FPGA と協調動作する準同型暗号ライブラリの構築

代表研究者　　川　村　一　志　　東京工業大学科学技術創成研究院　特任助教

1 はじめに

本研究調査では、秘密計算の実用化に向け、準同型暗号ライブラリと FPGA を協調動作させるための演算処理ならびに通信処理のアクセラレータ開発に取り組んだ。準同型暗号を用いると、暗号化された電子データに対して復号することなく任意の処理を施すことが可能となるため、秘密計算による安全性の高いクラウドサービスを実現できる。一方で、準同型暗号を用いて暗号化した秘密情報は非常に巨大なデータとなることから暗号文演算にかかる時間が大きく、実用性に乏しい。図 1 は秘密計算の仕組みと課題を示す。

本研究では、秘密計算に用いられる暗号文演算の高速化を達成するため、FPGA（機能の書き換えが可能なハードウェア）を活用したアクセラレーションに取り組んだ。FPGA を用いて暗号文演算の高速化を実現するためには、演算処理の高速化のみならず、CPU と協調動作させるための通信処理の高速化も必要となる。そのため、本研究では演算処理と通信処理のそれぞれに対しアクセラレータを開発した。

本研究調査では、以下の手順に従ってアクセラレータ開発を遂行した。(1)既存の準同型暗号ライブラリを実際に動作、解析することでボトルネック演算を特定し、(2)ボトルネック演算処理のアクセラレータ、ならびに(3)CPU/FPGA 間の通信処理のアクセラレータを設計、実装した。(1)については、既存の準同型暗号ライブラリ HElib [1,2]に 16bit 比較演算処理を実装し、実行時間を解析した。暗号空間上での 16bit 比較演算では乗算オペレータが 252 回呼び出されており、そのオペレータ中に含まれる Bluestein FFT 演算がボトルネックとなっていることを確認した。高速フーリエ変換（FFT: Fast Fourier Transform）を発展させたアルゴリズムである Bluestein FFT は通常の FFT よりも複雑な処理を必要とする反面、標本数を任意に設定可能な特長を持つ。乗算オペレータは HElib において最頻の演算子であることから、Bluestein FFT の高速化により HElib を用いた様々な演算処理の高速化が期待される。また、他の準同型暗号ライブラリ FHEW [3,4]、TFHE [5,6]においても同様の傾向が見られたことから、Bluestein FFT に対するアクセラレータを構築することで各種ライブラリの高速化に寄与することができる。

本報告書では、2 章で Bluestein FFT アクセラレータの設計手法を示し、3 章で CPU/FPGA 間の通信処理アクセラレータの設計手法を示す。前者が手順(2)の内容に相当し、後者が手順(3)の内容に相当する。

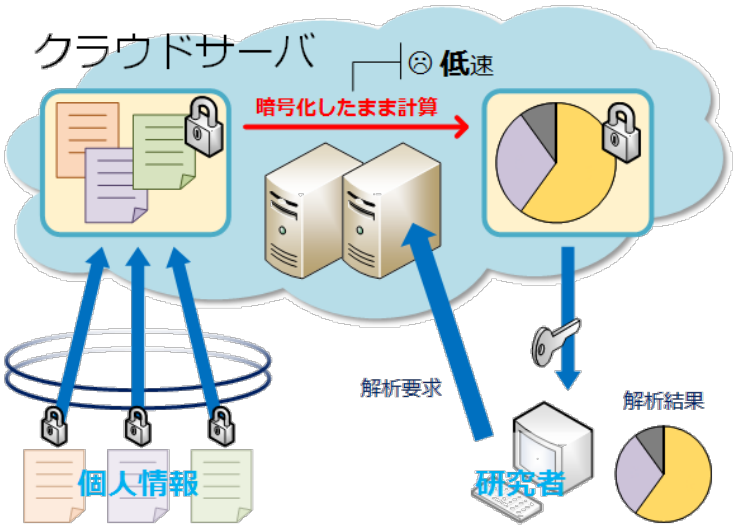


図 1 ：秘密計算の仕組みと課題．秘密計算の仕組みを用いると，個人情報を保護したうえで統計的な解析が可能となる．一方，その解析は現状非常に低速であり，計算処理の高速化が求められる．

2 演算処理アクセラレータの設計

2-1 アクセラレータの構成

標本数が N の Bluestein FFT では、N 個の信号 x(n)を N 個の信号 X(k)に変換する。このとき、下記のように式変形を施すことで、変換式を線形畳み込みの形に帰着させることができる。

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} \qquad \left(W_N = e^{-\frac{2\pi i}{N}}\right)$$

$$\Leftrightarrow X(k) = \sum_{n=0}^{N-1} x(n) W_N^{\frac{n^2}{2}+\frac{k^2}{2}-\frac{(n-k)^2}{2}} = W_N^{\frac{k^2}{2}} \sum_{n=0}^{N-1} x(n) W_N^{\frac{n^2}{2}} W_N^{-\frac{(n-k)^2}{2}} = W_{2N}^{k^2} \sum_{n=0}^{N-1} x(n) W_{2N}^{n^2} W_N^{-(n-k)^2}$$

$$\Leftrightarrow X(k) = \beta_k^* \sum_{n=0}^{N-1} \alpha_n \beta_{n-k} = \beta_k^* (\alpha_k * \beta_k) \qquad \left(\alpha_n = x(n) W_{2N}^{n^2}, \beta_n = W_{2N}^{-n^2}\right)$$

