次世代計算基盤に係る調査研究
チーム
成果報告書

牧野 淳一郎
神戸大学システム調査研究チーム

2026 年 3 月 24 日

本報告書は、文部科学省の令和４～６年度科学技術試験研究委託事業による委託業務として、国立大学法人 神戸大学が実施した「次世代計算基盤に係る調査研究（システム調査研究チーム）」の成果を取りまとめたものです。

目次

著者一覧

牧野 淳一郎	国立大学法人 神戸大学	第 1 章
		第 2.1 節
		第 4.2 節
		第 5 章
塩谷 亮太	国立大学法人 東京大学	第 2.2 節
小泉 透	国立大学法人 名古屋工業大学	第 2.3 節
鯉渕 道紘	大学共同利用機関法人 情報・システム研究機構 国立情報学研究所	第 2.4 節
平澤 将一	大学共同利用機関法人 情報・システム研究機構 国立情報学研究所	第 2.4 節
中里 直人	公立大学法人 会津大学	第 3.1 節
綱島 隆太	国立大学法人 神戸大学	第 3.2 節
岩澤 全規	独立行政法人 国立高等専門学校機構 松江工業高等専門学校	第 3.3 節
遠藤 克浩	国立研究開発法人 産業技術総合研究所	第 3.4 節
姫野 龍太郎	学校法人 順天堂	第 4.1 節
鎌谷 洋一郎	国立大学法人 東京大学	第 4.3 節
堀 宗朗	国立研究開発法人 海洋研究開発機構	第 4.4 節
八代 尚	国立研究開発法人 国立環境研究所	第 4.5 節
塩谷 隆二	学校法人 東洋大学	第 4.6 節
河合 浩志	学校法人 東洋大学	第 4.6 節
荻野 正雄	学校法人 大同大学	第 4.6 節
押山 淳	国立大学法人 東海国立大学機構 名古屋大学	第 4.7 節
岩田 潤一	株式会社 Quemix	第 4.7 節
石川 健一	国立大学法人 広島大学	第 4.8 節
藤井 通子	国立大学法人 東京大学	第 4.9 節

第 1 章　はじめに

牧野 淳一郎 [国立大学法人 神戸大学]

本報告書は、2022 年度から 2024 年度に行われた「次世代計算基盤に係る調査研究」のうち神戸大学チームによるシステム調査研究の成果を取りまとめるものである。

「次世代計算基盤に係る調査研究」は 2020 年に完成したスーパーコンピューター「富岳」のフィージビリティスタディとして行われたものである。

以下で、まず「富岳」およびその一つ前の世代の「京」について開発前の検討過程と完成したシステムのアーキテクチャ、主要な成果と課題について振り返り、ポスト富岳 FS の課題についてまとめる。

1.1　「京」

日本において、ナショナル・フラグシップ・スーパーコンピュータとして開発されたシステムは、2011 年に完成した「京」が初めてであった。「京」の開発は、「次世代スーパーコンピュータ」開発プロジェクトとして 2006 年度から 2011 年度までの 6 年間で行われた。そのための事前の検討としては、2004 年 8 月から 2006 年 7 月までに計算科学技術推進WGがおかれた他、2006 年 4 月から 8 月の短期間で、『次世代スーパーコンピュータ：概念構築に関する共同研究』が行われた。[1)] その共同研究では、目標性能 10 PF はあらかじめ設定され、その制限の下で共同研究に参加した 6 組織（東大、筑波大、国立天文台、富士通、日立、NEC）が提案するアーキテクチャに対して理研が提示する 21 のターゲット・アプリケーションの性能を評価する、という形式をとった。筑波大、富士通、日立、NEC の 4 提案は汎用システムで、消費電力は 10 MW から 30 MW、東大と国立天文台の 2 提案はアクセラレータであり、消費電力は配布資料にはどちらも「-10 MW」と記載される一方「※平成 24 年 6 月公開時の注意書き 消費電力については、10 MW 以下、10–20 MW、20–30 MW の範囲でまとめたもの。提案は、ホスト部を除いて、1.7 MW であった。」「※平成 24 年 6 月公開時の注意書き 消費電力については、10 MW 以下、10–20 MW、20–30 MW の範囲でまとめたもの。提案は、ホスト部を除いて、0.68 MW（案 1)、0.88 MW（案 2）であった」と「公開時に」注意書きが付記される、というものであった。この結果、最終的に NEC+日立案、富士通案の 2 つが検討され、「2 者のシステム構成により、目標性能達成の見込みが確認できたため、アク

1) 情報科学技術委員会 次世代スーパーコンピュータ概念設計評価作業部会（第 2 回）配付資料 https://www.mext.go.jp/b_menu/shingi/gijyutu/gijyutu2/022/shiryo/1321896.htm 資料 2-2 次世代スーパーコンピュータの概念設計について（続き）https://www.mext.go.jp/b_menu/shingi/gijyutu/gijyutu2/022/shiryo/__icsFiles/afieldfile/2012/06/11/1321896_4.pdf

セラレータの採用は考慮しない。」となり、NEC+日立案のベクトルプロセッサと富士通案のスカラープロセッサのヘテロジニアスシステムとして「京」の開発はスタートした。

「京」では、開発 4 年目の 2009 年 5 月に、NEC が撤退する、という国家プロジェクトとしてはかなり珍しい事態が発生し、計画の見直しが行われた。とは言え、元々 NEC が担当するベクトル部は全体システムの性能への寄与が必ずしも大きくはなく、おそらく製造コストあたりの性能でも電力あたりの性能でも優位であったであろうスカラ部に一本化されたことには結果的にはメリットがあったと思われる。

「京」は 2011 年に完成し、Top500 リストで 2011/6 と 2011/11 の 2 回にわたる 1 位 (2012/6 は IBM BlueGene/Q の Sequoia、2012/11 は NVIDIA K20x の Titan である) を獲得し、さらに 2011 年、2012 年と連続してゴードンベル賞を受賞する等の大きな成果をあげた。

「京」と国立大学や主要な国立研究開発法人（国研）のスパコンのアーキテクチャを比較すると、2011 年の段階では主要な国立大学基盤センターでメインのマシンがベクトル機であるのは東北大学、大阪大学のみ、国研では地球シミュレータ後継を設置した JAMSTEC 以外には大規模なシステムはなくなっており、ほぼ CPU (ほとんどは Intel/AMD の x86 アーキテクチャ) となっていた。東工大の TSUBAME スパコンは 2010 年 11 月稼働の 2.0 から GPU が主体のシステムとなり、2.4 PF と高いピーク性能を実現していた。

「京」の CPU である SPARC64 VIIIfx は、HPC2500 に採用された SPARC64 V や FX1 に採用された SPARC64 VII の経験を生かした優れたプロセッサであった。特に重要なのは、比較的深い浮動小数点演算パイプライン (6 段) に対して、レジスタウィンドウを使うものの非常に多くのアーキテクチャレジスタを用意した HPC-ACE 命令セットであり、これをコンパイラがループアンロールやソフトウェアパイプライニングで有効に使うことで、演算密度の高いカーネルでは確実に高い実行効率をあげることができた一方、 B/F = 0.5 と高いメモリバンド幅によってメモリアクセスが主体になるカーネルでもある程度の実行効率を実現できた。当時の x86 コアに比べて整数命令の IPC ではかなり落差があったが、この 2 つの特性によって広い範囲のアプリケーションで高い実行性能を実現できたといえる。

「京」の 10 PF という目標は、2006/6 の Top1 であった LLNL 設置の BlueGene/L の 367 TF に比べると 30 倍で、ムーアの法則が生きていても 5 年で 10 倍がトレンドであることから見ると 3 倍ほど大きいが、消費電力は BlueGene/L の 1.4 MW から 12.7 MW とほぼ 10 倍になっており、技術的に困難な目標というわけではなかったと考えられる。

1.2 「富岳」

「京」の後継、ポスト「京」(以降、最終の名称の「富岳」を用いる) は、「京」が完成した 2011 年度から検討が始まった。まず今後の HPC 技術の研究開発の検討WG（2011 年 4 月〜7 月）が開催され、その提言をうけてアプリケーション作業部会、コンピュータアーキテクチャ・コンパイラ・システムソフトウェア作業部会が 2012 年 2 月にまとめられ、その結果をうけて 2012 年度、2013 年度の 2 年間で「将来の HPCI システムのあり方の調査研究」が行われた。ここでは、システム設

計について東京大学+富士通、筑波大学+日立、東北大学+NEC の 3 グループが参加し、それぞれ汎用スカラ、アクセラレータ (加速部)、ベクトルのシステムの性能検討を行った。

この結果 2014 年 4 月から「エクサスケール・コンピューティング・プロジェクト」として「富岳」の開発がスタートする。当初は汎用部と加速部の両方を持ち、目標演算性能が 1 EF「級」、消費電力が 30〜40 MW を目標とした。

半導体技術的には、「京」の時との大きな違いは、ムーアの法則による性能向上、特に電力あたり性能の向上が大きくスローダウンし、トランジスタ密度の向上もそこまでではないがスローダウンしたことである。

ナショナル・フラグシップ・マシンというわけではないが「京」の 9 年前に完成した初代地球シミュレータは 180 nm プロセス、「京」は 45 nm プロセスであり、どちらもバルク CMOS であるのでほぼスケーリングが効く。とは言え、動作電圧は 1/4 ではなく 1/2 から 1/3 程度と想像されるが、 1/3 としてもトランジスタレベルで消費電力が 1/36 に低下する。実際の電力性能は 70 倍程度であり、これはスカラーアーキテクチャとベクトルアーキテクチャの差、特に B/F が 4 から 0.5 に低下していることを考えると妥当であろう。

一方、当初「富岳」が完成予定であった 2019 年に利用可能と想定されるプロセスは N10 と呼ばれるものであるが、これは「京」に使われた 45 nm に対してトランジスタの 1 次元サイズが 1/4.5 になるもの「ではない」。 CMOS プロセスは TSMC の場合 20 nm の次から FinFET に移行する。FinFET に移行した第一世代は N16、第二世代は N10、第三世代は N7 と呼ばれるが、N16 のトランジスタ密度は 20 nm とほぼ同じ、 N7 で N16 の「最大 3 倍」である。45 nm に対して 20 nm は 6.25 倍になっているとしても、トランジスタ密度の向上は N7 で最大で 19.75 倍、N10 では おそらく 12 倍程度にとどまったと思われる。

実際には、 TSMC は N10 ノードの開発に失敗し、目標の性能を実現できなかったため、「富岳」は N7 ノードで開発されることになり、予定よりも 1 年遅れた 2020 年に完成した。

「富岳」の公式の目標は「アプリケーションで最大「京」の 100 倍」となっている。上にみたように半導体の性能向上だけでは実現できないものになっている。

まず、N7 ノードのため、トランジスタ数の増加はかなり大きいが、それでも面積あたりで 20 倍程度にとどまる。また、はるかに大きな問題は電力性能の向上であり、動作電圧の減少はおそらくあるとしても 40%程度のため、この分による電力性能向上は最大でも 3 倍程度、微細化による電力性能向上は 3–4 倍程度のため、半導体技術からくる電力性能向上は 10 倍程度以下となる。

実際に「富岳」ではどうなったかというと、HPL での電力性能が「京」の 0.8 GF/W から 15 GF/W とほぼ 20 倍になった。アーキテクチャの改良により 2 倍程度以上の電力性能向上を実現したことがわかる。

これは大きな成果である一方、SIMD 演算幅が 4 倍、演算器や L1D キャッシュのレイテンシが 6 サイクルから 9 サイクルへと 1.5 倍となった。また、L2D キャッシュのバンド幅が低下し、レイテンシが大きく増加した。さらに、浮動小数点演算の命令セットアーキテクチャに HPC-ACE に変わって新しく策定した ARM SVE を採用した結果、レジスタウィンドウが廃止されアーキテクチャレジスタが 32 本となった。このため、「京」では可能であった、多数のレジスタを使うことで

の演算レイテンシの隠蔽が困難になった。物理レジスタはアーキテクチャレジスタより多いので、OoO 実行とレジスタリネーミングでパイプラインを埋める、というアプローチであったと思われるが、命令ウィンドウがあまり大きくなく、また物理レジスタ数も少ないため、実際にはコンパイラが普通に生成したコードでは実行時にパイプラインが埋まらず、演算リッチなカーネルでも実行効率が高くならない、という問題が起こっている。また、L2D キャッシュやメインメモリのレイテンシが非常に大きいため、SIMD 幅が大きくなったこと、キャッシュを共有するコア数が増えたことと合わせて、主記憶アクセスのレイテンシ隠蔽も困難になった。

これらの困難のため、「富岳」では、「京」で高い実行効率を実現できたプログラム、Intel x86 で高い実行効率を実現できたプログラムのどちらでも、大きな実行効率低下を起こしているのが現状である。

「富岳」のもうひとつの問題は、AI への対応が遅れたことである。「富岳」の AI 対応は FP16 SIMD 演算の採用にとどまっており、FP16 演算のピーク性能が 12 TF である。これは、N12 を採用した、「富岳」の時期からすると前世代の GPU である NVIDIA V100 の 120 TF の 1/10 となっており、CPU と GPU の違いはあるとは言え、高いとはいいがたい。特に、 FP64 性能では 3 倍程度の差であることから、現在もっとも重要になっている AI 性能が相対的に低くなったのはハードウェアの利用価値に大きな影響があったといわざるを得ない。アーキテクチャを最終的に決定した 2016 年時点では AI の重要性は明らかであり、PFN では 2016 年初頭から MN-Core の開発を開始した。プロジェクト開始時の決定、性能目標をそのまま実現するのではなく、状況に応じて柔軟に変更することが必要であったと考えられる。

1.3 ポスト「富岳」FS

本報告書の対象となるポスト「富岳」FS は、「富岳」完成後 2 年経った 2022 年度から 2024 年度までの 3 年間にわたって実施されたが、2023 年度の終わりに実質的なシステム構成提案の評価が行われ、理研チームから複数出ていたシステム構成提案の中から、富士通 CPU と NVIDIA GPU の組み合わせによるものが採用された。「京」、「富岳」と採用されなかったアクセラレータがようやく採用されたことには一定の評価ができる一方、神戸大学提案の他にもあったと思われる国内でのアクセラレータ開発提案が採用されなかったことは残念である。

ここではまず、「京」、「富岳」の時期との状況の違いについて整理しておく。

大きな変化は

a) ムーアの法則による半導体技術の進歩が大きくスローダウンし、特に電力性能の向上、SRAM 容量の向上がほぼ期待できなくなったこと

b) 社会、科学技術のあらゆる面で AI、特にトランスフォーマーに代表される基盤モデルの応用が急速に進歩し、社会そのものの基盤となりつつあること

の 2 つといえる。a) は、演算性能、特に FP64 性能の大きな向上は非常に難しくなった、ということを意味する。「富岳」の電力あたり性能は Green500 の測定値で FP64 15 GF/W であり、すで

に述べたようにこれは「京」の 20 倍程度である。一方、2025 年時点での最大スコアは GH200 の 73.3 GF/W であり、5 倍程度であるが、NVIDIA は A100 でも 33 GF/W を記録しており、そこからの向上は 2 倍程度である。N2 や N1.4 プロセスになっても、GPGPU アーキテクチャでは 150 GF/W まではいかないと考えられ、「富岳」に比べても 10 倍程度の向上にとどまる。

が、FP64 性能という観点からは、各社の GPGPU のターゲットが FP64 性能ではなく、 AI 向けの FP8、さらには FP4 性能になってきていることが大きな問題である。前処理・後処理に時間をかけられる密行列積では低精度の表現での演算を複数組み合わせて高精度にする尾崎スキーム等の利用もあり得る一方、HPC 応用で倍精度が必要になるケースのうち密行列積はごく一部であり、それらにどのように対応するかは未解決の問題となっている。

一方、ハードウェアから専用設計する場合には、乗算器アレイ等を共通化することで、複数の演算精度に対応することはそれほど困難ではない。実際、MN-Core およびを MN-Core 2 プロセッサでは、そのような、乗算器アレイを共通化する回路構成とすることで、比較的小さな回路規模で FP16 に対する FP32、FP64 の性能比率を高く維持している。

さらに、ここ数年での AI 計算需要の変化も大きい。これまでの深層学習では、CNN に代表される、比較的パラメータ数の小さなネットワークが主流であった。2 次元の画像認識で、典型的にはパラメータ数は 100M 以下であった。

一方、現在主流になっているトランスフォーマーベースの LLM では、非常に小さなモデルでパラメータ数が 8B（80 億）、DeepSeek-R1 では 671B に達する。DeepSeek では MoE (Mixture of Experts) が採用されており、トークン毎に全てのパラメータを読出すわけではないが、それでもトークン毎に 37B パラメータを読出す。これに対して、演算量はそれほど大きくならない。まず、バッチサイズを大きくしていくと、バッチ毎に採用するエキスパートが違うため、ある程度のバッチサイズでは結局ほぼ全部パラメータを読出すことになる。また、KV キャッシュはバッチ毎に別に構築されるため、こちらが大きくなると演算量とメモリアクセスの比は非常に小さくなる。

このことは、 LLM の性能はほぼ完全にメモリバンド幅リミットであり、演算性能はそれほど重要ではなくなってきている、ということである。

このため、NVIDIA も、 AMD も、最新のモデルではこれまで以上にメモリバンド幅を増やしている。H100 では 3 TB/s、 H200 では 5 TB/s、 B300 で 8 TB/s だったものを、 Rubin では一気に 22 TB/s まで、また AMD MI455X でも 19.6 TB/s まで増やしている。Rubin, MI455X のどちらも HBM4 であり、Rubin の消費電力は 2.3 kW、MI455X は 1.7 kW 程度と見込まれており、電力あたりのメモリバンド幅向上は H200 からそれほど大きくない。MI455X で 1 ビットアクセスあたりの消費電力が 11 pJ/bit 程度となる。

このことは、キャッシュ階層を持つ共有メモリアーキテクチャで、 HBM のような 2.5D 構造、すなわちプロセッサダイとメモリダイを平面的にパッケージ内で横に並べる構造では、実現できるメモリアクセスあたりの消費エネルギーに下限があることを示している。まずメモリダイ側は 10 mm 程度のサイズがあるので、メモリアレイからインターフェイスまでの距離がその程度はある。また、プロセッサダイ側はさらに大きいため、全体では典型的には 30 mm 程度の距離をデータが移動する。シングルエンドの配線のキャパシタンスは 0.2 pF/mm 程度はあるので、30 mm では 6

pF、電圧スイングが 1 V とすればスイッチングごとに 3 pJ の電力を消費する。変化確率が 50% とすれば 1.5 pJ/bit である。 これは配線だけを考えた場合で、 実際には多数のバッファやパイプラインレジスタのために電力が増え、さらにキャッシュ機構によって大きく電力が増える。

ポスト「富岳」を汎用 HPC の発展の観点からのみ見ると、第一の問題は演算の電力性能の向上が困難になってきていることだが、実際の主要なアプリケーションになってきている LLM の側からは、演算性能よりも電力あたりのメモリバンド幅のほうがより大きな制約になってきていることがわかる。

1.4 アーキテクチャの変革の必要性

ここで、ハイエンドの計算機アーキテクチャの、過去の変化を振り返っておく。これは大雑把に以下の 4 時代に区分できる。

- スカラープロセッサの時代。1975 年 CDC 7600 まで。
- 共有メモリベクトルプロセッサの時代。 1976 年 Cray-1 から 1992 年まで。
- 分散メモリ CPU マシンの時代。 1993 年 Cray T3D から 2007 年まで。
- アクセラレータの時代。 2008 年 Roadrunner から。

この変化は、基本的には半導体技術の進歩がドライブしている。スカラープロセッサからベクトルプロセッサへの変化は、完全にパイプライン化した演算器を実装可能なところまで集積度が上がったこと、磁気コアメモリより桁違いにアクセスタイムが短い半導体メモリが利用可能になったことの 2 つによって可能になった。第一の要因のため、1 つないし複数のパイプライン化された浮動小数点演算ユニットを単一の計算機に集積可能になり、第二の要因によってこれらの浮動小数点演算ユニットに高いバンド幅で主記憶からデータ供給することが可能になり、キャッシュ階層が不要になった。

共有メモリベクトルプロセッサから分散メモリ CPU マシンへの変化は、単一チップに完全にパイプライン化した演算器を集積可能になった結果、システム全体で物理的な共有メモリを維持しつつ全体性能を向上させることが困難になったことによる。CPU チップの演算性能が急速に向上した一方、オフチップのメモリバンド幅向上は、単一の CPU チップに DRAM メモリを直結した場合でも演算器にデータ供給するのに十分ではなく、複数 CPU を物理的な共有メモリにつなぐシステムではさらにメモリバンド幅が小さくなった。これらのシステムではキャッシュメモリが必須となった。

単一の CPU とメモリを直結させた計算ノードを構成要素とし、多数のノードをそれほどバンド幅が高くないネットワークで結合すれば、システム全体を共有メモリに結合させるのに比べてシステムコストを大幅に切り下げつつ演算性能もトータルのメモリバンド幅も向上させることができる。もちろん、並列化には MPI のようなの分散メモリ環境でのプログラミングが必要になり、書き換えや新規開発のコストは非常に大きいが、性能向上を続けるにはそれ以外の方法はなかったといえる。

米国ではこの時点でベクトルアーキテクチャは事実上消滅し、キャッシュを持つマイクロプロセッサベースのシステムが主流になったのに対し、日本では 1993 年にベクトルプロセッサを分散メモリアーキテクチャにした数値風洞が出現し、富士通 VPP500 シリーズとなった。また、 NEC も 1994 年発表の SX-4 以降で同様な分散メモリアーキテクチャを採用した。

日本では 2001 年の地球シミュレータでもベクトルアーキテクチャが継承され、NEC は現在までベクトルアーキテクチャの製品開発を続けている。富士通も 1999 年の VPP5000 まで分散メモリベクトルアーキテクチャが継承された。これらは、マイクロプロセッサベースのシステムに比べてアプリケーションチューニングが容易で高い実行効率を出しやすかったことは否定しがたいが、ピーク性能あたりのコストは高価なものになってきていた。

2008 年頃になると、CPU ベースのシステムからアクセラレータベースのシステムへの変化が起こる。Top500 では Sony/IBM の Cell プロセッサを使った Roadrunner の登場が大きな意義を持つが、その後主流になるのは NVIDIA および AMD の GPGPU ベースのシステムである。2010 年に投入された NVIDIA Fermi GPU は 630 GF と当時の Intel CPU の 10 倍程度のピーク性能を持ち、Top500 上位のシステムはアクセラレータを持つものがほとんどとなった。

ここで、CPU と GPU の違いは何か、を見ておくと、どちらも多数の演算コアを持ち、物理的な共有メモリをオフチップに持つ、というところはあまり変わりがない。一つの大きな違いはコアのアーキテクチャとプログラミングモデルで、GPU は SIMD 構造を多数のスレッドを実行させることに使うため、演算器やキャッシュ、メインメモリのレイテンシを隠蔽しやすい。また、多数のスレッドのため、大容量のレジスタファイルを持つので、レジスタのデータを再利用できれば非常に高い実行効率を実現できる。CPU は明示的な SIMD 幅をコンパイラがケアする必要があり、また同時実行スレッド数も少なく、アーキテクチャレジスタ数も少ないため、レイテンシ隠蔽が困難で、実行効率をあげにくく、また演算器のハードウェアレイテンシをあまり大きくできないために電力性能の向上が難しい。また、キャッシュコヒーレンシの扱いが大きく違う。CPU ではかなり厳密にコヒーレンシを維持するプロトコルが使われ、特にコア数が増えた時に消費電力の増加やキャッシュアクセスレイテンシの増加の大きな要因になっているが、GPU ではなんらかの緩和されたコヒーレンスモデルを採用することである程度スケールするようにしている。このため、GPU では CPU に比べると高い性能が出しやすい。また、DDR メモリではなく GDDR や最近では HBM を使うことで、GPU ダイと DRAM ダイの配線を短くし、消費電力を下げ、バンド幅を向上させてきた。

しかし、前節でみたように、GPU でも現状では HBM がオフチップで GPU ダイの横にあり、アーキテクチャ的にはコア間で共有されている、という構造自体が性能向上の制約になっている。これは、1980 年代後半から 1990 年代前半の共有メモリベクトルプロセッサや、あるいは当時多数開発されていたマイクロプロセッサベースの共有メモリプロセッサで、物理的な共有メモリがメモリバンド幅向上の制約になっていたのと同じ構造である。

すでに述べたように、この時にシステムレベルで起こった変革は共有メモリから分散メモリへの変化であった。一方、チップレベルでも物理的な共有メモリに限界がきているのは同じだが、現在までは GPU アーキテクチャと HBM のような オンパッケージまで DRAM 配線を短くする方向で

の対応にとどまっていた。

これは、CPU と伝統的な DRAM モジュールの組み合わせに比べると大きな電力削減になっていた一方、現在では明らかに限界にきている。とはいえ、これに代わる現実的なアプローチがまだ存在していないのも事実である。

1.4.1 本調査研究でのアプローチ

システムレベルでは分散メモリに移行したので、チップレベルでも分散メモリに移行すれば良いわけだが、問題は DRAM がオフチップであるなら (HBM のように同一パッケージ内でも) ある程度の配線長は発生するので、分散メモリアーキテクチャにしてもアクセスエネルギーの大幅な減少は期待できないことである。

つまり、「オフチップ」メモリでは駄目で、「オンチップ」でなければならない。「オンチップ」であれば、プロセッサコアと主記憶をの距離を 1 mm 程度まで短くすることは可能であり、データ移動によるエネルギー消費を 0.1 pJ/bit 程度にできる。現状では DRAM 自体の読出しエネルギーが 0.5 pJ/bit 程度あるため、こちらが大半を占めるが、それでも HBM を使ったメモリプロセッサに比べて 1 桁以上メモリアクセスエネルギーを下げることができる。

とは言え、DRAM を「オンチップ」にする現実的な実装技術はこれまで存在していなかった。

HBM はメモリ自体は 8–12 層の積層で、これはマイクロバンプを使っている。ピッチは 30 μm 程度である。HBM の DRAM スタックは、一番下にロジックダイと呼ばれる、プロセッサダイとのインターフェイスのためのダイがもう 1 枚ある構造を持つ。単純には、この「ロジックダイ」にプロセッサ機能を集積すればよいよう見えるが、これはそうではない。これは、HBM メモリではマイクロバンプはダイの中央部に集中しており、DRAM 上でもプロセッサダイ上でも 5 mm 程度のデータ移動が起こる。また、DRAM 上で高速インターフェイスを使って信号線数を減らしており、この部分の消費電力も無視できない。

また、実装上の大きな問題として、マイクロバンプを使った積層のため、熱抵抗が大きく、ロジックダイの上に DRAM スタックを載せる構造では冷却が困難になる。

2010 年頃までに、これらを解決できる要素技術はそろってきていた。最重要なのはハイブリッドボンディング、ないし Cu-Cu 直接接合と言われる技術で、バンプを形成しないで、SiO_2 層と同じ平面に形成した Cu パッド同士を熱処理で結合させるものである。これは Sony が CMOS センサーとロジックダイの接合のために開発した技術であるが、パッドサイズ 1 μm 程度の極めて微細なパッドが可能になっている。また、TSMC では自社のダイ同士の積層にハイブリッドボンディングが実用化されており、例えば Zen 3 以降の AMD CPU で採用されている。

ハイブリッドボンディングと同程度に重要なのは、「カスタム DRAM」が少なくとも技術的には可能になってきたことである。DRAM はロジック LSI と異なり、標準品の大量生産が行われてきた。これは現在でも大きく変わっておらず、 DDR、GDDR、LPDDR、HBM はそれぞれ JEDEC 標準があり、Top 3 DRAM ベンダは基本的にこれらの標準品だけを製造している。しかし、Top 3 ではない台湾系 DRAM ベンダは、プロセスノードでは遅れている一方、マスクコストは低く、設備コストも低いため、ビットあたりの製造コストには大きな差がない。このため、台湾系 DRAM

ベンダは、2020 年前後から、カスタム設計による 3 次元積層メモリという新しいビジネスモデルを作ろうとしている。

3 次元積層メモリの基本的な考え方は、上のハイブリッドボンディング技術を使うことで、DRAM-DRAM および DRAM-ロジックダイの接合パッド数を飛躍的に増大させ、データ移動距離を短くすることで、 HBM に比べて大きく消費電力を削減しよう、というものである。技術的には、非常に多くのパッドを使えるため、面積あたりのバンド幅もあげることが可能になる。一方、ロジックダイと DRAM ダイを積層するため、HBM ではなかった非常に多様な設計上の問題が発生する。

しかし、ハイブリッドボンディングと TSV による積層は要素技術としては十分に確立したものであり、DRAM とロジックの積層とそれによる分散メモリアーキテクチャを実用化できれば非常に高いメモリバンド幅を非常に低くアクセスエネルギーで実現できる、新時代のプロセッサになることは確実である。これは、AI、特に LLM の画期的な性能向上をもたらすだけでなく、汎用 HPC でも大きな性能向上につながると考えられる。このことは、「京」、「富岳」で高いメモリバンドを維持することの重要性、意義が示されていることからも明らかであろう。

このため、本調査研究では、GPU アーキテクチャによるアクセラレータの次にくるものとして、3D 積層メモリによるオンチップ分散メモリアーキテクチャを持つアクセラレータの可能性について検討した。

得られた主要な知見は以下のとおりである。

- オンチップ分散メモリアーキテクチャによって非常に高いメモリバンド幅を実現することは近い将来に十分可能である。
- 多くのアプリケーションで、並列化やアクセラレータへのオフロードの手法は現在の GPU と大きく違うものではない。もちろん、分散メモリに対応して、空間分割が必要になるが、これは基本的には MPI 並列化で行っている手法が使える。
- LLM 応用ではこのアーキテクチャのアドバンテージは極めて大きく、近い将来にこのタイプのアーキテクチャを実現したプロセッサが主流になることは避けられない。

本報告では、2 章にアーキテクチャとその物理的な面も含めた実現性検討、3 章にシステムソフトウェア・コンパイラの検討、4 章に個別分野のアプリケーションの検討をまとめた。

第 2 章　アーキテクチャ調査研究

2.1　アーキテクチャ調査研究の統括およびアクセラレータ評価

牧野 淳一郎 [国立大学法人 神戸大学]

アーキテクチャ調査研究全体の統括を行うとともに、独自アクセラレータアーキテクチャについて、リファレンス設計を定め、N7 ないし N5 のプロセスルールでの動作速度・電力性能を評価した。具体的には、NVIDIA、AMD のそれぞれのプロセッサの性能を比較評価し、TSMC の N16 から N3 までの性能評価を行った。結果は以下にまとめる。

2.1.1　商用アクセラレータの電力性能トレンドの推定

商用アクセラレータとして NVIDIA 社および AMD 社の GPGPU のこれまでのトレンドを調査し、将来の予測を実施した。

まず、GPGPU の世代間の性能向上が、TSMC 社のプロセスノードの性能改善との相関を調査した。TSMC 社のプロセスノード間の性能改善が表 2.1.1 である。

表 2.1.1　TSMC 社のプロセス情報

プロセス情報	N16/12	N10	N7	N5	N3	N3E	N2
Speed Improvement @Same Power	50%	15%/N16	20%/N10	20%/N7	15%/N5	18%/N5	15%/N3E
Power Reduction @Same Speed	60%	35%/N16	40%/N10	40%/N7	30%/N5	34%/N5	30%/N3E
Logic Density	x2	x1.6	x1.8	x1.7	x1.6	不明	
出典	TSMC	TSMC	TSMC	TSMC WikiChip	TSMC	TOM's Hardware	TOM's Hardware

ここで注意しなければならないのは、速度向上はともかく同一速度での電力削減は、複数の世代にわたって掛け算で効くものではなくなっていることである。これは、同一速度での電力削減にはかなり電圧を落とす必要があるが、この表にある TSMC 公式の値は、前世代での標準電圧から、新世代での同じ動作クロックになる電圧に落とした時の値であり、前世代の標準電圧から新世代の標準電圧にした時の値ではないからである。また、セルライブラリが通常は 0.55 V、超低電圧動作向けプロセスでも 0.4 V までしか動作保証がないため、前世代ですでにこれらの限界電圧まで落としていれば、新世代になっても電圧が下がらないため、消費電力の低下はこの表にあるよりもずっと小さくなる。

上記 TSMC 社のプロセスノード間性能改善と、NVIDIA 社 GPGPU の電力性能トレンドとの相関の調査および、将来予測結果を示したものが、表 2.1.2 である。

2024 年末になって、NVIDIA からは B200、B300 の詳細と、さらに次世代の Rubin、Rubin Ultra の概要が発表され、2 つの新製品が発表されるということになった。さらに、2026 年初頭には Rubin の性能が大きく変わり、Rubin Ultra は一旦ロードマップから消えたように見える。表 2.1.2 はそれらを反映している。

B200、B300 は H100 からの変更は大きくないが、FP4 がサポートされること、FP64 の性能が大きく低下すること、メモリバンド幅が向上することが主な変更点である。Rubin でも、FP64 の性能は H100/200 に比べると低下したままとなる。

これは、以下の 3 つを意味していると考えられる。

1. NVIDIA が AI マーケット、特に FP8、FP4 が重要と考えられている生成 AI や LLM を主要なターゲットとした。
2. NVIDIA アーキテクチャでは FP64/32 性能と FP16/8/4 性能にはトレードオフがあり、今後の製品では FP64/32 性能の向上は期待できない。Blackwell 以降 では低精度演算の組み合わせで DGEMM についてはある程度の性能を実現する尾崎スキーム等の活用が必須となるように見える。
3. 特に LLM でメモリバンド幅が極めて重要になるため、電力増加を容認してでもメモリバンド幅を増やす方向になっている。

表 2.1.2　NVIDIA 社 GPGPU のトレンド

製品名	P100	V100	A100	H100	B300	Rubin
製品発表年	2016	2017	2020	2022	2025	2026
プロセス	N16	N12	N7	N4	N4P	N3
プロセス量産開始年	2015	2017	2018	2021	2023	2024
演算性能@FP8 (TF)	N.A.	N.A.	N.A.	1979	4500	17500
演算性能@FP16 (TF)	21.2	125	312	989	2250	4000
演算性能@ FP64 (TF)	4.7	7.8	19.5	67	1.25	33
FP16/FP64	4.5	16.0	16.0	14.7	1800	121
周波数 (MHz)	1328	1455	1410	1780	1837	1800
TDP (W)	300	300	400	700	1400	2500
電力効率 (GF/W@FP64)	15.67	26.00	48.75	95.71	0.89	13.20
B Tr 数（トランジスタ数 [10 億]）	15.3	21	54	80	216	336
Die Size (mm^2)	610	815	826	814	1600	1600?
DRAM バンド幅 (GB/s)	720	900	2000	3000	8000	20000
DRAM 電力あたりバンド幅 (GB/J)	2.4	3.0	5.0	4.3	5.7	6.7

1 は、ここ数年の AI マーケットの急速な膨張が今後も続くと想定されることから当然と考えられる。2 は、設計として FP64/32 と FP16/8/4 が回路を共有していない、あるいは共有部分が小さいことを意味しており、興味深い。PFN の MN-Core では、精度が異なる演算間で可能な限り乗算器等のユニットを共有する構成（特許取得済）になっており、FP64 の性能をカットしても回路規模の大きな減少にはつながらない。また、そもそも演算性能に対してシリコン面積が小さく、他社のプロセッサに比べて大きな優位性を持っている。

3 については、LLM 推論では明らかにメモリバンド幅向上が望ましいが、メモリバンド幅向上のための手段は基本的に HBM の世代交代と数の増加によるということを意味しているように見える。実際、電力あたりのメモリバンド幅 (表では単位 GB/J とした) の改善が非常に小さく、ここ 10 年で 3 倍以下であることがわかる。改善が小さいのは、 HBM というフォームファクタが P100 以降同じだからである。このため、データが移動する物理的な距離は HBM の世代が変わっても大きくは変わらず、またインターフェイスの電圧も HBM3 までは大きく変わらなかったため、メモリアクセスエネルギーはほぼ同じか、高クロック化によって若干上昇した。

HBM3e まではデータ線が 1024 本である。HBM では 6.4 Gbps であった転送速度を HBM3e では当初 9.6 Gbps まで上げるとされていた。実際に製品に採用されたものではまだ 8 Gbps にとどまっており、また開発中のものも 9.2 Gbps にとどまる模様である。

一方、HBM4 は、HBM3e とは以下の 3 つの点で異なる。第一は、データ線が 2048 本に増えることである。これはもちろんバンド幅の増加をもたらす。第二は、標準設計においても、いわゆるベースダイに DRAM プロセスではなくロジックプロセスが用いられることである。これは、ベースダイがより低電圧、低消費電力で高速に動作することを可能にし、消費電力の削減を可能にする一方、無視できない価格上昇をもたらす。第三は、各社が計画している「Custom HBM」である。これは、ベースダイの設計をカスタマイズすることで、HBM4 で定義された標準のチップ間インターフェイスではなく、例えば UCIe や他の D2D に特化した高速・低電力で面積も小さいインターフェイス IP を使うことを可能にするものである。これらにより、ある程度の消費電力の削減が期待できる。

とは言え、HBM4 を採用するとみられる Rubin においても、電力あたりメモリバンド幅は大きくは改善していない。

システム全体として見ると、もちろん HBM の DRAM 自体およびロジックダイの DRAM I/F までの消費電力はある程度減る一方、ロジックダイ内部のデータ移動の消費電力は大きく減らすことは困難である。このことが、NVIDIA GPU のメモリバンド幅、特に電力あたりのバンド幅の向上があまり大きくないことの原因と考えられる。

このように、NVIDIA 社の GPGPU の世代間の演算性能および消費電力の改善は、Rubin に至るまで基本的には TSMC 社の公称改善率以内に収まること、さらに、今後は FP32/64 性能は大きく低下することが確認できた。

V100 および H100 では演算性能が前の世代から大きく改善しているが、ここは Tensor コアの採用や Tensor コアの大型化、半導体プロセスの改善以上の周波数向上等により演算器当たりのトランジスタ数を改善したことによると思われる。

その結果、前の世代に比べてダイサイズの大型化、トランジスタ当たりの消費電力量が改善していない等の状況が見てとれ、基本的には TSMC 社の公称改善率以内での改善に収まるペースでの改善トレンドであると言える。

「ポスト富岳」世代では、半導体プロセスの量産化が順調に進めば N2 世代の Rubin Next の世代、若干の遅延が生じると N3 の Rubin 世代となると思われる。これらでは、FP64/FP32 の性能は極めて大きく低下する。従来とは全く違った数値計算法が必須になると考えられる。

表 2.1.3 は AMD 社 GPGPU のトレンドである。2022 年当初に、こちらも NVIDIA 社 GPGPU と同様の分析を実施した。現在の表は 2026 年時点でのアップデートである。

こちらも NVIDIA 社と同様に、基本的には TSMC 社の公称改善率以内に収まることが確認できた。 AMD の GPGPU の NVDIA GPGPU との大きな違いは FP64 性能を維持してきたことであるが、 MI400 系以降は同一世代でアプリケーションエリアによって複数のモデルを提供する方向になるようである。MI430X が HPC 向けとされている。しかし、その性能は未公表であり、アーキテクチャも不明なためピーク性能、電力性能ともに予測は困難であるが、N2 プロセスを採用し、N5 から最大限の電力性能改善を実現したとすると、FP64 で 300 TF、200 GF/W 程度、消費電力 1500 W 程度と予測される。

表 2.1.3　AMD 社 GPGPU のトレンド

製品名	MI8	MI25	MI50	MI100	MI250X-OAM	MI300X	MI355X	MI455X
製品発表年	2017	2017	2018	2020	2021	2023	2025	2026?
プロセス	N28	GF14	N7	N7	N6	N5	N3	?
演算性能@FP16 (TF)	8.2	26.4	26.5	186.4	383	1300	3000	?
演算性能@FP8 (TF)	-	-	-	-	-	-	5000	20000
演算性能@ FP64 (TF)	0.512	0.768	6.6	11.54	47.85	81.3	79	?
FP16/FP64	16.0	34.4	4.0	16.2	8.0	16.0	38.0	?
周波数 (MHz)	1000	1500	1725	1502	1700	2100	2100	?
TDP (W)	175	300	300	300	560	750	1400	2500
電力効率 (GF/W@FP64)	2.93	2.56	5.75	5.01	85.45	109.0	56.4	?
B Tr 数	8.9	12.5	13.2	21	58.2	153.0	245	300
Die Size (mm^2)	596	510	331	763.2	1448	1017	1100?	1100?
DRAM Size (GB)	16	16	32	128	128	192	288	432
DRAM バンド幅 (GB/s)	436	1020	1230	3200	3200	5300	8000	19600

次に、アクセラレータの費用の分析を実施した。16/12 nm から 5 nm まで、ファウンドリから販売される Wafer の価格を調査し、それをもとに 3 nm、2 nm の価格を推定したものが表 2.1.4 である。

ここから、ファウンドリから販売される Wafer の価格は世代が進むごとに指数関数的に上昇しており、さらに、世代が進むごとに金額の上昇率自体が上がっている傾向が見てとれた。

3 nm、2 nm では 5 nm 時点の 1.82 倍以上の倍率となることが容易に想像できるが、最低ラインである同じ倍率で試算している。

「富岳」では 7 nm のプロセスを使用していたが、仮に本プロジェクトにて 3 nm を採用すれば、Wafer 価格は最低でも 3.3 倍、2 nm を採用すれば最低でも 6 倍となることになる。

本プロジェクトでは、この半導体価格の上昇が大きな制約となることが予想される。

表 2.1.4　Wafer 価格と価格上昇率

プロセスノード	Wafer 価格	価格上昇率
16 nm/12 nm	$3,984	–
10 nm	$5,992	1.50 倍
7 nm	$9,346	1.56 倍
5 nm	$16,988	1.82 倍
3 nm	$30,901（予測）	1.82 倍（予測）
2 nm	$56,209（予測）	1.82 倍（予測）

Wafer 価格の出典: https://www.techpowerup.com/272267/alleged-prices-of-tsmc-silicon-wafers-appear

2.1.2　商用 CPU の電力性能トレンドの推定

商用 CPU のトレンド調査のため、2015 年以降に発売された Intel 社 CPU のうち、サーバー向けである Platinum ランクからモデルを抜粋したものが表 2.1.5 である。

2015 年発売の Skylake 世代から、2022 年発売の Sapphire Rapids 世代までの 8 年間で、最大演算性能は 3.66 倍に上昇している。しかしながら、2016 年発売の NVIDIA 社 GPGPU P100 と 2022 年発表の H100 SXM は、7 年間で倍精度演算の演算性能が 14 倍まで上昇していることを考慮すると、CPU の演算性能の向上は極めて緩やかであることが伺える。

この演算性能の向上は最大周波数および演算コア数によってもたらされているが、Sapphire Rapids はチップレット技術を採用することで、より多くのコア数を実現している。これを除けばコア数はほぼ横ばいであり、その演算性能の向上は周波数の向上によってもたらされていることがわかる。

また、電力効率も 8 年間で 2 倍程度の改善と、NVIDIA 社 GPGPU が 5.22 倍であることを踏まえると、こちらも極めて緩やかであると言える。

表 2.1.5　Intel 社 CPU のトレンド

コードネーム	発売年	ランク	モデル	コア数	最大周波数 [GHz] (AVX512)	TDP [W]	最大演算性能	GF/W
Sapphire Rapids	2022	Platinum	8490H	60	2.9	350	5568	15.9
IceLake	2019	Platinum	8368Q	38	3.3	270	4013	14.9
Cascade Lake	2019	Platinum	8280L	28	2.4	205	2150	10.5
Skylake	2015	Platinum	8180M	28	1.7	205	1523	7.4

2.1.3　MN-Core 後継の可能な電力性能の推定

PFN・神戸大学で、これまで開発を行ってきたアクセラレータについて実機での計測結果および設計時の評価結果をもとに電力性能の推定を行った。これは、MN-Core (12FFC プロセス)、MN-Core2 (N7 プロセス)、MAU-Shuttle (N5 プロセス) について、MAB と呼んでいるプロセッサ

ブロックの電力性能と、その中の演算器の部分のみの電力性能をまとめたものを表 2.1.6 に示す。MAU-Shuttle についてはシミュレーション、他は実測である。

この数値と、プロセッサボード全体の消費電力を使い、さらに半導体技術の進歩を考慮することで、次世代システムの電力性能を見積もる。

表 2.1.6　アクセラレータ・テクノロジーノード別 推定電力性能

アクセラレータ名称 (動作電圧)	Technology Node	(MAB) 電力性能 (TFLOPS/W@FP16)	演算器のみ (TFLOPS/W@FP16)	Target Clock (MHz)
MN-Core (0.55 V)	(12 nm Node)	1.82	NA	500
MN-Core2 (0.55 V)	(7 nm Node)	2.75	6.26	610
MAU-Shuttle (0.65 V)	(5 nm Node)	NA	8.55	1100

