第 4 章

アプリケーション調査研究グループ

4.1 調査研究の概要および方針

4.1.1 AI for Science の潮流

近年、生成型 AI や大規模言語モデル（LLM）の急速な発展により、科学研究における AI 活用の重要性が飛躍的に高まっている。R-CCS の FS3.0 概要資料によれば、従来の数値シミュレーションに加え、データ駆動型の解析や自動化、自律的な研究支援といった AI for Science に基づくアプリケーションが重要になっている。これに対応するため、HPC と AI を統合した計算基盤を構築し、革新的なアプリケーションの開発・利用スタイルに適応することが求められている。

4.1.2 アプリケーション調査研究の目的

FS3.0 では、生成 AI や自律化を取り込んだ科学アプリケーションのトレンドを把握し、次世代 HPCI システムに必要な機能・性能要件を明確化することを目指している。申請書に記載されたミッションは以下の通りである。

1. **Japan Scientist AI Jam による先端事例の調査** TRIP-AGIS が主催する Japan Scientist AI Jam に参加し、OpenAI、Anthropic、Google の協力を得ながら最先端の AI 技術を科学分野でどのように活用できるかを検証する。特に大規模言語モデルや生成 AI の活用方法を研究し、理化学研究所計算科学研究センターの HPC/AI 支援部門の最新知見を反映する。
2. **HPC+AI 活用課題の中間整理** 対象となるアプリケーション群を調査し、HPC と AI が混在するワークフローの課題を抽出・整理する。特に、数値シミュレーションと AI の後処理・解析型（ポスト処理）、シミュレーションと AI が相互作用する in situ 型、AI 学習から HPC 検証に逆方向へフィードバックする型、デジタルツインやストリーミング型の四つのパターンを想定して、それぞれの機能要件を洗い出す。
3. **ベンチマーク整備と性能評価** 抽出した代表ワークロードについてベンチマークコードを整備し、現行システムや試作システムで性能評価を行う。評価結果を基に、次世代 HPC システムに求められる目標性能を定量化する。
4. **他グループとの協議と実装方針の策定** 抽出課題と必要要件をアーキテクチャ検討グループやシステムソフトウェア検討グループと共有し、共通ベンチマーク環境の構築や CI/CD/CB 連携による持続的な

性能改善を提案する。

4.1.3 国内外の動向と関連研究

4.1.3.1 HPC+AI 統合プラットフォーム

次期フラッグシップ「富岳 NEXT」は、Fujitsu の MONAKA-X CPU と NVIDIA GPU を NVLink Fusion で緊密に接続し、シミュレーションと AI の双方で高い性能を達成することを目指している。また、MONAKA-X に行列計算エンジン Arm SME を統合し、GPU と 高帯域・低遅延で密結合した構成により AI 推論の低遅延処理と HPC アプリケーションの高速化を両立させる。このような CPU-GPU の密結合は、AI 推論をシミュレーションに組み込む in situ ワークフローに不可欠である。

米国では DOE の AI for Science 計画に基づき、RHAPSODY などのマルチランタイムミドルウェアが提案されている。RHAPSODY は、MPI ベースのシミュレーション、長寿命の AI サービス、細粒度タスクを同時に実行できる仕組みを提供し、低遅延の AI-HPC カップリングと異種タスクの共存を実現している。同論文では、科学的ワークフローが simulation + training + inference を同時に扱うようになり、ランタイムに長時間の MPI プロセス、高スループット推論サービス、細粒度の AI タスクなどの極めて複雑な要求が課されている。このようなミドルウェア事例は、FS3.0 におけるワークフロー管理の参考になる。

4.1.3.2 デジタルツインとストリーミング

デジタルツインは、物理システムの状態や挙動をリアルタイムに追跡し、仮想空間上で再現することで最適制御や予測を行う技術である。米国バークレー研究所は、「デジタルツインが実世界のシステムをモデル化し、測定データと AI・シミュレーションを用いてリアルタイムの意思決定を支援する」と説明している。デジタルツインは物理システムからのストリーミングデータと連動するため、HPC システムにはデータストリーミング専用 NIC や計算と独立したデータ受信経路、耐障害性のあるノード構成が求められる。また、Argonne 国立研究所は、HPC 環境に適したイベントストリーミングフレームワーク Mofka を開発し、RDMA 対応ネットワークや大規模生産者-消費者ワークロードに最適化した persistent streaming を提供している。こうした研究は、デジタルツイン型ワークフローに対応するためのシステム設計に重要である。

4.1.3.3 NVMe バーストバッファとノードローカル SSD

計算ノードに直接接続された NVMe SSD（バーストバッファ）は、シミュレーションデータを高速に保存・取り出すために利用される。米国 Oak Ridge 国立研究所の Frontier では各ノードに 1.92 TB の NVMe デバイスを 2 基搭載しており、ピーク読み取り 5.5 GB/s、書き込み 2 GB/s の性能を持つノードローカル SSD を「バーストバッファ」として利用できると説明されている。ユーザはジョブ実行時に `-C nvme` オプションを指定することでバーストバッファを使用し、データを `/mnt/bb/<userid>` に配置できる。ノードローカル SSD は、ポスト処理型ワークフローで大量のデータを高速に書き出す際に有効だが、共有ファイル I/O を支援するためには SCR ライブラリなどの中間ソフトウェアが必要である。

4.1.3.4 メタデータ管理とストレージ

データ駆動型アプリケーションでは、膨大な数のファイルやオブジェクトを扱うため、メタデータ管理の高速化が欠かせない。米国 Lawrence Berkeley 国立研究所の SoMeta プロジェクトでは、オブジェクト中心の

メタデータ管理システムを開発し、約 27.7 万個のデータオブジェクト検索において従来の Lustre ファイルシステムより 15〜40 倍高速、SciDB より 10〜90 倍高速なメタデータ検索を実現したと報告している。また、Hammerspace などの商用データプラットフォームは、GPU/CPU サーバに搭載されたローカル NVMe をグローバルファイルシステムに統合し、高速読書き性能と高速メタデータ操作、効率的な小ファイル更新を提供することが HPC に不可欠だと指摘している。

4.1.3.5 スケジューラと小規模ジョブへの対応

AI 学習や推論タスクでは多数の短時間ジョブやインタラクティブセッションが発生する。Argonne 国立研究所の報告によると、実験データの即時解析や AI による意思決定を伴うワークロードでは、短命なジョブや持続的なインタラクティブセッションを大量に起動する必要があり、従来の HPC スケジューラは長時間ジョブのスループット最適化を優先するため対応が難しいことが指摘されている。一部のセンターではデバッグパーティションやプリエンプト可能なキューなどを導入しているが、近未来の AI 活用にはより柔軟で低遅延なスケジューリング機構が求められる。

4.2 Japan Scientist AI Jam Session を踏まえた HPCI 整備計画に関する調査

4.2.1 Japan Scientist AI Jam Session

4.2.1.1 イベントの概要

Japan Scientist AI Jam Session は、理化学研究所の TRIP 事業本部・AGIS 事務局が中心となって実施した、科学者と Frontier AI パートナーが協働しながら、最新 AI を研究課題の探索と解決にどう生かせるかを実践的に検証するプログラムである。研究者が最先端 AI ツールを使って研究を加速する方法を体験すると同時に、その協働過程から得られた知見を AI 開発側へのフィードバックとして還元する場と位置づけられている。構成は、11 月 11 日のオンラインチュートリアル、11 月 28 日のドライラン、12 月 16 日の東京本番セッション、12 月 18 日の神戸本番セッションの 4 段階で、オンラインでの基礎習得から会場での実践までを段階的に設計している。テーマはオンラインチュートリアルを踏まえて参加者が各自で設定し、対象者は研究者に加えて研究マネジメント・研究コミュニケーション人材、大学院生、URA にも開かれている。公式ページでは AWS、Anthropic、Google、OpenAI の最先端モデルへのアクセスと企業スタッフによる現地支援が案内され、添付資料では OpenAI、Anthropic、Google、AWS、NVIDIA のモデル提供に加え、Discord による Q&A、Chrome browser extension を用いた collection tool、RIKEN open model platform + MCP server の利用も示されている。各セッションの参加者数は、オンラインチュートリアル 225 人、ドライラン 50 人、東京 124 人、神戸 95 人であった。

4.2.1.2 事後のアンケート結果

事後アンケートは 134 件の回答に基づいており、参加者の所属は研究機関が 64.2% と中心で、大学が 24.6% と続いた。研究分野はライフサイエンスが 46.3% で最も多く、次いで情報・AI が 20.1% で、物理、材料・工学、医学・薬学など、複数分野から参加があった。取り組んだ課題はコード生成が 60.4% で最も多く、文献調査 38.1%、手法分析 29.9%、実験計画 26.1%、データ解釈 24.6% が続いており、AI が発想支援だけでなく、研究実務の具体的な作業にも広く利用されていたことがうかがえる。利用モデルは OpenAI 85.8%、Google

75.4%、Anthropic 73.9% が上位で、複数モデルを比較しながら活用する参加者が多かった。

満足度と効果に関する結果は、全体として前向きである。事前案内の分かりやすさは 5 段階中「3」が 41.0% で最も多く、課題の準備状況も「3」が 34.3% で最多となっており、事前準備段階の評価は中程度にとどまった。オンラインチュートリアルの満足度も「3」が 43.3% で最も多いが、「4」「5」を合わせると 39.6% となり、大きな不満が支配的だったわけではない。一方、本番セッションの運営満足度は「4」「5」が計 70.9% に達し、今回の体験が今後の研究に影響すると答えた人は「4」「5」で 85.1%、次回も参加したいという回答は「4」「5」で 89.6% に達した。つまり、事前案内や準備段階の評価は中庸だった一方で、本番での体験価値や今後への期待は非常に高かったといえる。自由記述でも、Frontier AI 企業による現地支援、最新モデルを比較できた点、高性能モデルを無償で利用できた点、他参加者との対話から学べた点などが好意的に挙げられている。

その一方で、改善点も明確である。AI ツール利用時の主な障壁として最も多かったのは「自分自身の知識不足」58.2% で、次いで「時間が短すぎた」47.8%、「チュートリアル資料が不十分」33.6% であった。モデル性能そのもの 17.2% やドキュメント不足 17.9% よりも、学習支援や導入支援の不足が大きな課題として認識されていたことが分かる。また、研究で AI を使う際の懸念としては、「誤用・誤解釈」60.4%、「プライバシー」56.7%、「バイアス」41.8%、「再現性」37.3%、「説明可能性」26.1% が上位に並んでおり、利便性への期待と同時に、研究倫理や研究品質への慎重な意識も強い。否定的な自由記述では、AI Jam の時間が短いこと、自由形式よりも一定のガイドがほしいこと、イベントの目的を事前により明確に示してほしいこと、チュートリアルの実演例を増やしてほしいこと、アカウント設定の案内をもっと早くしてほしいことなどが挙げられている。

以上から、Japan Scientist AI Jam Session は、研究者に最先端 AI を紹介するだけのイベントではなく、実際の研究課題に AI を適用し、その有効性、運営上の課題、教育支援の必要性、倫理的論点までを同時に可視化する実践的な場として機能したとまとめられる。資料末尾では、Claude Skills を用いたラボオートメーション支援の例も紹介されており、AI 活用の対象が文献調査やコード生成にとどまらず、実験プロトコルや研究設備の運用支援へ広がりうることも示唆されている。今後は、事前準備や伴走支援をさらに充実させることで、参加者の高い満足度と継続参加意向を、研究現場での本格的な実装につなげていくことが期待される。

4.2.2 アプリケーションの分類と必要要件

本調査では、HPC と AI が混在するアプリケーションを以下の四つのスタイルに分類し、それぞれの課題と必要要件を整理した。

4.2.2.1 シミュレーション → AI（後処理・解析）型

数値シミュレーションの結果を保存し、後処理として AI による解析や可視化を行うスタイルである。例えば、流体シミュレーションの大規模データをニューラルネットワークで特徴抽出し、異常検知やパラメータ最適化に利用する場合などが該当する。大量の出力データを高速に書き出し、AI 処理の入力として読み出す必要がある。要件と課題は以下の通りである。

1. **バーストバッファ・ノードローカル SSD** シミュレーションの出力をいったんノードローカル SSD にバッファリングし、後処理フェーズで順次 AI に読み込ませる仕組みが重要である。Frontier のバーストバッファは各ノード 1.92 TB × 2 で読み取り 5.5 GB/s、書き込み 2 GB/s の性能を持ち、ユーザーが ‘/mnt/bb‘ 経由で直接利用できる。ノードローカル SSD の利用によりピーク I/O 性能が線形に向

