【SWEST/ACRi 共同企画セッション】FPGAが変えるスーパーコンピューティングの世界

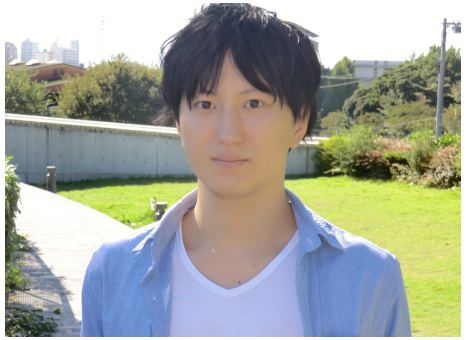
小林 諒平

東京科学大学 総合研究院 スーパーコンピューティング研究センター

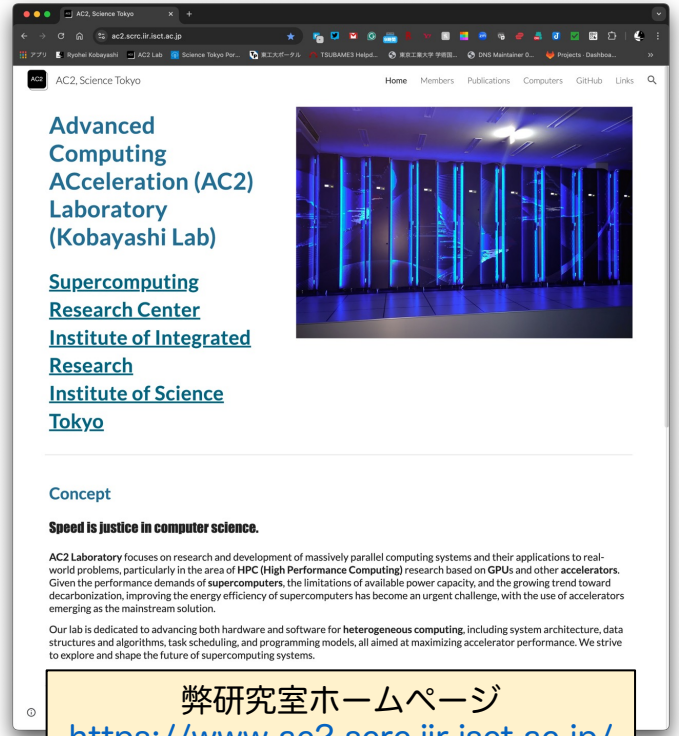
第27回 組込みシステム技術に関するサマーワークショップ（SWEST27）
セッションs3a
2025/08/29 10:30 – 11:40

自己紹介

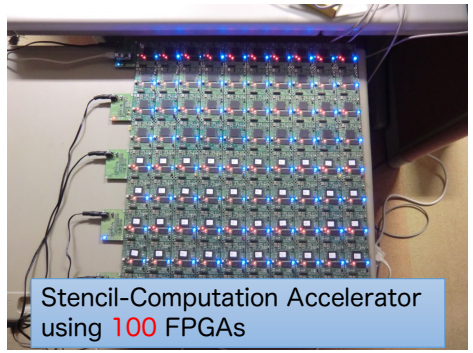
- 名前：小林 諒平
  - 東京科学大学 准教授
- 略歴
  - 2024年10月 ~ 現在東京科学大学 准教授
  - 2016年4月 ~ 2024年9月 筑波大学 助教
  - 2016年3月 東工大 博士 (工学)
  - 2013年3月 東工大 修士 (工学)
- 主な研究分野
  - 計算機システム，高性能計算，演算加速器 (GPU, FPGA, etc.)，並列プログラミング，並列アプリケーションの高速化
- もともとはFPGA屋です

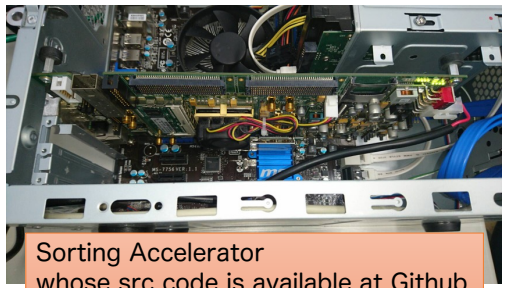


https://sites.google.com/site/ryokbya/



弊研究室ホームページ
https://www.ac2.scrc.iir.isct.ac.jp/


Stencil-Computation Accelerator using 100 FPGAs


Sorting Accelerator
whose src code is available at Github
https://github.com/monotone-RK/FACE

アカウントもってる
SNSなどなど
(お気軽にフォロー下さい)

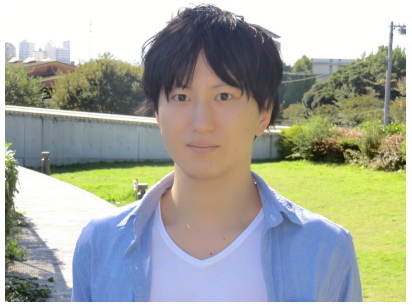
先端演算加速研究室 (小林研究室)
Advanced Computing ACceleration (AC2) Laboratory

研究室HP
https://www.ac2.scrc.iir.isct.ac.jp/

当研究室のコンセプト

コンピュータサイエンスにおいて速いことは正義である

当研究室では，スーパーコンピュータのための演算加速技術や
演算加速する実アプリケーションを研究していきます
(要はスパコンに関することは何でもやります！)



研究室主宰者：小林 諒平
https://sites.google.com/site/ryokbya/

- 研究キーワード
  - OpenMP
  - MPI
  - OpenACC
  - SYCL
  - 演算加速器
  - GPU
  - FPGA
  - リコンフィギャラブルコンピューティングシステム
  - 並列アプリケーションの高速化
  - etc...

場所はすずかけ台 (来年度から横浜)
キャンパスです
- G2棟11階

スーパーコンピュータ Pegasus
(筑波大学 計算科学研究センターで稼働中)

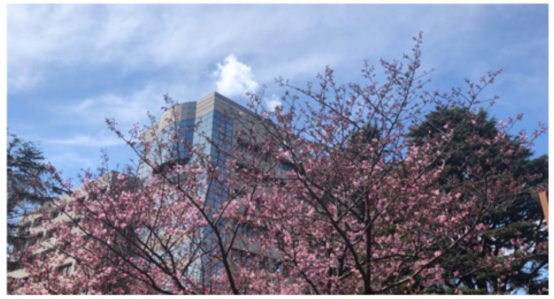
スーパーコンピュータ TSUBAME 4.0
(今日も元気にすずかけ台キャンパスで稼働中)

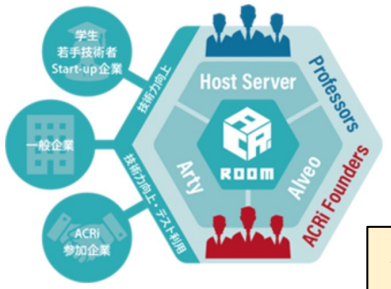
ACRiとは？

- 正式名称は Adaptive Computing Research Initiative（アダプティブコンピューティング研究推進体）
  - 要は日本の FPGA を盛り上げていこうと頑張っている団体です

ACRiルーム

アダプティブコンピューティング研究推進体 (ACRi) では、FPGA を活用する高性能なアダプティブコンピューティング・システムの開発及びその設計を効率化するための FPGA 活用基盤の開発を推進するために、100枚を超える FPGA ボードを含むサーバ計算機をリモートからアクセスして利用できる FPGA 利用環境として ACRi ルームを開設しました。





- 3時間単位で機材を無償で貸出し
- リモートからアクセスして利用するスタイル
- Forum を通じて大学教員や ACRi 企業が技術支援を実施

タダで使えるFPGA開発環境 ACRiルーム

https://gw.acri.c.titech.ac.jp/wp/manual/welcome

FPGAやりたいけど買うの・環境作るの面倒くさい，という方は是非ご利用をご検討頂ければと思います．

YouTubeを使ったウェビナーを定期的に開催

FPGA技術に関する交流会を現地開催

高性能計算と演算加速器

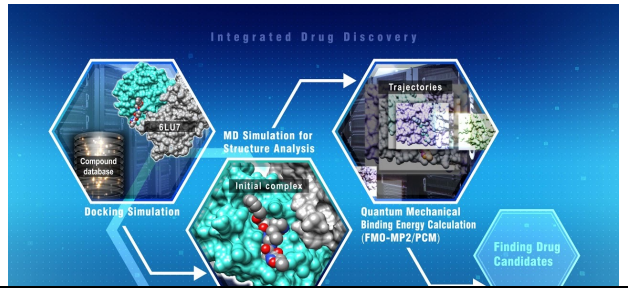
[1] https://www.ccs.tsukuba.ac.jp/pr/movie/#photo
[2] https://www.youtube.com/watch?v=KZrUbk13BC4
[3] https://www.nvidia.com/ja-jp/data-center/v100/
[4] https://www.bittware.com/ja/fpga/520n/

Institute of SCIENCE TOKYO

●高性能計算 (HPC: High-performance computing)

- 大規模かつ高性能なコンピュータシステムを用いて，膨大な計算を必要とする問題を高速に処理することを目指す研究分野

大規模かつ高性能なコンピュータシステム
(例：Cygnus スーパーコンピュータ [1])



膨大な計算を必要とする問題
(例：COVID-19対応創薬シミュレーション [2])

●演算加速装置 (アクセラレータ) の活用が HPC のトレンド

- プログラム中の支配的な演算部を加速するためのハードウェア
  - ✓ スパコンの電力あたりの性能 (電力効率) を引き上げるのが狙い
- 広く多用されているのは GPU (Graphics Processing Unit)
  - ✓ ただし，部分的に並列性が縮小したり，条件分岐が頻出するような不規則な演算は苦手
- そこで，上記の欠点を FPGA (Filed-programmable gate array) でカバーし，GPU と併用するアプローチに着目する

NVIDIA Tesla V100 (Volta) GPU [3]

BittWare 520N
Intel Stratix 10 FPGA ボード [4]

FPGAのHPC利用

Institute of SCIENCE TOKYO

●近年のハイエンドFPGA

- 真の意味でアプリケーションとハードの codesigning
- プログラミング環境： OpenCL, C++, etc.
- 高性能の光インターコネクト: 100Gb x N
- 柔軟な精度コントロールが可能
- （比較的）低電力

BittWare 520N with Intel Stratix 10 FPGA equipped with 4x 100Gbps optical interconnection interfaces

●課題

- GPUより低い演算性能 (FLOPS)
- メモリテクノロジ，バンド幅: GPUより1世代程度遅い
- 高位合成ベースの開発コストは決して低くない
  - ✓ FPGAに特化した記述やコンパイラ指示子挿入
  - ✓ ライブラリ・デバッグツールの不足

各計算デバイスの特性

	performance (FLOPS)	external comm.	programming effort
CPU	△	○	◎
GPU	◎	△	○
FPGA	○	◎	× -> △? recently getting better

GPUが得意なことをFPGAにやらせようとすると爆死します (経験談)
→ 「FPGAは永遠の二番手」なので，使いどころの見極めが大事
→ 我々は GPU が不得手な部分のみのオフローディングに活用

各デバイスをアプリの特性に基づいて適材適所的に活用するのが重要

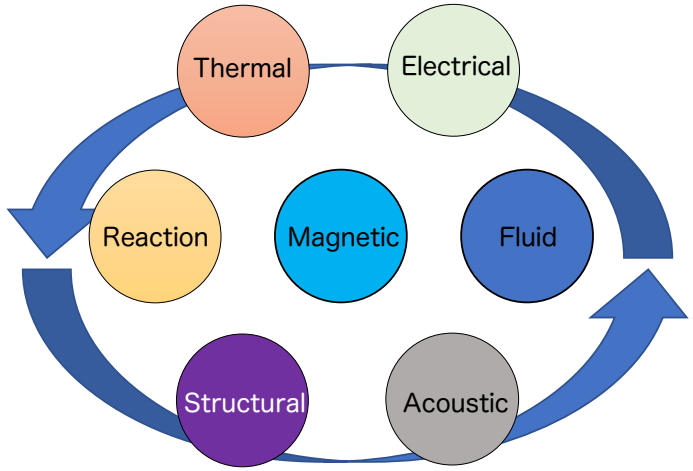
CHARM: Cooperative Heterogeneous Acceleration with Reconfigurable Multidevices

Institute of SCIENCE TOKYO

- GPUとFPGAを適材適所的に活用してアプリケーション全体の性能を向上
  - CPUは，GPUおよびFPGAの制御を担当 (カーネルを起動・両デバイスを同期)
  - GPUは，アプリケーション中の並列性が高い部分を実行
  - FPGAは，GPUの苦手とする演算および通信を実行 (演算と通信の融合)

multi-physics/multi-scale complicated problem

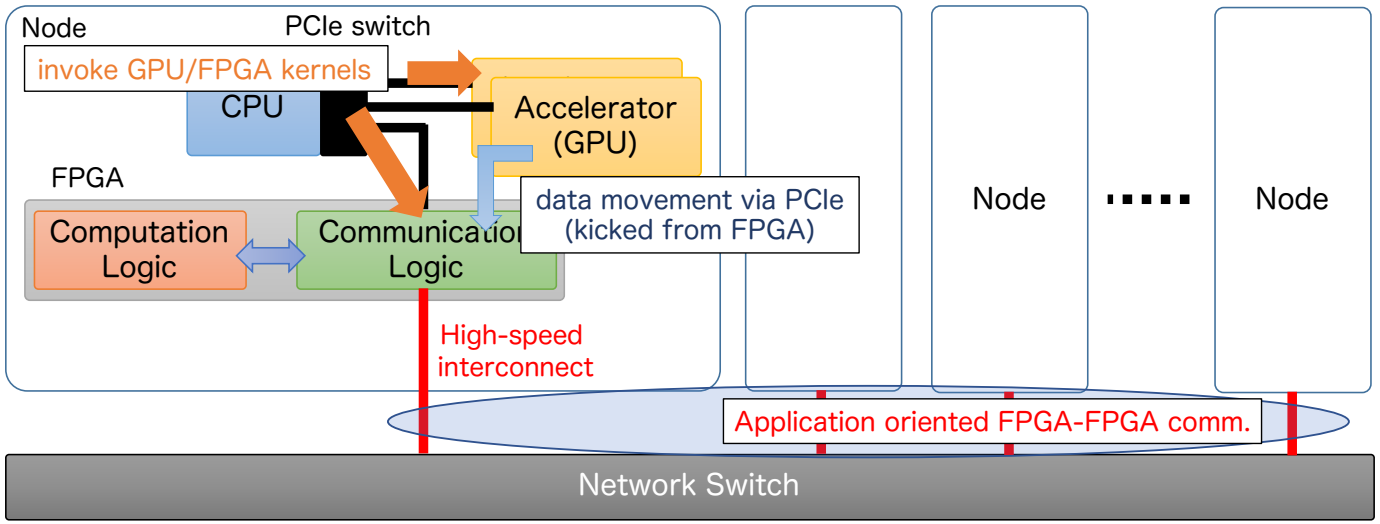
例：石炭の燃焼解析 (化学反応と流体計算)



シミュレーション内に様々な特性の演算が出現するので，
GPUに不適合な演算が部分的に含まれる可能性がある

→ GPUだけではアプリケーションを十分に演算加速し切れない

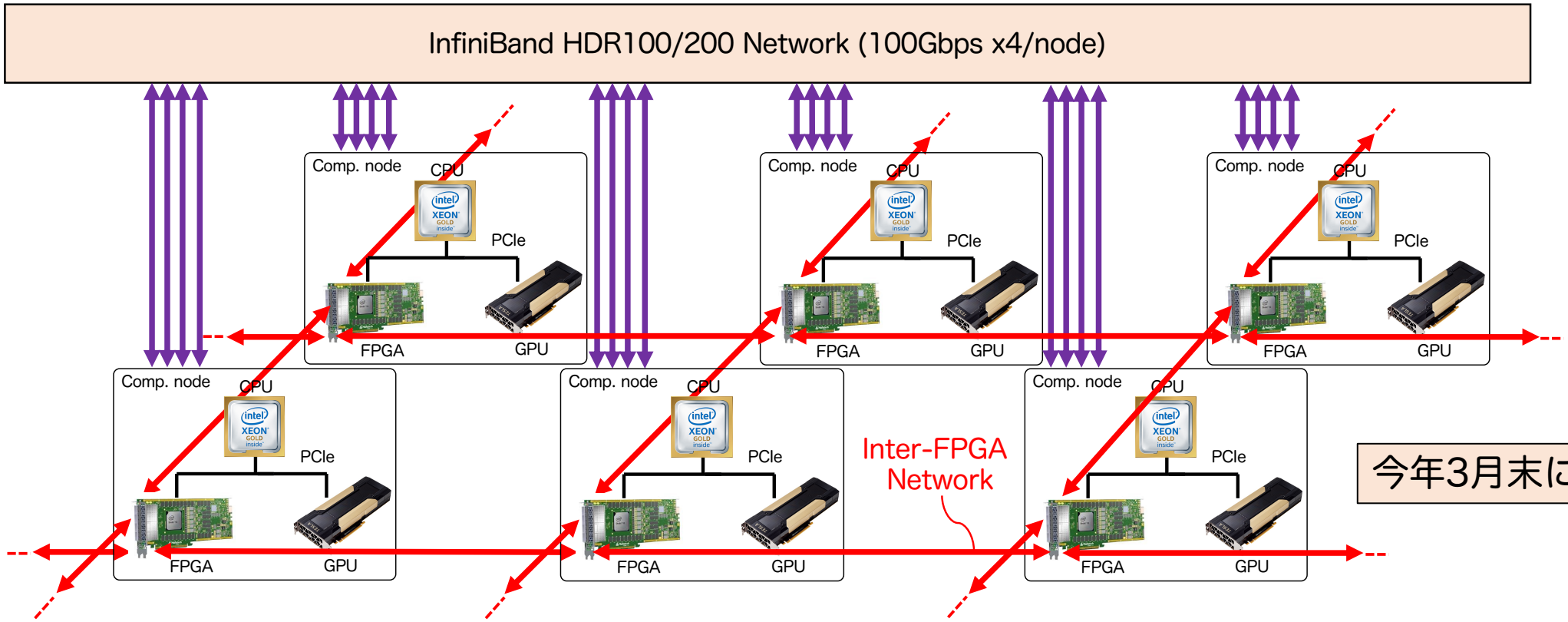
CHARMコンセプトの概要図



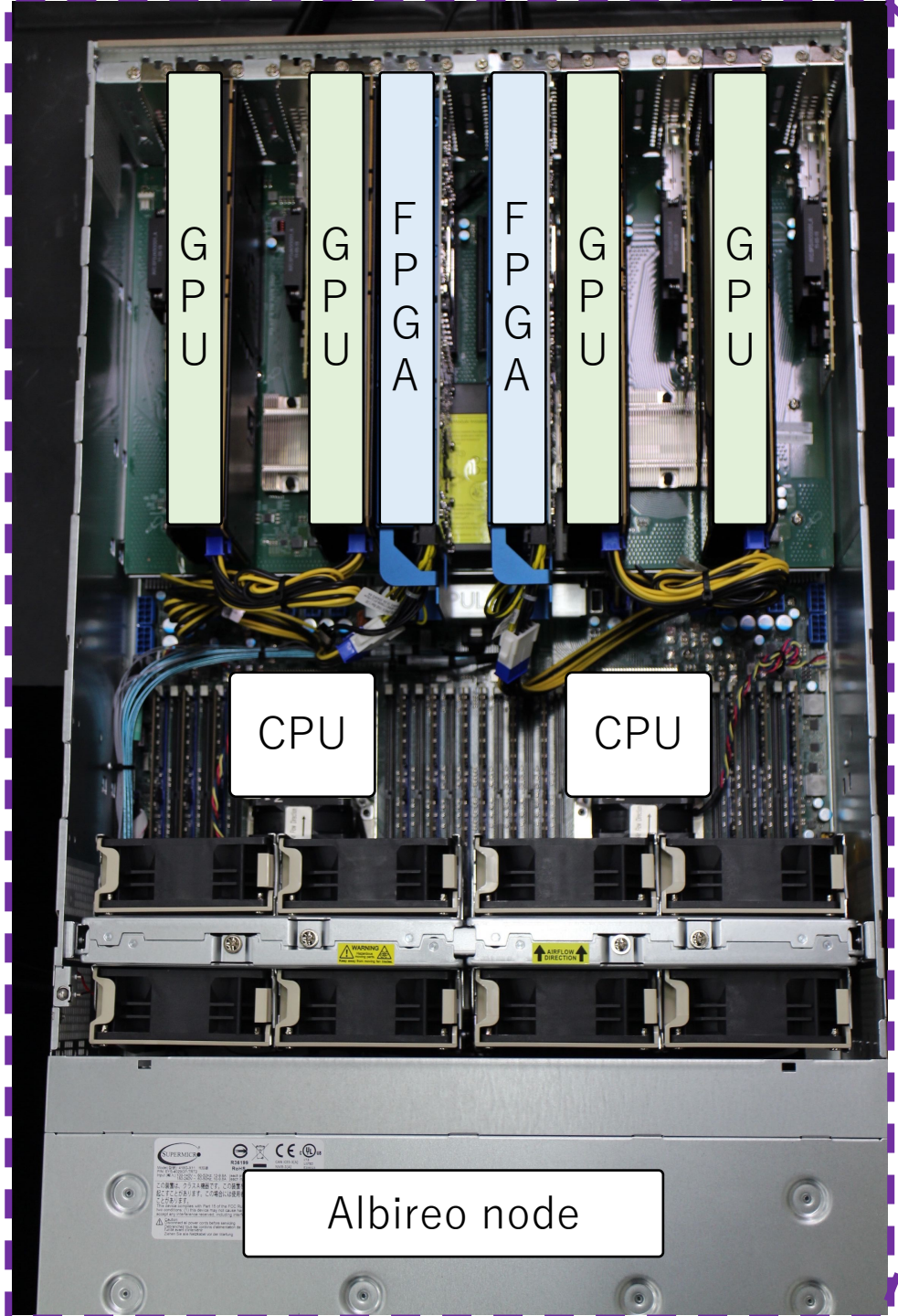
このコンセプトの実証システムがスーパーコンピュータ Cygnus

スーパーコンピュータ Cygnus の概略図

- GPUとFPGAをアクセラレータとして搭載 (Albireo ノードのみ)
  - DenebノードはGPUのみ搭載
- 2種類の相互結合網を有する
  - InfiniBand (全ノード共通，InfiniBand HDR100 x 4)
  - FPGA間直接網 (Albireoノードのみ，2次元トーラスネットワーク)



