GPU向け高速Tree Reductionアルゴリズム

小池 敦[1,a)]　定兼 邦彦[2,b)]

概要：GPU を用いた Tree reduction アルゴリズムを扱う．Tree reduction は並列計算における最も基本的な問題の一つであり，四則演算を用いて書かれた数式の並列評価や木に対する各種クエリ等の多くの問題の一般化となっている．GPU 向けアルゴリズムを設計する際はマルチプロセッサの SIMD アーキテクチャを考慮する必要があるが，木構造に対する処理では，一般に参照データが入力に大きく依存するため，GPU アーキテクチャを有効に活用することは難しい．しかし，本論文では，GPU アーキテクチャを適切に考慮し，高速に Tree reduction を計算するアルゴリズムを提案する．提案アルゴリズムは木の形に依存せず I/O 計算量が最適となる．

1. はじめに

プロセッサの動作クロック周波数の向上は限界を迎えており，周波数向上に代わるパフォーマンス向上の手段として並列アーキテクチャが注目されている．GPU (Graphics Processing Unit) は元々はグラフィック処理のための専用プロセッサとして開発された．しかし，非常に高い並列性を持っていることから，グラフィック処理以外にも GPU が使われ始めている．汎用の処理に GPU を使用することは GPGPU (general-purpose GPU) と呼ばれており，安価に超並列環境が構築できることから注目されている．

GPU は多数のコアを用いて効率よく処理を行うため，特殊なアーキテクチャとなっている [1]．GPU プログラミングにおいては，このアーキテクチャを適切に考慮する必要がある．NVIDIA 社は GPGPU のための開発環境として，CUDA[2] を提供しており，CUDA 上で開発することにより，様々な GPU モデル上で動作するプログラムを実装することができる．しかし，最適なパフォーマンスを得るためには，GPU アーキテクチャを適切に考慮してアルゴリズムを設計する必要がある．GPU マルチプロセッサは SIMD アーキテクチャとなっており，マルチプロセッサ内のすべてのコアが常に同一の命令を実行している．SIMD コアによるメモリアクセスは，特定の規則でアクセスする場合に効率的になるため，入力に依らずに規則的にメモリアクセスできるようにアルゴリズムを設計する必要がある．筆者らは GPU アルゴリズムに対し漸近解析を行うためのモデルとして，AGPU モデルを提案している [3]．

Tree reduction は並列計算における最も基本的な問題の一つである．木に対する各種クエリ，数字と算術演算子を用いた数式の計算などを含む多くの問題が Tree reduction に一般化され [4], [5], [6]，また，コンパイラ最適化 [7] やメッセージの復号 [8] 等多くの問題で活用されている [9]．Miller と Reif[5] は Tree reduction を計算するための Parallel tree contraction アルゴリズムを提案した．その後も様々な研究が行われている [9], [10], [11], [12], [13]．また，古典的な並列計算の教科書 [14] にも多くの研究成果が紹介されている．

入力が行きがけ順 (preorder) に直列化（シリアライズ）されている場合の tree reduction も重要である．多くのデータがこの方式でシリアライズされているが，XML もこのデータ構造の一つである．本論文ではこのデータ構造を XML 表現と呼ぶ．近年リレーショナルデータベースでの扱いが難しい非構造データが急速に増大しており，これらを柔軟に扱うことができるデータベースとして，XML データベースが注目されている．XML データベースに対するクエリ言語として XQuery [15] があり，クエリ処理の高速化が求められているこのシリアライズ手法は Euler tour technique [16] に対応しており，シーケンシャルに Tree reduction を計算するためには，スタックを用いてシリアライズデータを 1 回スキャンするだけでよい．また，木に対する基本的なクエリの幾つか（木の高さ等）は，このデータ構造に対して並列 Prefix sum を行う時間で解けることが知られているしかし後述するように，Prefix sum では解くことができないような問題が多く存在するため，高速に，より広いクエリを扱うことができるようなアルゴ

[1] 総合研究大学院大学 複合科学研究科 情報学専攻
[2] 東京大学 大学院 情報理工学系研究科
[a)] koike@nii.ac.jp
[b)] sada@mist.i.u-tokyo.ac.jp

リズムが求められている．

Tree reduction を効率的に並列計算するために制約をつけることを考える．Matsuzaki ら [17], [18] は拡張分配則（Extended distributivity）という制約を追加した場合の Tree reduction について提案しており，Kakehi ら [19] は BSP モデル [20] 上での XML 表現に対する tree reduction に関して，拡張分配則を用いた高速アルゴリズムを提案した．計算量は $\mathcal{O}(n/p+p)$ である．また，Emoto,Imachi [21] は上記アルゴリズムを Map-Reduce モデル [22] 向けに改良した．一方，Morihata,Matsuzaki [13] は Tree reduction 計算のための別の制約として，部分縮約則を提案している．この制約を追加しても，なお多くのクエリがカバーされ（詳細は後述する），効率の良い並列計算が可能となる．また，この制約を満たす問題は，上記の Prefix sum で解ける問題を包含している

本論文では，Morihata,Matsuzaki[13] による部分縮約則を活用し，XML 表現に対する Tree reduction を GPU 上で高速計算するアルゴリズムを提案する．上記の BSP モデルや Map Reduce モデル上でのアルゴリズムでは，各コアは与えられた入力に対し，個別に計算を行うが，GPU マルチプロセッサは SIMD アーキテクチャのため，内部のコアは個別に計算を行うことはできない．すなわち，マルチプロセッサ内のコアは常に同一の命令を実行する必要がある．特にメモリアクセスについては，アクセスするデータアドレスがコアごとに入力に依存し大きく変わるため，GPU の効率的なメモリアクセスの仕組み（コアレッシング等）を活用することは難しい．本論文では上記の問題を解決し，高速なアルゴリズムを設計する．提案アルゴリズムは木の形に依存せずに AGPU 上で I/O 計算量が最適となる．AGPU(p, b, M) モデル上で n 要素に対して Tree reduction を行う時，時間計算量は $\mathcal{O}\left(\frac{n \log b}{p}\right)$，I/O 計算量は $\mathcal{O}(n/b)$ である．

2. 準備

2.1 AGPU モデル

AGPU モデルは，GPU 向けアルゴリズムの計算量の漸近解析を行うための並列計算モデルである．AGPU モデルを用いることで，GPU デバイスの詳細仕様に依らない汎用的なアルゴリズム設計と評価を行うことができる．まず，AGPU モデルのアーキテクチャを説明した後，GPU 向けアルゴリズムの評価基準について説明する．

2.1.1 アーキテクチャ

AGPU モデルのアーキテクチャを図 1 に示す．AGPU モデルのアーキテクチャは並列計算を行うためのデバイス (GPU) とデバイスを制御するためのホスト (CPU) の異種混載システムとなっている．デバイスは p 個のコアを備えている．コアのワード長は w ビットであり，コアはワード単位でデータにアクセスする．また，デバイスは k 個のマルチプロセッサで構成されており，各マルチプロセッサは b 個のコアを備えている．すなわち $p = kb$ である．マルチプロセッサはホストから起動されたプログラムを個別に実行する．すなわち，マルチプロセッサは他のマルチプロセッサとの通信手段および同期手段を持たない．ホストはすべてのマルチプロセッサの処理完了を待つことにより，マルチプロセッサ間の同期を行うことができる．しかし，マルチプロセッサの処理完了時，共有メモリのデータはすべて削除される．後で参照する必要があるデータはマルチプロセッサの処理終了時にすべてグローバルメモリに書き込む必要がある．

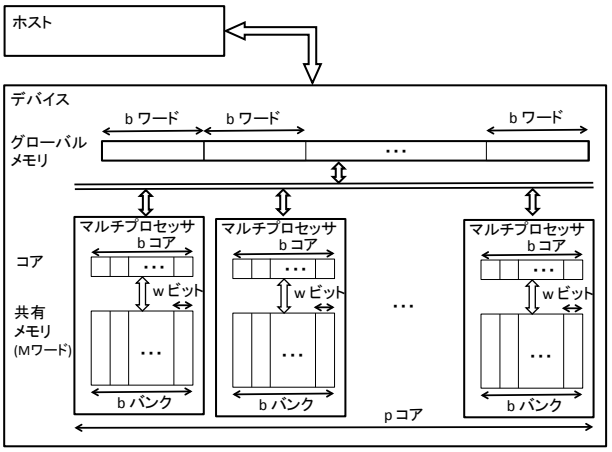


図 1　AGPU モデルのアーキテクチャ

マルチプロセッサ内の b 個のコアは b 個のスレッドに対し，常に同一の命令を実行する．この時，各コアは同一命令を並列に実行するという．1 つのマルチプロセッサ内で並列に処理されるスレッドの集合をワープと呼ぶ．ただし，オペランドに指定されるデータアドレスについてはコアごとに指定することができる．また，命令には実行条件を含めることができ，条件を満たすコアのみ命令を実行させることができる．一方，各コアは複数のスレッドを時分割で切り替えながら同時実行することができる．この時，各コアは複数スレッドを並行に実行するという．言い換えれば，マルチプロセッサは複数ワープを並行に実行することができる．GPU のこの機能をマルチスレッディングと呼ぶ．マルチスレッディングにはグローバルメモリアクセスのレイテンシ（待ち時間）を隠ぺいする効果がある．すなわち，あるワープがグローバルメモリアクセスにより，待ち状態になっている場合に，マルチプロセッサは他のワープを実行することによりコアの使用率を高めることができる．図 2 に具体例を示す．マルチスレッディングは GPU における効率的なメモリアクセスのキーとなる技術である．

