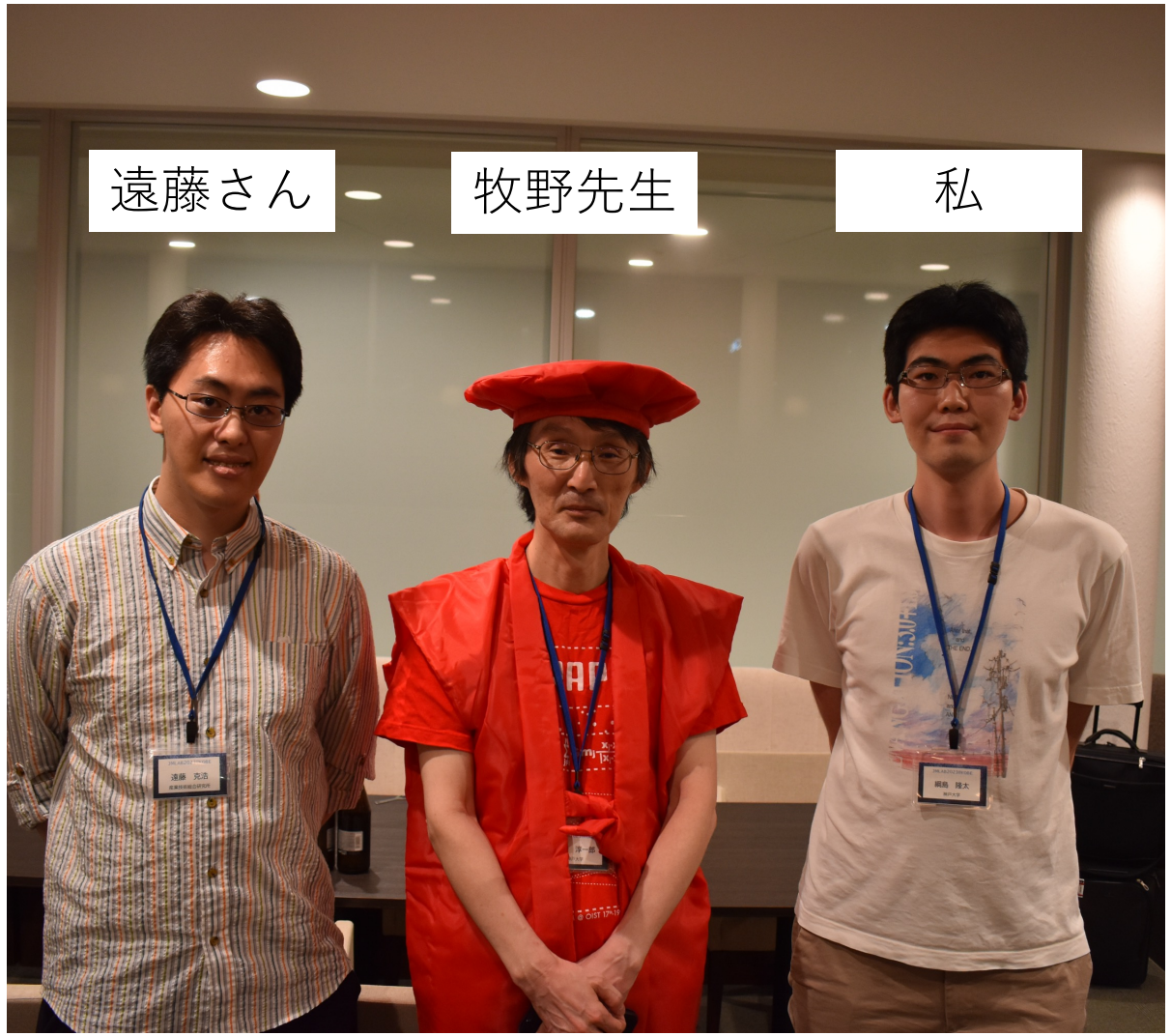
OpenACC for MN-Core

神戸大学 理学研究科 惑星科学研究センター（牧野研究室）

綱島隆太

自己紹介



- 2016年3月都立産業技術高専荒川キャンパス
  ものづくり工学科 医療福祉工学コース 卒業
- 2018年3月琉球大学工学部情報工学科 卒業
- 2020年3月筑波大学大学院システム情報工学研究科
  コンピュータサイエンス専攻 博士前期課程 修了
- 2023年3月筑波大学大学院理工情報生命学術院
  システム情報工学研究群 情報理工学位プログラム
  博士後期課程 単位取得満期退学
  - 色々あってまだ博論執筆中
- 2023年4月神戸大学大学院理学研究科 教育研究補佐員
  - 牧野研究室@CPS（惑星科学研究センター）
  - 去年のCPSセミナーで発表していた遠藤さん（産総研）の後任

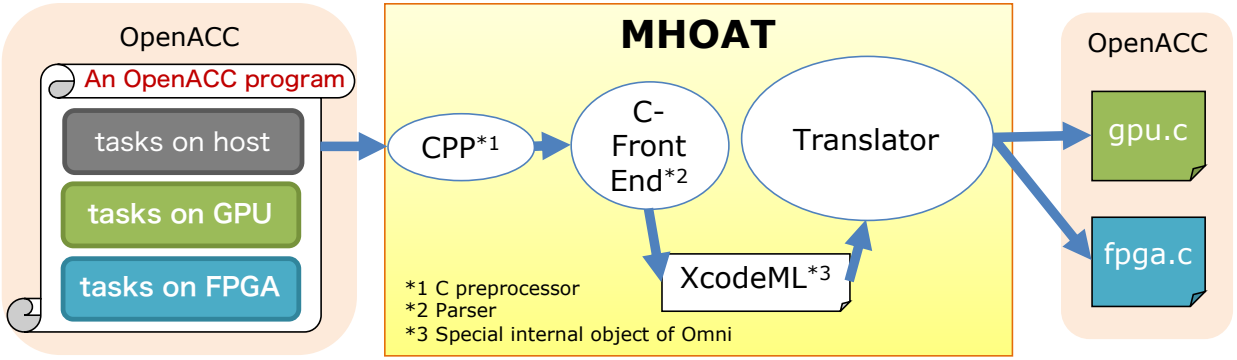
←先月の研究会にて

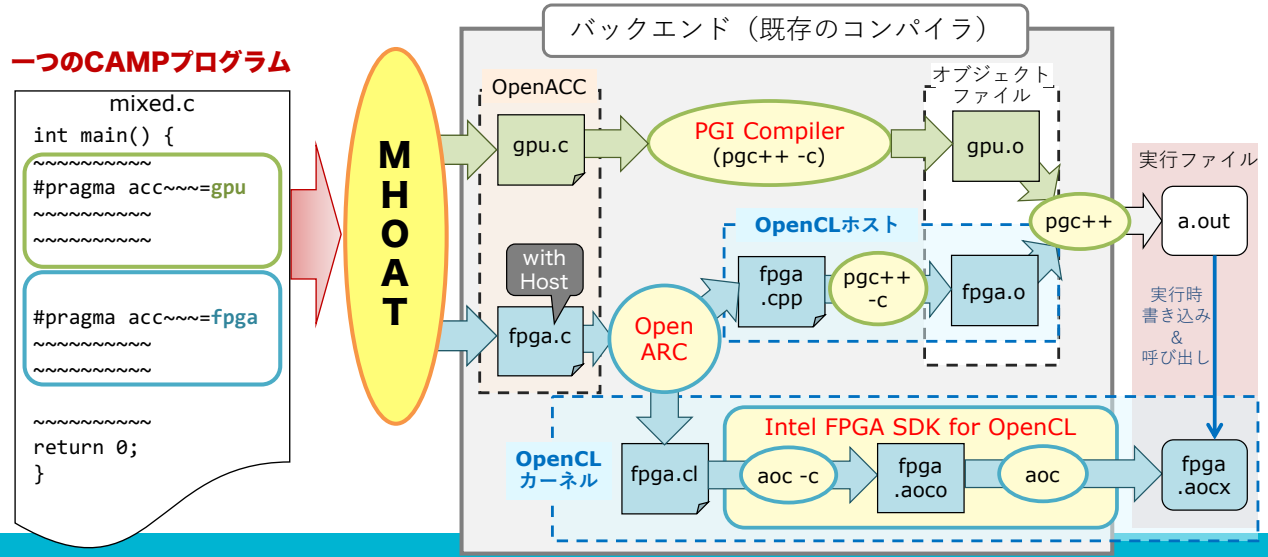
これまでやってきたこと

出典：Intel

出典：NVIDIA

- 筑波の大学院でのテーマ：
  GPU＋FPGA協調計算を統一的に記述できる
  OpenACCプログラミング環境の研究
  - Source-to-sourceコンパイラ MHOATを開発
  - 既存の処理系と組み合わせてOpenACCからGPU+FPGA混合プログラミングができるようにした
  - 実アプリでの評価をし、性能向上と、生産性の改善を確認した





