最先端高性能計算システムにおける第一原理電子動力学シミュレーションのコデザイン

最先端高性能計算システムにおける
第一原理電子動力学シミュレーションの
コデザイン

2018年9月

廣川 祐太

最先端高性能計算システムにおける第一原理電子動力学シミュレーションのコデザイン

廣川 祐太

システム情報工学研究科

筑波大学

2018 年 9 月

概要

今日の高性能計算システムでは，消費電力に対する演算性能 (以下，“電力効率” と呼ぶ) からメニーコア CPU システムや GPU に代表されるアクセラレータを搭載したスーパコンピュータ，システムが広く用いられており，2018 年 6 月の TOP500 リストでは上位 10 件のうち 9 件がメニーコア CPU または GPU を搭載している．メニーコア CPU や GPU は，消費電力と演算性能が低いコアを一般的な CPU に比べ非常に多数接続したプロセッサで，高い並列性と理論性能を持つ．しかしながら，理論性能と実性能には大きな乖離があり，高い並列性を持つアルゴリズムの適用や大規模システムにおける通信削減など，実アプリケーションへの要求は厳しさを増している．

特に理論性能と実性能の乖離は，高性能計算と計算科学の両分野が協力して解決しなければならない問題であり，“コデザイン (協調設計)” が注目されている．コデザインは，高性能計算と計算科学の協調による両分野の相互発展を狙うもので，今後必須の開発手法となることが予測されるが，実性能に依らない様々な評価が必要と考えられる．本研究の目的はメニーコア CPU と GPU，各最先端アーキテクチャおよびそれらを採用するシステムにおいて，電子動力学シミュレーションのコデザインによる最適化と性能評価を行い，各アーキテクチャで得られる性能や記述性，可搬性などの議論から，実アプリケーションのコデザイン手法の模索と，電子動力学へ寄与することである．

本研究では，まず実アプリケーションのコデザインについて定義し，電子動力学シミュレーション ARTED について最適化と性能評価を実施した．メニーコア CPU では，アプリケーションの支配的な計算であるステンシル計算について，512-bit SIMD 命令を用いたベクトル計算の手動実装を行い，Intel の最新メニーコア CPU である Knights Landing において，HPL 性能比でおよそ 38% の実行効率を獲得した．世界最大規模の Knights Landing クラスタである Oakforest-PACS では，核となるハミルトニアン計算で 4 PFLOPS，HPL 比で約 30% の実行効率を達成し，高いスケーリング性能を示した．GPU はアプリケーション全体をディレクティブベース言語である OpenACC で実装することで，開発コストの削減を狙ったが，やはりメニーコア CPU における SIMD 命令と同じく，ネイティブな CUDA プラットフォームを用いた最適化が要求されている．各アーキテクチャのステンシル計算性能は HPL 比で 40–50% の性能を達成し，本研究で実施した最適化は各アーキテクチャの性能をよく引き出せている．

アプリケーションの性能，保守性，最適化の水準や可搬性について，各項目が如何に達成できているかの議論により，実アプリケーションのコデザインに必要な要件について評価と考察を行った．各議論より，各アーキテクチャの導入メリットとデメリットをまとめ，最後にエクサスケール以降のシステムにおいてコデザインのために必要なアーキテクチャの技術やプログラミングモデルについて議論した．得られる実性能に関しては，アクセラレータである GPU が最も良い性能かつ高い電力効率を有しているが，ア

クセラレータによる新たなプログラミングモデルの導入が必要となり，保守性を阻害してしまう．メニーコア CPU はデファクトスタンダードである OpenMP+MPI のハイブリッド並列構造を維持可能で，計算科学との協調が容易というメリットがある．実アプリケーションにおけるコデザインの実施には，アプリケーションの開発とその最適化が同時並行で協調して行われる必要があり，保守や拡張といったメンテナンス性が非常に重要と考えられる．ただし，性能とメンテナンス性にはトレードオフがあり，どちらをより重要視するかという議論はコデザインにおいて重要な論点である．

目次

概要 **i**

第 1 章 序論 **1**
1.1 研究背景 1
1.2 研究目的 2
1.3 本論文の構成 2

第 2 章 メニーコア CPU と GPU **5**
2.1 今日のスーパーコンピュータ開発 5
2.2 メニーコア CPU 6
2.2.1 Sunway SW26010 6
2.2.2 Intel Xeon Phi 7
2.3 GPU (Graphics Processing Unit) 9
2.4 理論性能と実性能の乖離: HPL と HPCG ベンチマーク 11

第 3 章 高性能計算と計算科学のコデザイン **15**
3.1 コデザインの実例 15
3.1.1 ハードウェアにおけるコデザイン – 計算機開発 15
3.1.2 ソフトウェアにおけるコデザイン – 実アプリケーション最適化 16
3.2 現在進行中のコデザインベースプロジェクト 17
3.2.1 FLAGSHIP 2020 17
3.2.2 Exascale Computing Project 19
3.3 コデザインの必要性と本研究の意義 20
3.4 本研究におけるコデザインの定義と評価指標 21

第 4 章 ARTED: 電子動力学シミュレータ **23**
4.1 概要 23
4.2 計算・並列化方法 24
4.2.1 マルチスケール計算 24
4.2.2 ハミルトニアン計算 26

4.2.3 ステンシル計算 . . . 27
4.3 コデザインに基づく実装と最適化 . . . 29
4.4 関連研究 . . . 30

第 5 章 メニーコア CPU における最適化と性能評価 31
5.1 評価環境 . . . 31
5.1.1 KNC クラスタ: COMA . . . 31
5.1.2 KNL クラスタ: Oakforest-PACS . . . 32
5.2 ベクトル並列におけるステンシル計算の最適化 . . . 33
5.2.1 マルチコアプロセッサでのコンパイラベクトル最適化の促進 . . . 34
5.2.2 KNC プロセッサへのさらなる最適化 . . . 36
5.2.3 512-bit SIMD 命令を用いた手動ベクトル化 . . . 38
5.2.4 問題依存のさらなる最適化 . . . 42
5.2.5 Knights Corner から Knights Landing へ . . . 43
5.3 ヘテロジニアス環境におけるクラスタレベル最適化 . . . 44
5.3.1 通信アルゴリズム . . . 45
5.3.2 静的ロードバランス . . . 46
5.4 アーキテクチャ依存最適化 . . . 46
5.4.1 メニースレッド環境におけるベクトル和の並列化 . . . 46
5.4.2 MCDRAM と DRAM の併用 . . . 48
5.5 性能評価 . . . 50
5.5.1 ステンシル計算の性能評価 . . . 50
5.5.2 Knights Corner/Landing クラスタ間の比較 . . . 51
5.5.3 Knights Landing クラスタにおける計算全体の性能評価 . . . 52
5.6 考察 . . . 54
5.6.1 動的クロック調整機能によるロードインバランス . . . 54
5.6.2 他のメニーコアプロセッサへの適用 . . . 54

第 6 章 GPU における最適化と性能評価 57
6.1 評価環境 . . . 57
6.2 アプリケーションの実装方法: OpenACC+CUDA . . . 57
6.3 アーキテクチャへの最適化 . . . 60
6.3.1 スレッド数・レジスタ数の調整 . . . 60
6.3.2 メモリアクセス最適化 . . . 60
6.3.3 Multi-Process Service によるスループット改善 . . . 63
6.4 性能評価 . . . 64
6.4.1 ステンシル計算 . . . 64
6.4.2 計算全体 . . . 65

6.5 考察 66

第 7 章 考察・議論 **69**

7.1 コデザインの検証 69

7.1.1 各アーキテクチャが達成する実性能の妥当性 69

7.1.2 実アプリケーションの協調開発 71

7.1.3 次世代システムへの適用可能性 72

7.1.4 総括 73

7.2 その他の議論 74

7.2.1 アーキテクチャ間評価 74

7.2.2 他アプリケーションへの最適化の適用 76

7.3 エクサスケールコンピューティングとその先に向けて 77

第 8 章 結論 **79**

8.1 まとめ 79

8.2 今後の課題 80

参考文献 **81**

謝辞 **87**

付録 A 研究業績 **89**

図目次

2.1 SW26010 アーキテクチャ 7
2.2 Knights Corner アーキテクチャ 8
2.3 Knights Landing アーキテクチャ 9
2.4 Volta アーキテクチャ (SM 構造) 10

3.1 実アプリケーションの開発サイクル 21

4.1 ARTED の計算領域: 3 次元マクスウェル方程式 + TDKS 方程式 24
4.2 MPI の並列化方法 25
4.3 ハミルトニアン計算全体の流れ 27
4.4 25 点ステンシル計算のメモリアクセスパターン 28
4.5 ステンシル計算のオリジナル実装 28

5.1 コンパイラによる自動ベクトル化に最適化したステンシル計算の実装 35
5.2 倍精度複素数積 $(a, bi)(0, -i)$ を展開した実装 39
5.3 Z 次元のメモリアクセス最適化 40
5.4 Z 次元のメモリアクセスの IMCI 実装 41
5.5 プリプロセッサを用いた IMCI から AVX-512 へのコード変換 44
5.6 (a) InfiniBand FDR を用いたノードをまたぐ PingPong 通信性能，(b) Symmetric 実行における倍精度浮動小数点数ベクトルの `MPI_Allreduce` 通信性能 45
5.7 ベクトル和の並列化における各実装の性能 47
5.8 ベクトル和の並列化の各実装 48
5.9 KNL のメモリモードの違いによる計算全体の性能比較 49
5.10 ステンシル計算の性能評価 50
5.11 KNC/KNL クラスタ間における強スケーリングの性能比較 51
5.12 Oakforest-PACS 全系による性能評価 53
5.13 時間発展計算の計算時間内訳 53
5.14 Skylake-SP プロセッサにおけるステンシル計算性能の評価 55

6.1 OpenACC+CUDA 実装の例: OpenACC/Fortran で記述されたアプリケーション本体 59

6.2 OpenACC+CUDA 実装の例: CUDA/C で記述・最適化された GPU カーネルコード . 59
6.3 本研究のステンシル計算の並列化方法 (OpenACC) 61
6.4 ステンシル計算における通常のメモリアクセス (X 次元の差分計算のみ) 61
6.5 グローバルメモリへのリードアクセスを削減した場合 (X 次元の差分計算のみ) 62
6.6 Y-Z 平面のグローバルメモリへのリードアクセス削減 62
6.7 CPU/GPU のアフィニティ設定例 63
6.8 CUDA MPS の効果 63
6.9 ステンシル計算の OpenACC と CUDA 実装の性能比較 64
6.10 計算全体の性能評価 66

7.1 Performance vs. Maintainability & Portability 75

表目次

2.1 理論性能 [PFLOPS] と実性能 (HPL と HPCG) [PFLOPS] の比較 11
2.2 本研究のターゲットアーキテクチャにおける HPL 性能 [GFLOPS] 12

3.1 ポスト「京」重点課題とターゲットアプリケーション 18
3.2 Exascale Computing Project stack 19

5.1 COMA 諸元 32
5.2 Oakforest-PACS 諸元 32
5.3 SPARC64 VIIIfx 諸元 33
5.4 L1D キャッシュミスおよびメモリプリフェッチ発行数 34
5.5 100 反復時のキャッシュメモリアクセス待ち時間 [sec] 35
5.6 100 反復時の演算待ち時間 [sec] 35
5.7 SPARC64 VIIIfx でのステンシル計算性能 36
5.8 コア内スレッド数による性能への影響 36
5.9 Knights Corner でのコンパイラベクトル化性能 37
5.10 倍精度浮動小数点数の load/gather 命令数 37
5.11 L1D キャッシュミス数とヒット率 37
5.12 手動ベクトル最適化の効果 41
5.13 周期境界領域拡張の効果 42
5.14 各シミュレーションのデータセット 52
5.15 ISSP System C 諸元 55

6.1 Pre-PACS-X (PPX) 諸元 58
6.2 OpenACC と CUDA のレジスタ使用数 65

7.1 各アーキテクチャの性能比較 70
7.2 想定される計算ノード構成 70
7.3 GPU に要した追加実装 (SLOC: source lines of code) 71
7.4 コデザインの達成状況 74
7.5 アーキテクチャ間評価 75

7.6　本研究で取り上げたうち条件を要する最適化 76

第 1 章

序論

1.1 研究背景

今日の高性能計算 (HPC, High-Performance Computing) システムではメニーコア CPU と GPU が広く用いられており，2018 年 6 月の TOP500 リストでは，上位 10 システムのうち 9 システムでメニーコア CPU または GPU が利用されている [1]．これらのアーキテクチャは，消費電力に対する演算性能 (以下，“電力効率”) が従来の CPU に対して優れていることが知られており [2]，消費電力 [Watt] あたりの性能 [GFLOPS] として 50 GFLOPS/Watt が目標とされるエクサスケールスーパーコンピュータ開発において必須技術と考えられている．メニーコア CPU と GPU は，回路規模が小さく単体性能が低い演算コアを非常に多数接続し，並列性能の向上によって高い理論演算性能を達成する．しかし現在運用中のこれらのアーキテクチャを用いたシステムは，理論性能と達成可能な実性能の間に大きな乖離がある．加えて並列性の大幅な増加と高度な最適化，アクセラレータのための新たなプログラミングモデルの導入などによって，高性能計算システムにおける実アプリケーション開発の難易度は飛躍的に上昇し，非常に大きな問題となっている．

これらの問題を解決するための手法として，高性能計算と計算科学による “コデザイン (協調設計)” が注目されている．高性能計算と計算科学のコデザインは，実アプリケーションを高速化するためのハードウェアとシステムの開発や，高性能計算システムに対する実アプリケーションの実装と最適化により，両分野の協調による発展を目指すものである．理論性能と実性能の乖離に代表される様々な問題は，高性能計算と計算科学の双方が協力して解決しなければならず，その手段こそがコデザインにほかならない．その中で実アプリケーションのコデザインは，“実アプリケーションの実装と最適化” という形で実現されてきたが，最適化によって引き起こされる難読化，保守性，可搬性の低下など，協調された開発とは言い難く，得られる実性能とのバランスについて議論が必要である．本研究では，最先端アーキテクチャであるメニーコア CPU および GPU に対してコデザインを行うことで，実性能に限らないコデザインの評価を考える．

筑波大学計算科学研究センターでは，光と物質の相互作用の第一原理計算を目的として，電磁気学と電子動力学を組み合わせたマルチスケール電子動力学シミュレータ ARTED (Ab-initio Real-Time Electron Dynamics simulator) を開発している [3]．ARTED は電磁気学との組み合わせにより，光の周期よりも短い時間でのシミュレーションによる基礎科学への貢献や，レーザー加工技術といった応用研究

に対しても正確かつ予測可能なシミュレーションを提供可能で，新たな計算科学分野として発展が期待されている．よく知られているように第一原理計算は非常に巨大な計算量を持つため，現在のスーパーコンピュータでも現実的な計算は数万原子規模かつピコ秒といった非常に短い時間スケールでしか実施できず [4, 5]，メニーコア CPU や GPU といった最先端アーキテクチャとシステムの活用が強く求められている．しかしながら，現在までに最先端アーキテクチャによる最適化や性能評価は実施されておらず，ARTED がターゲットとするシステムも「京」コンピュータと一般的な Xeon CPU の利用にとどまっている．

1.2 研究目的

本研究ではメニーコア CPU と GPU，各最先端アーキテクチャおよびそれらを採用するシステムにおいて，電子動力学シミュレーションのコデザインによる最適化と性能評価を行い，各アーキテクチャの実効性能や記述性，可搬性などの議論から，実アプリケーションのコデザイン手法を模索し，電子動力学への寄与を目的とする．

筑波大学計算科学研究センターとの共同研究より，同センター主導で開発されている電子動力学シミュレーションコード ARTED のコデザインを行う．同アプリケーションは，計算時間の大半をステンシル計算に費やしており，実シミュレーションには非常に多くの計算資源が要求される．ステンシル計算は計算科学分野において頻出する一般的な計算パターンで，その最適化と性能評価はコデザインにおいて重要な評価指標と考えられる．

Intel Xeon Phi は汎用 CPU として，高性能計算でのデファクトスタンダードである OpenMP+MPI 並列アプリケーションを直接実行可能で，実アプリケーションの可搬性の確保が容易と考えられる．実アプリケーション開発の観点では，アクセラレータを用いる場合は専用 API やディレクティブベース言語を用いたコードの修正や追記が必須となる．したがって，Intel Xeon Phi のように OpenMP+MPI の並列化構造を維持できる汎用メニーコア CPU が望ましいが，電力効率の観点ではアクセラレータは汎用メニーコア CPU に対し優位である．にもかかわらず，同じ実アプリケーションをメニーコア CPU と GPU 両方に実装・性能評価を行った例は極めて稀で，記述性や可搬性などを含めた包括的な議論は行われていない．本研究では，高性能計算システムで広く用いられている Intel Xeon Phi プロセッサをメニーコア CPU として，NVIDIA GPU をアクセラレータとしてそれぞれ最適化と性能評価を行い，各アーキテクチャとシステムが達成する実効性能と実アプリケーションのコデザインにおける議論を行う．

1.3 本論文の構成

本論文の構成を以下にまとめる．第 2 章にて，スーパーコンピュータシステムで利用されているメニーコア CPU と GPU のアーキテクチャについて，それらの特徴や抱える問題についてまとめる．第 3 章では，高性能計算と計算科学のコデザインの実例とコデザインベースで実施されているスーパーコンピュータの開発プロジェクトを参照し，最先端アーキテクチャが抱える問題からコデザインの重要性と本研究の意義，本研究におけるコデザインと評価方法について定義を行う．第 4 章でコデザインのターゲットアプリケーションである電子動力学シミュレーションコード ARTED について，計算と並列化内容をまとめ，

コデザインに従った実装と最適化の指針を述べる．

第 5 章および第 6 章にて，メニーコア CPU と GPU への ARTED の最適化と性能評価を実施し，各アーキテクチャで得られる性能や最適化手法についてまとめる．最適化と性能評価から，第 7 章でコデザインの考察を行い，協調開発のために何が必要となるのか議論を展開する．最後に，エクサスケールコンピューティングとその先について議論し，第 8 章で本研究をまとめる．

第 2 章

メニーコア CPU と GPU

本章では，メニーコア CPU と GPU のアーキテクチャ概要と，その特性について述べる．

2.1　今日のスーパーコンピュータ開発

今日の高性能計算システムでは，メニーコア CPU と GPU が広く用いられている．2018 年 6 月の TOP500 では，上位 10 件のうち 9 件がメニーコア CPU システム，または GPU に代表されるアクセラレータを搭載したシステムである [1]．これらのアーキテクチャは，消費電力と演算性能が低い演算コアを非常に多数接続したプロセッサで，高い並列性と理論演算性能を持ち，電力効率に優れることが知られている [2]．

演算コア単体の逐次演算性能は一般的な CPU に比べて非常に低く，Intel Xeon CPU がベースクロックで 2–3 GHz で動作するのに対し，メニーコア CPU の Intel Xeon Phi は 1.1–1.4 GHz 程度で動作する．これらのアーキテクチャは，並列実行可能な計算を多くのコアとスレッド，加えて SIMD 演算によって並列処理することで，高い演算性能を達成できる．したがって，実アプリケーションが巨大な並列実行領域を持たなければ，その高い演算性能を享受することは極めて困難である．実アプリケーションはプロセッサの逐次処理性能に頼ることはできず，高い並列性を提供できるように，アルゴリズムを最適化または実装する必要がある．アルゴリズムの改良を求められるにも関わらず，メニーコア CPU や GPU が高性能計算システムにおいて利用される背景には，電力効率がスーパーコンピュータ開発における致命的な問題となっていることが大きい．

2020 年前半を目処に各国で開発が進められているエクサスケールスーパーコンピュータは，電力 [Watt] あたりの演算性能 [GFLOPS] として 50 GFLOPS/Watt (供給電力上限はシステム全体で 20–30 MW 程度) が目標とされている．そこで，電力制約を満たしながら演算性能を向上させるためのアーキテクチャとして，メニーコア CPU と GPU に代表されるアクセラレータが注目されるようになった．前述の通り，これらのアーキテクチャは理論演算性能が一般的な CPU に比べて高く，電力効率に優れている．多くの HPC ベンダがこのアプローチにより高い電力効率を達成し，Intel では self-bootable なメニーコア CPU として Xeon Phi を，NVIDIA はアクセラレータとして Tesla P100, V100 GPU を提供している．しかしながら，2018 年 6 月の時点で世界最高の電力効率を持つシステムでも約 18.4 GFLOPS/W しか達成できておらず，設定されている目標は極めて困難である [2]．

運用コストの観点からも，電力効率は非常に重要視されている．例えば，東京電力の法人契約中，工場向けの “高圧電力” 契約から概算すると，システムで 4 MW の電力を必要とする JCAHPC (最先端共同 HPC 基盤施設) の Oakforest-PACS では，年間で 5 億円程度が電気料金として発生する．スーパーコンピュータは概ね 5 年の運用と保守を基本としているため，システム運用開始から運用終了までに要する電気料金は 25 億円となる．Oakforest-PACS の落札価格は政府調達公示より約 72 億円で，5 年間の運用とすると，調達と運用コストは合計で約 100 億円と全体で 20% 以上を運用コストが占めており，無視できない割合である．

中国の Sunway TaihuLight は，プロセッサあたり 256 個の演算コアを搭載したメニーコア CPU システムで，2018 年 6 月時点で，世界第 2 位の性能を持つ．消費電力は約 15 MW と日本の「京」コンピュータに対し約 1.2 倍しか増加していないにもかかわらず，9 倍以上の実効性能を達成している．ORNL (Oak Ridge National Lab.) では 2018 年 6 月に IBM Power9 CPU と V100 GPU による GPU クラスタ “Summit” の運用を開始し，100 PFLOPS を超える実効性能を世界で初めて達成した [6]．Summit は約 8.8 MW と Sunway TaihuLight の約半分の消費電力でありながら，約 1.3 倍の実効性能を達成しており，アクセラレータ型システムが高い電力効率を持っていることがわかる．アメリカエネルギー省ではエクサスケールシステムとして Intel Xeon Phi を用いた “Aurora (A21)” システムが計画され，2018 年には 180 PFLOPS，2021 年には 1 ExaFLOPS の到達を目標としている．日本においても，PEZY Computing により PEZY-SC プロセッサが開発され，MIMD で動作するアクセラレータ型クラスタとして，2018 年 6 月時点で世界一の電力効率を達成している．また，国家プロジェクト “フラッグシップ 2020” として開発が進められている “ポスト「京」コンピュータ” では ARM プロセッサをベースとしたメニーコア CPU システムが計画されている [7]．

各国の開発状況より，メニーコア CPU や GPU は今後の高性能計算システム開発における重要なキーであることは疑いの余地がない．しかし前述の問題から，これらの最先端アーキテクチャを用いたアプリケーションの開発においては，高い並列性を持つアルゴリズムの適用や大規模システムにおける通信削減など，プロセッサやシステムに対する理解と最適化がより強く求められることは避けられない．

2.2 メニーコア CPU

メニーコア CPU は，従来 CPU で実装される演算コアを単純化して回路面積と消費電力を小さくすることで，電力制約を満たしつつ従来よりも多くの演算コアを接続したプロセッサである．本節では SW26010 プロセッサと，Intel Xeon Phi プロセッサについてそれぞれ概要を述べる．

2.2.1 Sunway SW26010

SW26010 プロセッサは，2018 年 6 月現在で世界第 2 位の計算性能を持つ Sunway TaihuLight に用いられている [8]．SW26010 プロセッサのアーキテクチャを，図 2.1 に示す [9]．

プロセッサは，管理コアにあたる MPE (Management Processing Element) と計算コアにあたる CPE (Compute Processing Element) に分かれており，どちらも 64-bit RISC (Reduced Instruction Set Computer) コアとして実装されている．CPE は二次元のメッシュネットワークで 64 コアが接続さ

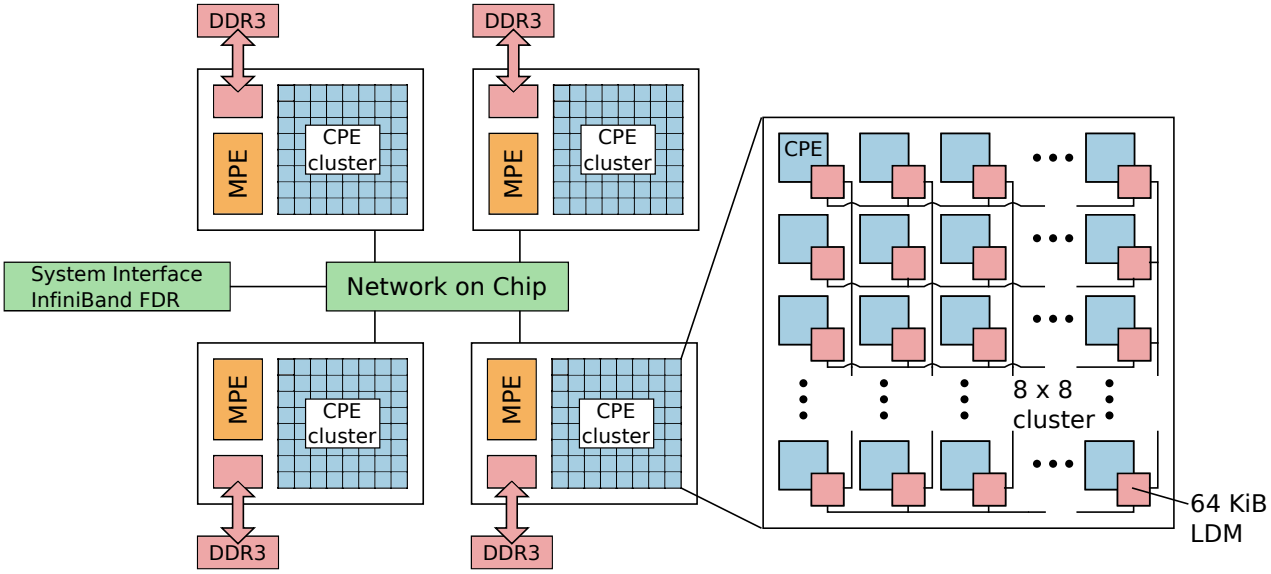

図 2.1　SW26010 アーキテクチャ

れており，CPE のメッシュネットワーク外に 1 個の MPE を配置し，1 MPE + 64 CPEs で 1 つの CG (Core Group) を構成する．CG はダイ上に 4 つ，4 MPEs + 256 CPEs で 1 個のプロセッサが構成され，1.45 GHz で動作し約 3.062 TFLOPS の理論性能を持つ．

MPE はメモリや通信の制御，タスクスケジューリングなどの計算以外の処理を行い，メインの計算は CPE で行う．各 CPE は 256-bit 幅の SIMD 演算器を持ち，各サイクルで 1 回の積和演算を実行，8 FLOP/cycle の処理が可能である．また，各 CPE は 32 本の浮動小数点数レジスタと，“スクラッチパッドキャッシュ” として 64 KiB の LDM (Local Data Memory) を持つ．スクラッチパッドキャッシュは，通常の CPU が提供するキャッシュとは異なり，ユーザがデータ管理を行うキャッシュメモリである．LDM は，スクラッチパッドキャッシュまたはソフトウェア制御の LLC (Last-Level Cache) として利用できる．

メインメモリは，各 CG に対して 8 GiB の DDR3 メモリが一枚しか接続されておらず，総メモリバンド幅はノードあたり 136.5 GB/s と，後述する Intel Xeon Phi や NVIDIA GPU に比べ極めて低い．[8, 9] でも述べられている通り，SW26010 プロセッサでは data locality を上げて LDM を有効活用できるかが重要となる．

2.2.2　Intel Xeon Phi

Xeon Phi は，Intel が開発した x86 互換のメニーコアプロセッサのプロダクトである．HPC 用途においては，第 1 世代の KNC (Knights Corner) と第 2 世代の KNL (Knights Landing) の 2 つのアーキテクチャが提供されている．