上し、I/O ばらつきが小さい利点があるが、共有ファイル I/O が難しいという欠点がある。

2. **多対少通信パターンへの対応** シミュレーションノードが多数のデータを I/O ノードや AI サーバへ送る場合、通信パターンが many-to-few となり、ネットワーク輻輳を引き起こす。バーストバッファによる分散書き込みや通信パターンの最適化が課題である。
3. **大容量ファイルの効率的な移動** バーストバッファに保存したデータを並列ファイルシステムに非同期で書き戻す仕組みが必要である。SCR ライブラリやインメモリファイルシステムなどのミドルウェアによりチェックポイントの分散管理を行う。
4. **AI 推論サービスとの連携** AI の後処理は高いメモリ帯域を必要とし、推論モデルのホスト GPU までのデータ転送効率が全体の性能を左右する。大規模言語モデル推論では 70 B 〜 100 B パラメータ級モデルで 50 ms 以下の応答時間を求めるケースがあり、バッチ処理や GPU メモリ内での効率的なデータ準備が重要である。

4.2.2.2 シミュレーション　AI（in situ / in loop）型

シミュレーションと AI モデルがリアルタイムに相互作用し、AI がシミュレーションを制御したり、シミュレーション結果を即座に AI が解析するスタイルである。例えば、分子動力学計算中に生成 AI がパラメータを予測し、シミュレーションを方向付ける AI-in-HPC や、シミュレーション結果を AI が即時に評価してフィードバックする AI-out-HPC が該当する。要件と課題を以下にまとめる。

1. **CPU – GPU 間の低レイテンシ通信** In situ 処理ではシミュレーションと AI 推論が同一ノード上で並行実行されるため、CPU と GPU の通信遅延を極小化する必要がある。富岳 NEXT 計画では、MONAKA-X CPU と GPU を高帯域・低遅延で密結合する NVLink Fusion を検討し、AI 推論を組み込んだ HPC アプリケーションの性能向上を目指している。
2. **細粒度同期が可能なランタイム** AI 推論を含む多数のタスクがシミュレーションに割り込むため、タスク依存関係を動的に解決しながら非同期実行できるランタイムが必要である。RHAPSODY [15] は MPI 実行、長時間実行 AI サービス、細粒度タスクを統合するマルチランタイムであり、低遅延の AI-HPC カップリングと高スループット推論を実現している。
3. **リアルタイム性を保証するノード配置** シミュレーションと AI を同ノードまたは隣接ノードに配置し、通信遅延を最小化するためのノード配置アルゴリズムが必要である。高精度シミュレーションと高速推論を重ね合わせることで、AI エージェントがリアルタイムにパラメータ探索や制御を実現する。
4. **I/O 回避とデータストリーミング** In situ 型ではファイルシステムへの書き出しを避け、メモリ内のデータを直接 AI モデルに渡す必要がある。イベントストリーミングフレームワークは RDMA 対応ネットワークや多リンク構成で大容量データを持続的にストリーミングする仕組みを提供しており、HPC 環境でのストリーミング処理に活用できる。

4.2.2.3 AI 学習 → HPC 検証（逆方向）型

先に AI モデルを学習させ、その結果を基に HPC シミュレーションで検証や詳細解析を行うスタイルである。たとえば大規模言語モデルに科学知識を学習させ、その推論結果を基に数値シミュレーションで現象の検証を行うケースがこれに当たる。要件と課題は以下の通りである。

1. **ワークフロー制御ノード** 大量の推論リクエストを管理し、結果に応じてシミュレーションジョブを起

動するための制御ノードが必要である。分析プラットフォームと HPC ジョブ管理を連携させ、AI モデルの出力に応じたシミュレーションを自動投入する。

2. **小規模・短時間ジョブに強いスケジューラ** AI 学習や推論では短いジョブが多数発生するが、従来の HPC スケジューラは長時間ジョブのスループットを最適化するため小規模ジョブがスケジュールされにくい。Argonne NL の報告では、リアルタイム実験データ解析や AI ステアリングを含むアプリケーションでは短いジョブやインタラクティブセッションが大量に必要となり、従来のスケジューリングポリシーでは応答性と柔軟性が不足すると指摘している。デバッグパーティションやプリエンプティブキューの導入が進んでいるが、より柔軟なスケジューリングが不可欠である。
3. **メタデータ高速管理** AI 学習では大量の小ファイル（データセットや重み）を素早く取り扱う必要がある。SoMeta のような分散メモリ型メタデータ管理システムは、数十万オブジェクトのメタデータ検索で Lustre の 15〜40 倍の性能を示しており、高速メタデータ処理のためのアーキテクチャやソフトウェアが重要である。

4.2.2.4 デジタルツイン・ストリーミング型

物理システムの観測データをストリーミング受信し、リアルタイムに仮想モデル（デジタルツイン）で再現するスタイルである。現実世界の機器やセンサーから送られる連続データを HPC システムが受信し、AI モデルやシミュレーションで解析して制御信号を返す。要件と課題は以下の通りである。

1. **データストリーミング専用 NIC と独立した受信経路** デジタルツインでは計算とデータ受信を並行するため、計算ノードとは独立したネットワーク経路とストリーミング専用 NIC が必要である。バーストバッファと同様、ノード故障時にもデータ受信を継続できる耐障害性が求められる。
2. **耐障害性のあるノード構成** 物理システム側のセンサーやネットワークに障害が起きてもサービスを継続するため、冗長ノードやフェイルオーバー機構を備えた構成が必要となる。
3. **リアルタイム AI 推論と高頻度制御** デジタルツインは物理システムに対するフィードバックを提供するため、推論レイテンシをミリ秒オーダーに抑え、大量のストリーミングデータに対して高頻度で推論を実行できるシステム構成が求められる。

4.2.3 目標性能の定量化

調査の一環として、FS3.0 では代表的なワークロードに基づき次世代 HPC の目標性能を定量化した。本報告書では以下の目標値を提案する。

これらの指標は、AI 推論と HPC シミュレーションを組み合わせたワークロードが持つリアルタイム性と大量並列性を踏まえて設定したものである。例えば、70 B パラメータ級の大規模言語モデルのインタラクティブ推論では 50 ms 以下の応答が望まれ、vLLM のような GPU 内バッチ準備技術が必要である。また、ベクトル検索と推論を組み合わせた RAG 型エージェントでは、検索レイテンシを 10 ms 以下に抑えつつ数百万のオブジェクトを扱うメタデータ操作が求められる。

4.2.4 調査結果に基づく提言

4.2.4.1 アーキテクチャ設計

- **CPU - GPU 密結合とスケーラブルなノード設計** 富岳 NEXT のような高帯域・低遅延な CPU - GPU 結合（NVLink Fusion）を採用し、AI 推論を含む in situ ワークフローの遅延を削減する。
- **ノードローカル NVMe やバーストバッファの強化** 各ノードに高速 NVMe バーストバッファを搭載し、シミュレーション出力のバッファリングや AI モデルのチェックポイント保存に利用する。ノードローカル SSD をグローバルファイルシステムに統合し、チェックポイントやホットデータを共有する仕組みを整備する。
- **高速メタデータ管理の採用** SoMeta のようなオブジェクト中心のメタデータ管理システムを導入し、AI 学習データセットや実験データの探索を高速化する。分散ハッシュテーブルやブルームフィルタを利用することで膨大なファイル数にも対応できる。
- **デジタルツイン用ストリーミングインフラ** RDMA 対応のイベントストリーミングフレームワーク（Mofka [16] など）を組み込み、データストリーミング専用 NIC を備えたノードを設置してリアルタイムデータを受信する。耐障害性のある冗長構成とフェイルオーバー機構を整備する。

4.2.4.2 システムソフトウェア・ランタイム

- **マルチランタイム環境の整備** RHAPSODY のように MPI 実行、AI サービス、細粒度タスクを同時に扱うマルチランタイムを活用し、in situ ワークフローや agentic ワークフローを効率的に実行する。
- **スケジューラの改善と小規模ジョブ対応** 従来のスループット重視型から、短時間ジョブやインタラクティブセッションを迅速にスケジュールできるポリシーへ移行する。Argonne NL の調査で指摘されたように、AI 推論や実験データ解析では短命なジョブを大量に起動する必要があり、現行スケジューラでは応答性が不足している。デバッグパーティションやプリエンプティブキューの拡充に加え、ジョブ依存関係と優先度を動的に管理する機能を持つ次世代スケジューラを開発する。
- **CI/CD/CB と性能評価基盤** 抽出したベンチマークコードを用いて継続的な性能測定を行い、CI/CD/CB（継続的インテグレーション／デプロイメント／ベンチマーキング）パイプラインを構築

表 4.2.1 代表的な AI+HPC のワークロードと目標値

ワークロード	主要指標	目標性能
	推論レイテンシ（70B〜100B モデル相当）	$\leq$ 50 ms/推論
HPC 制御型 AI エージェント	同時エージェント数	$\geq$ 10 000 並列エージェント
	エージェント介入による HPC 制御遅延	$\leq$ 5%
	推論頻度	$\geq$ 1 000 推論/秒/ノード
in situ AI 推論	計算とのオーバーラップ率	$>$ 90%
	1 GPU あたり同時常駐コンテキスト数	4–8 個
	ベクトル検索レイテンシ	$<$ 10 ms/クエリ
RAG 型 AI エージェント	メタデータ操作性能	$\geq 10^9$ objects, $\geq 10^6$ ops/s
	AI 推論＋検索の合成遅延	$<$ 100 ms

する。ユーザがコードの変更を即座に評価し、次期システムの設計にフィードバックできる環境を整備する。

4.2.4.3 アプリケーションとワークフロー

- **アプリケーション共同設計** ユーザ・アプリケーション開発者とアーキテクトが早期から協調し、HPC システムの機能を最大限に引き出すアプリケーション開発手法を推進する。特に AI モデルのオンプレミス学習や推論に対応したワークロード管理、データ管理を計画段階から統合する。
- **代表ワークロードの整備** 各スタイルに対応する代表アプリケーションを選定し、ベンチマークコードや自動化されたワークフローを整備する。例えば、流体力学シミュレーション＋ AI 可視化、分子動力学＋生成モデル制御、科学知識ベース＋ RAG 推論＋ HPC 検証、デジタルツインによるリアルタイム制御などが候補となる。ベンチマークを通じて性能要求を評価し、次期システムの設計指針を得る。
- **安全性と機密性の確保** 国産大規模言語モデル（Swallow など）をローカル環境で利用する可能性を検討し、機密性の高い研究データに対応できる支援体制を整備する。データ主権を確保しつつ、外部クラウドサービスとの連携や AI モデルの更新を安全に行える環境を提供する。

4.2.5 次年度の調査研究の計画

生成 AI の性能向上に伴い、より多くのアプリケーションが生成 AI の推論に依存するようになることが予想される。また、推論においても MoE や KV Cache、Linear Attention、4 次元並列の次元配分などにより、演算性能・メモリ帯域・通信帯域・レイテンシに関する要件が大きく変わる。つまり、モデルアーキテクチャの変遷とその高性能実装手法に関する情報を常に最新になるように更新し続けることが重要である。また、実際のワークロードを計測する意味でもこれらの最新のモデルの最高性能の実装を開発し続けることを目的とする。具体的には、事前学習・SFT・強化学習・推論の４つのステージを Megatron-LM、vLLM で実装しながらその演算性能・メモリ帯域・通信帯域・レイテンシに関する要件を調査する。

4.3 次世代アプリケーションニーズ調査研究サブグループ

4.3.1 調査研究の目的

次世代 HPC アプリケーション開発に関する調査方法を検討し、分担機関との役割分担、評価観点の整理、調査対象分野の優先順位付け、次世代 HPC・AI 開発支援拠点形成事業との連携体制構築など、実施体制の整備と分野横断的な連携枠組みの構築を重点的に行う。その後、次世代 HPC 環境におけるアプリケーションの開発手法について調査を開始する。また、HPCI の幅広いアプリケーションエリアからの要望・要件を調査するための体制を確立する。

4.3.2 調査研究の結果

次世代 HPC・AI 研究開発支援センター（HAIRDESC）との連携体制（特に中核拠点の一つである東京大学）を構築し、HAIRDESC が支援する幅広い分野のアプリケーションコードの GPU 化支援事例を本サブグループによるニーズ調査に活用するための準備を進めた。