今回の内容

- MN-Core用のOpenACCについて内容が固まってきたので、それを説明
  - 現実的にできるものだ、ということを伝えたい
  - あと、プログラミングがそんなに大変にならないということも
- 目次
  - MN-Coreの紹介
  - MN-Core for OpenACCの研究目的
  - OpenACCとは？
  - MN-Coreの構造とプログラミングに必要なこと
  - 設計思想
  - 設計
  - 想定している記述例
  - 言語処理系実装「案」

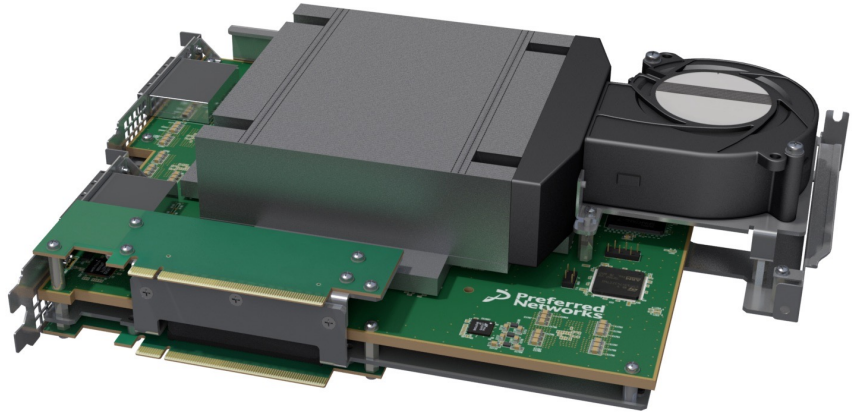
用語等（Glossaries）

用語	解説
アクセラレータ	CPUに接続して使う、より速いプロセッサ。GPUやMN-Coreなどのこと
ホスト	アクセラレータに対してCPU側のこと
デバイス	（ここでは）＝アクセラレータ。ホストと対比するときはデバイスと呼ぶことが多い
カーネル	デバイス上で実行される処理
FS	フィージビリティ・スタディ（ここでは次期スパコンの事前調査研究プロジェクトのこと）
PFN	株式会社Preferred Networks（プリファード・ネットワークス）
MN-Coreと牧野研	元々牧野先生とPFNでアクセラレータの共同研究をしていた
PyTorch	深層学習用のPythonライブラリ
API	Application Programming Interface（≠Library、何らかのプログラミングインターフェイス）

- MPI、OpenMPについては計算科学屋さん（＝アプリの人）が多いので、知ってる前提で説明
- CUDA、OpenCLについても詳細は省略、このほか各スライドで補足

MN-Coreの紹介

- 株式会社Preferred Networksの深層学習を高速化する
  専用プロセッサ（アクセラレータ）
- MN-Coreを搭載したスーパーコンピューター「MN-3」は世界のスパコンの
  省電力性能ランキングGreen500で、 2020~2021年に1位を3度獲得
  - 39.4GFlops/W（当時）
  - GPUスパコンよりも電力性能比が高かった（重要）
  - 現在後継機が開発中
- 電力性能（Flops/W）とは？
  - 電力1Wで1秒あたりに実行できる
    浮動小数点数演算の回数
  - 値が高いほど、少ない電力でたくさんの計算
    をすることができる



MN-Core Board
出典：㈱ Preferred Networks

OpenACC for MN-Coreの研究目的

- 目的： HPCでMN-Coreを現実的に使えるようにする
- なぜ必要？
  - MN-Core自体は深層学習だけでなく、汎用計算（普通の計算）も可能
  - ポスト富岳FS*でMN-Core系列のアクセラレータ搭載が検討中
  - しかし、現状のMN-Coreには汎用計算用のプログラミング環境が存在しない
  - アプリの人のことを考えるとこれではダメ（勿体ない）

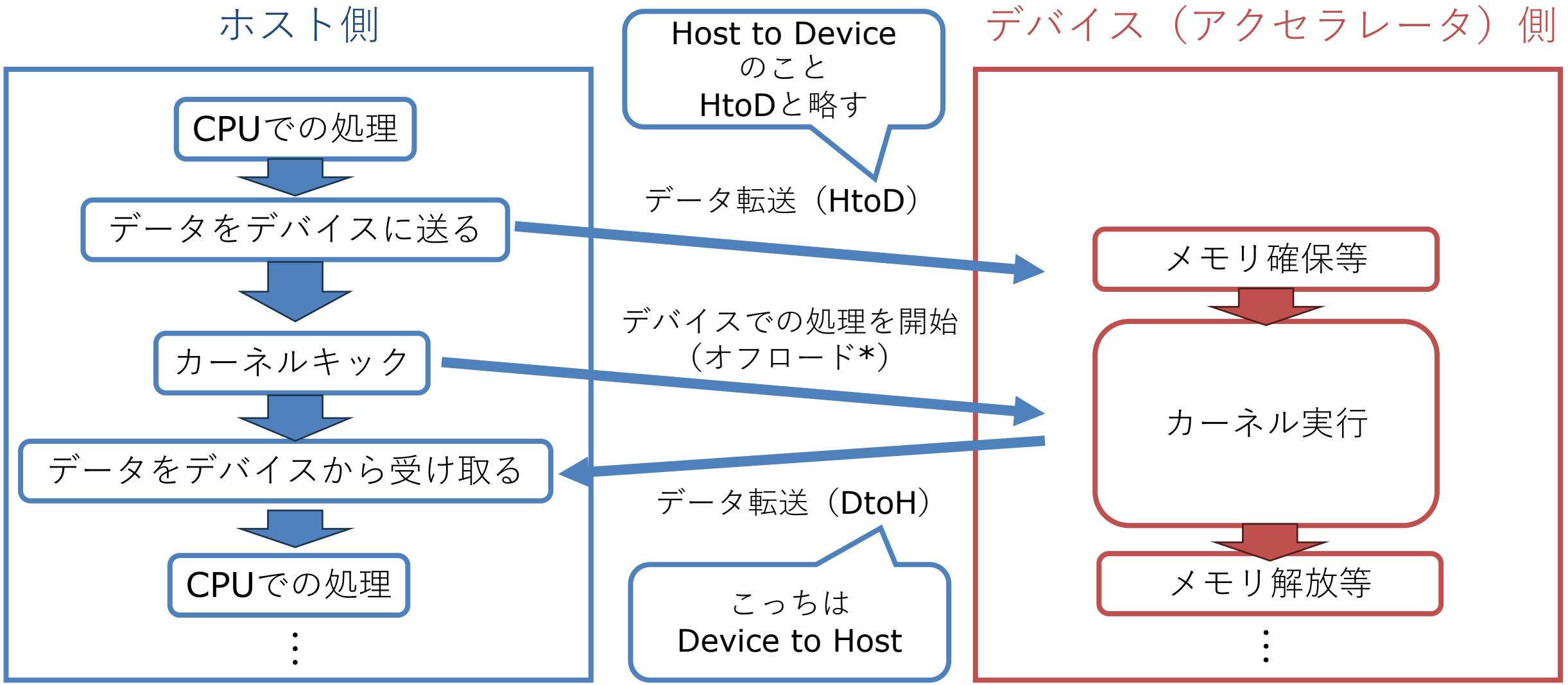
多用途にプログラミング可能なインターフェイスとしてOpenACCを検討

*次世代計算基盤に係る調査研究事業（ポスト「富岳」FS）
（富岳の次のスパコンの調査研究プロジェクト）
https://www.cps-jp.org/~comps_fspf/?ml_lang=ja

プログラミング環境を整えることの重要性

- プログラミング環境を整えないと使いものにならないし、誰も使わない
- 現状でもMN-Coreは全く汎用計算をプログラミングできない訳では無い
  - ただし、汎用計算プログラミングを想定していない
  - 非常にプログラミングで苦労する（PyTorch、アセンブリ直書きなど）
- 結局アプリの人が使えないと意味がない（以下、HPCにおける要望？）
  - 「現実的に」と言っているのはこの部分
    - 理論的には書けても、めちゃくちゃ難しいとか、性能チューニングしにくいとかではダメ
  - 手間が多いのも良くない（＝そこそこ簡単に書けることが必要）
  - できれば既存のプログラムをそのままMN-Coreで走らせたい

