Bernstein 及び A.K.Lenstra らの素因数分解（行列計算ステップ）回路に関する調査報告書

2003 年 1 月 31 日

2003年1月31日

Bernstein 及び A.K. Lenstra らの素因数分解 (行列計算ステップ) 回路に関する調査報告書

1 はじめに

本報告書では文献 [1] 及び [9] で述べられた素因数分解 (行列計算ステップ) 回路に関して調査報告する.

素因数分解問題の困難性に安全性の根拠をおく公開鍵暗号プリミティブの多くは, 一般的な素因数分解アルゴリズムである一般数体ふるい法（GNFS）の計算量を基にして, その安全性が評価される. 最近, そのGNFS を, より効率的に行う方法として専用 H/W を用いる方法が提案された（[1, 9] 参照）. 素因数分解アルゴリズムへの専用 H/W 適用の先行研究としては, [8, 10, 11] があるが, [1] ではそれらに対する優位性も主張されている.

ここでは GNFS で計算負荷の高い部分を, “関係式収集ステップ ”と “行列計算ステップ ”の 2 ステップに大きく分けて記述する. [1] では, 両ステップに対して専用 H/W アプローチの提案が述べられているが, 特に, 行列計算ステップに対してある種のソーティングアルゴリズムを用いる具体的改良法が記述されている. また [1] においては, 同ステップに対してルーティングアルゴリズムの適用可能性も触れられている. その後, [9] で, 同ステップに対しルーティングを基にした専用 H/W が具体的に提案された.

[1, 9] では, GNFS にかかる “スループットコスト”（時間 × 金額）を測度にして, 行列計算ステップへの専用 H/W 使用の有効性を見積もっている. [9] において主張されているように, $1.17 \cdot n$ ビット合成数に対する従来法と n ビット合成数に対する改良法をほぼ同等のスループットコストとする漸近的な見積もりが妥当である.

スループットコストを測度にして評価することには, まだ議論の余地が残されているが, 上述の結果は [1, 9] のアプローチの有効性を示している.

また [9] では, [1] での専用 H/W と比べ, よりコンパクトな H/W 実現法を提案することにより, 1024 ビット合成数に対する行列計算ステップは, （マスク生成費を除いて）約 5 千ドルの専用 H/W を用いれば数時間でできるようになると主張している. 但し, 著者らは上述の評価を “optimistic”な前提でのものであると断り書きをしている. つまり, その見積もりの際に想定されている疎行列（“小行列”）のサイズは, 漸近的なスループットコスト関数を最適化した時に得られる値であり, 実際の 1024 ビット合成数 GNFS に関して述べる時には, 注意を払う必要があるということである.

[9] では, 512 ビット合成数 GNFS に成功した [4] での実際の行列サイズを基に, 従来の漸近関数を使っての 1024 ビット合成数に対する行列サイズ（“大行列”）も見積もっており, また, それに対する実際的なスループットコストも見積もっている. 現在 GNFS 全体で, 関係式収集ステップが律速ステップであることを考慮すると, これら評価値も重要な判断基準になると思われる (詳細は [9] 参照).

[9] に記述された専用 H/W の実現には技術的困難は依然存在すると思われるが, 量子計算機などと比べて, 現時点でも, この専用 H/W の実現可能性は高いと思われる.

[9] においては, この専用 H/W の提案により, 1024 ビット RSA の安全性の根拠は, GNFS 中の関係式収集ステップの計算困難性に集約されたとし, 現時点で 1024 ビット RSA は依然安全であるとしている.

また, 最近の関連研究として [7, 12] がある.

本報告書では, [1, 9] に基づいて行列計算ステップ専用 H/W に関して調査報告する. まず, 2 節でソーティングを基にした Bernstein の回路 (専用 H/W) について述べた後, 3 節でルーティングを基にした A.K. Lenstra らの回路について説明する. 4 節では, 2, 3 節での専用 H/W を用いた時の GNFS の漸近的スループットコストに関して [9] に従って述べる. 5 節でも [9] に従って 1024 ビット素因数分解を目的にした時の専用 H/W (カスタムビルド, FPGA) を用いた行列計算ステップスループットコストを見積もり, 通常の PC を用いた時のスループットコストとの比較に関して述べる.

2 ソーティングを基にしたBernstein の回路

2.1 Schimmler ソーティングアルゴリズム

ソーティングを基にした Bernstein の回路では, Schimmler ソーティングアルゴリズムが用いられるが, そのアルゴリズムの基本構成要素は偶奇転置 (odd-even transposition) アルゴリズムと呼ばれるアルゴリズムであるので, まずその記述より始める.

偶奇転置とは, “(奇-偶) 位置間での転置” と “(偶-奇) 位置間での転置”という 2 ステップの繰り返しよりなるアルゴリズムである. [1] より例を引くと,

Time 0:	8	9	7	9	3	2	3	4
Time 1:	8	9	7	9	2	3	3	4
Time 2:	8	7	9	2	9	3	3	4
Time 3:	7	8	2	9	3	9	3	4
Time 4:	7	2	8	3	9	3	9	4
Time 5:	2	7	3	8	3	9	4	9
Time 6:	2	3	7	3	8	4	9	9
Time 7:	2	3	3	7	4	8	9	9
Time 8:	2	3	3	4	7	8	9	9

という図により表される置換である. ここで, $\frac{a}{c}\ \frac{b}{d}$ は, $c = \min\{a, b\}, d = \max\{a, b\}$ を表す. m 個の要素に対して, m ステップでソーティングが行われる (上記の例では $m = 8$).

この偶奇転置を用いて, Schimmler ソーティングアルゴリズムは, 例えば表 1 のように記述される. 簡単のために m を 2 の冪としておく. Schimmler ソーティングは m^2 個の要素を $m \times m$ のメッシュ状に並べて, それらを $8m - 8$ 回の (並列) ステップでソートするアルゴリズムである. まず, メッシュ上での次の 2 種類の順序 (左から右, 蛇状) に注意する.

左から右: $(1,1), (1,2), \ldots, (1,m), (2,1), \ldots, (2,m), (3,1), \ldots\ldots, (m,m)$

蛇状: $(1,1), \ldots, (1,m), (2,m), \ldots, (2,1), (3,1), \ldots, (3,m), \ldots\ldots,$

再び [1] より例を引いて, 表 1 の左上図のようにメッシュ状に配置された数値列をソートする. 本アルゴリズムは $4 = (2 \times 2)$ ブロックに対して再帰的に行われるので, ステップ 1 の後には各ブロック内ではソートが完了することになる.