ここで、線形畳み込み（$\alpha_k * \beta_k$）の計算はフーリエ変換とその逆変換、ならびに要素ごとの乗算の組み合わせにより実現されるが、ここでのフーリエ変換に用いる標本数は M≧2N-1 を満たす任意の M とすることができる。したがって、M の値を 2 の冪乗に設定（M=2^m）すれば、任意の N に対して x(n)から X(k)を効率的に計算可能となる。Bluestein FFT が標本数を任意に設定可能でありながら高速に処理できる要因は、標本数が 2 の冪乗ではないフーリエ変換を 2 の冪乗のフーリエ変換（すなわち FFT）へと帰着する上記の手順による。しかしながら、同程度の標本数の FFT に比べてその計算量は数倍にも及ぶため、Bluestein FFT アクセラレータの性能向上が必要となる。

図２は線形畳み込み（$\alpha_k * \beta_k$）の計算を標本数が M の FFT モジュールならびに逆 FFT モジュールを用いて実現した場合の構成図を表す。本研究では、従来手法にて単一のハードウェアモジュールとして設計されていた FFT モジュールと逆 FFT モジュールを、図３に示すように各々複数モジュールに切り分けて設計することで、パイプライン効率を格段に向上させる。具体的には、標本数が M=2^m の FFT（逆 FFT）処理全体を m 個の細粒度モジュールとして個別設計し、それらを相互に接続させて m 段のパイプラインステージから成る大モジュールを構成する。以降、細粒度モジュール内部の設計について、その設計手法を示す。

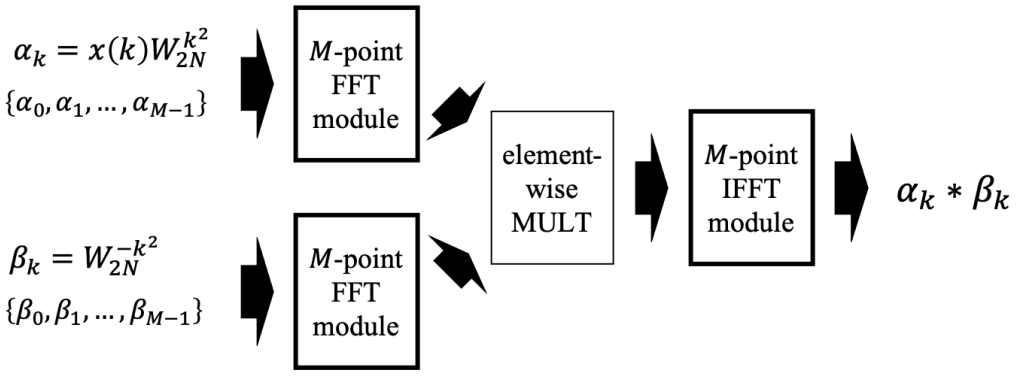


図２：線形畳み込み（$\alpha_k * \beta_k$）の計算を標本数が M の FFT モジュールと逆 FFT モジュールを用いて実現する場合の構成図．通常，M は M≧2N-1 を満たし M=2^m で表すことのできる最小の値とする．例えば N=8191 に対しては M≧16381 となるため，標本数 M=16384 とする．

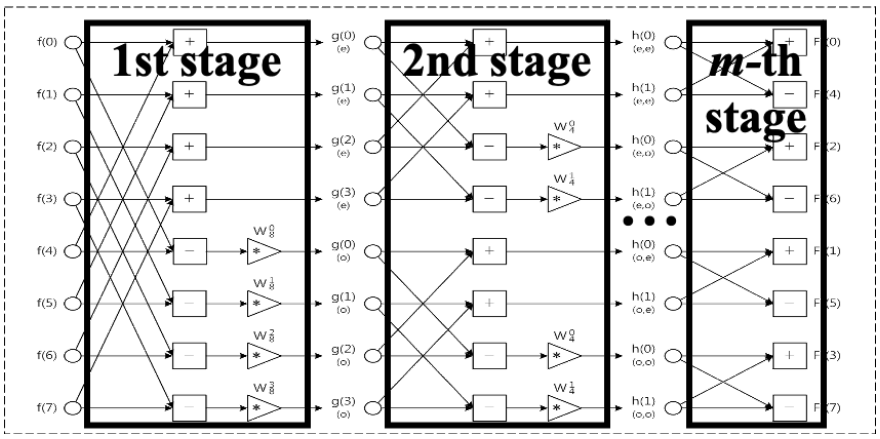


図３：標本数が M=2^m の FFT を対象とした細粒度モジュール設計．FFT を単一モジュールではなく m 個の処理に分割し，m 段のパイプラインステージから成るモジュールとして設計する．

2-2 モジュール設計

FFT は基本演算として加算、減算、乗算を含み、その中でも乗算が処理時間のボトルネックとなる。すなわち、乗算処理を効率的に実行可能なハードウェア設計が重要である。FFT の処理フローをよく観察してみると、乗算は減算とセットになった差積演算（$(a-b)\times c$）の形で登場することから、差積演算ユニットの効率的な設計により乗算処理を効率的に実行可能になるものと考えた。本研究調査では、基本的な論理回路として知られる「セレクタ」に着目した差積演算ユニットの効率的な設計に取り組み、それにもとづいて標本数が M の FFT モジュールを設計した。