デバイスは 2 種類のメモリを備えている．1 つ目はグローバルメモリである．これは低速であるが大容量であり，すべてのマルチプロセッサおよびホストからアクセス可能である．グローバルメモリは b ワードごとのブロック

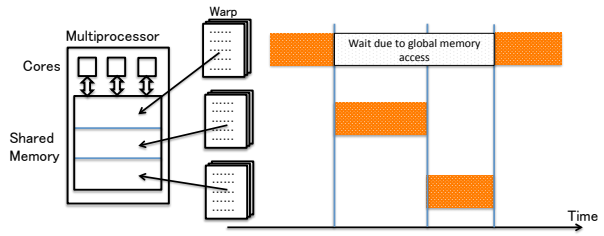


図 2　マルチスレッディングによるレイテンシ隠ぺいの例

に分割されている．同一命令を実行するマルチプロセッサ内の全コアが同一ブロックにアクセスする時，1 回のメモリアクセスで全コア分のデータにアクセスすることができる．これはコアレッシングと呼ばれており，処理時間に大きな影響を与える．一方，コアが複数の異なるブロックにアクセスする時は，各ブロックのに対して 1 回のアクセスが必要となる．2 つ目は共有メモリである．各マルチプロセッサは内部に容量 M ワード $(b \leq M)$ の共有メモリを備えている．これは高速であるが低容量である．また，マルチプロセッサ内部のコアからのみアクセス可能である．共有メモリは b 個のバンクから構成される，同一命令を実行する b 個のコアのそれぞれが異なるバンクにアクセスする時，単位時間でデータにアクセスできる．複数のコアが同一のバンクにアクセスする時は，処理がシリアライズされる．これはバンクコンフリクトと呼ばれており，これも処理時間に大きな影響を与える．以上で定義される計算モデルを AGPU(p, b, M, w) と記載する．ただし M, w については，省略される場合がある．

2.1.2　アルゴリズムの評価基準

まず，アルゴリズムの実行時間を評価する基準として，時間計算量と I/O 計算量を使用する．時間計算量は，各マルチプロセッサで実行されるプログラムの命令発行数である．共有メモリへのアクセスでバンクコンフリクトが発生する場合，コンフリクト数に応じた時間が時間計算量に加算される．また，グローバルメモリへのアクセスについては，b ワードのブロックに対する書き込みまたは読み込みの時間計算量を 1 とする．マルチプロセッサごとに命令発行数が異なる場合には，最も多い発行数を時間計算量とする．I/O 計算量については，上記で説明したグローバルメモリアクセス回数のすべてのマルチプロセッサでの合計値とする．I/O 計算量を時間計算量とは別に評価する理由は，グローバルメモリアクセス処理に要する時間が他の処理に比べて大きくなるためである．また，グローバルメモリに対して同時にアクセス可能なマルチプロセッサ数が限られているため，アクセス回数については，すべてのマルチプロセッサでの合計値とする．

次に，メモリ使用量を評価する基準として，グローバルメモリ使用量と共有メモリ使用量を使用する．使用される単位はビットである．また，共有メモリ使用量は各マルチプロセッサで使用されるメモリ使用量の最大値とする．大規模データを扱う場合，グローバルメモリ使用量を少なくすることは特に重要である．また，共有メモリ使用量は M ワード以下にする必要がある．また，共有メモリ使用量は以下で説明する多重度にも影響する．

2.1.1 節で述べた通り，マルチスレッディングは GPU におけるメモリアクセスのキーとなる技術である．しかし，I/O 計算量の値はマルチスレッディングの効果とは無関係であるため，マルチスレッディングの効果を I/O 計算量を用いて評価することはできない．そこでマルチスレッディングの効果を評価する値として多重度を導入する．

マルチスレッディングの効率を上げるためには，マルチプロセッサに割り当てるワープ数を増やせば良い．しかし，割当ワープ数は共有メモリ使用量に依存する．マルチプロセッサ内のすべてのワープは同一の共有メモリを使用するため，全ワープでの共有メモリ使用量の合計値が共有メモリサイズを超えることはできない．AGPU(p, b, M) 上で設計されたアルゴリズムについて，各ワープの共有メモリ使用量を m とすると，多重度 $\mathcal{M}$ は $\mathcal{M} := M/m$ と定義される．これは CUDA のオキュパンシに対応する値であるが，多重度は AGPU モデルのパラメータを使用して計算することができる．共有メモリ使用量が大きい場合，多重度の値は小さくなり，マルチスレッディングによるレイテンシ隠蔽の効果が小さくなる．

2.2　Tree reduction

木上の頂点 v に対して，以下の関数を定義する．

$$f(v, \otimes, \oplus, h) = \begin{cases} h(w_v) & (v \text{ が葉}) \\ w_v \otimes \left(\bigoplus_{u \in S(v)} f(u, \otimes, \oplus, h) \right) & (\text{その他}) \end{cases}$$

ここで，$S(v)$ は v の子の集合を表し，w_v は頂点 v の重みを表す．$\otimes, \oplus$ は 2 つの引数を取る関数であり，$x \otimes y$ は $\otimes(x, y)$ を意味し，$x \oplus y$ は $\oplus(x, y)$ を意味する．h は 1 つの引数を取る関数である．これらは入力として与えられる．また，$\bigoplus$ は以下のような計算を行う．

$$\bigoplus_{i=0}^{n-1} T[i] = T[0] \oplus T[1] \oplus \cdots \oplus T[n-1].$$

この時，木の根 r に対して，$f(v, \otimes, \oplus, h)$ を算出する計算を Tree reduction と呼ぶ．本論文では，$\oplus$ は結合則を満たすものとする．$\otimes$ は指定する場合以外は結合則を満たしている必要はなく，また，$\otimes, \oplus$ は可換である必要もない．また，$\otimes, \oplus$ は単位元を持っていると仮定する．$\otimes$ または $\oplus$ が単位元を持たない場合は単位元を添加する．また，関数 f は基本型もしくは基本型を要素とする tuple を返すものとする．すなわち，関数 f はリストを含むデータを返すことはできない．また，関数 $\otimes, \oplus, h$ は入力値によらずに定

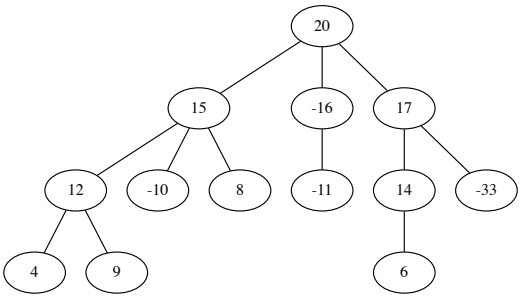


図 **3**　木の具体例

数時間で計算できるものとする

図 3 の木に対して，Tree reduction 計算する例をいくつか挙げる．

例.1 (sum)

木の全頂点の重みの合計を求める．この時，関数 $\otimes, \oplus, h$ を以下のように指定して Tree reduction を計算すれば，所望の値 37 が得られる．

$$\otimes(x, y) = x + y$$
$$\oplus(x, y) = x + y$$
$$h(x) = x$$

例.1' (count)

頂点の重みが 10 以上の頂点数を求める．この場合，頂点重みをあらかじめ以下の関数を用いて変換しておけば，例.1 と同様に扱うことができ，所望の値 5 が得られる．当然，事前に上記の変換を行う代わりに，頂点重みを取得する際に，常に以下の関数の戻り値を取得するようにしてもよい．

$$g(w_v) = \begin{cases} 0 & \text{if } w_v < 10, \\ 1 & \text{otherwise} \end{cases}$$

本論文では，以下，このような場合は例.1 と同様に扱う．

例.2 (max path weight)

根から葉までのパスのうち，パスコスト（パス上の頂点の重みの合計）が最大のパスのパスコストを求める．この時，関数 $\otimes, \oplus, h$ を以下のように指定して Tree reduction を計算すれば，所望の値 57 が得られる．対応するパスは根から重み 6 の葉までのパスである．

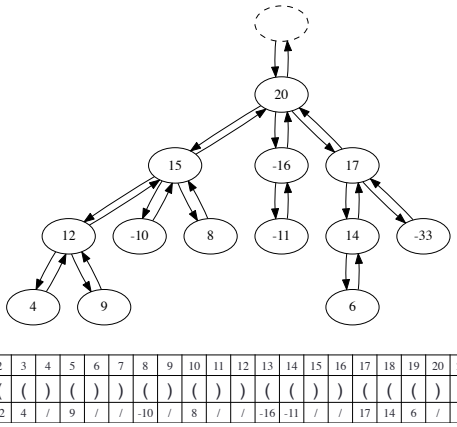
$$\otimes(x, y) = x + y$$
$$\oplus(x, y) = \max(x, y)$$
$$h(x) = x$$

例.3 (max subtree)

木に対するすべての部分木（ある頂点とその頂点のすべての子孫からなる木）のうち，頂点コストの和が最大になるもののコストを求める．この時，関数 $\otimes, \oplus, h$ を以下のように指定して Tree reduction を計算すれば，出力されるペアの第一項が所望の値 38 となる．対応する subtree は重み 15 の頂点を根とする subtree である．

$$\otimes(x, t) = (\max(x + s_t, m_t), x + s_t)$$
$$\text{(ただし } t = (m_t, s_t))$$
$$\oplus(t, u) = (\max(m_t, m_u), s_t + s_u)$$
$$\text{(ただし } t = (m_t, s_t), u = (m_u, s_u))$$
$$h(x) = (x, x)$$

これらの例以外でも木に対する様々なクエリが Tree reduction で表せる [4], [5], [6], [14], [18].



Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
BP	(	(	(	(	)	(	)	)	(	)	(	)	)	(	(	)	)	(	(	(	)	)	(	)	)	)
W	20	15	12	4	/	9	/	/	-10	/	8	/	/	-16	-11	/	/	17	14	6	/	/	-33	/	/	/

図 **4**　図 3 の BP 表現と XML 表現 (配列 W)

2.3　XML 表現

木の直列化表現として，BP 表現と XML 表現を説明する．木を preorder で探索 (traverse) する時，下向き探索時には開きカッコ (を出力し，上向き探索時には閉じカッコ) を出力したものが BP 表現である．図 3 の木に対する BP 表現を図 4 に示す．元の木に対して，根の親にダミーの頂点を追加し，各枝を両向きの 2 つの有向枝に入れ替えると，BP 表現の各 index と有向枝は 1 対 1 対応する．図 4 の木の有向枝には，対応する index を記載した．ある index に対応する頂点とは，下向き探索時は探索の終点頂点，上向き探索時は探索の始点頂点のことを指す（このようにすることで開きカッコと対応する閉じカッコが同じ頂点に対応するようになる）．BP 表現は根付きの ordered tree の構造を保持できるが，重みについては保持できない．本論文で使用する XML 表現は以下のように定義される．BP 表現で開きカッコの index には対応する頂点の重みを格納し，閉じカッコに対応する index には / を格納する．図 4 の配列 W が図 3 の XML 表現である．これは木を preorder でシリアライズする際に一般的に用いられる定義である [21]. XML 表現は頂点数の 2 倍の数の配列を用いて木の構造と頂点の重みの両方を保持できる．

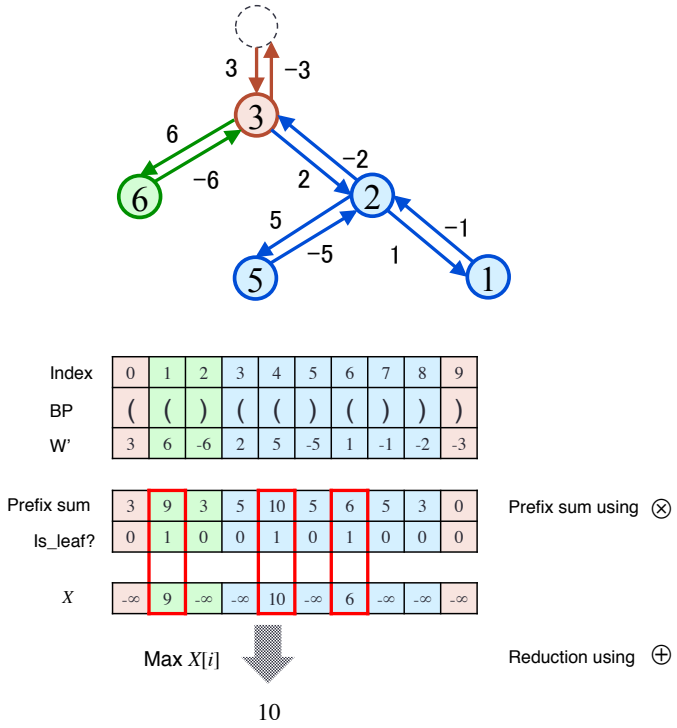


図 5　Prefix sum を用いた max path weight の計算

2.4　XML 表現に対する Tree reduction の先行研究（その 1）

XML 表現に対する Tree reduction はスタックを用いて入力を 1 回スキャンすることで求められる．また，限られた条件においては Tree reduction は効率的に並列計算できることが知られている本節では，$\otimes$ は結合的であり，逆元を持つものとする．また，$\otimes$ は $\oplus$ に対して分配則 (distributivity) を満たすものとする．すなわち $a\otimes(b\oplus c)=(a\otimes b)\oplus(a\otimes c)$ が成り立つ．入力としては，BP 表現の配列と前節の W において / の代わりに対応する開きカッコの重みの $\otimes$ 逆元を代入した配列 W' を与える．配列 W' は Parentheses matching アルゴリズムを用いることで配列 W から計算できる．max path weight の場合の例を図 5 に示す．この場合の戻り値は 10 であり，対応するパスは重みが $3,2,5$ の頂点である．また，枝には index でなく，index に対応する配列 W' の値を書いてある．

Tree reduction を求める手順は以下の通りである．まず，配列 W' に対し，$\otimes$ を用いて prefix sum を求める．max path weight のときは $\otimes$ は $+$ である．次にこの計算結果から葉に対応する index の値のみを取り出す．その他の場合は $\oplus$ の単位元を格納する．max path weight の場合は単位元は $-\infty$ である．最後に，これらの値に対して，$\oplus$ を用いて，reduction を計算する．max path weight の場合は，max の演算となる．この結果，所望の Tree reduction の値が得られる

以上の結果より，もし $\otimes$ が結合的で逆元を持ち，また，$\otimes$ が $\oplus$ に対して分配則を満たすとき，XML 表現に対する

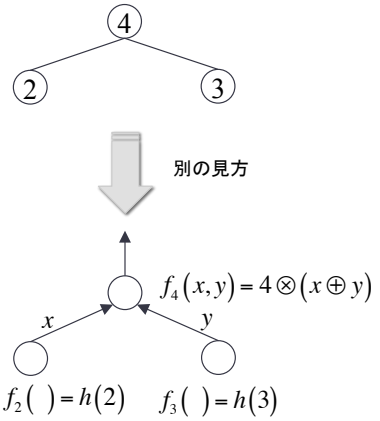


図 6　Tree reduction の回路図表現の例

tree reduction は，Parentheses matching と prefix sum の計算時間で計算できることがわかる．

しかし，$\otimes$ が結合的で，$\otimes$ が $\oplus$ に対して分配則を満たすというのは，かなり強い制約である．この場合，根から葉までのすべてのパスに関して，個別に $\otimes$ による reduction を行ったのち，それらの結果に対して，$\oplus$ による reduction を行うと，Tree reduction の値になるということが証明でき上記アルゴリズムもこの性質を活用している．また，$\otimes$ による reduction も $\otimes$ についての結合則を利用して並列計算できる．すなわち，この場合には自明な並列性を持っていると言える．また，多くのクエリで使用される $\otimes,\oplus$ は分配則を満たさない．例えば，2.2 節の例 1(sum) のような基本的なクエリでさえも分配則を満たしていない．

2.5　Tree reduction の回路図表現

以下の説明のために，まず，Tree reduction の回路図表現

Tree reduction の式を見ると，ある頂点の値を計算するために参照するのは，その子の値のみであり，関数の返り値は親頂点にのみ使用される．これを踏まえると，Tree reduction 計算を別の見方でみることができる．簡単な例を図 6 に示す．葉を除く各頂点は任意の数の子を入力とし，出力（Tree reduction 定義の 2 行目の計算結果）を親に送る回路であると考える．葉は常に定数値を返す回路である (Tree reduction 定義の 1 行目に相当する)．このような表現を本論文では回路図表現と呼ぶことにする．本論文では今後常に Tree reduction をこのように解釈する．

$\otimes,\oplus$ が結合的であり，$\otimes$ が $\oplus$ に対して分配則を満たす時，図 7 のような変換が可能である．Tree reduction を定義に従って計算する場合，葉から根の方向に順に計算していく必要があるが，$\otimes$ が $\oplus$ に対して分配則を満たす場合は，内部頂点に対してこのような変換を行うことで木の頂点数を減らすことができるため，効率的な並列計算が可能となる．

