input3_ptr = reinterpret_cast<uint64_t*>(input3.data());
    for (size_t i = 0; i < level; ++i) {
      // Perform vector-to-vector FMA
```

```
      EltwiseFMAModAVX512<64, 1>(input1_ptr, input1_ptr, input2_ptr, input3_ptr,
                                 input_size, moduli[i]);
      input1_ptr += input_size;
      input2_ptr += input_size;
      input3_ptr += input_size;
    }
  }  // END of benchmark block
}

BENCHMARK(BM_EltwiseVectorVectorFMAModAVX512)
    ->Unit(benchmark::kMicrosecond)
    ->Iterations(10000)                                          // # of iterations
    ->ArgsProduct({{static_cast<int64_t>(std::pow(2, 10)),   // logN = 10
                    static_cast<int64_t>(std::pow(2, 11)),   // logN = 11
                    static_cast<int64_t>(std::pow(2, 12)),   // logN = 12
                    static_cast<int64_t>(std::pow(2, 13)),   // logN = 13
                    static_cast<int64_t>(std::pow(2, 14)),   // logN = 14
                    static_cast<int64_t>(std::pow(2, 15))},  // logN = 15
                   {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
                    15}});  // level, from 1 to 15
```

5.1.2 実験結果

表 5.2 は，乗算と加算を順番に適用したもの（MultAdd）と，新たに実装した融合積和演算（FMA）との平均実行時間を比較したものである．入力データサイズ（input_data_size）は，$64 * (3 * N * l + l)/8$[bytes] で算出した．なお，表は input_data_size で昇順にソート済みである．表 5.2 のデータを，横軸を入力データサイズ，縦軸を実行時間としたグラフとして図示したものが図 5.2 である．

表 5.2: 乗算と加算を順番に適用したもの（MultAdd）と融合積和演算（FMA）との平均実行時間（シングルスレッドで 10,000 回実行．60-bit modulus を使用．入力データは N を多項式の次数，l を整数としたとき，$N * l$ 要素の 64bit 整数配列 3 つ，および l 要素の 64bit 整数配列（moduli）1 つとする．入力データサイズ（input_data_size）は，$64 * (3 * N * l + l)/8$[bytes] で算出した．）

$(\log_2 N, l)$	input_data_size [bytes]	MultAdd [us]	FMA [us]	speedup
(10,1)	24,584	0.81	0.87	0.927×
(11,1)	49,160	1.53	1.72	0.889×
(10,2)	49,168	1.57	1.75	0.893×
(10,3)	73,752	2.51	2.72	0.922×
(12,1)	98,312	3.38	3.54	0.955×
(11,2)	98,320	3.30	3.57	0.923×
(10,4)	98,336	3.36	3.63	0.927×

Continued on next page

Continued from previous page

$(\log_2 N, l)$	input_data_size [bytes]	MultAdd [us]	FMA [us]	speedup
(10,5)	122,920	4.20	4.53	0.928×
(11,3)	147,480	4.97	5.37	0.926×
(10,6)	147,504	5.05	5.45	0.926×
(10,7)	172,088	5.90	6.35	0.930×
(13,1)	196,616	6.72	7.04	0.955×
(12,2)	196,624	6.80	7.07	0.961×
(11,4)	196,640	6.63	7.16	0.925×
(10,8)	196,672	6.72	7.25	0.928×
(10,9)	221,256	7.58	8.16	0.928×
(11,5)	245,800	8.28	8.96	0.924×
(10,10)	245,840	8.41	9.09	0.926×
(10,11)	270,424	9.26	9.98	0.928×
(12,3)	294,936	10.20	10.66	0.956×
(11,6)	294,960	9.93	10.77	0.922×
(10,12)	295,008	10.10	10.89	0.927×
(10,13)	319,592	10.93	11.80	0.927×
(11,7)	344,120	11.60	12.56	0.924×
(10,14)	344,176	11.79	12.71	0.928×
(10,15)	368,760	12.63	13.64	0.926×
(14,1)	393,224	13.45	14.46	0.930×
(13,2)	393,232	13.46	14.11	0.954×
(12,4)	393,248	13.58	14.17	0.959×
(11,8)	393,280	13.24	14.34	0.924×
(11,9)	442,440	14.88	16.13	0.923×
(12,5)	491,560	16.95	17.71	0.957×
(11,10)	491,600	16.61	17.91	0.927×
(11,11)	540,760	18.25	19.72	0.925×
(13,3)	589,848	20.22	21.14	0.957×
(12,6)	589,872	20.36	21.24	0.959×
(11,12)	589,920	19.92	21.50	0.927×
(11,13)	639,080	21.56	23.29	0.926×
(12,7)	688,184	23.75	24.79	0.958×

Continued on next page

Continued from previous page

$(\log_2 N, l)$	input_data_size [bytes]	MultAdd [us]	FMA [us]	speedup
(11,14)	688,240	23.24	25.02	0.929×
(11,15)	737,400	24.90	26.88	0.926×
(15,1)	786,440	26.70	29.91	0.893×
(14,2)	786,448	26.77	28.71	0.932×
(13,4)	786,464	26.94	28.19	0.955×
(12,8)	786,496	27.14	28.33	0.958×
(12,9)	884,808	31.05	32.28	0.962×
(13,5)	983,080	35.25	36.79	0.958×
(12,10)	983,120	34.92	36.09	0.967×
(12,11)	1,081,432	40.34	39.76	1.01×
(14,3)	1,179,672	44.77	43.34	1.03×
(13,6)	1,179,696	43.95	43.93	1.00×
(12,12)	1,179,744	44.20	43.76	1.01×
(12,13)	1,278,056	50.08	47.81	1.05×
(13,7)	1,376,312	54.14	51.49	1.05×
(12,14)	1,376,368	53.62	52.11	1.03×
(12,15)	1,474,680	57.87	55.34	1.05×
(15,2)	1,572,880	62.15	60.41	1.03×
(14,4)	1,572,896	63.22	59.51	1.06×
(13,8)	1,572,928	63.11	58.90	1.07×
(13,9)	1,769,544	71.13	66.50	1.07×
(14,5)	1,966,120	79.82	74.84	1.07×
(13,10)	1,966,160	81.16	74.69	1.09×
(13,11)	2,162,776	91.11	81.55	1.12×
(15,3)	2,359,320	98.21	93.58	1.05×
(14,6)	2,359,344	97.69	90.31	1.08×
(13,12)	2,359,392	99.15	90.17	1.10×
(13,13)	2,556,008	108.67	98.04	1.11×
(14,7)	2,752,568	116.52	105.28	1.11×
(13,14)	2,752,624	117.46	105.69	1.11×
(13,15)	2,949,240	126.23	113.10	1.12×
(15,4)	3,145,760	134.62	120.21	1.12×

Continued on next page

Continued from previous page

$(\log_2 N, l)$	input_data_size [bytes]	MultAdd [us]	FMA [us]	speedup
(14,8)	3,145,792	134.09	120.35	1.11×
(14,9)	3,539,016	151.28	135.47	1.12×
(15,5)	3,932,200	168.28	156.92	1.07×
(14,10)	3,932,240	167.81	150.42	1.12×
(14,11)	4,325,464	184.76	165.53	1.12×
(15,6)	4,718,640	202.02	180.42	1.12×
(14,12)	4,718,688	201.47	180.54	1.12×
(14,13)	5,111,912	218.30	195.61	1.12×
(15,7)	5,505,080	236.05	210.59	1.12×
(14,14)	5,505,136	234.98	210.65	1.12×
(14,15)	5,898,360	252.28	225.85	1.12×
(15,8)	6,291,520	270.52	240.56	1.12×
(15,9)	7,077,960	305.02	283.78	1.07×
(15,10)	7,864,400	339.45	303.38	1.12×
(15,11)	8,650,840	378.02	332.47	1.14×
(15,12)	9,437,280	413.32	367.01	1.13×
(15,13)	10,223,720	455.42	399.35	1.14×
(15,14)	11,010,160	495.77	435.95	1.14×
(15,15)	11,796,600	539.31	475.72	1.13×

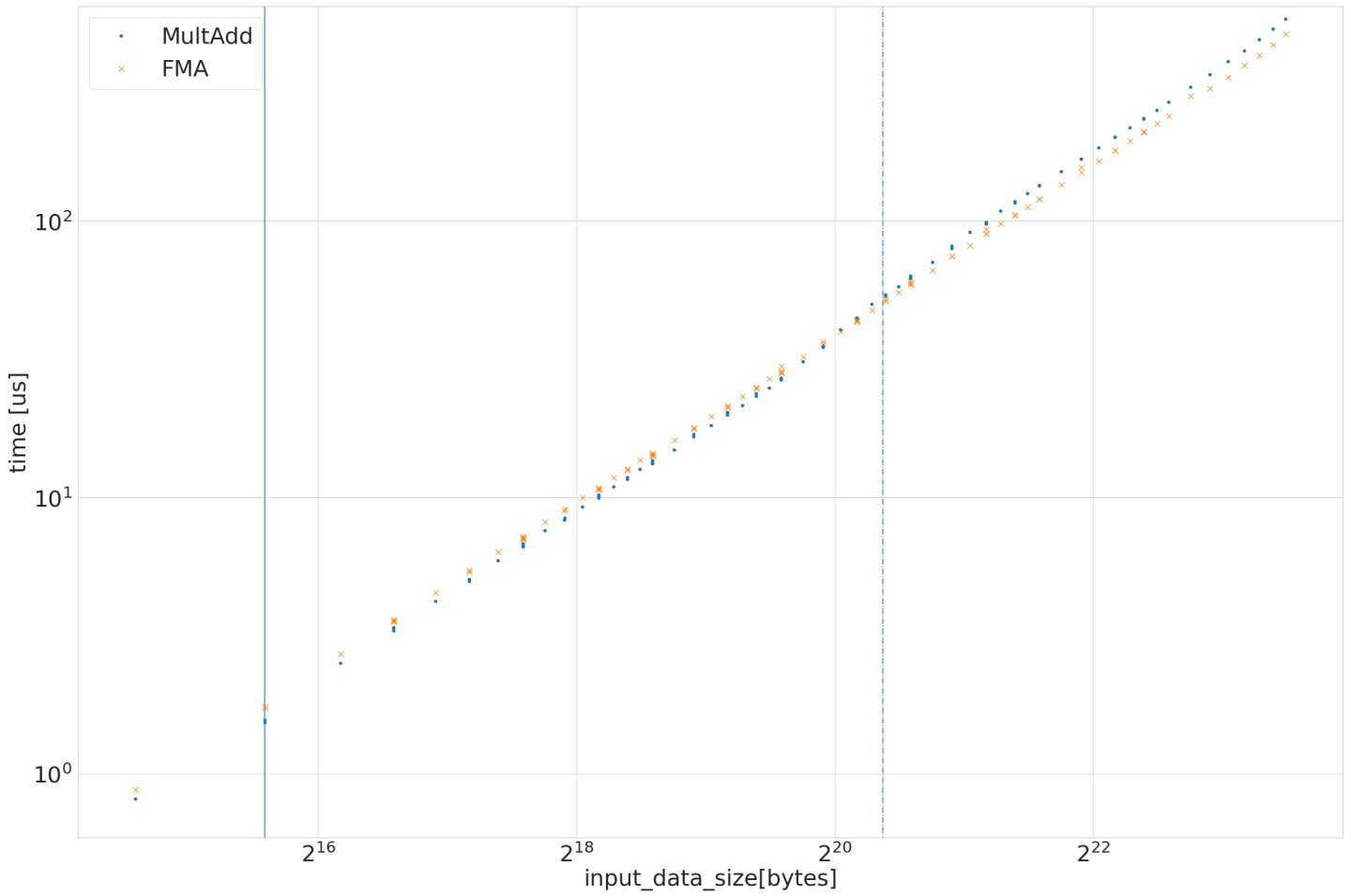

図 5.2　乗算と加算を順次適用したもの（MultAdd）と融合積和演算（FMA）の入力データサイズ（input_data_size，単位は bytes）ごとの平均実行時間（シングルスレッドで 10,000 回実行．60-bit modulus を使用．入力データは N を多項式の次数，l を整数としたとき，$N*l$ 要素の 64bit 整数配列 3 つ，および l 要素の 64bit 整数配列（moduli）1 つとする．入力データサイズは，$64*(3*N*l+l)/8$[bytes] で算出した．縦実線と縦破線は，それぞれ L1 キャッシュと L2 キャッシュのサイズを示している．）

5.1.3　考察

図 5.2 によると，入力データサイズが 2^{19}bytes 付近で実行時間が逆転している．2^{19}bytes 付近より小さいデータサイズの際は MultAdd の方が性能が高く，対して 2^{19}bytes 付近を超えると FMA の方が性能が高くなる．これは，実装とキャッシュサイズに依るものと考えられる．図 5.2 より，逆転が起きている箇所は，L2 キャッシュサイズ付近であることがわかる．FMA は，処理を一つの関数にまとめた上で，剰余演算を返却時の 1 回に集約を行なっている．しかし，FMA の実装はナイーブであるのに対し，MultAdd で使用している剰余乗算の実装は，内部処理を手動でアンローリングしている．ゆえに，FMA は MultAdd に比べて L1〜L2 までのキャッシュを有効に使うことができず，入力データサイズが L2 キャッシュに収まる範囲では速度が劣っていると考えられる．一方，入力データサイズが L2 キャッシュに乗り切らなくなった際には，別々の関数呼び出しとなっている MultAdd が速度的に不利となる．これは，それぞれの関数で SIMD レジスタに入力を読み込んでいるため，L3 キャッシュアクセスのレイテンシによる影響を受けていると考えられる．対して，FMA は一度の関数呼び出しのため，SIMD レジスタへの読み込みは 1 回で済む．実際は L2 キャッシュサイズよりも小さいサイズで実行時間の逆転が起こっている．これは，入力データサイズに考慮されていない関数内部の一時変数が影響していると考えられる．

5.2 Trace-Type Function

本節では，Trace-Type Function の評価を行う．

5.2.1 実験方法

シングルスレッドとマルチスレッドの双方で実行時間の測定を行う．マルチスレッドでは，使用可能なコアを全て活用するため，使用するスレッド数は 8 に設定した．多項式の次数 N，scaling factorΔ は，それぞれ $\log_2 N = 15$，$\Delta = 2^{40}$ に設定した．測定は各パラメータ（rotate-and-sum の適用回数 M，暗号文のレベル l，KeySwitch に使用する特殊な法の数 k，loop-unrolling 後のイテレーション数 h）の組 $(M, l, k, h) \in \{1, 7, 15\} \times \{1, 4, 9\} \times \{1, 5, 10\} \times \{1, 2, 3, ..., 14, 15\}$ ごとに 100 回ずつ行い，その平均実行時間により比較を行う．なお，初回実行時の影響を排除するため，測定の前段階として 1 回余分に実行を行なっている．

ソースコード 5.3　Unrolled Trace-Type Function

```
double EvalTraceUnrollHKMult(
    const size_t M,
    const CryptoContext<DCRTPoly> cc,
    const LPKeyPair<DCRTPoly>& keys,
    const uint32_t L,
    const size_t num_unroll,
    const size_t num_mod_reduce,
    const vector<vector<usint>>& auto_indices_parallel,
    map<usint, std::pair<std::vector<DCRTPoly>, std::vector<DCRTPoly>>>&
        inv_evks) {
  // Input
  vector<complex<double>> x = {0, 0, 0, 0, 0, 0, 0, 1};
  Plaintext ptxt = cc->MakeCKKSPackedPlaintext(x);

  cout << "Input x: " << ptxt << endl;
  auto c = cc->Encrypt(keys.publicKey, ptxt);
  vector<complex<double>> x1(cc->GetRingDimension() / 2, 1);
  auto c1 = cc->Encrypt(keys.publicKey, cc->MakeCKKSPackedPlaintext(x1));
  // We need to reduce ciphertext level from L to 1,
  // applying HMul and Rescale (L - 1) times
  for (size_t i = 0; i < num_mod_reduce; ++i) {
    c = cc->EvalMult(c, c1);
    c = cc->Rescale(c);
  }

  TimeVar t;
  TIC(t);
  // BEGIN benchmark block
  for (size_t numitr = 0; numitr < num_unroll; ++numitr)
    c += EvalHoistedAutomorph(c, auto_indices_parallel[numitr], inv_evks);
  // END benchmark block
  double timing = TOC(t);

  Plaintext result;
  cc->Decrypt(keys.secretKey, c, &result);
  result->SetLength(5);
```

```
37
38    auto m = cc->GetCyclotomicOrder();
39
40    size_t num_ans = 0;
41    int ans = 1;
42    size_t num_expected_ans = 1UL << M;
43    if (M == size_t(log2(m / 2))) {
44      ans = 2;
45      num_expected_ans = (1UL << (M - 1));
46    }
47
48    // See all the slot values
49    for (size_t i = 0; i < m / 4; ++i) {
50      if (round(result->GetCKKSPackedValue()[i].real()) == ans) num_ans += 1;
51    }
52
53    return timing;
54  }
```

5.2.2　実験結果（シングルスレッド）

表 5.3 から 5.5 にシングルスレッドでの実行結果を示す．また，測定結果を箱髭図としてプロットしたものを図 A.1 から A.27 に示す．ただし，normal と hexl は，それぞれ unrolled trace-type function（AVX512 未適用）と提案手法（AVX512 適用済）を指す．さらに，表 5.6 から 5.8 には提案手法による実行時間の改善率を示す．なお，改善率は normal の実行時間/hexl の実行時間 で算出した．図 5.3 から 5.5 は，実行時間の改善率をグラフとして図示したものである．

表 5.3　rotate-and-sum の適用回数 M=2 のときの Unrolled Trace-Type Function の平均実行時間（シングルスレッドで 100 回実行．単位は ms．結果は normal/hexl という形式で表示．ただし N/A はメモリ不足により測定不能であることを示す．loop-unrolling 後のイテレーション数 h，暗号文のレベル l，KeySwitch に使用する特殊な法の数 k）

$h\backslash k/l$	1/1	1/4	1/9	5/1	5/4	5/9	10/1	10/4	10/9
1	28.020 / 23.071	101.94 / 78.162	343.89 / 247.22	36.141 / 33.030	62.980 / 56.020	147.00 / 123.22	52.000 / 45.000	82.010 / 70.030	132.01 / 118.98
2	41.182 / 38.000	140.70 / 124.20	459.00 / 397.75	54.000 / 50.566	95.273 / 90.101	213.13 / 197.68	76.000 / 71.374	119.94 / 113.70	198.80 / 189.09

表 5.4　rotate-and-sum の適用回数 M=7 のときの Unrolled Trace-Type Function の平均実行時間（シングルスレッドで 100 回実行．単位は ms．結果は normal/hexl という形式で表示．ただし N/A はメモリ不足により測定不能であることを示す．loop-unrolling 後のイテレーション数 h，暗号文のレベル l，KeySwitch に使用する特殊な法の数 k）

$h\backslash k/l$	1/1	1/4	1/9	5/1	5/4	5/9	10/1	10/4	10/9
1	504.64 / 288.20	2,061.6 / 1,009.7	7,089.8 / 3,078.2	618.21 / 410.02	1,025.8 / 672.31	2,499.9 / 1,482.2	909.52 / 599.35	1,299.6 / 890.02	2,017.4 / 1,405.1
2	124.19 / 86.020	457.87 / 285.53	1,635.3 / 911.07	156.77 / 121.93	247.02 / 192.22	605.17 / 425.31	217.24 / 168.27	323.19 / 251.14	505.13 / 398.57
3	100.06 / 78.010	377.09 / 270.55	1,229.9 / 822.33	129.16 / 108.52	218.91 / 185.94	506.39 / 404.17	183.30 / 152.37	279.97 / 236.68	449.94 / 386.00
4	105.64 / 90.636	375.30 / 292.63	1,242.3 / 932.01	136.58 / 119.41	240.09 / 216.18	541.09 / 464.07	194.18 / 170.32	299.72 / 268.22	484.91 / 437.42
5	120.55 / 102.68	414.87 / 347.32	1,385.4 / 1,101.5	153.90 / 138.79	269.10 / 246.33	603.66 / 532.96	218.52 / 199.47	340.37 / 310.24	553.41 / 508.74
6	131.10 / 119.00	453.03 / 389.75	1,485.2 / 1,254.0	171.20 / 157.68	302.40 / 286.64	669.81 / 607.29	243.28 / 222.56	380.65 / 355.87	619.27 / 580.86
7	144.18 / 132.35	492.81 / 435.20	1,602.3 / 1,386.0	190.99 / 180.34	334.68 / 317.42	739.30 / 685.53	265.02 / 250.02	420.67 / 397.10	686.34 / 652.47

表 5.5　rotate-and-sum の適用回数 M=15 のときの Unrolled Trace-Type Function の平均実行時間（シングルスレッドで 100 回実行．単位は ms．結果は normal/hexl という形式で表示．ただし N/A はメモリ不足により測定不能であることを示す．loop-unrolling 後のイテレーション数 h，暗号文のレベル l，KeySwitch に使用する特殊な法の数 k）

$h\backslash k/l$	1/1	1/4	1/9	5/1	5/4	5/9	10/1	10/4	10/9
1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2	1,504 / 858.25	6,150.4 / 2,967.7	21,021 / 9,145.3	1,847.3 / 1,271.7	2,968.9 / 1,965.1	7,359.1 / 4,331.2	2,699.2 / 1,845.7	3,861.7 / 2,632.1	5,984.4 / 4,158.8
3	409.46 / 247.42	1,664.3 / 877.85	5,666.8 / 2,780.3	501.94 / 351.88	830.61 / 585.25	2,025.6 / 1,292.1	776.51 / 514.47	1,142.9 / 769.26	1,777.7 / 1,227.5
4	268.73 / 179.74	1,041.5 / 611.49	3,539.8 / 1,902.8	334.93 / 248.64	553.06 / 422.85	1,341 / 922.06	486.15 / 362.03	726.54 / 551.04	1,150.3 / 900.12
5	226.57 / 159.43	822.6 / 534.02	2,821.7 / 1,801.5	275.47 / 217.25	502.29 / 378.2	1,095 / 825.54	400.48 / 312.95	600.97 / 490.6	979.25 / 784.7
6	217.79 / 172.39	812.72 / 554.22	2,671.7 / 1,748.7	286.28 / 233.29	500.71 / 435.34	1,130.4 / 858.32	414.49 / 327.99	658.39 / 555.62	985.38 / 843.94
7	214.12 / 172.51	784.68 / 572.48	2,592.8 / 1,794.8	276.37 / 244.12	480.37 / 416.72	1,080 / 881.88	394.52 / 332.21	615.22 / 520.45	965.46 / 852.65
8	218.57 / 180.85	784.56 / 604.62	2,588.3 / 1,896.5	280.78 / 246.39	487.93 / 437.68	1,104.6 / 934	404.46 / 347.98	618.96 / 546.32	997.48 / 892.7
