

# Direct Universal Access: Making Data Center Resources Available to FPGA

---

Ran Shu<sup>1</sup>, Peng Cheng<sup>1</sup>, Guo Chen,<sup>2</sup><sup>1</sup>, Zhiyuan Guo,<sup>3</sup><sup>1</sup>, Lei Qu<sup>1</sup>, Yongqiang Xiong<sup>1</sup>, Derek Chiou<sup>4</sup>, Thomas Moscibroda<sup>4</sup>

<sup>1</sup>Microsoft Research, <sup>2</sup>Hunan University, <sup>3</sup>Beihang University, <sup>4</sup>Microsoft Azure

- FPGA の大規模利用がちょくちょく話題に上がる Microsoft さんの論文。
- 実際に使われているかはともかく、何を問題視していて、どのように解決したかという過程が知りたい。

# Contents

概要

背景と目的

問題の解析と理想のアーキテクチャ

目的

DUA overview / DUA components

Evaluation

結論

## 概要

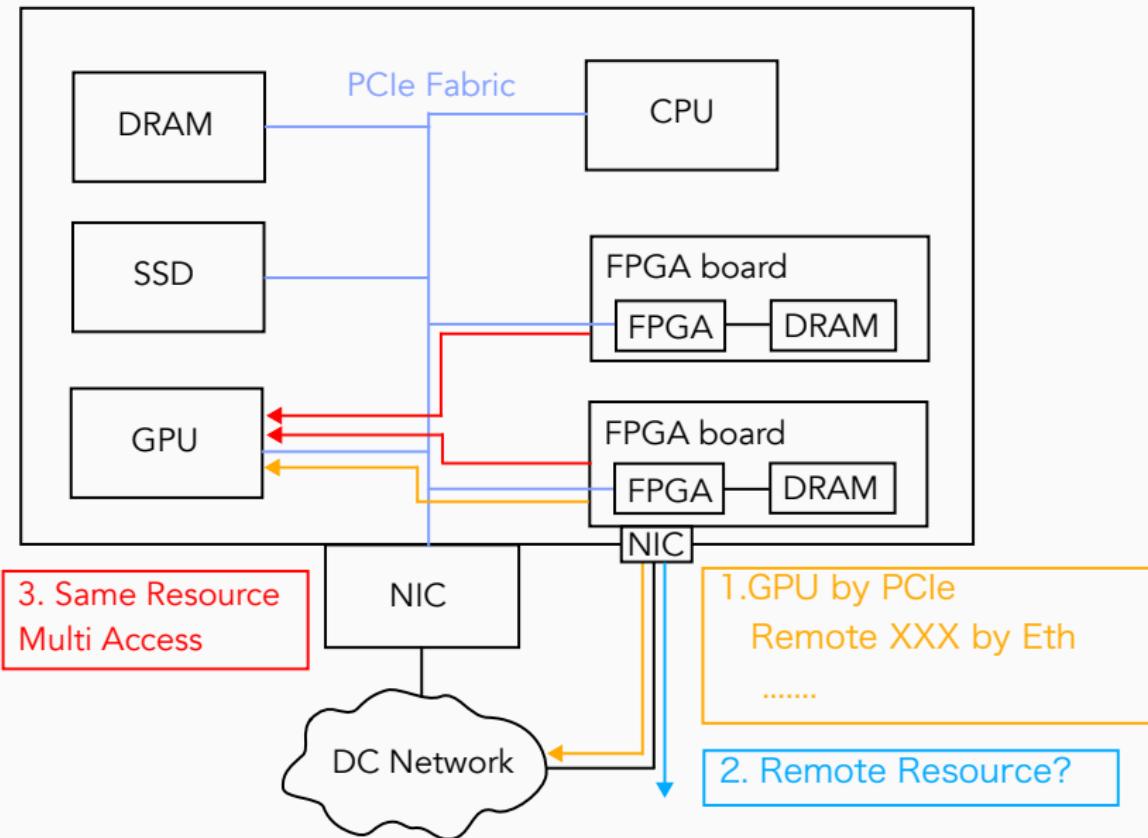
---

- FPGA を大規模データセンターにデプロイすることを考える。
- このとき、FPGA からデータセンターのリソースをうまく利用するのはちょっとむずかしい。
  - オンボードのホストのリソースに関してはまあ良い。
  - 問題はデータセンター内のリモートホストのリソースを利用することを考えたとき。
- FPGA からデータセンター内リソースへのアクセスを抽象化して同一方法で様々なリソースを取り扱えるようにした！ (Direct Universal Access (DUA))
- 評価したところ実装に必要な回路サイズも小さく、性能も良かった。

## 背景と目的

---

- 大規模 FPGA アプリケーションを構築するには複数の FPGA でコミュニケーションとったり、それ以外のリソースともやり取りできてほしいが、概ね以下のようないい問題がある。
  - 1 別ホストに対して、様々なリソースを、様々な通信方式で要求することはプログラムを非常に複雑にする。
  - 2 server-centric で、FPGA から直接的なリモートリソースの特定ができない。
  - 3 リソースマルチプレクシングができない。  
これは開発者が処理を記述する必要がある。  
同じリソースを使用する複数 FPGA が存在する場合、同時アクセスなど問題はより深刻化する。



