重複排除ストレージにおけるSHA-1計算のSSEによるスループット向上手法

坪内佑樹[†2]　置田真生[†1]　伊野文彦[†1]
山川聡[†3]　柏木岳彦[†4]　萩原兼一[†1]

本稿では，重複排除ストレージのためのSHA（Secure Hash Algorithm）-1計算の高速化を目的として，SSE（Streaming SIMD Extensions）命令に基づくスループット向上手法を提案する．提案手法は，異なる入力ファイルに対する処理が独立であることに着目し，SSE命令によるベクトル処理およびOpenMPによるマルチスレッド処理を併用する．実験では，単一ファイルを並列処理する既存手法と比較して1.5倍の速度向上を得ている．このときの実行効率は93%に達し，計測したスループット80 GbpsはPCI Expressの実効帯域幅を超えている．したがって，重複排除ストレージにおける性能ボトルネックを除去できていると考える．

An SSE-Based Method for Increasing Throughput of SHA-1 Computation for Deduplication Storage

Yuki Tsubouchi,[†1] Masao Okita,[†1] Fumihiko Ino,[†1]
Satoshi Yamakawa,[†3] Takehiko Kashiwagi[†4]
and Kenichi Hagihara[†1]

This paper presents a streaming SIMD extensions (SSE)-based method for increasing the throughput of secure hash algorithm (SHA)-1 computation for deduplicated storage systems. Our method exploits the data independency between different input files, realizing SSE-based vectorization and OpenMP-based multithreaded execution. In experiments, we achieve a speedup of 1.5 times over a previous method that parallelizes computation for a single file. This leads to an efficiency of 93% and the measured throughput reaches 80 Gbps, which is higher than the effective bandwidth of the PCI Express bus. Thus, we think that our method eliminates a performance bottleneck of deduplicated storage systems.

1. はじめに

重複排除ストレージ（DS：Deduplication Storage）[1]とは，ストレージシステムにおけるデータ格納プロセスの前段に，既に格納済みのデータとの重複の有無を判定する重複排除機能を付与したものである．書き込み要求とともに転送されたデータについて，重複の有無を判定し，非重複データのみをストレージデバイスに書き込むことにより、デバイス容量を削減できる．例えば，ファイルサーバなどのバックアップに応用されている．

重複の有無はファイルを分割したブロックごとに判定されている．ブロックごとにハッシュ値を計算しておけば，ブロック全体の代わりにハッシュ値を比較することにより重複判定を高速化できる．代表的なハッシュ関数としては，SHA（Secure Hash Algorithm）-1[2]が挙げられる．近年，PCI Expressなどの入出力バスにおける帯域幅の増加やSSD（Solid State Drive）などのストレージデバイスの高速化により，SHA-1計算がDSの性能ボトルネックとなりつつある．

SHA-1計算の高速化を目的として，GPU（Graphics Processing Unit）[3]などのアクセラレータやCPUのベクトル命令SSE（Streaming SIMD Extensions）[4]を用いる既存研究がある．GPUはグラフィクス処理のためのアクセラレータであり，CPUと比較して10倍ほど高いメモリ帯域幅を持つ．望月ら[5]は，GeForce GTX 580上において300 Gbpsのスループットを達成している．しかし，DSでは入出力バスを経てGPUへデータを転送する必要があり，さらに同一ブロックに対してSHA-1計算を反復することはないため，SHA-1計算のスループットはPCI Expressの実効帯域幅（約45 Gbps）で抑えられる．一方，Locktyukhin[6]はSSEによる高速化手法を提案している．この手法は主記憶上のデータを直接操作でき，データ転送に起因するオーバヘッドはない．1個のブロックに対するSHA-1計算を1つのタスクとみなし，単一タスク内における算術演算命令の一部をSSEに

†1 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University
†2 大阪大学基礎工学部情報科学科
Department of Information and Computer Sciences, School of Engineering Science, Osaka University
†3 日本電気株式会社システムプラットフォーム研究所
System Platform Laboratory, NEC Corporation
†4 日本電気株式会社 IT ソフトウェア事業本部
IT Software Division, NEC Corporation

