IoT機器に適したFPGAを用いた文字列分割の高速化

大和　一洋 *
ミラクル・リナックス株式会社

2016/12/1

概 要

本稿は、IoT ゲートウェイやデータセンターのサーバーに適する FPGA を用いた文字列分割の高速化についての研究成果を報告する。ザイリンクス社の高位合成コンパイラを用いたプロトタイプは、200MHz で動作し、クロック毎に最大 32 の ASCII 文字を処理する。それは、PCI Express インターフェイスを介して、ホストコンピュータと FPGA ボード間でデータ転送を行うために OpenCL と自製のフレームワーク (Volvox) のいずれかを利用する。評価では、Volvox を使ったプロトタイプによる処理が、CPU による処理よりも約 10 倍高速であることが示された。

1　はじめに

IoT (Internet of Things) 機器の数は、2020 年までに 208 億台に達すると予想されている [1]。膨大な数の機器を扱うためのキーとなるコンポーネントのひとつは、IoT ゲートウェイである。それは、データを機器から収集してインターネットサーバーに転送することに加えて、集約、フォーマット変換、不要なデータの破棄などの前処理も行う。IoT ゲートウェイは、データセンター以外にも様々な場所に設置されることが想定される。多くの機器からのトラフィックを処理しなければならないにもかかわらず、そこでは、設置スペース、冷却、電力などの制約はより厳しい。ネットワーク・プロトコルやフォーマットには HTTP や JSON のようにテキスト形式のものが多くあるので、トラフィックにはそれらテキスト文字列が含まれる。また、近年、小型機器でも動作する軽量プログラム言語 (例えば、Python や Ruby) を IoT 機器に用いる場合、入出力にテキストを使用するケースもある。そのため、効率的なテキスト処理は、優れた IoT ゲートウェイの重要な要件のひとつとなる。

一方、データセンターでも IoT 機器からの大量のトラフィックを処理しなければならない。また、テキストを使用する機会も多い。例えば、(Hadoop [2] を用いたような) 収集データの分散バッチ処理、自然言語の構文解析、AI (Artificial Intelligence) との対話、多量のサーバーのログ解析などである。

FPGA (Field Programmable Gate Array) は、これらの課題に対する有効な解決策のひとつと考えられる。インテル社は、2020 年までに 1/3 のクラウドを構成するサーバーに FPGA が搭載されると述べている [3]。FPGA を使うと、ユーザーは自身で目的に応じて回路を構成できる。良好な設計がなされると、高い性能と電力効率が得られる [4]。しかしながら、一般的なソフトウェアエンジニアや運用管理者がそれらを使いこなすことは難しい。なぜなら、その設計には、デジタル回路の知識をはじめ、システムバスや OS などのコンピュータに関する広範な知見が必要だからである。加えて、アプリケーションプログラムが、構築された FPGA 内の回路を使用するように新規開発または修正を行う必要もある。

我々は、GNU C Library (glibc) [5] のような基本的なライブラリから FPGA を透過的に使用する OS を提供することで、これらの課題を解決することを計画している。この方式の優位点は、アプリケーションの修正が不要なことである。また、最近の CPU と FPGA を統合するトレンドによって、このコンセプトは Intel プラットホームで ARM でも実現可能である。インテル社は、FPGA 技術の先進的な企業であるアルテラ社の買収が完了したとアナウンスした [6]。もうひとつの主要な企業であるザイリンクス社は、ARM ベースのプロセッサと FPGA による構成可能な機能をひとつにした SoC (System on Chip) である Zynq を販売している。ARM 社のオーナーであるソフトバンクの CEO は、IoT のために ARM ベースのプロセッサを活用することに積極的な姿勢を表明している [8]。

我々は、計画実現の第一歩として、テキスト処理のもっとも基本的な機能のひとつである文字列の単語へ分解を行う分割器のプロトタイプを開発した。本稿ではそのアーキテクチャと評価結果について述べる。

*Email: kazuhiro.yamato@miraclelinux.com

2 プロトタイプ

2.1 全体構成

プロトタイプは、図 1 に示したようにホストコンピュータと、PCI Express インターフェイスで接続された FPGA ボード (表 1 参照) で構成される。図中のコンポーネントは、3 つのレイヤーに分類できる。下位レイヤー (灰色の背景) は、ハードウェアであり、その機能は固定である。中間レイヤー (青い背景) は、DMA 転送や割り込み処理など様々なアプリケーションで共通して使われ、下位と上位レイヤーの橋渡しを行う。上位レイヤー (赤い背景) は、アプリケーションに固有の処理を持つ。

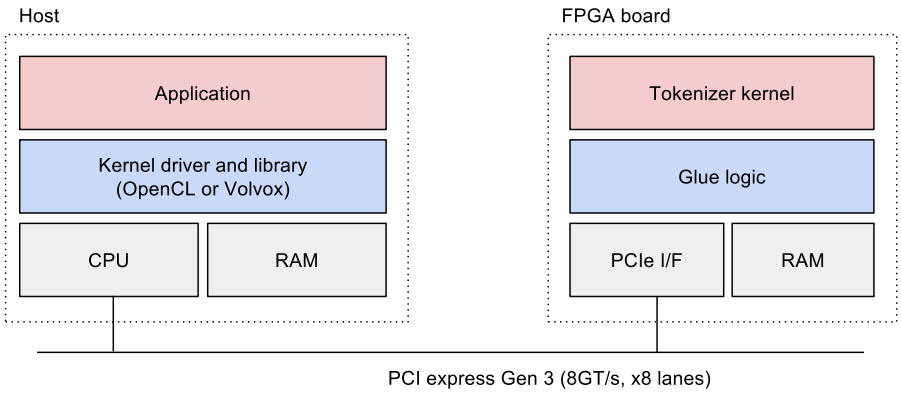

図 1: 全体構成

もっとも主要な役割を担う分割器のカーネル (Tokenizer kernel) [1] を 2.2 節で説明する。中間レイヤーに 2 つの異なるフーレムワーク、OpenCL と Volvox を用いて、それぞの場合について評価を行った。これらを 2.3 と 2.4 節で説明する。上位レイヤーでのホストアプリケーションについては 2.5 節で解説する。