2.1.4　独自アクセラレータアーキテクチャの検討

現時点で、アクセラレータのアーキテクチャとしては様々なものが提案・実装されている。本節では、まず 2.1.4.1 節でそれらを概観したあと、2.1.4.2 節以降で独自アクセラレータアーキテクチャについて検討する。

2.1.4.1　既存アクセラレータアーキテクチャの比較

以下のものを検討対象とする。

1. NVIDIA GPU
2. PEZY-SCx
3. Sunway SW26010(Pro)
4. Intel Max GPU
5. MN-Core

これらについて、命令処理方式とメモリ階層、コア間通信の実現方法、結果としての面積・消費電力の観点から考察する。

2.1.4.1.1　命令処理方式

ここで比較するのは、コア間 SIMD か MIMD かとコア内で SIMD ユニットを採用しているかどうかである。PEZY-SCx と Sunway はコア間は MIMD である。Sunway はコア内で SIMD を採用しており、SIMD 幅は 256 ビット (Pro では 512 ビット) である。PEZY-SC2 まではコア内はスカラー演算、SC3 は 128 ビット SIMD である。これらにおける SIMD 方式は x86 の SSE/AVX 等と同様であり、1 命令で処理されるデータ幅が広い。レジスタ内では要素毎の独立な並列演算となる。メモリアクセスはおそらく unaligned load はサポートしているものと思われる。ストライドアクセスや間接アクセスについては不明である。

PEZY-SCx では最大 8 スレッドが時分割でサイクル毎に切り替わって動作する。この機能により、演算や L1 アクセスのレイテンシを隠蔽でき、コンパイラによる命令スケジューリングやハー

ドウェアによる OoO 実行を不要にしている。Sunway ではこのような SMT のサポートはなく、OoO 実行もリネームレジスタもないため、高い性能を出すには物理レジスタを注意深くアサインする必要がある。PEZY ではスレッド毎にレジスタファイルが必要になるため、一見大規模なハードウェアリソースが必要なように思われるが、複数スレッド間で共有の必要がないため、ポート数の少ない複数のメモリでレジスタファイルを実装でき、ハードウェア規模や消費電力へのインパクトは小さい。

1 コアで多数のスレッドが走ることで、これらのスレッド間の同期に時間がかかると大きなオーバーヘッドになり性能低下を招くが、PEZY はよく考えられたアーキテクチャであり、キャッシュ階層単位での同期が可能になっている。

NVIDIA GPU では、SM という単位内での SIMD である。A100 では、1 チップに 128 個の SM があり、これが FP32 Cuda コア 64 個、Tensor コア 4 個をもつ。

1 つの Tensor コアは、16 ビットの場合の行列に対しての演算を行う。従って、データ幅は 256 ビット (積算項はその 2 倍) である。これが 4 つあることでデータ幅は 1024 ビット、積算項については 2048 ビットとなる。演算数は 256FMA である。

Cuda コアにおける行列演算でない FP32/FP64 演算の性能は Tensor コアによる行列演算性能の 1/16 であるので、おそらく FP32 ユニットは実装されておらず、FP16 ユニットが SM 当たり 16 個あるものと推測される。レジスタファイルは SM 当たり 256 kB である。

NVIDIA の資料では、Tensor コア毎に L0 I キャッシュ、Warp Scheduler と Dispatch unit を持つことになっているので、この場合 SM の中で Tensor コア毎の 4 ユニットが MIMD 動作し、データ幅は 256 ビットということになる。つまり、AVX(2) 命令をもつ x86 プロセッサのような、256 ビット幅の SIMD 命令をもつプロセッサの集合体とみなすこともできる。

NVIDIA GPU の特徴は、Tensor コア毎に 64 kB と巨大なレジスタファイルをもつこと、この巨大なレジスタファイルを利用して、非常に多数のスレッドを起動できることである。ここで注意するべきことは、この「スレッド」は SIMD ユニットの中で動いているので、通常のスレッドのような MIMD 動作するものではなく、ソフトウェアから見える SIMD ユニットの幅が増えたようなものであることである。これは、ハードウェアのレイテンシを効率的に隠蔽する効果をもち、またアプリケーションの並列性の記述を簡潔にできる、極めて有効な機構である一方、メモリバンド幅自体の不足を解消できるものではない。

NVIDIA GPU は、L1 や L2 キャッシュに比べてもオンチップメモリの総容量ではレジスタファイルが大きい、という非常に特異なアーキテクチャをとっていた (H100 以降は変わる)。このため、キャッシュブロッキングのような、データ再利用をもっとも効果的に行うためには、データをキャッシュではなくレジスタファイルに置く必要があった。一方、レジスタファイルはスレッド間で共有されないため、複数スレッドで共有したいデータをオンチップでもつためにはキャッシュを使う必要がある。このため、NVIDIA GPU では、オンチップメモリを使ったデータ再利用を効率的に実現するにはアーキテクチャに固有の特異なコーディングをする必要があり、その場合でもキャッシュのバンド幅はそれほど高くないため有効性には限界がある。

主記憶アクセスに対しては間接アクセス、すなわち、その時のスレッド毎に独立なアドレスを

もって主記憶アクセスができる。これはもちろんレイテンシが高く、アドレスの分布によってはスループットも低いが、スレッド数が大きい時にはレイテンシは隠蔽される。

Intel MAX GPU は、単一の Xe コアに Vector Engine (VE)、Matrix Engine (XMX) がそれぞれ 8 個搭載され、VE は FP64/32/16 演算を 32/32/64 個、XMX は FP16 で 512 演算となっている。一方、データ幅は 512 ビットと記載されている。FP32 に対してはサイクル毎に 16 データで演算幅とあっているが FP64 では不足しているように見える。

Intel 発表資料ではチップ全体で FP64/32 52 TFLOPS、レジスタバンド幅 419 TB/s、L1 バンド幅 105 TB/s となっており、VE 数は 1024 であるので動作クロックは 1.6 GHz、ベクトル演算のデータ幅は FP64 で 1024 ビットとなる。FMA 演算では 2 演算に対して 3 入力 1 出力であり、サイクル毎に VE1 つで 4096 ビットのレジスタアクセスが発生するはずであるが、Intel 資料からは 1 サイクルで VE がアクセスできるレジスタは 2048 ビットで不足している (FP32 では問題ない)。

レジスタ構成、スレッドをサポートしているかどうか、主記憶アクセスにスレッド毎の独立アドレスをサポートするかどうか等は不明である。ただし、レジスタファイルは VE 当たり 64 kB と巨大であり、NVIDIA GPU と同様の SMT サポートがある可能性が高い。

MN-Core は完全 SIMD であり、チップ上の全てのユニットが同期して同一命令を処理する。間接アクセスは各ユニットが持つローカルメモリに対して行われる。

アプリケーションの実行効率については、MIMD が高く、SIMD では低いのではないかと考えがちであるが、これに明確な根拠があるとはいいがたい。その理由の 1 つは、MIMD といいながら、PEZY-SCx 以外の全てのプロセッサはコア内や SM 内は SIMD であることである。このため、条件分岐はマスク演算に置き換えられる。MIMD が有効に働くためには、プロセッサコアや SM 単位で違う内容を、条件分岐の形ではなく実行するようなコードである必要がある。これは、通常のループ並列化や OpenCL、Cuda 等による並列化で発生することは稀である。

特にコア数が大きい時には、並列化の効率に大きな影響をもつのは同期や放送・縮約等の大域通信のオーバーヘッドである。これについては 2.1.4.1.4 節で述べる。

2.1.4.1.2 メモリ階層

全てのプロセッサはメモリ階層をもつが、その実装はプロセッサ毎に大きく異なる。以下それぞれについて述べる。

PEZY-SCx はノンコヒーレントな 3 層キャッシュをもつ。SC の特徴は、キャッシュラインが L2、L3 とメインメモリに近づくほど大きくなることであり、メモリコントローラやキャッシュコントローラを複雑にせずに高い実効転送バンド幅を実現している。ノンコヒーレントなので、他のコアが変更したデータを触るためには明示的にキャッシュをフラッシュする必要がある。フラッシュ命令は階層毎に用意されている。

もうひとつの特徴は、各 PE がキャッシュ階層とは独立にローカルメモリをもつことである。このローカルメモリはメモリ階層から完全に切り離されており、低消費電力で高速にアクセスできる。PEZY-SCx では総容量ではキャッシュよりもこのローカルメモリが大きく、大きな容量がソフトウェアから明示的に利用できるためデータ再利用が容易になっている。

Sunway では、プロセッサは 1 個の Management Processing Element(MPE) と 64 個の Computer Processing Element(CPE) に分かれており、MPE は I キャッシュと L1、L2D キャッシュを持ち、CPE は I キャッシュを持つが D キャッシュは持たない。その代わりに 64 kB のローカルメモリをもち、主記憶とローカルメモリの間は DMA でデータ転送する。主記憶のストライドアクセスはあると推測される。間接アクセスはなく、必要なら複数の DMA を起動する。

64 個の CPE はの 2 次元格子構造をもち、x 方向、y 方向のそれぞれで Point-to-point、放送、総和をレジスタ間通信で行うことができる。この機能は並列化効率の向上に非常に有効である。

NVIDIA GPU は PEZY と同様なノンコヒーレントな階層キャッシュをもつ。ラインサイズは 128 byte 固定である。L1 を分割してローカルメモリとして使用できる。

Intel MAX GPU は階層キャッシュである。コヒーレンシサポート、ラインサイズは不明である。L1 を分割してローカルメモリとして使用できる。

MN-Core は、階層メモリ構造はとるがキャッシュではなく、階層間のデータ転送を命令で明示的に与える。このデータ移動命令も SIMD である。階層間での放送・総和等もサポートする。

2.1.4.1.3 オフチップメモリ

全てのプロセッサで、なんらかの形のオフチップメモリをつけている。表 2.1.7 に外部メモリの主要なパラメータについてまとめる。

表 2.1.7 オフチップメモリ構成

	ピーク演算性能 (FP64)	ピークメモリ転送速度	B/F	備考
PEZY-SC2 4.1	4.1 TFLOPS	95 GB/s	0.023	DDR3
Sunway SW26010	3.06 TFLOPS	137 GB/s	0.045	DDR4
NVIDIA A100	19.5 TFLOPS	1.56 TB/s	0.080	HBM2
Intel Max GPU 52	52 TFLOPS	3.2 TB/s	0.062	HBM2e
MN-Core	38.4 TFLOPS	400 GB/s	0.014	LPDDR4

Intel MAX GPU は、FP64 のベクトル演算に対する値としている。FP32 等では行列演算があり、FP64 でも行列演算のサポートがあるとピーク性能が 2 倍、B/F は半分の 0.03 になる。

「京」で B/F = 0.5、「富岳」でも 0.36 を維持したことを考えると、これらのアクセラレータでは B/F の値が非常に低くなっていることがわかる。特に、Intel Max GPU では L2 キャッシュの B/F も 0.25 と極めて低いものになっており、アプリケーション性能へのインパクトがあるのではないかと想像される。

2.1.4.1.4 コア間通信について

前節で、メモリ階層についてハードウェアの観点からまとめたが、本節ではどのような操作にどのようなオーバーヘッドがあるか、という観点からチップ内コア間通信がどのように実現されるかをまとめる。

- Point-to-point 転送

  Sunway と MN-Core 以外の全てのプロセッサで、キャッシュを経由する。このため、キャッシュラインのフラッシュ、同期、リードのステップが必要となり、オーバーヘッドは大きくなる傾向がある (キャッシュフラッシュには容易にマイクロ秒オーダーの時間がかかる)。

  Sunway では、64 個の CPE グループの中では、任意のコア間で x 方向、y 方向の 2 ステップのレジスタ間通信によってデータを送ることができる。このレイテンシはナノ秒オーダーであり、非常に高速である。

  MN-Core では、各階層の共有メモリを経由することでデータ転送が可能である。SIMD である関係上コア間でランダムな通信では多数回のアクセスが必要になるが、規則的な移動ならば効率的な転送が可能である。

- 同期・放送・縮約

  Sunway と MN-Core 以外の全てのプロセッサで、キャッシュを経由する P2P 通信によって実装される。このため、関係するプロセッサ数の対数オーダーのオペレーションが必要になり、レイテンシは極めて大きくなる。NVIDIA GPU では典型的には 100 マイクロ秒程度が総和にはかかる。この時間は多くのアプリケーションでは致命的といえるほどに長く、アクセラレータの広い範囲のアプリケーションへの適用を妨げる大きな要因になっている。

  Sunway では、64 個の CPE グループの中では、縦、横の 2 回のオペレーションで同期・放送・縮約ができる。このレイテンシはナノ秒オーダーであり、非常に高速である。

  MN-Core では、全体 SIMD であるため同期は不要である。また、各階層の共有メモリ経由で放送・縮約が可能であり、チップ全体に対してナノ秒オーダーでの放送・縮約が可能である。

  コア間縮約のオーバーヘッドは、多くのアプリケーションで並列化効率に直結する。単純な例としては行列ベクトル積がある。コア間縮約が遅いため使えない場合には、結果の 1 要素の演算は 1 コアで行う必要があり、行列の 1 次元方向の要素数よりも多くのコアを使うことは難しい。縮約が十分速いなら、1 つの結果要素のための内積演算を複数コアで並列に行えるため、並列化効率が飛躍的に向上する。また、多くのアプリケーションで全体での最大値や総和といった演算が実際に発生する。典型的な例は有限要素法の反復計算で現れる内積演算である。MPI によるノード間通信での総和は、近年は IB スイッチが縮約をサポートすることもあり高速化しており、チップ内での総和のサポートは必要度が高いものになっている。

2.1.4.1.5 電力コストについて

ここでは、命令実行方式、メモリ階層構造およびオフチップメモリについて、消費電力の観点から検討する。

まず、MIMD/SIMD の選択と電力コストについては、命令フェッチ・デコード・スケジュールユニットのコストを考えるとその個数が少ないことが望ましいのは自明である。ただし、汎用 CPU のようなコア内 SIMD 命令では、無制限に SIMD 幅を増やすことはどこかの時点で消費電力増加につながる。これは、レジスタ内のシャッフル命令等がビット幅に比例以上のコストがかかるこ

と、演算ユニット、メモリユニットともに物理的に大きく、配線長が長くなるため、配線遅延をカバーするためには大電力のドライバが必要になること等による。これを避けるためには、原理的にはシャッフル命令を排除、またはそのスループットを下げ、さらにメモリ・レジスタ間転送で非アラインド転送を排除して、演算器の近くに L1D キャッシュをもってくればよい。これはしかし、キャッシュ階層をもつアーキテクチャでは困難である。

これらの観点から、「電力コストだけを考えた時には」命令実行方式としてはなるべく大きな単位での SIMD 方式が望ましいが、その場合にはメモリ階層の構造も合わせて考える必要があることがわかる。

次に、メモリ階層構造について検討する。2.1.1 節の数値からは、2029 年頃のアクセラレータについて、半導体技術の進展、アーキテクチャの進歩から想定されるボードレベルの電力当たり性能は FP64 性能を十分重視したアーキテクチャでは 500 GF/W が可能と考えられる。この値から、「1 演算につきデータをどの程度の距離チップ内で移動可能か」が決まる。チップ内の電気信号によるデータ移動に必要な消費電力は、配線長さ l の単位長当たりのキャパシタンス c と信号の電圧振幅 V によって、以下のように与えられる。

$$w = \frac{clV^2}{2} \tag{2.1.1}$$

配線からグラウンドプレーンまでの距離と配線の幅ないし高さが同程度で絶縁体の誘電率があまり大きくない時、c は次式の程度である。

$$c = 0.2 \text{ (fF/}\mu\text{m)} \tag{2.1.2}$$

l 電圧振幅 V を 0.6 V、配線長 l を 10 mm とした時、w は 0.36 pJ である。従って、64 ビットのデータを 5 mm 移動すると、12 pJ のエネルギーが消費される。

仮に電力性能の目標を 1 TFLOPS/W とすると、これは 1 演算当たり 1 pJ である。すなわち、64 ビットデータを 0.43 mm 移動するだけで 1 pJ 消費する。B/F = 1 の LLC があるとして、演算器からメモリセルまでの平均配線長が $0.43 \times 8 = 3.5$ mm を超えると、LLC までのデータ移動だけで 1 TFLOPS/W を実現するための電力をすべて消費してしまうことがわかる。従って、ある程度の規模でチップ内コアから物理的に共有される LLC をもつ限り、2029 年頃に半導体技術的には実現可能と考えられる 1 TF/W を実現するのは実際には不可能である。

これに対する対応としては以下のようなものが考えられる。

1. 共有キャッシュを排除し、Sunway のようなメインメモリへの DMA のみ、あるいは MN-Core のようなチップ内ネットワークのみとする。
2. キャッシュメモリのバンド幅を下げる。
3. 3 次元実装、チップ上光通信等の技術でデータ移動のエネルギー消費を下げる。
4. 上の (1)-(3) の組合せ等。例えば、PEZY-SC のようにキャッシュとローカルメモリを組み合わせて、さらに DMA やチップ内ネットワークももつ。

消費電力とコストだけを考えると 1 が望ましいが、すでに GPGPU にポートされているアプリケーションや、特に OpenMP 等で記述されているアプリケーションのポータビリティを考えると、キャッシュがあるメモリ構成が望ましい。一方、OpenACC の場合には配列のレイアウト等を指定できることもあり、むしろ DMA やチップ内ネットワークのほうが効率的な実装ができる。

最後に、オフチップメモリについて検討する。

HBM3 を使った場合に、データ転送のエネルギー消費は 3–4 pJ/bit と言われている。これは、DRAM チップ上、サブストレート上、プロセッサチップ上でそれぞれ 5–10 mm 程度のデータ移動があり、特に DRAM チップ上では駆動電圧をあまり下げられないことから原理的な限界といえる。このことから、DRAM へのデータ移動の消費エネルギーが全体の 20%、目標電力性能が 1 TF/W とすると、1 演算当たり DRAM アクセスの消費エネルギーが 0.2 pJ、移動できるデータは演算当たり 0.07–0.1 ビット、B/F の値としては 0.008–0.0125 となる。

B/F が 0.01 ということは、オフチップメモリ 1 語アクセスにつき 1 千演算程度しないとメモリバンド幅リミットになる、ということである。大規模な密行列の操作が主体となるアプリケーションではそのような状況もあり得るが、大多数のアプリケーションではこれは現実的ではない。

このことは、HBMx の現在の実装の延長では、多くのアプリケーションが有効に利用できるオフチップメモリをアクセラレータにつなげることは困難である、ということを意味する。従って、より低い消費電力でアクセスできるオフチップメモリ実装方式の検討は必須である。そのオプションとしては以下のようなものが考えられる。

1. HBMx メモリの設計はそのままで、ロジックダイ上に 3 次元実装する
2. HBMx メモリの実装を変更し、特に DRAM ダイ上でのデータ移動を減らす
3. カスタムあるいは今後期待される標準品 DRAM をロジックダイ上に 3 次元実装する
4. SRAM ダイをロジックダイ上に 3 次元実装する

本調査研究では、これらのオプションについて、消費電力とアプリケーション性能への影響の観点から比較検討した。

2.1.4.1.6　メモリ階層のオプション

本調査研究では、人的な資源の制約もあり、チップ内メモリ階層については 1 のチップ内ネットワークをベースにしてアプリケーション実装はどのようになるか、キャッシュに比べてどのような優劣があるかを検討する。オフチップメモリについては、コスト・バンド幅の観点からは最善といえる 3 のカスタム DRAM メモリの 3 次元実装をリファレンスとした。

2.1.4.2　MN-Core 後継のベースラインアーキテクチャの決定

ここまでのアーキテクチャ、メモリ階層の検討結果を踏まえ、本調査研究でのアプリケーション評価、アーキテクチャ評価のリファレンスとなるアーキテクチャを策定した。

電力性能については、MN-Core は TSMC の 12FFC プロセスで製造され、ボードレベルで FP64 演算に対して 70 GF/W の性能を実現している。従って、単純に TSMC のノミナルな低電力化率

を使うと、N2 プロセスで 434 GF/W、N1.4 では 620 GF/W となり、アーキテクチャ改良なしで GPU 等の世界のトレンドよりも高い電力性能が実現できることになる。実際には動作電圧の設定等の問題があり、ここまで上がることはありそうにないが、ここから非常に大きくずれるものではない。従って、いくつかのアーキテクチャ改良を加えることで、FP64 で 900-1000 GF/W を N1.4 プロセスで実現することは可能と考える。

ベースラインアーキテクチャのパラメータは以下のようになる。これは N2 世代を想定したものである。

1. 演算器は倍精度 (単精度、FP16、BF16/Int8、FP8/4) の行列ベクトル乗算ユニット
2. 1 パッケージ当たりの演算ユニット数は MN-Core の 6 倍、クロックは 2.5 倍、性能は 15 倍 (FP64 500 TF)
3. FP8 性能は FP64 性能の 64 倍。32 PF
4. 消費電力は 1000 W 以下
5. 外部メモリは 3 次元実装 DRAM、容量 256–384 GB、バンド幅 100 TB/s (B/F = 0.2)
6. 大容量 DRAM はホスト CPU 側を想定する
7. 汎用 CPU インターフェイスは PCIe gen7(物理実装は UCIe)、バンド幅は片側 1 TB/s
8. 汎用 CPU 側に HBM または LPDDR6 による大容量メモリ。容量 256 GB 程度、バンド幅 1–2 TB/s
9. ホスト CPU 消費電力最大 150W、128 コア、3 GHz、256 bit SIMD2 or 単一の 512 bit SIMD ユニット、FP64 12 TF、B/F = 0.08–0.16

他社の現状や次世代への外挿に比べて、

- パッケージあたりのメモリバンド幅は 5 倍程度
- 電力あたりのメモリバンド幅は 15 倍程度
- 電力あたりの FP64 性能は 20 倍程度以上
- 電力あたりの FP8 性能は 4 倍程度以上

となる。非現実的に高い数値に見えるかもしれないが、電力あたりの演算性能についてはすでにみたように MN-Core シリーズからのそれほど無理のない外挿となっている。一方、電力あたりのメモリバンドは HBMx を使ったアーキテクチャに比べて 1 桁以上高い。これが現実的かどうかを以下で検討する。

アクセラレータの外部メモリは、プロセッサコアのローカルメモリの延長として実装される。すなわち、キャッシュ階層ではなく、プロセッサコアが直接アクセス可能な単一の DRAM ブロックが各プロセッサコアに直結するものとする。

2.1.4.3 3D 実装メモリの検討

本 FS ではまずは DRAM ベンダに 3DDRAM マクロの設計、電力シミュレーションを委託し、実現性と性能について様々な面からの検討を行った。想定した構成は、DRAM ベンダが提供するマ

クロを 2 次元的にしきつめたセミカスタム DRAM とロジックダイのハイブリッドボンディングによる接合である。DRAM は高密度 (現在 10 μm ピッチ、ポスト「富岳」時点では 5 μm ピッチ) の TSV で 4 ないし 8 層程度スタックされるとした。現在 HBM 等で使われている TSV は 50 μm 程度のピッチであり HBM ダイ当たりパッド数は数千程度だが、ハイブリッドボンディングではさらに 2 桁程度多いパッドが実現できる。このため、HBM では必要になっている DRAM ダイ上でのシリアル・パラレル変換が不要になり、また多数の小さなマクロ毎に I/O パッドを置くことで DRAM 上でもロジックダイ上でもデータ移動を小さくできる。この結果、連続アドレス（同一ページ内) リードについては現在の製造技術で 0.6 pJ/bit 以下のアクセスエネルギーが実現可能という結果が得られた。4 pJ/bit 程度である HBM3/3e に比べて極めて大きなエネルギーの削減である。

実現性については以下の点を検討した。

1. 構成方法。具体的には、基板側にくるのがロジックダイか DRAM ダイか
2. パッケージ外に引き出す信号の SI (Signal Integrity)
3. 電源供給方法と PI (Power Integrity)
4. 冷却方法
5. DRAM および 3D 接合の欠陥発生率と冗長設計手法

以下これらについて検討結果の概要を述べる。

2.1.4.3.1 構成方法

DRAM を含めた 3D 積層の構成方法として、単純には

a) ロジックダイを下 (DRAM とパッケージのサブストレートの間) に置く
b) DRAM を下 (ロジックダイとパッケージのサブストレートの間) に置く

の 2 とおりがあり得る。近年の CPU や GPU では、SRAM ダイとロジックダイの集積が使われるケースが多くなっているが、例えば AMD の Zen 4 CPU では a)、同じ AMD の MI-300X GPU では b)、Intel MAX GPU では a) と b) どちらも使われている。ただし、Zen 4 CPU では「Face down」と呼ばれる、配線層が基板側になるアプローチがとられており、SRAM ダイとプロセッサダイの接続はプロセッサダイに TSV をあけている。この TSV のピッチが比較的大きいためか、3D 接続が可能なように設計された Zen 4 コアはこの機能を削除した Zen 4c コアに比べて 1.5 倍程度の面積になっている。

Zen5 では、メモリダイが下、ロジックダイが上になる構成になった。また、富士通 MONAKA でも、同様にメモリダイが下、ロジックダイが上である。

一般には、a) を採用すると以下の問題が発生する。

a1) ロジックを Face up で実装する必要があるため、電源、信号等の基板側にいく全てのパッドが TSV で実現される必要があり、ダイエリアを消費する。また、SERDES や PCIe 等の高速信号では、負荷として加わる TSV とそこまでの配線を考慮する必要がある。元々の IP 自体

が Face down の実装を前提にパッド配置を決めているので、物理実装の新規開発が必要になる可能性もある。

a2) DRAM ダイがロジックダイとヒートシンクの間に入るため、ロジックダイの温度が上昇する。

a3) DRAM の電源供給もロジックダイの TSV を経由するため、ロジックダイのエリアを消費する。

一方、b) を採用すると以下の問題が発生する。

b1) ロジックダイと基板間の全ての信号が積層した DRAM の TSV 経由となるため、そのインピーダンスを考慮する必要がある。

b2) ロジックダイの電源のために DRAM ダイに非常に多くの TSV が必要となり、エリアを消費する。また、カスタム設計が必要になる可能性がある。

b3) DRAM の主な冷却パスがそれ自体高温なプロセッサダイ経由となるため、DRAM の温度が非常に上がり、安定動作が困難になる可能性がある。

これらについて、シミュレーションによって検討した。詳細は以下の項目で述べる。

2.1.4.3.2 パッケージ外に引き出す信号の SI

上の a)、b) 双方について spice シミュレーションによる評価を行った。シミュレーションでは、a) の場合には追加されるロジックダイ上の配線と TSV、b) の場合には追加されるロジックダイ上の配線と DRAM ダイ上の TSV を加え、基板上の配線までを考慮したシミュレーションを行った。信号としては PAM4 を採用した PCIe gen6 の信号を例にした。

結果として、どちらの場合でも、TSV の配置等を配慮すれば十分な信号品質が得られることがわかった。これは、逆にいうと、この配慮が必要ということであり、b) の場合にロジックの高速信号については十分なケアが必要ということだが、原理的な問題はないことが示された。

2.1.4.3.3 電源供給方法と PI

ロジックダイの消費電力が圧倒的に大きいため、b) のケースについて検討を行った。今回検討したマクロの場合には、TSV に使用できるエリアほぼ全てを使い、DRAM 信号以外を基本的にすべて電源とグラウンドに使っても、DRAM TSV での電圧降下がまだ大きいという結果になった。これは、ロジックダイのコア電圧 0.4 V、消費電力 800 W で電流 2000 A を想定した場合である。原理的には TSV エリアと数を増やすことで対応可能ではあるが、a) のアプローチのほうが好ましいといえる。

TSV の抵抗値は構造に非常に大きく依存するため、電源向けの TSV を開発することで解消する問題ともいえる。

具体的には、TSV が Si のベース層から最下層のメタルまでで、配線層はビアで最上層メタルまでもってきている構造の場合、特に古い DRAM プロセスではビアの抵抗が非常に大きい。これは

ビアがタングステンであること、ビアの断面積が小さいこと等による。言い換えると、配線層も貫通する TSV にする、あるいはビアの面積を増やす、といったことで大きく TSV の抵抗を下げることは可能である。

ただし、ここまで面積あたりの消費電力を増やす必要はなくなってきている。これは、ダイやプロセッサの製造コストとライフタイムの電力コストを比較した時、1 kW では 5 年使えて 30 円/kW とすれば電力だけで 150 万円、データセンターや電源ユニットのコストを考えるとその数倍になるため、これはシステムコストよりも大きいからである。このため、電力性能を上げることが最重要になってきており、面積あたりの性能を上げる必要は薄れてきている。

2.1.4.3.4 冷却方法

ロジックダイの消費電力を 800 W 程度とした場合、発熱が一様という単純な仮定の下では a) のケースで DRAM が 4 層の場合のロジックダイの追加の温度上昇が 5.1 K となった。これ自体は十分小さいが、発熱が完全に一様ではないことには注意が必要である。MN-Core の場合には、通常の CPU とは異なり、演算器とそれに接続したローカルメモリがチップエリアの大半を占めるため、発熱は一様に近いが、PCIe 等の IP では局所的に発熱が高いエリアが存在する。これらについてより詳細な熱シミュレーションが必要であるが、基本的には大きな温度上昇が起こると考えられ、ロジックダイを下に置く構造では冷却は困難であると考えられる。

b) のケースではこの問題はなく、ロジックダイの消費電力が非常に大きい場合には b) のアプローチが必須であるといえる。

2.1.4.3.5 DRAM および 3D 接合の欠陥発生率と冗長設計手法

高密度のハイブリッドボンディングを使うためには、現在のところで各社とも WoW(Wafer-on-wafer) の工程を必要としている。ポスト「富岳」の時期には CoW(Chip-on-wafer) や CoC(Chip-on-chip) の工程も可能になってくる可能性があるが、ここでは WoW を前提に欠陥発生率とその対応についての検討を行った。

欠陥は、基本的に以下の 4 箇所で発生するものと考えられる。

1. ロジックダイ自体
2. DRAM ダイ自体
3. DRAM ダイの TSV 加工と DRAM ダイ間のハイブリッドボンディング
4. ロジックダイと DRAM ダイのハイブリッドボンディング

WoW の場合、ロジックダイ、DRAM ダイに事前に十分なテストをするのは困難であるため、単純なやり方では総合的な良品率がこの 4 箇所での良品率の積になってしまい、非常に悪くなる。従って、これを防ぐ必要がある。

防ぐための基本的なアプローチは冗長設計である。MN-Core の場合には、すでに PE レベルで冗長設計を行っており、ある程度の良品率の改善を得ている。また、SRAM マクロ等ではマクロ内でも冗長設計が採用されている。DRAM についても、マクロ内での冗長性とマクロ自体の冗長

性の両方を適用する。

問題はハイブリッドボンディングであるが、これについては、DRAM の制御線については複数の TSV を 1 シグナルに割り当てることでエラーレートを減らし、本数の多いデータ線については代替線を用意することでエラーレートを減らすことで対応可能であるという結論になった。

Cu マイグレーション等についても、寿命に対して TSV の電流上限値が求められており、十分に長い寿命が得られるとわかった。

2.1.4.3.6 まとめ

3DDRAM について、実際にハイブリッドボンディングでのテストチップ製造経験のあるベンダにシミュレーション等の評価を委託して、設計上の問題点について検討を行った。ポスト「富岳」のような極めて発熱が大きい場合には、ロジックダイが上 (基板と反対側) にくるアプローチが望ましいが、この場合には DRAM ダイを通した電源供給について TSV の増強等の配慮が必要であること、また PCIe や SERDES 等の高速信号についても事前の検討が必要であることが確認できた。また、WoW の製造方法をとることによる不良品率の増加については、基本的に各部での冗長設計により対応するべきという結論になった。なお、発熱分布等についてさらに検討が必要であるが、ロジックダイが下にくる方式を否定するものではない。

これらから、DRAM についてセミカスタム設計に対応するベンダであれば、WoW による超多ピン接続で、20 TB/s を超える高いバンド幅と 0.6 pJ/bit を下回る低い消費エネルギーを両立させることは可能であるといえる。しかしながら、本 FS の範囲内では当初計画していた試作による評価は行うことができなかったので、良品率、信頼性についての評価は十分とは言えない。

2024 年から、PFN において、LLM 推論をターゲットとして実際に 3D 積層 DRAM を採用したプロセッサの開発を始めている。ポスト「富岳」で想定しているような最先端プロセスではないが、 LLM 推論用途では十分な演算性能と非常に高いメモリバンドを実現したプロセッサを実現する計画である。

また、現在のところ DRAM についてセミカスタム設計に対応するベンダは最先端の DRAM プロセスをもつファブではなく、ニッチ向けの供給を行っている二線級のファブである。このため、消費電力はともかく、面積当たりの容量があまり大きくない、という問題が現状ではある。これについては、これらのファブのプロセス改善と、最先端の DRAM ファブのこのような設計への対応の両方について、ある程度の進展は期待できると考える。

2.1.5 全体性能の検討

MN-Core 後継のキャッシュをもたないローカルメモリアーキテクチャと、3D 積層 DRAM を組み合わせることで、表 2.1.8 に示すような構成が可能と評価された。

1 ノードの特徴としては、2029 年頃に想定される他のアーキテクチャでの構成に比べて、1 ノードあたりの FP64 演算性能、メモリバンド幅の双方で大きく上回り、電力は同程度ないしはそれ以下になることで大幅に電力性能を向上させていること、また 1 ダイ当りの性能が高いために価格性能比も良好であることがあげられる。

表 2.1.8　ノード性能

CPU FP64 性能	6 TF
CPU メモリバンド幅	1 TB/s
CPU メモリ容量	512 GB
アクセラレータ FP64 性能	512 TF
アクセラレータ FP16 性能	8 PF
アクセラレータ FP8 性能	32 PF
アクセラレータメモリ容量	512 GB
アクセラレータメモリバンド幅	128–256 TB/s
消費電力	1.5 kW

2.2　CPU 評価

塩谷 亮太 [国立大学法人 東京大学]

2.2.1　背景

高性能計算アプリケーションは一般に、並列化可能でアクセラレータにより高速に実行できる部分と、並列化が難しくシーケンシャルに汎用 CPU で実行される部分を含む。アムダールの法則として知られるように、プログラムの並列化を進めていくと最終的に並列化が難しいシーケンシャルな部分が残り、実行時間に対して支配的となる。したがって、計算機システムとしては処理の大半を担うアクセラレータに加え、シーケンシャルな処理を行う CPU の性能も重要となる。

そのような背景に基づき、本調査研究では高性能計算アプリケーションを対象とし、CPU の調査と評価を行った。具体的には RISC-V 等による独自実装の汎用 CPU および商用 CPU の評価のため、主にシミュレーションによるアプリケーション実行時性能評価を行った。特に、現在搭載を検討しているいくつかの企業で開発中の RISC-V CPU について性能解析を中心に評価を行い、現在主流である x86-64 CPU と比較して十分な性能が達成できるかを検討した。

2.2.2　RISC-V CPU

本評価において RISC-V CPU を主に評価したのは、低コストでアクセラレータと密結合したシステムを作ることができるためである。現在高性能 CPU の市場で支配的な x86-64 CPU は、IP (Intellectual Property) として設計が提供されることはなく、基本的には完成したチップのみが供給されている。このため、アクセラレータとは外部バスを通じて通信せざるを得ず、通信バンド幅やレイテンシにボトルネックを生じる。別の代表的な高性能 CPU である ARM CPU は IP としても提供されているものの、その価格は非常に高く、また内部を独自にカスタマイズする事は基本的にできない（あるいは極めて高額なライセンス料を必要とする）。

これに対し、RISC-V CPU は比較的柔軟な形で設計やチップが提供されており、低コストで独自のシステムを作成する観点では非常に魅力的である。また、RISC-V CPU では、さまざまなオープンソース設計も存在する。これらは完全に設計が公開されている事からコスト面で非常に有利なことに加え、柔軟なカスタマイズも可能となる。実際に、そのようなオープンソース設計に基づいて製造された商用チップも登場し始めている。

一方で、RISC-V CPU のハードウェアやそのソフトウェアスタックはまだ十分に成熟しているとは言えず、実際に高性能計算等に用いた場合の性能はよくわかっていない。そこで本評価では代表的な RISC-V CPU のベンダである Tenstorrent 社と SiFive 社の協力を得て、それぞれの提供している CPU の性能についての調査や、一部データ提供による我々独自の評価・解析を行った。以下ではこれらのうち、主に性能解析の結果について述べる。

2.2.3 評価内容

評価ではプログラム内に含まれるシーケンシャルな処理を抽出し、それらを CPU で実行した場合の性能を評価・比較した。この評価では大規模並列粒子法シミュレーションのための汎用高性能ライブラリである FDPS（https://github.com/FDPS/FDPS）を用いた。FDPS を対象としたのは、以下の理由による：

- ライブラリ等への依存関係が少ないため、発展途上の RISC-V のソフトウェアスタックでもバイナリの生成が容易である。
- 並列化可能な部分とシーケンシャルな部分をそれぞれ適度に含んでいる。

FDPS は C++で記述されており、これを各評価環境向けにコンパイルして性能を評価した。まず FDPS の実行をプロファイラによって解析し、プログラムの実行時間に対して支配的な関数を探した。それらの関数の中からアクセラレータで実行可能な部分と、シーケンシャルな部分を分類し、後者を評価対象とした。

比較評価対象の CPU としては、x86-64 CPU については、現在最も高速な実装の 1 つである AMD 社のものを複数種類用いた。これらは実際に AMD 社の CPU を搭載している計算機を用いてプログラムの実行と各種の測定を行った。また、RISC-V CPU については企業と交渉の結果、プログラム内の各命令がリタイアしたサイクル数の系列（トレース）を得ることができたため、これを解析して x86-64 CPU における実行結果の該当する部分を割り出し比較した。

2.2.4 評価における課題と解決

この評価では以下のような点が課題となる：

1. x86-64 CPU と RISC-V CPU の間では、駆動周波数や同じ処理を行うために必要な命令数が違うため、単純に実行時間を比較するだけでは意味がある比較とならない。
2. 一般に、CPU において関数のような小さい粒度で、実行サイクル数を正確かつ低オーバーヘッドに測定することは簡単ではない。

3. 各関数は呼び出し毎に実行命令数が大きく変化するため、動的に同じ呼び出し同士で比較を行う必要がある。
4. 機密保持の観点より、各企業において開発中の CPU 上での測定を我々が直接行う事ができない。

上記の課題に対し、まずプログラムに対して一定の変更を行い、測定対象となる関数の呼び出し回数と実行サイクル数を x86-64 CPU 上で軽量に取得できるよう一定のコードを挿入した。この測定のために挿入するコードは、当初は Linux システムコールを介してパフォーマンスカウンタの値を取得するものを使用していたが、その後の評価により、使用するカーネルのバージョン等に依存してサイクル数等がかなり変動することがわかった。このため、パフォーマンスカウンタに直接アクセスを行うなどの軽量な方法で、より精度の高い測定を行う方法を検討した。

x86-64 CPU については実機上の実行やプログラムの改編が自由に行えるため、柔軟な測定が行えた。一方で、RISC-V CPU ではトレースの提供のみが行われたため、これを解析することで必要な情報を得た。具体的には、命令アドレスの遷移や消費サイクル数などを手動で解析し、対応する関数呼び出しを割り出した。また、我々が独自に開発をしてきたパイプライン可視化ツール Konata による可視化を行い、それによる性能解析も行った。

これらにより、上記の課題を解決し、該当する関数同士を適切に比較できるようになった。

2.2.5 評価結果

上記の解析や評価の結果、以下のことがわかった。

- Tenstorrent 社および SiFive 社の RISC-V CPU の性能は、2025 年の時点では、AMD 社の x86-64 CPU の性能には総合的には及んでいない。しかし、高性能計算においてシーケンシャルな部分を処理する CPU として見た場合には良好な性能を持ち、RISC-V CPU がボトルネックとはならずに十分な性能を提供できていることがわかった（両社における機密保持の観点より、詳細な性能差などの分析結果はここでは省略する）。
- 特に、Tenstorrent 社の実装はクロックサイクルあたりの実行命令数の観点で、AMD 社の Zen4 世代を含む x86-64 CPU よりも相当に高い結果を示している。このため Tenstorrent 社の実装は比較的低い動作周波数でも高い性能が達成できる見込みであり、電力効率上でかなり有利であることが期待できる。また、CPU とアクセラレータ間のデータ転送に必要な時間が大きく改善されることを考慮すると、両者が密結合されたシステムを構成することで大きな性能向上を得られる可能性がある。
- 定性的な内部構造等の比較検討も行ったが、各社の CPU の性能の傾向の違いは、AMD 社（と Intel 社の）x86-64 CPU が 5 GHz〜6 GHz の極めて高いクロック周波数を達成するために、同時実行命令数や各種のバッファのサイズなどを（相対的に）あまり大きくしないよう設計している事に主に起因すると推測している。この結果、それらの CPU はクロックサイクルあたりの実行命令数がある程度抑えられてしまっている。一方で、Tenstorrent 社の CPU は、よりクロックサイクルあたりの実行命令数を高める方向に設計方針を振っているようであり、その違

いが評価結果に表れたものと推定している。

2.3 アクセラレータアーキテクチャ評価

小泉 透 [国立大学法人 名古屋工業大学]

2.3.1 背景と解決すべき課題

既存の多くのアクセラレータは、複数の精度（例えば、FP64、FP32、FP16、FP8、INT8 など）での演算に対応している。独自アクセラレータも、AI 向け（推論および学習）および科学技術計算向けに、同様に複数の精度に対応する必要がある。一方で、この目的を達成する優れた構成方法は明らかではない。本調査では、これら複数の精度向けの行列積演算器回路を省面積・省電力で実装する方法について調査した。

2.3.2 複数の演算の間での共有による省面積省電力化

行列乗算においては、複数の異なる値に対して同じ値を掛ける操作があり、この特性を生かすと通常の演算器を単純に並べただけよりも効率の良い実装が可能となる。具体的には、Booth エンコーディングを行う部分の共有や、Radix-8/Radix-16 Booth 乗算器で必要となる被乗数の 3 倍・5 倍・7 倍回路などの共有が可能となる。Booth エンコーディングは既存の乗算器でも用いられる技法であるが、行列演算器のような高度な共有が行われる場合は優劣のバランスが変化するため、検討しなおす必要がある。

Galal *et al.* (2013) の浮動小数点数演算器構成方法の研究では、符号なし二進乗算器、符号なし四進乗算器、Radix-4 Booth 乗算器、Radix-8 Booth 乗算器、Radix-16 Booth 乗算器、について調査されている。この 5 種類について、乗算器の幅を n とした時 n^2 のオーダーを持つ項である、部分積あたりに必要なトランジスタ数を調査すると表 2.3.1 のようになる。Booth 乗算器は符号ビットが必要となることから、n のオーダーを持つ項のコストが大きいことが知られている。したがって、行列乗算器には符号なし四進乗算器が最も適していると結論付けた。

乗算器構成方法	部分積あたりトランジスタ数（n^2 項）
符号なし二進乗算器	28 トランジスタ/部分積
符号なし四進乗算器	18 トランジスタ/部分積
Radix-4 Booth 乗算器	20 トランジスタ/部分積
Radix-8 Booth 乗算器	18 トランジスタ/部分積
Radix-16 Booth 乗算器	18.5 トランジスタ/部分積

表 2.3.1　乗算器構成方法ごとの部分積あたりトランジスタ数（n^2 項）

2.3.3　複数の精度の間での共有による省面積化

乗算器（部分積生成回路および加算器木回路）は、精度ごとに配置するのではなく、複数の精度で共有できれば全体の回路面積を小さくすることができ、配線遅延・配線での消費電力・チップ面積、をいずれも小さくできるため好ましい。これは例えば、10 bit 乗算器を単独で配置するのではなく、5 bit 乗算器 4 個と多少の追加回路により構成するべきであるということである。ただし、多数の精度間で完全に共有することはできないから、適宜選択器を挿入する必要がある。この選択器を挿入した分だけ、精度毎に回路を作成した場合よりも消費電力が増大してしまうから、この選択器の挿入は最小限にとどめる必要がある。