まず、$(a-b)\times c$の形式で表される差積演算に対し、各変数を n ビット変数と見なしてビットレベルで式変形を施す。以下では、変数$\boldsymbol{x}$の i ビット目をx_iと表記している。

$$(a-b)\times c = ac + b(-c)$$

$$= \sum_{i=0}^{n-2}\sum_{j=0}^{n-2}\left(a_i c_j + b_i \overline{c_j}\right)2^{i+j} + a_{n-1}2^{n-1}\left(-\sum_{i=0}^{n-2} c_i 2^i\right) + c_{n-1}2^{n-1}\left(-\sum_{i=0}^{n-2} a_i 2^i\right) + b_{n-1}2^{n-1}\left(-\sum_{i=0}^{n-2} \overline{c_i} 2^i\right)$$

$$+ \overline{c_{n-1}}2^{n-1}\left(-\sum_{i=0}^{n-2} b_i 2^i\right) - b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i + (a_{n-1}c_{n-1} + b_{n-1}\overline{c_{n-1}})2^{2n-2}$$

$$= \sum_{i=0}^{n-2}\sum_{j=0}^{n-2}\left(a_i c_j + b_i \overline{c_j}\right)2^{i+j} + \sum_{i=0}^{n-2} b_i 2^i + \sum_{i=0}^{n-2}\left\{(b_{n-1}c_i + a_{n-1}\overline{c_i}) + \left(\overline{a_i}c_{n-1} + \overline{b_i}\overline{c_{n-1}}\right)\right\}2^{n+i-1}$$

$$+ (a_{n-1}c_{n-1} + b_{n-1}\overline{c_{n-1}} - a_{n-1} - c_{n-1} - b_{n-1} - \overline{c_{n-1}})2^{2n-2} + (a_{n-1}+1)2^{n-1}$$

ここで、上の第 2 式は変数$a\sim c$をビットレベル表現することで得られ、第 3 式は第 2 式に登場する負の項を 2 の補数表現を用いて書き直すことで得られる。第 3 式に登場する2^{2n-2}項の係数は以下のように変形できることから、

$$\begin{aligned} &a_{n-1}c_{n-1} + b_{n-1}\overline{c_{n-1}} - a_{n-1} - c_{n-1} - b_{n-1} - \overline{c_{n-1}} \\ &= (a_{n-1}c_{n-1} - a_{n-1} - c_{n-1} + 1) - 1 + (b_{n-1}\overline{c_{n-1}} - b_{n-1} - \overline{c_{n-1}} + 1) - 1 \\ &= \overline{a_{n-1}c_{n-1}} + \overline{b_{n-1}}c_{n-1} - 2 \end{aligned}$$

最終的に以下の式を得る。

$$(a-b)\times c = \sum_{i=0}^{n-2}\sum_{j=0}^{n-2}\left(a_i c_j + b_i \overline{c_j}\right)2^{i+j} + \sum_{i=0}^{n-2} b_i 2^i + \sum_{i=0}^{n-2}\left\{(b_{n-1}c_i + a_{n-1}\overline{c_i}) + \left(\overline{a_i}c_{n-1} + \overline{b_i}\overline{c_{n-1}}\right)\right\}2^{n+i-1}$$
$$+ \left(\overline{b_{n-1}}c_{n-1} + \overline{a_{n-1}c_{n-1}}\right)2^{2n-2} + a_{n-1}2^{n-1} + 2^{n-1} - 2^{2n-1}$$

本式から、各変数がnビットの差積演算では部分積が$2n^2+n+2$項存在する。一方、本式中の$xz+y\bar{z}$の形で表される項はセレクタを用いて圧縮可能（詳細は後述）であり、これにより部分積がn^2+n+2項に削減される。結果として、セレクタを用いて差積演算ユニットを構成すれば部分積がn^2個削減され、効率的な差積演算処理が実現される。図４は４ビットの差積演算の部分積を表すが、点線により仕切られた２個の部分積はセレクタを用いて１個の部分積へと圧縮可能であり、この場合、部分積の個数が38個から22個へと削減される。式を見れば明らかな通り、nが大きいほど部分積の削減割合は向上する。

差積演算ユニットを用いたFFTモジュールの設計について説明する前に、セレクタを用いて$xz+y\bar{z}$の形で表される項（部分積）を圧縮可能な理由を説明する。セレクタは２ビットの入力信号x,yを１ビットの制御信号zによって選択するものであり、その出力結果が$xz+y\bar{z}$となる。これは１ビットの制御信号zが１の場合に出力信号がxと等しくなり、0の場合に出力信号がyと等しくなることを意味し、したがって出力信号は必ず１ビットとなる。$xz+y\bar{z}$が２個の部分積から構成されるにも関わらず結果が１ビットである（２ビットにならない）ことから、部分積の圧縮が可能となる。

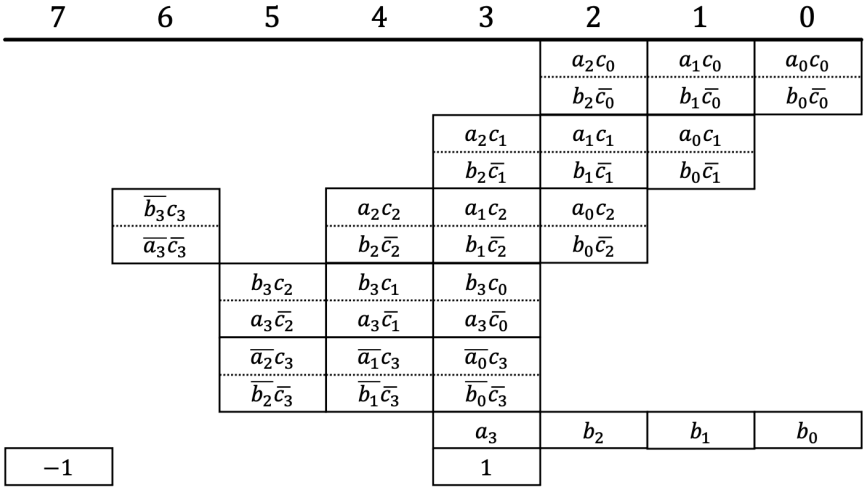


図４：4ビット変数$a\sim c$の差積演算（$(a-b)\times c$）の部分積．点線により仕切られた２個の部分は$xz+y\bar{z}$の形で表され，セレクタを用いて１個の部分積へと圧縮可能である．結果として，n=4ビットの差積演算では部分積をn^2=16個削減できる．