3	1	4	1	5	9	2	6		1	1	2	3	2	2	2	3		1	1	0	0	2	2	1	0
5	3	5	8	9	7	9	3		3	3	3	3	4	5	5	6		1	1	2	2	2	2	2	3
2	3	8	4	6	2	6	4		3	4	4	5	6	6	7	7		3	3	3	3	4	4	4	3
3	3	8	3	2	7	9	5		5	8	8	8	9	9	9	9		3	4	3	3	5	5	5	6
0	2	8	8	4	1	9	7		1	1	0	0	2	2	1	0		4	4	4	5	6	6	7	7
1	6	9	3	9	9	3	7		4	4	3	2	5	4	4	3		5	6	5	5	9	8	7	7
5	1	0	5	8	2	0	9		7	6	5	5	9	8	7	7		7	8	8	8	9	9	9	9
7	4	9	4	4	5	9	2		9	9	8	8	9	9	9	9		9	9	8	8	9	9	9	9

$\Longrightarrow$ 各ブロック内ソート (左から右) $\Longrightarrow$ 各列内ソート (上から下) $\Longrightarrow$ 各行内ソート (蛇状)

0	0	0	1	1	1	2	2		0	0	0	1	1	1	1	1		0	0	0	1	1	1	1	1
3	2	2	2	2	2	1	1		3	2	2	2	2	2	2	2		2	2	2	2	2	2	2	3
3	3	3	3	3	4	4	4		3	3	3	3	3	3	3	3		3	3	3	3	3	3	3	3
6	5	5	5	4	3	3	3		4	4	4	5	4	4	4	4		4	4	4	4	4	4	4	5
4	4	4	5	6	6	7	7		6	5	5	5	6	5	5	5		5	5	5	5	5	5	6	6
9	8	7	7	6	5	5	5		7	8	7	7	6	6	7	7		6	6	7	7	7	7	7	8
7	8	8	8	9	9	9	9		9	8	8	8	9	9	8	8		8	8	8	8	8	9	9	9
9	9	9	9	9	9	8	8		9	9	9	9	9	9	9	9		9	9	9	9	9	9	9	9

$\Longrightarrow$ 各列内ソート (上から下) $\Longrightarrow$ 各行内ソート (左から右)

表 1: Schimmler ソーティングの例

ステップ 2 以降は, $4m$ 回で行えることが, 先の偶奇置換のステップ数よりわかり, 今 $m = 2^s$ とすると, この操作が $i = 1, \ldots, s$ にわたって行われることがわかる. つまり, 総ステップ数は $\sum_{i=1}^{s} 4 \cdot 2^i = 4 \cdot \frac{2^{s+1}-2}{2-1} = 8m-8$ であることがわかる.

2.2 ソーティングを基にした回路

次に 2.1 節のソーティングアルゴリズムを用いて, GNFS における “行列計算”を行う方法について述べる. その計算は, 疎な $\mathbb{F}_2$-値の $D^{1+o(1)} \times D^{1+o(1)}$ 行列 A が定める線形変換の非自明な核ベクトル (1 個) を求めることである (ターゲット n が 512, 1024 ビットの時の具体的な D の値については 5.1 節参照). そして, それは Wiedemann (又は Lanczos) のアルゴリズムを用いれば, A と $D^{1+o(1)}$ 行縦ベクトル v との積を約 $D^{1+o(1)}$ 回求めることに (計算量的に dominant という意味で) 帰着される. よって, その行列 · 縦ベクトル積演算を 2.1 節のソーティングアルゴリズムを用いて効率的に実現することを考える.

今までの GNFS での実験報告より, その $\mathbb{F}_2$-値 行列 A の要素中の 1 の総数は $D^{1+o(1)}$ というオーダーとして話を行う. まず, 2 次元メッシュの縦横を $D^{0.5+o(1)}$ として, そのメッシュ上に $\log_2 D$ ビット幅のレジスターを 3 個 (r_1, r_2, r_3) もったプロセッサを配列する. それらプロセッサのことを, 以後 [9] に従いノードと呼ぶ ([1] ではセルと呼んでいる).

まず, r_1 には v 内で値が 1 の位置を保持する. そして, それらと異なるノードの r_2, r_3 に A 内で値が 1 の成分のインデックスを保持する. それらをそれぞれタイプ 1, タイプ 2 のノードと呼ぶことにする. 以下の表では, タイプ 1 ノードは r_1 の値で表し, タイプ 2 ノードは $(r_2$,$r_3)$ で表すこととする.

最初のノードの間の順序関係は, タイプ 1 のノード同士には r_1 の値による順序を入れ, タイプ 2 とタイプ 2 のノード同士には r_3, r_2 という順に比較する辞書式順序を入れる. そして, タイプ 1 とタイプ 2 のノードの間には, それぞれの r_1 と r_3 の値を比較し, それらが等しい場合は, タイプ 1 のノードを小とすることで順序を入れる. それ以外のノードは全てタイプ 1, 2 のノードより小とする (それらをタイプ 0 と呼び以下の表では 0 で表す). このような順序に関して蛇状ソートされた後のメッシュの例を [1] より引用する.

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
(3,3)	(2,3)	(3,2)	(2,1)	(1,1)	1	0	0
(8,3)	5	(1,5)	(4,5)	6	7	(6,7)	(15,7)
(1,12)	12	(13,11)	(1,10)	(1,9)	(8,8)	(2,8)	8
(10,12)	13	(1,13)	(2,13)	14	(1,14)	(3,14)	(4,14)
(11,16)	(3,16)	(2,16)	16	(5,15)	(4,15)	(2,15)	(1,15)

次に, タイプ 1 のノードは, その r_1 の値と等しい r_3 の値を持つタイプ 2 ノードと通信して, その r_2 の値を r_1 に保持する. また, その操作を終えると, タイプ 1 ノードはタイプ 0 になる. 次のような状態となる.

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
(3,3)	(2,3)	(3,2)	2,(2,1)	1,(1,1)	0	0	0
(8,3)	0	1,(1,5)	4,(4,5)	0	0	6,(6,7)	15,(15,7)
1,(1,12)	0	(13,11)	(1,10)	(1,9)	8,(8,8)	2,(2,8)	0
10,(10,12)	0	1,(1,13)	2,(2,13)	0	1,(1,14)	3,(3,14)	4,(4,14)
11,(11,16)	3,(3,16)	2,(2,16)	0	(5,15)	(4,15)	(2,15)	(1,15)

次に, r_1 のみに関する順序を考え, r_1 の値を, タイプ 0 より小としてソートする.

1	1	1	1	1	2	2	2
10	8	6	4	4	3	3	2
11	15	0	0	0	0	0	0
(3,3)	(2,3)	(3,2)	(2,1)	(1,1)	0	0	0
(8,3)	0	(1,5)	(4,5)	0	0	(6,7)	(15,7)
(1,12)	0	(13,11)	(1,10)	(1,9)	(8,8)	(2,8)	0
(10,12)	0	(1,13)	(2,13)	0	(1,14)	(3,14)	(4,14)
(11,16)	(3,16)	(2,16)	0	(5,15)	(4,15)	(2,15)	(1,15)

次に, $\mathbb{F}_2$ 上での行列演算であることから, 蛇状順序で “(奇-偶) 位置”の対同士で r_1 の値を比較し, 同じであれば 0 に置き換えて, 再度 r_1 に関してソートする.

1	2	2	3	3	4	4	6
0	0	0	0	15	11	10	8
0	0	0	0	0	0	0	0
(3,3)	(2,3)	(3,2)	(2,1)	(1,1)	0	0	0
(8,3)	0	(1,5)	(4,5)	0	0	(6,7)	(15,7)
(1,12)	0	(13,11)	(1,10)	(1,9)	(8,8)	(2,8)	0
(10,12)	0	(1,13)	(2,13)	0	(1,14)	(3,14)	(4,14)
(11,16)	(3,16)	(2,16)	0	(5,15)	(4,15)	(2,15)	(1,15)

この時点で, r_1 の重複度は 2 以下であることに注意して, 最後に, r_1 の値が隣接する両側の値と一致したら, 先と同様 0 にしてソートする.