幅広いコミュニティにおいて開発されてきたコードを次世代システム向けに移植していくためにはコミュニティごとに多様な要請があると想定される。そこで、天文コミュニティにおける代表的なアプリケーションであるとともに演算律速なアプリケーションの代表例でもある重力多体計算を題材としてCUDA/HIP/SYCL/Kokkos/OpenACC/OpenMP target という多様な開発環境を駆使した GPU 実装・最適化の比較検証（4.3.2.1 節）を行うことで、単なるニーズの吸い上げだけではなく各種ニーズに応じた情報提供を行うための素材作成にも取り組んだ。

具体的な情報収集としては、第 255 回 ARC・第 202 回 HPC 合同研究発表会や SCA/HPCAsia 2026 といった HPC コミュニティの研究会、「富岳成果創出加速プログラム」基礎科学合同シンポジウム 2025 や日本天文学会 2026 年春季年会といった基礎科学コミュニティの研究会に参加し、HPC コミュニティにおける動向および基礎科学コミュニティにおけるコード開発動向を調査した。4.3.2.1 節における成果は、「富岳成果創出加速プログラム」基礎科学合同シンポジウム 2025 や日本天文学会 2026 年春季年会、第 203 回 HPC・第 17 回 QS 合同研究発表会などで発表し、幅広い分野に向けての成果還元を行った。

加えて、生成 AI ツールを用いてのコード開発支援についても試み（4.3.2.2 節で紹介）、また次世代アプリケーション開発手法調査研究サブグループとの情報交換を行った。

以下では、特に GPU 向けの多様なプログラミング手法に関する調査研究（4.3.2.1 節）について詳細に報告し、また生成 AI ツールを用いてのコード開発支援試行（4.3.2.2 節）についても詳細を報告する。

4.3.2.1 GPU 向けの多様なプログラミング手法に関する調査研究

■はじめに 富岳の後継機である「富岳 NEXT」への演算加速器搭載が決定しており、NVIDIA 社が GPU 設計を担当することが発表された[2]。今までの NFS（National Flagship System）においては GPU 搭載が見送られてきたこともあり、国内コミュニティによって開発されてきた科学技術計算コードの GPU 対応は遅れており、この先急ピッチで GPU 移植・最適化を進めていく必要がある。その一方でこれまで NIS（National Infrastructure System）として導入・運用されてきたスパコンには GPU 搭載システムが多数含まれており、特に近年では GPU 搭載事例が増加している。

2024 年度までに導入された国内の GPU スパコンはすべて NVIDIA 製 GPU を搭載したものであったが、2025 年度には量子科学技術研究開発機構（QST）および核融合科学研究所（NIFS）のプラズマシミュレータ “双星” には AMD MI300A を全 280 基搭載するサブシステム B を導入、筑波大学計算科学研究センターの Sirius にも AMD MI300A が全 96 基搭載されるなど、AMD 製 GPU を搭載したシステムの導入も始まっている。したがって GPU ベンダーに依らないコード開発の必要性も高まっており、特にこれからはじめて GPU 移植に着手する研究者がはじめからベンダーニュートラルな GPU コードを開発できれば、コード移植回数自体を削減できる可能性もある。

GPU 向けコード開発環境としては、NVIDIA 製 GPU のみを対象とする CUDA、NVIDIA 製 GPU と AMD 製 GPU 両方に対応できる HIP、より多様なデバイスに対応できる SYCL といった低レベル言語、OpenACC [17] や OpenMP target [18] といった指示文ベースの手法があり、主要な開発環境および NVIDIA/AMD/Intel GPU への対応状況については [19] にまとめられている。また、複数のプログラミングモデルを統合的に扱える環境としては、低レベル言語や OpenMP などをバックエンドとする Kokkos [20, 21] や GPU 向け指示文を簡易に切り換えられる Solomon [22] などがあげられる。

本研究では今後の GPU 移植に向けた知見を得るため、直接法に基づく N 体計算コードを、CUD-

[2] https://www.riken.jp/pr/news/2025/20250822_1/index.html

表 4.3.1　GPU の性能比較。
†1 AMD 製 GPU については Vector 性能。
†2 NVIDIA 製 GPU については GPU Boost Clock for FP32 non-Tensor Core Ops、AMD 製 GPU については Boost Clock。
†3 パッケージ全体の値。GPU 部分に限れば 700 W。
†4 水冷モデルの場合。550 W の製品もある。

	NVIDIA GH200	NVIDIA B200	AMD MI300A
FP32 理論ピーク性能 †1	66.91 TFlop/s	74.45 TFlop/s	122.6 TFlop/s
FP32 コア数	16896 (132×128)	18944 (148×128)	14592 (228×64)
動作周波数 †2	1980 MHz	1965 MHz	2100 MHz
グローバルメモリ容量	HBM3 96 GB	HBM3e 192 GB	HBM3 128 GB
メモリバンド幅	4.0 TB/s	7.7 TB/s	5.3 TB/s
TDP	1000 W†3	1000 W	760 W†4
プロセスルール	TSMC 4N	TSMC 4NP	TSMC 5N

A/HIP/SYCL に加えて Kokkos と Solomon を用いて実装・最適化し、NVIDIA GH200/B200 および AMD MI300A 上での性能を比較した。直接法に基づく N 体計算の演算カーネルは

$$\frac{\mathrm{d}^2 \boldsymbol{r}_i}{\mathrm{d}t^2} = \sum_{j=0, j\neq i}^{N-1} \frac{G m_j (\boldsymbol{r}_j - \boldsymbol{r}_i)}{\left(|\boldsymbol{r}_j - \boldsymbol{r}_i|^2 + \epsilon^2 \right)^{3/2}} \tag{4.1}$$

と表される。ここで $\boldsymbol{r}_i$，m_i はそれぞれ N 個の粒子中の i 番目の粒子の位置と質量、G は重力定数、ϵ は重力ソフトニング長であり、ここでは Plummer softening を採用した。直接法においてはこの重力計算の計算量は $\mathcal{O}(N^2)$ となり、演算律速なアプリケーションの代表例として知られている。また、銀河や宇宙の大規模構造といった無衝突系のシミュレーションにおいては単精度浮動小数点演算で十分な精度が得られるため、本研究でも単精度浮動小数点演算を用いる。

本研究において性能評価対象とした GPU は NVIDIA GH200、NVIDIA B200、AMD MI300A であり、その性能や仕様は表 4.3.1 に示したとおりである。すべて 2024 年に国内で稼動を開始した GPU スパコンに搭載されているものであり、NVIDIA GH200 は最先端共同 HPC 基盤施設（JCAHPC）が運用する Miyabi の演算加速ノード Miyabi-G に、NVIDIA B200 は東京工科大学 AI テクノロジーセンターが運用する青嵐に、AMD MI300A は QST/NIFS が運用するプラズマシミュレータ “双星” のサブシステム B に搭載されている。JCAHPC が運用する Miyabi は HPCI に資源提供されているものの、青嵐およびプラズマシミュレータ “双星” は HPCI には資源提供されていない。2026 年度には筑波大学計算科学研究センターが運用する Sirius（AMD MI300A を搭載）および名古屋大学情報基盤センターが運用する「不老・弐」（Type II サブシステムに NVIDIA GB200 を搭載）が HPCI に資源提供する予定であり、本研究における AMD MI300A や NVIDIA B200 の性能評価はこれらのシステムにおける性能の参考値にもなる。

以下、比較対象とした GPU 向けプログラミング手法の概要および適用した最適化手法をまとめ、各実装の性能評価結果を報告する。最後に GPU プログラミング手法の性能比較結果をまとめる。

■各実装におけるパラメータ調整　各プログラミングモデルにおける実装においては、最終的な性能最適化として実装手法の選択や、性能に大きな影響を与えるパラメータの決定が必須である。本研究では、粒子数

$N = 4\,194\,304$ においてパラメータ探査を行い、最も高性能となるパラメータを全粒子数に対して適用する方針を採用した。

NVIDIA GH200 上の最終実装　CUDA C++ 実装においては、`memcpy_async()` 命令を用いた実装においてブロックあたりのスレッド数を 512、ループアンローリング段数を 16、命令レベルの並列性を 4 に設定した実装が最も高性能であり、`nvcc` のコンパイルオプションとして`-O3 -ftz=true -gencode arch=compute_90,code=sm_90` を指定した。

SYCL 実装においては、シェアードメモリをダブルバッファ的に利用する実装においてブロックあたりのスレッド数を 512、ループアンローリング段数を 32、命令レベルの並列性を 8 に設定した実装が最も高性能であり、`acpp` のコンパイルオプションとして`-O3 -ffast-math -fgpu-flush-denormals-to-zero --acpp-targets=cuda:sm_90` を指定した。

Kokkos 実装においては、`Kokkos::TeamPolicy` を用いたシェアードメモリ実装に raw pointer を渡し、チームあたりのスレッド数を 128、ループアンローリング段数を 32、命令レベルの並列性を 4 に設定した実装が最も高性能であり、CMake を通じて CUDA C++ や SYCL と同等のコンパイルオプションが指定されるよう設定した。

指示文実装においてはコンパイラに示唆するブロックあたりのスレッド数のみが最適化パラメータであり、OpenACC の `kernels` 構文においては 64、`parallel` 構文においては 128、OpenMP target の `loop` 指示文においては 64、`distribute` 指示文においては 1024 と指定した。`nvc++` のコンパイルオプションについては`-O3 -Mfprelaxed=rsqrt -gpu=cc90,fastmath` を指定し、その他に OpenACC あるいは OpenMP target での GPU オフローディングに必要な指定を行った。

NVIDIA B200 上の最終実装　Kokkos 実装においては、`Kokkos::TeamPolicy` を用いたシェアードメモリ実装に raw pointer を渡し、チームあたりのスレッド数を 256、ループアンローリング段数を 32、命令レベルの並列性を 4 に設定した実装が最も高性能であり、CMake については Blackwell 世代の指定だけが NVIDIA GH200 環境との差分である。

AMD MI300A 上の最終実装　HIP の packed FP32 命令を用いた実装においてはシェアードメモリを用いない実装の方が高速であり、ブロックあたりのスレッド数を 256、命令レベルの並列性を 4 に設定した実装が最も高性能であり、`hipcc` のコンパイルオプションとして`-O3 --offload-arch=gfx942` を指定した。Packed FP32 命令を用いない実装においては、シェアードメモリを用い、ブロックあたりのスレッド数を 1024、ループアンローリング段数を 1024 として命令レベルの並列性を設定しない実装が最高性能であった。コンパイルオプションについては、さらに`-ffast-math -fgpu-flush-denormals-to-zero` を付与した性能測定も行ったが、性能向上は見られなかった。

SYCL 実装において AdaptiveCpp を用いてコンパイルした際には、packed FP32 命令を用いた実装では HIP 実装と同設定、packed FP32 命令を用いない実装ではシェアードメモリをダブルバッファ的に利用する実装においてブロックあたりのスレッド数を 512、ループアンローリング段数を 64 として命令レベルの並列性を設定しない実装が最高性能であった。`acpp` のコンパイルオプションとしては packed FP32 命令を用いる場合には`-O3 --acpp-targets=hip:gfx942` を、packed FP32 命令を用いない場合には`-O3 -ffast-math -fgpu-flush-denormals-to-zero --acpp-targets=hip:gfx942` を指定した。

SYCL 実装において Intel oneAPI を用いてコンパイルした際には、シェアードメモリを用いずにブロックあたりのスレッド数を 256、命令レベルの並列性を 2 とする実装が最高性能であった。`icpx` のコンパイルオプションとしては`-O3 -fsycl -fsycl-unnamed-lambda -fsycl-targets=amd_gpu_gfx942` を指定した。コンパイルオプションについては、さらに`-ffast-math -ftz` を付与した性能測定も行ったが、有意な性能変

表 4.3.2　NVIDIA GH200 上の性能測定結果。

	$N = 4\,194\,304$		$N = 33\,554\,432$	
	最高値 $[\mathrm{s}^{-1}]$	中央値 $[\mathrm{s}^{-1}]$	最高値 $[\mathrm{s}^{-1}]$	中央値 $[\mathrm{s}^{-1}]$
CUDA C++	2.05×10^{12}	2.05×10^{12}	2.08×10^{12}	2.08×10^{12}
SYCL (acpp)	2.11×10^{12}	2.11×10^{12}	2.16×10^{12}	2.15×10^{12}
Kokkos	2.12×10^{12}	2.10×10^{12}	2.13×10^{12}	2.11×10^{12}
OpenACC (`kernels`)	1.69×10^{12}	1.68×10^{12}	1.71×10^{12}	1.70×10^{12}
OpenACC (`parallel`)	1.69×10^{12}	1.68×10^{12}	1.70×10^{12}	1.69×10^{12}
OpenMP (`loop`)	1.69×10^{12}	1.68×10^{12}	1.70×10^{12}	1.70×10^{12}
OpenMP (`distribute`)	1.56×10^{12}	1.56×10^{12}	1.57×10^{12}	1.57×10^{12}