続いて、これまでに説明した差積演算ユニットをベースにFFTモジュールを構成する手法について説明する。FFTはBluestein FFT同様、N個の信号x(n)をN個の信号X(k)に変換するが、その標本数Nは２の冪乗に限定される。以下では、標本数Nが２の冪乗となる場合の変換式を対象とする。

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{nk}\qquad\left(W_N=e^{-\frac{2\pi i}{N}}\right)$$

$$\Leftrightarrow X(k)=\sum_{n=0}^{\frac{N}{2}-1}x(n)W_N^{nk}+\sum_{n=\frac{N}{2}}^{N-1}x(n)W_N^{nk}=\sum_{n=0}^{\frac{N}{2}-1}\left\{x(n)+W_N^{\frac{N}{2}k}x\left(n+\frac{N}{2}\right)\right\}W_N^{nk}$$

上では、変換式の和を前半部分と後半部分に分割している。ここで、Nは２の倍数であることから、$k=2m$の場合と$k=2m+1$の場合に分けて考えると、

$$X(2m) = \sum_{n=0}^{\frac{N}{2}-1} \left\{ x(n) + W_N^{Nm} x\left(n + \frac{N}{2}\right) \right\} W_{\frac{N}{2}}^{nm} = \sum_{n=0}^{\frac{N}{2}-1} \left\{ x(n) + x\left(n + \frac{N}{2}\right) \right\} W_{\frac{N}{2}}^{nm}$$

$$X(2m+1) = \sum_{n=0}^{\frac{N}{2}-1} \left\{ x(n) + W_N^{N\left(m+\frac{1}{2}\right)} x\left(n + \frac{N}{2}\right) \right\} W_{\frac{N}{2}}^{n\left(m+\frac{1}{2}\right)} = \sum_{n=0}^{\frac{N}{2}-1} \left\{ x(n) - x\left(n + \frac{N}{2}\right) \right\} W_N^n W_{\frac{N}{2}}^{nm}$$

となり、それぞれが標本数 N/2 の変換式へと帰着される。ここで、N が 2 の冪乗である点に着目すると、上記の過程を繰り返し適用可能であり、最終的には標本数 2 の変換式を導くことができる。上記の過程を 1 回適用することで標本数が半分になることから、標本数が$M = 2^m$の場合、同過程をm回繰り返すことで最小標本数（標本数 2）の変換式が導かれる。結果的に、ナイーブに処理する場合に O(N^2)必要であった変換式の計算量を O(logN)にまで削減でき、高速な変換処理が実現される。例として、標本数が 8 の FFT 処理を書き下してみると、以下のようになる。

$$X(0) = \{(x(0) + x(4)) + (x(2) + x(6))\} + \{(x(1) + x(5)) + (x(3) + x(7))\}$$
$$X(1) = \{(x(0) - x(4))W_8^0 + (x(2) - x(6))W_8^2\} + \{(x(1) - x(5))W_8^1 + (x(3) - x(7))W_8^3\}$$
$$X(2) = \{(x(0) + x(4)) - (x(2) + x(6))\}W_4^0 + \{(x(1) + x(5)) - (x(3) + x(7))\}W_4^1$$
$$X(3) = \{(x(0) - x(4))W_8^0 - (x(2) - x(6))W_8^2\}W_4^0 + \{(x(1) - x(5))W_8^1 - (x(3) - x(7))W_8^3\}W_4^1$$
$$X(4) = \left\{\{(x(0) + x(4)) + (x(2) + x(6))\} - \{(x(1) + x(5)) + (x(3) + x(7))\}\right\}W_2^0$$
$$X(5) = \left\{\{(x(0) - x(4))W_8^0 + (x(2) - x(6))W_8^2\} - \{(x(1) - x(5))W_8^1 + (x(3) - x(7))W_8^3\}\right\}W_2^0$$
$$X(6) = \left\{\{(x(0) + x(4)) - (x(2) + x(6))\}W_4^0 - \{(x(1) + x(5)) - (x(3) + x(7))\}W_4^1\right\}W_2^0$$
$$X(7) = \left\{\{(x(0) - x(4))W_8^0 - (x(2) - x(6))W_8^2\}W_4^0 - \{(x(1) - x(5))W_8^1 - (x(3) - x(7))W_8^3\}W_4^1\right\}W_2^0$$

これらの式の中には多数の差積演算が含まれている。一方、入力信号 x(n)は複素数で表されることから、セレクタを用いて FFT を実現するためにはさらに式変形が必要となる。例えば、X(1), X(3), X(5), X(7)に含まれる$(x(1) - x(5))W_8^1$に着目すると、

$$x(1) = a + bi,\ x(5) = c + di,\ W_8^1 = e + fi,\ (x(1) - x(5))W_8^1 = A + Bi$$
$$A = (a - c)e + (d - b)f$$
$$= \sum_{i=0}^{n-2}\sum_{j=0}^{n-2}\left\{\left(a_i e_j + c_i \overline{e_j}\right) + \left(d_i f_j + b_i \overline{f_j}\right)\right\}2^{i+j} + \sum_{i=0}^{n-2}(c_i + b_i)2^i$$
$$+ \sum_{i=0}^{n-2}\left\{(c_{n-1}e_i + a_{n-1}\overline{e_i}) + \left(b_{n-1}f_i + d_{n-1}\overline{f_i}\right) + (\overline{a_i}e_{n-1} + \overline{c_i}\overline{e_{n-1}}) + \left(\overline{d_i}f_{n-1} + \overline{d_i}\overline{f_{n-1}}\right)\right\}2^{i+n-1}$$
$$+ \left\{\left(\overline{b_{n-1}}c_{n-1} + \overline{a_{n-1}c_{n-1}}\right) + \left(\overline{d_{n-1}}f_{n-1} + \overline{d_{n-1}f_{n-1}}\right)\right\}2^{2n-2} - 2^{2n} + (a_{n-1} + d_{n-1})2^{n-1} + 2^n$$