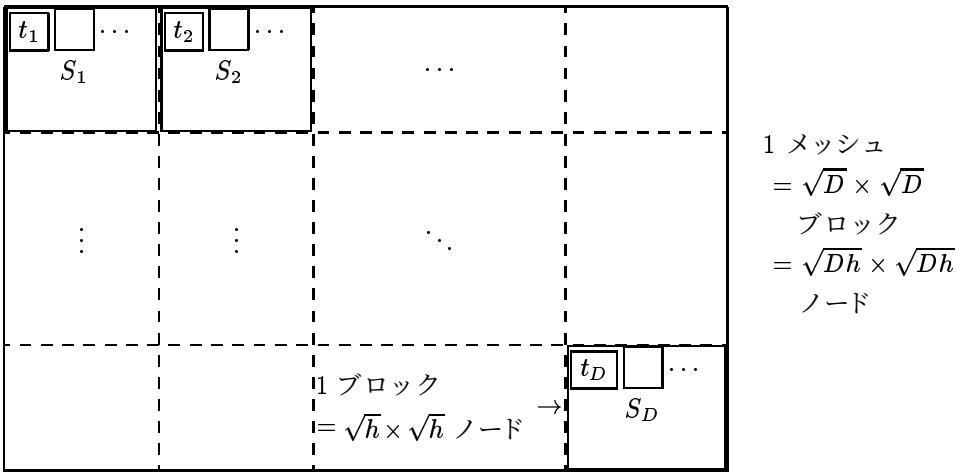
1	6	8	10	11	15	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
(3,3)	(2,3)	(3,2)	(2,1)	(1,1)	0	0	0
(8,3)	0	(1,5)	(4,5)	0	0	(6,7)	(15,7)
(1,12)	0	(13,11)	(1,10)	(1,9)	(8,8)	(2,8)	0
(10,12)	0	(1,13)	(2,13)	0	(1,14)	(3,14)	(4,14)
(11,16)	(3,16)	(2,16)	0	(5,15)	(4,15)	(2,15)	(1,15)

これで, Av の要素が 1 になるインデックスが計算されたと共に, 次の A と Av の乗算を行う (ソーティングされる前の) 初期状態が得られた.

3 ルーティングを基にした A.K. Lenstra らの回路

3.1 基本となる方式

次に A.K. Lenstra らによるルーティングアルゴリズムに基づく Av の計算法について述べる. まず, A の行数 D と一列あたりの 1 の平均総数 h に対し, $m = \sqrt{Dh}(= D^{0.5+o(1)})$ として, サイズが $m \times m$ のメッシュコンピュータを考える. それらをサイズが $\sqrt{h} \times \sqrt{h}$ の D 個のブロック S_i $i = 1, \ldots, D$ に分ける. S_i は列に関する左右順に並んでいるとする. そして, S_i 内の左上隅のノードプロセッサを t_i と書き, “値 i のターゲット”と呼ぶことにする. 各ノードは, $\log_2 D$ ビットのレジスタ $Q[j], R[j]$ を持ち, t_i は更に, 1 ビットを保持できるレジスタ $P[i]$ を持つとする.



メッシュは次のように初期化される. v のインデックス i の値を, t_i の $P[i]$ に順次格納し, A の i 列目で 1 が格納された行インデックスを S_i にあるノードの $Q[\cdot]$ に (任意順に) 格納する. それ以外の $Q[\cdot]$ には nil を格納する (ここで nil は $1, \ldots, D$ の値いずれとも異なる特殊値とする). Av は以下の手順に従い計算される.

1. 全ての i に対し, $P[i]$ の値を t_i から S_i にある全てのノードにブロードキャストする (このブロードキャストは $2\sqrt{h}-2$ ステップで行われる).
2. 全ての i と S_i にある全てのノード j に対して, もし $P[i]$ の値が 1 であるなら, $R[j] \leftarrow Q[j]$ と代入し, そうでないなら $R[j] \leftarrow \mathrm{nil}$ とする.
3. 全ての i に対し, $P[i] \leftarrow 0$ とする.
4. $R[j]$ に対して, メッシュに基づいたパケットルーティングアルゴリズムを行い, nil でない値を保持した $R[j]$ の値を, それが属するターゲット $t_{R[j]}$ に向かってルーティングしていく. 各 t_i はパケットを受け取るたびに, それを破棄して, $P[i]$ の値をフリップする (これにより $\mathbb{F}_2$ の加算が行われる).

パケットルーティングアルゴリズムとしては, “時計回り転置 (clockwise transposition)”アルゴリズムを用いる. 表 2 に示された 4 ステップを繰り返し行うことによりルーティングを行うのであるが, その際, 両矢印で示された 2 ノード $j, j+1$(縦並びなら上下, 横並びなら左右の順) 間で, 次に示す比較-交換が行われる.

1. 現在の $R[j], R[j+1]$ の値を比較し, どちらかが nil でない時にのみ以下の操作を行う.
2. もし $R[j] = R[j+1] (\neq \mathrm{nil})$ であれば, $R[j], R[j+1]$ の値を nil に設定する.
3. $R[j], R[j+1]$ 共に nil でなく, $R[j] \neq R[j+1]$ であれば, $t_{R[j]}$ の列 (又は行) インデックス c_j と $t_{R[j+1]}$ の列 (又は行) インデックス c_{j+1} を比較し, もし $c_j > c_{j+1}$ であれば, $R[\cdot]$ レジスタ間の値を交換する.
4. もし $R[j]$ が nil であれば, $t_{R[j+1]}$ の列 (又は行) インデックス c_{j+1} と $R[j+1]$ が位置する列 (又は行) インデックス d_{j+1} を比較し, もし $c_{j+1} < d_{j+1}$ であれば, $R[\cdot]$ レジスタ間の値を交換する. $R[j+1]$ が nil である場合も同様とする.

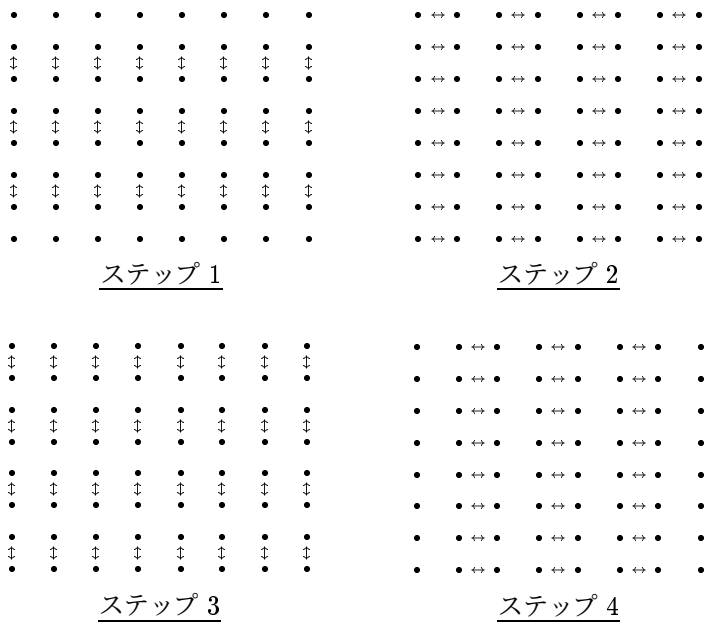


表 2: 時計回り転置アルゴリズム

このアルゴリズムには, ソーティングを基にした回路の時とは異なり, 理論的なステップ数の上限が得られていない. が, [9] では, サイズが 13000×13000 までのランダムなメッシュを用いて実験を行ったところ, そのステップ数が $2m$ を超えないと報告している. ルーティングの最大ステップ数の (自明な) 下限は $2m-2$ であるので, これは非常に効率が良いと主張している. また, ルーティングを基にした回路の方がノードの構造やその動作が簡単であるため, H/W での実現を考慮した場合に有利であると主張している.