前提知識：アクセラレータの基本的な使い方



*オフロード：ホストの一部処理をアクセラレータ（デバイス）で実行すること

OpenACCとは？

- ディレクティブ形式でアクセラレータのプログラミングが可能なオープン規格API
  - ディレクティブとは `#include` などのような コンパイラ指示文のこと
  - 要するにOpenMPのアクセラレータ版
  - 主にNVIDIA GPUで使われている
  - OpenACCで記述されたアプリは結構増えてきている

OpenACCのコード例（オフロード対象処理の指定）

```
#pragma acc parallel loop
for (i = 0; i < N; i++) {
    out[i] = in1[i] * in2[i];
} // 計算コードはそのまま
```

OpenACCの利点

- CPUと同じ言語を使用して、既存コードに追記する形でアクセラレータ対応できる
  - 特にOpenMPを指定しているような単純なループなら、ループの書き換えは不要。CPUで動くのでデバッグも楽
- API初期化やキューの作成など細かいことをいちいち書く必要がないので、書くべきことが減らせる
- アクセラレータ上のデバイス変数、デバイスポインタをいちいち宣言しなくて良い（データ転送タイミングの指示構文を持ちながら、変数スコープの分離がない）
- CUDAやOpenCLに比べると、当然全体としてのコード量も減る

なぜOpenACCにするのか？

- OpenACCの利点の裏返しが他のAPI（言語）だと起こる

CUDA、OpenCLの場合

- CPUコードからループをカーネル関数化してループ変数をスレッドIDにするなど、根本的な書き換えが必要
  - OpenACCでは、解かせたい問題の計算内容の記述と、並列化の記述が分離できるとも言える
- API初期化やキューの作成など、常にいちいち細かいことを書く必要がある
- アクセラレータ上のデバイス変数、デバイスポインタはホストとは別に別途確保する必要がある（そもそもホストコードとデバイスコードに別れている）
- 当然全体としてのコード量は増えるし、手間も掛かる

- 実情を言ってしまえば、OpenCLなんて（CUDAすら）書きたくないという人が結構いる
- ただし、OpenACCにも欠点はあるし、他のAPIの方が良い場合もある

OpenCLとの比較（まとめ表）

- CUDAはNVIDIA専用なので、MN-Core用としての他の候補はOpenCL
  - 一応、OpenCLの方がCUDAよりちょっと大変だけど、だいたい同じ感じ

	OpenCL	OpenACC
書き換え	**大変** 言語レベルで書き換えが必要 （特にFortran→OpenCL）	**楽** 追記だけで良い （Fortran、C、C++対応）
記述難易度	**大変** 細かいことを全部書かないといけない	**楽** 色々省略可。調整したいときに追記、ということができる
細かい最適化	**できる** 上記の裏返し 性能チューニングでは重要	**あまりできない** ある程度自動でやってくれるが、その分、マニュアル制御が限定される （GPUの事例は後で言及）
コンパイラ実装難易度	**楽** 記述抽象度が低いので、一般論としてOpenACCよりは相対的に楽	**大変** 簡潔に記述できる分、コード生成に頼ることが多いのでコンパイラ開発は大変

とはいえOpenACCは難易度調整が効く

- OpenACCではまず簡単に書いて、少しずつ追記しながらチューニングできる
  - この「**とっつきやすさ**」は、大きなOpenACCのアドバンテージ
- コンパイラ開発でも、必須のディレクティブが少なく、開発者に裁量がある
  - なるべく書き方を固定させたり、独自拡張を追加することも可能
  - 書くことが増えると良くないように思えるが、コンパイラへ与えられる最適化ヒントが増えるので、実装が楽になるし、ユーザーも性能チューニングしやすくなる
  - ここを調整することで、バランスを取ることは可能

OpenMPは？

- 実はOpenMPもアクセラレータ対応している
  - ただし、OpenACCよりも後発（2、3年遅い）
  - IntelがXeon Phiやってたあたりで登場
  - 今のところというか登場から10年、規格の中身はOpenACCの後追いになっている
  - OpenMPはIntelやAMDが推進している
    - 採用数でいうとこちらが優勢と出てきた当時から言われたりもしてるが、実際のところAMDよりNVIDIAのほうがちゃんとOpenMP対応しようとしている（確認できる資料の性能や対応バージョンを見る限り）
  - 実は同じループにホストとデバイスの指定を併記できない問題もある
    - OpenACCと使い分ければ普通にできるが、OpenMPだけだとifdefを使わないといけない
  - OpenACCのほうがユーザー数が多い（そもそもNVIDIA優勢なので）
    - つまり、既存のコードもOpenACCの方が多い

他のAPIとの関係性（MN-Coreにおいて）

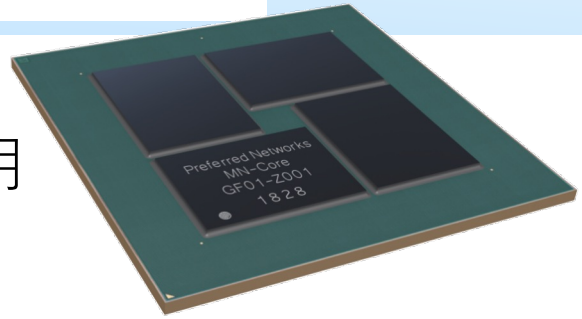
- OpenACCに一本化する訳では無い
  - OpenACCでできる限りのことはできた方が良いが、それは理想論なので他がいらない訳では無い
    - 一応、通常はOpenACCで事足りるぐらいだと、理想的だとは思う
  - OpenCLではより低レイヤーな記述ができる
    - 細かいチューニングができる
    - OpenCL for MN-Coreも別途準備されている
  - 機械学習ならPyTorchの方が楽
    - これは既にPFNで開発されている

ここからは、MN-Core上でのプログラミングについて

ハードウェアの構成から、どういうふうにプログラミングすべきかを説明

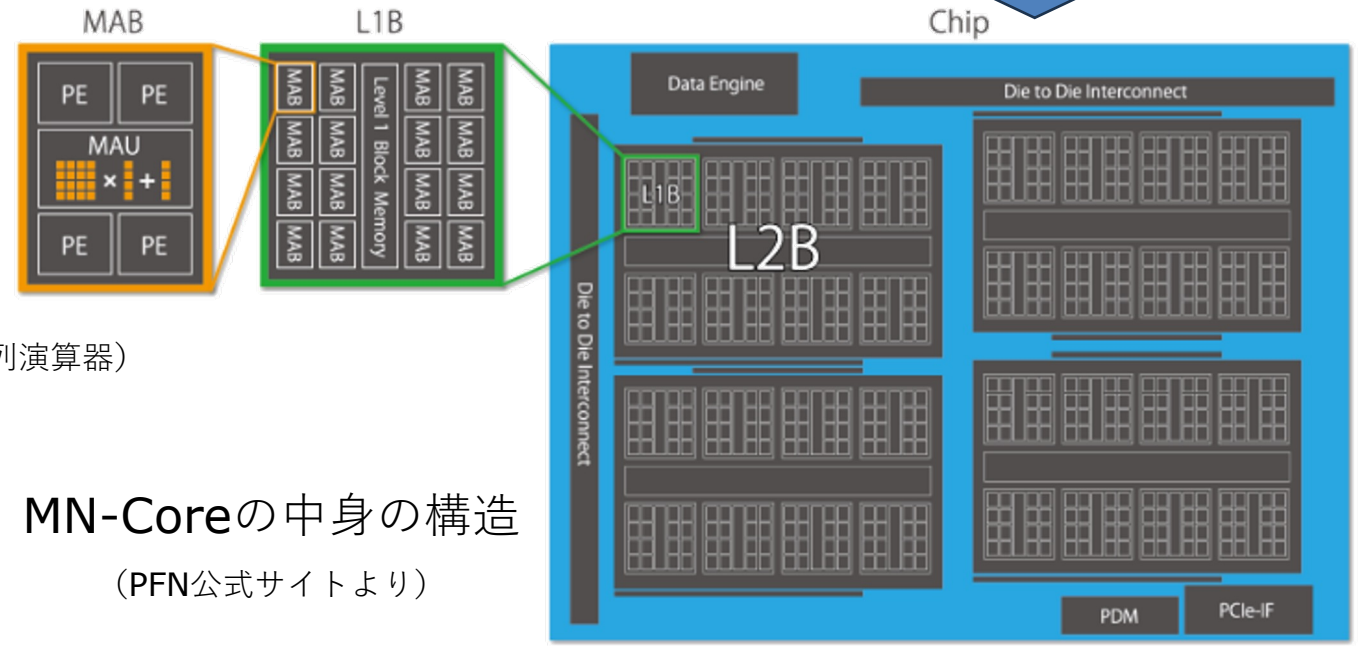
MN-Coreの構造

*SIMD (Single Instruction, Multiple Data)：
異なる複数のデータ(Multiple Data)に
同じ処理（Single Instruction）をすること