化は見られなかった。

Kokkos 実装においては、`Kokkos::TeamPolicy` を用いたシェアードメモリ実装に raw pointer を渡し、チームあたりのスレッド数を 512、ループアンローリング段数を 64、命令レベルの並列性を 2 に設定した実装が最も高性能であった。AMD GPU 上で HIP や SYCL を用いた際にはシェアードメモリを用いない方が高性能になる場合も多いが、Kokkos 実装においてはシェアードメモリを用いないと大幅に性能が低下する。シェアードメモリを用いない HIP 実装を Kokkos から直接呼び出すと性能が改善することから、Kokkos が生成するカーネルがインデックス計算などのオーバーヘッドを付与しているためと考えられる[3]。

OpenMP target 実装においては、`distribute` 指示文を用いた実装においてブロックあたりのスレッド数を 1024 とした実装が最も高性能であり、コンパイルオプションとして`-O3 -fopenmp --offload-arch=gfx942` を指定した。`amdclang++` のコンパイルオプションについては、さらに`-ffast-math -fgpu-flush-denormals-to-zero` を付与した性能測定も行ったが、有意な性能変化は見られなかった。AMD MI210 上では OpenMP target の `loop` 指示文と `distribute` 指示文の両方の性能はほぼ一致した [22] が、AMD MI300A 上では `loop` 指示文を用いた実装の性能が極端に低かったために性能測定を打ち切った。`loop` 指示文はより多くをコンパイラに委ねる実装手法であり、AMD MI300A では CPU と GPU が統合されているため、CPU 側で実行されて性能が低下した可能性がある。

■性能評価手順　性能評価にあたっては N 体計算における 1 ステップ分の重力計算に要する時間を測定対象としており、CPU-GPU 間のデータ転送に要する時間は含めていない。また、OpenMP の task 指示文を利用して、別スレッドで 0.1 s 間隔で GPU の温度・消費電力・動作周波数を取得した [23]。NVIDIA 製 GPU においては NVIDIA Management Library（NVML）を用い、AMD 製 GPU においては ROCm SMI ライブラリを用いて測定を行った。また、NVIDIA GH200 および AMD MI300A においては、CPU と GPU モジュールが統一されているために sysfs を用いた電力測定 [24] もさらに別のスレッドを立てて実行した。各測定は粒子数 N ごとに 10 回ずつ実施し、測定開始前には 5 分間スリープさせて GPU 温度を十分に下げた状態から測定を開始した。

性能測定結果　図 4.3.1a および表 4.3.2 に示した NVIDIA GH200 上の性能測定結果では、どの実装手法においても演算性能（単位時間あたりの相互作用計算数）は粒子数 N に依らないかわずかに上昇する傾向が見られた。これは今回の測定範囲とした粒子数 N が 4 194 304 以上であり、GPU の演算コアを埋め尽くすだけの十

[3] `https://github.com/kokkos/kokkos/issues/8738`

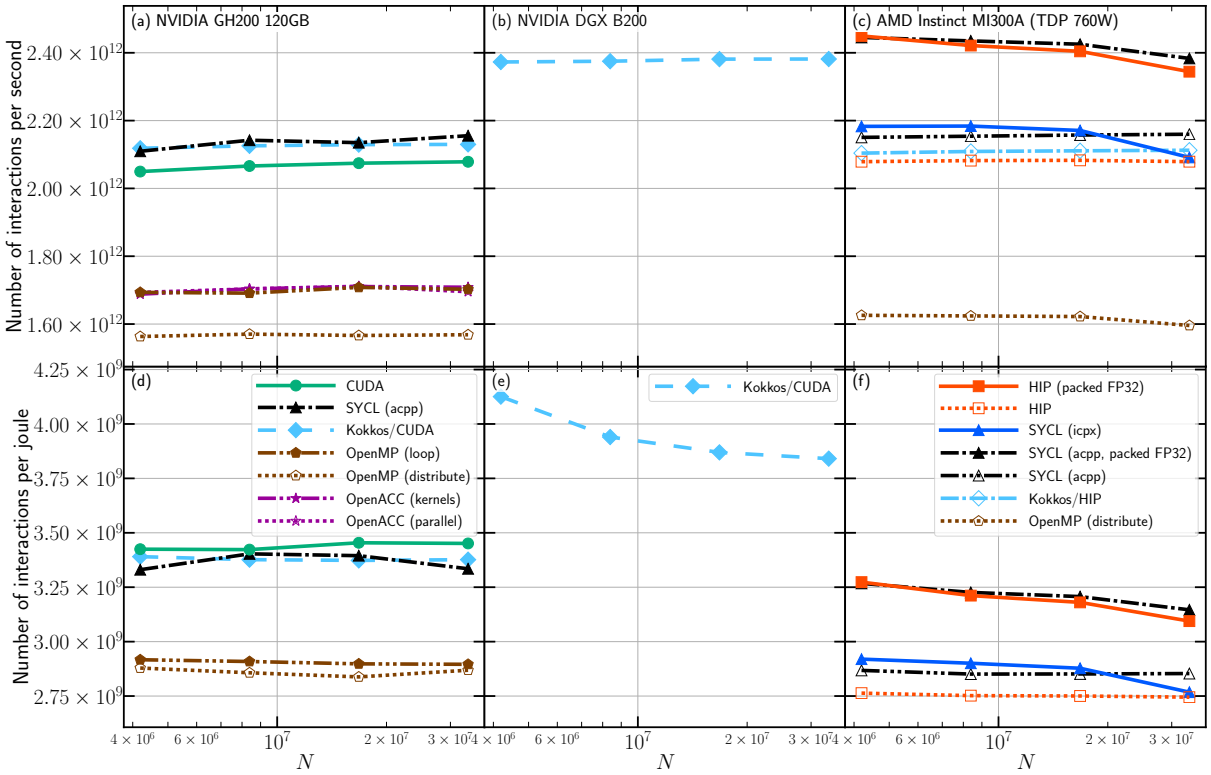

図 4.3.1　N 体計算の性能測定結果。1 秒あたりの相互作用計算数（上段）および消費エネルギーあたりの相互作用計算数（下段）の、10 回測定した結果の最高値をそれぞれ粒子数 N の関数として示す。(a, d) JCAHPC が運用する Miyabi-G 上の NVIDIA GH200 上の測定結果を示す。電力測定については、GH200 モジュール全体の消費電力を用いた。(b, e) 東京工科大学が運用する青嵐上の NVIDIA B200 上の測定結果を示す。電力測定については、NVML を用いて測定した GPU 部の消費電力を用いた。(c, f) QST/NIFS が運用するプラズマシミュレータ “双星” 上の AMD MI300A 上の測定結果を示す。電力測定については、MI300A 全体の消費電力を用いた。

分な演算量を確保できていたためである。最高性能となったのは粒子数 $N = 33\,554\,432$ における SYCL 実装（acpp）であり、CUDA C++ 実装よりも 3.70% 高かった。また Kokkos 実装についても、CUDA C++ 実装よりも 2.47% 高性能となり、SYCL や Kokkos といった性能可搬プログラミングを用いた実装が、CUDA C++ 実装と同等以上の性能を発揮できると示された。

指示文を用いた実装手法は、CPU 実装とコードを共通化できる、GPU 移植に要する工数を大幅に削減できるといった利点がある反面、低レベルな最適化ができないために得られる性能は低下する。OpenACC と OpenMP target の `loop` 指示文の実装はほぼ同性能であり、粒子数 $N = 33\,554\,432$ において CUDA C++ 実装の 82% 程度という指示文実装としては高い性能が得られた。OpenMP target の `distribute` 指示文の実装は CUDA C++ 実装の 75.5% であり、他の 3 通りの指示文実装よりも性能が低下しているものの、依然として指示文実装としては十分な性能が得られている。

NVIDIA B200 上の演算性能については、Kokkos 実装についてのみ測定した（図 4.3.1b）。粒子数 $N = 33\,554\,432$ における 10 回の測定の最高値は $2.38 \times 10^{12}\,\mathrm{s}^{-1}$（中央値は $2.38 \times 10^{12}\,\mathrm{s}^{-1}$）であり、NVIDIA GH200 上の Kokkos 実装の性能の 1.12 倍である。両者の理論ピーク性能比は 1.13 倍であるため、理論ピーク性能比相当の性能向上が得られていることが分かる。

表 4.3.3　AMD MI300A 上の性能測定結果。

	$N = 4\,194\,304$		$N = 33\,554\,432$	
	最高値 $\left[\mathrm{s}^{-1}\right]$	中央値 $\left[\mathrm{s}^{-1}\right]$	最高値 $\left[\mathrm{s}^{-1}\right]$	中央値 $\left[\mathrm{s}^{-1}\right]$
HIP (packed FP32)	2.45×10^{12}	2.44×10^{12}	2.34×10^{12}	2.34×10^{12}
HIP (vector FP32)	2.08×10^{12}	2.08×10^{12}	2.08×10^{12}	2.07×10^{12}
SYCL (acpp, packed FP32)	2.45×10^{12}	2.44×10^{12}	2.38×10^{12}	2.38×10^{12}
SYCL (acpp, vector FP32)	2.15×10^{12}	2.15×10^{12}	2.16×10^{12}	2.16×10^{12}
SYCL (icpx)	2.18×10^{12}	2.18×10^{12}	2.09×10^{12}	2.08×10^{12}
Kokkos	2.10×10^{12}	2.10×10^{12}	2.11×10^{12}	2.11×10^{12}
OpenMP (`distribute`)	1.63×10^{12}	1.63×10^{12}	1.60×10^{12}	1.59×10^{12}

図 4.3.1c および表 4.3.3 に示した AMD MI300A 上の性能測定結果では、演算性能がほぼ粒子数 N に依らない実装（packed FP32 命令を用いない HIP および SYCL（acpp）実装、Kokkos、OpenMP target）と、粒子数 N を増やすと性能が低下する実装（packed FP32 命令を用いた HIP および SYCL（acpp）実装、SYCL（icpx））という 2 種類の挙動が見られた。SYCL（icpx）実装は部分的に packed FP32 命令を用いるバイナリを生成するため、packed FP32 命令の使用有無がこうした挙動の違いの主な要因だと考えられる。また演算性能については、packed FP32 命令を用いて実装した HIP および SYCL（acpp）実装、packed FP32 命令を明示的には用いずに実装した HIP および SYCL（acpp, icpx）実装、そして指示文実装である OpenMP target 実装という 3 グループに大別される。

最高性能は粒子数 $N = 4\,194\,304$ における HIP 実装（packed FP32 命令使用）の $2.45 \times 10^{12}\,\mathrm{s}^{-1}$ であり、これは NVIDIA GH200 上の最高性能（SYCL（acpp）、粒子数 $N = 33\,554\,432$）の 1.14 倍である。両者の理論ピーク性能比は 1.83 倍であるため、理論ピーク性能比に比べて性能向上率が著しく小さい。

この理由を理解するには、AMD MI300A の使用電力および動作周波数の挙動を調べる必要がある。図 4.3.2 に、粒子数 $N = 16\,777\,216$ における各 GPU 上での N 体計算中の GPU 温度、消費電力、動作周波数の時間変化を示す。図 4.3.2i の AMD MI300A 上での動作周波数を見ると、計算開始直後の時点で既にブーストクロック周波数（点線）を大幅に下回っており、packed FP32 命令を使っている場合にはより低い周波数で動作していることが分かる。この動作周波数の低下は、供給電力不足が原因だと考えられる。消費電力の挙動（図 4.3.2f）を見ると、消費電力は TDP（破線）に達しており、また計算完了直前に一部の CU が演算処理を終了するのに伴い消費電力が低下すると同時に動作周波数が上昇している。また、動作周波数の低下と GPU 温度（図 4.3.2c）には目立った相関が見られないことから、動作周波数低下の主要因は温度によるサーマルスロットリングではないと考えられる。供給電力不足による性能低下は、計算の開始・終了処理の寄与が小さくなる長時間計算、つまり大粒子数において顕著になる。これが、AMD MI300A 上で packed FP32 命令を用いた際に粒子数 N が増加するにつれて演算性能が低下していく主な要因だと考えられる。NVIDIA 製 GPU 上では電力供給能力に余裕があり、また GPU 自体も十分に冷却されていたために、動作周波数の低下は見られずに常時ブーストクロックで動作していた（図 4.3.2g, h）。