四則演算による算術計算も回路図表現が可能である．$(4\times5)+(6\times7)$ に対応する回路図表現を図 8 に示す．

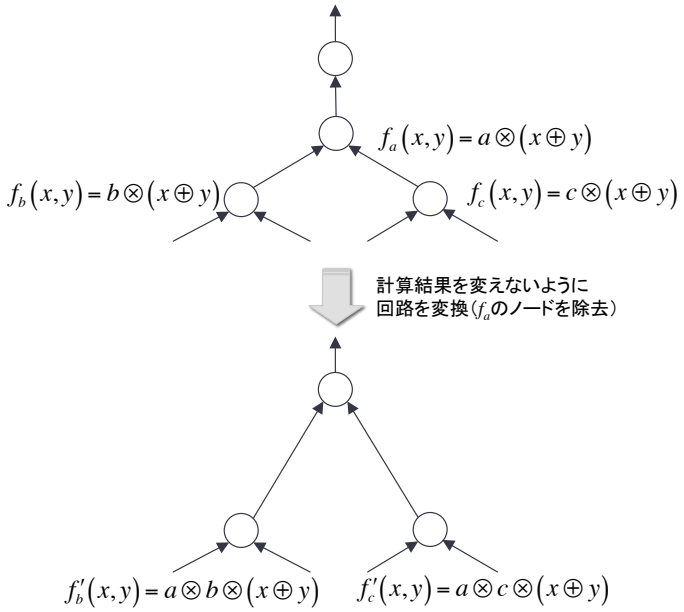


図 **7**　分配則を利用した回路変換の例

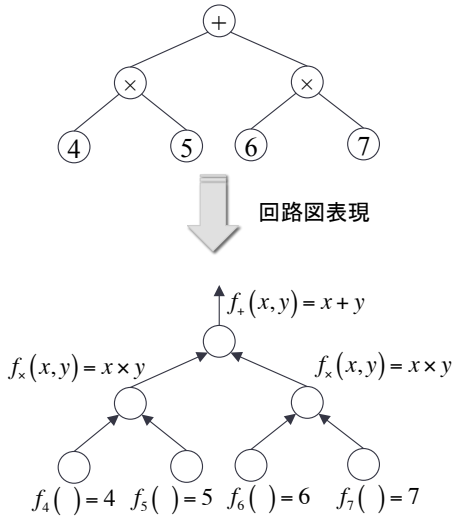


図 **8**　算術計算 $(4\times5)+(6\times7)$ に対応する回路図表現

2.6　XML 表現に対する Tree reduction の先行研究（その２）：拡張分配則（Extended distributivity）

Tree reduction を効率的に解くための制約として，拡張分配則を説明する．

定義 2.1 (拡張分配則 [17], [18])　$\oplus$ は結合的とする．また，$f_0(x)=a_0\otimes(b_0\oplus x\oplus c_0)$ とし，$f_1(x)=a_1\otimes(b_1\oplus x\oplus c_1)$ とする．この時，ある関数 p_1,p_2,p_3 が存在し，任意の a_0,b_0,c_0,a_1,b_1,c_1 に対し，以下が成り立つ時．$\otimes$ は $\oplus$ に対して，拡張分配則を満たすという．

$$
\begin{aligned}
f_0\circ f_1(x) &= f(x)\\
f(x) &= a\otimes(b\oplus x\oplus c)\\
a &= p_1(a_0,b_0,c_0,a_1,b_1,c_1)\\
b &= p_2(a_0,b_0,c_0,a_1,b_1,c_1)
\end{aligned}
$$

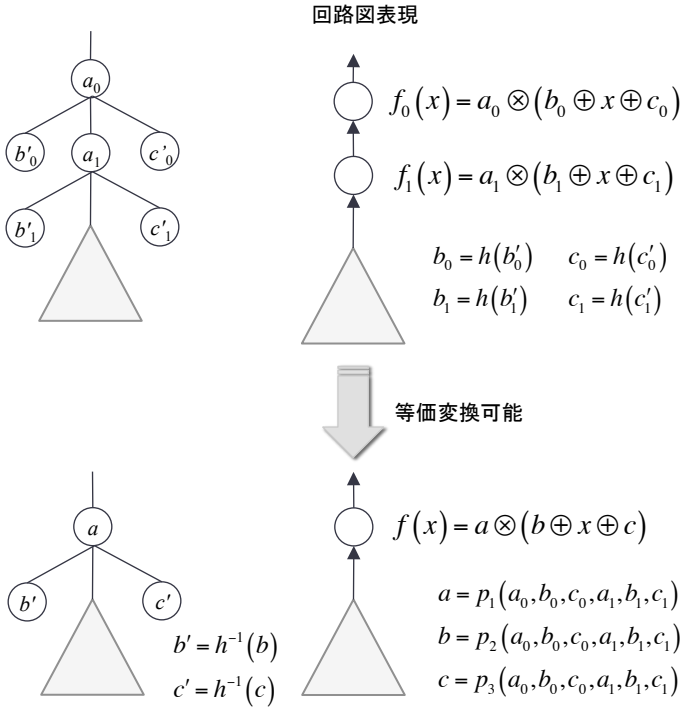


図 **9**　拡張分配則と Tree reduction との対応関係（h が全単射の時）

$$c = p_3(a_0,b_0,c_0,a_1,b_1,c_1)$$

□

上記の状況および h が全単射の場合の Tree reduction との対応を図 9 に示す．h が全単射でない時は，変換後の関数に対応する入力の木は存在しないが，特に問題はない．内部頂点に対し，拡張分配則を用いて変換を行うことで頂点数を減らすことができる．これにより，分配則の場合と同様に効率的な並列計算ができるようになる．

Kakehi ら [19] は BSP モデル [20] 上での XML 表現に対する tree reduction に関して，拡張分配則を満たす場合の高速アルゴリズムを提案した．n を要素数，p をコア数とすると，計算量は $\mathcal{O}(n/p+p)$ である．本論文で提案するアルゴリズムもこのアルゴリズムと同様のアイデアを使用している．また，Emoto,Imachi [21] は上記アルゴリズムを Map-Reduce モデル [22] 向けに改良した．

2.7　部分縮約則

本論文では，$\otimes,\oplus$ が部分縮約則を満たす場合の高速 GPU アルゴリズムを提案する．そこで，まず，Morihata, Matsuzaki [13] によって提案されている部分縮約則について説明する．ただし，本論文のこれまでの説明に合わせて，彼らの説明とは異なる方法で説明する．また，簡単のため，特に断らない限り $h(x)=x$ と仮定して説明する．

定義 2.2 (部分縮約則 [13])　$\oplus$ は結合的とする．また，$f_0(x)=a_0\otimes(b_0\oplus x\oplus c_0)$，$f_1(x)=a_1\otimes(b_1\oplus x\oplus c_1)$，$f_2(x)=a_2\otimes(b_2\oplus x\oplus c_2)$ とする．この時，入力値によらずに定数時間で計算可能な関数 ρ_1,ρ_2,ρ_3 が存在し，任

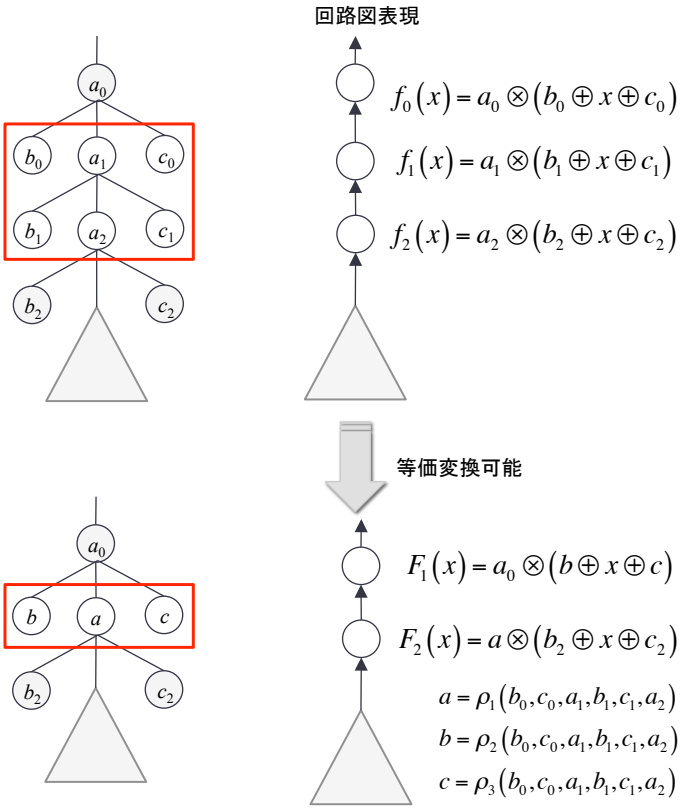


図 **10**　部分縮約則と Tree reduction との対応関係（$h(x) = x$ の場合）

意の $a_0, b_0, c_0, a_1, b_1, c_1, a_2, b_2, c_2$ に対し以下が成り立つ時，$\otimes, \oplus$ は部分縮約則を満たすという．

$$
\begin{aligned}
f_0 \circ f_1 \circ f_2(x) &= F_1 \circ F_2(x) \\
F_1(x) &= a_0 \otimes (b \oplus x \oplus c) \\
F_2(x) &= a \otimes (b_2 \oplus x \oplus c_2) \\
a &= \rho_1(b_0, c_0, a_1, b_1, c_1, a_2) \\
b &= \rho_2(b_0, c_0, a_1, b_1, c_1, a_2) \\
c &= \rho_3(b_0, c_0, a_1, b_1, c_1, a_2)
\end{aligned}
$$

□

部分縮約則と Tree reduction との対応を図 10 に示す．これは，3 つ以上の関数の合成を 2 つの関数の合成で表せるということを意味する．ただし，縮約前後で a_0, b_2, c_2 の値が変わっていないこと，および，関数 ρ_1, ρ_2, ρ_3 の引数に a_0, b_2, c_2 が含まれていないことに注意が必要である．これにより，並列化が容易になる．例えば，図 11 において部分縮約則を満たす場合，X と Y の縮約を並列に実行することができる．

$\otimes$ が結合的で，$\otimes$ が $\oplus$ に対して分配則を満たす時，$\otimes, \oplus$ は部分縮約則を満たす．この時，a, b, c は以下のように計算できる．

$$
\begin{aligned}
a &= a_1 \otimes a_2 \\
b &= b_0 \oplus (a_1 \otimes b_1)
\end{aligned}
$$