MN-Coreパッケージ

- プログラミングを考えるにあたりMN-Coreのアーキテクチャの説明
  - MN-Coreパッケージあたり8192個のプロセッサエレメント（PE）を搭載
  - すべてのPEが並列で計算を行うSIMD*アーキテクチャ
  - PEはツリー構造で配置
    - 1パッケージ
      - 4Chip（ダイ）
        - 4 L2B(Level 2 Broadcast Block)
          - 8 L1B（Level 2 Broadcast Block）
            - 16 MAB（Matrix Arithmetic Block）
              - 4 PE＋ローカルメモリ
              - 1 MAU（Matrix Arithmetic Unit、行列演算器）
- 要するに階層構造になっている
  - ただし、これはGPUも同じ
  - 大きな違いは分散メモリ型である点（詳細は次頁）



MN-Coreの中身の構造

（PFN公式サイトより）

MN-Coreの動作

- ベクトル命令はあるが、スレッドは無い
  - 本当に完全SIMD動作
  - その分ハードウェアは単純化されている
- 各PEにローカルメモリはあるが、コヒーレンシは無い
  - つまり、ハードウェアレベルで各ローカルメモリの値を監視したり、自動で更新された値を共有したりすることはない（スクラッチパッドメモリという）
    - ハードの単純化だけでなく、ここは性能チューニングにおいてかなり重要なところ（後述）
    - GPUにもコヒーレンシのないシェアードメモリというものがあるが、機能的には実質キャッシュである（シェアードメモリも後述）
    - MN-Coreのローカルメモリはキャッシュではない。常にPEの近くに計算するデータを置く設計（DRAMは遠くにある）
  - このあたりはすべてプログラムに委ねられる＝明示的に制御可能
  - 要するに、プログラムでMPIと同様のデータ管理をすれば良い

つまり、何をすればよいか？

- アクセラレータ上で、MPIみたいなことを行えれば良い
  - MPIみたいな、というのはつまり、分散メモリ型のプログラミング

面倒くさそうだけど…

*キャッシュブロッキング：
ループをキャッシュに乗るサイズに細かく分割することでキャッシュヒット率を向上させて高速化を図る手法

- 実際のところ、キャッシュのせいで性能チューニングが辛くなってるところがある
  - ユーザーからはキャッシュの動きが明確にわからない
    - さらにはレイテンシの問題もある
  - そもそもキャッシュブロッキング*を書くのが面倒すぎる（あれは果たしてプログラミングと呼べるのか…？）
- 特に近年ではB/F比の低下、つまり計算に対してデータ移動がボトルネックになることが多く、ここを明示的に制御できたほうがチューニングし易いはず
  - つまり、キャッシュによるチューニングの苦労から解放されるためには、分散メモリ型プログラミングのような明示的な制御への移行が一つの解
    - MN-Core自体がそのように設計されている
    - 実は昔のスパコンはこれだったという話がある
  - 大変そうだが、過去の事例から、OpenACCではMPIよりは楽に書けるはず（後述）

OpenACC特有の事情（1）

- そもそも、配列を確保するメモリ階層をOpenACCでは指定できない
  - デバイスに送る、ホストに返すという操作しか書けない
    （コード上の変数スコープが、ホストとデバイスで別れないので）
  - というか、キャッシュブロッキングみたいなことを書けないと適切にシェアードメモリを割り当てることはできないので、ディレクティブ記述だけだと根本的に無理
  - コンパイラが頑張れば、という話はあるかもしれないけど、それこそ机上の空論
  - OpenACCの登場から10年以上経っても、GPUのシェアードメモリはちゃんと使えるようになっていない

OpenACC特有の事情（2）

- 実際、OpenACCからGPUのシェアードメモリを使うのは大変難しい
  - GPUではシェアードメモリを使いこなすことが性能チューニングの肝
  - 普通に行列積を書いただけだとOpenACCでは酷い結果になる
    - シェアードメモリがあまり効かない通信ボトルネックなコードでは、CUDAとの差は相対的に縮まるが
  - 一応シェアードメモリのための指示文もあるが、逐次コードからシェアードメモリを適切に使うようにコード生成することが（おそらく）難しく、対応不十分
  - 元のコードをキャッシュブロッキングするとうまくいくが、結局CUDAと同じぐらい大変になる
    - 元のコードをいじるのは本来のOpenACCの使い方ではなく、ディレクティブ形式APIの思想からはかけ離れる
- なので、 OpenACCとの親和性は、GPUのようなメモリ階層のあるハードより、PEのローカルメモリだけ考えれば良いMN-Coreのほうが高いと言える
  - むしろ、OpenACCは結構GPUのことしか考慮されてないので、分散メモリ型を我々が提案する

実際できそうなのか？

- OpenACC for MN-Coreで、追加で考えるべきところは、分散メモリ型プログラミングをどうするかという部分
  - ここは現状のOpenACCでは非対応
- 実はディレクティブ形式で分散メモリ型プログラミングAPIというのは既にある
  - XcalableMP（XMP）*：ディレクティブ形式でノード並列プログラミングができるAPI
    - MPIよりも簡単にノード並列プログラミングを可能にすることが目的
    - High Performance Fortran (HPF)をより実用的にした
    - HPFは通信の記述を省略しすぎてチューニングがしづらくなってしまっていた
    - また、それによってコード生成の実装コストも高くなりすぎてしまったので、XMPではそこを改善
- これはちゃんと実装されているので、そういうコード変換ができるというのは実証済み
  - これぐらいのディレクティブがあれば、逐次コードからコンパイルできる、という参考になる

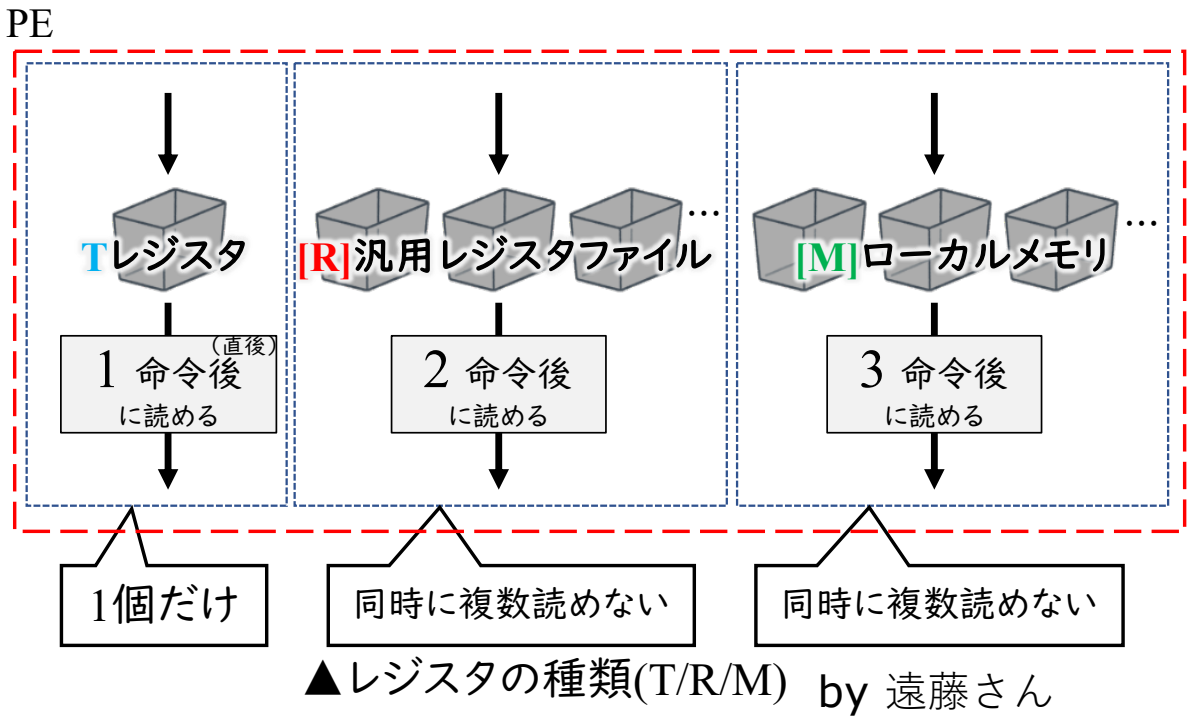
*XacalableMP：https://xcalablemp.org/ja/

OpenACCでどうすればよいかというと

- 基本的には既存のOpenACC記述でオフロード（MN-Core上の計算を起動）
- データ転送やPEへの演算割当てについては分散メモリ型であることを考慮する
  - ここは独自拡張を定義し、記述できるようにする