基本的には、選択しなければならない信号線の本数が少なくなる、加算器木の後段で選択するように工夫すべきであり、またそのようなことが可能であると結論付けた。具体的には、低精度乗算器を組み合わせて高精度乗算器を作成する際、低精度演算での内積方向加算（$a_1b_1 + a_2b_2 + a_3b_3 + a_4b_4 + \cdots$）と、高精度演算での同じ桁位置の加算（$a_1^{hi}b_1^{lo} + a_1^{lo}b_1^{hi} + a_2^{hi}b_2^{lo} + a_2^{lo}b_2^{hi} + \cdots$）が共有できるようなレイアウトを考えればよい。これを自動的に行うアルゴリズムは作成できていないが、手作業でいくつかのレイアウトを試すことで、実用上ほぼ最適なレイアウトが得られるようである。

2.3.4　パイプライン段数削減による省電力化

各パイプラインレジスタを構成するフリップフロップは、演算回路と比較可能なほど電力を消費するから、演算のレイテンシを削減してパイプライン段数をなるべく削減することが肝要である。一方で、パイプライン段数を削減する既存手法はいずれも、浮動小数点数演算回路の面積を大きく増大させてしまうから、その採用が全体の消費電力を削減するかはよく検討する必要がある。

パイプライン段数を削減する技術の中でも、leading zeros anticipation (LZA) と呼ばれる技術に着目した。この技術は、加算結果を確定させる前にその先行ゼロカウントを 1 以内の誤差で予測するものであり、加算結果を確定させてから先行ゼロカウントを行うのと比べ遅延を削減できる。一方で、その回路は高速加算器と同程度であり、回路規模の増大が問題となりうる。しかしながら、積和演算器と違い行列演算器では、加算器木が内積長（例えば 8、16、32 など）倍だけ大きいのに対し、LZA 回路の大きさは変わらないから、LZA を導入した場合の回路規模の増大の影響は 1/8、1/16、1/32 などとなり、その影響は相対的に小さくなる。したがって、LZA を導入すべきと結論付けた。

2.4　ネットワークアーキテクチャ評価

鯉渕 道紘 [大学共同利用機関法人 情報・システム研究機構 国立情報学研究所]
平澤 将一 [大学共同利用機関法人 情報・システム研究機構 国立情報学研究所]

2.4.1 性能要求

計算ノードのリファレンス設計に対して、ネットワーク側に必要な性能・機能を見積った。まず HPL と HPCG ベンチマークを対象とした。HPL では、高い実行効率を得るために必要なネットワークバンド幅 (縦横合計) は次式で概算される。

$$b > b_{min} = 32c/(x^{1/2}) \tag{2.4.1}$$

b はバイト/秒単位のネットワークバンド幅、c は 1 秒あたりの演算回数、x はバイト単位の主記憶サイズである。

ここで、$c = 256$ TFLOP、 $x = 128$ GB とした場合、

$$b_{min} = 22.9 \text{ GB/s} \tag{2.4.2}$$

となる。つまり、相互結合網を 2 次元格子にマップした時に、4 つのリンクのそれぞれの転送バンド幅が 11.5 GB/s 以上必要ということになる。つまり、合計で約 50 GB/s である。これは直接網の場合に該当する。

HPCG では主要な通信は袖交換である (マルチグリッド法なのでレイテンシもある程度重要だがここでは省略する)。参照アーキテクチャにおいてアクセラレータ主記憶が 32 GB とすると、約 $(500)^3$ 程度のグリッドが入る大きさである。B/F = 0.1 を仮定すると実行効率は 1.6%となり、この時の 1 イテレーション (最大格子サイズ) の実行時間は 1.2 ミリ秒となる。袖のサイズは 12 MB 程度なので、通信時間を計算時間の 1/3 の 0.4 ミリ秒として必要なネットワーク速度は 6 方向合計で約 30 GB/s となる。

2.4.2 通信データの圧縮技術

必要となるネットワークバンド幅を抑えるために、アーキテクチャレベルで通信データの圧縮を行い、通信量を削減する実装が最近登場している。例えば、2 倍の圧縮率が得られた場合、必要となるネットワークバンド幅を半分に抑えることができる。Anton3、FPGA 専用計算機では、対象とするアプリケーション (例：Molecular Dynamics(MD) シミュレーション) の通信データの内容に局所性が強いことを利用して、実際に転送するデータ転送量の削減に成功している。Anton3 の相互結合網では、先頭ビットから連続して “0” が生じるように INZ(interleaved non-zero encoding) を採用し、そのビットを除去して転送することでデータ量を削減している。佐野らの FPGA ネットワークの研究では、2 次元 Lattice Boltzmann Methods (2D-LBMs) において同一送受信間の通信において、前後の転送データの依存関係を利用して高い圧縮率を得ることで、1.7 倍の性能向上を達成したことが報告されている。これらの通信データの圧縮は、(1) データが 32 ビット表現である点、(2) アプリケーションで発生する通信データの局所性を利用したシンプルな可逆圧縮方式を採用している点が共通している。一般的に、64 ビットの浮動小数点数を対象とする可逆圧縮アルゴリズムは処理が複雑である。そのため 1 マイクロ秒以下の通信時間が前提の相互結合網に導入でき

る技術は限られている。特に Anton3 の相互結合網は 1 hop あたり 55 ナノ秒と極限まで小さな通信遅延である。

2.4.3 ネットワークトポロジと機能

大域通信としてもっとも重要なのは放送と縮約である。京、富岳で採用されている Tofu ネットワークでは、縮約操作 (MPI_ALLreducde) が京 9,216 ノードにおいてメッセージ 16 KiB の時に 50 MB/s、 32KiB の時に 100 MB/s の実効速度が得られている。すなわち、これらのメッセージサイズでレイテンシは 320 マイクロ秒である。一方、Mellanox のスイッチを使った 1,500 ノードのネットワークでは、Mellanox の ISC2019 での発表によれば、スイッチ側で縮約を行う SHARP を使わない場合に 60 マイクロ秒程度、 SHARP を使うと 10 マイクロ秒程度となり、ネットワーク演算処理を行うことで顕著な高速化を達成している。また、このレイテンシの差は、Tofu ネットワークにおいてネットワークの直径が Mellanox などの間接網に比べて大きいことが影響していると思われる。

2.4.4 ネットワークトポロジと光電融合

光サーキットスイッチ (OCS) により、ネットワークトポロジを動的に更新する技術が研究開発され、実用化されつつある。OCS は新たなトレンドであり直接網、間接網のネットワーク設計に影響を与える可能性がある。これまでの相互結合網は、InfiniBand を用いた間接網やカスタム設計した直接網が用いられることが多かった。しかし、本必要帯域であれば、光電融合技術が現実的に利用可能である。具体的には、光サーキットスイッチ (OCS: Optical Circuit Switch) と電気ネットワークを統合した光電ネットワーク構成が盛んに実用化されている。Noctua 2 スーパーコンピュータ (CALIENT S320 OCS) (Paderborn Center for Parallel Computing 2024)、Google TPUv4 クラスタ (Jouppi *et al.* 2023; Liu *et al.* 2023)、Jupiter データセンターネットワーク (Poutievski *et al.* 2022) が挙げられ、多くの OCS の試作機が研究開発されている (Ballani *et al.* 2020; Mellette *et al.* 2017; Wang *et al.* 2023)。いずれの OCS も上記の要求帯域を満たしている。

OCS を用いた HPC 向け相互結合網は、図 2.4.1 に示した 3 とおりに分類されるが、近年は電気パケットスイッチ (EPS: Electric Packet Switch) と OCS を相補完的に使うハイブリッド型 (図 2.4.1) が主流である。そこで、ハイブリッド型を主に検討した。

図 2.4.2 に光サーキットスイッチを導入した研究開発の分類を示す。本図において「ネットワークトポロジ」は元となる電気スイッチのネットワークトポロジを表している。同「専用 OCS (Optical Circuit Switch) デバイス」とは、3 次元 MEMS や光パッチパネルなどのコモディティとして商用化されている光サーキットスイッチデバイス以外のデバイスを指す。なお、ハイブリッドネットワークにおいて直接網と間接網の区分は、元となる電気ネットワークの構成に基づく。これは、光サーキットスイッチは、実際にトラヒックが流れる前に回線を設定するため、アプリケーション側からはリンクの一部のように見えるためである。

Palomar 光サーキットスイッチ (Liu *et al.* 2023)、2022 年発表の Jupiter (Poutievski *et al.* 2022)、Tale of Two (Wang *et al.* 2023) は、典型的な Fat ツリー構造の Google データセンターにおいてジョブ内通信遅延を削減する構成を可能とする研究成果である。一方、同目的を達成しつつコスト

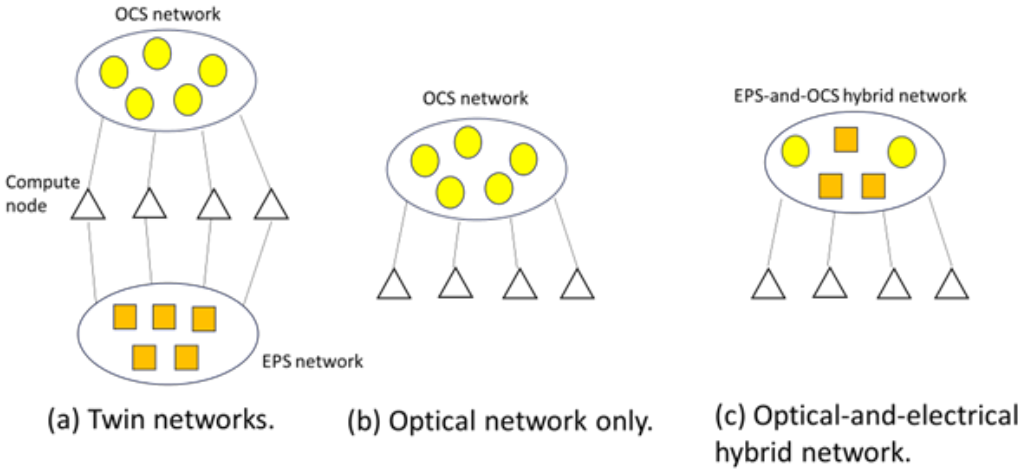

図 2.4.1　光サーキットスイッチと電気パケットスイッチを用いた結合網の分類。

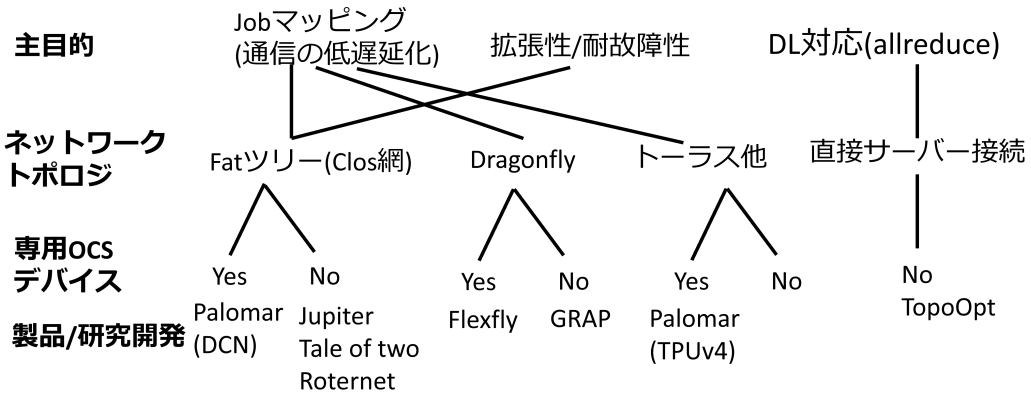

図 2.4.2　光サーキットスイッチを利用したデータセンターと並列計算機の分類。

を抑えるべく、直接網の Dragonfly ネットワークトポロジを対象にした研究である Flexfly (Wen *et al.* 2016) などがある。同様にトーラスにおいてサブトポロジの embedding を容易にすることで通信遅延を削減する目的を達成することができる TPUv4 (Liu *et al.* 2023) などが報告されている。また、Allreduce 処理を高速化するために光サーキットスイッチのみでネットワークを構成する TopoOpt が提案されている。これらにおいて専用光サーキットスイッチを用いる理由は、商用光サーキットスイッチと比べて故障率の改善、光サーキットスイッチのスイッチ時間の短縮の 2 点が挙げられる。なお、いずれの光サーキットスイッチもポート数が 100 を越える高次元化を達成しており、現実的に大規模結合網の構築が可能となっている。一方で課題としては、去年の報告書でも簡単に指摘したとおり、通信パターンの予測性、リアルタイム性への対応とそのためのソフトウェア開発が依然、挙げられる。

図 2.4.2 においてネットワークトポロジがトーラス、専用 OCS デバイスの利用を想定していない区分にあたる研究開発が空欄である。しかし、異なる通信パターンを有するアプリケーションを 1 つの並列計算機に低コストでサポートする場合、この空欄となっている部分を埋める研究開発が

重要となるであろう。特に通信パターンが予測できない場合、Tale of Two (Xia *et al.* 2017) のように局所的にランダム性を導入して、各パケットが経由する平均電気パケット数を削減して通信遅延を削減することが期待できる。

以上の検討に基づき、OCS を用いた相互結合網の開発として 2 つの方向が考えられる。

- 汎用 MEMS OCS スイッチを用いた相互結合網の開発、および実運用での安定性・性能向上を目的とした応用手法の研究開発
- プロトタイプ実装やシミュレーションに基づく、高速な再構成が可能なカスタム化 OCS を用いる相互結合網のピーク性能の向上を図る研究開発

第 2 章の参考文献

Ballani, H., P. Costa, R. Behrendt, D. Cletheroe, I. Haller, K. Jozwik, F. Karinou, S. Lange, K. Shi, B. Thomsen, and H. Williams (2020). “Sirius: A Flat Datacenter Network with Nanosecond Optical Switching”. In: *the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 782–797.

Galal, S., O. Shacham, J. S. Brunhaver II, J. Pu, A. Vassiliev, and M. Horowitz (2013). “FPU Generator for Design Space Exploration”. In: *2013 IEEE 21st Symposium on Computer Arithmetic*, pp. 25–34.

Jouppi, N., G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. A. Patterson (2023). “TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings”. In: *the 50th Annual International Symposium on Computer Architecture*.

Liu, H., R. Urata, K. Yasumura, X. Zhou, R. Bannon, J. Berger, P. Dashti, N. Jouppi, C. Lam, S. Li, E. Mao, D. Nelson, G. Papen, M. Tariq, and A. Vahdat (2023). “Lightwave Fabrics: At-Scale Optical Circuit Switching for Datacenter and Machine Learning Systems”. In: *the ACM SIGCOMM 2023 Conference*, pp. 499–515.

Mellette, W. M., R. McGuinness, A. Roy, A. Forencich, G. Papen, A. C. Snoeren, and G. Porter (2017). “RotorNet: A Scalable, Low-complexity, Optical Datacenter Network”. In: *the Conference of the ACM Special Interest Group on Data Communication*, pp. 267–280.

Paderborn Center for Parallel Computing (2024). “Noctua 2 Supercomputer”. In: *Journal of large-scale research facilities* 8, A187. DOI: 10.17815/jlsrf-8-187.

Poutievski, L., O. Mashayekhi, J. Ong, A. Singh, M. Tariq, R. Wang, J. Zhang, V. Beauregard, P. Conner, S. Gribble, R. Kapoor, S. Kratzer, N. Li, H. Liu, K. Nagaraj, J. Ornstein, S. Sawhney, R. Urata, L. Vicisano, K. Yasumura, S. Zhang, J. Zhou, and A. Vahdat (2022). “Jupiter evolving: transforming google’s datacenter network via optical circuit switches and software-defined networking”. In: *Proceedings of the ACM SIGCOMM 2022 Conference*. SIGCOMM ’22. Amsterdam, Netherlands: Association for Computing Machinery, pp. 66–85. ISBN: 9781450394208. DOI: 10.1145/3544216.3544265.

Wang, W., M. Khazraee, Z. Zhong, M. Ghobadi, Z. Jia, D. Mudigere, Y. Zhang, and A. Kewitsch (2023). “TopoOpt: Co-optimizing Network Topology and Parallelization Strategy for Distributed Training Jobs”. In: *NSDI*, pp. 739–767.

Wen, K., P. Samadi, S. Rumley, C. P. Chen, Y. Shen, M. Bahadori, K. Bergman, and J. Wilke (2016). “Flexfly: enabling a reconfigurable dragonfly through silicon photonics”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

Xia, Y., X. S. Sun, S. Dzinamarira, D. Wu, X. S. Huang, and T. S. E. Ng (2017). "A Tale of Two Topologies: Exploring Convertible Data Center Network Architectures with Flat-tree". In: *ACM Special Interest Group on Data Communication*. SIGCOMM, pp. 295–308.

第 3 章　システムソフトウェア・ライブラリ調査研究

3.1　システムソフトウェア・ライブラリ調査研究の統括およびコンパイラ検討

中里 直人 [公立大学法人 会津大学]

2022 年 10 月より開始となった本調査研究では、MN-Core をアクセラレータとする大規模並列コンピュータシステムの構築を構想していた。システムソフトウェア・ライブラリ調査研究のグループでは、MN-Core のプログラミングシステムおよびライブラリについて検討をおこなった。

3.1.1　調査研究開始時の状況

調査研究開始時点で利用可能であった、MN-Core (GPFN2) を使ったシステムでは、プログラミングシステムとして PyTorch が利用可能であった。そのため、Convolutional Neural Network (CNN) 等の機械学習計算は、既存の PyTorch による実装が動作していた。

本調査研究では、High Performance Computing (HPC) のアプリケーションを MN-Core で実行することを目的としており、典型的な HPC アプリケーションを PyTorch により記述することは、多くのハードルがあることは検討の初期段階で判明した。そのため、GPU のプログラミングシステムに近い、カーネルコンパイラについて検討をおこなった。以下、実装したコンパイラの概要、それによるアプリケーションの評価、そしてカーネルでは記述できないデータ転送の扱いについて記述する。

3.1.2　カーネル用コンパイラの実装

幅広いアプリケーションの性能評価をおこなうために、データ並列言語として、C 言語をベースとしたカーネル記述用コンパイラのプロトタイプを実装した。この実装には、LLVM プロジェクトによるコンパイラフロントエンド clang を利用した。clang により C 言語で記述された演算カーネルのソースコードを LLVM IR (中間言語) に変換し、その IR を MN-Core 用命令に変換する。これを使い、N 体計算カーネルや、ステンシル計算のカーネルから MN-Core 用命令を生成し、シミュレータとハードウエア上で性能評価と計算結果の検証をおこなった。

三次元ステンシル計算の例として姫野ベンチを元に修正を加えて実装した演算カーネルを示す。

```
1  __kernel void mncore_kernel(__local double *GRF,
2                              __local double *LM0,
3                              __local double *LM1)
```

```
4  {
5    const double omega = 1.0e-3;
6
7    double p[NBB][NBB][NBB];
8    double A1[NB][NB][NB];
9    double wrk1[NB][NB][NB];
10   double wrk2[NB][NB][NB];
11   double bnd[NB][NB][NB];
12
13   const int nb_size1 = NBB*NBB*NBB;
14   const int nb_size2 = NB*NB*NB;
15
16   load_lm((double *)p,    LM0, nb_size1);
17
18   load_lm((double *)A1,   LM1+0*nb_size2, nb_size2);
19   load_lm((double *)wrk1, LM1+1*nb_size2, nb_size2);
20   load_lm((double *)bnd,  LM1+2*nb_size2, nb_size2);
21
22   double gosa = 0.0;
23   for(int k = 1; k <= NB; k++) {
24     for(int j = 1; j <= NB; j++) {
25       for(int i = 1; i <= NB; i++) {
26         double s0, ss;
27         s0 = A1[i-1][j-1][k-1] * p[i+1][j ][k ] + A2[i-1][j-1][k-1] *
                p[i ][j+1][k ] + A3[i-1][j-1][k-1] * p[i ][j ][k+1]
28           + B1[i-1][j-1][k-1] * ( p[i+1][j+1][k ] - p[i+1][j-1][k ]
29                                  - p[i-1][j+1][k ] + p[i-1][j-1][k
                                       ] )
30           + B2[i-1][j-1][k-1] * ( p[i ][j+1][k+1] - p[i ][j-1][k+1]
31                                  - p[i ][j+1][k-1] + p[i ][j-1][k-1]
                                       )
32           + B3[i-1][j-1][k-1] * ( p[i+1][j ][k+1] - p[i-1][j ][k+1]
33                                  - p[i+1][j ][k-1] + p[i-1][j ][k-1]
                                       )
34           + C1[i-1][j-1][k-1] * p[i-1][j ][k ] + C2[i-1][j-1][k-1] *
                p[i ][j-1][k ]
35           + C3[i-1][j-1][k-1] * p[i ][j ][k-1] + wrk1[i-1][j-1][k-1];
36         ss = (s0 * A4[i-1][j-1][k-1] - p[i][j][k]) * bnd[i-1][j-1][k
              -1];
37         gosa = gosa + ss * ss;
38         wrk2[i-1][j-1][k-1] = p[i][j][k] + omega * ss;
39       }
40     }
41   }
```

```
42
43    store_lm(LM1+3*nb_size2, (double *)wrk2, nb_size2);
44
45    LM1[nb_size2*4+1] = gosa;
46  }
```

この演算カーネルは、PE のローカルメモリにあるデータの演算のみを記述するものである。つまり、PE での演算のみを指定するもので、PE 間のデータ移動は記述しない。カーネル関数の引数である GRF は PE のレジスタメモリ、LM0, LM1 は PE のローカルメモリを示す。カーネル内の関数 load_lm(),store_lm() はローカルメモリとカーネル変数間とのコピー操作である。

配列のサイズ NB を変えることで、PE 当たりのメモリ量と演算量が変わる。MN-Core の命令を生成した例として、次のテーブルに NB を変えたときの LLVM IR の命令数と、MN-Core アセンブリの命令数を示す。

NB	LLVM IR 行数	アセンブリ命令数
2	577	302
3	1797	978
4	4145	2278
5	7993	4414

PE あたりに 5^3 のメモリを割り当てた場合までは、命令生成が可能であった。ただし、この命令生成では、PE 当たりのベクトル長は 1 を仮定している。これらの結果に基づき、PE あたりのベクトル長を変更した場合の性能を評価した。このステンシル計算では、毎回隣接 PE 間でのデータ交換 (袖交換) が必要であり、MN-Core (GPFN2) のアーキテクチャでは、(1) MAU 内での袖交換, (2) L1 ブロック内での袖交換, (3) L2 ブロック内での袖交換, (4) チップ内 L2 ブロック間の袖交換, (5) チップ間の袖交換という、5 階層でのデータ移動が必要となる。

コンパイラの実装では、アプリケーションで必要な倍精度演算に対応した三角関数の MN-Core 用実装を行い、数学関数ライブラリの初期的な評価も実施した。

3.1.3 アプリケーションの性能評価

以下では、PE 間のデータ移動が必要ない単純な並列計算について、性能評価を示す。

MN-Core のハードウェアで、アプリケーションの性能を評価するために、OpenCL API をベースとしてホスト API を定めてライブラリとして実装した。この API には、MN-Core ボードの初期化、メモリの割り当ておよび転送、演算カーネル実行のための関数を実装した。

API ライブラリを利用して、MN-Core 用の N 体計算カーネルの性能評価をハードウエア上でおこなった。DRAM からデータを PE に分配・回収する命令を含めて粒子数 (N) に応じて演算カーネルを生成するプログラムを作成し、いくつかの N について演算性能の測定と演算誤差の評価を

おこなった。演算を倍精度で行った場合、N が 64K,128K, 256K の場合の 1 ステップあたりの演算性能は、それぞれ 1029, 1196, 1545 GFLOPS であった。ホスト側で計算した結果と比較して演算誤差の平均は $\sim 10^{-15}$ となった。

カーネル用コンパイラを利用して、積分変数変換に三角関数を必要とする、ファインマンループ積分の Quasi-Monte Carlo (QMC) 積分法による性能評価を行った。性能は 1 秒あたりの積分点の処理数 (単位は秒あたり 10^9 points) で性能評価した。性能評価の結果を次のテーブルに示す。

積分点数 ($\times 10^9$)	性能 (GPoints/sec)
1.22	0.8563657535
2.147	0.9931252071
8.599	1.189134253
12.14	1.220734277
24.29	1.246782282

N 体問題と同様に、データ移動のないアプリケーションでは、問題サイズが十分大きいければ、PE の稼働率が高くなりより高性能となる。結果、MN-Core (GPFN2) での性能は最大で約 1.2 GPoints/sec であった。この性能を GPU (NVIDIA H100) での同じ計算と比較すると、おおよそ 4 分の 1 の性能に相当する。なお、MN-Core (GPFN2) と H100 では、チップ実装の半導体プロセス世代は、それぞれ 12 nm と 4 nm であり大きな隔たりがある。一方で、MN-Core 用命令のプロファイリングにより、この QCM カーネルにおける積和命令の割合は約 25 % であった。また、命令スケジューリング上必要な nop 命令の割合も大きく、まだ性能向上の余地がある。この評価から、現状のカーネルコンパイラはレジスタおよびローカルメモリの割り当ては十分最適化されているが、LLVM IR からの MN-Core 用命令への変換時に、不必要な命令をより削減する最適化が必要であることがわかった。

3.1.4 PE 間データ移動の検討

実アプリケーションでは、複数の PE 間でのデータ転送や総和処理を記述できなければならない。階層的に接続されている PE の間でデータ転送等を実現するための手法について詳細を検討し、一部の実装をおこなった。

MN-Core アーキテクチャでは、MAU 内の 4 個 PE 間では直接データ移動をするパスがある。しかし、L1 ブロック内の他の PE とは直接データをやりとりするパスはない。そのため、L1 ブロックの共有メモリを介することで、PE 間のデータ移動を実現できる。この方法では、転送先 PE の ID とデータを共有メモリからブロードキャストし、PE の宛先と個々の PE の ID が一致するときのみマスク命令によりレジスタ・ローカルメモリで書き込むことで、任意のデータ移動を実現できる。

異なる L1 ブロックに属する PE 宛てにデータを移動するには、L2 ブロックの共有メモリを介して、同様の方法でデータ移動を実現できる。ひとつの L1 ブロック内でのデータ移動命令については、2 次元ステンシル計算について評価をおこなった。

PE の ID としては、アーキテクチャでハードコートされた PE ID を使うこともできるし、多次元のデータを PE のメモリに割り付けて、最適なデータ移動が可能となるように別途 ID を割り振ることもできる。後者の場合、ID を重複して割り当てることで、一度の転送で複数のデータ書き込みも可能となり、様々な応用が考えられる。

3.2 OpenACC 検討

綱島 隆太 [国立大学法人 神戸大学]

神戸大において、本調査研究チームが提案したスーパーコンピュータシステムに搭載される MN-Core 後継アーキテクチャのアクセラレータへ、ユーザーが容易に移行できるようにするために、コンパイラ指示文（ディレクティブ）形式の API である OpenACC プログラミング環境の検討および開発を行った。なお、本検討の内容は、SWoPP2024 内で行われた情報処理学会第 195 回ハイパフォーマンスコンピューティング研究発表会 (綱島 ほか 2024) や The 7th R-CCS International Symposium などで発表している。

3.2.1 コンパイラ指示文形式 API の動向調査

アクセラレータの汎用利用における従来のコード記述方法は CUDA や OpenCL といった、低レベルの詳細な記述を要するプログラミング言語によるものであり、アクセラレータへの移行や参入の障壁となっていた。そこで、アクセラレータに対応するコンパイラ指示文形式の API が出てきた。コンパイラ指示文（以下、ディレクティブ）形式 API はコンパイラへのヒントとなる指示文をプログラムのソースコードに専用書式のコメントとして記述するものである。コンパイラはその指示文の情報を元にコードの最適化や変換を行う。CUDA や OpenCL と比べてディレクティブ形式 API のメリットには、以下のような点がある。

- CUDA や OpenCL が全面的な書き換えを要する一方、既存の CPU コードのアクセラレータ対応が追記のみで可能
- API 初期化やキューの管理でホストコードを逐一書く必要が無いため、書くべきことを減らせる
- アクセラレータ上のデバイス変数、デバイスポインタを逐一宣言する必要がない
- ディレクティブはコンパイラの設定によって無視することができ、デバイス間におけるコードの可搬性が高められる

これらによって、記述量を減らすことができ、既存コードの移植や新規参入にかかるプログラミングの障壁を抑えられる。特に、既存の CPU コードを修正する必要がなく、必要な指示文を追記す

るだけで良いため、CUDA や OpenCL に比べてプログラムの生産性を大きく向上させることができる。

さらに、ディレクティブ形式の良いところは、本来やらせたいアルゴリズムをそのまま書くのに近いシーケンシャルなコードの記述と、並列化のための記述を分離できることである。最近では、CUDA や OpenCL の代わりとして SYCL や Kokkos といった、CPU で使われている言語でネイティブに記述する言語やライブラリ形式のものも出てきているが、はじめから並列化した際の複雑な動作を考えながらコードを記述する必要がある。一方で、ディレクティブ形式であれば、まず初めに CPU 用のコードを書いて基本的なアルゴリズムを検証してから、ディレクティブを追加してアクセラレータへ移植する、といったことが容易にできる。

3.2.1.1 OpenACC

OpenACC は 2011 年に登場した、C、C++、Fortran に対応するアクセラレータ向けの標準ディレクティブ API である。今日、OpenACC は GPU プログラミングにおけるディレクティブ形式 API のデファクトスタンダードとなっている (三木・塙 2024)。2008 年の東工大の TSUBAME 1.2 への採用以降、NVIDIA は HPC における GPU 活用を牽引しており、本調査研究開始時点で NVIDIA が HPC における最も主要な GPU ベンダとなっていた。そのような状況の中、NVIDIA は OpenACC のコンパイラ開発に力を入れており、OpenACC で書かれているアプリケーションの実例も多数ある状況であった。

3.2.1.2 OpenMP との比較

OpenMP も 2013 年の仕様からアクセラレータに対応している。Intel と AMD のデバイスでは OpenACC がベンダ製コンパイラで対応していない一方で、OpenMP に対応しているため、単純に対応しているベンダの数から OpenMP の方がユーザーにとって都合が良いとする意見も見られる。しかし、既存の GPU ユーザーにとってより手慣れた手段で容易に MN-Core 後継アーキテクチャに移行できるようにすることは極めて重要である。OpenACC の方が先に登場したため、本調査研究の初期段階で、先述のとおり OpenACC のほうが普及している状況であった。OpenMP に優先対応することで既存の多くの OpenACC ユーザーには学習コストが発生する。また、記述性、機能面について、並列数の明示的な指定や非同期実行のキューの指定など OpenACC に比べて対応が不十分な点があり、結果として既存アクセラレータにおける性能の高速化においても OpenMP より OpenACC の方が優勢な状況であった。

OpenMP の仕様は先に登場した OpenACC の後追い状態となっていることから、より機能的な記述が可能な OpenACC に対応しておけば、OpenMP への変換は比較的容易であると言える。実際に、米国オークリッジ国立研究所をはじめとして両者の間で変換を行うトランスレータの開発が進められている。

さらに、OpenACC は基本的にアクセラレータにターゲットが限定されており、仕様が OpenMP より小さいため処理系実装しやすいというメリットがある。

したがって、本調査研究では OpenACC の対応を検討することにした。

3.2.2 MN-Core における OpenACC の対応検討

本調査研究の提案アクセラレータは MN-Core の後継に当たるものであるが、現行の MN-Core とのアーキテクチャの最大の違いは DRAM がオンチップかオフチップかということである。現行の MN-Core では DRAM がオフチップであるが、後継アーキテクチャでは Processing Element（PE）ごとのローカルメモリとしてオンチップに存在している。しかしながら、現行の MN-Core と後継アーキテクチャにはともに PE ごとに SRAM のローカルメモリが存在しており、DRAM を除いた基本的なアーキテクチャは同じである。言い換えれば、PE のローカルメモリのサイズが後継では DRAM を含めた大きなサイズになるが、現行では SRAM のサイズに限定される。したがって、本検討では PE のローカルメモリに収まる範囲のデータサイズに限定して、現行の MN-Core（MN-Core 2 ではなく、本調査研究開始時点で存在していた MN-Core [通称「無印」]）に対応する OpenACC の検討を行った。

OpenACC の記述と MN-Core のアーキテクチャの整合性であるが、問題は少ないことがわかった。MN-Core もアクセラレータであり、ホストからアクセラレータを操作するという基本的なプログラミングモデルが適用できるためである。ただし、メモリモデルに関して、GPU などの既存のアクセラレータが共有メモリ型であるのに対し、MN-Core が各 PE にローカルメモリを持つ分散メモリ型であり、前提としているメモリモデルが異なっていることから、特にこの点についての対応検討が必要であることもわかった。

3.2.2.1 OpenACC と MN-Core の親和性

また、OpenACC のプログラミングモデルと MN-Core のアーキテクチャの親和性については、非常に高いことがわかった。

OpenACC のようなディレクティブ形式の API では、ユーザーによる記述が少なくなる分、並列化のための分割やベクトル化などの処理をユーザーが細かく決めるのではなく、言語処理系によって決定するようになっている。GPU に限らず CPU でもそうであるが、既存のアーキテクチャの場合、並列化したコードの性能向上のためにはスレッド数の調整やキャッシュ使用率向上などの為の性能最適化が必要である。しかし、これらは事前に最適な値が正確に分からず、ユーザーは既存のアーキテクチャから実行時の動きを予想して最適と思われる値になるようにプログラミングするか、実際に繰り返し実行してみて性能評価をしたうえで最適化を試みている。ユーザーもこのように高度なハードウェアの知識と、経験則を頼りに最適化を試みているなか、コンパイラで同様のことを実現するのは非常にハードルが高く、実際、GPU において OpenACC は CUDA での手動最適化に対して性能が劣ると言われてきた。

多くの場合、この原因は記述性の問題であり、高度な抽象化によって細かい記述ができないからであるという説明がなされるが、根本的な問題は、ソフトウェアから明示的にコントロールできないハードウェア上の動作が性能に直結するということであり、これがコンパイラでの性能最適化のハードルが高い原因でもある。しかし、MN-Core ではキャッシュやスレッドなどが存在せず、データ移動や並列度といった性能に関わる部分がプログラマブルである。よって、MN-Core

ではこれらが実行前に明示的にコントロール可能であるため、コンパイラにおける事前に制御できる範囲のプログラム性能最適化を理論上正確に行うことができる。実際に、遠藤 (2022) によって MN-Core 上のレジスタ割り当てを理論限界性能まで最適化できることが確認されており、MN-Core 上の動作の部分的な最適化はすでにコンパイラで正確にできることがわかっていた。

したがって、MN-Core では API の設計次第でユーザーが既存のアーキテクチャより性能を出しやすいプログラミング環境を実現できることが見込まれたため、それに向けて調査研究に取り組んだ。

3.2.3 API 仕様と言語処理系設計の検討

3.2.3.1 概要

MN-Core 向け OpenACC（以下、OpenACC/MN-Core または OpenACC/MN）の仕様について検討した。また、仕様を検討するにあたって、言語処理系の設計は密接に関わることであり、実装コストを考慮することも現実的に重要であるため、言語処理系設計の検討も行った。HPC の典型的なアプリケーションにはステンシル計算が含まれていることが多いため、プロトタイプの目標としてステンシル計算コードを記述できることとした。

OpenACC/MN プロトタイプの仕様は以下のようなものである。

- 対応言語は C、Fortran
- 基本的に OpenACC 1.0 までの仕様
- 一部 2.0 以上の仕様（特に分割コンパイルに必要な仕様）
- 加えて独自拡張

対応言語は標準の OpenACC と異なり、C++に未対応としているが、これは後述する実装ツールが C++に未対応であることによる。C++は C と比べて仕様の規模が膨大であり、独自に対応しようとした場合、非常に実装コストが高い。C++から C 言語のコードを呼べることから、現実的にひとまずこのような仕様とした。ただし、将来的に C++に対応する場合には、C 言語版と同様の仕様となる想定である。

対応する API は OpenACC 1.0 までの標準仕様を基本とした。また、`enter/exit data`、`routine` など、OpenACC 2.0 以上の仕様は主に分割コンパイラ対応のためのものに優先的に対応する。加えて、MN-Core のアーキテクチャのための独自拡張にも対応する。他にも OpenACC の独自拡張を行った例には PEZY-SC シリーズ向けの OpenACC(Tabuchi *et al.* 2016; 田渕 2018) や複数種類のアクセラレータへ同時にオフロードすることを目的とした CAMP（Cooperative Acceleration by Multi-device Programming）システムの OpenACC (綱島 ほか 2023; 綱島 2024) がある。なお、OpenACC 標準にはディレクティブ形式 API の他にランタイム関数も含まれているが、ディレクティブで同様の記述が可能であることから、本検討において OpenACC/MN には含めないこととした。

3.2.3.2　言語処理系設計

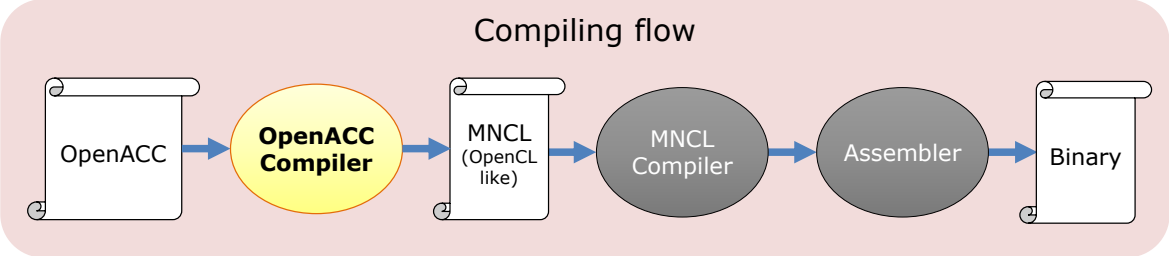

図 3.2.1　言語処理系の設計概要

言語処理系の設計概要を図 3.2.1 に示す。OpenACC/MN コンパイラは、OpenACC/MN から OpenCL 相当の MN-Core 専用汎用言語 MNCL へ変換する Source-to-Source コンパイラとした。MNCL からバイナリファイルへの変換は MNCL コンパイラにより行うようにする。このようにすることで MNCL と処理系を一本化することができるため、実装コストを削減できるばかりでなく、開発作業を分担することができる。このような OpenACC の Source-to-Source コンパイラには複数の実装例があり、理研の Omni OpenACC コンパイラや米国オークリッジ国立研究所の OpenARC などが挙げられる。また、Source-to-Source コンパイラとして実装することにより OpenACC から MNCL の関数呼び出しを容易に実装可能である。OpenACC から低レイヤー言語を呼び出すことを想定した構文は OpenACC 標準仕様に存在しており、実際に GPU でも OpenACC から CUDA 呼び出しができる。これにより、OpenACC では表現できない細かい最適化を行うことが可能となる。

3.2.3.3　仕様設計方針

本検討における設計方針として、典型的にアクセラレータで実行される SIMD 的に全体または分岐した条件の範囲が一斉に同じ動作をする処理を OpenACC/MN のターゲットとし、複雑なコードは MNCL で書かれる想定とした。ディレクティブ形式の API ではすべての並列実行単位が同じ動作をする SIMD 実行を表現するのは比較的容易であるが、例えば、分散メモリ上の明示的な通信制御をするうえで、個々の PE 間で異なるパターンで通信させたいといった場合には、コメント調のディレクティブ形式 API では複雑さや冗長さが出てきてしまう。このような場合にはディレクティブではなく、専用言語で記述できたほうが簡潔なコードになることが予想できる。なお、OpenACC から MNCL を呼び出すことが可能な設計のため、OpenACC で表現できない箇所が一部のみである場合には、すべて MNCL で記述する場合に対し、プログラミングの負担を最小限に抑えることができる。

3.2.4　MN-Core 向け OpenACC の基本設計（GPU 向け OpenACC との違い）

本検討では、分散メモリ型のアクセラレータにおけるディレクティブ形式のプログラミング手法を新たに検討し、提案することとした。この基本設計の違いにより記述時の前提も変わってくる。

MN-Core 向けの独自拡張は主に分散メモリに対応するためのものである。OpenACC/MN のデバイス固有の独自拡張の内容は以下のとおりである。

- MN-Core 上の各階層でのデータ分割の指示
- ホストと PE 間の集団通信の指示
- PE 間の集団通信の指示

これらは OpenACC が先述のとおり GPU をはじめとする共有メモリ型のアクセラレータを前提としているのに対し、MN-Core が分散メモリ型アーキテクチャであることから、よりハードウェアに近い制御を可能にして性能チューニングの際のユーザービリティを高めるために定義される。

これらの検討経緯であるが、標準の OpenACC をそのまま分散メモリ型アーキテクチャの MN-Core に当てはめようとすると、ユーザーが直感的に操作できるようなインターフェイスではなくなり、HPC におけるアプリケーションの性能最適化に適さなくなることが危惧された。具体的には、GPU では共有メモリを前提とした並列実行であるため、ユーザーは個々のデータ配置について意識する必要はなく、並列粒度はループに `gang`、`worker`、`vector` を指定してループ分割をするようになっている。しかし、このプログラミングスタイルをそのまま MN-Core に適用しようとすると、分散メモリであるため、データの配置が直接計算結果に関わるばかりか、プログラミング時にデータの置き場所とマッチしないループ分割を記述できてしまい、そもそも計算自体の可否に関わるという問題が起こる。

このプログラミングスタイルを厳格に守り、変換時に必要な通信を自動生成する、通信コストを無視して DRAM を共有メモリ代わりに使うなど、記述とのギャップをコンパイラやランタイムライブラリなどの言語処理系で解決することはできなくはないが、その場合は処理系の実装コストが上がるばかりでなく、直感的ではない、性能チューニングの難しいプログラミングをユーザーに強いることになる。この場合、明示的な通信制御を行うインターフェイスではなくなり、MN-Core のアーキテクチャによる予測可能な動作に基づいた性能最適化をユーザーが直接行えなくなってしまう。また、分散メモリではデータの分割方法が通信コストへ顕著に影響をするため、その制御は重要である。

そこで本検討では、MN-Core 上の分散メモリのための OpenACC の独自拡張を検討した。

3.2.4.1 MPI 相当のノード間並列実行用ディレクティブ形式 API

分散メモリ型のハードウェアを想定したディレクティブ形式 API には、アクセラレータを対象としたものではないものの、通常 MPI で記述されるノード間での分散メモリ並列計算のためのプログラムをディレクティブ形式で記述できるようにした High Performance Fortran（HPF）(High Performance Fortran Forum 1997) および XcalableMP（XMP）(Nakao *et al.* 2018) が存在する。これらはディレクティブから分散メモリ上で明示的に通信を制御する実行コードを生成する。分散メモリ型の API の利点として、共有メモリ型では制御が困難なメモリ通信について明示的に記述できるため、API レベルでユーザーによる最適化が容易となることが挙げられる。これらは HPC 向けであり、特に XMP はリファレンス実装が存在し、実アプリケーションで使用可能な実装が実現

できていることから、これらのノード間並列計算用ディレクティブ形式 API を参考にした。

勿論、OpenACC/MN はアクセラレータのための API であるため、MPI の代わりとしてノード間並列処理を行うための HPF や XMP とは役割が異なる。また、アクセラレータのためのプログラミングではホストとデバイス間のデータ転送や、オフロードしてデバイス上で実行される箇所の指定が必要である。加えて、MN-Core はツリー状の階層構造となっているため、この部分との整合性を取る必要があり、OpenACC はもちろんのこと、HPF、XMP などの既存の API と全く同じものをそのまま持ってくることは不可能である。そのため、あくまで OpenACC が基礎であり、そこに HPF や XMP を参考にした要素を取り入れる設計とした。あくまでも OpenACC であることから、OpenACC 標準に倣って記述対象もノード間並列処理ではなく、1 ノード内のアクセラレータにおける並列処理である。

HPF や XMP は PGAS（Partitioned Global Address Space）モデルの API である。PGAS モデルでは、ハードウェアレベルにおいて分散メモリ環境であっても、仮想的に大域的な名前空間をユーザーに提供する。分割方法はディレクティブで制御できるが、配列自体は共有メモリの場合と同様に記述でき、ユーザーはメモリ階層のスコープを意識しなくて良い。また、データの場所は言語処理系が管理するため、分散したメモリ間の通信記述を簡略化することができる。OpenACC/MN でも元の OpenACC の共有メモリモデルや HPF、XMP に倣って PGAS モデルを採用することにした。