## 問題の解析と理想のアーキテクチャ

---

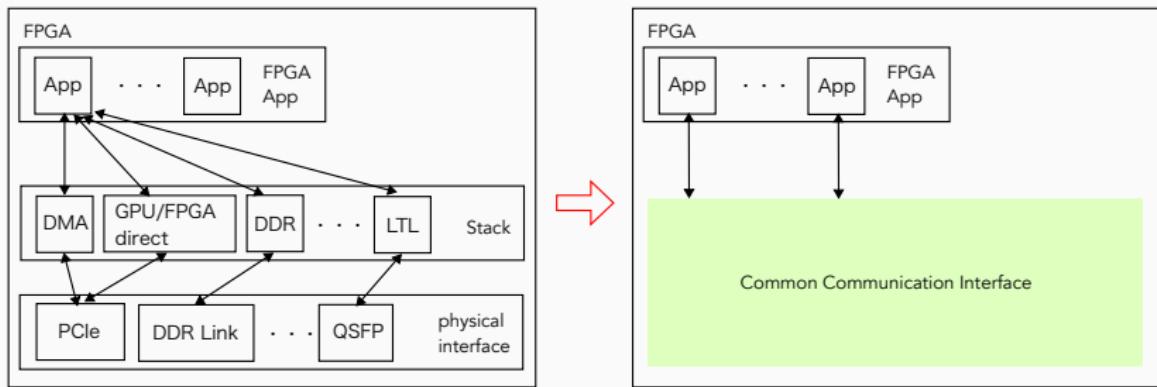
# 問題 1

- 異なるリソース／異なるリソースロケーション／異なる通信方式
  - Resource: DRAM, CPU, GPU, FPGA, ...
  - Location: ローカルホスト、リモートホスト
  - Stack: PCIe, network, ...
- FPGA アプリケーションが何を利用するかによるが非常にプログラムが複雑になる。
- 表は各リソース、Stack の実装に何行程度必要かを示す図。

Resource	Communication stack	LoC
Host DRAM	DMA	294
Host CPU	FPGA host stack	205
Onboard DRAM	DDR	517
Remote FPGA	LTL	1356

# 問題 1 へのアプローチ

- Application からは共通のインターフェースを利用するように抽象化

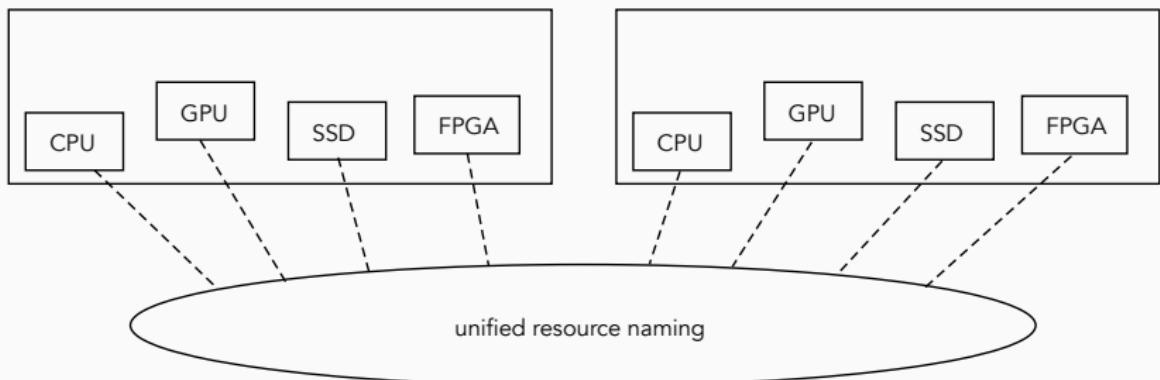


## 問題 2

- Server-Centric (Server 中心の namespace によるアクセシビリティの悪さ)
  - 例えば PCIe アドレスはそのホスト内部からの利用のみが可能な namespace
- 例えば FPGA からリモートの SSD にデータを書き込みたい場合
  - 1 local CPU で動作している daemon process へデータ転送
  - 2 remote CPU で動作する daemon process へデータ転送
  - 3 リモートの SSD に書き込みをする
- このような記述をリソースコミュニケーション種類ごとに記述する。

## 問題 2 へのアプローチ

- global unified naming scheme
- DC ネットワークリソースを一意に特定するような命名をしてリソース特定可能にする。
- またこの名前からリソースに対してアクセスできるような Routing を実装する

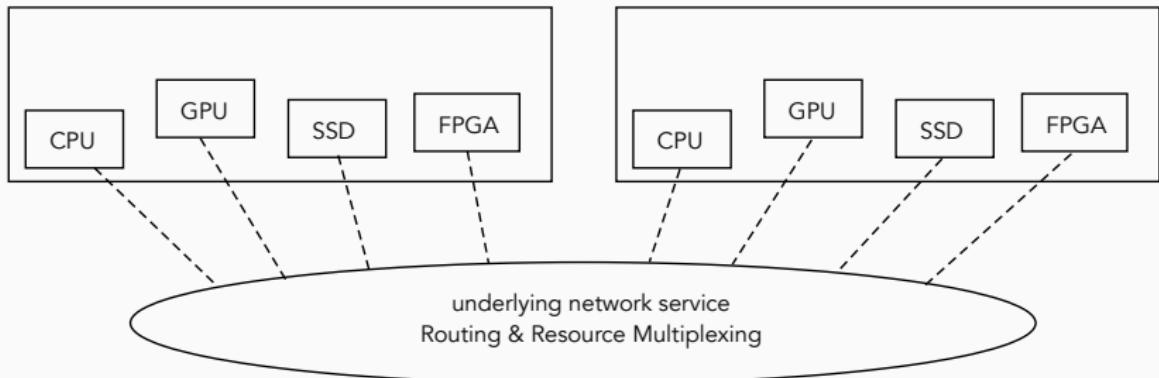


## 問題 3

- 通信におけるリソースマルチプレクシングが貧弱
  - 例えば PCIe を利用するが、アクセス先は DMA/GPU/FPGA と異なる場合
  - FPGA 開発者はこれを適切にプログラムでハンドルする。
  - Application と密結合になってしまふ
- 更に、同じリソースへの同時アクセスケースのハンドリングも。。。
  - 2つの FPGA 上の Application から同じ SSD に書き込む場合