ⓒ 2012 Information Processing Society of Japan

よりベクトル処理している．しかし，60%の命令にフロー依存があり，その速度向上率は 1.4 倍に留まる．

そこで本研究では，大量のブロックに対する SHA-1 計算のスループット向上を目指して，複数のブロックをまとめて SSE および OpenMP により並列処理する手法を提案する．具体的には，ブロックごとの処理が同一の算術演算命令を同一の手順で適用する点に着目し，それらを SSE によりベクトル処理する．この工夫により，ベクトル化の対象となる命令の数を増やせる．また，その実行効率を高めるために，主記憶に対するストライドアクセスを回避できるようにメモリ領域を確保する．さらに，マルチコア CPU の性能を引き出すために，OpenMP[7] によるマルチスレッド化を施す．

以降では，まず 2 章で重複排除処理を簡潔に紹介し，3 章で SHA-1 の概要および既存の高速化手法を示す．次に，4 章で提案手法について述べ，5 章で評価実験の結果を示す．最後に，6 章で本稿をまとめ，今後の課題を述べる．

2. 重複排除処理

図 1 に重複排除処理の仕組みを示す．図 1 の赤い長方形は重複するブロックを表す．重複排除処理は n 個のファイル $F_1, F_2, \ldots, F_n$ を入力として，これらのファイルから N $(> n)$ 個のブロックを生成し，$(1-R)N$ 個の非重複ブロックを出力する．ここで，R はデータの重複率を表し，重複するブロック数を T として $R = T/S$ である．

データの重複はファイルではなく，それらを分割したブロックごとに検出する．これにより重複率 R を高め，格納すべきデータサイズを削減できる．一般的にブロックサイズは 4〜512 KB である．

典型的なバックアップ環境では，重複率 R は 95%に達する．この場合，格納すべきブロックの数 $S-T$ は小さく，重複排除のためのハッシュ値計算が DS の性能ボトルネックである．逆に重複率 R が低い場合，$S-T$ は大きい．この場合，ストレージデバイスへの書き込みが性能ボトルネックとなる．

3. Secure Hash Algorithm（SHA）-1 の概要および既存の高速化手法

SHA-1[2] は米国政府の標準ハッシュ関数として採用されている．SHA-1 関数 H は 2^{64} ビット未満のビット列を，160 ビットのビット列，すなわちハッシュ値に変換する．

$$H : \{0,1\}^{2^{64}-1} \rightarrow \{0,1\}^{160} \qquad (1)$$

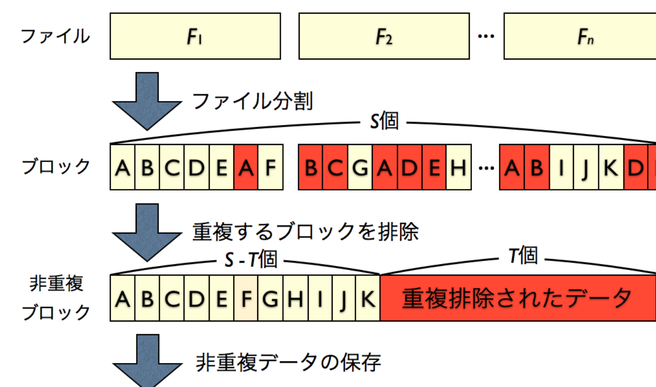


図 1 重複排除処理の仕組み

ここで，$\{0,1\}^{2^{64}-1}$ は $2^{64}-1$ ビットのビット列であり，$\{0,1\}^{160}$ は 160 ビットのビット列である．

3.1 ハッシュ値の計算手順

まず，入力として与えられるブロックのサイズが 512 の倍数になるように，ブロックにパディングを施す．M を自然数として，パディング後におけるブロックのビット長は $512M$ で表せる．その後，ブロックを 512 ビットごとのチャンクに分割し，先頭のチャンクから順に圧縮処理を施す．圧縮処理は，処理の過程において 80 個の 32 ビットワードの並び（拡張チャンク）を生成し，最後に 5 個の 32 ビットワード（160 ビットのビット列）を出力する．以降では，ブロックおよびチャンクの順番を区別するために，i 番目のブロック B_i における j $(0 \leq j \leq M-1)$ 番目のチャンクを $C_{i,j}$ とし，$C_{i,j}$ における k $(0 \leq k \leq 79)$ 番目の拡張チャンクを $W_{i,j,k}$ とする．