このプログラミングモデルの設計は、元の OpenACC に近いとは言え、分散メモリ管理のための記述が増えるため、一見して元の OpenACC と比較してプログラミング難易度が上がるかのように見える。しかし、性能チューニングのコストは下がることが予想される。GPU では特に演算律速の場合にシェアードメモリを用いることが性能向上のために必須であったが、OpenACC では CUDA や OpenCL と異なりシェアードメモリを確実に利用できる明示的な制御方法はなく、最適化のコンパイラ依存性が非常に高いため、この部分のユーザーによるチューニングは困難である。これはループに対して分割を指定するようになっていることにより、キャッシュブロッキングのようなスコープを限定した多次元分割の表現が困難であるという根本的な問題も含んでいる。一方で、OpenACC/MN ではシェアードメモリを考える必要がなく、分割方法についてもデータの分割を明示的に指示するため、分散メモリアーキテクチャである MN-Core 上でユーザーが記述したとおりに素直にプログラムが実行でき、ローカルメモリの高いバンド幅を有効に活用できる。さらに、前述のとおり、動作が決定論的で性能予測が可能なアーキテクチャであることから、コンパイラの最適化の精度も限界まで向上させることが理論上可能である。

3.2.5 具体的な OpenACC/MN のインターフェイス

OpenACC/MN のインターフェイスはなるべく既存の OpenACC と記述上の差異が少なくなるようにした。これはユーザーの GPU からの移行のしやすさを考慮したことに加えて、実装コストを抑えるためである。プロトタイプコンパイラの実装には既存の標準 OpenACC パーサーを拡張するため、独自拡張が増えるほど実装コストが上がってしまうことから、標準仕様でデバイス・実装依存となっている箇所は独自の解釈を定義し、解釈を拡張することで対応できる部分は API の拡張

はせずに解釈の拡張のみとした。

OpenACC 標準のインターフェイスを用いる主な機能には以下が挙げられる。

- ホスト・デバイス間データ転送
- デバイスメモリの確保と解放
- すでにデバイスに存在するデータの明示
- デバイスへオフロードする範囲の指定
- ループの並列実行階層（並列粒度）の指定
- リダクションの指示

独自に拡張したインターフェイスを用いる機能には以下が挙げられる。

- MN-Core 上の各階層でのデータ分割の指定
- 袖領域の指定
- 袖交換のタイミングの指示
- その他リダクション以外の集団通信（分配、放送、収集、データ同期、並び替えなど）

まず、OpenACC 標準のインターフェイスのうち、実装依存の定義について説明する。

3.2.5.1 デバイスへオフロードする範囲の指定

OpenACC でオフロード範囲を指定するディレクティブには `parallel`、`kernels`、`serial` の 3 つがある。このうち、`serial` は `parallel` の並列粒度をすべての階層で 1 を指定した場合（つまり、GPU では 1 スレッドで実行）のエイリアスとなっている。また、これらと `loop` ディレクティブを組み合わせた `parallel loop` 等の書き方がある。

`parallel` と `kernels` の違いは以下のとおりである。`parallel` ディレクティブの場合はある `parallel` ディレクティブの適用範囲がすべて同じ並列粒度で実行され、その適用範囲を実行したときの結果はユーザーによって保証されている前提でコンパイルされるのに対し、`kernels` の場合はその適用範囲下に書かれている `loop` ディレクティブごとに別々のカーネル関数と見做して並列粒度を変えることができ、個々の `loop` ディレクティブの並列化の判断はオプション節の指示がない限りコンパイラの判断となる。

`kernels` の動作は MN-Core の SIMD アーキテクチャに合致しない。`kernels` はスレッドを前提とした設計となっているが、MN-Core ではスレッドが無く、8192 個の PE が常に一斉に動くため、ハードウェアの動作として、カーネルごとに異なる PE 数で動かすようなことはできない。したがって、OpenACC/MN のプロトタイプコンパイラではひとまず `parallel` に対応することとした。

ただし、GPU では `parallel` と `kernels` は実使用上ほぼ違いがなく、あまり区別なく使われる傾向にある。よって、`parallel` ディレクティブで記述することを推奨するが、将来的には GPU からの移植を考慮して、`kernels` を用いている場合にもコンパイラで内部的に `parallel` と解釈することで変換自体は受け付けるようにする想定とした。`serial` についても同様である。

3.2.5.2 並列実行階層の指定

ループの並列展開で指定できる gang、worker、vector の各階層については、OpenACC の仕様上、その解釈はデバイスごとに実装依存となっているため、MN-Core でも拡張はせずに独自の解釈を行う。OpenACC/MN では、以下のように解釈する。

- gang：PE レベル並列
- worker：割当てなし
- vector：ベクトル命令並列

gang の並列数は PE 数の 8192 で固定とし、num_gangs で gang の数を指示した場合は無視される。なお、loop ディレクティブが記述されている場合、デフォルトで gang レベルが指定されているものと解釈し、PE の数だけ並列展開する。

worker の割当てが無いことは、適当なハードウェア階層がないことに加えて、最も普及している NVIDIA GPU では、慣習的に worker があまり使用されないことを踏まえている (成瀬 2017)。

vector については NVIDIA GPU ではスレッドと解釈されているが、OpenACC/MN では本来の仕様の意味に合致するベクトル命令とした。ベクトル長はデフォルト（指定なし）では最大値に指定されているとコンパイラに解釈されるが、vector_length 節で最大の値まで任意に指定することも可能とする。例えば、MN-Core のベクトル命令では 1 演算命令で倍精度 1 要素分の演算を 4 サイクル実行できるため、倍精度演算で指定できる最大ベクトル長は 4 である。

なお、ループに対する階層の指定とループで参照されるデータの配置が一致しない場合は、コンパイルエラーとなる。

3.2.5.3 ネストループの並列展開指示

ネストループについては collapse 節を記述することで、複数ループをまとめて並列展開する。

MN-Core ではスレッドが存在せず、並列化は PE レベルでの展開が基本で、それ以外はベクトル命令による実行ができるかどうかのみであるため、基本的に gang レベルで一とおりまとめて並列展開させる必要がある。さらに、OpenACC 2.0 で gang ループの中に gang 節を含んだループを含めることを禁止している。よって、collapse 節を使わずに、ループそれぞれに loop ディレクティブを記述して並列階層（gang、worker、vector）を指示、もしくは省略した場合、GPU 版 OpenACC のように collapse 節を指定したときと同様の変換が行われることはない。

次に、MN-Core 向け独自拡張について説明する。

3.2.5.4 MN-Core 上の各階層でのデータ分割の指定

MN-Core のチップ内部はツリー構造となっており、各層に通信バッファのためのメモリが存在している。これらのメモリ（具体的には L2BM、L1BM）はあくまで通信時に使用されることを想定しており、レイテンシや電力などの面からデータを留めておくキャッシュとしての使用は想定されていない。さらに、前述したとおり、本検討ではローカルメモリに収まる範囲のデータ量に限っ

て扱うことから、計算中はすべてのデータが PE に配置される。データ転送を考えると、ホストからデバイスへデータを送る際には通信バッファメモリを介して一気に PE までデータを送ることになる。逆にデバイスからホストにデータを送る際も同様に PE からホストへ一気にデータが送られる。よって、この際のデータ分割を指示する記法について検討した。

なお、後述するとおり、デフォルトでは分割指定の記述を省略することができる。

分割の指定はホスト・デバイス間の通信、もしくはデバイスメモリの確保と解放と一緒に節として記述する。具体的には `copyin` 節、もしくは `copyout` 節の指定できるディレクティブでサポートする。プロトタイプコンパイラではひとまず `enter data`、`exit data` ディレクティブでの記述に対応する。

リスト 3.2.1　分割数指定のための拡張節（C 言語の場合）

```
1 #pragma acc enter data copyin(a[0:N][0:N],b[0:N][0:N],c[0:N]) \
2     L2(a,b[4][4];c[16]) L1(a,b[4][2];c[8]) \
3     mab(a,b[2][8];c[16]) pe(a,b[4][1];c[4])
```

リスト 3.2.2　分割数指定のための拡張節（Fortran の場合）

```
1 !$acc enter data copyin(a(1:N, 1:N),b(1:N, 1:N),c(1:N)) &
2     !$acc& L2(a,b(4, 4);c(16)) L1(a,b(4, 2);c(8)) &
3     !$acc& mab(a,b(2, 8);c(16)) pe(a,b(4, 1);c(4))
```

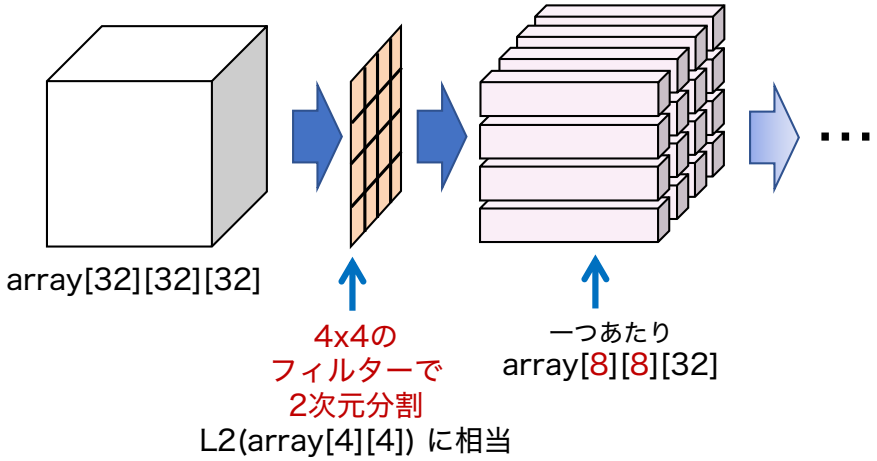

図 3.2.2　例：`L2(array[4][4])` という記述があった場合の分割イメージ

独自拡張の各節は C 言語ではリスト 3.2.1、Fortran ではリスト 3.2.2 のように記述し、階層ごとに分割数を定義する。`L2` 節は 1 パッケージあたりの L2BM の、`L1` 節は一つの L2B 内部の L1BM の、`mab` 節は一つの L1B 内部の MAB の、`pe` 節は一つの MAB 内部の PE の分割数をそれぞれ示している。例えば `L2` 節であれば、1 パッケージ内の全 L2BM を用いてどのように分割するかということを記述する。

L2 などに続く丸括弧の中には配列名とその配列の各次元の分割数を記述する。a、b、c はそれぞれ配列名である。それぞれの次元の分割数は C 言語の場合、角括弧 [ ] の中に 0 以外の自然数で指定できる。Fortran の場合、配列名に続く丸括弧（ ）の中に各次元ごとにカンマ , で区切って、0 以外の自然数で指定できる。ただし、分割数の値は各次元の分割数を掛けた値がその階層の個数の合計値を超えない範囲で指示する必要がある。

各配列の記述はセミコロン（ ; ）で区切る。ただし、同じ分割を行う配列は、配列名の直後にコロンで区切って続けて書く省略記述が可能である。

なお、ここで設定するのは分割後のサイズではなく分割数であるため、配列サイズを変化させる場合でも、各次元が元のサイズの倍数で変化する場合には書き換え不要である。

分割の考え方は図 3.2.2 のように、各階層で指定したとおりに分割された、格子状のフィルターを通していくイメージがわかりやすい。複数階層あるため、複数の格子状のフィルターを通して、最終的な格子 1 つあたりのサイズが 1 つの PE に配置されるデータサイズとなる。数式上は階層のフィルターを通すたびに分割数で割られていく。

記述を省略し、デフォルト分割する場合、正方形や立方体に最適化された以下の分割が行われる。

- 2 次元では L2B, L1B, PE を [4][4], [4][2], [(4) *2][8] で分割
- 3 次元では L2B, L1B, PE を [4][4][1], [1][1][8], [4][4][(4)] で分割
- 1 次元では MAB の中の PE0 と PE1 などで交互に（インターリーブで）配列を配置

2、3 次元のうち、(4) は同じ MAB に属することを示している。また、上記を合算すると 2 次元では [128][64]、3 次元では [16][16][32] の格子に分割され、それぞれの次元の格子数で割った値が 1PE あたりの担当要素数となる。逆に言えば、この値未満の 2、3 次元配列では 1 要素も分配されない PE が発生し、全 PE を使い切らない。

基本的に正方形や立方体の形であれば、上記のデータ分割方法を指定することで通信効率を上げられる。正方形や立方体から大きく崩れなければ、通信効率は高く保たれる。

また、1 次元配列をこのようにするのは、このようにしなかった場合、袖交換の際に MAB 単位の放送のバンド幅を有効活用するために、一旦境界の PE からその他 3 つの PE にデータを分散して、更に受け取った MAB 側でもそのデータをもう一回境界の PE に集約する必要があり、データ転送が 2 回余計に増えてしまうためである。

正方形や長方形でない場合やプログラマー自身で設定したい場合にはリスト 3.2.1 やリスト 3.2.2 のような記述をすることでデフォルトでないデータ分割を行うことができる。

3.2.5.5 袖領域、袖交換の指示

MN-Core は分散メモリ型であるが、特にステンシル計算のようなループ回ごとの依存性、すなわち PE をはじめとした並列実行単位間の計算の依存性が発生する処理の場合は、共有メモリと異なり明示的な通信コントロールが必要となる。したがって、袖交換のためのインターフェイスを検討した。

袖交換の指示記述については、既存の計算科学ユーザーが移行しやすくするため、前述のノード間並列計算用ディレクティブ形式 API と互換性のあるものとした。ステンシル計算のサンプルコードの C 言語版をリスト 3.2.3、Fortran 版をリスト 3.2.4 に示す。袖領域の指定を shadow 節で、袖通信のタイミングを reflect ディレクティブでコンパイラに指示する。なお、このサンプルコードではデフォルトが存在する OpenACC/MN 記述については省略しているため、独自拡張は袖に関するものだけである。

shadow 節は配列の袖領域を指定するための節である。shadow に続き丸括弧の中に配列名、それに続いてそれぞれの次元の袖の幅を角括弧（Fortran の場合は丸括弧）の中に続ける。なお、C 言語の場合は次元ごとに角括弧を記述するが、Fortran では同じ丸括弧の中に次元ごとにカンマで区切って記述する。袖の幅は 0 以外の自然数で記述する。なお、先頭の袖の幅と末尾の袖の幅が同じ場合、リスト 3.2.3、リスト 3.2.4 のように省略記述が可能である。そうでない場合、先頭の袖の幅と末尾の袖の幅をコロン : で区切って記述する。複数の配列を指定する場合はカンマ , で区切って、別の配列の記述を続ける。shadow 節の記述場所は copy、 copyin、 copyout、 create、 present の節が指定されるディレクティブとする。

リスト 3.2.3　ステンシル計算のサンプルコード（C 言語版）

```
#pragma acc parallel create(temp[0:128][0:128][0:128]) \
    copy(a[0:128][0:128][0:128]) shadow(a[1][1][1])
for(count=0; count<N; count++) {
#pragma acc loop gang vector 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++){
                 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*a[i][j][k];
    } } }
#pragma acc loop gang vector 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++){
        a[i][j][k] = temp[i][j][k];
    } } }
#pragma acc reflect(a)
}
```

リスト 3.2.4　ステンシル計算のサンプルコード（Fortran 版）

```
!$acc parallel create(temp(1:128, 1:128, 1:128)) &
    !$acc& copy(a(1:128, 1:128, 1:128)) shadow(a(1, 1, 1))
do count=1, N
```

```
 4 !$acc loop gang vector collapse(3)
 5     do i=2, 128-1
 6         do j=2, 128-1
 7             do k=2, 128-1
 8                 temp(i,j,k) = &
 9                     c1*(a(i-1,j,k)+a(i+1,j,k) &
10                     +a(i,j-1,k)+a(i,j+1,k) &
11                     +a(i,j,k-1)+a(i,j,k+1)) &
12                     +c2*a(i,j,k)
13     end do; end do; end do
14 !$acc loop gang vector collapse(3)
15     do i=2, 128-1
16         do j=2, 128-1
17             do k=2, 128-1
18                 a(i,j,k) = temp(i,j,k)
19     end do; end do; end do
20 !$acc reflect(a)
21 end do
22 !$acc end parallel
```

前述のノード間並列計算用ディレクティブ形式 API では shadow は独立したディレクティブであるが、OpenACC/MN では節としている。これは OpenACC ではどの指示に関しても実行時の動作に即して決まったタイミングで記述するようになっており、文法規則を統一するためである。逆に言えば、XMP ではどこでも書けてしまうようになっているが、これはコンパイラ実装の面でも、コード保守性の面でもメリットがないため、他の独自拡張でも避けて設計している。

reflect ディレクティブは袖通信のタイミングを指示する指示文である。通信の発生するタイミングで記述する。引数は配列名である。複数の配列を指定する場合はカンマ（, ）で区切る。サイズの指定は不要である。reflect ディレクティブで指示している配列が、その箇所で有効となっている shadow 節に含まれていない場合、エラーとなる。

なお、袖が巡回する場合や斜めの通信がいらない場合は XMP と同様のオプションを記述する必要がある。

3.2.5.6 その他の集団通信（袖交換、リダクション以外）

ここでは OpenACC に標準で存在しているリダクションと前述した袖交換以外の集団通信について述べる。

OpenACC/MN ではデフォルトとして、ホストからデバイス（PE）へのデータ転送は配列の場合には分割して分散配置、スカラー変数の場合には放送としている。

それ以外の記述についても検討を行った。C 言語はリスト 3.2.5、Fortran はリスト 3.2.6 に示す。なお、これらのコードはあくまで記述例を示すためのものであり、具体的な計算のコードサンプルではないことを補足する。

ホストから来る配列を放送したい場合は bcast 節を用いる。引数は配列名であり、カンマで区切って複数の配列を指定する記述が可能である。copy、copyin、create の各節に放送したい配列が記述されているディレクティブに一緒に記述する。なお、配列が create 節で指定されている場合、実際には放送されないが、すべての PE に同じ要素数の配列が確保されることをコンパイラに知らせるために bcast 節を記述する。

bcast_divide 節は loop ディレクティブの指示されたループにおいて計算自体は分割するが、すべての配列要素を各 PE に重複保持したい場合に指示する節である。引数は配列名であり、カンマで区切って複数の配列を指定する記述が可能である。

この節を用いることで、メモリが足りている限りは計算が進む毎に発生する PE 間のデータの依存性を解決できる。共有メモリ型では PE 間のデータを互いに参照することが可能であるが、MN-Core は分散メモリ型であるため、事前に必要なデータを同期しておく必要がある。この節で指定された配列は放送されて各 PE に重複して確保されるものの、各 PE で計算される index 範囲は決まっており、各 PE 上の分割対象のループではそれぞれの担当範囲のみが計算される。同期については各 PE 間のデータを Allgather の要領で同期する async_data ディレクティブを合わせて用いる。

async_data ディレクティブは bcast_divide 節と対になる存在であるため、送り元にスカラー変数を指定するなどの MPI の Allgather と完全に同じ使い方はできず、あくまで繰り返し SIMD 実行される中で、各 PE で別々の要素が計算される配列の同期を目的としたディレクティブである。

このような集団通信は大雑把であり、一見してコストが掛かると考えられ、MN-Core でも通信コストは発生する。しかし、MN-Core の構造上、一旦ツリーの上の階層にデータを集めてから各 PE に放送するという処理は比較的容易に実装でき、さらに SIMD 動作で常に同期通信されるため、集団通信のための命令によってツリー構造でない場合に比べて低コストで実行することができる。

また、対象のハードウェアが事前にわかっていることから、パターン化された通信はインターフェイスを簡略化することができ、同じく全体で通信をする Alltoall についても同様のインターフェイスでの記述が見込めることがわかった。OpenACC 標準のリダクションも同じ理由で MN-Core に流用できる。

なお、これらのインターフェイスはオプションを拡張することで、MN-Core 全体を通してではなく、各 L1BM、各 L2BM の PE 単位で行うといったことも想定している。

リスト 3.2.5　その他の MN-Core 拡張（C 言語）

```
1 #pragma acc enter data copyin(a[:N][:N],b[:N],c[:N]) \
2     bcast(b) bcast_divide(c)
3
4 #pragma acc parallel
5         for(count=...) {
6                 ...
7 #pragma acc async_data(c)
8         }
```

リスト 3.2.6　その他の MN-Core 拡張（Fortran）

```
1 !$acc enter data copyin(a(:N, :N),b(:N),c(:N) &
2     !$acc& bcast(b) bcast_divide(c)
3
4 !$acc parallel
5         do count=...
6                 ...
7 !$acc async_data(c)
8         end do
9 !$acc end parallel
```

3.2.6　OpenACC/MN コンパイラの開発

処理系のうち、MNCL コンパイラは別途実装が検討されていたため、本節では綱島が担当し、開発した OpenACC/MN コンパイラについて説明する。なお、MNCL 実装は C 言語ベースであることから、OpenACC プロトタイプ実装についても C 言語版のプロトタイプを開発した。OpenACC とデバイスコードは並行して検討を進め、本検討から MNCL への提案も行った。

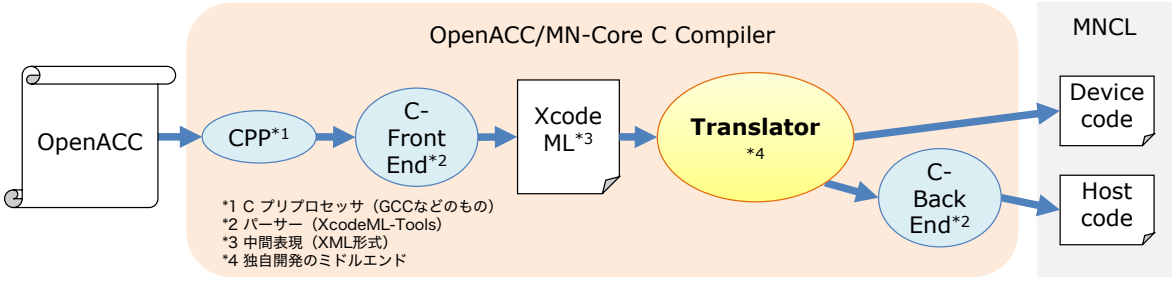

図 3.2.3　OpenACC/MN-Core コンパイラ実装

図 3.2.3 に OpenACC/MN C コンパイラの実装を示す。このコンパイラは前述のとおり Source-to-Source コンパイラであり、OpenACC/MN から MNCL のデバイスコードとホストコードへ変換する。C と Fortran のプロトタイプ実装をするための実現性や実装コストを考慮し、中間コード表現には XcodeML を使用する。XcodeML は C++には対応していないものの、C、Fortran との完全互換性があり、可逆変換に対応している。XcodeML・ソースコード間の変換ツールとして C、Fortran に対応した XcodeML-Tools が公開されている。また、XcodeML は前述の XMP のリファレンスコンパイラの実装ツールである Omni コンパイラ基盤で使用されている。

OpenACC/MN C コンパイラの構成と変換の流れについて説明する。OpenACC ソースコードから XcodeML への変換（図 3.2.3 中の C-FrontEnd）、XcodeML から MNCL ホストコードへの変換（図 3.2.3 中の C-BackEnd）には既存の XcodeML-Tools を修正して使用する。なお、C-FrontEnd の前にプリプロセッサ（図 3.2.3 中の CPP）を呼び出し、事前にそのソースファイルでインクルード指定されているコードを全て書き込み、マクロも全て置き換えたうえで C-FrontEnd により XcodeML に変換する。XcodeML-Tools の C-FrontEnd は OpenACC のパーサーを備えている。

本実装では OpenACC/MN 独自拡張の変換のために C-FrontEnd の拡張を行った。ホストコードは C 言語であり、既存の XcodeML-Tools により C に戻したほうが実装コストが低いことから、XcodeML-Tools の C-BackEnd を用いている。一方、デバイスコードは独自の言語であるため、独自実装の Translator から直接生成される。

なお、当初は利用者の多さや実質的なコンパイラ基盤のデファクトスタンダードとなっていることから、LLVM での実装を検討していた。しかし、C、Fortran ともに既存の OpenACC 実装が不完全な状態であることに加えて、OpenACC 実装で必要となる部分のドキュメントや参照実装の少なさ等から実装コストが高いことが判明した。また、Omni コンパイラ基盤についても同様の懸念があった。よって、プロトタイプとしては上記ツールで XcodeML に変換したうえで、Translator を独自に開発することとした。

Translator はミドルエンド、つまりコンパイラの中核を担っている。Translator の実装は Python の XML 処理ライブラリ lxml[1)] を使用した。XcodeML は XML の拡張であり、コードのフォーマットは XML であるため、XML の汎用処理が可能なツールを用いることができる。Translator ではディレクティブの中に現れる変数解析や分割のための処理など、コード全体を読み込み、必要な情報を検索し、様々な書き換えを行う。ホストコードとデバイスコードを適切に分離し、ホストコード用の XcodeML とデバイスコードのソースファイルを生成する。

3.2.7 コード変換と性能評価

OpenACC/MN プロトタイプコンパイラでは、3 次元までの配列を扱うコードの基本的なコンパイル機能を実装した。なお、以下で出てくる OpenACC/MN のコードは全て、プロトタイプコンパイラで対応している C 言語のコードである。

まず、簡単なテストとして、OpenACC/MN の 1 次元ベクトル加算コードをコンパイルして、MNCL への変換、および実機での実行ができるか確認した。リスト 3.2.7 に OpenACC/MN コードを示す。このコードは 8192 要素の配列の依存性のない、エレメントワイズなベクトル加算をするコードである。

このコードを実際に OpenACC/MN プロトタイプコンパイラでコンパイルした。変換後の MNCL ホストコードをリスト 3.2.8、MNCL デバイスコードをリスト 3.2.9 に示す。なお、MNCL は仕様が決まっていなかったため、ホストコードについては仮の関数へ変換している。ヘッダーをインクルードしたコードから始まるため、それらは前略している。また、`# 6 "acc_vecadd1.c"` などはコンパイラによって自動的に挿入される行番号である（このコードをコンパイルする際にはコメントとして扱われる）。

デバイスコードについては別途実装中の MNCL コンパイラの実装に合わせたコードへ変換しているため、そのまま MNCL コンパイラでコンパイルすることができる。PE 数と同じ 8192 要素の配列を計算するため、それぞれの PE で 1 つの index 番号の要素を計算するように変換している。変数宣言の修飾子の `__global` は DRAM または PDM、`__private` はレジスタを含めた PE のロー

1） https://lxml.de/tutorial.html

カルメモリ、bm2 は L2BM、bm1 は L1BM の確保を意味している。前述のとおり、__private 以外は__global メモリも含めて通信バッファとして使用される。distribute 関数は L2BM から L1BM へのデータ分配、distribute_pe 関数は L1BM から PE へのデータ分配、collect_PE 関数は PE から L1BM へのデータ収集、collect 関数は L1BM から L2BM へのデータ収集を行う関数である。ループについては PE あたりループ 1 回の計算であるため、for 文を取り払い、ループの中身のみを取り出すようにコンパイルしている。

実際にこのデバイスコードを MNCL コンパイラでコンパイルし、ホストコードは MNCL 実装が対応していないことからリスト 3.2.8 を元にしたホスト側からの制御コードを別途用意して、MN-Core の実機で実行を試みた。その結果、正しい計算が実行できることを確認した。

リスト 3.2.7　OpenACC/MN のベクトル加算コード

```
1  #include <stdio.h>
2  #define N 8192
3
4  int main() {
5      double array1[N], array2[N], array3[N];
6      for(int j=0; j<N; j++) {
7          array1[j] = 0.0;
8          array2[j] = 1.0;
9          array3[j] = 1.0;
10     }
11
12 #pragma acc enter data copyin(array1[:N],
13 array2[:N], array3[:N])
14
15 #pragma acc parallel loop gang present(array1[:N], array2[:N], array3
       [:N])
16     for(int i=0; i<N; i++) {
17         array1[i] = array2[i] + array3[i];
18     }
19
20 #pragma acc exit data copyout(array1[:N], array2[:N], array3[:N])
21
22 }
```

リスト 3.2.8　MNCL に変換後のベクトル加算のホストコード

```
1  // 前略
2  #include "MN_acc.h"
3  # 6 "acc_vecadd1.c"
4  # 6 "acc_vecadd1.c"
5  int main()
6  {
7  # 7 "acc_vecadd1.c"
```

```
 8 double array1[8192];
 9 # 7 "acc_vecadd1.c"
10 double array2[8192];
11 # 7 "acc_vecadd1.c"
12 double array3[8192];
13 {
14 # 9 "acc_vecadd1.c"
15 int j;
16 # 9 "acc_vecadd1.c"
17 for(j = (0); j < (8192); j++) {
18 {
19 # 10 "acc_vecadd1.c"
20 (array1[j]) = ((float)0.0);
21 # 11 "acc_vecadd1.c"
22 (array2[j]) = ((float)1.0);
23 # 12 "acc_vecadd1.c"
24 (array3[j]) = ((float)1.0);
25 }
26 }
27 }
28 # 15 "acc_vecadd1.c"
29 _ACC_MN_runtime_HtoD(array1, 8192);
30 _ACC_MN_runtime_HtoD(array2, 8192);
31 _ACC_MN_runtime_HtoD(array3, 8192);
32 # 18 "acc_vecadd1.c"
33 _ACC_MN_runtime_kernel_kick("main_kernel0");
34 # 23 "acc_vecadd1.c"
35 _ACC_MN_runtime_DtoH(array1, 8192);
36 _ACC_MN_runtime_DtoH(array2, 8192);
37 _ACC_MN_runtime_DtoH(array3, 8192);
38 }
```

リスト 3.2.9　MNCL に変換後のベクトル加算のデバイスコード

```
 1 __kernel void main_kernel0(
 2 __global double* _arg_array1,
 3 __global double* _arg_array2,
 4 __global double* _arg_array3
 5 ) {
 6     __private double array1[1];
 7     bm2 double array1_bm2[1 * 512];
 8     bm1 double array1_bm1[1 * 64];
 9     __private double array2[1];
10     bm2 double array2_bm2[1 * 512];
11     bm1 double array2_bm1[1 * 64];
```

```
12      __private double array3[1];
13      bm2 double array3_bm2[1 * 512];
14      bm1 double array3_bm1[1 * 64];
15
16      distribute(array1_bm1, array1_bm2, _arg_array1, 1);
17      distribute_pe(array1, array1_bm1, 1);
18      distribute(array2_bm1, array2_bm2, _arg_array2, 1);
19      distribute_pe(array2, array2_bm1, 1);
20      distribute(array3_bm1, array3_bm2, _arg_array3, 1);
21      distribute_pe(array3, array3_bm1, 1);
22
23      {
24          array1[0] = array2[0] + array3[0];
25      }
26
27      collect_pe(array1_bm1, array1, 1);
28      collect(_arg_array1, array1_bm2, array1_bm1, 1);
29      collect_pe(array2_bm1, array2, 1);
30      collect(_arg_array2, array2_bm2, array2_bm1, 1);
31      collect_pe(array3_bm1, array3, 1);
32      collect(_arg_array3, array3_bm2, array3_bm1, 1);
33  }
```

また、OpenACC/MN プロトタイプコンパイラでは袖交換のコード変換に対応した。リスト 3.2.10 に OpenACC/MN の 3 次元配列による 7 点ステンシル計算コードを示す。なお、このコードは科学計算アプリケーションの中に出てくる典型的なコードを模倣しているが、具体的な科学計算を解いているわけではないことを補足する。プロトタイプコンパイラではデータ分割記述省略時のデフォルト分割に対応しており、このコードでもデータ分割の記述を省略している。補足として、このコードのホスト・デバイス間通信は本来であればさらに最適化可能であるが、プロトタイプにつきこのような実装としている。

変換後の MNCL デバイスコードをリスト 3.2.11 に示す。なお、ホストコードはリスト 3.2.8 と同様の仮の関数への変換であるため省略する。`__private` で確保されているサイズは袖領域を含めた 1PE あたりのデータサイズである。ここでは [128][128][128] のサイズを [16][16][32] で分割しているため、袖を含まないサイズは [16][16][32] で割った [8][8][4] である。これは袖領域の存在しない `__private` 配列 temp のサイズに一致する。一方、配列 a は変換前のコードであるリスト 3.2.10 の `shadow(a[1][1][1])` のとおり、各次元の前方と後方の要素に 1 つずつ袖領域を持つ。そのため、`__private` 配列 a は各次元に前後計 2 要素を足したサイズ [10][10][6] を PE 上に確保している。袖が付くため、変換後のループでは袖を計算しないようにコンパイルされている。なお、元のコードで境界が計算されないため、変換後のコードも境界に当たる要素を計算から弾かなければならないが、MNCL 側の仕様が決まっていないことから暫定的にリスト 3.2.11 のような変換としている。なお、OpenACC 側では SIMD 命令のマスク処理による if 文の実行を変換後のコードで表現するた

めに、

```
if(L2B_id == 0 && L1B_id == 0 && MAB_id == 0 && PE_id == 0 && i == 0)
    continue;
```

といった記述を MNCL 側でできるようにし、境界要素がある PE の場所とループ回の指定を表現できるようにする MNCL 仕様の提案を行った。

halo_comm 関数についても、MNCL 側で未対応であることから OpenACC 側で提案した関数である。引数は左から順に、配列名、1 要素あたりのサイズ、合計次元数、各次元の要素数のリスト（配列か構造体）、各次元の袖領域の幅（配列か構造体）である。暫定的な関数仕様ではあるが、デフォルト分割とそれに応じた PE 上のデータ配置が決まっていれば処理が可能な関数としている。

その他、異なるカーネル関数で同じデータを扱う際に、PE 上にデータを保持し続ける MNCL 構文の提案を OpenACC 側から行った。

リスト 3.2.10　OpenACC/MN の 3 次元ステンシル計算コード

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #define N 128
 4 #define N_X N
 5 #define N_Y N
 6 #define N_Z N
 7
 8 int main() {
 9     double (*a)[N_Y][N_Z] = (double(*)[N_Y][N_Z])malloc(sizeof(double
       ) * N_X * N_Y * N_Z);
10     double c1 = -1.5;
11
12     for(int i=0; i<N_X; i++) {
13         for(int j=0; j<N_Y; j++) {
14             for(int k=0; k<N_Z; k++) {
15                 a[i][j][k] = 1.0;
16             }
17         }
18     }
19
20     double (*temp)[N_Y][N_Z] = (double(*)[N_Y][N_Z])malloc(sizeof(
       double) * N_X * N_Y * N_Z);
21
22 #pragma acc enter data copyin(temp[:N_X][:N_Y][:N_Z], a[:N_X][:N_Y][:
    N_Z])
23
24 #pragma acc parallel present(a[:N_X][:N_Y][:N_Z], temp[:N_X][:N_Y][:
    N_Z]) shadow(a[1][1][1])
25     for(int count=0; count<1000; count++) {
```

```
26 #pragma acc loop gang collapse(3)
27         for(int i=1; i<128-1; i++){
28             for(int j=1; j<128-1; j++){
29                 for(int k=1; k<128-1; k++){
30                     temp[i][j][k] =
31                         c1*(a[i-1][j][k]+a[i+1][j][k]
32                         +a[i][j-1][k]+a[i][j+1][k]
33                         +a[i][j][k-1]+a[i][j][k+1]
34                         +a[i][j][k]);
35                 }
36             }
37         }
38 #pragma acc loop gang collapse(3)
39         for(int i=1; i<128-1; i++){
40             for(int j=1; j<128-1; j++){
41                 for(int k=1; k<128-1; k++){
42                     a[i][j][k] = temp[i][j][k];
43                 }
44             }
45         }
46
47 #pragma acc reflect(a)
48
49     } // count の for 文（＝オフロード適用範囲）の終了
50
51 #pragma acc exit data copyout(a[:N_X][:N_Y][:N_Z], temp[:N_X][:N_Y][:
       N_Z])
52 }
```

リスト 3.2.11　MNCL に変換後の 3 次元ステンシル計算のデバイスコード

```
1  __kernel void main_kernel0(
2  __global double _arg_temp[128][128][128],
3  __global double _arg_a[128][128][128],
4  __global double _arg_c1
5  ) {
6      // 1PEあたりの袖領域を含めたデータサイズ
7      __private double temp[8][8][4];
8      bm2 double temp_bm2[8 * 4][8 * 4][4 * 32];
9      bm1 double temp_bm1[8 * 4][8 * 4][4 * 4];
10     __private double a[10][10][6];
11     bm2 double a_bm2[10 * 4][10 * 4][6 * 32];
12     bm1 double a_bm1[10 * 4][10 * 4][6 * 4];
13     __private double c1;
14
```

```
15      distribute(temp_bm1, temp_bm2, _arg_temp, 8 * 8 * 4);
16      distribute_pe(temp, temp_bm1, 8 * 8 * 4);
17      distribute(a_bm1, a_bm2, _arg_a, 10 * 10 * 6);
18      distribute_pe(a, a_bm1, 10 * 10 * 6);
19      broadcast(&c1, &_arg_c1, 1);
20
21      {
22          int count;
23          for(count = 0; (count) < (1000); (count)++)
24          {
25              {
26                  int i;
27                  for(i = 1; (i) < (9); (i)++)
28                  {
29                      {
30                          int j;
31                          for(j = 1; (j) < (9); (j)++)
32                          {
33                              {
34                                  int k;
35                                  for(k = 1; (k) < (5); (k)++)
36                                  {
37                                      temp[(i) - (1)][(j) - (1)][(k) -
                                            (1)] = (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)])) + (a[i][j][k
                                            ]));
38                                  }
39                              }
40                          }
41                      }
42                  }
43              }
44              {
45                  int i;
46                  for(i = 1; (i) < (9); (i)++)
47                  {
48                      {
49                          int j;
50                          for(j = 1; (j) < (9); (j)++)
51                          {
```

```
52                                  {
53                                      int k;
54                                      for(k = 1; (k) < (5); (k)++)
55                                      {
56                                          a[i][j][k] = temp[(i) - (1)][(j)
                                               - (1)][(k) - (1)];
57                                      }
58                                  }
59                              }
60                          }
61                      }
62              }
63              int a_size[3] = {10, 10, 6};
64              int a_shadow[3][2] = {{1, 1}, {1, 1}, {1, 1}};
65              halo_comm(a, sizeof(double), 3, a_size, a_shadow);
66          }
67      }
68
69      collect_pe(temp_bm1, temp, 8 * 8 * 4);
70      collect(_arg_temp, temp_bm2, temp_bm1, 8 * 8 * 4);
71      collect_pe(a_bm1, a, 10 * 10 * 6);
72      collect(_arg_a, a_bm2, a_bm1, 10 * 10 * 6);
73  }
```

前述のとおり、変換後の MNCL が検討中であるため、暫定的なコードではあるものの、計算の意味を変えないように 3 次元以下の配列のみに修正した姫野ベンチマークのコードでも同様の変換を行えることが確認できた。

コードが長いため、中核の計算箇所である jacobi 関数のみ抜粋した OpenACC/MN のコードをリスト 3.2.12 に、コンパイル後の MNCL デバイスコードをリスト 3.2.13 に示す。なお、MNCL 側で仕様が固まっていなかったため、暫定的に `reduction` の変換を保留しているが、

```
1  reduce_max_pe(buf_bm1, &v, 1);
2  reduce_max(out_v, buf_bm2, buf_bm1, 1);
```

といったすでに変換可能な distribute 関数や collect 関数に近い関数として MNCL 側で実装される想定である。

リスト 3.2.12　OpenACC/MN の姫野ベンチマークのソースコード

```
1  float
2  jacobi(int nn)
3  {
4    float gosa,s0,ss;
5
6  #pragma acc enter data copyin(p[:MIMAX][:MJMAX][:MKMAX], wrk1[:MIMAX
       ][:MJMAX][:MKMAX], \
```

```
7    wrk2[:MIMAX][:MJMAX][:MKMAX], bnd[:MIMAX][:MJMAX][:MKMAX]) shadow(p
        [1][1][1])
8
9  #pragma acc parallel present(p[:MIMAX][:MJMAX][:MKMAX], wrk1[:MIMAX
       ][:MJMAX][:MKMAX], \
10   wrk2[:MIMAX][:MJMAX][:MKMAX], bnd[:MIMAX][:MJMAX][:MKMAX]) shadow(p
        [1][1][1])
11 {
12   int i,j,k,n;
13
14   for(n=0 ; n<nn ; ++n){
15     gosa = 0.0;
16
17 #pragma acc loop gang collapse(3) reduction(+:gosa)
18     for(i=1 ; i<IMAX-1 ; ++i)
19       for(j=1 ; j<JMAX-1 ; ++j)
20         for(k=1 ; k<JMAX-1 ; ++k){
21           s0 = a012 * p[i+1][j  ][k  ]
22              + a012 * p[i  ][j+1][k  ]
23              + a012 * p[i  ][j  ][k+1]
24              + b * ( p[i+1][j+1][k  ] - p[i+1][j-1][k  ]
25                                       - p[i-1][j+1][k  ] + p[i-1][j-1][k  ]
                                           )
26              + b * ( p[i  ][j+1][k+1] - p[i  ][j-1][k+1]
27                                       - p[i  ][j+1][k-1] + p[i  ][j-1][k-1]
                                           )
28              + b * ( p[i+1][j  ][k+1] - p[i-1][j  ][k+1]
29                                       - p[i+1][j  ][k-1] + p[i-1][j  ][k-1]
                                           )
30              + c * p[i-1][j  ][k  ]
31              + c * p[i  ][j-1][k  ]
32              + c * p[i  ][j  ][k-1]
33              + wrk1[i][j][k];
34           ss = ( s0 * a3 - p[i][j][k] ) * bnd[i][j][k];
35           gosa += ss*ss;
36
37           wrk2[i][j][k] = p[i][j][k] + omega * ss;
38         }
39
40 #pragma acc loop gang collapse(3)
41     for(i=1 ; i<IMAX-1 ; ++i)
42       for(j=1 ; j<JMAX-1 ; ++j)
43         for(k=1 ; k<JMAX-1 ; ++k)
44           p[i][j][k] = wrk2[i][j][k];
```

```
45
46 #pragma acc reflect (p)
47
48   } /* end n loop */
49
50 }
```

リスト 3.2.13　MNCL に変換後の姫野ベンチマークのデバイスコード

```
1  __kernel void jacobi_kernel1(
2  __global float _arg_p[64][64][128],
3  __global float _arg_wrk1[64][64][128],
4  __global float _arg_bnd[64][64][128],
5  __global float _arg_wrk2[64][64][128],
6  __global int _arg_nn,
7  __global float _arg_gosa,
8  __global float _arg_s0,
9  __global float _arg_a012,
10 __global float _arg_b,
11 __global float _arg_c,
12 __global float _arg_ss,
13 __global float _arg_a3,
14 __global float _arg_omega
15 ) {
16     __private float p[6][6][6];
17     bm2 float p_bm2[6 * 4][6 * 4][6 * 32];
18     bm1 float p_bm1[6 * 4][6 * 4][6 * 4];
19     __private float wrk1[4][4][4];
20     bm2 float wrk1_bm2[4 * 4][4 * 4][4 * 32];
21     bm1 float wrk1_bm1[4 * 4][4 * 4][4 * 4];
22     __private float bnd[4][4][4];
23     bm2 float bnd_bm2[4 * 4][4 * 4][4 * 32];
24     bm1 float bnd_bm1[4 * 4][4 * 4][4 * 4];
25     __private float wrk2[4][4][4];
26     bm2 float wrk2_bm2[4 * 4][4 * 4][4 * 32];
27     bm1 float wrk2_bm1[4 * 4][4 * 4][4 * 4];
28     __private int nn;
29     __private float gosa;
30     __private float s0;
31     __private float a012;
32     __private float b;
33     __private float c;
34     __private float ss;
35     __private float a3;
36     __private float omega;
```

```
37
38      distribute(p_bm1, p_bm2, _arg_p, 6 * 6 * 6);
39      distribute_pe(p, p_bm1, 6 * 6 * 6);
40      distribute(wrk1_bm1, wrk1_bm2, _arg_wrk1, 4 * 4 * 4);
41      distribute_pe(wrk1, wrk1_bm1, 4 * 4 * 4);
42      distribute(bnd_bm1, bnd_bm2, _arg_bnd, 4 * 4 * 4);
43      distribute_pe(bnd, bnd_bm1, 4 * 4 * 4);
44      distribute(wrk2_bm1, wrk2_bm2, _arg_wrk2, 4 * 4 * 4);
45      distribute_pe(wrk2, wrk2_bm1, 4 * 4 * 4);
46      broadcast(&nn, &_arg_nn, 1);
47      broadcast(&gosa, &_arg_gosa, 1);
48      broadcast(&s0, &_arg_s0, 1);
49      broadcast(&a012, &_arg_a012, 1);
50      broadcast(&b, &_arg_b, 1);
51      broadcast(&c, &_arg_c, 1);
52      broadcast(&ss, &_arg_ss, 1);
53      broadcast(&a3, &_arg_a3, 1);
54      broadcast(&omega, &_arg_omega, 1);
55
56      {
57          int i;
58          int j;
59          int k;
60          int n;
61          for(n = 0; (n) < (nn); ++(n))
62          {
63              gosa = 0.0;
64              for(i = 1; (i) < (5); ++(i))
65              {
66                  for(j = 1; (j) < (5); ++(j))
67                  {
68                      for(k = 1; (k) < (5); ++(k))
69                      {
70                          s0 = ((((((((((a012) * (p[(i) + (1)][j][k]))
                                 + ((a012) * (p[i][(j) + (1)][k]))) + ((
                                 a012) * (p[i][j][(k) + (1)]))) + ((b) *
                                 ((((p[(i) + (1)][(j) + (1)][k]) - (p[(i) +
                                  (1)][(j) - (1)][k])) - (p[(i) - (1)][(j)
                                 + (1)][k])) + (p[(i) - (1)][(j) - (1)][k
                                 ])))) + ((b) * ((((p[i][(j) + (1)][(k) +
                                 (1)]) - (p[i][(j) - (1)][(k) + (1)])) - (p
                                 [i][(j) + (1)][(k) - (1)])) + (p[i][(j) -
                                 (1)][(k) - (1)])))) + ((b) * ((((p[(i) +
                                 (1)][j][(k) + (1)]) - (p[(i) - (1)][j][(k)
```