今年3月末に運用終了



IB HDR100 x4
→ HDR200 x2

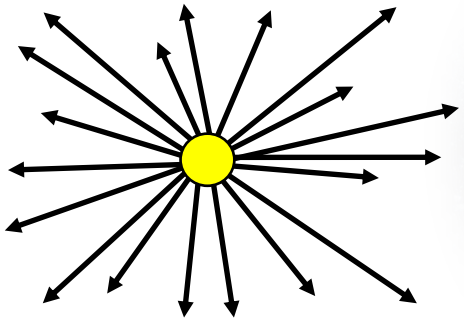
100Gbps x4
FPGA optical
network

IB HDR200
switch (for
full-bisection
Fat-Tree)

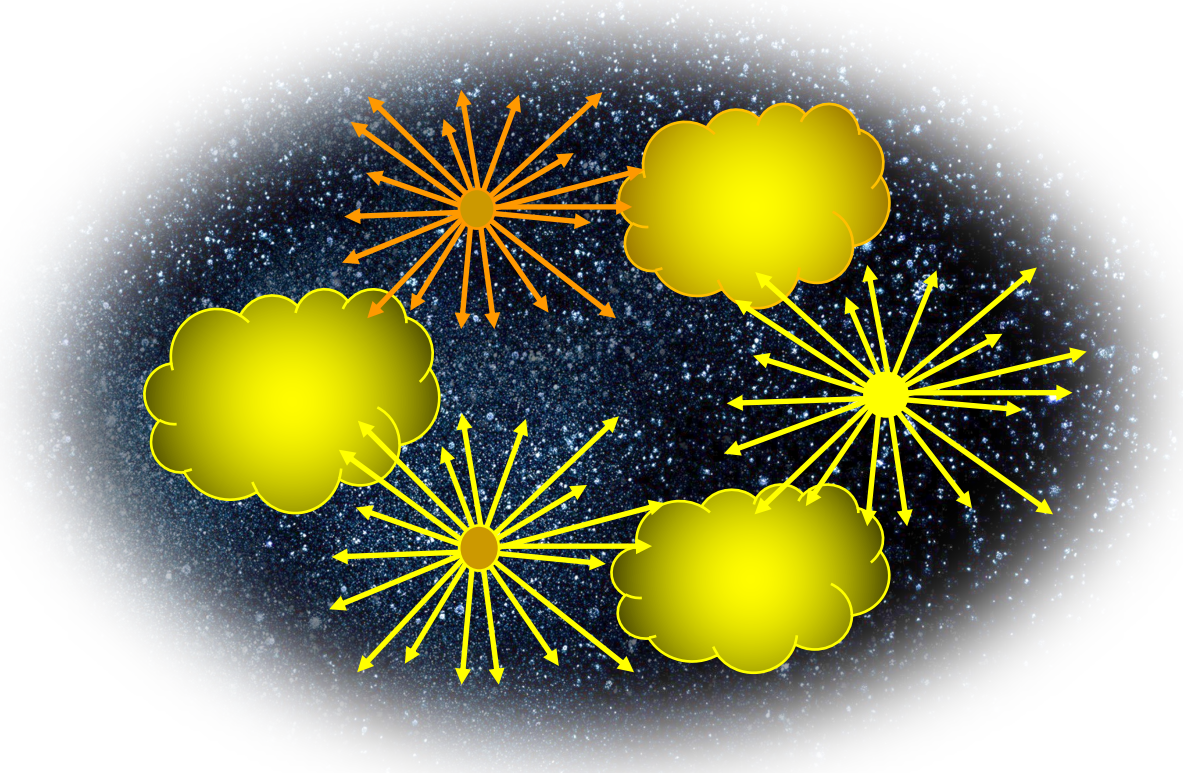
ターゲットアプリケーション

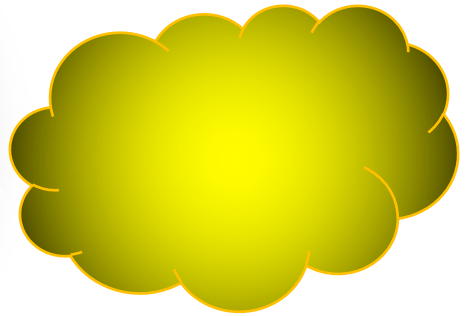
●宇宙輻射輸送シミュレーションコード ARGOT

- 点光源と空間に分散した光源の2種類の輻射輸送問題を組み合わせて，
  初期宇宙の天体形成をシミュレーションする (筑波大学計算科学研究センターにて開発)



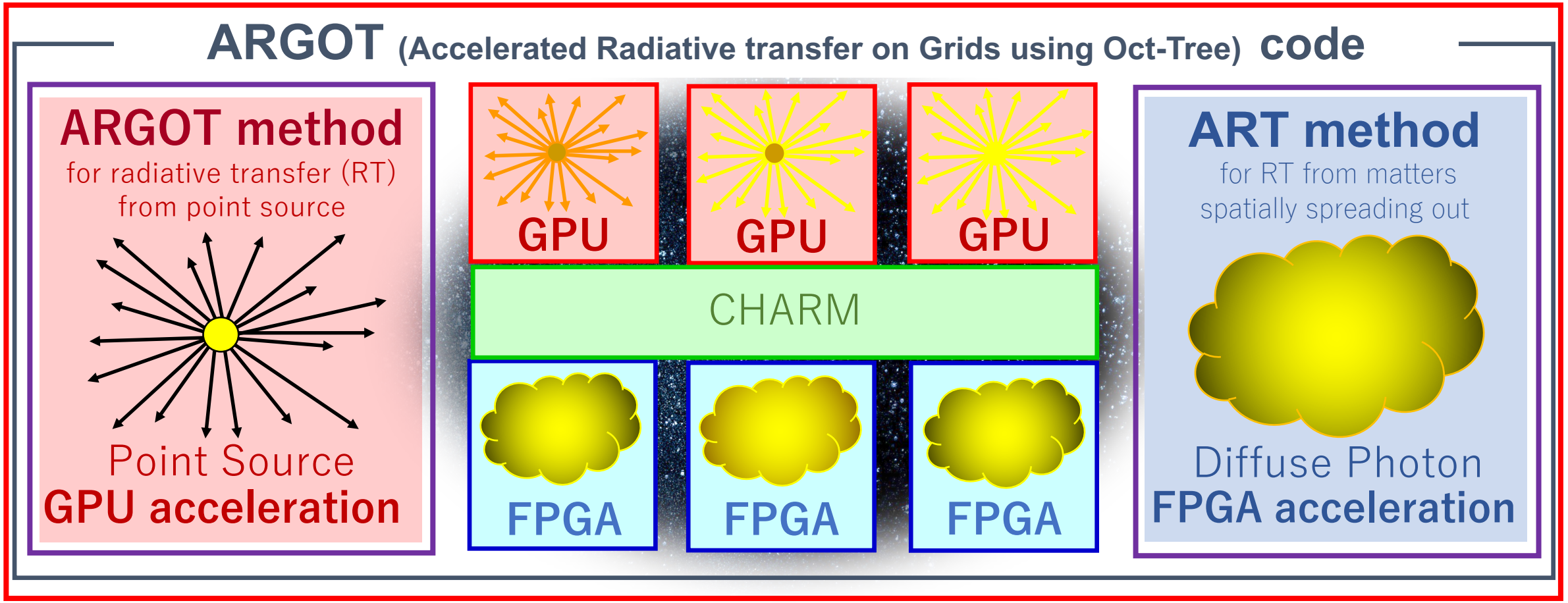
Point Source





Diffuse Photon

ARGOT code with CHARM

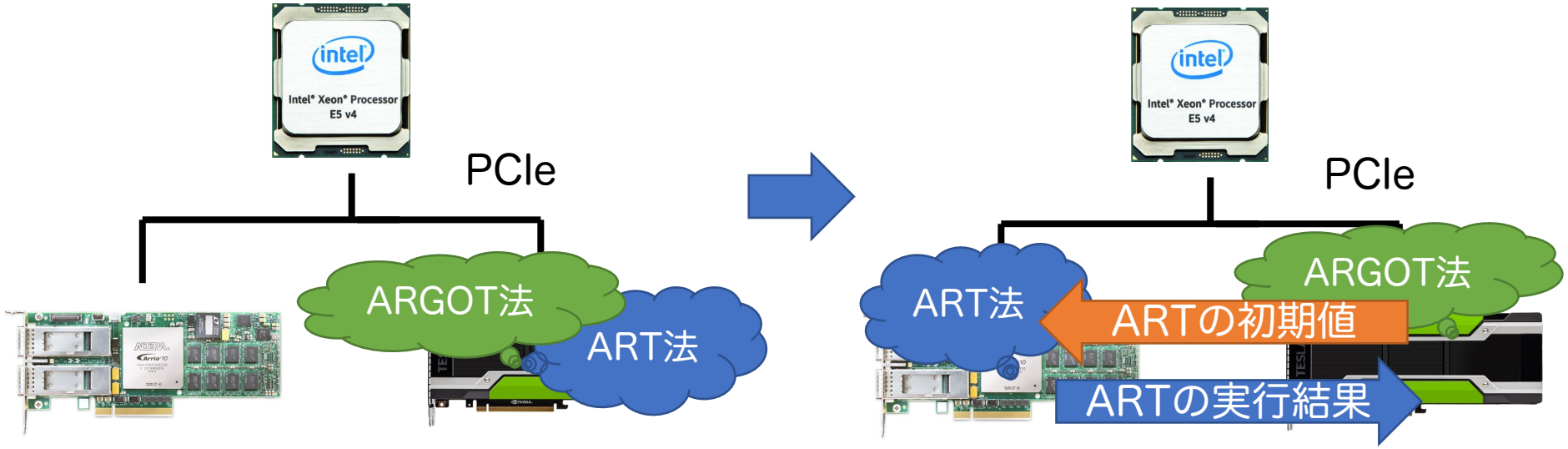


GPU・FPGA複合演算加速ARGOTコード

Institute of SCIENCE TOKYO

●実装の概要

- ARGOT法・ ART法をどちらもGPUで実装していたコードをベースに，ART法の部分をFPGA実装に置き換える
- ART法に必要な初期データは，GPUで生成しFPGAへ転送
- ART法の演算終了後，GPUに演算結果を返信する

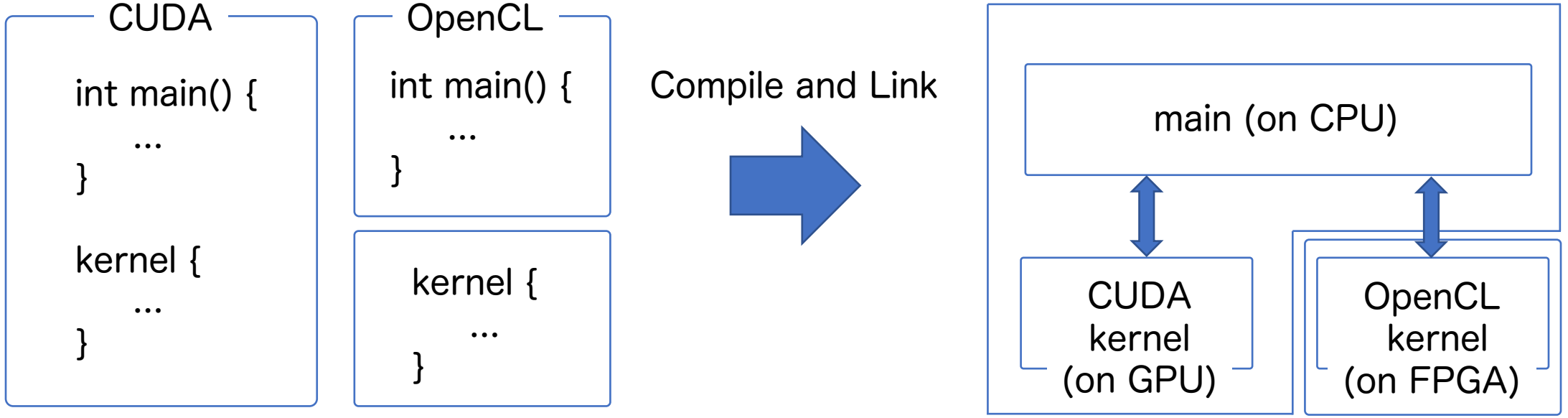


GPU・FPGA複合演算加速の実装方式

- CUDA と OpenCL の混合プログラミング
  - GPUで動作する計算カーネルをCUDAで記述
  - FPGAで動作する計算カーネルをOpenCLで記述

* 中道 安祐未, 小林 諒平, 藤田 典久, 朴 泰祐, “GPU・FPGA混載ノードにおけるヘテロ演算加速プログラム環境に関する研究”, HPC168

- これまでの研究* で CUDA と OpenCL はお互いにシンボルコンフリクトを起こさず，リンクできることが分かっている

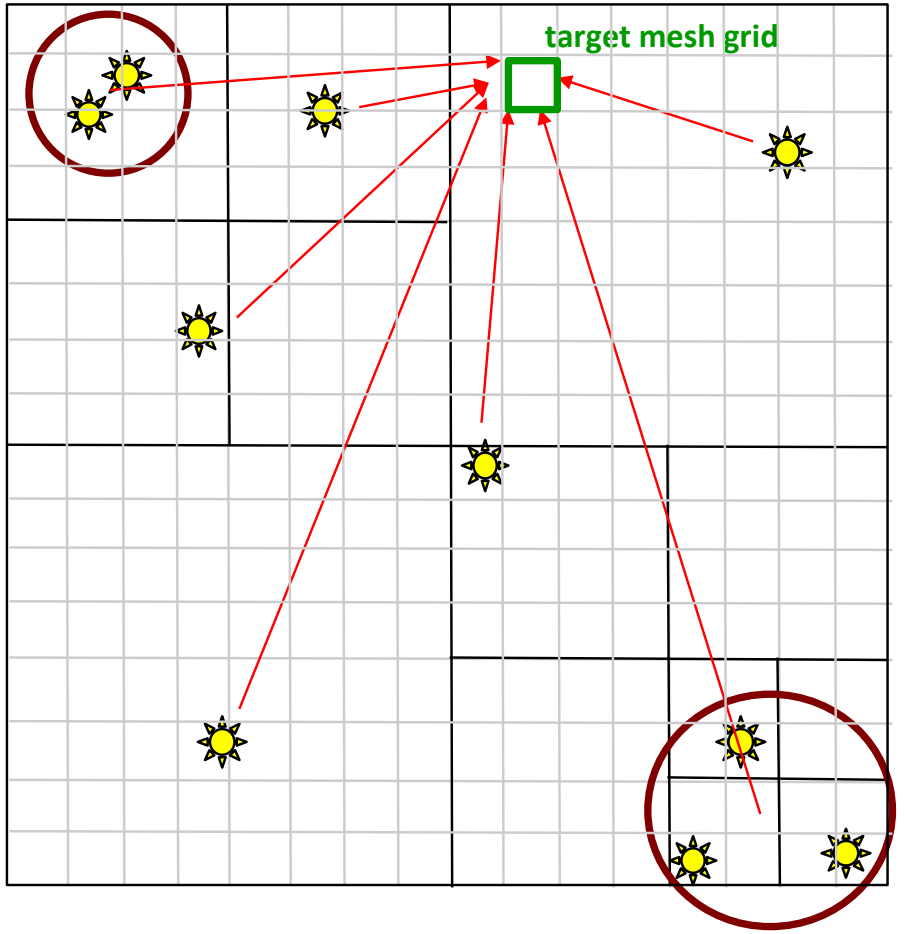


ARGOT法

- 点光源からの輻射輸送を計算
- 光源の分布を八分木のデータ構造で扱う
  - 3次元の問題空間なので八分木
  - ターゲットメッシュから見て，ある一定の角度の中に光源が入ると1つの光源として扱う，

effective number of sources : N_s ➡ log N_s

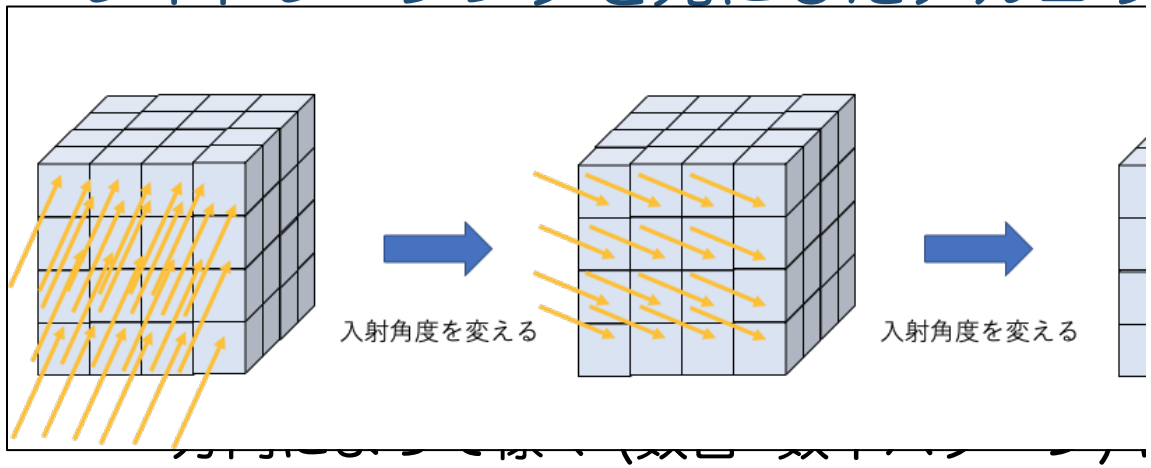
- 重力計算におけるTree-Codeに似た手法
  - GPU 実装に適している
  - 各光源からの輻射輸送を各スレッドにマップして，計算



T. Okamoto et al., "ARGOT: accelerated radiative transfer on grids using oct-tree", Monthly Notices of the Royal Astronomical Society, Volume 419, Issue 4, February 2012, Pages 2855–2866

ART法 (ARGOTコードで最も支配的な演算)

●レイトレーシングを元にしたアルゴリズム



✓本質的にランダムアクセス → 低いキャッ

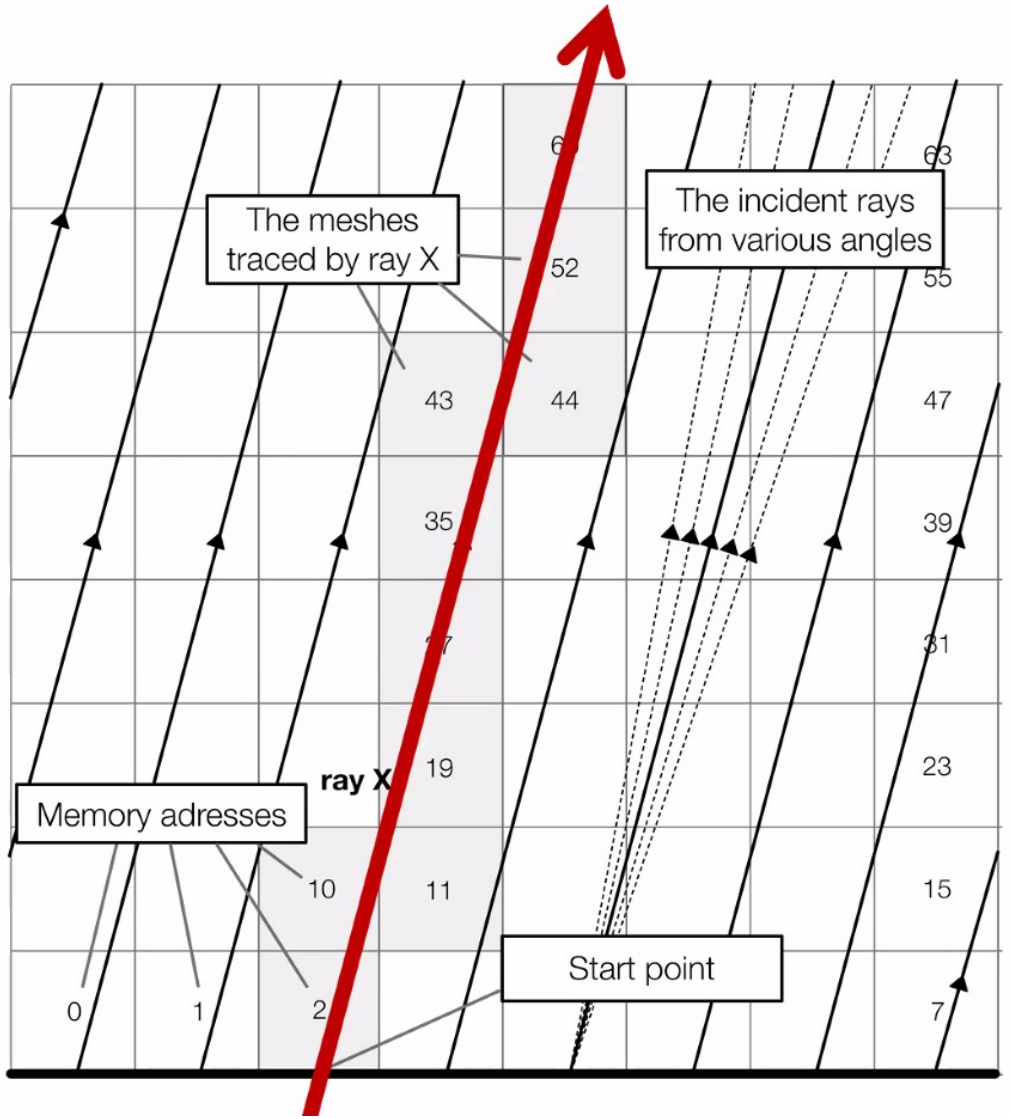
➢同じメッシュを隣接した複数のレイが通過
メッシュ上の複数のレイの効果を重ね合わ
ならない