図 2 にハッシュ値の計算手順を示す．チャンク $C_{i,j}$ に対する圧縮処理は，以下のステップ（1）〜（4）を含む．これらのステップを先頭のチャンクから順に（$j=0$ から $M-1$ まで）適用し，最後に得た一時ハッシュ値が SHA-1 の出力となる．

（1） チャンクの分割：チャンク $C_{i,j}$ を 16 個のワード $W_{i,j,0}, W_{i,j,1}, \ldots, W_{i,j,15}$ に分割する．

（2） チャンクの拡張：式 (2) に基づいて，16 個のワードから新たに 64 個のワード $W_{i,j,16}, W_{i,j,17}, \ldots, W_{i,j,79}$ を生成する．

ⓒ 2012 Information Processing Society of Japan

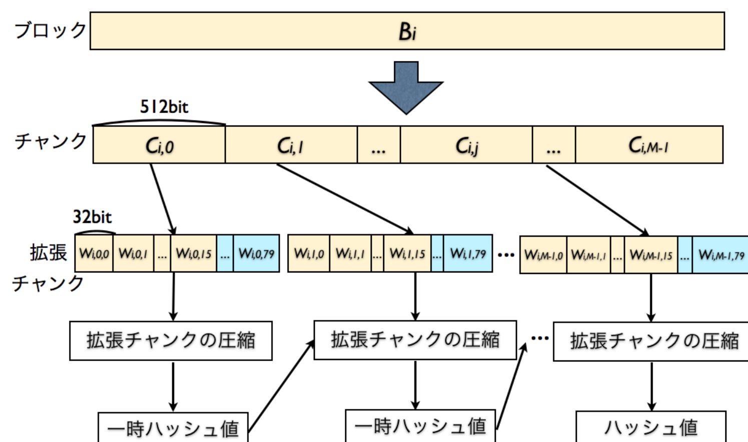


図 2 ハッシュ値の計算手順

$$W_{i,j,k} = (W_{i,j,k-3} \oplus W_{i,j,k-6} \oplus W_{i,j,k-14} \oplus W_{i,j,k-16}) \text{ leftrotate } 1 \quad (2)$$

ここで，leftrotate は左巡回シフトを表し，$16 \leq k \leq 79$ である．

(3) 一時ハッシュ値の初期化：直前のチャンク $C_{i,j-1}$ で得た一時ハッシュ値を参照して一時ハッシュ値を初期化する．$C_{i,j}$ が最初のチャンクならば，あらかじめ定められた値[2] を一時ハッシュ値とする．

(4) 拡張チャンクの圧縮：ワード $W_{i,j,k}$ を先頭から順に（$k = 0$ から 79 まで）参照し，3 種類の論理関数を使用して拡張チャンクを圧縮する．論理関数の詳細については文献 2) を確認されたい．圧縮して得た値を一時ハッシュ値として出力する．

なお，ステップ（3）においてチャンク $C_{i,j-1}$ から $C_{i,j}$ へのフロー依存が存在する．したがって，複数のチャンクを並列処理することはできない．しかし，ブロックごとの処理は互いに独立であるため，複数のブロックを並列処理することは可能である．

表 1 に 1 個のチャンクを処理するために必要な算術演算命令の内訳を示す．チャンク 1 個あたり 976 個の算術演算命令が必要である．したがって，ブロックのサイズ S が 64 KB の場合，1 ブロックあたりの算術演算命令数 $A = 976S/(512/8)$ は 999,424 個となる．

3.2 既存の高速化手法

1 章で述べた通り，Locktyukhin[6] はベクトル処理による SHA-1 の高速化手法を提案している．具体的には，ステップ（2）におけるワードの生成を SSE によりベクトル処理し，

表 1 1 個のチャンクを処理するために必要な算術演算命令の数

演算命令	ADD	AND	OR	XOR	ROTATE	合計
チャンクの拡張	0	0	0	192	64	256
拡張チャンクの圧縮（$0 \leq k \leq 19$）	80	20	0	40	40	180
拡張チャンクの圧縮（$20 \leq k \leq 39$）	80	0	0	40	40	160
拡張チャンクの圧縮（$40 \leq k \leq 59$）	80	60	40	0	40	220
拡張チャンクの圧縮（$60 \leq k \leq 79$）	80	0	0	40	40	160
合計	320	80	40	312	224	976