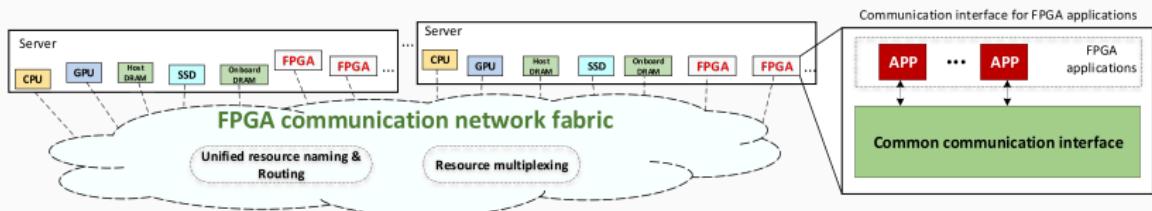
## 問題 3 へのアプローチ

- underlying network service の実装
  - unified naming scheme を利用したリソースの routing
  - multiplexing 機能も実装



# Desicred Communication Architecture

- まとめると FPGA の世界に以下のようなアーキテクチャを導入したい
  - common communication interface
  - global, unified naming scheme
  - underlying network service
- これを実装したものが **Direct Universal Access (DUA)** である。



目的

---

## 目的

- 目的：FPGA の世界に理想的なコミュニケーションアーキテクチャ（DUA）を導入する。
  - 1 共通のコミュニケーションインターフェースの導入
  - 2 全てのリソースに対してアドレスを付加し、リソースの場所に関わらずアクセス可能にする。
  - 3 ルーティングやリソースマルチプレクシングできるようにネットワークサービスを導入。

## DUA overview / DUA components

---

# DUA Overview

- 結果的にこのような実装により DUA を実現します
- 以降、この図 +  $\alpha$  を部分ごとに確認していきます

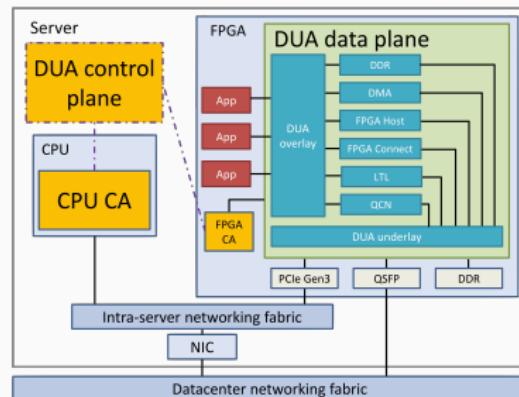


Figure 2: DUA architecture.

## Resource Address Format

- まずは **global, unified naming scheme** の話から。
- DUA では DC 内でグローバルにユニークなアドレスを各リソースにつける
- ただし新しいアドレスフォーマットを考えるのは懸命ではない → 様々なリソースの既存の命名法を活用する。
- UID = server ID : device ID : resource INST
  - server ID : IP address
  - device ID : 各サーバ内で unique な ID
  - resource INST: 各デバイスにおける既存の addressing scheme

UID (serverID:deviceID)	Address /port	Resource description
192.168.0.2:1	0x00000001CFFFF000	1st block of host DRAM
192.168.0.2:1	0x000000019FFFF000	2nd block of host DRAM
192.168.0.2:2	0x80000000	1st block of FPGA onboard
192.168.0.2:3	8000	1st application on FPGA
192.168.0.2:3	8001	2nd application on FPGA

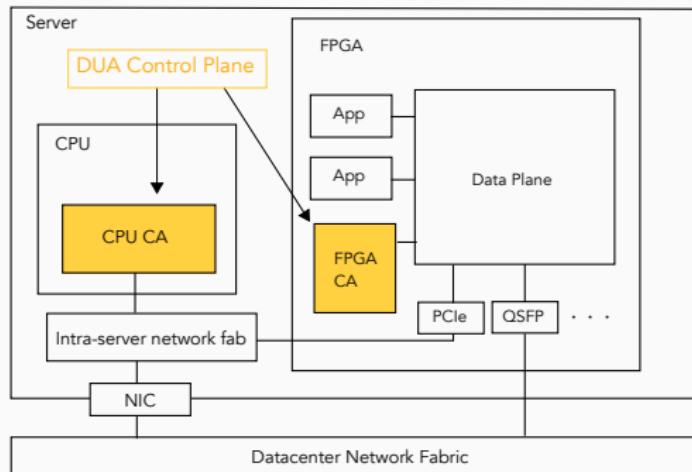
- 次に、Common Communication Interface の話
- 共通のインターフェース (API)
- プログラマはこのインターフェースを利用するのみを考えればいい。
  - アプリケーションでは BSD ソケットのようなノリで取り扱える。
  - Connection setup/close primitive (CONNECT/LISTEN/ACCEPT/CLOSE)
  - Data transmission primitive (SEND/RECV, READ/WRITE)
  - socket プログラミングみたいに両方で開いて送受信してクローズみたいな形のコード記述

