Vol.2016-ARC-221 No.39
2016/8/10

人工知能指向アーキテクチャの現代的アプローチ

竹岡 尚三[†1,1,a)]　中條 拓伯[1,b)]

概要：本稿は，人工知能向けアーキテクチャを，現代的な知見の元に見直す試みである．かつての記号処理専用マシンは，単一の素朴なパイプラインでのデータ処理に加えて，タグ・チェックを並列に行う程度であった．現在は，アプリケーション記述層では，マルチスレッド，*Map&Reduce* に代表されるデータセントリック処理，関数型プログラミングが一般的になり，ハードウェアに近い層では，スーパスカラ，投機的実行などの技術が一般的である．また，細粒度では，記号処理分野でも，データ・レベル並列を利用した並列処理が盛んに研究されたこともある．本稿では，記号処理データの配置に関するアイデア，記号処理におけるデータ・レベル並列の再考，キャッシュ・ミスヒット時の処理などについて述べ，現代的な要素を人工知能指向アーキテクチャに持ち込むことについて，考察する．

キーワード：人工知能，記号処理，並列処理，プロセッサアーキテクチャ

Contemporary Approach for AI Oriented Computer Architecture

TAKEOKA Shozo[†1,1,a)]　NAKAJO Hironori[1,b)]

Abstract: In this paper, we try to review an AI processor based on contemporary knowledge and technology. In the past few decades, dedicated symbolic processing machines were just checking tags in parallel in addition to data processing with a single and simple pipeline. Currently data centric processing or functional programming such as multi-threading or *Map&Reduce* have been generally utilized in a layer of application description as well as super-scalar or speculative execution have been general in a layer around hardware. Furthermore, fine grained parallel processing based on data level parallelism has been investigated in symbolic processing field. This paper describes some ideas on arrangement of data and reconsideration of data level parallelism in symbolic processing as well as handling processing in cache misses in order to bring contemporary elements into an AI oriented computer architecture.

Keywords: Artifitial intelligence, symbolic processing, parallel processing, processor architecture

1. はじめに

本稿は，記号処理をベースとした旧来の人工知能志向システムの並列性の分析を行い，その後，実用的な人工知能を目指し，現状の技術動向に基づいた今後のアプローチについて述べる．

人工知能技術は，大きく機械学習システムと，推論システムなどの記号処理システムに分けられる．本章ではまず，従来の記号処理用アーキテクチャと従来の大規模並列プロセッサ・システムにおける記号処理における並列性に関してまとめ，続いて，機械学習系における並列性について述べる．

1.1 旧来の記号処理用アーキテクチャにおける低水準並列性

これまでの記号処理用アーキテクチャ [1] では，低水準，すなわち細粒度並列性はあまり高くなかった．

旧来の典型的な記号処理アーキテクチャの細粒度並列処理では，以下のような処理が同時に並列に行われていた．

[1] 東京農工大学
Tokyo University of Agriculture and Technology
[†1] 現在，(株) アックス
Presently with AXE, Inc.
[a)] take@axe.bz
[b)] nakajo@cc.tuat.ac.jp

- 演算処理
- データ・フェッチ
- タグ・チェック
- ガベージコレクション

演算処理は非常に単純であり，パイプライン深度は浅く，パイプライン深度方向の並列度は低かった．また，投機実行はほぼ行われていないか，単純なプリフェッチ程度の極めて素朴なものだけが実現されていた．

記号処理の基本データであるリストは，実直な実現を行うと，ポインタによる実現であるため，一度フェッチして，それをインデックスとしなければ，次のデータをアクセスできなかった．そして，データ・フェッチに関する投機実行は，プリフェッチ程度であったので，その次以降のデータ・フェッチのほとんどは，フェッチし演算処理を行なった結果を使用していた．そのため，データ・フェッチのみを次々と行うことはできなかった．

旧来の記号処理アーキテクチャでは，データの供給に対して，演算が非常に単純であったために，CPU コア内部での細粒度並列性を上げることが難しかった．

並列性に関してはやや遠のくが，以下では，メモリ管理などについて述べる．ガベージコレクションは，1980 年前後において，コピー方式は，オーバヘッドが大きいとされ，あまり使用されなかった．しかしながら，1990 年前後から現在において，キャッシュメモリやデマンド・ページング方式を備えたシステムが広く一般的になったことから，キャッシュのヒット率向上や，データのメモリ・ページ内への集積を行ったほうが性能が上がり易いため，なんらかのコピーを行う方式が採用されることが通常である．