単一ブロックに対する SHA-1 計算を高速化する．

$W_{i,j,k}$〜$W_{i,j,k+3}$ をベクトル処理するために，式 (2) を以下のように書き換える．

$$W_{i,j,k} = (W_{i,j,k-3} \oplus W_{i,j,k-8} \oplus W_{i,j,k-14} \oplus W_{i,j,k-16}) \text{ leftrotate } 1 \quad (3)$$
$$W_{i,j,k+1} = (W_{i,j,k-2} \oplus W_{i,j,k-7} \oplus W_{i,j,k-13} \oplus W_{i,j,k-15}) \text{ leftrotate } 1 \quad (4)$$
$$W_{i,j,k+2} = (W_{i,j,k-1} \oplus W_{i,j,k-6} \oplus W_{i,j,k-12} \oplus W_{i,j,k-14}) \text{ leftrotate } 1 \quad (5)$$
$$W_{i,j,k+3} = (0 \oplus W_{i,j,k-5} \oplus W_{i,j,k-11} \oplus W_{i,j,k-13}) \text{ leftrotate } 1 \quad (6)$$
$$W_{i,j,k+3} = (W_{i,j,k} \oplus W_{i,j,k+3}) \text{ leftrotate } 1 \quad (7)$$

式 (3)〜式 (6) は 1 個のベクトル命令で処理でき，式 (7) は 1 個のスカラ命令を用いて処理できる．さらに $k \geq 32$ に対しては，式 (2) を次のように書き換えることにおり，式 (7) のためのスカラ命令を除去できる．

$$W_{i,j,k} = (W_{i,j,k-6} \oplus W_{i,j,k-16} \oplus W_{i,j,k-28} \oplus W_{i,j,k-32}) \text{ leftrotate } 2 \quad (8)$$

上記のベクトル処理により，チャンクの圧縮処理のための算術演算命令を約 18%削減できる．

4. 提 案 手 法

提案手法を示す前に，CPU 上で高速化を図る理由について述べる．そのために，CPU における SHA-1 計算のピークスループットを分析する．

まず，CPU のピーク性能を秒間あたりの算術演算回数 P で与える．

$$P = fvcs \quad (9)$$

ここで，f，v，c および s はそれぞれ CPU のクロック周波数，1 クロックあたりに実行可能な算術演算命令数，コア数およびソケット数を表す．評価実験で用いた計算機（Xeon X5677 CPU）では，SSE を用いて $f = 3.47 \times 10^9$，$v = 4$，$c = 6$ および $s = 2$ である．ここで，$v = 4$ とした理由は，多くの SSE 命令が 1 命令/サイクルの演算スループットを持つためである[8]．また，CPU および GPU のうち，どちらをシステムに追加すべきかを議論するために，$s = 2$ とした．結果，$P = 166.56 \times 10^9$ となり，この値は 166.56 GIPS

ⓒ 2012 Information Processing Society of Japan

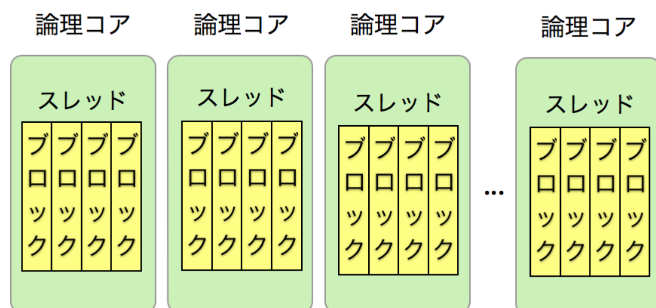


図 3 提案手法におけるタスクの割り当て

(Giga Instructions Per Second)に相当する．

次に，SHA-1 計算のピークスループット T は次式で計算できる．

$$T = S/(A/P) \tag{10}$$

3.1 節の $A = 999,424$ および $S = 65,536$ を使えば，$T = 1.07 \times 10^7$ となる．この値は 85.6 Gbps に相当する．