Packed FP32 命令を用いない実装および指示文を用いて GPU 化した実装においては、それぞれ NVIDIA GH200 上の低レベル実装および指示文実装の性能とほぼ同等であった。このことから、AMD MI300A 向けに特化した最適化を施さずに性能可搬プログラミングを用いた実装が、NVIDIA GH200 上の同様の実装と同等の性能を発揮できることが示された。さらに AMD MI300A 向けに特化した実装として packed FP32 命令

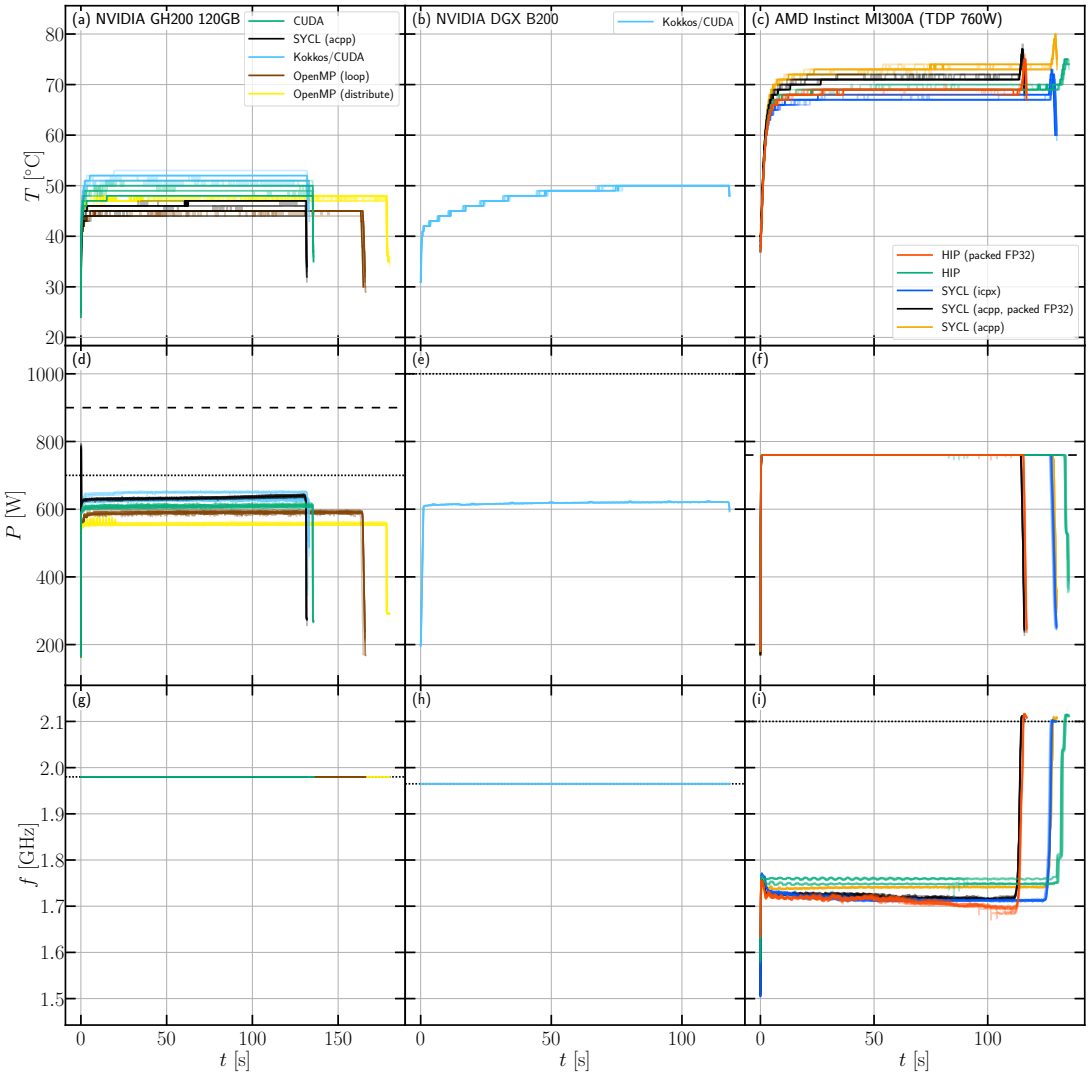

図 4.3.2　N 体計算中の GPU 状態の時間変化（粒子数 $N = 16777216$）。上段に GPU 温度、中段に GPU の消費電力、下段に GPU のクロック周波数を示す。各開発環境ごとに 10 回測定した結果を全てプロットしている。(a, d, g) JCAHPC が運用する Miyabi-G 上の NVIDIA GH200 上の測定結果を示す。(d) の電力測定については GH200 モジュール全体の消費電力を用い、GH200 全体での電力制限値（破線）と GPU 部分の電力制限値（点線）を表示した。(b, e, h) 東京工科大学が運用する青嵐上の NVIDIA B200 上の測定結果を示す。(e) の電力測定については NVML を用いて測定した GPU 部の消費電力を用い、B200 の TDP を点線で示した。(c, f, i) QST/NIFS が運用するプラズマシミュレータ “双星” 上の AMD MI300A 上の測定結果を示す。(f) の電力測定については MI300A 全体の消費電力を用い、MI300A の TDP を破線で示した。(g, h, i) には GPU のブーストクロック周波数を点線でプロットした。

を用いた実装を採用することで、NVIDIA GH200 上の性能を上回り、また NVIDIA B200 と同等の性能が得られることも分かった。

消費エネルギーあたりの性能測定結果　図 4.3.1d および表 4.3.4 に、NVIDIA GH200 上での消費エネルギーあたりの性能測定結果を示す。演算性能においては CUDA C++ 実装は他の低レベル実装に比べてわずかに劣っていたが、消費エネルギーあたりの性能においては CUDA C++ 実装が最も高いという結果になった。

表 4.3.4　NVIDIA GH200 上の消費エネルギーあたりの性能測定結果（$N = 33\,554\,432$）。

	GH200 モジュール全体		GPU 部（sysfs）		GPU 部（NVML）	
	最高値 $\left[\mathrm{J}^{-1}\right]$	中央値 $\left[\mathrm{J}^{-1}\right]$	最高値 $\left[\mathrm{J}^{-1}\right]$	中央値 $\left[\mathrm{J}^{-1}\right]$	最高値 $\left[\mathrm{J}^{-1}\right]$	中央値 $\left[\mathrm{J}^{-1}\right]$
CUDA C++	3.45×10^9	3.41×10^9	3.65×10^9	3.61×10^9	3.64×10^9	3.60×10^9
SYCL (acpp)	3.33×10^9	3.32×10^9	3.66×10^9	3.65×10^9	3.54×10^9	3.53×10^9
Kokkos	3.38×10^9	3.37×10^9	3.60×10^9	3.59×10^9	3.58×10^9	3.58×10^9
OpenMP (`loop`)	2.90×10^9	2.86×10^9	3.08×10^9	3.04×10^9	3.07×10^9	3.03×10^9
OpenMP (`distribute`)	2.87×10^9	2.83×10^9	3.06×10^9	3.02×10^9	3.08×10^9	3.04×10^9

表 4.3.5　AMD MI300A 上の消費エネルギーあたりの性能測定結果。

	$N = 4\,194\,304$		$N = 33\,554\,432$	
	最高値 $\left[\mathrm{J}^{-1}\right]$	中央値 $\left[\mathrm{J}^{-1}\right]$	最高値 $\left[\mathrm{J}^{-1}\right]$	中央値 $\left[\mathrm{J}^{-1}\right]$
HIP (packed FP32)	3.27×10^9	3.26×10^9	3.09×10^9	3.09×10^9
HIP (vector FP32)	2.76×10^9	2.76×10^9	2.74×10^9	2.74×10^9
SYCL (acpp, packed FP32)	3.27×10^9	3.26×10^9	3.15×10^9	3.15×10^9
SYCL (acpp, vector FP32)	2.87×10^9	2.87×10^9	2.85×10^9	2.85×10^9
SYCL (icpx)	2.92×10^9	2.92×10^9	2.77×10^9	2.76×10^9

ただし、実装手法間の違いは小さい。また、Miyabi-G の計算ノードの割り当てはジョブスケジューラによって動的に決定されるため、各測定においては異なる計算ノードが割り当たる。つまり、各計算ノードの個体差が反映されている可能性があり、今回の測定結果だけを用いて CUDA C++ 実装が最も高い消費エネルギーあたりの性能を持つと結論することはできない。その一方で、指示文を用いた OpenMP 実装においては、消費エネルギーあたりの性能が CUDA C++ 実装よりも 10% 以上低下しており、最大のエネルギー効率を目指す際には低レベル実装を採用する必要があることが分かる。OpenACC 実装については、NVIDIA HPC SDK のコンパイルエラーのために GPU 状態監視スレッドを立てられなかったため、消費エネルギーあたりの性能測定はできていない。

表 4.3.4 には、sysfs を用いて GH200 モジュール全体あるいは GPU 部のみの消費電力を測定した場合と、NVML を用いて GPU 部の消費電力を測定した場合の 3 通りの消費エネルギーあたりの性能を示している。Sysfs を用いた測定どうしの比較から CPU 部分である Grace などが全体の 1 割程度の消費電力を占めていることが分かり、また GPU 部を対象とした sysfs と NVML の測定結果が SYCL（acpp）を除いてほぼ一致することが分かる。SYCL（acpp）実装において、sysfs と NVML の測定結果が異なる理由は不明である。

NVIDIA B200 の消費電力測定においては sysfs が利用できないために NVML を用いて GPU 部の消費電力を測定し、その結果を用いて得られた消費エネルギーあたりの性能を図 4.3.1c に示す。NVML の測定は sysfs に比べて時間分解能が低く実際の消費電力上昇に追随できていないため、特に実行時間が短い場合には電力性能を過大評価（消費電力を過小評価）する傾向がある。今回の測定において最も信頼性の高い粒子数 $N = 33\,554\,432$ における Kokkos 実装の消費エネルギーあたりの性能は $3.84 \times 10^9\,\mathrm{J}^{-1}$（中央値は $3.84 \times 10^9\,\mathrm{J}^{-1}$）であり、NVML による NVIDIA GH200 上の GPU 部電力測定結果に基づいて評価した Kokkos 実装の消費エネルギーあたり性能よりも 7.20% 向上していた。この性能向上は、製造プロセスルールが TSMC 4N から TSMC 4NP に改良されたためだと考えられる。

図 4.3.1f および表 4.3.5 に、AMD MI300A 上での消費エネルギーあたりの性能測定結果を示す。粒子数依存性については演算性能と同様の傾向であり、また packed FP32 命令の使用によって消費エネルギーあたりの性能も向上していることが分かった。しかし packed FP32 命令を用いた実装においても、NVIDIA GH200 上の消費エネルギーあたり性能には及ばず、また packed FP32 命令を用いない際には NVIDIA GH200 上の指示文実装と同等の性能であった。この結果は AMD MI300A の製造プロセスルールが TSMC 5N であり、NVIDIA GH200/B200 の TSMC 4N/4NP に比べて一世代古いことが主な要因であると考えられる。Kokkos 実装については GPU 状態監視機能有効時にコンパイルエラーが発生したため、また OpenMP target 実装については GPU 状態監視機能有効時に演算性能が極端に低下したために、消費エネルギーあたりの性能測定は行っていない。

■まとめ 本研究では、NVIDIA GH200、NVIDIA B200、AMD MI300A という 3 種類の最新 GPU 上で、直接法に基づく N 体計算コードを CUDA/HIP/SYCL/Kokkos/Solomon という複数の GPU プログラミング手法で実装し、それぞれの性能を比較した。

図 4.3.1 に示したように、性能可搬プログラミングである SYCL や Kokkos を用いた実装であっても、低レベルな CUDA/HIP 実装と同等かそれ以上の演算性能を発揮できることが分かった。こうした低レベルな性能可搬プログラミングは、いずれも C++ のラムダ式を活用するものであり、GPU プログラミングにおける低レベルな最適化を行いつつも、コードの可搬性を高めることができる。ただし Kokkos については、SYCL よりもさらに抽象度の高いプログラミング環境となっているために、自分の実装と最終的に発行される命令との対応関係を把握しづらく、デバッグおよび性能最適化に際してはより時間がかかることも分かった。