また，リスト表現については，CDR-Coding といった，リストを連続領域に配置する実装が行われたマシンがある?. CDR-Coding は, CONS ペア表現を用いるか, CDR-Coding 表現を行うかの判断のオーバヘッドや CDR-Coding を解消し CONS ペア表現に変換するときの処理があり，結局，総合的にはオーバヘッドが大きいものとされ，現在はほぼ使用されていない．

2. 旧来の記号処理用アーキテクチャにおける高水準の並列性

プログラマが意識する高水準な並列性としては，マルチスレッドによる粗粒度の並列性がある．また，Lisp などの言語では，並列な *map* 計算が提案され，実現も数多くある．これらは，各データの依存性が低くなるようにプログラマが記述すれば，高い並列度を得られる．例えば，木探索を行うようなシステムでの，並列な探索は高い並列度を得られることが多い．ただし，極めて旧式のメモリ容量の乏しいシステムでは，メモリ・アロケータに要求が集中し，そこがボトルネックとなることが多かった．

近代的なシステムにおいては，メモリは各演算コアごとに独立に十分に用意できるので，記号処理特有の問題は生じない．当然，データに依存性があるようなプログラムでは並列度が上がらないが，現在の多くのプログラマはマルチスレッド・プログラミングに習熟しているため，並列プログラミングでは，データに依存性があるようなプログラムを記述しないようにといった知見が浸透している．

Haskell，OCaml，Erlang などの単一代入な関数型言語が一般的に知られ，実用システムで広く使われるようになってきており，同時に，*Hadoop* などによって，*Map&Reduce* 計算も一般に知られるようになり，並列 *map* が抵抗なく一般的なプログラマに使われるようになってきている．また，メモリが潤沢になり，並列度を上げたいため，最近の記号処理アプリケーションでは，特に必要性がない限り，リストの破壊的操作は行われない．

2.1 従来の大規模並列プロセッサ・システムにおける記号処理と並列性

1990 年前後の大規模並列プロセッサ (MPP)・システムは，SIMD システムが多く，それらの MPP 上での記号処理は，データ並列性を発揮するように実現されていた．例えば，Connection Machine 上には Xector というデータ表現と，それを操作する Xapping を備えた Lisp[3] が実現された．

現在主流の MIMD ベースのクラスタ計算機システムは MPMD(Multiple Programs Multiple Data stream) ではあるが，そこで実行される一般的なアプリケーションは，MPI 通信や Hadoop を用いて，粗粒度でみると SIMD 的な考えで設計されている．

また，Hadoop などの *Map&Reduce* 系のシステムは，基本的にデータ並列性を活用し，データの移動を行わないようなデータセントリックなシステムとして，並列計算プログラミングが容易になるような方針づけを行っている．データ並列処理は，データ間の独立性が高いアプリケーションを対象にしており，細粒度でも高い並列度を持つ．MIMD ベースのクラスタ計算機であれば，各ノードで独立にリスト処理を行えるので，リスト処理を使用した探索処理も，高並列度で処理できる．

2.2 機械学習系人工知能の並列性

機械学習はデータの配置に規則性を持たせることが容易で，単純な配列に対する演算が行えることが多い．したがって，ベクトル計算機や GPGPU のような，深度の深いパイプライン・アーキテクチャでも高効率で実行できる．このことから，細粒度並列性が高いといえる．

また，例えば Support Vector Machine のように学習対象データを分割して，独立して学習させ，最後に学習結果をマージすることが容易な方式もある．したがって，データ並列アーキテクチャや，各ノードが独立した MIMD 計

算機でも高い並列度で学習を行わせることができる．これについは，粗粒度並列性が高いといえる．

2.3 遺伝的アルゴリズムの人工知能の並列性

遺伝的アルゴリズム (Genetic Algorithm，以下 GA と記述)) は，基本的に試行を多量に繰り返す．その試行は，基本的に独立している．したがって，GA の評価機構の構造に関わらず，独立試行を高い並列度で実行することができる．粗粒度並列性は高いといえる．