もう一つ重要な問題：レジスタ割付

- MN-Coreにはローカルメモリも含め、複数種類のレジスタが存在
  - 書き込み後に読み出しできる動作サイクル数、物理的な個数が異なる
  - 制約の中で、遅延がなるべく最小限になるように使わないといけない
    - そうしないと性能が出ない
    - 結構クリティカルな問題
- ここは遠藤さんが既に解決済み
  （昨年10月のCPSセミナーで発表）
  - これは組合せ最適化問題
  - シミュレーテッドアニーリングで上手くいくことがわかった
  - OpenACCでもこれを取り入れる予定



▲レジスタの種類(T/R/M) by 遠藤さん

どう設計すべきか？

設計思想（特にインターフェイス）

GPU版OpenACCでの課題

- 性能チューニングに関すること
- GPUの場合、スレッド数調節でチューニングというよくわからない作業が発生する
  - 性能決定要因が複雑すぎて、実行してみないとわからないところがある
  - これがなかなか辛い
- 加えて、OpenACCの場合、調整するパラメータがハードと完全に一致していない
  - オープン規格なので、汎用化のためにそうなっている（OpenCLでも同じ）
  - しかし、こういうことをされるとユーザーが実際にプログラミングするときに困る
  - gang, worker, vector*という3階層のうち、どれをどう当てはめるかはコンパイラによる
    - NVIDIA GPU（CUDA）だとblock, threadの2階層なので、どれかがダブるか、どれかが無効になる
  - NVHPCでもどれが実際のハードにどう反映されるかは、コンパイル結果を見ないとわからない
    - ついでに言えば、指定した数値通りにコンパイラがコンパイルしてくれるかすらわからない

*OpenACCのループ割当を指定するパラメータ。階層だけでなく、具体的な数値も指定可能

FPGAでの話

- FPGAとは？：再構成可能ハードウェア
  - 普通のハードのコンパイルでは、ある処理をするための機械語命令が生成される
  - FPGAでは、ある処理をするための回路自体が生成される（専用回路を生成）
- 抽象化の問題はGPUよりずっと顕著
  - OpenACC for FPGAを触っていたが、わりとOpenCLに掛かる問題
  - ベンダー提供コンパイラのインターフェイスがOpenCLなので、OpenCLの制約に縛られる
    - OpenACC → OpenCLというSource-to-sourceコンパイルをしてから、OpenCLコンパイルする
  - OpenCLでは表現できないことを無理やりOpenCLで表現するようになっている
    - 本来はハードウェア記述言語という専用言語で細かく書かないといけないところを普通のソフトウェアのように書けるようにしている
    - 性能を出すにはキャッシュブロッキングのような、コンパイル後のハードウェアを想定したコード記述が必要
- OpenACC for FPGAでは変にOpenCLを抽象化していなかった
  - むしろそんなことしたらチューニング不可能になる
  - 私は冗長な記述を要する部分以外は、これ以上抽象化するべきではないという考え

つまり、どうすればよいか

- 性能にダイレクトに関わるところはユーザーが明示的に制御できたほうが良い
  - ただし、コンパイラがちゃんと最適化できるならそのほうが良い
  - HPCに限らず、アクセラレータなど性能を求めるハードでは、下手に抽象化を採用すべきではない、ということでもある
- 但し、ハードと噛み合わない論理的構造のインターフェイスを採用してはいけない
  - 制御できたとして、ユーザーにとって何がどうなるのか分からなければダメ
  - まず最初にどうしたら良いかもよくわからなくなる
- 実現できるかわからないコード自動生成を前提として、仕様を決めてはいけない
  - インターフェイスばかり先に考えていてはダメ
  - ユーザーがそんなに考えなくても書ける部分は、冗長でも最初は書かせてしまったほうが良い
    - つまり実装優先。そうしないと永遠に実装ができなくて、誰もうれしくない結果になる

ユーザーは何を書けばよいか？

- コンパイラ視点で考えてみると...
  - 生成するコードやアセンブリを決定できる情報が欲しい
    - 逐次コードに含まれない並列化に関する情報
    - コード解析が難しそうなメタな情報（例えば、袖領域はどこからどこまで？）
    - これらをディレクティブで教えてあげればちゃんとコンパイルできるはず
  - ディレクティブはコンパイラへのヒントである
    - 最低限、生成するコードを決定するのに必要な情報が揃ってればコンパイルできる
    - そういう考えのもと、与えるべきヒントを考える必要がある
  - 細かい計算とかはコンパイラでやらせれば良い
    - 簡単な計算なら大したコストではない
    - （実装コストとトレードオフだが、簡単なことであれば、）
      なるべく冗長なことはユーザーに書かせずにコンパイラでやってしまうべき

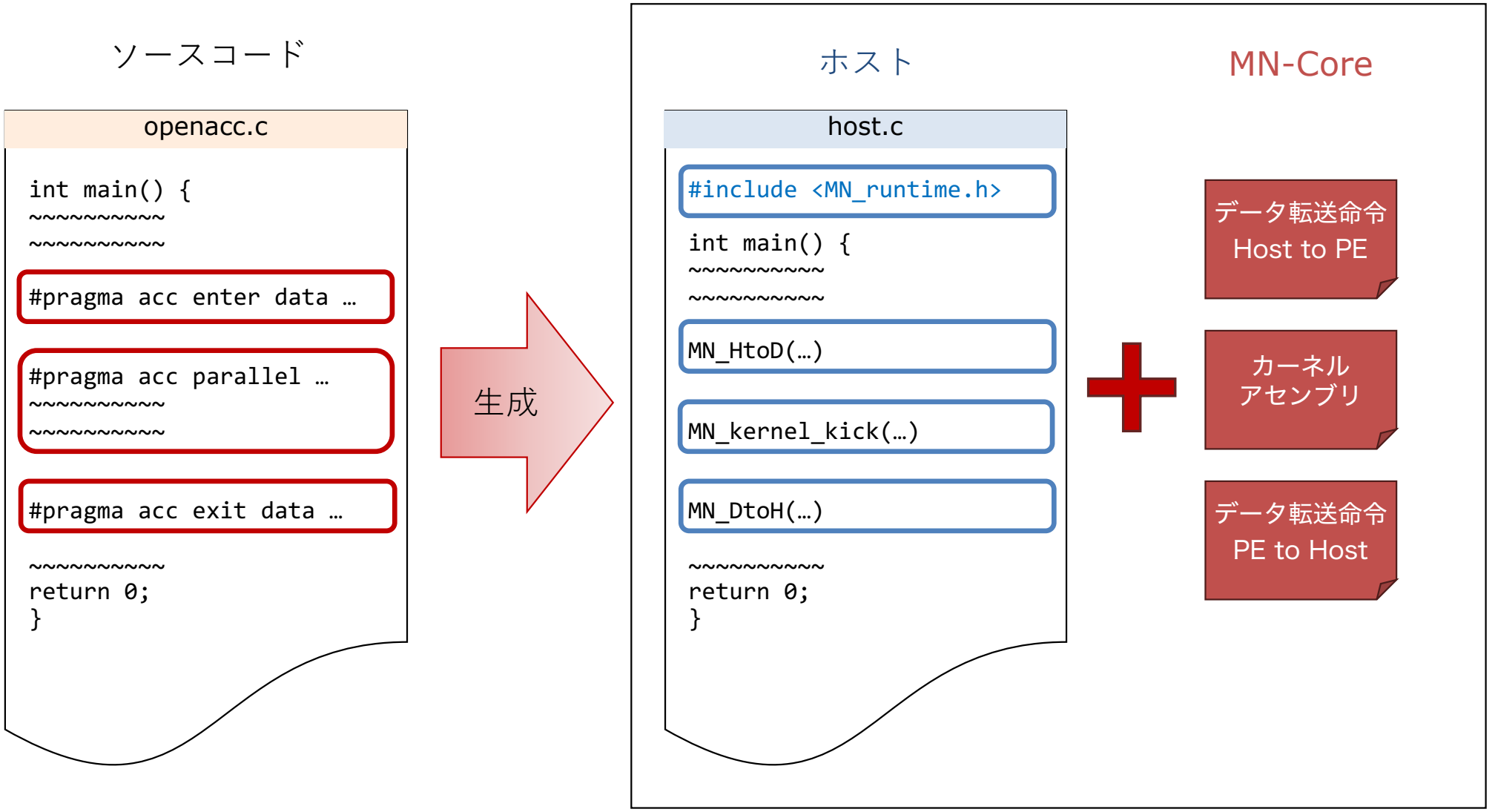
設計

要件、大まかな仕様、言語処理系の変換の概要、プログラムの動作

要件・仕様

- OpenACCでそんなにたくさんのことができなくても良い
  - OpenCLも別途準備中なので
- 分散メモリ型並列プログラミングができるようにする
  - 基本的な部分はOpenACC、分散メモリの部分だけ独自拡張
  - XMPを参考に（XMPと同じような記述をするが、インターフェイスはOpenACCに合わせる）
- 袖交換通信ができるようにする
  - 要するにステンシル計算ができると、結構なアプリが動く
  - XMPでもそういう感じだったので、このあたりを目指す
- Fortran実装優先で
  - Fortranコードのアプリが多いため