また, [13] において, 行列計算ステップに対する 3 次元回路の提案もなされている.

3.2 改良法

[9] では, 更に, 前節で述べた回路の改良の提案が述べられている.

- **ターゲット t_i の密度増加**
  ターゲット t_i の密度を上げることで, 物理的実現可能性を高めることができる. ρ をノード 1 個あたりの $P[\cdot]$ レジスタの平均個数とする. 3.1 節に述べたスキームでは, $\rho = h^{-1}$ となっていた. $P[\cdot]$ レジスタの総数は D に固定されているので, ρ を増加すると, その分メッシュ中の総ノード数は $h\rho$ に比例して小さくなり, 1 回のルーティングにかかる時間を $\sqrt{h\rho}$ に比例して減少させることができる.

  この時 A 内の hD 個の 1 を値とするインデックスすべてを 1 回でルーティングすることができなくなってしまう. この問題は次のようにルーティングプロセスを部分的に逐次化することで解決される.

  ノード毎に 1 個の行列成分を表す $Q[\cdot]$ を保持するかわりに, ノード毎に $h\rho$ 個の $Q[\cdot]$ 値を保持するようにする. 各ノード i には ρ 個の行 $c_{i,1}, \ldots, c_{i,\rho}$ に対応するレジスタ $P[c_{i,1}], \ldots, P[c_{i,\rho}]$ を付属させ, 行 $c_{i,1}, \ldots, c_{i,\rho}$ の中身を $R[i]$ に 1 個ずつ代入する (実際の実装では, 行列のインデックス $Q[\cdot]$ は各々のノードに近接している DRAM バンクにストアされている). 乗算を行うために, 順次 $Q[\cdot]$ レジスタから次の値をとってきては時計回り転置ルーティングを行うということを繰り返す. よって, 乗算 1 回あたりのステップ総数は $\sqrt{h\rho}(= h\rho/\sqrt{h\rho})$ 倍される. ρ が増加すれば, 生産コストは A のインデックスを保持する hD 個の DRAM ノードが支配的になる.

  漸近的にみると, この改善はスループットコスト (4 節参照) を増加させ好ましくない. しかし, このことで, 少しコストが増加したとしても, 全体の占有面積が小さくなるため, 回路の物理的実現可能性を大きく向上させるという利点が存在する (5.2 節参照).

- **ブロック数 $K > 1$ のブロック Wiedemann アルゴリズム**
  ブロック数 $K > 1$ のブロック Wiedemann アルゴリズムを用いることで, 高効率化を図ることができる. [9] では, Wiedemann アルゴリズムを用いる時に $K = 1$ では, $20D$ 回の行列・ベクトル乗算が必要であるのに対し, K を 32 以上に上げることにより, それは $3D$ 回で足りるとしている. また, 行列・ベクトル乗算そのものも次のように効率化される. ブロック Wiedemann アルゴリズム ([6]) では, K 個のベクトル v_i $(i = 1, \ldots K)$ に対して, まず $A^k v_i$ $(i = 1, \ldots K,\ k = 1, \ldots 2D/K)$ が計算されて, その後 (別の)K 個のベクトルに対して同様な積が $k = 1, \ldots D/K$ に対して計算される. これら K 個のベクトルに対する乗算を 1 つの回路で効率的に実現するのである. 3.1 節の方法では, 各ノードは 1 ステップ間に, 高々1 個のメッセージ (行インデックス = ターゲットのアドレス, 長さ $= \log_2(D)$) を隣接ノードに送付していたが, 改良法では, それに更に K ビットのデータを付加して送付する. この付加 K ビット中の 1 に対応するベクトルに関しては, その行インデックスは有効であるであることを示す.

以上の改良を加えたアルゴリズムを記述する (特に $\rho > 1$ とする). まず, $C_j := \{c | 1 \le c \le D, (j-1)\rho \le c - 1 < j\rho\}$ とする ($\{1, \ldots, D\} = \cup_{j \in \{1, \ldots, D/\rho\}} C_j$). 各ノード j は 1 ビットレジスタ $P_i[c], P'_i[c]$ $(i = 1, \ldots K, c \in C_j)$ と $\log_2(D) + K$ ビットレジスタ R_j を持つ. また, ノード j には $Q_j = \{(r, c) | A_{r,c} = 1, c \in C_j\}$ というデータが付随している (下記アルゴリズムでは各 $c \in C_j$ は I_j という変数によってインデックス付けされているとする). 初期状態では, ベクトル v_i の各要素は $P_i[\cdot]$ レジスタに格納されているとする.

1. $\forall i, c$ に対して, $P'_i[c] \leftarrow 0$ とし, $\forall j$ に対して, $I_j \leftarrow 1$ とする.
2. 次を $h\rho$ 回繰り返す.
   (a) $\forall j$ に対して, $(r, c) \leftarrow Q_j[I_j], I_j \leftarrow I_j + 1, R[j] \leftarrow \langle r, P_1[c], \ldots, P_K[c] \rangle$ とする.
   (b) $R[\cdot]$ に対して, $R[j] = \langle r, \ldots \rangle$ が $r \in C_j$ となる t_j にルーティングされるように時計周り転置ルーティングアルゴリズムを行う.
   ここで, ノード j が $r \in C_j$ となるメッセージ $\langle r, p_1, \ldots, p_K \rangle$ を受取ったら, $P'_i[r] \leftarrow P'_i[r] \oplus p_i$ $(i = 1, \ldots K)$ としてそのメッセージを破棄する. また, 2 メッセージ $\langle r, p_1, \ldots, p_K \rangle$ と $\langle r, p'_1, \ldots, p'_K \rangle$ が隣接ノード間で比較されたら, $r \in C_j$ となるノード j から遠い方のメッセージは破棄され, もう一方は $\langle r, p_1 \oplus p'_1, \ldots, p_K \oplus p'_K \rangle$ に置き換えられる (ここで, $p_1 \oplus p'_1, \ldots, p_K \oplus p'_K$ が全て 0 の時には, それらメッセージは nil に置き換えられる).

3. $\forall i, c$ に対して, $P_i[c] \leftarrow P'_i[c]$ とする.

上記ステップを終えた後には $P_i[\cdot]$ には $A^k v_i$ の要素が格納されており, 次の乗算の初期状態となっている. また, $u_j(A^k v_i)$ $(j = 1, \ldots K)$ という内積計算のためにメッシュに更に配線しなければならないが, [9] では, それは低コストに抑えられるとしている. 最後に, $K > 1$ としたために, 3.1 節で述べた方式よりも輻輳が問題になるが, これも [9] では, さしたる効率低下を招かないとしている.