2.4 熱力学的計算

量子アニーリングを使用した D-Wave[4] のシステムが Google などにより採用されたことで，焼きなまし法などのような熱力学的に最適に近い解を得る計算が，最近また注目されている．焼きなまし法は，計算の分割が容易で，分割されたものは独立性が高いので，容易に並列化できる．これらは，非常に局所的な演算を行うため，並列度は演算装置の台数にほぼ正比例する．

2.5 現代的な応用の期待

人工知能の応用に関して，機械学習系については，第 1 次 AI ブームと似たような応用分野に注目が集まっており，ほぼあらゆる分野で機械学習系の人工知能が使用できることが期待されている．

記号処理おいては，自然言語処理について，より精度の高い解析が求められている．人工知能は人間同士の自然言語のやりとりから，情報を抽出することなどが求められている．具体的な例として，金融分野では，各所からのニュースリリース文を正確に読み取り，投資の方針決定の補助となることが求められている．

このような場合，記号処理による推論システムを使用し，その中心機能は，ユニフィケーションである．また，いわゆるビッグデータ時代になり，記号処理の対象となるデータの量が増え，推論の探索空間も大きくなりつつある．同時に，独立性の高いデータが増大し，また，アプリケーション・プログラマもスレッド・プログラミングや，Map などのデータセントリックなプログラミングを行うようになったため，高水準での並列度が高まっている．

また，自動車のような移動体において，運転者と人工知能が自然言語による対話を行うことは，すでに要請されている．しかしながら，自動車などは，電波の届かないところを走行することも多い．したがって，車載の人工知能は，サーバやクラウドなどと通信すること無く，単独で自然言語処理をすることが要求されている．つまり，性能/電力比の良い記号処理ハードウェアが望まれている．

3. AI プロセッサの動向

1997 年に IBM 社のディープ・ブルーがチェス王者に勝利して以来，人工知能に対する期待が再燃し，2016 年 3 月には囲碁界のトップ棋士を打ち破るに至った．そこには，人工知能ソフトウェアが重要な要素ではあるものの，その計算量は膨大なものとなり，ハードウェア性能，とりわけプロセッサ処理能力が重要な鍵を握る．本章では，現在注目されている人工知能プロセッサに焦点をあて，そのコンセプトや応用について紹介する．

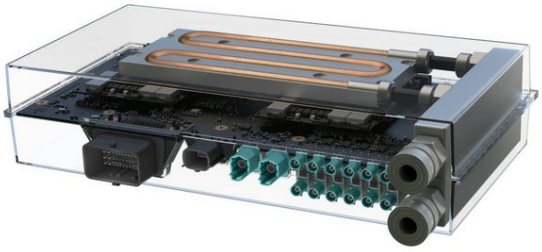

図 1 NVIDIA DRIVE PX2 (2016 NVIDIA Corporation All rights reserved)

Fig. 1 NVIDIA DRIVE PX2 (2016 NVIDIA Corporation All rights reserved)

3.1 Nvidia 社 DRIVE PX

GPU において最先端をリードする NVIDIA 社は，自動車の自律走行を目指して DRIVE PX を開発した [5].

安全な自動運転には，まず自車の位置の精確な判断，周囲の障害物の認識，安全運転のための最適ルートの探索のための膨大な計算を持続する必要がある．

自車とその周辺環境の認識には，カメラと他のセンサからの情報とともに，ナビゲーション機器からのデータと照合し，一方で最も安全なルートの算出を行うこととなり，これらすべてをリアルタイムで処理するための計算パワーが要求される．そのために，DRIVE PX はディープラーニング，センサフュージョン，360 度の視野を提供する画像取得機能といった技術を統合している．

これらの処理のために Tegra X1 を 2 個搭載し，グラフィックス，コンピュータービジョン，ディープラーニング用の豊富なミドルウェアを提供している（**図 1**）．

しかしながら，NVIDIA 社は専用の AI プロセッサを開発するといった道を取らず，あくまでも GPU を活用していくという方針で進めている．

3.2 Google 社 TPU (Tensor Processing Unit)

2016 年 5 月に Google 社は，ディープラーニング専用プロセッサとして Tensor Processing Unit (TPU) を発表した．その TPU モジュールを**図 2** に示す．

TPU はディープラーニングの学習演算のために開発した ASIC であり，これまで他社がディープラーニングに GPU や FPGA によるアクセラレーションを行っていたものとは一線を画する．その詳細はまだ明らかではないが，