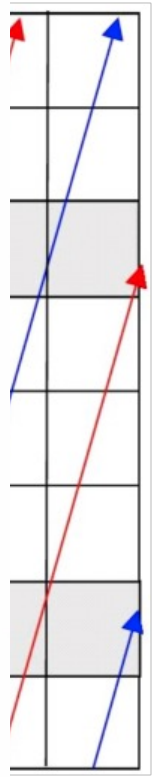
✓方法1: atomic演算を使う

- atomic演算自体がそもそもオーバーヘッ
ら使用するのは避けたい

✓方法2: 隣接するレイは同時に計算しない

- コア数に対して並列性が不足する
- メモリアクセスパターンがより飛び飛び
(ただでさえメモリアクセスで問題を抱え
なのにその問題を更に悪化させてしまう)



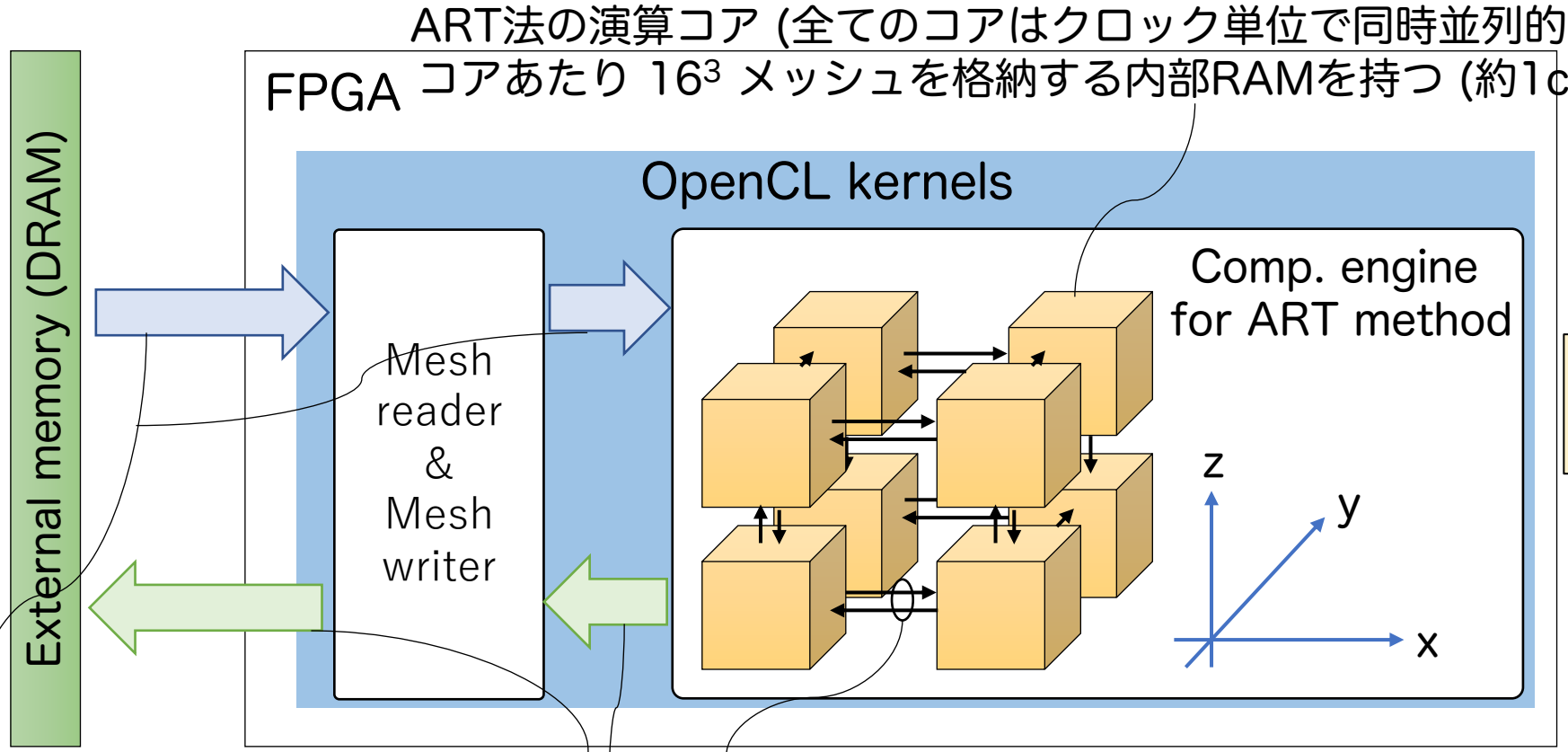


応を

scheme
parallel
mical
ust 2015,

ART法のFPGA実装

●コア間をIntel Channelで接続し，レイデータを転送して計算を実行



時間並列性 (パイプライン並列) および空間並列性を活用して ART 法の実効性能を最大化

→ Comm. path of ray data (Intel Channelを使用)

load mesh data into comp. engine

store comp. result (mesh data) in DRAM

```
#pragma OPENCL EXTENSION cl_intel_channels : enable

channel rt_ch[6][NPE_X][NPE_Y][NPE_Z];

// PE が保持するメッシュ空間
#define PE_MESH_SIZE (16 * 16 * 16)

// 1 つの面に 1 入力 / 1 出力 あるので、12 本のチャネルが必要
#define PE_INPUT_X_POS rt_ch[1][1][0][0]
#define PE_INPUT_X_NEG rt_ch[0][1][0][0]
#define PE_INPUT_Y_POS rt_ch[3][0][1][0]
#define PE_INPUT_Y_NEG rt_ch[2][0][1][0]
#define PE_INPUT_Z_POS rt_ch[5][0][0][1]
#define PE_INPUT_Z_NEG rt_ch[4][0][0][1]
#define PE_OUTPUT_X_POS rt_ch[0][0][0][0]
#define PE_OUTPUT_X_NEG rt_ch[1][0][0][0]
#define PE_OUTPUT_Y_POS rt_ch[2][0][0][0]
#define PE_OUTPUT_Y_NEG rt_ch[3][0][0][0]
#define PE_OUTPUT_Z_POS rt_ch[4][0][0][0]
#define PE_OUTPUT_Z_NEG rt_ch[5][0][0][0]

__kernel void PE_x0_y0_z0(...){
  // メッシュ空間のデータ (光学的厚みと source function) を格納する配列に Block RAM を
  __local struct radiation_mesh rmesh[PE_MESH_SIZE];

  // メッシュにおける光学的厚み、メッシュの source function を格納
  set_radiation_mesh(rmesh);

  // 768 x N^2 の全てのレイに対してレイトレーシングを実行 (N = 16)
  bool exit = false;
  while (!exit) {
    bool x_neg;
    bool x_pos;
    bool y_neg;
    bool y_pos;
    bool z_neg;
    bool z_pos;

    // レイの入力判定
    INPUT_COND(x_neg);
    INPUT_COND(x_pos);
    INPUT_COND(y_neg);
    INPUT_COND(y_pos);
    INPUT_COND(z_neg);
    INPUT_COND(z_pos);

    // 隣接 PE から入力がある場合、レイデータを受信する
    if      (x_neg) ray = read_channel_intel(PE_INPUT_X_NEG);
    else if (x_pos) ray = read_channel_intel(PE_INPUT_X_POS);
    else if (y_neg) ray = read_channel_intel(PE_INPUT_Y_NEG);
    else if (y_pos) ray = read_channel_intel(PE_INPUT_Y_POS);
    else if (z_neg) ray = read_channel_intel(PE_INPUT_Z_NEG);
    else if (z_pos) ray = read_channel_intel(PE_INPUT_Z_POS);

    // ART 法の演算カーネルを実行 (輻射強度の計算)
    calc_intensity(&ray, rmesh);

    bool x_neg_out;
    bool x_pos_out;
    bool y_neg_out;
    bool y_pos_out;
    bool z_neg_out;
    bool z_pos_out;

    // レイの出力判定
    OUTPUT_COND(x_neg_out);
    OUTPUT_COND(x_pos_out);
    OUTPUT_COND(y_neg_out);
    OUTPUT_COND(y_pos_out);
    OUTPUT_COND(z_neg_out);
    OUTPUT_COND(z_pos_out);

    // レイの次の計算領域が隣接 PE の保持するメッシュ空間である場合、レイデータを送信する
    if      (x_neg_out) write_channel_intel(PE_OUTPUT_X_NEG, ray);
    else if (x_pos_out) write_channel_intel(PE_OUTPUT_X_POS, ray);
    else if (y_neg_out) write_channel_intel(PE_OUTPUT_Y_NEG, ray);
    else if (y_pos_out) write_channel_intel(PE_OUTPUT_Y_POS, ray);
    else if (z_neg_out) write_channel_intel(PE_OUTPUT_Z_NEG, ray);
    else if (z_pos_out) write_channel_intel(PE_OUTPUT_Z_POS, ray);

    // レイトレーシングの終了判定
    CHECK_TERMINATION(exit);
  }
}
```

(x, y, z) = (0, 0, 0)
に位置する PE の擬似コード

Intel Channel
を宣言

通信経路を
マクロで分かりやすく

1コア分のOpenCLカーネル関数

メッシュデータを格納する
ワーキングメモリを宣言

ワーキングメモリの初期化

ART法の演算ループ
(この部分がパイプラインHWとして実装される)

隣接コアからレイデータが来るか判定

レイデータの受信
(データが来る場合)

輻射強度の計算
(ART法で一番大事なところ)

隣接コアにレイデータが行くか判定

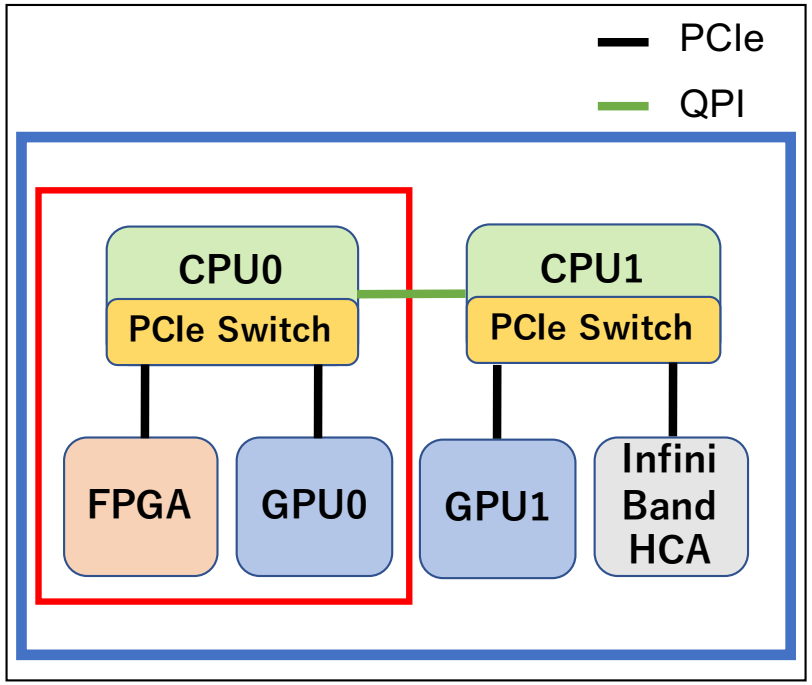
レイデータの
送信
(データが行く場合)

ART法の終了判定

評価環境

- 筑波大学計算科学研究センターで稼働中のPPX (Pre-PACS version X) クラスタシステムにて実装及び評価を行った
  - ミニCygnusのような実験用クラスタ

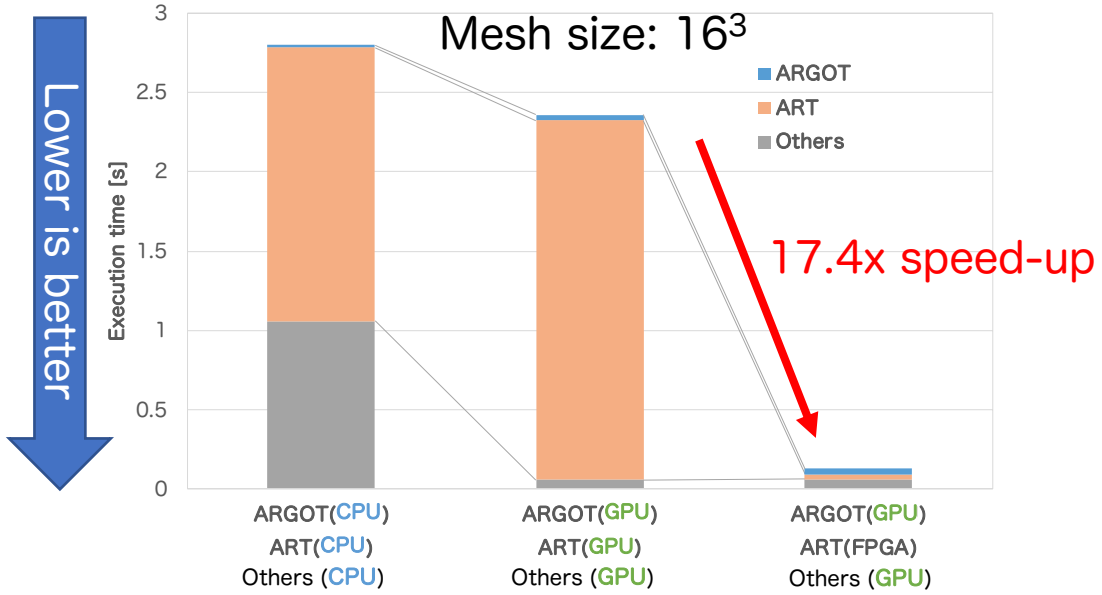
Hardware specification	
CPU	Intel Xeon E5-2660 v4 x2
Host Memory	DDR4-2400 16GB x4
GPU	NVIDIA Tesla P100 x2 (PCIe Gen3 x16)
FPGA	Intel Arria 10 GX (BittWare A10PL4) (PCIe Gen3 x8)
Software specification	
OS	CentOS 7.3
Host Compiler	gcc 4.8.5, g++ 4.8.5
GPU Compiler	CUDA 9.1.85
FPGA Compiler	Intel Quartus Prime Pro Version 17.1.2.304

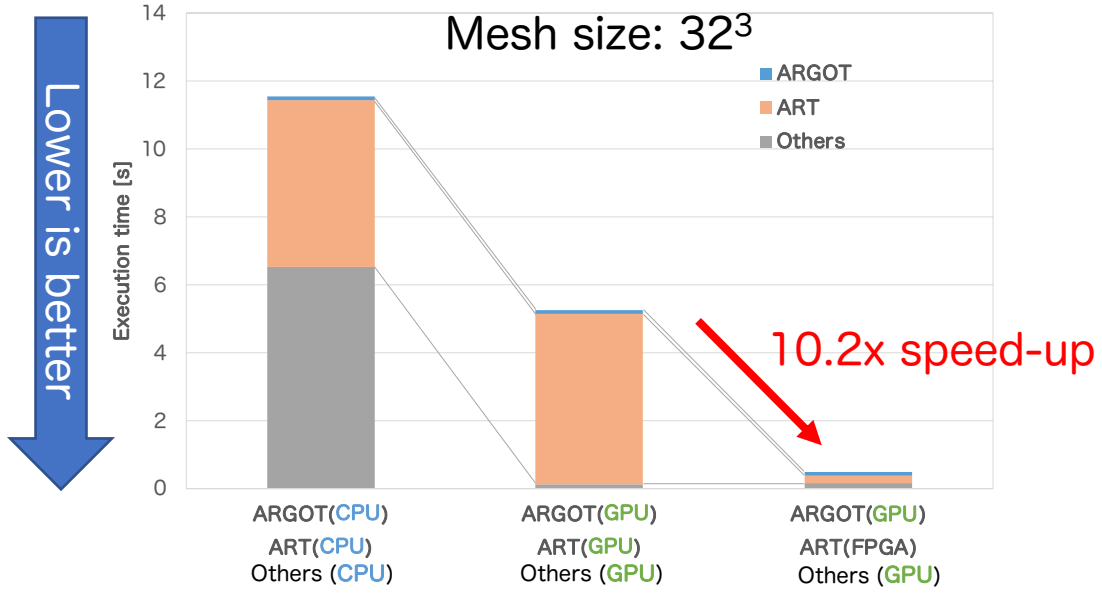


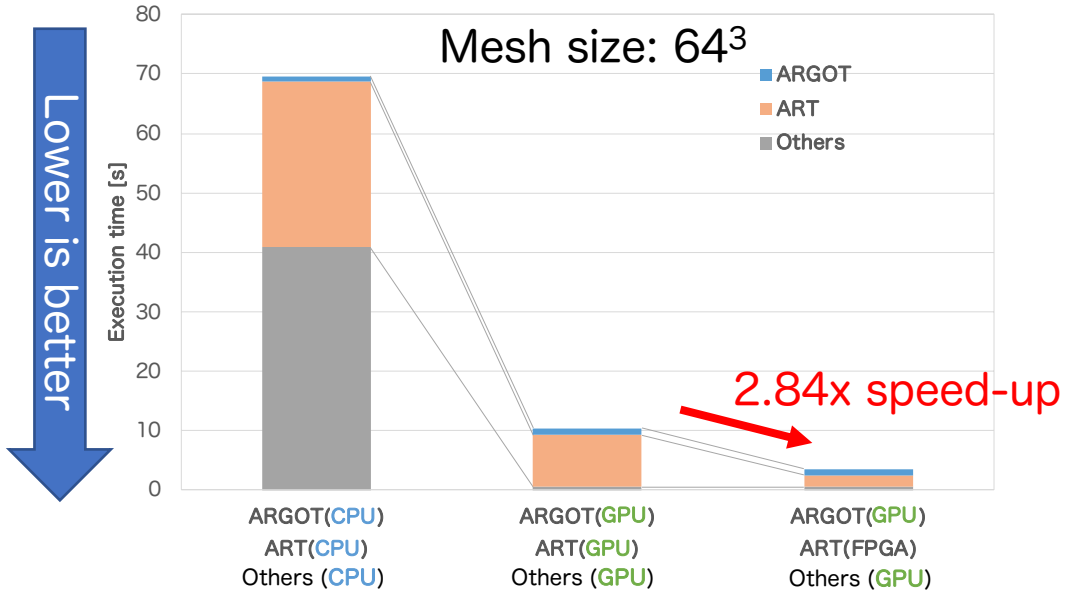
A computation node of PPX

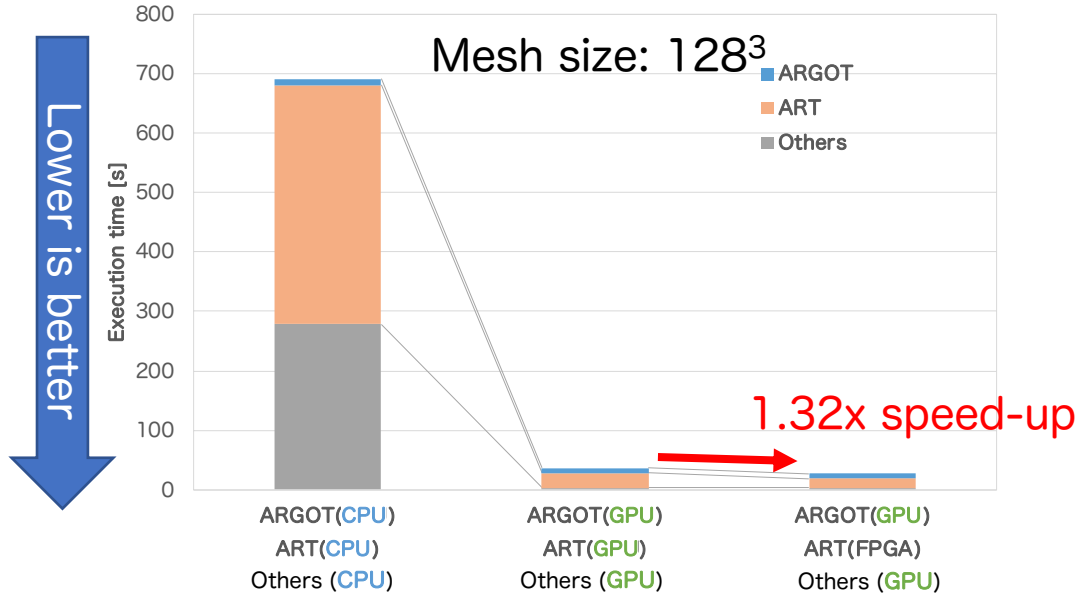
PPX (フロント側)