言語処理系の変換の概要

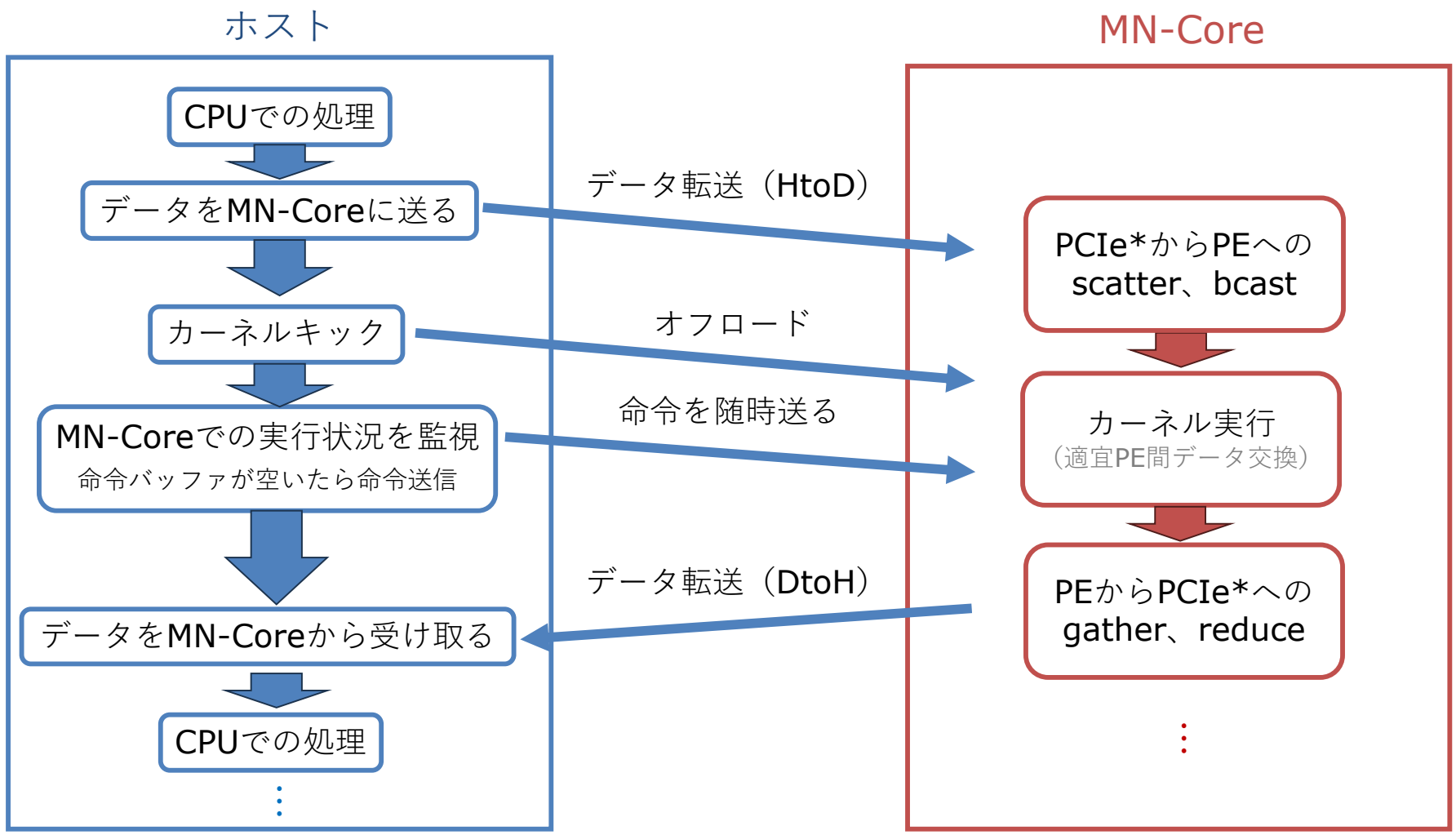


- 要するに
  - データ転送命令生成機能
    - HtoD
    - DtoH
  - カーネル生成機能
    - 袖交換含む
  - ホスト側制御のための ランタイム ライブラリ

が必要

プログラムの動作概要

*PCIe：PCI Express。CPUとデバイスとの接続インターフェイス



- 大体さっきの基本的なアクセラレータの図と同じ
- 違いは、
  - HtoDのときに決められた分配方法に従い、データをそれぞれのPEのローカルメモリに配置すること
    - このあたりは集団通信
  - MN-Coreの命令バッファの容量を考慮していること
  - カーネル実行時にPE間通信する場合があること
    - ここも基本的に集団通信

実際にどう記述するか？

インターフェイスを示す
サンプルコードを見せながら解説
実装中に変わるかもしれないので確定ではないが、これに沿う

各PEへのデータ分配（配列分割方法）指定

- acc data copy(...) l2(a[分割数][分割数]...) l1(a[分割数][分割数]...)
  mab (a[分割数][分割数]...) pe(a[分割 or 倍数][分割 or 倍数]...)
  - 各階層（L2B、L1B、MAB、PE）の分割数を指定するデータ転送ディレクティブの節*
    - data, enter dataディレクティブに指定できる
  - 分割数とは、各階層の全個数をどう分割するかということ
    - たとえば、二次元配列aがあるとして、L2Bに1次元目は8要素、2次元目は2要素割り当てたい場合
      L2Bは16個あるので、1次元目は2分割すれば良いからa[2]と指定
      2次元目は8分割すれば良いからa[8]と指定
  - []は配列の次元の数だけ指定
    - ただし、 [MAX]の場合はその階層では次元を割り当てないということになる
    - 最高次元（一番右の次元）が[MAX]の場合は記述省略可
  - PEについては、l2、l1を指定すると決まるので省略可
    - PEは倍数を指定可
  - MABはMABで分割したい時に指定（分割数はPE同様、自動的に決まるので省略可）
    - 指定しない場合はMABで分割しないで、直接PEを割り当てる

*節： clause。ディレクティブ中のオプション指定のこと

配列分割指定の補足

- XMPではtemplateというものがあるが、今回は独立指示分ではなく節として規定
  - 節にしないとユーザーの抵抗があると思われる
  - 独自拡張とはいえ、変化しすぎると良くない
  - よって、OpenACCの書き方に合わせて、既存のHtoDデータ転送と一緒に指定
- 複数の配列に同じ分割方法を指定する場合は、省略記述可
  - templateを採用しないので、省略記述を用意
  - ,（カンマ） で続けることで同時指定可
    - a,bが同じ次元数で、同じ分割方法の場合：I2(a,b[4][4])
  - なお、別の分割方法の配列を指定したい場合は ;（セミコロン）で区切る
    - 異なる次元数の配列cも一緒に指定したい場合： I2(a,b[4][4]; c[4])
  - データ転送ディレクティブの全配列が同じ分割方法なら、変数名は省略可：I2([2][2][4])

分割数を指定することについて

- これが一番最初に思いついた
  - ちょっと面倒なので色々別案も考えてみたが、これがおそらく一番安全
  - 例えば、L2Bの分割を考えるときには
    「L2Bの個数は16だから、8要素割り当てるには2分割だな」
    といった感じで、ハードについて強制的に考えることになるので変な割当てになりにくい
  - コンパイラがコード生成する上で必要な情報は基本的にこれで表せてしまう
    - 本当はハード的にはもうちょっと複雑なことができるが、基本はこれで良い
- 抽象度ゼロなので、分割数を指定するとハード依存になるという問題はある
  - ただし、OpenACCにはdevice_type節という、指定をあるデバイスに限定させる節がある
    - device_type(*DEVICE_NAME*) のように使用
    - 複数の異なるアクセラレータにそれぞれの節の指定を定義できる
    - この節以降の節が*DEVICE_NAME*で指定したアクセラレータ専用になる
    - この節を使うようにすれば、コードの可搬性は保持できるはず→

```
// device_type節の適当なコード例
#pragma acc data copyin(a[0:1024], b[0:1024])
copyout(c[0:1024]) ¥
device_type(MN_Core) l2(a,b,c[8]) l1(a,b,c[1]) ¥
device_type(MN_Core2) l2(a,b,c[16]) l1(a,b,c[1])
```