KNC は第 1 世代 Xeon Phi プロセッサとして提供された，最大 61 コアのプロセッサである．KNC プロセッサのアーキテクチャを，図 2.2 に示す [10]．各演算コアは 512-bit SIMD 演算機が 1 つ用意され，サイクル辺り 1 回の倍精度浮動小数点数の積和演算を実行，16 FLOP/cycle で処理できる．コアあたり

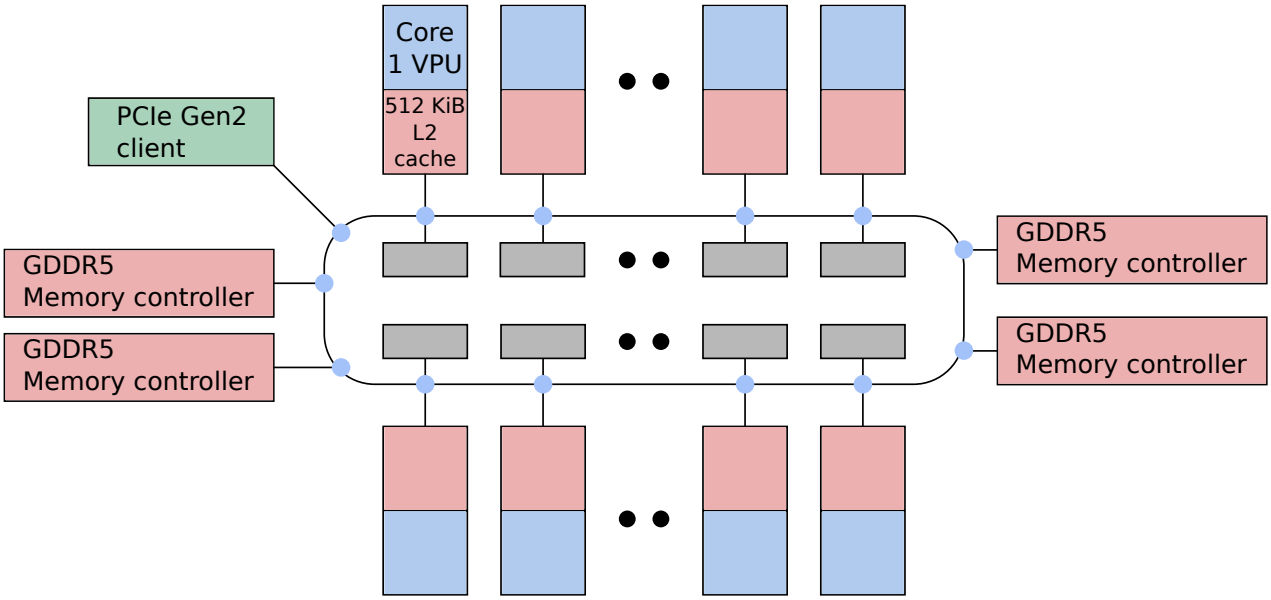

図 2.2　Knights Corner アーキテクチャ

32 KiB の L1 キャッシュと 512 KiB の L2 キャッシュメモリを持ち，各演算コアはメモリアクセスのレイテンシを隠蔽するために 4 つのハードウェアスレッドを処理できる．演算コアあたり 4 つのハードウェアスレッドを処理する場合，スレッドあたり最低 128 KiB の L2 キャッシュが割り当てられる．最大 61 コアがリングバスで接続され，最大 244 スレッドを同時実行可能である．各演算コアは約 1.1–1.2 GHz で動作し，プロセッサ単体の理論演算性能は 1 TFLOPS を超える．

Intel Xeon などの汎用 CPU に対し，KNC はプロセッサカードとして PCIe (PCI-Express) で接続され，起動や通信などの一部制御を CPU に委任している．しかし，KNC 上にはマイクロ Linux カーネルが動作し，KNC 自体が 1 個の計算ノードとして振る舞うことが可能で，これは CPU の下位デバイスとして扱われる GPU には不可能である．KNC が計算ノードとして振る舞えることで，x86 CPU で実装された実アプリケーションをほぼそのまま実行できるだけではなく，ポスト「京」コンピュータといったメニーコア CPU を唯一の計算資源として提供するシステムを想定した実アプリケーションの最適化や性能評価が可能となっている．GPU のようにアクセラレータとしても利用可能で，Intel MKL (Math Kernel Library) では，Level3 BLAS や LAPACK など計算コストの高いカーネルを KNC にオフロード可能である [11].

KNL は，第 2 世代 Xeon Phi プロセッサとして提供されている，最大 72 コアのプロセッサである．KNC と異なり通常のソケット型 CPU として提供され，KNC は Xeon CPU を組み合わせたヘテロジニアスシステムを構築する必要があったのに対し，KNL を唯一の計算資源とするメニーコアシステムの構築が可能となった．KNL のアーキテクチャを，図 2.3 に示す [12].

KNL は 2 つの演算コアをまとめて “Tile” と呼び，最大 36 枚の Tile を 2 次元のメッシュネットワークで接続することで，最大 72 コアをプロセッサダイ上に収めている．各演算コアは 32 KiB の L1 キャッシュと 512-bit SIMD 演算機を 2 つ持ち，サイクル辺り 2 回の倍精度浮動小数点数の積和演算を実行可能で，KNC の倍となる 32 FLOP/cycle を提供する．各演算コアが同時実行可能なハードウェアスレッド

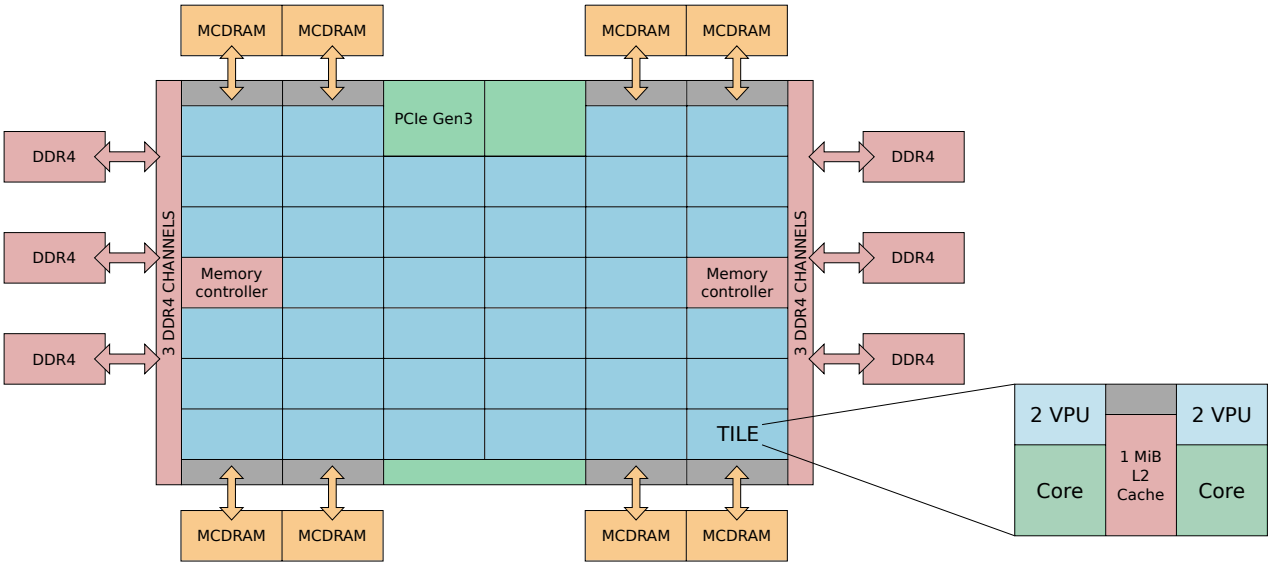

図 2.3　Knights Landing アーキテクチャ

の数は変わらず，プロセッサは最大 288 スレッドを同時処理可能である．L2 キャッシュは 512 KiB/core から 1 MiB/tile となるが，スレッドあたりの L2 キャッシュサイズは最低 128 KiB で，条件は KNC と変わらない．

チップ内には高バンド幅メモリとして MCDRAM が提供され，実測値で 490+ GB/s のメモリバンド幅を提供する[*1]．メモリコントローラを介してチップ外に最大 6 枚の DDR4 メモリが接続され，合計 115.2 GB/s の理論メモリバンド幅を提供する．KNL の MCDRAM は SW26010 と同様にスクラッチパッドキャッシュ，LLC としての利用の他に，メインメモリとしての利用も可能である．スクラッチパッドキャッシュとメインメモリの利用は “Flat モード” と呼ばれ，MCDRAM と DDR4 メモリがそれぞれ NUMA メモリとして扱われる．LLC としての利用は “Cache モード” と呼ばれ，Flat と Cache を組み合わせた “Hybrid モード” も選択できる．

チップのメッシュネットワークは，アクセスアルゴリズムに応じて “Quadrant,” “SNC (Sub-NUMA Clustering),” “Hemisphere” など複数のモードを選択可能だが，これらによる性能の違いはわずかしかないことが報告されている [13]．したがって，本研究では評価環境が提供する Quadrant での性能評価を実施する．

2.3　GPU (Graphics Processing Unit)

代表的な GPU には NVIDIA の Telsa や AMD の RadeonPro などがあるが，本章では HPC 用途におけるデファクトスタンダードとなっている NVIDIA GPU について述べる．図 2.4 に最新アーキテクチャである “Volta” における，GPU の演算コアである SM (Streaming Multiprocessor) の構成を示す [14]．NVIDIA GPU は Xeon に代表される x86 CPU に対し PCIe で接続され，カーネルの実行制御や

[*1] Intel は現在までに MCDRAM の理論メモリバンド幅を公開していない．

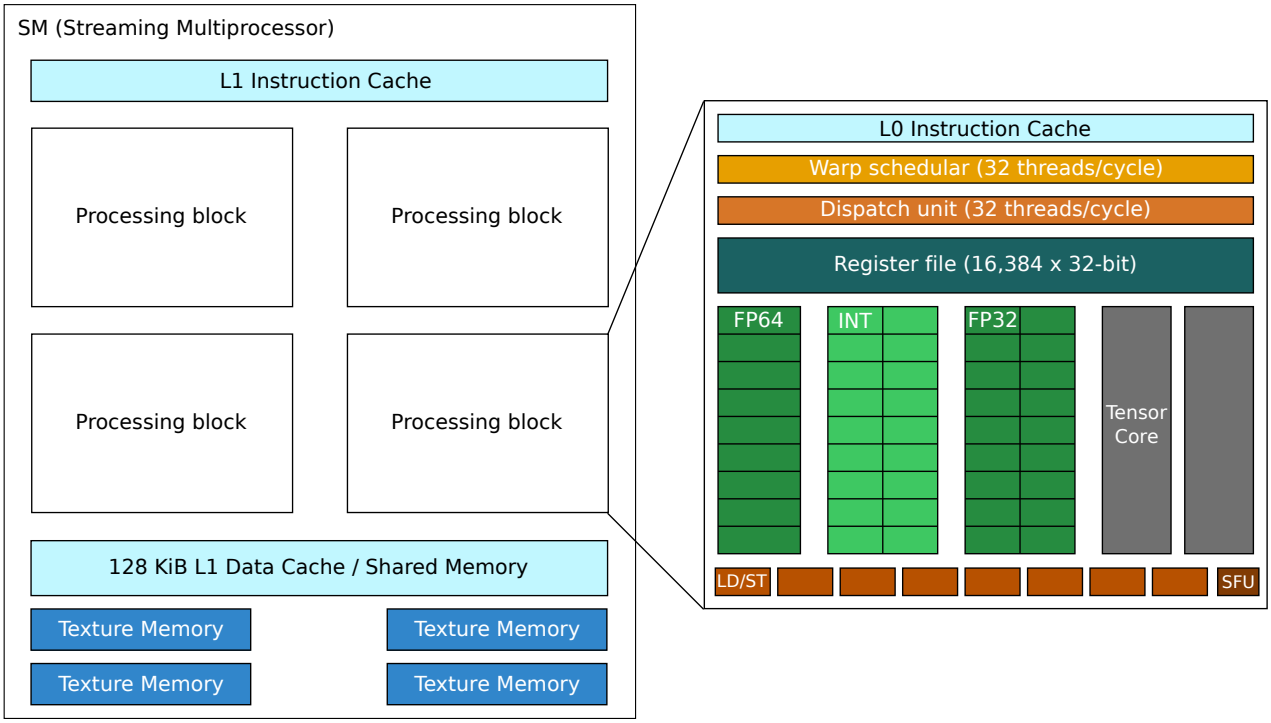

図 2.4 Volta アーキテクチャ (SM 構造)

計算データの初期化など，計算以外の制御をすべて CPU に委任する．したがって，NVIDIA GPU 単体での動作は不可能である．

NVIDIA が提供する GPGPU (General Purpose on GPU) コンピューティングのためのプラットフォームである CUDA (Compute Unified Device Architecture) では，計算の最小単位を “スレッド” と呼び，32 スレッドを “Warp” という単位にまとめ，Warp 単位での計算，メモリアクセスを行う [15]．複数のスレッドをまとめたものを “ブロック” と呼び，複数のブロックをまとめて “グリッド” と呼ぶ．グリッドは GPU に対し 1 個しか存在せず，グリッド内のブロックとブロック内のスレッドは 3 次元配置が可能である．NVIDIA はこの実行モデルを SIMT (Single Instruction Multiple Threads) と呼んでいるが，幅が 32 の SIMD と捉えることが可能である．

Volta アーキテクチャでは最大 84 個の SM が接続され，プロダクトの Tesla V100 ではうち 4 個が歩留まり改善のために利用され 80 個の SM が有効化される．各 SM は 64 個の 32-bit 浮動小数点数演算機, 32-bit 整数演算機と，32 個の 64-bit 浮動小数点数演算機を持つ[*2]．Warp 単位での命令スケジューリングを行う Warp スケジューラは，各 SM に 4 個が接続され，クロックあたり 1 Warp を制御できる．Warp スケジューラあたりで見ると，16 個の 32-bit 浮動小数点数演算機，32-bit 整数演算機と，8 個の 64-bit 浮動小数点数演算機が用意される．

メモリアクセスは，Volta アーキテクチャでは 1 Warp がまとめて実行されるが，以前のアーキテクチャでは Half-Warp (16 スレッド) 単位で処理される．Volta アーキテクチャを採用する Tesla V100 は高バンド幅メモリとして HBM2 を 16 GiB または 32 GiB 提供し，キャッシュメモリには SM 内で共有さ

[*2] 他に機械学習用の “Tensor-core” が実装されているが，本研究では使用しないため説明を省略する．

表 2.1　理論性能 [PFLOPS] と実性能 (HPL と HPCG) [PFLOPS] の比較

システム	Rank	システム分類	理論性能	HPL	HPCG
Summit	#1	アクセラレータ (GPU)	187.66	122.30	2.926
Sunway TaihuLight	#2	メニーコア CPU	125.44	93.01	0.481
Sierra	#3	アクセラレータ (GPU)	119.19	71.61	1.796
Piz Daint	#6	アクセラレータ (GPU)	25.33	19.59	0.486
Sequoia	#8	マルチコア CPU	20.13	17.17	0.330
Trinity	#9	メニーコア CPU	43.90	14.14	0.546
Cori	#10	メニーコア CPU	27.88	14.01	0.355
Oakforest-PACS	#12	メニーコア CPU	24.91	13.55	0.385
K computer	#16	マルチコア CPU	11.28	10.51	0.603

れる 128 KiB の L1 キャッシュ，プロセッサ全体で共有される 6 MiB の L2 キャッシュがある．CUDA GPU における特殊なメモリとして，L1 キャッシュメモリと共有される “シェアードメモリ” と，読み込み専用キャッシュの “テクスチャメモリ” が提供される．シェアードメモリは，ブロック内で共有されるワーキングメモリで，畳み込み演算でのスレッド間のデータ畳み込みや他スレッドへのデータ受け渡しなどに利用される．Volta では L1 キャッシュとシェアードメモリが 128 KiB の領域を共有するため，シェアードメモリを使う場合は L1 キャッシュが小さくなることに留意されたい．

2.4　理論性能と実性能の乖離: HPL と HPCG ベンチマーク

各最先端アーキテクチャの特徴をまとめると，各アーキテクチャは電力効率の向上のため，下記の方法によりアーキテクチャ単体の演算性能を増加させていることが分かる．

1. 演算コア数の増加によるスレッド並列性の増加
2. SIMD 幅の増加によるベクトル並列性の増加
3. メモリ階層の増加によるメモリバンド幅の改善

しかし，これらは同時に下記の問題を引き起こしている．

1. スレッド並列性増加: 巨大な並列処理可能領域の確保が必要となる
2. ベクトル並列性増加: ループ長が短くなり，演算のパイプライン化が困難になる
3. メモリ階層の増加: スクラッチパッドキャッシュの操作といったデータ移動処理が複雑化する
4. 演算コアの簡略化: 逐次処理性能が低下する

プロセッサや計算ノードの TDP や，システム全体に供給可能な電力の制限を考えれば，高性能計算システムの性能を向上させるには演算コア数を増やし，並列処理性能を増加することでしか解決できない．しかし上記の問題によって，実アプリケーションの開発の難易度は今後さらに上昇することが予想され，高性能計算システムへの最適化が必須となる．

表 2.1 に, 2018 年 6 月における TOP500 リストの上位から抜き出したシステムの理論ピーク性能, HPL (High-Performance Linpack) および HPCG (High-Performance Conjugate Gradient) ベンチマークの

表 2.2 本研究のターゲットアーキテクチャにおける HPL 性能 [GFLOPS]

システム	アーキテクチャ	プロセッサ	理論性能	HPL 性能
QURIOSITY	Skylake-SP	Xeon Gold 6148	1,024	992.1
Oakforest-PACS	Knights Landing	Xeon Phi 7250	3,046	1,993.3
DGX Saturn V	Pascal	Tesla P100	4,800	3,333.6
DGX Saturn V Volta	Volta	Tesla V100	7,000	4,053.0

性能を併記する [1, 16]．ここではシステム分類を簡便化し，メニーコアまたはマルチコア CPU，アクセラレータ型システムとだけ記載する．マルチコア CPU を持つ Sequoia および「京」コンピュータが理論性能比で 85–93% の HPL 性能を持つのに対し，メニーコア CPU システムは最大 74%，GPU システムは最大 77% と，従来のマルチコアシステムに比べて理論性能と実性能との間に乖離が見られる．

GPU などのアクセラレータを搭載したシステムでは，従来のシステムに対しメモリ管理や演算カーネルのスケジューリングなどアクセラレータの制御が必要となるため，これらが実行オーバーヘッドとなりやすい．Summit は 4608 ノードのシステムを Full-bisection Fat-tree network で接続し，PCIe gen 3.0 よりも高速なノード内インターコネクトである NVLink によって CPU と GPU 全てが接続されながらも，理論性能比に対して 65% しか得られていない．メニーコア CPU では，Sunway TaihuLight が理論性能比で 74% と，他のメニーコア CPU システムと比べれば高い効率を示しているものの，およそ 30 PFLOPS の理論性能を活かしきれていない．ネットワーク的に最も有利な Full-bisection Fat-tree network である Oakforest-PACS でも理論性能比で高々 54% 程度の実性能しか得られておらず，KNL システムにおける乖離はより深刻になっている．

現在の最先端アーキテクチャ，特に Intel と NVIDIA に代表されるコモディティのプロセッサは TDP を維持しつつ高い性能を達成するため，多くが動的なクロック調整機構を持っている．例えば KNL プロセッサでは，動的なクロック調整によって 100% の速度を保ったまま AVX-512 命令を処理できず，SIMD 幅やコア数を増加させることで向上させてきた理論性能は，その多くが達成困難なカタログスペックとなっている．NVIDIA GPU においても “NVIDIA GPU Boost” という名称で，TDP 制約に合わせてクロック周波数を動的に切り替えるため，同様の問題を抱えている．HPL はこの問題を評価するための指標として，十分に活用できると考えられる．

HPL は，開発当時において通信性能が演算性能に対し極めて高かったため，演算性能の測定に重点を置き，密行列を係数行列として持つ連立一次方程式の求解性能を測定する．密行列の求解は O (N^3) の計算量を持つため計算律速なベンチマークであり，HPL ではネットワークやメモリのバンド幅など，データ移動全般の通信性能が影響しない．演算性能よりも通信性能の進化スピードが鈍化し，通信性能がボトルネックになるアプリケーションが大多数を占めるようになると，理論演算性能どころか HPL で得られる演算性能すら達成が困難となった．

HPCG はメモリや通信といったデータ移動の性能に重点を置いたベンチマークで，疎行列を係数行列に持つ連立一次方程式の求解したときの演算性能を測定する．HPCG は連立一次方程式の求解法として反復法 (CG 法，Conjugate Gradient method) を用い，O (N^2) まで計算量が低下する．反復法は，メモリバンド幅ボトルネックとなる疎行列ベクトル積や，`MPI_Allreduce` や `MPI_Allgather` などの通信レイテンシがボトルネックとなる集団通信によって構成され，データ移動に関わる性能に実効性能が律速

される．HPL と HPCG を提案した University of Tennessee の Jack Dongarra は 2 つのベンチマークは性能のブックエンドだと述べており，HPL と HPCG をそれぞれ最高性能と最低性能として．実アプリケーションの性能はこの間に位置すると主張する．

本研究では HPL を達成可能な最大の実性能として扱い，理論性能比および HPL 性能比を併記によって本研究で得られる実性能の妥当性について明瞭な議論を行う．TOP500 の HPL 性能は，問題サイズなどの選択はシステムで得られる最高性能を登録できるため，TOP500 の記録がシステムが達成可能な最高性能とみなせる．2018 年 6 月の TOP500 の結果より，各アーキテクチャを採用したシステムにおけるプロセッサ 1 台あたりの HPL 性能を表 2.2 に示す．同表は TOP500 における各システムの HPL 性能より，CPU または GPU あたりの HPL 性能を算出している．GPU クラスタでは，HPL の主要計算である行列積が GPU で計算されるため，GPU クラスタの HPL 性能を GPU 性能と捉えることができる．“DGX Saturn V”, “DGX Saturn V Volta” は NVIDIA が運用するプライベートスーパーコンピュータだが，GPU ベンダである NVIDIA が運用するため，HPL の結果は非常によく最適化されていると期待される．

Skylake-SP や Broadwell-EP など近年の Intel CPU のアーキテクチャでは，SIMD 命令である AVX, AVX2, AVX-512 を使用する場合に，基本とする動作周波数から約 0.3 GHz 低下した周波数での動作を最低限保証する，“AVX base clock” が定義されている．したがって，理論ピーク性能は基本的に AVX base clock から求めることが自然と考えられるが，TOP500 に記載されている理論性能 (“Rpeak”) は計算すると基本動作周波数から算出されており，注意が必要である．同機構は Turboboost と同じく，TDP 制約の中で最高性能を達成するための仕組みと言えるが，これにより理論性能は最早カタログスペック以上の意味を持たない．表 2.2 では，AVX base clock から理論性能を算出している．

第 3 章

高性能計算と計算科学のコデザイン

本章ではまず，高性能計算におけるハードウェア (計算機の開発) とソフトウェア (実アプリケーションの最適化) のコデザインについて，実際のシステムや実アプリケーションにおけるコデザインと，現在実施されているコデザインベースのプロジェクトを紹介し，コデザインの重要性と本研究の意義を述べる．本論文の考察では，それらが達成されているかについて議論する．

3.1　コデザインの実例

3.1.1　ハードウェアにおけるコデザイン – 計算機開発

コデザインによる計算機開発は，実アプリケーションの中でボトルネックとなる計算カーネルの高速化，通信コストの削減などを目的として行われ，“専用計算機” の開発を意味することが多い．GRAPE (GRAvity piPE) は，特定の実アプリケーションに特化した専用計算機の中で，特に成功した代表例のひとつと言える [17].

GRAPE は，東京大学で開発された，計算天文学における重力多体計算の高速化に焦点を当てた専用ハードウェアである．1989 年に開発された GRAPE-1 [18] は，天文学における重力多体計算は非常にマクロなスケールで実施されることから，大部分を 8-bit の浮動小数点数で保持し，加算や積算回数により精度が必要な計算は 16-bit または 48-bit の固定小数点数で計算することで，計算コストを削減している．また 8-bit 浮動小数点数計算は結果が $2^8 = 256$ 通り，加算や積算を行っても 2^{16} 通りしかないことから，計算をテーブルルックアップにより省略し，さらなる計算コストの削減を行っている．IEEE 754 に定義されない bit 幅を持つ浮動小数点数演算による計算コストの削減は，演算回路から自由に開発できる専用ハードウェアならではの手法と言える．その後，GRAPE は現在に至るまで 10 以上のバージョンが開発されている．

GRAPE-2 や GRAPE-4 など偶数番のバージョンでは，高精度演算を対象としてハードウェアが設計され，最大 64-bit 幅で計算上必要となる精度に圧縮した 38-bit, 32-bit，29-bit 幅の浮動小数点数演算により高精度化と高速化が行われた [19]．GRAPE-4 はプロセッサ単体で 523 MFLOPS の理論性能を持ち，合計 1692 台を接続した最高 884 GFLOPS のシステムにより，1995 年および 1996 年のゴードン・ベル賞を受賞した [20, 21].