4 漸近的スループットコストの見積もり

2 節及び 3 節での回路を用いた専用 H/W GNFS の効果を測るのに, [1, 9] では, 通常の計算量ではなく, スループットコストというものを用いて議論している. これは VLSI デザインの分野での "AT コスト" (Area $\times$ Time) と呼ばれるものに相当すると [2, 9] にはあり, また Area を "デバイスの値段"と読み替えて, [9] では, 値段と計算時間の積をメジャー (measure) にして最適化を論じている.

以下では, [9] に従い, 専用回路を用いた GNFS を circuit-NFS と略記し, 用いない場合を standard-NFS と呼ぶことにする. そして, standard-NFS を用いる場合でも [5] による効率化を図ったものを improved-NFS (複数多項式を使用して漸近計算量を削減) と呼び, そうでない場合を ordinary-NFS と呼ぶ (4.2 節参照). [3] において, improved-NFS は 1024 ビット (ぐらいの小さい)n に対しては実際的には効率的でないと言及されていることに注意しておく.

細かい議論に入る前に, 行列計算ステップに関しての総オペレーション数 (逐次的計算量) とスループットコストの大雑把な見積もりを述べる. 2 節及び 3 節で見たように $D^{1+o(1)} \times D^{1+o(1)}$ のサイズの行列 A に対して, メッシュの大きさは $m = D^{0.5+o(1)}$ として $m^{1+o(1)} \times m^{1+o(1)}$ で与えられた. A は $D^{1+o(1)}$ 個の非零エントリからなる疎行列であるので, 2.2 節で述べたように従来のアルゴリズムでは, $D^{1+o(1)} \cdot D^{1+o(1)} = D^{2+o(1)}$ が計算量であり, メモリは $D^{1+o(1)}$ 用意しておけばよかった. circuit-NFS では全体の (逐次的) 計算量は $m^{1+o(1)} \cdot D^{1+o(1)} \cdot D^{1+o(1)} = D^{2.5+o(1)}$ であるが, (並列化した) ステップ数は $m^{1+o(1)} \cdot D^{1+o(1)} = D^{1.5+o(1)}$ となる. その金銭コスト ($\approx$ 回路規模) は従来同様 $D^{1+o(1)}$ で見積もられる. つまり表 3 にあるような見積もりが得られる.

	総オペレーション数	ステップ数 (実時間)	装置規模	スループットコスト
standard-NFS	$D^{2+o(1)}$	$D^{2+o(1)}$	$D^{1+o(1)}$	$D^{3+o(1)}$
circuit-NFS	$D^{2.5+o(1)}$	$D^{1.5+o(1)}$	$D^{1+o(1)}$	$D^{2.5+o(1)}$

表 3: 行列計算ステップのコスト (概略)

以下, 4.1 節では, GNFS の漸近計算量の算出法について述べ, それを基にして 4.2 節において, スループットコストに関し最適化した場合のアルゴリズムのスループットコスト見積もりについて述べる.

4.1 GNFS の漸近的計算量

$\alpha, r \in \mathbb{R} (0 \leq r \leq 1)$ に対して $L_x[r; \alpha]$ で $\exp((\alpha + o(1))(\log x)^r (\log\log x)^{1-r})$ $(x \to \infty)$ という漸近評価を満たす $x (> 0)$ の任意の実数値関数とする.

この時, $0 < s < r \leq 1, \alpha > 0, \beta > 0$ という固定されたパラメータに対して,

$$\begin{aligned} & L_x[r-s; -\alpha(r-s)/\beta] \\ &= \quad \mathrm{Prob}[z : L_x[s;\beta]\text{-スムース} \,|\, z \in_{\mathrm{R}} \mathbb{Z} \ \text{ s.t. } \ 0 < z \leq L_x[r;\alpha]] \end{aligned} \tag{1}$$

という関係式が GNFS 計算量評価の基本となる. また, 素因数分解のターゲットとする n に対して, 頻繁に出てくる L_n を L と, $L[1/3; \alpha]$ を $L(\alpha)$ と略記する. これらの記法を用いると, (1) 式の特別な場合が, "$L[2/3; \alpha]$

以下のランダム正整数が $L(\beta)$-スムースになる確率は $L(-\alpha/3\beta)$ である" と表せる．また, "$\zeta = x + o(1)$ $(x \to \infty)$"のことを "$\zeta \stackrel{o}{=} x$"と書くこととする．

まず [9] に従い, standard-NFS の記述を行うことより始める．

$$d \approx \delta(\log n/\log\log n)^{1/3}$$

となる正整数 d に対して, 既知正整数 $\overline{m}(\approx n^{1/(d+1)})$ が $f(\overline{m}) \equiv 0 \mod n$ を満たすような 1 つのモニックでない d 次の非斉次多項式 $f(X) = \sum_{i=0}^{d} f_i X^i$ を考える．ここで, δ は後で調節されるパラメータであり, $f_i \approx n^{1/(d+1)}$ とする．有理サイド, 代数サイドのスムースネス上限をそれぞれ B_r, B_a と書くことにする．[9] では, 後で調節されるパラメータ $\beta(>0)$ を用いて B_r も B_a も共に $L(\beta)$ で与えている．これはもちろん $B_r = B_a$ ということでなく, それらの量を算出する関数が漸近的に $L(\beta)$ となる関数であることを意味している．また, 後で調節されるパラメータ $\alpha(>0)$ を用いて $|a| < L(\alpha), 0 < b < L(\alpha), \gcd(a,b) = 1$ である $(a,b) \in \mathbb{Z}^2$ で $a - b\overline{m}$ が B_r-スムースかつ $b^d f(a/b)$ が B_a-スムースであるものを [9] では "関係式" と単に呼んでいる．[9] では, 漸近評価を主眼にして, それが見易いように, 大素数を用いた場合の関係式の分類やその扱いというようなデリケートなことには触れていない．これら関係式は疎な D 次元 $\mathbb{F}_2$-値ベクトルを与える．ここで D は $\pi(X)$ を x 以下の素数の個数関数として

$$\begin{aligned} D &\approx \pi(B_r) + \#\{(p,r) | p : B_a\text{以下の素数}, f(r) \equiv 0 \mod p\} \\ &\approx \pi(B_r) + \pi(B_a) \end{aligned} \tag{2}$$

で与えられる． 素数定理より $\pi(x) \approx x/\log x$ $(x \to \infty)$ であるので, $\pi(L(\beta)) = L(\beta)$ であって, 更に $2L(\beta) = L(\beta)$ であることに注意すると, (2) 式より $D = L(\beta)$ がわかる．

更に $n^{1/(d+1)} = \exp(\frac{\log n(\log\log n)^{1/3}}{\delta(\log n)^{1/3} + (\log\log n)^{1/3}}) = \exp((1/\delta + o(1))(\log n)^{2/3} \times (\log\log n)^{1/3}) = L[2/3, 1/\delta]$ であることより "一般の関係式" $(a,b) \in \mathbb{Z}^2$ に対して

$$|a - b\overline{m}| = L[1/3, \alpha] n^{1/(d+1)} = L[1/3, \alpha] L[2/3, 1/\delta] = L[2/3, 1/\delta] \tag{3}$$