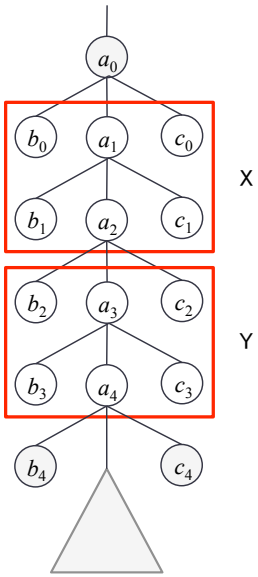


図 **11**　並列に部分縮約則を適用する例

$$
c = (a_1 \otimes c_1) \oplus c_0
$$

これは，図 10 上図において，a_1 を展開することで得られる．$X = b_2 \oplus x \oplus c_2$ とおくと，

$$
\begin{aligned}
f_0 \circ f_1 \circ f_2(x) &= a_0 \otimes (b_0 \oplus (a_1 \otimes (b_1 \oplus (a_2 \otimes X) \oplus c_1)) \\
&\quad \oplus c_0) \\
&= a_0 \otimes (b_0 \oplus (a_1 \otimes b_1) \oplus (a_1 \otimes a_2 \otimes X) \\
&\quad \oplus (a_1 \otimes c_1) \oplus c_0) \\
&= a_0 \otimes (b \oplus (a \otimes X) \oplus c) \\
&= a_0 \otimes (b \oplus (a \otimes (b_2 \oplus x \oplus c_2)) \oplus c) \\
&= F_1 \circ F_2(x)
\end{aligned}
$$

2.2 節の例.1，例.2，例.3 は部分縮約則を満たしている．

例.1 (sum)

a, b, c を以下のように定めることができる．（実際には $\oplus$ は可換なので c は不要）

$$
\begin{aligned}
a &= a_1 + a_2 \\
b &= b_0 + b_1 \\
c &= c_0 + c_1
\end{aligned}
$$

例.2 (max path weight)

$\otimes$ が $\oplus$ に対して分配則を満たしているので，上記の説明をそのまま適用して a, b, c を定めることができる．（実際には $\oplus$ は可換なので c は不要）

$$
\begin{aligned}
a &= a_1 + a_2 \\
b &= \max(b_0, a_1 \otimes b_1) \\
c &= \max(a_1 \otimes c_1, c_0)
\end{aligned}
$$

例.3 (max subtree)

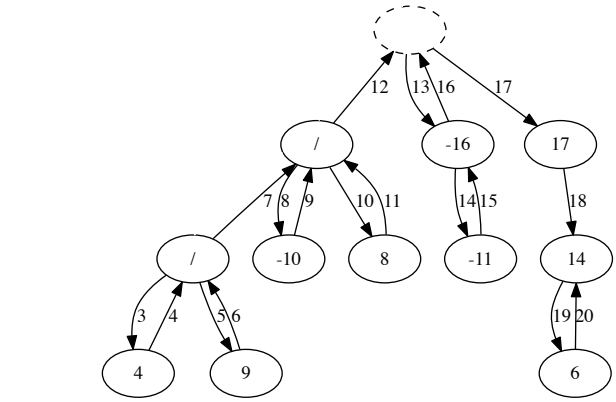
やや複雑なので，まず，$\oplus$ の可換性を利用して b_i と c_i

を縮約することにする．頂点 b_i と頂点 c_i を重み $b_i \oplus c_i$ の頂点に置き替え，これを頂点 b_i とする．頂点 c_i は削除する．また，このケースでは $h(x) = x$ は成り立たない．回路図表現におけるは b_0, b_1 は以下のような値を持っているとする．

$$b_0 = (m_{b_0}, s_{b_0})$$
$$b_1 = (m_{b_1}, s_{b_1})$$

この時，a, b を以下のように定めることができる．

$$a = \max(a_1 + s_{b_1} + a_2, a_2)$$
$$b = (\max(m_{b_0}, m_{b_1}), s_{b_0} + a_1 + s_{b_1} + a_2 - a)$$



Index	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
W[3..20]	4	/	9	/	/	-10	/	8	/	/	-16	-11	/	/	17	14	6	/

図 **12** 図 4 の W[3..20] に対応する擬似木

3. 定式化および提案アルゴリズム

3.1 定式化

AGPU(p, b, M, w) を用いて，以下の入力から以下の出力を計算する．

input $W[n]$: $n/2$ 頂点からなる木の XML 表現

output $f(r, \otimes, \oplus, h)$: Tree reduction 関数の木の根 r に対する出力値

入力はグローバルメモリ上に置かれており，出力もグローバルメモリに置く．W の各要素のサイズは AGPU のワード長 w と一致しているものとする．

ただし，関数 $\otimes, \oplus$ は以下の条件を満たしているものとする．

- 関数 $\otimes, \oplus, h$ は入力値によらずに定数時間で計算可能である
- $\oplus$ は結合則を満たす
- $\otimes, \oplus$ は部分縮約則を満たす

$\otimes$ は結合的でなくてもよく，また，$\otimes, \oplus$ は可換でなくてもよい．

なお，$\otimes, \oplus$ は単位元を持っていると仮定することができる．もし，持っていない場合は添加することにする．以後，$\otimes$ と $\oplus$ の単位元を区別せずに I と書く．2 項演算において引数の一つが I の時は演算を行わなわず，もう片方の値を返す．両方 I の時は I を返す

3.2 擬似木と縮約木

まず，提案アルゴリズムにおいてキーとなるデータ構造である擬似木と縮約木について説明する．入力配列に対する（連続する index からなる）部分配列を擬似木と呼ぶ．例として，図 4 の XML 表現に対する部分配列 W[3..20] を考える．この配列の走査の様子を図示すると図 12 の上図のようになる．根に関しては対応する index が存在しないので，点線で記載してある（上向き探索では枝に対応する頂点は始点であることに注意）．このような頂点を擬似根と呼ぶことにする（すなわち，最も高い位置にある対応する index がない頂点が擬似根である）．すべての木，擬似木に対して，擬似根を持っていると仮定することができる．また，一部の頂点については重みの値を持っていないため，その場合の重みは / と書いた．図 12 上図において，2 頂点間が両向きの 2 本の枝で接続されている場合，その枝のペアを paired-edge と呼び，片方向の枝でしか接続されていないものを solo-edge と呼ぶことにする．擬似木の各枝は preorder の走査に対応しているので，solo-edge は擬似木の左縁か右縁にしか存在しない．言い換えれば，solo-edge は left-visible もしくは right-visible である．

Tree reduction を計算する場合，擬似木に対しても，一部の頂点について計算を行うことができる．図 12 に対して部分的に max path weight 問題のための tree reduction を行った結果を図 13 に示す．この木を縮約木と呼ぶことにする．縮約木を算出するアルゴリズムについては後述する．縮約木の性質として，まず，擬似木のすべての solo-edge は縮約木にも含まれる．また，paired-edge には必ず葉が接続されており（そうでなければもっと contraction できる），各頂点は高々 1 つの paired-edge としか接続されない（$\oplus$ が assosiative であると仮定したことにより，隣り合う sibling に対して，常に $\oplus$ による演算を行うことができる）．

以上の考察により，縮約木を保持するためには，以下の値を保持すれば十分であることがわかる．

- 配列 A: right-visible な solo-edge に対応する頂点の重みを格納する配列
- 配列 B: 配列 A の各頂点に paired-edge で接続された葉の重みを格納する配列
- 配列 C: left-visible な solo-edge に対応する頂点および擬似根に paired-edge で接続された葉の重みを格納する配列
- N_A: 配列 A の要素数

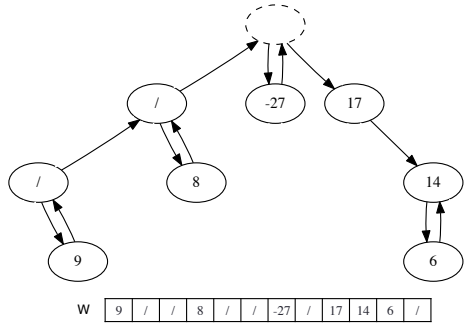


図 **13**　図 12 に対して max path weight 算出のための contraction を行った結果（縮約木と呼ぶ）

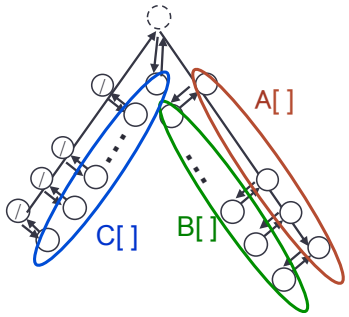


図 **14**　縮約木を保持するためのデータ構造

- N_C: 配列 C の要素数

これらの値を以後，5 つ組と呼ぶことにする．図 14 に配列 A, B, C を図示する．配列 B の要素数は N_A である．また，left-visible な edge に対応する頂点は重みの値を持たないため，これらのための配列は不要である．なお，right-visible または left-visible な頂点が paired-edge に接続された葉を持たない場合 (例えば図 13 の重み 17 の頂点)，単位元の重みを持つ葉を接続することにする．このようにすることで right-visible または left-visible な頂点は常に paired-edge に接続された葉を持つことになる．追加される頂点は高々 n 個であり，このようにしても Tree reduction の結果を変えない．以上より，right-visible な edge の数は N_A であり，left-visible な edge の数は $N_C - 1$ である．

また，配列 A, B に関しては上の要素から順に要素を格納し，配列 C に関しては下の要素から順に要素を格納する．

また，上記のデータ構造と XML 表現は相互に容易に変換できる．

3.3　提案アルゴリズムの概要

AGPU(p, b, M, w) のマルチプロセッサの数を k とおく．すなわち，$k = p/b$ である．提案アルゴリズムは以下の 4 つの手順で Tree reduction を計算する．図 15 に図示する．

