資料１－３

評価委員会資料 2025/3/6

牧野淳一郎
神戸大学

「次世代計算基盤に係る調査研究」第 13 回評価委員会 2025/3/6

報告概要

- FS の目的
- 神戸大学チームの目標
- アーキテクチャ検討からわかったことのまとめ
- システムソフトウェア検討・アプリケーション検討の概要

FS の目的

２．事業の概要

（１）調査研究の内容

本調査研究では、ポスト「富岳」時代の次世代計算基盤の具体的な性能・機能等について、サイエンス・産業・社会のニーズを明確化し、それを実現可能なシステム等の選択肢を提案する。その際、我が国として独自に開発・維持するべき技術を特定しつつ、要素技術の研究開発等を実施する。

具体的には、システム（アーキテクチャ、システムソフトウェア・ライブラリ、アプリケーション）、新計算原理、運用技術を対象に、技術動向等を調査し、技術的課題や制約要因を抽出しつつ、次世代計算基盤に求められる具体的な性能・機能を明らかにする。

研究チームのミッション

Ⅰ．システム調査研究チーム

アーキテクチャ、システムソフトウェア・ライブラリ、アプリケーションを対象として調査研究を行う。本チーム内において、アーキテクチャ調査研究、システムソフトウェア・ライブラリ調査研究、アプリケーション調査研究を実施するグループをそれぞれ編成する。また、アーキテクチャ調査研究グループにおいては、複数のアーキテクチャを検討するにあたり、必要に応じて複数のサブグループを編成すること。

達成目標

- チームを構成する各グループの検討結果を総合し、次世代計算基盤として想定されるシステム（ アーキテクチャ、システムソフトウェア・ライブラリ、アプリケーション） を提案する。

Ａ．アーキテクチャ調査研究グループ

達成目標

- アーキテクチャに関する技術動向調査及び「C.アプリケーション調査グループ」の検討を踏まえ、評価指標として考慮すべき項目の検討を行い、達成すべき目標とその優先順位を提案する。
- 複数のアーキテクチャシステムについて「C.アプリケーション調査グループ」との連携により抽出した評価対象アプリケーションによるシステム評価を行う。

研究内容

- アーキテクチャ（ プロセッサ、メモリ、ストレージ、I/O等） およびそれに関連する技術分野を研究対象とする。
- 複数のアーキテクチャを対象に比較検討を行う。　　等

研究チームのミッション

B． システムソフトウェア・ライブラリ調査研究グループ

達成目標

- システムソフトウェア・ライブラリに関する技術動向調査及び「C.アプリケーション調査グループ」の検討を踏まえ、評価指標として考慮すべき項目の検討を行い、達成すべき項目とその優先順位を決定する。
- 「C.アプリケーション調査グループ」との連携により抽出した抽出した評価対象対象アプリケーション等によるシステム評価を行う。

研究内容

- OS、コンパイラ、ファイルシステム、ライブラリ、フレームワーク、言語等およびそれに関連する技術分野を研究対象とする。 等

C． アプリケーション調査研究グループ

達成目標

- 次世代計算基盤の果たすべき役割及び最新の計算科学ロードマップを踏まえた、システムが満たすべき具体的な性能・機能検討の提案を行う。
- 他チームと連携し、評価対象アプリケーションの抽出を行う。

研究内容

- デジタルツイン技術の進化を支え、世界をリードする研究成果の創出やSociety5.0の推進、SDGsの達成に貢献するプラットフォームとして必要なアプリケーション分野（ AI・データ科学やデジタルツインをはじめとする分野） 等を研究対象とする。
- また、従来分野に加え、社会科学分野や、オープンな計算環境や商用展開を想定した適切な評価アプリケーションを検討対象に含める必要がある。 等

神戸大学チームの目標

調査研究の概要（アーキテクチャ調査研究グループ）

取組概要

分野自体の技術トレンド、
半導体プロセス技術動向からポスト富岳時代に実現可能なシステム
がどのようなものかを検討し、評価対象アプリケーションによるシステム評価を行う

調査研究体制

神戸大学

◎Preferred Networks

東京大学

国立情報学研究所

調査内容

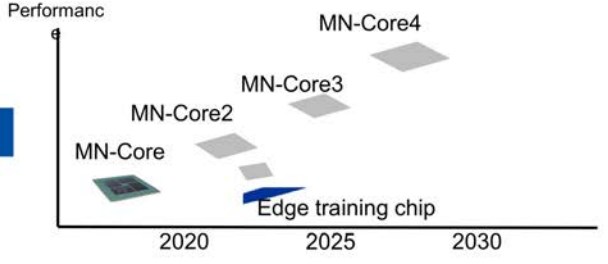
独自アクセラレータ・CPU構造見直しによる省電力化・効率改善に重点

アクセラレータ側:
現時点で同じ半導体技術では世界最高の電力当り性能を実現している

MN-Core をベースに 一層高い電力性能、チップ面積あたり性能を目指すと共に、アプリケーションとのコデザイン により高い実行効率を実現する

汎用プロセッサ側:

RISC-V ベースの独自設計で世界最高レベルの性能を目指す。



スケジュール

初年度
アーキテクチャのレファレンス設計

2年度
アーキテクチャ・システムソフトウェア・アプリケーショングループの共同による性能評価/アルゴリズム、設計改良、性能評価の繰り返し

調査研究の概要（システムソフトウェア・ライブラリ調査研究グループ）

取組概要

近年HPCアプリケーションで大きな課題となってきている「実行効率の低下」と「アプリケーション開発の困難さの増大」に対する新しいアプローチによる解決の可能性をアーキテクチャグループ・アプリケーショングループと協力しつつ検討する。

階層キャッシュ・コア間共有メモリに頼る「力任せな」並列化ではなく、データの局所性を最大限に活用するソフトウェア制御による高効率な並列化をアプリケーション開発の負担を軽減しつつ実現する方向性を調査する。

調査内容

DSL・フレームワークによる高効率コードの自動生成に重点

深層学習では高レベルDSL/フレームワークが一般的なアプローチになった：TensorFlow, PyTorch, JAX

アプリケーション開発者が個別アーキテクチャ向けのコーディング、チューニングする必要がなくなっている

(PFN は Chainer/Pytorch、CuPy 等で実績がある

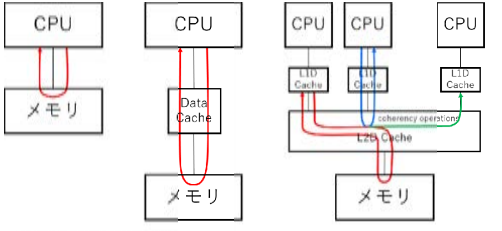
他のアプリケーションでも同様の方向での高い効率と開発の容易さを実現する可能性の評価に重点をおく。