として出力の実部Aが得られる。この表現によりセレクタを用いた部分積の圧縮が第 1 項、第 3 項、第 4 項の括弧内に適用可能である。同様にして出力の虚部$B = (a - c)f + (b - d)e$に対しても部分積の圧縮が可能であり、また、$(x(1) - x(5))W_8^1$以外の差積演算に対しても同様の帰着が可能である。ただし、標本数が 8 の FFT における処理時間のボトルネックは$(x(1) - x(5))W_8^1$および$(x(3) - x(7))W_8^3$であることが実験的に確認されている。これは、$W_8^0 = W_4^0 = W_2^0 = 1$および$W_8^2 = W_4^1 = -i$であることから、上記において$e = 0$または$f = 0$が成り立ち、計算が単純化されることに起因する。そこで、FFT モジュールの構成にあたっては、その計算過程に登場する差積演算のうち、W_P^0および$W_P^{P/4}$が登場「しない」箇所にのみセレクタを用いた差積演算ユニットを利用することとする。例として、標本数が 8 の場合の FFT モジュールを図示すると図 5 のようになる。標本数が 16 以上になった場合にも同様の方法にて FFT モジュールを設計する。

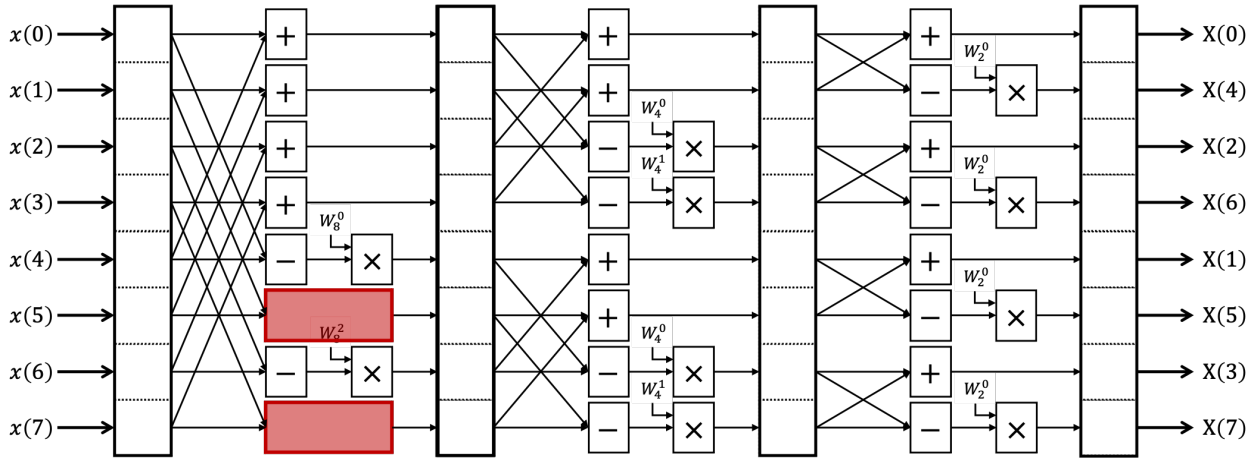


図５：標本数８の FFT モジュール．赤枠が処理時間のボトルネックとなる差積演算であり，これらの箇所をセレクタベースの差積演算ユニットとして構成する．

3 通信処理アクセラレータの設計

3-1 アクセラレータの要件

CPU/FPGA 間の効率的なデータ通信を実現するためには、優先度キュー（PQ: Priority Queue）を効率的に実装することが求められる。優先度キューはよく知られた抽象データ型のひとつであり、有名なところではグラフ探索アルゴリズムやネットワークスイッチ等に用いられてきた。優先度キューに格納される各要素はデータと優先度をペアで持ち、以下に示す基本操作を備える。

- 要素の挿入（Enqueue）：キューに要素を加える。
- 要素の削除（Dequeue）：最大の優先度を持つ要素をキューから削除する。

優先度キューにおけるデータの格納と基本操作を効率的に実現する目的で FPGA アクセラレータが提案されているものの、既存の FPGA アクセラレータにはいくつかの課題が存在する。第一に、優先度キューに格納しておくことのできる要素数が少ない点、第二に、実用的な優先度キューが備えるべき 2 つの性質を実装することが難しい点が挙げられる。ここで 2 つの性質とは、①同じ優先度を持つ 2 個の要素に対しては挿入された順に従って優先度が与えられ、②完全に満たされた状態の優先度キューに要素を挿入する際には最も優先度の低い要素が削除されることを指す。既存の実装で最も有力かつ頻繁に用いられるヒープベースの優先度キューは、FPGA 上のメモリブロックを利用して大規模なキューを実装できる一方、先に示した 2 つの性質を満たすことが難しい。その背景にはヒープベースの優先度キューが最大の優先度を持つ要素にのみ着目した実現方式となっている点がある。故に、これらの性質を満たした大規模な優先度キューを FPGA 上に実現するためには、異なる実現方式に着目する必要がある。