その後も，現在までに GRAPE-9 までが開発され，第 7 世代にあたる GRAPE-7 以降は再構成可能ハードウェアとして注目されている FPGA (Field Programmable Gate Array) を中心として開発が行われている [22]．他にも GRAPE の開発をヒントに，同種の計算パターンを持つ分子動力学においても専用ハードウェアが構想され，GRAPE を分子動力学に応用した MDGRAPE-4 などが現在までに開発されている [23]．

PEACH2 (PCI Express Adaptive Communication Hub version 2) は，GPU などのアクセラレータ技術の要素技術におけるノード間アクセラレータの直接結合による直接通信，通信性能向上を目指した TCA (Tightly Coupled Accelerators) 機構のためのハードウェアである [24]．PEACH2 は特に GPU クラスタにおける GPU 直接通信をターゲットとした通信専用ハードウェアだが，TCA 機構のコンセプトそのものは GPU に限らずアクセラレータ全般としている．

PEACH2 ボードは FPGA を用いて実装されており，PCIe gen 2 デバイスとして GPU とともに接続され，PCIe 外部接続ケーブルを用いて隣接ノードを接続，ノード間直接通信を実現する．FPGA から GPU が持つメモリへの直接アクセスには，NVIDIA が提供する GDR (GPU Direct for RDMA) 機構が用いられる．InfiniBand を通じたノード間の GPU 通信にも GDR が用いられるが，PEACH2 は PCIe のプロトコルで通信できるため InfiniBand が必要とするプロトコル変換が不要で，より低レイテンシな通信が実現されている．しかしながら，PCIe 外部接続ケーブルの通信可能距離 (ケーブル長) や，当時の FPGA の制約もあり，直接接続可能な規模は計算ノード 16 台までに限られる．その後，PCIe gen 3 デバイスとして PEACH3 が開発され，通信帯域幅の向上が図られている [25]．

PEACH2 と PEACH3 は，アクセラレータ間通信の高速化のための専用ハードウェアとして開発が行われてきたが，FPGA をさらに活用するために PEACH2/PEACH3 を実装した FPGA チップ上の未使用の回路リソースを用いて，演算のオフロード，すなわち通信デバイスとアクセラレータを兼ねたハードウェアとしての利用も試みられている [26]．近年，高性能計算においても FPGA の利用が注目されている [27, 28, 29] が，GRAPE や PEACH2 はその先駆者と言える．

3.1.2 ソフトウェアにおけるコデザイン – 実アプリケーション最適化

コデザインに則ったソフトウェア開発は，特に既存の実アプリケーションをどのように最先端アーキテクチャとシステムに実装するか，実行効率を高められるかに焦点が当てられる．ゴードン・ベル賞は，最先端のスーパコンピュータにおける実アプリケーションの性能と社会へのインパクトを重視しており，実アプリケーションのコデザインの実例として最適である．

Sunway TaihuLight は現代のシステムにおいて最も重要視されている Byte/FLOP が他のトップレベルのスーパコンピュータに比べ遥かに劣っており，実アプリケーションで高い実効性能を得るのは並大抵のことではない．このことは，HPL と HPCG で得られる演算性能からも一目瞭然である．しかし [8] では，同システムにおいて FD (Finite Difference) 法による地震シミュレーションを最適化し，最高で 18.9 PFLOPS，HPL 性能比で約 20% の効率を達成し，2017 年のゴードン・ベル賞を受賞した．

計算においては，SW26010 の各演算コアに 64 KiB しかない LDM (スクラッチパッドキャッシュ) に DDR3 メモリから如何にデータを供給するか，また書き戻すかが重要となる．最適化の多くは，システムの低い Byte/FLOP の克服を目的としており，線形方程式を解く CG 法における LDM ブロッキング，

Athreads による 2 次元分割を用いたスレッド間のメモリアクセス最適化といった複数の並列化階層における最適化が行われている．LDM へのデータ転送では，計算領域と袖領域の合計サイズを LDM 容量に合うように調整し，非同期コピーを用いた計算とのオーバーラップにより DDR3 メモリの長いアクセスレイテンシを隠蔽している．また，SW26010 プロセッサの計算性能を活かし on-the-fly のデータ圧縮により，計算性能を 15.2 PFLOPS から 18.9 PFLOPS に引き上げている．on-the-fly のアルゴリズムは，16-bit 浮動小数点数として LDM にコピーしたデータを 32-bit 浮動小数点数としてレジスタに展開しながら計算し，16-bit に圧縮しながら LDM に書き戻すことで実現するが，16-bit の圧縮フォーマットは，実際の値を 1–2 の範囲で正規化することで指数部を持たずに符号 1-bit，残り 15-bit をすべて仮数部に割り当てている．

[30] では第一原理計算による実時間密度汎関数法 (RSDFT, Real-Space Density Functional Theory) について，「京」コンピュータ上での最適化を行い，当時としては最大規模の 10 万原子を超えるシリコンナノワイヤの電子状態計算を達成し，2011 年のゴードン・ベル賞を受賞した．

「京」コンピュータの CPU である SPARC64 VIIIfx プロセッサは，8 の物理コアを持つマルチコア CPU で，各コアは 128-bit 幅の SIMD 演算を提供している．メインメモリがプロセッサあたり 16 GiB と，「京」コンピュータ以前のシステムに比べるとコアあたりのメモリサイズは小さく，OpenMP を用いたスレッド並列化と，data locality への最適化が要求されている．RSDFT は，CG 法，グラムシュミット法による正規直交化，部分対角化の 3 つの計算が大部分を占めており，特に O (n^3) の計算量を持つグラムシュミット法と部分対角化に焦点を当てて最適化が行われている．ノード内並列について，グラムシュミット法と部分対角化はどちらも三角行列計算になるため，スレッド並列化による最適化が困難である．そこで，部分行列と部分三角行列に分割し，部分行列を O (n^3) の計算量かつ data locality への最適化が容易な行列積で計算し性能向上を図った．MPI による並列分散では，計算式のさらなる分割についても議論されている．計算対象となる電子の波動関数は，電子軌道，エネルギーバンド，実空間格子点の 3 つで構成され，MPI の並列化は電子軌道にのみ行われていた．しかしエネルギーバンド間には依存性がなく，MPI の sub-communicator を導入し，主たる計算カーネル内における通信コストの削減と，並列性の確保を行っている．

3.2　現在進行中のコデザインベースプロジェクト

3.2.1　FLAGSHIP 2020

日本では「京」コンピュータの次のフラッグシップシステム “ポスト「京」コンピュータ” の開発プロジェクトが進行し，文部科学省の推進事業として “FLAGSHIP 2020” プロジェクトを展開している [7]. FLAGSHIP 2020 では，理化学研究所計算科学研究センター (R-CCS, 旧計算科学研究機構) 主導のシステム開発と，各大学・研究機関が主導する 9 つの重点課題および 4 つの萌芽的課題で構成されたアプリケーション開発が各々のフィードバックを踏まえ，システム開発とアプリケーション開発それぞれのコデザインを実施し，“性能と使いやすさの両立” を掲げている．同プロジェクトのコデザイン対象はハードウェアおよびアーキテクチャ，ミドルウェア (システムソフトウェア)，アルゴリズムと数値演算ライブラリを挙げており，重点課題各 9 分野から選択したアプリケーションを代表として，これらのコデザイン

表 3.1 ポスト「京」重点課題とターゲットアプリケーション

課題名	アプリケーション
生体分子システムの機能制御による革新的創薬基盤の構築	タンパク質の分子動力学計算
個別化・予防医療を支援する統合計算生命科学	ゲノム解析
地震・津波による複合災害の統合的予測システムの構築	地震シミュレーション
観測ビッグデータを活用した気象と地球環境の予測の高度化	全球雲解像シミュレーション
エネルギーの高効率な創出、変換・貯蔵、利用の新規基盤技術の開発	分子科学計算
革新的クリーンエネルギーシステムの実用化	Large-eddy simulation
次世代の産業を支える新機能デバイス・高性能材料の創成	第一原理電子状態計算
近未来型ものづくりを先導する革新的設計・製造プロセスの開発	Computer aided engineering
宇宙の基本法則と進化の解明	Lattice QCD 計算

が進められている [31]．各分野と，選定されたアプリケーションを表 3.1 に示す．なお，同表ではアプリケーション名ではなくアプリケーションの計算内容について記載する．それぞれのアプリケーションは，計算や通信のパターンの違いだけではなく，ストレージ性能の重要性も大きく異なる．

2018 年 7 月現在，設計段階が終了し現在は開発段階に入っており，すでに試作機の公開も行われている．CPU は ARM v8 による 64-bit アーキテクチャを採用し，48 core (12 core × 4 NUMA) + 2 or 4 assistant core 構成で，ARM SVE (Scalable Vector Extension) を用いた 512-bit SIMD 命令の提供が予定されている．また，倍精度浮動小数点数演算に対する Byte/FLOP は，高バンド幅メモリを用いて約 0.4 が予定されている．「京」コンピュータと比較すると 6 倍増の計算コア，4 倍増の SIMD 幅，約 0.8 倍減の Byte/FLOP と，「京」コンピュータよりも高並列なアプリケーションが求められている．メインメモリのサイズが公開されていないが，現在高メモリバンド幅として主流な HBM2 がプロセッサあたり 16 GiB や 32 GiB で提供されていることを考えると，「京」コンピュータと同程度しか用意されず，メモリ制約はより厳しいことが予想される．

上記も含めた各問題に対し，理化学研究所では 4 つのチームによって下記の研究開発が進行している．

システムソフトウェア開発

メニーコア CPU に最適化した OS カーネルの開発，高効率な MPI 通信ライブラリの開発，ファイル I/O ミドルウェアの開発

アーキテクチャ開発

CPU シミュレータ開発による設計アーキテクチャ検証，アプリケーション最適化の先行実施と準備，プログラミング環境の整備

アプリケーション開発

ターゲットアプリケーションを用いた最適化，システム評価用のアプリケーションベンチマーク確立，ライブラリやフレームワークなどのアプリケーション基盤技術の開発

コデザイン推進

ターゲットアプリケーションの性能見積もり，ハードウェアとシステムソフトウェアも絡めた性能改善，ハードウェアの性質に合わせた新しいアルゴリズムの検討，DSL (Domain Specific Language) やアプリケーション開発フレームワークの開発

表 3.2　Exascale Computing Project stack

Application Development	
Chemistry and Materials Applications	Lattice-QCD，分子動力学等
Energy Applications	風力発電，原子炉，粒子加速器シミュレーション等
Earth and Space Science Applications	超新星爆発，地震，気候モデルシミュレーション等
Data Analytics and Optimization Applications	都市モデル，Deep Learning によるガン解析等
Co-Design	数値アルゴリズム，ソフトウェアフレームワーク開発
Software Technology	
Programming Models and Runtimes	プログラミングフレームワーク開発
Development Tools	性能最適化ツール，コンパイラ開発等
Mathematical Libraries	数値計算ライブラリ開発
Data and Visualization	可視化，ファイル I/O ミドルウェア開発等
Software Ecosystem and Delivery	OS 開発

理化学研究所ではポスト「京」コンピュータだけでなく広範に使えるソフトウェアスタックとして，“McKernel” や “XcalableMP” などのシステムソフトウェアや開発環境を提供している．メニーコア CPU に向けた軽量カーネル “McKernel” は，高性能計算システムで用いられる Linux カーネルと実アプリケーションの間に挟まれる，軽量 OS である．McKernel によって，実アプリケーションはシステムコールや通信，割り込み処理などの OS ノイズの影響を低減し，実アプリケーションの本来の性能を引き出すことが可能である [32]．“XcalableMP (XMP)” は，高並列システムにおける高性能高生産性プログラミング環境のために開発されている，ディレクティブベースのプログラミング言語である [33]．XMP は従来 MPI で記述していた分散並列環境をディレクティブベース言語に置き換えるもので，スレッド並列 API である OpenMP との連携が可能である．アクセラレータ用である OpenACC と連携した “XcalableACC” も提案され，アクセラレータ用通信機構やディレクティブベース言語間の連携についても議論されている [34]．

3.2.2　Exascale Computing Project

アメリカでは，ORNL (Oak Ridge National Lab.) の Titan と LLNL (Lawrence Livermore National Lab.) の Sequoia に対し 50 倍以上の性能を達成できるエクサスケールシステムを持続可能なエコシステムとして 2021 年に実現することを目標として，ECP (Exascale Computing Project) を進めている．ECP は FLAGSHIP 2020 と同様に，複数の分野でアプリケーションの開発とコデザインを推進し，人材育成にも主眼を置いている．

ECP は大きく “Application Development”，“Software Technology”，“Hardware and Integration” の 3 分野に分かれ，各分野で複数の開発チームが研究開発を進めている．“Application Development” と “Software Technology” の 2 分野で行われている研究開発について，表 3.2 に示す [35]．ECP の一貫として，多くのプログラミングフレームワークやコンパイラなどのソフトウェアスタックの開発が進められており，軽量スレッドライブラリの Argobots [36]，PGAS (Partitioned Global Address Space) 言語拡張である UPC++ [37]，LLVM ベースコンパイラの Flang [38] などがある．

ECP では，複数のシステムを同時並行して開発・運用しシステム規模の拡大を細かくすることで，エクサスケールシステムへの円滑な移行を計画しており，「京」コンピュータから短いステップでのエクサスケールシステムへの移行が計画されている FLAGSHIP 2020 とは対照的である．現在，ORNL と LLNL が IBM Power9 + Volta GPU クラスタとしてそれぞれ Summit と Sierra の運用を開始したが，今後 Lawrence Berkeley National Lab. の NERSC-9 や Los Alamos National Lab. と Sandia National Lab. の Crossroads などの “Pre-exascale system” の導入を計画している．エクサスケールシステムには，Argonne National Lab. の Aurora (A21) を初めとした 3 システムが計画されており，Aurora では Intel と協力しメニーコア CPU システムの構築を計画している．LLNL では，エクサスケールシステムとして Sierra の次に El Capitan を計画している．システム概要はこれまでに公開されていないが，現システムである Sierra が GPU クラスタであること，ECP がエコシステムを推進していることから，El Capitan もアクセラレータ型システムと推察される．

3.3 コデザインの必要性と本研究の意義

第 2 章にて述べたように，今日の大規模高性能計算システムには，ハードウェアが理論性能と呼んでいるものと達成可能な実性能の間に大きなギャップが存在する．計算律速である HPL ですら，実際には理論性能に対して 50–70% 程度の性能しか得られない．また，高い並列性を持つアルゴリズムの開発，計算ノード間の通信の削減，アクセラレータを用いた新たな並列化モデル・ミドルウェアの導入など，高性能計算システムから実アプリケーションへの要求は年々複雑化し，解決が困難となっている．これらの問題は，より高性能な計算システムを開発，利用促進をしなければならない高性能計算と，高性能計算システムを用いて科学を探求する計算科学の双方が協力して解決しなければならない．その手段こそコデザインにほかならず，“高性能計算と計算科学のコデザイン” は今後必須の開発手法になると考えられる．

実アプリケーションのコデザインは大局的に 2 つのアプローチが考えられる．

1. 実アプリケーションの実装を積極的に書き換え，ハードウェアに最適化する
2. 実アプリケーションの開発とハードウェアへの最適化を同時並行で実施する

高性能計算側から考えた時のコデザインは，その多くが前者を指していると考えられる．前者は達成可能な実性能を最大限引き出すことが期待されるが，実アプリケーションの積極的な書き換えによって開発が妨げられる可能性がある．後者は実アプリケーションの開発とハードウェアへの最適化の両立を目指すが，実効性能を引き出すのは前者に比べて困難である．どちらの視点も非常に重要だが，高性能計算からは制約条件と達成可能な実性能の提示しかできず，コデザインにおいては計算科学との議論と選択が必要である．近年，国家予算で行われるプロジェクトの一環で実施されるソフトウェア開発は，各国が成果物を OSS (Open Source Software) として公開を求める傾向にあり，Linux に代表される OSS の持続可能なエコシステムが重要視されている．また，現在進行している各コデザインベースプロジェクトは使いやすさやエコシステムに注力しており，各々が協調，すなわち協力して行うことが重要であると強調している．したがって，本研究では “実アプリケーションの開発とハードウェアへの最適化を協調する” ことに着目したコデザインについて議論する．

実アプリケーション開発は，PDCA サイクルに近い図 3.1 のような開発サイクルが考えられる．順序

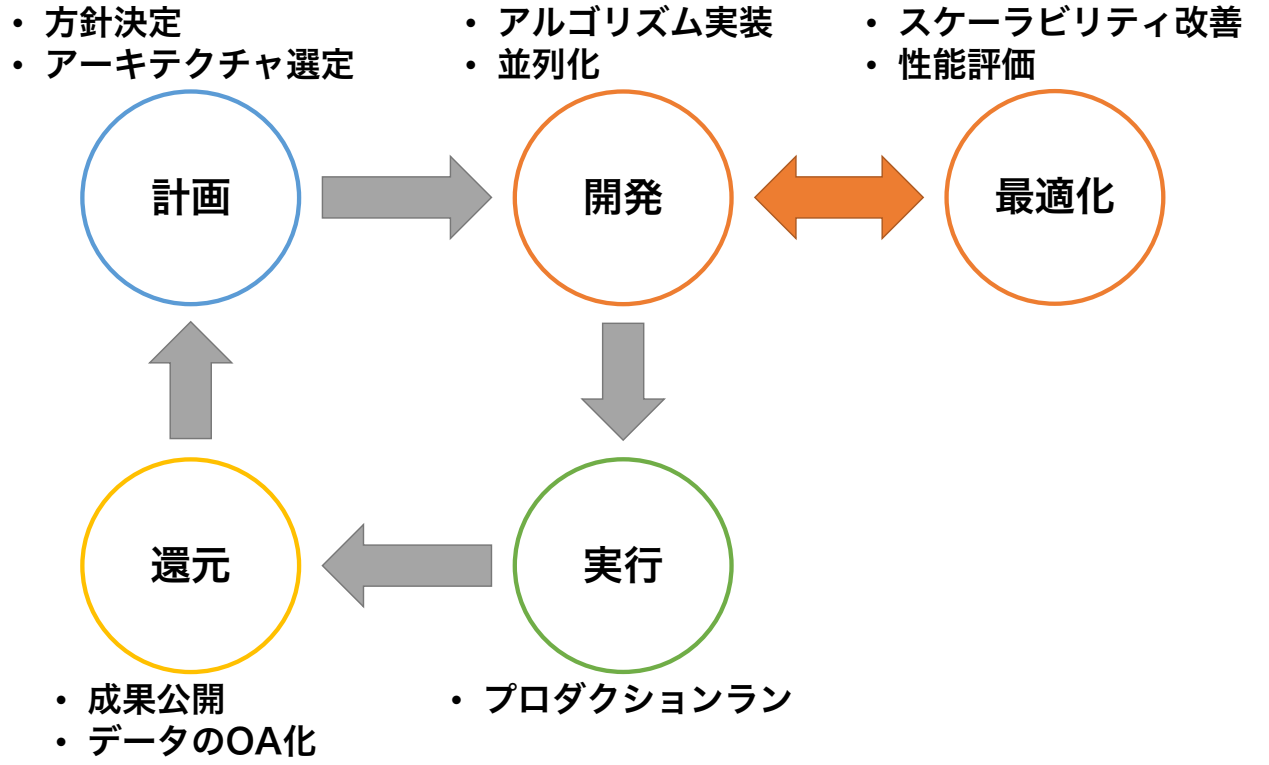

図 3.1　実アプリケーションの開発サイクル

として，何を開発するか，どのようなシステムを想定するかといった計画を行い，計画に沿って実アプリケーションの開発と最適化を反復実施する．十分な計算性能が得られる目処がついたら，プロダクションランを行って成果やデータを公開し，得られた結果や問題点から次の目標と方法を計画し，開発を実施する，と進んでいく．この中で，例えばプロダクションランで十分な性能が得られなかったとき，一度立ち戻って開発と最適化を反復実施したり，計画から修正することも考えられる．

高性能計算と計算科学によるコデザインで何が重要となるかを考えるに，もちろん達成可能な実性能について議論が必要だが，持続可能な実アプリケーション開発を高性能計算はどのようにサポートできるのか，各アーキテクチャを実際に用いた時にコデザインにとって何が起こるのか，実性能に限らない評価が必要と考えられる．高性能計算と計算科学による実アプリケーションの協調開発を実現するために，これらの議論により手法の体系化につなげることが必要である．

3.4　本研究におけるコデザインの定義と評価指標

協調開発を考えると，すでに述べた通り “実アプリケーション開発を妨げない” ことが重要である．実アプリケーションの最適化に関する研究は，その多くが “選択可能な最大限の最適化” を行っており，次世代計算機への可搬性の議論はほとんど行われていない．したがって，実アプリケーションを構成する各計算に対し，最適化水準を考える必要がある．最適化水準とは，ハードウェアの特性を理解した上で，SIMD 命令によるベクトル演算の最適化，CUDA による GPU 実装などの高度かつ最大限の最適化が可

能なのか，それとも計算科学がアクセス可能な範囲にとどめるべきなのか，を意味する．

成果物が他のシステムへどのように移行できるのか，可搬性の議論も非常に重要と考えられる．導入されたハードウェアのために，開発言語が変更される場合も考えられるが，本研究では実アプリケーション開発を妨げないことを重要視するため，実アプリケーションの基本構造は維持されるべきである．

以上より，実アプリケーションのコデザインについて以下の通り定義する．各項目が，本研究においてどのように遂行されるのかは，第 4 章にて述べる．

1. 実アプリケーションの構造の維持した上で，計算機への最適化が行われる
2. 実アプリケーションの開発は最適化と協調して行われ，それを妨げない
3. 最適化はコードの重要性と変更頻度を踏まえて，その水準を決定する

第 7 章では，上記の要件定義から実アプリケーションのコデザイン達成の可否について，下記の項目について議論する．

1. 対象アーキテクチャが達成可能な実性能
2. 最適化による実アプリケーション開発への影響
3. 成果物が次世代システムへ適用できるか

1 はシステムやプロセッサの絶対性能や HPL 性能からの妥当性を検証し，最適化について定量的評価を行う．2 はどの程度の最適化コードが必要となったか，またどのような書き換えが必要となるかを定性的に評価する．3 は，本研究で実施した最適化が現在のメニーコア CPU や GPU だけでなく，今後開発されるアーキテクチャやシステムに対し適用できるか，定性的に評価を行う．

本研究では，第 5 章および第 6 章でメニーコア CPU と GPU に対し，実アプリケーションの最適化と性能評価を行う．最適化と性能評価を踏まえ，第 7 章でコデザインに必要な要素，また各最先端アーキテクチャがコデザインにどう影響するか，などの議論を展開する．

第 4 章

ARTED: 電子動力学シミュレータ

本章では，実計算科学アプリケーションとして取り上げる電子動力学シミュレータ ARTED の概要，計算や並列化方法，最適化を行うカーネルとその戦略について述べる．

4.1　概要

ARTED (Ab-initio Real-Time Electron Dynamics simulator) は，筑波大学計算科学研究センターにて開発されている，光と物質の相互作用の第一原理計算を目的とした電子動力学シミュレータである [3]．同シミュレータは，パルス光のもとでの電子動力学計算に加え，電子動力学と電磁気学の 2 つの方程式を同時に解くマルチスケール計算が可能である．電子動力学では，電子軌道に対する時間依存 Kohn-Sham (TDKS, Time-Dependent Kohn-Sham) 方程式において，実時間・実空間法を用いて電子の波動関数の記述及び求解を行い，光電磁場では，電磁気学の基礎方程式であるマクスウェル方程式を時間領域差分 (FDTD, Finite Difference Time-Domain) 法により解く．図 4.1 に，3 次元マクスウェル方程式と TDKS 方程式を解く際に用いる空間格子について示す．

例えば，実験での観測が非常に困難となるアト秒 (10^{-18} seconds) 空間において，ARTED によって光の周期よりも短い時間での電子動力学シミュレーションを可能とし，強レーザー光による電子の励起過程やエネルギーバンド上の電子の運動のメカニズムを解明した [39, 40]．高強度超短パルス光を用いた加工技術は，質の高い微細加工や難削材料への適用可能性により，応用上重要な加工技術として精力的に研究が進められている [41]．ARTED によって，これまで経験的なパラメータを用いた研究にとどまっていたレーザー加工の初期過程に対して，TDKS 方程式とマクスウェル方程式を組み合わせることで正確かつ予測可能なシミュレーションの提供が可能で，応用上も加工の精度や効率を向上が期待される．

よく知られているように，第一原理計算は非常に巨大な計算量を持ち，「京」コンピュータや Sequoia BlueGene/Q など大規模システムを利用しても数万原子を持つ電子状態計算が限度 [4, 5] で，時間スケールでは非常に短い時間 (ピコ秒, 10^{-12}) でのシミュレーションが現実的に可能な範囲にとどまっている．電子動力学では高い計算性能を持つスーパーコンピュータが依然として求められており，ARTED は「京」コンピュータや Xeon CPU クラスタなどをターゲットとして開発が行われているが，メニーコア CPU や GPU といった最先端アーキテクチャの活用が必要とされている．

ARTED はシミュレーションの初期条件として，RSDFT と同様の方法を用いて基底状態を求める

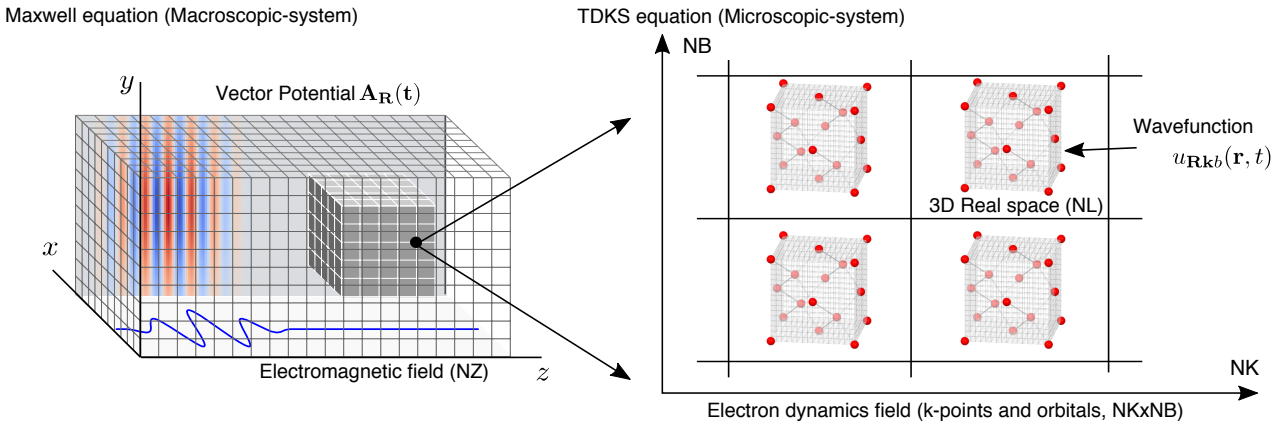

図 4.1　ARTED の計算領域: 3 次元マクスウェル方程式 + TDKS 方程式

[42]．ただし，RSDFT が 1,000–100,000 原子といった大規模な系を対象としているのに対し，ARTED ではシミュレーションの特性から 10–100 原子程度の小規模なセルを非常に多くの個数計算する必要がある．ARTED は時間発展が計算時間の大部分を占め，その初期状態である基底状態を求める計算時間は相対的に非常に短い．電子の波動関数の時間発展は，時間発展演算子の 4 次のテイラー展開によって計算され，1 ステップあたりに波動関数に対する 4 回のハミルトニアンの演算が発生する．時間発展はおおよそ 10,000–100,000 ステップ行われるため，ARTED では時間発展計算が支配的であり，大部分は後述するステンシル計算に費やされる．

大規模系を対象とする RSDFT は，実空間を domain decomposition により分割し MPI で並列計算を行うため，隣接する MPI プロセス間で袖領域交換が必要となり，通信の隠蔽が大きな課題となっている．一方，ARTED は TDKS 方程式において実空間ではなく，より大きな並列化可能な空間であるブロッホ波数空間を MPI で並列計算し，実空間の分散並列化は行わない．1 つの波数に属する実空間計算は，計算ノード 1 台が持つ計算能力とメモリ容量のみで十分に実施可能である．

実空間を分割した際に発生する袖領域の交換が不要な代わりに，軌道関数から密度分布を得るために全ての波数空間上の実空間データについて，`MPI_Allreduce` 通信を用いて計算結果を束ねる必要がある．マクスウェル方程式では FDTD 法による隣接格子間の通信のみを行うため，相対的に TDKS 方程式が高い通信コストを持つ．TDKS 方程式が持つ 3 次元実空間の 1 個あたりのサイズは RSDFT に対し非常に小さく，1 ステップあたりの計算時間が 10^{-6} [sec] レベルに対し通信は 10^{-9} [sec] レベルのオーダである．したがって，ARTED は RSDFT に対して通信コストが低く大規模並列システム向けのアプリケーションと言える．

4.2　計算・並列化方法

4.2.1　マルチスケール計算

図 4.1 に示すように，ARTED のマルチスケール計算では 2 つの方程式を解くが，マクスウェル方程式を離散化したときの空間格子を “マクロ格子点 (`NZ`)” と呼び，TDKS 方程式は下記の 3 つのパラメータ