表 1: FPGA ボートと関連する諸元

FPGA ボード	ALPHA DATA ADM-PCIE-7V3 [11]
FPGA デバイス	Xilinx Virtex-7 XC7VX690T-2
最大レーン幅	×8
最大リンク速度	8 GT/s
DDR	2 つの 8GB ECC-SODIMM, 速度は最大 1333MT/s

2.2 分割器のカーネル (Tokenizer kernel)

分割器のカーネルは、各行ごとに、入力される文字列の全ての単語に対して、開始と終端位置の組を算出する。現在のプロトタイプの諸元を表 2 に示す。分割器は、最大 L 行 (表 4 参照) を一度に処理する。分割器は、C++で記述され、ザイリンクス社の Vivado (HLS) 2016.1 によって合成されているので、その入出力は表 3 に示すとおり C 言語の関数のように定義される。

図 2 に、2 つの行 '□□MIRACLE□□□LINUX' と 'Corporate□color□is□green.' (□ はスペースを表す) の分割を例示する。入力パラメータ num_lines と total_length は、それぞれ、1 回のカーネル実行での処理される行数とそれらの合計長を示す。lengths は各行の長さを格納する配列である。その配列長は、num_lines と同一である。配列 lines は、処理される文字列をすべて含み、それらにはパディングや改行を示すコードは含まれない。'\n' が含まれていても、それは単なる文字のひとつとして扱われる。

[1] FPGA 内に構成される処理機構をカーネルと呼ぶ。Linux カーネルとは異なるので混同に注意。

表 2: 分割器のカーネルのプロトタイプ諸元

区切り文字	半角スペース
マルチバイト文字	非対応
連続した区切り文字	ひとつの区切り文字として扱う
1 行の最大長	65535

表 3: 分割器のカーネルの入出力

パラメータ	I/O	ビット幅	種類	プロトコル	要素数/チャンク
num_lines	IN	32	scalar	AXI slave (register)	N/A
total_length	IN	32	scalar	AXI slave (register)	N/A
lines	IN	8	array	AXI master	32
lengths	IN	16	array	AXI master	16
markers	OUT	32	array	AXI master	8
positions	OUT	16	array	AXI master	16

出力パラメータ `markers` は、長さ `num_lines`+1 の配列である。その要素は、対応する行の分割結果を格納する領域の先頭を指す。最後の要素は、`positions` の次の要素を指す。配列 `positions` は、単語の開始と終端の位置の組を含む。その配列長は、入力文字列によって変わり、`markers` の最後の要素を使って次のように算出できる。

$$N_p = Z/B_p \tag{1}$$

ここで N_p, Z と B_p は、それぞれ、`positions` の要素数、`markers` の最後の値、`positions` の要素の大きさである。同様に単語数 N_t は次のように得られる。

$$N_t = N_p/2 \tag{2}$$

カーネルのブロック図を図 3 に示す。入出力のためのポートは、実際にはデータ幅 W_D ビットのひとつの AXI マスターインターフェイスなので、配列中の複数の要素が一度に読み書きされる。その単位をチャンクとここでは呼ぶ。チャンク中の要素数を表 3 の '要素数/チャンク' 列に示す。

`Reader` は、まず `lengths` の全ての要素を、続いて `lines` を読み出す。この方式は、メモリをシーケンシャルにアクセスし、AXI のバーストリードを引き起こす。バーストリード /ライトは、1 回の手続きで連続したアドレスの複数のデータを転送することにより高いスループットを実現する。バースト転送を行うことは、高速なデータ入出力のための最も重要なポイントのひとつである。`Dispatcher` は、読み出された `lines` のチャンクを循環的にひとつの `Splitter` に送り出す。`Splitter` は、クロック毎に 1 文字 (8 ビット) を解析し、その繰り返し間隔は $W_D/8$ クロックである [2]。繰り返し間隔と同じ数 N_s 個の `Splitter` を配置することで、合計 N_s 文字が 1 クロックで処理される。`Splitter` は、いくつかのチャンクに分割されたもののひとつを解析するので、行の中の完全な位置を算出するのは不可能である。そのため、完全な位置を後段のブロックで計算出来るように、いつくかの変数も算出する。`Linearizer` は、循環的に `Splitter` から解析結果を集めて、再度ひとつのデータストリームを生成する。`Unifier` は、以前の結果と組み合わせて完全な positions と markers を計算する。このアーキテクチャの

表 4: 分割器のカーネルの構成パラメータ

パラメータ	値	概要
L	8192	分割器が 1 度に処理できる最大行数
W_D	256	AXI マスターインターフェイスのデータ幅
N_S	32	Splitters の数
Q_p^W	64	positions に対するバーストライト長