この目的に合致した優先度キューの実現方式のひとつに、循環リストベースの優先度キューがある。ところが、本実現方式にもとづいてアクセラレータを構築する場合、素直に設計するだけでは基本操作の実行効率が非常に悪く、アクセラレータとしての役割を果たせない。したがって、基本操作の性能向上が必須要件となる。本研究では、以下に示す条件を満足させつつ、基本操作の性能向上を目指す。

1) 各要素の優先度は自然数で表され、値の大きさで優先度を示す。
2) 等しい優先度を持つ 2 つの要素がキュー内に存在するとき、後から挿入された要素ほど高い優先度を持つ。
3) 完全に満たされた状態のキューに要素を挿入するとき、キュー内で最も優先度の低い要素が自動的に追い出される。

3-2 アクセラレータの設計

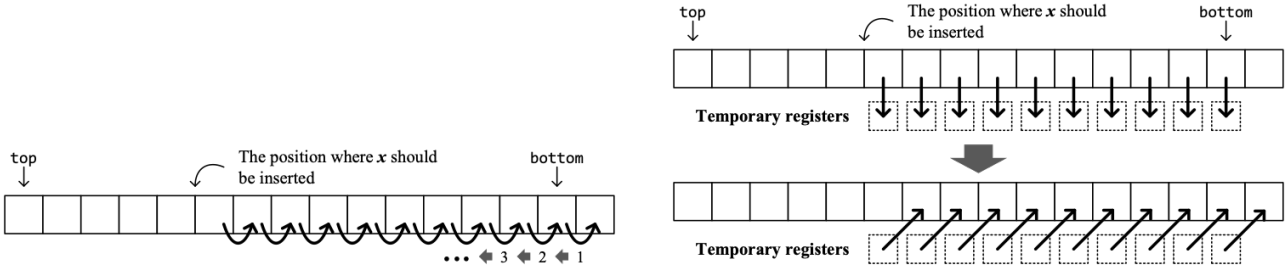
本研究では、循環リストベースの優先度キューを想定し、これにもとづくアクセラレータ設計を工夫することで性能向上を目指す。

循環リストベースの優先度キューに対する基本操作（要素の挿入、削除）はアルゴリズム 1 ならびにアルゴリズム 2 に従って実現される。要素の挿入や削除を実行する間、2 個のポインタ（top, bottom）を用いて優先度キューの状態を監視する。ここで、ポインタ top は先頭の最大優先度を持つ要素を、ポインタ bottom は末尾の最小優先度を持つ要素を示す。アルゴリズム 1 にあるように、新しい要素が挿入される際には既にキュー内に存在していた要素と順次比較をおこなうことで、キュー内に存在する要素が常に優先度順に並ぶようにする必要がある。この比較操作は逐次的におこなわれる必要があることから、要素の挿入にはキュー内の要素数に比例した時間を要し、実行効率が悪い。一方、アルゴリズム 2 が示すように、最大優先度の要素を削除する際にはポインタ top を変更するだけでよく、極めて効率的に実現できる。すなわち、アルゴリズム 1 が効率的に実現されるようアクセラレータを設計する必要がある。

アルゴリズム 1：循環リストベースの優先度キューに対する要素の挿入
入力：優先度キュー**Q**、ポインタ top, bottom、新しい要素 x
出力：優先度キュー**Q**、ポインタ top, bottom
01. s = bottom, d = bottom + 1
02. d が top と等しくなるか、x の優先度が **Q**[s]の優先度よりも小さくなるまで 3-4 行目を繰り返し
03. **Q**[d] = **Q**[s]
04. s = s − 1, d = d − 1
05. **Q**[d] = x, bottom = bottom + 1

アルゴリズム 2：循環リストベースの優先度キューに対する要素の削除
入力：優先度キュー**Q**、ポインタ top, bottom
出力：優先度キュー**Q**、ポインタ top, bottom、削除される要素 x
01. x = **Q**[top], top = top + 1

ここでは、アルゴリズム 1 を FPGA 上で効率的に実行させるため、要素の挿入に関わる操作のうち、要素の比較（アルゴリズムの 02 行目に相当）と要素の移動（アルゴリズムの 03 行目に相当）を分離し、図６(b)に示すように要素の移動を並列に実行することを考える。本報告書では、これを「並列シフト法」と呼ぶ。並列シフト法を採用する場合、新しい要素の挿入位置を逐次探索ではなく二部探索で探すことができるようになり、キュー内の要素数 n に対して $O(\log_2 n)$ の時間でアルゴリズムを完遂できる。一方で、順次比較を用いる図６(a)の方法ではアルゴリズムの実行に $O(n)$ の時間を要する。しかしながら、並列シフト法では多数の要素を同時にメモリから読み出し、同時にメモリへと書き込む必要がある。FPGA 上のメモリブロックに対して読み書き可能なデータの個数は高々2 個程度であるため、並列シフト方をそのまま実現することは現実的でない。そこで、k 個のメモリブロックに対してキュー内の要素を周期的に配置することにより、並列シフト法を k 並列で実現する手法を提案する。提案手法では k の値を様々変更することで、並列化による恩恵と並列化を導入するうえで必要なオーバーヘッドとのトレードオフを考慮した設計が可能になる。



(a) 従来手法：順次シフト　　(b) 提案手法：並列シフト

図６：メモリに格納された要素の移動戦略．従来手法(a)では要素の移動を順次繰り返すのに対し，提案手法(b)では複数の要素を同時に読み出し，同時に書き込む．提案手法では新しい要素の挿入位置を事前に探索する必要があるが，二部探索により $O(n)$ ではなく，$O(\log_2 n)$ の時間で実行可能になるものと期待される．