(1) 入力配列を k 個の擬似木に分割し，k マルチプロセッサは個別に各擬似木に対して縮約を行い，縮約木を生成する．(Local Tree Contraction for Pseudo Trees) 各マルチプロセッサは縮約木の情報を表す 5 つ組を出力する．全体として，このステップでは k 個の 5 つ組が得られる．

(2) k 個の 5 つ組を用いて，k 個の縮約木に対する parentheses matching を行う (solo-edge に対して対応する edge を探す)．この際，$\mathcal{O}(k)$ 個の区間を用いてカッコの対応関係を表現できる (Global Parentheses Matching)．これは，シーケンシャルアルゴリズムを用いて $\mathcal{O}(k)time$ で計算できる．

(3) 各マルチプロセッサは，自身に割り当てられた縮約木中の開きかっこに対して，上記の parentheses matching の情報を用いて閉じかっこの情報 (具体的には，配列 C の値) を得る．そして，部分縮約則を用いて，縮約を行う．この結果，縮約されたの木の頂点数は高々 $\mathcal{O}(k)$ になる．(Local Tree Contraction for Contracted Trees)

(4) シーケンシャルアルゴリズムを用いて，残った頂点の縮約を行う．これは $\mathcal{O}(k)time$ で計算できる．(Global Tree Contraction)

以下，それぞれのステップについて，詳細を説明する．

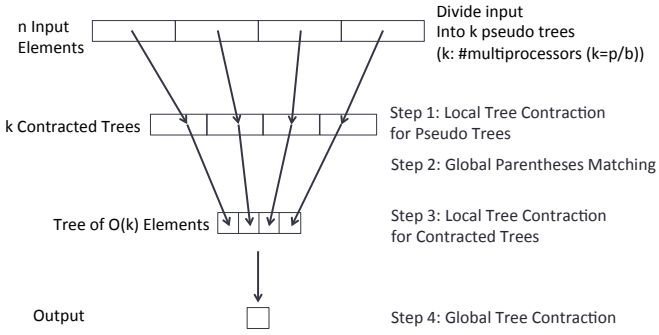


図 **15**　提案アルゴリズムの概要

3.4　Local Tree Contraction for Pseudo Trees

AGPU の各マルチプロセッサは高々 $\lceil n/k \rceil$ 個の要素から構成される擬似木に対し，縮約木を生成する．

まず，擬似木から縮約木を生成するためのシーケンシャルアルゴリズムを説明し，その後，そのアルゴリズムを用いて GPU 向けアルゴリズムを設計することにする．シーケンシャルアルゴリズムを Algorithm 3.1 に示す．これは，XML 表現に対するシーケンシャルな Tree reduction 計算アルゴリズムを擬似木が扱えるように改良したものである．入力に XML 表現（部分配列でないもの）を入れた場合は，$N_C = 1, N_A = 0$ となり，Tree reduction 結果は $C[0]$ に格納される．

次に，これを用いて，GPU マルチプロセッサ向けアルゴリズムを設計する．まず，処理の概要を図 16 に示す．入力の擬似木は b^2 要素のサブブロックに分割され，サブブロックごとに処理を行う．各サブブロックに対する処理をまとめると以下のようになる．

(1) b^2 個の要素をグローバルメモリから共有メモリに読み

Algorithm 3.1 縮約木算出のためのシーケンシャルアルゴリズム

```
1: procedure GenerateContractedTree(W, n)
                                          ▷ W[n]: pseudo tree
2:     N_A ← 0                            ▷ Size of arrays A and B
3:     C[0] ← I
4:     N_C ← 1                            ▷ Size of array C
5:     for i ← 0 to n do
6:         if W[i] ≠ '/' then
7:             A[N_A] ← W[i]
8:             B[N_A] ← I
9:             N_A ← N_A + 1
10:        else
11:            if N_A == 0 then
12:                C[N_C] ← I
13:                N_C ← N_C + 1
14:            else if N_A == 1 then
15:                N_A ← N_A − 1
16:                C[N_C] ← C[N_C] ⊕ (A[N_A] ⊗ B[N_A])
17:            else
18:                N_A ← N_A − 1
19:                B[N_A − 1] ← B[N_A − 1] ⊕ (A[N_A] ⊗ B[N_A])
20:            end if
21:        end if
22:    end for
23:    return A, B, C, N_A, N_C
24: end procedure
```

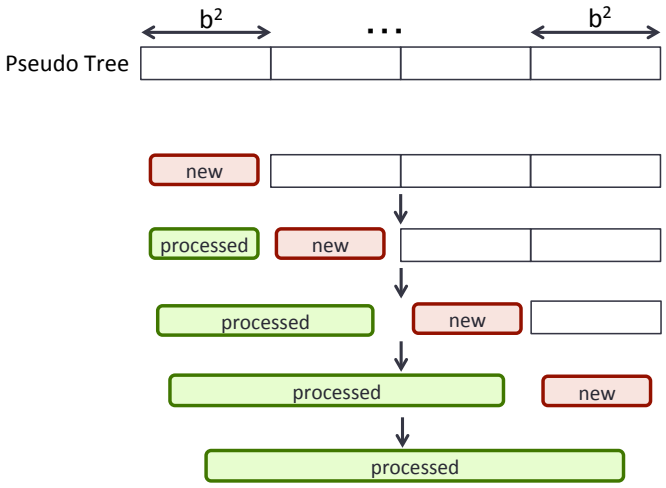


図 **16**　GPU マルチプロセッサ向けアルゴリズムの概要

込む

(2) 読み込んだデータは $b \times b$ の行列として表現され，マルチプロセッサ内の b コアは 1 行の b 要素を担当する．各コアは 1 行のデータに対し，Algorithm 3.1 を用いて縮約木を生成する．すると b 個の縮約木が生成される．

(3) 上記の b 個の縮約木に対し，左の縮約木から順に処理済みの縮約木とのマージを行う．この際，部分縮約則を用いることで b コアで並列に縮約処理を行う．

以下，各ステップの詳細について説明する．

3.4.1　グローバルメモリからのデータ読み込み

b^2 個の要素をグローバルメモリから共有メモリに読み込む処理について，詳細を説明する．b^2 個の要素を b 行 b 列の 2 次元配列 $w[b][b]$ とみなすことにする．先頭の b 要素は 1 行目，次の b 要素は 2 行目に配置されている．そして，アルゴリズムは 1 行ずつグローバルメモリから共有メモリにコピーする．ただし，コピー先アドレスに関して，後の処理を効率化するために工夫する．共有メモリのメモリアドレス ($w[0][0]$ のアドレスからの相対値) については図 17(a) のようになっている．この時，図 17(b) に示すように，要素 $w[i][j]$ はアドレス $bi + ((i+j)\%b)$ に配置される．この処理はグローバルメモリに対して常にコアレスアクセスを行っており，また，共有メモリに対するバンクコンフリクトは発生しない．

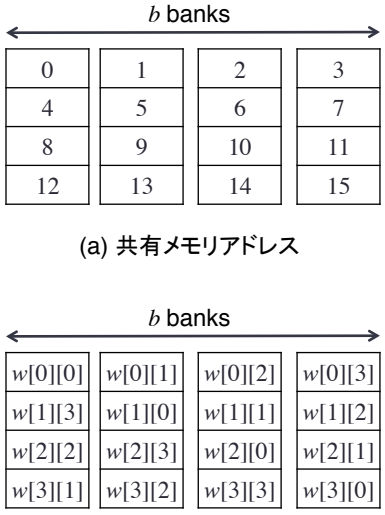


図 **17**　配列 $w[b][b]$ の共有メモリへの配置方法 ($b = 4$ の場合)

3.4.2　縮約木生成のための並列処理

図 17(b) に示すように共有メモリに読み込まれた b^2 個の要素に対し，b コアを用いて，並列に縮約木生成処理を行う．マルチプロセッサ内のコア i は i 行目を担当し，マルチプロセッサ全体で b 個の縮約木を生成する．各コアは Algorithm 3.1 を並列に実行する．ただし，メモリアクセスについては，若干修正が必要である．まず，シーケンシャルアルゴリズムが $W[j]$ にアクセスする時，コア i は $w[i][j]$ にアクセスする．よって，$w[0][j], w[1][j], \cdots, w[b-1][j]$ が並列にアクセスされることになるが，図 17(b) のメモリ配置により，バンクコンフリクトが発生しない．次に，その他のメモリについては，コア i のためのメモリはバンク i に確保する．具体例を図 18 にあげる．シーケンシャルアルゴリズムが配列 $A[b]$ を使用する時，コア i は配列 $A_i[b]$ を使用するものとする．そして，$A_i[b]$ のすべての要素はバンク i に配置される．これらの配列はコアごとに異なる index にアクセスすることがあるが，このようなメモリ配置にすることで，それらのメモリアクセスによるバンクコンフリクトを回避することができる．

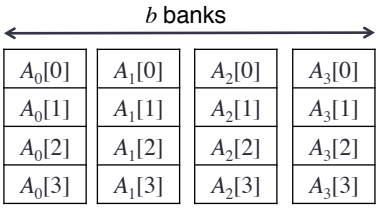


図 **18**　配列 $A_0[b], A_1[b], \cdots, A_{b-1}[b]$ の共有メモリへの配置方法 ($b = 4$ の場合)