[2]チャンク中に 16 を超える単語がある場合、より大きいクロック数を要する

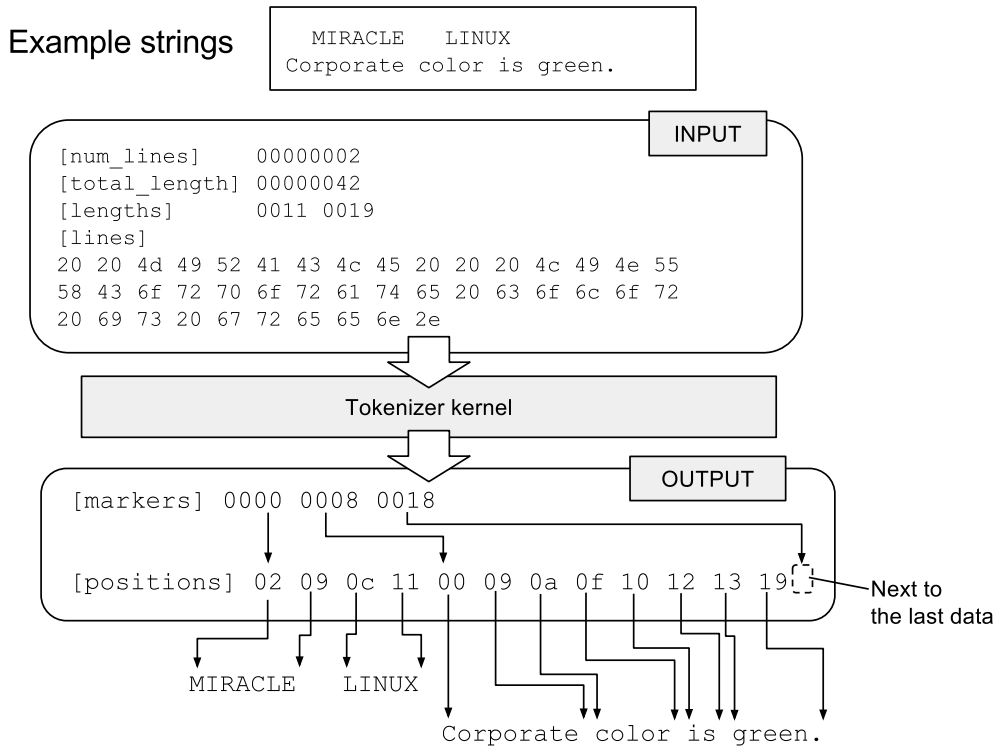

図 2: 単語分割の例。`markers` と `positions` は、見やすくするために実際のビット幅より小さく表示されていることに注意。

利点は、レイテンシ (処理時間) が入力される行の長さの分布に依存せず、主として全体の大きさに比例することである。

算出された positions と markers は、それぞれ、`Positions formatter` と `Marker formatter` に送られる。それらのブロックは、送られてくるデータ要素から W_D ビット長のチャンクを生成する。Q_p^W 個の positions が準備できる度に、`Writer` はそれらを書き出す。一方、markers は、すべての positions が書き出されるまで `Marker formatter` 内に蓄積される。`markers` の要素数は最大でも $L+1$ なので、一時的にこれらを保持することはそれほど困難ではない。この手法もバーストライトを引き起こすために用いられている。

表 5 のように AXI マスターインターフェイスのパラメータを調整している。上述のとおり `positions` と `markers` のバースト長は、それぞれ、Q_p^W と $L+1$ である。なお、ここでのバースト長は、`memcpy()` または、`for` 文の中で使用される値のことである。これは、AXI インターフェイスのパラメータのひとつであり、デフォルト値 (15) のままである `AWLEN` と異なる点に注意のこと。表中の値は試行錯誤的に決められた。これらのパラメータなしでは、バーストライトと次のバーストライトの間隔が、Vivado HLS の C/RTL co-simulation の結果から期待される値より長くなった。

表 5: 明示的に指定した AXI インターフェイスパラメータ

パラメータ	latency	max_read_burst_length
lines	0	32
lengths	0	32
markers	0	default (16)
positions	0	default (16)

分割器のカーネルは 200MHz で動作するように設計されている。これは、FPGA ボード ADM-PCIE-7V3 で OpenCL を使う場合の変更できない要件である。Vivado HLS の co-simulation によると、1 つのチャンク [3] が入

[3] サイズが N_s バイト以下

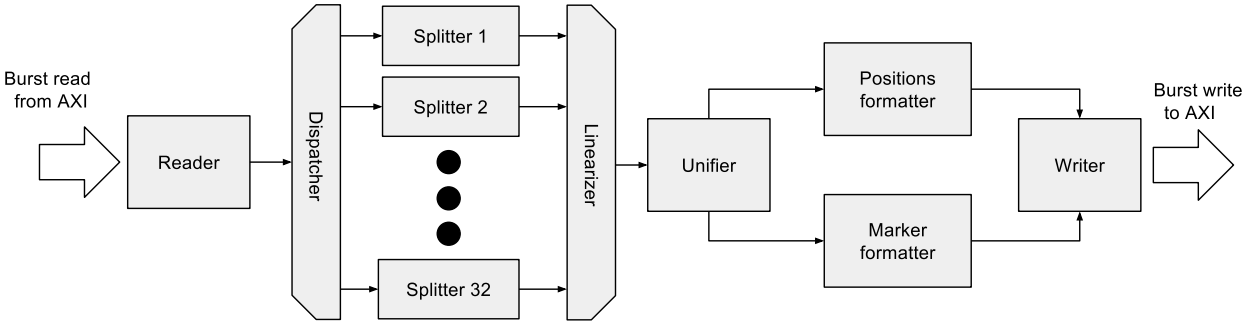

図 3: 分割器のカーネル (Tokenizer kernel) のブロック図

力された場合のレイテンシは 179 であり、N_s バイト ごとに増加する。

2.3 OpenCL フレームワーク

OpenCL は、The Khronos Group [10] によって標準化されているフレームワークである。開発者は、CPU、GPU、FPGA や専用アクセラレータなど種々のデバイスを C 言語をベースにした 1 つの API セットで使用できる。その API は、ホストとデバイスの両方に対して定義されている。加えて、ザイリンク社の OpenCL 開発環境である SDAccel では、OpenCL C のみならず、pragma 指示子を用いた C と C++で FPGA のカーネルを記述することができる。我々は、分割器のカーネルを SDAccel を使って C++で記述した。