9	231.26 / 194.66	831.35 / 656.57	2,707.7 / 2,051.4	301.22 / 263.63	522.12 / 468.09	1,173.5 / 1,029.3	430.08 / 374.57	674.94 / 594.03	1,094.9 / 963.44
10	242.34 / 212.09	860.9 / 698.59	2,892.2 / 2,210.9	314.91 / 283.1	555.48 / 505.86	1,237.7 / 1,079.4	449.61 / 401.41	718.31 / 633.72	1,152.2 / 1,037.3
11	255.26 / 225.64	897.82 / 752.12	2,942.3 / 2,361.6	346.82 / 303.81	590.78 / 538.6	1,309.7 / 1,159.7	490.83 / 431.88	757.07 / 677.19	1,215.9 / 1,106.8
12	271.47 / 241.41	958.48 / 791.67	3,068.4 / 2,503.9	361.04 / 322.6	621.76 / 573.53	1,378.4 / 1,231.1	520.18 / 454.15	799.67 / 720.88	1,297.8 / 1,177.5
13	283.26 / 259.01	987.33 / 839.52	3,190.3 / 2,658.9	373.03 / 345.73	654.09 / 610	1,463.7 / 1,312.5	545.2 / 489.42	832.38 / 763.61	1,335.1 / 1,249.5
14	294.72 / 267.89	1,034.8 / 893.77	3,319.5 / 2,815.7	391.09 / 359.2	685.44 / 648.49	1,511.9 / 1,381.4	550.32 / 506.85	877.15 / 815.87	1,408.7 / 1,350.5
15	308.51 / 283.64	1,061.2 / 940.05	3,432.1 / 2,964.4	402.51 / 383.66	721.6 / 679.81	1,579.9 / 1,459.7	574.33 / 538.67	899.97 / 854.62	1,477.1 / 1,401.4

表 5.6　rotate-and-sum の適用回数 M=2 のときの Unrolled Trace-Type Function 実行時間の改善倍率（シングルスレッドで 100 回実行．(改善倍率) = (normal の実行時間)/(hexl の実行時間) で算出．ただし N/A はメモリ不足により測定不能であることを示す．loop-unrolling 後のイテレーション数 h，暗号文のレベル l，KeySwitch に使用する特殊な法の数 k）

$h\backslash k/l$	1/1	1/4	1/9	5/1	5/4	5/9	10/1	10/4	10/9
1	1.21	1.30	1.39	1.09	1.12	1.19	1.16	1.17	1.11
2	1.08	1.13	1.15	1.07	1.06	1.08	1.06	1.05	1.05

表 5.7　rotate-and-sum の適用回数 M=7 のときの Unrolled Trace-Type Function 実行時間の改善倍率（シングルスレッドで 100 回実行．(改善倍率) = (normal の実行時間)/(hexl の実行時間) で算出．ただし N/A はメモリ不足により測定不能であることを示す．loop-unrolling 後のイテレーション数 h，暗号文のレベル l，KeySwitch に使用する特殊な法の数 k）

$h\backslash k/l$	1/1	1/4	1/9	5/1	5/4	5/9	10/1	10/4	10/9
1	1.75	2.04	2.30	1.51	1.53	1.69	1.52	1.46	1.44
2	1.44	1.60	1.79	1.29	1.29	1.42	1.29	1.29	1.27
3	1.28	1.39	1.50	1.19	1.18	1.25	1.20	1.18	1.17
4	1.17	1.28	1.33	1.14	1.11	1.17	1.14	1.12	1.11
5	1.17	1.19	1.26	1.11	1.09	1.13	1.10	1.10	1.09
6	1.10	1.16	1.18	1.09	1.06	1.10	1.09	1.07	1.07
7	1.09	1.13	1.16	1.06	1.05	1.08	1.06	1.06	1.05

表 5.8　rotate-and-sum の適用回数 M=15 のときの Unrolled Trace-Type Function 実行時間の改善倍率（シングルスレッドで 100 回実行．(改善倍率) = (normal の実行時間)/(hexl の実行時間) で算出．ただし N/A はメモリ不足により測定不能であることを示す．loop-unrolling 後のイテレーション数 h，暗号文のレベル l，KeySwitch に使用する特殊な法の数 k）

$h\backslash k/l$	1/1	1/4	1/9	5/1	5/4	5/9	10/1	10/4	10/9
1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2	1.75	2.07	2.3	1.45	1.51	1.7	1.46	1.47	1.44
3	1.65	1.9	2.04	1.43	1.42	1.57	1.51	1.49	1.45
4	1.5	1.7	1.86	1.35	1.31	1.45	1.34	1.32	1.28
5	1.42	1.54	1.57	1.27	1.33	1.33	1.28	1.22	1.25
6	1.26	1.47	1.53	1.23	1.15	1.32	1.26	1.18	1.17
7	1.24	1.37	1.44	1.13	1.15	1.22	1.19	1.18	1.13
8	1.21	1.3	1.36	1.14	1.11	1.18	1.16	1.13	1.12
9	1.19	1.27	1.32	1.14	1.12	1.14	1.15	1.14	1.14
10	1.14	1.23	1.31	1.11	1.1	1.15	1.12	1.13	1.11
11	1.13	1.19	1.25	1.14	1.1	1.13	1.14	1.12	1.1
12	1.12	1.21	1.23	1.12	1.08	1.12	1.15	1.11	1.1
13	1.09	1.18	1.2	1.08	1.07	1.12	1.11	1.09	1.07
14	1.1	1.16	1.18	1.09	1.06	1.09	1.09	1.08	1.04
15	1.09	1.13	1.16	1.05	1.06	1.08	1.07	1.05	1.05

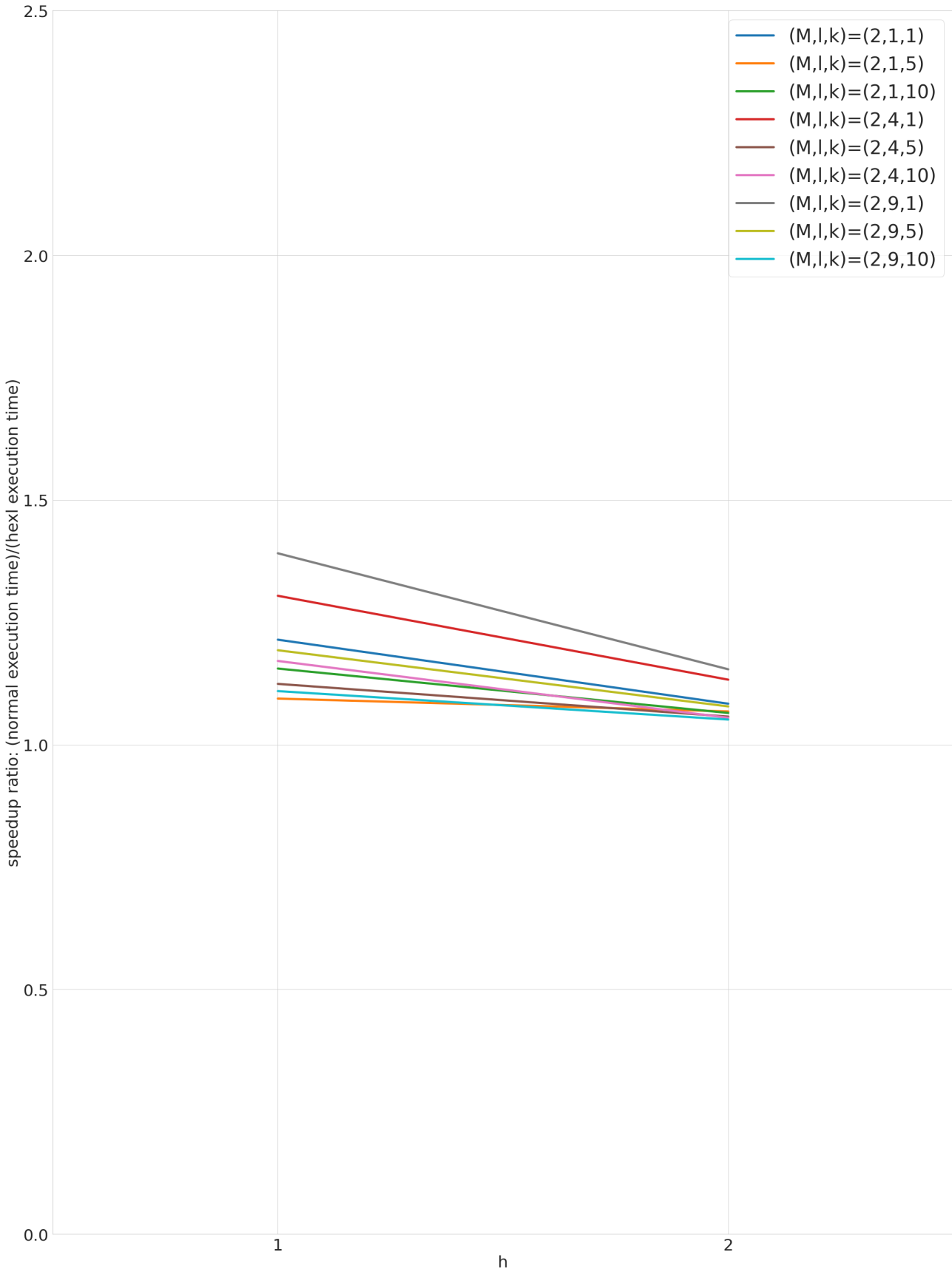

図 5.3　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間の高速化率（スレッド数 1，rotate-and-sum の実行回数 M=2）

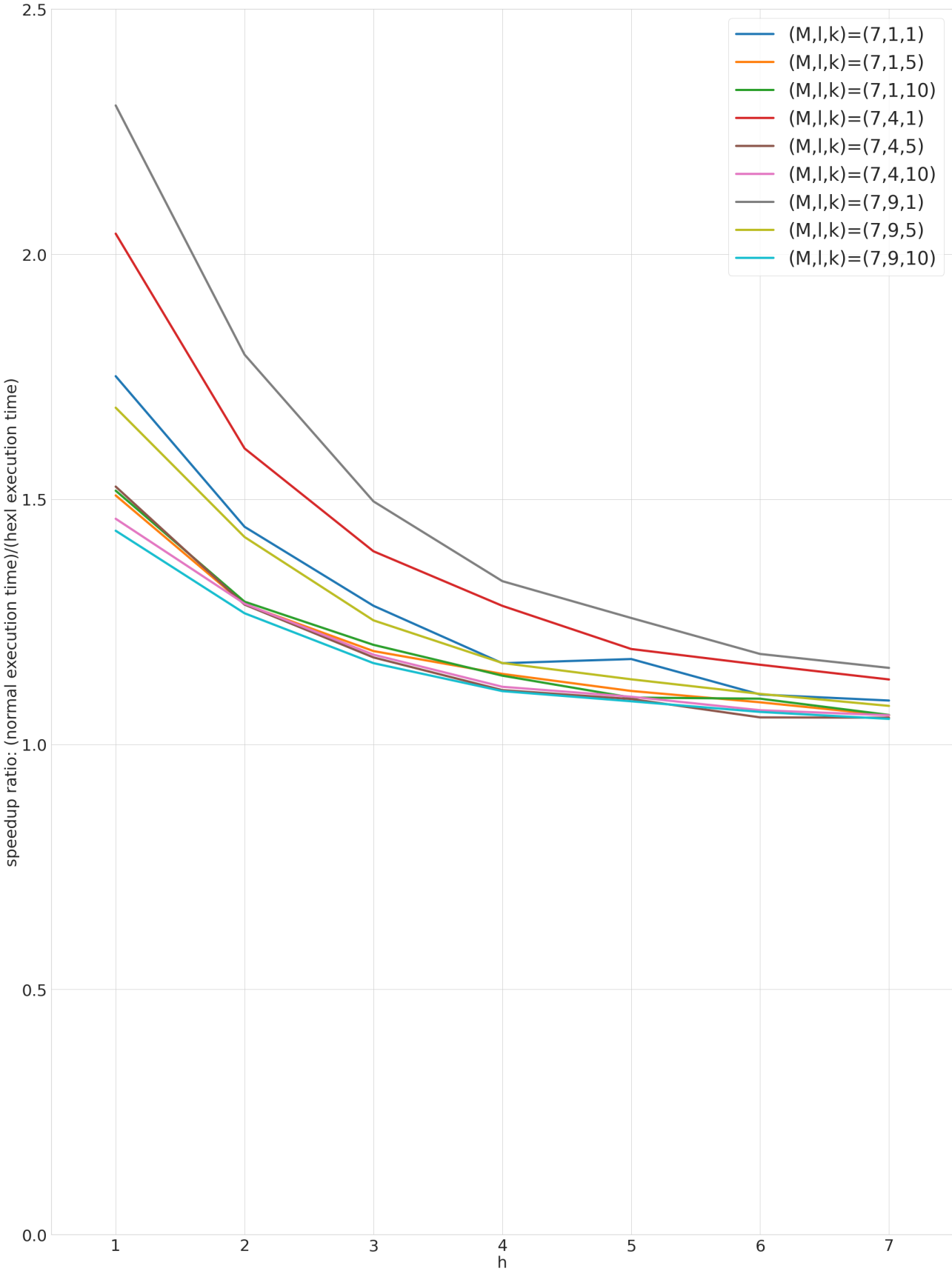

図 5.4　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間の高速化率（スレッド数 1，rotate-and-sum の実行回数 M=7）

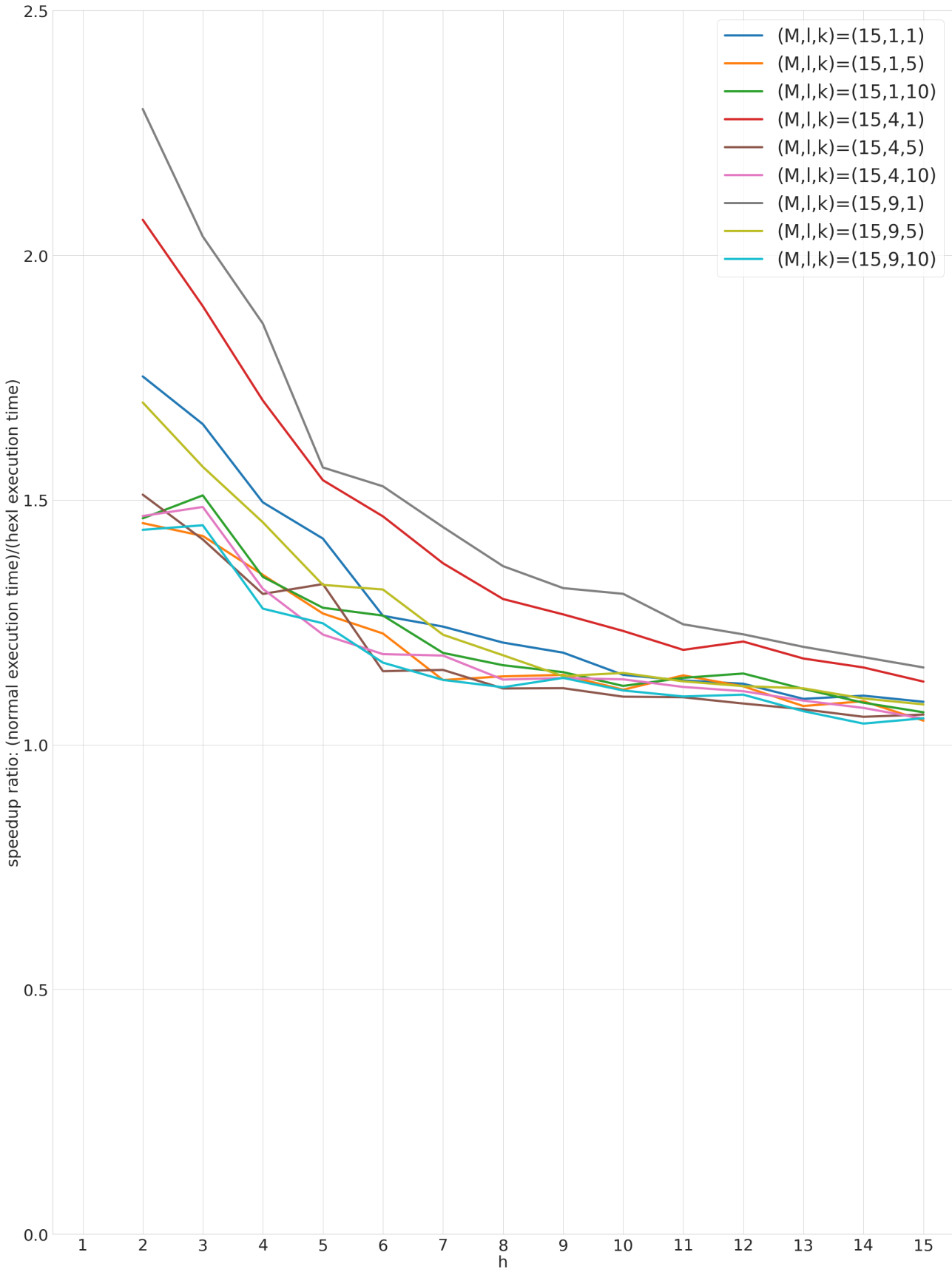

図 5.5　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間の高速化率（スレッド数 1，rotate-and-sum の実行回数 M=15）

5.2.3　実験結果（マルチスレッド）

表 5.9 から 5.11 にマルチスレッドでの実行結果を示す．また，測定結果を箱髭図としてプロットしたものを図 A.28 から A.54 に示す．ただし，normal と hexl は，それぞれ Ishimaki ら [5] の従来手法（AVX512 未適用）

と提案手法（AVX512 適用済）を指す．を指す．さらに，表 5.12 から 5.14 には提案手法による実行時間の改善率を示す．なお，改善率は normal の実行時間/hexl の実行時間 で算出している．図 5.6 から 5.8 は，実行時間の改善率をグラフとして図示したものである．

表 5.9　rotate-and-sum の適用回数 M=2 のときの Unrolled Trace-Type Function の平均実行時間（スレッド数 8 で 100 回実行．単位は ms．結果は normal/hexl という形式で表示．ただし N/A はメモリ不足により測定不能であることを示す．loop-unrolling 後のイテレーション数 h，暗号文のレベル l，KeySwitch に使用する特殊な法の数 k）

$h\backslash k/l$	1/1	1/4	1/9	5/1	5/4	5/9	10/1	10/4	10/9
1	12.010 / 10.121	36.949 / 28.101	133.86 / 102.88	13.434 / 12.202	22.636 / 20.010	57.020 / 48.899	18.889 / 16.061	27.010 / 24.010	51.030 / 46.030
2	23.051 / 19.828	71.313 / 54.869	255.81 / 193.82	24.475 / 21.030	40.960 / 36.020	107.01 / 90.394	33.434 / 28.646	48.040 / 41.636	94.121 / 84.505

表 5.10　rotate-and-sum の適用回数 M=7 のときの Unrolled Trace-Type Function の平均実行時間（スレッド数 8 で 100 回実行．単位は ms．結果は normal/hexl という形式で表示．ただし N/A はメモリ不足により測定不能であることを示す．loop-unrolling 後のイテレーション数 h，暗号文のレベル l，KeySwitch に使用する特殊な法の数 k）

$h\backslash k/l$	1/1	1/4	1/9	5/1	5/4	5/9	10/1	10/4	10/9
1	119.62 / 87.242	370.12 / 245.42	1,153.1 / 728.81	167.49 / 133.96	277.55 / 217.79	558.03 / 423.75	234.32 / 194.94	345.29 / 284.41	546.69 / 452.26
2	34.283 / 28.364	107.98 / 78.121	350.8 / 264.22	41.970 / 36.828	69.030 / 59.202	160.94 / 136.05	59.101 / 50.323	84.131 / 72.081	150.52 / 132.56
3	38.848 / 33.273	114.46 / 90.020	402.05 / 310.03	43.737 / 37.828	73.131 / 64.081	179.04 / 154.19	60.222 / 51.990	86.899 / 75.535	163.44 / 147.26
4	49.051 / 42.525	147.89 / 113.88	518.35 / 398.19	53.889 / 46.253	90.040 / 78.374	226.05 / 192.10	73.808 / 63.303	106.03 / 91.606	204.06 / 183.44
5	62.222 / 53.111	182.46 / 141.01	643.90 / 491.03	64.212 / 55.071	108.05 / 95.071	273.36 / 234.06	87.818 / 76.000	127.05 / 109.85	245.46 / 220.42
6	70.778 / 61.919	215.80 / 167.47	764.23 / 576.61	75.394 / 65.323	127.20 / 111.57	329.23 / 274.92	102.58 / 87.566	148.42 / 128.57	288.25 / 258.78
7	81.535 / 73.596	251.82 / 195.56	881.82 / 671.3	86.152 / 74.192	144.64 / 127.38	371.62 / 316.15	117.24 / 100.21	171.46 / 147.14	331.65 / 299.16

表 5.11　rotate-and-sum の適用回数 M=15 のときの Unrolled Trace-Type Function の平均実行時間（スレッド数 8 で 100 回実行．単位は ms．結果は normal/hexl という形式で表示．ただし N/A はメモリ不足により測定不能であることを示す．loop-unrolling 後のイテレーション数 h，暗号文のレベル l，KeySwitch に使用する特殊な法の数 k）

$h\backslash k/l$	1/1	1/4	1/9	5/1	5/4	5/9	10/1	10/4	10/9
1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2	317.67 / 231.62	1,101.1 / 739.01	3,360.6 / 2,067.9	488.83 / 391.84	810.32 / 659.09	1,631.8 / 1,209	714.75 / 570.35	1,017.8 / 869.48	1,611.9 / 1,330.1
3	98.182 / 75.192	313.04 / 216.49	1,010.3 / 669.33	133.76 / 108.1	217.4 / 176.77	515.02 / 398.14	190.05 / 162.13	292.64 / 238.1	470.79 / 415.69
4	75.273 / 60.535	226.69 / 170.01	752.81 / 556.32	94.626 / 78.515	154.16 / 131.05	365.42 / 295.66	131.77 / 112.28	188.47 / 160.16	334.87 / 297.11
5	69.687 / 59.889	203.32 / 161.81	683.15 / 540.59	81.222 / 70.879	135.75 / 121.31	319.73 / 278.06	113.39 / 98.465	162.77 / 143	305.25 / 269.26
6	79.97 / 67.778	235.33 / 184.67	798.05 / 625.08	90.586 / 80.99	153.15 / 132.58	363.09 / 319.16	124.77 / 108.56	181.77 / 167.43	341.24 / 318.64
7	88.758 / 81.182	265.47 / 210	916.6 / 707.69	99.97 / 85.909	165.06 / 145.8	406.83 / 349.26	137.4 / 117.7	199.59 / 173.06	376.02 / 335.09
8	99.768 / 86.333	298.95 / 231.71	1,039.5 / 807.19	111.28 / 94.465	182.58 / 160.44	454.01 / 389.36	150.52 / 128.49	216.51 / 190.14	411.03 / 386.48
9	117.7 / 97.03	355.39 / 262.76	1,159.1 / 879.13	120.44 / 103.45	200.49 / 176.01	501.94 / 430.21	164.63 / 140.83	236.64 / 215.35	454.18 / 411.95
10	121.55 / 106.47	368.13 / 286.18	1,284.2 / 976.27	130.36 / 112.94	218.6 / 191.25	553.22 / 471.34	179.67 / 159.71	263.36 / 231.86	495.82 / 459.8
11	136.17 / 114.71	402.32 / 310.93	1,398.5 / 1,073	141.48 / 128.17	237.46 / 208.23	599.27 / 510.73	193.46 / 176.61	281.14 / 252.92	539.24 / 501.21
12	143.3 / 125.57	437.86 / 338.98	1,512.1 / 1,158.5	154.07 / 133.43	256.29 / 229.49	651.99 / 555.53	210.06 / 186.91	308.04 / 273.95	595.32 / 531.18
13	155.91 / 135.38	472.29 / 362.24	1,655.4 / 1,250.2	163.4 / 143.81	281.14 / 242.22	721.17 / 596.97	224.27 / 205.69	336.56 / 293.43	635.51 / 577.76
14	164.34 / 143.33	512.4 / 389.4	1,762.2 / 1,342.5	175.97 / 152.79	294.39 / 256.45	749.54 / 640.28	243.17 / 212.78	352.68 / 299.43	671.49 / 631.54
15	175.78 / 150.62	535.99 / 420.16	1,901.7 / 1,423	185.22 / 164.62	309.86 / 273.1	796.4 / 682.49	254.36 / 234.85	365.67 / 339.69	718.39 / 658.13

表 5.12　rotate-and-sum の適用回数 M=2 のときの Unrolled Trace-Type Function 実行時間の改善倍率（スレッド数 8 で 100 回実行．(改善倍率) = (normal の実行時間)/(hexl の実行時間) で算出．ただし N/A はメモリ不足により測定不能であることを示す．loop-unrolling 後のイテレーション数 h，暗号文のレベル l，KeySwitch に使用する特殊な法の数 k）

$h\backslash k/l$	1/1	1/4	1/9	5/1	5/4	5/9	10/1	10/4	10/9
1	1.19	1.31	1.30	1.10	1.13	1.17	1.18	1.12	1.11
2	1.16	1.30	1.32	1.16	1.14	1.18	1.17	1.15	1.11

表 5.13　rotate-and-sum の適用回数 M=7 のときの Unrolled Trace-Type Function 実行時間の改善倍率（スレッド数 8 で 100 回実行．(改善倍率) = (normal の実行時間)/(hexl の実行時間) で算出．ただし N/A はメモリ不足により測定不能であることを示す．loop-unrolling 後のイテレーション数 h，暗号文のレベル l，KeySwitch に使用する特殊な法の数 k）

$h\backslash k/l$	1/1	1/4	1/9	5/1	5/4	5/9	10/1	10/4	10/9
1	1.37	1.51	1.58	1.25	1.27	1.32	1.20	1.21	1.21
2	1.21	1.38	1.33	1.14	1.17	1.18	1.17	1.17	1.14
3	1.17	1.27	1.30	1.16	1.14	1.16	1.16	1.15	1.11
4	1.15	1.30	1.30	1.17	1.15	1.18	1.17	1.16	1.11
5	1.17	1.29	1.31	1.17	1.14	1.17	1.16	1.16	1.11
6	1.14	1.29	1.33	1.15	1.14	1.2	1.17	1.15	1.11
7	1.11	1.29	1.31	1.16	1.14	1.18	1.17	1.17	1.11

表 5.14　rotate-and-sum の適用回数 M=15 のときの Unrolled Trace-Type Function 実行時間の改善倍率（スレッド数 8 で 100 回実行．(改善倍率) = (normal の実行時間)/(hexl の実行時間) で算出．ただし N/A はメモリ不足により測定不能であることを示す．loop-unrolling 後のイテレーション数 h，暗号文のレベル l，KeySwitch に使用する特殊な法の数 k）

$h\backslash k/l$	1/1	1/4	1/9	5/1	5/4	5/9	10/1	10/4	10/9
1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2	1.37	1.49	1.63	1.25	1.23	1.35	1.25	1.17	1.21
3	1.31	1.45	1.51	1.24	1.23	1.29	1.17	1.23	1.13
4	1.24	1.33	1.35	1.21	1.18	1.24	1.17	1.18	1.13
5	1.16	1.26	1.26	1.15	1.12	1.15	1.15	1.14	1.13
6	1.18	1.27	1.28	1.12	1.16	1.14	1.15	1.09	1.07
7	1.09	1.26	1.3	1.16	1.13	1.16	1.17	1.15	1.12
8	1.16	1.29	1.29	1.18	1.14	1.17	1.17	1.14	1.06
9	1.21	1.35	1.32	1.16	1.14	1.17	1.17	1.1	1.1
10	1.14	1.29	1.32	1.15	1.14	1.17	1.12	1.14	1.08
11	1.19	1.29	1.3	1.1	1.14	1.17	1.1	1.11	1.08
12	1.14	1.29	1.31	1.15	1.12	1.17	1.12	1.12	1.12
13	1.15	1.3	1.32	1.14	1.16	1.21	1.09	1.15	1.1
14	1.15	1.32	1.31	1.15	1.15	1.17	1.14	1.18	1.06
15	1.17	1.28	1.34	1.13	1.13	1.17	1.08	1.08	1.09

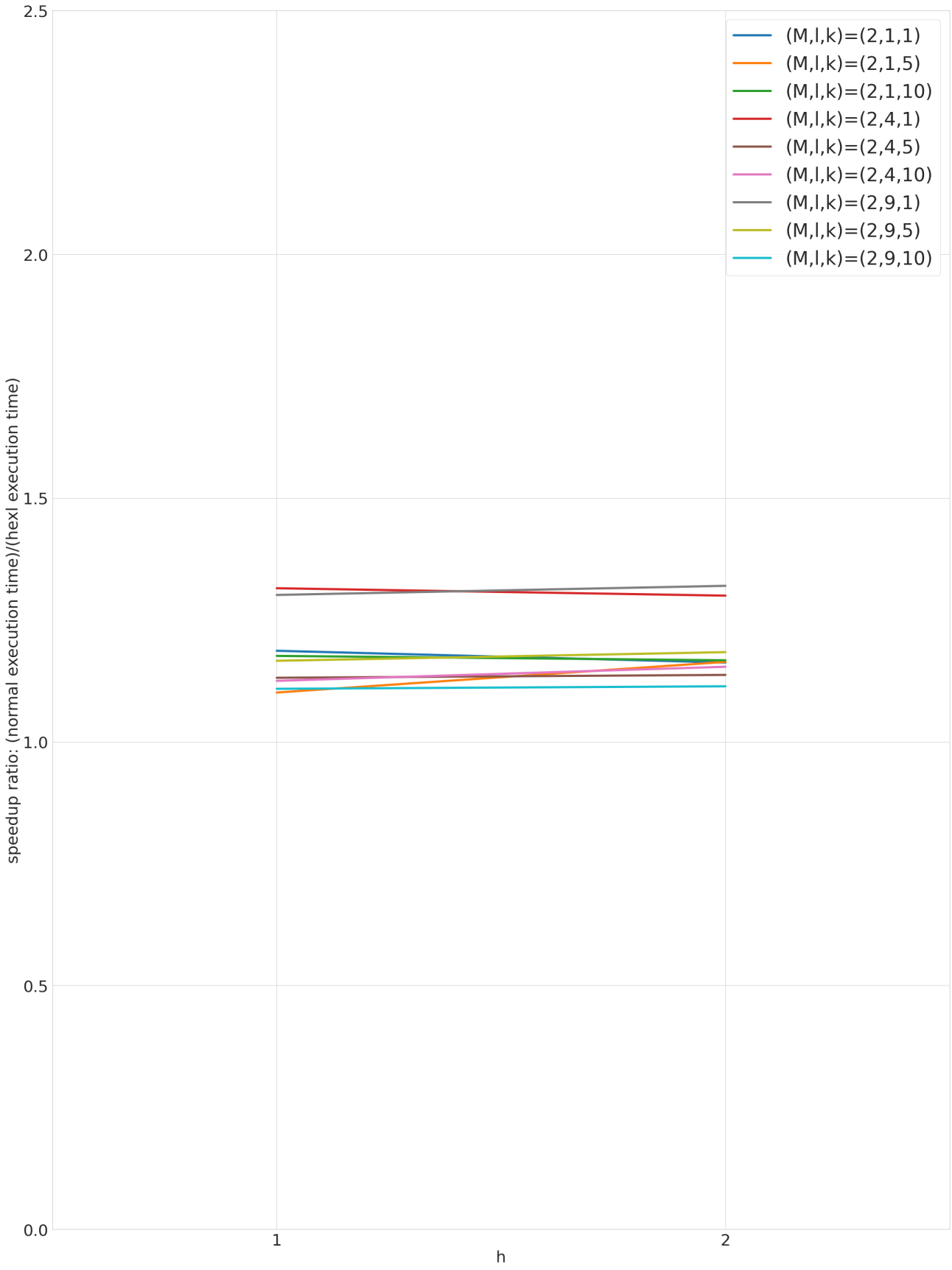

図 5.6　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間の高速化率（スレッド数 8，rotate-and-sum の実行回数 M=2）

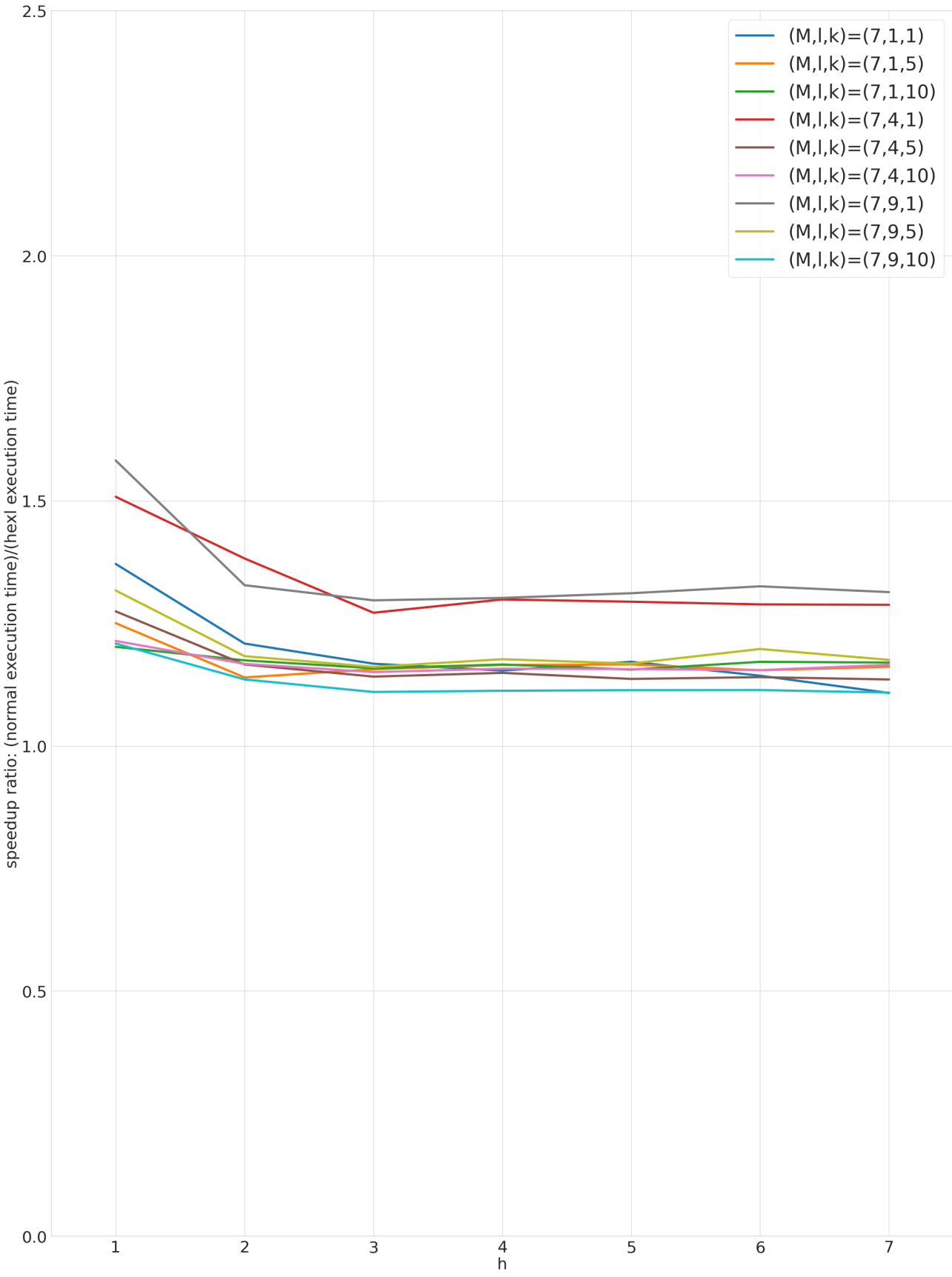

図 5.7　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間の高速化率（スレッド数 8，rotate-and-sum の実行回数 M=7）

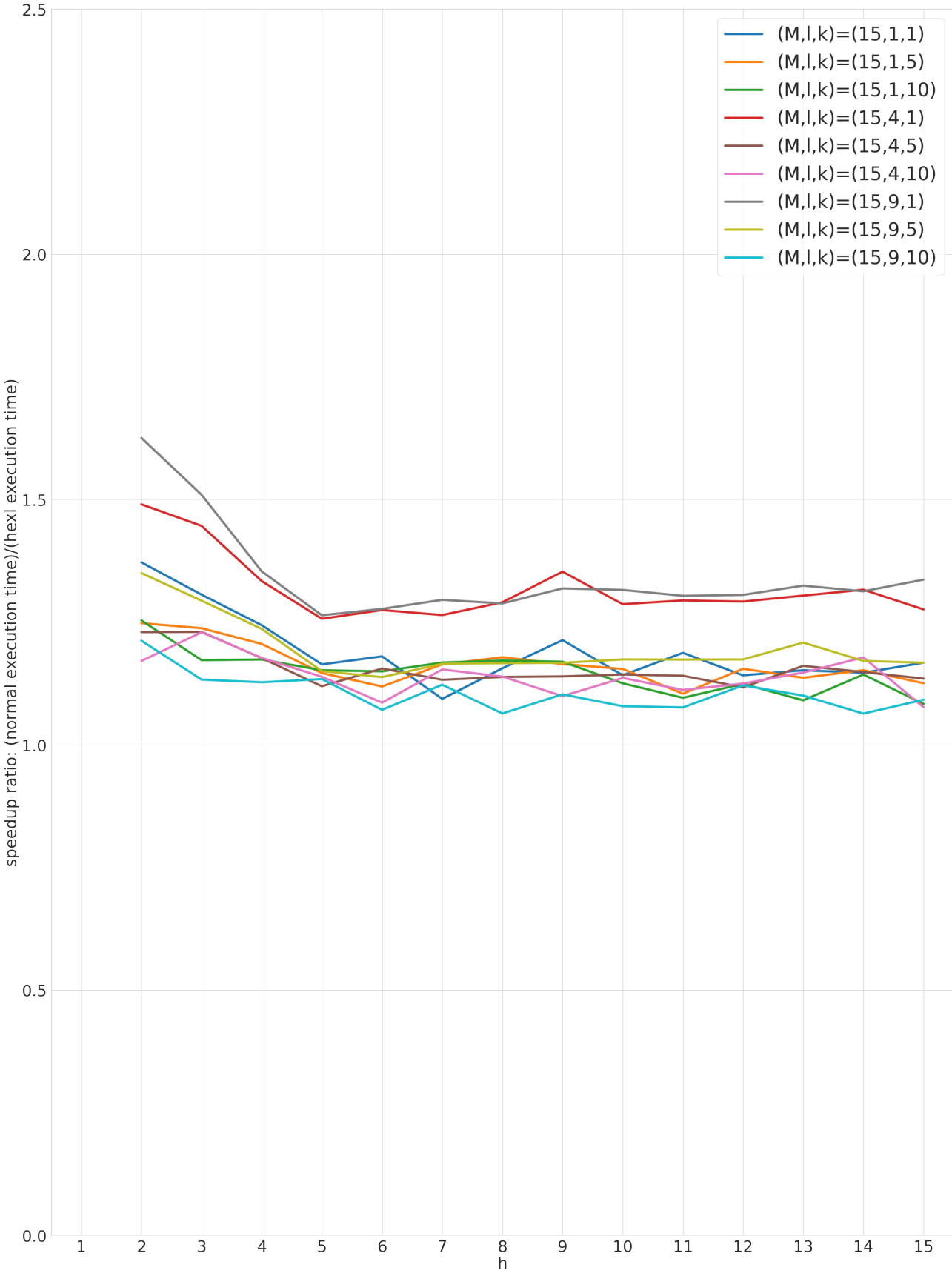

図 5.8　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間の高速化率（スレッド数 8，rotate-and-sum の実行回数 M=15）

5.2.4　考察

図 5.3 から 5.8 を見ると，M と k を固定したとき，l が大きくなるにつれて高速化率は高くなっている．これは，l が大きくなる，すなわち暗号文のデータサイズが増加することにより，SIMD によって一度に処理されるデータ

が増加することに起因するものであると考えられる．

また，図 5.3 から 5.5 について，M と l を固定したときの k，および，各パラメータ対における h に注目すると，k や h が大きくなるにつれて高速化率は下がっている．これは，SIMD によって一定の高速化が見込めるものの，全体の計算量は増加するため，SIMD で一度に処理されるデータの量に変化がないことが原因であると考えられる．ただし，一部の結果，とくに M=15 の場合については，全体の傾向と異なる挙動をしている箇所がある．これは，実行回数が 100 回に固定されていることから，図 A.19 から A.27 に示す外れ値の影響を大きく受けているものと考えられる．

マルチスレッドの結果（図 5.6 から 5.8）を確認すると，前述の特徴は軽微に現れているものの，全体として高速化率はほぼ一定，かつシングルスレッドよりも低いことがわかる．これは，マルチスレッドにおいて，各スレッドの同期コストによるものと考えられる．

第 6 章

まとめ

本稿では，trace-type function の演算に焦点を絞り，AVX512 を活用した SIMD 化による最適化を行った．実装にあたって，補助ライブラリとして Intel HEXL[8] を活用した．また，Intel HEXL には実装されていないベクトル同士の剰余積和演算を新たに実装し，Ishimaki らの unrolled trace-type function[5] へ適用した．実験の結果，AVX512 未適用の unrolled trace-type function と比較して 1.05-2.30 倍の計算速度向上を実現した．

現状の課題としては，新たに実装したベクトル同士の剰余積和演算の性能が，パラメータが小さい場合に悪化することが挙げられる．また，実際に trace-type function を活用したアプリケーションに適用した上での性能評価，および，本実験で無効化したハイパースレッディングやターボブーストを有効化した上での検証など，実運用を想定した検証を行っていく必要がある．

謝辞

本研究は，JST CREST（JPMJCR1503）の支援を受けたものである．

また，本研究を進めるにあたり，多くのご指導を頂いた山名早人教授に厚く御礼申し上げます．また，研究に関して様々な意見を下さった山名研究室の皆様に深く感謝いたします．

参考文献

[1] International Data Corporation (IDC). Data creation and replication will grow at a faster rate than installed storage capacity, according to the idc global datasphere and storagesphere forecasts. `https://www.idc.com/getdoc.jsp?containerId=prUS47560321`, 3 2021. (Accessed on 01/06/2022).