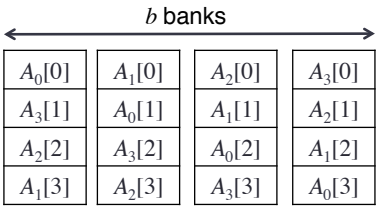


図 **19**　配列 $A_0[b], A_1[b], \cdots, A_{b-1}[b]$ の共有メモリ配置の変換 (縮約木生成後；$b = 4$ の場合)

3.4.3　処理済みの縮約木とのマージ処理

まず，メモリ配置について確認する．処理済みの縮約木 T はグローバルメモリに保持され，処理中の b 個の縮約木 t_i $(0 \leq i < b)$ は共有メモリに保持されている．$t_0, \cdots, t_{b-1}$ を順に T にマージするが，t_i を T にマージする際は，T に関する必要なデータをグローバルメモリから取得し，マージ処理したのち，再び処理結果をグローバルメモリに書き込む．マージ処理は具体的には，t_i の left-visible edge と対応する T の right-visible edge を探し，見つかった場合は，配列 A, B の値を取得し，部分縮約則を用いて並列に縮約を行う（配列 A, B, C に対し，関数 ρ_1, ρ_2, ρ_3 を使用して要素数を減らしていく）．

上記の並列縮約処理において，バンクコンフリクトを回避するため，あらかじめ共有メモリ上のメモリ配置を変換する．前節の処理が完了したとき，図 17(b) に示すように t_i のデータはすべて bank i に格納されている．マージ処理を行う前に，これを図 19 のように変換する．正確には $A_i[j]$ がアドレス $bj + ((i + j)\%b)$ に配置されるようにする．この処理は 1 行ずつデータを上書きしていくことにより行う．よって，この処理は $2b$ 回の共有メモリアクセスにより行うことができる．配列 A_i, b_i, C_i のすべてに対して，この処理を行う．この変換により，配列 A_i の異なる index に対し，コアが並列でアクセスできるようになる．

ここで，連続する 2 つの縮約木に関する性質について確認する．1 つめの擬似木に対して走査を行った際の最終到達頂点と 2 つめの擬似木における走査開始頂点は一致する．また，縮約木においてもこれらの頂点が縮約されることはない．よって，連続する 2 つの擬似木においてそれぞれ縮約木を生成すると，1 つめの縮約木の最右頂点と 2 つめの縮約木の最左頂点が一致する．この状況の例を図 20 に示す．また，1 縮約木のマージ処理において，最大でも b 頂点しかマージされないため，T の最右頂点から上方向に最大 b 個の right-visible edge の情報を取得すれば十分である．よって，1 回のマージおけるグローバルメモリアクセスの回数は A, B それぞれに対して 2 回ずつで十分である

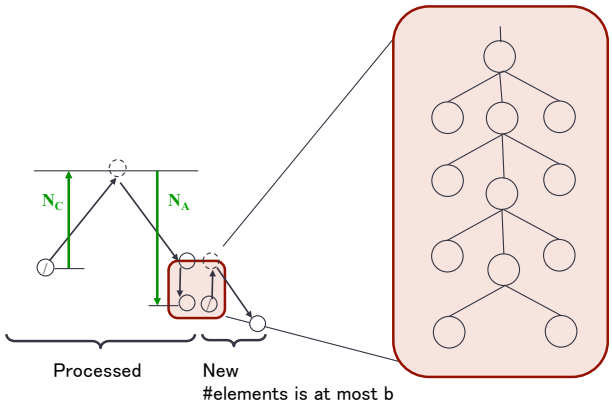


図 **20**　処理済みの縮約木と処理中の縮約木のマージ処理の例

次に，マージ結果の書き込みについて説明する．t_i の right-visible edge を T に追加するとき，T の縮約されずに残った最右頂点の先に接続される．right-visible edge については，上（根）から下（葉）方向にデータが格納されているため，T のデータを修正することなく，高々 2 回のグローバルメモリアクセスで，書き込みが完了する．t_i の left-visible edge を T に追加するとき，T の擬似根の上に接続される．left-visible edge については，下（葉）から上（根）方向にデータが格納されているため，T のデータを修正することなく，高々 2 回のグローバルメモリアクセスで，書き込みが完了する．

3.5　Global Parentheses Matching

前節までの処理で，k 個の縮約木が得られる．k 個の縮約木は連続する 2 つの縮約木の終点と始点を接続することにより，1 つの大きな擬似木であると考えることができる．そこで，次にこの擬似木に対してさらに縮約を行うことを考える．なお，これ以降のアルゴリズムは Kakehi ら [19] のアルゴリズムと同様のアイデアを使用している．各縮約木に対する N_A, N_C をスキャンすることで，right-visible edge と left-visible edge の対応を計算することができる（各縮約木に対する N_A, N_C を格納する配列を $N_A[k], N_C[k]$ とする）．この処理を本論文では Parentheses matching と呼ぶことにする（本処理は対応する BP 表現上での Parentheses matching になっているため）．$k = 4$ の場合の具体例を図 21(a) に示す．right-visible edge と left-visible edge の対応が見つかった場合，right-visible edge が left-visible edge の情報を取得することにより，図 21(b) のような形状の木となる．この部分に対して，部分縮約則を用いて縮約を行うことができる．もし木の内部にこのような構造があったとしても，縮約が可能であることに注意する．なお，左 3 つ

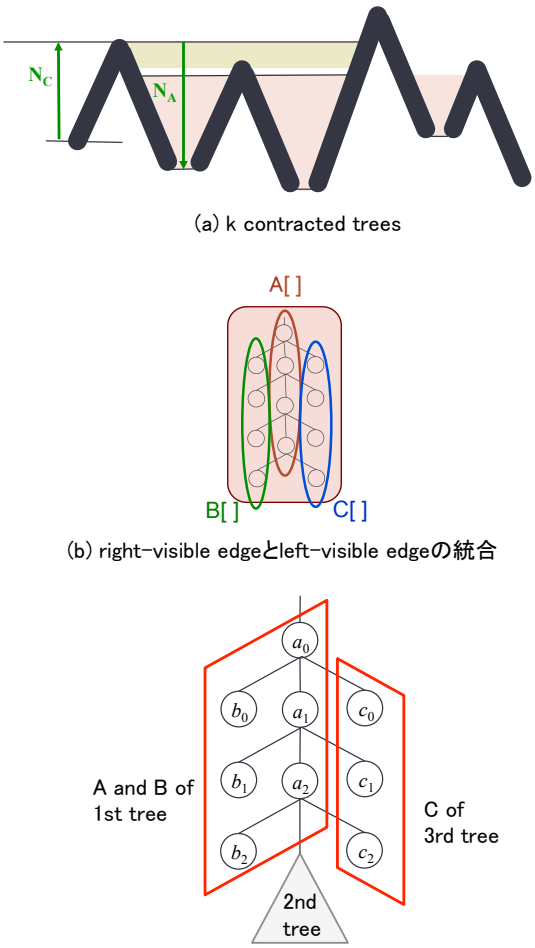


図 **21**　k 個の縮約木に対する Parentheses matching ($k = 4$)

の縮約木の結合部は図 21(c) のようになっている．頂点 a_2 は 2 番目の擬似木の擬似根に対応している．また，1 つめの縮約木と 2 つめの縮約木の縮約前は b_2 は子孫を持っている．また，3 つめの縮約木の c_2 よりも深い頂点は，2 つめの縮約木に読み込まれる．

Parentheses matching の詳細を述べる前に，まず，使用する用語を整理する．木の深さを 1 つめの縮約木の始点頂点からの相対的な深さで表現する．1 つめの縮約木の始点頂点の深さを 0 とし，その頂点より上にある頂点の深さは負の値をとる．深さ h から ℓ ($h \leq \ell$) において，縮約木 i の right-visible edge（に対応する頂点）と縮約木 j の left-visible edge（に対応する頂点）に対応があるとき，それを 4 つ組を用いて (h, ℓ, i, j) と表すもし対応関係が見つかっていない場合は，(h, ℓ, i, ϕ) や (h, ℓ, ϕ, j) と書くことにする．また，各縮約木 i は上記 right-visible edge の対応関係を保持するための配列 $R_i[b]$，および対応関係のない left-visible edge の区間を保持するための変数 L_i を持つ．Parentheses matching のためのシーケンシャルアルゴリズムを Algorithm 3.2 に示す．$R_i[b]$ は深い区間ほど前に格納されることに注意する．また，対応関係を表す 4 つ組の総