袖領域・袖交換

- `acc data copy(…) l1(…) l2(…) shadow(a[1:1][1:1][1:1])`
  - 各次元の袖の要素を指定 [マイナスの要素数:プラスの要素数]
  - これもdata, enter dataディレクティブに指定できる
  - XMPと内容は同じだが、これも独立した指示文ではなく、節とする
- `acc loop reflect(a)`
  - 袖更新の発生をコンパイラに知らせる
  - これもXMPと内容は同じだが、節とする

gang, worker, vectorの指定

- OpenACC for MN-Coreでは無視する
- 配列分割方法の指定で決まってしまうので不要
  - XMPでもループに改めて分割方法を指定したりはしない

ベクトル加算（サンプルコード）

```
void vecadd(...) {
    ...
// データ転送（copyinがHtoD、copyoutがDtoH）
// 赤字が分配方法指定の独自拡張
#pragma acc data copyin(a[0:1024], b[0:1024])
copyout(c[0:1024]) l2(a,b,c[8]) l1(a,b,c[1])
{
// オフロードするコードブロックの指定
#pragma acc parallel
#pragma acc loop independent
    for(int i=0; i<1024; i++)
        c[i] = a[i]+b[i];
}
    ...
}
```

- VectorAddはGPUとあまり変わらない
  - 追加部分は赤字だけ

7点ステンシル（サンプルコード）

```
void stencil() {
    ...
#pragma acc enter data create(temp[0:128][0:128][0:128]) ¥
copyin(b[0:128][0:128][0:128], a[0:128][0:128][0:128])  ¥
l2(a,b[4][4]) l1(a,b[4][2]) mab(a,b) ¥ // [1][2][1]は省略可
// 各次元の袖の要素を指定[マイナスの要素数:プラスの要素数]
shadow(a[1:1][1:1][1:1])
    ...
    for(count=0; count<N; count++) {
    ...
#pragma acc parallel
// 袖交換の発生をコンパイラに知らせる
#pragma acc loop reflect(a)
        for(int i=1; i<128-1; i++){
            for(int j=1; j<128-1; j++){
                for(int k=1; k<128-1; k++){
                    temp[i][j][k] = c1*(a[i-1][j][k]+a[i+1][j][k]
                                       +a[i][j-1][k]+a[i][j+1][k]
                                       +a[i][j][k-1]+a[i][j][k+1])
                                      +c2*b[i][j][k];
                }
            }
        }
        for(int i=1; i<128-1; i++){
            for(int j=1; j<128-1; j++){
                for(int k=1; k<128-1; k++){
                   a[i][j][k] = temp[i][j][k];
        } } }
    }
    ...
#pragma acc exit data copyout(a[0:128][0:128][0:128]) delete(b,temp)
    ...
}
```

- こっちもそこまでたくさん変化があるわけではない
  - loopディレクティブはgang, worker, vectorの指定が無くなったので、最外ループだけ書けば良くなっている
- スカラー変数はOpenACCの仕様では何もしなくてもfirstprivate
  - なので、c1, c2は各PEに放送される

これだけで良いのか？

- 結局のところ、機械語にしても関数にしても、以下が決まればコード生成できる
  - どういう処理をさせるのか（ループの中身の演算子で決まる）
  - どのデータを処理するのか（配列分割指定とループの中身の変数で決まる）
- MN-Coreは完全SIMDで、分岐処理もないので、考慮すべきことは少ない
  - 命令バッファ容量とかの話もあるので、コンパイラ実装するときにこれだけで良い訳では無いが、単純な処理なら考慮することがかなり少ないのは事実
    - そもそもMN-Coreは命令数もそんなにたくさんあるわけではない

話を具体的にすると

- 配列の各要素がどのPEにあるのか、どのような計算をさせればよいかがわかってれば計算できるはず
  - 1つのPEで2回以上ループを回すとしても、1PEあたりの配列要素割当て数が分かれば良い話
- 配列の各要素の情報（PEの場所など）は、配列のパラメータを入れる構造体をコンパイラ内で用意しておけば簡単に保持できる
- ハードの方の数字は完全に固定されているので、四則演算だけで色々決まる
  - 例えば、l2, l1だけ配列分割を書いた場合の各PEのデータ割当て数とか
  - 実装にもよるが、袖交換通信に関しても、どこが袖領域で、更新後どこが必要かが分かってれば良い

配列分割も自動でできるかも？

- 配列分割についても結局組み合わせ最適化問題なので、コンパイラで自動的にコード生成できると嬉しい
  - ツリー構造における集団通信のバンド幅や命令サイクルは決まっているので、ちゃんと計算すれば最適な分割方法は決まるはず
  - ただし、これを実装できるかどうかはそれこそ机上の空論なので、まずは現実的に手動チューニングできるインターフェイスを実装する

処理系実装案

周辺調査を行って固まりつつある

コンパイラ

- XcodeML＋何かコード変換ツール
  - XcodeMLはXML形式のOmniCompiler(Omni)*の中間コード表現
    - C、Fortran対応でソースコードの情報を完全保持（＝可逆変換）
  - XcodeML自体の解析には、 Omniで使われているオープンソースの フロントエンド・バックエンドパーザー xcodeml-tools を流用
    - xcodeml-toolsはOpenACC対応していて、独自拡張ディレクティブの対応も比較的容易
  - Omni （＋xcodeml-tools）自体は綱島が博論で触っていたツール
    - XMPもこのコンパイラで実装されているので、おそらく参考にできる
    - ただし、これをMN-Core対応させるのはドキュメントが無いので大変なため、コード変換には別ツールを使う
  - Omniを使わないでOpenACCのSource-to-sourceコンパイラ開発、というのは前例がある
- 実はNVIDIAと米国ORNLがLLVMでOpenACCコンパイラを作っている
  - まだちゃんとできていないようだが、最終的にはこれのMN-Core版を実装してコミットしたい

*OmniComipler：https://omni-compiler.org/

ランタイムライブラリ

- ホスト側はランタイムライブラリが必要
  - 少なくともPFN側にはC++のものしかないので、PFNの人と協力してちゃんとしたものを開発する予定（よろしくお願いします）
  - 神戸大学にはCで書かれたものがあるものの、少し難があるということを聞いている
  - Cで書かれていると、Fortranでも、C++でもリンク可能になる
    - 一応、FSではアプリの数的にFortran優先
  - コンパイラ開発を考えると、ランタイムライブラリはCでないと色々大変

まとめ

まとめ

- MN-Core用のOpenACCについて考案した
- OpenACCの特徴や既存のデバイスでの課題点を洗い出した
- MN-Coreの構造に踏まえて必要なプログラミングモデルについて考えた
- 既存のデバイスにおける性能チューニングの問題について取り上げた
- これらを踏まえ、 OpenACC for MN-Coreをどのようなインターフェイスにすべきかまとめ、具体的に示した
- 言語処理系についても設計と、現時点での案を示した