スケジュール

初年度
DSL/フレームワーク/言語仕様検討・ プロトタイプ実装開発

2年度
アーキテクチャ・システムソフトウェア・アプリケーショングループの共同による性能評価/アルゴリズム、設計改良、性能評価の繰り返し

調査研究体制



階層キャッシュシステムではメモリアクセスもコア間通信もエネルギー消費大

調査研究の概要（アプリケーション調査研究グループ）

取組概要

重要なアプリケーション、アプリケーションカーネルの抽出、 次世代システム向けアルゴリズムの検討と性能評価を合わせて行う。伝統的なHPCアプリケーションだけでなく、AI応用、OSS等についても十分な検討を行う。

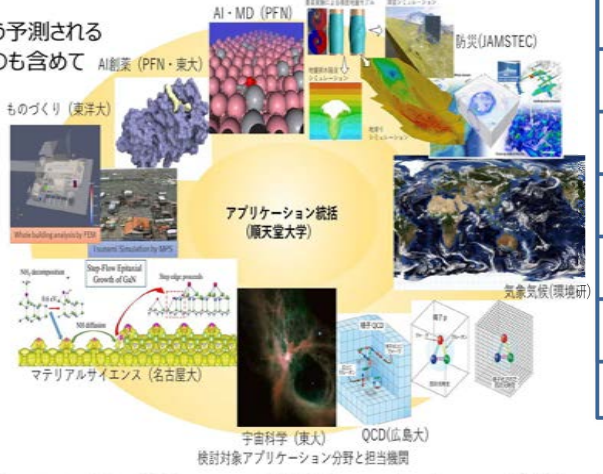
最重要なポイント:多くの分野において、実際に演算性能を有効に使えるようにアルゴリズム・アーキテクチャ双方の改良を進めること

分野抽出の理由:・これまで重要なアプリケーションであり、今後もそう予測される
・多様なアルゴリズムをカバーすることで、現在想定されていないものも含めて
十分に広いアプリケーションで高い実行効率を実現する

調査内容

アプリケーション分野とそれを担当する研究機関は以下の通りである

- 創薬/MD/AI シミュレーション応用 PFN
- ゲノム科学 東京大学
- 防災・減災(地震・津波) JAMSTEC
- 防災・減災(気象・気候) 環境研
- ものづくり(CAE) 東洋大学
- 物性/材料 名大
- 基礎科学(素核) 広島大学
- 基礎科学 (宇宙) 東大
- OSS/商用アプリケーション



調査研究体制

- 神戸大学
  - ◎順天堂大学
  - Preferred Networks
  - JAMSTEC
  - 環境研究所
  - 東洋大学
  - 名古屋大学
  - 広島大学
  - 東京大学

評価内容:抽出したアプリケーション/アルゴリズムに対して、演算性能、メモリアクセス性能については想定するアーキテクチャでの演算カーネルの性能評価、それに基づいたアーキテクチャ、アルゴリズム双方の改良を行う。ネットワークについても同様に、想定アーキテクチャのレイテンシも考慮した性能評価を行う。

スケジュール

初年度
　商用および独自コードのアルゴリズム調査・カーネル抽出、ヘテロジニアスアーキテクチャでの性能評価
2年度
　アーキテクチャ・システムソフトウェア・アプリケーショングループの共同による性能評価/アルゴリズム、設計改良、性能評価の繰り返し

FS の目標設定と神戸大学チーム目標のずれ

本調査研究では、ポスト「富岳」時代の次世代計算基盤の具体的な性能・機能等について、サイエンス・産業・社会のニーズを明確化し、それを実現可能なシステム等の選択肢を提案する

⇳

・分野自体の技術トレンド、
・半導体プロセス技術動向からポスト富岳時代に実現可能なシステム
がどのようなものかを検討し、評価対象アプリケーションによるシステム評価を行う

神戸大学チーム側では、技術トレンド、特に半導体プロセス技術動向を重視(他が重視してないというわけではない)

何故半導体技術トレンドを重視するか？

- 何ができるか(どう発展するか)を決める極めて大きな要因は半導体技術
- 計算機アーキテクチャの歴史と国策プロジェクトの歴史からこれを簡単に振り返っておく

計算機アーキテクチャの歴史(HPC向け)

- 1976 年まで: スカラー計算機。最後は CDC 7600
- 1976 年から 1992 年まで: ベクトル (共有メモリ並列含む) 計算機。Cray-1 から C-90 まで。
- 1993 年から 2008 年まで: (マルチコア含む) マイクロプロセッサの分散メモリ並列計算 Cray T3D から、、、Red Storm あたりまで。
- 2008 年から: GPU アーキテクチャのアクセラレータとマイクロプロセッサの組合せ: IBM RoardRunner (Cell なので GPU じゃないけど)

大体 15 年毎。GPU の次はまだでてきてないが、そろそろでてくる時期。

何故これらの変化が起こった (必要になった) か？

スカラー計算機→ベクトル計算機

- スカラー計算機の進歩: 増えるトランジスタを「1 つの演算器を速くする」に
- 主記憶は磁気コアで、半導体よりはるかに遅いことが想定=キャッシュが必要
- CDC 7600 (S. Cray の設計) あたりが最後

1970 年前後にデバイス技術の大きな変革が 2 つ起こった

- IC の開発で使えるゲート数が 1 演算器を超えて大きくなった
- 半導体メモリの開発でメモリが非常に高速になった

ベクトル計算機: 非常に高バンド幅の半導体メモリを活用して、少数の演算器を 100% 近くまで使い切るアーキテクチャ。
(今までの計算機の歴史のなかで唯一、メモリバンド幅が余るほどあった時代)

ベクトル計算機→マイクロプロセッサ超並列

- ベクトル計算機の進歩: 1 プロセッサのパイプライン数や主記憶を共有するプロセッサの数を増やす
- メモリもプロセッサも多数の IC から構成されて、沢山内部配線があるので、その間の配線も沢山あっても大丈夫、が前提
- 完全なパイプラインをもつプロセッサが 1 チップにはいると破綻。チップ間の配線をチップ内に比べて増やせなくなる。
- このため、メモリバンド幅の進歩が計算速度の進歩に追い付かなくなり、キャッシュベースのアーキテクチャに戻る。
- プロセッサチップ 1 つと少数のメモリチップで 1 ノードを構成し、その間はネットワークでつなぐ構成が有利になる。
- 初期の代表: Cray T3D。

マイクロプロセッサ超並列→GPU

- 1チップの中に沢山プロセッサがはいるようになる。
- これは実はシステムレベルでのベクトル並列プロセッサの限界と同じ。
- マルチ(メニー)コアプロセッサでは、キャッシュコヒーレンシを維持した階層キャッシュとオフチップの共有メモリ
- キャッシュコヒーレンシの維持のための電力が支配的になる。
- キャッシュコヒーレンシを緩和したり、捨てたりできるアーキテクチャが有利に。

GPU→???