したがって，マルチコア CPU における実行効率が 53%を超えれば，そのスループットは CPU・GPU 間の実効帯域幅 45 Gbps を上回る．ゆえに，本研究では CPU による高速化手法を提案する．なお，CPU・GPU 間の実効帯域幅は CUDA SDK[9] を用いて計測した．

4.1 工夫 1：マルチスレッド処理によるコアの活用

図 3 に，提案手法におけるスレッドへのタスク割り当てを示す．マルチコア CPU のコアをすべて活用するために，OpenMP[7] によるマルチスレッド処理を実現する．そのために，各コアに 1 個のスレッドを割り当て，各スレッドが 4 個のブロックをベクトル処理する．

さらに，CPU が提供するハイパースレッディング（HT：Hyper-Threading）技術により，各論理コアに 1 個のスレッドを担当させる．したがって，X 個の物理コアをもつ CPU は $2X$ 個のスレッドを並行処理する．SSE によるベクトル処理と合わせて，ソケットあたり $8X$ 個のブロックを並行処理できる．

なお，効率のよいメモリアクセスを実現するために，提案手法はプロセッサアフィニティを指定して特定の物理コアにスレッドを固定している．

4.2 工夫 2：SSE 命令によるベクトル処理

図 4 に SSE 命令によるスループット向上手法を示す．提案手法は大量のブロックに対するスループットを向上するために，4 個のブロックをまとめて SSE 命令によりベクトル処理する．具体的には，図 4 に示すように算術演算命令 $E_{i,a}$，$E_{i+1,a}$，$E_{i+2,a}$ および $E_{i+3,a}$ をベクトル化する．ここで，$E_{i,a}$ はブロック B_i における n $(1 \le a \le A)$ 番目の算術演算命令を表す．

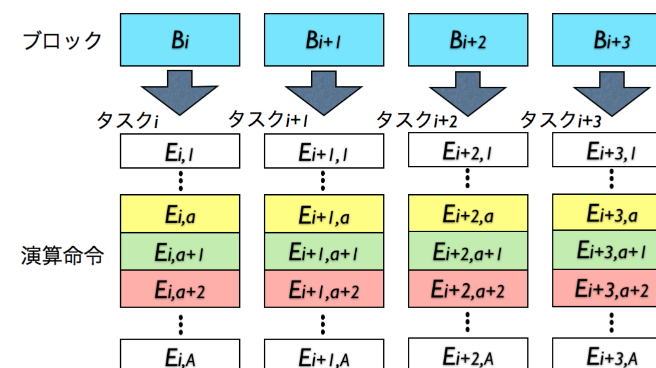


図 4 SSE によるスループット向上手法

提案手法は，複数ブロックに対するスループットを高めるものであり，単一ブロックに対する実行時間は短縮できない．しかし，複数のブロックをまとめて並列処理することにより，ステップ（2）およびステップ（4）をベクトル化の対象にできる．ここで，ステップ（2）およびステップ（4）の算術演算命令数は 2 : 3 の比を持つ．したがって，ステップ（2）をベクトル化の対象とする既存手法[6] と比較して，提案手法は約 2.5 倍の算術演算命令をベクトル化できる．

4.3 工夫 3：ストライドアクセスの回避

ストライドアクセスを回避することは，ベクトル演算の効率を高めるために重要である．そこで，提案手法はステップ（2）において 4 個のタスクが生成するワード $W_{i,j,k}$，$W_{i+1,j,k}$，$W_{i+2,j,k}$ および $W_{i+3,j,k}$ を連続した主記憶領域に格納する．これらを連番の主記憶アドレスに割り当てることにより，ステップ（4）におけるストライドアクセスを回避できる．そのために，一連のワードを格納する配列を以下のように宣言する．

```
int W[80][4];
```

仮に，`W[4][80]` のように宣言してしまうと，k 番目のワード $W_{i,j,k}$，$W_{i+1,j,k}$，$W_{i+2,j,k}$ および $W_{i+3,j,k}$ を参照するときにストライド幅が 80 のメモリアクセスが発生してしまう．