```
                                + (1)])) - (p[(i) + (1)][j][(k) - (1)]))
                               + (p[(i) - (1)][j][(k) - (1)]))))) + ((c) *
                                (p[(i) - (1)][j][k]))) + ((c) * (p[i][(j)
                                - (1)][k]))) + ((c) * (p[i][j][(k) -
                               (1)]))) + (wrk1[(i) - (1)][(j) - (1)][(k)
                               - (1)]);
71                          ss = (((s0) * (a3)) - (p[i][j][k])) * (bnd[(i
                               ) - (1)][(j) - (1)][(k) - (1)]);
72                          gosa += (ss) * (ss);
73                          wrk2[(i) - (1)][(j) - (1)][(k) - (1)] = (p[i
                               ][j][k]) + ((omega) * (ss));
74                      }
75                  }
76              }
77              for(i = 1; (i) < (5); ++(i))
78              {
79                  for(j = 1; (j) < (5); ++(j))
80                  {
81                      for(k = 1; (k) < (5); ++(k))
82                      {
83                          p[i][j][k] = wrk2[(i) - (1)][(j) - (1)][(k) -
                                (1)];
84                      }
85                  }
86              }
87              int p_size[3] = {6, 6, 6};
88              int p_shadow[3][2] = {{1, 1}, {1, 1}, {1, 1}};
89              halo_comm(p, sizeof(float), 3, p_size, p_shadow);
90          }
91      }
92
93      collect_pe(p_bm1, p, 6 * 6 * 6);
94      collect(_arg_p, p_bm2, p_bm1, 6 * 6 * 6);
95      collect_pe(wrk1_bm1, wrk1, 4 * 4 * 4);
96      collect(_arg_wrk1, wrk1_bm2, wrk1_bm1, 4 * 4 * 4);
97      collect_pe(bnd_bm1, bnd, 4 * 4 * 4);
98      collect(_arg_bnd, bnd_bm2, bnd_bm1, 4 * 4 * 4);
99      collect_pe(wrk2_bm1, wrk2, 4 * 4 * 4);
100     collect(_arg_wrk2, wrk2_bm2, wrk2_bm1, 4 * 4 * 4);
101 }
```

最後に性能評価について述べる。MNCL 実装が 2 次元以上の配列のコンパイルに非対応のため、実機での実行は難しかったものの、MN-Core は固定周波数であり、1 命令あたりのサイクル数（＝実行時間）も決まっているため、アセンブリを確認することで定量的な性能予測が可能である。

よって、リスト 3.2.10 の 3 次元ステンシル計算コードを OpenACC/MN コンパイラで変換して生成されたリスト 3.2.11 の MNCL デバイスコードについて、アセンブリによる命令数ベースでの性能評価を行った。

ループの中身の変換のみに限定されるものの、LLVM ベースの MNCL の簡易実装を別途用意してバイナリと対になるアセンブリの手前の抽象アセンブリまで変換した。抽象アセンブリでも命令数の確認は可能である。ベクトル命令の指示構文は MNCL 側で明確になっていないため、OpenACC/MN コンパイラでも保留としたが、この簡易実装で対応するベクトル命令化のためのコードを追記して、ベクトル命令の使用を試みた。

結果、ループの中身について、ベクトル命令の使用には実装上の課題が残るものの、ストールの無い理論上の限界性能に相当するアセンブリに変換できることを確認した。リスト 3.2.14 はアセンブリの一部（リスト 3.2.10 の 37 行目、ループの中身の計算箇所）を抜粋したものである。

リスト 3.2.14　実際に OpenACC/MN からのコンパイルにより生成された抽象アセンブリ（抜粋）

```
1 passad in=array[127]s1 out=%pass.0 masks: nop=0
2 faddd in=array[7]s1 fb_alu out=%add23_s0_double t masks: nop=0
3 faddd in=fb_mau array[61]s1 out=%add31_s0_double t masks: nop=0
4 faddd in=fb_mau array[73]s1 out=%add39_s0_double t masks: nop=0
5 faddd in=fb_mau array[66]s1 out=%add47_s0_double t masks: nop=0
6 faddd in=fb_mau array[68]s1 out=%add55_s0_double t masks: nop=0
7 fmuld in=fb_mau CONST_double_1.000000e-01 out=array[67]s1 t masks:
    nop=0
```

3.3　DSL 検討

岩澤 全規 [独立行政法人 国立高等専門学校機構 松江工業高等専門学校]

主に並列粒子法プログラム開発フレームワーク (FDPS) の開発を行った。特に近年提案された、Particle Mesh Multipole Method(PM^3)(Nitadori 2014) のモジュールを開発し、FDPS から呼び出せるように改良を行った。

3.3.1　PM^3 の概要

PM^3 は主に周期境界条件下での粒子間相互作用を効率よく計算する手法である。計算の流れは FMM の様に粒子分布から多重極展開を作り、そこから局所展開を求め粒子への力を計算する。局所展開の式は以下のように書かれる。

$$L_i = \sum_j G_{i-j} M_j \tag{3.3.1}$$

ここで、L_i は格子点 i での局所展開の係数、M_j は格子点 j での多重極展開の係数、G_{i-j} は格子点 i と j の距離に依存する変換行列である。FMM では行列積として M2L 変換を行うが、PM^3 ではこ

の変換を FFT による畳み込み計算を用いて以下のように計算する事で計算量を減らす。

$$\tilde{M}_k = \mathcal{F}\{M_j\} \tag{3.3.2}$$

$$\tilde{G}_k = \mathcal{F}\{G_{i-j}\} \tag{3.3.3}$$

$$\tilde{L}_k = \tilde{G}_k \tilde{M}_k \tag{3.3.4}$$

$$L_i = \mathcal{F}^{-1}\{\tilde{L}_k\} \tag{3.3.5}$$

3.3.2 多重極展開、局所展開、ポテンシャルの計算方法

ここでは、多重極展開から局所展開を求め最終的にポテンシャルを計算する方法を示す。まず、$1/r$ ポテンシャルを念頭に以下のような展開を考える。

$$\frac{1}{|\mathbf{S}-\mathbf{R}|} = \sum_{l=0}^{\infty}\sum_{m=-l}^{l} S_l^{-m}(\mathbf{S}) R_l^m(-\mathbf{R}), \tag{3.3.6}$$

$$R_l^m(r,\theta,\phi) = \frac{r^l}{(l+m)!} P_l^m(\cos\theta) e^{im\phi}, \tag{3.3.7}$$

$$S_l^m(r,\theta,\phi) = (-1)^{l+m}\frac{(l-m)!}{r^{l+1}} P_l^m(\cos\theta) e^{im\phi}. \tag{3.3.8}$$

ここで、$|\mathbf{S}| > |\mathbf{R}|$ とする。また、R_l^m は regular solid harmonics、S_l^m は singular solid harmonics と呼ばれる。さらに、$\mathbf{r}_i$ にある電荷 q_i が作る $\mathbf{r}_M$ での多重極展開、$M_l^m|_{\mathbf{r}_M}$ を以下のように定義する (P2M)。

$$M_l^m|_{\mathbf{M}} = \sum_i q_i R_l^m(\mathbf{M}-\mathbf{r}_i). \tag{3.3.9}$$

$\mathbf{r}_M$ での多重極展開 ($M_l^m|_{\mathbf{r}_M}$) から $\mathbf{r}_L$ での局所展開 ($L_l^m|_{\mathbf{r}_L}$) への変換は以下のように表すことができる (M2L)。

$$L_l^m|_{\mathbf{r}_L} = \sum_{\lambda=0}^{p}\sum_{\mu=-\lambda}^{\lambda} M_\lambda^\mu|_{\mathbf{r}_M} S_{l+\lambda}^{-(m+\mu)}(\mathbf{r}_L-\mathbf{r}_M). \tag{3.3.10}$$

局所展開を用いると $\mathbf{r}$ でのポテンシャル $\Phi(\mathbf{r})$ は以下のように書ける (L2P)。

$$\Phi(\mathbf{r}) = \sum_{l=0}^{\infty}\sum_{m=-l}^{l} L_l^m|_{\mathbf{r}_L} R_l^m(\mathbf{r}-\mathbf{r}_L). \tag{3.3.11}$$

ただし、このポテンシャルが収束するためには $|\mathbf{r}-\mathbf{r}_L| < |\mathbf{r}-\mathbf{r}_M|$ である必要があり、$\mathbf{r}$ の近傍の粒子からの寄与は粒子同士の直接相互作用として計算する必要がある。本実装では Barnes-Hut Tree 法を採用してこの部分を高速に計算している。

3.3.3 PM^3 での計算手順

ここでは、PM^3 の具体的な計算手順に関して記す。

先ず、計算領域全体を規則格子に分割し、各格子内の粒子が作る多重極展開 M_l^m を計算する。

次に M2L により局所展開を求める。PM^3 モジュールでは周期境界条件を考慮しているため、全てのセルからの多重極モーメントによる局所展開は以下のようになる。

$$L_l^m|_{\mathbf{r}_L} = \sum_{\mathbf{n}} \sum_{M} \sum_{\lambda=0}^{p} \sum_{\mu=-\lambda}^{\lambda} M_\lambda^\mu|_{\mathbf{r}_M} S_{l+\lambda}^{-(m+\mu)}(\mathbf{r}_L - (\mathbf{r}_M + \mathbf{r}_\mathbf{n})). \tag{3.3.12}$$

ここで、M に関する和は、計算領域全体の格子点に対する和を表し、$\mathbf{n}$ は周期境界条件により無限に存在するイメージセルを表す。また、$\mathbf{r_n} = (n_x L_x, n_y L_y, n_z L_z)$ とし (L_x, L_y, L_z) は計算領域の長さであり、(n_x, n_y, n_z) はそれぞれの方向の格子数を表す整数ベクトルである。

さらに $L_l^m|_{\mathbf{r}_L}$ の収束を速めるために以下のような分割を考える。

$$G_{l,\lambda}^{m,\mu}|_{\mathbf{r}_L - \mathbf{r}_M} = \sum_{\mathbf{n}} S_{l+\lambda}^{-(m+\mu)}(\mathbf{r}_L - (\mathbf{r}_M + \mathbf{r}_\mathbf{n})) = \tag{3.3.13}$$

$$\left(\sum_{\mathbf{n}} \frac{\Gamma(l+\lambda+1/2, \alpha^2 |\mathbf{r}_L - (\mathbf{r}_M + \mathbf{r}_\mathbf{n})|^2)}{\Gamma(l+\lambda+1/2)} S_{l+\lambda}^{-(m+\mu)}(\mathbf{r}_L - (\mathbf{r}_M + \mathbf{r}_\mathbf{n})) \right. \tag{3.3.14}$$

$$\left. + \sum_{\mathbf{n} \neq \mathbf{0}} \frac{(-i\pi)^{(l+\lambda)}}{\sqrt{\pi}} \frac{\exp(-(\pi k_\mathbf{n})/\alpha)^2}{\Gamma(l+\lambda+1/2)} \frac{k_\mathbf{n}^{2(l+\lambda)-1}}{V} S_{l+\lambda}^{-(m+\mu)}(\mathbf{k_n}) \cdot \exp(2\pi i \mathbf{k_n} \cdot (\mathbf{r}_L - \mathbf{r}_M)) \right) \tag{3.3.15}$$

ここで、第一項は実空間での計算、第二項は波数空間での計算を表す。また、$\mathbf{k}_n = (n_x/L_x, n_y/L_y, n_z/L_z)$, $V = L_x L_y L_z$ は計算領域の体積である。

これを用いて、

$$L_l^m|_{\mathbf{r}_L} = \sum_{\lambda=0}^{p} \sum_{\mu=-\lambda}^{\lambda} \sum_{M} M_\lambda^\mu|_{\mathbf{r}_M} G_{l,\lambda}^{m,\mu}|_{\mathbf{r}_L - \mathbf{r}_M} \tag{3.3.16}$$

と書ける。

これを畳み込み演算とみなし、FFT を用いて高速に $L_l^m|_{\mathbf{r}_L}$ を計算する。

3.3.4 FDPS による PM^3 の計算手順

PM^3 モジュールを用いた FDPS での相互作用は以下のような手順で計算される。

1. 逆格子空間での変換行列 $\tilde{G}$ をあらかじめ計算し全プロセスで保存しておく。
2. 計算領域を分割し、各プロセスに担当領域を割り当て, 粒子をプロセス間で交換する。
3. 各プロセスは担当の粒子を用いてローカルツリー構造を構築し、ツリーセルの多重極展開を計算する。この時 PM^3 のセルとツリーのセルがあるレベルで一致するように構成する。
4. 各プロセスは近傍粒子からなるツリーの情報 (LET) を近傍プロセスと交換する。

5. 各プロセスは再び多重極展開を求める。ここで、近傍の PM^3 セルの多重極展開が求まる。
6. 各プロセスは多重極展開を用いて近傍粒子との相互作用を Tree 法で計算する。
7. M2L 変換により局所展開を求め、遠方粒子からの寄与を計算する。これは以下の手順で行う。
   (a) 全プロセス数 (N_p) と多重極展開の要素数 ($(p+1)^2$) を比較し、$N_p \geq (p+1)^2$ ならば、複数のプロセスが一つの多重極展開要素を、それ以外ならば複数の要素を 1 つのプロセスが担当するようにプロセスグループを決め、要素を各プロセスに分配する。
   (b) 各プロセスグループで多重極展開をフーリエ変換する。この時点で、プロセスは担当する多重極展開要素のフーリエ変換 $\tilde{M}_k$ を得る。
   (c) 各プロセスグループで $\tilde{M}_k$ と $\tilde{G}_k$ を用いて $\tilde{L}_k$ を計算する。この際、$\tilde{M}_k$ の全要素が必要なため、逆格子空間座標 k 方向に分割した部分的な $\tilde{M}_k$ ごとに適当なプロセスに分配し、そのプロセスで部分的な $\tilde{L}_k$ を求め、集約して $\tilde{L}_k$ を得る。
   (d) $\tilde{L}_k$ を逆フーリエ変換し局所展開を求め、それを必要とするプロセスに送信する。
   (e) 各プロセスは局所展開を用いて遠方粒子からの寄与を計算する。

ここで、手順 1 と 7 が PM^3 モジュールの担当であり、それ以外の手順は FDPS が担当する。

3.3.5 精度の確認

精度の確認の為に、立方体領域に一様に粒子をばらまいた時の重力を、PM^3 と TreePM 法の計算精度の比較を行った。図 3.3.1 は粒子の加速度の相対誤差の累積分布である。粒子数は 4096 とした。また、PM^3 のメッシュ数は 8^3、TreePM 法のメッシュ数は 256^3 とした。PM^3 ははるかに少ないメッシュ数で高い精度を達成できることが分かる。

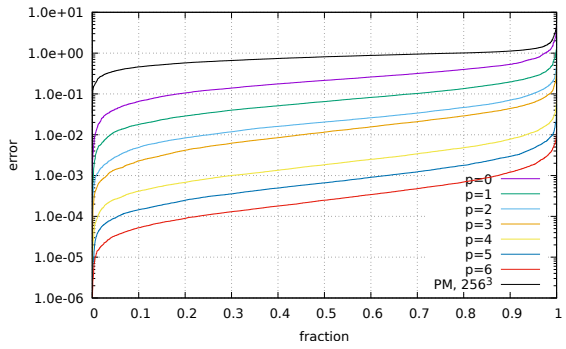

図 3.3.1　PM^3 法と TreePM 法の誤差の比較。

また、実際に宇宙論的大規模構造のシミュレーションも PM^3 法で行い、正しくシミュレーションできていることが確認された。

3.4　アクセラレータむけ最適化コンパイラの検討

遠藤 克浩 [国立研究開発法人 産業技術総合研究所]

産総研では、粒子系シミュレータの粒子間相互作用を計算する MN-Core 向け計算カーネルを出力するコンパイラの開発を進めた。粒子系シミュレータとは、多数の粒子が粒子間に働く相互作用によって力を受け、その合力によって運動する様子を数値的に計算するシミュレーションプログラムのことである。粒子間相互作用計算を PyTorch 実装に置き換えることは、計算効率の観点から望ましくなく、粒子系シミュレータのうち主要な計算処理を占めてる粒子間相互作用計算について、その実装を効率化する汎用的なコンパイラを実装することが求められた。

粒子間相互作用計算では、入力と出力がそれぞれ各粒子の位置と各粒子にかかる合力であるため、入出力サイズは粒子数 N に比例する。一方、相互作用計算の演算数は N の二乗に比例する（厳密には、近距離相互作用が無視できない粒子に対する計算量）。これは相互作用計算の対象となる粒子数が十分に大きければ、アクセラレータを使用することによる通信のオーバーヘッドを隠蔽できることを意味し、アクセラレータの活用が期待される。

産総研では、本プロジェクト開始前から粒子間相互作用計算を MN-Core で効率的に実行できるようにするための要素技術の開発を進めてきた。本プロジェクトでは、本プロジェクト以前の要素技術も組み合わせ、粒子間相互作用を計算する MN-Core 向け計算カーネルを出力するコンパイラ（以降、単にコンパイラと称する）の開発を進めた。本プロジェクトによる開発では、MN-Core だけではなく類似の構造を持つシステムでも汎用的に利用可能なコンパイラの開発に成功している。またユーザーがコンパイラを使用できるようにするためのソフトウェア整備を行った。3.4.1 節には、このコンパイラの仕様と構成についてまとめる。また、3.4.2 節には、コンパイラによって生成されたカーネルの例（最適化前・最適化済）を記載し、最適前後の効率化について説明する。さらに 3.4.3 節には、実際に MN-Core 2 を使用した重力多体シミュレーションを実行し、その性能評価について説明する。なお、以下の記述については、厳密な表記よりも説明のわかりやすさを優先した記述が含まれる。

3.4.1　コンパイラの仕様

コンパイラは、PIKG 高級言語で記述されたユーザーコードを入力とし、ユーザーコードに記述された計算内容を MN-Core で実行可能なカーネルに変換する機能を持つ。PIKG 高級言語で記述されたユーザーコードの例を表 3.4.1 に示す。表 3.4.1 のユーザーコードは、粒子相互作用計算のうち、1 つの粒子ペア、すなわち粒子 i と粒子 j の位置等の情報が与えられたときにその粒子間に働く力のベクトルを求める方法が記述されている。コンパイラが出力するカーネルは、この相互作用計算の方法を基に、複数の粒子に対するすべての粒子ペアの相互作用を求め、各粒子ごとにその合力を計算して出力する。

上記の機能を達成するため、図 3.4.1 のとおり、コンパイラはユーザーコードを次の順序で処理

表 3.4.1　PIKG 高級言語で記述されたユーザーコードの例（重力相互作用）

```
F32 eps2
rij = EPI.pos - EPJ.pos
r2 = rij * rij + eps2
rinv = rsqrt(r2)
mrinv = EPJ.mass * rinv
mrinv3 = mrinv * rinv * rinv
FORCE.acc += rij * mrinv3
```

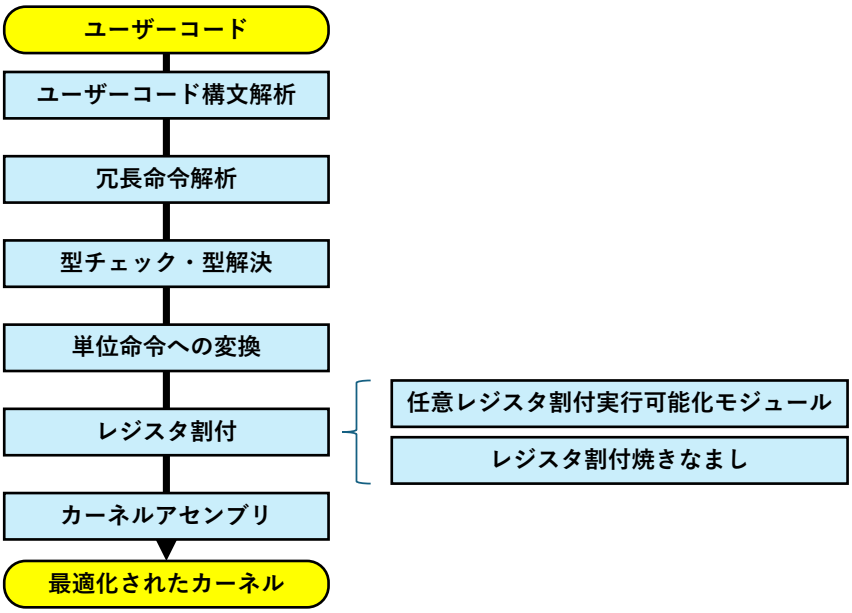

図 3.4.1　コンパイラのカーネル生成フロー

する。

- **ユーザーコード構文解析部分**：入力されたユーザーコードに対して構文解析処理を実行し、逐次的に処理できる形式の命令列に変換する。
- **冗長命令解析部分**：ユーザーコード構文解析部分によって生成された命令列のうち、変数の不要な複製などの冗長な命令を抽出し、同等の計算が可能なより短い命令列に変換する。
- **型チェック・型解決部分**：ユーザーコードの中で型が指定されていない変数について、型推論により型を割り当て、また型の指定に矛盾がないか判定する。
- **単位命令変換部分**：上記までに処理された命令列を、MN-Core で実行可能な単位命令と 1 対 1 対応する命令列に変換する。この処理には、積和演算などの 1 命令で実行可能な演算を発見してまとめる処理も含まれる。
- **レジスタ割付部分**：各命令の入出力を MN-Core のどのレジスタに割り当てるかを決定する。MN-Core ではレジスタ種類ごとに読み出しおよび書き込みの遅延が異なるため、レジスタ割

付を最適化することによって計算速度を改善することができる。詳細は後述する。

- **カーネルアセンブリ部分**：これまでに得られた命令列を含め、MN-Core 内部におけるデータ転送を含めたカーネルの生成を行う。

以上の処理によって、ユーザーコードから MN-Core で実行可能なカーネルが生成される。なお、レジスタ割付部分については、焼きなましによるレジスタ配置最適化を採用した。これは、次の手順に基づく。

- **任意レジスタ割付実行可能化モジュール**：このモジュールでは、どのようなレジスタ割付が与えられたとしても、最小限の遅延命令と代入命令を挿入することで、実行可能な命令列を生成する。不適切な割り当てとなっていれば、遅延命令と代入命令が多く挿入されることになり、生成された命令列は要求された計算を正しく計算可能であると保証されるが、その分計算速度が低下する。
- **レジスタ割付焼きなまし**：任意レジスタ割付実行可能化モジュールを使用することで、どのようなレジスタ割付パターンが与えられても、それによって生成される正しく実行可能な命令列の命令列長を評価することができる。これを用いて、命令列長が最小となるようなレジスタ割付パターンを焼きなまし法によって求める。焼きなまし法では、はじめランダムにレジスタ割付を変更しつつ、時間経過とともに、命令列長が大きくなりすぎる方向への変更を徐々に許容しないようにすることで、命令長を最小化する。

3.4.2 コンパイラにより生成されたカーネルの例

表 3.4.1 にある重力相互作用を計算するユーザーコードをコンパイラによってカーネルを記載する。リスト 3.4.2 は、「レジスタ割付部分」による最適化を実行せずに生成したカーネルを示す。また、リスト 3.4.3 は「レジスタ割付部分」による最適化を実行して生成したカーネルを示す。

リスト 3.4.2 および 3.4.3 に示すカーネルは、リスト 3.4.1 に示すような形式を 1 命令とし、それを逐次的に実行する形式となっている。各命令は、命令種別、型、入力レジスタ群、出力レジスタ群、マスク設定から構成されている。命令内容の詳細は省略するが、MN-Core が実行可能な単位命令と一対一している。

リスト 3.4.2 と 3.4.3 の比較からわかるとおり、最適化無しの場合は合計 52 命令、最適化済みの場合は合計 23 命令となっており、最適化による恩恵を受けていることがわかる。MNCore では 1 命令当たりの実行時間は一定であるため、命令数の削減が計算の高速化に直接寄与する。したがって、相互作用計算単体の計算時間においては、コンパイラ内部の最適化によって約 2.2 倍の性能向上を達成したことになる。

カーネル最適化の大部分は、無駄な待機命令が排除されたことによるものである。最適化前のカーネルでは、'nop' という入出力レジスタのない命令が大量に挿入されていることが見てとれる。この命令は実質的に計算を実行しておらず、その前後の命令で行ったレジスタ入出力が正しく処理されるために待機している。最適化後のカーネルでは、入出力のレジスタ種別が適切に選択される

ことにより、'nop' の挿入をすべて回避できていることがわかる。

リスト 3.4.1　カーネルの各命令の表記法

```
['命令種別:型', ['レジスタ種別.レジスタ番号（入力1）', ... ],
      ['レジスタ種別.レジスタ番号（出力1）', ... ], マスク設定]
```

リスト 3.4.2　コンパイラによって生成されたカーネル（最適化なし：52 命令）

```
['sub:F64', ['I.xi@x', 'J.xj@x'], ['R.10'], 0],
['sub:F64', ['I.xi@y', 'J.xj@y'], ['R.0'], 0],
['sub:F64', ['I.xi@z', 'J.xj@z'], ['R.21'], 0],
['set:F64', ['R.10'], ['T.'], 0],
['fma:F64', ['T.', 'T.', 'R.4'], ['R.12'], 0],
['set:F64', ['R.0'], ['T.'], 0],
['fma:F64', ['T.', 'T.', 'R.12'], ['R.7'], 0],
['set:F64', ['R.21'], ['T.'], 0],
['fma:F64', ['T.', 'T.', 'R.7'], ['R.8'], 0],
['nop:', [], [], 0],
['rsqrt:F64', ['R.8'], ['R.1'], 0],
['nop:', [], [], 0],
['mul:F64', ['R.1', 'R.1'], ['R.15'], 0],
['set:F64', ['R.8'], ['T.'], 0],
['mul:F64', ['R.15', 'T.'], ['R.9'], 0],
['set:F64', ['R.14'], ['M.s'], 0],
['set:F64', ['R.9'], ['T.'], 0],
['nop:', [], [], 0],
['fmi:F64', ['M.s', 'T.', 'R.13'], ['R.18'], 0],
['set:F64', ['R.1'], ['T.'], 0],
['mul:F64', ['T.', 'R.18'], ['R.17'], 0],
['nop:', [], [], 0],
['mul:F64', ['R.17', 'R.17'], ['R.2'], 0],
['set:F64', ['R.8'], ['T.'], 0],
['mul:F64', ['R.2', 'T.'], ['R.20'], 0],
['set:F64', ['R.22'], ['M.s'], 0],
['set:F64', ['R.20'], ['T.'], 0],
['nop:', [], [], 0],
['fmi:F64', ['M.s', 'T.', 'R.3'], ['R.19'], 0],
['set:F64', ['R.17'], ['T.'], 0],
['mul:F64', ['T.', 'R.19'], ['R.16'], 0],
['nop:', [], [], 0],
['mul:F64', ['R.16', 'R.16'], ['R.6'], 0],
['set:F64', ['J.mj'], ['T.'], 0],
['mul:F64', ['T.', 'R.16'], ['R.11'], 0],
['set:F64', ['R.6'], ['T.'], 0],
['mul:F64', ['T.', 'R.11'], ['R.5'], 0],
['nop:', [], [], 0],
```

```
39    ['set:F64', ['R.5'], ['M.s'], 0],
40    ['set:F64', ['R.10'], ['T.'], 0],
41    ['nop:', [], [], 0],
42    ['fmi:F64', ['M.s', 'T.', 'F.acc@x'], ['F.acc@x'], 0],
43    ['set:F64', ['R.5'], ['M.s'], 0],
44    ['set:F64', ['R.0'], ['T.'], 0],
45    ['nop:', [], [], 0],
46    ['fmi:F64', ['M.s', 'T.', 'F.acc@y'], ['F.acc@y'], 0],
47    ['set:F64', ['R.5'], ['M.s'], 0],
48    ['set:F64', ['R.21'], ['T.'], 0],
49    ['nop:', [], [], 0],
50    ['fmi:F64', ['M.s', 'T.', 'F.acc@z'], ['F.acc@z'], 0],
51    ['set:F64', ['F.pot'], ['T.'], 0],
52    ['sub:F64', ['T.', 'R.11'], ['F.pot'], 0]]
```

リスト 3.4.3　コンパイラによって生成されたカーネル（最適化済み：23 命令）