GPU・FPGA連携ARGOTコードの性能評価

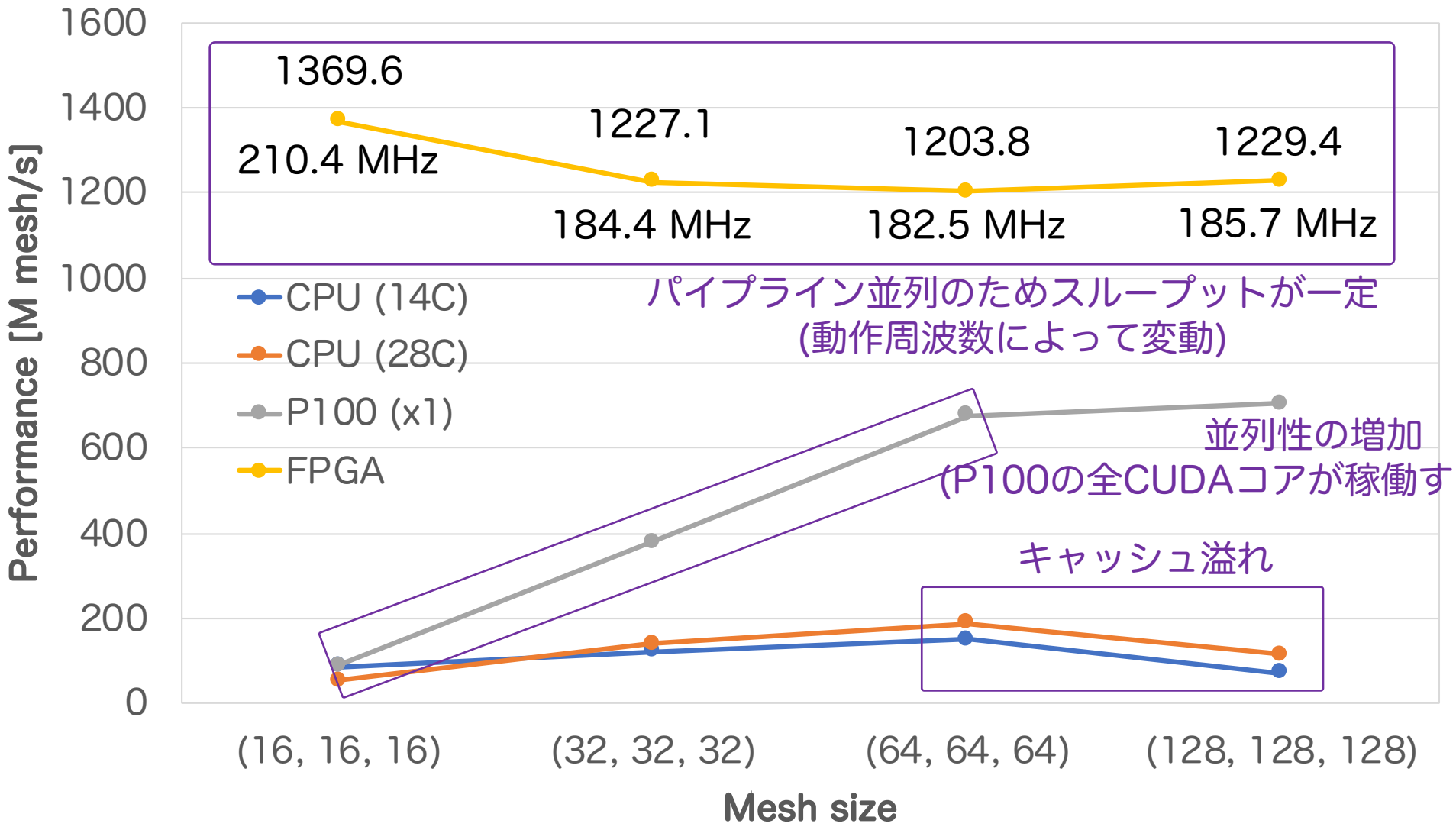








Performance evaluation of the ART method

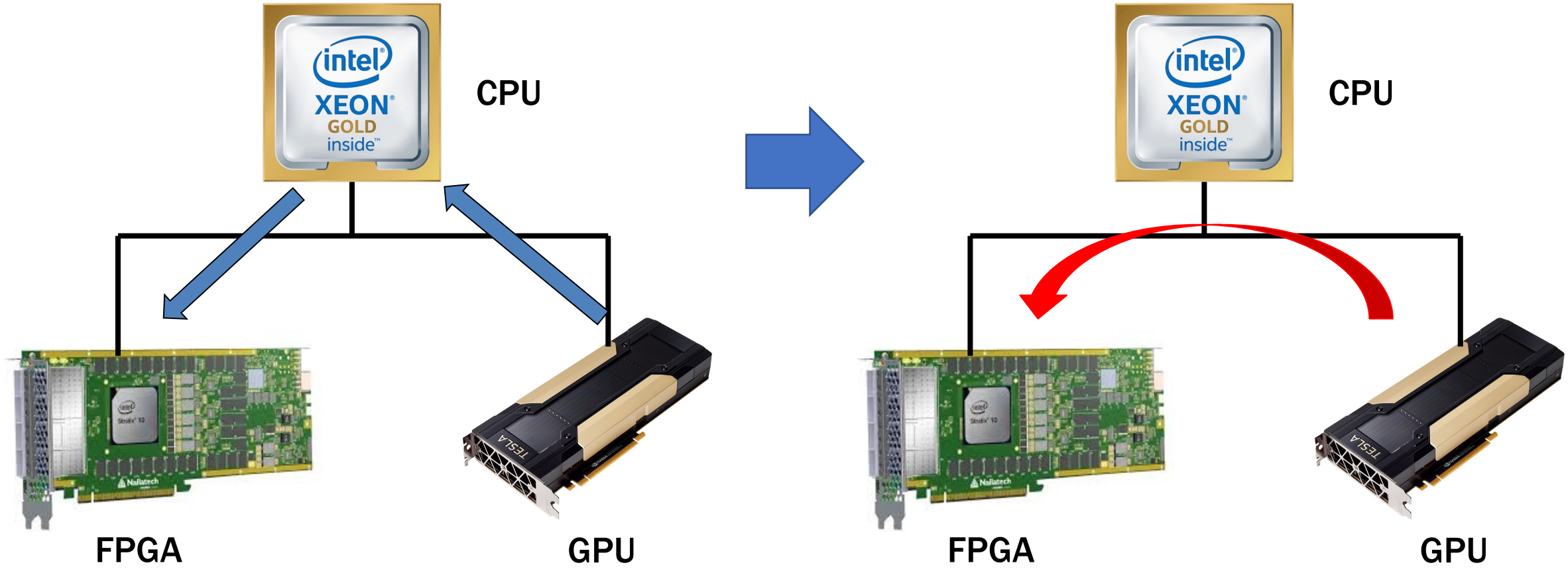


14C:
14 cores
(1 socket)

28C:
28 cores
(2 sockets)

GPU・FPGA連携ARGOTコードの更なる性能向上に向けて

- 両デバイス間のデータ転送を最適化
  - CPUを介さず，FPGA (OpenCLカーネル) が自律的にDMA転送できるようにする



GPU-FPGA間DMA転送

Institute of SCIENCE TOKYO

●概要

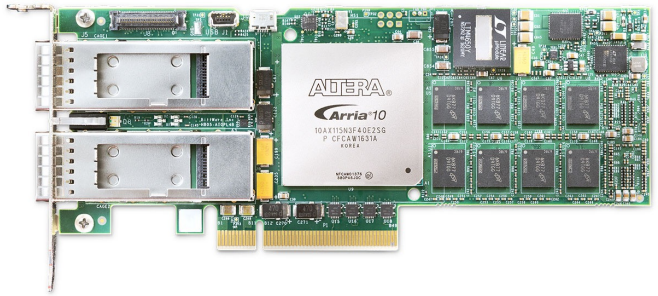
- GPUデバイスのグローバルメモリ，FPGAデバイスの外部メモリをPCIeアドレス空間にマッピングすることで，FPGAのPCIeコントローラIPの持つDMA機構を用いて双方のメモリ間でデータのコピーを行う

●DMAを実行する手順

- ① GPUデバイスのグローバルメモリをPCIeアドレス空間にマッピング
- ② ①でマップしたメモリアドレス情報をFPGAに送信
- ③ OpenCLカーネルからPCIe DMA機構を駆動

CPUが実行

FPGAが実行



① GPUデバイスのグローバルメモリをPCIeアドレス空間にマッピング

- 先行研究*のAPIを活用
  - PCIeアドレス空間にマップされたGPUメモリのアドレスである paddr を取得
  - 内部的にはNVIDIAが提供する Kernel APIを用いて GPUメモリアドレスをPCIeアドレスにマップし，そのアドレスを取得 (GPU Direct for RDMA)

```
1  #define SIZE  1000000
2
3  tcaresult tcaCreateHandleGPU(unsigned long long *
       paddr, void *ptr, size_t size);
4
5  int main(void) {
6    uint32_t data[SIZE/4];
7    void* ptr;
8    cudaSetDevice(0);
9    cudaMalloc(&ptr, SIZE);
10
11   unsigned long long paddr;
12   tcaCreateHandleGPU(&paddr, ptr, SIZE);
13
14   printf("paddr␣=␣0x%016llx\n", paddr);
15
16  return 0;
17 }
```

* Hanawa, T et al., Interconnection Network for Tightly Coupled Accelerators, 2013 IEEE 21st Annual Symposium on High-Performance Interconnects, pp 79-82

② ①でマップしたメモリアドレス情報をFPGAに送信

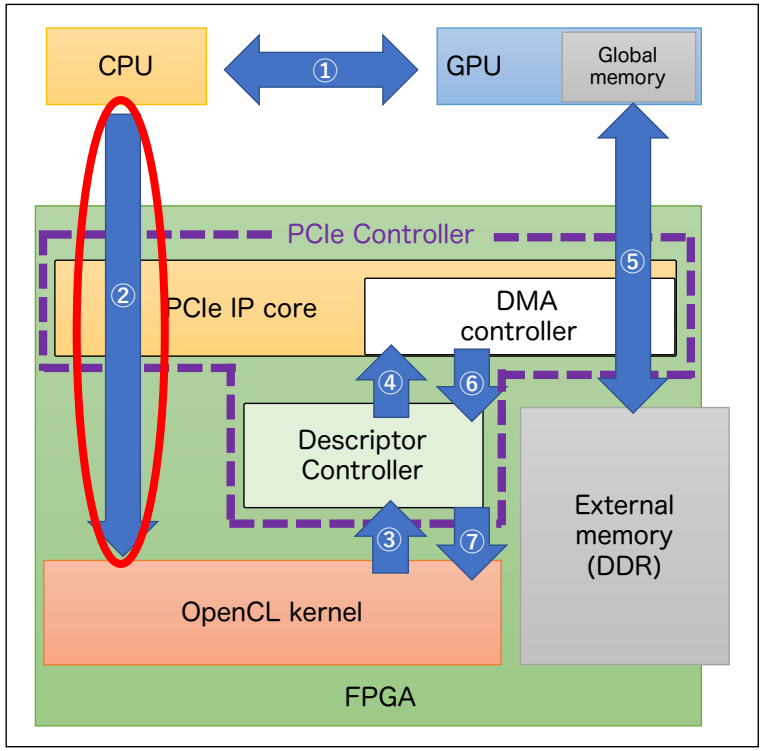
- OpenCL API (clSetKernelArg) を使って送信
  - OpenCLカーネルコードの引数にセット

Host code

```
status = clSetKernelArg(kernel, argi++, sizeof(unsigned long long), &paddr);
aocl_utils::checkError(status, "Failed to set argument PADDR");
```

Kernel code

```
__kernel void fpga_dma(
        __global uint *restrict RECV_DATA,
        __global const uint *restrict SEND_DATA,
        __global ulong *restrict E_CYCLE,
        __global const uint *restrict NUMBYTE,
        const ulong PADDR,
        const uint DEBUG_MODE
)
```



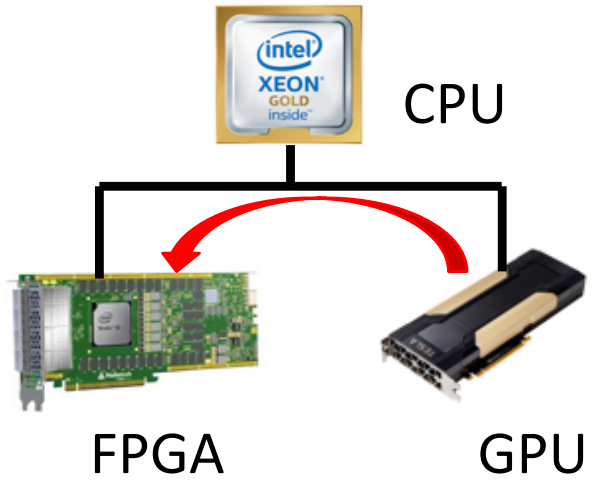
③ OpenCLカーネルからPCIe DMA機構を駆動

- DMA機構にディスクリプタを入力して，DMAを実行する
- ディスクリプタ: DMA転送に必要となる構造体
  - 送信元，送信先，データ長，ディスクリプタの識別子を含む
    - ✓例えば，送信元に paddr (gpu_memadr), 送信先に FPGAの外部メモリアドレスをセットすることで FPGA ← GPU の DMA転送を実行する

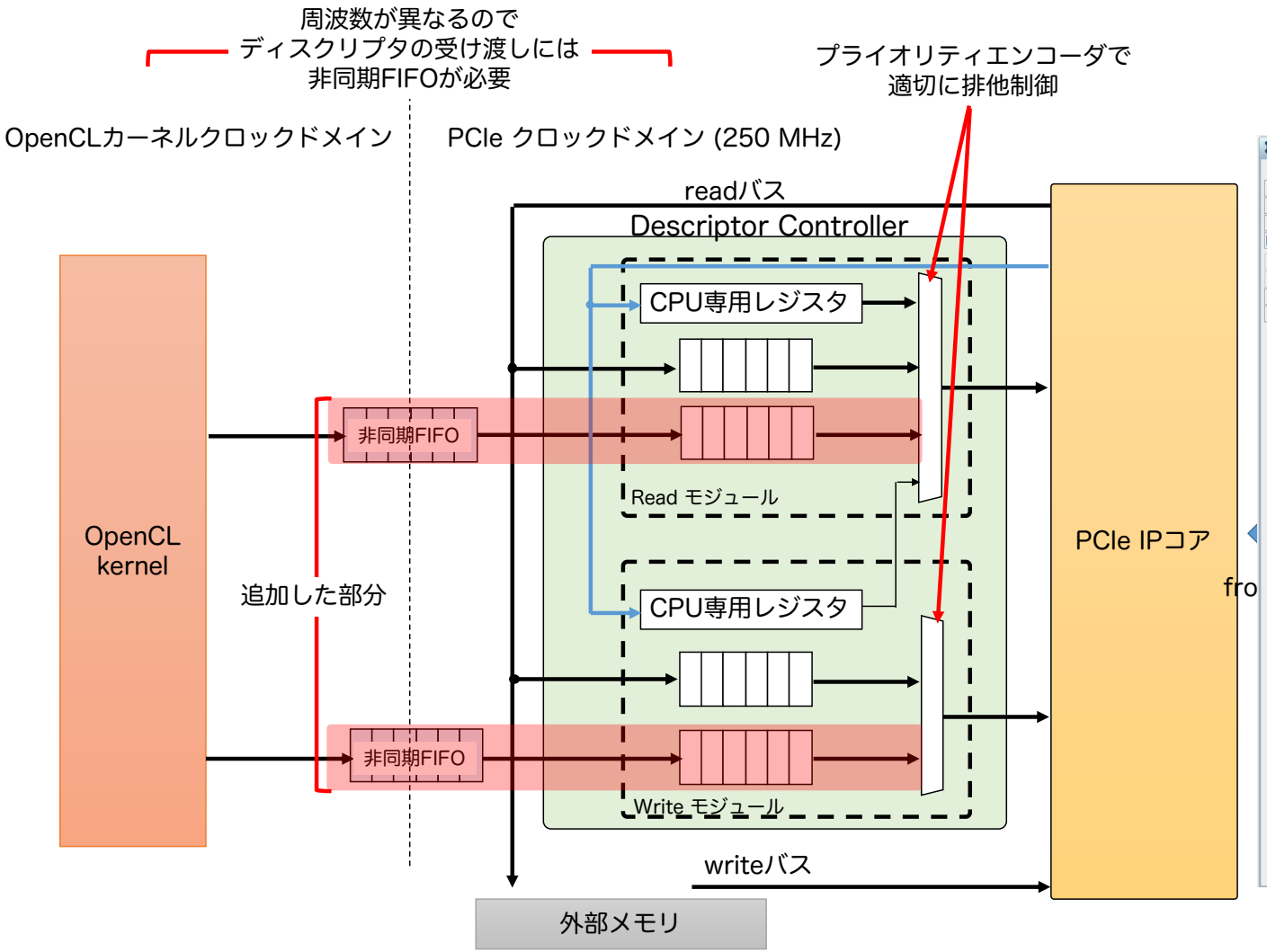
Kernel code

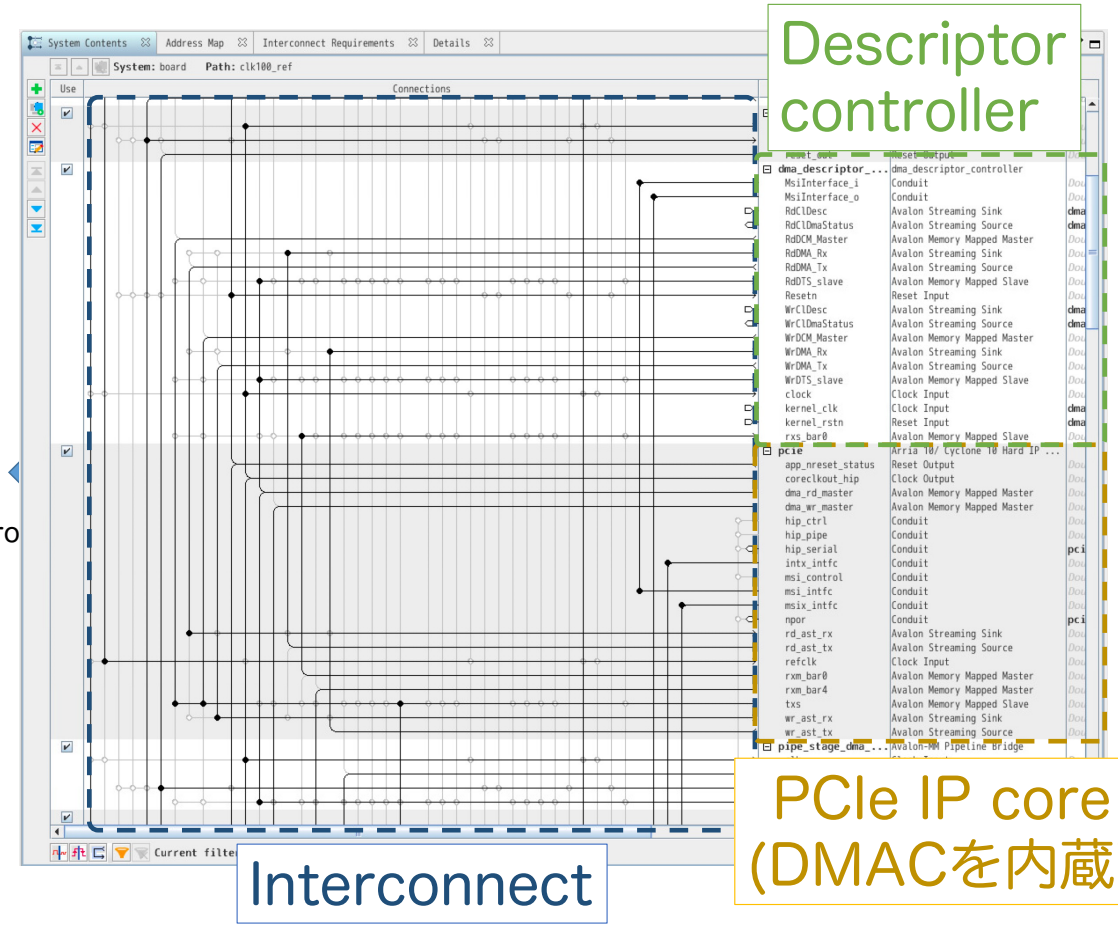
```
__kernel void fpga_dma(__global float *restrict fpga_mem,
                       const ulong gpu_memadr,
                       const uint id_and_len)
{
  cldesc_t desc;
  // DMA transfer GPU -> FPGA
  desc.src = gpu_memadr;
  desc.dst = (ulong)(&fpga_mem[0]);
  desc.id_and_len = id_and_len;
  write_channel_intel(fpga_dma, desc);
  ulong status = read_channel_intel(dma_stat);
}
```