[2] Craig Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, pp. 169–178, New York, NY, USA, 2009. Association for Computing Machinery.

[3] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 10624 LNCS, pp. 409–437. Springer Verlag, 2017.

[4] N P Smart and F Vercauteren. Fully homomorphic SIMD operations. *Designs, Codes and Cryptography*, Vol. 71, No. 1, pp. 57–81, 2014.

[5] Yu Ishimaki and Hayato Yamana. Faster Homomorphic Trace-Type Function Evaluation. *IEEE Access*, Vol. 9, pp. 53061–53077, 2021.

[6] Wonkyung Jung, Eojin Lee, Sangpyo Kim, Jongmin Kim, Namhoon Kim, Keewoo Lee, Chohong Min, Jung Hee Cheon, and Jung Ho Ahn. Accelerating fully homomorphic encryption through architecture-centric analysis and optimization. *IEEE Access*, Vol. 9, pp. 98772–98789, 2021.

[7] Fabian Boemer, Rosario Cammarota, Daniel Demmler, Thomas Schneider, and Hossein Yalame. Mp2ml: A mixed-protocol machine learning framework for private inference. Cryptology ePrint Archive, Report 2020/721, 2020. `https://ia.cr/2020/721`.

[8] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe de Souza, and Vinodh Gopal. Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '21, pp. 57–62, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A Full RNS Variant of Approximate Homomorphic Encryption. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 11349 LNCS, pp. 347–368. Springer Verlag, 2019.

[10] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. `https://ia.cr/2012/144`.

[11] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012.

[12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *ACM Transactions on Computation Theory*, 2014.

[13] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 10031 LNCS, pp. 3–33. Springer Verlag, 2016.

[14] P V Ananda Mohan. *Residue Number Systems*. Springer International Publishing, 2016.

[15] PALISADE Lattice Cryptography Library (release 1.9.2). https://palisade-crypto.org/, 4 2020.

[16] Jean Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 10532 LNCS, pp. 423–442. Springer Verlag, 2017.

[17] Kyoohyung Han and Dohyeong Ki. Better bootstrapping for approximate homomorphic encryption. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 12006 LNCS, pp. 364–390. Springer, 2020.