となる．また, $\log(L(\alpha)^d) = d(\alpha + o(1))(\log n)^{1/3}(\log\log n)^{2/3} = \delta(\log n)^{1/3} \times (\log\log n)^{-1/3}$ $(\alpha + o(1))(\log n)^{1/3}(\log\log n)^{2/3} = \delta(\alpha + o(1))(\log n)^{2/3} \times (\log\log n)^{1/3}$ より $L(\alpha)^d = L[2/3, \alpha\delta]$ がわかり, 従って一般の $(a,b) \in \mathbb{Z}^2$ に対して

$$\begin{aligned} |b^d f(a/b)| &= (d+1) n^{1/(d+1)} L(\alpha)^d \\ &\qquad / * \text{項数} \times f_i\text{の大きさ} \times a^i b^{n-i}\text{の大きさ} * / \\ &= L[2/3, 1/\delta] L[2/3, \alpha\delta] = L[2/3, \alpha\delta + 1/\delta] \end{aligned} \tag{4}$$

が導かれる．$a - b\overline{m}$ と $b^d f(a/b)$ がスムースネスに関して独立な確率分布をもつという仮定の下で, 上述した (1) 式の特別な場合の見積もりを適用すれば, $a - b\overline{m}, b^d f(a/b)$ 共に $L(\beta)$-スムースになる確率は, (3), (4) 式より

$$L(-\frac{1/\delta}{3\beta}) \cdot L(-\frac{\alpha\delta + 1/\delta}{3\beta}) = L(-\frac{\alpha\delta + 2/\delta}{3\beta}) \tag{5}$$

となる．関係式を見つけるための (a,b) の探索空間の大きさは $2L(\alpha)^2 = 2L(2\alpha) = L(2\alpha)$ であるから GNFS に成功するためには,

$$L(2\alpha) \cdot L(-\frac{\alpha\delta + 2/\delta}{3\beta}) = L(\beta) \quad (= D)$$

でなければならない．これより

$$\alpha \stackrel{o}{=} \frac{3\beta^2 + 2/\delta}{6\beta - \delta} \tag{6}$$

が導出される．パラメータ α, β, δ 間の (6) 式が [9] での議論の基本となる．

次に上記のパラメータを用いて計算量の見積もりを行うと, 関係式導出ステップは $L(2\alpha)$ の計算量であることが見て取れる. 行列計算ステップは, 行列 A の行数が $D = L(\beta)$ であり, その各列のハミング重みは $L(0)$ と見積もれるので, A と $\mathbb{F}_2$-値ベクトルとの積を $O(D)$ 回行うことを考えると, 全体の計算回数は $O(D) \cdot L(0) \cdot L(\beta) = L(2\beta)$ となる. 通常は, 計算量という観点から $L(\alpha) + L(\beta)$ を最小化するように (6) 式を用いて, α, β, δ の値が決定される. 結論として, $\alpha = \beta$ の時が最適なパラメータとなり, その時は $\alpha = \beta = 2/3^{2/3}, \delta = 3^{1/3}$ となる. また, その時の計算量は $L(4/3^{2/3}) = L((64/9)^{1/3}) = L(1.922\ldots)$ であることがわかる.

4.2 漸近的スループットコストの最適化

[9] では, standard-NFS のスループットコストを小さくするようにパラメータ設定を行っており, そのために行列計算のスループットコストを $L(3\beta)$ として, $L(2\alpha) \stackrel{o}{=} L(3\beta)$ という式を満たす最適なパラメータが決定されている (表 3 参照). ここで, 関係式収集ステップのスループットコストを $L(2\alpha)$ としていること, つまり, ふるいなしで小素因数分解は定数ステップとして見積もっていることに注意.

結論として, [9] では $\delta \stackrel{o}{=} 2/3^{1/3} = 1.386\ldots, \alpha \stackrel{o}{=} 3^{2/3}/2 = 1.040\ldots, \beta \stackrel{o}{=} 3^{-1/3} = 0.693\ldots$ というパラメータが得られている. この時, 関係式収集ステップ及び行列計算ステップのスループットコストは共に, $L(3^{2/3}) = L(2.080\ldots)$ で与えられることがわかる (表 4 参照).

また, circuit-NFS で, スループットコストに関し最適化すると, その パラメータは $\delta \stackrel{o}{=} (5/3)^{1/3}(6/5) = 1.422\ldots, \alpha \stackrel{o}{=} (5/3)^{1/3}(5/6) = 0.988\ldots, \beta \stackrel{o}{=} (5/3)^{1/3}(2/3) = 0.790\ldots$ で与えられることがわかる. この時, 関係式収集ステップ及び行列計算ステップのスループットコストは共に, $L((5/3)^{4/3}) = L(1.976\ldots)$ で与えられる (表 4 参照).

次に [5] でなされた複数多項式を用いた方式 (improved-NFS) について, スループットコスト見積もりを行う. α, δ をそれぞれ以前と同じように “ふるい範囲パラメータ”, “多項式次数パラメータ”とする. B_r, B_a を定めるパラメータには以前とは異なり, $B_r = L(\beta), B_a = L(\gamma)$ と 2 個のパラメータを用いる. そして, d 次で, 法 n で $\overline{m}$ を根に持つ $B_r/B_a = L(\beta - \gamma)$ 個の整係数多項式のある集合を G で表す. 今回は対 (a, b) s.t. $\gcd(a, b) = 1,\ b > 0$ が関係式であるとは, $a - b\overline{m}$ が B_r-スムースで, 少なくとも 1 個の $g \in G$ に対して, $b^d g(a/b)$ が B_a-スムースになる時と定義する.

漸近計算量を最適化するようにパラメータを選ぶと, $\delta \stackrel{o}{=} 1.401\ldots, \alpha = \beta \stackrel{o}{=} 0.9509\ldots, \gamma \stackrel{o}{=} 0.8259\ldots$ となり, その計算量は $L(1.901\ldots)$ で与えられる ([5]). improved-NFS を用いるための必要条件 $\beta \geq \gamma$ を満たして, improved-NFS をスループットコストに関して最適化したパラメータを選んでも, ordinary-NFS の最適スループットコストより良いスループットコストが得られないので, [1] ではスループットコストに対し最適化しない standard-NFS の評価を行うには, improved-NFS が用いられるが, スループットコストに対し最適化して standard-NFS 又は circuit-NFS の評価を行うのには, ordinary-NFS をアルゴリズムのベースにして評価を行っている.

また, improved-NFS で, 関係式収集ステップの計算時間のみに着目した時のその時間見積りは $L(1.868\ldots)$ で与えられる. よって, circuit-NFS において, もし行列計算ステップを無視してコストの見積もりを行うなら, そのコストは $L(1.868\ldots)$ になると [9] では述べている.

表 4 で, スループットコスト欄に着目し, circuit-NFS の有効性を見る. スループットコストに対して最適化していない standard-NFS と比較すると, ターゲット数 n のビット長が $\left(\frac{2.852\ldots}{1.976\ldots}\right)^3 = 3.009\ldots$ 倍の時の circuit-NFS が, 従来の standard-NFS と同等のスループットコストで実行可能であることがわかる. また, スループットコストに対して最適化した standard-NFS と比較すると, n のビット長が $\left(\frac{2.080\ldots}{1.976\ldots}\right)^3 (< 1.17)$ 倍の時の circuit-NFS が, 従来法と同等のスループットコストで実行可能なことがわかる. [1] では, 先の $3.009\ldots$ という数字が見積もられているが, [9] では standard-NFS のパラメータ α, β, δ をスループットコスト関数に最適化し直すことで後者の 1.17 という数字に訂正されることを示している.