- コヒーレンシを諦めても、階層キャッシュでチップ内部でデータを移動させること自体が性能の制限になってきている
- チップ内部であまりデータ移動させないアーキテクチャがあればそっちにいけるが。
- PEZY-SC(キャッシュもあるけど)、Sunway SC26010、MN-Core 等では演算器の物理的に近くに大容量のメモリを配置
- 但し、依然としてメインメモリはオフチップ DRAM。アプリケーションによってはこれでもよいが万能とはいいがたい(全てのアプリケーションで CPU や GPU よりましではある)

何故データ移動が制約になるか？

配線遅延・配線消費電力の問題
微細化しても、「単位長さ辺りのキャパシタンス」はかわらない、ところが、「単位長さあたりの抵抗」は配線幅の二乗に反比例して増える:

- **配線長がスケールしても配線遅延は「デザインルールに反比例して」増える**
- **配線長一定だと遅延は二乗で増える。電力は減らない**

データ移動距離と電力

- 1cm の配線は 2pF のキャパシタンスがある。これは微細化してもかわらない (線と平面の間のキャパシタンス) 電圧 1V だと、この線は 1pJ/bit を消費する。
- DDR5 メモリは 20pJ/bit くらい。10cm くらいの配線で電圧 1.3V とか。。
- LPDDR5 と GDDR6x: 10 pJ/bit くらい。配線が DDR5 のモジュールより短くて電圧も低い。
- HBMx: 3-4 pJ/bit。概ね 25mm くらいまで配線短くなっている。

3pJ/bit は 64 ビットワード移動に 0.2nJ なので、HBM でも 5G ワード移動すると 1J 消費する。B/F=4 の計算機作ると 10GF/W しかでない。0.1 にしても DRAM メモリだけで 400GF/W までしかいかないのでキャッシュの分とかいれると 100GF/W あたり。富岳の 5 倍にしかならない。

デバイス技術の重要な変化

- 集積度の向上: 量的だが質の変化を起こしてきた
  - １システムにパイプライン化した演算器
  - １チップにパイプライン化した演算器
  - １チップに「非常に多数」の演算器
- メモリバンド幅の変化: 磁気コアから半導体への１度だけ

ベクトルアーキテクチャは、半導体メモリの開発でメモリバンド幅が飛躍的に向上してから、１チップに演算器がはいって相対的にメモリバンド幅が上がらなくなるまでの間だけ成立したアーキテクチャ。

少し違う観点から

	システム	チップ
1 演算器	スカラー/ベクトル	スカラー
少数演算器	共有メモリスカラー/ベクトル	共有メモリ CPU
多数演算器	分散メモリスカラー	共有メモリ GPU

システムの進化をチップがなぞるなら、チップ内分散メモリなるものがでてきていてもいい。
現時点の現実はそうなっていない。主記憶がオフチップ DRAM である限り (HBM のような「2.5D」になっても) 分散しても意味がない。
意味があるためには、 DRAM 主記憶を多数に分割して演算器の近くにもってくる必要がある。

DRAM と演算器を近くに

- アイディアは少なくとも 1990 年代にさかのぼる。Processing-in-Memory (Kogge 他)
- ここまでの議論からいえること: 今までは時期尚早だった (オフチップ DRAM+階層メモリアーキテクチャでなんとかなった)
- DRAM プロセスで演算器を作る話は SK Hynix, Samsung 等がやっているが、演算器性能が低くなりすぎる。
- ロジックダイと DRAM の貼り合わせ+複数 DRAM の貼り合わせによる 3 次元化が必須。

3D積層技術

色々なところで実際に使われるようになっている

- DRAM 同士: HBM で 12-16 層。現在はマイクロバンプ (この辺あとでもうちょっと詳しく)
- CMOS センサー: センサーとロジックを接続。ソニーでは「Cu-Cu 接合」技術を実用化
- ロジックとロジック: TSMC で実現。マイクロバンプ、ハイブリッドボンディング (これも後述) の両方。 AMD の CPU と GPU が代表例。

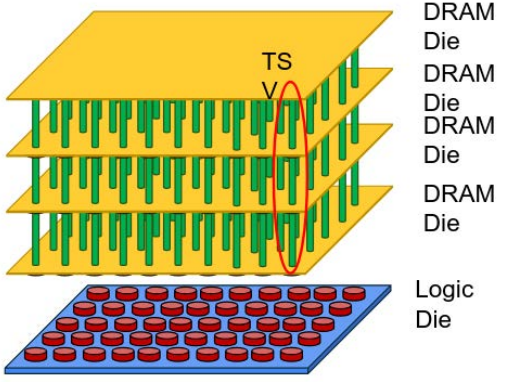
DRAM とロジック: 利用可能になってきている。

システムとチップの対応

	システム	チップ
1 演算器	スカラー/ベクトル	スカラー
少数演算器	共有メモリスカラー/ベクトル	共有メモリ CPU
多数演算器	分散メモリスカラー	~~共有メモリ~~ GPU 3D 積層による分散メモリ

近い将来に移行が起こる、という想定で、神戸大学チームでは 3D 積層メモリによるチップ内分散メモリアーキテクチャの可能性を検討した。

3D積層



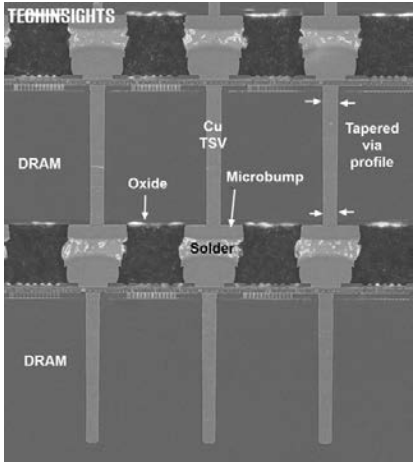
複数の DRAM ダイとロジックダイを非常に多数の TSV とパッドで接続
現行技術の例:　800mmsq,　4stack:　総容量 400Gbit これを 64Mbit のブロック 6400 個 (1 枚 1600 個) で構成。それぞれが 64 ビット幅の I/O (合計 400k)。500MHz の SDR DRAM 構成で 200Tbit/s = 25TB/s

1 ダイに 64Mbit の SDRAM ブロックを 1600 個集積。(例えば 40 x 40 に配置) これが 4 枚重ねられる
1 ブロックあたり 1000-2000 本の TSV が必要。 $4\mu\mathrm{m}$ ピッチなら面積の 5% 程度。
電力は 100-200W。HBM なら 25TB/s は 1kW を大きく超える。