[18] Gentry Craig, Shai Halevi, and Smart Nigel P. Fully Homomorphic Encryption with Polylog Overhead. In Pointcheval David and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pp. 465–482, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[19] Shai Halevi and Victor Shoup. Algorithms in HElib. In A. Garay, Juan and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pp. 554–571, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[20] Craig Gentry, Shai Halevi, Charanjit Jutla, and Mariana Raykova. Private Database Access with HE-over-ORAM Architecture. In Malkin Tal, Vladimir Kolesnikov, and Lewko Allison BishopPolychronakis Michalis, editors, *Applied Cryptography and Network Security*, pp. 172–191, Cham, 2015. Springer International Publishing.

[21] Ahmad Al Badawi, Louie Hoang, Chan Fook Mun, Kim Laine, and Khin Mi Mi Aung. PrivFT: Private and Fast Text Classification With Homomorphic Encryption. *IEEE Access*, Vol. 8, pp. 226544–226556, 12 2020.

[22] Intel Corporation. Intel ® intrinsics guide. `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`. (Accessed on 01/22/2022).

[23] Aguilar-Melchor Carlos, Jorisand Guelton Serge Barrier, Guinet Adrien, Killijian Marc-Olivier, and Lepoint Tancrède. NFLlib: NTT-Based Fast Lattice Library. In Kazue Sako, editor, *Topics in Cryptology - CT-RSA 2016*, pp. 341–356, Cham, 2016. Springer International Publishing.

[24] Shai Halevi and Victor Shoup. Faster Homomorphic Linear Transformations in HElib. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pp. 93–120, Cham, 2018. Springer International Publishing.

[25] Bossuat Jean-Philippe, Christian Mouchet, Troncoso-Pastoriza Juan, and Hubaux Jean-Pierre. Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys. In Canteaut Anne and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, pp. 587–617, Cham, 2021. Springer International Publishing.

[26] PALISADE Lattice Cryptography Library (release 1.11.5). `https://palisade-crypto.org/`, 2021.

[27] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption standard. Cryptology ePrint Archive, Report 2019/939, 2019. `https://ia.cr/2019/939`.

付録 A

loop-unrolling 後のイテレーション数 h を変化させた際の実行時間