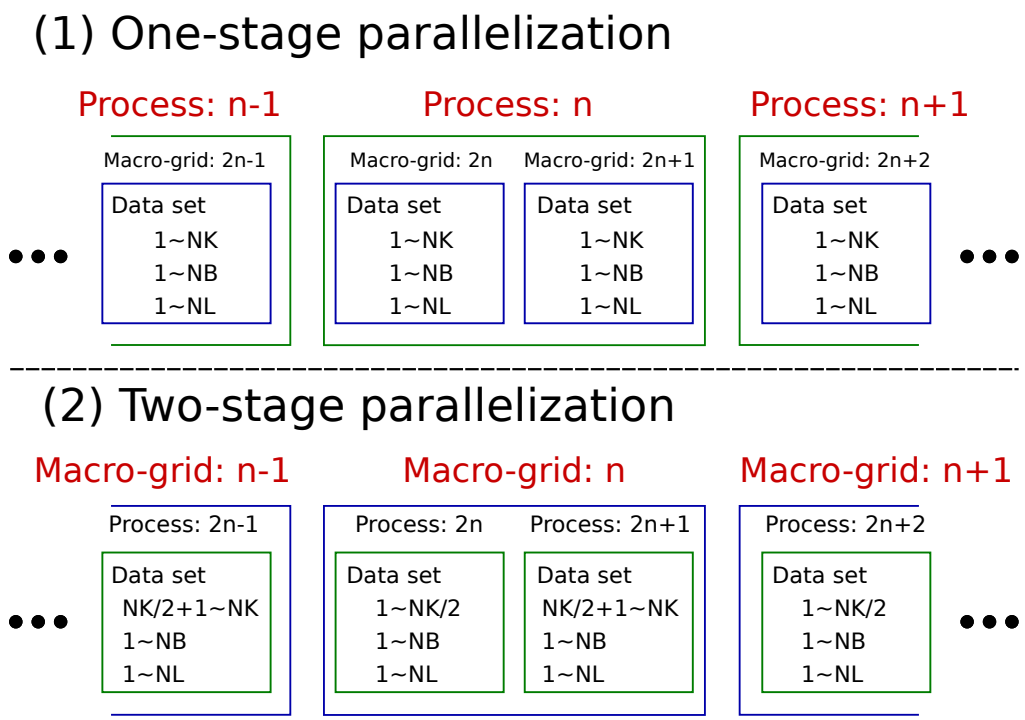

図 4.2　MPI の並列化方法

で構成されている．

- ブロッホ波数空間格子 (NK)
- バンド (NB)
- 3 次元実空間格子点 (NL = (NLx, NLy, NLz))

2 つの方程式を解くため，MPI コミュニケータがそれぞれの方程式に対し定義されるが，実行時の MPI の並列化方法は 2 つに分かれる．

- **複数**のマクロ格子点をまとめ，**1 つ**の MPI プロセスで**複数**の TDKS 方程式を解く
- **1 個**のマクロ格子点に対し，**複数**の MPI プロセスで **1 個**の TDKS 方程式を解く

図 4.2 に，2 つの並列化方法を図示する．この図では 2 個のマクロ格子点を 1 MPI プロセスで解く場合を (1)，2 MPI プロセスで 1 個の TDKS 方程式を解く場合を (2) に示している．マクロ格子点 1 つあたりのデータサイズが小さい場合，複数のマクロ点を束ねて計算することで，より広い物理空間・より大きい物質でのシミュレーションを実行できる．マクロ格子点を 1 つの MPI プロセスが複数持つため，各 MPI プロセスは複数の閉じた系 (TDKS 方程式) を独立に計算する．(1) の場合，TDKS 方程式を計算している間，各 MPI プロセスは通信なしに計算を行えるが，マクスウェル方程式の計算時に TDKS 方程式のデータを全 MPI プロセス間で共有するための通信がどちらのパターンにおいても必要となる．(1) のパターンでは，MPI のコミュニケータはグローバルに 1 つだが，(2) のパターンでは 2 つのコミュニケータを要する．マクロ格子点 1 つあたりのデータサイズが大きい場合，計算時間や計算ノードのメモリサイズなどの関係から，1 個のマクロ格子点を分割し 1 個の TDKS 方程式を複数の MPI プロセスで計算

する．TDKS 方程式の MPI 並列化は波数空間 (NK) を分割するため，TDKS 方程式の計算中，1 個のマクロ格子点を計算する MPI プロセス群を MPI のサブコミュニケータとして定義し，サブコミュニケータ間で波数空間を束ねる通信を行う必要がある．ただし，3 次元実空間格子は分割されないため，どちらの並列化方法においても袖領域交換は発生しない．

TDKS 方程式において，1 個の式を計算する MPI プロセス数を NP とすると，各 MPI プロセスでは (NK/NP)×NB 個の電子の波動関数（3 次元実空間格子点: NL）を，OpenMP を用いてスレッド並列化する．各 3 次元実空間は独立に存在しており，ステンシル計算は各 OpenMP スレッドが独立かつ逐次的に行う．時間発展計算中の通信は，1 個の TDKS 方程式を計算する MPI プロセス間と，マクスウェル方程式の全格子点 (全 MPI プロセス) 間の 2 つが必要となる．どちらも MPI_Allreduce で，最大でサイズ NL の倍精度浮動小数点ベクトルの総和を行う．前者は TDKS 方程式の波数空間を束ねる通信，後者はマクスウェル方程式の袖領域に相当する通信である．袖領域交換は通常 1 対 1 通信が用いられるが，1 対 1 通信にすることによって起こる計算の煩雑化を防ぐため，MPI_Allgather に該当する通信を MPI_Allreduce で行っている (各プロセスが担当する領域以外をゼロクリアしておくことで結果的に MPI_Allgather と等価になる)．したがって通信サイズはマクロ格子点数に相当する．しかし，マクロ格子点数は大規模システムである Oakforest-PACS の全系を用いても最大 2^{15} 個で，通常は MPI プロセス数やノード数に等しく非常に少ないため，ボトルネックとはなっていない．

4.2.2 ハミルトニアン計算

波数空間格子の規模はシミュレーション対象や要求精度により異なるが，最終的には各実空間格子点に対するステンシル計算の性能が支配的となる．実空間格子のサイズは，シリコン結晶 (Si) を対象とした場合 $(16, 16, 16)$，α クォーツ (SiO_2) を対象とした場合 $(20, 36, 50)$ がアプリケーションの計算精度上の最小値となる．

計算領域である電子の波動関数は倍精度複素数で，周期境界条件による 25 点ステンシル計算が行われる．電子軌道の時間発展計算に 4 次のテイラー展開法が用いられているため，1 ステップの時間発展に 4 回のハミルトニアン計算が必要となる．ハミルトニアン計算の流れについて，図 4.3 に概略を示す．同計算は主にステンシル計算と擬ポテンシャル計算で構成され，異なるメモリアクセスパターンを持つ [3]．

まず，各スレッドの一時計算領域 (tmp_in) に，計算する実空間をコピーする．コピーした一時計算領域上でステンシル計算と擬ポテンシャル計算を行い，計算結果を格納した配列 (tmp_out) を用いて電子軌道配列 Zu を更新する，これを 4 回繰り返す．前述の通り，波数空間並列で，各実空間を逐次的に計算するため OpenMP を用いて波数空間およびバンドのループ両方を並列化する．

1 個の実空間の計算は OpenMP の 1 スレッドで行われるため，各スレッドは 1 回の時間発展で 4 回のステンシル計算が含まれるハミルトニアン計算を逐次的に行い，複数個の実空間の計算を行う．各実空間は独立であり，1 回の時間発展で行われる 4 回のステンシル計算において OpenMP のスレッド同期または MPI による通信が発生しない．したがって，本研究ではステンシル計算を行う 1 個の実空間を，計算のワーキングセットまたはドメインと呼ぶ．

擬ポテンシャル計算はステンシル計算とは異なり，Scatter/Gather のメモリアクセスを繰り返して実空間上に存在する原子相当の格子点のみを更新するアルゴリズムである．同計算は，物理的な直感によっ

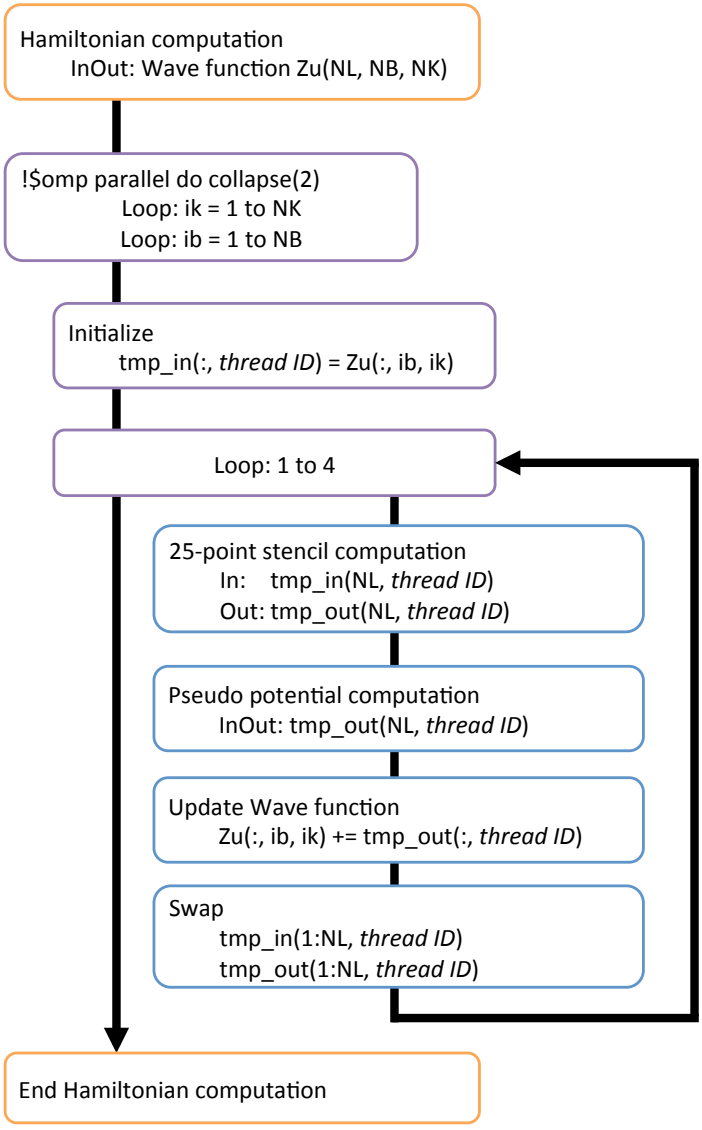

図 4.3　ハミルトニアン計算全体の流れ

て実装される計算量削減アルゴリズムであるため，最適化が非常に困難である [43]．ハミルトニアン計算はステンシル計算のコストが 8 割以上を占めており，ステンシル計算の最適化が極めて重要である．

4.2.3　ステンシル計算

図 4.4 に 25 点ステンシル計算のメモリアクセスパターンを示し，図 4.5 に ARTED のステンシル計算のオリジナル実装を示す．図 4.5 に示す計算は，図 4.3 の “25-point stencil computation” で行われる．ただし，Fortran では一般的に配列のインデックスが 1 で始まる 1-origin だが，最適化のため 0 始まりの 0-origin で受け取っている．

図 4.5 において，`A`，`B`，`C`，`D` は全て係数である．実空間格子は `E`，`F` に格納され，`E` が入力データ，`F`

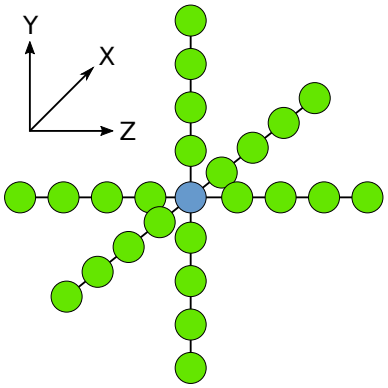

図 4.4 25 点ステンシル計算のメモリアクセスパターン

```
integer,intent(in)     :: NL
integer,intent(in)     :: IDX(-4:4,0:NL-1)
integer,intent(in)     :: IDY(-4:4,0:NL-1)
integer,intent(in)     :: IDZ(-4:4,0:NL-1)
real(8),intent(in)     :: A,B(0:NL-1)
real(8),intent(in)     :: Cx(4),Cy(4),Cz(4)
real(8),intent(in)     :: Dx(4),Dy(4),Dz(4)
complex(8),intent(in) :: E(0:NL-1)
complex(8),intent(out):: F(0:NL-1)
complex(8),parameter   :: zI = (0.d0, 1.d0)
integer     :: i
complex(8) :: v(3),w(3)

do i=0,NL-1
  ! x computation
  v(1)=Cx(1)*(E(IDX(1,i))+E(IDX(-1,i))) + Cx(2)*(E(IDX(2,i))+E(IDX(-2,i)))&
  &    +Cx(3)*(E(IDX(3,i))+E(IDX(-3,i))) + Cx(4)*(E(IDX(4,i))+E(IDX(-4,i)))
  w(1)=Dx(1)*(E(IDX(1,i))-E(IDX(-1,i))) + Dx(2)*(E(IDX(2,i))-E(IDX(-2,i)))&
  &    +Dx(3)*(E(IDX(3,i))-E(IDX(-3,i))) + Dx(4)*(E(IDX(4,i))-E(IDX(-4,i)))
  ! y computation
  ...
  ! z computation
  ...
  ! store
  F(i) = B(i)*E(i) + A*E(i) - 0.5d0*(v(1)+v(2)+v(3)) - zI*(w(1)+w(2)+w(3))
end do
```

図 4.5 ステンシル計算のオリジナル実装

が計算結果になり，図 4.3 中の tmp_in および tmp_out に相当する．実空間格子のメモリ配置は，Z 次元が連続，Y 次元が Z-stride，X 次元が ZY-stride となっている．配列 v，w に係数が異なる近傍点の加減算結果が格納されており，2 種類のベクトル演算が混在し，各次元で長さ 4 のベクトル演算が計 6 回行われる．周期境界を考慮したインデックス計算を省略するため，間接参照配列 IDX，IDY，IDZ に予め計算した近傍点のインデックスを格納している．必要な演算をまとめると，複素数の加減乗算，複素数のスカラ倍が必要となり，演算数は格子 1 点の更新に対し 158 FLOP，約 2.68 Byte/FLOP である．

既に述べたように，サイズ (NLx, NLy, NLz) のステンシル計算を各 OpenMP スレッドが独立で行うため，ステンシル計算は 1 個のタスクと見なすことができ，各タスクのワーキングセットは 1 MiB 未満と非常に規模が小さい．すなわち，現行のプロセッサが持つ L2 キャッシュに 1 個のタスクが，L3 キャッ

シュがあれば複数のタスクが収まる規模といえる．また各ワーキングセットはメモリ空間上独立しているため，巨大な 1 個の実空間を OpenMP で分割計算する場合に比べ，メモリアクセスコストが低い．加えて，各スレッドは 1 回のハミルトニアン計算のループで 4 回のステンシル計算を行うため，キャッシュミスの抑制が期待される．

ステンシル計算は計算科学のアプリケーションにおいて頻出する計算パターンのため，本研究で実施する最適化は他分野のアプリケーションに対しても効果があるものと期待される．しかし前述した同計算の特殊性から，キャッシュブロッキングに代表されるメモリ上不連続な領域へのアクセスコスト削減手法は ARTED のステンシル計算では効果が低いことが予測され，他分野アプリケーションへの適用時には考慮が必要となる可能性が高い．

4.3　コデザインに基づく実装と最適化

アプリケーションの計算上のボトルネックは，各プロセス各スレッドが独立に計算できるハミルトニアン計算で，特に 25 点ステンシル計算である．共同研究における議論により，ステンシル計算は現在の四次差分で十分な精度が安定して得られており，将来のコード変更の可能性が非常に低いことがわかった．そのためステンシル計算のメモリアクセスパターンが変わる可能性は低く，512-bit SIMD 命令を用いた手動のベクトル最適化や，CUDA を用いた GPU への最適化といった実装コストが高い最適化を行ったとしても，実アプリケーションの開発サイクルに影響を与えないと考えられる．

実アプリケーション全体は Fortran90 で開発されており，また主たる開発者の主な利用プログラミング言語が Fortran であることから，ステンシル計算を除く最適化はすべて Fortran 上で行う．計算内容の一部変更だけでなく，計算カーネル全体の書き換えが必要になった場合でも，アプリケーションユーザ (共同研究者，以下ではユーザ) が理解した上で実装可能なようにするためである．開発言語を固定しても，並列化方法はループ順序や計算順序など，細かいレベルで書き換えが必要になると考えられる．その多くはコンパイラが最適化しやすいコードへの最適化となるが，慣れ親しんだ言語であるため，別言語による最適化に比べユーザは理解しやすいと期待できる．

以上から，3.4 節に述べた定義，“実アプリケーションの構造の維持”，“同時並行で行われる開発を妨げない”，“最適化水準はコードの重要性と変更頻度に従う” の全てを満たす最適化戦略は以下の通りである．

1. ユーザが計算の修正や新しい実装を行えるように，主たる開発言語 (Fortran) は変更しない
2. ただし，実アプリケーションのボトルネックであるステンシル計算はユーザが実装詳細を把握可能かに依らず，高度な最適化を実施する
3. また，Fortran 上の最適化は，コンパイラによる最適化をアシストするように行う

しかしながら，GPU への最適化はそもそも新しい並列化モデルを追加する必要があるため，既存の OpenMP+MPI 並列化構造の範疇では十分な性能を達成できないことは容易に想像できる．したがって実アプリケーションの構造維持は非常に困難で，第 7 章では OpenMP+MPI 実装に対し，GPU 実装はどれだけの変更が加えられたかを評価する．

4.4 関連研究

第一原理に基づく密度汎関数法を計算するオープンソースのソフトウェアパッケージとして，OCTOPUS が開発，広く利用されている [44]．OCTOPUS は ARTED と異なり，波数空間に加え実空間についても分散並列化を行い，[45] では 4,096 ノード (16,384 演算コア) で構成された BlueGene/P システムにおける性能評価を報告している．また，OpenCL/C + Fortran のハイブリッドプログラミングにより，NVIDIA GPU のみならず AMD GPU にも対応した GPU 実装が行われている [46]．しかしながら，現在の最新アーキテクチャにおける性能評価は報告されていない．

別の実時間実空間法に基づくシミュレーションとして，“GCEED” が開発されており，「京」コンピュータを用いた 7,920 ノード (63,360 演算コア) の大規模計算の性能評価が報告されている [5]．大規模シミュレーションにおける性能評価では，LLNL (Lawrence Livermore National Laboratory) に設置された Sequoia BlueGene/Q を用いた大規模計算において，98,304 ノード (1,572,864 演算コア) を用いて 8.75 PFLOPS の実効性能を達成している [4]．しかしながら，これらの先行研究は最大 16 コアのマルチコア CPU システムでの評価にとどまっており，メニーコア CPU での大規模性能評価は行われていない．

アメリカエネルギー省では，15 の実アプリケーションについて KNL での性能評価を報告している [47]．Xeon CPU で開発してきた実装を単純に移植しただけでは，KNL の性能は MCDRAM のメモリバンド幅に律速されてしまい，いくつかのアプリケーションはスレッド数の大幅な増加により OpenMP のオーバーヘッドが全体性能を低下させており，OpenMP の並列化コードの改善が必要な場合があるとしている．TACC (Texas Advanced Computing Center) では，2 つのアプリケーションベンチマークおよび 2 つの実アプリケーションを通して KNL の性能評価を行った [13]．KNL はメモリモードだけではなく，チップ内の 2 次元メッシュネットワークについて 4 種類のネットワークモードを提供するが，ネットワークモードの違いは僅かな性能差しか生まないことを報告している．

[48] では，Lattice-QCD (格子量子色力学) 計算における主要計算である Wilson-Dslash operator で表れる 4 次元ステンシル計算の性能評価を行った．この研究では，MCDRAM だけを用いて 4 threads/core の実行で 505 GFLOPS を達成し，約 221 GB/s の実効メモリバンド幅が得られている．[49] はステンシル計算において，temporal blocking をさらに発展させた temporal wave-front tiling を提案し，3 次元ステンシル計算において Cache-mode において MCDRAM の容量を超えた場合でも約 800 GFLOPS の性能を達成した．

GPU におけるステンシル計算の最適化は高い性能が得られることが知られており，現在では新しいアルゴリズムの提案による高度な最適化や自動チューニングなど様々な議論が行われている [50, 51, 52]．

GPU は高いバンド幅を持つ GDDR5 や HBM2 といったメモリが搭載されているが，これらが提供する容量は一般的な CPU に対して小さい．実行されるアプリケーションが大規模であるほど，メモリ容量が不足し GPU クラスタでの実行が困難になる可能性が高くなる．そこで，GPU が持つメモリを超えた場合でも高い性能を達成するために GPU (MPI+CUDA) 用のステンシル計算フレームワークが提案されており，メモリ容量制約を克服するための手法も議論されている [53]．

第 5 章

メニーコア CPU における最適化と性能評価

本章では，KNC クラスタおよび KNL クラスタにおける電子動力学アプリケーションの最適化と性能評価，得られた性能に対する考察を述べる．

5.1　評価環境

5.1.1　KNC クラスタ: COMA

KNC の性能評価には，筑波大学計算科学研究センターの COMA を用いる [54]．COMA は，日本で初めて KNC を採用した PC クラスタとして 2014 年に稼働を開始し，同センターでは最初のペタスケールシステムとして運用され，後述する Oakforest-PACS などメニーコア CPU システムへの準備環境としても注目された．COMA の諸元について，表 5.1 に示す．

システムは 393 台の計算ノードで構成され，各ノードには 2 台の KNC が PCIe で接続されている．KNC の理論ピーク演算性能は 1,074 GFLOPS で，理論演算性能は計算ノードあたり約 2.5 TFLOPS となる．PCIe デバイス間通信では，NUMA 構成で Xeon CPU 間を接続するための Intel QPI (QuickPath Interconnect) を経由した場合に通信性能が大幅に低下することが知られている [55]．この問題を解決するために，COMA の計算ノードは KNC と InfiniBand FDR がすべて片側の CPU に接続されている．

本研究では，CPU のみの実行 (以下，CPU 実行)，Native 実行，Symmetric 実行の 3 つについて性能評価を行う．CPU 実行では，CPU の各ソケットに対し MPI プロセスを 1 個割り当て，各ノードあたり 2 個の MPI プロセスが割り当てられる．COMA の Xeon CPU は Intel Hyper Threading (SMT, Simultaneous Multi-Threading) が無効になっているため，各プロセスの OpenMP スレッド数は物理コア数に等しい．Native 実行では，各 KNC に対し 1 個の MPI プロセスを割り当て，CPU 実行と同じく各ノードあたり 2 個の MPI プロセスが割り当てられる．Symmetric 実行では CPU 実行及び Native 実行を組み合わせ，各ノードに 4 個の MPI プロセスが割り当て，計算ノード単位では Symmetric 実行は CPU 実行に対して 2 倍以上の計算資源を持つことになる．

COMA に接続されている Xeon Phi 7110P は 61 個の物理コアを持ち，最大で 244 スレッドでの並列

表 5.1 COMA 諸元

CPU	Intel E5-2670v2 with 10 cores, 2.5 GHz ×2 + Intel Xeon Phi 7110P with 61 cores, 1.1 GHz ×2
# of nodes	393 (use up to 128)
Memory	64 GiB of CPU DDR3 and 8 GiB ×2 of Xeon Phi GDDR5
Interconnect	Mellanox Connect-X3, InfiniBand FDR
Topology	Full bisection bandwidth of Fat-Tree
Compiler & MPI	Intel compiler 16.0.2 andIntel MPI 5.1.3

表 5.2 Oakforest-PACS 諸元

CPU	Intel Xeon Phi 7250 with 68 cores, 1.4 GHz base clock
# of nodes	8208 (use up to 8192)
Memory	16 GB of MCDRAM and 96 GB of DDR4-2400
Interconnect	Intel Omni Path Architecture with 100 Gbps link
Network topology	Full bisection bandwidth of Fat-Tree
File system	26 PB Lustre by DDN SFA7700X, 500 GB/s bandwidth
File cache	940 TB Burst Buffer by DDN IME14K, 1560 GB/s bandwidth
Cooling	Water colling for CPU and Air cooling for others
Operating system	CentOS 7 and McKernel (developed by RIKEN)
Compiler and MPI	Intel compiler 17.0.1 and Intel MPI 2017 update 1

化が可能である．しかしながら，そのうち 1 コアは OS のプロセス制御に用いられているため，本研究では 1 コアを除外し 60 コアでの並列化を考える．

5.1.2 KNL クラスタ: Oakforest-PACS

KNL の性能評価には，筑波大学と東京大学が共同設置する JCAHPC で運用されている Oakforest-PACS を用いる [56]．Oakforest-PACS は 2016 年 11 月の TOP500 リストにおいて，ピーク理論性能 25 PFLOPS，HPL 性能 13.55 PFLOPS を達成し，世界第 6 位にランクしたスーパーコンピュータである [1]．表 5.2 に，システムの基本性能について示す．

同じく KNL を採用した LBNL に設置されている Cori や，LANL に設置されている Trinity が Cray 社が提供する MPP (Massively Parallel Processing) システムであるのに対し，Oakforest-PACS はコモディティ PC クラスタとして世界最大規模のシステムである．各計算ノードは約 3 PFLOPS のピーク演算性能を持つ Xeon Phi 7250 を唯一の計算リソースとして持ち，8,208 台の計算ノードで構成される．通信デバイスとして Intel OPA (Omni-Path Architecture) [57] が採用され，8,208 ノードは Full bisection の Fat-tree network を構成する．

本研究では，扱いやすい 2 の冪乗ノード数での評価が合理的であるとして 8,192 ノード，システム全体の 99.8% を用いた性能評価を行う．同システムは Burst Buffer が利用できるが，ARTED はファイル IO が一般的なシミュレーションに比べて非常に少ないため，本研究では用いていない．上述の理由より，アプリケーションはファイルへのデータの書き出しを時間発展計算の終了後に一括処理する．したがっ

表 5.3　SPARC64 VIIIfx 諸元

Core	8
L1 Data Cache/Core	32 KB
L2 Cache	6 MB (shared)
GFLOPS	128
Memory Bandwidth	64 GB/s
Byte/FLOP	0.5
Compiler	Fujitsu K 1.2.0-19
Optimize Option	`-O3 -Kfast,ocl,openmp`

て，性能評価ではファイル IO の時間が含まれていないことに注意されたい．

KNL プロセッサでは，コア間ネットワークモードとメモリモードが複数用意されているが，Oakforest-PACS ではコア間ネットワークは Quadrant モードで固定され，メモリは MCDRAM を DDR4 メモリと同等に扱う Flat モードと，MCDRAM を DDR4 メモリのキャッシュとして扱う Cache モードの 2 種類のみが提供されている．この組合せを通常，“Flat-Quadrant” と “Cache-Quadrant” と呼ぶ．ARTED はステンシル計算が支配的な計算となるため，断りのない場合は全ての性能評価において Flat-Quadrant モードで MCDRAM のみを利用している．

Oakforest-PACS は，OS ジッタの影響回避のためタイマー割込をコア 0 のみ受け付ける，tickless 設定が行われている [58]．このため，Xeon Phi 7250 が持つ 68 コアのうち 64 コアの利用を推奨しており，本研究でも 64 コア・最大 256 スレッドで性能評価を実施する．また，KNL では 2 つのコアが 1 つの Tile を構成し 1 MiB の L2 キャッシュを共有するため，コア 0 で発生する割り込みがコア 1 にも影響を与えると考えられる．したがって，本研究ではコア 0 および 1 を避けてスレッドの割当を設定し，割り込みの影響を回避する．

5.2　ベクトル並列におけるステンシル計算の最適化

既に述べたとおり，本研究のステンシル計算はワーキングセットを 1 スレッド計算するため，スレッドとベクトル並列に対する最適化が重要となる．特にベクトル並列化は，従来プロセッサよりも長い SIMD 幅を持つメニーコア CPU において，極めて重要である．

ステンシル計算の最適化においては，性能評価にシリコン結晶 (Si) を用いる．Si は，1 個のステンシル計算領域である実空間格子のサイズ (`NL`=`NLx`×`NLy`×`NLz`) が (`NLx`, `NLy`, `NLz`)= $(16, 16, 16)$ となる．MPI と OpenMP によって並列化される波数空間格子は，実アプリケーションでは 4^3 から 24^3 程度，バンドは 16 と設定するが，ここでは KNC のメモリ制限の関係でそれぞれ 8^3，16 と設定する．すなわち，サイズが 16^3 の逐次ステンシル計算が $8^3 \times 16$ 個あり，これを言わば “タスク” として，各 OpenMP スレッドが計算する．

表 5.4 L1D キャッシュミスおよびメモリプリフェッチ発行数

	元実装	最適化実装	+ パディング
ミス率 [%]	7.7	4.23	2.02
プリフェッチ数	1.48×10^7	1.41×10^{11}	1.41×10^{11}
ロードストア数	1.13×10^{12}	1.09×10^{12}	1.09×10^{12}