```
1     ['sub:F64', ['I.xi@x', 'J.xj@x'], ['N.10'], 0],
2     ['sub:F64', ['I.xi@y', 'J.xj@y'], ['N.0', 'T.'], 0],
3     ['sub:F64', ['I.xi@z', 'J.xj@z'], ['T.'], 0],
4     ['set:F64', ['T.'], ['M.21', 'T.'], 0],
5     ['fma:F64', ['N.10', 'N.10', 'R.4'], ['R.12', 'T.'], 0],
6     ['fma:F64', ['N.0', 'N.0', 'T.'], ['R.7', 'T.'], 0],
7     ['fma:F64', ['M.21', 'M.21', 'T.'], ['R.8', 'T.'], 0],
8     ['rsqrt:F64', ['T.'], ['R.1', 'T.'], 0],
9     ['mul:F64', ['T.', 'T.'], ['R.15', 'T.'], 0],
10    ['mul:F64', ['T.', 'R.8'], ['R.9', 'T.'], 0],
11    ['fmi:F64', ['R.14', 'T.', 'N.13'], ['R.18', 'T.'], 0],
12    ['mul:F64', ['R.1', 'T.'], ['R.17', 'T.'], 0],
13    ['mul:F64', ['T.', 'T.'], ['R.2', 'T.'], 0],
14    ['mul:F64', ['T.', 'R.8'], ['R.20', 'T.'], 0],
15    ['fmi:F64', ['N.22', 'T.', 'R.3'], ['R.19', 'T.'], 0],
16    ['mul:F64', ['R.17', 'T.'], ['R.16', 'T.'], 0],
17    ['mul:F64', ['T.', 'T.'], ['R.6'], 0],
18    ['mul:F64', ['J.mj', 'T.'], ['M.11', 'T.'], 0],
19    ['mul:F64', ['R.6', 'T.'], ['R.5', 'T.'], 0],
20    ['fmi:F64', ['T.', 'N.10', 'F.acc@x'], ['F.acc@x'], 0],
21    ['fmi:F64', ['T.', 'N.0', 'F.acc@y'], ['F.acc@y'], 0],
22    ['fmi:F64', ['T.', 'M.21', 'F.acc@z'], ['F.acc@z', 'T.'], 0],
23    ['sub:F64', ['F.pot', 'M.11'], ['F.pot', 'T.'], 0]]
```

3.4.3　コンパイラにより生成されたカーネルの性能評価

この節では、MN-Core 2 を使用して、重力多体系のシミュレーションを行った場合の結果について述べる。前節のとおり、コンパイラにより生成したカーネルを使用した計算を行うため、ホスト

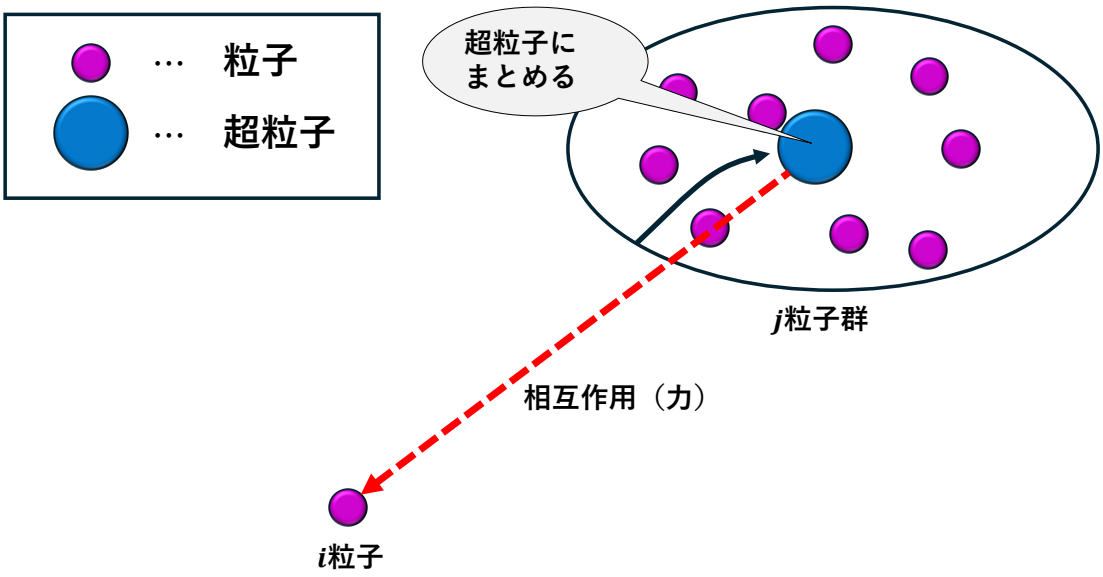

図 3.4.2　超粒子の生成による計算量の削減（概念図）

側と MN-Core 2 双方にまたがる Barnes-Hut 木アルゴリズムを使用した N 体シミュレーションを実装し、評価した。Barnes-Hut 木は、シミュレーション系全体のセルを再帰的に小さいセルへと分割し、各セルに属する粒子群という形の構造化データとしてシミュレーション中の全粒子を保持する。Barnes-Hut 木を使用した相互作用計算アルゴリズムでは、ある 1 つ粒子（i 粒子とよぶ）との相互作用を計算する際、自分自身以外のすべての粒子（j 粒子とよぶ）との相互作用を計算するのではなく、距離が離れた粒子群をある程度まとめて 1 つの超粒子としてみなし相互作用計算を行う。これにより、粒子数の二乗の相互作用計算を愚直に計算することを回避し、計算量を大幅に削減する。どの範囲の複数の粒子を超粒子にまとめるのかを構成する際に参照されるのが Barnes-Hut 木の再帰的な分割構造である。具体的には、i 粒子から見た際の角度が閾値を下回るような最も大きい分割セルが 1 つの超粒子に変換される（図 3.4.2）。これにより、より遠い粒子ほど、より多くの粒子をまとめた超粒子として変換されることになる。

ホスト側と MN-Core 2 双方にまたがる Barnes-Hut 木アルゴリズムでは、Barnes-Hut 木による超粒子への変換等はホスト側で実行する。MN-Core 2 側は、Barnes-Hut 木にもとづく粒子変換処理の結果から生成された、相互作用を計算すべき粒子群ペアのデータをもとに、その相互作用と合力の計算結果を、ホスト側に返却する。合力の計算結果をもとにした時間方向への数値積分の実行などのその他の計算は、ホスト側で行う。ホスト側の計算の大部分は、粒子シミュレータ開発フレームワークである FDPS を使用して実装を行った。一部、特にホスト側で動作する FDPS と MN-Core 2 の接続のための、MN-Core 特有の API に基づくデータ転送制御は、本プロジェクトでカスタム実装を行った。

具体的なシミュレーションの初期設定としては、次のとおりである。本検証では、Cold Collapse と呼ばれる、重力の下で銀河や銀河団などの構造がどのように形成されるかを考える 1 つのモデル

	CPU / MN-Core 2
粒子数	229376
ステップ数	16
一括処理粒子数	32 (CPU only), 64, 128, 256, 1024, 2048, 4096, 8192 (MN-Core 2 only)

表 3.4.2 Parameters of N-body benchmarks for CPU and accelerators (GH200 and MN-Core 2).

の計算を扱う。Cold Collapse に基づくシミュレーションモデルとして、初期状態として温度の低い均一な粒子分布から始まり、重力相互作用によって構造が形成される様子を計算した。このシミュレーションのパラメータ設定は表 3.4.2 のとおりである。ここで、並列計算において重要な一括処理粒子数について説明する。粒子を 1 つずつ処理すると並列計算の恩恵を得ることが難しいため、「一括処理粒子数」個分の粒子をグループ化して、一括で相互作用計算を実行する。一括処理粒子数が少なすぎれば並列化の影響を受けられず全体の計算性能が低下するが、一方一括処理粒子数が多すぎる場合は、無駄な相互作用計算をする可能性が上昇するため、同様に全体の計算性能が低下する。一括処理粒子数の適切な値は系の状況に依存するため、様々な一括処理粒子数での性能評価をすることが望ましい。そこで、本検証では、表 3.4.2 のように異なる一括処理粒子数を設定した場合の評価を行った。

上記の実験における性能評価の結果は、図 3.4.3 のとおりとなった。図 3.4.3 は、横軸に一括処理粒子数、縦軸にスループット（1 秒間に計算できたステップ数）を記載している。スループットは、一括処理粒子数が 64 から 512 ではおよそ 10 程度、1024 から 4096 ではおよそ 40 程度となっていることがわかる。比較用に、ホスト側で計算処理も行った場合の評価を図 3.4.4 に示す。ホスト側 CPU は、Intel Core i5 14500 であり、8 個の Efficient-core と 6 個の Performance-core を持つ。CPU での計算においては、AVX2 並列計算を実装している。図 3.4.4 より、ホスト側のみの計算によるスループットは、32 および 64 ではおよそ 15、128 から 512 ではおよそ 25 程度、1024 から 4096 では 15 程度となった。よって、一括処理粒子数が 1024 以上であれば、MN-Core を使用してシミュレーションを加速できることがわかった。また、CPU と MN-Core では、スループットのピーク性能が出る一括処理粒子数の範囲に大きな違いがあることもわかる。

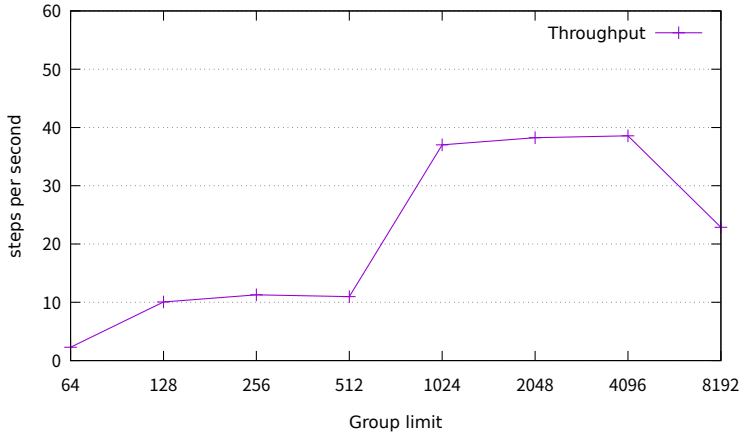

図 3.4.3　MN-Core 2 のスループット評価。

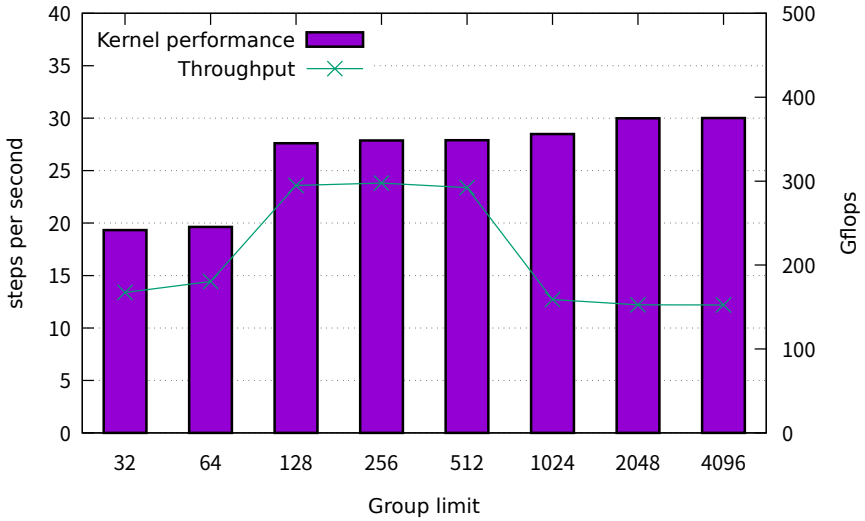

図 3.4.4　CPU(AVX2 並列化) のスループット評価（比較用）。

第 3 章の参考文献

遠藤 克浩 (2022). MN-Core 向け相互作用計算カーネルコンパイラ PIKG/MN-Core、 CPS セミナー. 惑星科学研究センター. URL: https://www.cps-jp.org/calendar/fy2022/2022-10-28/index.htm.

High Performance Fortran Forum (1997). *High Performance Fortran Language Specification*. High Performance Fortran Forum. URL: http://hpff.rice.edu/versions/hpf2/hpf-v20.pdf.

三木 洋平, 塙 敏博 (2024). "GPU 向け指示文統合マクロの実装". In: 情報処理学会研究報告. Vol. 2024-HPC-195. 2, pp. 1–11.

Nakao, M., H. Murai, T. Boku, and M. Sato (2018). "Performance Evaluation for Omni XcalableMP Compiler on Many-Core Cluster System Based on Knights Landing". In: *Proceedings of Workshops of HPC Asia*. HPC Asia '18. Association for Computing Machinery, pp. 52–58. DOI: 10.1145/3176364.3176372.

成瀬 彰 (2017). OpenACC で始める GPU コンピューティング：ループの最適化. 日本計算工学会. URL: https://www.jsces.org/activity/journal/files/tutorial_2201.pdf.

Nitadori, K. (2014). *Particle mesh multipole method: An efficient solver for gravitational/electrostatic forces based on multipole method and fast convolution over a uniform mesh*. arXiv: 1409.5981 [astro-ph.IM]. URL: https://arxiv.org/abs/1409.5981.

Tabuchi, A., Y. Kimura, S. Torii, H. Matsufuru, T. Ishikawa, T. Boku, and M. Sato (2016). "Design and Preliminary Evaluation of Omni OpenACC Compiler for Massive MIMD Processor PEZY-SC". In: *OpenMP: Memory, Devices, and Tasks: : 12th International Workshop on OpenMP (IWOMP 2016)*, pp. 293–305.

田渕 晶大 (2018). "演算加速機構を持つ並列システム向け PGAS 言語コンパイラの研究". 博士論文. 筑波大学.

綱島 隆太, 小林 諒平, 藤田 典久, 朴 泰祐, Lee Seyong, Vetter Jeffrey S., 村井 均, 中尾 昌広, 辻 美和子, 佐藤 三久 (2023). "OpenACC 単一記述による GPU+FPGA 複合デバイス処理システム". In: 情報処理学会論文誌コンピューティングシステム（ACS） 16.2, pp. 1–15.

綱島 隆太, 遠藤 克浩, 中里 直人, 井町 宏人, 牧野 淳一郎 (2024). "ポスト富岳世代の MN-Core ベースアクセラレータ対応 OpenACC のインターフェイスとコンパイラの検討および開発". In: 情報処理学会研究報告. Vol. 2024-HPC-195. 1, pp. 1–10.

綱島 隆太 (2024). "複数種の演算加速機構を持つ高性能計算システムにおけるプログラミングに関する研究". 博士論文. 筑波大学.

第 4 章　アプリケーション調査研究

4.1　アプリケーション調査研究の統括

姫野 龍太郎 [学校法人 順天堂]

アプリケーション調査研究全体を統括し、問題規模の設定、アルゴリズムの検討に対して必要なサポートを行った。また他グループへのアプリケーションからの提案点要求を整理し、コデザインを有効に進めた。具体的には、8 つのアプリケーション分野それぞれで、現在の計算速度では計算時間に問題があり、次世代のスーパーコンピュータを必要とし、今後発展するであろうコードを選定した。さらに、そのコードの現在の計算規模、および今後の計算規模を見積もり、カーネルの抽出に取り組んだ。この時、同時に、計算規模の見積もりも行った。これにより、最も重要なパラメータである計算とメモリーアクセスの比や通信量と回数、その間の演算量などを見積もることができるようになった。

それぞれのアプリケーション研究者にとって、アプリケーション全体での評価は厳しいため、基本的にはカーネル部分を抽出しての評価が中心となっている。これに関しては会津大学で開発中の、C 言語をベースとしたカーネル記述用コンパイラコンパイラがうまく対応できるようになれば、アプリケーション研究者にとっては面倒な部分がかなり解消されることになる。この点で種々のアプリケーションを取り込む上で敷居を低くすることにつながり、期待が高い。一方で、ものづくりアプリケーションの分野ではプログラムの Python 版を経由することで MN-Core 性能評価や最適化を検討できることを示してくれた。この方法は、他のアプリケーションでも利用可能であり、もうひとつの期待の方向である。

さらに、これまでのアプリケーション分野での検討に加えて、近年バイオ分野で次世代シーケンサーの能力が飛躍的に高まり、ヒトゲノムの変異を見つけるソフトウェアとハードウェアの高速化のニーズが急速に高まっている。現在開発に取り組んでいるプロセッサであるメニーコア・アーキテクチャで、この速度を飛躍的に高めることが可能であることが示唆されていることから、AMD のメニーコア・プロセッサにより、どの程度の高速化が可能であるか、計測することとした。このアプリケーション・ソフトウェアの開発を行った企業（先端加速システムズ株式会社）と協議し、ソフトウェアの使用の許諾を得るとともに、AMD 社製 64 コアの PC の導入を行ない、評価を行った。その結果、一般的な PC の 10 倍以上の計算速度を示し、メニーコア・アーキテクチャ用に最適化された計算コードが非常に高速に道産することが示された。（現在論文を準備中）

4.2 創薬と深層学習応用アプリケーションの検討

牧野 淳一郎 [国立大学法人神戸大学]

当初、創薬のためのアプリケーションとしてタンパク MD、深層学習応用として古典的な CNN を想定していたが、ここ数年で明らかにもっとも重要になるのはトランスフォーマーベースの LLM となり、これは近い将来に大きく変わらないように思われる。実際、 LLM を非常に長時間使うことで、プログラム開発や数学の未解決問題の解決といった、「知的」なタスクを実行可能であることが明らかになってきており、 LLM の重要性は急速に高まっている。このため、本節ではトランスフォーマーの性能推定についてまとめる。

LLM のメモリアクセス性能要求は、従来の CNN とは根本的に異なっている。典型的なトランスフォーマーでは、メモリアクセスはバッチあたりで

$$10C^2 + BLC/4 \tag{4.2.1}$$

計算量は

$$(20C^2 + LC/4)B \tag{4.2.2}$$

で与えられる。ここで C は チャネル数で、モデルサイズやタイプにあまり大きく依存しないで 8192 程度、 B は バッチサイズでこれは実行するシステムや実現したい速度に大きく依存し、 L は コンテクスト長 (文章の長さ) である。

このことから、モデルパラメータにほとんど依存しないで、アプリケーションが要求する B/F 値が以下で与えられることがわかる。

$$\mathrm{B/F} = \begin{cases} 1/(2B) & (\ L \ll C/B) \\ 1 & (L \gg C/B) \end{cases} \tag{4.2.3}$$

つまり、

- コンテクスト長が小さいうちは、バッチサイズを大きくすることで要求 B/F を小さくし、効率をあげることができる。
- しかし、$L \sim C/B$ くらいから B/F 要求が大きくなるので、大きなバッチサイズを要求するシステムでは長いコンテクストは扱えない。

ということがわかる。注意するべきことは、この B/F 値は FP8 や FP4 といった現在 LLM で実際に用いられている演算に対する B/F 値であることである。また、コンテキスト長は実際の応用では非常に長くなる傾向がある。例えば論文要約や、マニュアルを読ませてソフトウェア開発をさせるといったタスクでは現状ではこれらはコンテキスト側に入る。このため、バッチサイズにあまり依存しないで、必要な B/F 値が 1 に近付く傾向がある。

もちろん、論文要約の例では論文を読ませるところはいわゆる Prefill になり必要なメモリバンド幅は高くない。しかし、タスクが複雑なものになると Prefill ではない本体部分の比重が高くなり、メモリバンド幅要求が高くなる。このため、LLM 応用では、メモリバンド幅を有効に使えるようになっていれば、基本的にメモリバンド幅律速の性能が実現できる。

リファレンスアーキテクチャについて、LLM での実行効率を推定した。また、PFN では実際に 3D 積層 DRAM を用いた LLM むけプロセッサの開発を進めており、メモリバンド幅律速の限界に対して 50% 以上の実行効率を実現できる見込みである。

トランスフォーマーが AI の主流になったことは、ここで見るようにアプリケーション性能が完全にメモリバンド幅リミットになる、という従来の AI 応用とは全く異なるハードウェア要求をもたらしている。この要求に対応するには、プロセッサとメモリの物理的な距離を小さくすることが必須であり、なんらかの技術を使った主記憶とプロセッサの積層がもっとも有力な解答になるであろう。

4.3 ゲノム科学アプリケーションの検討

鎌谷 洋一郎 [国立大学法人 東京大学]

ゲノム科学の中でもヒトの全ゲノムシークエンス（WGS）解析は、特に大規模集団解析を行う場合、計算時間を要するプロセスであり、その効率化が求められている。実際、英国 UK Biobank や米国 All of Us などのプロジェクトでは 50 万人程度の WGS データが算出されている。さらにはシンガポール、タイ、アラブ首長国連邦などを含む世界各国で数万人規模以上の WGS を実施する大規模計画が次々と発表され、また進行している。このような背景から、計算性能の向上に加え、環境負荷の低減に配慮した計算基盤の重要性も高まっていると考えられる。

4.3.1 ショートリード・シークエンス解析の実行時間

WGS 解析では、一般的に 150 塩基対程度の短い DNA 配列（ショートリード）データを大量かつ高速に生成したのち、それらを参照ヒトゲノム配列にマッピング（bwa）した後、局所的に再構築された候補ハプロタイプ配列を用いて参照ゲノム配列との違い（遺伝的バリアント）を同定する（Haplotype Caller）。最後に集団レベルでのバリアントのフィルタリングと洗練化を行う一連の標準的パイプラインが存在する。

本検討では、比較対照として、全ゲノム解析等実行計画の難病での実施プロトコルに採用されている Parabricks についてパフォーマンスをまとめた。

全ゲノムシークエンス解析において、対応が必要とされるゲノムデータのファイルサイズは以下のとおりである（500 人の実データの平均値を元に算出）。

一人について実際の実行時間は以下のとおりである（東京大学医科学研究所　Shirokane5 を使用）。GPU 枚数と SSD のみ条件を振っている。

大容量ファイルを解析するとはいえ、ファイル IO はそれほどのボトルネックとならず、GPU 枚

表 4.3.1　ゲノムデータのファイルサイズ

ファイル形式	1 人 [GB]	2,000 人 [TB]	20,000 人 [TB]
fastq	55	110	1,100
bam	65	130	1,300
vcf	22	44	440

表 4.3.2　GPU 構成別の処理時間比較

構成	メモリ指定	CPU**	fq2bam (min)	Haplotype Caller (min)	bgzip + validate_pbrun (min)	
V100* 1 枚 ssd あり	256 GB	16	157.70	108.9	129.6	*
V100 1 枚 ssd なし	256 GB	16	164.65	102.1	–	
V100 2 枚 ssd あり	256 GB	16	78.52	56.2	–	
V100 2 枚 ssd なし	256 GB	16	87.20	56.5	–	

NVIDIA Tesla V100 SXM2, ** Intel Xeon Gold 6126

数で実行時間は決まることがわかった。

4.3.2　Illumina 社以外のショートリード・シーケンサー

AVITI は、Illumina 社とは別のショートリード・シークエンス技術で、基本原理は異なり、Illumina 社シーケンス結果と比較すると精度の高さで知られている。生成される塩基データの分布にも違いがあると考えられる。

AVITI 生成データの WGS 解析の実行時間について、同じ検体由来の DNA 解析結果による NovaSeq とも比較した結果を示す。

表 4.3.3　シーケンサー・キット別の実行条件および処理時間

シーケンサー・キット	塩基長	メモリ指定*	GPU	CPU	fq2bam (min)	HaplotypeCaller (min)
NovaSeq6000	150	960 GB	2	80	67.78	24.80
AVITI CloudBreak	150	480 GB	2	80	35.95	27.90
AVITI UltraQ	150	960 GB	2	80	27.60	20.62

* 最大メモリ使用量はいずれの条件においても 400G 以下であった。

同じ fq2bam パイプラインを使用し、同じ計算環境であっても、AVITI シーケンサーのデータ解析の方がかなり実行時間が速くなっていた。CloudBreak は NovaSeq より塩基コールの品質が高く、UltraQ はさらに高い。実行時間のトレンドが見えていることからも、精度が上がっているため、マッピングの過程でベストマッチの配列位置を探索しやすく、実行時間の高速化となったものと考えられる。

4.3.3 ロングリード・シークエンス解析の実行時間

Nanopore PromethION はナノスケールの穴を DNA 分子が通過する際に発生する電流をセンサで計測することで塩基配列を得る技術で、非常に長い塩基配列（ロングリード）を読むことができるロングリードシーケンサーの一つである。世界を代表するバイオバンクである UK Biobank が 5 万人の Nanopore シーケンサー解析を開始しており、Nanopore、あるいは競合の PacBio 社のロングリード・シークエンサーの大規模データが生成される傾向にある。これもゲノム科学アプリケーションの大きなトピックであると考えられる。そこでロングリードシークエンスの標準解析のパフォーマンスを対照として検討した。

電流データである fast5 から、塩基配列データである fastq ファイルに変換するには guppy（v 5.0.17）ソフトウェアを使用した。これは 3 層の CNN、5 層の RNN、Linear CRF Encoder からなる Deep Learning フレームワークで、2 GPU（NVDIA Tesla V100 SXM2）を使用したところ 815 min CPU 時間（user+sys）を要した。

その結果得られた塩基配列データは N50（長さの中央値に対応）が約 9,400 塩基対（bps）であった（NovaSeq6000 の場合は 150 塩基対なので、中央値で 60 倍を超える長さの配列データを得た）。総塩基数は 20 Gbps で、通常 100 Gbps 前後である NovaSeq6000 よりも少なく、シークエンス深度は 7x 程度である（昨年度報告した NovaSeq 結果では 30x であった）。

さらに、PromethION 技術では DNA メチル化（下記のワークフローでは、具体的には 5 メチル化シトシン）を検出できるためこれをさらに実施した。minimap2 を用いたアラインメントに 57808.554 sec CPU 時間（10CPU 使用で real 5950.485 sec）、メチル化を検出するプロセス（nanopolish call-methylation）に 5970 min CPU 時間（10CPU 使用で real 2978 min）であった。これらをまとめると、PromethION による配列解析は、NovaSeq の結果と比較しても、高速化を要する現状であると考えられた。

4.4 地震と構造物シミュレーションアプリケーションの検討（低次非構造有限要素解析）

堀 宗朗 [国立研究開発法人 海洋研究開発機構]

地震シミュレーションの対象は、長期の地殻変動、地震直後の地殻変動、断層から地表までの地震動伝播、地表近傍の堆積層内での地震動の増幅、地盤と構造物の連成を考慮した構造物の地震時挙動など多岐に渡る。これらの挙動は対象物の幾何形状および地表における応力フリーのなどに代表される境界条件の影響を強く受けるため、幾何形状を適切にモデル化出来、かつ、境界条件の取り扱いに優れた、低次非構造四面体要素を用いた有限要素解析が望まれる。一方で、地震シミュレーションでは領域が広大かつ要求分解能が高く大規模問題となる。加えて、地殻や地盤などの構造情報の不完全さや観測データを用いた最適化・データ同化のニーズも高まりつつあるため、これらのシミュレーションのさらなる高速化が望まれる状況にある。

このような状況を踏まえ、まず、ベースとなる低次非構造四面体要素を用いた有限要素解析につ

いて、既存のアルゴリズムでの演算カーネルの抽出、ノード間通信、I/O の必要量の推定を行い、新しいアルゴリズムの検討の方向性を明らかにした。具体的には、実機で想定される加速処理やデバイスあたりの問題サイズなどを検討し、GPU 実装での経験をもとに有限要素法における element by element 法（EbE 法）の効率的な実装により *time to solution* を改善するというアルゴリズム開発の方針を定めた。また、これらを実施するための疎行列ループの切り出しを行い、ターゲットとなるカーネルコードの整備を進めた。

上記を踏まえ、汎用 CPU での実行プロファイルとアクセラレータで実行する部分の性能評価に基づき、アプリケーションの性能評価を行った。具体的には、本解析（低次非構造四面体要素を用いた有限要素解析）による時刻歴解析では、時間ステップ毎に $Ax = f$ なる方程式を反復型 solver（共役勾配法）にて求解するが、その共役勾配法内のコアカーネルとなる行列ベクトル積 $q = Ap$ について EbE 法によりオンザフライで計算しており、この演算の性能により本解析の性能が左右される（なお、現状の計算機よりも演算性能に対してメモリ容量・メモリ帯域が相対的に小さくなる計算機においても性能を発揮できるような検討を進めていくためにも、本コアカーネルでは行列格納型の疎行列ベクトル積ではなく、行列ベクトル積の都度要素行列を生成し、疎行列ベクトル積をオンザフライで計算する EbE 法により演算を検討している。また、通常の有限要素法では A をメモリ上に保持するため必要メモリが多くなるが、EbE 法ではその必要がなくなる。そのため大自由度の問題を扱う本解析には EbE 法が適している）。すなわち、EbE 法は、オンフライで要素毎に要素剛性行列と関連する変数の積を計算し足しこんでいくため、計算量が多くなり、この効率的な処理を実現することがポイントとなる。実際、全体の計算量の内訳は、おおよそ EbE カーネル部分が 6〜7 割、共役勾配法の諸計算が 2 割強、変数の時間方向の update が 1 割弱となっている。この全体の計算量の 6〜7 割を占める EbE カーネル部分に対して、メモリアクセスや演算の実装の工夫を行い、既存の計算機である「富岳」に搭載されている A64FX CPU (Armv8.2-A) および ABCI の NVIDIA A100 GPU (SXM4 40 GB) における性能を測定した。表 4.4.1 に A64FX CPU 1 個、および、A100 GPU 1 個で得られた性能値を示す。A64fx では FP32 ピーク比で 9.6%、A100 では FP32 ピーク比で 12.7% という実効性能が得られており、演算量の多い EbE 法を用いているものの、time to solution の観点からも効率的な解析が実現されていることが分かる（なお、GH200 においても性能検証を行い、同様の高い性能が得られることを確認している）。なお、この EbE 法では CRS などにより疎行列をメモリ上に保持しなくてもよいため、省メモリとなるだけではなく、アーキテクチャによっては疎行列を CRS で保持した疎行列ベクトル積よりも高速に疎行列ベクトル積が実行可能となる場合がある。実際、演算性能に対してメモリ帯域が相対的に大きい A64FX CPU では CRS と EbE 法は同程度の計算速度になるが、演算性能が相対的に高い A100 GPU では EbE 法が CRS の約 3 倍高速になることが分かった。

以上の検討により、MN-Core での実測値と比較することで、当アプリケーションの適合性を評価できる状況となり、上記の EbE 法による有限要素法カーネルの実行効率の評価を進めた。その結果、EbE 法ではある程度メモリバンド幅が低い場合でも高い実行効率が得られるため、A64fx や A100 と同様の理論ピークに対して 10%程度ないしはそれより高い実行効率が期待できることが分かった。上記を踏まえ、実際にベンチマークコードを机上検討した結果では、MN-Core では EbE

法が必要とする主記憶への間接アクセスを高速に実行できるため、主記憶アクセス時間が計算時間に遮蔽可能であることがわかった。このため、実行効率はベクトル性能に対して 80%程度（積和にならない部分があるため)、理論ピークの 20%程度が期待できることが分かった。

表 4.4.1　Elapsed time and obtained performance of EbE-method computed in FP32.

System	Elapsed time	Ratio to FP32 peak	Ratio to peak memory bandwidth
A64FX	0.70 s	9.6%	7.5%
A100 SXM4 40 GB	0.14 s	12.7%	12.7%

4.5　気象・気候シミュレーションアプリケーションの検討

八代 尚 [国立研究開発法人 国立環境研究所]

気候・気象シミュレーションに用いられる代表的なアプリケーションとして、大気モデル、陸域モデル、海洋モデルが挙げられる。大気モデルは大気の風速、気圧、気温、水物質等の時間発展を球面上の流体現象として解く部分（流体力学過程）と、大気放射過程、水物質の微物理過程、サブグリッドスケールの乱流拡散過程、下端境界（地表面）での水熱交換過程、化学物質の微物理過程や化学反応過程をそれぞれ解く部分（諸物理過程）を持つ。大気モデルには地球全体の大気を同時に解く全球モデルと、限定した地域のみを対象とし側面境界条件を与える領域モデルがある。それぞれについて、スペクトル展開して波数空間上で流体力学を解くスペクトル法モデルと、格子に区切って計算する格子法モデルが存在する。計算並列化では水平方向に領域分割を行い分散メモリでの計算を行うことが主流である。大気の鉛直方向には大気放射過程や乱流混合過程のように、陰解法で解くことが必要な計算があるため、鉛直方向の計算プロセス分割は一般的ではない。陸域モデルは、地球の陸面上を規則的または不規則的な領域で区切り、各領域での植物、土壌を介したエネルギー収支、水・炭素量等の物質収支を解く部分と、河川等による領域間の水の移動を解く部分を持つ。河川以外の過程のほとんどは領域間の依存関係がないため、水平方向に分割して並列化を行うことが容易である。河川の移流計算のためのネットワーク通信は複雑であるが、シミュレーション中に通信相手が変わることはなく、また全体からみて所要計算時間は短い。海洋モデルは大気モデルと同様に、流体力学過程と諸物理過程を持つが、海洋の場合陸地が存在するため、水平方向の境界条件がより複雑である。そのため全球海洋モデルにおいてもスペクトル法はほとんど用いられず、格子法モデルが一般的である。上記で挙げたシミュレーションモデル以外にも、土石流モデルや生態系モデル、電場を解く雷モデル等が存在する。また、気象・気候分野は古くから地球観測データ等を用いたデータ科学と密接に連携して研究を進めてきており、変分法やカルマンフィルタに代表されるデータ同化システムを利用することは大前提となる。それぞれのシミュレーションで必要となる時空間離散化やアルゴリズムは多種多様であり、検討するアーキテクチャにおいてこれ

らすべてを網羅可能であるかを検討することは人的・時間的制約のため難しい。そこで本研究では、最も代表的なアルゴリズムに絞った検討を進めた。

4.5.1 全球大気シミュレーションに関する検討

気象・気候シミュレーションを代表するアプリケーションとして、本研究では格子法を用いた大気モデルを選定した。このモデルの流体力学計算部分は、演算アルゴリズムとしては構造化された格子を用いて、隣接する格子の情報を利用する構造格子ステンシル計算に分類される。水平・鉛直方向にデータ依存があり、またシミュレーション精度向上のために高い空間解像度が要求される。大気モデルにおいて非構造格子があまり用いられない理由は、モデル全体が多数の変数を同時に利用するにも関わらず、高解像度化したい空間領域が同時に多数存在し、また時々刻々と変化するためである。このような場合は AMR のような手法を用いたときにロードバランスをとるためのデータ移動が頻発し、均一な高解像度格子を使う場合と比べて大幅な高速化が見込めない。また、構造格子であっても計算領域全体を波数展開するスペクトル法の場合は、多数のプロセスが関わる通信が頻発し、ネットワーク通信律速となる。そのため、メモリ分散での超高並列計算では間接参照を減らし最小限のデータ移動で演算を実現する構造格子ステンシル計算が効率的である。さらに流体計算の数値精度と高い演算密度を実現するガラーキン法も存在するが、これも構造格子ステンシル計算のひとつである。

本研究では、さらに具体的なソフトウェアとして全球非静力大気モデル NICAM(Satoh *et al.* 2014) を選定した。NICAM は水平方向の格子点をプロセス分割する際に、プロセス間のトポロジーを非構造的に、プロセス内の格子点を構造的に扱う「準構造格子系」を採用している。ただし、ステンシル計算を行う場合のアクセスパターンはシンプルなカーテシアン座標系とは異なり、姫野ベンチに代表されるようなごく基本的なステンシル計算とは異なる演算バターンを評価することが可能である。実計算による評価には、フルアプリケーションではなく演算カーネルを用いることとした。NICAM の演算カーネルは、「富岳」開発プロジェクトである FLAGSHIP2020 プロジェクトや、日独仏エクサスケールコンピューティングのためのソフトウェア開発研究事業 SPPEXA の一つである AIMES プロジェクト (Kunkel *et al.* 2020) においてすでに抽出・整備されており、これらを用いたソフトウェアパッケージ「IcoAtmosBenchmark v1 (AIMES project 2018)」を用いた。本パッケージには、6 つの演算カーネル（水平拡散、水平発散、鉛直三重対角行列演算、水平移流フラックス計算、水平移流制限項計算、鉛直移流制限項計算）とステンシル計算に用いるメトリクス計算カーネル、さらにプロセス間通信カーネルが含まれている。

気候・気象分野での科学的なマイルストーンとして、現在各国の研究機関は 2030 年に全球 1 km 程度の水平解像度での気候シミュレーションを、24 時間で 1 年分計算可能（1 Simulation Year Per wall-clock Day）にすることを目指している。本研究においても、同様の目標設定が妥当であると考える。NICAM はスーパーコンピュータ「京」の 1/4 系を用いて、世界で初めて全球 870 m メッシュでの気象計算を実現した。この時の計算速度は 0.0006 SYPD に相当した。現在、「富岳」の 1/4 系を用いて同じ問題サイズは 0.024 SYPD で計算可能であり、およそ 40 倍の高速化を実現している。これを踏まえ、日本の次世代フラッグシップマシンで再び 40 倍の高速化を実現可能であれ

ば、1 SYPD を達成可能である。ただし、「京」から「富岳」への高速化はハードウェアの性能向上だけで実現したものではなく、一部はアプリケーションの改良（特に、混合精度演算の積極的な利用）に起因している。次世代計算基盤においても、ハードウェアの性能向上のみではなく、ソフトウェアの改良を含めて、上記の実現目標を達成するものと想定する。

プログラミングモデルやソフトウェア再実装のコストを度外視して、今回検討するアーキテクチャへの最適化が理想的（少なくとも現行のアプリケーション並み）に完了できると仮定して、その演算性能を引き出すために重要な要素としては 3 点が挙げられる。一つ目は大きな並列度の確保である。演算加速機構は L2, L1, 演算ユニットと階層構造になっており、それらを使い切るためには SIMD で動作する十分に大きな並列度を確保することが必要であり、NICAM ではデータ構造の最内に配置される水平格子数を並列度の指標としてみることができる。二つ目の要素は演算加速機構が主に用いる高速メモリの容量であり、これは一つ目の要素とも密接に関連している。1 時刻ステップをできる限りに高速に計算するという命題において、ホスト側の大容量メモリへのアクセスは無視できないオーバーヘッドとなる。最近の GPU を用いた計算においても、課題は同一であり、GPU 上のメモリで全ての演算を完結させてはじめて、アーキテクチャの性能に見合う計算速度を得ることができる。高速メモリの容量によって、確保可能な並列度は制限を受ける。三つ目の要素は高速メモリの転送速度である。一般的に速いメモリはコストが高く容量が限られており、演算加速器用に大容量の高速メモリを搭載しようとすると、メモリ転送性能が犠牲になる。演算密度が低い気象・気候アプリケーションの演算性能は、ほぼメモリ転送性能によって決定される。そのため、並列度＝メモリサイズとメモリ転送性能のトレードオフの中で、性能要求を考える必要がある。

表 4.5.1 に、NICAM を 870 m メッシュで計算する場合の、分割数と要求されるマシン性能についてまとめる。NICAM は空間領域を水平方向に二次元タイル分割することでプロセス並列を実現しており、分割数によって水平方向のタイルあたり格子数が変動する。表より、想定するメモリ容量に収まるのは 10000 タイル以上に分割したケースであるが、40000 タイル以上に分割すると必要とする並列度の 1/4 程度しか確保できないということがわかった。十分な並列度とメモリ容量制限をクリアする設定では、要求されるメモリ転送性能が極端に高く（「富岳」の CMG あたりメモリ転送性能の 500 倍以上）、現実的ではない。このことからも、ハードウェア性能の向上のみで目標とする計算性能を達成するのは難しいことが明らかになった。ノード間通信性能およびファイル I/O の見積もりについても検討を行った。現行のシミュレーションでは、通信、ファイル I/O ともに総計算時間の 5%程度に抑えるような実験設定で計算を行っており、ストロングスケーリングによって利用するプロセス数を増加させると、それらの過程に要する経過時間の比率は大きくなる。本検討では経過時間を 10%まで許容したが、並列度として最適なプロセス分割数（10240 並列）を選んだ場合には、必要となる通信性能は 60 GB/s、ファイル I/O 速度は 430 MB/s と、それぞれ「富岳」の 2.2 倍、14 倍となる。

表 4.5.1　ターゲット問題のプロセス並列数の違いによる要求性能のまとめ。ただし、次の条件下での値とする。1) 時間刻みあたり必要経過時間は 870 m メッシュ全球シミュレーションを 24 時間で 1 年分計算する (1 SYPD) 際に必要となる計算速度。2) 必要となるメモリ転送速度についての仮定：シミュレーション全体の平均としてピークメモリ転送性能の 10%で実行可能であること。3) 必要となる通信速度についての要件：1 回の時間ステップの中で、4 ノードへのバイセクション通信を 50 回行う経過時間が、総経過時間の 10%に収まること。4) 必要なファイル I/O 速度についての要件：シミュレーション内の 1 時間ごとに、三次元変数 20 個を出力する経過時間が、総経過時間の 10%に収まること。

ケース番号	分割タイル数	タイルあたり水平格子数	袖領域格子数	鉛直層数	時間刻み（力学コア）[sec]	時間刻みあたり必要経過時間 [sec]
Case.1	2560	264196	2052(0.8%)	78	2	0.0055
Case.2	10240	66564	1028(1.5%)	78	2	0.0055
Case.3	40960	16900	516(3.1%)	78	2	0.0055
Case.4	163840	4356	260(6.0%)	78	2	0.0055

ケース番号	タイルあたりメモリ消費量 [GiB]	ファイル I/O 頻度	必要となるメモリ転送速度 [TB/s]	必要となる通信速度 [GB/s]	必要なファイル I/O 速度 [MB/s/die]
Case.1	63.0	every 10 sec	573	119	1708
Case.2	15.9	every 10 sec	143	60	430
Case.3	4.0	every 10 sec	36	30	109
Case.4	1.0	every 10 sec	9	15	28

4.5.2　構造格子ステンシルカーネルのプログラム再実装に関する検討

高い実装密度と電力性能を実現するために、近年の計算機アーキテクチャはドラスティックな変遷をしている。多くの演算ワークロードを担当する部分が CPU から GPU に変わったことはその最たるものである。さらに AI 研究とそのための計算機需要の拡大に後押しされ、計算機のトレンドは GPU の先に進みつつある。本研究において対象としている MN-Core もその一つである。このような計算機アーキテクチャの変更に追従するには、現在のソフトウェアが主に利用している Fortran や C 言語のような世代の言語では困難である。例えば、既存の Fortran で記述されたプログラムを拡張することで GPU に対応しようとした場合、OpenACC や OpenMP のようなディレクティブベースの言語拡張を利用することで、CPU と GPU の両方で動作が可能なソフトウェアを維持することができるが、OpenACC と OpenMP のいずれを用いても、科学技術計算に用いることの可能な NVIDIA 社、AMD 社、Intel 社のすべての GPU で高い性能を発揮することが現時点では出来ない。プログラムの実装にかかる工数をできる限り減らし、広範な種類の CPU、GPU、AI チップに対する性能可搬性を実現するには、新たな言語環境と開発エコシステムに移行することが必要である。しかし、言語環境の移行はプログラムコードの量が百万行を超えることも珍しくない気象・気候アプリケーションにとって簡単なことではない。そこで、プログラム移行の容易さと演算性能の両面において評価を行った。対象としたプログラミング言語は Python と Julia である。Python については、さらに Google JAX ライブラリを利用することを前提とした。これら 2 つの言語を選定した理由は、すでに海外の研究において実装例がある点と、Fortran とコーディングが似ている点にある。なお、独自あるいは既存のドメイン固有言語を用いたり、C++のテンプレート

プログラミングを利用した開発フレームワークを用いるという選択肢については今回検討しなかった。ドメイン固有言語については、将来に渡り開発を維持継続させ、多くの新たなアーキテクチャのサポートを増やそうとした場合、コミュニティを拡大し広く普及させる必要があるが、現状において 10 年後に最も普及しているであろうドメイン固有言語を適切に選び取ることは困難である。C++フレームワークは現時点においても GridTools や Kokkos 等の利用実績があるが、Fortran と C++は言語の設計思想が大きく異なり、日本の気象・気候シミュレーションモデルの開発コミュニティにとっては学習コストが最も大きい。これらの理由より、今回の検討対象からは除外した。

Python (JAX) を用いた再実装では、「IcoAtmosBenchmark v1」に含まれる 6 つの演算カーネルを対象とした。JAX は Google 社が開発した Python 上の機械学習フレームワークである。JAX がもつ主な機能は、1) 自動微分の計算が容易、2) その場 (Just In Time: JIT) コンパイルによる計算高速化、3) vmap によるベクトル化、4) pmap による並列化である。このうち HPC にとって有用なのは JIT コンパイラとしての機能である。NumPy を用いて、普通の Python のソースコードとして記述された任意の数値計算処理を、その場でコンパイルして CPU のみならず、GPU や TPU 等のアクセラレータ上で高速に実行することが可能になり、HPC アプリケーションを書く研究者側の自由度が向上する。MN-Core に関しても、JAX を用いて記述された Python コードを MN-Core 上で動作させるための開発が行われており (PFN Blog 2022)、将来的に JAX は本調査研究で対象とする独自アクセラレータアーキテクチャでサポートされることが期待される。

まず第一段階として、オリジナルの Fortran コードをなるべく模倣した形で、Python への移植を実施した。図 4.5.1 にオリジナルコードの一部を示す。この計算は隣り合う 3 つの六角形の各中心点（分割された正 20 面体格子系の格子点であり変数の配置された地点）から、3 つの六角形が共有する六角形の頂点の一つへ値を内挿する操作となっており、また l と k のループはそれぞれ、分割されたタイルと鉛直層毎に操作を繰り返すことを表している。これに対して、図 4.5.2 に for ループを用いて Python で実装した結果を示す。NumPy を用いることで、多次元配列を Fortran で宣言した形状に揃えたこともあり、両者のソースコードは非常によく似ている。一方で、演算速度については両者は大きく異なる。表 4.5.2 に 6 つの演算カーネルの経過時間の結果をまとめる。計測にあたっては、AMD Ryzen5 5560U を搭載したノート PC を用い、Winodws11 上の WSL2 にて実行環境を構築して計算を行った。表 4.5.2 より、オリジナル版と Python 版 (Ver.1) では、実行時間に 400-3400 倍もの違いが出た。次に、実行時間の短縮を測るために、NumPy 多次元配列の要素毎の計算に用いられる「スライシング」という実装への変更を行った。図 4.5.3 にソースコードの例を示す。スライシングを用いる場合、右辺と左辺の配列の次元数を一致させる必要がある。また、前述の Fortran 版、Python (Ver.1) 版ではループ内でのみ用いられる中間配列について、鉛直層とタイルの次元が無い小さな配列サイズを用いていたが、スライシングを用いる場合は最終的なターゲットとなる配列の次元・サイズに合わせる必要がある。これにより、演算の並列性を確保することが可能になるが、一方で一時的に消費するメモリサイズが増えるという問題がある。表 4.5.3 にスライシングで記述した Python (Ver.2) 版の経過時間を示す。Ver.2 でのコードは Fortran に近いアルゴリズムの記述スタイルをやめることによって、dyn_diffusion で Ver.1 から 58 倍程度の高速化を実現した。さらに、Ver.2 を元に JAX を用いた実装を行った。JAX での実装において

も、単純に for ループで記述した Python コードに JIT コンパイルを指定しただけでは、ほとんど高速化は得られず、以下のようにスライシングを行う、

$$at[i:n].set(moderator) \tag{4.5.1}$$

という記述方法をとることで、CPU では NumPy と同程度の速度を達成した（表 4.5.3)。図 4.5.3(下) に示したソースコードは、JAX で記述した際の例であり、NumPy のスライシング（図 4.5.3(上)）と比較した場合、ほとんど同一のコードとなる。。表 4.5.3 には GPU を用いて計算した結果も示している。このとき、GPU には NVIDIA A100 を用いた。倍精度演算性能としては本実験での CPU 計算環境と比較して、50 倍弱の違いがあるが、経過時間としてはあまり CPU と変わらない結果となった。これは、問題サイズがあまり大きくないことに起因する。演算性能の向上は今後の課題であるが、GPU の利用に対してプログラムを一切書き換える必要がない点は非常に高く評価できる。

表 4.5.2　IcoAtmosBenchmark v1 の実行時間比較。

カーネル名	説明	経過時間 (Fortran) [sec]	経過時間 (Python v1) [sec]
dyn_diffusion	3 次元データに対して水平 2 次元方向に格子点を参照するステンシル計算	0.010	20.2
dyn_divdamp	3 次元データに対して 3 次元方向に格子点を参照するステンシル計算	0.014	11.1
dyn_vi_rhow_solver	3 次元データに対して鉛直 1 次元方向に 3 重対角行列を解くステンシル計算	0.008	3.3
dyn_horiz_adv_flux	3 次元データに対して水平 2 次元方向のフラックス項を計算	0.018	13.5
dyn_horiz_adv_limiter	3 次元データに対して水平 2 次元方向のフラックス制限項を計算	0.037	22.8
dyn_vert_adv_limiter	3 次元データに対して鉛直 1 次元方向のフラックス制限項を計算	0.003	10.3

表 4.5.3　IcoAtmosBenchmark v1 の実行時間比較（その 2)。

カーネル名	(Fortran) [sec]	(Python v1) [sec]	(v2) [sec]	(v3,CPU) [sec]	(v3,GPU) [sec]
dyn_diffusion	0.010	20.2	0.35	0.95	0.01

以上をふまえ、Fortran から Python (JAX) へのプログラム記述言語を変更した場合の利点・問題点について以下にまとめる。

- JAX を用いた場合、ユーザープログラミングの視点からはほぼ GPU の有無を意識することなくプログラムを記述することが可能であり、また NumPy を用いた記述に習熟している場合は NumPy → JAX への移行は学習コストが極めて小さく済む。
- Fortran で記述した場合、do ループによる構造化によって配列の演算を行うが、Python (NumPy, JAX) で記述した場合にはスライシングによる配列の演算を採用することが必須であ

```
Fortran Ver. (抜粋)

do l = 1, lall
   do k = 1, kall
      do d = 1, nxyz
         do g = gminm1, gmax
            ij     = g
            ip1j   = g + 1
            ip1jp1 = g + iall + 1
            ijp1   = g + iall

            vt(g,d,TI) = ( ( + 2.0 * coef_intp(g,1,d,TI,l) &
                             - 1.0 * coef_intp(g,2,d,TI,l) &
                             - 1.0 * coef_intp(g,3,d,TI,l) ) * scl(ij    ,k,l) &
                         + ( - 1.0 * coef_intp(g,1,d,TI,l) &
                             + 2.0 * coef_intp(g,2,d,TI,l) &
                             - 1.0 * coef_intp(g,3,d,TI,l) ) * scl(ip1j  ,k,l) &
                         + ( - 1.0 * coef_intp(g,1,d,TI,l) &
                             - 1.0 * coef_intp(g,2,d,TI,l) &
                             + 2.0 * coef_intp(g,3,d,TI,l) ) * scl(ip1jp1,k,l) &
                         ) / 3.0
         enddo
      enddo
   enddo
enddo
```

図 4.5.1　演算カーネル（水平拡散）のソースコードの一部。

る。これはアルゴリズムの記述方法としては大きな方針転換であり、機械的な書き換え作業だけでは完了できない。

- 演算性能については、問題サイズが小さい場合は Fortran よりも未だ低速であるが、適切なスケーリング性能評価を進めることによって、十分に許容可能な演算性能を得ることが期待される。

Julia を用いた再実装では、「IcoAtmosBenchmark v1」に含まれる 6 つの演算カーネルのうち、水平拡散と鉛直三重対角行列演算の 2 つを対象とした。Julia を利用する利点は複数あり、Python とその利用コミュニティが発展させている機械学習ライブラリや可視化ライブラリとの親和性が高いこと、アーキテクチャ・低レベル言語環境の異なる多種の GPU への対応を進めており、将来のプログラム移植が容易であることが期待されること、既存の気象・気候アプリケーションが過度に依存している Fortran と言語仕様が似ている（配列添字の開始が 1 である、多次元配列のメモリ格納順が Column major order である）こと等が挙げられる。MN-Core は Julia をサポートしていないが、集中的な開発資源の投入を行えばサポート実現のための障壁はあまり高くないと期待する。

Python 実装の際と同様の方針で、まずはオリジナルの Fortran コードをなるべく忠実に再現した形で、Julia への移植を実施した。図 4.5.4 に同じ区間を Julia で記述した際のコードを示す。両者は非常に似た記述を行うことが可能であり、ループを do 構文と for 構文で記述するかや、継続行の扱い、配列の添え字を書く際の括弧の種類、程度しか違いが存在しない。Python への移植では、Google JAX を用いた高速化を達成するために「スライシング」記法に書き換える必要があった他、配列添え字やループの開始インデックスを注意深くずらす必要があった。Julia の場合は後述する最適化を適用した後も、for 文を用いたループでの記述を変更する必要がなく、「スライシング」に相当する Julia の「ブロードキャスト」演算を有効にするための記述変更は必ずしも高速化する

```
Python Ver.1 (抜粋)

for l in range(0, lall):
    for k in range(0, kall):
        for d in range(0, nxyz):
            for g in range(gminm1-1, gmax):
                ij     = g
                ip1j   = g + 1
                ip1jp1 = g + iall + 1
                ijp1   = g + iall

                vt[g,d,TI] = ( (   2.0 * coef_intp[g,0,d,TI,l]
                                   - 1.0 * coef_intp[g,1,d,TI,l]
                                   - 1.0 * coef_intp[g,2,d,TI,l] ) * scl[ij    ,k,l]
                             + ( - 1.0 * coef_intp[g,0,d,TI,l]
                                   + 2.0 * coef_intp[g,1,d,TI,l]
                                   - 1.0 * coef_intp[g,2,d,TI,l] ) * scl[ip1j  ,k,l]
                             + ( - 1.0 * coef_intp[g,0,d,TI,l]
                                   - 1.0 * coef_intp[g,1,d,TI,l]
                                   + 2.0 * coef_intp[g,2,d,TI,l] ) * scl[ip1jp1,k,l]
                             ) / 3.0
```

図 4.5.2　図 4.5.1 に同じ。ただし Python を用いて、Fortran を模倣して記述されたもの。

わけではなかった。これら As Is コードの経過時間について表 4.5.4 に示す。計測にあたっては、Intel Core i9-12900K を搭載したワークステーションを用い、Ubuntu 22.04 上に実行環境を構築して計算を行った。言語環境はそれぞれ GNU Fortran 11.4.0、Python3.12.8、Julia 1.10.8 を用い、スレッド並列化は適用していない。As Is コードでは Python と Julia の計算時間に大きな違いはなく、いずれも Fortran と比較して 100〜1000 倍低速であった。

表 4.5.4　IcoAtmosBenchmark v1 の実行時間比較（その 3)。

カーネル名	経過時間 (Fortran) [sec]	経過時間 (Python) [sec]	経過時間 (Julia) [sec]
dyn_diffusion	0.007	12.2	12.2
dyn_diffusion	0.006	1.89	0.93

次に、カーネルのうち dyn_diffusion に対して、実行時間を短縮するための高速化手法を適用した。検討の結果、最も劇的な改善がみられたのは配列サイズを決めている変数について const を用いて宣言することであった。配列サイズを格納する変数は全プログラムファイルで共有することが多く、そのため global 変数の扱いとなる。global 変数の値と型はいつでも変更される可能性があるため、コンパイラの最適化を著しく阻害する。それらの変数の const 宣言により、メモリアロケーションの回数とメモリ使用量がそれぞれ 100,000,000 分の 1、10,000,000 分の 1 に減少し、As Is コードの 800 倍程度まで高速化した。実際にフルアプリケーションにおいて、該当する配列サイズの const 宣言を行うことができるかについては、実行時の前処理を工夫することによって可能である。すなわち、Fortran の場合コンパイル済みの実行バイナリを事前に準備し、実行時にパラメータファイルを読み込ませて配列サイズを確定させているが、Julia の場合は事前コンパイルが必要ではないため、実験設定に合わせて必要な配列サイズを記述したプログラムファイルを生成し

```
Python Ver.2 (抜粋)

vt[gminm1-1:gmax,0:kall,0:nxyz,0] =
            ( (   2.0 * coef_intp[gminm1-1:gmax,None,0,0:nxyz,TI,0]
                - 1.0 * coef_intp[gminm1-1:gmax,None,1,0:nxyz,TI,0]
                - 1.0 * coef_intp[gminm1-1:gmax,None,2,0:nxyz,TI,0] ) * scl[gminm1-1:gmax          ,0:kall,None,0]
            + ( - 1.0 * coef_intp[gminm1-1:gmax,None,0,0:nxyz,TI,0]
                + 2.0 * coef_intp[gminm1-1:gmax,None,1,0:nxyz,TI,0]
                - 1.0 * coef_intp[gminm1-1:gmax,None,2,0:nxyz,TI,0] ) * scl[gminm1:gmax+1          ,0:kall,None,0]
            + ( - 1.0 * coef_intp[gminm1-1:gmax,None,0,0:nxyz,TI,0]
                - 1.0 * coef_intp[gminm1-1:gmax,None,1,0:nxyz,TI,0]
                + 2.0 * coef_intp[gminm1-1:gmax,None,2,0:nxyz,TI,0] ) * scl[gminm1+iall:gmax+iall+1,0:kall,None,0]
            ) / 3.0
```

```
Python Ver.3 (抜粋)

vt = vt.at[gminm1-1:gmax,0:kall,0:nxyz,0].set(
            ( (   2.0 * coef_intp[gminm1-1:gmax,None,0,0:nxyz,TI,0]
                - 1.0 * coef_intp[gminm1-1:gmax,None,1,0:nxyz,TI,0]
                - 1.0 * coef_intp[gminm1-1:gmax,None,2,0:nxyz,TI,0] ) * scl[gminm1-1:gmax          ,0:kall,None,0]
            + ( - 1.0 * coef_intp[gminm1-1:gmax,None,0,0:nxyz,TI,0]
                + 2.0 * coef_intp[gminm1-1:gmax,None,1,0:nxyz,TI,0]
                - 1.0 * coef_intp[gminm1-1:gmax,None,2,0:nxyz,TI,0] ) * scl[gminm1:gmax+1          ,0:kall,None,0]
            + ( - 1.0 * coef_intp[gminm1-1:gmax,None,0,0:nxyz,TI,0]
                - 1.0 * coef_intp[gminm1-1:gmax,None,1,0:nxyz,TI,0]
                + 2.0 * coef_intp[gminm1-1:gmax,None,2,0:nxyz,TI,0] ) * scl[gminm1+iall:gmax+iall+1,0:kall,None,0]
            ) / 3.0
)
```

図 4.5.3　図 4.5.1 に同じ。ただし Python を用いて、NumPy のスライシング手法を用いて演算を記述したもの（上）と、JAX を用いて演算を記述したもの（下）。。

実行することで、実験毎に配列サイズを変更しつつ、const 宣言を適用することによる高速化を達成可能である。さらに、配列添字の範囲検査を省略する@inbounds マクロを適用し、オーバヘッドを削減するとともに、固定長配列のためのパッケージ StaticArrays.jl を利用し、一部のサイズの小さい mutable な一次元ワーク配列に対して MVector 型での宣言を適用することで静的かつ高速な固定長配列に変更した。加えて、ブロードキャスト演算についても評価を実施した。ほとんどの区間では@. マクロを最内ループ内の各演算行の先頭に適用し、元のアルゴリズム記述を変えずにブロードキャスト演算への変換を行うとともに、部分配列のアクセスがある場合は@view マクロを併用して配列のアクセス範囲を適用した。これらの高速化の適用結果について表 4.5.5 に示す。配列サイズの const 宣言で Fortran の実行速度に肉薄するところまで改善し、さらに@inbounds マクロの適用によって Fortran と同等の速度となった。StaticArrays の適用によって、Fortran よりも高速化するケースがあることも示された。一方で、ブロードキャスト演算については for ループのまま計算する場合よりもメモリアロケーションやメモリ使用量が増加し、結果としてプログラム修正がかなり必要であるにも関わらず、実行時間がある程度以上高速化しなかった。これは部分配列アクセスなどが存在する場合に、都度ビューを設定する必要があり、メモリアロケーション回数を削減しずらいことに起因すると考えられる。

ここまで CPU 上での Julia への移植は非常に開発コストが低く高性能であることが示されたが、GPU での最適化を進める場合、CPU での最適化とは全く異なる戦略を取る必要が明らかになった。Julia の GPU 最適化では、配列プログラミングとカーネルプログラミングのいずれかを用いる。前者の場合は、用いる配列をすべて GPU デバイス用の配列（仮に GPU 用配列と呼ぶ）に置き換え、実行する。GPU 用配列はループを用いた演算には向いておらず、基本的にブロードキャスト演算が要求される。そのため、Python (JAX) を用いた場合と同じく、かなり多くのカーネルを書き換える必要が発生し、ループで記述していた Fortran コードをそのまま移植し性能を発揮できるという Julia の利点は発揮できなくなる。後者のカーネルプログラミングを採用した場合は、