以下の図は，第 5.2 節で行なった実験において，loop-unrolling 後のイテレーション数 h を変化させた際の実行時間を箱髭図としてプロットしたものである．

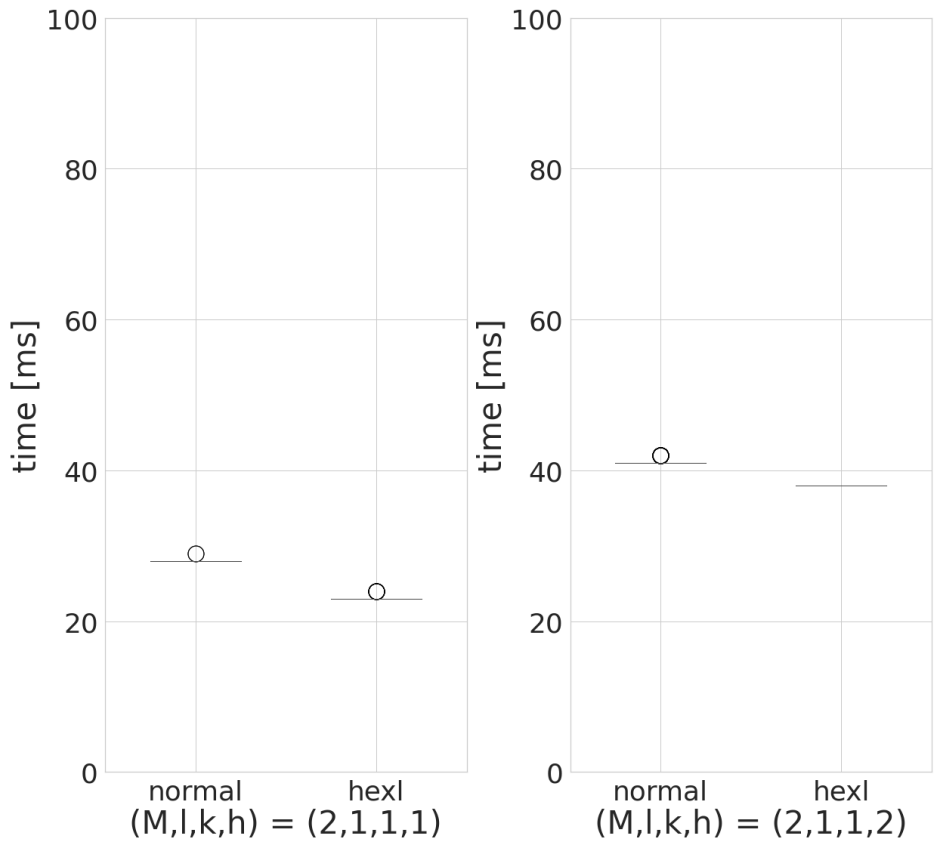

図 A.1　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1，rotate-and-sum の実行回数 M=2，暗号文のレベル l=1，KeySwitch に使用する特殊な法の数 k=1）

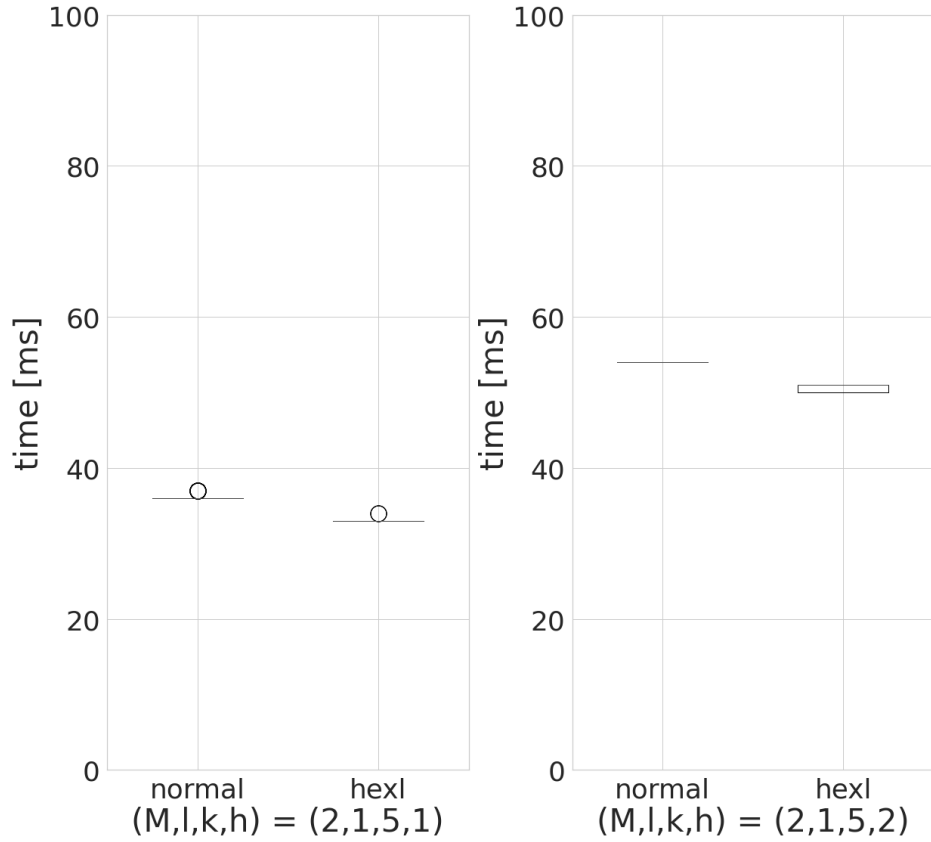

図 A.2　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1，rotate-and-sum の実行回数 M=2，暗号文のレベル l=1，KeySwitch に使用する特殊な法の数 k=5）

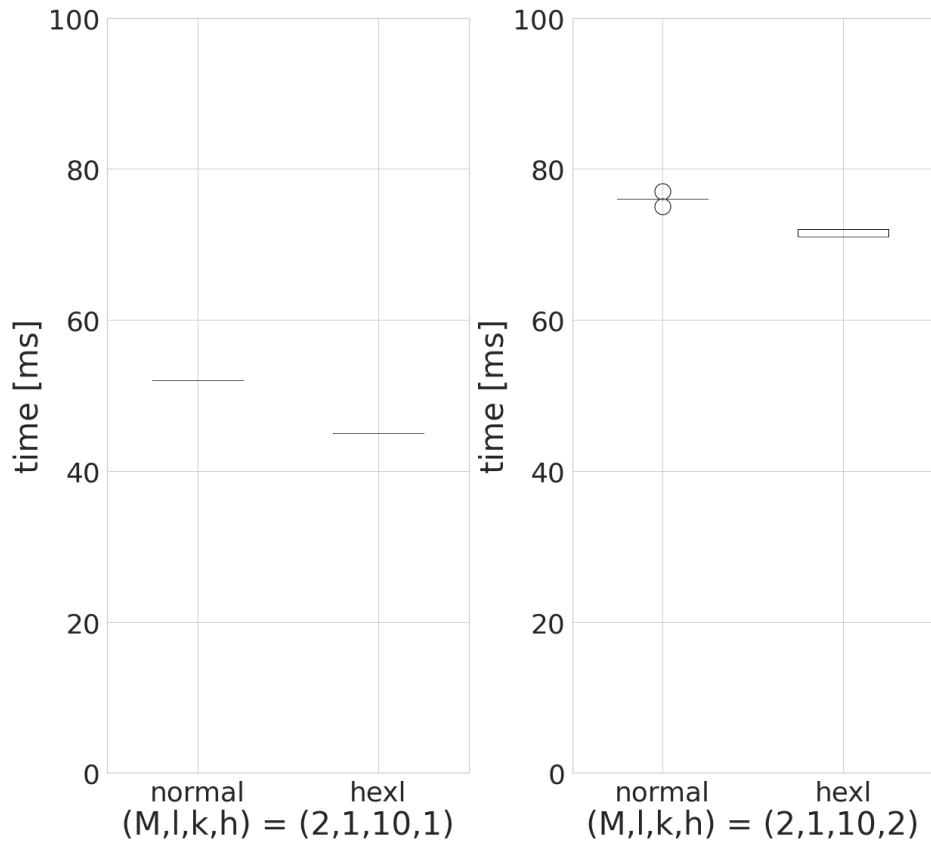

図 A.3　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1，rotate-and-sum の実行回数 M=2，暗号文のレベル l=1，KeySwitch に使用する特殊な法の数 k=10）

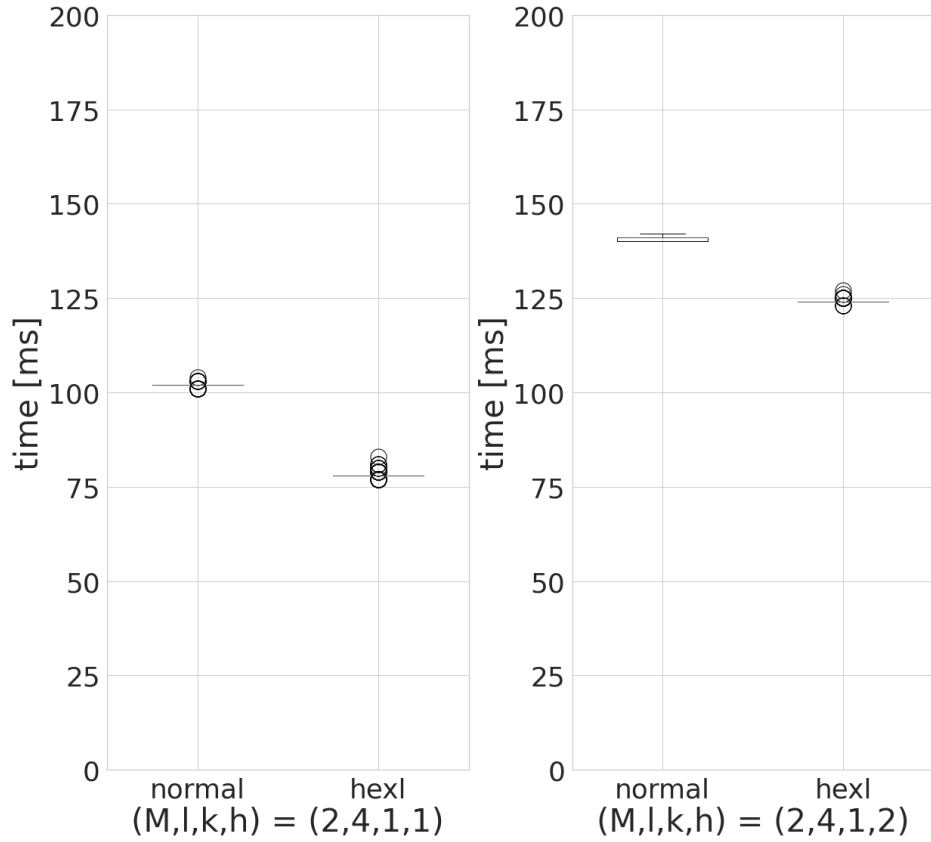

図 A.4　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1，rotate-and-sum の実行回数 M=2，暗号文のレベル l=4，KeySwitch に使用する特殊な法の数 k=1）

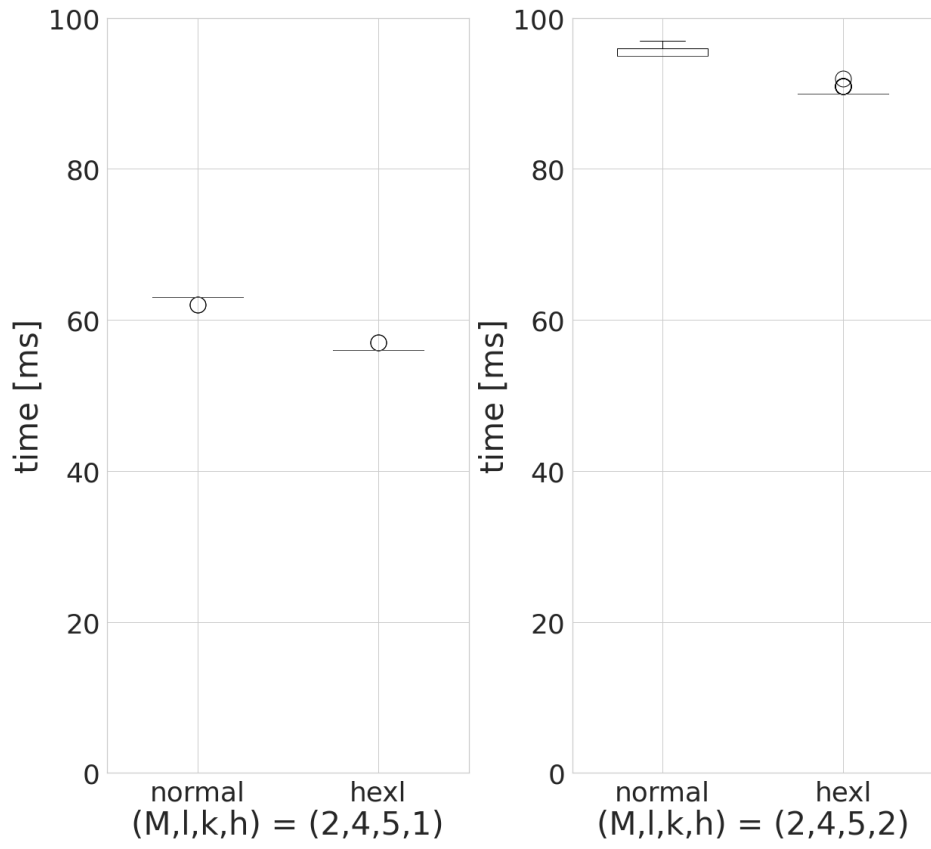

図 A.5　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1，rotate-and-sum の実行回数 M=2，暗号文のレベル l=4，KeySwitch に使用する特殊な法の数 k=5）

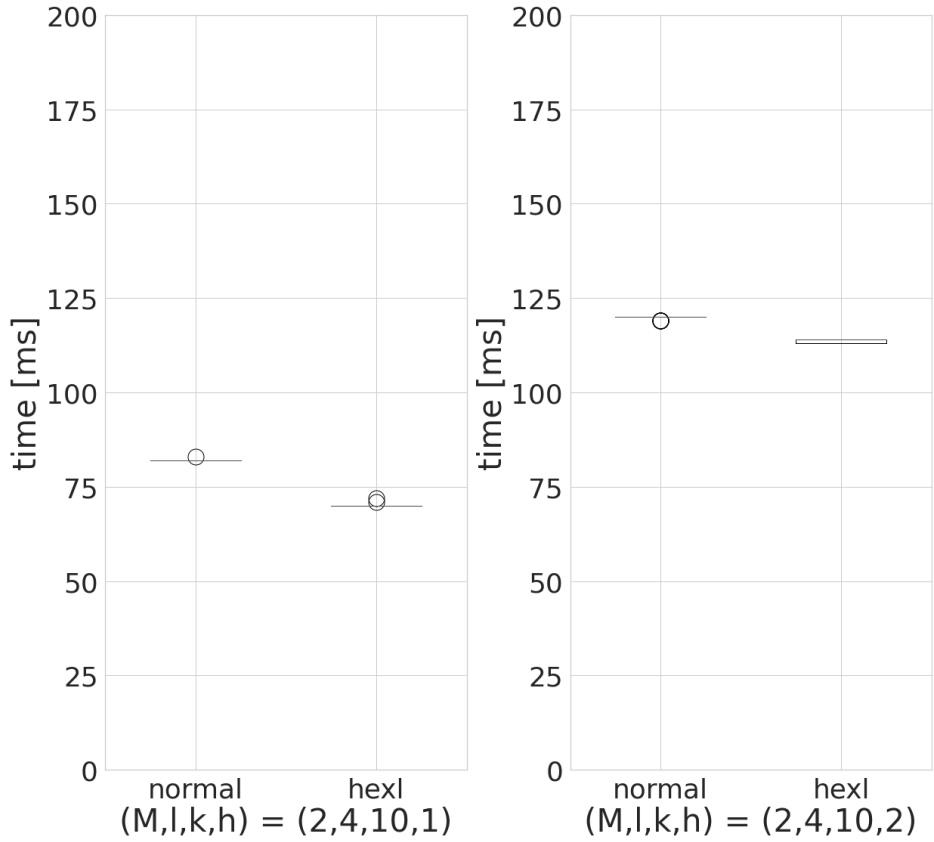

図 A.6　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1，rotate-and-sum の実行回数 M=2，暗号文のレベル l=4，KeySwitch に使用する特殊な法の数 k=10）

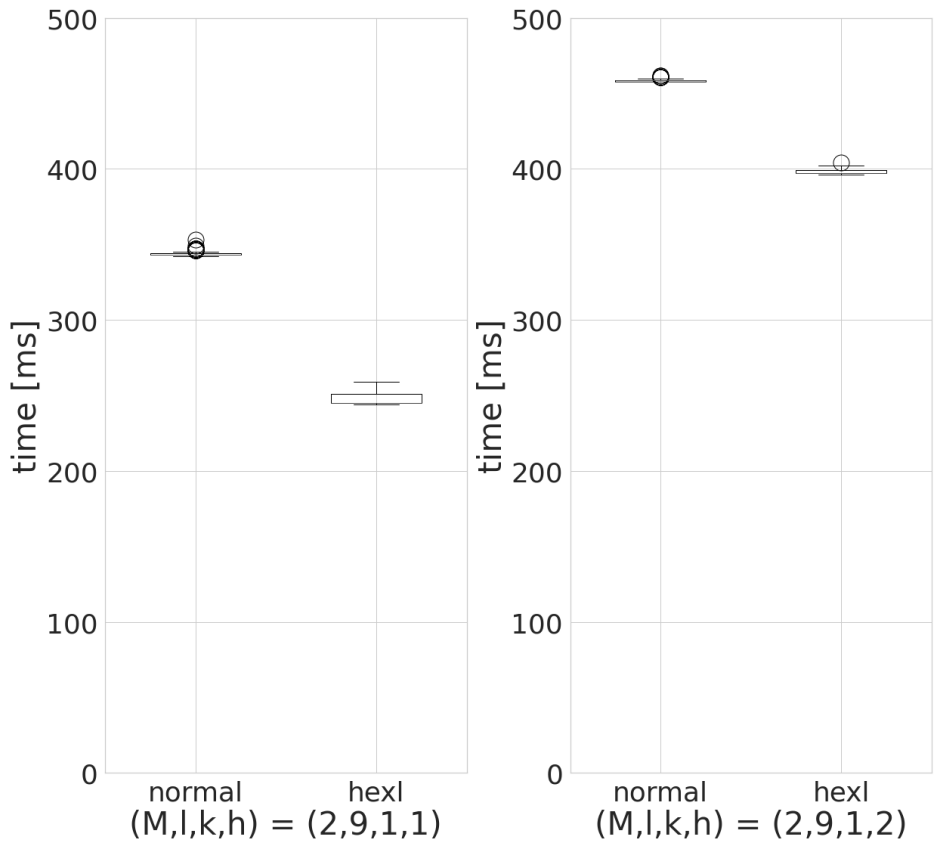

図 A.7　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1，rotate-and-sum の実行回数 M=2，暗号文のレベル l=9，KeySwitch に使用する特殊な法の数 k=1）

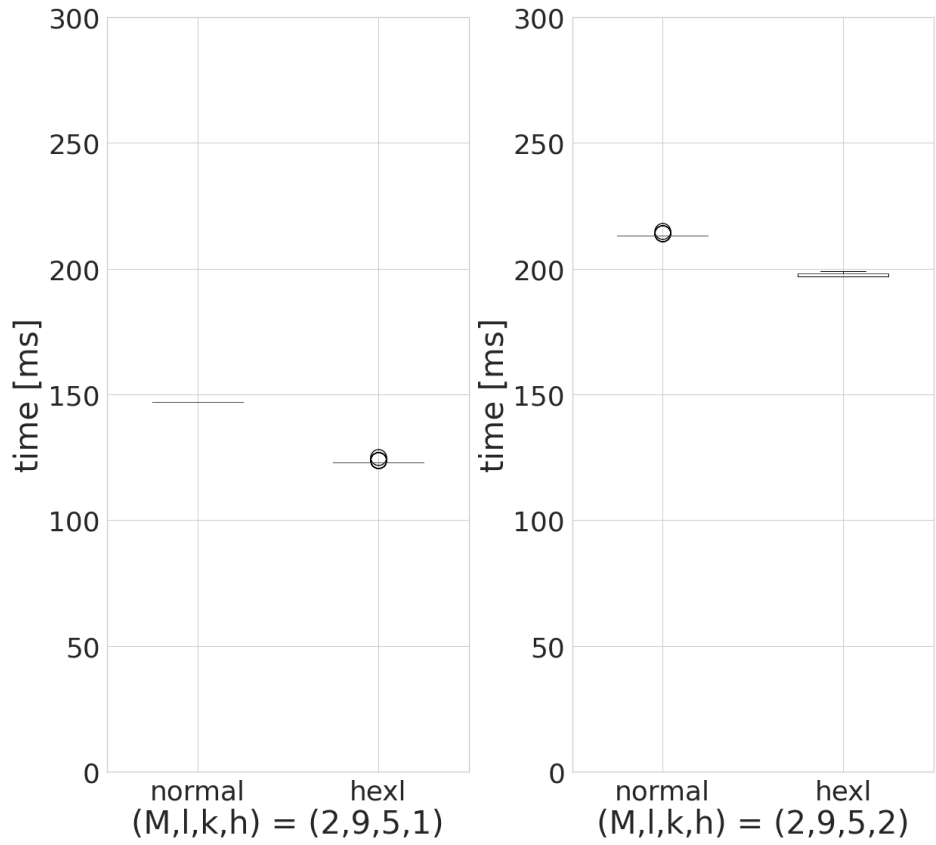

図 A.8　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1，rotate-and-sum の実行回数 M=2，暗号文のレベル l=9，KeySwitch に使用する特殊な法の数 k=5）

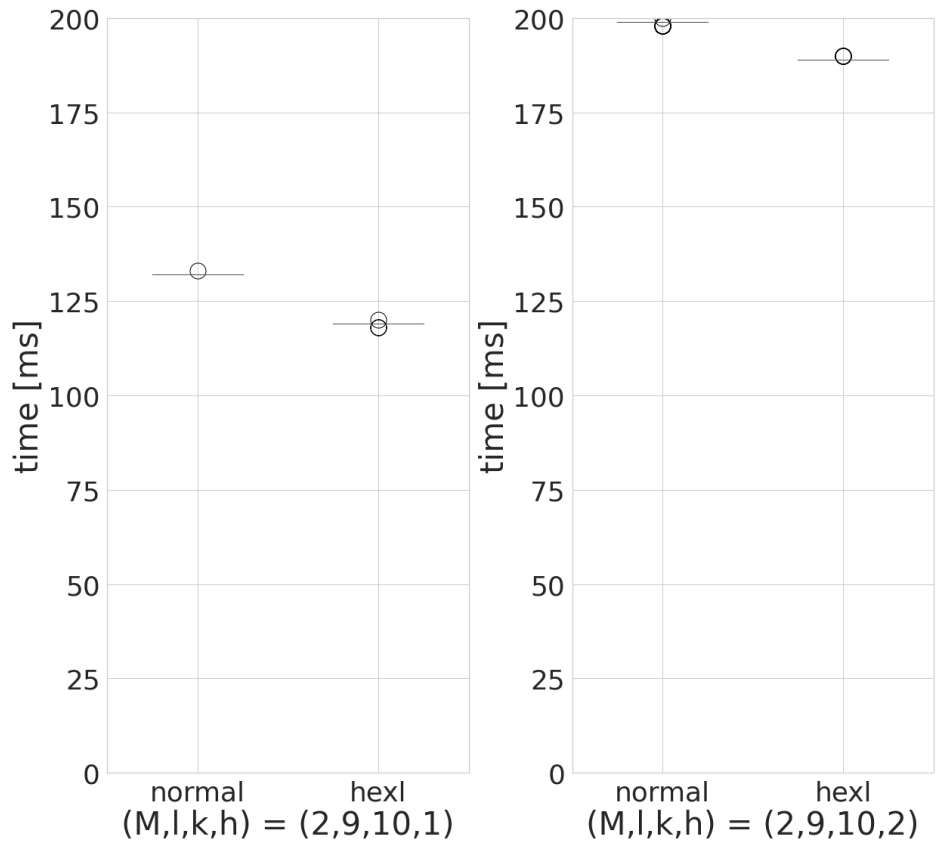

図 A.9　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1，rotate-and-sum の実行回数 M=2，暗号文のレベル l=9，KeySwitch に使用する特殊な法の数 k=10）

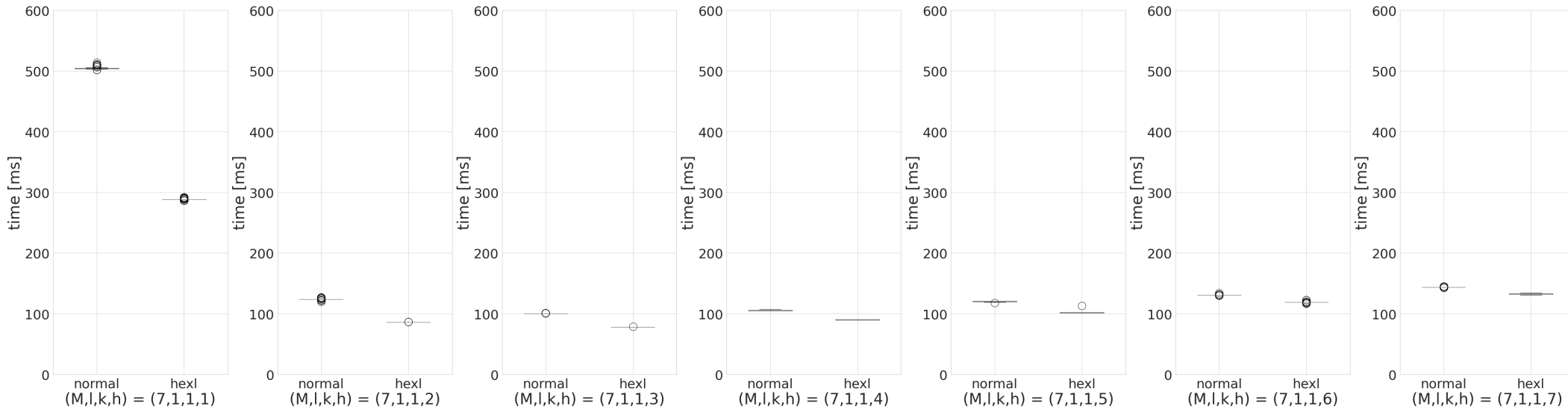

図 A.10　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1，rotate-and-sum の実行回数 M=7，暗号文のレベル l=1，KeySwitch に使用する特殊な法の数 k=1）

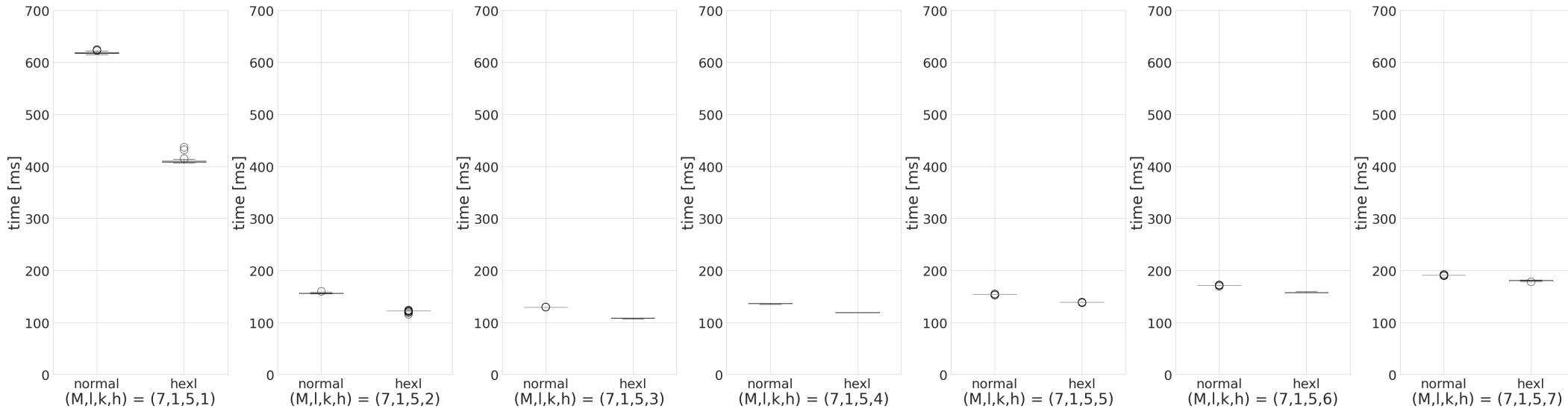

図 A.11　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1，rotate-and-sum の実行回数 M=7，暗号文のレベル l=1，KeySwitch に使用する特殊な法の数 k=5）

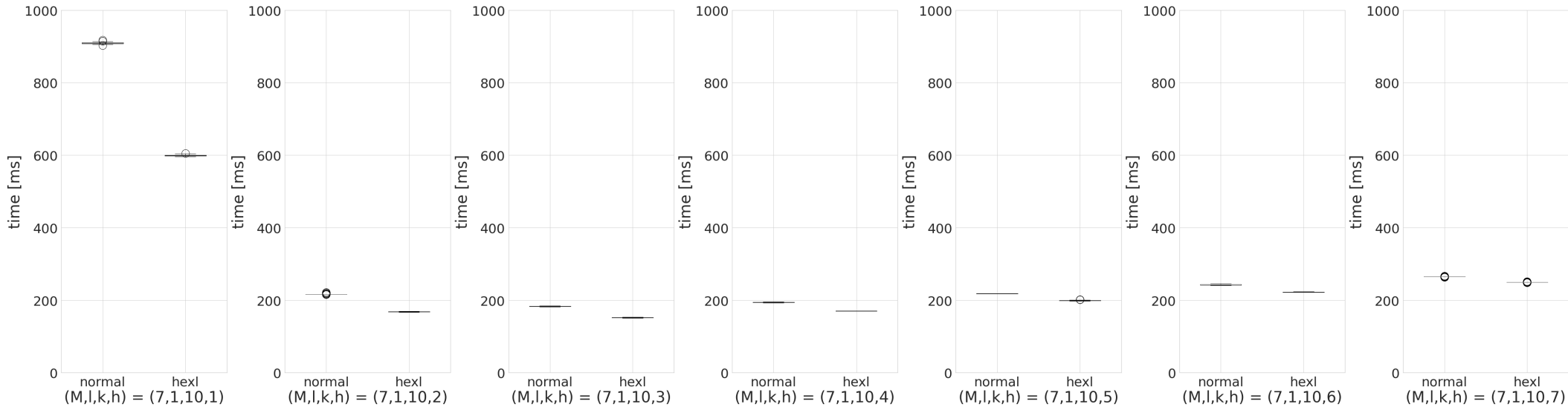

図 A.12　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1，rotate-and-sum の実行回数 M=7，暗号文のレベル l=1，KeySwitch に使用する特殊な法の数 k=10）

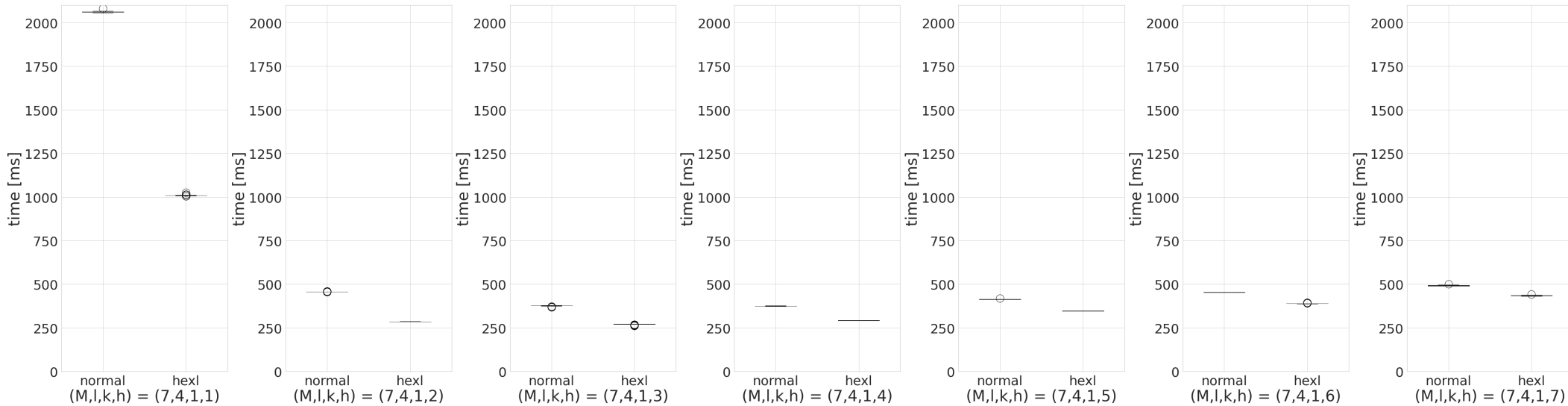