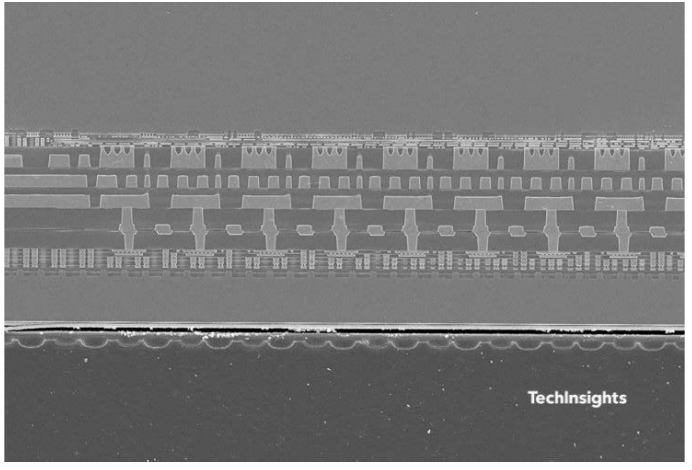
接合技術の現状

基礎となる技術:

- **接合: マイクロバンプ or ハイブリッドボンディング**
- **TSV: TSV-middle 等のファインピッチ技術**



マイクロバンプ



ハイブリッドボンディング

マイクロバンプ

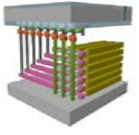
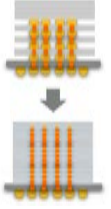
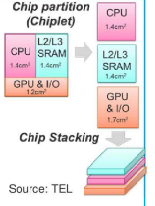
- 基本的にハンダボールや C4 バンプの延長の技術
- 現状で利用可能なピッチ: $\sim 40\mu\mathrm{m}$、Intel MAX GPU では $37\mu\mathrm{m}$。
- 将来的には $10\mu\mathrm{m}$ を切るといわれている。
- HBM3e までの各社の HBM で採用
- 現状では $1\mathrm{mm}^2$ あたり 600 本程度なので、 $800\mathrm{mm}^2$ のチップだと全面において 50 万本。 50TB/s 程度までなら実現可能
- 複数積層だと TSV があるエリアを集中させたいのでもう 1 桁程度高密度にしたい
- ダイ間を樹脂で充填するので、熱抵抗が大きい。(現在の HBM で大きな問題)
- CoW/CoC なので、イールドの問題は少ないが加工費用が高価

ハイブリッドボンディング

- ダイ表面の Cu パッドを直接接合。
- 最初の実用化はソニーの センサー
- 現状で多くのベンダで利用可能なピッチ: $\sim 5\mu\mathrm{m}$
- 将来的には $1\mu\mathrm{m}$ を切るといわれている。
- 現状では $1\mathrm{mm}^2$ あたり 4 万本程度なので、 $800\mathrm{mm}^2$ のチップだとエリアの 2.5% でも 100 万本。 1Gbps で 1Pbit/s、つまり概ね 100TB/s が実現可能。
- ダイ間に樹脂がはいらないので、熱抵抗が小さい。
- Wafer-on-Wafer なので加工費用は安価 (BSPDN でも同様な技術が使われる) が、イールドの問題はある。1Gbit DRAM のイールドは 90-95% といわれており、それを前提にした冗長設計が必須になる。

技術の現状

貼り合わせ接合適用例

Application	CIS	NAND	DRAM	Logic		Advanced Package	
Stacking Device	BSI Pixel + (Peripheral) + Logic	3D NAND Cell + Peripheral	3D DRAM Cell + Peripheral	Backside PDN Logic + Bare Si	Sequential CFET Logic + Logic	HBM DRAM ( ) ⋮ DRAM ( ) + Logic	Disaggregation / Chiplets A B C
Bonding	W-W (Ox Fusion) Cu Hybrid	W-W Cu Hybrid	W-W Cu Hybrid	W-W Ox Fusion	W-W Ox Fusion	D-W u Bump → Cu Hybrid	D-W / D-D Cu Hybrid
3D I/O Pitch	3um →1um	1um →0.5um	1um →0.5um	Sub um (nTSV)	Sub um (nTSV)	40um →25um	10um →1um
Structure			 Source: TEL 3D DRAM structure				
Status	HVM	R&D~HVM	R&D	R&D	R&D	R&D	R&D~HVM

次世代デバイスへ向けて、貼り合わせ接合技術の導入が拡大

東京エレクトロンの IR 資料 2025/2 サブミクロンピッチの WoW、TSV は実用範囲

冷却の問題

以下のような主張をみることがある

- HBM ですでにそうであるように、3D 積層構造では冷却が最大の問題
- 横につむのをやめて、縦構造(細いメモリダイを一杯並べる)にしないといけない

実際には

- 現状の HBM で熱抵抗が大きい層はマイクロバンプ接合を樹脂で埋めているところ。30$\mu\mathrm{m}$ 程度あり熱伝導率が小さい。ハイブリッドボンディングではこの層がないため、1 層あたりの熱抵抗が 1 桁下がる
- このため、 $1\mathrm{W}/\mathrm{mm}^2$ 程度の発熱でも温度上昇は 1K/層 程度。16 層程度までは大きな問題ではない。

DRAM 3D実装技術の現状

	HBM	Custom HBM	Bufferless HBM	Customized Bufferless	Full Custom CoW(etc)	Full Custom WoW
信号数	2K	2K	2K	4-8K?	> 10K	> 100K
電力 pj/bit	3-4	2-3?	2-3?	1-2?	~ 1	0.25-1
ベンダ	A/B/C	A/B	B	F	D/E/F	D/E/F
時期	now	2026?	2027?	2026?	now?	now?
NRE	-	非常に高い	非常に高い	高い	高い	安い可能性
量産コスト	高	高	高	高	中	低
容量	大	大	大	中	小 - 中	小 - 中
ロジック	最先端	最先端	最先端	微妙	古い	古い

- HBM は世代が進んでも大きくアクセスエネルギーが下がらない。このため、多様なソリューションが検討されている。
- 原理的にアクセスエネルギーを最小にできるのは、信号数を増やせる WoW
- 各社 WoW による 3D 積層に積極的になってきた。(Major 3 社以外)
- 0.25pJ/bit だと 100TB/s でも 200W。今の HBM3 より実質低い電力でメモリバンド幅を 20 倍にできる。