ⓒ 2012 Information Processing Society of Japan

表 2 実験環境

環境	1	2
CPU	Xeon X5677	Xeon E7-4850
周波数 f（GHz）	3.47	2.00
ソケットあたりのコア数 c	6	10
ソケット数 s	2	4
全 CPU のピーク性能 P（GIPS）	166.56	320
Hyper-Threading	有効	無効
主記憶（GB）	24	256
OS	CentOS 6.0 64-bit	CentOS 6.0 64-bit
コンパイラ	gcc 4.6.2	gcc 4.6.1

表 3 各手法の実測スループットおよび実行効率

項目	提案手法		OpenSSL-SSE 手法		OpenSSL 手法	
	環境 1	環境 2	環境 1	環境 2	環境 1	環境 2
実測スループット（Gbps）	80.1	129.1	52.0	79.1	36.6	63.2
実行効率（%）	93	76	60	47	42	37

5. 評価実験

提案手法の性能を評価するためにスループットを計測した実験結果を示す．比較のために，SSE を用いない CPU 実装として OpenSSL[10] の実装を用いた．OpenSSL は C 言語で記述されたオープンソースのライブラリであり，暗号化関数を実装している．さらに，既存手法[6] として，開発版 OpenSSL の SSE による実装（OpenSSL-SSE 手法）を用いた．実験に用いた計算環境を**表 2** に示す．

いずれの実験においても，6 GB のデータを主記憶上に読み終えた状態から SHA-1 計算を開始しスループットを測定した．なお，ブロックサイズ S は 64 KB である．

5.1 スループットの評価

表 3 に，各手法の実測スループットおよび実行効率，すなわちピークスループット T に対する実測スループットの比を示す．なお，環境 2 のピークスループット T（167.9 Gbps）は表 2 の値を基に計算した．環境 1 において，提案手法の実行効率は約 93%に達している．一方，OpenSSL 手法の実行効率は 42%であり，提案手法の速度向上率は 2.4 倍である．4 演算を 1 命令でベクトル処理しているにも関わらず，速度向上率が 4 倍に達しなかった理由は，実行効率が 100%で抑えられるためである．

図 5 に，スレッド数 t を変えたときの各手法の実測スループットを示す．提案手法および OpenSSL 手法は，t に比例してスループットを向上できている．特に 40 コアを持つ環境 2 においてスループットを向上できていることから，提案手法は高いスケーラビリティを持つ．環境 1 において $t \geq 10$ のとき，提案手法のスループットは CPU・GPU 間の実効帯域幅（45 Gbps）を上回る．したがって，ソケットあたり 6 コア程度の CPU であれば，SHA-1 計算を GPU よりも CPU に担当させた方がよい． さらに，OpenSSL-SSE 手法と比較して，提案手法の実測スループットは約 1.5 倍である．

図 6 にブロックサイズ S および実測スループットの関係を示す．いずれの手法においても，ブロックサイズ S に比例してスループットが増大する．この理由は，ブロックサイズ S に依存しない処理，すなわち初期化およびパディングに要する時間が S の増大とともに相対的に減少するためである．SHA-1 計算のスループットを向上するためには，ブロックサイズ S を増大すればよい．しかし，S の増大によりストレージデバイスに格納すべきデータの重複率 R が低下するため，ストレージデバイスへの書き込み時間が増大する．したがって，SHA-1 計算のスループットおよびストレージデバイスへの書き込み時間はトレードオフの関係にある．