図 A.13　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1, rotate-and-sum の実行回数 M=7, 暗号文のレベル l=4, KeySwitch に使用する特殊な法の数 k=1）

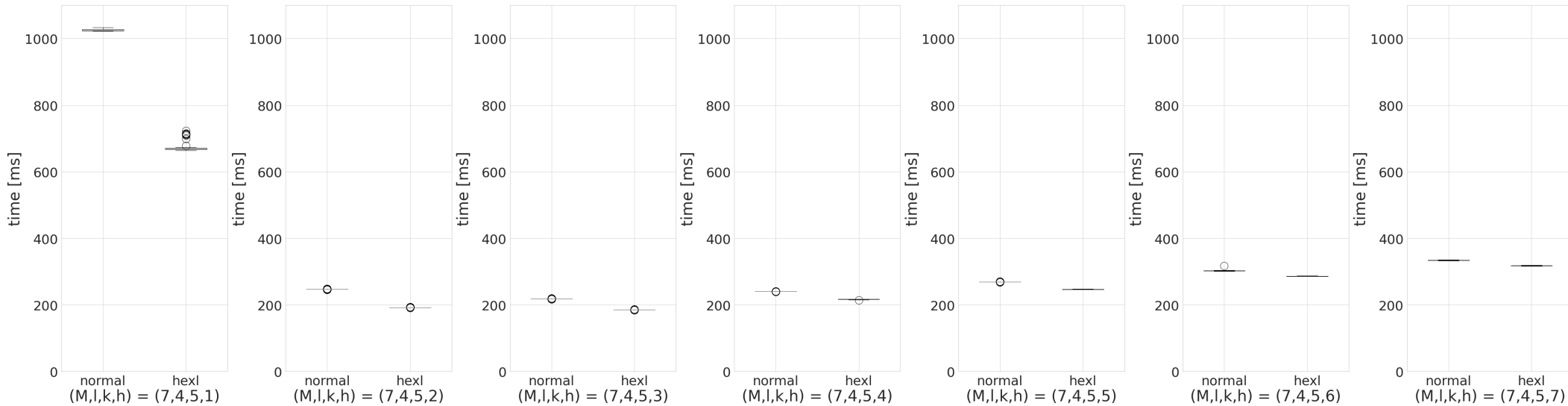

図 A.14　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1，rotate-and-sum の実行回数 M=7，暗号文のレベル l=1，KeySwitch に使用する特殊な法の数 k=5）

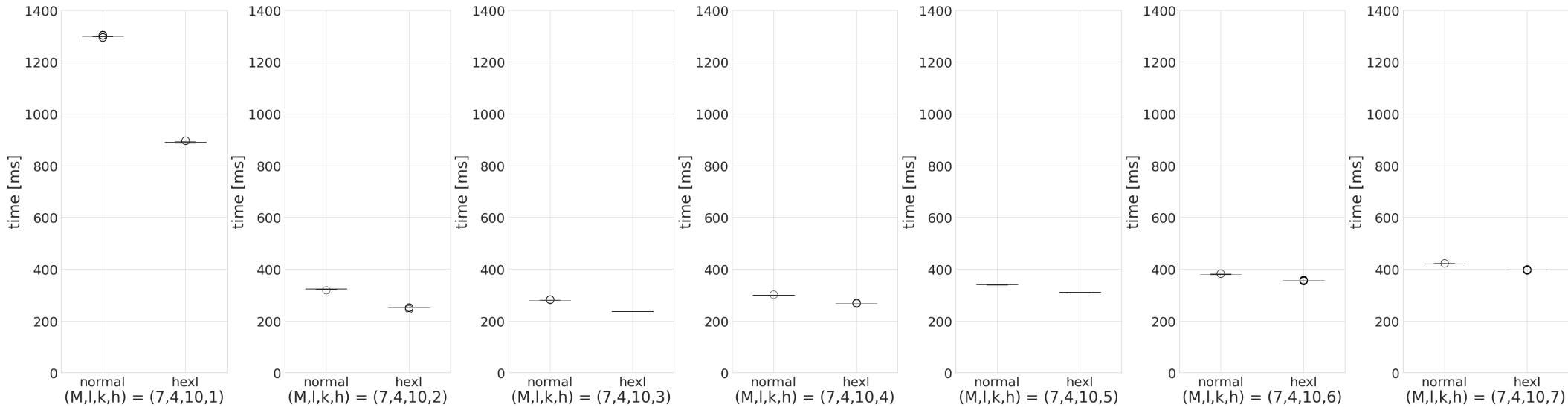

図 A.15　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1，rotate-and-sum の実行回数 M=7，暗号文のレベル l=1，KeySwitch に使用する特殊な法の数 k=10）

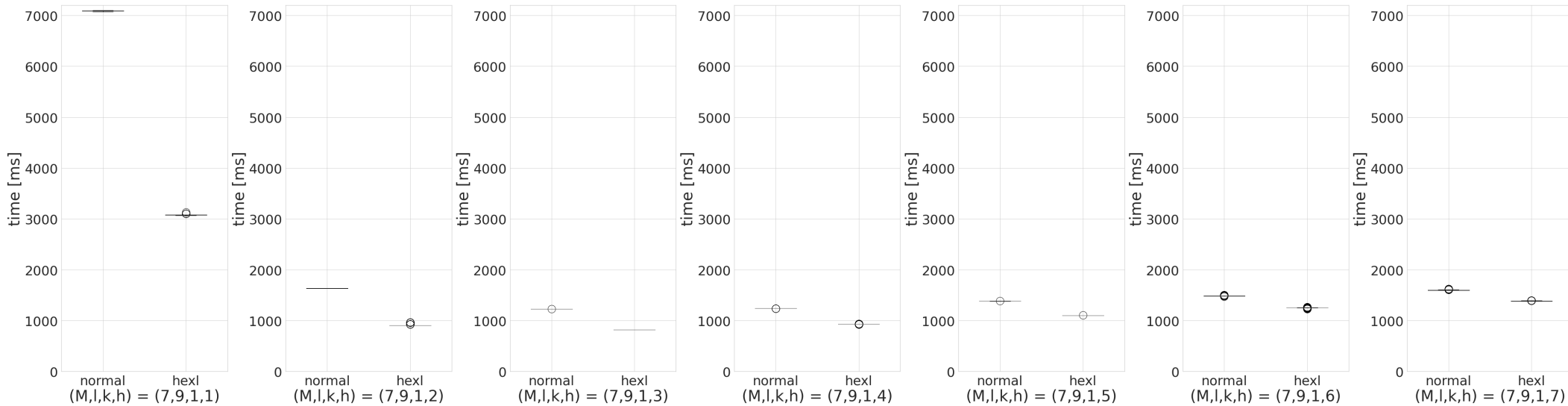

図 A.16　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1, rotate-and-sum の実行回数 M=7, 暗号文のレベル l=9, KeySwitch に使用する特殊な法の数 k=1）

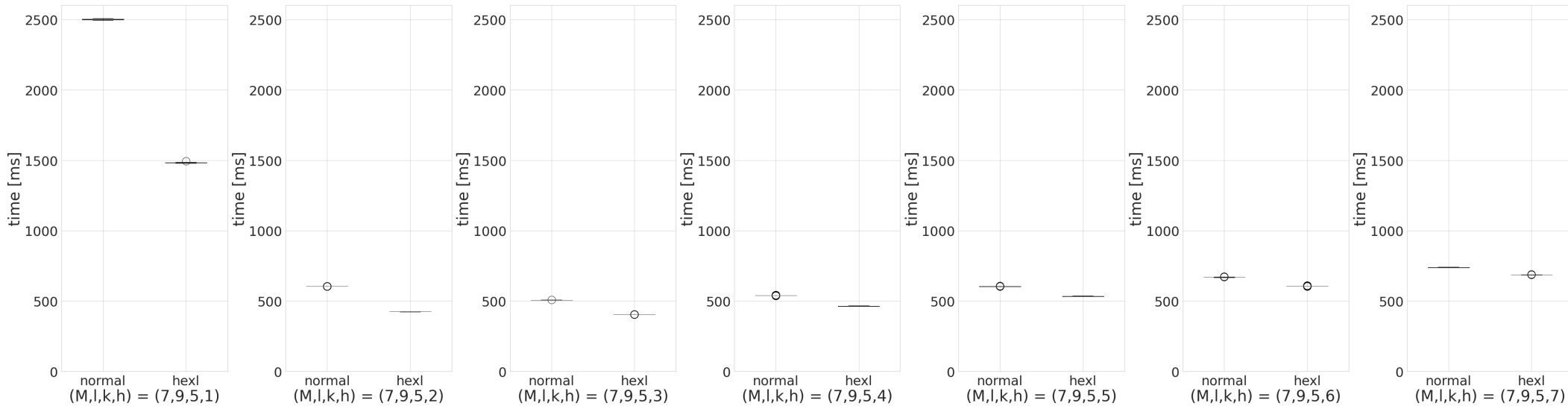

図 A.17　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1, rotate-and-sum の実行回数 M=7, 暗号文のレベル l=9, KeySwitch に使用する特殊な法の数 k=5）

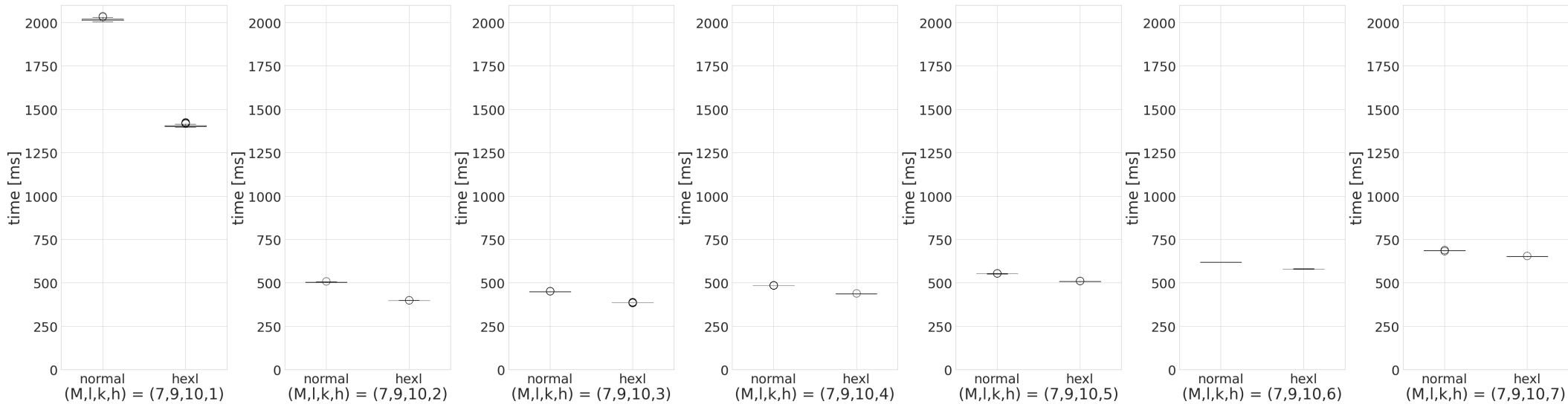

図 A.18　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1, rotate-and-sum の実行回数 M=7, 暗号文のレベル l=9, KeySwitch に使用する特殊な法の数 k=10）

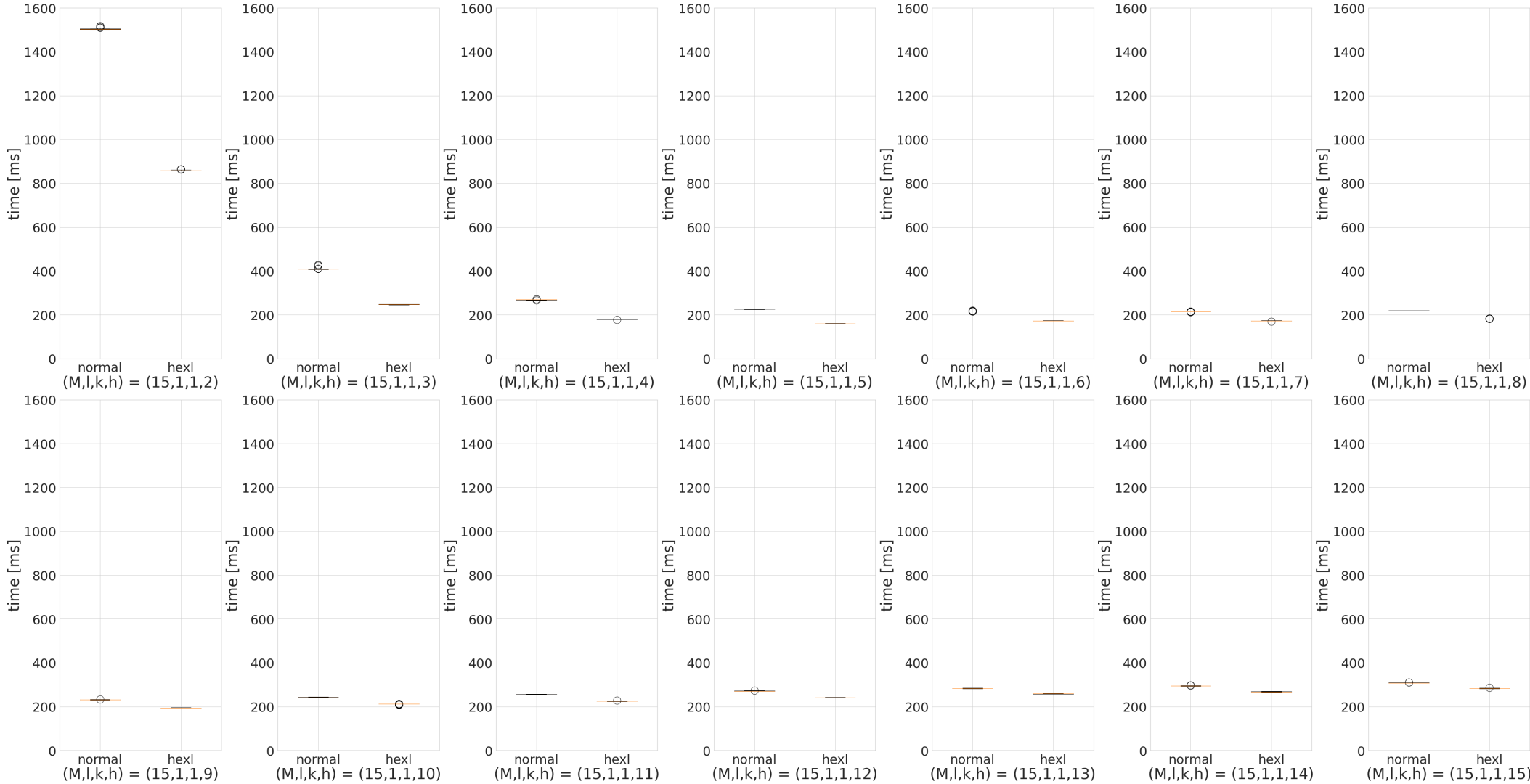

図 A.19　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=1, KeySwitch に使用する特殊な法の数 k=1）

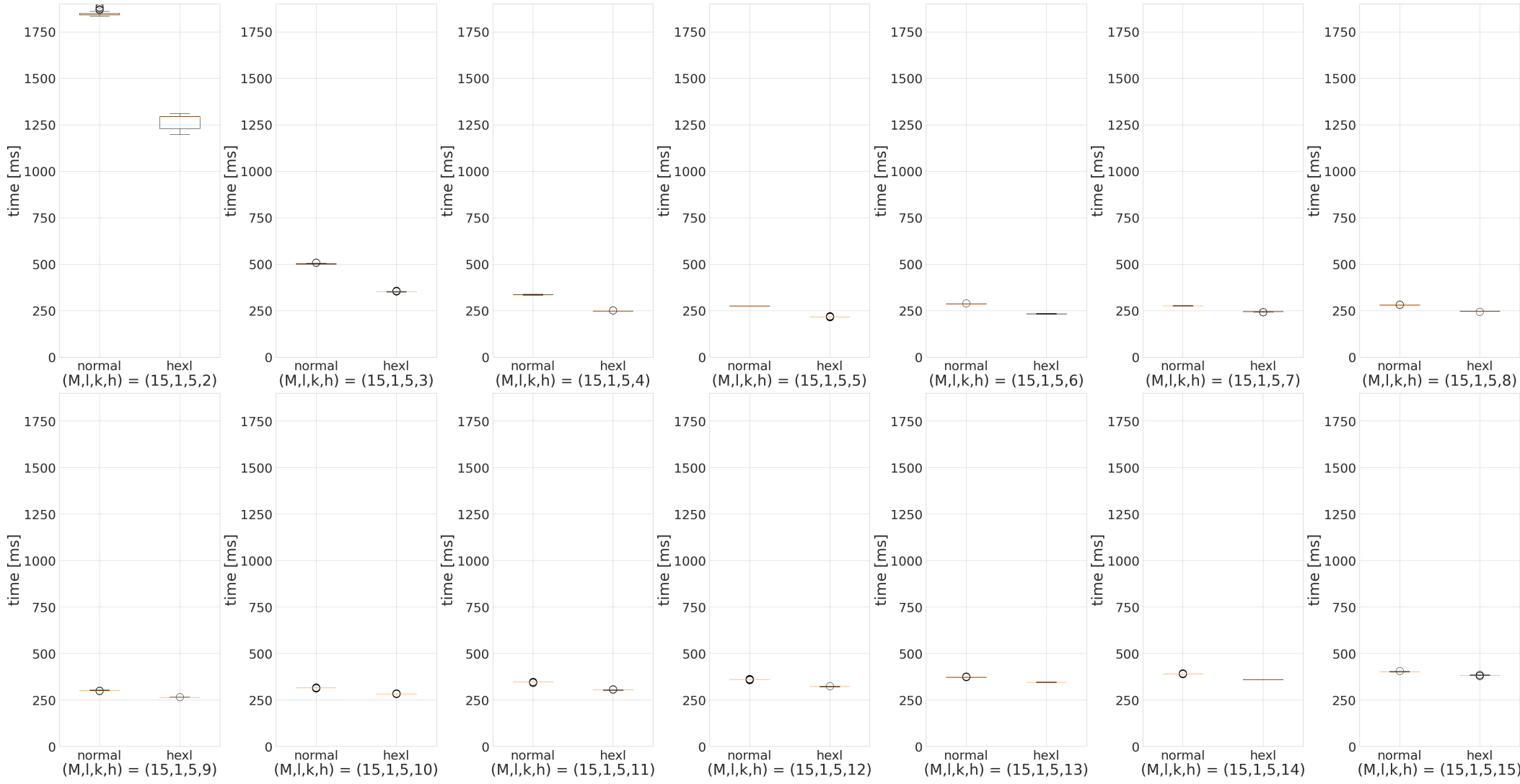

図 A.20　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=1, KeySwitch に使用する特殊な法の数 k=5）

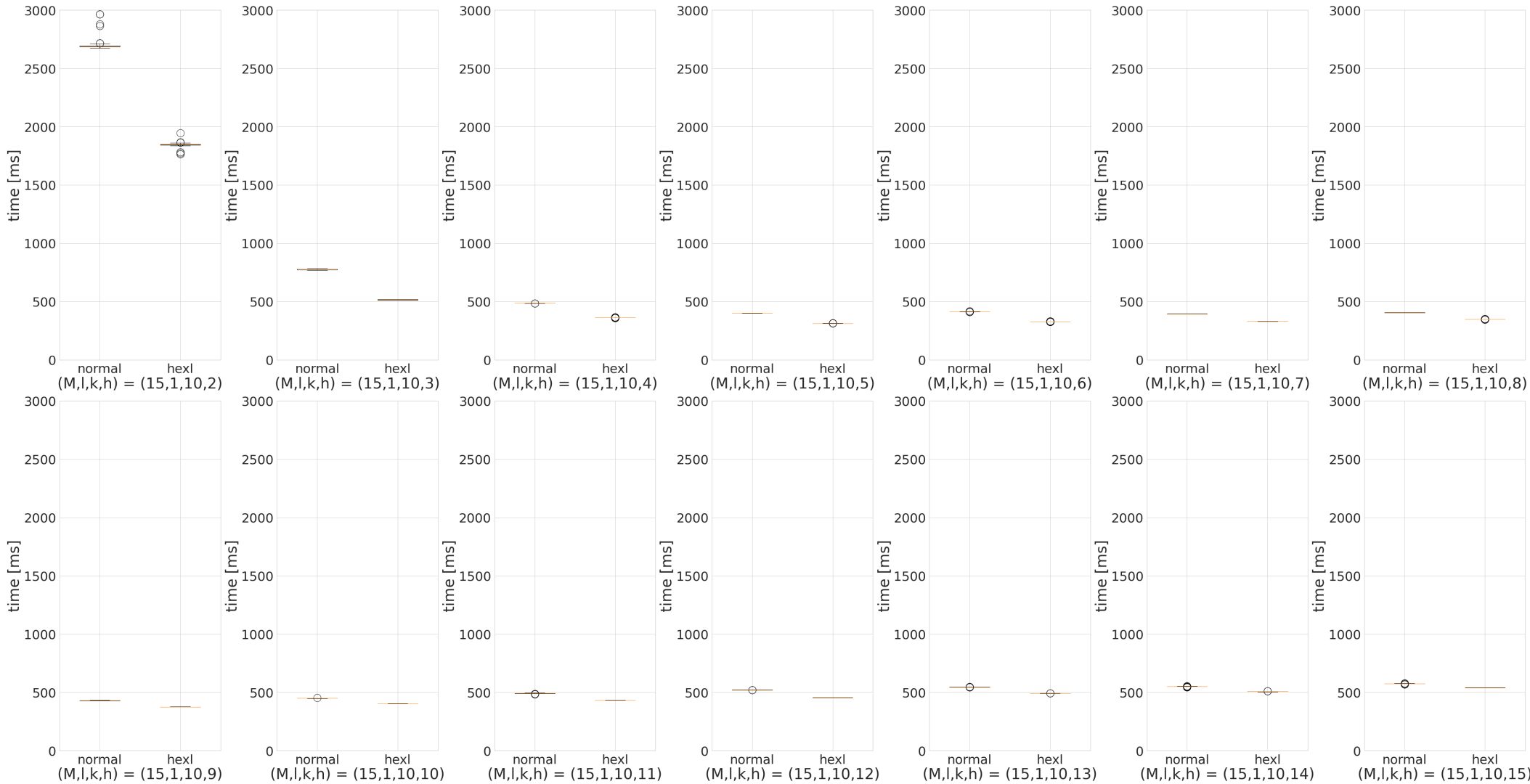

図 A.21　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=1, KeySwitch に使用する特殊な法の数 k=10）

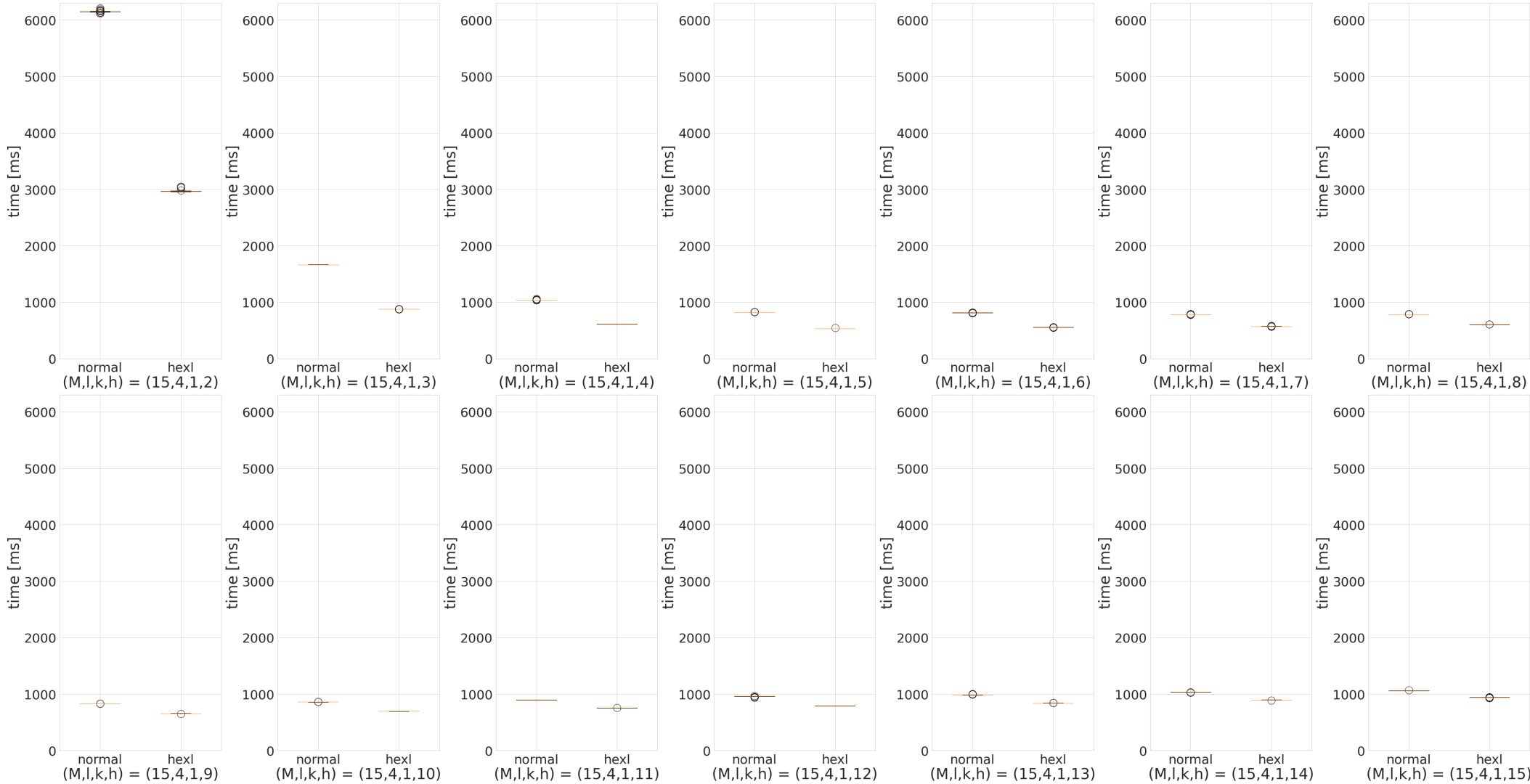

図 A.22　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=4, KeySwitch に使用する特殊な法の数 k=1）

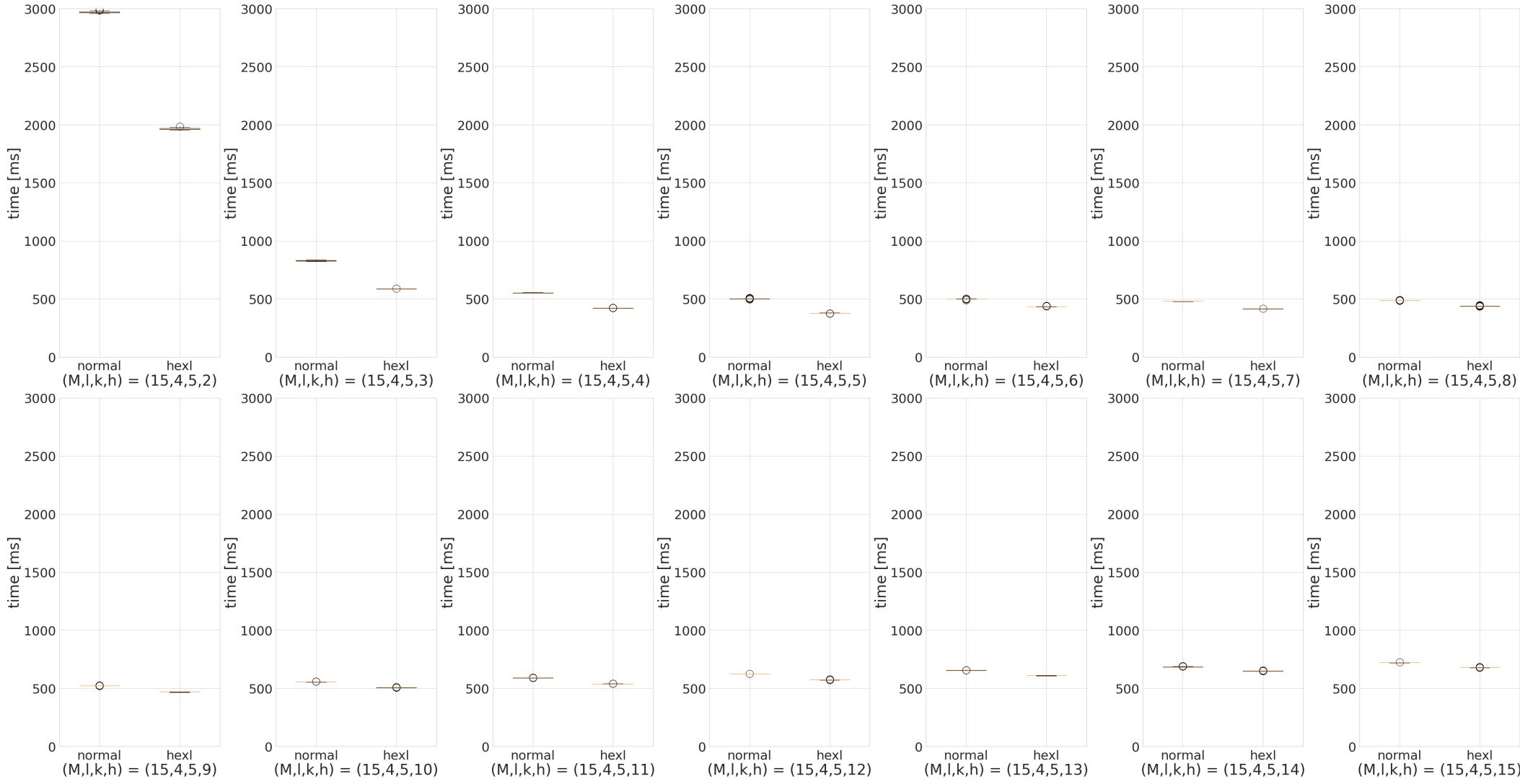

図 A.23　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=4, KeySwitch に使用する特殊な法の数 k=5）

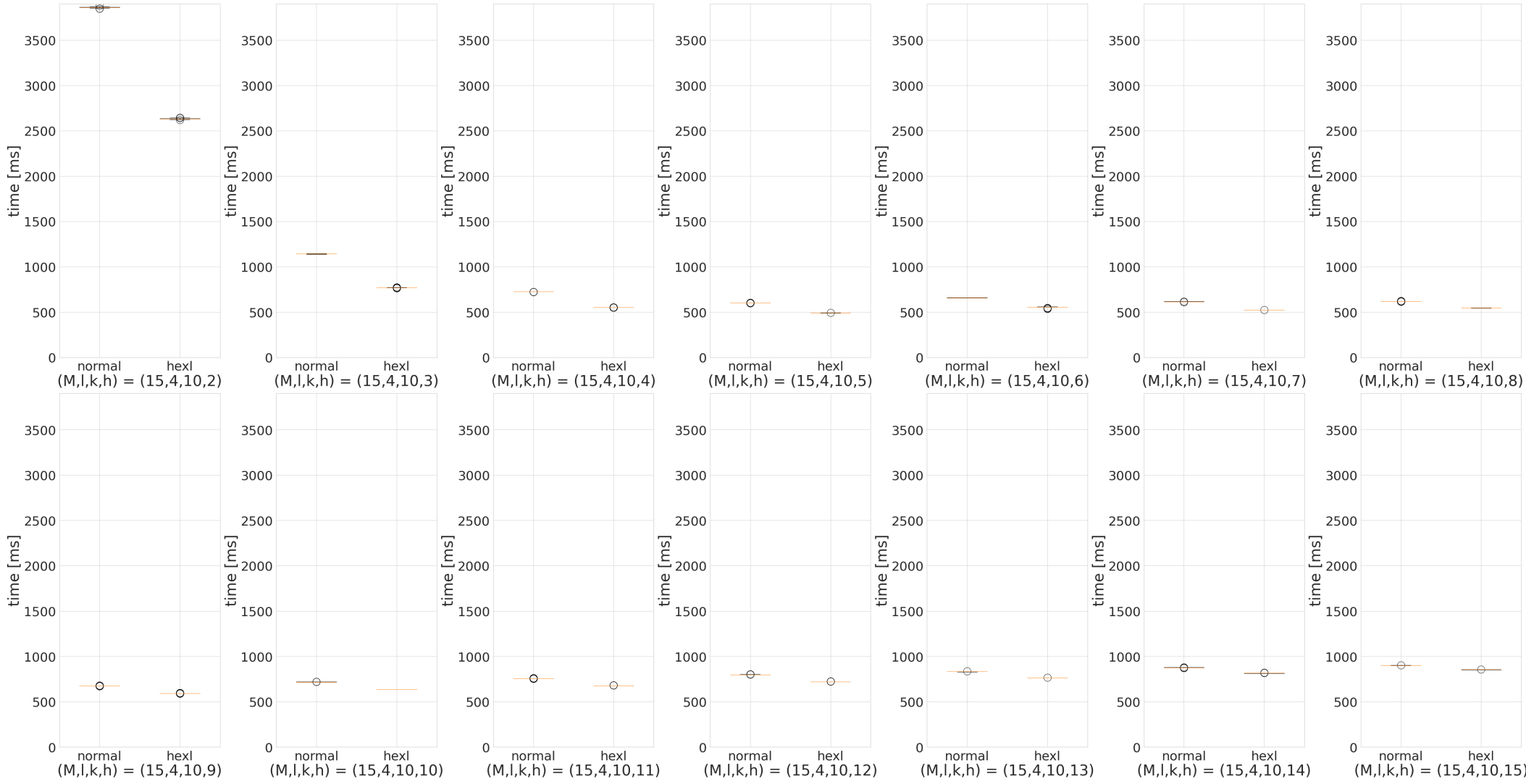

図 A.24　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=4, KeySwitch に使用する特殊な法の数 k=10）

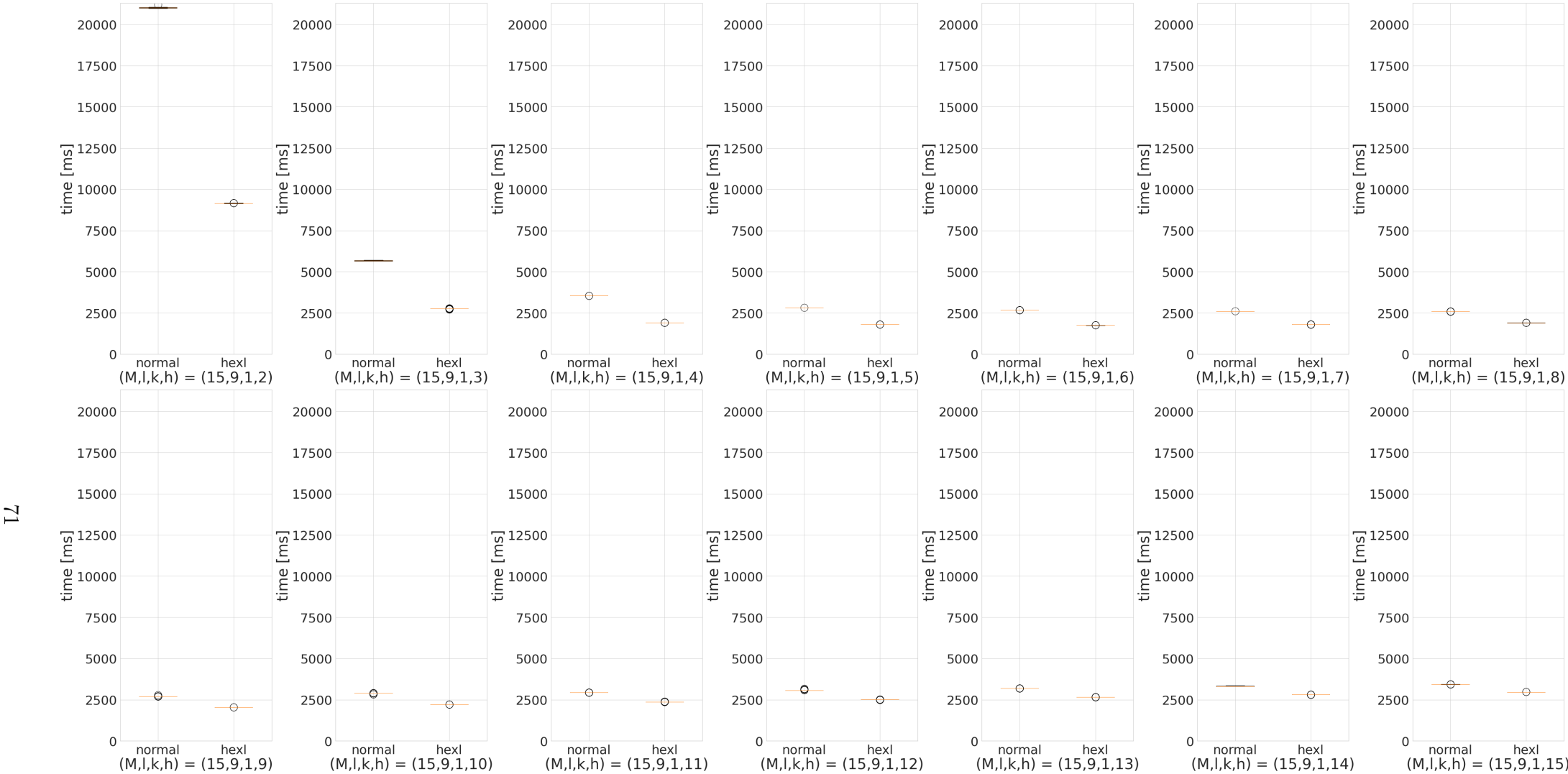

図 A.25　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=9, KeySwitch に使用する特殊な法の数 k=1）

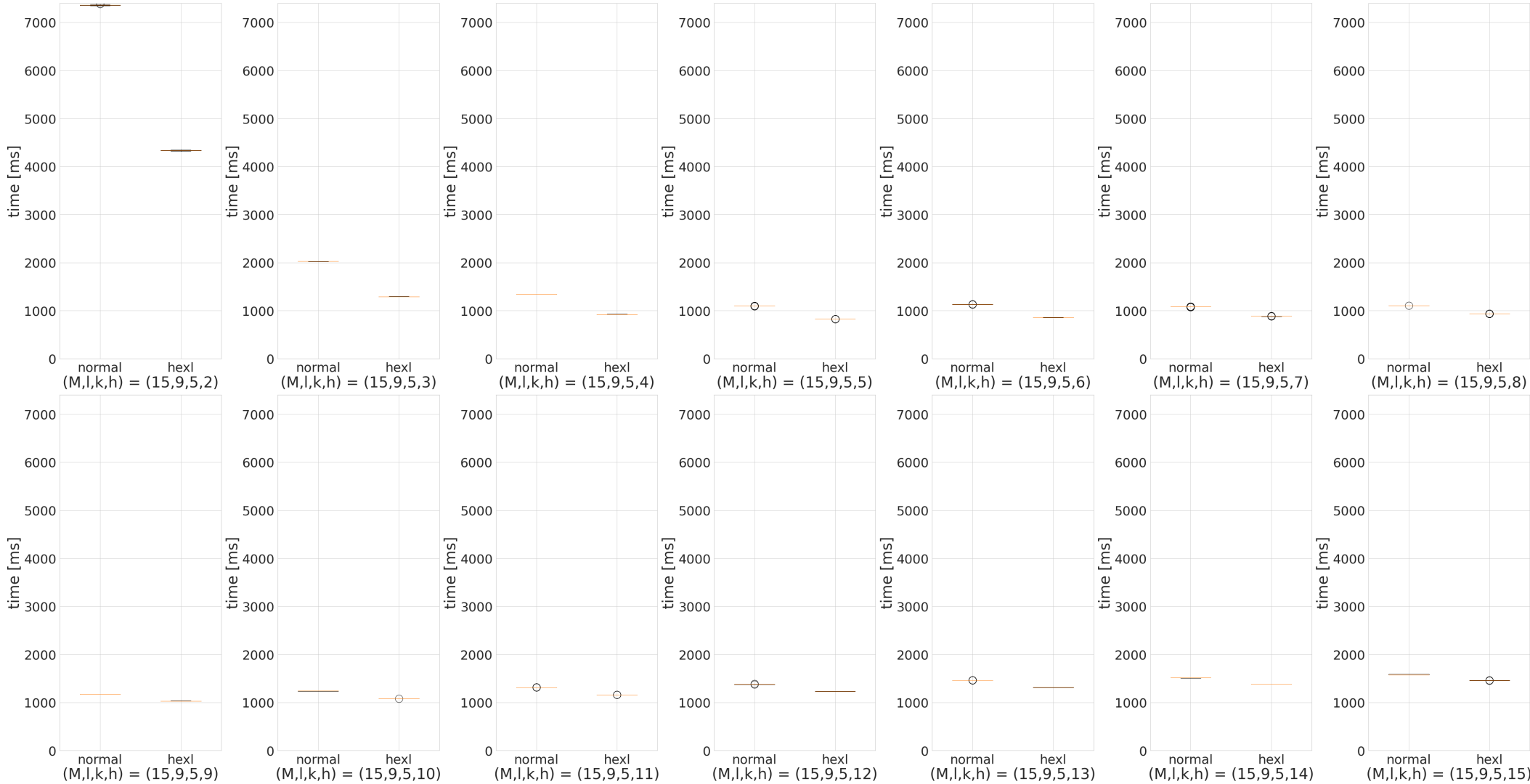

図 A.26　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=9, KeySwitch に使用する特殊な法の数 k=5）

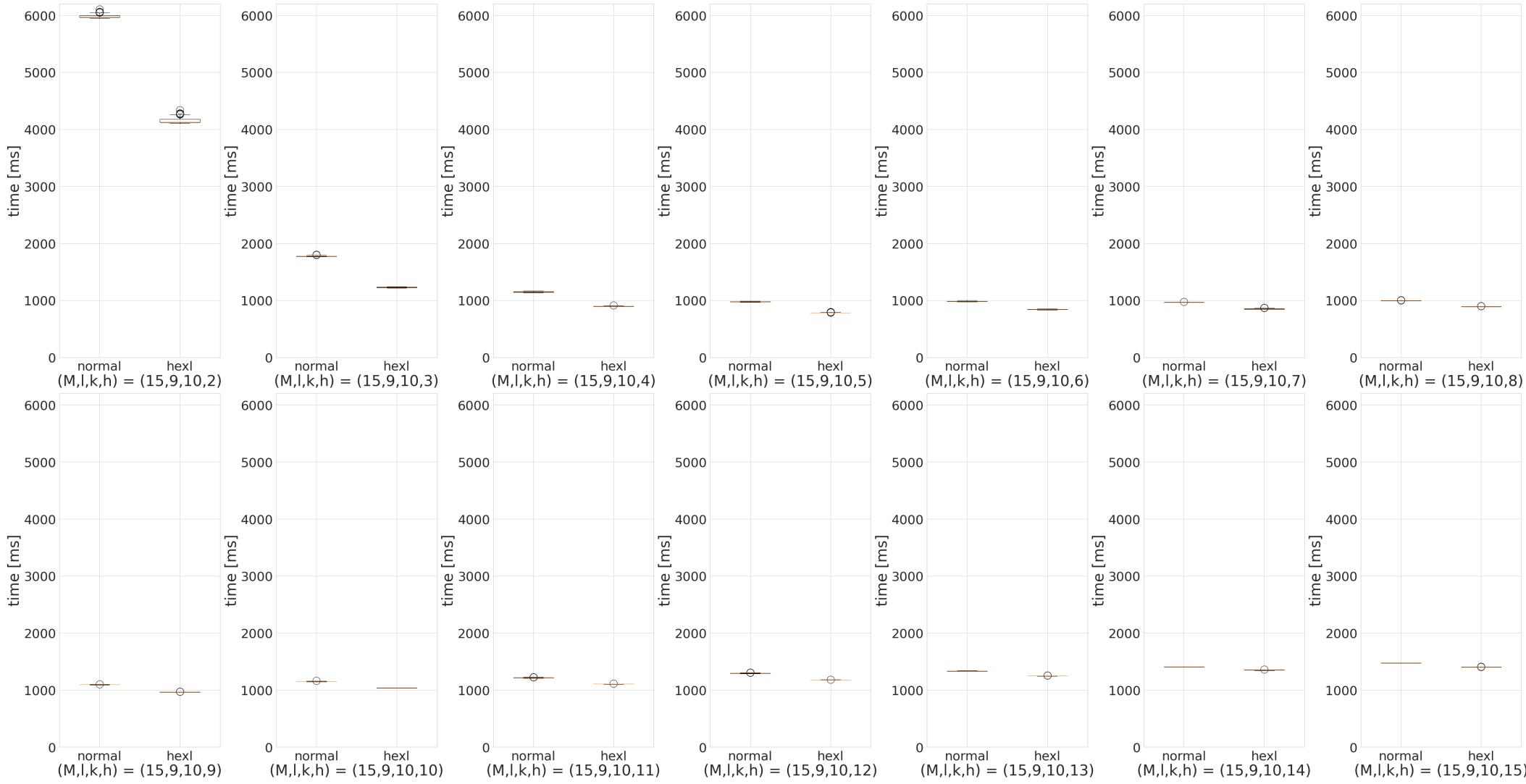

図 A.27　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 1, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=9, KeySwitch に使用する特殊な法の数 k=10）