GPU-to-FPGA DMA kick



実装をチョット見せる

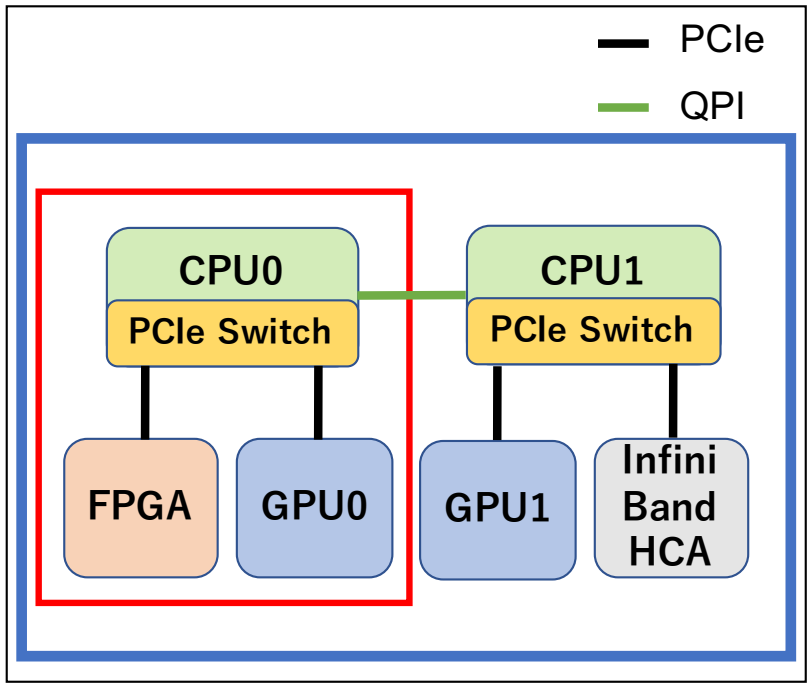




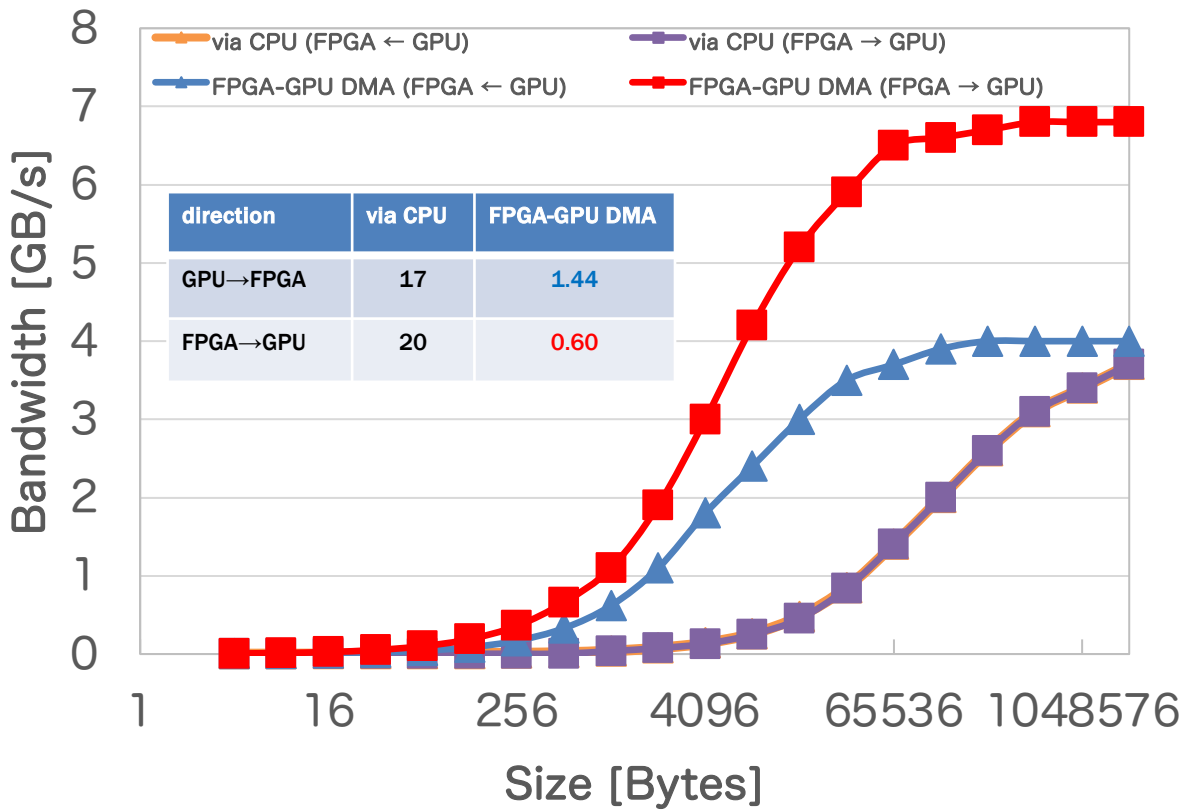
ディスクリプタコントローラのハードウェアブロック図　Qsys ツールを使ってハードウェアをポチポチ接続

通信レイテンシ・バンド幅を評価

- PPXを利用
  - GPU: NVIDIA P100 GPU
  - FPGA: Intel Arria10 FPGA

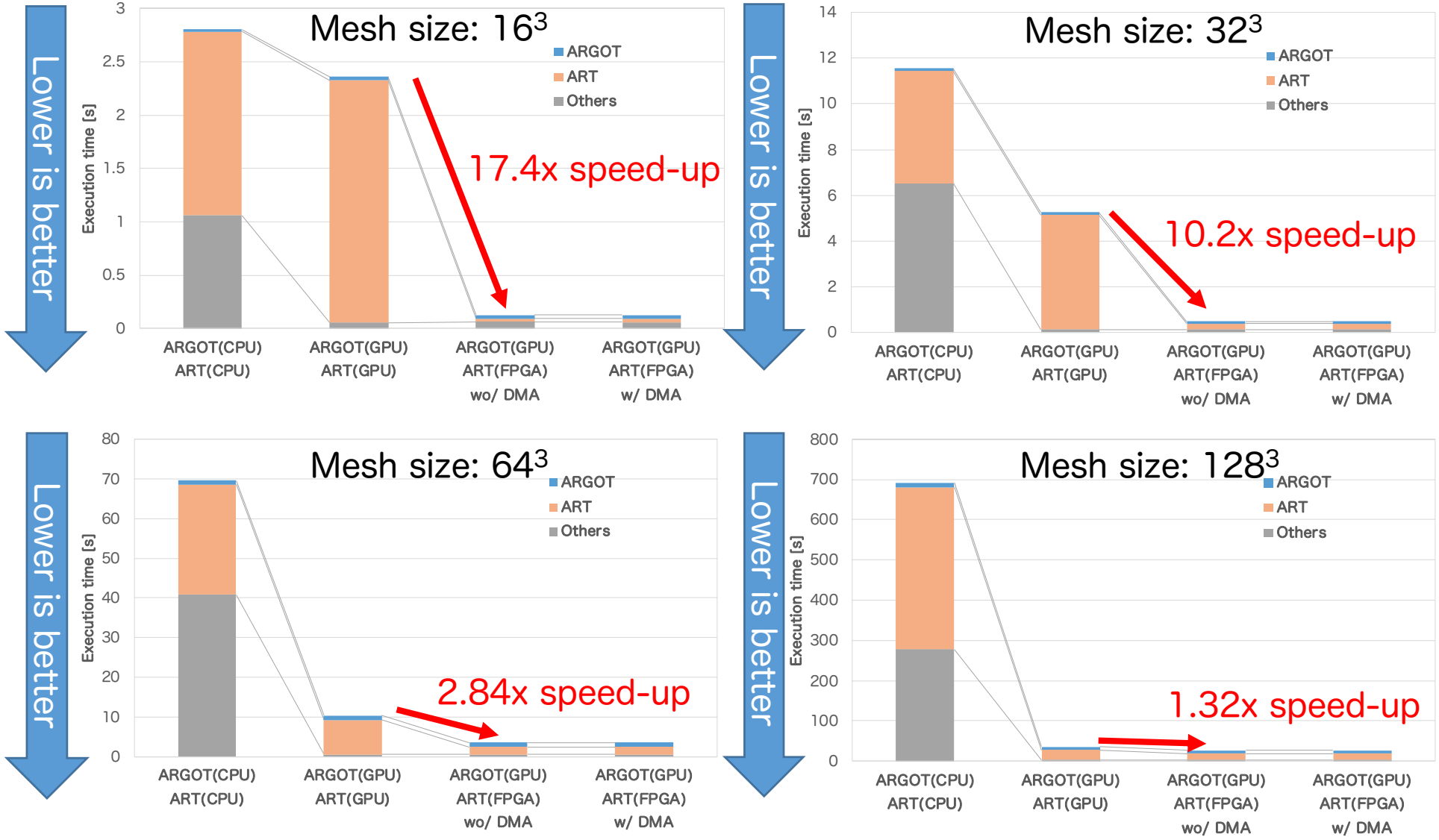


A computation node of PPX



direction	via CPU	FPGA-GPU DMA
GPU→FPGA	17	1.44
FPGA→GPU	20	0.60

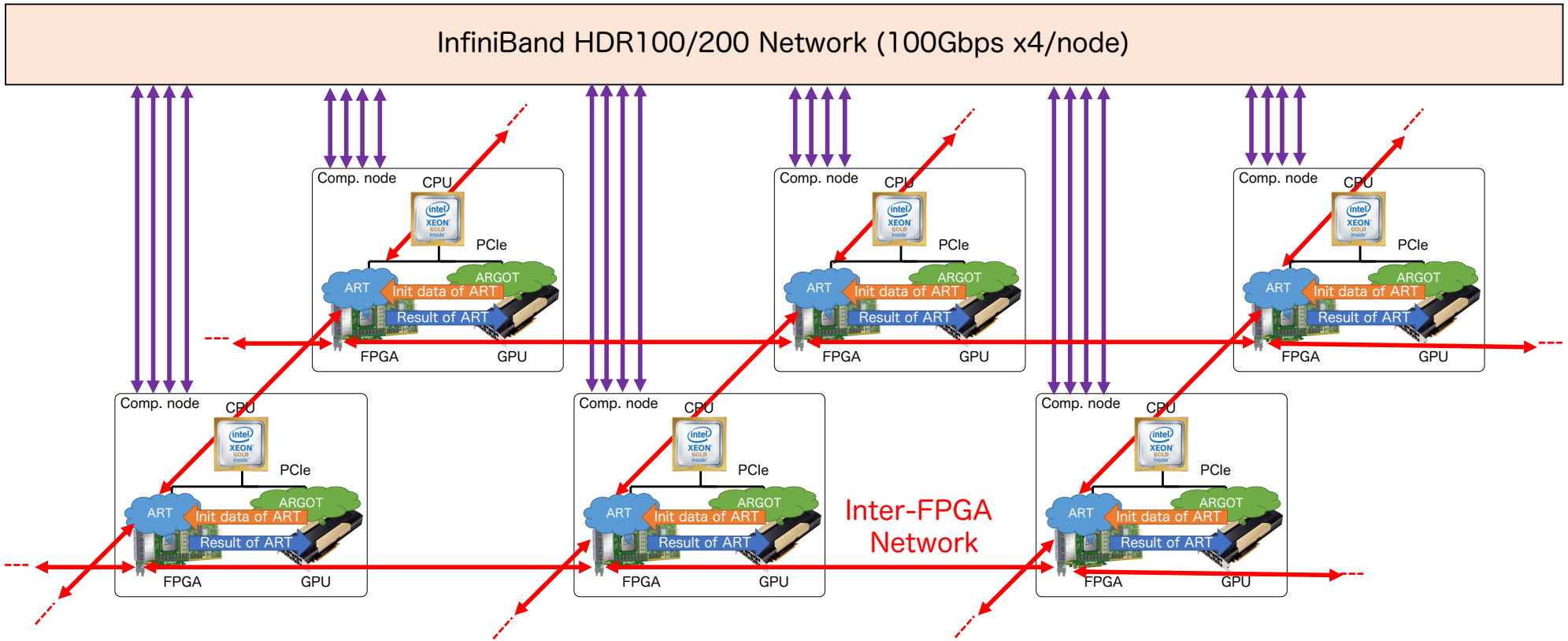
GPU・FPGA連携ARGOTコード with / without DMAの性能比較



GPU-FPGA間通信処理はARGOTコード全体の1%程度であったため，GPU-FPGA間DMAの付与は性能向上に寄与しなかったが，開発したDMA機構が実アプリで正しく動作することを実証した

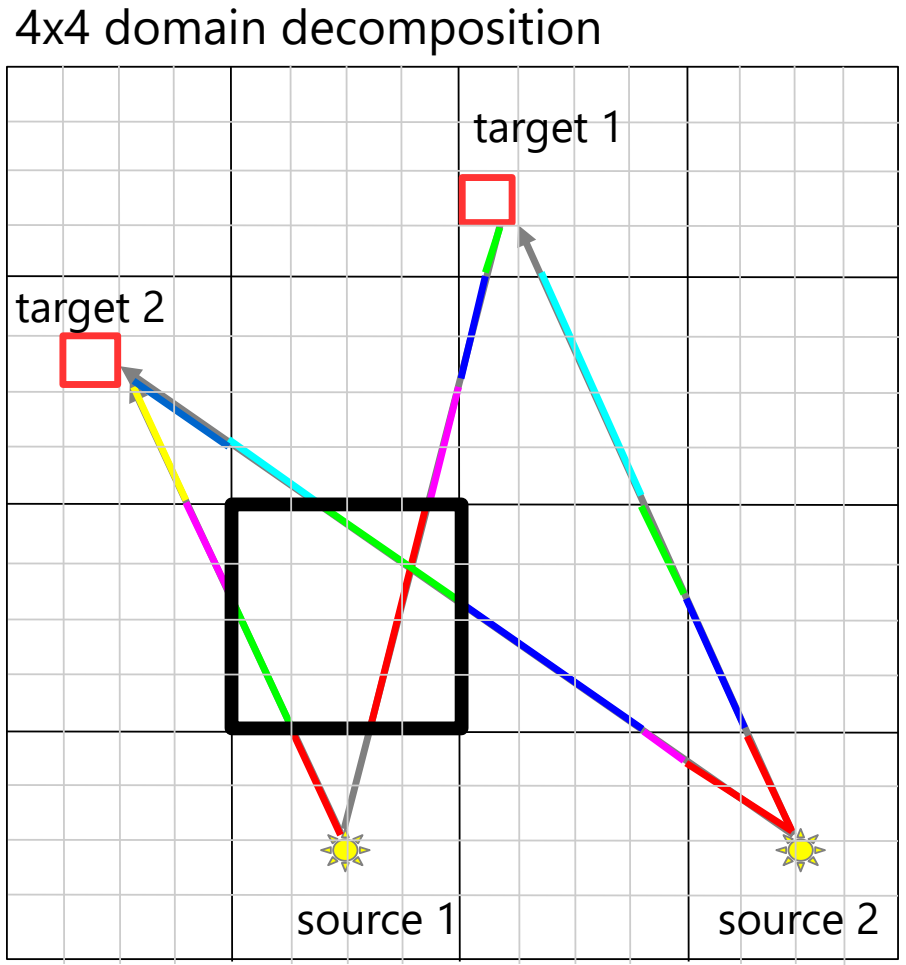
GPU・FPGA連携ARGOTコードの多ノード並列化

- 1ノードにつき 1 MPI プロセスをアサインし，GPUとFPGAを制御
  - ARGOT法は複数のGPUで並列化
    - ✓ CUDA + MPIプログラミングで実現 (ARGOTコードに元々実装済み)
  - ART法を複数のFPGAで並列化
    - ✓ ART法に必要なデータはFPGA間ネットワークで送受信する (OpenCLで制御可能)



ARGOT法のノード並列化

- オリジナルのARGOTコードに元々実装済み [4]
  - シミュレーション空間 (3次元) を各次元に均等に分割
    - ✓1 GPU / プロセスの割り当て
  - 複数のノードにまたがる光線はノード間の境界で「レイセグメント」分割される
    - ✓各ノードが担当するセグメント同士は互いに独立
      → CUDAスレッドに割り当てられ並列処理
    - ✓各ノードで各セグメントの光学的厚みを計算する
  - 最後に各セグメントの光学的厚みの計算結果の和を求める

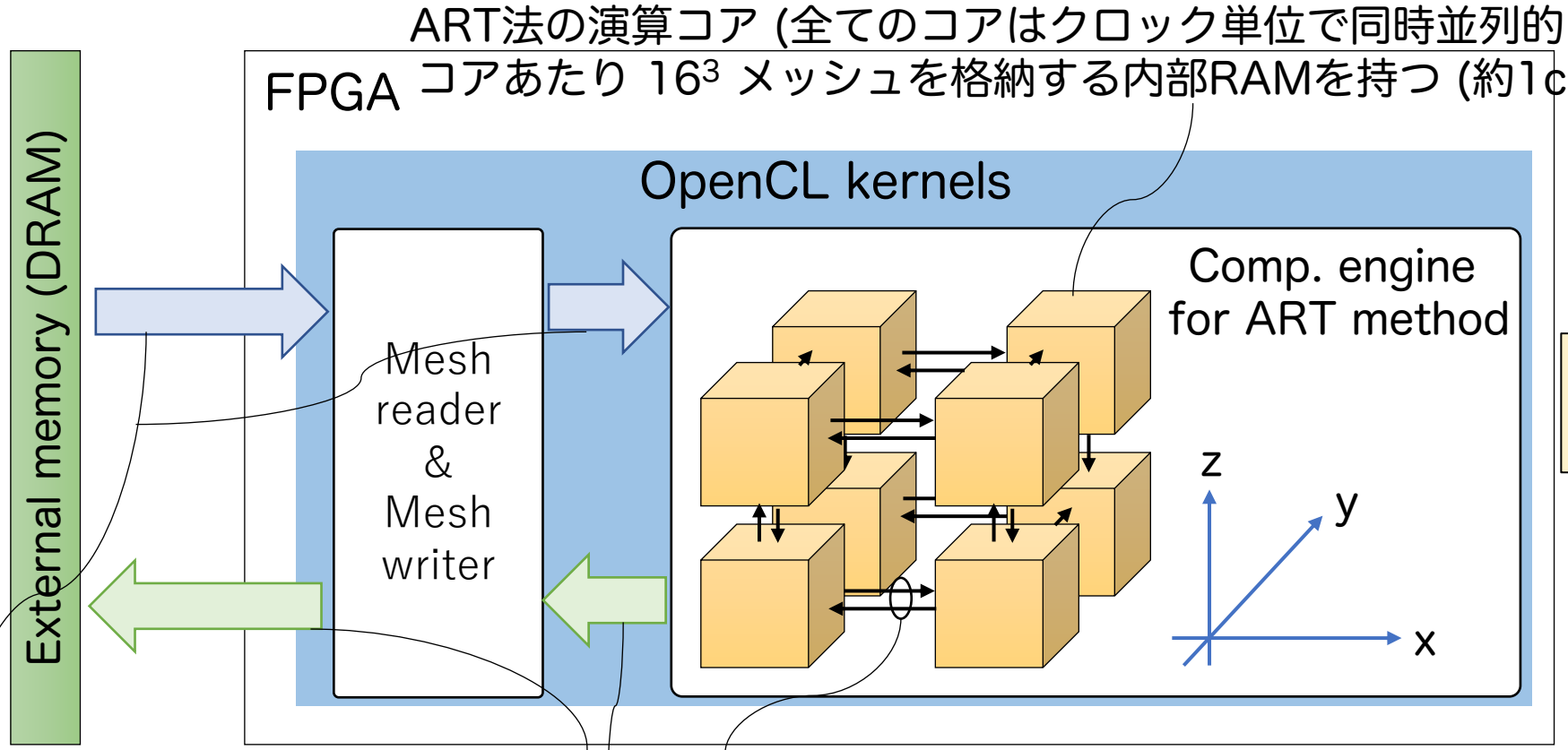


4 × 4 (16ノード) の領域分割
光線がノードをまたぐ度に，色分けされている
(セグメント分割されている)

[4] Okamoto, T., Yoshikawa, K. and Umemura, M.: argot: accelerated radiative transfer on grids using octtree, Monthly Notices of the Royal Astronomical Society, Vol. 419, No. 4, pp. 2855–2866 (online), DOI:10.1111/j.1365-2966.2011.19927.x (2012).