図 4 に OpenCL を使う場合のデータの流れの一例を示す。OpenCL は、'Host application' および 'OpenCL framework on Host' と名付けられたボックス間のインターフェイス (API) を定義するので、厳密なシーケンスは、そのライブラリ実装に依存する。図は仕組みを容易にイメージするための補助である。

ラベルの先頭の丸括弧の中の数字は、それらが使用される典型的な順序を示す。ラベルはそれらと関連のあるコンポーネントの近くに配置されている。図ではデータが一旦 FPGA ボード上の RAM を経由しているが、実際のホストと FPGA ボード間の転送方法は、上述のとおり実装依存である。開発者は、目的固有の処理のみを OpenCL C (もしくは C/C++) を使って設計すればよく、PCI Express や DDR RAM のようなハードウェアの詳細を知る必要がない。これは OpenCL を使うもっと有益な点のひとつである。

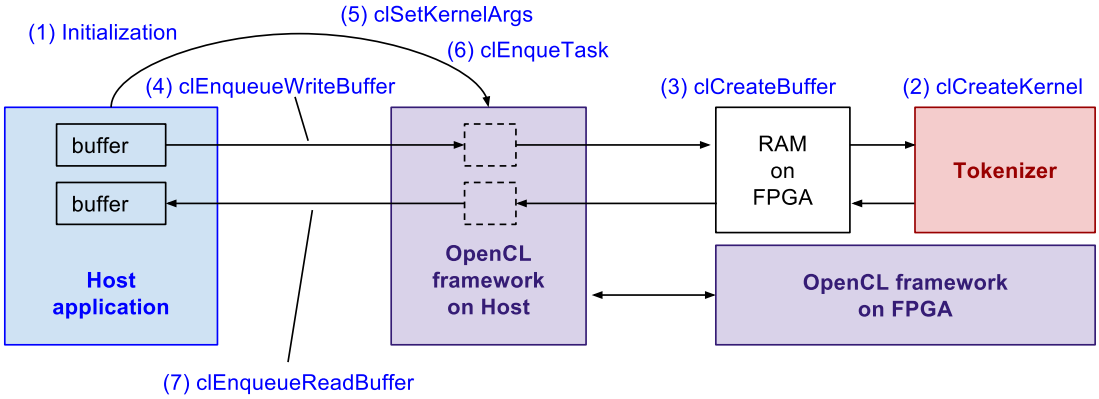

図 4: Volvox フレームワークを使ったデータの流れの例

2.4 Volvox フレームワーク

Volvox フレームワークは、自製のソフトウェアと FPGA のためのグルー・ロジックで構成される。これを開発した理由は次の 2 つである。ひとつは、OpenCL の API セットが様々なプラットホームで種々の目的のために設

計されていること。すなわち、それは本研究の目的に対して最良とは限らないこと。他の理由は、SDAccel がまだ正式版でない (β 版な) ので、それを使った結果にはまだ改善の余地があるかもしれないためである。

分割器のカーネル自身は、何ら変更なしに OpenCL と Volvox フレームワークで共通して使用できる。しかし、ホスト側では、出来る限りの性能を提供するために、上位レイヤーとのインターフェイスは OpenCL のそれとは非互換である。Volvox の特徴のひとつは、分割器のカーネルが、図 5 に示すようにアプリケーション・プログラムのメモリ空間にマップされているバッファに対して直接読み書きすることである。また、FPGA ボード上の DDR RAM は使用されず、Linux カーネルとアプリケーションプログラム間のデータのコピーもない。

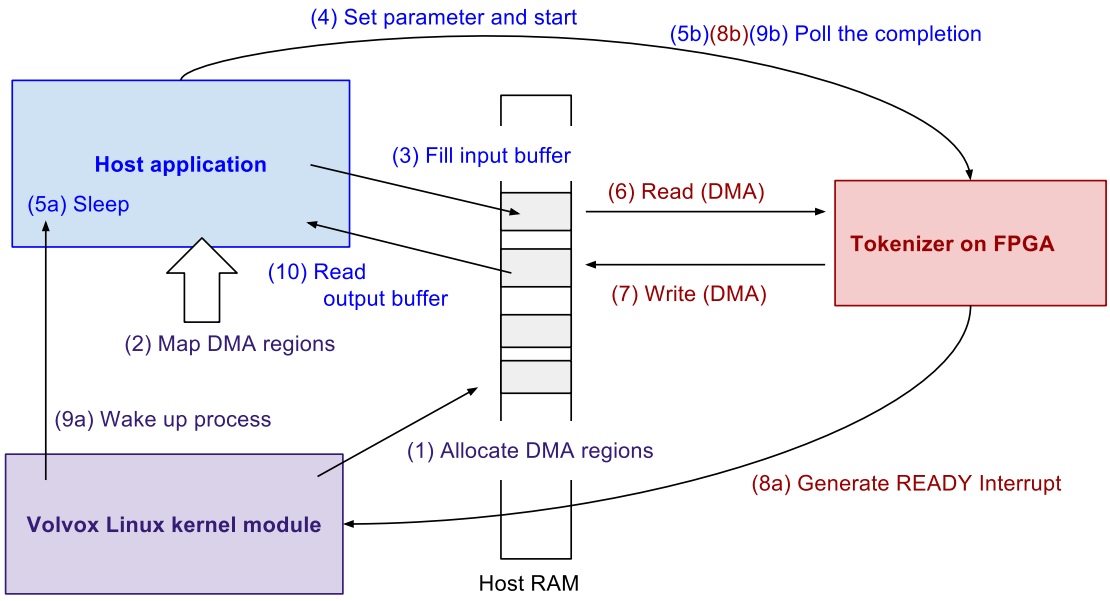

図 5: Volvox フレームワークでのデータの流れ

2.4.1 FPGA 内のグルー・ロジック

図 6 に、分割器が FPGA に搭載されているハードウェアブロックを使って PCI Express インタフェースにアクセスするためのグルー・ロジックを示す。分割器を除くコンポーネントはすべて LogiCORE と呼ばれるザイリンクス社の IP である。それらは、×8 レーン・8GT/s の PCI Express からのトラフィックを処理するために、256 ビットの AXI インターフェイスの RDATA および WDATA を用いて 250MHz で動作する。