アルゴリズム 3：循環リストベースの優先度キューに対する並列シフト法を用いた要素の挿入
入力：優先度キュー**Q**、ポインタ top, bottom、新しい要素 x、並列度 k
出力：優先度キュー**Q**、ポインタ top, bottom
01. t = top, b = bottom + 1, h = floor((b + t) / 2)
02. t と b が等しくなるまで 03-04 行目を繰り返し
03. x の優先度が **Q**[h]の優先度以上ならば b = h、そうでなければ t = h + 1
04. h = floor((b + t) / 2)
05. shift = bottom − t + 1
06. bH = floor((bottom + 1) / k), bL = (bottom + 1) − bH * k
07. s = bottom, d = bottom + 1
08. 09-10 行目を bL 回繰り返し（事前処理）
09. **Q**[d] = **Q**[s]
10. s = s − 1, d = d − 1
11. shift = shift − bL
12. cH = floor(shift / k), cL = shift − cH * k
13. 14-17 行目を cH 回繰り返し
14. i = 0 から k − 1 に対し 15-16 行目を並列実行
15. tmp(i) = **Q**[s − i]
16. **Q**[d − i] = tmp(i)
17. s = s − k, d = d − k
18. 19-20 行目を cL 回繰り返し（事後処理）
19. **Q**[d] = **Q**[s]
20. s = s − 1, d = d − 1
21. **Q**[d] = x, bottom = bottom + 1

循環リストベースの優先度キューに対し、並列度が k の並列シフト法を用いて要素の挿入を実行する手順をアルゴリズム 3 に示す。本アルゴリズムを FPGA 上で実現するには、キュー内の各要素を k 個のメモリブロックに周期的に配置しておく必要がある。これにより、番地が連続する k 個の要素への同時アクセスを許容し、k 並列の並列シフト法を実現できる。例えば、図 7 に示すキューに対し k = 4 でアルゴリズム 3 を実行する場合、図 8 に示すように 4 個のメモリブロックへの配置がおこなわれる。これに対して 4 個の一時レジスタ（tmp(0)～tmp(3)）を用意しておけば、4 個の連続する要素を同時並行で読み書き可能である。

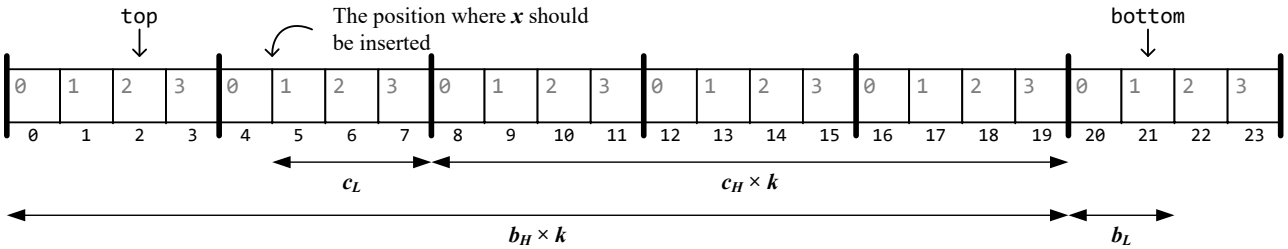


図 7 ：キューの例．キュー下部に示した数値はメモリ番地を表し，キュー内部に示した数値は k = 4 でアルゴリズム 3 を実行する場合のメモリブロックインデックスを表す．本図にはアルゴリズムを実行させた際に登場する各種変数（bH, bL, cH, cL）が例示されている．

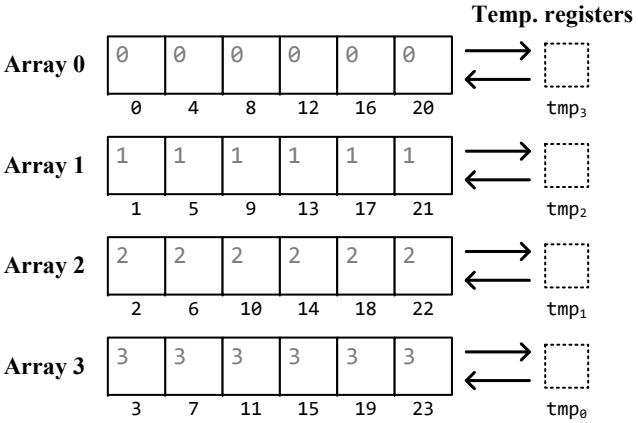


図８：4 個のメモリブロックに対する周期的な要素の配置.

アルゴリズム 3 は以下のように前半と後半に分けられる。

- 前半（01-05 行目）：二分探索を用いて要素の挿入位置を探索する。
- 後半（06-21 行目）：要素の移動と新しい要素の配置をおこなう。

前半の手順にて求まった shift は後半に移動させる要素の個数を表す。後半の要素の移動では、k 並列の並列シフト法を効率的に実現するために事前処理（08-10 行目に相当）と事後処理（18-20 行目に相当）が必要となる。事前処理は各メモリブロックと一時レジスタを一対一に対応させる役割を持ち、FPGA 上に展開した際にマルチプレクサの使用を抑制する効果がある。また、事後処理は並列シフト法で要素を移動させた際に余ってしまった k 個より少ない要素を移動させる役割を持つ。13-17 行目は並列シフト法の本体に相当する処理であり、与える k が大きいほど同時に移動させる要素数が増える。しかしながら、事前処理と事後処理で移動させる要素はワーストケースで k-1 個あり、両方ともの処理が図６(a)に示すような順次移動となることから、k が大きいほどこれらの処理に余分な時間を消費する。故に、最適な k の設定値は自明にはならず、並列数の増加に伴う並列シフトの高速化と事前・事後処理のオーバーヘッドとのバランスを考える必要がある。k の値に応じたアクセラレータ性能の評価は 2.3 節にて議論する。事前処理や事後処理を導入するうえで必要となる各種変数（bH, bL, cH, cL）は図７に例示されているが、このうち bL, cH, cL はそれぞれ事前処理、並列シフト、事後処理にて移動させる要素数を表している。