```
Julia Ver. (抜粋)

@inbounds begin
for l = 1:ADM_lall
   for k = 1:ADM_kall
      for d = 1:ADM_nxyz
         for g = gminm1:gmax
            ij      = g
            ip1j    = g + 1
            ip1jp1 = g + iall + 1
            ijp1    = g + iall

            vt[g,d,TI] = ( ( + 2.0 * coef_intp[g,1,d,TI,l]
                             - 1.0 * coef_intp[g,2,d,TI,l]
                             - 1.0 * coef_intp[g,3,d,TI,l] ) * scl[ij    ,k,l]
                         + ( - 1.0 * coef_intp[g,1,d,TI,l]
                             + 2.0 * coef_intp[g,2,d,TI,l]
                             - 1.0 * coef_intp[g,3,d,TI,l] ) * scl[ip1j  ,k,l]
                         + ( - 1.0 * coef_intp[g,1,d,TI,l]
                             - 1.0 * coef_intp[g,2,d,TI,l]
                             + 2.0 * coef_intp[g,3,d,TI,l] ) * scl[ip1jp1,k,l]
                         ) / 3.0
         end
      end
   end
end
end # @inbounds
```

図 4.5.4　図 4.5.1 に同じ。ただし Julia を用いて演算を記述したもの。。

カーネルが GPU 用コードに事前コンパイルされるため、ループでの記述が許容される。しかし、そのままの関数をまるまるカーネル化しようとしても計算結果が一致しないケースが多く、カーネル内での中間配列準備や、並列化したくない演算、非同期で実行できない演算等を制御するために、一つの大きなカーネルをいくつもの小さなカーネルに細分化しなければいけない。これまでの OpenACC の言語仕様の進展で見られたように、Julia での GPU カーネルプログラミングにおいても、これらの挙動のコントロールについて仕様や機能が今後追加されることを期待する。dyn_diffusion を用いて配列プログラミングとカーネルプログラミングの書き方で GPU 最適化を適用した結果、経過時間は Apple M1 Ultra の GPU (48 コア) を用いてそれぞれ 28 秒、9 秒であった。これは CPU を用いた演算と比べて圧倒的に低速であり、さらなる検討が必要である。

以上をふまえ、Fortran から Julia へのプログラム記述言語を変更した場合の利点・問題点について以下にまとめる。

- Julia を用いた場合、少なくとも CPU 上での実行では言語の学習コストや移行先の言語への特別な配慮（インデックス番号の扱いやメモリ配置順の違い）をほとんど意識することなく移植が可能であり、またループを基本とした記述スタイルを変えずに Fortran と遜色ない速度で計算を行うことが可能である。
- Julia での計算高速化において型や値、配列サイズを事前に確定しておくことは非常に重要である。実行時に配列サイズが変わらない、型が変わらない、immutable である、配列外参照の検査が必要ない、等の追加情報を適切に付与することで速度とメモリ使用量の最適化が促進される。
- GPU の利用においては、CPU 最適化時に有効だったループでの演算をやめ、ブロードキャス

表 4.5.5　dyn_diffusion カーネルの高速化適用結果。

	経過時間	メモリアロケーション	
	[sec]	回数	サイズ
Julia As Is コード	12.2	405M	8.3 GiB
ケース 1			
配列サイズの const 宣言	0.015	4	793 KiB
＋@inbounds マクロの適用	0.0072	4	793 KiB
＋ StaticArrays の適用	0.0039	3	792 KiB
ケース 2			
配列サイズの const 宣言+ブロードキャスト演算	0.21	11K	772 MiB
＋@inbounds マクロの適用	0.22	11K	772 MiB

ト演算に切り替える必要があるが、なるべくコードの書き換えコストをかけずに高速化することが現時点では難しく、例外処理等について細かい配慮を必要とする。

4.6　ものづくりアプリケーションの検討

塩谷 隆二 [**学校法人 東洋大学**]
河合 浩志 [**学校法人 東洋大学**]
荻野 正雄 [**学校法人 大同大学**]

次世代計算基盤がものづくりアプリケーションに求める演算数やメモリ容量を見積もることを目的とし、汎用 CPU での実行プロファイルとアクセラレータで実行する部分の性能評価に基づき、ものづくり分野のアプリケーション性能評価を実施した。この調査には、オープンソース CAE ソフトウェア ADVENTURE[Ogino *et al.* 2005] のベンチマークコードが用いた。

数値シミュレーション全体の性能向上には、並列効率だけでなく、計算ノードあたりの性能がこれまで以上に重要になっている。本研究では、現在のマルチコアアーキテクチャ上で領域分割法（DDM）に基づく有限要素コードの性能を最適化するためのアプローチを採用した。

4.6.1　非構造格子有限要素解析ソフトウェア ADVENTURE の解析と最適化

非構造格子有限要素解析ソフトウェアである ADVENTURE は、領域分割法を用いた並列計算を特徴とし、C 言語、MPI、OpenMP で開発されている。計算カーネルである部分領域ソルバーの実装においては、メニーコアおよび GPU、アクセラレータへの有効な実装となりうる **Local Schur Complement (LSC) アプローチ**に着目し、スカラー型計算機向けのミニアプリを開発した。LSC アプローチでは、Schur 補行列が明示的に計算される。このアプローチにより、部分領域の剛性行列を LSC 行列として保持することで、従来の反復ごとの行列分解を LSC 行列を用いた対称密行列

ベクトル積演算に置き換え、大幅な計算時間の短縮を実現した。提案手法は、既存の手法と比較してメモリ使用効率が優れているだけでなく、計算速度も速いことを性能予測モデルと Intel アーキテクチャでの数値例により実証している (Kawai *et al.* 2016)。

ADVENTURE の利用促進と MN-Core 等の次世代計算基盤への適応を目指し、ファイル入出力や有限要素解析部分をライブラリ化した **Python アプリケーション**を開発した。これにより、非構造格子由来の疎行列を係数とする連立一次方程式を用いた MN-Core の性能評価や最適化が可能となり、既存コードの再利用による効率的な開発が実現した。また、領域分割法の並列求解部分の性能評価を行う **C+OpenMP アプリケーション**を開発し、ADVENTURE の MN-Core への直接的な移植時の性能評価を可能にした。

DDM は、非重複 DDM と重複 DDM に大別され、本研究で採用しているのは非重複 DDM である。非重複 DDM における Schur 補行列は、直接法と反復法、およびストレージベースとストレージフリーに分類される。LSC アプローチは、直接ストレージ（DS）型の DDM に分類される。

LSC アプローチでは、部分領域の剛性行列 K_i は、内点部分 u_{Ii} と境界点部分 u_{Bi} に分割して表現される。

$$u_i = \begin{pmatrix} u_{Ii} \\ u_{Bi} \end{pmatrix} \tag{4.6.1}$$

これに伴い、K_i は以下のように分解される。

$$K_i = \begin{pmatrix} K_{IIi} & K_{IBi} \\ K_{IBi}^T & K_{BBi} \end{pmatrix} \tag{4.6.2}$$

全体的な Schur 補行列 S は、各部分領域の Schur 補行列 S_i の総和として表される。

$$S = \sum_{i=1}^{N} {R_{B_i}}^T S_i R_{B_i} \tag{4.6.3}$$

ここで、

$$S_i = K_{BB_i} - {K_{IB_i}}^T {K_{II_i}}^{-1} K_{IB_i} \tag{4.6.4}$$

反復計算における行列ベクトル積 $y = Sp$ は、各部分領域における

$$y_i = S_i p_i \tag{4.6.5}$$

の計算と、その後の集約

$$y = \sum_{i=1}^{N} {R_{B_i}}^T y_i \tag{4.6.6}$$

によって行われる。

LSC アプローチによる DS-LSC 実装と従来の DS-Sky 実装を比較すると、準備段階では DS-Sky が因数分解を行うのに対し、DS-LSC は LSC の計算を行う。DDM 反復段階では、DS-Sky が前方/後方代入を行うのに対し、DS-LSC は行列ベクトル積を行う。

LSC アプローチの性能を評価するために、プレートモデルを用いた数値実験を行った。このモデルでは、各サブドメインの自由度（DOF）が増加すると、DS-Sky の計算時間は N_{dof}^2 に比例し、T_{ds-sky}^{prep} は n^7 に比例する。一方、DS-LSC の計算時間は N_{dof} に比例し、T_{ds-lsc}^{prep} も n^7 に比例するが、その係数は DS-Sky よりも小さくなる。これにより、大規模な問題において LSC アプローチが優位に立つことが示唆される。

LSC アプローチが従来の直接ソルバーにおける前方/後方代入に大きく依存する既存アプローチよりも、メモリ使用効率が高く、高速であることを裏付けている。LSC アプローチの B/F 値（Byte/FLOPS）は、DGEMM や DSYMV といった BLAS レベル 3 およびレベル 2 の関数を活用することで向上させることができる。これにより、メモリ帯域幅がボトルネックとなる環境においても高い性能を引き出すことが可能となる。

4.6.2 PyTorch を利用したソルバーモデルのグラフ化と MN-Core 評価

MN-Core 評価では、既存のコード資産を有効活用しつつ、低コストでの新規コード開発を方針とした。具体的には、ADVENTURE をライブラリ化し、PyTorch でアプリケーションを開発し、ソルバー部分を MN-Core で実行した。ADVENTURE のソースコードは C 言語で約 5 万行で構成されているが、線型方程式のソルバーモデルの行列ベクトル積部分のみを MN-Core で実行するために、PyTorch を利用して計算グラフ化を行った。PyTorch のソースコードは数十行での実装となった。ソルバーモデルのグラフを図 4.6.1 に示す。小規模問題を用いた Xeon CPU プロセッサと MN-Core での性能評価結果を表 4.6.1 に示す。

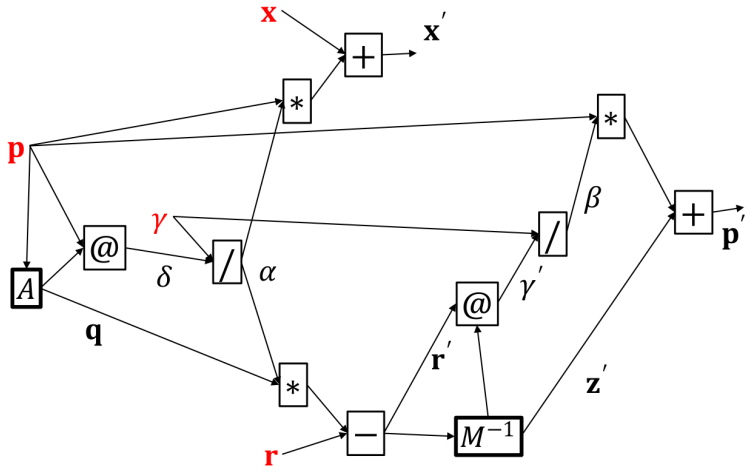

図 4.6.1 Linear Equation Solver Models.

4.6.3 新たなターゲットアプリケーション

ADVENTURE のベンチマークコードを用いた評価から、次世代計算基盤に対する要求が明らかになった。大規模問題においては、ノードあたり初期計算時に 10^{12} オーダー、反復計算時に 10^9

表 4.6.1　Performance evaluation of MN-Core using unstructured grid and small-scale problems [sec].

	Xeon Platinum (8260M, 8 threads)	**MN-Core (offloading)**
setup data (ADV)	0.2421	
solve equation (one iteration)	21.502 (0.005244)	10.124 (0.002469)
in total	21.739	12.498

オーダーの演算数、そして行列計算用に数百メガバイトのメモリ容量が必要となることが判明した。

これまで主に、応力解析ソフトウェア ADVENTURE_Solid を中心に性能評価を行ってきたが、新たなターゲットアプリケーションとして、高周波電磁界解析ソフトウェア ADVENTURE_FullWave に着目し、その GPU への移植と性能評価を実施した。ADVENTURE_FullWave による 160 億有限要素数の人体内部の超大規模解析の可視化結果を図 4.6.2 に示す。情報通信機器や医療機器の電磁環境適合性対応に不可欠な高周波電磁界解析は、数十から数百億要素という大規模モデルを扱う必要があり、その計算負荷は極めて高い。ADVENTURE_FullWave の GPU 等アクセラレータへの移植は、このような大規模計算を効率的に処理するために、アクセラレータが不可欠であることを示唆している。

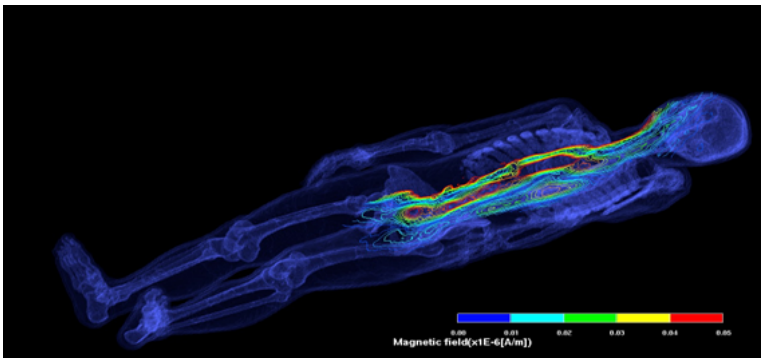

図 4.6.2　High-frequency electromagnetic field analysis inside the human body (frequency: 300[MHz], number of elements: 16 billion).

4.6.4　まとめと今後の展望

本研究では、領域分割法における Local Schur Complement (LSC) アプローチの性能評価を実施した。LSC アプローチは、部分領域の剛性行列を LSC 行列として保持することで、従来の反復ごとの行列分解を LSC 行列を用いた対称密行列ベクトル積演算に置き換え、大幅な計算時間の短縮を実現した。このアプローチは、メモリ使用効率が高く、既存手法よりも高速であることが示された。

ADVENTURE の利用促進と MN-Core 等の次世代計算基盤への適応を目指し、ファイル入出力や有限要素解析部分をライブラリ化した Python アプリケーションおよび C+OpenMP アプリケー

ションを開発した。これらの成果は、高性能計算環境におけるものづくりアプリケーションの高度化に貢献するものである。今後のより大規模かつ複雑なシミュレーションの実現には、GPU をはじめとする高性能アクセラレータの活用が鍵となる。MN-Core 等のアクセラレータを含む多様な計算資源を最適に活用するための技術開発が、今後の研究開発において重要であることが確認された。

4.7 マテリアルサイエンス応用アプリケーションの検討

押山 淳 [国立大学法人 東海国立大学機構 名古屋大学]
岩田 潤一 [株式会社 Quemix]

マテリアルサイエンス分野における主要アプリケーションである第一原理電子状態計算を実行する RSDFT (Real-Space Density-Functional Theory) コードおよび RS-CPMD (Real-Space Car-Parrinello Molecular Dynamics) コードの、次世代アーキテクチャ上での性能を定量的に予測し、性能向上の可能性を調査する。具体的にはアプリケーションのカーネル部分を抽出し、まずは現在のアーキテクチャ上での性能およびボトルネックを明らかにする。それを踏まえ、今後の高速化および多機能化の指針を示す。

4.7.1 コード調査

「富岳」コンピュータ上での RSDFT コードの実行により、その重要なカーネルは、部分空間反復法 (Rayleigh-Ritz 法) による外部固有値問題の求解部分であり、特に問題サイズが大きい場合 (10 万原子規模) は、Gram-Schmidt 直交化および部分空間対角化がボトルネックとなり、いずれもターゲット物質のサイズ N の 3 乗に比例する計算コストを要することが明らかになっている。ただしボトルネック部は行列積演算が主となるため、計算全体でも 40〜50%の高い実行効率を達成することが可能である (10,000 ノード使用時)。一方、小〜中規模サイズ (1 万原子以下) では差分行列および非局所擬ポテンシャルと呼ばれるランク 1 の行列の演算部分が、N^3 のボトルネック部に比べ無視できない計算時間を要するようになり、それに伴って実行効率も下がる。RSDFT の発展形であり、マテリアルサイエンスにおいて重要な Molecular Dynamics (MD) 計算を DFT の枠内で実行することも可能となっている (コード名:RS-CPMD, Real-Space Car-Parrinello Molecular Dynamics)。この RS-CPMD コードでの主要カーネルは、Gram-Schmidt 直交化に対応するラグランジュ乗数部分の計算であり、システムサイズの 3 乗に比例する計算コストとなる。また一般的に MD は時間方向になるべく長くシミュレーションを行う必要があり、1 ステップに要する計算時間を抑えるために系のサイズを大きく取れない。そのために、高効率で実行可能な N^3 のボトルネック部の計算と、低効率の差分および非局所ポテンシャルの行列演算の計算時間が同程度となり、全体的な実行効率も下がる傾向にある。ノード間通信は、N^3 ボトルネック演算部分で必要となるリダクション (MPI_Allreduce) と、差分および非局所擬ポテンシャル行列演算に伴う隣接通信 (MPI_Isend, MPI_Irecv) が主となる。並列化は空間および固有ベクトルの軸について行うことが

可能で、大規模系では両者の軸についての並列化を組み合わせた場合、計算時間全体の 30%ほどが通信時間となる。小〜中規模系では隣接通信のコストが主となるが、演算部分と合わせ、この部分のコスト削減が次世代機で実現できれば大幅な性能向上が見込めると思われる。I/O については、計算の最後に、大きくても使用メモリサイズ相当のデータを書き出すのみで、現在のターゲットサイズ、10,000–100,000 原子規模計算では、それほどの重大な問題は引き起こさないと思われる。

4.7.2 GPU を導入した場合のベンチマークおよび「富岳」との比較

GPU に対応した RSDFT コードを用いて、グリッド数 4,374,000 点、固有ベクトル本数 2592 本 (1000 原子規模) の系において「富岳」との比較を行った。「富岳」では 108 ノード (12MPI /ノードで実行) を用いた場合、反復計算 1 ステップの実行時間は 15.7 秒であった。一方、NVIDIA DGX A100 を用いて GPU を 8 台搭載したノード 1 つで同じ計算を実行したところ、実行時間は 18.6 秒であった。主要カーネルごとの計算時間の内訳を、表 4.7.1 および表 4.7.2 に示す。

表 4.7.1 「富岳」、108 ノード (12 MPI/ノード) での実測値（単位：秒）

	SCF (1 step)	SD	GS	CG	Others
Fugaku	15.7	4.50	0.97	7.33	2.90

表 4.7.2 A100×8、１ノードの GPU マシンでの実測値（単位：秒）

	SCF (1 step)	SD	GS	CG	Others
DGX A100	18.6	4.25	3.07	9.56	1.72

4.7.3 GPU による性能向上の可能性について

RSDFT には元々、差分および非局所擬ポテンシャルに関する通信を束ねて通信回数を削減する機能が実装されており、この機能がほぼそのまま GPU オフロードの効率化に利用できる。その結果、「富岳」であまり効率の出ていない部分が加速され、「富岳」100 ノード相当の計算時間が GPU 機 1 ノードで達成できるという結果が得られた。

4.7.4 RSDFT のマルチノード環境での性能の机上評価

RSDFT は、密度汎関数法（DFT）に基づく第一原理計算により様々な物質のシミュレーションを行うためのコードで、実空間差分法により DFT の基礎方程式である Kohn-Sham 方程式を解く実装となっている。数値的には、空間をメッシュに切って得られる行列に対する固有値問題、固有値の小さいものから順に、系に含まれる電子の個数に相当する本数の固有ベクトル求めることになる。RSDFT では、その求解に部分空間反復法（Rayleig-Ritz 反復法）を採用している。主要な計算は、部分空間対角化（SD)、Gram-Schmidt 直交化（GS)、および共役勾配法（CG）からなり、SD および GS の演算量はシステムサイズ（＝原子数）の 3 乗に比例し、CG はシステムサイズの 2 乗に比例するため、計算全体の演算量もシステムサイズの 3 乗に比例することになる。

表 4.7.2 のデータを元にして、ポスト「富岳」提案環境での性能評価を行う。評価を行う問題サイズ（グリッド点数、バンド本数（∝電子数））および問題を乗せるのに必要なノード数を、表 4.7.2 の系も併せて、表 4.7.3 に示す。

表 4.7.3　性能評価を行う問題サイズ

原子数	バンド本数	グリッド点数	ノード数	メモリ/ノード (GB)	差分袖データ（一方向）（MB）
1000	2592	4374000	1	170	-
8000	20736	35251200	8	340	0.547
16000	41472	70502400	128	340	0.362

評価の方法であるが、演算時間については、表 4.7.2 の結果を元に、ノード当たりのグリッド点数とバンド本数から、システムサイズの 3 乗のスケーリングを用いてノード当たりの演算量を算出し、A100 とポスト「富岳」の性能比を鑑みて評価する。複数ノードを使用する計算では、これらにさらにノード間通信に要する時間を上乗せする。各ルーチンでの主要な通信は、SD は、ベクトルの内積から行列要素を計算するために必要な Allreduce、および差分の行列を作用させるために必要な差分の袖の隣接通信である。GS は、SD と同様な内積計算が必要で、そのために Allreduce が必要となる。SD および GS の Allreduce はバンド本数の 2 乗個の要素をまとめて一回で行うとする。CG はベクトルの本数 ×8 回の内積計算の Allreduce、および差分の隣接通信となる。CG の内積の Allreduce はベクトル複数本分をまとめて処理することができるため、ここでは表 4.7.2 計測を行ったときと同じ 128 本分をまとめて処理することとした。また Allreduce についてはのバタフライアルゴリズムを想定した。さらに Allreduce および差分の隣接通信ともにレイテンシは無視した。

表 4.7.4 に評価結果をまとめる。なお、表 4.7.2 にあった Others の項目は、最終的な物理量の計算の時間であり、他の主要ルーチンに比して寄与が小さいため、今回の見積からは除外した。

表 4.7.4　ポスト「富岳」での計算時間評価（単位：秒）。カッコ内は通信時間。

原子数	SCF (1 step)	SD	GS	CG
1000	5.27	1.33	0.96	2.98
8000	170 (0.84)	85.1 (0.28)	61.4 (0.02)	23.7 (0.53)
16000	79.2 (1.11)	42.6 (0.38)	30.7 (0.03)	5.92 (0.70)

現在、第一原理計算は、数百原子以下の系においても、SCF 反復計算の 1 ステップに数十秒から数百秒を要することはざらにあることを考えると、かなり少ないノード数で 10000 原子規模の系を、現在と同程度の計算時間で扱えるようになることは、かなり複雑な原子モデルで、リアルな材料のシミュレーションが実行可能になるという有望な結果といえる。

4.8　素粒子・原子核物理応用アプリケーションの検討

石川 健一 [国立大学法人 広島大学]

素粒子・原子核物理における大規模計算を必要とする応用アプリケーションとして格子量子色力学 (格子 QCD) 向けのアプリケーションの MN-Core アーキテクチャ上での性能を机上評価した。

4.8.1　格子量子色力学と数値計算手法

量子色力学 (QCD) は原子核を構成する陽子や中性子などの性質をクォークとグルーオンという素粒子の相互作用で説明する理論である。格子 QCD は QCD を計算機で第一原理的に計算することができるようにした理論である。ここでは格子 QCD に用いる数値計算手法を概観し、その数値計算における主要部分であるクォークソルバーについて説明する。

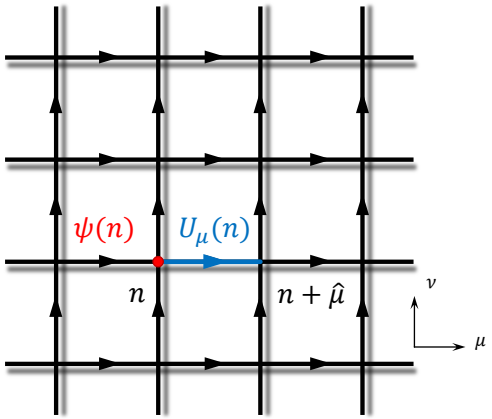

図 4.8.1　格子 QCD の変数の 4 次元格子上の 2 次元平面での配置の模式図

格子 QCD では 4 次元時空を離散化（格子化）し格子点や格子点をつなぐ辺の上にクォークやグルーオンの自由度を配置する。さらに量子力学的にこれらの自由度を取り扱うためにファインマン経路積分法を用いる。素粒子実験にて観測される物理量は格子 QCD では、ファインマン経路積分法を通じたクォークやグルーオンの自由度を積分変数とする以下のような重み付き多重積分で表される。

$$
\begin{aligned}
\langle O \rangle &= \frac{1}{Z} \prod_{n,\mu} \int \mathrm{d}U_\mu(n) \prod_n \int \mathrm{d}\bar{\psi}(n)\, \mathrm{d}\psi(n) O[U,\bar{\psi},\psi] e^{-S_{\mathrm{LQCD}}[U]}, \\
Z &= \prod_{n,\mu} \int \mathrm{d}U_\mu(n) \prod_n \int \mathrm{d}\bar{\psi}(n)\, \mathrm{d}\psi(n) e^{-S_{\mathrm{LQCD}}[U]}, \\
S_{\mathrm{LQCD}}[U] &= S_{\mathrm{gluon}}[U] + S_{\mathrm{quark}}[U,\bar{\psi},\psi], \\
S_{\mathrm{quark}}[U,\delta\psi,\psi] &= \sum_{n,m} \bar{\psi}(n) D_{\mathrm{quark}}[U](n,m) \psi(m).
\end{aligned}
\tag{4.8.1}
$$

ここで、$U_\mu(n)$ はグルーオンの自由度を表す 3×3 の特殊ユニタリー行列で格子点 n から μ 方向に伸びた単位格子辺上にある。$\psi(n)$ と $\bar{\psi}(n)$ はそれぞれクォークと反クォークの自由度を表す格子点 n 上にある 3×4 成分の複素グラスマン数である。（図 4.8.1 参照）$S_{\rm gluon}[U]$ と $S_{\rm quark}[U]$ はそれぞれ格子上でのグルーオンの作用積分とクォークの作用積分である。古典力学ではこれらの作用積分からグルーオンとクォークの運動方程式が得られる。$D_{\rm quark}[U](n,m)$ はクォークの格子上での運動方程式の元となる格子上の差分演算子である。この差分演算子は $U_\mu(n)$ を含んでおり、この差分演算子を通じてクォークとグルーオンの相互作用が生じる。$O[U,\bar{\psi},\psi]$ は一般的な物理量を表し、これらはグルーオンとクォークの自由度で記述される。クォークの自由度であるグラスマン数の積分は解析的に行うことができ、以下のようなグルーオンの自由度 $U_\mu(n)$ のみの多重積分で記述することができる。

$$\begin{aligned}\langle O\rangle &= \frac{1}{Z}\prod_{n,\mu}\int \mathrm{d}U_\mu(n) O[U,G_{\rm quark}[U]]\det[G_{\rm quark}[U]]e^{-S_{\rm gluon}[U]},\\ Z &= \prod_{n,\mu}\int \mathrm{d}U_\mu(n)\det[D_{\rm quark}[U]]e^{-S_{\rm gluon}[U]},\\ G_{\rm quark}[U](n,m) &= \left[\left(D_{\rm quark}[U]\right)^{-1}\right](n,m).\end{aligned}\tag{4.8.2}$$

ここで、$G_{\rm quark}[U](n,m)$ は格子上の差分演算子 $D_{\rm quark}[U](n,m)$ の逆行列であり、4 次元時空での格子点 m から n へのクォークの伝播の量子力学的確率振幅を表す量となっている。格子 QCD で取り扱う自由度は 4 次元格子の点の数に比例しおよそ $\mathcal{O}\left(100^4\right)$ 次元の多重積分であるので、積分の評価にはモンテカルロ法が用いられる。

クォークを含む格子 QCD では上述のように $G_{\rm quark}[U]$ や $\det[D_{\rm quark}[U]]$ のような $D_{\rm quark}[U]$ の逆行列や行列式を取り扱う必要がある。4 次元格子の一辺の格子点数が 100 のときの行列 $D_{\rm quark}[U]$ は成分が複素数の $3\times4\times100^4$ 次元の正方行列であるので、厳密にこの逆行列や行列式を評価することは不可能である。格子 QCD の数値シミュレーションでは逆行列の評価には反復法を用い、行列式の評価にはガウス積分を通じたモンテカルロ積分を用いる。格子 QCD でのシミュレーションではこれらのことを考慮したモンテカルロ積分法のハイブリッドモンテカルロ法を用いる。ハイブリッドモンテカルロ法での物理量 O の積分式は以下のようになる。

$$\begin{aligned}\langle O\rangle &= \frac{1}{Z}\prod_{n,\mu}\int \mathrm{d}P_\mu(n)\prod_{n,\mu}\int \mathrm{d}U_\mu(n)\prod_n\int \mathrm{d}\phi^\dagger(n)\,\mathrm{d}\phi(n) O[U,G_{\rm quark}[U]]e^{-H[P,U,\phi^\dagger\phi]},\\ Z &= \prod_{n,\mu}\int \mathrm{d}P_\mu(n)\prod_{n,\mu}\int \mathrm{d}U_\mu(n)\prod_n\int \mathrm{d}\phi^\dagger(n)\,\mathrm{d}\phi(n) e^{-H[P,U,\phi^\dagger\phi]},\\ H[P,U,\phi^\dagger,\phi] &= \sum_{n,\mu}\frac{1}{2}\,\mathrm{Tr}\left[\left(P_\mu(n)\right)^2\right]+S_{\rm gluon}[U]+\sum_{n,m}\phi^\dagger(n)G_{\rm quark}[U](n,m)\phi(m).\end{aligned}$$

ここで、$\phi^\dagger(n),\phi(n)$ は行列式 $\det[D_{\rm quark}[U]]$ を再現するように導入された複素変数である。$P_\mu(n)$ はハイブリッドモンテカルロ法で導入された 3×3 のエルミート行列であり、$U_\mu(n)$ に対する補助変数である。この積分表示は統計力学の熱平衡状態での物理量の値を評価する正準集団の評価式と形式的に同じである。ハイブリッドモンテカルロ法ではここからマルコフ連鎖モンテカルロ法とメ

トロポリス法を組み合わせて $U_\mu(n)$ をサンプリングしていく。

ハイブリッドモンテカルロ法での $U_\mu(n)$ のサンプリングに用いるマルコフ連鎖の遷移には、$H[P,U,\phi^\dagger,\phi]$ を古典力学のハミルトニアンとみなした時間発展を用いる。このとき、$P_\mu(n)$ は $U_\mu(n)$ に対する正準共役な運動量とみなす。また、$\phi^\dagger(n),\phi(n)$ は正規分布 $e^{-\phi^\dagger G_{\rm quark}[U]\phi}$ から生成された外場として取り扱う。$P_\mu(n)$ と $U_\mu(n)$ に対する運動方程式は

$$\begin{aligned}\dot{U}_\mu(n) &= \mathrm{i}P_\mu(n)U_\mu(n),\\ \dot{P}_\mu(n) &= -\left(\frac{\partial S_{\rm gluon}}{\partial U_\mu(n)}\right)[U] + \phi^\dagger G_{\rm quark}[U]\left(\frac{\partial D_{\rm quark}}{\partial U_\mu(n)}\right)[U]G_{\rm quark}[U]\phi,\end{aligned} \tag{4.8.3}$$

となる。ここで、$\phi^\dagger(n),\phi(n),D_{\rm quark}(n,m),G_{\rm quark}(n,m)$ の格子座標に関する和はベクトルと行列の積として表現して省略した。格子 QCD のシミュレーションで最も時間を要する部分の一つが、このハイブリッドモンテカルロ法で用いる古典力学の運動方程式を数値的に積分するところである。特に $P_\mu(n)$ の運動方程式に含まれるクォークからの寄与の部分に $U_\mu(n)$ に依存した逆行列 $G_{\rm quark}[U]$ の部分がある。古典力学の運動方程式 (4.8.3) を数値的に積分する手法として分子動力学法を用いるが、このとき時間刻みのステップ毎に $U_\mu(n)$ が変化し、それに応じて各ステップで逆行列の計算が必要となる。

この逆行列の計算を行う部分を「クォークソルバー」と呼び、格子 QCD 計算を効率良く進めるためにはクォークソルバーの性能が大変重要である。本報告ではクォークソルバーの MC-Core アーキテクチャ上での性能を机上評価する。

4.8.2 クォークソルバー

クォークの格子上での運動方程式は $O(a)$ 改良型 ウィルソンフェルミオン（クローバーフェルミオン）を用いる。クォークソルバーの計算アルゴリズムとして BiCGStab 法を用いる。クローバーフェルミオンの係数行列および解くべき連立方程式は以下で与えられる。

$$\sum_{m\in(\text{4D lattice sites})}\sum_{b=1}^{3}\sum_{\beta=1}^{4} D[U]^{a,b}_{\alpha,\beta}(n,m)x^b_\beta(m) = b^a_\alpha(n), \tag{4.8.4}$$

$$D[U]^{a,b}_{\alpha,\beta}(n,m) = \delta_{n,m}F^{a,b}_{\alpha,\beta}(n) - \kappa\sum_{\mu=1}^{4}\Big[(I-\gamma_\mu)_{\alpha,\beta}U^{a,b}_\mu(n)\delta_{n+\hat{\mu},m} + (I+\gamma_\mu)_{\alpha,\beta}\big(U^{b,a}_\mu(m)\big)^*\delta_{n,m+\hat{\mu}}\Big]. \tag{4.8.5}$$

ここで $D^{a,b}_{\alpha,\beta}[U](n,m)$ が係数行列 (添字 $_{\rm quark}$ を省略した) の成分で、n,m は 4 次元格子座標、a,b は QCD の色自由度で a,b はそれぞれ $1,2,3$ の値を取り、α,β はクォークのスピンの自由度を表しそれぞれ $1,2,3,4$ の値を取る。$x^b_\beta(m)$, $b^a_\alpha(n)$ はクォーク場を表している。$F^{a,b}_{\alpha,\beta}(n)$ は $O(a)$ 改良のために導入した $U_\mu(n)$ で構成される量でクローバー項と呼ばれる項である。係数行列は図 4.8.2 のような一階差分のステンシル構造を持つ。以下では式 (4.8.4) の添字を省略した連立方程式を

$$Dx = b \tag{4.8.6}$$

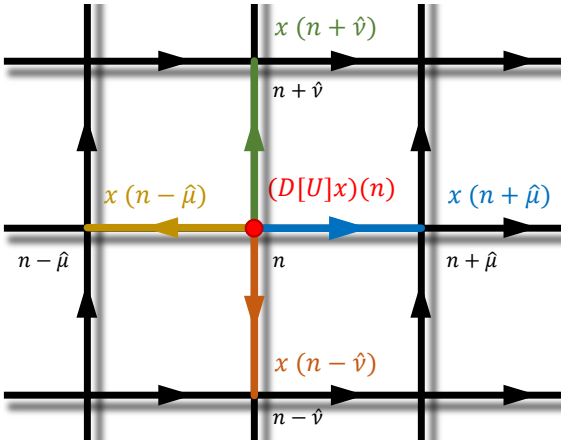

図 4.8.2　クローバフェルミオンの係数行列のステンシル構造の模式図

と表記する。

式 (4.8.6) の y と D が与えられたときに x を解く解法には反復解法の BiCGStab 法を用いる。本調査研究では式 (4.8.6) を解く BiCGStab 法ソルバーの性能を机上評価する。さらに、式 (4.8.6) のような格子上での差分演算子を係数行列に持つ連立方程式の反復解法には数値解の収束の加速のために前処理法が使われる。ここでは、4 次元の各方向の格子点の数は偶数であり 4 次元格子には周期境界条件を課した場合を考える。このときに有効な前処理法として Red-Black （または Even-Odd）前処理を用いる。4 次元格子点座標 n を $n = (n_x, n_y, n_z, n_t)$ とし、各方向の座標 $n_i, i = x, y, z, t$ は $n_i = 0, \ldots, N_i - 1$ を取るものとする。N_i は偶数である。格子点座標 n の偶奇性 eo を $eo = n_x + n_y + n_z + n_t \mod 2$ とする。このとき、格子点座標 n の集合 n は $eo = 0$ の偶格子座標の集合 n_e と $eo = 1$ の奇格子座標の集合 n_e に分割される。係数行列 (4.8.5) に現れる格子差分（ステンシル）構造は一階差分の形のため、格子座標の差分に関して差分はつねに偶の格子座標と奇の格子座標をつなぐようになっている。ベクトル x, b のベクトルの成分の並べ方を偶格子の集合に属するものを先に置き、奇格子の集合に属するものを後ろに置くようにすることで連立方程式 (4.8.6) は以下のようにブロック表示される。

$$\begin{pmatrix} D_{ee} & D_{eo} \\ D_{oe} & D_{oo} \end{pmatrix} \begin{pmatrix} x_e \\ x_o \end{pmatrix} = \begin{pmatrix} b_e \\ b_o \end{pmatrix}. \tag{4.8.7}$$

ここで、x_e（b_e）と x_o（b_o) はそれぞれ偶格子座標を持つ成分と奇格子座標を持つ成分である。D_{eo} に D_{oe} に格子点座標に関する差分が含まれ、D_{ee} と D_{oo} には格子点座標に関する差分は含まれない。式 (4.8.7) は以下の方程式に簡略化される。

$$\tilde{D}_{ee} x_e = \tilde{b}_e, \tag{4.8.8}$$

$$x_o = D_{oo}^{-1} b_o - D_{oo}^{-1} D_{oe} x_e, \tag{4.8.9}$$

ここで、

$$\tilde{b}_e \equiv D_{ee}^{-1} b_e - D_{ee}^{-1} D_{eo} D_{oo}^{-1} b_o, \quad \tilde{D}_{ee} \equiv I_{ee} - D_{ee}^{-1} D_{eo} D_{oo}^{-1} D_{oe}$$

である。式 (4.8.8) を x_e について解き、式 (4.8.9) を用いることで全体の解ベクトル $x = (x_e, x_o)^T$ を得ることができる。式 (4.8.8) が Red-Black 前処理を施した後の連立方程式である。本調査研究では式 (4.8.8) を BiCGStab 法で解いた場合の性能を机上評価する。BiCGStab 法のアルゴリズムを Alg. 1 に示した。以下では表記の簡略化のため $\tilde{D}_{ee} \to A$, $x_e \to x, \tilde{b}_e \to b$ とする。行列ベクトル積 $q = Ap$ の計算の詳細は Alg. 2 に示した。

Algorithm 1 BiCGStab アルゴリズム

input x,b, Solve $Ax = b$, $x = 0$ start.
$x = 0$
$r = b$
$\tilde{r} = r$ ($\tilde{r}$ can be arbitrary.)
$p = r$
$\sigma = |b|$
$\rho_0 = \langle \tilde{r} | r \rangle = |b|^2$
for **do**
  $q = Ap$
  $\alpha = \rho_0 / \langle \tilde{r} | q \rangle$
  $x = x + \alpha p$
  $r = r - \alpha q$
  err $= |r|$
  if err$/\sigma < \epsilon$ **then**
    exit
  end if
  $t = Ar$
  $\omega = \langle t | r \rangle / |t|^2$
  $x = x + \omega r$
  $r = r - \omega t$
  err $= |r|$
  if err$/\sigma < \epsilon$ **then**
    exit
  end if
  $\rho = \langle \tilde{r} | r \rangle$
  $\beta = (\alpha/\omega)(\rho/\rho_0)$
  $\rho_0 = \rho$
  $p = p - \omega q$
  $p = r + \beta p$
end for

Algorithm 2 行列ベクトル積の計算の詳細

input p, output $q = Ap$, where $p = p_e, A = \tilde{D}_{ee}, q = q_e$
$y_o = D_{oo}^{-1} D_{oe} p_e$
$q_e = D_{ee}^{-1} D_{eo} y_o$
$q_e = p_e - q_e$

4.8.3 計算時間の見積もりと性能評価

将来必要となる、素粒子・ハドロン物理学に用いる格子 QCD の計算の格子の大きさはおおよそ $256^4 \sim 512^4$ と見積もられている。MN-Core アーキテクチャでは演算器 (Processing Element, PE) に局所メモリ (Local Memory, LM) を持ち、また多くのレジスターファイル (General Regisger File, GRF) を持つことで性能を上げる。LM の容量は 4096×2 長語 (Long Word, LW) であり、倍精度実数を 4096×2 個保持できる。MN-Core の 1 ボードには PE が 8192 PE 搭載されており、ボードあたりの LM 容量は 64 MiLW である。将来の MN-Core アーキテクチャではこれらの容量は増強さ

れると見込まれる。まずデータサイズについて評価した後、計算時間の見積もりを行う。計算時間の見積もりではステンシル計算に必要な袖交換の時間を見積もるが、そのネットワーク性能として「富岳」の Tofu D ネットワークの性能を基に評価する。BiCGStab 法 Alg. 1 の反復を 2000 回行ったときの計算時間とその性能を評価する。

4.8.3.1 データサイズ

1 ボードの LM 容量からボードあたりの格子サイズを $16^3 \times 32$ とする。前節で述べたように Red-Black 前処理を施した場合のクォーク場のベクトルの格子サイズはその半分の 16^4 となる。これを元に、BiCGStab (Arg.1) の実行に必要なデータサイズを見積もると表 4.8.1 のようになる。必要なデータサイズの評価には 1 複素数の保持に 2LW を用いるとしている。この評価からステンシル計算に必要な PE 間やボード間での袖交換のためのバッファをのぞくデータ数は 31.5 [MiLW] である。

ステンシル計算に必要な PE 間やボード間での袖交換のためのバッファについての見積もりをおこなう。特に、PE 間での袖交換は高速に行う必要があるため、LM 上でのバッファリングを行った後に交換する。現在の MN-Core 1 ボードの PE 数は 8192 個であるので PE あたりの格子サイズは 2^4 となる。Red-Black 前処理した場合の 4 次元ステンシル計算 (1 階差分) で各上下方向に袖を交換するためのバッファサイズは $2^3 + 3 \times 2 \times 2^2 = 32$ 格子点である。ここで T 方向は Red-Black 前処理のため一方向のみ送受し X, Y, Z 方向はそれぞれ上下方向送受する。送信用バッファと受信用バッファの両者を用意するので $32 \times 2 = 64$ 格子点と見積もる。ボードあたりは $64 \times 8192 = 524288$ 格子点である。フェルミオンベクトルは格子点に 3×4 複素数の要素を持つが袖交換の場合はスピン射影 $I \pm \gamma_\mu$ の性質から 4 スピンの内 2 スピンのみ交換すればよいので 3×2 複素数を交換する。送信用と受信用のバッファをそれぞれ用意する。袖交換のためのバッファサイズは $64 \times 3 \times 2 \times 2 \times 8192$[複素数] $= 12582912$ [LW] $= 12.0$ [MiLW] と見積もる。現在の MN-Core の 1 ボードの LM サイズは 64 [MiLW] である。上述の主要なデータサイズ 31.5 [MiLW] と袖交換に必要なバッファサイズ 12.0 [MiLW] は 1 ボードの LM サイズ 64 [MiLW] に収まっている。将来の MN-Core アーキテクチャでは LM 容量は増えることが予想されるので、ボードあたり $16^3 \times 32$ の格子サイズの計算は可能であると見積もる。

4.8.3.2 計算時間

計算時間の見積もりは BLAS1 相当のベクトル計算の計算時間とステンシル計算の計算時間の見積もりをそれぞれ評価した。上述の見積もりからデータはすべて LM 上に乗っていることとし、LM から必要なデータをロードして演算器で計算し結果を LM にストアするのに必要なサイクル数を評価し、それを MN-Core の動作周波数（500 MHz）で除して計算時間を見積もった。ボードあたりのベクトルの格子点数は Red-Black 前処理のため 16^4 である。表 4.8.2 に Alg. 1 に含まれる BLAS1 相当のベクトル計算の計算時間を示す。内積やノルムの計算ではボード内縮約のための縮約・放送の時間（1LW あたり約 1 μsec）を含んでいる。縮約・放送のためのボード間での通信時間は「富岳」 Tofu D ネットワークの性能を基に考察する。「富岳」でのベンチマークの結果では

変数	個数	[複素数] 要素数/個数	合計データサイズ [LW]
BiCGStab 法 (Alg. 1:$x, b, r, \tilde{r}, p, q, t$)	7	$3 \times 4 \times 16^4$	$7 \times 3 \times 4 \times 16^4 \times 2$
行列ベクトル積 (Alg. 2：y_o)	1	$3 \times 4 \times 16^4$	$3 \times 4 \times 16^4 \times 2$
リンク変数 $U_\mu(n)$	1	$3 \times 3 \times 4 \times 16^3 \times 32$	$3 \times 3 \times 4 \times 16^3 \times 32 \times 2$
クローバー項 $F(n)$	1	$21 \times 2 \times 16^3 \times 32$	$21 \times 2 \times 16^3 \times 32 \times 2$
合計			33030144 [LW]
袖交換バッファー	64 格子点/PE	$64 \times 3 \times 2 \times 2$/PE	$64 \times 3 \times 2 \times 2 \times 2 \times 8192$
バッファー込みの合計			33030144 + 12582912 [LW] = 43.5 [MiLW]

表 4.8.1　$16^3 \times 32$ 格子サイズでの BiCGStab 方を用いる場合に必要なデータサイズ

$48 \times 12 \times 48$ ノードの縮約・放送の時間はデータサイズが 12 Byte より小さい場合（Tofu バリアあり） 9 [μsec] 、（Tofu バリアなし） 70 [μsec] でほぼ一定、12 Byte より大きい場合は（Tofu バリアあり） 60 [μsec] 、（Tofu バリアなし） 100 [μsec] から増えていく。ここでは実数 1 つの縮約に 100 [μsec] かかると仮定した。内積ノルム計算の時間は縮約と放送のためのボード間の通信時間が支配的である。

係数行列の掛け算（行列ベクトル積・ステンシル計算）$q_e = D_{ee}^{-1} D_{eo} p_o$ の計算時間の評価は以下のとおりである。格子点 1 点あたりに必要な命令数は 792fma + 264add + 12mul（ここで、fma は積和算命令 add は加減算命令、mul は積算命令）であり、1068 命令である。MN-Core ではこれらの命令はおおよそ 1 cycle で評価できるがデータの依存性や LM へのデータの配置、演算命令が連続的に発行されるためには条件が整っている必要がある。正確なサイクル数を見積もるには実際にアセンブリ (vsm 言語) プログラムを書き、これらの条件を評価検討する必要がある。ここでは、正確なサイクル数を求めるまで至っていないため、不確定要素として因子 2 を導入し、必要な計算サイクル数をおよそ 2000 cycle と評価する。ボードあたりの格子点数は Red-Black 前処理のため 16^4 であり、PE 数は 8192 個である。これを処理するのに要する計算時間は (2000[cycle/格子点/PE]) $\times$ (16^4[格子点])/(8192[PE])/(500[MHz]) = 32[μsec] と評価した。この時間にはボード内の PE 間のステンシル計算の袖交換に要する時間は含まれていない。