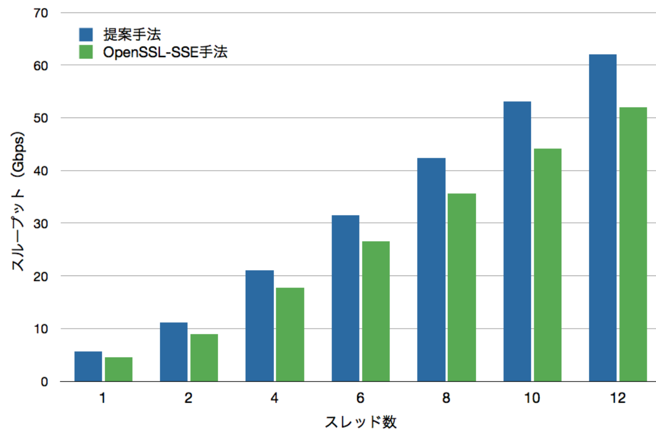
5.2 スループット向上の内訳分析

4 章で示した工夫を評価するために，環境 1 において各々の工夫を使い分けてスループットを計測した．**図 7** に，OpenSSL 手法に提案手法の工夫を段階的に施したときの実測スループットを示す．いずれのスレッド数においても，ストライドアクセスを回避すること（工夫 3）により，提案手法（工夫 1〜2）と比較してスループットを約 1.5 倍に増大できている．したがって，コア数に関わらずストライドアクセスを回避することは有用である．特に，PCI Express の実効帯域幅（45 Gbps）を上回るスループットを得るためには，工夫 3 が有用である．

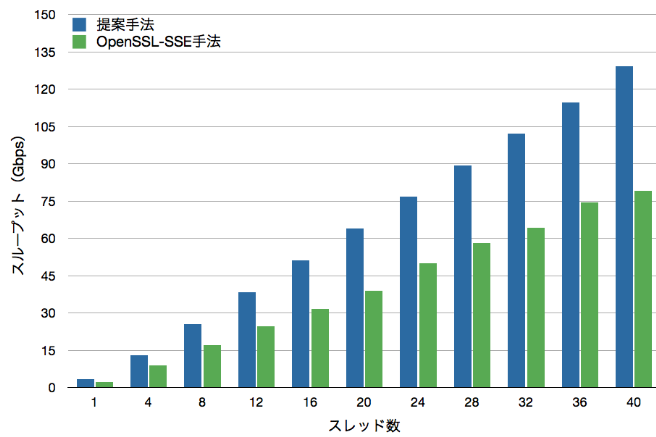
次に，提案手法（工夫 1〜2）および OpenSSL 手法（工夫 1）を比較することにより，SSE による性能向上の度合いを調べた．いずれのスレッド数においても，スループットは約 1.1〜1.2 倍に向上できている．したがって，ストライドアクセスが発生する場合においても，SSE によるベクトル処理（工夫 2）は有用である．

最後に，環境 1 において HT 技術による性能向上の度合いを調べた．コアすべてを使う場合，提案手法の実測スループットは 62.0 Gbps から 80.1 Gbps に増大した．一方，OpenSSL-SSE 手法の実測スループットは 52.0 Gbps から 46.4 Gbps に低下した．提案手法における増大の理由は主記憶へのアクセス時間を HT 技術により隠蔽できたことによる．一方，OpenSSL-SSE 手法における低下の理由は主記憶へのアクセス時間を隠蔽できる余地