Solomon を用いた指示文実装においては、低レベル実装である CUDA/HIP/SYCL/Kokkos に比べて数十パーセント程度性能が低下したものの、CPU 向け実装とコードを共通化できるため、GPU 移植に要する工数を大幅に削減できることがあらためて確認された。また NVIDIA GH200 上で OpenMP target を用いる際には `loop` 指示文の方が高速であったのに対し、AMD MI300A 上では `distribute` 指示文の方が高速になるという、環境ごとの性能差が見られた。実際に用いる指示文や構文をコンパイル時点で選択できるという Solomon の特徴は、こうした環境ごとの性能差に対応する上でも有用だと考えられる。

NVIDIA GH200 と NVIDIA B200 の性能比較においては、本研究では Kokkos 実装のみの比較にとどまったものの、理論ピーク性能比にほぼ等しい性能向上が得られた。この性能比較においては、ブロックあたりのスレッド数の調整だけを施すだけという汎用的な性能最適化のみで、各世代に特化した性能最適化を施すとこなく両 GPU アーキテクチャ上で高い性能可搬性が得られることが分かった。

NVIDIA GH200 と AMD MI300A の性能比較においては、AMD MI300A の理論ピーク性能が NVIDIA GH200 の 1.83 倍であるにも関わらず、実際の性能向上は 1.14 倍にとどまった。その要因は AMD MI300A の電力供給能力不足に起因する動作周波数の低下であると考えられる。これは前世代の AMD MI210 においても観測されていた現象であり、AMD 製 GPU 上で演算律速なアプリケーションを実行する際の特性として認識しておく必要がある。AMD MI300A 上で最大性能を発揮するには packed FP32 命令を用いる実装が有効だと確認でき、また一世代前の CDNA 2 向けのコードからの変更も必要なかった。

消費エネルギーあたりの性能を比較すると、低レベル実装である CUDA/HIP/SYCL/Kokkos が指示文実装に比べて高い性能を発揮することが分かった。用いた GPU ごとの比較という観点では、NVIDIA B200、NVIDIA GH200、AMD MI300A の順に高い消費エネルギーあたり性能が得られたことから、半導体の製造プロセスルールの世代が新しいほど高いエネルギー効率が得られていると解釈できる。

4.3.2.2 生成 AI ツールを用いた Solomon 改良の試行

Solomon は、GPU 向け指示文である OpenACC [17] や OpenMP target [18] を簡易に切り替えて使用するために開発されたマクロライブラリである [22]。

Solomon においては、GPU オフローディングしたい処理に `OFFLOAD()` や `PRAGMA_ACC_KERNELS_LOOP()`、`PRAGMA_OMP_TARGET_TEAMS_DISTRIBUTE_PARALLEL_FOR()` といった指示文を模した記述を追加することで、OpenACC や OpenMP target における指示文を生成することで GPU オフローディングを支援する。こうした指示文はそれぞれ指示節やリストを受けられるようになっており、カンマ区切りで渡された複数の入力を受け付けられる。公開当初のバージョンの Solomon においては、指示節として `AS_INDEPENDENT` を付与する際に制約があった。この指示節は OpenMP をバックエンドとすると `simd` と読み替えられ、例えば `Pragma("omp target teams distribute parallel for simd")` などへと変換される。しかし OpenMP においては、`simd` は指示節ではなく構文の一部であるため、この例においては `for` と `simd` の間に別の指示節が挿入されるとエラーとなってしまう。したがって、`AS_INDEPENDENT`（あるいは `ACC_CLAUSE_INDEPENDENT`、`OMP_TARGET_CLAUSE_SIMD`）は全てのオプション指示節の先頭に記述する必要があった。

こうした制約はユーザーの利便性を損ねるため、2025 年 12 月時点の Claude Code を用いて機能拡張に取り組んだ。目指した機能は指示節ごとに優先度を設定し、この優先度ごとに自動的に指示節をソートする機能である。本機能の実現により、ユーザーが指示節の順序を意識せずに入力できるようになる。Solomon はプリプロセッサマクロを利用したマクロライブラリであり、このソート処理はコンパイル時点において完了する必要があるためにその具体的実装は非自明なものとなる。そこで本研究では Claude Code にこうした機能の実現可能性を問い合わせたうえで実装を依頼した。

Claude Code を用いた実装作業中においては、テストコンパイル結果を見ながらの修正処理が進行したものの、問題が生じたために作業を中断することとした。コンパイルエラーが生じた際に、`#pragma omp target teams distribute parallel for simd` という記法が OpenMP の仕様に沿っていないためにコンパイルエラーが生じたと Claude Code 側が判断し、Solomon の OpenMP target 向け出力を削除するよう提案してきたためである。実際には上記の構文は OpenMP の仕様にのっとったものであり、この事実を Claude Code に指摘しても頑強に抵抗してきたために作業を中断することとした。本試行が失敗した要因としては、Claude Code に与えたタスクが一定程度複雑な思考を要する高度なものであった点、OpenACC に比べて OpenMP target に関する学習量が十分でなかった可能性などが考えられる。

また Claude Code を用いた試行では実装完了に至らなかったソート機能の実現であるが、Claude Code が生成したコードを人手でデバッグし、さらにソート機能ではなく `simd` のみを出力するフィルターと `simd` 以外を出力するフィルター機能として実装することで同等の機能を実現した。本機能は最新版の Solomon に取り込み済みであり、今回の試行は Solomon の利便性改善につながったと言える。

4.3.3 次年度の調査研究の計画

HAIRDESC との連携により HPCI のアプリケーション分野からの要望調査を開始し、また重点分野における要件整理とベンチマーク・サンプルコード候補の抽出を実施する。あわせて主要な国際会議である ISC に参加して海外における最新動向を調査するとともに、次世代アプリケーション開発手法調査研究サブグループとも連携しながら生成 AI を活用しながら多様な開発手法への対応方針についても調査する。さらに代表的なベンチマークとサンプルコードを作成・評価し、性能要件の明確化と開発支援手法の技術的検証を

進める。ここで得られた知見を調査研究内の各グループに横展開するのに加え、HPC 研究会や日本天文学会など様々なコミュニティにおいて積極的に発表することで、幅広いコミュニティに情報を還元していく。また、HAIRDESC において東京大学で雇用予定の教員に本サブグループにも参画してもらうことにより、HAIRDESC との連携を強化する。

4.4 次世代アプリケーション開発手法調査研究サブグループ

4.4.1 調査研究の目的

本サブグループでは、次世代アプリケーションニーズ調査研究サブグループと連携し、次世代計算機環境におけるアプリケーション開発手法について調査を行うことを目的とする。

近年、大規模言語モデル（LLM）の発展により、ソースコード生成や既存コードの最適化を支援する技術が急速に発展している。これらの技術は、科学技術計算アプリケーションの開発効率を向上させる可能性がある一方で、大規模 HPC アプリケーションへの適用可能性は十分に検証されていない。

そこで本サブグループでは、代表的な科学技術計算アプリケーションを対象として、Claude Code など民間サービスの LLM を用いた GPU 向けコード生成およびプログラムの高性能化を試行する。また、民間サービスの LLM を用いたアプリケーション開発の課題を整理するとともに、ローカル環境で動作する LLM を用いたアプリケーション開発手法についても検討する。

4.4.2 調査研究の結果

4.4.2.1 LLM を用いた代表的アプリケーションの GPU 移植

本研究では、大規模言語モデル（Large Language Models, LLM）を用いた科学技術計算アプリケーションの GPU 移植の可能性を評価するため、有限要素解析アプリケーション GeoFEM を対象として GPU 向けコード生成の試行を行った。GeoFEM は三次元弾性解析などを対象とした並列有限要素解析アプリケーションであり、MPI+OpenMP によるハイブリッド並列化が施された Fortran コードとして長年にわたり開発されてきた HPC アプリケーションである。近年のスーパーコンピュータでは GPU を標準的に搭載する構成が一般化しており、シミュレーション分野においても GPU 利用は不可欠となりつつある。一方で、CPU 向けに開発された既存アプリケーションを GPU 向けに移植する作業は依然として大きな負担となっている。

従来の CPU 向けスーパーコンピュータでは MPI+OpenMP による並列化が標準的であったのに対し、GPU 向け並列プログラミングでは CUDA，OpenACC，OpenMP target など複数のプログラミングモデルが存在する。さらに GPU ベンダごとにサポート状況が異なるため、アプリケーション開発者にとってはプログラミングモデル選択と実装の両面で大きな負担が生じる。特に Fortran で記述されたレガシーアプリケーションでは利用可能な GPU プログラミング手法が限られており、コード移植の困難さがさらに増大する。

こうした問題を解決する手段として、近年急速に発展しているコード生成 AI の活用が期待されている。コード生成 AI が既存の CPU 向けアプリケーションから GPU 向けコードを生成し、さらにテスト実行によって CPU 版との結果一致を確認できるのであれば、アプリケーション開発者は従来の CPU 向け開発を継続しながら GPU 対応を進めることが可能となる。本研究では、コード生成 AI の一つである Claude Code (Opus 4.1) を用いて GeoFEM/Cube ミニアプリケーションの GPU 移植を試行し、コード生成能力および開発プロセス全体を評価した。

GeoFEM/Cube は GeoFEM から性能評価用に抽出されたミニアプリケーションであり、三次元弾性静

解析問題における疎行列連立一次方程式を反復法で解く処理を対象としている。プログラムは fixed-form Fortran で記述され、MPI+OpenMP による並列化が施されている。本アプリケーションは主に以下の二つの処理から構成される。

- 係数行列生成（coefficient matrix assembly）
- 共役勾配法（Conjugate Gradient, CG）ソルバ

このうち CG ソルバ部分は疎行列ベクトル積や内積、AXPY などの基本的な数値カーネルから構成される比較的単純なループ構造を持つ。一方、係数行列生成部分は深い多重ループと条件分岐を含む複雑な処理であり、GPU 向け並列化を行うためにはループ構造の再設計が必要となる場合が多い。

GPU コード生成の評価では、Claude Code を複数の独立したディレクトリで実行し、同一の入力コードに対して GPU コード生成を試行した。Claude Code はコード生成だけでなく、コンパイル、実行、テストといった一連の作業を自律的に実行できるため、本研究では GPU コード生成からテスト実行までを含む開発プロセス全体を評価対象とした。コード生成の再現性を評価するため、同一ソースコードを用いた複数回の実行を行い、生成されたコードの性能および生成に要した時間を比較した。

また、入力ソースコードの形式がコード生成に与える影響を評価するため、以下の三種類のコードを準備した。

- fixed-form Fortran（`.f`）
- free-form Fortran（`.f90`）
- OpenMP 指示文を除去した free-form Fortran

fixed-form Fortran は古い Fortran コードに多く見られる形式であり、行長制限や implicit 型宣言などの特徴を持つ。著者らの経験から、これらの要素はコード開発時のバグを誘発しやすく、コード生成 AI による開発効率にも影響を与える可能性があるため、その影響を評価することとした。

実験の結果、fixed-form Fortran を入力とした場合にはコード生成時間が長くなる傾向が確認された。これは fixed-form 特有の書式制約によりコード修正時のエラーが増加するためであると考えられる。また implicit 型宣言により変数型が暗黙的に決定される場合、変数名変更時に新しい変数として解釈されることがあり、これが誤動作の原因となるケースも観測された。これらの結果は、人間の開発者と同様にコード生成 AI においても可読性の高いコードが開発効率を向上させることを示している。

生成された GPU コードの性能評価結果のまとめについて、図 4.4.1 に示す。図 4.4.1 は人手による実装と Claude Code による実装の実行時間を示しており、CG ソルバの時間と行列生成部分の時間の積み上げである。実行時間は JCAHPC が運用するスーパーコンピュータ Miyabi-G の GH200 GPU を用いており、10 回計測したうちの中間パターンを示している。全体として、CG ソルバ部分については比較的良好な GPU 実装が生成されるケースが確認された。ただし、単に OpenACC を用いて GPU 実装を指示した場合（左から 3 番目）、最もよく観察されたパターンは、OpenACC のスレッド数を指定するための指示文が指定されておらず、不十分な性能であった。この実装に対して、「さらに速くせよ」と指示を加えることで、スレッド粒度を指定するための指示節が追加され、人手によるコードと同等の性能を得ることができた。

一方で、係数行列生成部分の GPU 並列化については、単に「GPU を使って高速化せよ」と指示しただけでは、そもそも実施されなかった。左から 4 番目のケースを見れば、明らかに係数行列生成部分がボトルネックであり、Claude Code 自身がテスト実行によりそれを把握しているにも関わらず、「さらに高速化」を指示した 5 番目のケースにおいても、実施した全 10 ケース全てにおいて並列化の対象とならなかった。この事例