ART法のFPGA実装 (再掲)

- コア間をIntel Channelで接続し，レイデータを転送して計算を実行



時間並列性 (パイプライン並列) および空間並列性を活用して ART 法の実効性能を最大化

→ Comm. path of ray data (Intel Channelを使用)

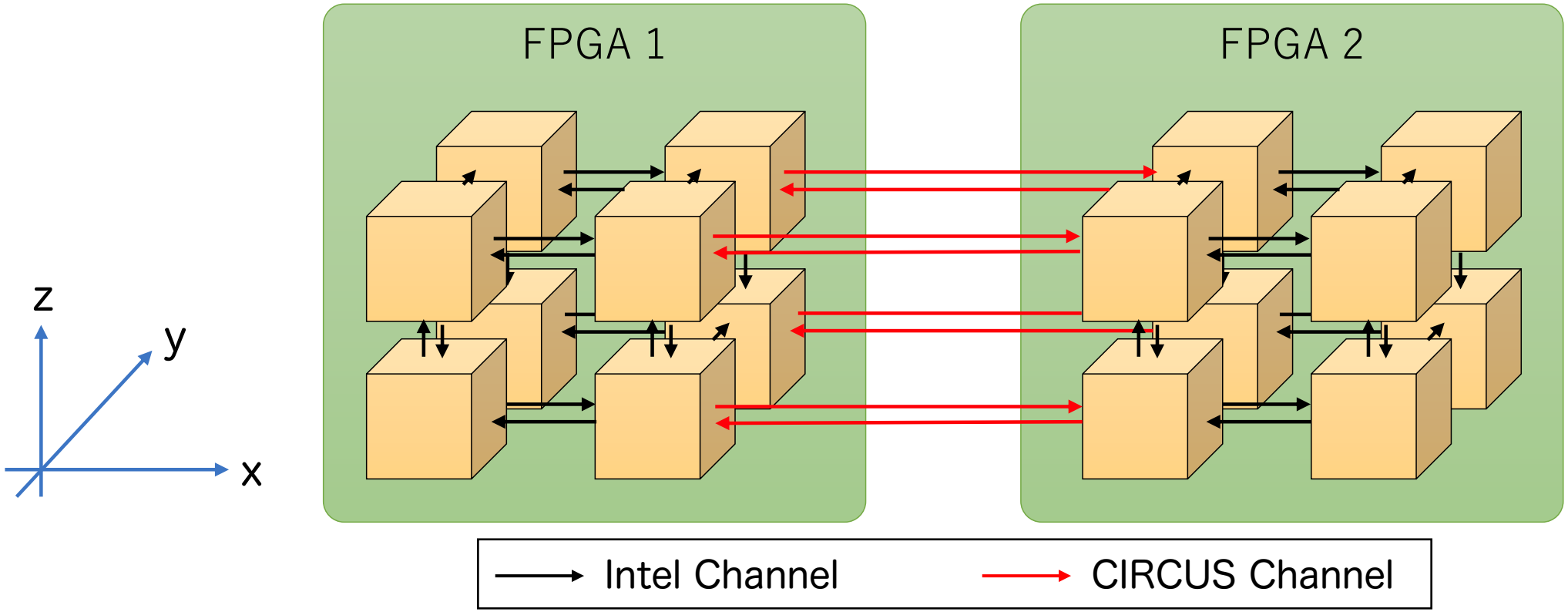
load mesh data into comp. engine

store comp. result (mesh data) in DRAM

マルチFPGAを用いたART法の並列化の概要

[2] 藤田ら, “OpenCL プログラミングを用いた並列 FPGA 処理 システムの性能評価”, ACS70, Vol. 13, No. 3, pp. 13–28

- コア間のIntel Channel接続をCIRCUSを用いて異なるFPGA間に拡張する
  - 複数のFPGAを用いて，ART演算コアのクラスタを構築するイメージ
- CIRCUS: OpenCLから制御可能なFPGA間通信技術
  - Communication Integrated Reconfigurable CompUting System [2]
  - 演算パイプラインから直接データを送受信する (演算と通信の融合)



CIRCUSの通信モデル

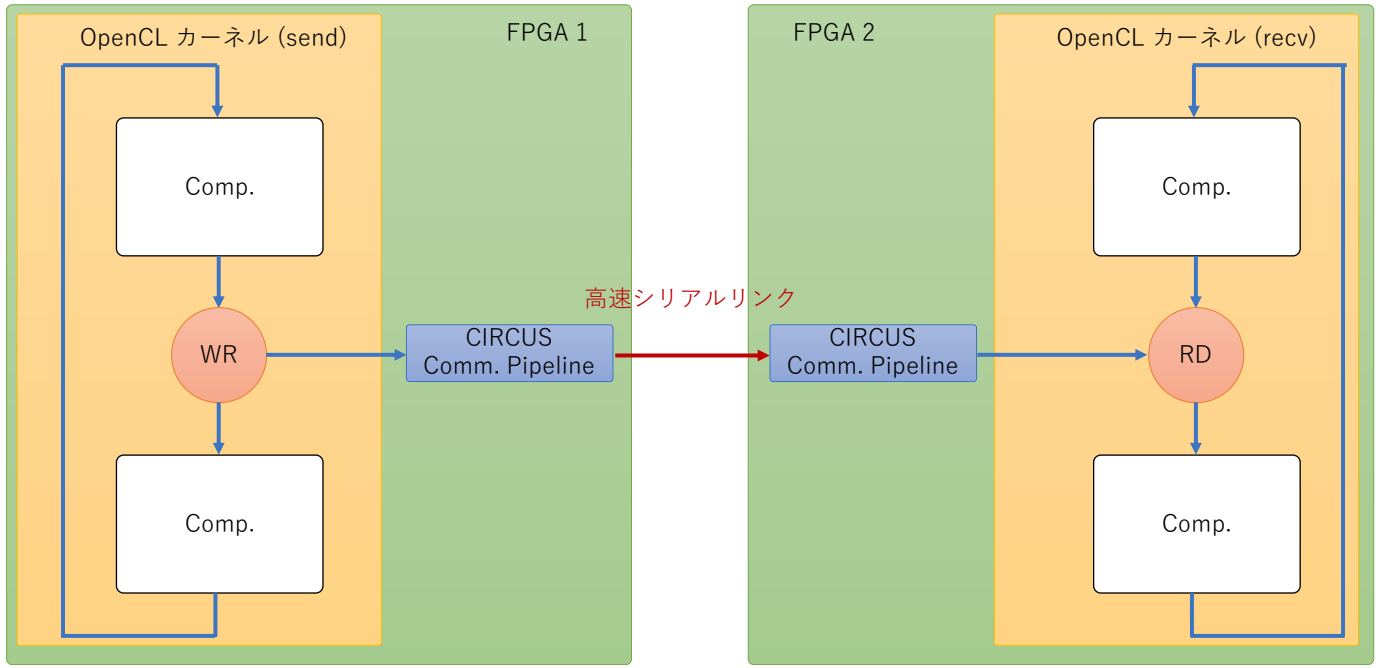
- パイプライン通信を前提とする
  - 全ての論理がハードウェアとして並列に動作するFPGAならではの強みを活かす
  - HPCで一般的な通信手法は Socket or MPI だが FPGAには不向き

sender code on FPGA1

```
sender(__global float* restrict x, int n) {
  for (int i = 0; i < n; i++) {
    float v = x[i];
    write_channel_intel(simple_out, v);
  }
}
```

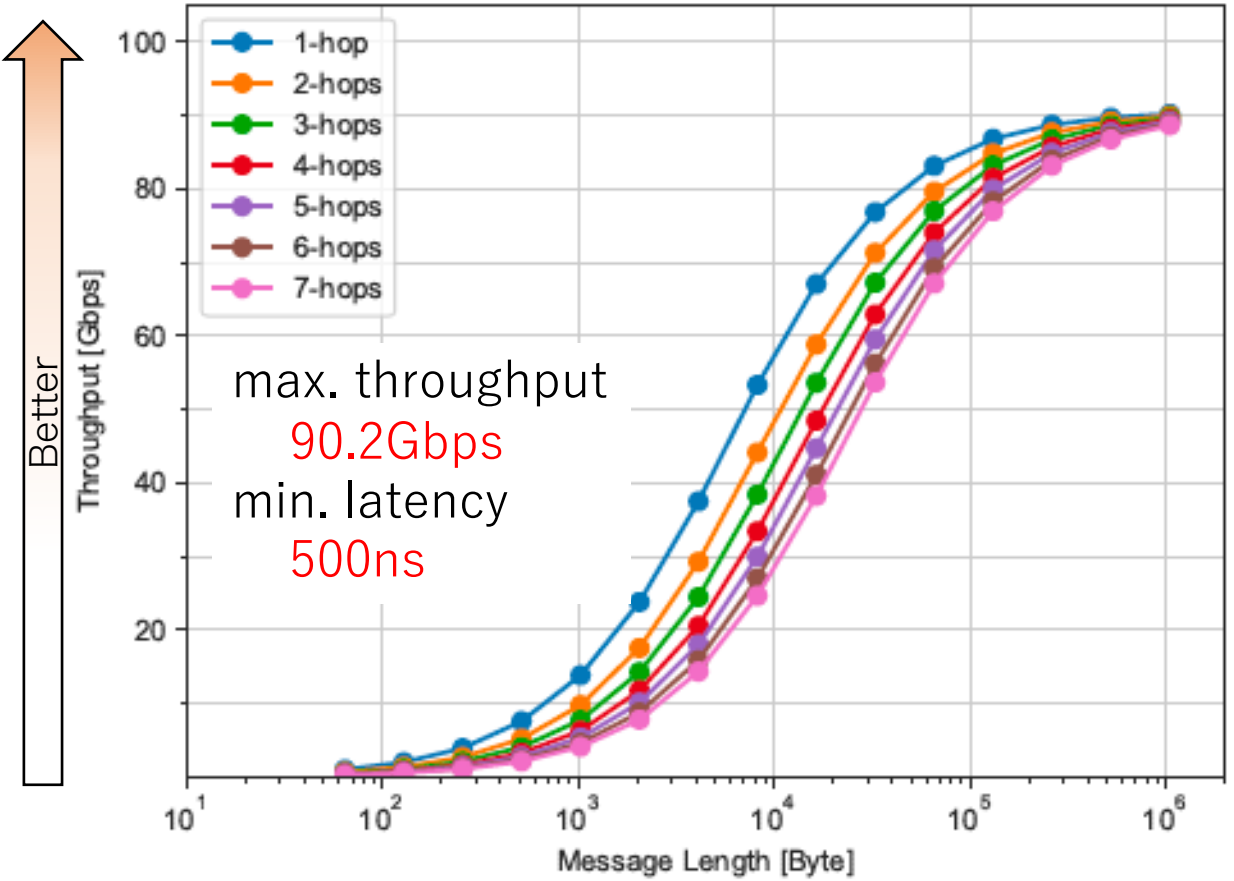
receiver code on FPGA2

```
receiver(__global float* restrict x, int n) {
  for (int i = 0; i < n; i++) {
    float v = read_channel_intel(simple_in);
    x[i] = v;
  }
}
```

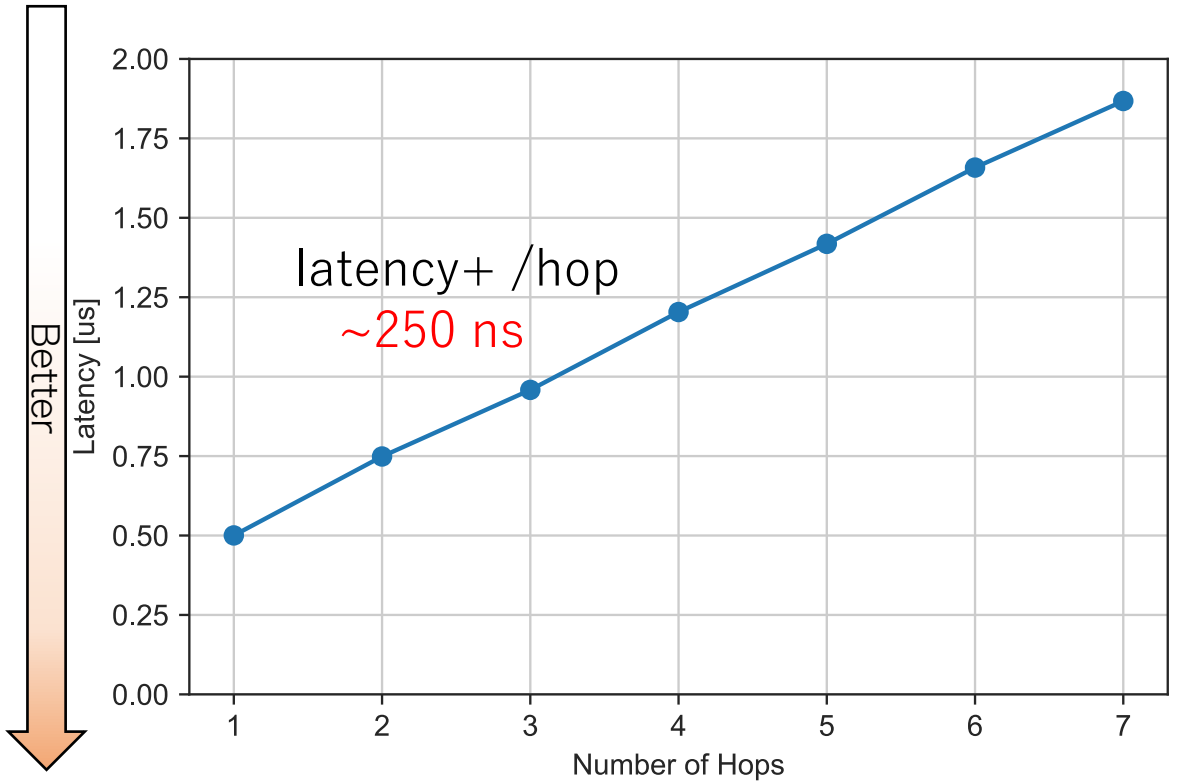


CIRCUSが構築するパイプライン構造

CIRCUSの通信性能 (pingpong ベンチマークで評価)



Throughput（1hop~7hops）



Latency（1hop~7hops）

Evaluated on up to 8 Bittware 520N FPGA boards in Cygnus supercomputer at CCS, University of Tsukuba [14]

[14]: N. Fujita, et al., “Performance Evaluation of Pipelined Communication Combined with Computation in OpenCL Programming on FPGA”, AsHES2020.

ART+CIRCUS実装の擬似コード

```
#pragma OPENCL EXTENSION cl_intel_channels : enable

// CIRCUS カーネルをインクルード
#include <circus.h>

channel rt_ch[6][NPE_X][NPE_Y][NPE_Z];

// CIRCUS Channel
channel extout_x_neg_x0_y0_z0;
channel extin_x_neg_x0_y0_z0;

// PE が保持するメッシュ空間
#define PE_MESH_SIZE (16 * 16 * 16)

// 1 つの面に 1 入力 / 1 出力 あるので、12 本のチャネルが必要
#define PE_INPUT_X_POS rt_ch[1][1][0][0]
#define PE_INPUT_X_NEG extin_x_neg_x0_y0_z0
#define PE_INPUT_Y_POS rt_ch[3][0][1][0]
#define PE_INPUT_Y_NEG rt_ch[2][0][1][0]
#define PE_INPUT_Z_POS rt_ch[5][0][0][1]
#define PE_INPUT_Z_NEG rt_ch[4][0][0][1]
#define PE_OUTPUT_X_POS rt_ch[0][0][0][0]
#define PE_OUTPUT_X_NEG extout_x_neg_x0_y0_z0
#define PE_OUTPUT_Y_POS rt_ch[2][0][0][0]
#define PE_OUTPUT_Y_NEG rt_ch[3][0][0][0]
#define PE_OUTPUT_Z_POS rt_ch[4][0][0][0]
#define PE_OUTPUT_Z_NEG rt_ch[5][0][0][0]

__kernel void PE_x0_y0_z0(...) {
  // メッシュ空間のデータ (光学的厚みと source function) を格納する配列に Block RAM を利用する
  __local struct radiation_mesh rmesh[PE_MESH_SIZE];

  // メッシュにおける光学的厚み、メッシュの source function を格納
  set_radiation_mesh(rmesh);

  // 768 x N^2 の全てのレイに対してレイトレーシングを実行 (N = 16)
  bool exit = false;
  while (!exit) {
    bool x_neg;
    bool x_pos;
    bool y_neg;
    bool y_pos;
    bool z_neg;
    bool z_pos;

    // レイの入力判定
    INPUT_COND(x_neg);
    INPUT_COND(x_pos);
    INPUT_COND(y_neg);
    INPUT_COND(y_pos);
    INPUT_COND(z_neg);
    INPUT_COND(z_pos);

    // 隣接 PE から入力がある場合、レイデータを受信する
    if      (x_neg) ray = read_channel_intel(PE_INPUT_X_NEG);
    else if (x_pos) ray = read_channel_intel(PE_INPUT_X_POS);
    else if (y_neg) ray = read_channel_intel(PE_INPUT_Y_NEG);
    else if (y_pos) ray = read_channel_intel(PE_INPUT_Y_POS);
    else if (z_neg) ray = read_channel_intel(PE_INPUT_Z_NEG);
    else if (z_pos) ray = read_channel_intel(PE_INPUT_Z_POS);

    // ART 法の演算カーネルを実行 (輻射強度の計算)
    calc_intensity(&ray, rmesh);

    bool x_neg_out;
    bool x_pos_out;
    bool y_neg_out;
    bool y_pos_out;
    bool z_neg_out;
    bool z_pos_out;

    // レイの出力判定
    OUTPUT_COND(x_neg_out);
    OUTPUT_COND(x_pos_out);
    OUTPUT_COND(y_neg_out);
    OUTPUT_COND(y_pos_out);
    OUTPUT_COND(z_neg_out);
    OUTPUT_COND(z_pos_out);

    // レイの次の計算領域が隣接 PE の保持するメッシュ空間である場合、レイデータを送信する
    if      (x_neg_out) write_channel_intel(PE_OUTPUT_X_NEG, ray);
    else if (x_pos_out) write_channel_intel(PE_OUTPUT_X_POS, ray);
    else if (y_neg_out) write_channel_intel(PE_OUTPUT_Y_NEG, ray);
    else if (y_pos_out) write_channel_intel(PE_OUTPUT_Y_POS, ray);
    else if (z_neg_out) write_channel_intel(PE_OUTPUT_Z_NEG, ray);
    else if (z_pos_out) write_channel_intel(PE_OUTPUT_Z_POS, ray);

    // レイトレーシングの終了判定
    CHECK_TERMINATION(exit);
  }
}
```

Intel Channel, CIRCUS Channel を宣言

通信経路をマクロで分かりやすく

1コア分のOpenCLカーネル関数

メッシュデータを格納するワーキングメモリを宣言

ワーキングメモリの初期化

ART法の演算ループ
(この部分がパイプラインHWとして実装される)

隣接コアからレイデータが来るか判定

レイデータの受信
(データが来る場合)

輻射強度の計算
(ART法で一番大事なところ)

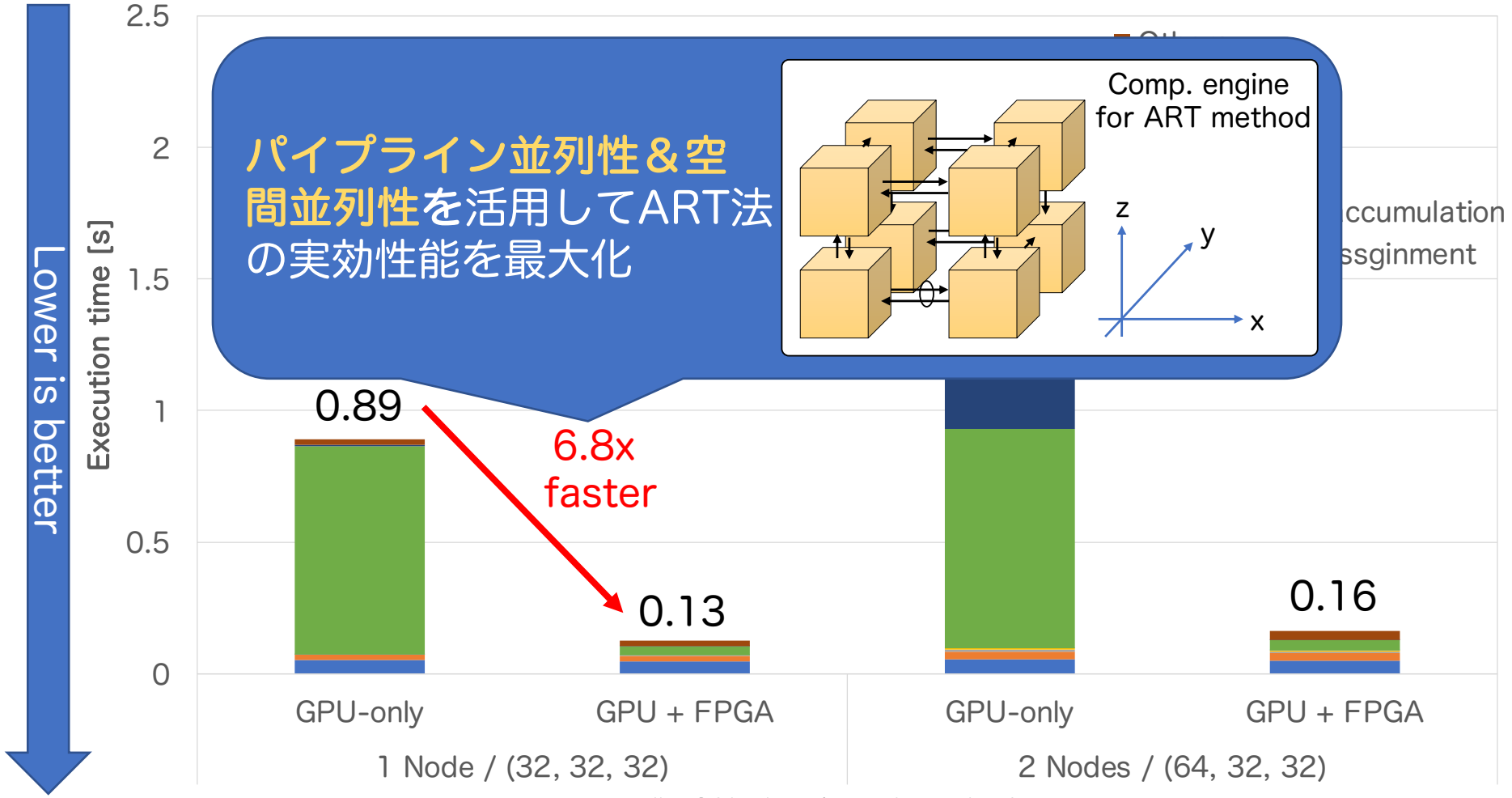
隣接コアにレイデータが行くか判定

レイデータの送信
(データが行く場合)