アクセラレータの予測と比較

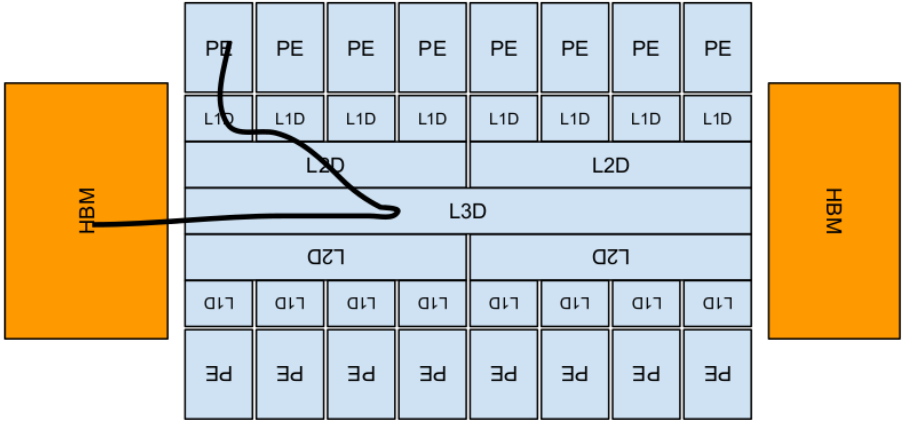
予測	NVIDIA	AMD	MN-Core
製品発表年	2028	2028	2028
プロセス (量産開始年)	A16(26)	A16(26)	—
演算性能@FP16(TF)	5480	3289	—
演算性能@ FP64(TF)	98	207	—
FP16/FP64	55	16	16
周波数 (MHz)	2000	2625	—
TDP(W)	1500	1000	1000
電力効率 GF/W@FP64	65.3	207	—
Die Size(mm2)	1600	1100(x2)	—
DRAM サイズ (GB)	384	384	384
DRAM バンド幅 (TB/s)	(24)	(12.8)	100
電力効率 GB/J@FP64	16	12.8	100

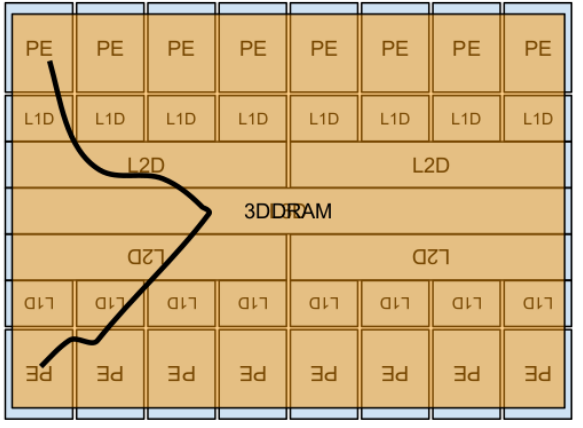
他ベンダの DRAM バンド幅と電力は、HBM (Bufferless でも) と現在の階層キャッシュアーキテクチャを継続するなら同時には実現困難。MN-Core 後継アーキテクチャは他社と比べて FP64 電力性能で 2 倍以上、電力あたりメモリバンド幅で 5 倍以上を実現可能。

他ベンダは 3DDRAM 使っても高いメモリバンド幅実現は困難

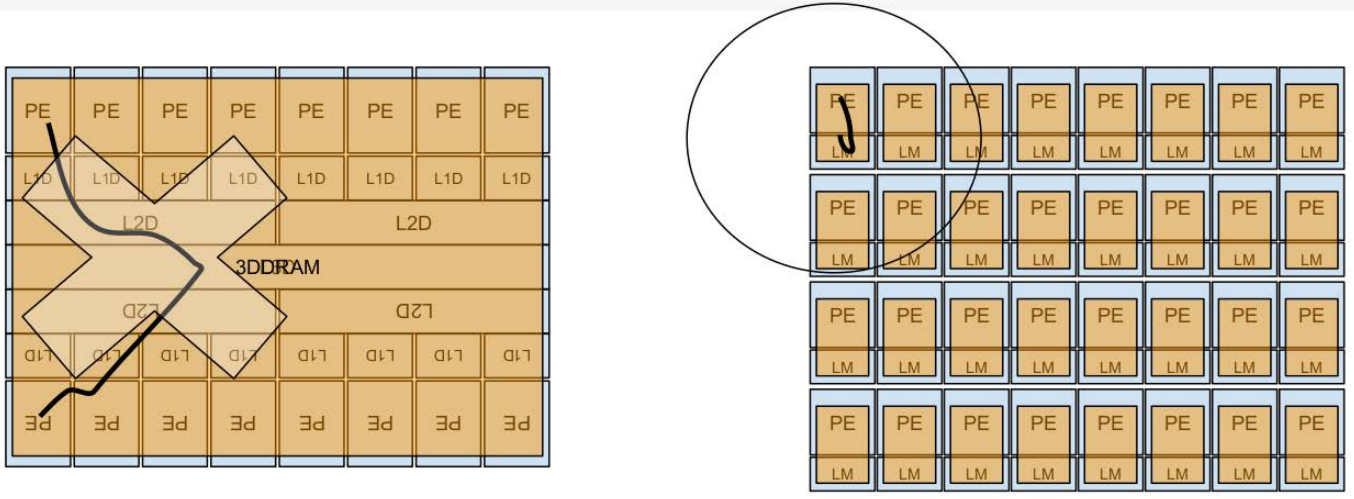
共有メモリでは 3D にしても電力性能上がらない

理由: 階層キャッシュによる共有メモリアーキテクチャはそれ自体がメモリアクセスのボトルネックを持つ。3D でもデータ移動距離があんまりかわらない。



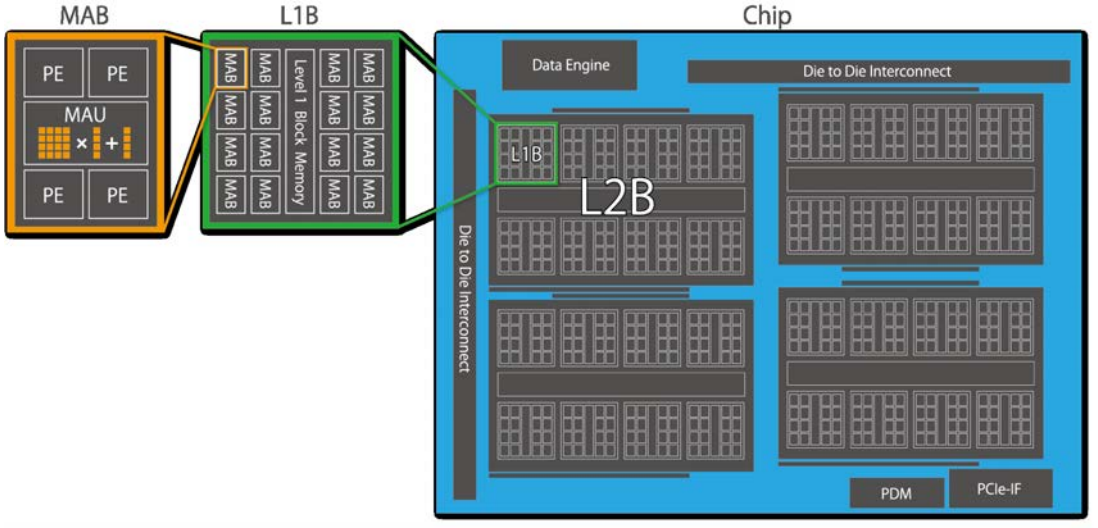


共有メモリと分散メモリの違い



物理的なデータ移動距離を 1 桁以上短くできる。
MN-Core は分散メモリアーキテクチャ。DRAM アクセスコスト小さい。
GPGPU は基本的に共有メモリ。3D 構造にしても DRAM アクセスコストは大きくは下がらない。