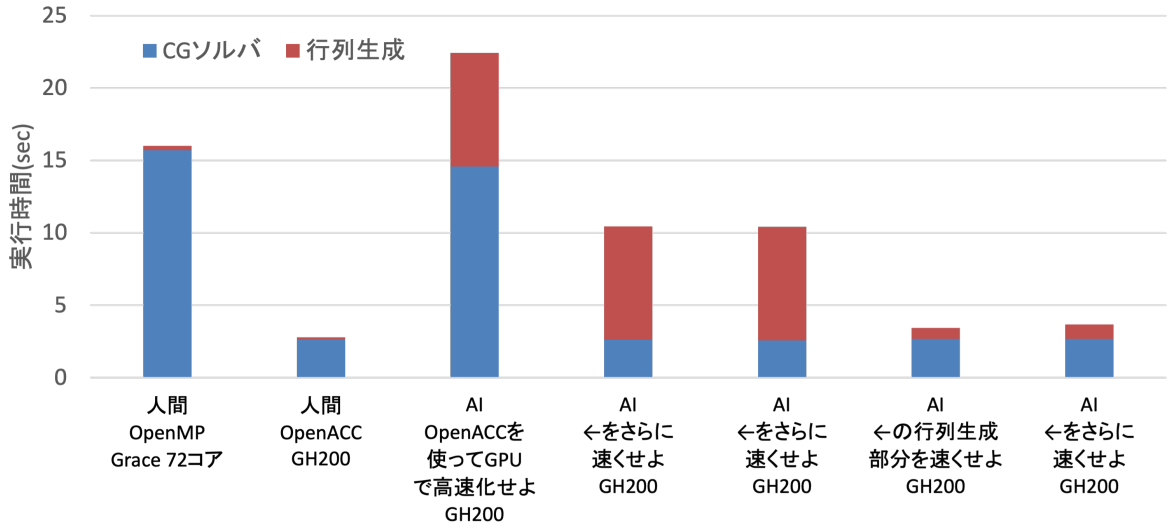

図 4.4.1　GH200 における GeoFEM の性能評価まとめ。左から元コード、人間による OpenACC コード、以降が AI による生成コードである。

から、Claude Code は実測性能のフィードバックから性能改善を提案しているわけではなく、学習に基づき CG 部分が GPU 並列化の対象であると判断したと推察される。

係数行列生成部分の GPU 向け並列化を指示するためには、具体的に「行列生成部分を高速化せよ」（右から 2 番目）と指示する必要があった。ただし、計算結果は正しいものの性能は不十分であり、全 10 ケースで最も高速なコードであっても、人手によるコードと比較して 6 倍以上低速であった。「さらに高速化」を指示した場合（1 番右）であっても、より遅くなるだけで、満足なコードは生成されなかった。この部分ではアプリケーション固有の複雑な処理が多く含まれるため、コード生成 AI がループ並列化の可能性を十分に判断できない場合が多いと考えられる。すなわち、CG ソルバのような一般的な数値計算カーネルについては AI が既存知識を利用して適切な並列化を行うことができる一方、アプリケーション固有のロジックについては適切な並列化を行うことが困難である可能性が示唆された。

図 4.4.2 に係数行列生成部分の Claude Code の実装例、図 4.4.3 に、人手による OpenACC 実装例を示す。元の OpenMP 実装では、3 行目の `icel0` ループに対して `!$omp parallel do` による並列化が施されている。Claude Code による実装例では、元の OpenMP 実装同様、3 行目の `icel0` ループをスレッド並列化している。これ自体は間違っていないが、より性能を求める場合には、外側のループをスレッドブロック並列化、内側のループをスレッド並列化する必要がある。また、`icel0` ループの並列化においては、20 行目の `atomic` 指示文は不要である。このループ構造を機械的に眺めると、21 行目の代入分は書き込み競合する可能性があるため、保守的に `atomic` 指示文を導入することはある種正しい判断ではあるが、1 行目のカラーリングループによって、3 行目の `icel0` ループによる書き込み競合は発生しないよう実装されているため、性能を求めるなら `atomic` 指示文を書くべきではない。このように、アプリケーション固有の知識を用いた性能最適化について、プログラマが LLM に与えてあげる必要性があると考えられる。図 4.4.3 の人手による OpenACC 実装例では、`icel0` ループを `gang` レベル並列化し、`je` ループを `ie` ループの直後に配置し直すことで一重化可能ループにした上で、 `collapse` 節を用いて長さ 64 のループとして並列化している。この実装の場合においては、`ie, je` ループの並列化に伴う書き込み競合を回避するために、22 行目の `atomic` 指示

```
do icol= 1, ELMCOLORtot
     !$acc parallel loop gang vector private(...)
     do icel0= ELMCOLORindex(icol-1)+1, ELMCOLORindex(icol)
          ...
          do ie= 1, 8
               ip = nodLOCAL(ie)
               if (ip.le.N) then
                    do je= 1, 8
                         jp = nodLOCAL(je)
                         ...
                         do kpn= 1, 2
                              do jpn= 1, 2
                                   do ipn= 1, 2
                                        ...
                                   enddo
                              enddo
                         enddo
                         ...
                         if (IDlu.eq.1) then
                              !$acc atomic update
                              AU(9*kk-8)= AU(9*kk-8) + a11
                              ...
                         endif
                         ...
                    enddo
               endif
enddo; enddo; enddo
```

図 4.4.2　行列生成部の Claude Code による OpenACC 実装例。

文が必要となる。この場合、ie，je ループの処理が一つのスレッドブロック内で閉じるため、キャッシュを有効利用することができ、かつ本来大きなオーバーヘッドとなる atomic 処理についても L1 キャッシュの範囲内で実施できるため、低オーバーヘッドで並列化できる。このような GPU の特性を考慮した最適化は、プログラマが指示するか、あるいはより GPU プログラミングに特化したコード生成 AI を開発する必要があると考えられる。

以上の結果から、LLM を用いた GPU コード生成はアプリケーションの一部については有効であるものの、アプリケーション固有の複雑な処理に対しては十分な性能を持つコードを生成することが困難であることが明らかとなった。また、入力コードの可読性や並列構造の明確さがコード生成成功率に大きく影響することも確認された。

なお、本研究の成果は SCA/HPCAsia 2026 Workshop（LLM4HPCAsia 2026）において発表 [25] され、Best Paper Award を受賞した。本研究は、コード生成 AI を用いた実アプリケーションの GPU 移植プロセスを体系的に評価した研究として高く評価されたものである。

```
do icol= 1, ELMCOLORtot
    !$acc parallel num_gangs(...) vector_length(64)
    !$acc loop gang private(...)
    do icel0= ELMCOLORindex(icol-1)+1, ELMCOLORindex(icol)
        ...
        !$acc loop collapse(2) vector(64)
        do ie= 1, 8
            do je= 1, 8
                ip = nodLOCAL(ie)
                if (ip.le.N) then
                    jp = nodLOCAL(je)
                    ...
                    do kpn= 1, 2
                        do jpn= 1, 2
                            do ipn= 1, 2
                                ...
                            enddo
                        enddo
                    enddo
                    ...
                    if (IDlu.eq.1) then
                        !$acc atomic update
                        AU(9*kk-8)= AU(9*kk-8) + a11
                        ...
                    endif
                    ...
                endif
            enddo
enddo; enddo; enddo
```

図 4.4.3　行列生成部の妥当な OpenACC 実装例。

4.4.2.2　コード生成 AI による GPU 移植の課題

前節では、LLM を用いた実アプリケーションの GPU 移植の可能性について GeoFEM を対象に評価した。一方で、HPC 分野において実際に GPU 移植の対象となるコードは、CPU 向けに長年にわたり最適化されてきたレガシーコードである場合が多い。このようなコードでは、可読性よりも CPU 上での性能を優先した設計が行われているため、コード生成 AI による理解や変換が困難になる可能性がある。

そこで本研究では、CPU 向けに高度に最適化された HPC コードに対するコード生成 AI の適用可能性を評価するため、ICTCG 法ソルバを対象とした GPU 移植実験を実施した。

対象コードは 3 次元ポアソン方程式を解く反復ソルバであり、約 3,500 行の Fortran 90 プログラムで構成されている。このコードには以下のような HPC 特有の最適化が含まれている。

- SCS（Sell-C-Sigma）形式による疎行列格納
- 明示的スレッド管理による OpenMP 並列化

- プリプロセッサマクロによる実装切替
- グラフ彩色による並列化

これらの最適化は CPU 上での性能向上には有効であるが、GPU 並列化とは並列化パラダイムが異なるため、GPU 移植時の障壁となる可能性がある。特に OpenMP の明示的スレッド管理は GPU 向け並列化と構造的に不整合を生じやすく、コード生成 AI が並列構造を正しく解釈できない場合がある。

GPU 移植実験では CLI 型コード生成 AI として Claude Code，Codex，Gemini CLI の 3 種類を比較対象とした。これらのツールは自然言語による指示に基づいてコード生成や修正を行い、コンパイルエラーや実行時エラーを解釈して修正案を提示するなど、対話的なソフトウェア開発を支援する機能を持つ。

実験は以下の三段階で実施した。

1. シンプルアプローチ
2. 段階的アプローチ
3. de-optimization アプローチ

シンプルアプローチでは、「このコードを GPU で実行できるように移植してほしい」という単純な指示のみを与えて GPU 移植を試みた。その結果、Claude Code のみが GPU 移植に成功し、CPU 実装と比較して約 2.1 倍の高速化を達成した。一方で Codex および Gemini CLI は GPU 移植に失敗した。

次に、AI にコード構造を理解させてから GPU 移植を行う段階的アプローチを試みた。この方法ではまず AI にコード全体の構造を説明させ、その説明を参照しながら GPU 移植を行わせた。その結果、Claude Code は CPU 比約 2.6 倍の高速化を達成したが、Codex と Gemini CLI は依然として移植に失敗した。

これらの結果から、CPU 向け最適化によるコード構造の複雑さが GPU 移植失敗の原因である可能性が示唆された。そこで本研究では CPU 固有の最適化を段階的に除去する *de-optimization* 手法を検討した。

本研究では de-optimization を以下の二段階で実施した。

第 1 段階　コメントアウトコードや未使用コードの削除、プリプロセッサマクロの整理など、可読性向上を目的としたコード整理

第 2 段階　OpenMP 並列化を除去し、シリアル実行版へ変換

第 1 段階の de-optimization を適用したコードに対して Codex および Gemini CLI による GPU 移植を試みたが、両者とも移植に失敗した。これはコードの複雑さの主因がコメントやマクロではなく CPU 並列化構造そのものであることを示唆している。

一方、第 2 段階の de-optimization を適用したコードでは Codex による GPU 移植が成功し、CPU 実装と比較して約 6.3 倍の高速化を達成した。

さらに、全国大会発表時には de-optimization 後のコードを入力として Claude Code による GPU コード生成実験を追加実施した。その結果、元コードを入力とした場合には Claude Code による GPU 実装は CPU 実装に対して約 2.6 倍の高速化にとどまったのに対し、de-optimization 後のコードを入力とした場合には約 14.5 倍の高速化を達成した。また、同一の de-optimization 後コードを入力として生成された GPU 実装を比較すると、Claude Code の実装は Codex の実装より約 2.3 倍高速であった。

この結果は、CPU 向け最適化を除去してコード構造を単純化することで、複数のコード生成 AI が効率的な GPU 実装戦略を選択できるようになることを示している。また、入力コードの構造が生成される実装戦略および性能に大きな影響を与えることも明らかとなった。

以上の結果から、CPU 向けに高度に最適化された HPC コードに対してコード生成 AI を適用する場合には、コード構造の複雑さが大きな障壁となること、また de-optimization によるコード単純化が GPU 移植成功率および性能向上に有効である可能性が示された。

なお、本研究の成果は情報処理学会第 88 回全国大会において発表 [26] され、学生著者による優れた研究として学生奨励賞を受賞した。

4.4.2.3 ローカル LLM を用いたアプリケーション開発

前節まででは、主にクラウド型 LLM を用いた HPC アプリケーションの GPU 移植について評価を行った。しかし、クラウド型 LLM の利用には外部ネットワーク接続や利用コストなどの制約があり、スーパーコンピュータ環境において常に利用できるとは限らない。また、研究コードや未公開データを扱う場合には、外部サービスへのコード送信が難しい場合もある。

そのため、本研究ではローカル環境で動作するオープン LLM を用いた HPC アプリケーション開発支援についても検討 [27] した。具体的には、コード生成モデルとして公開されている Qwen 系列の LLM を対象とし、HPC コード生成能力を向上させるための学習手法を検討した。