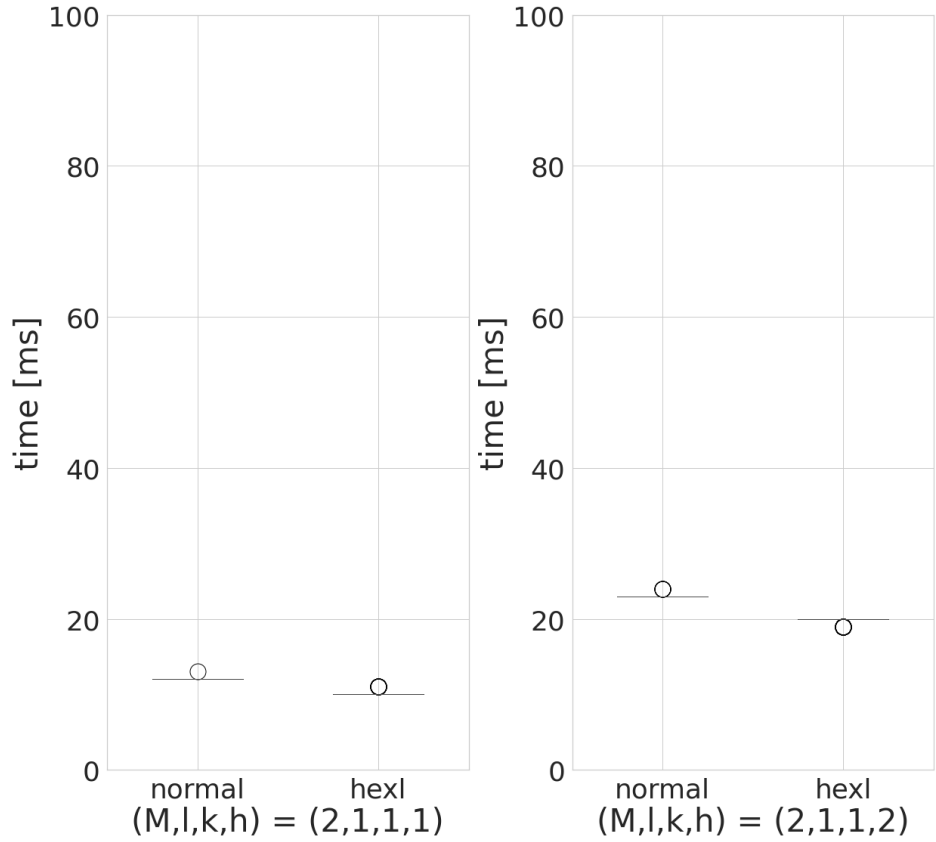

図 A.28　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=2，暗号文のレベル l=1，KeySwitch に使用する特殊な法の数 k=1）

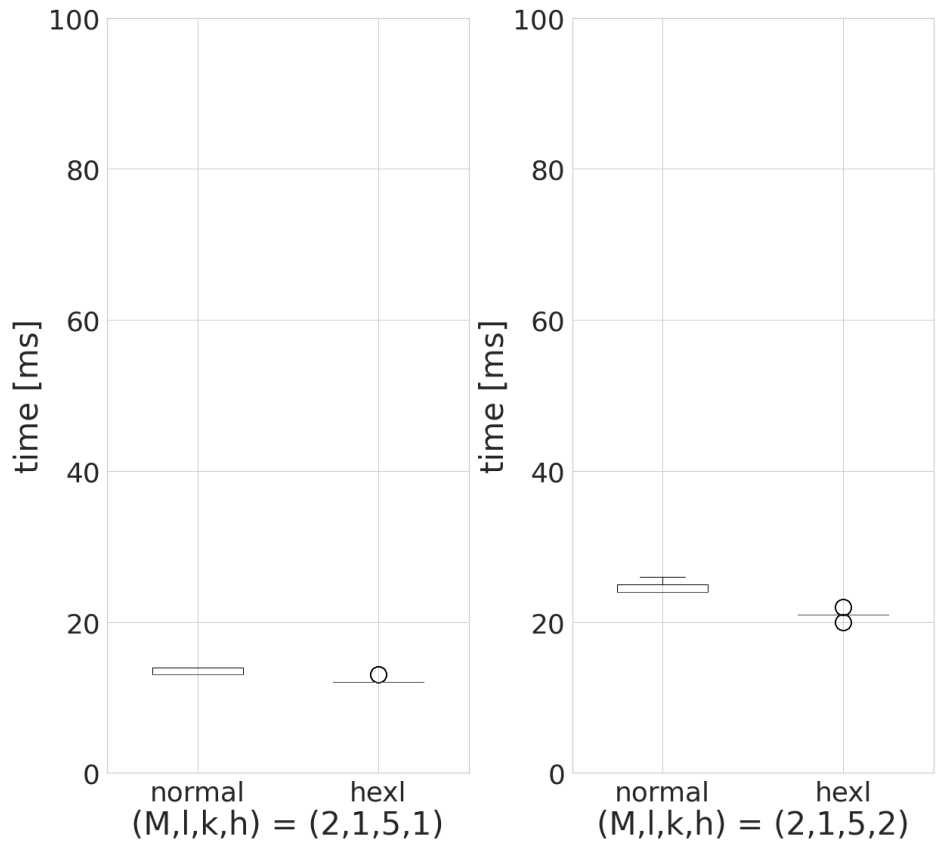

図 A.29　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=2，暗号文のレベル l=1，KeySwitch に使用する特殊な法の数 k=5）

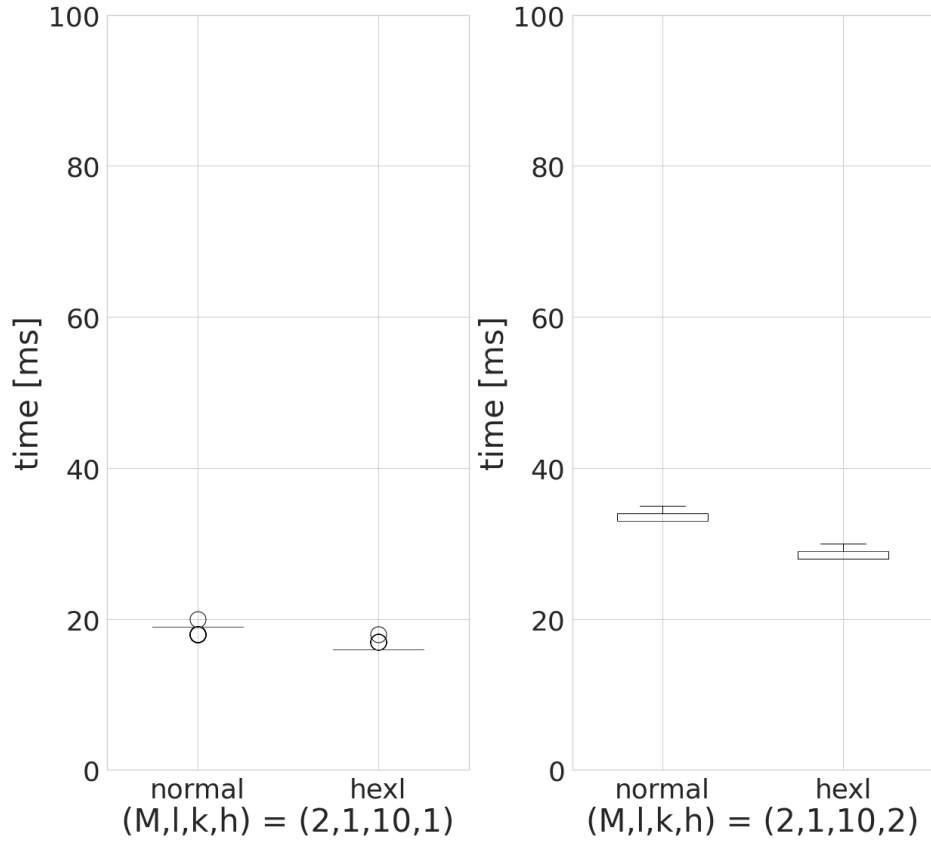

図 A.30　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=2，暗号文のレベル l=1，KeySwitch に使用する特殊な法の数 k=10）

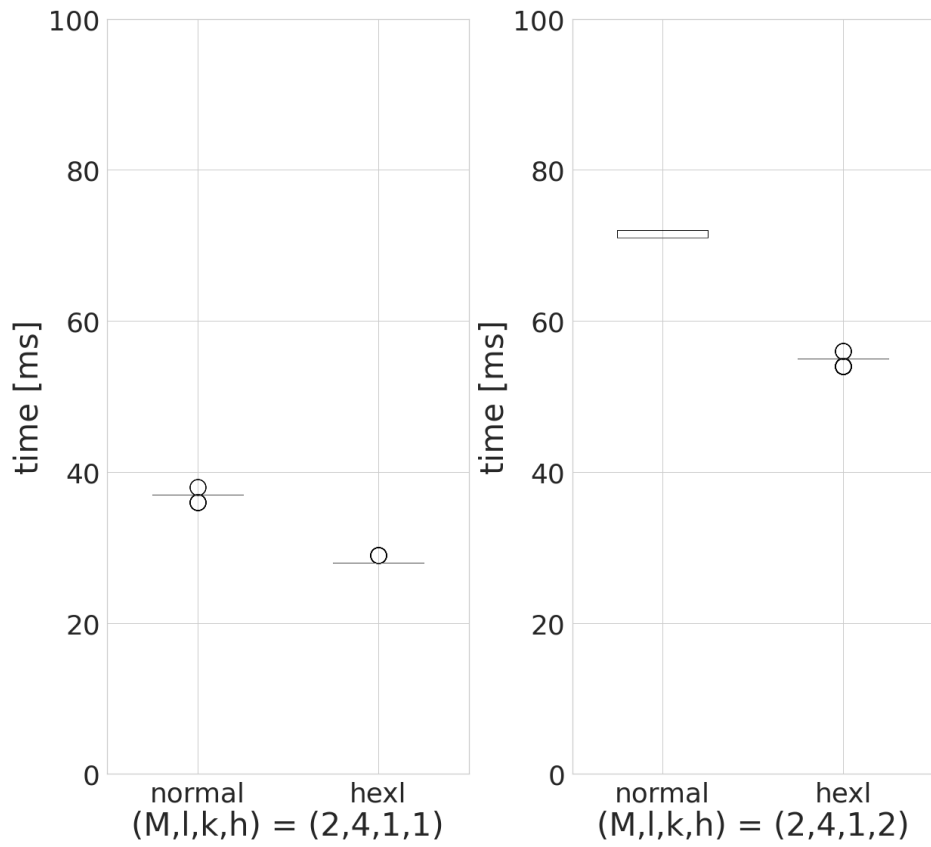

図 A.31　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=2，暗号文のレベル l=4，KeySwitch に使用する特殊な法の数 k=1）

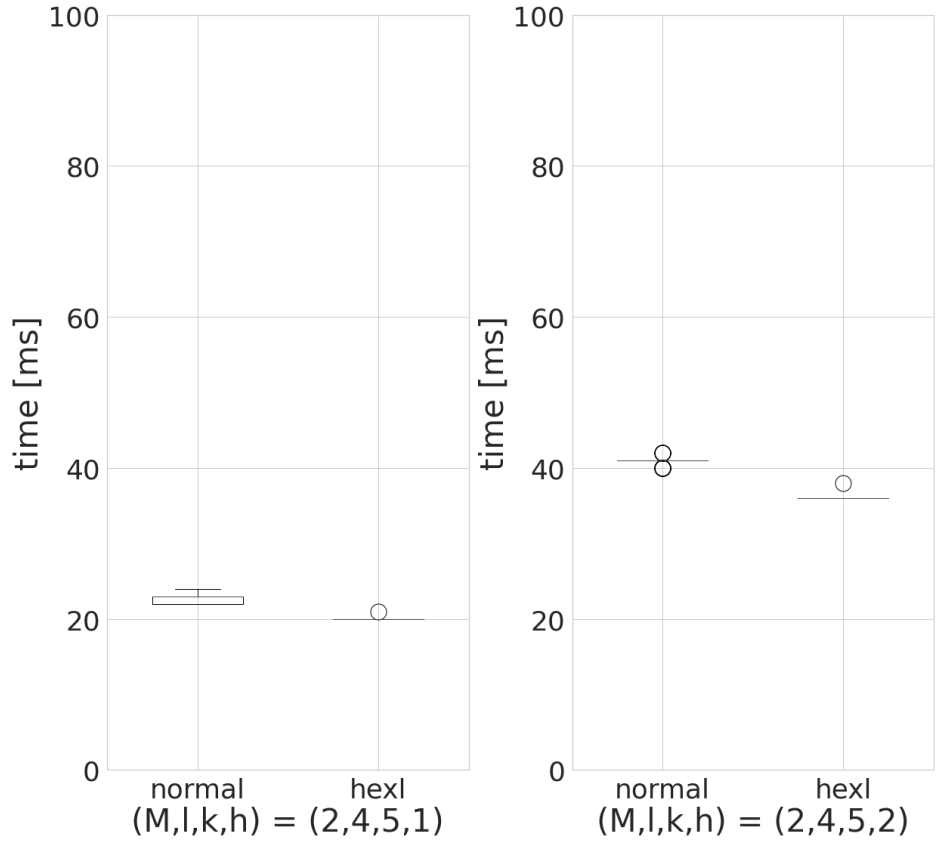

図 A.32　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=2，暗号文のレベル l=4，KeySwitch に使用する特殊な法の数 k=5）

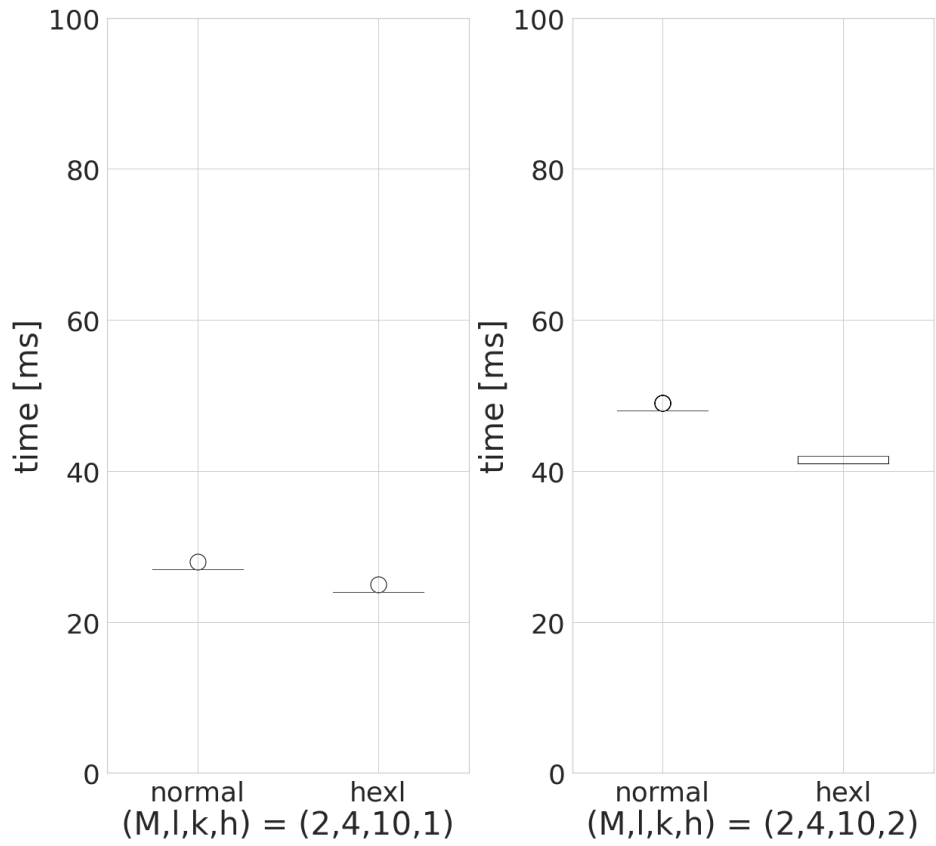

図 A.33　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=2，暗号文のレベル l=4，KeySwitch に使用する特殊な法の数 k=10）

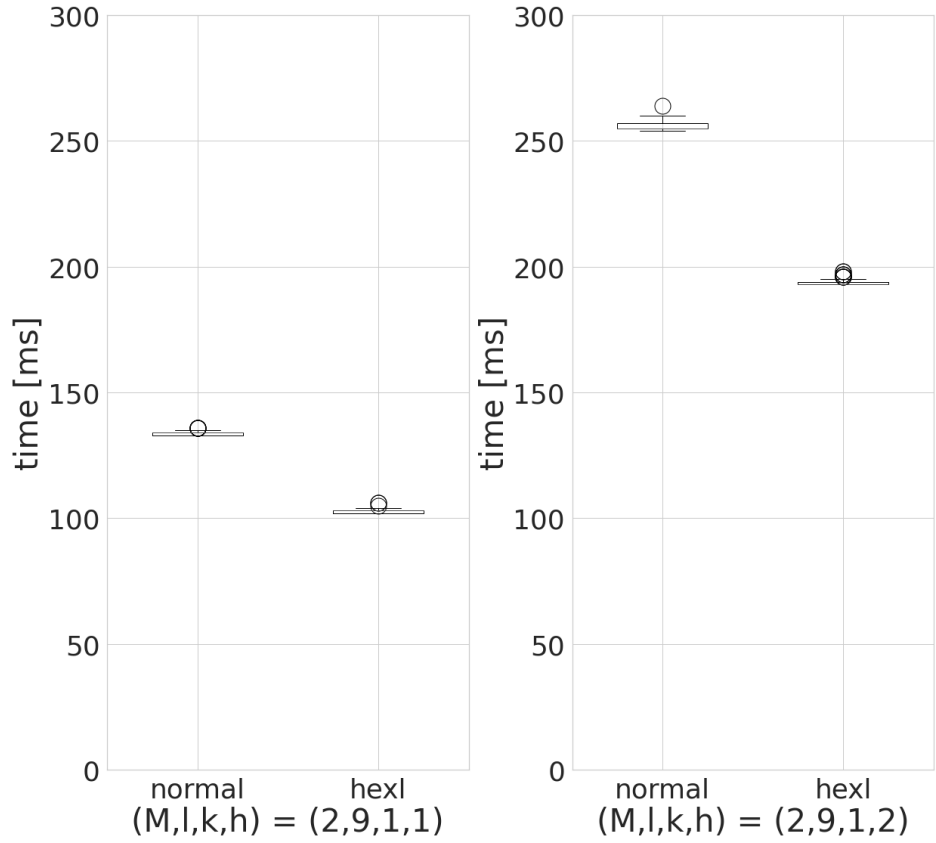

図 A.34　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=2，暗号文のレベル l=9，KeySwitch に使用する特殊な法の数 k=1）

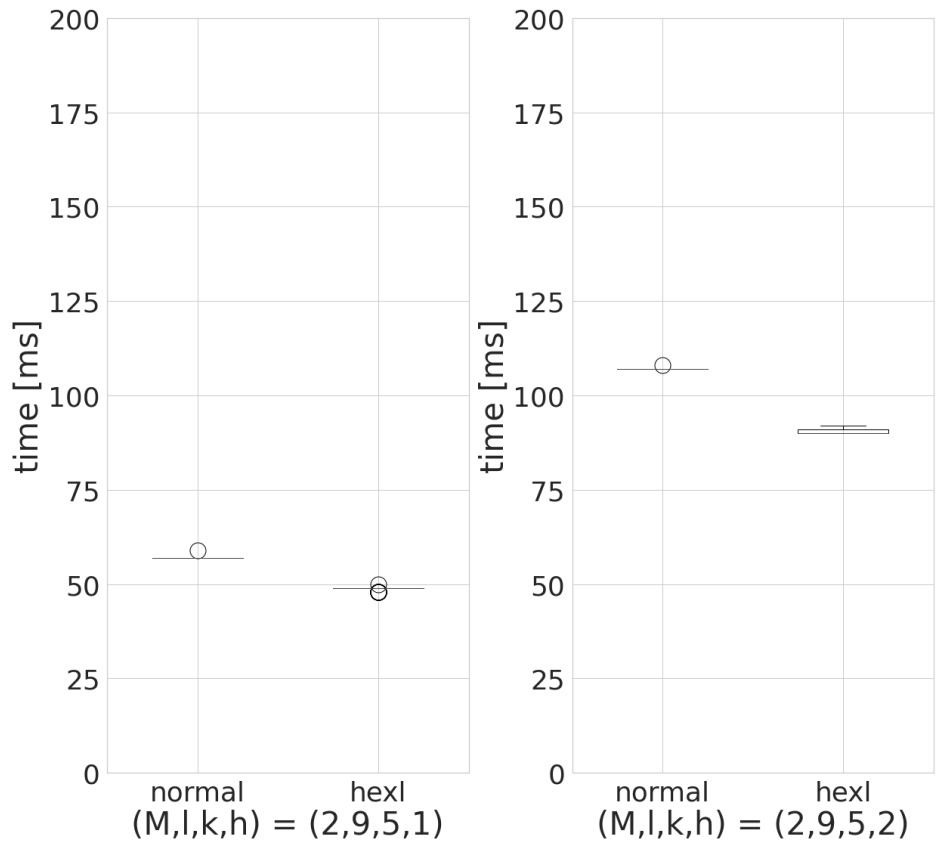

図 A.35　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=2，暗号文のレベル l=9，KeySwitch に使用する特殊な法の数 k=5）

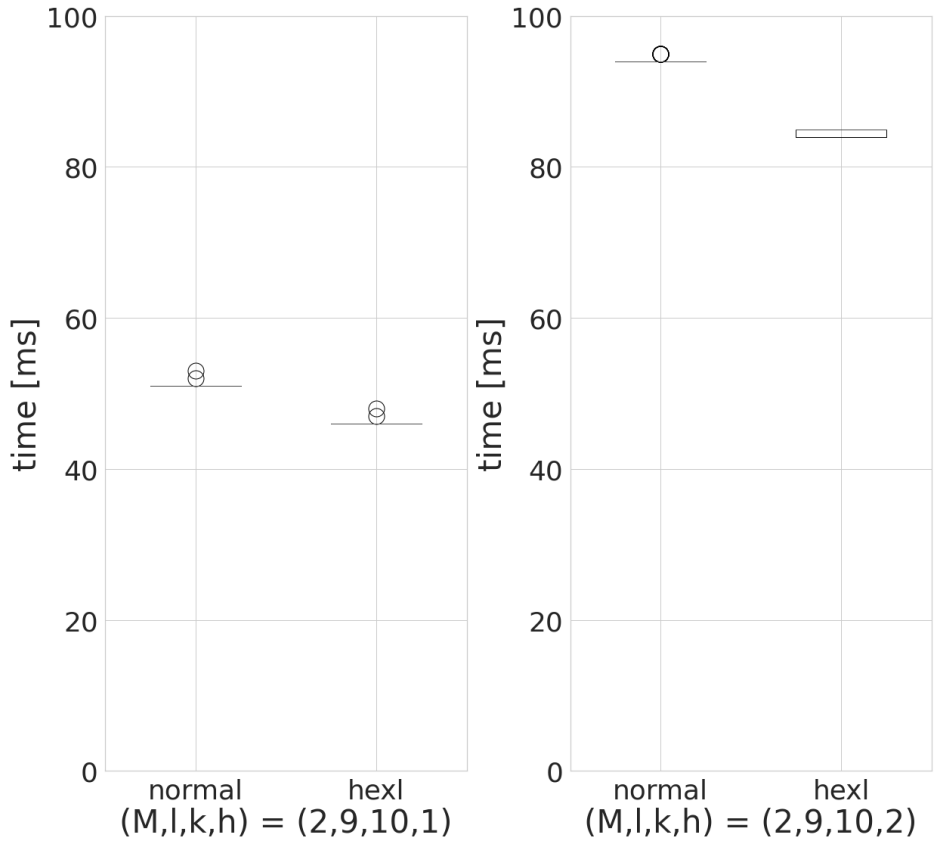

図 A.36　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=2，暗号文のレベル l=9，KeySwitch に使用する特殊な法の数 k=10）

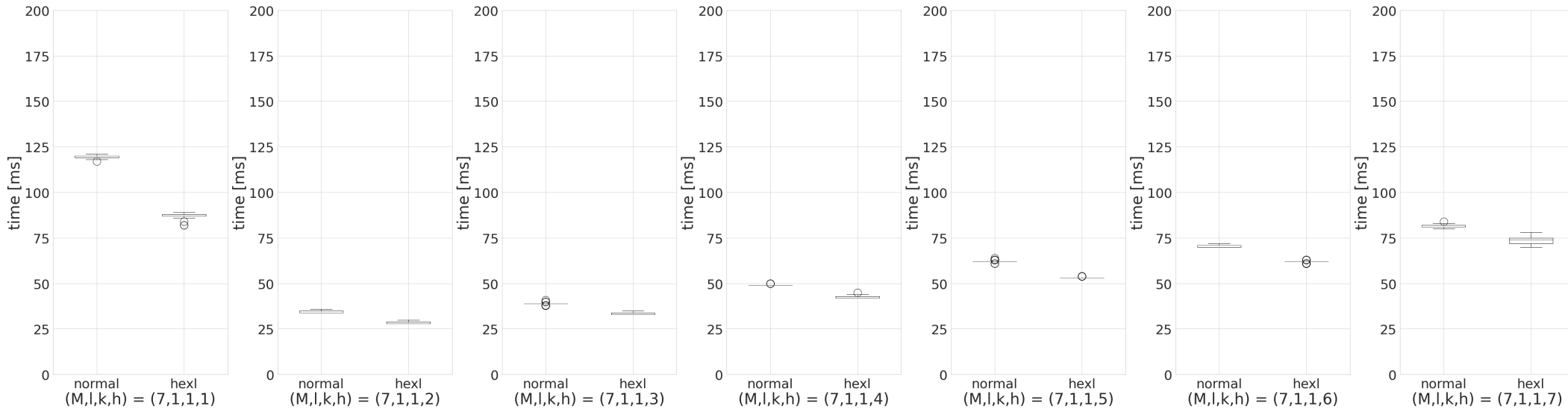

図 A.37　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=7，暗号文のレベル l=1，KeySwitch に使用する特殊な法の数 k=1）

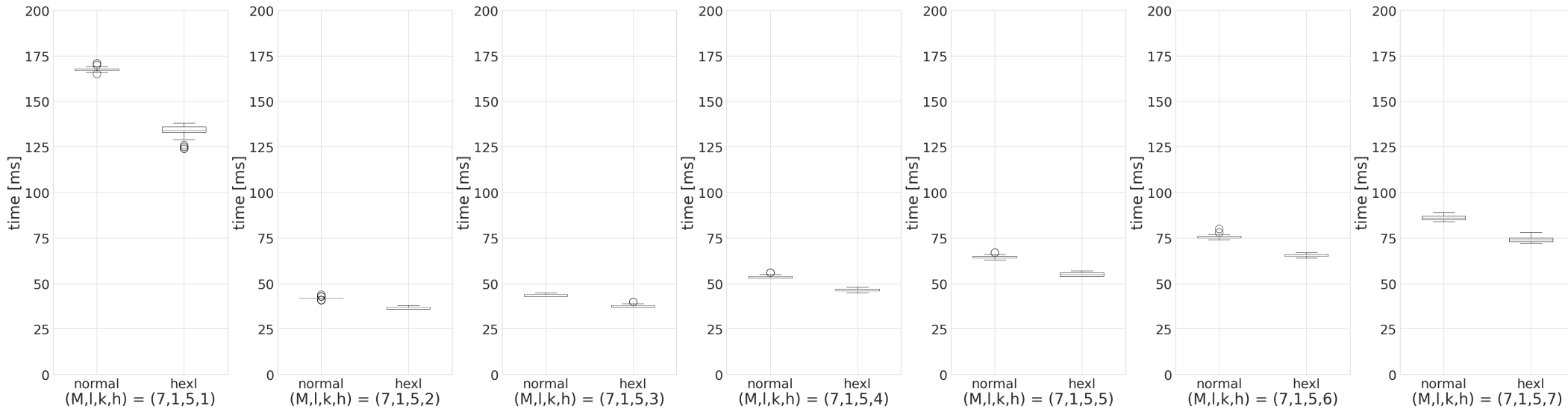

図 A.38　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=7，暗号文のレベル l=1，KeySwitch に使用する特殊な法の数 k=5）

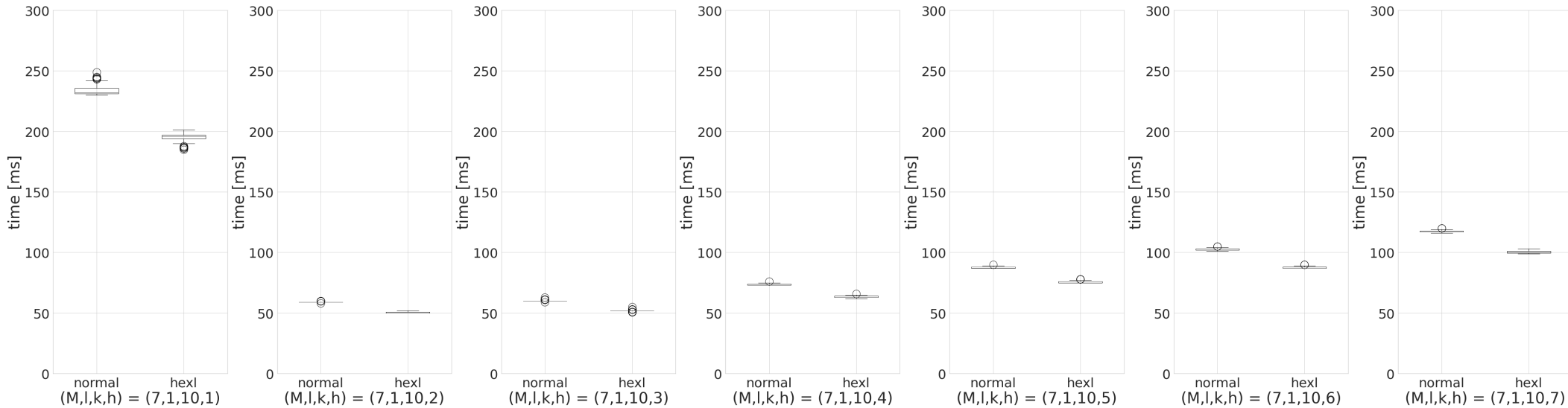

図 A.39　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=7，暗号文のレベル l=1，KeySwitch に使用する特殊な法の数 k=10）

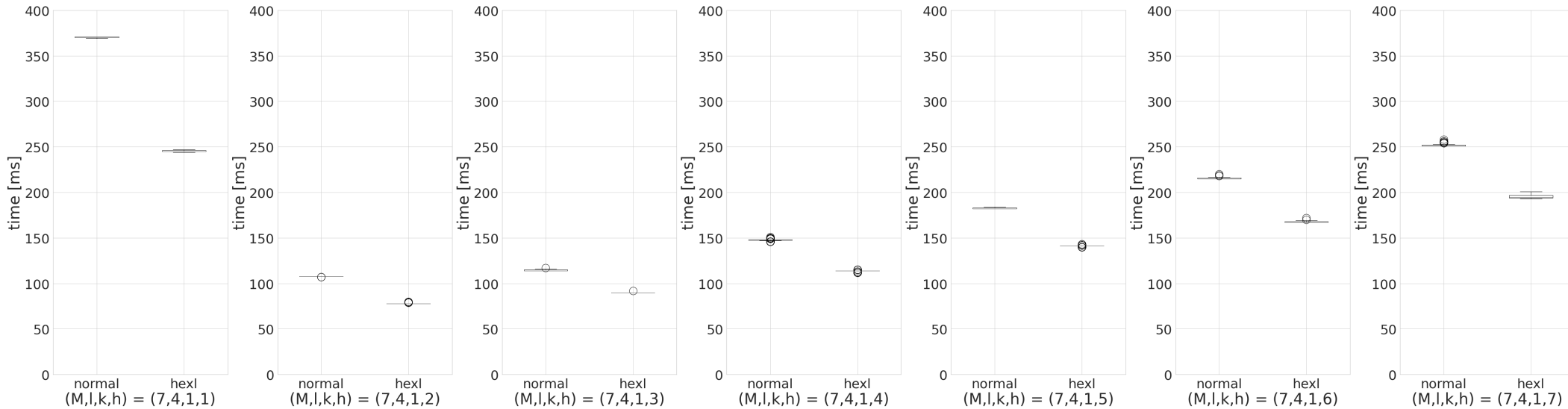

図 A.40　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=7，暗号文のレベル l=4，KeySwitch に使用する特殊な法の数 k=1）

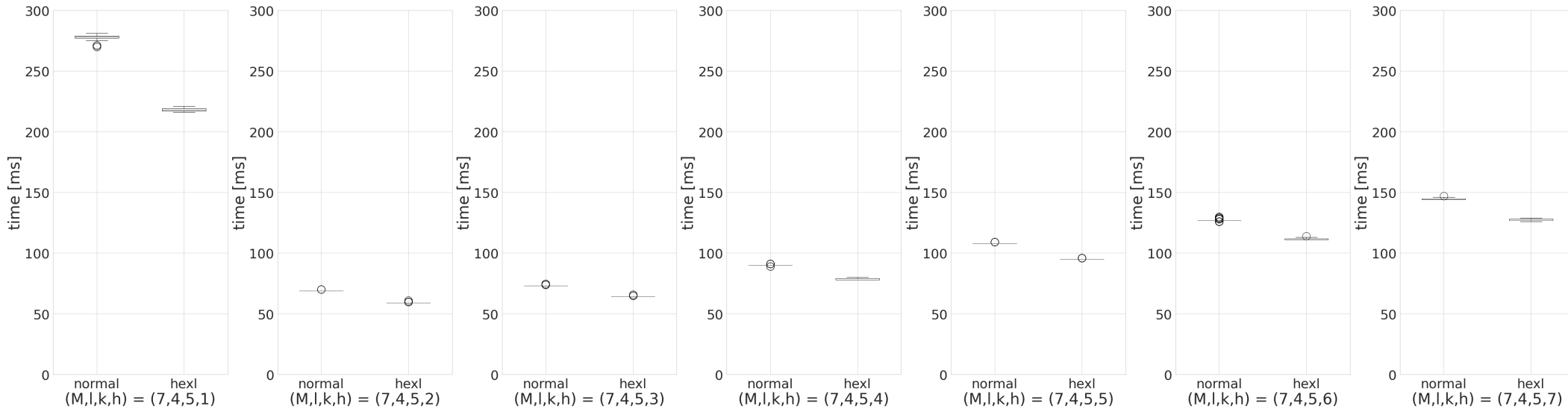

図 A.41　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=7，暗号文のレベル l=4，KeySwitch に使用する特殊な法の数 k=5）

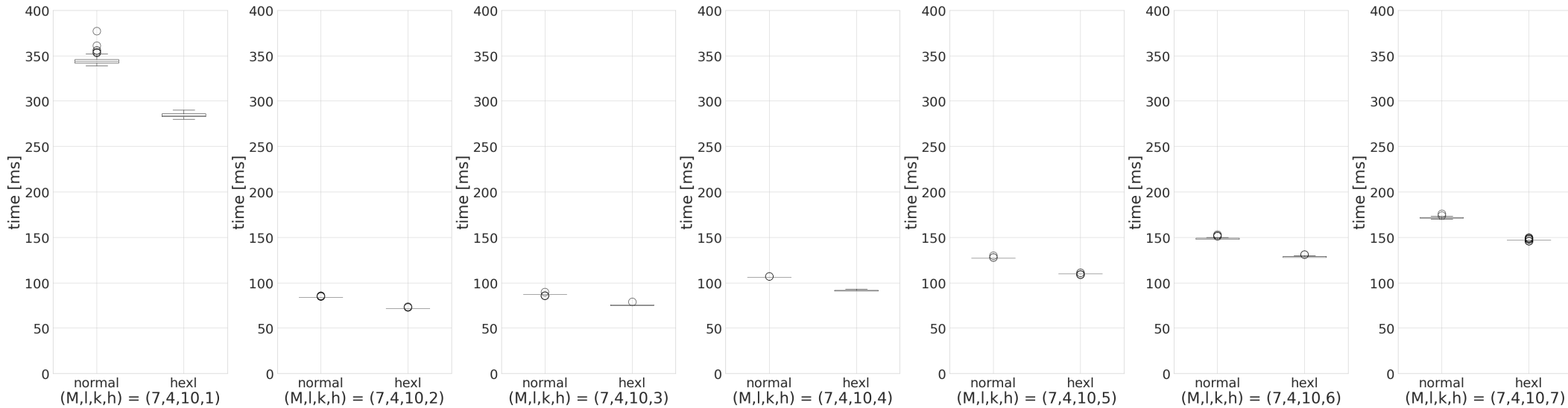

図 A.42　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=7，暗号文のレベル l=4，KeySwitch に使用する特殊な法の数 k=10）

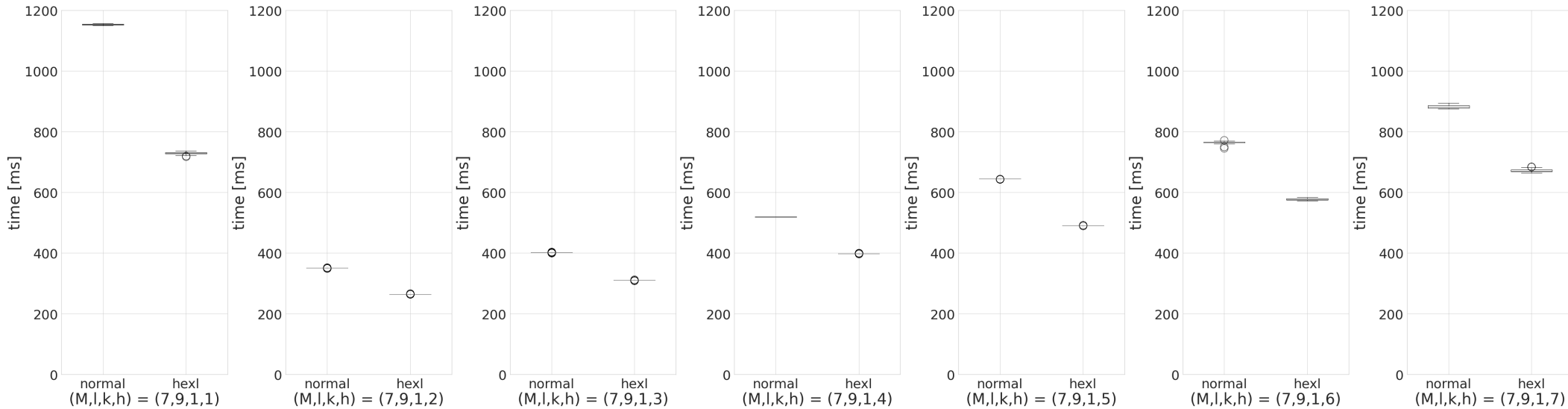