## DUA underlay network

- 最後に、Underlying network service の話（これが長い。。。）
- 大きく分けて Control Plane / Data Plane に分かれている
- これらの協調作業により Routing や Resource Multiplexing を実現する

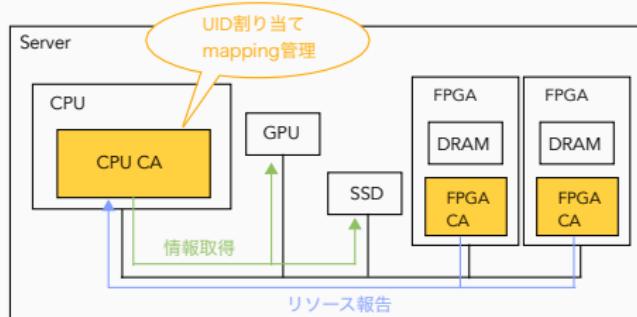
# DUA Control Plane

- まず DUA の Control Plane
- CPU Control Agent (CPU CA) と FPGA Control Agent (FPGA CA)
  - 複雑な control logic は CPU CA で行う
  - ローカルリソース監視や control plane command を Data Plane へ運ぶ役割
- CPU / FPGA CA で Resource Management, Routing Management, Connection Management を行う



# Resource Management

- FPGA CA がオンボードのリソースに関する情報を CPU CA に伝える
- CPU CA は上記 + その他メモリ/GPU 情報などを集め UID を割り当てる
  - device ID はホストでユニークに降るため、CPU CA は device ID とローカルリソースの mapping を持っている。
  - もちろん plug-in/plug-out/failure などでリソース変更が起きた際には mapping の update を行う。
- 現実装ではまだ naming service はないため UID 指定でリソースを利用する



## Routing Management

- 各サーバー内のリソースの接続情報から routing path を決定する。
- FPGA CA が communication stack と physical interface の情報を CPU CA に伝える
- CPU CA は各サーバーごとに interconnection table を保持、管理しこれをもとに routing path を決める
  - ただし DC network fabric との接続性を確認した場合は、他サーバのあるリソースを意味するエントリーを挿入する。
- FPGA1(192.168.0.2:4) → FPGA3(192.168.11.5:3/On Remote Host) のルーティングパスは、  
**FPGA1 → (FPGA Connect) → FPGA2 → (LTL) → FPGA3**

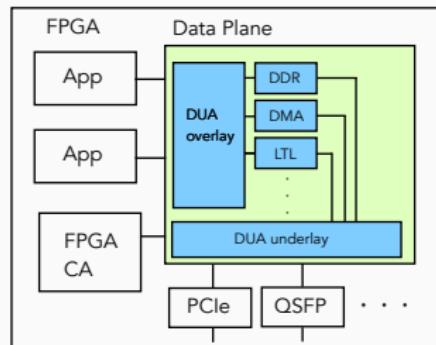
Src Resource (UID)	Dst Resource (UID) / Stack
FPGA 1 (192.168.0.2:1)	FPGA 2 (192.168.0.2:2) / FPGA Connect
	Host DRAM (192.168.0.2:3) / DMA
	Onboard DRAM (192.168.0.2:4) / DDR
FPGA 2 (192.168.0.2:2)	FPGA 1 (192.168.0.2:1) / FPGA Connect
	Host DRAM (192.168.0.2:3) / DMA
	Resources on other servers (*:*) / LTL

## Connection Management

- Connection の管理も Control Plane で行う。
- まずアクセス可能かどうかを policy で判断し、良ければ routing path を決定してそれに沿った forwarding table をデータプレーンに配る
- Stack によっては大量同時接続が非サポートのこともある（LTL は 64 しかない）が、複数の DUA connection を同一の tunnel connection でマルチプレクシングできる
- Application がコネクションをクローズしたときには Stack の tunnel connection をクローズし forwarding table を data plane から削除する。
- コネクション成立/失敗時にアプリケーションに通知する

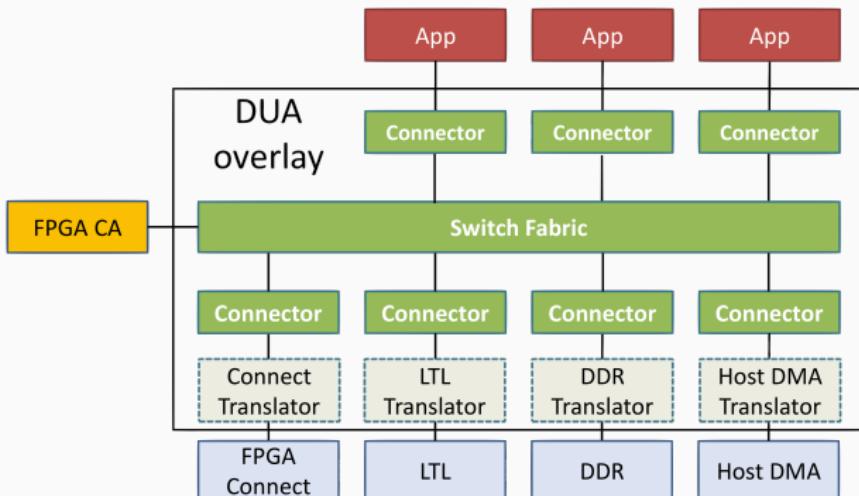
## DUA Data Plane

- 次に DUA の Data Plane
- overlay, stack, underlay からなる。
  - overlay : 異なる Application と Stack の間のデータ転送を行う Router
  - stack : 既存の communication stack
  - underlay : stack と physical interface をつなぎ、異なる stack 用に物理インターフェース上で multiplexing する