ART法の終了判定

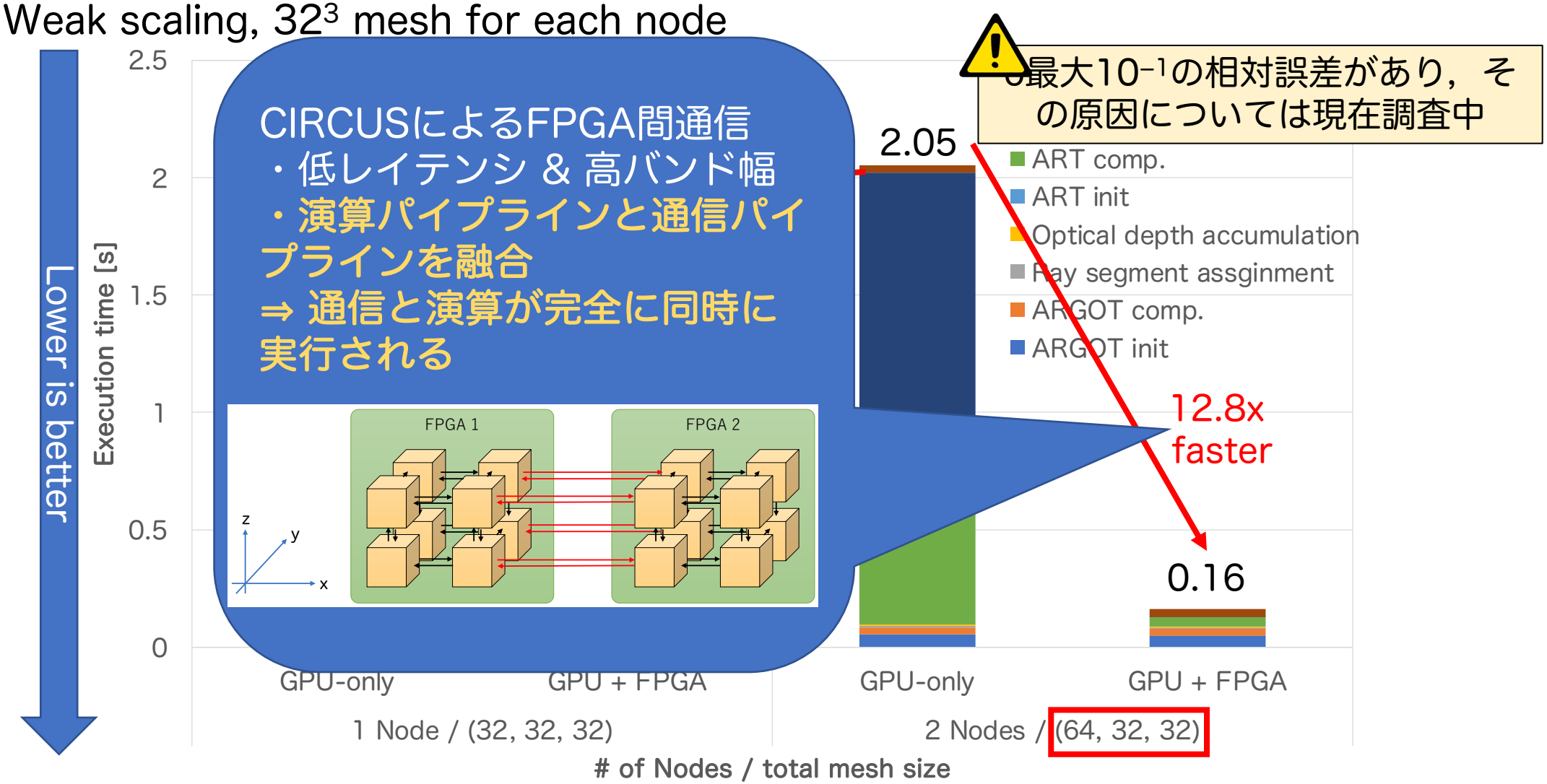
Performance evaluation of GPU–FPGA-accelerated ARGOT code on multiple nodes

- ARGOT with 2 nodes (2 GPUs/IB + 2 FPGAs/CIRCUS)
  - Weak scaling, 32^3 mesh for each node



Performance evaluation of GPU–FPGA-accelerated ARGOT code on multiple nodes

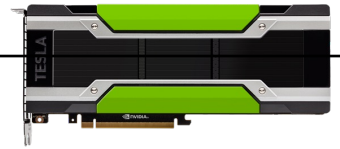
- ARGOT with 2 nodes (2 GPUs/IB + 2 FPGAs/CIRCUS)
  - Weak scaling, 32^3 mesh for each node



最大10^{-1}の相対誤差があり，その原因については現在調査中

What has contributed to the improved performance?

●1 node performance

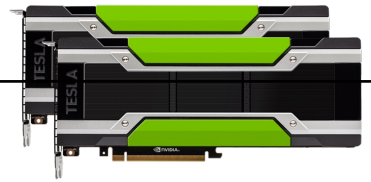


- Random memory access...
- Short vector length...

6.8x faster

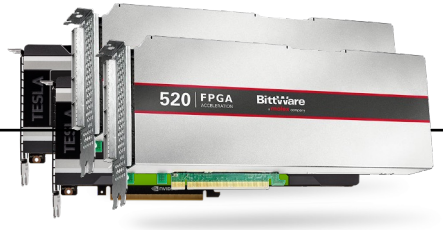
- BRAM-based memory logic
- Pipeline and spatial parallelism

●2 nodes performance



- Random memory access...
- Short vector length...
- large overhead by small chunks of multiple data copy...

12.8x faster

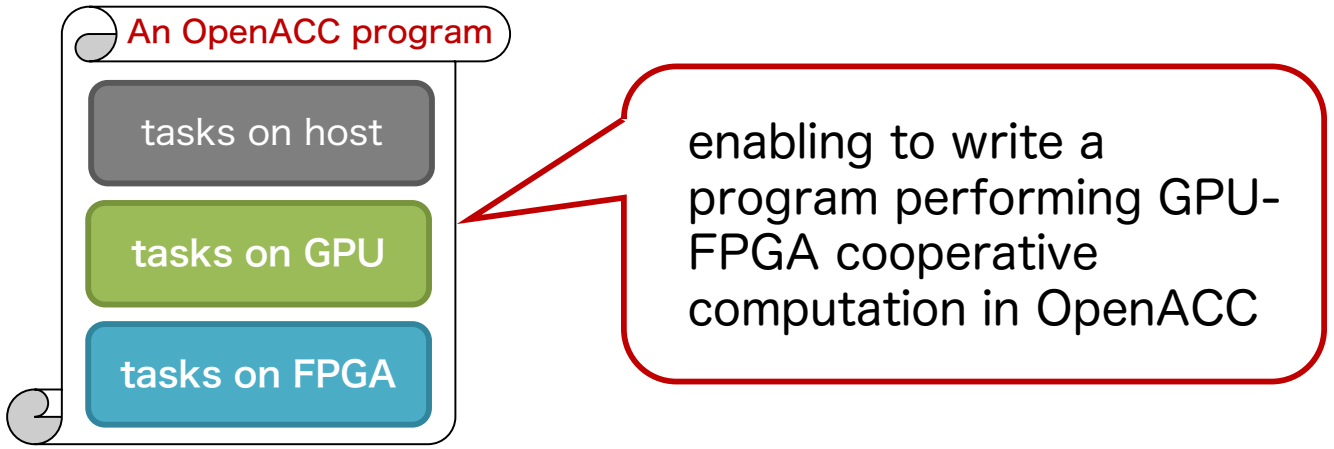


- BRAM-based memory logic
- Pipeline and spatial parallelism
- integrating comp. and comm. pipelines (comp. and comm. work completely simultaneously)

GPU・FPGA連携のためのプログラミング

- これまでの実装方式
  - CUDA + OpenCL の混合プログラミング
    - ✓プログラマに多大な負担を強いる

  → よりユーザビリティの高いプログラミング環境が必要
- 提案手法: OpenACC programming on multi-hetero devices
  - GPU-FPGA混載クラスタシステムにおける両デバイスをOpenACCによる統一的記述によって利用可能にする
  - 米国ORNLとの共同研究成果
    - ✓ORNLが開発しているFPGA用OpenACCコンパイラを利用

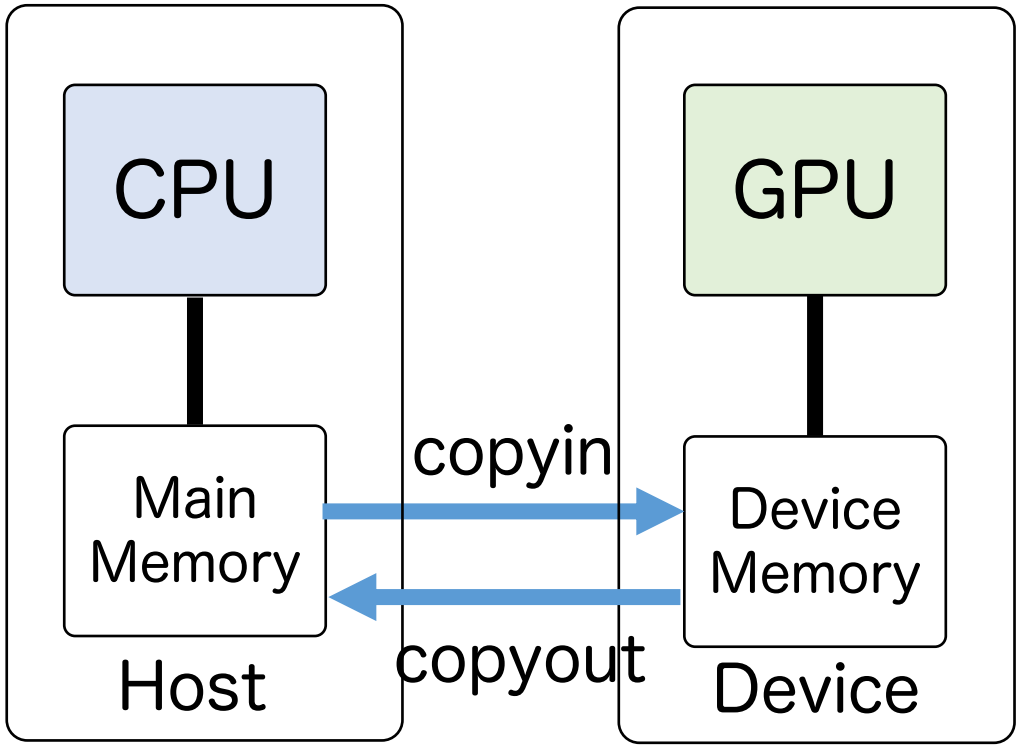


OpenACCとは？

●コンパイラ指示文（pragma）を書き添える形のGPUプログラミング

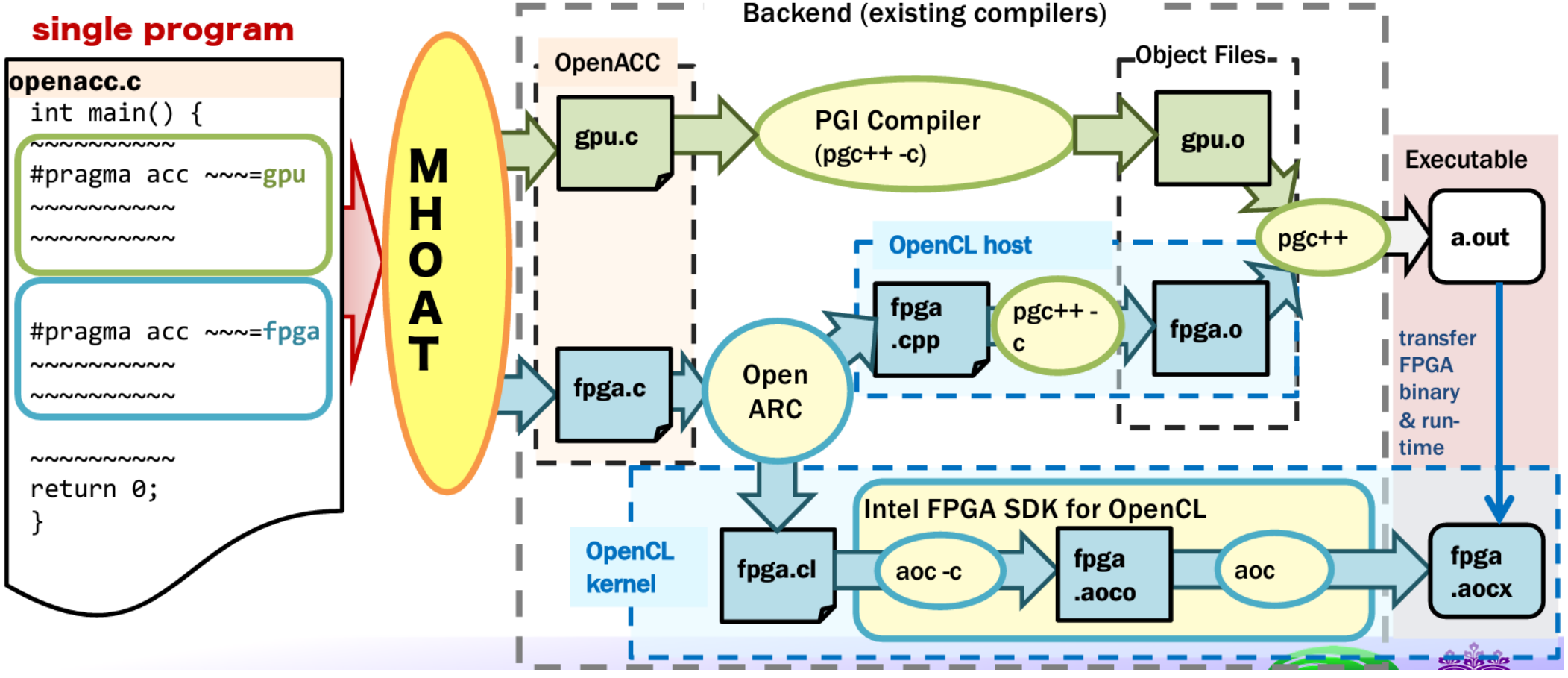
```c
int main(void) {

  // ~~~ CPU Processing ~~~

  // Offloading the following to an accelerator
  #pragma acc data copyout(c[:N]) copyin(a[:N], b[:N])
  {
    #pragma acc parallel loop
    for (i = 0; i < N; i++) {
      c[i] = a[i] + b[i];
    }
  }

  // ~~~ CPU Processing ~~~

  return 0;
}
```

サンプル C コード vecadd.c



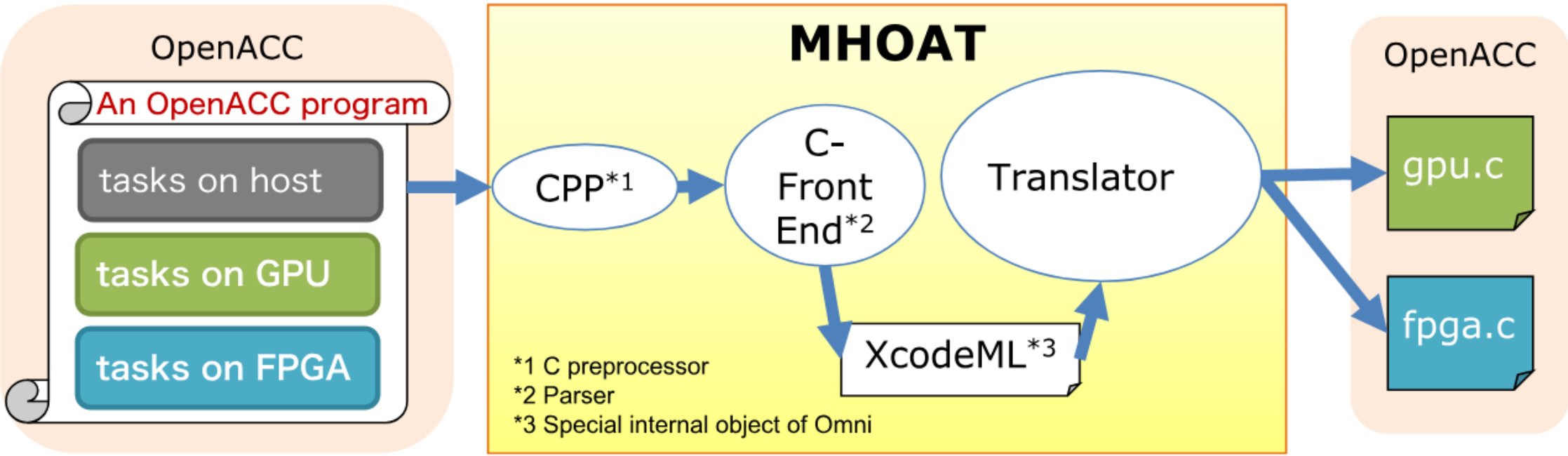
MHOAT: Multi-Hetero OpenACC Translator

- GPUとFPGAをOpenACCの枠組みで利用できるようにする
  - ORNLが開発したOpenARCはバックエンドコンパイラとして利用する
  - R. Tsunashima, et al., “OpenACC unified programming environment for GPU and FPGA multi-hybrid acceleration”, HLPP2020, Jul. 2020.