一般的なコード生成 LLM は、単体テストを通過するかどうかなどの機能的正しさを主な学習目標として訓練されている。しかし HPC 分野では、機能的に正しいコードであっても実行性能が大きく異なる場合があり、性能を考慮したコード生成が重要となる。例えば同じアルゴリズムを実装した場合でも、ループ順序やメモリアクセスパターン、並列化戦略などの違いによって実行時間が数倍以上変化することがある。

そこで本研究では、生成されたコードを実際の計算機上で実行し、得られた性能を報酬として LLM を学習させるオンライン強化学習手法を検討した。この手法では、生成されたコードをコンパイルして実行し、得られた GFLOPS などの性能指標をモデルの報酬として利用する。これにより、手動で作成した教師データを用いなくても、実行性能に基づいてコード生成モデルを改善することが可能となる。

さらに、本研究では Staged Quality-Diversity（SQD）と呼ばれる段階的探索アルゴリズムを導入した。SQD ではコード最適化を複数の段階に分け、以下のような最適化技術を段階的に許可する。

- レジスタブロッキング
- キャッシュブロッキング
- SIMD 命令
- OpenMP 並列化
- メモリプリフェッチ

このように探索空間を段階的に拡張することで、モデルは基本的な最適化から高度な最適化までを体系的に学習することができる。

図 4.4.4 は、行列積を対象とし、SQD によって強化学習した際の学習時ベンチマークの性能を示している。`-O0`，`-O3` はコンパイラオプションを表しており、コンパイラの最適化の有無による学習への影響差を調査している。横軸が木の深さを示しており、学習が進むにつれ、概ね性能が向上していることがわかる。

強化学習の結果出来上がったモデルの評価実験では、学習に用いた行列積の他に、行列ベクトル積、Jacobi 法など複数の HPC カーネルを対象に性能を評価した。その結果、強化学習を適用したモデルは元のモデルと比較して大幅な性能向上を示した。例えば行列積カーネルでは最大で約 7.8 倍の性能向上が確認され、また行列ベクトル積 や Jacobi などのメモリ帯域律速カーネルでも大きな改善が観測された。

これらの結果は、ローカル LLM を用いた HPC コード生成においても、実機性能を利用した学習によって

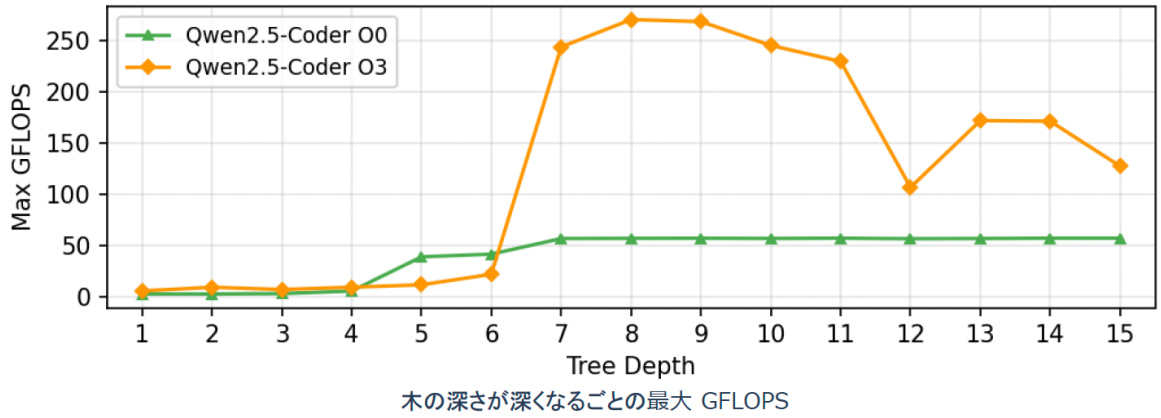

図 4.4.4　SQD による行列積強化学習時のベンチマーク結果。

コード生成能力を向上させることが可能であることを示している。今後は GPU カーネルやより大規模なアプリケーションを対象とした評価を行い、HPC アプリケーション開発支援におけるローカル LLM の活用可能性をさらに検討する予定である。

以上の結果から、本サブグループでは、LLM を用いた HPC アプリケーション開発支援について、以下の知見を得た。

第一に、GeoFEM のような実アプリケーションに対しても、LLM を用いた GPU 移植が一定程度可能であることを確認した。特に CG ソルバのような比較的単純なループ構造を持つ部分では、LLM による並列化が有効に機能することが分かった。

第二に、CPU 向けに高度に最適化された HPC コードでは、その最適化構造が LLM によるコード理解の障壁となる場合がある。de-optimization によりコード構造を単純化することで、複数のコード生成 AI が効率的な GPU 実装を生成できることを確認した。

第三に、クラウド型 LLM だけでなく、ローカル LLM を用いた HPC コード生成についても、実機性能を報酬とする学習により性能向上が可能であることを確認した。

これらの結果は、LLM を用いた HPC アプリケーション開発支援が今後の重要な研究課題であることを示している。

4.4.3　次年度の調査研究の計画

本年度の調査結果から、LLM を用いた HPC アプリケーション開発支援には一定の可能性がある一方で、大規模アプリケーションへの適用にはいくつかの課題があることが明らかとなった。

特に、大規模 HPC アプリケーションの GPU 移植は、数万行規模のコードに対する長時間の開発タスクとなる場合が多く、LLM による自動化には困難が伴う。例えば、LLM ツールのコンテキスト制約やセッションの自動圧縮（auto-compact）機能、あるいはスーパーコンピュータのインタラクティブノードの利用時間制限などにより、長時間にわたる開発タスクを継続的に実行することが困難となる場合がある。

そこで次年度以降は、大規模 HPC アプリケーションを対象とした LLM 支援型開発手法について、以下の二つの方向で研究を進める予定である。

第一に、大規模アプリケーションの GPU 移植タスクを適切に小規模な開発タスクへ分割する手法の検討で

ある。GPU 移植は通常、アルゴリズム理解、データ構造の変換、カーネル並列化、データ転送最適化など複数の作業から構成される。これらの作業を LLM が処理可能な粒度の小タスクに分割することで、長時間タスクを段階的に処理する開発ワークフローの構築を目指す。

第二に、複数の LLM を階層的に利用する AI エージェント型開発手法の検討である。具体的には、プロジェクト全体の進行を管理する上位エージェントを設置し、各サブタスクを下位の LLM に割り当てることで、大規模な開発タスクを分散処理する方式を検討する。このような階層的なエージェント構造を導入することで、単一の LLM では対応が困難な長大な開発タスクへの対応を可能とすることを目指す。

これらの研究を通じて、LLM を用いた HPC アプリケーション開発支援技術の実用化可能性を評価するとともに、次世代計算機環境における AI 支援型アプリケーション開発の方向性を明らかにする予定である。

参考文献

[15] Aymen Alsaadi et al. “RHAPSODY: Execution of Hybrid AI-HPC Workflows at Scale”. In: *ArXiv* abs/2512.20795 (2025).

[16] Matthieu Dorier et al. “Toward a persistent event-streaming system for high-performance computing applications”. In: *Frontiers in High Performance Computing* Volume 3 - 2025 (2025). ISSN: 2813-7337. DOI: 10.3389/fhpcp.2025.1638203.

[17] OpenACC-Standard.org. *The OpenACC Application Programming Interface Version 2.7*. 2018.

[18] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Version 5.2*. 2021.

[19] Andreas Herten. “Many Cores, Many Models: GPU Programming Model vs. Vendor Compatibility Overview”. In: *Proceedings of the SC ’23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W 2023, Denver, CO, USA, November 12-17, 2023*. ACM, 2023, pp. 1019–1026. DOI: 10.1145/3624062.3624178.

[20] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns”. In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003.

[21] Christian R. Trott et al. “Kokkos 3: Programming Model Extensions for the Exascale Era”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 805–817. DOI: 10.1109/TPDS.2021.3097283.

[22] Yohei Miki and Toshihiro Hanawa. “Unified Schemes for Directive-Based GPU Offloading”. In: *IEEE Access* 12 (Jan. 2024), pp. 181644–181665. DOI: 10.1109/ACCESS.2024.3509380. arXiv: 2411.18889 [cs.DC].

[23] 三木 洋平 and 塙 敏博. “NVIDIA H100 PCIe および AMD MI210 における N 体計算コードの性能評価”. In: **研究報告ハイパフォーマンスコンピューティング（*HPC*）** 2023-HPC-190.28 (July 2023), pp. 1–13. ISSN: 2188-8841.

[24] 塙 敏博, 三木 洋平, and 小原 亮真. “GH200 における電力性能最適化”. In: **研究報告ハイパフォーマンスコンピューティング（*HPC*）** 2025-HPC-199.8 (May 2025), pp. 1–7. ISSN: 2188-8841.

[25] Tetsuya Hoshino et al. “Evaluating Claude Code’s Coding and Test Automation for GPU Acceleration of a Legacy Fortran Application: A GeoFEM Case Study”. In: *Proceedings of SCA/HPCAsia 2026 Workshops (LLM4HPCAsia 2026)*. Best Paper Award. 2026. DOI: 10.1145/3784828.3785335.

[26] Shugo Sakaguchi et al. “ICTCG 法の計算プログラムの GPU 移植を対象としたコード生成 AI の比較”. In: **情報処理学会 第 *88* 回全国大会**. 7K-05, 学生奨励賞. 2026.

[27] Ryo Mikasa et al. “Improving HPC Code Generation Capability of LLMs via Online Reinforcement Learning with Real-Machine Benchmark Rewards”. In: *arXiv preprint arXiv:2602.12049* (2026).

様式第２１

学　会　等　発　表　実　績

委託業務題目「HPCI整備計画調査研究事業」
機関名:（代表機関）理化学研究所
１．学会等における口頭・ポスター発表

発表した成果（発表題目、口頭・ポスター発表の別）	発表者氏名	発表した場所（学会等名）	発表した時期	国内・外の別
Status updates from RIKEN -- FugakuNEXT, FS3.0（口頭発表）	佐藤賢斗	CEA-RIKEN Workshop 2026	2026/1/22	国内
Designing Next-Generation HPC Infrastructure: The FugakuNEXT Case（口頭発表）	佐藤賢斗	DREAM: 1ST WORKSHOP ON DATA REDUCTION AND ENERGY-AWARE DATA MOVEMENT in HPCAsia2026 https://dream-workshop.github.io	2026/1/29	国内
System Software Solutions for FugakuNEXT and Beyond（口頭発表）	佐藤賢斗	MulticoreWorld2026 https://multicore.world	2026/2/19	国外
Pathways to vendor-neutral GPU computing for sustainable simulation code development （口頭発表）	三木洋平	「富岳成果創出加速プログラム」基礎科学合同シンポジウム 2025 https://kds.kek.jp/event/57304/timetable/	2026/1/14	国内
多様なプログラミング手法を用いたN体計算コードのGPU実装：NVIDIA GH200およびAMD MI300A上での性能比較 （口頭発表＋ポスター発表）	三木洋平	日本天文学会 2026年春季年会	2026/3/6	国内

２．学会誌・雑誌等における論文掲載

掲載した論文（発表題目）	発表者氏名	発表した場所（学会誌・雑誌等名）	発表した時期	国内・外の別
FS3.0: 富岳NEXT時代を見据えたHPCI運用システム整備計画に関する調査研究	佐藤,賢斗, 小松,一彦, 高橋,慧智, 横田,理央, 小林,諒平, 佐藤,雅之, 冨嶋,茂樹, 遠藤,新, Mohamed,Wahib, Jens,Domke, 藤田,典久, 宮島,敬明, 深谷,猛, 建部,修見, 三木,洋平, 下川辺,隆史, 星野,哲也	情報処理学会 研究報告ハイパフォーマンスコンピューティング（HPC）巻 2026-HPC-203, 号 33, p. 1-6	2026/03/09	国内
N体計算におけるGPUプログラミング手法比較：NVIDIA GH200/B200およびAMD MI300Aでの性能評価	三木 洋平, 塙 敏博	情報処理学会 研究報告ハイパフォーマンスコンピューティング（HPC）巻 2026-HPC-203, 号 47, p. 1-10	2026/3/18	国内
AMD MI300A APUにおける共有メモリシステムの性能評価	藤田 典久, 吉川 耕司, 辻 美和子, 朴 泰祐, 建部 修見	情報処理学会 研究報告ハイパフォーマンスコンピューティング（HPC）, 巻 2026-HPC-203, 号 2, p. 1-8	2026/3/16	国内