5.2.1 マルチコアプロセッサでのコンパイラベクトル最適化の促進

はじめに，図 4.5 の元実装をベースに，「京」コンピュータ (SPARC64 VIIIfx プロセッサ) への最適化を行う [59]．表 5.3 に，プロセッサの諸元を示す．同プロセッサでは 128-bit SIMD 命令の HPC-ACE が提供されているが，本研究の最適化対象は倍精度複素数計算のため，1 要素しか計算できない．そのため，本研究では HPC-ACE を用いた手動ベクトル化は行わず，コンパイラによるベクトル化を促進する最適化を行う．

コンパイラによるベクトル化は，あくまでも記述されたコードを基に最適化を行うため，記述によってはベクトル演算が複雑化し期待した性能が得られていない可能性がある．ここで，図 4.5 の問題点を述べる．近傍点のアクセスに用いている 4 Byte 整数の間接参照配列 `IDX`，`IDY`，`IDZ` により，コンパイラはメモリアクセスパターンを把握できず，メモリアクセスが非常に非効率的となっている．また，24 個の近傍点にアクセスするため，メモリから 96×`NL` Byte のインデックス値のロードが必要となり，メモリ帯域を非常に圧迫してしまう．性能評価に用いる空間格子のサイズは 16^3 で，データサイズにすると 64 KiB となり，各点に対する係数 `B` も合わせて，各スレッドで 96 KiB のデータをメモリからロードする．

L2 キャッシュはコア間共有だが，プロセッサ全体で 6 MiB，コアあたり 768 KiB が利用できる．必要な係数なども含んでも，1 個のステンシル計算で必要なデータは L2 キャッシュに収まっている．図 4.4 に示す通り，メモリは Z 次元が連続となっているが，元実装では最もメモリ距離が遠い X 次元から計算しており，キャッシュが有効利用できておらず，計算順序を変更する必要がある．

以上より，ステンシル計算の最適化を行ったコードを図 5.1 に示す．1 次元配列として確保している 3 次元実空間格子を，カーネル中では (`NLz`, `NLy`, `NLx`) の 3 次元配列とした．この変更で配列へのアクセスが直接参照となり，コンパイラがメモリアクセスパターンを把握しやすくなることが期待される．

ステンシル計算は周期境界条件を持つため，インデックス計算には剰余が必要となる．実空間の各次元のサイズは，性能評価においては 2 のべき乗の 16 のため，論理積演算で代替可能かつ剰余演算よりも高速に求められるが，被除数が 2 のべき乗である必要がある．格子点を増やして 2 のべき乗にすることも可能だが，計算領域が肥大化しシミュレーションに制限が発生する可能性がある．そこで，各次元で剰余テーブル `modx`，`mody`，`modz` を用意し，剰余計算を省略した．剰余テーブルは次元のサイズを N，被除数を m とすると，$[0, N \times 2 + 4 - 1]$ の範囲で $m \bmod N$ の結果が保存されている．本研究では，剰余テーブルは各次元で 144 Byte が必要で，これは全てのインデックスを保管する間接参照配列の 96×`NL` Byte に対し非常に小さく，性能と可用性のバランスが取れている．

最後に，計算領域のサイズによりキャッシュスラッシングが多発する可能性があるため，Y 次元へのメモリパディングを行いこれを回避する．

```
real(8),   intent(in) :: B(0:NLz-1,0:NLy-1,0:NLx-1)
complex(8),intent(in) :: E(0:NLz-1,0:NLy-1,0:NLx-1)
complex(8),intent(out):: F(0:NLz-1,0:NLy-1,0:NLx-1)
integer,intent(in)    :: modx(0:NLx*2+4-1)
integer,intent(in)    :: mody(0:NLy*2+4-1)
integer,intent(in)    :: modz(0:NLz*2+4-1)
integer    :: ix,iy,iz
complex(8) :: v,w

#define IDX(dt) iz,iy,modx(ix+(dt)+NLx)
#define IDY(dt) iz,mody(iy+(dt)+NLy),ix
#define IDZ(dt) modz(iz+(dt)+NLz),iy,ix

do ix=0,NLx-1
do iy=0,NLy-1
do iz=0,NLz-1
  ! z computation
  v=(Cz(1)*(E(IDZ(1))+E(IDZ(-1))) + Cz(2)*(E(IDZ(2))+E(IDZ(-2))) &
  & +Cz(3)*(E(IDZ(3))+E(IDZ(-3))) + Cz(4)*(E(IDZ(4))+E(IDZ(-4))))
  ...
  ! y computation
  v=(Cy(1)*(E(IDY(1))+E(IDY(-1))) ... ) + v
  ...
  ! x computation
  ...
  ! store
  F(iz,iy,ix) = B(iz,iy,ix)*E(iz,iy,ix) + A*E(iz,iy,ix) - 0.5d0*v - zI*w
end do
end do
end do
```

図 5.1　コンパイラによる自動ベクトル化に最適化したステンシル計算の実装

表 5.5　100 反復時のキャッシュメモリアクセス待ち時間 [sec]

実装	浮動小数点数	整数	トータル
元実装	50.67	40.11	90.78
最適化実装	43.86	4.63	48.49
＋パディング	8.71	4.80	13.51

表 5.6　100 反復時の演算待ち時間 [sec]

実装	浮動小数点数	整数	トータル
元実装	9.03	0.57	9.60
最適化実装	8.05	1.55	9.60
＋パディング	5.81	1.48	7.29

各最適化について，「京」コンピュータで提供されている詳細プロファイラを用い，8 スレッドで性能評価を行った．ハミルトニアン計算には擬ポテンシャル計算が含まれるが，ステンシル計算に対し実行時間は比較的短いため，本評価では計算を省略する．つまり，図 4.3 中で，擬ポテンシャル計算を省略したハミルトニアン計算を 100 反復行い，キャッシュミスや演算時間などの検証を行う．

まず，元実装でのロードストア命令に対するキャッシュミスを計測すると，L1D ミスが 7.7%，L2 ミスが 0.03% となった．予測通り，各実空間格子は L2 キャッシュに収まっており，L1D キャッシュミスの削

表 5.7 SPARC64 VIIIfx でのステンシル計算性能

実装	GFLOPS	ピーク比 [%]
元実装	14.94	11.67
最適化実装	17.88	13.97
+ パディング	27.20	21.25

表 5.8 コア内スレッド数による性能への影響

OpenMP Threads	Relative performance
60 (1 Thread/core)	57.43
120 (2 Threads/core)	83.55
180 (3 Threads/core)	99.99
240 (4 Threads/core)	109.17

減が重要となっている．ロードストア命令比での L1D キャッシュミスを表 5.4 に示す．ここでは，図 4.5 の元実装，図 5.1 の最適化実装，最適化実装に対しパディングを行った際の結果を示している．最適化実装は元実装に対し，ロードストア命令数と L1D キャッシュミスを削減できている．また，プリフェッチ命令の発行数が 10^7 オーダから 10^{11} オーダに増加しており，間接参照配列の除去によってプリフェッチが効果的に動作している．

次に，キャッシュメモリへのアクセス待ち時間について，表 5.5 に示す．元実装では，100 反復時に浮動小数点数と整数のアクセス待ち時間が合わせて 90 秒発生していた．最適化実装では，間接参照配列を用いず近傍点のインデックスを直接計算することによって整数のアクセス待ち時間が約 1/10 まで削減された．浮動小数点数は最適化実装でも依然として待ち時間が非常に長いが，パディングによって 1/5 まで削減されている．

ここで，演算待ち時間を表 5.6 に示すが，インデックスを直接計算することによって，整数演算の待ち時間が増加している．しかしながら，キャッシュメモリのアクセス待ち時間と合わせて 100 秒近いボトルネックが約 20 秒まで削減されており，間接参照のコストが非常に高いということが分かる．計算順序を非連続方向から計算した場合，ロードストア命令の発行数が 1.25 倍に増加し，キャッシュメモリのアクセス待ち時間が 2 倍，演算待ち時間が 5 倍近く増加した．非連続方向から計算したことにより，コンパイラは計算をパイプライン化できないという警告を出力しており，効率的なベクトル化および演算ができていなかったと考えられる．

最後に，演算性能について表 5.7 に示す．元実装では 14.94 GFLOPS だったが，最適化およびパディングを行うことで 27.20 GFLOPS，ピーク性能比約 21 % まで改善した．本研究では，図 5.1 の最適化した自動ベクトル化実装を用いて，KNC への最適化を行う．

5.2.2 KNC プロセッサへのさらなる最適化

前述の SPARC64 VIIIfx プロセッサへの最適化を踏まえて，図 5.1 のコードにさらなる最適化を行い，KNC におけるコンパイラベクトル最適化の性能を評価する．[60] や [61] といった先行研究を参照すると，高レベル最適化を行ったとしてもピーク性能比 10% と，KNC において高い演算性能を得るのは非

表 5.9　Knights Corner でのコンパイラベクトル化性能

実装	GFLOPS	ピーク比 [%]
元実装	30.06	2.80
コンパイラベクトル化	109.17	10.16
近傍点ロード最適化	125.81	11.71
non-temporal store	129.00	12.01
L1D ブロッキング	129.48	12.06
メモリパディング	130.44	12.15

表 5.10　倍精度浮動小数点数の load/gather 命令数

実装	load	gather
コンパイラベクトル化	568	96
近傍点ロード最適化	456	64

表 5.11　L1D キャッシュミス数とヒット率

実装	ミス数	ヒット率 [%]
元実装	1.67×10^{11}	88.7
コンパイラベクトル化	7.83×10^{10}	85.8
近傍点ロード最適化	5.21×10^{10}	88.5
non-temporal store	4.76×10^{10}	89.6
L1D ブロッキング	4.79×10^{10}	89.0
メモリパディング	4.55×10^{10}	89.3

常に困難が予測される．KNC でのメモリのアラインメントは，実空間格子配列 `E(0:NLz-1, 0:NLy-1, 0:NLx-1)` としたとき，`E(0, :, :)` が 64 バイト境界にアラインされるように設定した．

まず，表 5.8 にコア内スレッド数による KNC でのステンシル計算性能への影響について示す．一般的に，ステンシル計算はメモリバンド幅律速のため，物理コア数と OpenMP スレッド数を一致させることが望ましいとされている．しかしながら，本研究においてはワーキングセットが 1 個の実空間であることから並列数も重要であることが表からもわかる．以上より，KNC は 60 コアを使用し，コアあたり 4 スレッドの 240 スレッドで全ての性能評価を行う．

表 5.9 に，KNC でのステンシル計算性能を示す．図 4.5 の元実装は，ピーク性能比で 2.8% しか得られておらず，KNC でも間接参照のコストが非常に高いことがわかる．“コンパイラベクトル化” は，図 5.1 を KNC で実行したときの性能で，元実装の約 3.6 倍に相当する 109.17 GFLOPS が得られた．同実装は SPARC64 VIIIfx に対して最適化を行ったものだが，従来のマルチコアプロセッサにおける最適化がメニーコア CPU でも有効であることを示しており，非常に重要な結果と考えられる．ここから，KNC に対するさらなる最適化を考える．

図 5.1 では，近傍点のロードがベクトル演算と同時に行われているため，一時配列配列に近傍点をロードした後に，ベクトル演算を行うように変更した．アセンブラコードから，load および gather 命令の出現数を表 5.10 に示す．KNC では，メモリのアラインが取られていない状態でのロードに対して

`loadunpackhi/lo` という 2 つの命令が用いられる．そのため，表での load 命令数は通常の aligned load 命令と合わせて 3 命令の総数を示している．アセンブラコードレベルでは，命令出現数が減少しており，処理内容が簡素化され効率が良くなったもしくは load 命令の実行回数が減少したと考えられ，性能が 125.81 GFLOPS に向上した．SPARC64 VIIIfx では，性能低下に繋がるため適用していない．

`F` には書き込みのみが行われ, 書き込まれたデータは計算中で参照されない．そこで, 最内の Z 次元ループに `vector nontemporal` ディレクティブを記述し，キャッシュを経由しない non-temporal store [62] とすることで，キャッシュメモリをさらに有効利用し 129 GFLOPS となった．`vector nontemporal` は Intel コンパイラで提供されるディレクティブで，指定された変数またはループにおける書き込みをキャッシュ非経由の streaming store 命令で行う．

SPARC64 VIIIfx では効果がなかったが，L1D に対して YX 次元に 2 次元ブロッキングを行うと，若干ながら性能が改善し 129.48 GFLOPS となった．スラッシングを回避するために，SPARC64 VIIIfx と同様に Y 次元に対しメモリパディングを行ったが 130.44 GFLOPS と小さな改善にとどまった．Z 次元のパディングを行った場合，KNC ではベクトル化時にループサイズがベクトル長で割り切れなくなってしまい，アドレスが 64 Byte 境界からずれてしまうため，効率的なベクトル化が困難となる．

L1D キャッシュミス数とヒット率について，Intel VTune プロファイラ [63] での計測結果を表 5.11 に示す．元実装に対し，コンパイラベクトル化はミス数のオーダが 1 桁下がっている．近傍点のインデックスを直接計算しているため，間接参照に使用していたキャッシュメモリを近傍点のキャッシュに利用できるようになったことが要因と考えられる．近傍点ロード最適化の効果が最も高く，L1D キャッシュミスが 2×10^{10} 程度減少した．他の最適化も適用することでミス数が減少しているが，L1D ブロッキング時には増加しており，効果が非常に小さいと考えられる．メモリパディングも行い，最終的な L1D キャッシュヒット率は 89.3% となった．

最後に，KNC におけるコンパイラによるベクトル最適化では，130.44 GFLOPS と理論性能に対して 12.15% を達成し，元実装に対して 4.3 倍の高速化を達成した．

5.2.3 512-bit SIMD 命令を用いた手動ベクトル化

元実装に対して大幅な高速化を達成したものの，理論性能を考えると非常に低い値しか得られていない．KNC から SIMD 命令の bit 幅が 512-bit になり，従来の Xeon プロセッサが提供する SIMD 幅に対して倍のデータ量となった．Intel コンパイラは，256-bit 幅のベクトル演算には強力な最適化機構を提供しているが，512-bit 幅へのノウハウ蓄積ができておらず，ベクトル最適化が成熟していない可能性がある．そこで，KNC で提供されている 512-bit SIMD 命令の IMCI (Initial Many-Core Instructions) Intrinsics を用いて，手動のベクトル演算最適化を行う．Intrinsics は C 言語のインターフェイスとして提供されるため，Fortran から利用できない．したがって，本研究ではステンシル計算コードのみを C 言語の関数として抜き出し，手動ベクトル最適化を行う．

本研究では，コンパイラベクトル最適化で得られた結果から，次に示す方法で手動ベクトル最適化を行う．まず，`F` に対して non-temporal store を用いるが，ストア先のアドレスが 64 Byte でアラインされていなければならない．演算ベクトル長はサイズが 128-bit の倍精度複素数のため 4 となり，4 つの格子点を同時更新する．前提条件として，先に述べたアラインメントの問題を踏まえ，連続なメモリアクセス

```
inline __m512d dcomplex_mul_c(__m512d a) {
  __m512i b, c, d = (__m512i)a;
  b = _mm512_set4_epi64(1LL<<63,0,1LL<<63,0);
  c = _mm512_shuffle_epi32(d,_MM_PERM_BADC);
  return (__m512d) _mm512_xor_si512(c,b);
}
```

図 5.2　倍精度複素数積 $(a, bi)(0, -i)$ を展開した実装

となる Z 次元について格子サイズをベクトル長 4 の倍数として最適化を行った．大きく分けて次の 3 点の最適化を行い，それぞれの効果について検証する．

- 倍精度複素数積の最適化
- 近傍点のインデックス計算のベクトル化
- 非アラインメモリアクセスの最適化

計算には倍精度複素数が用いられているが，IMCI では Intel SSE/AVX と異なり，複素数積を高速に計算するための命令が提供されていない [64]．KNC で複素数積を行う際には，bit masked 命令を使って実部あるいは虚部のみを計算する必要があり，ベクトル演算ユニットの潜在性能の半分しか利用できない．これは Intel コンパイラによるベクトル最適化，Intrinsics を用いた手動ベクトル最適化のどちらの場合でも問題となる．この問題から，KNC の複素数計算で高い性能を得るのは実数計算の場合よりも困難と考えられる．図 4.5 では，配列 `w` と定数 `zI` で積が必要となるが，定数積のため，式展開により演算を省略できる．複素数積 $(a, bi)(0, -i)$ を展開すると $(b, -ai)$ となり，(1) 実部と虚部を入れ替え，(2) 虚部の符号を反転させる，の 2 ステップで計算可能となる．図 5.2 に，式展開を行った IMCI 実装を示す．`b` は定数として扱われ，XOR を用いて虚部の実数の符号ビットを反転する．

近傍点は各方向に ± 4 点にアクセスし，各次元で 8 点，計 24 点となる．Z 次元は後述するメモリアクセスの最適化によりインデックスの計算は不要で，Y および X 次元の 16 点のインデックスを求める必要がある．インデックスは 4 Byte 整数で表現可能な範囲で十分足りるため，16 点のインデックスを 512-bit ベクトル演算でまとめて求めることが可能で，スカラでのインデックス計算を最小限にできる．Z 次元の計算では，後述の最適化でメモリから 512-bit ベクトル 3 本のみをロードするため，図 5.4 に示すようにスカラでインデックス計算を行っている．

連続方向である Z 次元の計算では，必ず非アラインメモリアクセスが発生するため，コンパイラは gather 命令を発行しデータを集める．しかしながら，同命令はレイテンシが高く，性能上のボトルネックとなっている可能性が高い [65]．そこで，本研究では alignr 命令 (concatenate shift 命令) を用いて gather 命令を用いずに Z 次元の近傍点を集める．alignr 命令は 2 つの 512-bit ベクトルを連結して 1024-bit ベクトルとし，32-bit 単位で右論理シフトを行った後，下位 512-bit を返す命令である．この最適化は [66] をベースとしているが，[66] が周期境界条件を考慮していないのに対し，本最適化は境界条件によらず利用できる．

ここで，メモリの連続方向と非連続方向での計算パターンの違いについて考える．連続方向では，alignr 命令を用いて 1 点の更新に必要な近傍点を 1 つのベクトルに集めることが可能である．しかし，非連続方向では 1 回の load 命令を用いて 4 点の更新に必要な近傍点のうち，±4 のいずれか 1 点を一度に取得で

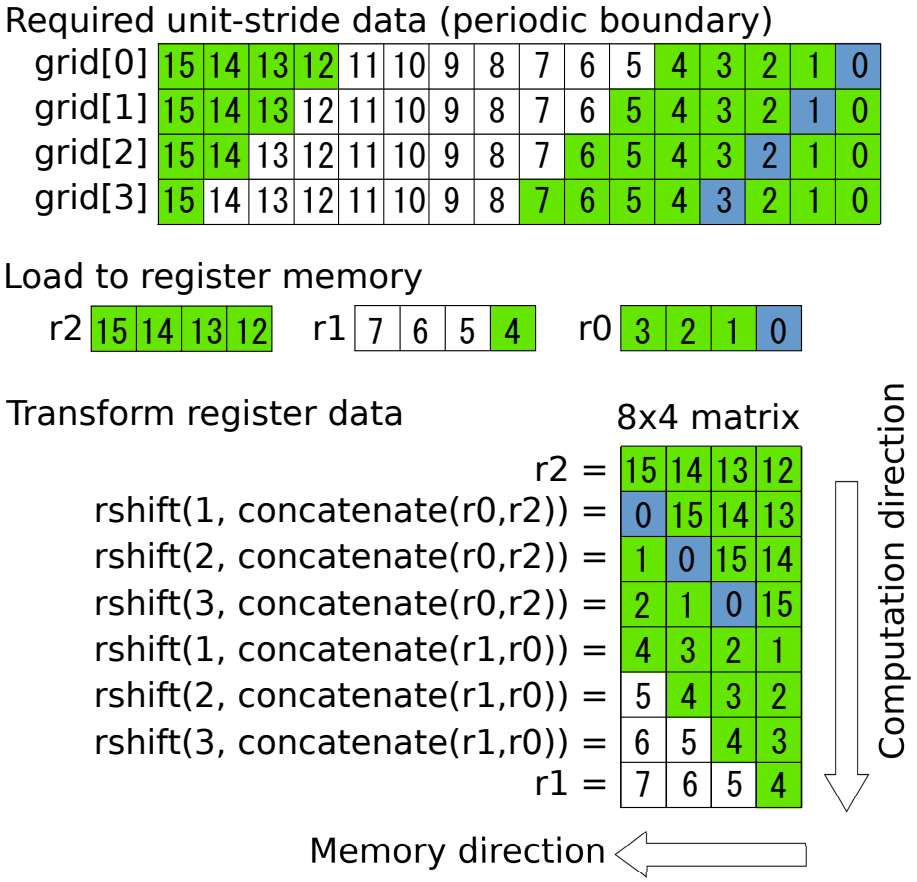

図 5.3　Z 次元のメモリアクセス最適化

きる．この場合，非連続方向はメモリに対して垂直方向に計算すれば，4 点を同時更新することができるが，連続方向では水平方向に計算する必要がある．SIMD での水平方向計算は IMCI には実装されておらず，SSE/AVX では実装されているものの垂直方向計算に比べて実行レイテンシが高い．したがって，連続方向でのメモリアクセスを，非連続方向と同じ垂直方向の計算となるように最適化を行う．

連続方向の Z 次元の非アラインメモリアクセスを，非連続方向と同じ計算パターンとなるように実装したイメージが図 5.3 となる．(1) まず，aligned load 命令を用いて更新対象の前の 4 点，更新対象の 4 点，その次の 4 点をそれぞれベクトル `v0`，`v1`，`v2` としてロードし，(2) alignr 命令を用いて，`v0`，`v1`，`v2` から各点の更新に必要な近傍点ベクトルを生成する．この時，更新点 ± 4 でそれぞれ 1 つの行ベクトルが 1 個のベクトルレジスタに収まり，負方向と正方向の近傍点で 8×4 の行列を生成する．例えば，更新点 `F[0]` では，近傍点 15–12 および 1–4 が必要となる．生成した行列を見ると，非連続方向でのメモリパターンと同様に，必要な近傍点は 1 列に揃っており，計算はメモリに対して垂直方向に行えばよく，水平方向の計算が必要ない．このメモリアクセスを，Intrinsics を用いて実装すると図 5.4 となる．alignr 命令の 32-bit シフト回数は即値で，コンパイル時には値が確定している必要がある．

手動ベクトル最適化を行った実装に対し，前述した各最適化を個別に適用し，その効果を表 5.12 に示す．“初期実装” は前述の最適化を行わず，単純に Fortran で書かれたコードを C 言語に書き換えただけの実装である．この場合，性能は 113.65 GFLOPS となり，コンパイラベクトル最適化に対して性能が低

```
/* Stencil computation loop */
for(ix = 0 ; ix < NLx ; ++ix) {
for(iy = 0 ; iy < NLy ; ++iy) {
double complex const* e = E[ix*NLy*NLz + iy*NLz];
...
for(iz = 0 ; iz < NLz ; iz += 4) {
  __m512i v0, v1, v2; __m512d m[4], p[4];

  /* Load to register memory */
  v0 = _mm512_load_epi64(e + modz[iz-4+NLz]);
  v1 = _mm512_load_epi64(e + iz);
  v2 = _mm512_load_epi64(e + modz[iz+4+NLz]);

  /* Transform register data */
  m[3] = (__m512d) v0;
  m[2] = (__m512d) _mm512_alignr_epi32(v1, v0,  4);
  m[1] = (__m512d) _mm512_alignr_epi32(v1, v0,  8);
  m[0] = (__m512d) _mm512_alignr_epi32(v1, v0, 12);
  p[0] = (__m512d) _mm512_alignr_epi32(v2, v1,  4);
  p[1] = (__m512d) _mm512_alignr_epi32(v2, v1,  8);
  p[2] = (__m512d) _mm512_alignr_epi32(v2, v1, 12);
  p[3] = (__m512d) v2;

  /* Z-dir. computation */
  for(n = 0 ; n < 4 ; ++n) {
    __m512d a = _mm512_add_pd(p[n], m[n]);
    __m512d b = _mm512_sub_pd(p[n], m[n]);
    v = _mm512_fmadd_pd(C[n], a, v);
    w = _mm512_fmadd_pd(D[n], b, w);
  }
  /* Y-dir. and X-dir. computation */
  /* non-temporal store to F */
} } }
```

図 5.4　Z 次元のメモリアクセスの IMCI 実装

表 5.12　手動ベクトル最適化の効果

実装	GFLOPS	ピーク比 [%]
初期実装	113.65	10.58
複素数積最適化	114.89	10.70
インデックス計算ベクトル化 (A)	142.74	13.29
非アラインアクセス最適化 (B)	179.39	16.70
(A) + (B)	221.38	20.61
全最適化適用時	224.45	20.90

い．L1D キャッシュミス数は全ての最適化で約 3.75×10^{10}，ヒット率は 92% 程度となり，コンパイラによる最適化に対し良好な結果となっている．

初期実装に対し複素数積展開を行うと 114.89 GFLOPS と若干ながら性能が向上し，メモリバンド幅に律速され，複素数積を計算するコストが非常に小さいといえる．本研究のコードでは各点の更新に対し複素数積を一度しか行わず，演算数が少ないため，複素数積が非常に多く必要となる計算では考慮が必要と考えられる．インデックス計算をベクトル化することで 142.74 GFLOPS の性能が得られ，インデックスをベクトル演算で求める利点があることが分かった．非アラインメモリアクセス最適化は，179.39 GFLOPS と本研究の最適化の中で最も高い効果が得られ，gather 命令が大きなボトルネックとなってい

表 5.13 周期境界領域拡張の効果

実装	Compiler Vec.	Explicit Vec.
従来 [GFLOPS]	130.44	224.45
Z 次元拡張 [GFLOPS]	165.63	220.66
ハミルトニアン計算相対性能	1.03	0.93

ること，load 命令の実行回数を削減できたことが大きく影響していると考えられる．

さらに，インデックス計算のベクトル化と非アラインメモリアクセス最適化の 2 つを組み合わせ，221.38 GFLOPS を達成した．初期実装に対し約 107 GFLOPS の性能向上が得られており，2 つの性能の増分からさらに 10 GFLOPS ほど追加の性能向上が得られている．命令数は変わっておらず，組み合わせによってレジスタの有効利用や計算の待ち時間が減少しさらなる性能向上に繋がったと考えられる．これらの最適化全てを適用することで，初期実装に対し 1.97 倍の 224.45 GFLOPS とピーク性能比 20.9 % を達成した．コンパイラにベクトル化を任せた場合 (130.4 GFLOPS) に比べ，およそ 1.7 倍の性能が得られており，Intel コンパイラのベクトル最適化が 512-bit SIMD 命令に対して成熟していないことを示唆している．

5.2.4 問題依存のさらなる最適化

本研究のステンシル計算は，周期境界のためキャッシュミスが発生しやすく，性能上不利なメモリアクセスが発生しやすい．そこで，連続方向の Z 次元を拡張し周期境界領域をコピーすることで，Z 次元のメモリアクセスを線形化しさらなる最適化を図る．

本研究のステンシル計算は，1 個の実空間格子に対し 4 回行われるため，各スレッドが計算用の一時領域にコピーした状態で，ステンシル計算を行っている．ここでは，この一時領域のサイズを元の `E(0:NLz-1, 0:NLy-1, 0:NLx-1)` から，`E(-4:NLz+4-1, 0:NLy-1, 0:NLx-1)` と拡張し，連続方向の次元を線形アクセス化した．この最適化によって，各ステンシル計算の直前に `E(-4:-1, :, :)` と `E(NLz:NLz+4-1, :, :)` の領域に対しデータコピーが必要となる．