MN-Core 階層構造



- 従来は複数の L2B がオフチップ DRAM 1 ユニットに接続
- 提案アーキテクチャでは各 PE がそれぞれ自分にローカルな DRAM に接続。いわゆるスクラッチパッドメモリに DRAM を使い、アクセラレータの主要なメモリがそれになる。

メモリアクセスのエネルギーの理論限界

- DRAM セルの容量：$10 \sim 20$ fF $\rightarrow$ アクセスの消費電力は 10fJ 程度
- 1 ゲートの消費電力: 1 ゲートあたり $0.1 \sim 1$ fJ 程度
- 配線の寄生容量: 1cm あたり 2pF: アクセスの消費電力は 1pJ 程度。現状では 10pJ に近い。 HBMx では 4-7pJ。

つまり:

- 現在の DRAM およびロジック製造技術でも、メモリアクセスエネルギーはもう 2-3 桁下げる余地がある (配線長を 1mm 以下にすれば)
- Memory Wall(Wulf and McKee 1995, Wilkes 1995) は 3 次元実装と分散メモリアーキテクチャで本当に解消する。BF=4 の計算機も将来にわたって可能。

アーキテクチャの実現可能性

- 基本アーキテクチャは MN-Core の延長であり、実績もあるもので実現に問題ない。
- 技術的に未知なのは DRAM 3D 積層の実現性 (良品率、耐久性等) であるが、本 FS 内では試作による評価は認められなかったので行っていない。バンド幅、電力についてはシミュレーションで詳細な評価を行った。
- 3D 積層のキー技術であるハイブリッドボンディングについては、AMD は TSMC のウェファ + TSMC のウェファでは実用になっている (MI300X 以降)
- DRAM 積層は現状で各社 DRAM + Logic は実績あり、 DRAM 複数積層は開発中。ポスト富岳の実現時期には十分実用になっていると思われる。
- (中国はもうちょっと実装・量産が進んでいる模様)

アーキテクチャ調査研究で得られた特に重要な知見

- HBMx を使っても、「プロセッサチップの横に共有メモリを提供する DRAM がある階層キャッシュアーキテクチャ」自体が電力性能向上への足枷になっている。
- DRAM 3D 積層は、製造技術の側から電力性能向上を実現する解を与える。これの有効利用には、物理的な共有メモリから分散メモリへのアーキテクチャの変革が必須になる。
- MN-Core アーキテクチャはこの変革に容易に対応できる。GPU、他の AI むけプロセッサは未知数である。

システムソフトウェア・ライブラリ調査研究の概要

当初の調査内容

- DSL・フレームワークによる高効率コードの自動生成に重点をおく。
- 深層学習は Pytorch, JAX 等からのコード生成。
- 他のアプリケーションでも同様の方向での高い効率と開発の容易さを実現する可能性の評価に重点をおく。

実際の調査研究

- 深層学習については当初の方針通り
- 一般の HPC アプリケーションについては、
  - DSL が適しているもの
  - 行列演算ライブラリ、FFT ライブラリ等のライブラリ利用が中心になるもの
  - どちらでもなく、汎用言語でのアプリケーションカーネルの記述が必要なもの

  に対してそれぞれ対応する。
- 3D DRAM による分散メモリアーキテクチャをターゲットにする。

このため、
(1)DSL 性能評価、 (2) ライブラリ性能評価、 (3) 汎用言語の設計・評価
をそれぞれ進める。

アプリケーションとアプローチの対応

アプリケーション	タイプ	
創薬と深層学習応用	粒子・深層学習	DSL
ゲノム科学	ツリー探索	汎用環境
地震と構造物	FEM(EbE)	汎用環境
気象・気候	規則格子差分	汎用環境
ものづくり	疎/密行列	ライブラリ
マテリアルサイエンス	密行列	ライブラリ
素粒子・原子核	規則格子差分	汎用環境
宇宙・惑星科学	粒子	DSL

- 規則格子向けの DSL を当初検討していたが、OpenACC ライクな言語環境で性能的には十分であり、記述の柔軟性が高いと判断した。
- ツリー探索や EbE 法は DSL よりも低レベルの記述が現実的である

システムソフトウェア・ライブラリ調査研究の主要な成果 (1) DSL

- 粒子法については、オフロードモデルを想定した。これは現在 GPU を使っているコードのほとんどが、粒子間相互作用計算のみを GPU で行うオフロードモデルであること、またシステム提案が CPU 側の演算能力、バンド幅が十分あり、オフロードモデルで高い効率を期待できることによる。(性能評価はアプリケーションのところで)
- 粒子間相互作用について、高レベルの記述から最適化されたカーネルのアセンブリコード (理論限界の性能がでる) を生成するコンパイラと、ホストとのデータ通信等の関数を自動生成する DSL を開発した。これは理研 RCCS 粒子系シミュレータ開発チームで開発していた FDPS/PIKG がベースである。
- 深層学習については、PFN 社内で開発してきた PyTorch からのコード生成で十分な性能がでることを確認できた。

FDPS/PIKG

粒子法については、理研-神戸大学で開発してきたフレームワーク FDPS の相互作用計算カーネルコンパイラ PIKG を、MN-Core の最適化したコードを生成する。

```
rij = EPI.pos - EPJ.pos
r2 = rij * rij + eps2
r_inv  = rsqrt(r2)
r2_inv = r_inv * r_inv
mr_inv  = EPJ.mass * r_inv
mr3_inv = r2_inv * mr_inv
FORCE.acc -= mr3_inv * rij
FORCE.pot -= mr_inv
```

左のコードから最適化した命令列を生成する。MN-Core での命令数 (実行時間そのもの) は反復あたり 23 命令であり、人間がアセンブリ言語を限界まで最適化したものと同じ命令数となった。
カーネル部分だけの実行効率はベクトル性能ピークに対して 65.2% (行列乗算ピーク性能に対して 16.3%) となる。

コード全体でも 20% を超える実行効率が得られた。

システムソフトウェア・ライブラリ調査研究の主要な成果 (2) ライブラリ

- 基本的な数値ライブラリとして、密行列向けの LAPACK と、FFT ライブラリを開発中である。どちらも、PE/MAB 単位の中では 80% から 95% 程度の実行効率が得られている。
- 特に、FFT でも、行列乗算ユニットを有効に使って 2 段 (倍精度) や 3 段 (単精度) のバタフライ演算を一度に行なうことで、ベクトル演算の場合よりも高速化できるため、ベクトル演算で性能が落ちるという欠点が大きな問題になりにくいこと、むしろ、ローカルメモリのバンド幅が高いため、GPU や CPU よりも高い実行効率が得られることがわかった。
- 最終報告では PE 間通信も考慮した性能推定を与える。多くの場合に通信は隠蔽でき、また提案システムは B/F=0.2 と高い DRAM バンド幅と大きなオンチップメモリを持つため、大きな FFT でも高い実行効率が得られることがわかった。