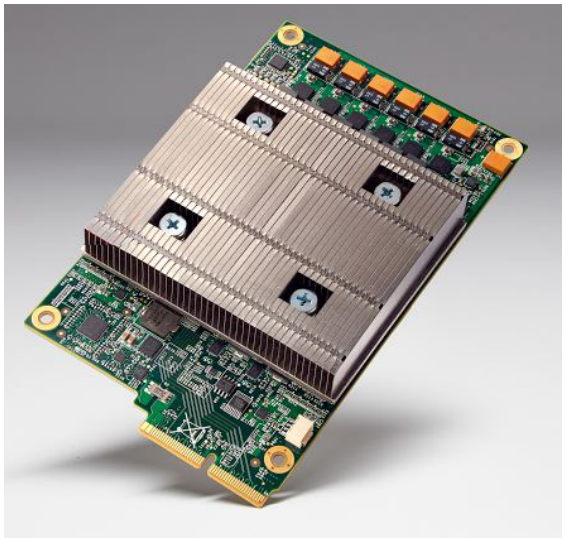

図 2　TPU モジュール (　2016 Google All rights reserved)
Fig. 2　TPU Module (　2016 Google All rights reserved)

現時点での発表によると消費電力当たりの性能は GPU や FPGA の 10 倍と言われている．

大きな特徴としては，機械学習アルゴリズムには，それほど精度が必要とされないという点に着目し，演算ビットを 8 ビットにした点が消費電力に大きく影響しているものと思われる．

なお，TPU は前述の囲碁のトップ棋士を打ち破った AlphaGo にも利用されており，その開発には初めて GHz を超えた RISC プロセッサであった Alpha チップの開発者であった Norman Jouppi が加わっている．TPU に関しては，今後詳細な内容が公開されていくものと期待される．

4.　本アプローチの方針

ここまでの分析から，これまでの記号処理用アーキテクチャでは演算部へのデータの供給が不十分あることが言える．リスト・データを十分に供給するには間接アドレシングによるリスト表現では限界があると考えられる．

例えば，ベクトル計算機に備えられたような間接アドレス参照機構と，条件付きベクトル機構を組み合わせれば，ある程度のデータ供給の効率は高まると思われる．しかしながら，演算部分のパイプラインが単純で深度も浅いため，主にメモリ操作部だけが複雑ないわゆる CISC 的な構造となってしまい，その複雑さに比べて，性能向上を望めないのではないかと思われる．また，演算が軽いため，データ・フェッチのみではなく，リスト・データ (CONS ペア) をストアする機構も高能率であることが望ましく，メモリ・アロケータも必要であることから，ナイーブなリスト・データのロード/ストアの両面を考えると，メモリ操作部はかなりの複雑さを持ってしまうと考えられる．

そこで，本アプローチでは，ナイーブなリスト表現ではなく，記号処理においても CDR-Coding されたリストや，Xector のように整列し連続しやすいデータを扱うことを考える．

4.1　本アプローチのデータ表現

本アプローチでは，CDR-Coding に変更を加えたものを使用する (**図 3**)．CDR-Coding では，そのリスト長，リスト領域に特に制限は無かった．

本アプローチでは，あらかじめそのリストの連続領域に十分な大きさの最大長を決め，先頭から最大長以内の領域については，そこにアクセスしても特段の問題は出ないようにする．連続領域の大きさについては，アプリケーションにヒントを記述することとする．未使用のデータ・セルに対しては，未使用のタグを付加して，参照されたデータが無効であることが検出できるようにする．

4.2　本アプローチのデータ表現を用いた実行

本アプローチのようにリストデータが連続していれば，単純な連続領域に対するプリフェッチが可能になる．また，2 つのリストのユニフィケーション時や，ワイルドカードを含む一致によるデータ比較では，パイプライン化した比較などを行える．

演算コア部分とメモリ部のバンド幅が大きければ，複数ワードの並列比較なども行える．また，領域長が判っているので，領域を分割して，それらの各領域について独立して並列に比較などを行うこともできる．

5.　AI プロセッサの実現に向けて

従来の記号処理用アーキテクチャでは細粒度並列性が小さすぎて，ハードウェア資源に余裕があり，演算器や処理ユニットを増やせる可能性があっても，それらを生かすことができなかった．