以下, 5 節では, 1024 ビットをターゲットにした時に, standard-NFS (ordinary-NFS) 時に生じる関係式行列 ($\beta = 0.961\ldots$) を “大行列”と呼び, circuit-NFS 時の行列 ($\beta = 0.790\ldots$) を “小行列”と呼ぶ (行列 A

	総オペレーション数	関係式収集		行列計算		スループットコスト
		装置規模	実時間	装置規模	実時間	
standard-NFS:						
ふるいあり	$L(1.90)$	$L(0.95)$ RAM	$L(1.90)$	$L(0.95)$ RAM	$L(1.90)$	$L(2.85)$
ふるいなし		$L(0)$				
スループットコストに対し最適化 (ふるいなし)	$\underline{L(2.08)}$	逐次: $L(0)$	$L(2.08)$	$L(0.69)$ RAM	$L(1.39)$	$L(2.08)$
		並列: $L(0.69)$ プロセッサ	$L(1.39)$			
circuit-NFS:	$\underline{L(1.98)}$	逐次: $L(0)$	$L(1.98)$	$L(0.79)$ メッシュ	$L(1.19)$	$L(1.98)$
		並列: $L(0.79)$ プロセッサ	$L(1.19)$			

表 4: NFS のコスト

のサイズが $D \times D, D = L(\beta)$ であったことに注意).

5 $n = 1024$ ビット 時の専用 H/W のスループットコスト見積もり

5.1 行列 A の大きさ

実際に 512 ビットの素因数分解を行った [4] によると, その行列 A の行数 D は約 670 万行で, 行内平均密度 h は 63 であった. これを規準にして 1024 ビット circuit-NFS で現れる "小行列"のサイズを見積もる. まず, 1024 ビットの時の "大行列"のサイズを $L(2/3^{2/3})$ という漸近関数より出す (以下に出てくる漸近関数パラメータについては, 4.2 節参照).

$$6\ 700\ 000 \cdot \frac{L_{2^{1024}}[1/3, 2/3^{2/3}]}{L_{2^{512}}[1/3, 2/3^{2/3}]} \approx 1.8 \cdot 10^{10}$$

ここで, [9] ではこの "大行列"の行内平均密度を約 100 と仮定して進めている. そして, この約 10^{10} という行列のサイズを 4.2 節で得られたパラメータに従って, circuit-NFS に最適な行列のサイズに直すと,

$$1.8 \cdot 10^{10} \cdot \frac{L_{2^{1024}}[1/3, (5/3)^{1/3}(2/3)]}{L_{2^{1024}}[1/3, 2/3^{2/3}]} \approx 8.7 \cdot 10^{7} \tag{7}$$

が得られる. [9] では, 以降で, "小行列"のサイズを $4 \cdot 10^7$ と仮定し, 更にその行内平均密度も約 100 と仮定して進めている.

[4] での経験値より, 512 ビット時の関係式収集計算を 1GHz PC 1 台で 8 年で行えるとして, standard-NFS での 1024 ビットでの計算時間見積りを出すと

$$8 \cdot \frac{L_{2^{1024}}[1/3, 4/3^{2/3}]}{L_{2^{512}}[1/3, 4/3^{2/3}]} \approx 6 \cdot 10^{7} \quad (\text{年})$$

となり, これを "小行列"を導出する時間に計算し直すと,

$$6 \cdot 10^{7} \cdot \frac{L_{2^{1024}}[1/3, (5/3)^{4/3}]}{L_{2^{1024}}[1/3, 4/3^{2/3}]} \approx 3 \cdot 10^{8} \quad (\text{年})$$

と見積もられる. つまり, 約 5 倍の時間を要するようになるのであるが, いずれにしても不可能な時間である.

5.2 節において, 上記の “小行列”又は “大行列”を用いた場合の, 以下の各項目の実装による行列計算ステップの具体的な金銭コストと実行時間を見積もる.

- ソーティング回路を用いた専用 H/W
- ルーティング回路を用いた専用 H/W
- (ルーティング回路を用いた) FPGA 実装
- 従来法 (ordinary-NFS の S/W 実装)

5.2 H/W パラメータとスループットコスト見積もり比較

2 節, 3 節で述べた方法を比較するために 3.2 節で述べたアルゴリズムのパラメータ ρ, K 以外にも H/W に関するパラメータを設定しておく必要がある. それらは以下のものである.

- $\mathcal{A}_t, \mathcal{A}_f, \mathcal{A}_d$ は, それぞれ論理トランジスタ 1 個, フリップフロップ 1 個, DRAM ビット 1 個の (ウェハ内平均) 占有面積を表す.
- $\mathcal{A}_w$ はウェハ 1 枚の面積.
- $\mathcal{A}_\rho$ はウェハ間接続パッドの占有面積.
- $\mathcal{C}_w$ はウェハ 1 枚 (FPGA の場合は FPGA チップ 1 枚) の金銭コスト.
- $\mathcal{C}_d$ はウェハ外にある DRAM ビット 1 個あたりの金銭コスト (FPGA の時のみ関係あり).
- $\mathcal{T}_d$ はウェハ外にあるメモリ DRAM ビットへのアクセス速度 (FPGA の時のみ関係あり).
- $\mathcal{T}_l$ は回路内の (平均) シグナル速度 (論理素子間, 単位配線間など).
- $\mathcal{T}_p$ はウェハパッド間での 1 ビット送信速度

[9] では, 比較プラットフォームとして, 表 5 で与えられる 3 種類の H/W パラメータを想定しており, カスタム 1, カスタム 2 は各々“論理 (logic) ウェハ”又は “DRAM ウェハ”での実現を想定したパラメータと呼ばれている. 使用 FPGA には Altera Stratix EP1S25F1020C7 が想定されており, FPGA の場合, 表 5 での $\mathcal{A}_t, \mathcal{A}_w$ の値は $\mathcal{A}_f$ を 1 に正規化した相対値が記述されている. また, パラメータ $\mathcal{A}_p, \mathcal{A}_d$ は, FPGA 実装では不要であり, パラメータ $\mathcal{C}_d, \mathcal{T}_d$ はカスタム H/W 実装では不要であるので, $\emptyset$ と記載されている.