AXI Bridge for PCI Express Gen3 は、AXI と PCI Express 間でリードおよびライト操作を転送する。言い換えると、分割器のカーネルの AXI インターフェイスでの 1 つのバーストリード (ライト) 操作は、PCI Express でのバーストリード (ライト) を引き起こす。このアーキテクチャでは、分割器がホストメモリへアクセスする DMA エンジンとして動作する。

割り込みコントローラは、次の 2 つの目的に使用される。

(1) レベル割り込みをエッジ割り込みに変換。分割器の割り込みピンは、Vivado HLS によって自動的に生成され、開発者はその仕様を変更できない。

(2) AXI Bridge のクロックに同期した割り込み信号を生成。

表 6 に配置配線後 FPGA のリソースを示す。使用率の点からは、分割器内の並列度 (`Splitter` の数) を増やすことは可能である。しかしながら、分割器のカーネルの論理的な最大スループットは、すでに PCI Express の最大転送レートにほぼ匹敵している。そのため、これ以上の並列性は性能にほとんど寄与しない。

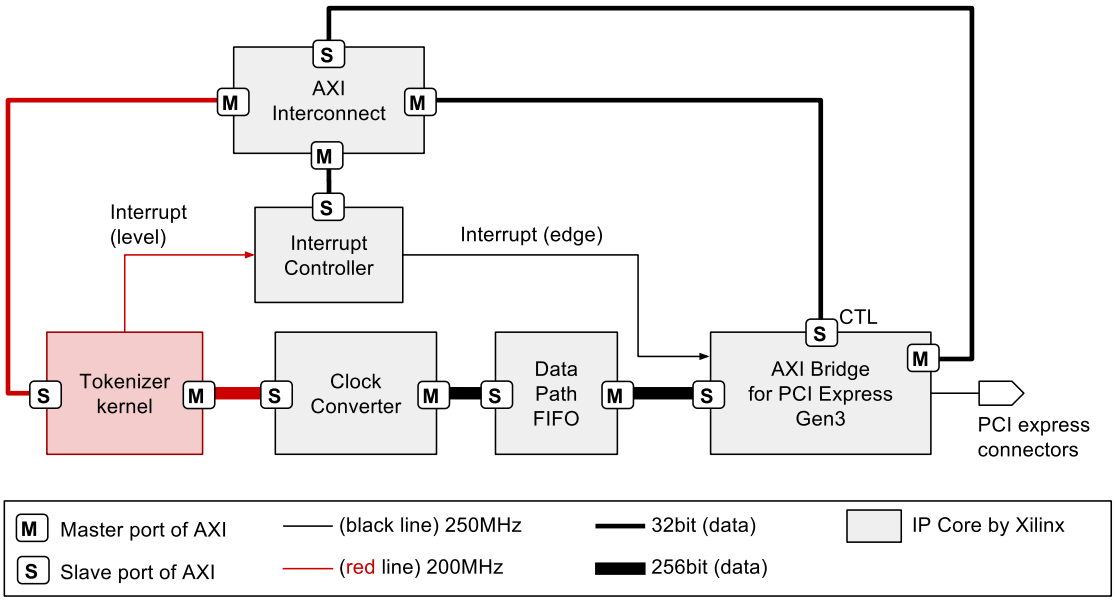

図 6: Volvox のグルー・ロジック

表 6: 分割器とグルー・ロジックのリソース

コンポーネント	LUT	LUTRAM	FF	BRAM
分割器とグルー・ロジック	20%	7%	11%	4%

2.4.2 ホストコンピュータのソフトウェア

Volvox フレームワークのホスト側でのコンポーネントは、Linux 用のローダーブルカーネルモジュールのみである。それは、サイズがそれぞれ 4MiB である DMA バッファ用の 2 つのメモリ領域を 2 組確保する。各領域は、それぞれ FPGA へのデータ、および、FPGA からのデータを格納する。2 組の領域を使うことで、ホストアプリケーションは、1 つの組を使って処理が行われている間に、次のカーネル実行のための入力データを準備したり、すでに処理が完了したデータにアクセス出来る。結果として、この機構も性能の向上に寄与する。Volvox のカーネルモジュールは、`mmap` システムコールを通じて、ホストアプリケーションに領域をマップする。

カーネルモジュールは、分割器からの割り込みを受けとった時に、ホストアプリケーションを起床する機能を持つ。しかしながら、割り込みの使用はオプションである。分割器のカーネルは、処理状態を示すビットを含むレジスタを提供するので、ホストアプリケーションはそのレジスタのポーリングにより処理の完了を知ることができる。カーネルモジュールは、Linux の streaming DMA API [12] に基づく DMA バッファの同期機能を提供する。ホストアプリケーションは、その機能により RAM のキャッシュのフラッシュや無効化を行わなければならない。