上記で提案したデータ表現を採用すれば，順方向に連続したリストの処理をパイプライン化でき，また，比較器を複数並べて，同時に並列に複数ワードの比較が行える．また，タグ・チェック機構も複数並べて，同時にそれらの複数ワードのタグ・チェックを同時に行うことが可能になる．

無効データをフェッチしていることは，そのデータのタグ・チェックが，計算結果の write back までに終了すればよく，それは容易である．以上をもとに現在構想しているシステムのブロック図を**図 4** に示す．

Xector のようなデータも，上記で提案したデータ表現を採用して実現すれば，MPP ではない，小規模なハードウェアでも高能率で処理が可能である．また，Xector データ長が十分に大きければ，本提案方式の領域を大きく分割して，それを複数の演算コアで分担して並列処理を行うことができる (**図 5**)．

先述したように，現在は，リストの破壊的な操作は特別な場合を除いて，ほとんど行われない．関数型言語におい

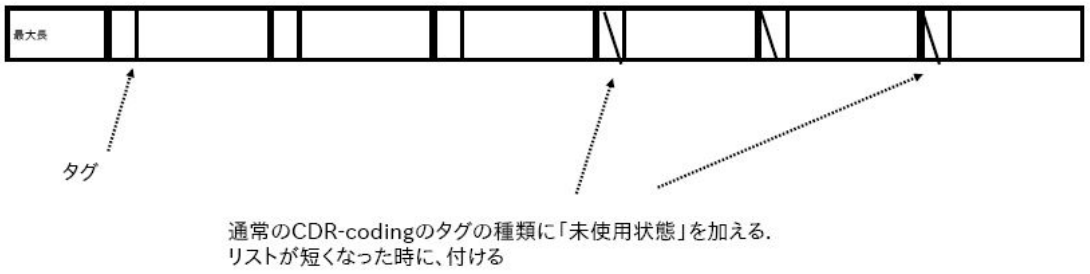

図 3　本アプローチの改変 CDR–Coding
Fig. 3　Revised CDR–Coding in our proposed approach

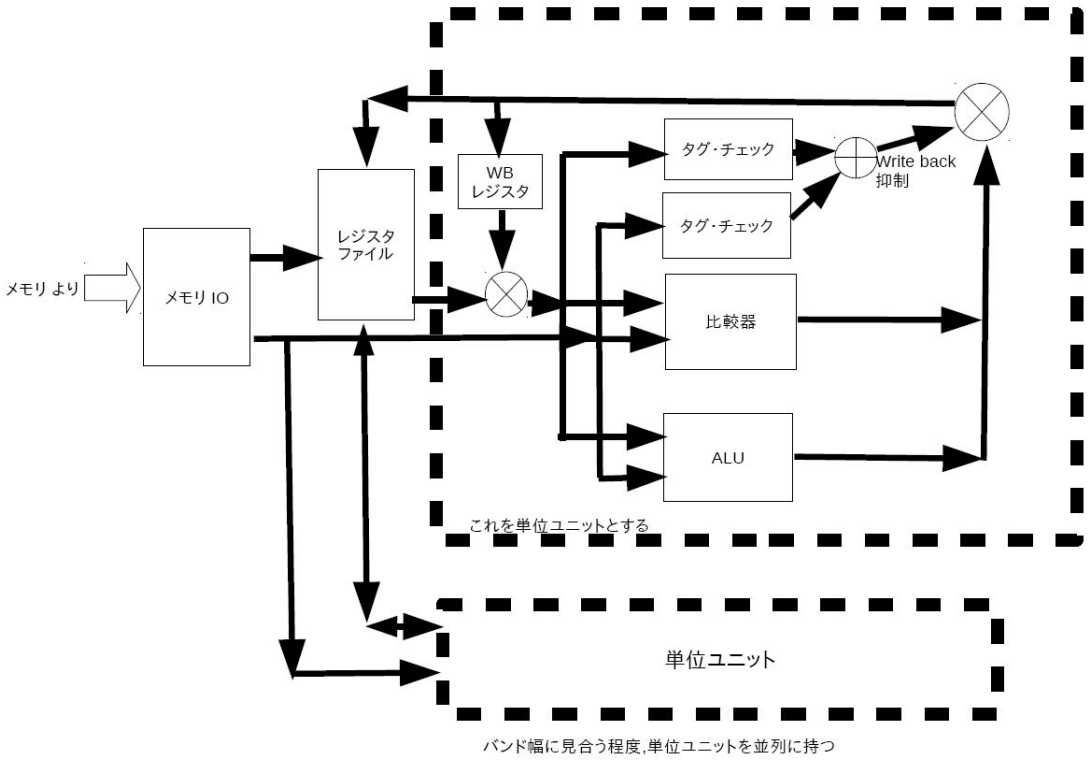

図 4　構想システムの内部ブロック図
Fig. 4　Internal block of the proposed system

ては，単一代入であるから，破壊的操作は皆無と言ってもよい．旧来は，CDR-Coding は，破壊的操作をした時のオーバヘッドを問題にしていたこともあるが，現在では，それはほぼ問題とならない．

本提案は，機械学習系を得意とする GPGPU など，深度の深いパイプラインと相反するものではない．記号処理用パイプラインも，GPGPU などと同様なメモリ IO 機構を使用できるならばその方がよい．つまり，機械学習系に向いた深度の深いパイプラインのすぐ横に，記号処理用のパイプラインを設け，処理内容によって，パイプラインを使い分けるようなハイブリッド構成も考えられる．

単体ノードの性能が向上すれば，粗粒度並列性を利用し，ノード数を増やせば，性能は向上する．実際の機械としては，タグ付きアーキテクチャで，複数のノードで構成されることを想定している． 1 ノードのメモリ空間は 4G 語程度として，32〜36bit 程度の基本データにタグを 4〜6bit 程度含む．