# DUA Overlay

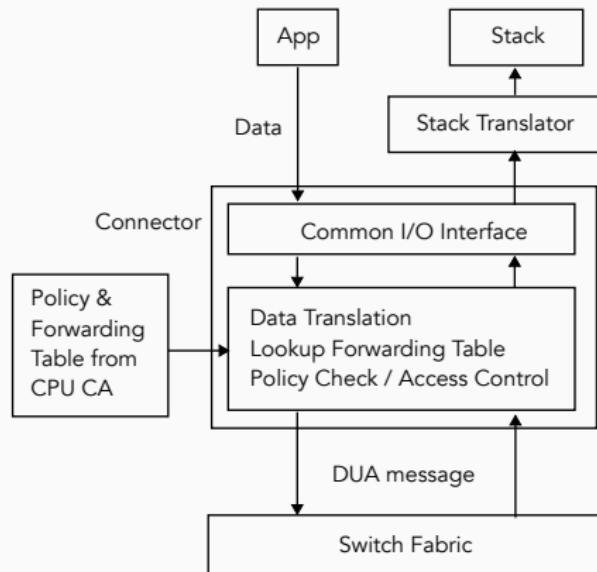
- overlay 部は更に、Connector, Switch Fabric, Stack Translator に分かれる。



- App/Stack と Switch fabric の間に存在する。
- 主に 2 つの機能がある。
  - データ転送 : I/O interface からのデータを encapsulate (DUA message) し switch fabric へ転送する / switch fabric からの DUA message を decapsulate し後続へ。場合によっては FPGA CA へ。
  - forwarding table の管理と lookup : forwarding table に沿って転送先 port を決定する
  - Access control : CPU CA で policy に基づいた routing path / forwarding table が決まり、policy 準拠の path のみが許される。

## DUA Overlay - Connector

- Connector を図にするとこんな感じ



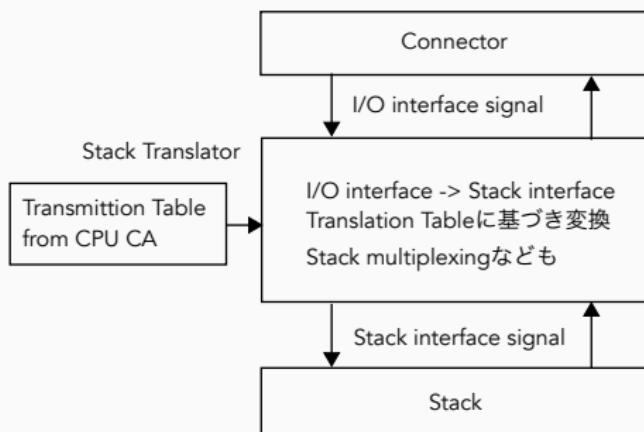
- App/Stack connector 間に存在する
- incoming connector から destination connector へ message を switch する
- バッファの話など細かい話があったがイマイチ何が言いたいのかわからなかつた。。。。

## DUA Overlay - Stack Translator

- Stack 側 Connector と Stack の間に存在する
- DUA interface を実際の Stack の interface へ変換する
- Control Plane が connection 設立時に stack translator に対応する translation table を各 stack translator へ送る
- translation table は、DUA message header と underlying stack header の mapping 情報でこれを元に変換する
- multiplex stack tunnel もここで制御できるらしい

## DUA Overlay - Stack Translator

- Stack Translator を図にするとこんな感じ

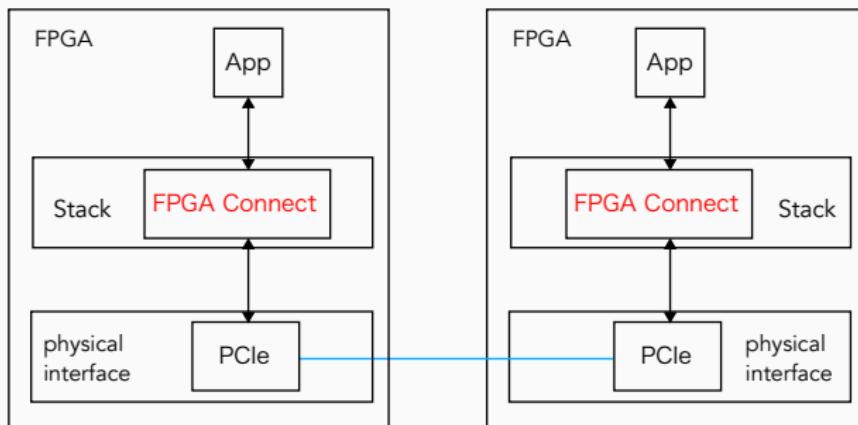


## Communication Stack

- 現実装では、既存の 4 つの Stack(LTL,DMA,FPGA-Host,DDR) を Data Plane に統合した。
- DUA は end-to-end の transport protocol に LTL を採用している。
  - LTL (Lightweight Transport Layer)
  - FPGA 間のネットワークを介した通信で UDP/IP 上に定義されるプロトコルらしい。
  - DC ネットワークの通信を FPGA から行うために利用し、輻輳制御は disable らしい。
- ただ、後で独自実装した FPGA Connect という Stack (こちらは PCIe 接続している FPGA 間での通信) も利用している