2.5 評価のためのホストアプリケーション

性能評価のために表 7 に示す 6 つのホストアプリケーションを開発した。それらは、2.5.1 と 2.5.2 節で解説される 2 つのグループに分類される。

2.5.1 実性能測定ためのホストアプリケーション

ひとつ目のグループの目的は、実性能を測定することである。アプリケーション FO_a と FV_a は、(a) 入力ファイルを読み込み、(b) その内容を分割器のカーネルが利用できる形式に変換して、(c) バッファに書き込み、(d) FPGA 内のカーネルの処理を開始する。ファイルの行数が L を超える場合、上記の (c)(d) が繰り返される。アプ

リケーション C_a は、CPUを用いて同様の処理を行う時間を知るために作成された。FO_a と FV_a と異なり、それは1行ごとに文字列分割を行う。

これらのアプリケーションは、最初のデータの準備終了から分割器のカーネルの実行完了までの時間を測定する。入力ファイルを読み込む時間の遅延を低減するため、すべての文字列は、一旦アプリケーション内のバッファに予めコピーされる。この時間は測定には含まれない。

2.5.2 コンセプト実証のためのアプリケーション

もう一方のグループの目的は、Cライブラリ (glibc) の関数のひとつである `strtok()` に相当する機能を実現し、その性能を調べることである。本稿では、その機能を `strtok_v()` と記す。アプリケーション FO_s と FV_s (C_s) は、単位時間あたり何回 `strtok_v()` (`strtok()`) を呼び出すかを測定する。1節で述べたように我々の目的は、FPGAを透過的に利用するOSを提供することである。これは、そのコンセプトのひとつのデモンストレーションである。ただし、現在の実装では、呼び出し元のアプリケーションの修正を不要とするまでの完全な透過ではない。

分割器のカーネルは、複数の行を一度に処理することが想定されている。1行のみを取り扱うことはできるが、その場合、3.4.1節で議論するようにいくつかのオーバーヘッドにより、不十分な結果になる。それ故、処理される複数行に、それらが `strtok_v()` に渡される前にアクセスできる必要がある。とはいえ、この制約があっても、例えば、大量のログのバッチ処理などに応用することは可能である。

アプリケーション FO_s と FV_s は、まず、複数行についての positions と markers をそれぞれ FO_a と FV_a の方法で得る。その後、それらは、`char *strtok_v(char *str)` の呼び出し毎に、NULLターミネーターをパラメータとして与えられた文字列に埋め込む。引数 `str` は、本来の `strtok` のそれと同じく分割される文字列である。返り値のポインタも、本来のそれと同じ意味を持つ。オリジナルの `strtok` との違いは、区切り文字が固定であり指定できないことである。アプリケーション C_s は、単純に `strtok()` を入力行ごとに呼び出す。C_a の分割エンジンについては我自身々で作成したが、こちらは第三者によって開発され、誰でも容易に使用できる GNU C Library の一部である。

表 7: ホストアプリケーション

アプリケーション	FO_a	FV_a	C_a	FO_s	FV_s	C_s
目的	実性能			strtok() 相当の機能		
デバイス	FPGA	FPGA	CPU	FPGA	FPGA	CPU
フレームワーク	OpenCL	Volvox	N/A	OpenCL	Volvox	N/A

3 評価

3.1 ホストマシン

表8に評価に用いたホストマシンの諸元を示す。

表 8: ホストマシンの諸元

CPU	Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
RAM	8 GB ×4
OS	CentOS 6.8 (x86_64)
Max payload size on PCIe	128

3.2 入力データ

英語文書から構成される 10 から 10^9 の 9 つのサイズの入力データを用いた。表 9 に、それらに含まれる単語と行の数を示す。図 7 と 8 に、単語長と連続した区切り文字の反復数の分布を示す。最頻出の語長は 3 である。90%と 99%の単語は、それぞれ、9 文字と 13 文字より小さい。

入力データサイズは、入力単語の組わせに依存し、入力行数が同じだとしても当然異なる。入力行数が最大 (すなわち L) の場合のおおよそサイズは数百キロバイトである。

表 9: 入力データの単語数と行数

サイズ (Bytes)	10	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9
単語数	2	18	154	1579	17194	171422	1743974	17237031	172432363
行数	1	3	43	247	1831	17390	194936	1988957	19911073

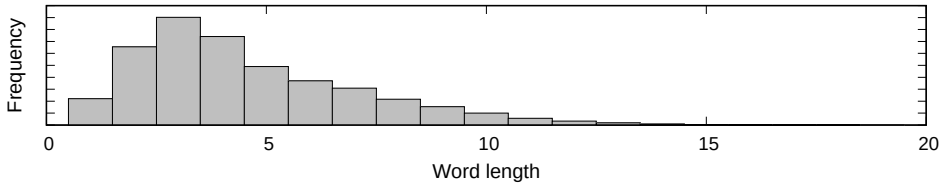

図 7: 入力データの単語長分布

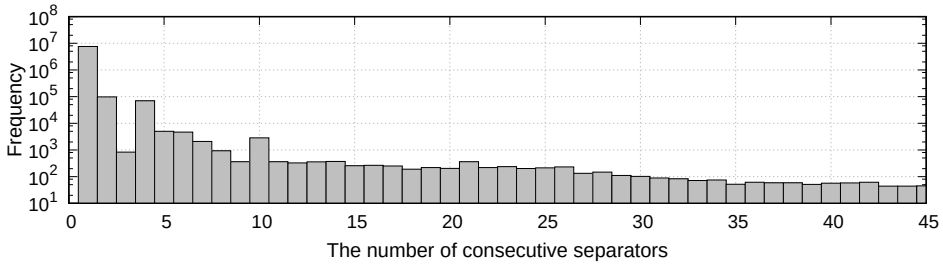

図 8: 入力データ中の区切り文字の反復数分布

3.3 アプリケーション

最大の性能を得るために 2.5 節に記載のアプリケーションをポーリングモードで使用した。割り込みを使った場合、おおよそ 5 から 10μs の追加的な遅延が分割器のカーネルの実行毎に発生した。