6. まとめ

本提案は，いわば，リストのベクトル化であり，それにより細粒度並列性が高められると考える．実直にナイーブなリスト処理を行っていては，ハードウェア・コスト対性能や電力消費対性能がよい記号処理システムの実現は難しいと思われる．

今後は，まず，ソフトウェア・シミュレータで本方式を実現して，我々が実際に実用している推論システムの中核部を動作させて，実際にどの程度の細粒度並列が得られるかを検証したい．

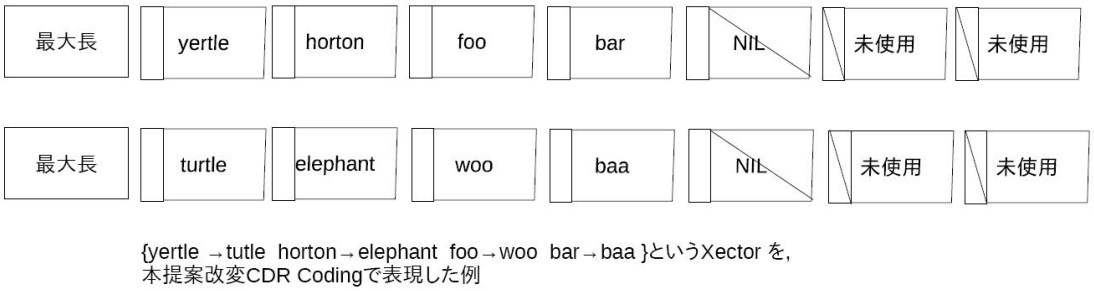

図 5 提案 CDR-Coding 例
Fig. 5 Example of the proposed CDR-Coding

参考文献

[1] Gerald Jay Sussman, Jack Holloway, G. L. Steel Jr., Alan Bell: Scheme-79 Lisp on a Chip, 1981/JUL，IEEE Computer (1981).
[2] Alan Bawden: Lisp Machine Progress Report AIM-444, The Lisp Mahcine Group, 1977/AUG，MIT AI Lab AI Memo (1977).
[3] Skef Wholey, Guy Steele Jr.: Connection Machine Lisp，Thinking Machines Technical Report PL87-6 (1987).
[4] Kamran Karimi, Neil G. Dickson, Firas Hamze, M.H.S. Amin, Marshall Drew-Brook, Fabian A. Chudak, Paul I. Bunyk, William G. Macready, and Geordie Rose: Investigating the Performance of an Adiabatic Quantum Optimization Processor, http://arxiv.org/abs/1006.4147 (2011) (accessed 2016.7.6).
[5] DRIVE PX: http://www.nvidia.co.jp/object/drive-px-jp.html (accessed 2016.7.6).