## FPGA Connect Stack

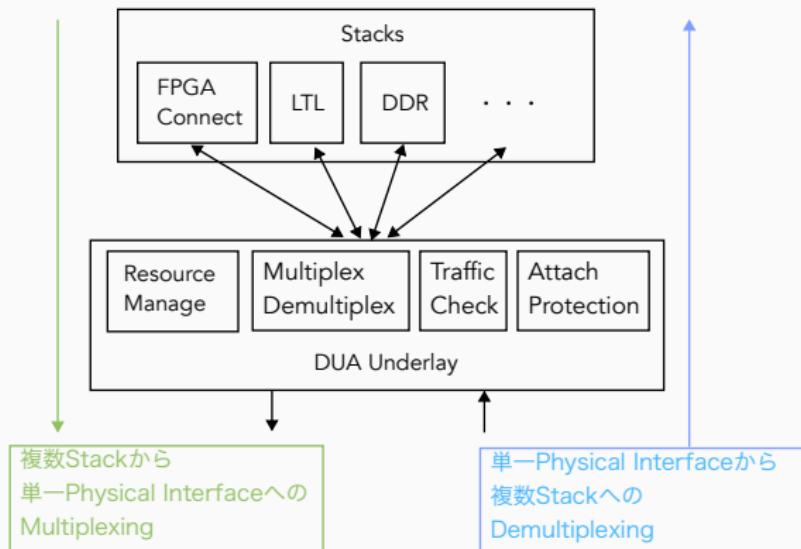
- 単一サーバー上に PCIe で接続された異なる FPGA 間で動作する複数 Application 間の direct communication をサポートする
- スライドでは詳細は省略。
- Microsoft 氏はこれも DUA の Data Plane に組み込んでいるよう。



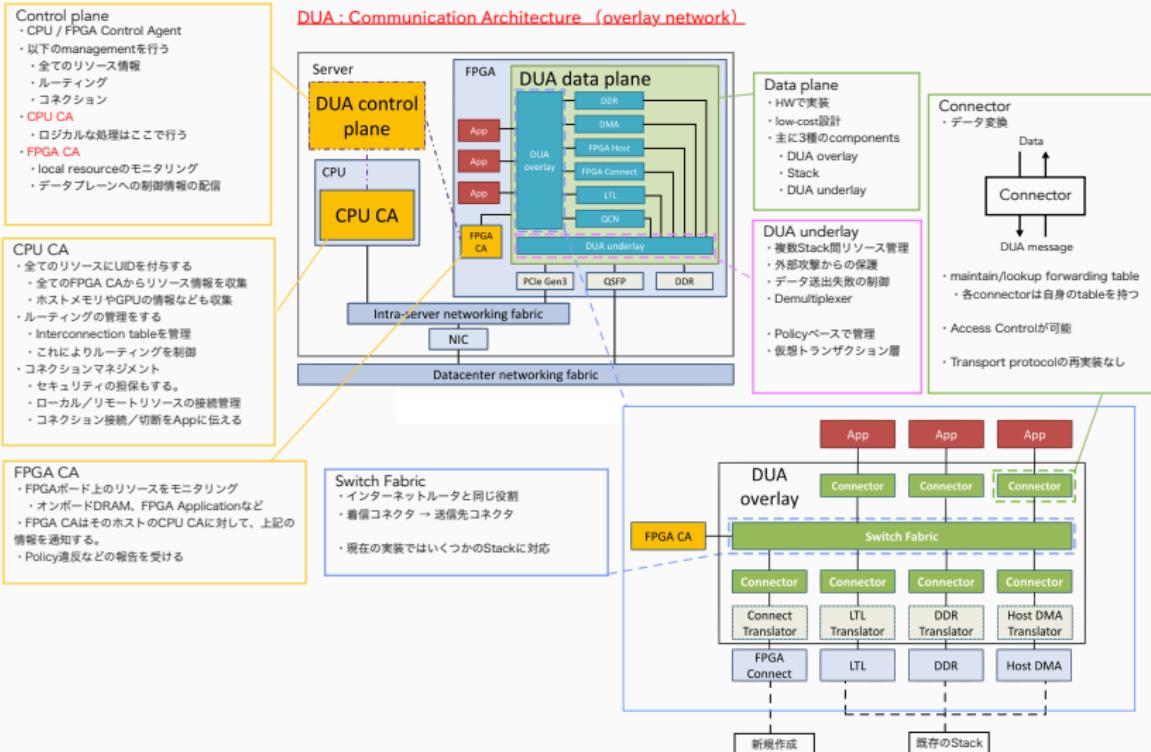
- Stack と Physical Interface の間に存在
- 主に以下のような処理を行う
  - 複数 Stack 間のリソース管理
  - 外部の攻撃からの Stack 保護
  - Traffic Check と FPGA CA への報告
  - Multiplexer/Demultiplexer