	カスタム 1	カスタム 2	FPGA
$\mathcal{A}_t$	$2.38\mu\mathrm{m}^2$	$2.80\mu\mathrm{m}^2$	0.05
$\mathcal{A}_f$	$19.00\mu\mathrm{m}^2$	$22.40\mu\mathrm{m}^2$	1.00
$\mathcal{A}_d$	$0.70\mu\mathrm{m}^2$	$0.20\mu\mathrm{m}^2$	$\emptyset$
$\mathcal{A}_p$	$4000\mu\mathrm{m}^2\times$ 秒	$4000\mu\mathrm{m}^2\times$ 秒	$\emptyset$
$\mathcal{A}_w$	$6.36{\cdot}10^{10}\mu\mathrm{m}^2$	$6.36{\cdot}10^{10}\mu\mathrm{m}^2$	25660
$\mathcal{C}_w$	\$ 5000	\$ 5000	\$ 150
$\mathcal{C}_d$	$\emptyset$	$\emptyset$	$\$\ 4{\cdot}10^{-8}$
$\mathcal{T}_d$	$\emptyset$	$\emptyset$	$1.1{\cdot}10^{-11}$ 秒
$\mathcal{T}_p$	$4{\cdot}10^{-9}$ 秒	$4{\cdot}10^{-9}$ 秒	$2.5{\cdot}10^{-9}$ 秒
$\mathcal{T}_l$	$1.46{\cdot}10^{-11}$ 秒/μm	$1.80{\cdot}10^{-11}$ 秒/μm	$1.43{\cdot}10^{-9}$ 秒

表 5: H/W パラメータ

それぞれの H/W に関するスループットコストの比較表を [9] より抜粋して, 表 6, 7 で与える. 5 列目は, カスタム H/W に関しては使用ウェハ数を, FPGA に関しては使用チップ数を, PC での S/W 実装に関しては使用 PC 数を表す. また, そこでの “金銭コスト” には, ウェーハ初期コスト (マスク生成費 $\leq$ \$1M) を除いた額を示す. アルゴリズムの項目 (1 列目) の “木構造”に関しては, [9] の補遺 B.3 を参照.

また, 表 6, 7 の 8 列目で (*) という印がついた値は, それぞれの H/W タイプでのスループットコストに関する最適値を表す.

その見積もり過程の詳細に関しては, [9] を参照することとし省略する. 表 6 の 2 行目にあるように, 小行列に対しては [9] の改良によって, “\$5000 $\times$ 6.1 時間” という低スループットコストで実現されることもさ

アルゴリズム	H/W タイプ	ρ	K	ウェハ数/チップ数/PC 数	金銭コスト ($)	実行時間 (秒)	スループットコスト ($ × 秒)
ルーティング	カスタム 1	0.51	107	19	94600	1440 (24 分)	$1.36 \cdot 10^8$(*)
ルーティング	カスタム 2	42.10	208	1	5000	21900 (6.1 時間)	$1.10 \cdot 10^8$(*)
ルーティング	カスタム 2	216.16	42	0.37	2500	341000 (4 日)	$8.53 \cdot 10^8$
ルーティング	カスタム 1	0.11	532	288	1440000	180 (3 分)	$2.60 \cdot 10^8$
ルーティング	FPGA	5473.24	25	64	13800	15900000 (184 日)	$2.20 \cdot 10^{11}$(*)
ルーティング	FPGA	243.35	60	2500	380000	1420000 (17 日)	$5.40 \cdot 10^{11}$
ソーティング	カスタム 1		1	273	4100000	1210000 (14 日)	$4.96 \cdot 10^{12}$
ソーティング	カスタム 1		$\gg 1$	273	4100000	182000 (50 時間)	$7.44 \cdot 10^{11}$
シリアル	PC		32	1	4460	125000000 (4 年)	$5.59 \cdot 10^{11}$
木構造	PC		32	66	24000	2290000 (27 日)	$5.52 \cdot 10^{10}$

表 6: $n = 1024$ ビット時の “小行列”に対する行列計算コスト見積もり

ることながら, 専用 H/W が 1 ウェハのみで設計可能であるということが注目すべき点であると [9] では主張されている. なぜなら, ウェハ間接続のような物理的困難を回避でき, その実現可能性を高めるからである.

アルゴリズム	H/W タイプ	ρ	K	ウェハ数/チップ数/PC 数	金銭コスト ($)	実行時間 (秒)	スループットコスト ($ × 秒)
ルーティング	カスタム 1	0.51	136	6030	30.1M	$5.04 \cdot 10^6$ (58 日)	$1.52 \cdot 10^{14}$(*)
ルーティング	カスタム 2	4112	306	391	2.0M	$6.87 \cdot 10^7$ (2.2 年)	$1.34 \cdot 10^{14}$(*)
ルーティング	カスタム 2	261.60	52	120	0.6M	$1.49 \cdot 10^9$ (47 年)	$8.95 \cdot 10^{14}$
ルーティング	カスタム 1	0.11	663	9000	500.0M	$6.40 \cdot 10^5$ (7.4 日)	$2.88 \cdot 10^{14}$
ルーティング	FPGA	17757.70	99	13567	3.5M	$3.44 \cdot 10^{10}$ (1088 年)	$1.19 \cdot 10^{17}$(*)
ルーティング	FPGA	144.41	471	$6.6 \cdot 10^6$	1000.0M	$1.14 \cdot 10^9$ (36 年)	$1.13 \cdot 10^{18}$
シリアル	PC		32	1	1.3M	27 万年	$1.16 \cdot 10^{19}$
木構造	PC		3484	813	153M	$3.17 \cdot 10^8$ (10 年)	$4.84 \cdot 10^{16}$

表 7: $n = 1024$ ビット時の “大行列”に対する行列計算コスト見積もり

参考文献

[1] D.J. Bernstein, “Circuits for integer factorization: a proposal”, preprint, available at `http://cr.yp.to/papers.html`.

[2] D.J. Bernstein, “Circuits for integer factorization”, web page, July 2002; `http://cr.yp.to/nfscircuit.html`.

[3] J.P. Buhler, H.W. Lenstra, Jr., C. Pomerance, “Factoring integers with the number field sieve”, *The development of the number field sieve, LNM* vol. 1554, Springer, 1993, 50-94.

[4] S. Cavallar, B. Dodson, A.K. Lenstra, W. Lioen, P.L. Montgometry, B. Murphy, H.J.J. te Riele, et.al., “Factorization of a 512-bit RSA modulus”, *Eurocrypt 2000, LNCS* vol. 1807, Springer, 2000, 1-17.

[5] D. Coppersmith, “Modifications to the number field sieve”, *J. Cryptology*, **6**(1993), 169-180.

[6] D. Coppersmith, “Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm”, *Math. Comp.*, **62**(1994), 333-350.

[7] W. Geiselmann, R. Steinwandt, "A dedicated sieving hardware", *PKC 2003, LNCS* vol. 2567, Springer, 2003, 254-266.

[8] A.K. Lenstra, A. Shamir, "Analysis and optimization of the TWINKLE factoring device", *Eurocrypt 2000, LNCS* vol. 1807, Springer, 2000, 35-52.

[9] A.K. Lenstra, A. Shamir, J. Tomlinson, E. Tromer, "Analysis of Bernstein's Factorization Circuit", *Asiacrypt 2002, LNCS* vol. 2501, Springer, 2002, 1-26.

[10] C. Pomerance, J.W. Smith, R. Tuler, "A pipeline architecture for factoring large integers with the quadratic sieve algorithm", *SIAM Journal on Computing*, **17**, 1988, 387-403.

[11] A. Shamir, "Factoring large numbers with the TWINKLE device", *CHES, LNCS*, vol.1717, Springer, 1999, 2-12.

[12] A. Shamir, E. Tromer, "Factoring large numbers with the TWIRL device (preliminary draft)", available at `http://psifertex.com/download/twirl.pdf`.

[13] M.J. Wiener, "The full cost of cryptanalytic attacks", accepted for publication in *J. Cryptology*.