図 A.43　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8, rotate-and-sum の実行回数 M=7, 暗号文のレベル l=9, KeySwitch に使用する特殊な法の数 k=1）

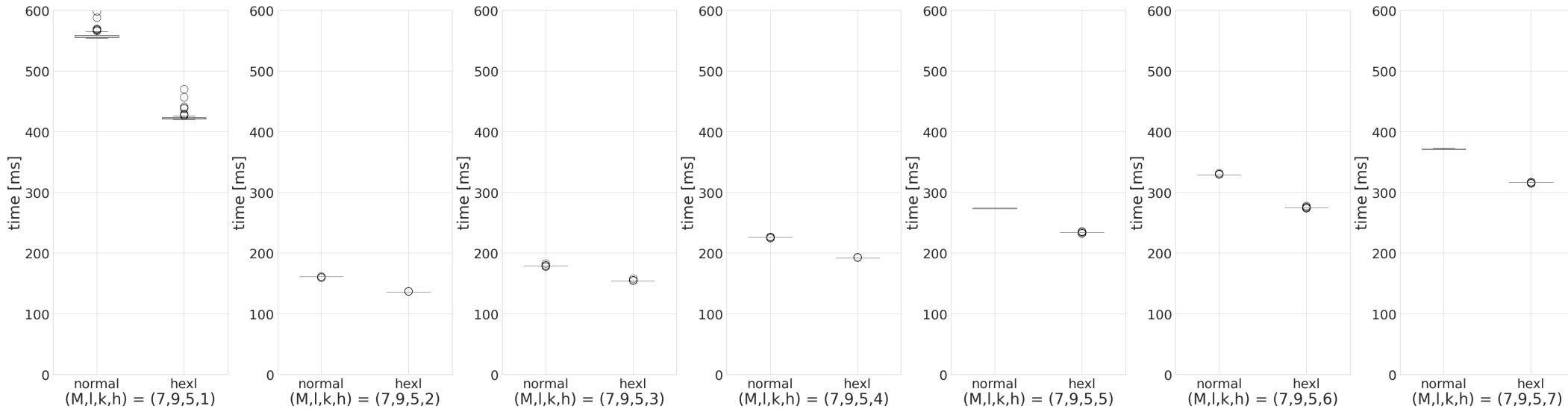

図 A.44　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=7，暗号文のレベル l=9，KeySwitch に使用する特殊な法の数 k=5）

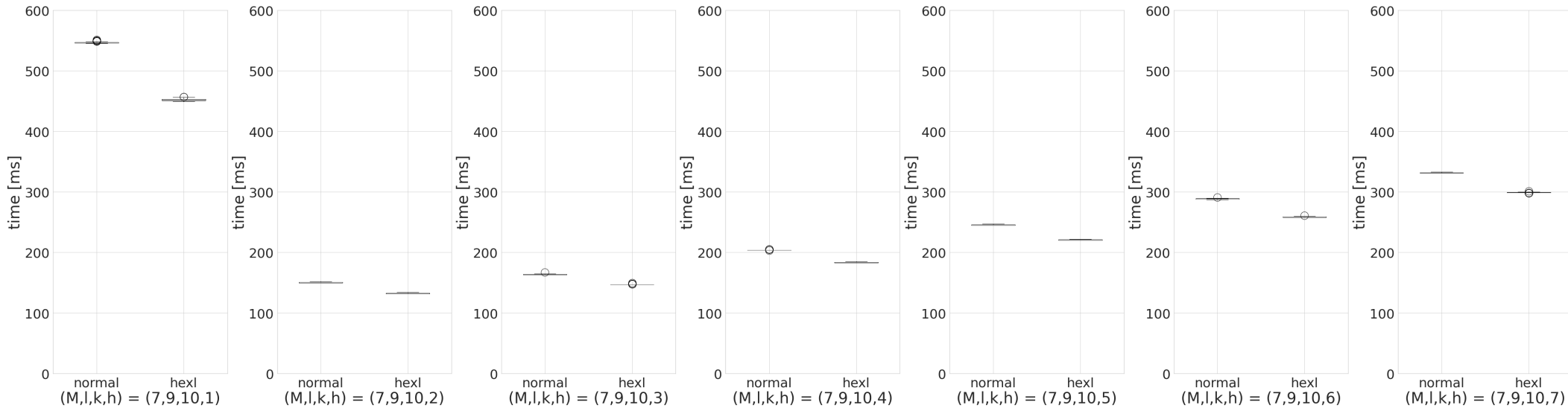

図 A.45　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8，rotate-and-sum の実行回数 M=7，暗号文のレベル l=9，KeySwitch に使用する特殊な法の数 k=10）

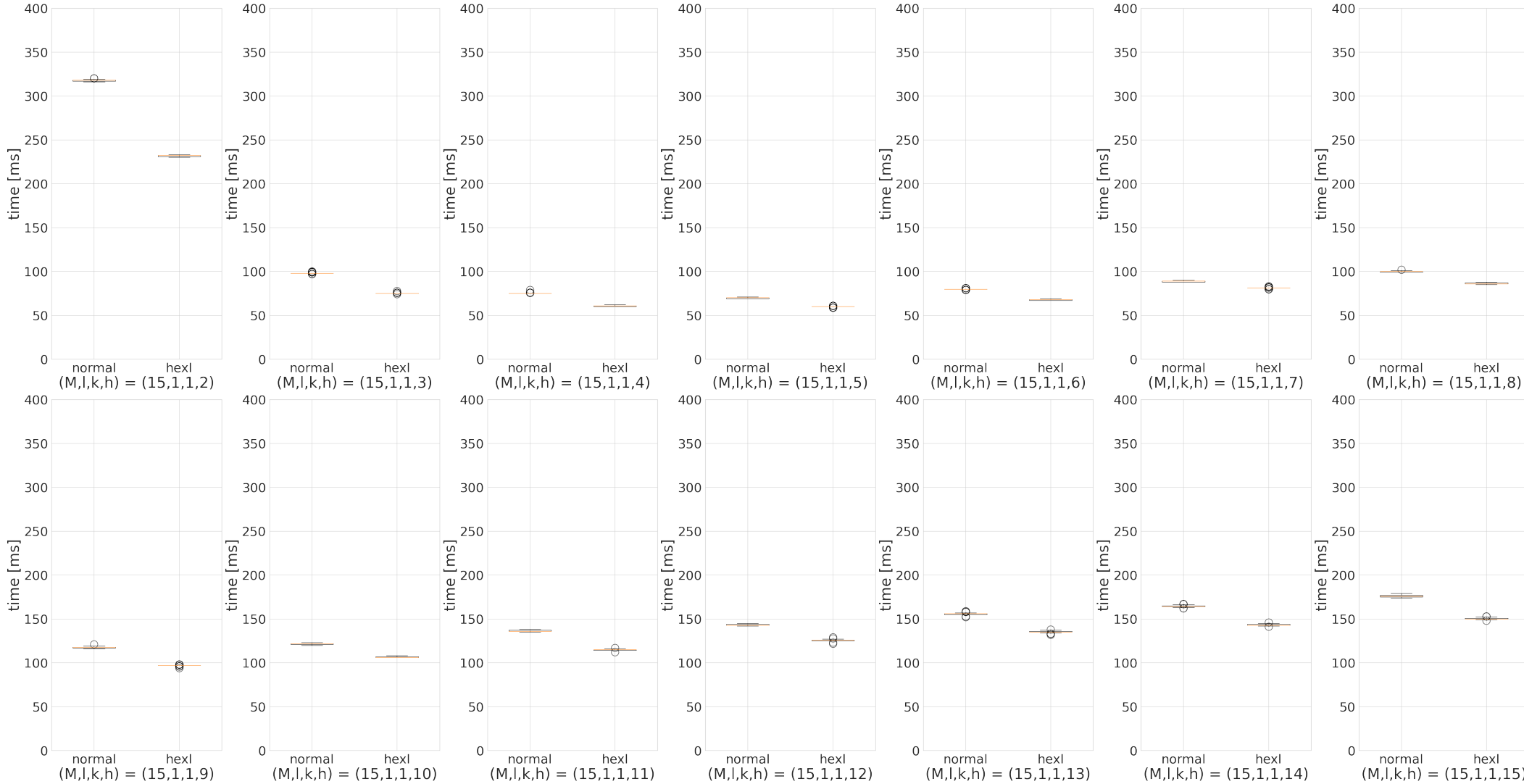

図 A.46　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=1, KeySwitch に使用する特殊な法の数 k=1）

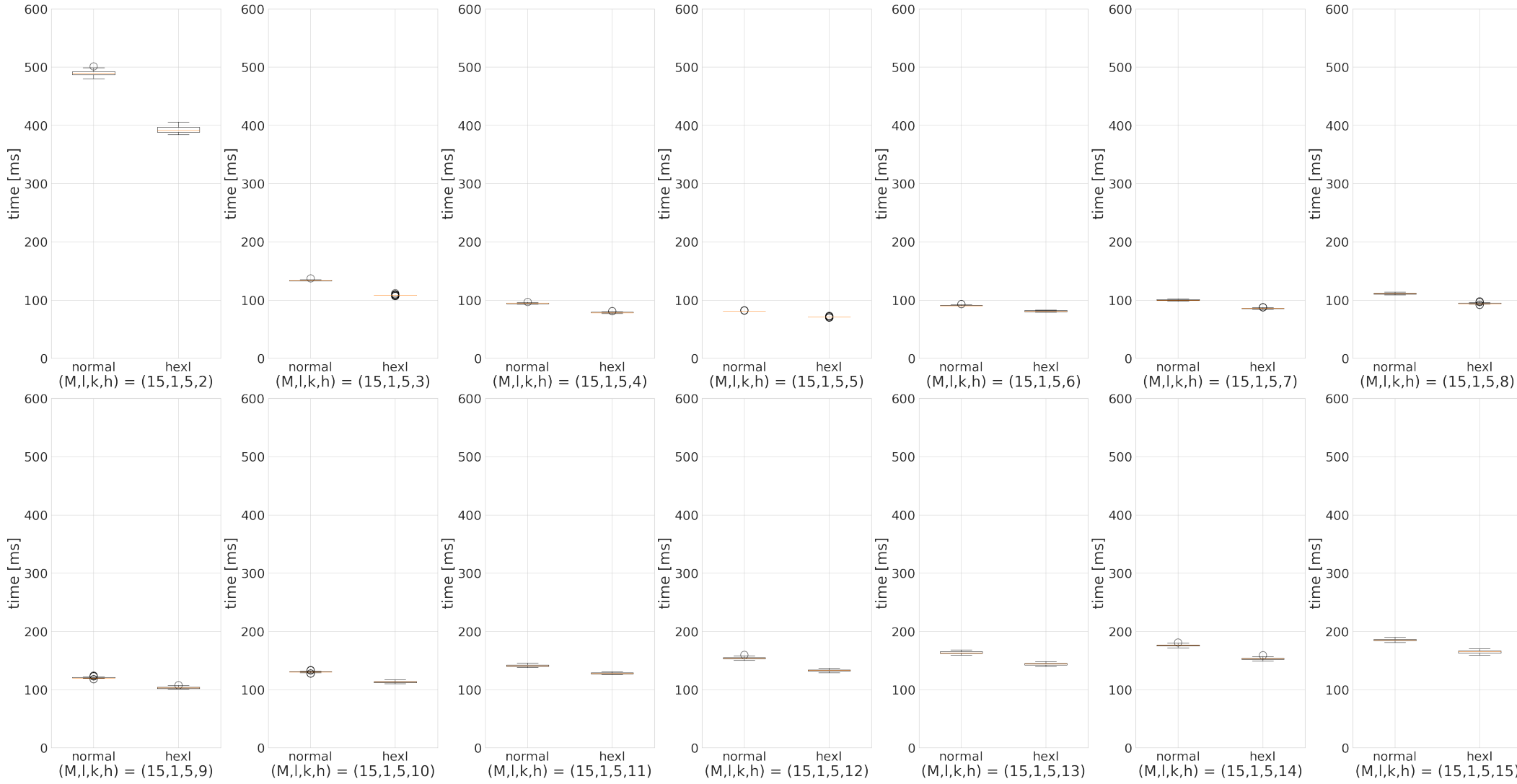

図 A.47　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=1, KeySwitch に使用する特殊な法の数 k=5）

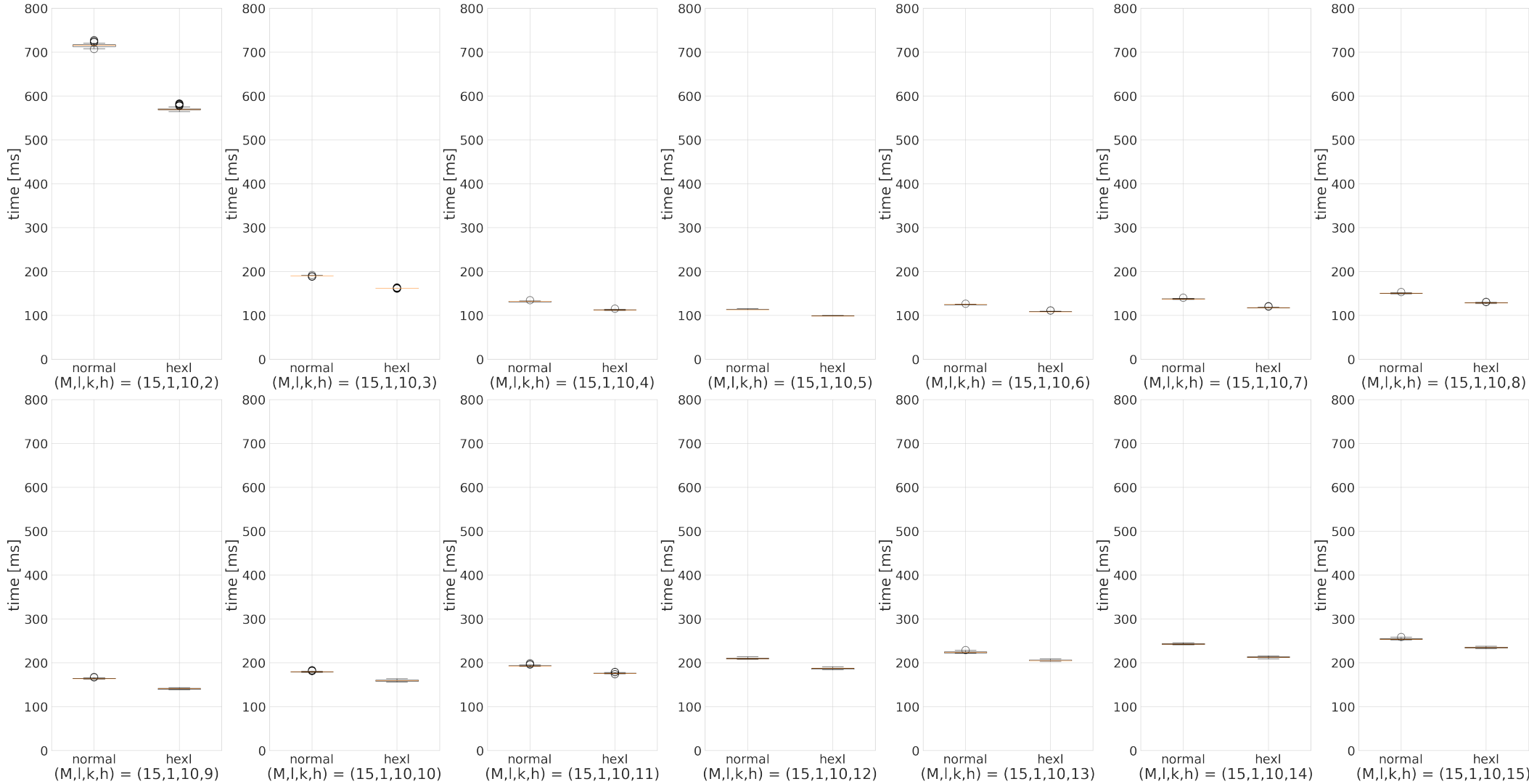

図 A.48　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=1, KeySwitch に使用する特殊な法の数 k=10）

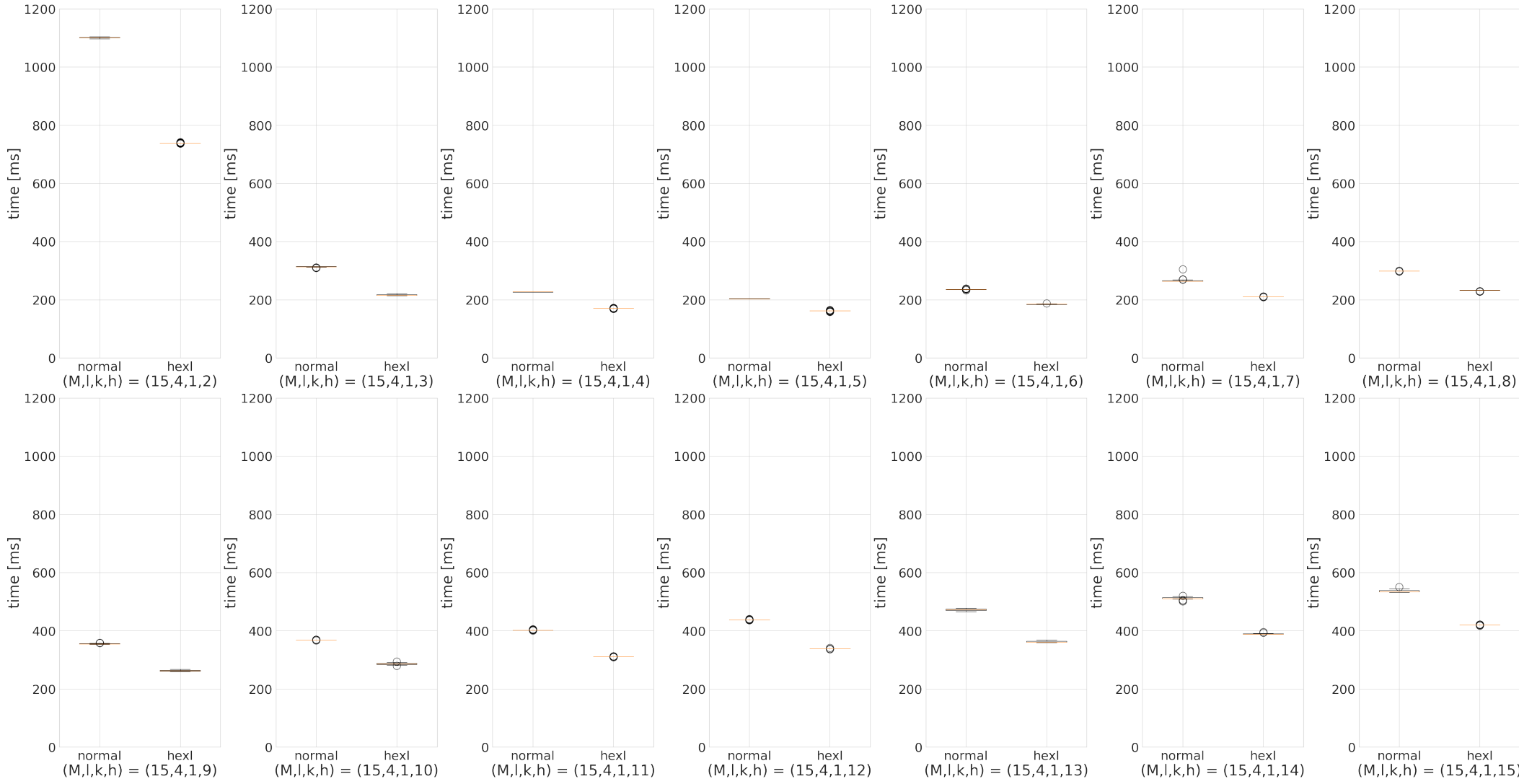

図 A.49　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=4, KeySwitch に使用する特殊な法の数 k=1）

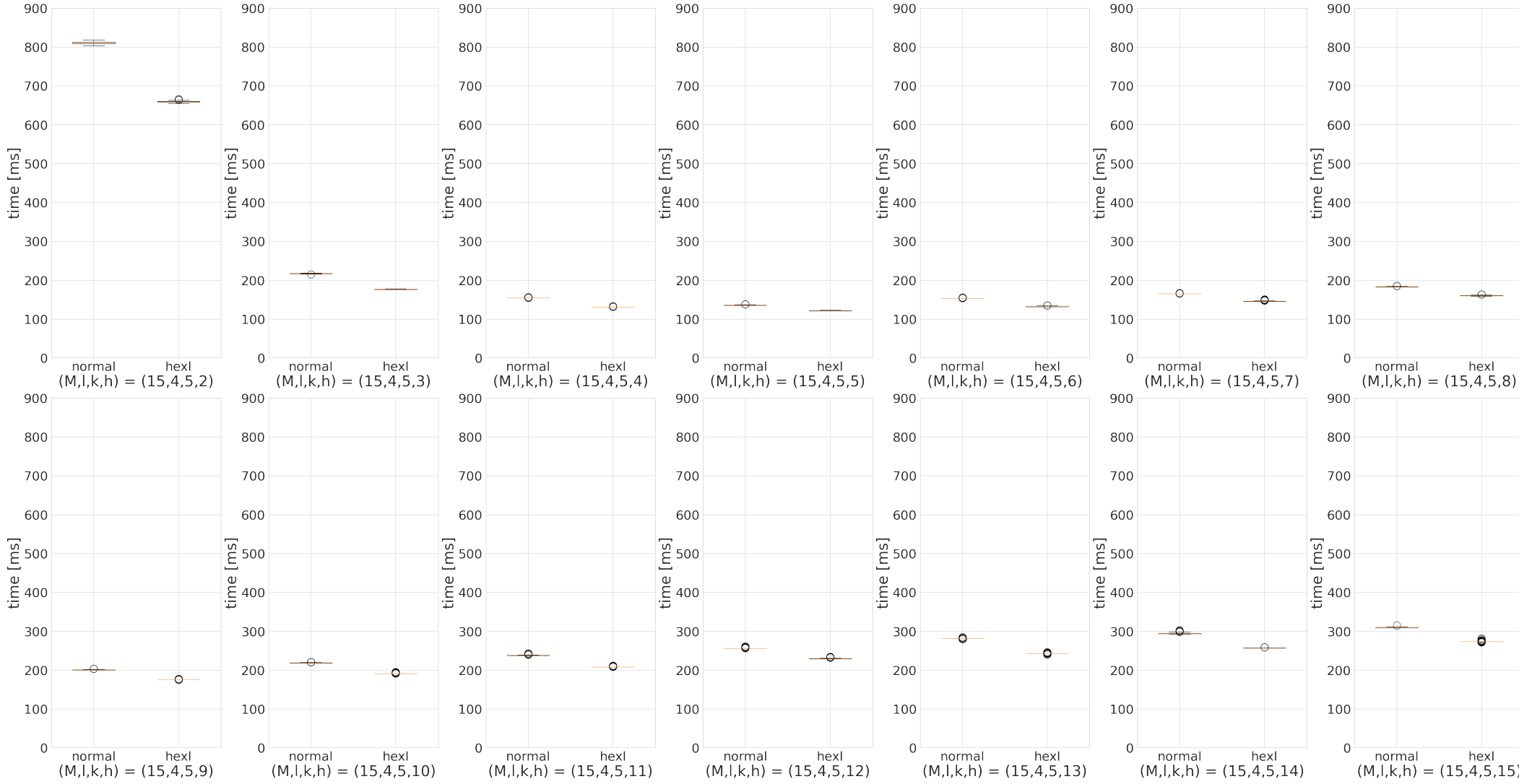

図 A.50　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=4, KeySwitch に使用する特殊な法の数 k=5）

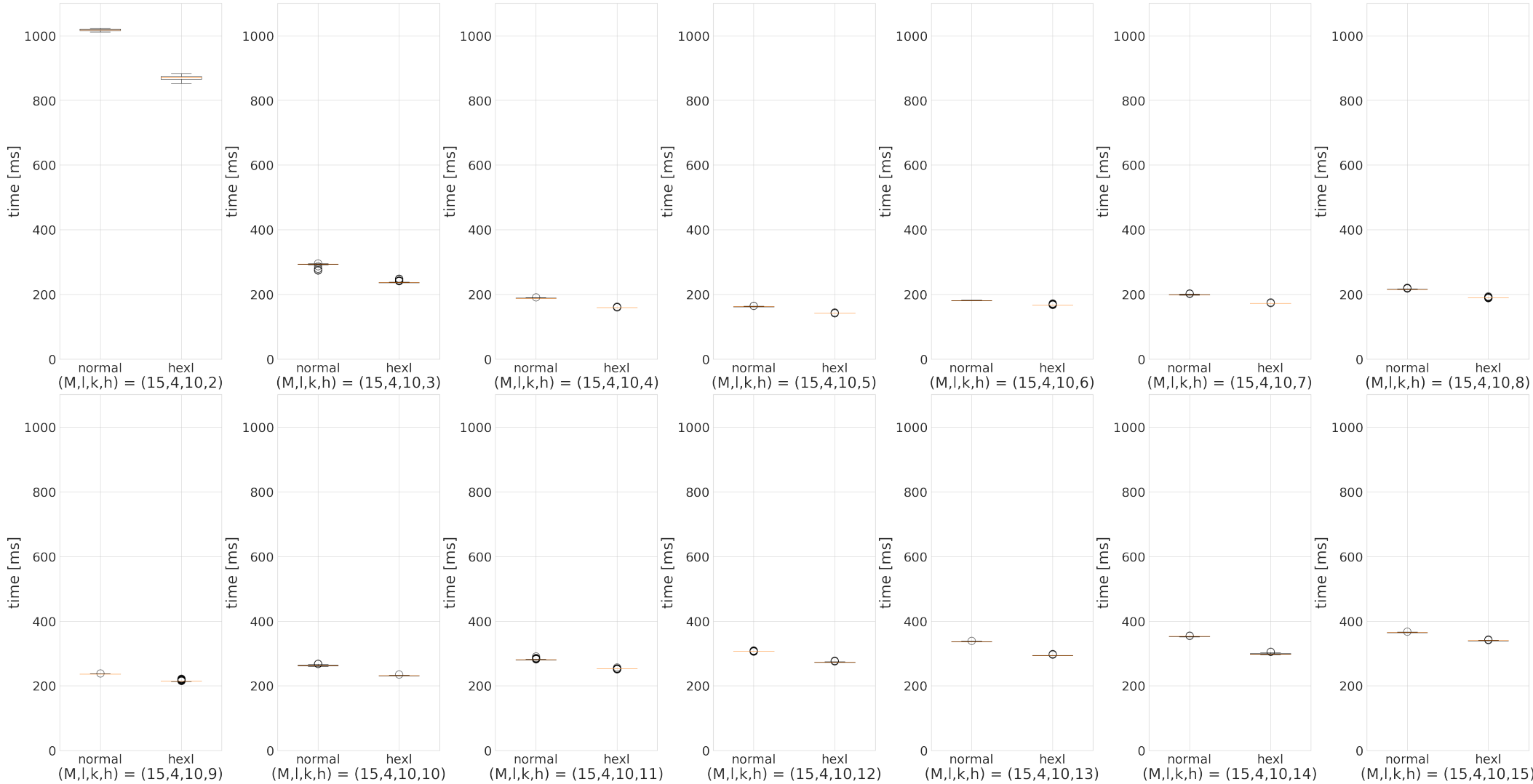

図 A.51　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=4, KeySwitch に使用する特殊な法の数 k=10）

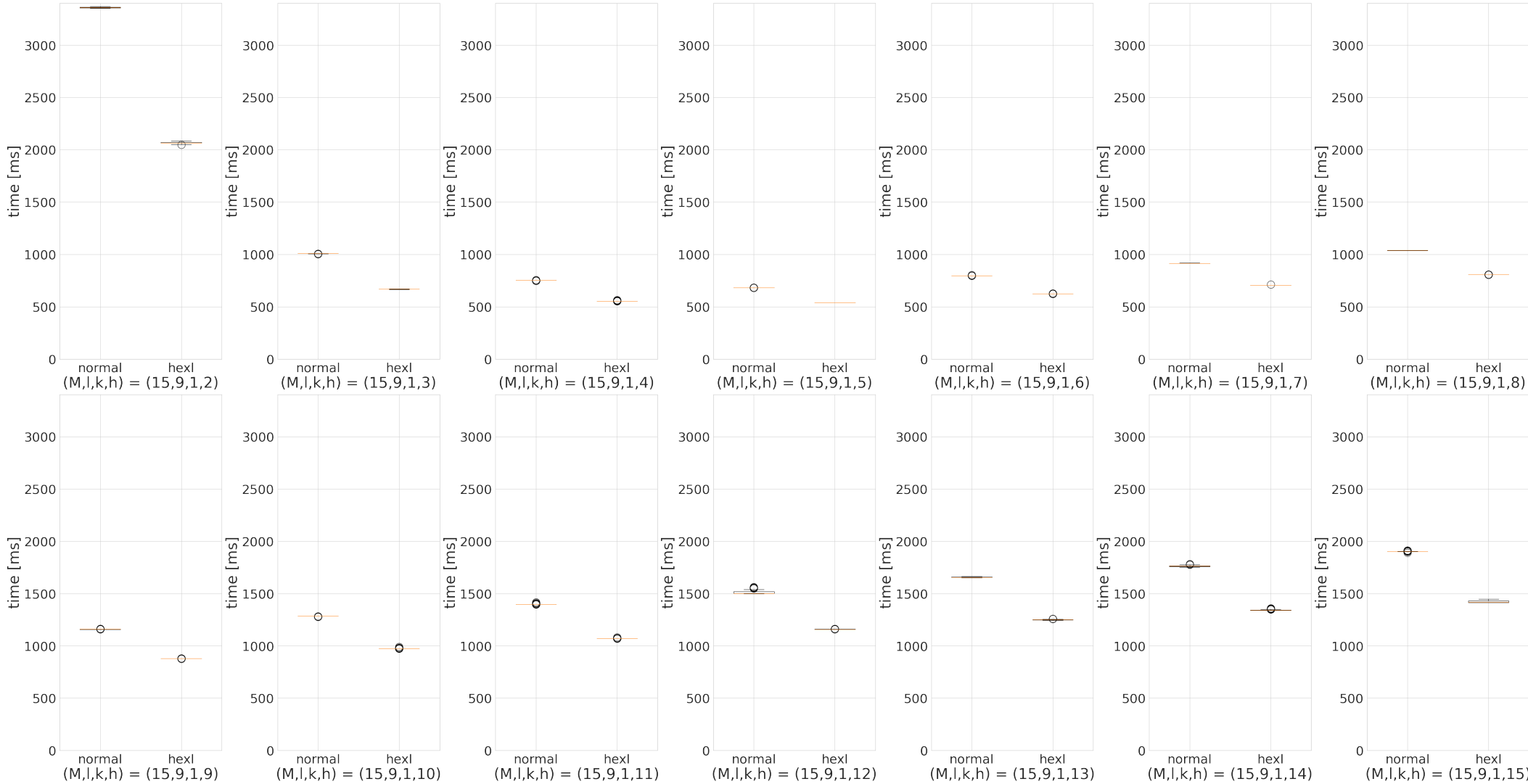

図 A.52　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=9, KeySwitch に使用する特殊な法の数 k=1）

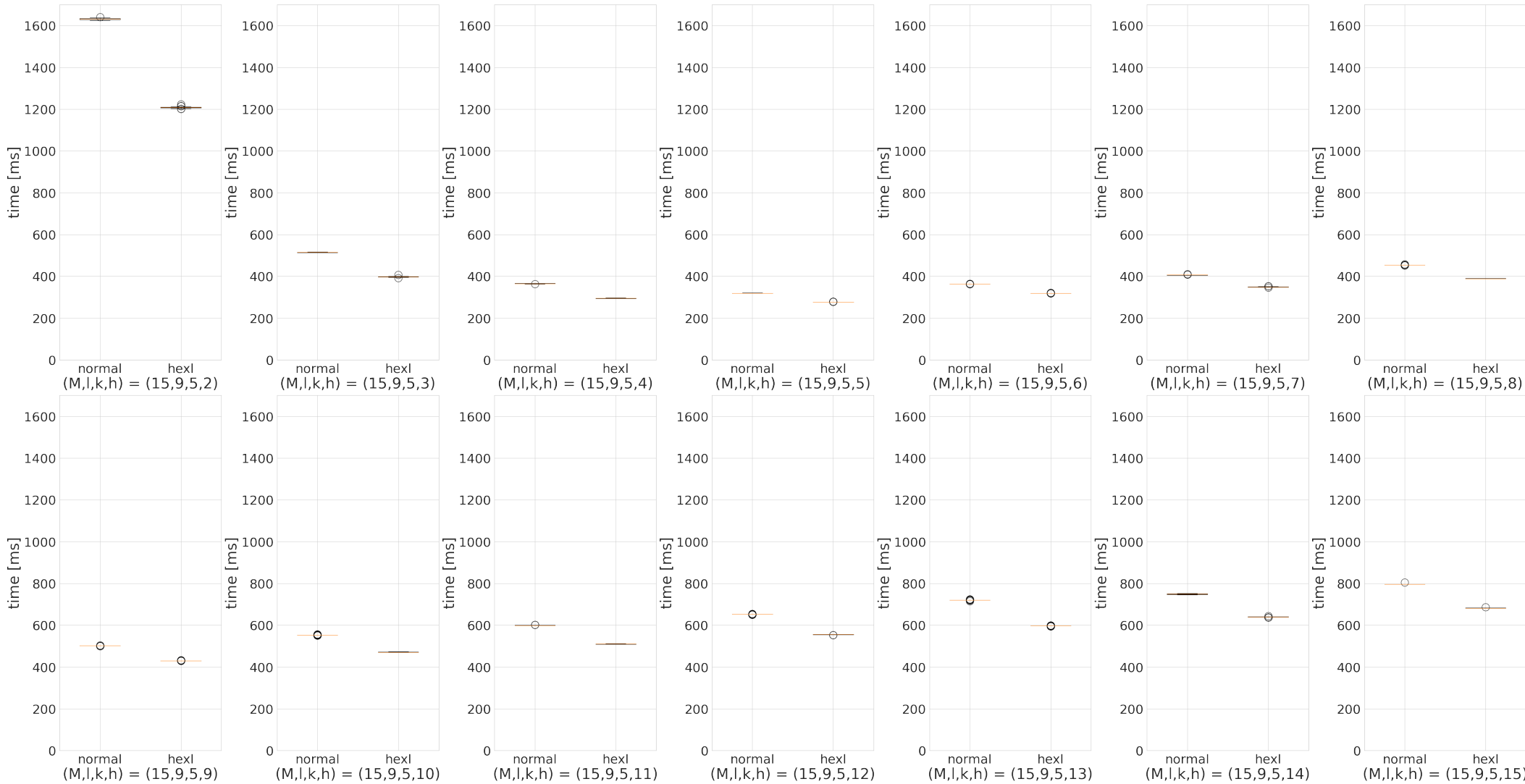

図 A.53　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=9, KeySwitch に使用する特殊な法の数 k=5）

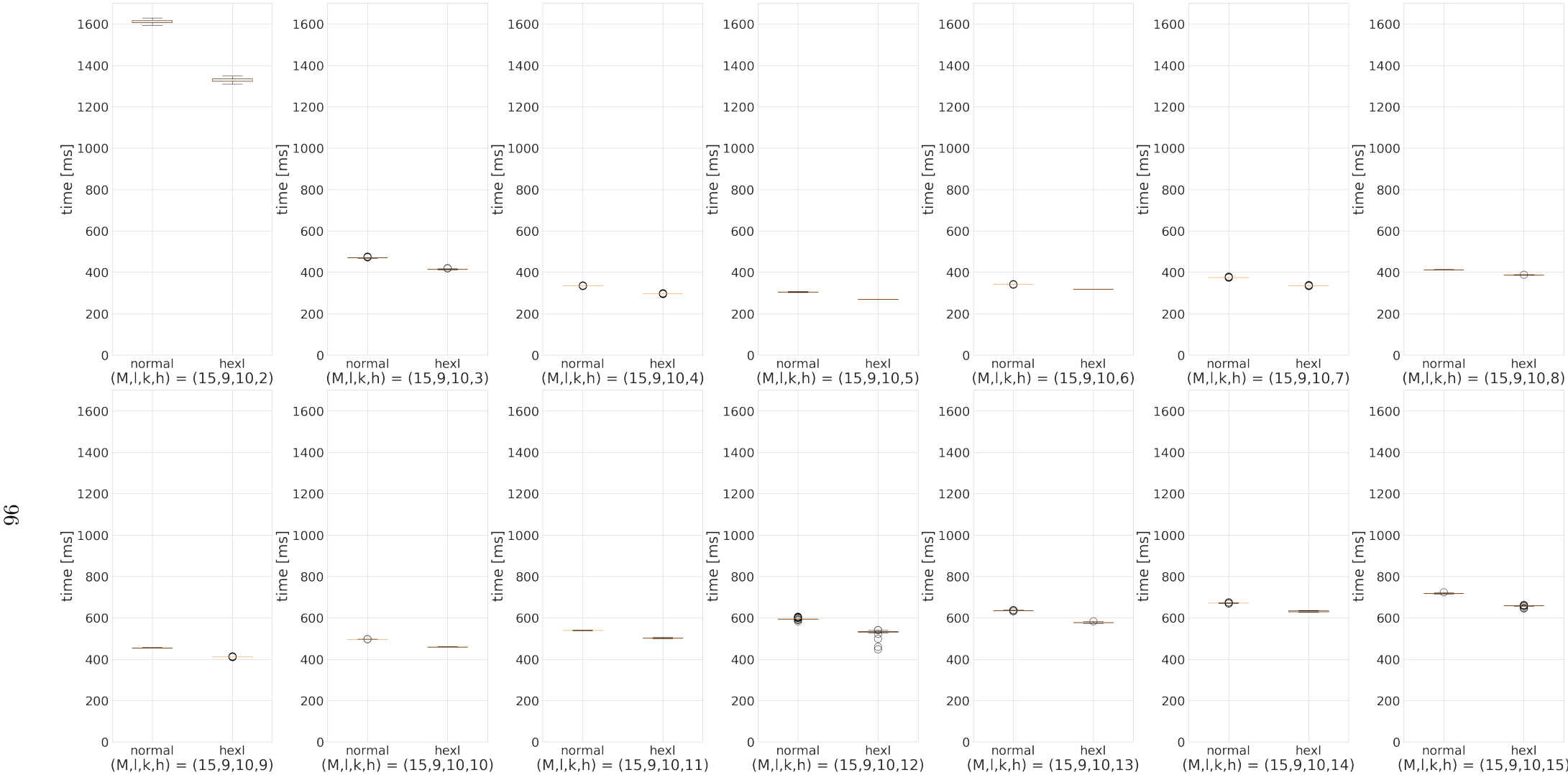

図 A.54　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間（スレッド数 8, rotate-and-sum の実行回数 M=15, 暗号文のレベル l=9, KeySwitch に使用する特殊な法の数 k=10）

研究業績

主著

1. **井上 紘太朗**，鈴木 拓也，山名 早人，”準同型暗号処理で多用される Trace-Type Function の AVX512 命令による高速化”，DEIM2022 第 14 回データ工学と情報マネジメントに関するフォーラム，(2022)，(発表予定).
2. **井上 紘太朗**，鈴木 拓也，山名 早人，“完全準同型暗号の高速化に向けたハードウェア利活用に関する研究調査”，FIT2020 第 19 回情報科学技術フォーラム，B-002，(2020).

共著

1. 牛山 翔二郎，高橋 翼，工藤 雅士，**井上 紘太朗**，鈴木 拓也，山名 早人，“完全準同型暗号下での差分プライバシ適用－レンジクエリを対象として－”，DEIM2021 第 13 回データ工学と情報マネジメントに関するフォーラム，G25-4，(2021).
2. 牛山 翔二郎，工藤 雅士，髙橋 翼，**井上 紘太朗**，鈴木 拓也，山名 早人，“差分プライバシと準同型暗号の組合せに関する研究動向調査”，CSS2020 コンピュータセキュリティシンポジウム 2020，1C5-1，(2020).