処理時間は、アプリケーション内に配置された `clock_gettime(CLOCK_MONOTONIC)` を使って測定された。この関数は測定毎に 2 回呼び出される。報告 [13] によると、この関数自身の遅延は約 50ns であり基本的に無視できる。

3.4 結果と議論

3.4.1 実性能

図 9 に、入力データサイズ A に対する処理時間 T を示す。黒、青、赤のマーカーは、それぞれ、アプリケーション FO_a、C_a および FV_a を用いた測定結果を表す。C_a の処理時間 (T_{C_a}) は、$A \lesssim 10^3$ では最小である。一方、FV_a (T_{FV_a}) は、$A \gtrsim 10^4$ で C_a のそれより小さい。比率 T_{C_a}/T_{FV_a} と、1 秒あたりに処理されるデータサイズであるスループット FV_s は、それぞれ、10 および 4.13×10^9 B/s である。

C_a の処理時間は、おおよそ入力データサイズに比例している [4] が、FV_a のそれは、$A \lesssim 10^3$ でほぼ一定で約 $7\mu s$ である。我々は、$A = 10$ での処理時間の内訳を調べた。FV_a には、以下の 3 つの工程がある。

(1) DMA バッファへの入力データの書き込みと分割器カーネルのレジスタ設定。

(2) DMA バッファの同期。

(3) 分割器のカーネルの実行。

それぞれの時間は、それぞれ、約 1、4、および 2 μs であった。工程 (1) と (3) は、入力サイズとともに増加するが、工程 (2) はほぼ一定であった。

(3) の時間に関して、2.2 節で述べたように設計 (C/RTL co-simulation) 上の分割器の最小レイテンシは 179 であるが、実際のそれは、クロック周波数が 200MHz なので約 400 であることを意味している。その違いは、ホスト RAM からデータを得るための遅延時間に起因する。図 10 に、分割器の AXI インターフェイスのいくつかの信号を示す。それは、FPGA 中の実信号のロジック・アナライザである ILA (Integrated Logic Analyzer) [14] を用いて取得された。最初の ARVALID の立ち上がり (読み込み要求の開始) から、対応する RVALID の立ち上がり (要求データの到着) までの間隔は、約 100 クロックである。分割器のカーネルは `lengths` と `lines` のために、2 度のバーストリードを行うので、その約 100 クロックの 2 倍の遅延が、設計上想定される値に加算される。仮に `lengths` と `lines` の両方を含むひとつの配列を使えば、遅延時間は減るであろう。しかし、それは可読性と保守性の低下をもたらす懸念がある。

FO_a の結果は、$A < 10^7$ において、$10^{-}3 \lesssim T \lesssim 10^0$ の広い 処理時間の分布を持ち、その平均時間は、$A = 10$ において、FV_a と C_a に対して、それぞれ、10^4 および 10^5 倍である。この理由は現在不明で、根本原因を探ることは OpenCL の実装がプロプライエタリであるために困難である。しかしながら、その影響は大きいサイズの入力 ($A > 10^8$) を処理する場合、無視できる。

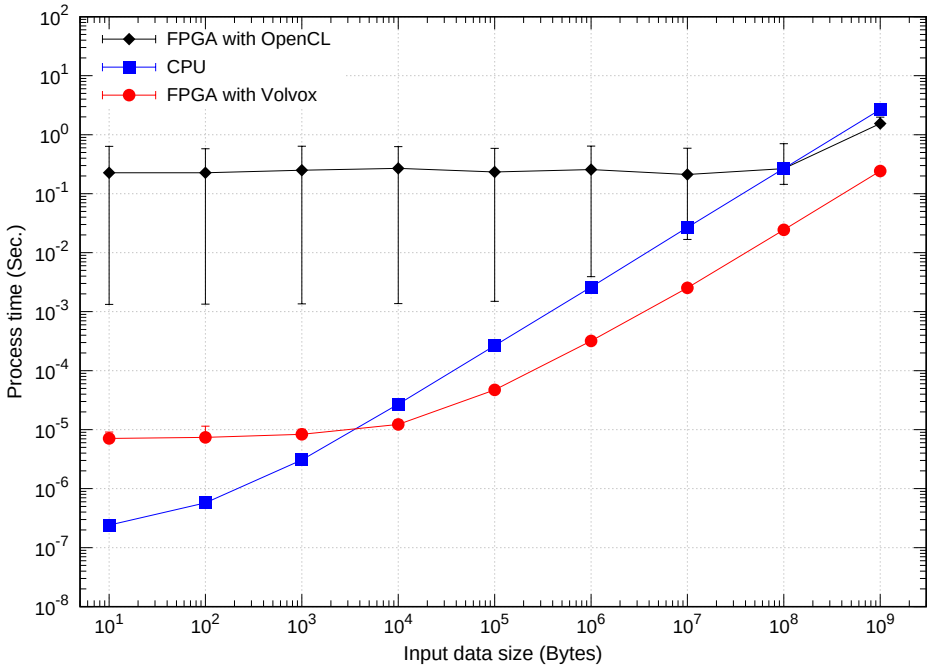

図 9: 入力データサイズに対する処理時間。マーカーは平均時間を示す。エラーバーは、結果の 90%が含まれる区間を表す。

3.4.2 strtok 相当関数の呼び出し回数

図 11 は、1 秒間の呼び出し回数の測定結果を示す。全体的な結果は、実性能のそれとほぼ合致する。C_s に対する FV_s の呼び出し回数の比率は、約 7 であり、前節で議論した実性能の比率である約 10 (T_{C_a}/T_{FV_a}) より小さい。