コンパイラベクトル最適化および手動ベクトル最適化実装に対し，周期境界領域拡張の効果について，表 5.13 に示す．以後，同最適化を Z 次元拡張とする．コンパイラベクトル最適化の実装では 30 GFLOPS 以上の性能向上に繋がったが，手動ベクトル最適化の実装では性能が低下した．本研究で実装した手動ベクトル最適化が，KNC に対し非常に最適化されており，同最適化の効果が小さくなっていると推測される．

最終行にはハミルトニアン計算の相対性能を示し，この値が 1 より大きい場合，Z 次元拡張がハミルトニアン計算全体で見たときに効果があることを意味する．Z 次元拡張のみ，拡張領域に対するデータコピーがステンシル計算の外で必要となっているため，ハミルトニアン計算の性能を比較する必要がある．コンパイラベクトル最適化と手動ベクトル最適化実装共に，連続領域のみにもかかわらず周期境界領域のコピーコストが高く，ハミルトニアン計算全体では実行時間が増加した．コンパイラベクトル最適化実装ではステンシル計算性能が向上したが，全体では 1.03 倍と小さな性能向上にとどまり，手動ベクトル最適化では 0.93 倍と全体の性能は低下した．本研究の場合，ハミルトニアン計算全体を考慮すると Z 次元

拡張の効果は低かったが，一般的なステンシル計算ではキャッシュミスなどのペナルティが大きくなるため，一考の余地があると言える．

5.2.5 Knights Corner から Knights Landing へ

手動ベクトル最適化のコードは，アーキテクチャが違っていても提供される命令セットが同じであれば，再コンパイルのみで実行可能である．SIMD 命令セットを用いた最適化は，特定アーキテクチャに依存してしまうことが多く，別のアーキテクチャへの移植を妨げてしまう．したがって，強力に最適化したコードを移植できるのかは非常に重要な観点である．

KNC では SIMD 命令として IMCI が提供されているのに対し，KNL は AVX-512 を提供するため，この 2 つの命令フォーマットの違いを吸収する必要がある [67]．IMCI と AVX-512 は，四則演算などの基本的な命令のフォーマットは同じだが，シャッフルや入れ替えといった命令のフォーマットが異なる．また，非アラインロード命令についても IMCI がキャッシュラインを跨がないように 2 命令を発行する必要があるのに対し，AVX-512 では AVX と同様に 1 命令で実行できる．

一部，IMCI では実装されているが AVX-512 では実装されていない命令も存在する．本研究で用いた IMCI を用いた手動実装のうち，AVX-512 への変換をするために書き換えが必要な命令，または計算は以下の 4 つである．

- 128-bit 単位での並べ替え (shuffle/permute 命令)
- non-temporal store 命令
- 倍精度浮動小数点数から倍精度複素数への変換
- 非アラインロード命令

128-bit 単位での並べ替えと non-temporal store 命令は，フォーマットや名前が異なるのみで，要求される引数は同一である．倍精度浮動小数点数から倍精度複素数への変換，非アラインロード命令は必要命令数や必要な命令そのものが異なるが，10 行未満のインライン関数の置換のみが必要となる．名前の変換とインライン関数実装の入れ替えのみを必要とするため，これらはプリプロセッサを用いることで，KNC から KNL へ容易に移行できる．プリプロセッサによるコード変換を図 5.5 に示す．IMCI 実装と AVX-512 実装の違いは，図 5.5 に示したコードだけである．`__AVX512F__`マクロは，AVX-512 をサポートするプロセッサで定義され，`__MIC__`は KNC でのコンパイル時に定義される．AVX-512 実装は，さらなる最適化を目的としたコードの追記は行わず，IMCI 実装に対しプリプロセッサを用いた置換のみで実装されている．

AVX-512 は複数のサブセットに分かれており，プロセッサ間で利用できる命令が異なる点に注意が必要となる．本研究で実装した手動ベクトル最適化コードでは，AVX-512 に対応する全てのプロセッサがサポートする基本命令セットの AVX-512F (Foundation) のみを利用している．すなわち，KNL や Skylake-SP など AVX-512 をサポートする全てのプロセッサで，本研究の実装は再コンパイルのみで利用可能である．

```
#ifdef __AVX512F__
/* AVX-512 codes */
#define _mm512_permute4f128_epi32(a, p) _mm512_shuffle_i32x4(a, a, p)
#define _mm512_storenrngo_pd            _mm512_stream_pd

inline __m512d dcast_to_dcmplx(double const *v) {
  __m512d w = _mm512_maskz_expandloadu_pd(0x55, v);
  return _mm512_movedup_pd(w);
}

inline __m512d dcomplex_mul(__m512d a, __m512d b) {
  __m512d ar = _mm512_movedup_pd(a);
  __m512d ai = _mm512_unpackhi_pd(a, a);
  __m512d bs = _mm512_shuffle_pd(b, b, 0x55);
           ai = _mm512_mul_pd(ai, bs);
  return       _mm512_fmaddsub_pd(b, ar, ai);
}
#elif __MIC_
/* IMCI codes */
inline __m512i _mm512_loadu_si512(int const* v) {
  __m512i w = _mm512_loadunpacklo_epi32(w, v + 0);
  return      _mm512_loadunpackhi_epi32(w, v + 16);
}

inline __m512d dcast_to_dcmplx(double const *v) {
  const __m512i perm = _mm512_set_epi32(7, 6, 7, 6, 5, 4, 5, 4, 3, 2, 3, 2, 1, 0, 1, 0);
  __m512d w = _mm512_loadunpacklo_pd(_mm512_setzero_pd(), v);
  return (__m512d) _mm512_permutevar_epi32(perm, (__m512i) w);
}

inline __m512d dcomplex_mul(__m512d a, __m512d b) {
  __m512d ze = _mm512_setzero_pd();
  __m512d s0 = _mm512_swizzle_pd(a, _MM_SWIZ_REG_CDAB);  /* s0 = [a.i a.r] */
  __m512d re = _mm512_mask_blend_pd(0xAA, a, s0);        /* re = [a.r a.r] */
  __m512d im = _mm512_mask_sub_pd(a, 0x55, ze, s0);      /* im = [-a.i a.i] */
  __m512d t0 = _mm512_mul_pd(re, b);                     /* t0 = [a.r*b.r a.r*b.i] */
  __m512d s1 = _mm512_swizzle_pd(b, _MM_SWIZ_REG_CDAB);  /* s1 = [b.i b.r] */
  return       _mm512_fmadd_pd(im, s1, t0);              /* [-a.i*b.i+a.r*b.r a.i*b.r+a.r*b.i] */
}
#endif
```

図 5.5　プリプロセッサを用いた IMCI から AVX-512 へのコード変換

5.3　ヘテロジニアス環境におけるクラスタレベル最適化

KNC クラスタでは，KNC と CPU それぞれに MPI プロセスを配布する Symmetric 実行についても性能評価を行う．Symmetric 実行は，PCIe デバイスとして接続されている KNC を 1 台の計算ノードとして扱えるため，OpenMP+MPI の構造を崩すことなく KNC クラスタが持つ全計算資源を利用することができる．また，現行アーキテクチャの KNL は PCIe デバイスではなく Xeon CPU と同じソケットタイプが主流となっており，KNL や他のメニーコア CPU への移植を考えると性能評価は極めて重要である．本節では，KNC+CPU のヘテロジニアス環境におけるクラスタレベルの最適化を示す．

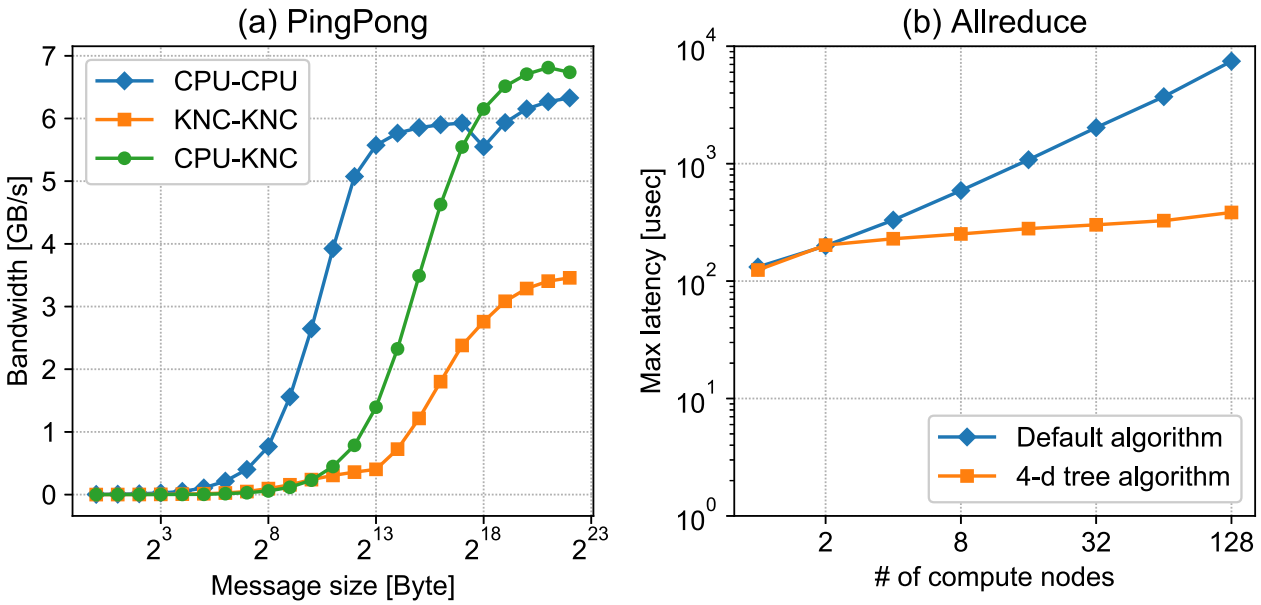

図 5.6　(a) InfiniBand FDR を用いたノードをまたぐ PingPong 通信性能，(b) Symmetric 実行における倍精度浮動小数点数ベクトルの `MPI_Allreduce` 通信性能

5.3.1　通信アルゴリズム

Native と Symmetric 実行の場合，KNC を 1 台の計算ノードとして，同じく PCIe で接続された InfiniBand を用いてノード間通信を行う．本研究の評価環境である COMA では，KNC と InfiniBand は全て同一ソケットに接続され，各計算ノードに接続された 2 台の KNC は Intel QPI (Quick-Path Interconnect) をまたがずに通信可能である．図 5.6-(a) に，InfiniBand FDR を用いたノードをまたぐ場合の PingPong 通信について，それぞれ CPU 間と KNC 間の測定結果を示す．CPU–KNC 間は，PCIe の通信バンド幅を意味する．CPU 間，KNC 間はどちらも InfiniBand FDR を用いているが，KNC の通信バンド幅は CPU に対して半分程度しか出ていない．この性能差は，KNC でのノード間通信が CPU と同じプロトコルで InfiniBand を用いていないことに起因する．Intel MPI ではプロキシモデルを用いて通信バンド幅を向上させている [68] が，COMA ではこの通信モデルを用いると通信したデータに不整合が発生してしまうため InfiniBand を直接利用する通信モデルに変更しており，結果として期待値よりも低い通信バンド幅しか利用できない．

前述の問題から，Symmetric 実行時には CPU と KNC の通信性能差を考慮した通信アルゴリズムが必要となる．Intel MPI の場合，通信アルゴリズムを手動変更が可能で，各 Collective 通信について複数の通信アルゴリズムが実装されている．通常は Intel MPI 側で最適と思われるアルゴリズムが自動選択されるが，本研究ではアルゴリズムを手動変更して通信性能を最適化する．

時間発展計算中で発生する通信は `MPI_Allreduce` による倍精度浮動小数点数，および小領域のサイズ NL と同じ長さの倍精度浮動小数点数ベクトルの総和演算のみである．したがって，`MPI_Allreduce` の通信アルゴリズムを最適化する必要がある．図 5.6-(a) から，計算ノード間の KNC による通信はバンド幅が低く，PCIe による CPU–KNC 間通信の最高性能は CPU 間通信より高い．つまり，最初に計算ノー

ド内で通信を行い，CPU で計算ノード間の通信を行えれば良く，基数を 4 とした 4-d tree のアルゴリズムが最適と考えられる [69]．同アルゴリズムでは，N を基数，P を MPI プロセス数として，$2log_N P$ 回で通信が行われる．N を 4 とすると，まずノード内の 4 MPI プロセス (2 CPU プロセス，2 KNC プロセス) で通信が行われ，あとは計算ノード 4 台単位で通信が行われる．このとき，計算ノード間の通信は 4 の倍数のランクの MPI プロセスで行われるが，COMA の Symmetric 実行において MPI プロセスは (CPU0, 1, KNC0, 1) の順序で割り当てられるため，計算ノード間の通信は必ず CPU で行われる．

サイズ NL での `MPI_Allreduce` の通信性能を図 5.6-(b) に示す．このとき，メッセージサイズは 64 KiB のため，ノードをまたぐ KNC 間通信より，PCIe 経由の CPU–KNC 間は高い通信性能を持つことが図 5.6-(a) からわかる．Intel MPI のデフォルトではミリ秒オーダまで通信性能が悪化しているが，4-d tree により 10^2 [usec] のオーダで安定している．既に述べたとおり計算時間のオーダは 1 ステップでミリ秒オーダであるため，この最適化により通信性能がボトルネックとならず，計算性能に注力できる．

5.3.2 静的ロードバランス

ここで，MPI プロセス単位で比較した場合の Symmetric 実行の性能についての予測を行う．Symmetric 実行で問題を CPU–KNC 間で均等に割り当てた場合，計算全体の性能は `MPI_Allreduce` により性能が低いプロセッサに律速される．したがって，KNC と CPU の計算時間が同一となるように計算量を調整することで，Symmetric 実行の性能を最大化する．

本研究では，実空間のサイズよりも波数空間のサイズが非常に大きいため，波動関数の波数空間パラメータである NK を MPI で分散されている．同並列化により各 MPI プロセスは袖領域の交換ではなく，`MPI_Allreduce` を用いた総和計算を行うため，通信は粗粒度かつ簡単化されている．以上より，NK の割当量を調整することで，極めて容易に負荷分散を実現可能である．性能評価では，性能に合わせて KNC と CPU の計算量を調整したものを “Symmetric (Balanced)” と記載し，均等割当は “Symmetric (Even)” と記載する．計算量調整により計算順序が変更されシミュレーション結果への影響が懸念されるが，現在までに影響は確認されていない．

5.4 アーキテクチャ依存最適化

5.4.1 メニースレッド環境におけるベクトル和の並列化

TDKS 方程式の求解において，波動関数全体を参照する電子密度計算が必要となる．同計算は，ベクトル和で表現され Zu を電子の波動関数配列としたときに，下記の計算式で表される．

$$il = 1, 2, \ldots, NL$$

$$rho_{il} = \sum_{ik=1}^{NK} \sum_{ib=1}^{NB} c_{ib,ik} |Zu_{il,ib,ik}|^2$$

この計算式は，NK×NB (波数空間のサイズ) 本の長さ NL (実空間格子のサイズ) のベクトルを束ねることを意味する．計算式における OpenMP の並列化方法を考えると，この計算式は非常に単純な計算ルー

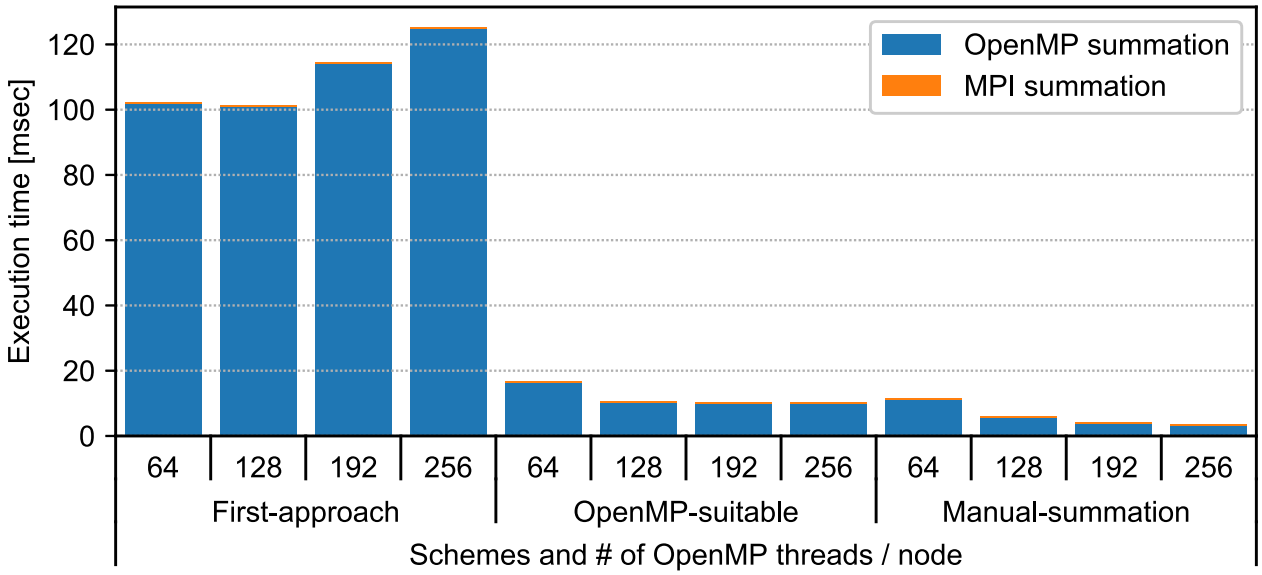

図 5.7　ベクトル和の並列化における各実装の性能

プで表されるが，最内ループに依存性があり単純な並列化が困難である．

ここで，ベクトル和計算の OpenMP 並列化実装について，図 5.8 に示す．最内ループに依存性があるため，最も単純化した実装の場合，`omp parallel` 句は最外に配置できるが，`omp do` 句は最内ループに必要となり，暗黙の同期によるオーバーヘッドが大きなコストとなる (“First-approach”)．しかし，`omp reduction` 句に指定する配列，すなわちここでは長さ NL のベクトルが各 OpenMP スレッドのスタック領域に収まる場合は，最外ループを並列化し `omp collapse` 句を使って並列粒度を粗粒度にできる (“OpenMP-suitable”)．OpenMP が提供するベクトルのリダクションでは性能が満足に出ない場合や，スレッドのスタック領域に収まらない場合，OpenMP の並列化ループにおいて並列和を手動で実装し必要なスタック領域を最小化することも可能である (“Manual-summation”)．“Manual-summation” には，リダクションのアルゴリズムとして非常に一般的な binary-tree algorithm を用いている [70]．同アルゴリズムは GPU 環境でも多用され，単純だが効果的なアルゴリズムとして知られる [71]．

図 5.7 に，32 台の KNL ノードにおけるベクトル和の性能を示す．まず，“First-approach” は OpenMP の暗黙の同期による非常に重いオーバーヘッドにより，非常に性能が低い．“OpenMP-suitable” のように修正することで，高い性能は得られるが，128 スレッド (2 threads/core) で性能は飽和状態にある．一方で，“Manual-summation” は “First-approach” に比べオーバーヘッドを 1/30 以上に削減し，“OpenMP-suitable” と異なり性能向上が見られ，OpenMP の `reduction` 句がメニーコア CPU におけるベクトル和に最適化されていないと推察される．また `MPI_Allreduce` による通信コストは，OpenMP の並列ベクトル和に比べてかなり低く，OpenMP の並列化方法が MPI 通信よりも極めて重要であることがわかる．

本最適化によって，ベクトル和の計算コストは 100 [msec] から 1 [msec] のオーダへ削減されたが，以降の章で示すとおり計算全体の性能は 10–100 [msec] オーダである．これらの結果は，メニーコア CPU で効率的にアプリケーションを実行するためにはコードの全体に渡る十分な最適化が必要であること，

```
! First-approach (Poor performance) ============
!$omp parallel
do ik=1,NK; do ib=1,NB
!$omp do
do il=1,NL
  rho(il)=rho(il)+c(ib,ik)*abs(Zu(il,ib,ik))**2
end do
!$omp end do
end do; end do
!$omp end parallel

! OpenMP-suitable ==============================
!$omp parallel do collapse(2) reduction(+:rho)
do ik=1,NK; do ib=1,NB; do il=1,NL
  rho(il)=rho(il)+c(ib,ik)*abs(Zu(il,ib,ik))**2
end do; end do; end do
!$omp end parallel do

! Manual-reduction =============================
!$omp parallel
tid=omp_get_thread_num(); tmp(:,tid)=0.d0

!$omp do collapse(2)
do ik=1,NK; do ib=1,NB; do il=1,NL
  tmp(il,tid)=tmp(il,tid)+c(ib,ik)*abs(Zu(il,ib,ik))**2
end do; end do; end do
!$omp end do

i=ceiling_pow2(omp_get_num_threads())/2
do while(i > 0)
  if (tid < i) tmp(:,tid)=tmp(:,tid)+tmp(:,tid+i)
  i=i/2
!$omp barrier
end do

!$omp do
do il=1,NL
  rho(il)=tmp(il,0)
end do
!$omp end do
!$omp end parallel
```

図 5.8 ベクトル和の並列化の各実装

OpenMP (スレッド並列化) における同期コストの考慮が必要であることを示している.

5.4.2 MCDRAM と DRAM の併用

KNL は MCDRAM と DDR4 の異なるバンド幅を持つ 2 つのメモリを提供するが，多くの実アプリケーションがメモリバンド幅ボトルネックであることから，性能を最大化するにはアプリケーションが MCDRAM で完結することが望ましい．しかし DDR4 メモリが最大 96 GiB と大容量であるのに対し，MCDRAM は高々 16 GiB しかなく，KNL クラスタの計算資源を最大限活用するためには，MCDRAM と DDR4 メモリを組み合わせる必要がある.

最も単純な方法として，MCDRAM をスクラッチパッドキャッシュにする方法が考えられる．L1 や L2 キャッシュがキャッシュするデータをハードウェア上で自動選択するのに対し，スクラッチパッドキャッシュはユーザがコード上でキャッシュするデータを選択する．スクラッチパッドキャッシュを用い

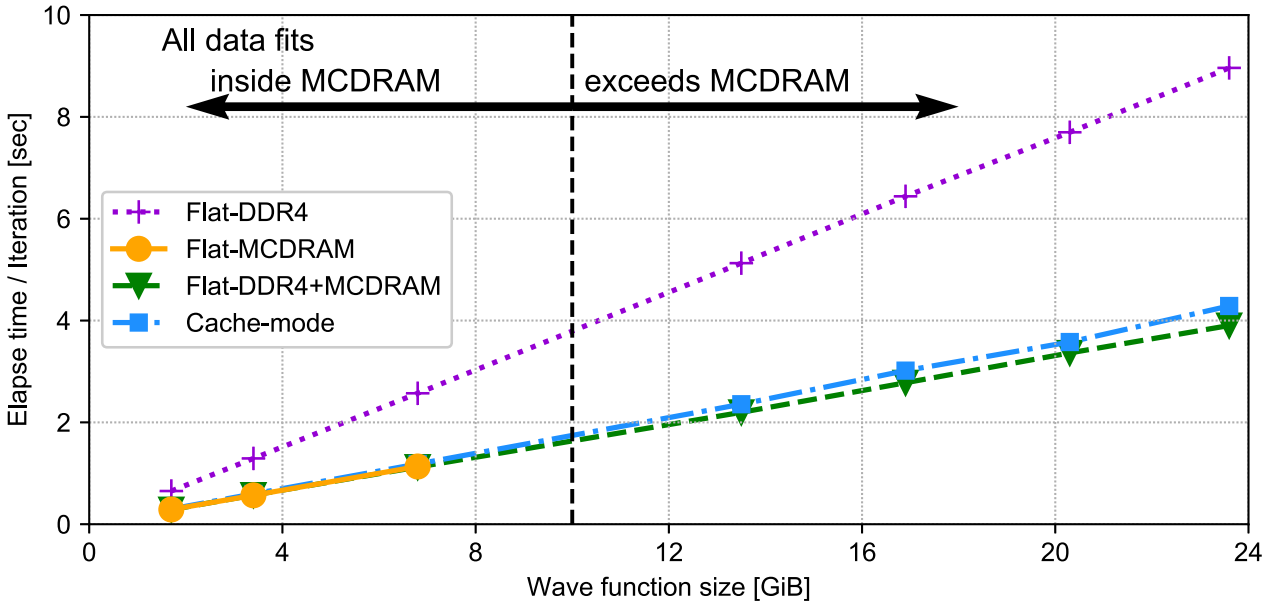

図 5.9　KNL のメモリモードの違いによる計算全体の性能比較

た MCDRAM と DDR4 メモリの活用方法は以下の通りである．

1. アプリケーションは全てのデータを DDR4 メモリに確保する
2. 計算開始前，DDR4 メモリからワーキングセットを MCDRAM に手動でコピーする
3. 計算終了後，計算結果を MCDRAM から DDR4 メモリに書き戻す

明示的なキャッシュデータの管理は，適切なキャッシュブロックサイズのサーチやキャッシュデータの追い出しタイミングなど，制御が非常に困難である．しかしながら，上述したスクラッチパッドキャッシュの使い方は図 4.3 で説明したハミルトニアン計算のワークフローに非常に適している．DDR4 メモリに電子の波動関数 (アプリケーションの計算領域) を確保し，各スレッドのワーキングメモリと計算に使用する各種係数を MCDRAM に確保すれば，計算アルゴリズムを変更することなく，MCDRAM をスクラッチパッドキャッシュとして利用できる．

図 5.9 に，アプリケーションのデータセットのサイズが MCDRAM の容量を超えた場合の性能について示す．“Flat-DDR4” と “Flat-MCDRAM” は，全てのアプリケーションデータが DDR4 メモリまたは MCDRAM に確保された状態で，それぞれのメモリで閉じて計算している．“Flat-DDR4+MCDRAM” は，DDR4 メモリと MCDRAM を併用し，それぞれメインメモリとスクラッチパッドキャッシュとして利用する．図のように，“Flat-DDR4+MCDRAM” によりアプリケーションの計算可能サイズを，極めて自然に広げることができた．“Cache-mode” は，MCDRAM をダイレクトマップキャッシュとして利用するモードだが，“Flat-DDR4+MCDRAM” に対して性能が若干低い．ダイレクトマッピングによるキャッシュコンフリクトは無視できないコストと考えられるが，各スレッドで閉じた計算のため，キャッシュコンフリクトが起こりづらいと考えられる．またこれらの結果より，制御が簡単であれば，スクラッチパッドキャッシュはダイレクトマップキャッシュに対し性能面で有利と言える．

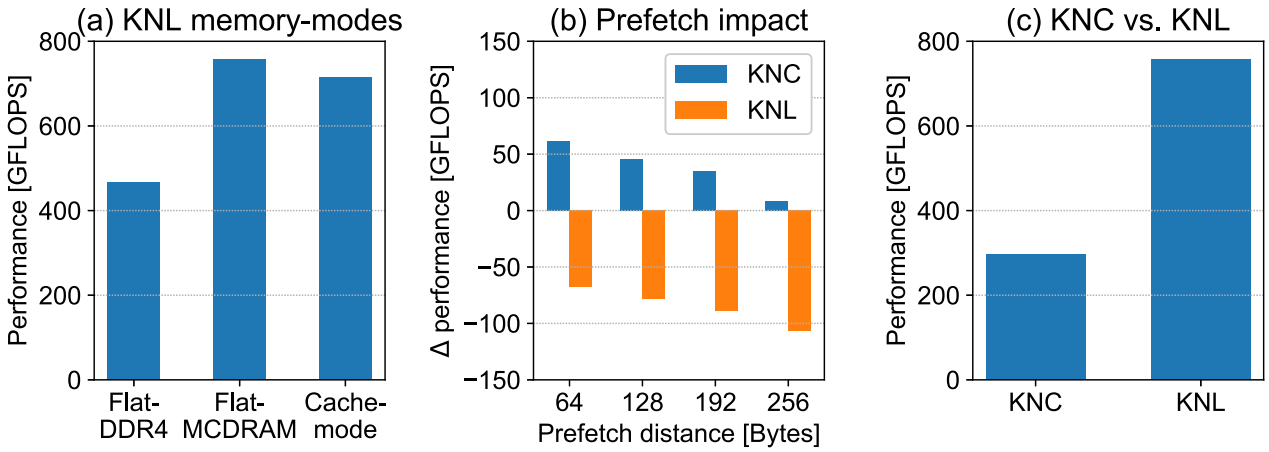

図 5.10　ステンシル計算の性能評価

5.5　性能評価

表 5.8 に示したように，ARTED の核となるステンシル計算はスレッド並列数が重要で，どちらのプロセッサでもコアあたり 4 スレッド，KNC は 240 OpenMP threads/processor，KNL は 256 OpenMP threads/processor で性能評価を行う．KNL の性能評価は，特に断りがない場合 Flat-MCDRAM で性能評価を行い，DDR4 メモリは全てシステムサービスに割り当てられる．