システムソフトウェア・ライブラリ調査研究の主要な成果 (3) 汎用言語環境

- Cuda/OpenCL 風言語、OpenACC 風コンパイラ指示によるアクセラレータへのオフロードの両方の検討を進めてきた。
- それぞれについて、これまでの検討で得られた知見をまとめる。

Cuda/OpenCL 風環境

どのような実装がよいかの比較検討のため、複数の実装を試みている。どれも LLVM ベースである。

- 会津大学によるもの
- 神戸大学の作成したものおよびそれをベースに外部委託開発したもの
- PFN 社内で独自開発したもの

これらの比較検討により多くの重要な知見が得られた。

重要な知見(1)

LLVM-IR から MN-Core アセンブリ言語 (vsm) への変換で十分に最適化されたコードをだすことはそれほど難しくない。

- **MN-Core では固定長ベクトル命令を採用することで、命令スケジューリングを不要にしている (直前の命令の結果を使える)。LLVM は固定長ベクトル命令 (SIMD 命令) に対応しているので命令生成できる。**
- **条件分岐はマスク実行に変換する必要があるが、この機能を LLVM がもっている (SIMD 命令用のマスク付きコード生成ができる)。**
- **現状ではアドレス計算はコンパイル時に解決され、(ループは展開される) ロード・ストア命令は次に述べるように消去されるため、生成される命令は基本的に演算命令だけになる。(袖交換等は別)**
- **このため、生成されるカーネルコードの実行効率は、ベクトル性能に対して最低でも 50%、典型的には 70% 前後になる。**

重要な知見(2)

通常のコンパイラバックエンドにはない特異な処理が必要である。

- **LLVM はロード・ストアアーキテクチャを仮定しているが、MN-Core はローカルメモリもオペランドにとれるため、ロード・ストア命令は不要である。このため、オペランドの書換えによってロード・ストアを消去する必要がある。**
- **MN-Core は複数のレジスタファイルとローカルメモリが命令セットから見える。このため、変数を適切に複数のユニットにわりつけてハザードを解決する必要がある。これはグラフ彩色問題に帰着でき、実用的な時間で十分良い解を求められる。**

重要な知見(3)

大規模 SIMD アーキテクチャー般の制約に加えて、現在の MN-Core の実装の持つ制約がある。特に

- 現在のところ実行時にループ反復回数を指定できない
- while ループを実装できない

これらは、機械学習では学習でも推論でも基本的に不要なため実装されておらず、現在の PFN 内製の命令生成系は ONNX 等のネットワーク記述からループが完全にアンロールされた命令列を生成する。
ポスト富岳向けシステムでは簡単なシーケンサを実装して実行時に回数を指定するループや while ループを可能にする。
MN-Core の前身である GRAPE-DR ではループ機能がオンボード FPGA によって実装されており、同様な実装を想定している。

While 等の実装

MNCL の関数内で

```
private double a = (PEの番号);
while(a > 0.1)  a *= 0.5;
```

みたいなコードがあると

```
private double a = (PEの番号);
global bool not_finished_g;
private bool not_finished = (a>0.1);
while( not_finished_g |= not_finished ){
   not_finished = (a>0.1);
   if (not_finished) a *= 0.5;
}
```

こんな感じに変換されればよい。

シーケンサ実装の今後の方針

- 現在開発中の推論向けプロセッサでは、命令シーケンサを外付け FPGA で実装予定。
- これは、複雑なループが必要だからというよりはコストダウンのため。ホストでの命令生成・転送のために必要なホストの能力、PCIe バンド幅を削減する。
- 実装の詳細は省略するが、詳しい情報があるものでもっとも近いのは FPS AP-120B/164 のシーケンサ部分。AP-120B では命令は inc, dec, com (アドレスレジスタへのコピー), clr, mov, add, sub, and, or, eqv, br, beq, bne, bge, bgt の 15 個。

OpenACC 風環境

- OpenCL に変換するものと、直接 vsm を生成するものの 2 バージョンを開発・評価中である。
- 後者は PFN 標準の API ではなく低レベル API を呼ぶため、低オーバーヘッドの実行が可能である。最終報告ではいくつかのベンチマークの結果を示す。

OpenACC 風環境

PFN 開発版

```
 for(step=0;step<nstep;step++){
#pragma goose parallel for loopcounter(i,j,k) result(c) ni_loop(NI) nj_loop(NJ) \
 nk_loop(NK) dflag(a:(step==0)?1:0) dflag_result(c:0)
   for(i=0;i<ni;i++){
     for(j=0;j<nj;j++){
       for(k=0;k<nk;k++) {
         c[i][j][k] = 4*a[i][j][k]-2*a[i-1][j][k]-3*a[i+1][j][k]-5*a[i][j-1][k]
                        -6*a[i][j+1][k]-7*a[i][j][k-1]-8*a[i][j][k+1];
         if((i==0)||(i==NI_M1)||(j==0)||(j==NJ_M1)||(k==0)||(k==NK_M1)) c[i][j][k] = 3.0;
       }
     }
   }
#pragma goose parallel for loopcounter(i,j,k) result(c) ni_loop(NI) nj_loop(NJ) \
 nk_loop(NK) dflag(c:0) dflag_result(a:(step==(nstep-1))?1:0)
   for(i=0;i<ni;i++){
     for(j=0;j<nj;j++){
       for(k=0;k<nk;k++) {
         a[i][j][k] = c[i][j][k];
       }
     }
   }
 }
```