ⓒ 2012 Information Processing Society of Japan

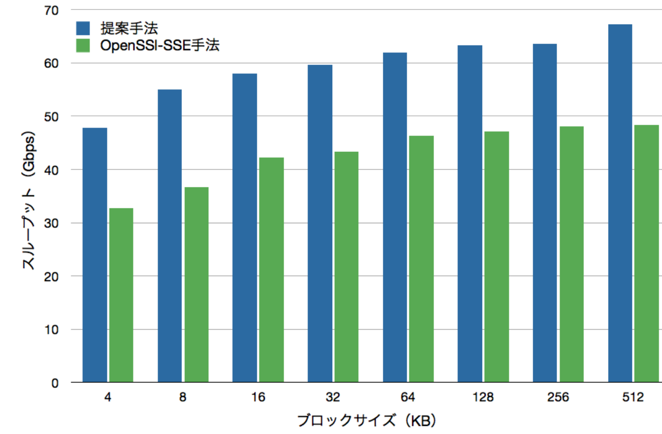


(a) 環境 1

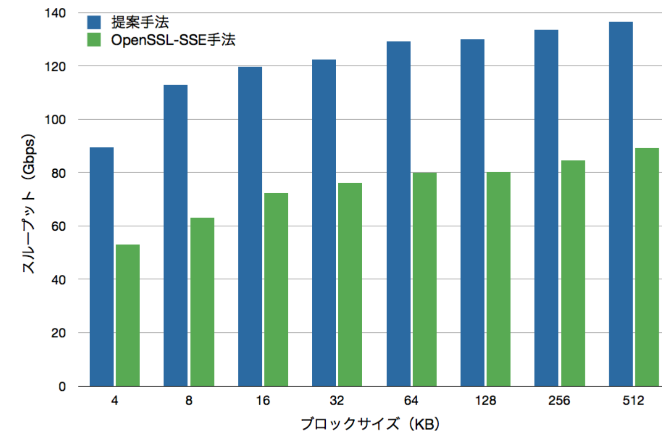


(b) 環境 2

図 5 各手法の実測スループット



(a) 環境 1



(b) 環境 2

図 6 ブロックサイズとスループットの関係

ⓒ 2012 Information Processing Society of Japan

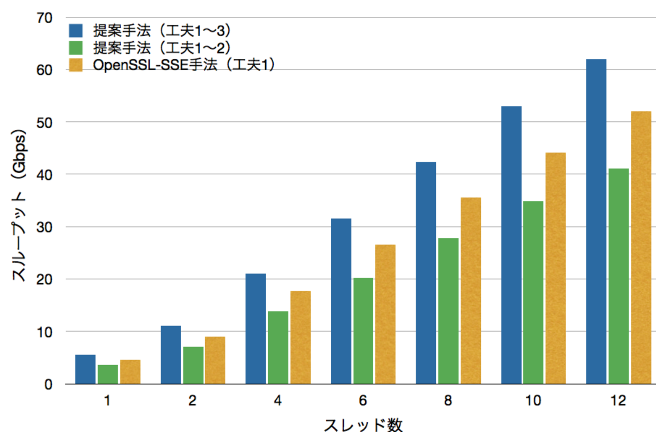


図 7 各工夫によるスループットの向上

が少なく，HT 技術によるスレッド切り替えのオーバヘッドがあることによる．

6. まとめと今後の課題

本稿では，重複排除ストレージのための SHA-1 計算のスループット向上を目的として，複数のブロックを SSE 命令および OpenMP により並列処理する手法を提案した．提案手法は，ベクトル処理の実行効率を高めるために，データ構造を工夫しストライドアクセスを回避する．

実験の結果，HT 技術を有効にした場合，OpenSSL-SSE 手法と比較して提案手法は約 1.5 倍のスループットを実現した．SSE を用いない OpenSSL 手法に対しては 2.4 倍のスループットを実現した．さらに，10 コア使用時のスループットが CPU・GPU 間の実効帯域幅（45 Gbps）を超え，実行効率は約 93%に達する．提案手法を用いることにより，重複排除処理のスループットを維持しつつクロックまたはコア数の少ない CPU を使用できる．

今後の課題としては，提案手法を重複排除ストレージに組み込み，ストレージシステムとしての性能を評価したい．

謝辞 本研究の一部は，科学研究費補助金基盤研究（B）（23300007）の支援を受けた．

参 考 文 献

1) Meyer, D.T. and Bolosky, W.J.: A Study of Practical Deduplication, *Proc. 9th USENIX Conf. File and Storage Technologies (FAST'11)*, pp.1–13 (2011).
2) National Institute of Standards and Technology: Secure Hash Standard, Federal Information Processing Standards Publication 180-2 (2002).
3) Lindholm, E., Nickolls, J., Oberman, S. and Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture, *IEEE Micro*, Vol.28, No.2, pp.39–55 (2008).
4) Klimovitski, A.: Using SSE and SSE2: Misconceptions and Reality, *Intel Developer Update Magazine* (2001).
5) 望月拓良，金子敏信：ハッシュ関数 SHA-1 の GPU(CUDA) によるソフトウェア並列高速実装，電子情報通信学会技術研究報告，IT2011-12, pp.13–18 (2011).
6) Locktyukhin, M.: Improving the Performance of Secure Hash Algorithm (SHA-1) (2010).
7) Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J. and Menon, R.: *Parallel Programming in OpenMP*, Morgan Kaufmann, San Mateo, CA (2000).
8) Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture (2011).
9) NVIDIA Corporation: CUDA Programming Guide Version 3.2 (2010).
10) OpenSSL Project: OpenSSL: The Open Source toolkit for SSL/TSL (2012).

ⓒ 2012 Information Processing Society of Japan