今後の予定

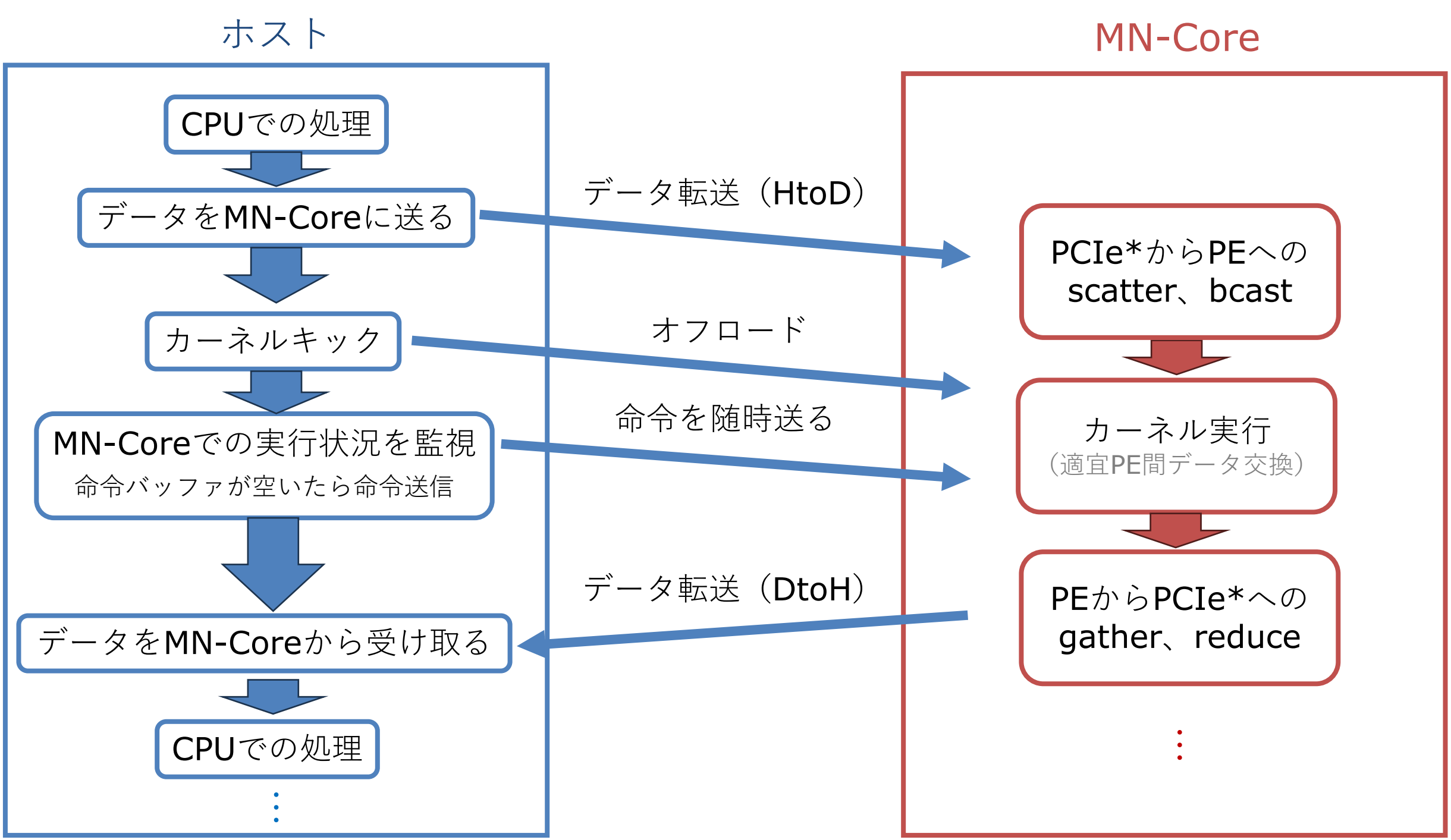
- まずはプロトタイプを開発し、ベクトル加算ができるというところまで示す
  - 12月ごろを目処に
  - 大まかなアセンブリでの処理の流れについても把握できている
- ドキュメントを用意する
  - 特に、分割方法についてはどのようにすべきか基本的な設定方法を示す

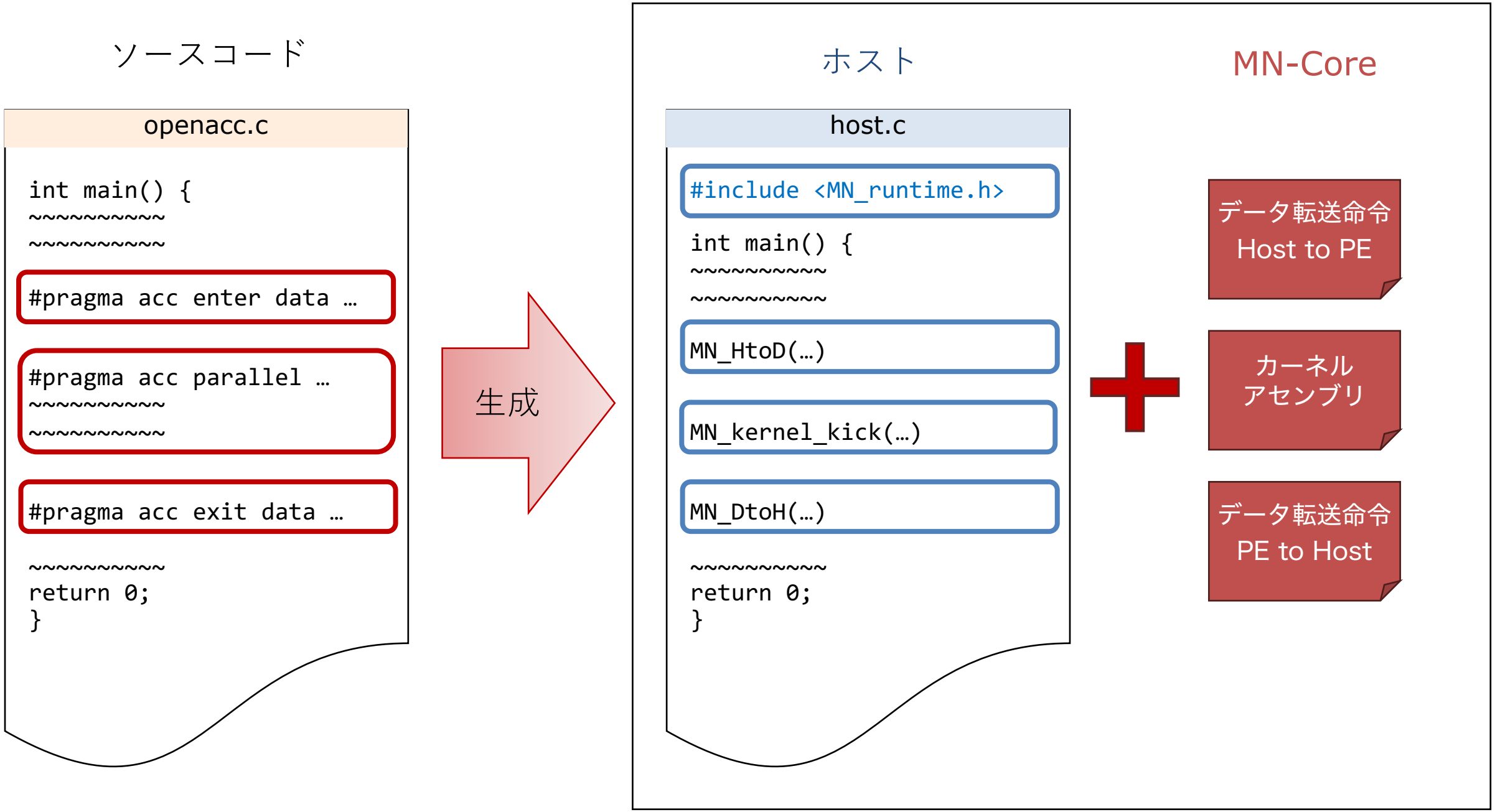
本研究は、文部科学省「次世代計算基盤に係る調査研究」事業の助成を受けたものです。
This work was supported by MEXT as " Feasibility studies for the next-generation computing infrastructure".

予備スライド

袖交換の考え方

- 各次元がどこで分割されているかにもよる
- L1B内に全て収まっている（PEで分割してない）次元を交換する場合
  - MAB内のPEのデータはお互い直接アクセスできる
  - MAB間はお互い交換が必要だけど、一旦L1Bを介さないといけない
    - 個別の通信についてはMAB単位のほうがまとめてデータを送受信できるので、袖交換を考えるとMAB単位での分割が良い
- L1Bで分割している次元を交換する場合
  - MAB間ではL1Bを介して交換（i-1の要素なら一つ前のMABから持ってくる）
  - 一つ前が別のL1Bの場合はL2Bを介してL1B間で交換
    - 最低次元がストライドになる場合はマスクが必要
    - なるべくストライドは避けるように分割する
- L2Bで分割している場合
  - まずL1Bで次元分割している場合と同じ手順
  - PDMを介して交換


ホスト
MN-Core
CPUでの処理
データをMN-Coreに送る
カーネルキック
MN-Coreでの実行状況を監視
命令バッファが空いたら命令送信
データをMN-Coreから受け取る
CPUでの処理
データ転送（HtoD）
オフロード
命令を随時送る
データ転送（DtoH）
PCIe*からPEへの
scatter、bcast
カーネル実行
（適宜PE間データ交換）
PEからPCIe*への
gather、reduce


ソースコード
openacc.c
int main() {
~~~~~~~~~~
~~~~~~~~~~
#pragma acc enter data …
#pragma acc parallel …
~~~~~~~~~~
~~~~~~~~~~
#pragma acc exit data …
~~~~~~~~~~
return 0;
}
生成
ホスト
MN-Core
host.c
#include <MN_runtime.h>
int main() {
~~~~~~~~~~
~~~~~~~~~~
MN_HtoD(…)
MN_kernel_kick(…)
MN_DtoH(…)
~~~~~~~~~~
return 0;
}
データ転送命令
Host to PE
カーネル
アセンブリ
データ転送命令
PE to Host