MHOAT の実装の概要

●理化学研究所計算科学研究センターと筑波大学計算科学研究センターが共同開発している Omni コンパイラ基盤*を用いて実装

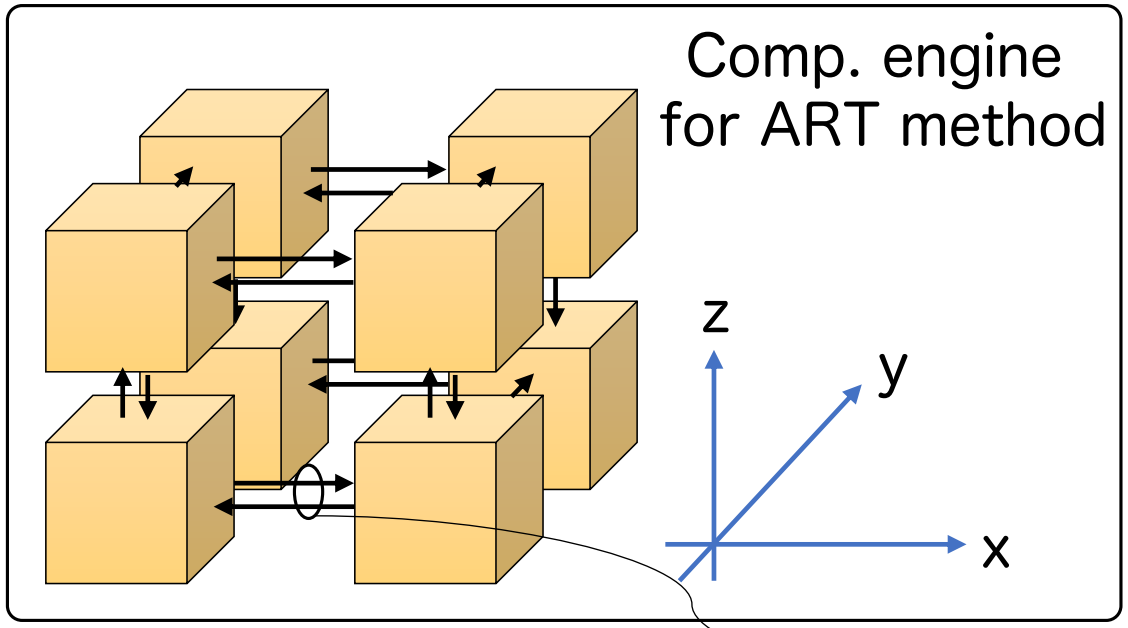


* Murai, H et al., Metaprogramming Framework for Existing HPC Languages Based on the Omni Compiler Infrastructure, 2018 6th International Symposium on Computing and Networking Workshops (CANDARW), pp.250–256

OpenACCによるART法のFPGA実装

Institute of SCIENCE TOKYO

●オリジナルのOpenCL実装と等価なハードウェア構成を記述可能




```
1  void fpga(...) {
2  #pragma accomn target_dev(FPGA)
3      int channel1[1];
4      int channel2[1];
5  ...
6  #pragma acc data copy(a[:N], b[:N]) pipe(channel1[:1],
       channel2[:1])
7  {
8  #pragma acc serial pipein(channel1) pipeout(channel2)
       async(0)
9  for(i=0; i < N/2; i++)
10 ...
11 #pragma acc serial pipein(channel2) pipeout(channel1)
       async(1)
12 for(i=N/2; i < N; i++)
13 ...
14 #pragma acc wait
15 } // acc data end
16 ...
17 }
```

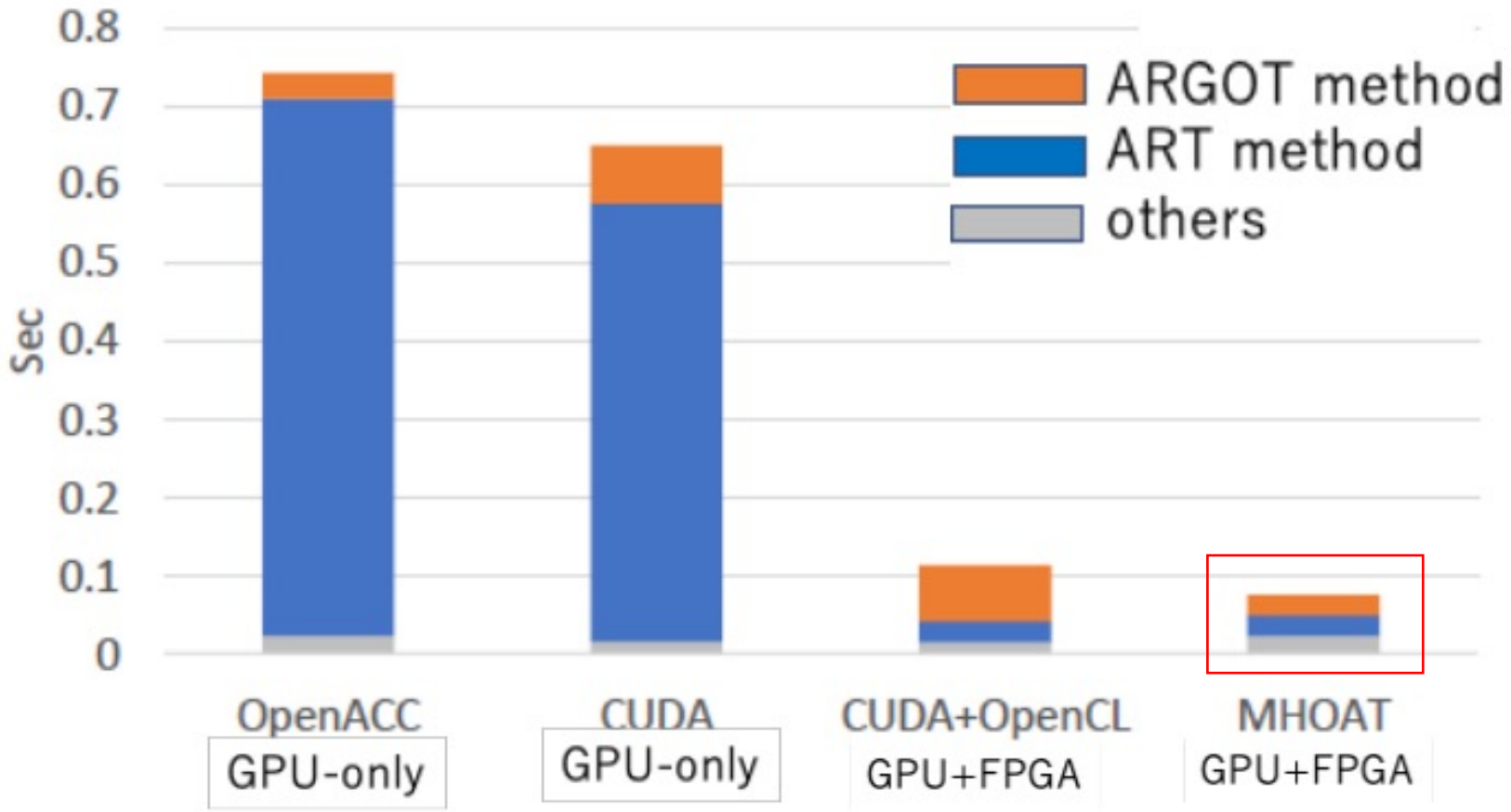
Processing element 0

Processing element 1

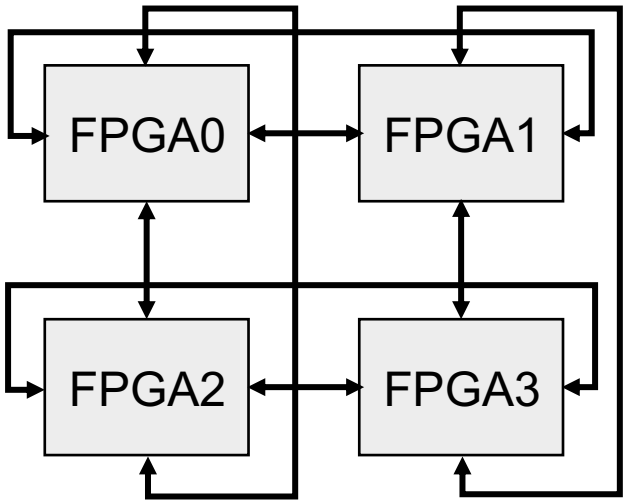
Code snippet of the FPGA impl. with OpenACC

4種のARGOTコード実装間での性能比較

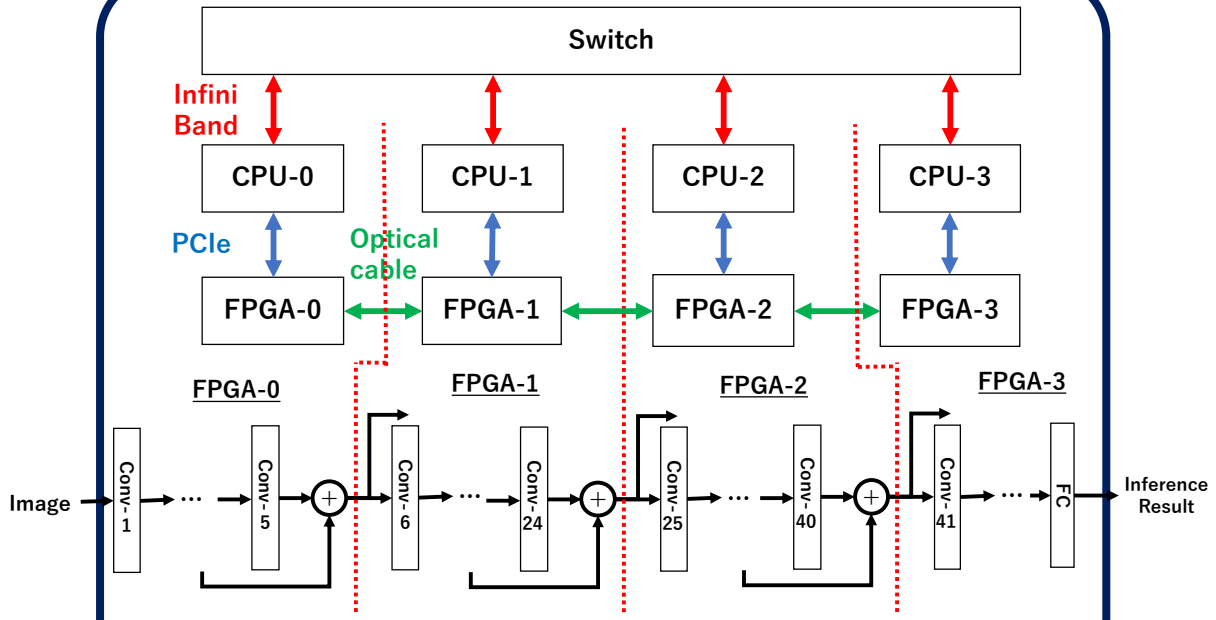
- 1GPU / 1 FPGAでの評価 [Taisuke Boku et al., H3 workshop (ISC'23)]
  - 問題サイズ：32^3



最近実施したFPGA-centricな研究



Making FPGA comm. more stable



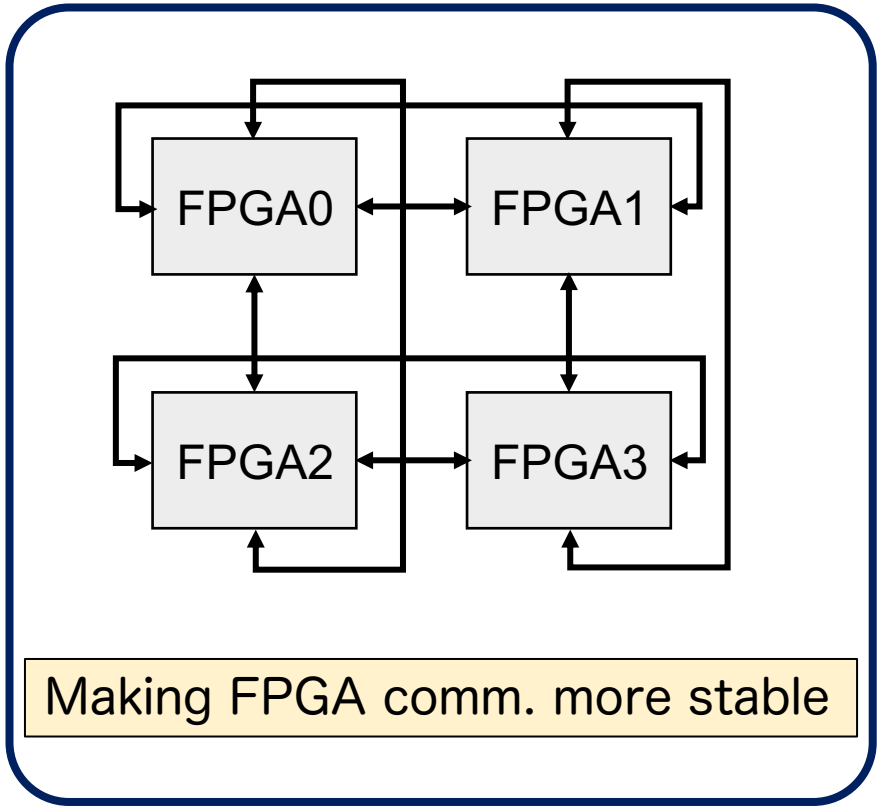
Making CNN run on multiple FPGAs

より安定したFPGA間通信フレームワークの開発

Institute of SCIENCE TOKYO

- CIRCUSの問題点：フロー制御に非対応
  - データ量が多いとバッファが溢れ、データが消失してしまう
  - 複雑な通信を含むプログラムの実装はほぼ不可能
    - ✓ 集団通信の実装等の障壁になる
- フロー制御を一から実装するのは大変

→フロー制御機能を備えた
通信コントローラ (Kyokko*) を利用



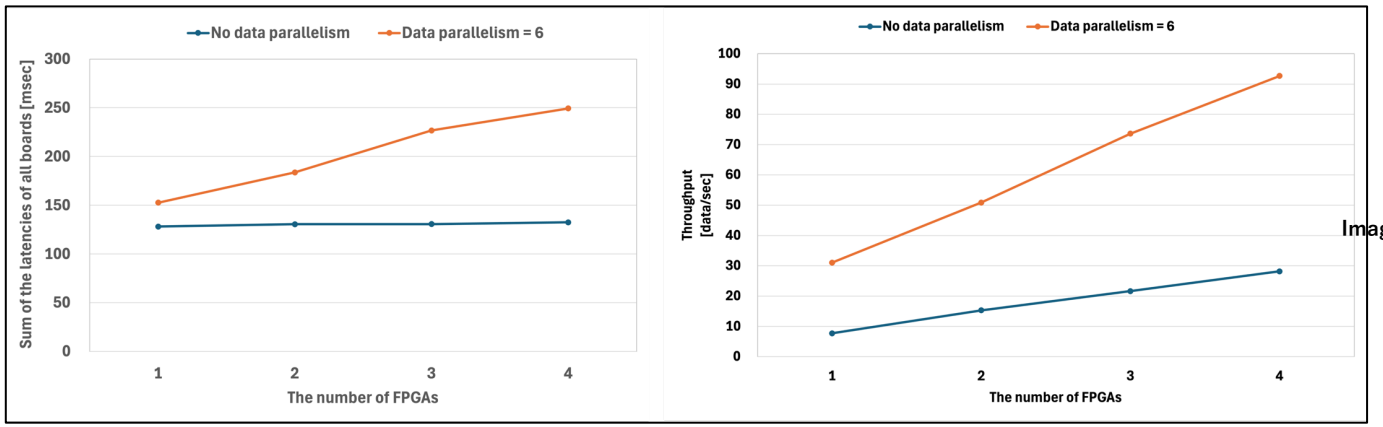
*Akinobu Tomori and Yasunori Osana., “Kyokko: a vendor-independent high-speed serial communication controller. “, HEART ’21

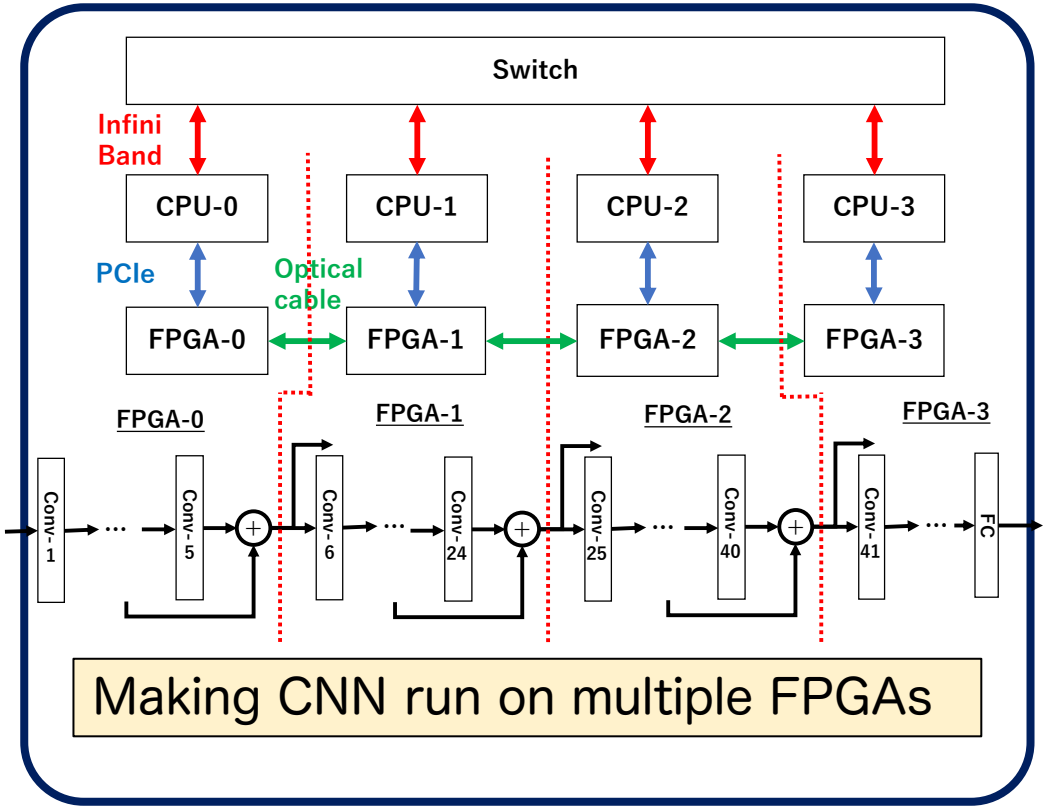
並列FPGAシステムによる深層学習推論処理の高速化

Institute of SCIENCE TOKYO

●CIRCUSフレームワークに基づくCNN推論ハードウェアアクセラレータ

- ユーザはOpenCLの抽象度に留まったまま，並列FPGAシステムのCNN加速を実現
- FPGAの台数を増やすほど，推論スループットは向上する
- レイテンシは，データ並列をしなければ殆ど一定 (データ並列時は通信チャネルの競合が発生)





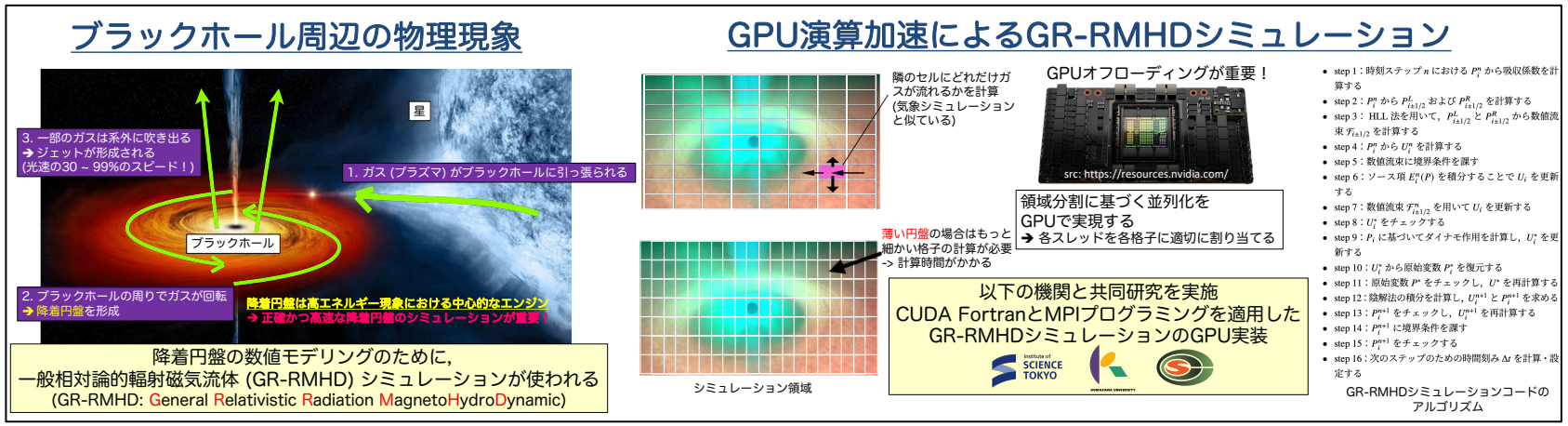
Takumi Suzuki et al., “Accelerating Deep Learning Inference with a Parallel FPGA System“, HEART’25, Pages 49 – 56, May 2025

FPGAを計算デバイスとして真に活用したいのなら

Institute of SCIENCE TOKYO

●私のスタンス：まずGPUやりましょう

●理由：だいたいの問題は並列化（GPUオフローディング）で速くなる



速くしたい何かがあるなら
まずGPUを試しましょう
（いきなりニッチなデバイス
つかうのやめましょう）

組み込み分野なら Jetson シリーズのGPUを試しましょう

現時点でCPUと比較して最大39倍の高速化を達成

以下のケースならFPGAつかうのありかもと思う領域

Institute of SCIENCE TOKYO

●電力要件がもの凄くシビアな場合

- GPUだと一瞬でバッテリーを食い尽くしかねないようなアプリケーション
- たとえば，ドローン制御とかでしょうか

●通信されるデータをリアルタイムかつ直に処理したい場合

- センサーで取得し，流れ込んでくるデータをフィルタ処理したいとか
- 通信と演算が密に結合した計算機システムを作らなければいけないのなら，FPGAは候補になりうる

それ以外の場合で，何かを速くするのにFPGAをつかってみよう，は正直お勧めしません.
速くするための手法はだいたい並列化に帰着しますし，そうなるとGPUの方が有利になることが多いです
もう一度言いますが，GPUが得意なこと (例：行列積) をFPGAにやらせてもまず間違いなく悲惨な目に遭います

まとめ

Institute of SCIENCE TOKYO

- スーパーコンピュータの電力効率の向上は喫緊の課題
  - その解として演算加速装置（アクセラレータ）の利活用が高性能計算分野の主流となりつつある
    - ✓ 現在最も多用されているアクセラレータはGPUだが，不規則な演算は苦手
    - ✓ FPGAを活用：GPUでは非効率となる演算を加速させるハードウェアを実現

→ 適材適所的に活用してアプリケーション全体の性能を向上 (CHARMコンセプト)

- これまで実施してきたGPU・FPGA連携のための研究を紹介
  - GPUとFPGAの併用による宇宙輻射輸送シミュレーションコード ARGOT の高速化
    - ✓ 2GPU / 2FPGA の性能：2GPUの性能と比べて 12.8 倍
  - 両デバイス間のDMAデータ転送技術の開発
    - ✓ CPUを介さないデータ転送をFPGAが自律的に実行する
    - ✓ OpenCLでDMA転送を記述可能
  - OpenACCの枠組みでのGPU・FPGA連携
    - ✓ CUDA + OpenCLベースの実装と比較して遜色ない性能
- 最近実施したFPGA-centricな研究
  - より安定したFPGA間通信フレームワークの開発 (フロー制御のサポート)
  - 並列FPGAシステムによる深層学習推論処理の高速化