## DUA Underlay

- DUA Underlay を図にするとこんな感じ



# DUA Overview

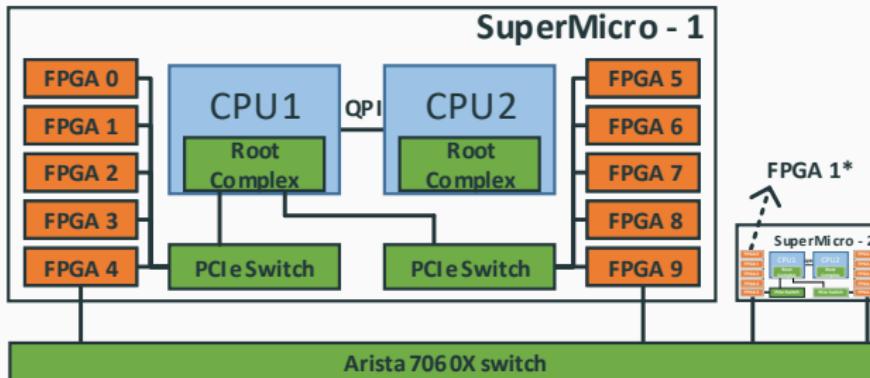


## Evaluation

---

# Evaluation

- MicroBenchmark と ApplicationBenchmark で性能を測定
- Testbed としては下図
  - Server : Supermicro SYS-4028GR-TR2 × 2 (一部の検証では Dell R720) with 40Gps NIC
  - FPGA : Altera Stratix V D5, 172.6K ALMs, 4G DDR3-1600 DRAM, PCIe Gen3 x8, 40GbE QSFP+ x2
  - Switch : Arista 706 0X
  - OS : Windows Server 2012 R2



## MicroBenchmark : FPGA Area Cost

- DUA の実装に関する FPGA リソースの消費の関係が下の表
- 4 stack (4-port switch, 4 connectors, 4 stack translator) には 9.29%
- Application 側に 4port 追加しても 19.86%
- Underlay は 4 stack - 3 physical interface で 0.25%
- Area cost としては低いし許容範囲  
(ハイエンド FPGA では無視できるレベル)

Component			ALMs	
DUA overlay	Switch fabric	2 ports	1272	0.74%
		4 ports	3227	1.88%
		8 ports	9366	5.45%
	Connector		3011	1.75%
	Stack translator	FPGA Connect	138.4	0.08%
		LTL	255.4	0.15%
		DMA	115.7	0.07%
		DDR	190.3	0.11%
DUA underlay	Stacks: FPGA Connect, LTL, DMA, DDR PHY interfaces: PCIe, DDR, QSFP		431.7	0.25%

# MicroBenchmark : Switch Fabric Performance

- スイッチファブリックの性能評価
  - 全 switch port は traffic generator/result checker の application に接続する
  - 混雑したトラフィックシナリオを想定してメッセージ送信
  - メッセージサイズは 32B から 4KB まで変化
  - switch fabric は 300MHz で動作、理想のスループットは 9.6GBps
  - 2 時間のテストで Latency / Throughput を計測。
- スループットは理論上の最大値に到達 & レイテンシは低い

Number of ports	Latency (ns)			Throughput (GBps)
	min	avg	max	
4	53	326	1086	9.6
8	56	649	1903	9.6

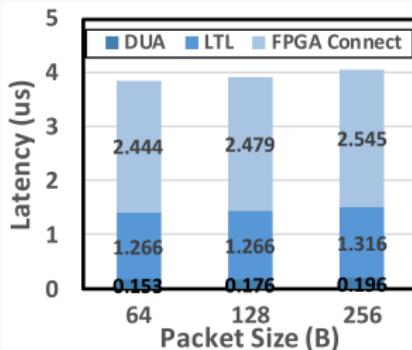
## MicroBenchmark : Routing Table Performance

- おそらく Connector での Forwarding Table matching の部分だと思う。。。
- 詳細がないが、Table entry に対する matching engine を組んだとのこと。
- table エントリと 1port あたりのエリアコスト、及び最大周波数をまとめた結果が下
  - 32 エントリのとき 1port あたり 0.83% エリアコスト、4-8port で考えると 3.3%-6.6% 程度
  - エントリ数が増加するとエリアコストはリニアに増加するが、最大周波数の減少はそれほど低下しない
  - このときのメッセージサイズの情報が書いてなかつたが、message-per-second の Throughput が clock frequency とおなじになる実装らしく、1port あたり 10GBps 位出るらしいので、計算するとだいたい 1message=20Byte 程度と思われる（多分）

Number of entries	ALMs (per port)	Fmax (MHz)
32	1435	0.83%
64	2810	1.63%
128	5571	3.23%

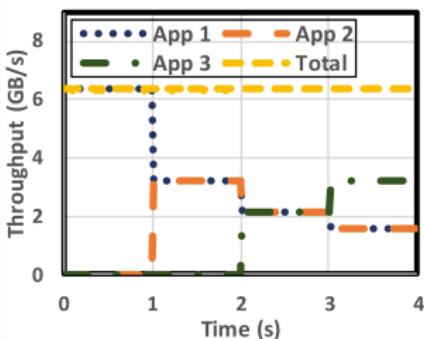
## Microbenchmark : Latency Overhead

- FPGA1 から FPGA1\* へのデータ送信
  - 1 FPGA1 → FPGA4 へ FPGA Connect でデータ送信
  - 2 FPGA4 → FPGA4\* へ LTL でデータ送信
  - 3 FPGA4\* → FPGA1\* へ FPGA Connect でデータ送信
- end-to-end の通信レイテンシを各レイテンシごとに確認
- Stack レイテンシに比べ、DUA のレイテンシ無視できるくらいはかなり低い