[4] 入力データサイズが小さいところでの非線形性は、3.3 節で言及した `clock_gettime()` に起因すると考えられる。

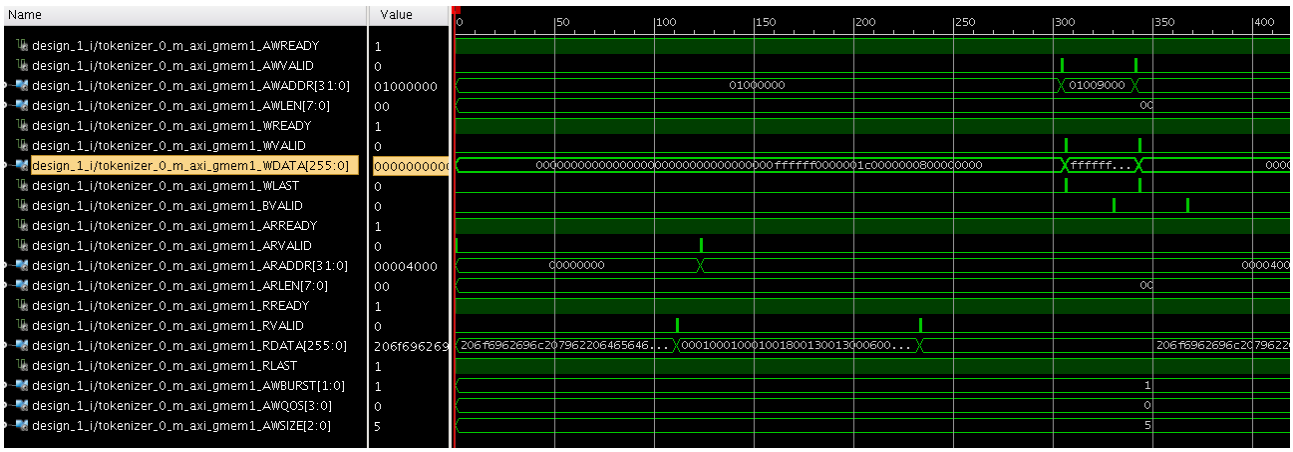

図 10: 10 バイトの文字列を使用した場合の分割器の AXI インターフェイスの信号

これは、`strtok_v()` の呼び出しと NULL ターミネーターの書き出しのために CPU による処理が増えたためと考えられる。

一般的に分割された単語に対して、さらなる処理が行われる。例えば、数値型への変換、大文字化、それらを使った新しい文字列の作成などである。次の複数行の処理を分割器のカーネルが処理している間に、これらを CPU で行うことができるので、CPU 負荷の観点からは分割処理がなくなったように見える。とにかく、この結果は我々のコンセプトが実現可能であることを示している。

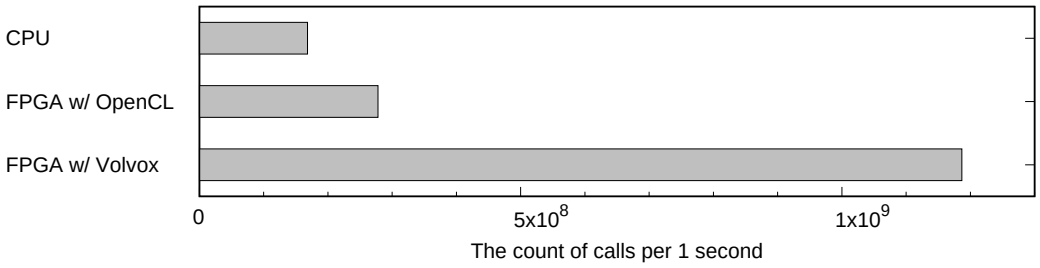

図 11: `strtok()` と相当する関数 `strtok_v()` の呼び出し回数の比較。

4 まとめ

本稿では、急速な増加が予想されている IoT 機器に向けた FPGA を用いた文字列分割の研究結果について報告した。ザイリンクス社の高位合成コンパイラを用いたプロトタイプは、200MHz で動作し、最大 32 の ASCII 文字をクロック毎に処理する。評価結果は、本目的のために作成された自製フレームワーク Volvox と、注意深く調整されたバースト転送のためのパラメータを用いたプロトタイプによる処理が CPU による処理より約 10 倍高速であった。我々のゴールは、1 節で述べたように FPGA を透過的に活用する OS を提供することである。今後、他の文字列処理機能と、FPGA 上でそれらを組み合わせた処理を実現していく。

ザイリンクス社に対して、OpenCL 開発環境である SDAccel の β 版をライセンスして頂いたことに感謝する。
OpenCL を用いた結果には不可解な部分もあるものの、協力してそれらを明らかにしていきたい。

参考文献

[1] Gartner's press release, Gartner says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015.

[2] http://hadoop.apache.org/

[3] http://simplecore.intel.com/newsroom/wp-content/uploads/sites/11/2016/03/Intel-Investor-Conference-Call-Deck.pdf

[4] https://www.xilinx.com/products/technology/power.html

[5] https://www.gnu.org/software/libc/

[6] https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/

[7] https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[8] http://www.armtechcon.com/from-trilobites-to-a-trillion-chips-the-iot-explosion/

[9] http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[10] https://www.khronos.org/

[11] http://www.alpha-data.com/dcp/products.php?product=adm-pcie-7v3

[12] https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt

[13] http://btorpey.github.io/blog/2014/02/18/clock-sources-in-linux/

[14] https://www.xilinx.com/products/intellectual-property/ila.html