ボード内での袖交換に必要な時間は PE 間での袖交換用バッファ（送信用と受信用）のデータサイズ、各 PE 間の LM の間での送受信の時間から見積もる。PE あたりの格子サイズは $2^3 \times 1$ であるが袖交換する方向は T 方向は上方下方のどちらか一方向、XYZ 方向については上下方の両方向が必要である。フェルミオンの袖のための格子の大きさは各 PE で $2^3 + 3 \times 2 \times 2^2 = 32$ 格子点である。カラーとスピンの自由度を考慮するとデータサイズは $3 \times 2 \times 32$[複素数] = 384[LW] である。送受信の 2 枚のバッファーサイズは 768 [LW] である。PE 間での袖交換は L2B メモリを経由して行う。このとき PE での LM 上のバッファへの読み書きの時間は無視できるとする。ボード内での袖交換の時間を減らすために、X 方向の分割がチップをまたぐ L2B メモリの間の袖交換、Y 方向はチップ内での異なる L2B メモリの間の袖交換とする。Z 方向と T 方向については同一の L1B/L2B 内での袖交換とする。チップをまた

計算の種類	計算時間 [μsec]	ボード間縮約時間 [μsec]
$y = y + \alpha x, y = x + \alpha y$	1–2	
$z = x - y$	~ 1	
$y = x$	~ 1	
$\alpha = \langle x \vert y \rangle$	2–3	~ 200
$\alpha = \vert x \vert^2$	1–2	~ 100

表 4.8.2　BLAS1 相当のベクトル計算の計算時間の見積もり

ステンシル計算	計算時間 [μsec]	ボード内袖交換時間 [μsec]	ボード間袖交換時間 [μsec]	時間 [μsec] (隠蔽なし)	時間 [μsec] (隠蔽あり)
$q_e = D_{ee}^{-1} D_{eo} p_o$	~ 32	~ 148	~ 162	~ 342	~ 194
$q_e = \tilde{D}_{ee} p_e$	$\sim (32 \times 2 + 1)$	$\sim 148 \times 2$	$\sim 162 \times 2$	~ 685	~ 389

表 4.8.3　ステンシル計算の計算時間の見積もり

ぐ L2B メモリの間のデータ転送速度はボードあたり 16LW/cycle, チップ内の L2B メモリの間のデータ転送速度はボードあたり 32 LW/cycle とし、同一 L2B メモリ内でのデータ転送時間は無視できるとする。このとき、X 方向はボード内で $16 \times 16^2 \times 3 \times 4 \times 2 \times 2 \times 4 = 786432$ [LW] のデータを移動する。移動の時間は 786432 [LW]/16 [LW/cycle]/500 [MHz] $\sim$ 98 [μsec] と見積もる。Y 方向はボード内で $4 \times 16^2 \times 3 \times 4 \times 2 \times 2 \times 16 = 786432$ [LW] のデータを移動しその時間は 786432 [LW]/32 [LW/cycle]/500 [MHz] $\sim$ 49 [μsec] と見積もる。したがって、ボード内での袖交換に要する時間は約 148 [μsec] と見積もる。

ボード間での袖交換に必要な時間は、ボード内の格子点 $16^3 \times 16$ の表面の格子点の数とボード間通信のネットワークのバンド幅から見積もる。ここで、ステンシル計算の構造からフェルミオンベクトルは $I \pm \gamma_\mu$ のスピン射影を施したものを送受信することでスピン自由度 4 のうちの半分の 2 個を送受信する。X,Y,Z 方向の袖の格子点の数は 16^3 個でそれぞれの方向の上下に複素数を 3×2 個送受信する。T 方向の一つの even/odd ベクトルの袖の格子点の数は 16^3 個で上方向または下方向の一つの方向に複素数を 3×2 個送受信する。ボードあたりの送信する（受信する）データサイズは $(3 \times 2 + 1)$ [方向] $\times 16^3 \times 3 \times 2$ [複素数] $= 344064$ [LW] である。LM 実数データを倍精度 64 bit 実数で換算して約 2.6 [MiB] である。一方向あたりは $16^3 \times 3 \times 2$[複素数] $= 49152$[LW] $= 384[KiB]$ である。ネットワークを通じた袖交換の時間を「富岳」の Tofu D の性能を基に見積もる。Tofu D のリンクあたりの性能は、レイテンシ 0.5 μsec、バンド幅 6.8 GB/s [1MB データサイズ] であった。ここから 1 方向の袖交換の時間は倍精度実数で 54 μsec と見積もることができる。今の場合の 1 方向のデータサイズは 384 [KiB] で Tofu D の有効バンド幅を引き出すことができない事が考えられる。データサイズが 1/3 のときに有効バンド幅も 1/3 になるとすると、162 μsec となる。4 次元の各方向に対する通信を同時に行うことができるとすると、ボード間での袖交換に必要な時間は 54–162 μsec と見積もる。ボード内の袖交換時間とボード間の袖交換時間はほぼ同じ時間である。ここで、ボード内の袖交換とボード間の袖交換を同時に行うことができる場合はボード内の袖交換の時間をボード間の袖交換の時間に隠蔽することが可能である。

表 4.8.3 に行列ベクトル積 $q_e = D_{ee}^{-1} D_{eo} p_o$ の計算時間の見積もりを記載した。反復法で解く係数行列の行列ベクトル積は $q_e = \tilde{D}_{ee} p_e$ であり、これは Alg. 2 にしたがって計算する。$q_e = D_{ee}^{-1} D_{eo} p_o$ の計算が 2 回とベクトルの差の計算を一回行うことから袖交換時間を含む計算時間は $\sim$ 685 または ~ 389 [μsec] と見積もる。

以上のベクトル計算やステンシル計算の時間の見積もりを基に、BiCGStab 法 (Alg. 1) の 2000 回の反復に要する時間を評価する。Alg. 1 の 8 行目から 30 行目の間のループ回数を $N_{\rm loop}$ とすると、計算時間は

$$T_{[\text{隠蔽あり}]} = 105 + (389 \times 2 + 2 \times 6 + 200 \times 3 + 100 \times 3) N_{\rm loop}\ [\mu\text{sec}] = 105 + 1690 N_{\rm loop}\ [\mu\text{sec}], \tag{4.8.10}$$

$$T_{[\text{隠蔽なし}]} = 105 + (685 \times 2 + 2 \times 6 + 200 \times 3 + 100 \times 3) N_{\rm loop}\ [\mu\text{sec}] = 105 + 2282 N_{\rm loop}\ [\mu\text{sec}], \tag{4.8.11}$$

となる。$N_{\rm loop} = 2000$ では

$$T_{[\text{隠蔽あり}]} \sim 3.4\ [\text{sec}], \tag{4.8.12}$$

$$T_{[\text{隠蔽なし}]} \sim 4.6\ [\text{sec}], \tag{4.8.13}$$

となる。

4.8.3.3 ボード単体での性能評価

ここでは、$16^3 \times 32$ 格子サイズに固定しボード単体での計算時間を評価する。ボード間の袖の交換に伴う通信時間とベクトルの内積に伴う縮約・放送に伴う通信時間を取り除いた BiCGStab 法の 2000 反復の計算時間は以下のように見積もられる。

$$T_{[\text{ボード単体}]} = 5 + (361 \times 2 + 2 \times 6 + 2 \times 3 + 1 \times 3) N_{\rm loop} = 5 + 743 N_{\rm loop}\ [\mu\text{sec}] \tag{4.8.14}$$

$$= 5 + 743 \times 2000\ [\mu\text{sec}] \sim 1.5\ [\text{sec}]. \tag{4.8.15}$$

総演算数 $N_{\rm FLO}$ は

$$N_{\rm FLO} = (48 + 8544 N_{\rm loop}) \times 16^4 = (48 + 8544 \times 2000) \times 16^4 \sim 1120\ [\text{GFLO}] \tag{4.8.16}$$

である。性能は

$$1120/1.5 \sim 750\ [\text{GFLOPS}] \tag{4.8.17}$$

となる。

ボードあたりのピーク性能は、小規模行列ベクトル積演算器を利用する場合：約 32.8 TFLOPS、利用しない場合約 8.2 TFLOPS である。本見積もりでは小規模行列ベクトル積演算器を利用していない想定による見積もりのため性能が低く出ている。小規模行列ベクトル積演算器を利用しない場合のピーク性能での効率は $0.75/8.2 \sim 0.09$ と見積もった。

アーキテクチャ	実行環境	数値精度	2000 反復の 実行時間 [sec]	性能 [GFLOPS]
CPU: Intel Xeon W-2223 3.6GHz, 4 cores	Intel OneAPI, ifort OpenMP 4 threads	倍精度	55.8	20.4
CPU: 同上 GPU: Nvidia A2000	Intel OneAPI, ifort OpenMP 4 threads Nvidia CUDA 12.1	混合精度	全体 5.12 (単精度部分 4.53)	251.2 (単精度)

表 4.8.4　比較対象アーキテクチャと性能

4.8.3.4　既存 CPU や GPU との性能比較

ボード単体での性能を既存の CPU や GPU で実測したものと比較検討した。表 4.8.4 に環境と実行結果の値を示した。BiCGStab 法の CPU プログラムは Fortran で実装され倍精度で計算する。一方、GPU プログラムの方はアルゴリズムを混合精度（倍精度と単精度を使う）を用いた方法を用いている。具体的には単精度の BiCGStab 法を倍精度の反復解法の前処理として用いることで倍精度の解を得る手法である。GPU プログラムの反復数は単精度の BiCGStab 法の反復数が 2000 となるようにして比較した。

CPU の倍精度の計算結果は約 20 GFLOPS の性能で MN-Core アーキテクチャでの性能の約 38 分の一であった。CPU のピーク性能は約 460 GFLOPS であるので、効率は約 0.04 であった。GPU の計算は単精度で行っているので単純に比較することはできないが、MN-Core で単精度で計算する場合の参考として GPU 計算の効率を評価する。該当 GPU1 ボードの単精度のピーク性能は約 8 TFLOPS である。効率は約 0.03 であった。

4.8.4　まとめ

素粒子・原子核物理における応用アプリケーションとして格子量子色力学の計算に用いられるクォークソルバーの MN-Core 上での性能を見積もった。クォークソルバーに用いられる反復法 BiCGStab 法の反復 2000 回の時間を見積もった。また、格子サイズ $16^3 \times 32$ についてボード単体での性能を見積もった。既存の CPU と GPU の環境で実際にベンチマークを取ったものと比較した。ボード単体での机上見積もりは、既存のアーキテクチャでの効率より良い結果を得た。また実行時間については、既存の単精度での実行時間と同じオーダーであった。実際の大規模な格子サイズ $256^3 \times 512$ での実行には多数のノードによる分散並列計算が必要となり、各ノードを結合するネットワーク性能が、全体性能に大きく影響を与える結果となった。ここ数年の GPU 加速器の技術動向はボードあたりの性能やメモリ容量を上げていき、分散並列度が減っていく傾向にある。また単精度や半精度の演算速度を倍精度演算速度よりも更に高速化したり専用の行列演算機構を搭載してメモリ容量やメモリバンド幅の制限を緩和する方向に進んでいる。MN-Core アーキテクチャの単精度の行列演算機構の利用や、さらなるローカルメモリ容量を大きくしていくことで、ボードあたりが担当する格子領域を増やし分散並列の度合いを低くすることができる。この場合ネットワーク性能が全体性能に与える影響を減らすことができると考えられる。

4.9 宇宙・惑星科学応用アプリケーションの検討

藤井 通子 [国立大学法人 東京大学]

宇宙・惑星科学応用アプリケーションの検討として、粒子法を用いた銀河進化シミュレーションコードの CPU および GPU クラスタを用いた性能評価を行った。

宇宙を構成する要素には、ダークマター、星、惑星などの粒子として扱うべき物質と、星間ガスのような流体として扱うべき物質がある。星やダークマター、惑星等は、重力相互作用のみを計算する粒子として、星間ガスは粒子化し、それぞれの粒子にカーネル半径と呼ばれる半径を設定し、広がった分布の重ね合わせとして流体の分布を表現する smoothed-particle hydrodynamics (SPH) 法を用いる。流体は、SPH 法ではなくメッシュによって表現することもできるが、ここでは、粒子法の場合についてまとめる。

重力の計算のために、粒子間の重力相互作用

$$\boldsymbol{a}_i = \sum_{i \neq j}^{N} \frac{G m_j (\boldsymbol{x}_j - \boldsymbol{x}_i)}{(|\boldsymbol{x}_j - \boldsymbol{x}_i|^2 + \epsilon^2)^{3/2}} \tag{4.9.1}$$

を全粒子ペアの分だけ評価する必要がある。ここで、ϵ は重力ソフトニングと呼ばれ、2 粒子が非常に近付いた時に加速度が発散してしまうことを防ぐ。この計算は粒子数 N に対し、$O(N^2)$ の計算コストとなり、粒子数が多くなると計算が困難となる。そのため、粒子を含む入れ子上のボックスのツリー構造を用い、近傍の粒子からの力は直接評価し、遠方の粒子からの力は、ボックスの重心で代替することで計算コストを $O(N \log N)$ に抑えることができるツリー法 (Barnes and Hut 1986) を用いる。

SPH 法 (Gingold and Monaghan 1977; Lucy 1977) では、ある粒子の物理量 (f_i) を、近傍の粒子を用いて、それらの重ね合わせ

$$f_i = \sum_j m_j \frac{f_j}{\rho_j} W(r_{ij}, h) \tag{4.9.2}$$

で表す。ここで、ρ_j、m_j は近傍の粒子の密度と質量、r_{ij} は 2 粒子間の距離である。h はカーネル半径と呼ばれ、カーネル関数 W のスケールを決めるパラメータであり、粒子分布の広がりを表現しており、粒子が集まる密度の高い場所で小さく、密度が低い場所で大きな値を取る。カーネル関数 W としては、スムーズに 0 から 1 まで変化する関数を用いる。SPH 法の実装にはいくつかのバリエーションがあるが、以下の計測では Density-Independent (DI) SPH 法 (Saitoh and Makino 2013) を用いている。

このようなの粒子系シミュレーションにおいて、計算コストが最も高い部分は、上記のような重力や流体の相互作用計算である。力を受ける粒子を i 粒子と呼び、力を及ぼす側の粒子を j 粒子と呼び、それぞれの粒子の配列（粒子リスト）を作り、それらの間の相互作用計算を行う。この相互作用計算を高速に行うための計算カーネルが計算の高速化において重要となる。相互作用計算カー

ネル生成プログラム「PIKG」[1)]を用いて生成した計算カーネルを使用することで、高速な相互作用計算を行える計算カーネルを生成した。これを用いて、汎用 CPU（Intel x86-64, Fujitsu A64FX）および GPU（NVIDIA GH200）で重力相互作用計算および流体相互作用計算の性能評価を行った。シングル CPU コアおよびシングル GPU での性能 (FLOPS、単精度ピーク性能) は表 4.9.1 のとおりである。(演算数は重力 27、流体（力）101、流体（密度・圧力）73 である。)

表にあるように、重力相互作用計算の方が流体計算よりも高いピーク性能を出た。GPU の場合は特に差が顕著であるが、これは、流体計算の方が重力計算と比べて力を及ぼす側の粒子が近傍粒子に限られ、j 粒子のリストが短いためである。また、流体計算の場合、相互作用計算が十分に速ければ、近傍粒子の探索の時間が相対的に長くなる。今後、近傍粒子探索の高速化も必要となってくる。重力計算について GPU を使用する場合、ツリーの構築、走査、相互作用リストの生成も含めて GPU 上で行うとさらに全体の計算時間を短縮できる (Bédorf *et al.* 2014)。

現在、GPU 並列計算に対応している他のコードには以下がある。

- **AthenaK**: Adaptive mesh refinement (AMR) 法を用いた磁気流体計算コード (Stone *et al.* 2024)
- **QUOKKA**: AMR 法を用いた輻射流体計算コード (He *et al.* 2024; Wibking and Krumholz 2022)
- **SHAMROCK**: SPH 法を用いた N 体/流体計算コード (David-Cléris *et al.* 2025)

	Fujitsu A64FX SVE		AMD EPYC^{TM} 9474F AVX2		AMD EPYC^{TM} 9474F AVX512		NVIDIA Grace Hopper H100	
カーネル	FLOPS	efficiency	FLOPS	efficiency	FLOPS	efficiency	FLOPS	efficiency
重力	37.3G	29.4	65.8G	50.2	90.6G	69.1	22.5T	38.0
流体 (力)	19.8G	15.4	29.4G	22.4	81.5G	62.1	1.88T	2.8
流体 (密度・圧力)	21.9G	17.1	15.1G	11.5	87.6G	66.8	0.555T	0.64

表 4.9.1　相互作用計算性能の評価。Fujitsu A64FX は「富岳」、AMD EPYC はアメリカの Flatiron Institute のシステム、NVIDIA Grace Hopper H100 は東京大学情報基盤センターの「Miyabi」を用いた。

1）https://github.com/FDPS/PIKG

第 4 章の参考文献

AIMES project (2018). *IcoAtmosBenchmark v1*. URL: https://aimes-project.github.io/IcoAtmosBenchmark_v1/.

Barnes, J., and P. Hut (1986). "A hierarchical O(N log N) force-calculation algorithm". In: *Nature* 324.6096, pp. 446–449. DOI: 10.1038/324446a0.

Bédorf, J., E. Gaburov, M. S. Fujii, K. Nitadori, T. Ishiyama, and S. Portegies Zwart (2014). "24.77 Pflops on a Gravitational Tree-Code to Simulate the Milky Way Galaxy with 18600 GPUs". In: *Proceedings of the International Conference for High Performance Computing*, pp. 54–65. DOI: 10.1109/SC.2014.10. arXiv: 1412.0659 [astro-ph.GA].

David-Cléris, T., G. Laibe, and Y. Lapeyre (2025). "The SHAMROCK code: I - smoothed particle hydrodynamics on GPUs". In: *Monthly Notices of the Royal Astronomical Society* 539.1, pp. 1–33. DOI: 10.1093/mnras/staf444. arXiv: 2503.09713 [astro-ph.IM].

Gingold, R. A., and J. J. Monaghan (1977). "Smoothed particle hydrodynamics: theory and application to non-spherical stars". In: *Monthly Notices of the Royal Astronomical Society* 181, pp. 375–389. DOI: 10.1093/mnras/181.3.375.

He, C.-C., B. D. Wibking, and M. R. Krumholz (2024). "A novel numerical method for mixed-frame multigroup radiation-hydrodynamics with GPU acceleration implemented in the QUOKKA code". In: *Monthly Notices of the Royal Astronomical Society* 535.4, pp. 3059–3076. DOI: 10.1093/mnras/stae2580. arXiv: 2407.18304 [astro-ph.GA].

Kawai, H., M. Ogino, R. Shioya, T. Yamada, and S. Yoshimura (2016). "Performance Evaluation of Local Schur Complement Approach in Domain Decomposition Method". In: *Transactions of JSCES* 20160006, pp. 1–10.

Kunkel, J., N. Jumah, A. Novikova, T. Ludwig, H. Yashiro, N. Maruyama, M. Wahib, and J. Thuburn (2020). "AIMES: Advanced Computation and I/O Methods for Earth-System Simulations". In: *Software for Exascale Computing - SPPEXA 2016-2019*. Ed. by H.-J. Bungartz, S. Reiz, B. Uekermann, P. Neumann, and W. E. Nagel. Cham: Springer International Publishing, pp. 61–102.

Lucy, L. B. (1977). "A numerical approach to the testing of the fission hypothesis". In: *The Astrophysical Journal* 82, pp. 1013–1024. DOI: 10.1086/112164.

Ogino, M., R. Shioya, H. Kawai, and S. Yoshimura (2005). "Seismic Response Analysis of Nuclear Pressure Vessel Model with ADVENTURE System on the Earth Simulator". In: *Journal of The Earth Simulator* 2, pp. 41–54.

PFN Blog (2022). JAX から MN-Core を利用する. URL: https://tech.preferred.jp/ja/blog/jax-on-mncore/.

Saitoh, T. R., and J. Makino (2013). "A Density-independent Formulation of Smoothed Particle Hydrodynamics". In: *The Astrophysical Journal* 768.1, 44, p. 44. DOI: 10.1088/0004-637X/768/1/44. arXiv: 1202.4277 [astro-ph.CO].

Satoh, M., H. Tomita, H. Yashiro, H. Miura, C. Kodama, T. Seiki, A. T. Noda, Y. Yamada, D. Goto, M. Sawada, T. Miyoshi, Y. Niwa, M. Hara, T. Ohno, S.-i. Iga, T. Arakawa, T. Inoue, and H. Kubokawa (2014). "The Non-hydrostatic Icosahedral Atmospheric Model: description and development". In: *Progress in Earth and Planetary Science* 1.1, p. 18. DOI: 10.1186/s40645-014-0018-1. URL: https://doi.org/10.1186/s40645-014-0018-1.

Stone, J. M., P. D. Mullen, D. Fielding, P. Grete, M. Guo, P. Kempski, E. R. Most, C. J. White, and G. N. Wong (2024). "AthenaK: A Performance-Portable Version of the Athena++ AMR Framework". In: *arXiv e-prints*, arXiv:2409.16053, arXiv:2409.16053. DOI: 10.48550/arXiv.2409.16053. arXiv: 2409.16053 [astro-ph.IM].

Wibking, B. D., and M. R. Krumholz (2022). "QUOKKA: a code for two-moment AMR radiation hydrodynamics on GPUs". In: *Monthly Notices of the Royal Astronomical Society* 512.1, pp. 1430–1449. DOI: 10.1093/mnras/stac439. arXiv: 2110.01792 [astro-ph.IM].

第 5 章　おわりに

牧野 淳一郎 [国立大学法人神戸大学]

まず、3 年間の調査研究に参加してくれた全員に感謝したい。本調査研究で得られた大きな成果は、参加者の貢献なしにはありえないものだった。また、提案を採択した文部科学省、評価委員、プログラムディレクターの皆様にも感謝したい。

本調査研究で提案したシステム構成そのものはポスト「富岳」で採択されるにはいたらなかったが、ポスト「富岳」に必要なアーキテクチャを原理的な問題や半導体技術、コンピュータアーキテクチャの発展の歴史と今後の方向性から検討することで、現在主流である GPU アーキテクチャの限界、それを超える新しいアーキテクチャの方向性、そのために必要な要素技術の状況を検討し、数年後のアーキテクチャの方向性を明らかにすることができた。

現代の CPU や GPU では、非常に多数の演算コアを 1 チップに集積できるようになり、非常に高い演算性能を実現できている。もちろん、ムーアの法則の限界により、電力あたりの演算性能の向上ペースはゆるやかになっているが、それでも演算性能の向上は続いている。特に AI 向けでは、演算精度を切り下げることで大幅な演算能力の向上が実現されている。

一方、主記憶はオフチップの DRAM であるため、メモリバンド幅にはチップ間の配線の本数と、配線そのものの消費電力に起因する限界がある。GPU では、メモリを伝統的な DRAM モジュールからオンボードの GDDRx パッケージ、さらにはオンパッケージの HBMx とより配線の短い実装形態に変えることで、メモリアクセスエネルギーの大きな減少とメモリバンド幅の増加を実現してきた。しかし、プロセッサダイの「横」にメモリダイを置く限り、メモリアクセスエネルギーにはどうしてもデータをチップ内で数十ミリメートル移動させることに伴う電力消費の成分が残る。これは HBM の規格が変わり、パッド数やバンド幅が増加しても基本的に変わらない。このため、HBM の消費電力はほぼバンド幅の増加に比例して増加しており、またシステム全体としての消費電力もほぼバンド幅の増加に比例して増加している。

この、共有メモリマルチコアプロセッサのメモリバンド幅の限界は、歴史的には 1990 年代初めに明らかになった共有メモリ並列プロセッサシステムですでに起こったことであり、Cray の共有メモリベクトルプロセッサや、1980 年代に多数あったマイクロプロセッサベースの共有メモリ並列システムが分散メモリ並列システムにとってかわられている。これは、1 ダイのマルチコアシステムでも、分散メモリ並列アーキテクチャになればより高い性能を実現できる、ということである。

しかし、プロセッサダイと DRAM ダイが平面的に並んでいる限り、分散メモリアーキテクチャをとってもデータ移動距離は大幅には短くならず、意味がある性能向上は得られない。言い換えると、分散メモリアーキテクチャで意味がある性能向上を実現するためには、メモリダイとプロセッ

サダイを積層することが必須である。

半導体ダイの 3 次元積層技術には長い研究開発の歴史があるが、実際に様々なシステムで利用されるようになったのは、HBM で使われている DRAM 積層を別にすると 2020 年前後になってからである。2026 年初頭の現時点で代表的なものは、 AMD Zen 3 CPU 以降で用いられている 3D V-cache と、やはり AMD GPU で用いられている base die (おそらくキャッシュメモリと、D2D インターフェースが集積されている) と compute die からなるシステムである。また、2027 年リリース予定の富士通 MONAKA プロセッサも、 Zen 4 以降と同様な、キャッシュメモリが中心の base die の上にプロセッサコアダイを載せるアプローチが採用されている。

これらのシステムは現在のところすべて、同一のロジックファブで製造した 2 ダイを積層するものであるが、積層技術にハイブリッドボンディングを用いることで、接合部分の熱抵抗をほぼゼロにするとともに、ピッチ 10 μm 以下の極めて高密度な接合を可能にしたのが従来の 3 次元積層技術との大きな違いである。例えば、2010 年代に規格化された Wide-IO DRAM では、接合は 45 μm ピッチのマイクロバンプで、 DRAM ダイの I/O 幅は 512 ビットにとどまる。また、熱抵抗も大きく、面接あたりの消費電力の大きな CPU や GPU での利用は現実的でなかった。

ハイブリッドボンディングが 2026 年時点、あるいは本調査研究開始の 2022 年時点で実用化されていたのは同一のロジックファブで製造したウェハ同士に限られていたが、主に台湾の DRAM ベンダが DRAM 自体のハイブリッドボンディングによる積層と、さらに自社ないし他社のロジックウェハとのハイブリッドボンディングによる積層の技術開発を進めており、2022 年頃には複数の DRAM ベンダが利用可能になってきていた。これらは、基本的には DRAM 自体にカスタム設計を適用し、アーキテクチャの要請に合わせたカスタム設計の DRAM とロジックダイを積層するものである。とはいえ、技術的には可能になっていたものの、まだ実際に採用したシステムが開発されるにはいたっていなかった。このことの大きな理由は、従来の GPU アーキテクチャ、すなわち物理的に主記憶を共有する並列プロセッサアーキテクチャでは、3 次元積層メモリを採用することのメリットがあまり大きくないことであると考えられる。階層キャッシュと物理的な共有メモリを採用する限り、DRAM がプロセッサダイの上 (あるいは下) にあったとしてもプロセッサから DRAM までのデータ移動距離や、キャッシュアクセスに伴う消費電力は大きく変わらず、消費電力の大きな減少やメモリバンド幅の大きな増加は実現できない。

ハイブリッドボンディングによる低い熱抵抗で、非常に多くのパッドを利用可能なダイ積層、分散メモリアーキテクチャのどちらも、それだけで大きな革新につながるものではなく、この 2 つを組み合わせることで初めて、GPU アーキテクチャを超える次世代のプロセッサが可能になる。このことを示し、またロジックダイと DRAM ダイの積層に伴う様々な技術的課題について検討を進めることができたことが、本調査研究の大きな意義であると考えている。

システムソフトウェア、コンパイラ、アプリケーションの観点でも、本調査研究で、分散メモリアーキテクチャでどのように実現するか、どのような性能が得られるかについての検討を進めることができた。分散メモリアーキテクチャを採用することの大きな意義は、極めて高いメモリバンド幅が現実的になることであり、B/F = 4 といった初代地球シミュレータ以前の古典ベクトルアーキテクチャ並みのメモリバンド幅を低電力で実現できる。このことは、チップ内ネットワークやプロ

セスコアアーキテクチャがアプリケーションの要請にあっていれば、古典ベクトルアーキテクチャ並みの、 50% を超えるような高い実行効率を多くのアプリケーションで実現できることが示された。また、人工知能応用でもっとも重要になる LLM でも、メモリバンド幅の限界で決まる性能に近いものを実現できることを、これは Preferred Networks で現在開発を進めている 3D 積層メモリを使ったプロセッサ MN-Core L1000 のシミュレーションで実証できた。

繰り返しになるが、本調査研究に関わった全ての人に感謝する。

外部発表一覧

発表者	題名	発表場所	日付
令和 4 年度			
牧野 淳一郎	ポストエクサスケールの計算機アーキテクチャ	第 16 回アクセラレーション技術発表討論会「高度計算科学の現状と未来」	2022/09/16
鯉渕 道紘	近似コンピューティングによる光演算の革新	日本光学会年次学術講演会	2022/11/14
牧野 淳一郎	パネルディスカッション「次世代 HPC 計算基盤構築に向けて夢を語ろう」ポジショントーク	第 22 回 PC クラスタシンポジウム 「HPC システム技術の最前線」	2022/12/06
牧野 淳一郎	富岳からポスト富岳へ—HPC システムの進化と計算基礎科学	「富岳で加速する素粒子・原子核・宇宙・惑星」シンポジウム	2022/12/13
牧野 淳一郎	研究報告 II システム研究調査の概要と検討状況	次世代計算基盤に係る調査研究に関する合同ワークショップ 〜 次世代高性能計算基盤の開発に向けて 〜	2023/02/22
令和 5 年度			
Fumiya Kono, Naohito Nakasato, Maho Nakata	Accelerating 128-bit Floating-Point Matrix Multiplication on FPGAs	2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)	2023/05/09
Naohito Nakasato	Evaluation of Various Arithmetic for Linear Algebra on GPU and FPGA	ICIAM 2023, Minisymposium “[01060] Exploring Arithmetic and Data Representation Beyond the Standard in HPC”	2023/08/21
牧野 淳一郎	シミュレーション天文学、専用計算機、汎用スパコン、深層学習	研究会「シミュレーション天文学のこれまでとこれから – ハードウェア・アプリケーション・サイエンス –」	2023/09/04

牧野 淳一郎	「次世代計算基盤に係る調査研究」システム調査研究・神戸大学「オリジナルアクセラレータアーキテクチャによる超低消費電力次世代計算基盤の評価」	第 17 回アクセラレーション技術発表討論会	2023/09/28
中里 直人、河野 郁也、中田 真秀	128-bit 浮動小数点演算による行列積とアプリケーションの性能評価	第 17 回アクセラレーション技術発表討論会「ポスト富岳アーキテクチャ」	2023/09/28
綱島 隆太	OpenACCfor MN-Core	CPS セミナー	2023/10/10
Michihiro Koibuchi	Approximate Interconnection Networks for Paralel Processing	13th international symposium on photonics and electronics convergence (ISPEC 2023) [招待講演]	2023/11/02
Michihiro Koibuchi	Photonic Approximate Communication Highlighting Ultimate Nature of Light	11th International Workshop on Computer Systems and Architectures, in conjunction with CANDAR2023 [招待講演]	2023/12/01
大淵 済	全球気象シミュレーションに関する考察—20 年前、10 年前、現在、そして、ポスト「富岳」に向けて—	京都大学防災研究所 災害気候研究分野セミナー	2023/12/12
牧野 淳一郎	研究報告 II システム研究調査の概要と検討状況	次世代計算基盤に係る調査研究に関する合同ワークショップ〜フィージビリティスタディ中間報告〜	2023/12/19
牧野 淳一郎	研究報告 II システム研究調査の概要と検討状況	「次世代計算基盤を利用した成果の最大化に向けて」に関する意見交換会	2024/01/15
N. Nakasato, Y.Murakami, F.Kono, M.Nakata	Evaluation of POSIT Arithmetic with Accelerators	”The International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2024)”	2024/01/27

令和 6 年度

Hisashi Yashiro, NICAM developers	Feasibility study for the next flagship supercomputer development and high-resolution climate modeling efforts in Japan	8th ENES HPC workshop on “High-resolution climate and weather modelling” , Lecce	2024/05/22
Hisashi Yashiro, Kazuya Yamazaki, Takashi Arakawa, Shuhei Matsugishi, Shereo Intyisyar, Kengo Nakajima	Efforts toward optimization of global non-hydrostatic atmospheric model on GPU supercomputer	JpGU2024, Makuhari	2024/05/29
Naohito Nakasato	Use of Local LLMs for Research Code Development	2024 InPEx (The International Post-Exascale Project) Workshop	2024/06/18
Hisashi Yashiro, NICAM developers	Development of a weather/climate model for Japan’s new flagship machine following Fugaku	Workshop on Global Storm-Resolving Analysis Bridging Atmospheric and Cloud Dynamics, Hakone	2024/06/19
Hisashi Yashiro, Kazuya Yamazaki, Takashi Arakawa, Shuhei Matsugishi, Shereo Intyisyar, Kengo Nakajima	Optimization, Translation, and Transformation of an Atmospheric Model to Utilize GPUs: A Case of NICAM	The 21st Annual Meeting of Asia Oceania Geosciences Society (AOGS2024), Pyeongchang	2024/06/26

鯉渕 道紘	光 × コンピューティングの研究動向	第84回OPT (Optical Packaging Technology) 公開研究会 [招待講演]	2024/07/26
綱島 隆太、遠藤 克浩、中里 直人、井町 宏人、牧野 淳一郎	ポスト富岳世代の MN-Core ベースアクセラレータ対応 OpenACC のインターフェイスとコンパイラの検討および開発	IPSJ 第 195 回 HPC 研究会 [情報処理学会研究報告 (Web), Vol.2024-HPC-195, No.1, 1-10.]	2024/08/08
Hisashi Yashiro, NICAM developers	Climate modeling at Exascale: Status, Challenges and Collaboration Opportunities	International Computing in the Atmospheric Sciences Symposium (iCAS2024), Stresa	2024/09/11
牧野 淳一郎	ポスト・エクサ、ポストムーア時代の HPC と AI	第 6 回アドバンス・シミュレーション・セミナー	2024/09/20
綱島 隆太	アクセラレータ用次世代プロセッサ MN-Core の動向の紹介（ポスト富岳調査の話を交えて）	EPnetFaN	2024/09/30
牧野 淳一郎	システム調査研究・神戸大学「オリジナルアクセラレータアーキテクチャによる超低消費電力次世代計算基盤の評価」結果報告	2024 年度「次世代計算基盤に係る調査研究」合同ワークショップ	2024/12/27
Hisashi Yashiro, NICAM developers	Development of Global Atmospheric Model NICAM towards Fugaku-NEXT era	HANAMI High-Level Symposium, Barcelona	2025/01/14

Ryuta Tsunashima, Katsuhiro Endo, Naohito Nakasato, Hiroto Imachi, Junichiro Makino	Development of OpenACCfor MN-Core: Part of the Post-Fugaku FS by Kobe University	第 7 回 R-CCS 国際シンポジウム	2025/01/23
大淵 済	HPC とスペクトル・モデル	福岡大学理学部地球圏科学科地球物理学グループセミナー	2025/02/18
中里 直人、河野 郁也、中田 真秀	様々な浮動小数点演算形式の評価プラットフォームとしての FPGA と GPU の比較	情報処理学会 第 193 回 HPC 研究会	2025/03/19
中里 直人	MN-Core アーキテクチャのためのプログラミングモデル	シンポジウム「ポスト富岳で拓くアプリの未来」	2025/03/25
藤井 通子	星一つ一つまで再現した銀河形成シミュレーションへ向けて	シンポジウム「ポスト富岳で拓くアプリの未来」	2025/03/25

事業参画者一覧

役職は参画当初のもの。

国立大学法人 神戸大学 (代表機関)		
牧野 淳一郎	教授	全体統括
林 祥介	教授	アプリケーションの検討と評価
大淵 済	特命教授	調査研究全体の運営と調整
斎藤 貴之	准教授	アプリケーションの検討と評価
高橋 芳幸	准教授	アプリケーションの検討と評価
永井 智哉	特命准教授	調査研究全体の運営と調整
樫村 博基	講師	アプリケーションの検討と評価
遠藤 克浩	特命助教	アーキテクチャ、システムソフトウェアの検討と評価
綱島 隆太	特命助教	システムソフトウェア、特にコンパイラの検討と評価
細野 七月	特命助教	アプリケーションの検討と評価
松嶋 俊樹	特命助教	アプリケーションの検討と評価
吉田 雄城	特命助教	アプリケーションの検討と評価
池部 美和子	事務補佐員	次世代計算基盤に係る調査研究補助
今村 リサ	事務補佐員	次世代計算基盤に係る調査研究補助
松本 史	事務補佐員	次世代計算基盤に係る調査研究補助
山口 良恵	事務補佐員	次世代計算基盤に係る調査研究補助
学校法人 順天堂 (分担機関)		
姫野 龍太郎	特任教授	アプリケーション調査研究の統括
学校法人 東洋大学 (分担機関)		
河合 浩志	教授	ものづくりアプリケーションの検討
塩谷 隆二	教授	ものづくりアプリケーションの検討
荻野 正雄	大同大学教授	ものづくりアプリケーションの検討
株式会社 Preferred Networks (分担機関)		
西川 徹	代表取締役社長	グループ統括
岡野原 大輔	代表取締役最高研究責任者	アーキテクチャ、システムソフトウェアの検討と評価
奥田 遼介	取締役最高技術責任者	アーキテクチャ、システムソフトウェアの検討と評価
田中 大輔	機械学習基盤担当 VP	アーキテクチャ、システムソフトウェアの検討と評価

土井 裕介	計算基盤担当 VP	アーキテクチャ、システムソフトウェアの検討と評価
前田 新一	研究担当 VP	アーキテクチャ、システムソフトウェアの検討と評価
牧野 淳一郎	VP/CTO of Processor Architecture	アーキテクチャ、システムソフトウェアの検討と評価
金子 紘也	EM	アーキテクチャ、システムソフトウェアの検討と評価
田形 健二	EM	アーキテクチャ、システムソフトウェアの検討と評価
平木 敬	シニアリサーチャー	アーキテクチャ、システムソフトウェアの検討と評価
浅井 大史	リサーチャー	アーキテクチャ、システムソフトウェアの検討と評価
三上 裕明	リサーチャー	アーキテクチャ、システムソフトウェアの検討と評価
井町 宏人	エンジニア	アーキテクチャ、システムソフトウェアの検討と評価
坂本 亮	エンジニア	アーキテクチャ、システムソフトウェアの検討と評価
高務 裕哲	エンジニア	アーキテクチャ、システムソフトウェアの検討と評価
中村 孝史	エンジニア	アーキテクチャ、システムソフトウェアの検討と評価
渡部 源太郎	エンジニア	アーキテクチャ、システムソフトウェアの検討と評価
菱田 快成	パートタイムエンジニア	アーキテクチャ、システムソフトウェアの検討と評価
坪内 美幸	プロジェクト管理	プロジェクト統括補佐
株式会社 Quemix (分担機関)		
松下 雄一郎	代表取締役 CEO (東京大学特任准教授)	マテリアルサイエンス応用アプリケーションの検討
岩田 潤一	プロダクトマネージャー/研究員	マテリアルサイエンス応用アプリケーションの検討
公立大学法人 会津大学 (分担機関)		
中里 直人	教授	システムソフトウェア・ライブラリ調査研究の統括およびコンパイラ検討
国立研究開発法人 海洋研究開発機構 (分担機関)		
堀 宗朗	部門長	地震と構造物シュミレーションアプリケーションの検討
古市 幹人	グループリーダー	地震と構造物シュミレーションアプリケーションの検討
国立研究開発法人 国立環境研究所 (分担機関)		
八代 尚	主任研究員	気象・気候シュミレーションアプリケーションの検討
国立研究開発法人 産業技術総合研究所 (分担機関)		
遠藤 克浩	研究員	アーキテクチャ、システムソフトウェアの検討と評価
国立大学法人 東京大学 (分担・協力機関)		

市村 強	教授	地震と構造物シュミレーションアプリケーションの検討 (協力)
鎌谷 洋一郎	教授	ゲノム科学アプリケーションの検討
塩谷 亮太	准教授	CPU 評価
藤井 通子	准教授	宇宙・惑星科学応用アプリケーションの検討
国立大学法人 東海国立大学機構 名古屋大学 (分担機関)		
白石 賢二	教授	マテリアルサイエンス応用アプリケーションの検討
押山 淳	特任教授	マテリアルサイエンス応用アプリケーションの検討
国立大学法人 名古屋工業大学 (分担機関)		
小泉 透	助教	アーキテクチャ、システムソフトウェアの検討と評価
国立大学法人 広島大学 (分担機関)		
石川 健一	准教授	素粒子・原子核物理応用アプリケーションの検討
大学共同利用機関法人 情報・システム研究機構 国立情報学研究所 (分担機関)		
鯉渕 道紘	教授	ネットワークアーキテクチャ評価
平澤 将一	特任助教	ネットワークアーキテクチャ評価
独立行政法人 国立高等専門学校機構 松江工業高等専門学校 (分担機関)		
岩澤 全規	講師	DSL 検討
国立大学法人 東北大学 (協力機関)		
平居 悠	学振特別研究員 (CPD)	宇宙・惑星科学応用アプリケーションの検討 (協力)
Leiden University (協力機関)		
Simon Portegies Zwart	Professor	宇宙・惑星科学応用アプリケーションの検討 (協力)
The University of Queensland (協力機関)		
Holger Baumgardt	Associate Professor	宇宙・惑星科学応用アプリケーションの検討 (協力)
Sun Yat-sen University (協力機関)		
Long Wang	Associate Professor	宇宙・惑星科学応用アプリケーションの検討 (協力)
Yale University (協力機関)		
David M. Hernandez	Postdoctoral Associate	宇宙・惑星科学応用アプリケーションの検討 (協力)

発行日

2026 年 3 月 24 日

発行責任者

牧野 淳一郎　国立大学法人 神戸大学