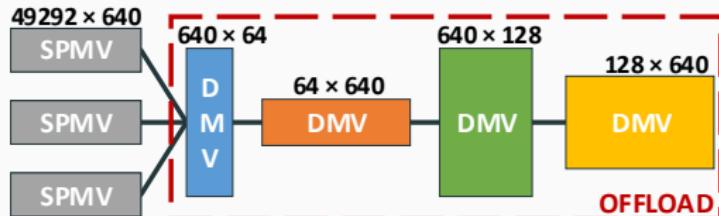
# MicroBenchmark : Handling Multiplexing

- DUA の Multiplexing 性能の測定
  - 同一の PCIe interface で Host DRAM にデータを書き込む Application1, 2, 3 (weight 1:1:2) を時間をずらしながら実行
  - 0s で App 1 を実行
  - 1s で App 2 を実行すると、App 1 と合計して Total Throughput に達成するよう平等にスケジュールされる
  - 2s で App 3 を実行すると、上記重みに従った Throughput に分かれる。
- 期待される Throughput への切り替わりもかなり早い



## ApplicationBenchmark : Deep Crossing

- Bing で使用されているらしい deep neural network らしい
- サービス応答に関わるレイテンシが重要
- モデル構造のうちの一部がベクトル計算集約的な部分のためそれを FPGA に offload し処理時間を短縮する
- 並列度をパラメータ化すると処理完了時間が短縮できるがロジックリソース消費量が増える
- 今回モデル内 DVM 全てを並列度 64 で実装し、最初 2 つと後ろ 2 つを分けて FPGA $\times 2$  で実装する
- 接続方式は FPGA Connect / LTL で確認



## Application Benchmark : Deep Crossing

- シングル FPGA は Area Cost 的に並列度 32 でしか実装できない。それとの比較
- シングル FPGA では発生しない Communication Latency が発生している
- 一方 DVM ごとの並列度が上がっているので、処理時間は向上しレイテンシは FPGA Connect では約 42%、LTL では約 37% 削減
- 2 つの FPGA ボードを繋ぐために必要な追加の OpenCL コード数は 26 行だった

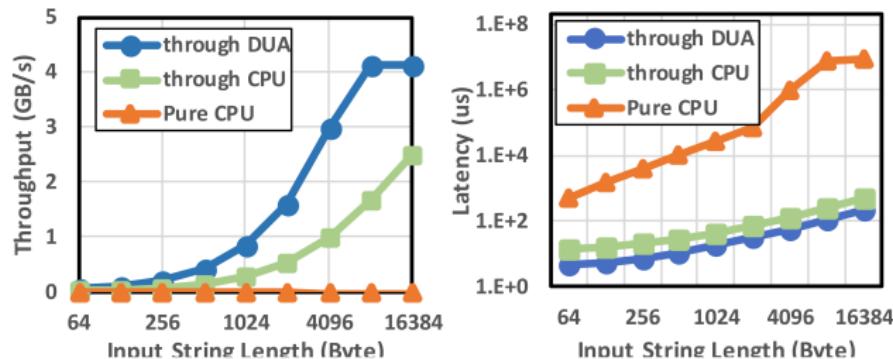
	Base	D1	D2	D3	D4	All
Parall = 32	26.0	11.0	9.4	11.0	9.8	67.2
Parall = 64		20.8	19.1	20.8	19.6	<b>106.3</b>

	D1	D2	D3	D4	Comm.	E2E
Single FPGA	7.27	6.58	13.30	12.96	N/A	40.12
Two FPGAs	4.17	3.48	7.22	7.09	1.419 (FC)	<b>23.38</b>
					3.354 (LTL)	<b>25.32</b>

- IDS(侵入検知システム) では典型的に正規表現が利用される
- DUA 上の 3FPGA で構成される fast multi-regular-expression matching プロトタイプを作成し測定
- 219 の正規表現ルールを作成し、3 分割 (73rules/stg) してステージ分けし、それぞれのステージを各 FPGA で実装した。
- 並列度は 32, 1cycle で 8bit 文字列をマッチングし、入力文字列がステージで照合完了して初めて次のステージに送られる
- 接続方式は FPGA Connect, 比較対象は CPU を通じた FPGA 間のデータ転送時、及び CPU でのマッチング性能。

# ApplicatonBenchmark : Multi-Regular-Expression Matching

- Latency と Throughput を測定
- Throughput : 文字列が大きくなると顕著になり、CPU を通じたデータ転送と比較しても 2 倍程度出る
- Latency : CPU を通じたデータ転送と比較してもあまり変化がないが、CPU 処理と比べると 3 倍近く高速になっていることがわかる
- 3 つの FPGA をつなぐために必要な追加の OpenCL コード数は 30 行未満だった



## 結論

---

## Conclusion

- FPGA の DC 利用を考えたときに出てくる各種問題を解決するような理想的なアーキテクチャを考え実装した
- FPGA から DC ネットワークのリソースを容易に利用できるようになった。
- 実装に関する Area Cost も低く、性能も Latency/Throughput の観点から非常に良いことがわかった。