Algorithm 3.2 Parentheses matching のためのシーケンシャルアルゴリズム

```
1:  procedure MatchParentheses(N_A, N_C, k)
                                                  ▷ N_A[k], N_C[k]
2:      N_{R_0}, N_{R_1}, ..., N_{R_{k-1}} ← 0   ▷ Size of arrays R_0, ..., R_{k-1}
3:      N_R ← 0                                   ▷ Size of arrays R
4:      d ← 0                                     ▷ Current depth
5:      L_0, L_1, ..., L_{k-1} ← NULL
6:      for i ← 0 to k − 1 do
7:          d ← d − (N_C[i] − 1)
8:          while N_R ≠ 0 do
9:              N_R ← N_R − 1
10:             (h, ℓ, p, q) ← R[N_R]
                        ▷ R stores unmatched right-visible edges
11:             if h > d then                     ▷ h is deeper than d
12:                 R_p[N_{R_p}] ← (h, ℓ, p, i)
13:                 N_{R_j} ← N_{R_j} + 1
14:                 if N_R == 0 and h > d + 1 then
15:                     L_i ← (d + 1, h − 1, φ, i)
16:                 end if
17:             else
18:                 R_p[N_{R_p}] ← (d + 1, ℓ, p, i)
19:                 N_{R_p} ← N_{R_p} + 1
20:                 R[N_R] ← (h, d, p, φ)
21:                 N_R ← N_R + 1
22:                 Break (exit While loop)
23:             end if
24:         end while
25:         if N_A[i] > 0 then
26:             R[N_R] ← (d + 1, d + N_A[i], i, φ)
27:             N_R ← N_R + 1
28:             d ← d + N_A[i]
29:         end if
30:     end for
31:     while N_R ≠ 0 do
32:         N_R ← N_R − 1
33:         (h, ℓ, p, q) ← R[N_R]
34:         R_p[N_{R_p}] ← (h, ℓ, p, φ)
35:         N_{R_p} ← N_{R_p} + 1
36:     end while
37:     return R_0, ..., R_{k-1}, N_{R_0}, ..., N_{R_{k-1}}, L_0, ..., L_{k-1}
38: end procedure
```

数は $\mathcal{O}(k)$ である．

3.6　Local Tree Contraction for Contracted Trees

前節で求めた $R_0, \cdots, R_{k-1}$ の情報に基づき，k 個のマルチプロセッサは並列に right-visible edge に対応する left-visible edge の取得と部分縮約則による縮約処理を行う．$R_i[0]$（先端が葉である区間）では定数値が求まり，それ以外（内部区間）では，3 つ組 (a, b, c) が求まる（図 10 参照）．すなわち，$\mathcal{O}(k)$ 個の区間のそれぞれは，定数個の要素数で表現される縮約木に縮約される．詳細は省略する．

3.7　Global Tree Contraction

区間の数は $\mathcal{O}(k)$ 個であり，各区間は前節の処理により，定数個の値に縮約される．最後に 1 コアを使って，これら

の区間をマージし，1 つの縮約木を得る．詳細は省略する．

3.8 計算量の解析

$n \geq p^2$ の場合の計算量を解析する．まず，時間計算量について解析する．Local Tree Contraction for Pseudo Trees に関して，b^2 要素に対する処理は $\mathcal{O}(b \log b)$time で行える (処理済み縮約木へのマージ処理に $\mathcal{O}(b \log b)$time かかり，他は $\mathcal{O}(b \log b)$time でできる)．よって，このステップ全体では $\mathcal{O}\left(\frac{n}{k}\frac{b}{b^2}\right) = \mathcal{O}\left(\frac{n}{p}\right)$time となる．また，Local Tree Contraction for Contracted Trees に関しては，非可換演算子による Reduction として計算できるので，Koike ら [3] により提案されているパイプラインアルゴリズムを使用することにより，計算時間は $\mathcal{O}(\frac{n}{kb}) = \mathcal{O}(\frac{n}{p})$time となる．その他の処理は $\mathcal{O}(k)$time で計算できる．よって，時間計算量は $\mathcal{O}(\frac{n \log b}{p})$time となる．

次に I/O 計算量については，どの要素にも高々定数回しかアクセスせず，1 アクセスでは必ず b 要素を取得している．よって，$\mathcal{O}\left(\frac{n}{b}\right)$ となる．また，グローバルメモリ使用量は $\mathcal{O}(n)$ ワード，共有メモリ使用量は $\mathcal{O}(b^2)$ ワードである．よって多重度は $\mathcal{O}\left(\frac{M}{b^2}\right)$ となる．

4. 結論

本論文では，Tree reduction のための GPU アルゴリズムを提案した．Tree reduction は木に対する多くのクエリの一般化になっている．使用する演算子が部分縮約則を満たす時も，なお，多くの問題がカバーされる．本論文では演算子が部分縮約則を満たす時の GPU 向けアルゴリズムを提案し，I/O 計算量が最適となることを示した．今後は，実装評価を行ったのち，木上のパターンマッチングや確率伝播法などの様々な実用的な応用に対し，提案アルゴリズムを適用したい．

参考文献

[1] Patterson, D. A. and Hennessy, J. L.: *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition (2013).

[2] NVIDIA Corporation: NVIDIA CUDA C Programming Guide version 4.2 (2012).

[3] Koike, A. and Sadakane, K.: A Novel Computational Model for GPUs with Applications to Efficient Algorithms, *International Journal of Networking and Computing*, Vol. 5, No. 1, pp. 26–60 (online), available from ⟨http://www.ijnc.org/index.php/ijnc/article/view/96⟩ (2015).

[4] Skillicorn, D. B.: Structured Parallel Computation in Structured Documents, *Journal of Universal Computer Science*, Vol. 3, No. 1, pp. 42–68 (online) (1997).

[5] Miller, G. L. and Reif, J. H.: Parallel Tree Contraction and Its Application, *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, SFCS '85, Washington, DC, USA, IEEE Computer Society, pp. 478–489 (online), DOI: 10.1109/SFCS.1985.43 (1985).

[6] Morihata, A., Matsuzaki, K., Hu, Z. and Takeichi, M.: The Third Homomorphism Theorem on Trees: Downward & Upward Lead to Divide-and-conquer, *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, New York, NY, USA, ACM, pp. 177–185 (online), DOI: 10.1145/1480881.1480905 (2009).

[7] Dekel, E., Ntafos, S. and Peng, S.-T.: Parallel tree techniques and code optimization, *VLSI Algorithms and Architectures* (Makedon, F., Mehlhorn, K., Papatheodorou, T. and Spirakis, P., eds.), Lecture Notes in Computer Science, Vol. 227, Springer Berlin Heidelberg, pp. 205–216 (online) (1986).

[8] Teng, S.-H. and Wang, B.: Parallel algorithms for message decomposition, *Journal of Parallel and Distributed Computing*, Vol. 4, No. 3, pp. 231 – 249 (online), DOI: http://dx.doi.org/10.1016/0743-7315(87)90035-9 (1987).

[9] Miller, G. L. and H. Teng, S.: Tree-Based Parallel Algorithm Design, *Algorithmica*, Vol. 19, No. 4, pp. 369–389 (online), DOI: 10.1007/PL00009179 (1997).

[10] Rao Kosaraju, S. and Delcher, A.: Optimal parallel evaluation of tree-structured computations by raking (extended abstract), *VLSI Algorithms and Architectures* (Reif, J., ed.), Lecture Notes in Computer Science, Vol. 319, Springer New York, pp. 101–110 (online), DOI: 10.1007/BFb0040378 (1988).

[11] Cole, R. and Vishkin, U.: The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time, *Algorithmica*, Vol. 3, No. 1-4, pp. 329–346 (online), DOI: 10.1007/BF01762121 (1988).

[12] Abrahamson, K., Dadoun, N., Kirkpatrick, D. and Przytycka, T.: A simple parallel tree contraction algorithm, *Journal of Algorithms*, Vol. 10, No. 2, pp. 287 – 302 (online), DOI: http://dx.doi.org/10.1016/0196-6774(89)90017-5 (1989).

[13] Morihata, A. and Matsuzaki, K.: A Practical Tree Contraction Algorithm for Parallel Skeletons on Trees of Unbounded Degree, *Procedia Computer Science*, Vol. 4, pp. 7 – 16 (online), DOI: http://dx.doi.org/10.1016/j.procs.2011.04.002 (2011). Proceedings of the International Conference on Computational Science, ICCS 2011.

[14] Reif, J. H.: *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition (1993).

[15] Robie, J. and Dyck, M.: XQuery 3.1: An XML Query Language (2014).

[16] Tarjan, R. E. and Vishkin, U.: Finding Biconnected Componemts And Computing Tree Functions In Logarithmic Parallel Time, *Proceedings of the 25th Annual Symposium onFoundations of Computer Science, 1984*, SFCS '84, Washington, DC, USA, IEEE Computer Society, pp. 12–20 (online), DOI: 10.1109/SFCS.1984.715896 (1984).

[17] Matsuzaki, K., Hu, Z., Kakehi, K. and Takeichi, M.: SYSTEMATIC DERIVATION OF TREE CONTRACTION ALGORITHMS, *Parallel Processing Letters*, Vol. 15, No. 3, pp. 321–336 (2005).

[18] Matsuzaki, K.: Parallel programming with tree skeletons, PhD Thesis, The University of Tokyo, Japan (2007).

[19] Kakehi, K., Matsuzaki, K. and Emoto, K.: Efficient Parallel Tree Reductions on Distributed Memory En-

vironments, *Proceedings of the 7th international conference on Computational Science, Part II*, ICCS '07, Berlin, Heidelberg, Springer-Verlag, pp. 601–608 (online) (2007).

[20] Valiant, L. G.: A bridging model for parallel computation, *Commun. ACM*, Vol. 33, No. 8, pp. 103–111 (online), DOI: 10.1145/79173.79181 (1990).

[21] Emoto, K. and Imachi, H.: Parallel Tree Reduction on MapReduce, *Procedia Computer Science*, Vol. 9, pp. 1827 – 1836 (online), DOI: http://dx.doi.org/10.1016/j.procs.2012.04.201 (2012). Proceedings of the International Conference on Computational Science, ICCS 2012.

[22] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, USENIX Association, pp. 10–10 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1251254.1251264⟩ (2004).