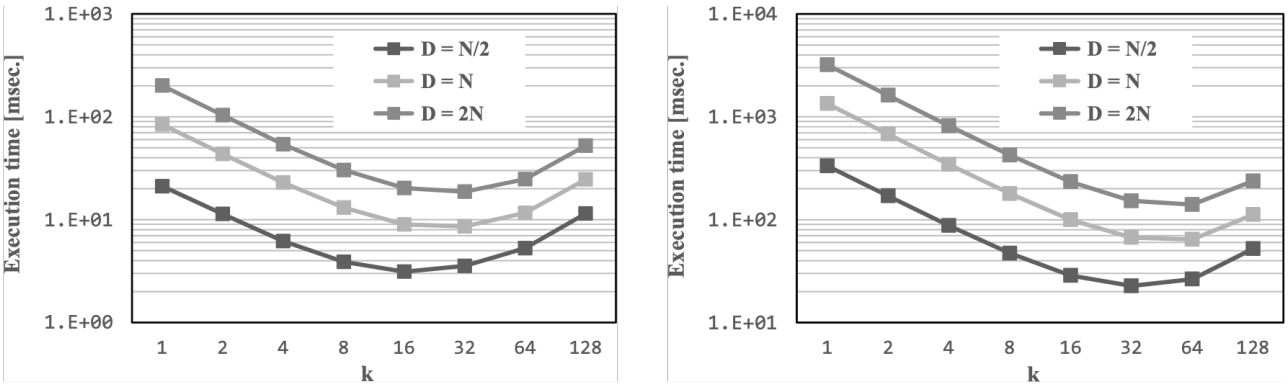
3-3 アクセラレータの実装と評価

本研究調査では、基本操作（要素の挿入と削除）を備えた循環リストベースの優先度キューを、アルゴリズム 3 ならびにアルゴリズム 2 に従って FPGA に実装し、通信処理アクセラレータを構成した。C++で記述したアルゴリズムをもとに高位合成ツールを用いてハードウェア化し、Xilinx 社の FPGA ボード Zynq-7000 上に実装した。キューに挿入される各要素はデータと優先度をペアで持つが、それぞれのフィールドは 32bit で表される。アクセラレータ評価のため、優先度キューに格納可能な最大要素数は 4096 と 16384 の 2 通りを想定し、それぞれの要素数に対して k = 1, 2, 4, 8, 16, 32, 64, 128 とした場合の実装を用意した。なお、k = 1 の場合は並列シフト法による恩恵をまったく受けないため、アルゴリズム 3 ではなく、アルゴリズム 1 に従った実装となっている。これらのアクセラレータに対し、以下の 2 ステップで要素の挿入と削除を実行し、その実行に要した時間を測定した。

1. 要素の挿入：優先度をランダムに設定した D 個の要素を順に優先度キューに挿入する。ただし、途中でキューが一杯になった場合には、先に示した条件に従って最も低い優先度を持つ要素を自動的に追い出す。
2. 要素の削除：優先度キューから完全に要素がなくなるまで要素を削除する。

最大要素数を 4096 に設定した優先度キューの実行時間を図９(a)に、最大要素数を 16384 に設定した優先度キューの実行時間を図９(b)に示す。グラフの横軸が並列度 k を、縦軸が実行時間を表す。今回の実験では、最大要素数 N に対して D = N/2, N, 2N としたときの実行時間を測定した。ここで、D = 2N では要素を挿入する過程で半数の要素が自動的に追い出される。図９(a)から、最大要素数が 4096 の場合には k = 16 の場合が最適であり、図９(b)から、最大要素数が 16384 の場合には k = 32 の場合が最適であることが確認で

きる。このように、最適な k の値は優先度キューの最大要素数に依存し、一般には最大要素数が大きいほど大きな k が最適になる可能性が高いと考えられる。最大要素数を 16384、並列度 k = 32 とした場合、ベースライン実装（k = 1）に対して 22.8 倍の高速化を達成した。



(a)　　(b)

図 9 ：実行時間の比較．(a)最大要素数が 4096 の場合．(b)最大要素数が 16384 の場合．

【参考文献】

[1] S. Halevi and V. Shoup, “Algorithms in HElib,” in *Annual Cryptology Conference*, pp. 554-571, 2014.

[2] HElib, https://github.com/shaih/HElib

[3] L. Ducas and D. Micciancio, “FHEW: bootstrapping homomorphic encryption in less than a second,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 617-640, 2015.

[4] FHEW, https://github.com/lducas/FHEW

[5] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, “TFHE: fast fully homomorphic encryption over the torus, Journal of Cryptology, vol. 33, no. 1, pp. 34-91, 2020.

[6] TFHE, https://github.com/tfhe/tfhe

〈発　表　資　料〉

題　　名	掲載誌・学会名等	発表年月
Implementation of a ROS-based autonomous vehicle on an FPGA board	International Conference on Field-Programmable Technology	December 2019
FPGA-based Heterogeneous Solver for Three-Dimensional Routing	Asia and South Pacific Design Automation Conference	January 2020