OpenACC 風環境

```
#pragma acc enter data copyin(array[0:128][0:128][0:128])

...

#pragma acc parallel present(array[0:128][0:128][0:128]) ¥
shadow(array[1:1][1:1][1:1])
for(int count=0; count<1000; count++) {

#pragma acc loop collapse(3)
    for(int i=1; i<128-1; i++){
        for(int j=1; j<128-1; j++){
            for(int k=1; k<128-1; k++){
                array[i][j][k] =
                    c1*(array[i-1][j][k]+array[i+1][j][k]
                    +array[i][j-1][k]+array[i][j+1][k]
                    +array[i][j][k-1]+array[i][j][k+1]);
    } } }

#pragma acc reflect(array)

} // count の for 文(=オフロード適用範囲)の終了

...

#pragma acc exit data copyout(array[0:128][0:128][0:128])
```

```
__kernel void main_kernel0(
__global double* _arg_array,
__global double _arg_c1) {

    // 1PEあたりの袖領域を含めたサイズ
    __private double array[10][10][6];
    bm2 double array_bm2[10 * 4][10 * 4][6 * 32];
    bm1 double array_bm1[10 * 4][10 * 4][6 * 4];
    __private double c1;

    // distribute、collect関数はデフォルト分割に対応している前提
    distribute(array_bm1, array_bm2, _arg_array, 10*10*6);
    distribute_pe(array, array_bm1, 10*10*6);
    broadcast(&c1, &_arg_c1, 1);

    for(int count=0; count<1000; count++){

        for(int i = 1; i < 9; (i)++) {
            for(int j = 1; j < 9; (j)++) {
                for(int k = 1; k < 5; (k)++) {
                    array[i][j][k] =
                        c1*(array[i-1][j][k]+array[i+1][j][k]
                        +array[i][j-1][k]+array[i][j+1][k]
                        +array[i][j][k-1]+array[i][j][k+1]);
                }
            }
        }

        // 袖通信(関数はデフォルト分割に対応している前提)
        int array_size[3] = {10, 10, 6};
        int array_shadow[3][2] = {{1, 1}, {1, 1}, {1, 1}};
        halo_comm(array, sizeof(double), 3, array_size, array_shadow);
    }

    collect_pe(array_bm1, array, 10*10*6);
    collect(_arg_array, array_bm2, array_bm1, 10*10*6);
}
```

アプリケーション調査研究の概要

当初の調査内容

- 重要なアプリケーション、アプリケーションカーネルの抽出、 次世代システム向けアルゴリズムの検討と性能評価を合わせて行う。伝統的な HPC アプリケーションだけでなく、AI 応用、OSS 等についても十分な検討を行う。
- 抽出したアプリケーション/アルゴリズムに対して、演算性能、メモリアクセス性能については想定するアーキテクチャでの演算カーネルの性能評価、それに基づいたアーキテクチャ、アルゴリズム双方の改良を行う。ネットワークについても同様に、想定アーキテクチャのレイテンシも考慮した性能評価を行う。

アプリケーションまとめ

アプリケーション	性能
創薬と深層学習応用	効率 30% (LLM 推論。学習なら 60-70%)
ゲノム科学	CPU のみの 10-20 倍
地震と構造物	効率 20%
気象・気候	効率 12%
ものづくり	効率 10%
マテリアルサイエンス	効率 50%
素粒子・原子核	効率 15%
宇宙・惑星科学	効率 10%

- この中で性能が主記憶リミットなのは LLM 推論、ものづくりの 2 つだけ。(主記憶バンド幅を増やしたからではある)
- (これも極端に主記憶バンド幅を増やしたからだが) 多くのアプリケーションで演算器リミットな性能になっており、行列演算ユニットを小さくしてベクトル性能をあげるほうが性能向上につながる。10-20% を 30-60% 程度にはできるが、ピーク性能や電力性能自体は低下する。

アプリケーションポーティングの大変さと実行効率

既に述べたように、アプリケーションポーティングの内容はアプリケーションのタイプ等で 3 つに分かれる。

1. DSL 型: 深層学習での PyTorch のようにアクセラレータで実行されるコード全体 (といっても行列行列積が主体) を DSL で書けるもの、あるいは粒子法のように、計算量のほとんどを占める部分 (粒子法では相互作用計算) をオフロードするもの。
2. ライブラリ型: DSL 型と同様に計算量の大半を占めるところをオフロードするが、その部分が行列の対角化や分解、FFT、疎行列ベクトル積等のライブラリコールで実現できるもの。
3. それ以外: さらに規則格子差分法とそれ以外に分かれる。

実行効率とポーティングの大変さ

1. DSL 型: 効率は高く、ポーティングはそれほど大変ではない。
2. ライブラリ型: 効率は高く、ポーティングはそれほど大変ではない。
3. それ以外:
    - 規則格子型: 効率は高く、ポーティングはそこそこ大変
    - それ以外: 効率も大変さもアプリケーションによる。

DSL型・ライブラリ型

- 深層学習では、PyTorch で書かれたものが CPU でも GPU でも他の AI アクセラレータでも動く。アプリケーションを書く側からみるとポーティングは特に大変ではない (各アーキテクチャ向けのコンパイラ開発は大変)
- 粒子のオフロードでも、DSL の相互作用計算部分の「コンパイラ」があればよい。実際に FDPS/PIKG では GPU と MN-Core で同一のコードが動作する。
- ライブラリ型も同様で、場合によってはライブラリのインターフェース API が違うがその書換え程度で動作する。

このようなタイプはかなり多いと考えられる。

それ以外:規則格子差分法

- 規則格子差分法は、工学応用はあまり多くないかもしれない (が、FDTD 法等重要なアプリケーションもある) が、素粒子、宇宙、気象の大規模シミュレーションでは広く使われている。
- 細かいバリエーションがあるので DSL を定義するよりも OpenACC (ないし XcalableMP/HPF) のような配列オリエンテッドな並列言語での記述が適切な抽象度であると考えられる。

それ以外: それ以外

- それ以外では、間接参照が発生するものが多い。が、間接参照を PE の中に閉じられる、あるいは前処理で対応可能なものが多い。以下にその例をあげる。
- ゲノム解析では、ツリー構造を辿るため、そのままのアルゴリズムでは反復回数が不定で実行時に判定が必要である。但し、通信リミットであることがわかっていると、固定回数反復で、終了していないものはホストで処理で十分であり、コードは PE 内で閉じるため特に困難はない。
- EbE 法では、計算実行部分はコンパイラで問題なくコード生成可能だが、間接参照部分には前処理が必要になる。前処理によって通信コードを生成すれば問題なく実行できる。

まとめ(1)

- 神戸大学チームでは、プロセッサダイと DRAM ダイを 3 次元積層することで可能になる、チップ内分散メモリアーキテクチャについて、その実現性、性能、プログラミング環境、アプリケーション性能を検討した。
- これは、歴史的に 3 次元積層への移行が起こる時期だと予想されるからである。
- 技術的には 3 次元積層は成熟してきており、実現可能である。メモリアクセスの電力を HBM に比べてバンド幅あたりで 1 桁以上下げることが可能であり、非常に高バンド幅の主記憶を低電力で実現できる。
- これを MN-Core アーキテクチャと組み合わせることで、電力性能が高く、B/F 値も非常に高いシステムが実現できる。
- アプリケーションポーティングの困難さは GPU へのオフロードと極端に違うわけではない。OpenACC のような環境も利用可能になる。

まとめ(2)

- アプリケーション実行効率は、DRAM バンド幅やローカルの SRAM のバンド幅が非常に高く、SRAM のレイテンシが小さく、また汎用 CPU でありがちなレジスタ数の制限等がないため高くなる傾向がある。