5.5.1　ステンシル計算の性能評価

はじめに，KNL のメモリモードによる性能の違いについて図 5.10-(a) に示す．スクラッチパッドキャッシュの性能評価で，既に MCDRAM を用いた性能が最良であることは述べているが，全てのデータを DDR4 で閉じて計算する Flat-DDR4 と，同じく MCDRAM で閉じて計算する Flat-MCDRAM では，約 1.7 倍程度の差しかなく，両者が提供するメモリバンド幅を考えれば MCDRAM での計算性能が低いと言える．これは，ワーキングセットのデータサイズによるものと考えられる．

各 OpenMP スレッドがハミルトニアンを計算する場合，スレッドローカルなワーキングメモリ領域で閉じて計算を行う．スレッドは，ハミルトニアンを計算する直前に MCDRAM または DDR4 メモリに確保したグローバルメモリ領域からワーキングセットをコピーする．ワーキングセットは，通常 L2 キャッシュに収まる程度に小さく，計算の大部分は L2 キャッシュで閉じる．メモリモードに依らず，計算時間の大部分は L2 キャッシュアクセスに費やされ，各メモリのバンド幅は MCDRAM or DDR4 メモリから L2 キャッシュにデータをキャッシュするわずかな時間の間しか影響せず，性能差が小さくなったと考えられる．以上より，Cache-mode の性能と Flat-MCDRAM の性能はほぼ同一であることが期待されるが，Cache-mode はダイレクトマップキャッシュのため，キャッシュコンフリクトが発生し Flat-MCDRAM よりも性能が低下していると考えられる．しかし計算は L2 キャッシュで閉じるため，性能低下が比較的小さく済んでいるとも考えられる．

次に，手動ベクトル最適化実装にて，キャッシュメモリへのプリフェッチ命令 (ソフトウェアプリフェッ

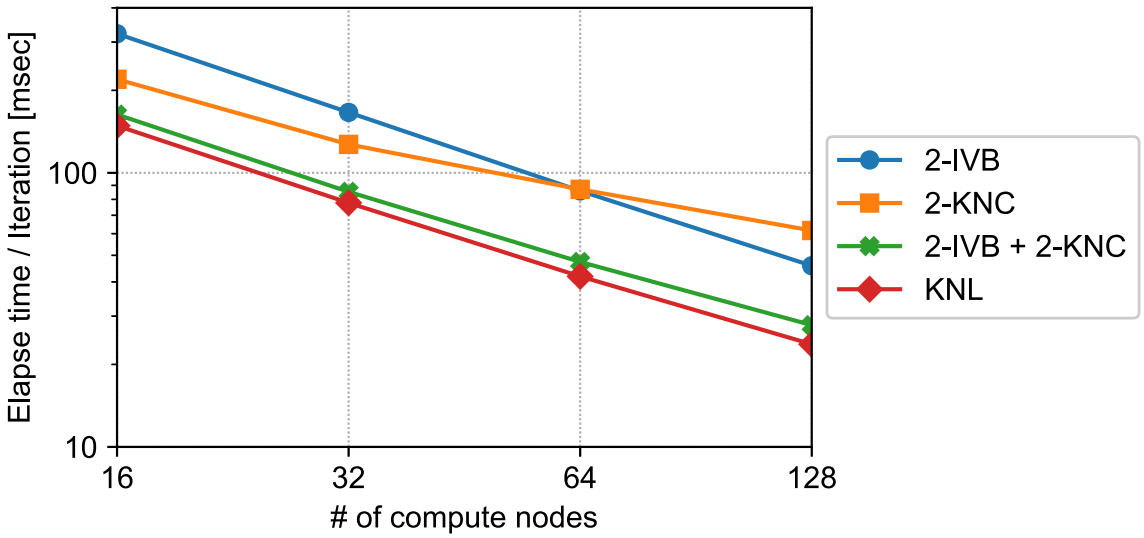

図 5.11　KNC/KNL クラスタ間における強スケーリングの性能比較

チ) を用いたさらなる最適化を考える．本研究の実装は，特に spatial locality に対する最適化を行い，計算においてレジスタに読み込んだデータを最大限活用している．現在のループ中で計算に必要なデータをロードした後，次の計算に必要なデータをプリフェッチするように load 命令の直後にプリフェッチ命令を発行する．ステンシル計算のワーキングセットは，キャッシュラインのサイズに等しい 64-byte ブロックでアラインされているため，プリフェッチ命令は次のループで必要なデータをキャッシュライン上の 1 個のブロックとしてまとめてプリフェッチできる．

図 5.10-(b) に，ソフトウェアプリフェッチの効果について，性能向上を差分で示す．グラフの横軸はプリフェッチ距離を示し，load 命令で読み込んだアドレスから 64-byte 先 (次のキャッシュライン) と 256-byte 先までのプリフェッチを評価した．KNL のハードウェアプリフェッチの性能向上に伴いソフトウェアプリフェッチの手動挿入が不要となりつつあり [48]，我々のステンシル計算コードでも KNC では 1 台あたり 60 GFLOPS 以上の性能が得られたのに対し，KNL では大幅な性能低下が見られた．

最後に，図 5.10-(c) に，KNC と KNL におけるステンシル計算の性能比較を示す．KNC は，ソフトウェアプリフェッチによりさらなる性能向上を達成し約 300 GFLOPS，理論性能に対して約 28% の効率が得られた．KNL は，KNC の約 2.56 倍となる 758.4 GFLOPS を達成し，理論性能に対して約 24.8% の効率を達成した．HPL の実効性能から，メニーコア CPU の達成可能な実性能を考えれば高い水準であると言える．KNL の実効性能は，KNC に対し最適化されたコードをベースとしていることから，KNC への最適化は KNL に対しても高い効果を発揮すると言える．

5.5.2　Knights Corner/Landing クラスタ間の比較

シングルセル計算における強スケーリングの性能評価を行い，KNC/KNL クラスタ間の性能比較を行う．本評価では，シミュレーションのサイズについて比較的大規模な計算 (NK, NB, NL) = (24^3, 16, 4096 = (16, 16, 16)) を設定し，計算資源の制約から計算ノード 128 台までの性能比較を行った．

表 5.14 各シミュレーションのデータセット

	シリコン	グラファイト
MPI procs. / Macro-grid	–	8
Macro-grid / MPI procs.	1, 2, 4	–
Total # of macro-grid (`NZ`)	32768	1024
# of wave count (`NK`×`NB`)	$8^3 \times 16$	7928×16
Size of 3-D real space (`NL`)	16^3	$26 \times 16 \times 16$

COMA は 2 台の Ivy-Bridge Xeon CPU をホストプロセッサとして，2 台の KNC をアクセラレータとして搭載しており，本評価では 2-IVB (CPU-only)，2-KNC (KNC-only, Native 実行)，2-IVB + 2-KNC (Symmetric 実行) の 3 つについて性能評価を行った．Symmetric 実行はヘテロジニアス実行であることから静的負荷分散を適用したが，32 ノード以下の比較的小規模なノード数でしか効果が得られなかった．本評価では，並列性が 24^3 となる波数空間の割当量を CPU と KNC で変更したが，強スケーリングにより KNC が要求する並列数を達成できなくなり，負荷分散の効果が得られなくなったと考えられる．

図 5.11 に，KNC/KNL クラスタの比較を示す．KNC-only は性能飽和が見られ，2-IVB に比べ性能が低下しているが，KNL は 2-IVB + 2-KNC の Symmetric 実行よりも高い性能を達成した．逐次演算性能の違いから，KNC が高い並列性を要求するのに対し，KNL は比較的低い並列性においても高い性能を達成しやすくなり，アプリケーションの移植コストが削減可能と考えられる．KNL は Flat-MCDRAM の実行では MCDRAM の実効バンド幅 490+ GB/s が利用できるが，COMA では CPU と KNC の 4 台を総計すると 430+ GB/s の実効バンド幅が利用できる．アプリケーションの性能はメモリバンド幅に律速されているため，総実効メモリバンド幅の差が，Symmetric 実行 (COMA) と KNL (Oakforest-PACS) 間の比較的小さな性能差の原因と考えられる．

5.5.3 Knights Landing クラスタにおける計算全体の性能評価

マルチスケール計算においてサポートされる 2 つの並列化方法について評価するため，表 5.14 に示す 2 つのシミュレーションセットを用いる．データセットは，Oakforest-PACS 全系を用いた実シミュレーションのパラメータをそのまま用いている．シリコンは TDKS 方程式のサイズが $8^3 \times 16 \times 16^3$ と小さいため，1, 2, 4 個のマクロ格子点をまとめて 1 個の MPI プロセスで計算する．グラファイトは TDKS 方程式のサイズがシリコンに対し約 25 倍と非常に巨大なため，各マクロ格子点を 8 個の MPI プロセスで計算する．8,192 ノードで計算できるマクロ格子点の数は，シリコンで 32,768 点，グラファイトが 1,024 点である．

弱スケーリングでは 128 ノードから計測し，シリコンは 512–32,768 点，グラファイトは 16–1,024 点のマクロ格子点を計算し評価した．強スケーリングはそれぞれ MPI の並列化方法が異なるため，MPI プロセスあたりの問題サイズの設定が異なっている．シリコンは 1,024 ノードから計測し，マクロ格子点を 8,192 点に固定して MPI プロセスあたりのマクロ格子点を減らして評価した．グラファイトは 512 ノードから計測し，マクロ格子点を 128 点に固定して，マクロ格子点あたりの MPI プロセス数を増やして評

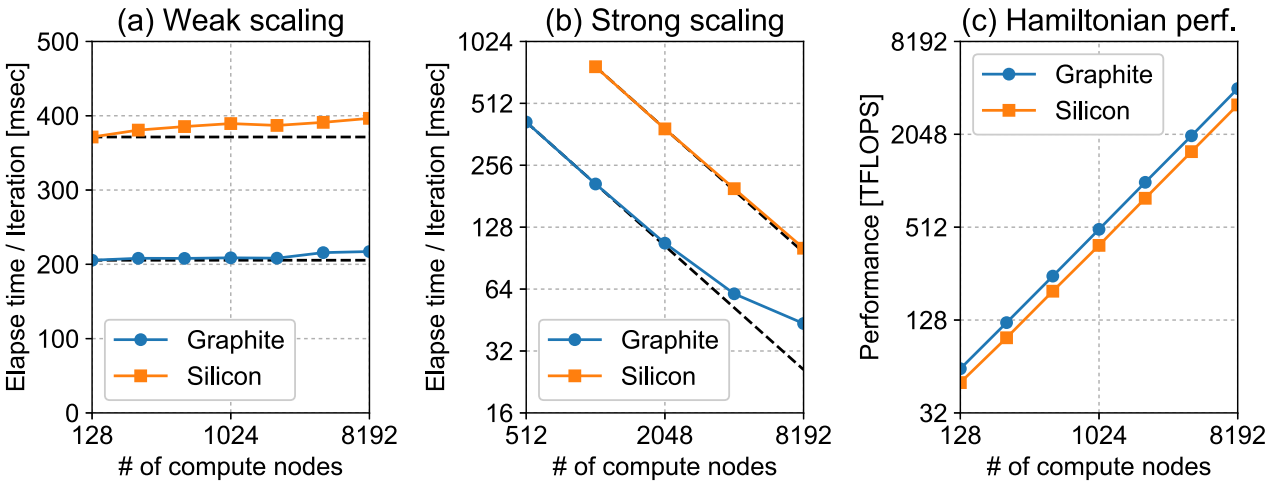

図 5.12　Oakforest-PACS 全系による性能評価

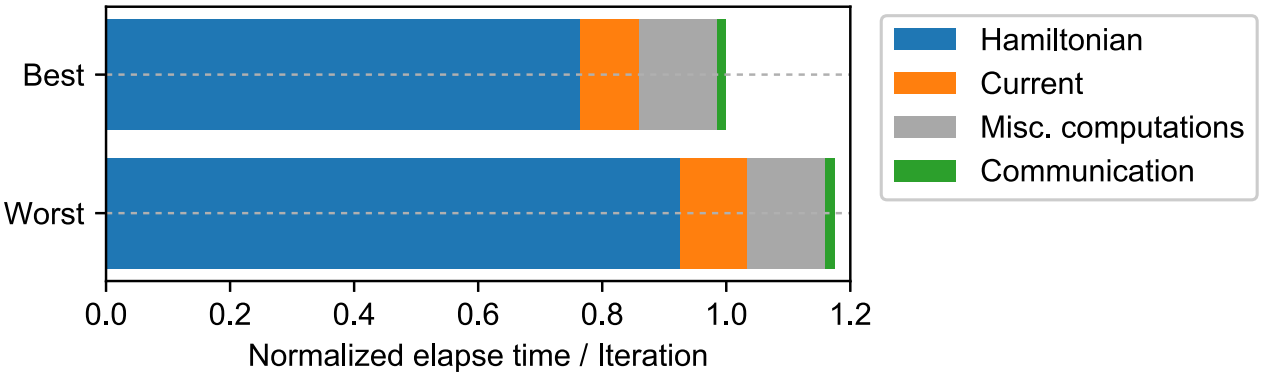

図 5.13　時間発展計算の計算時間内訳

価を行った．

図 5.12 に，Oakforest-PACS 全系を用いた性能評価の結果を示す．強スケーリングでは物質により結果が異なっているが，128 ノードから全系に相当する 8,192 ノードまでの弱スケーリングをほぼ完璧に達成している．弱スケーリングでは，時間発展計算で最も重要なハミルトニアン計算において，最大 4 PFLOPS，システム全系で理論性能に対して約 16% 相当，HPL 性能比で約 30% の計算効率を達成した．

シリコンは強スケーリングについてもほぼ完璧に達成していると言えるが，グラファイトの計算で大幅な性能低下が見られる．スケーリング性能の低下について，各計算ノード間でおよそ 20–30% 程度の計算時間の差が生じていることを発見しており，この差が原因であると考えている．全ての計算ノードには全く同じ計算量が与えられており，弱スケーリングの結果からも見て取れるように計算負荷はバランスが取れている．しかしながら，図 5.13 に示す計算ノード性能で比較した Best/Worst case の計算時間内訳より，通信を全く含まず計算のみで構成されるハミルトニアン計算の時間に大きな差が生じていることがわかる．

5.6 考察

5.6.1 動的クロック調整機能によるロードインバランス

通信を一切含まないハミルトニアン計算に発生しているノード間の性能差について，Turbo Boost に代表される動的クロック調整機能が原因と考えている．Turbo Boost は，動作クロックをプロセッサの TDP を守りつつベースクロックから最大で数百 MHz ほど動的に引き上げ，瞬間的に高い性能を得られる．しかしながら，Turbo Boost は当然プロセッサ温度も条件として含めているため，AVX-512 を使い演算器をフルに活用すれば温度上昇により Turbo Boost によるクロック向上の効果時間が短くなることは容易に予想できる．したがって AVX-512 を用いた計算を最高周波数で動かし続けることは困難で，このようなアーキテクチャで理論性能を達成することはほぼ不可能である．また Oakforest-PACS では 2 ノードペアでプロセッサを冷却する水冷却システムを導入しているため，1 番目に冷却されるノードと 2 番目に冷却されるノードでは，プロセッサ温度が異なったとしても不思議ではない．

シリコンとグラファイトでは強スケーリング時の性能が異なっており，上述の Turbo Boost によるノード間性能差だけでは説明できていない．これは，図 4.2 に示すように MPI の並列化方法の違いによるものと考えられる．シリコンでは計算コストの大部分を占める TDKS 方程式において MPI での分割を行っておらず，各 MPI プロセスが独立して複数の閉じた系を計算するため，`MPI_Allreduce` による MPI プロセス間同期が一切発生しない．したがって，シリコンは全 MPI プロセスでの全体同期だけが唯一必要となり，上述の性能ギャップはハミルトニアン計算を行ったあと，各時間ステップで行われる `MPI_Allreduce` によって即座に是正されると考えられる．

TDKS 方程式も MPI で分散並列化するグラファイトでは，同じ TDKS 方程式 (ここでは系とする) を計算する MPI プロセス群での同期，マクスウェル方程式での全 MPI プロセスの同期，2 種類の同期が必要となる．各系で閉じた同期は小数の MPI プロセス群，今回は 8 プロセス単位で発生するため，性能ギャップによる遅延はシリコンの場合に比べてランダム性が増加し，より広範囲に影響するものと考えられる．性能ギャップは最終的に通信レイテンシとして表面化し，通信時間の比が増加する強スケーリングの評価で問題が顕在化した．もしこのアルゴリズムと一切関係ない性能ギャップが改善した場合，我々のアプリケーションは強スケーリングにおいても高い性能を実現できると見込んでいる．

5.6.2 他のメニーコアプロセッサへの適用

AVX-512 命令をサポートした最初の Xeon CPU として，現在 Intel Skylake-SP (SKL) アーキテクチャが提供されており，本研究で実装した KNC および KNL 向けのステンシル計算の手動ベクトル最適化実装を，一切のコード変更なしに移植することが可能である．Xeon CPU との比較用に，本研究では東京大学物性研究所 (ISSP) にて試験稼働中の System C を用いる [72]．ISSP System C の諸元を表 5.15 に示す．同システムは，計算ノード 252 台で構成された SKL Xeon CPU を唯一の計算リソースとして持つ一般的な CPU クラスタである．

SKL はプロダクトが 6100 番台以降の場合，AVX-512 の演算ユニットが各コアに 2 個用意されるため，FLOP/Cycle は KNL と SKL で等価である．図 5.14 に，シングルソケットの Xeon Gold 6148 の

表 5.15　ISSP System C 諸元

CPU	Xeon Gold 6148×2 2.4 GHz base clock with 20 cores
Memory	192 GB DDR4-2666
Network	Mellanox InfiniBand EDR 4x
OS	Red Hat Enterprise Linux 7
Software	Intel Compiler 18.0.1, Intel MPI 2018 update 1

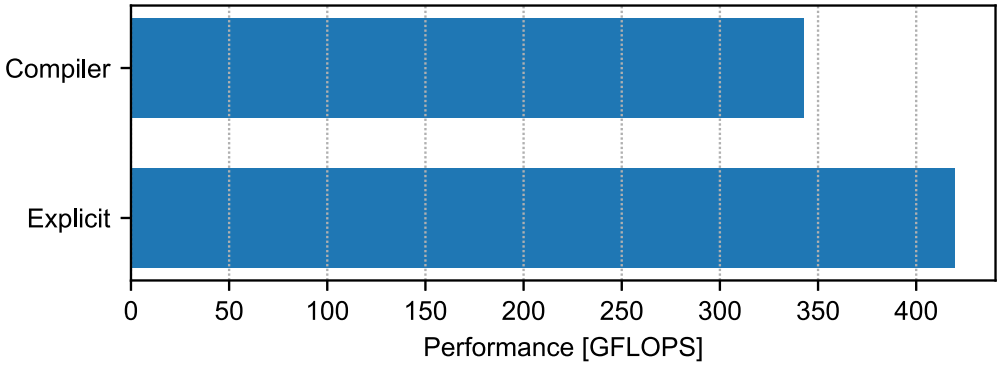

図 5.14　Skylake-SP プロセッサにおけるステンシル計算性能の評価

ステンシル計算性能について示す．“Compiler” は，最新の Intel コンパイラ ver. 18.0.1 によるベクトル最適化，“Explicit” は AVX-512 を用いた手動ベクトル最適化の性能を示す．SKL においても，Intel コンパイラが行う最適化に比べ我々の手動ベクトル化コードがより高い性能を示しており，現在のところ AVX-512 による手動ベクトル化が有効であることがわかる．しかし，性能の差が縮まっており，プロセッサの性能向上とコンパイラ習熟によりコンパイラによるベクトル最適化の性能が向上していると考えられる．

現在開発が進められているポスト「京」コンピュータでは，ARM プロセッサベースで 512-bit 幅で SIMD 命令セット ARM SVE (Scalable Vector Extension) を実装し，メニーコア CPU クラスタを実現する予定となっている．ポスト「京」コンピュータで採用される ARM プロセッサは，RISC プロセッサに分類され，ARM SVE も AVX-512 に比べてプリミティブな命令しか提供されない．これまで富士通が開発してきたシステムで提供された SIMD 命令，HPC-ACE/HPC-ACE2 は，HPC 用途では同じく 256-bit 幅の AVX/AVX2 命令に近い機能を提供してきたが，規格で定義される ARM SVE の範囲では，本研究の最適化を移植することは非常に困難である．例えば，本研究の最適化で重要な役割を持っていた alignr 命令も ARM SVE には存在しないため，複数の SVE 命令を組み合わせて実装する必要がある．そこで，富士通は HPC 向けの ARM SVE 拡張をポスト「京」コンピュータのプロセッサに追加することを予定しており，HPC-ACE に近い機能の提供が期待される．

第 6 章

GPU における最適化と性能評価

本章では，NVIDIA GPU クラスタにおける電子動力学アプリケーションの最適化と性能評価，得られた性能に対する考察を述べる．

6.1　評価環境

本研究では GPU クラスタとして，筑波大学計算科学研究センターで運用されている PPX (Pre-PACS-X) を用いる [54]．PPX は，同センターが長年開発している PACS シリーズの次世代機に向けた評価環境として運用されている小規模クラスタで，GPU として P100 と V100 を 2 枚搭載したノードがそれぞれ提供されている．PPX の諸元を表 6.1 に示す．PPX では，Intel Hyper Threading をオフにした Broadwell-EP Xeon CPU が 2 ソケット，P100 または V100 が各ソケットに PCIe で 1 台接続されており，InfiniBand EDR が片側に接続されている．表 6.1 の構成で P100 ノードが 4 台，V100 ノードが 1 台が運用され，計算ノードとして利用できる．ここで，各 GPU が持つ HBM2 の実効メモリバンド幅は GPU-STREAM ベンチマークを用いて計測した [73].

評価用のデータセットには，GPU で十分な並列性を確保できるように，計算領域 (電子の波動関数) のサイズを全体で 4 GiB とした．ステンシル計算のワーキングセットとなる 3 次元実空間格子点は 16^3 で，ワーキングセットあたり 64 KiB となる．計算資源の制限から，シングルセル計算の性能評価のみを実施する．

6.2　アプリケーションの実装方法: OpenACC+CUDA

アプリケーションを NVIDIA GPU へ対応させる場合，OpenACC または OpenMP によるディレクティブベース言語での開発か，CUDA または OpenCL による SDK (Software Development Kit) を用いた開発が考えられる．NVIDIA は OpenACC での開発を推奨しており，本節では OpenACC と CUDA による開発を考える．CUDA は NVIDIA による GPU コンピューティング用の統合開発環境で，当初は C/C++ 言語での提供にとどまっていた [15] が，現在では Fortran の開発環境が NVIDIA 傘下の PGI が提供する PGI コンパイラの機能として提供されている [74]．OpenACC は OpenMP に代表されるディレクティブベースのプログラミング言語で，OpenMP 4.0 から導入されたアクセラレータ拡張に先

表 6.1 Pre-PACS-X (PPX) 諸元

CPU	Xeon E5-2690 v4×2 2.6 GHz with 14 cores
GPU	NVIDIA Tesla P100×2 or V100×2 (PCIe card version)
Memory	64 GB DDR4-2400, 16 GB HBM2 / GPU
HBM2 Actual Bandwidth	P100: 551.3 GB/s, V100: 839.1 GB/s
Network	Mellanox InfiniBand EDR 16x with Mellanox OFED 3.4-2.0.0
OS	CentOS 7.3.1611 (Compute node)
Software	PGI Compiler 17.10, CUDA 9.0, OpenMPI 2.1.2

行する形で実装が進められた [75]．OpenMP のアクセラレータ拡張が厳格な並列化を求めているのに対し，OpenACC はシンプルな記述でも並列化を考慮した上でのコード変換とコンパイルが行われる．

本研究のターゲットは Fortran で実装されているため，利用可能なコンパイラは PGI コンパイラに限られるが，下記の理由から OpenACC による実装が最適と考えられる．

開発コスト

コード全てを CUDA に対応させるのは非常に時間がかかる，また GPU–CPU 間のデータ転送や計算カーネルの起動といった GPU の制御だけでも多くのコードの追加が必要

性能とのバランス

全てを CUDA で適切に実装することで最高性能を得られると期待されるが，最適化にかかるコストが釣り合わない可能性もある

実装者と利用者の技術レベルの乖離

HPC 研究者は CUDA で適切なコードを実装可能と期待されるが，実際の利用者は計算科学の研究者であり，CUDA コードの修正コストが高い

特に 3 つ目が最も重要で，第 3 章で述べたように，計算科学の中でも比較的新しい分野に属するアプリケーションはその大半が将来変更される可能性が高いコードと考えられる．このようなアプリケーションの全体を CUDA で実装した場合，その性能を保ったまま計算内容を修正するのは，アプリケーションのメイン開発者である計算科学者にとって非常にハードルが高い．OpenMP がスレッド並列 API のデファクトスタンダードとなって久しいが，近いインターフェイスを持つ OpenACC は，CUDA に比べ参入障壁を低く抑えられるものと期待される．しかし，細かい GPU への最適化が困難であることが起因し，OpenACC による単純な移植だけで高い性能を得ることは非常に困難である．OpenACC は，実アプリケーション全体の書き換えは CUDA に比べ容易だが，計算コストの高いカーネルでは CUDA による高度な最適化が必要と考えられる [76].

OpenACC version 2.0 からは，Native API (CUDA) との相互運用性が確保され，OpenACC で確保したメモリを Native API で利用できるポインタとして取得する `host_data` ディレクティブが追加されている．この仕様により，OpenACC と CUDA を共存させることが可能となった．Level1 BLAS の `daxpy` 計算を例として，`host_data` ディレクティブを用いた OpenACC+CUDA 実装を図 6.1 と図 6.2 に示す．アプリケーションが OpenACC/Fortran による実装の場合，図 6.1 の通り，まず `host_data use_device(X)` で OpenACC の API で確保したデバイスメモリを Native ポインタに変換し，C 言語で

```
integer, intent(in)    :: N
real(8), intent(in)    :: A, X(N)
real(8), intent(inout) :: Y(N)

!$acc data pcopy(X,Y)

!$acc host_data use_device(X,Y)
  call launch_cuda_daxpy(N,A,X,Y)
!$acc end host_data

!$acc end data
```

図 6.1　OpenACC+CUDA 実装の例: OpenACC/Fortran で記述されたアプリケーション本体

```
__global__
void cuda_kernel(int n, double a, double* x, double* y) {
  for(int i = ...; i < n ; i += ...)
    y[i] = a * x[i] + y[i];
}

extern "C"
void launch_cuda_daxpy(int *n, double *a, double *x, double *y) {
  cuda_kernel<<<128,((*n)+128-1)/128>>>(*n, *a, x, y);
}
```

図 6.2　OpenACC+CUDA 実装の例: CUDA/C で記述・最適化された GPU カーネルコード

実装した関数にポインタとして送る．後は図 6.2 の通り，CUDA の実行規則に従いデバイスメモリのポインタとして CUDA カーネルの引数に指定し，CUDA カーネルを起動する．CUDA カーネルは，引数に指定されたポインタの確保方法に依存することなく実行できる．本研究では，まず OpenACC 実装を行い，OpenACC 実装に対しボトルネックとなるコードのみを CUDA 化する OpenACC+CUDA 実装を行う．

NVIDIA 社との共同研究の中で，OpenACC と OpenACC+CUDA のベース実装をご提供頂いている[*1]．OpenACC 実装は次の通り修正を行った上で ARTED に取り込まれており，GPU クラスタにおいて利用可能な状態となっている．ベース実装では，ARTED で最も支配的な計算であるハミルトニアンを GPU で計算し，プロセス内の計算領域のリダクション，ハミルトニアンと同様のステンシル計算の一部が GPU 化されている．その後，共同研究の過程で計算アルゴリズムの一部が変更されたため，ベース実装では時間発展の毎ステップで計算領域すべてを GPU から CPU にコピーしなければならなくなり，メモリコピーが計算のボトルネックとなっていた．本研究では，ベース実装に対しさらにいくつかの計算カーネルの GPU 化を行い，計算領域を GPU に保持したまま計算可能にすることで，GPU–CPU 間のメモリコピーを最小限に抑えている．

[*1] https://github.com/anaruse/ARTED

6.3 アーキテクチャへの最適化

6.3.1 スレッド数・レジスタ数の調整

本節では，ステンシル計算の最適化の前に，CUDA スレッドと 32-bit レジスタの利用率について考える．本研究のステンシル計算は，先述の通り 25 点の倍精度複素数計算で，格子点 1 点の更新に必要なデータが 158 Byte と比較的多い．Tesla P100 以降，各 CUDA スレッドが使用できる 32-bit レジスタの数は最大 255 本となり，Tesla K40 と比べてほぼ倍になった．しかしながら，対象コードは倍精度複素数の計算が中心のため，Fortran90 のコードに OpenACC ディレクティブを付与するといった単純な実装では，適切なスレッド数の設定をすると各 SM が持つレジスタを使い切ってしまう．

各スレッドのレジスタ使用率の削減を考える前に，各スレッドブロックにいくつのスレッドを生成するかを考える．Warp スケジューラは， P100 では 32 CUDA コアあたり 1 個が用意され，各スケジューラはクロックあたり 2 Warp の命令をディスパッチできる．V100 は 1 SM (Streaming Multiprocessor) あたり 4 つの Warp スケジューラを持ち，各スケジューラは 32 スレッドの命令のディスパッチを 1 クロックで行う．また， P100 と V100 は SM あたり 64×2^{10} 個の 32-bit レジスタを持つ．スレッドあたりの最大レジスタ本数を 128 に設定すると，128 スレッドのブロック 4 つを 1 個の SM で実行可能な状態にできる．以上より，各 SM が持つ 32-bit レジスタをすべて活用し，かつ効率的なスケジューリングを行うためには，両 GPU ともに 128 スレッド/ブロックが最適なスレッド数と考えられる．しかしながら，スレッドあたりの 32-bit レジスタの数を 128 に制限すると，我々が対象とする倍精度複素数の 25 点ステンシル計算ではレジスタスピルが発生してしまう．

CUDA では，レジスタスピルが発生すると，SM のローカルメモリにデータを退避させる．ローカルメモリはスレッドあたり最大で 512 KiB が利用できるが，実体はグローバルメモリ (P100, V100 では HBM2) のため，アクセスレイテンシやバンド幅はグローバルメモリと等価である．また P100 は，ローカルメモリへのアクセスを常に L2 にキャッシュしている [77]．一方で，V100 ではローカルメモリへのアクセスがキャッシュされるという記述は見つかっていない．つまり，P100 では全 SM で共有される 4 MB の L2 キャッシュに，ローカルメモリ，すなわちスピルしたレジスタのデータが退避されるため，レジスタスピル発生時のローカルメモリへのアクセスコストが非常に高くなる．P100 でも，メモリアクセスがボトルネックとなる計算では，レジスタスピルは依然として性能低下の原因となる可能性が高い．

レジスタスピルの回避は，ループ分割による使用レジスタ本数の削減が最初に挙げられる．OpenACC でループ分割を行う場合，カーネルレベルで処理を分割することで，カーネルが利用するレジスタ数を削減することになる．しかしながら，現在のところ OpenACC の枠組みの中では，ループ分割による効果は得られていない．したがって，同コードについてステンシル計算を CUDA で実装し，レジスタ使用率の削減とメモリアクセスの最適化が行っている．

6.3.2 メモリアクセス最適化

まず，並列化方法について述べる．ARTED は 3 次元の実空間と波数空間を持っており，全波数空間について実空間のステンシル計算を行う必要がある．したがって，CUDA では波数空間をスレッドブロック

```
!$acc kernels
!$acc loop gang
do ikb=1,NKB
!$acc loop vector(128) collapse(2)
do ix=1,NX
do iy=1,NY
!$acc loop seq
do iz=1,NZ
  A(iz,iy,ix,ikb) = ...
```

図 6.3　本研究のステンシル計算の並列化方法 (OpenACC)

```
!$acc kernels
!$acc loop gang
do ikb=1,NKB
!$acc loop vector(128) collapse(2)
do ix=1,NX
do iy=1,NY
!$acc loop seq
do iz=1,NZ
  B(iz,iy,ix,ikb) = Cx * (A(iz,iy,mod(ix+1,NLx),ikb) - A(iz,iy,mod(ix-1,NLx),ikb))
end do
end do
end do
end do
```

図 6.4　ステンシル計算における通常のメモリアクセス (X 次元の差分計算のみ)

に割り当て，各スレッドブロックで 1 個の実空間を計算するのが最適な粒度と考えられる．CUDA GPU で動作させるコードを OpenACC で実装するが，この場合スレッドブロックは gang ループ，スレッドは vector ループと呼ばれる．OpenACC で，上記の並列化を行う場合図 6.3 のように実装する．ここで，`NX`，`NY`，`NZ` はそれぞれ 3 次元実空間のサイズ，`NKB` は波数空間のインデックスを示す．`!$acc kernels` は並列実行領域を示し，CUDA に変換される場合，単一また複数のカーネルが生成される．`collapse` でループを融合し，X-Y 平面を 1 つのループとして並列化する．これは，3 次元実空間が $(16, 16, 16)$ など比較的小さなブロックのため，ループ融合により並列性を確保している．最内の連続方向にあたる Z 次元は，`!$acc loop seq` を指定し 1 スレッドで逐次計算する．これをベースとして，メモリアクセスの最適化を考える．

本節では，NVIDIA との共同研究による実装の概要を述べる．CUDA 実装はループ分割を行い，3 次元空間のうち最外にあたる X 次元のみを計算，Y-Z 平面を計算する 2 つのカーネルに分割し各メモリアクセスを最適化する．スレッドインデックスの計算など煩雑な部分を省略するため，ここでは Fortran/OpenACC で 1 次差分のステンシル計算をコード例として記載する．

X 次元は，Y-Z ストライドのアクセスとなるため，アクセスコストが極めて高い．したがって，同カーネルではメモリアクセスのパターンを変更する．通常，図 6.4 のように，ステンシル計算では更新対象の格子点から見て，近傍点へのメモリアクセスを行うのが一般的である．この場合，各反復で複数の近傍点に対しメモリアクセスが行われるが，キャッシュメモリを活用する CPU のアーキテクチャでは，連続した近傍点に対しては効率的なメモリアクセスが期待できる．しかし GPU の場合，メモリアクセスは Warp 単位でまとめられるため，Warp 内の各スレッドが不連続にメモリアクセスを繰り返すと性能低下

```
B(iz,iy,ix,ikb) = 0
!$acc kernels private(g)
!$acc loop gang
do ikb=1,NKB
!$acc loop vector(128) collapse(2)
do ix=1,NX
do iy=1,NY
!$acc loop seq
do iz=1,NZ
  g = A(iz,iy,ix,ikb)
  B(iz,iy,mod(ix-1,NLx),ikb) = B(iz,iy,mod(ix-1,NLx),ikb) - Cx * g
  B(iz,iy,ix,ikb)            = B(iz,iy,ix,ikb) + g
  B(iz,iy,mod(ix+1,NLx),ikb) = B(iz,iy,mod(ix+1,NLx),ikb) + Cx * g
end do
end do
end do
end do
```

図 6.5 グローバルメモリへのリードアクセスを削減した場合 (X 次元の差分計算のみ)

```
complex(8) :: cache_y(3)

!$acc kernels private(cache_y)
!$acc loop gang
do ikb=1,NKB
!$acc loop vector(128) collapse(2)
do ix=1,NX
do iy=1,NY
  cache_y(1) = A(iz,NLy,ix,ikb)
  cache_y(2) = A(iz,  0,ix,ikb)

!$acc loop seq
do iz=1,NZ
  cache_y(3) = A(iz,mod(iy+1,NLy),ix,ikb)

  B(iz,iy,ix,ikb) = B(iz,iy,ix,ikb) &
  + Cy * (cache_y(3) - cache_y(1)) &
  + Cz * (A(mod(iz+1,NLz),iy,ix,ikb) - A(mod(iz-1,NLz),iy,ix,ikb))

  cache_y(1) = cache_y(2)
  cache_y(2) = cache_y(3)
end do
end do
end do
end do
```

図 6.6 Y-Z 平面のグローバルメモリへのリードアクセス削減

に繋がる．そこで，X 次元では図 6.5 のように，各反復でアクセスする 1 個の格子点が，どの格子点の更新に必要とされているか，という観点で計算を行う．このアクセス方法では，各反復でグローバルメモリへのリードアクセスは 1 回になり，またループを並び替えることで全スレッドがコアレスアクセスを実現できる．図 6.5 では，計算の一時データもグローバルメモリに書き込まれているが，実装ではローカルメモリを使用し，グローバルメモリへのライトアクセスも各格子点に対し 1 回になるように最適化している．

Y-Z 平面の計算カーネルでも，同様にメモリアクセス回数を削減する．連続方向である Z 次元は，最低限，L1 にキャッシュされることが期待されるため，Z 次元は通常通りのメモリアクセスを行う．Y 次

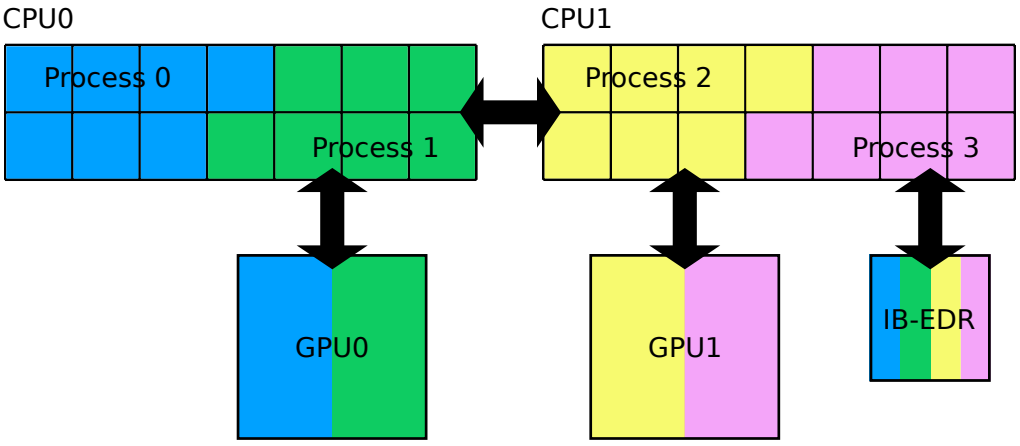

図 6.7　CPU/GPU のアフィニティ設定例

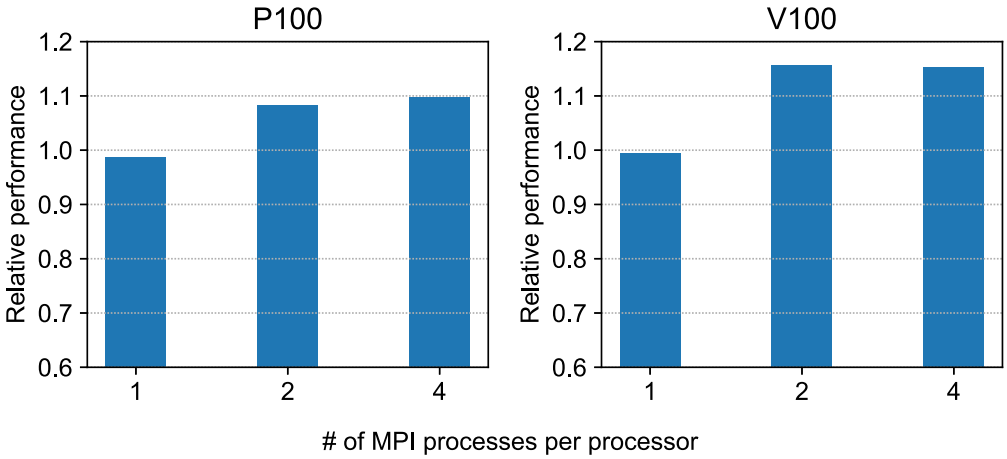

図 6.8　CUDA MPS の効果

元は Z ストライドのアクセスになるため，キャッシュから溢れる可能性が高い．したがって Y 次元については，必要な近傍点をローカルメモリに手動キャッシュすることでリードアクセスを削減する．Y-Z 平面のリードアクセス削減後のコードを図 6.6 に示す．計算時には，X 次元の計算，Y-Z 平面の計算の順でカーネルを実行する．

6.3.3　Multi-Process Service によるスループット改善

GPU クラスタは一般的な構成としてノードあたり CPU が 2 ソケット，GPU が各ソケットに 1 台以上接続されることが多く，MPI プロセスの CPU と GPU への割当方法は性能に大きく影響する．また CPU 実装の並列性の問題から，MPI による問題の細分化と複数 MPI プロセスによる GPU が共有 (非同期的な GPU へのタスク割当) される場合もある．そこで，GPU の計算資源の利用効率を最大化する仕組みとして，CUDA MPS (Multi-Process Service) が提供されている．

CUDA MPS は，ユーザまたはシステムレベルで GPU への実行キュー (CUDA カーネル) の制御を行うデーモンを立ち上げ，SM に余裕があり依存関係のないキューが投入されている場合に，同時に複数の

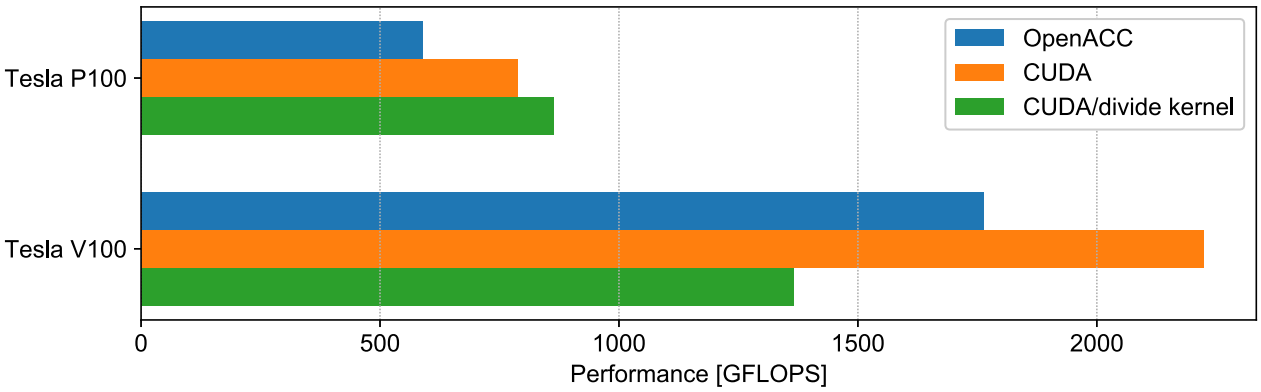

図 6.9 ステンシル計算の OpenACC と CUDA 実装の性能比較

キューを処理するようにスケジュールする．この仕組みにより，GPU は投入されたキューに対し適切に SM を割り当て，計算資源の利用効率を引き上げることができる．ただし，CUDA MPS デーモンが CPU プロセスと GPU の仲介を行うため，同時に複数キューを処理できない，すなわち SM に空きがない場合はすべて実行時オーバーヘッドとなる．Volta アーキテクチャでは，CUDA MPS をハードウェアベースで実装し，更なる性能向上を図っている [14]．また MPI プロセス間では，GPU カーネルに依存関係は発生しないため，各 MPI プロセスで投入されるカーネルを可能な限り同時に処理することが可能である．

CUDA MPS を経由して複数の CUDA カーネルが絶えず投入されることで，計算資源を常に最大限活用できることが期待される．PPX では GPU が各ソケットに 1 台接続されているため，ソケットに複数の MPI プロセスを割り当て，1 MPI プロセスで各 GPU を専有した場合との性能比較を行う．図 6.7 には，ソケットあたり 2 MPI プロセスで GPU を共有したときの CPU と GPU のアフィニティ設定例を示す．PPX ではソケットあたり 14 物理コアを持つため，ソケットあたり 2 MPI プロセスの場合，各 MPI プロセスは 7 OpenMP スレッドでプロセス内 CPU 並列を行う．MPI プロセスの割当数によっては，OpenMP の合計スレッド数は CPU が提供するコア数よりも少なくなる．

CUDA MPS の効果について図 6.8 に示す．このグラフでは CUDA MPS を無効にした状態で，GPU あたり 1 MPI プロセスで得られる実性能を基準とした相対性能を示す．グラフ横軸は，GPU を共有する MPI プロセス数を示し，1 MPI プロセスの場合は専有状態を意味する．1 MPI プロセスで 1 個の GPU を専有した場合，CUDA MPS はそのままオーバーヘッドとなり，1% と無視できる程度の性能低下が見られる．複数 MPI プロセスで GPU を共有することで CUDA MPS が適切に CUDA カーネルの実行を制御し，P100 では 1.1 倍，V100 では 1.15 倍とそれぞれ効果が得られている．

6.4 性能評価

6.4.1 ステンシル計算

まず，4 MPI プロセスで GPU を共有実行した場合のステンシル計算の性能を図 6.9 に示す．比較用に，PGI OpenACC の最適化メッセージを参照し，OpenACC 実装とほぼ同じ動作を想定した CUDA 実装 (シングルカーネル) を “CUDA” として評価した．CUDA は OpenACC と同一性能が期待されるが，

表 6.2　OpenACC と CUDA のレジスタ使用数

	OpenACC		CUDA	
Max # of registers	255	128	255	128
Used # of 32-bit registers	198	128	180	128
Spill stores+loads [B]	0	104+112	0	0

実際には OpenACC よりも高い性能が得られている．“CUDA/divide kernel” は，前述したメモリアクセス最適化やレジスタ数削減などの最適化が行われた実装である．

表 6.2 に，PTX の出力情報から P100 における OpenACC と CUDA 実装のレジスタ使用数を示す．V100 においても，ほぼ同じ傾向を示している．各スレッドが利用できる 32-bit レジスタ数は最大 255 で，本研究では各 SM にある全レジスタを活用できるように，各スレッドで利用可能なレジスタを 128 に制限している．OpenACC では，レジスタ数を制限することでレジスタスピルが発生しているが，一方で CUDA 実装では，レジスタ数を制限してもレジスタスピルは発生せず，使用レジスタ数だけが減少している．この違いは，レジスタ数制限の方法によって生じている．OpenACC では，PGI コンパイラに `-ta=tesla,maxregcount:128` とオプションを指定し，CUDA GPU の命令セットにあたる PTX コード上でレジスタ数を制限するが，CUDA ではカーネルに `__launch_bounds__(128,4)` を付与し 128 スレッド 4 ブロックで最低限動作する，つまりレジスタ数を 128 に抑えてコンパイルして PTX コードが出力される．CUDA でも，`__launch_bounds__`の変わりに PTX コードを出力する `ptxas` のプログラムオプションで`--maxrregcount=128` を指定するとレジスタスピルの発生を確認した．恐らく，`__launch_bounds__` では制限値に応じて最適化が CUDA コンパイラによって行われ，レジスタスピルがない PTX コードが出力されると考えられる．PGI OpenACC でも，同様の指定ができれば CUDA 実装と遜色ない性能が得られるはずである．

P100 の性能を検証すると，OpenACC 実装ではレジスタスピルにより，十分な性能が得られていない．レジスタスピルが発生すると，レジスタから溢れたデータはローカルメモリに退避される．ローカルメモリへのアクセスは，P100 では常に L2 キャッシュを経由するため，4 MiB しかない L2 キャッシュに GPU 上の全スレッドがデータを退避させることになり，性能低下に繋がると推察される．P100 では CUDA 実装のようにカーネルを 2 つに分割して，レジスタスピルの発生を回避したほうが高い性能を得られると考えられる．その一方で，V100 では最適化したはずの CUDA/divide kernel が OpenACC 実装よりも低い性能となっており，現在原因を調査中である．

6.4.2　計算全体

計算全体の性能評価について，図 6.10 に示す．横軸を GPU の台数としているが，使用している計算ノード数はこの半分であり，計算資源上の制約から P100 は 8 台，V100 は 2 台までの評価となる．計算全体であることから，MPI による通信や最小限の GPU-CPU 間のメモリコピーなどが発生していることに注意されたい．

ステンシル計算では P100 に対し約 2.5 倍の性能を V100 で達成しているが，計算全体の性能では約 2 倍にとどまる．ステンシル計算の場合，カーネル単位での性能評価のため純粋な GPU の性能だけが評価

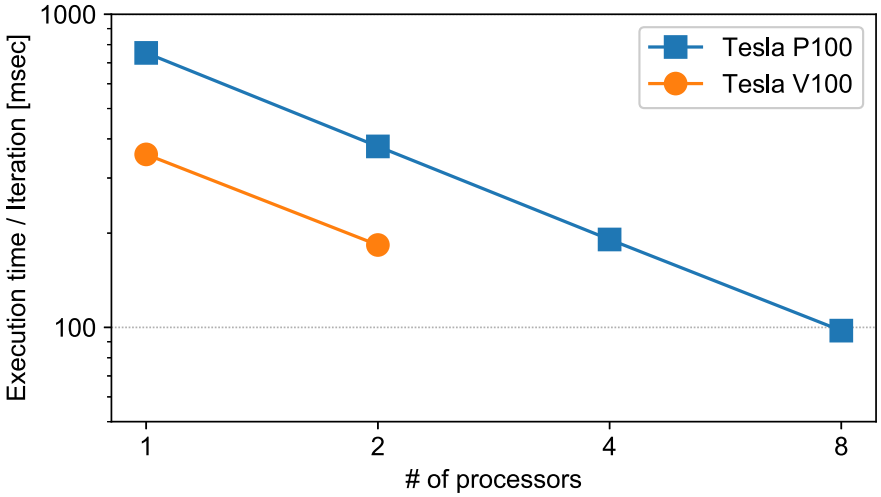

図 6.10 計算全体の性能評価

されているが，計算全体の場合は GPU と CPU 間のメモリ転送やカーネル実行のスケジューリングなど，本来必要な GPU の制御オーバーヘッドが発生する．結果として，計算全体の性能は純粋な計算性能よりも鈍化する傾向にあるが，それでも V100 は P100 に対して約 2 倍の実効性能を達成しており，理論性能の差よりも大きく性能向上している．また，P100 と V100 の評価環境は GPU 以外はすべて同じであることから，V100 の潜在的な計算性能が理論性能の差よりも高いことを示していると考えられる．

6.5 考察

V100 でも，OpenACC 実装は依然としてレジスタスピルが発生しており，十分な性能は期待できない．先に述べたように，P100 ではローカルメモリへのアクセスを L2 キャッシュ経由で行うことが明記されているが，V100 では現在その記述が見つかっていない．もし V100 がグローバルメモリに直接データを退避するとすれば，レジスタスピル時に L2 キャッシュへのアクセス集中を避けることができ，性能低下を緩和できると推察される．レジスタスピル時のローカルメモリのアクセスペナルティが緩和されたとすれば，レジスタスピルを避けるためにマルチカーネルで実行する CUDA 実装よりも，シングルカーネルでレジスタスピルが発生する OpenACC 実装が高速である理由を解決する手がかりとなる．

また図 6.9 の通り，V100 は P100 に対して約 2.5 倍の性能をステンシル計算で達成している．それぞれの GPU が持つ HBM2 のメモリバンド幅で比較すると，表 6.1 の通り V100 は P100 に対して約 1.5 倍のメモリバンド幅を持ち，ステンシル計算の性能が HBM2 のメモリバンド幅の差よりも広がっている．メモリバンド幅律速なアプリケーションにおいて，V100 での実効性能が単純なメモリバンド幅の差よりも大きく広がる場合があることがいくつかの研究で報告されている [78, 79] が，本研究のアプリケーションではワーキングセットのサイズと L1 キャッシュの性能向上が要因と考えられる．

V100 では，L1 キャッシュとシェアードメモリが統合され合計 128 KiB に変更されており，アクセスレイテンシやバンド幅も改善されている．特に，P100 と比べると L1 キャッシュのアクセスレイテンシは 82 cycles から 28 cycles，L1 キャッシュの転送スループットは 31.3 Bytes/cycle から 108.3 Bytes/cycle

と実測値で約 3.4 倍ほどの向上が報告されている [80]．本研究における性能評価では，スレッドブロックに割り当てられるワーキングセットのサイズが 64 KiB であるため，V100 が持つ 128 KiB の L1 キャッシュに収まっている．したがって計算性能は L1 キャッシュの性能が大きく影響し，HBM2 のメモリバンド幅や理論性能よりも実性能の差が広がったものと考えられる．またこれらの改善により，OpenACC 実装のように単純な実装でも高い性能が得やすくなったのではないかと考えられる．

第 7 章

考察・議論

本章では 3.4 節で述べた評価軸より，第 5 章と第 6 章で行った性能評価から，コデザインの考察を行う．最後に，エクサスケールコンピューティングとその先に向けたアーキテクチャやシステムへの期待をまとめる．

7.1　コデザインの検証

7.1.1　各アーキテクチャが達成する実性能の妥当性

本研究で対象としている Intel Xeon Phi 7250 (KNL) と NVIDIA Tesla P100 (Pascal), Tesla V100 (Volta)，さらに従来の CPU として SKL (Skylake-SP) アーキテクチャを採用する Intel Xeon Gold 6148 を用い，コデザインにおける最適化の定量的評価を行う．

表 7.1 に，各アーキテクチャの理論性能やメモリバンド幅，電力効率などの比較を行う．また，KNL の MCDRAM は理論メモリバンド幅が公表されていないため，STREAM ベンチマークで計測した実効メモリバンド幅を示す [81, 73]．SKL の実効メモリバンド幅は NUMA を適切に制御して 1 ソケットの値を記載し，理論性能は AVX-512 動作時の周波数から求めている．Pascal と Volta の各理論値は，すべて PCIe カードのプロダクトでの値を記載している．

ステンシル計算は，KNL が HPL 性能比で約 38% に対して，Volta は約 54.9% の実行効率を達成している．KNL での計算は MCDRAM で閉じており，Byte/FLOP が最も高いが，MCDRAM のメモリアクセスレイテンシが DDR4 メモリに比べて高いことが一因と考えられる．Volta は Pascal に対しステンシル計算の性能が約 2.2 倍と，メモリバンド幅の差を超えて性能が向上した．6.5 節で述べたとおり，Volta のステンシル計算は L1 キャッシュの性能に律速されていると考えられる [80]．SKL は高い実行効率を得ているが，SKL は Non-inclusive な L3 キャッシュを持ち，L2 キャッシュのアクセスレイテンシがこれまでのアーキテクチャに比べ短くなっている．本研究のステンシル計算は，タスクに相当する 1 個の実空間が，一般的な CPU が持つ L2 キャッシュ (512–1024 KiB) に収まる可能性が高い．比較的小さな実空間を非常に多数，OpenMP の各スレッドが独立して計算するため，L2 キャッシュのアクセスレイテンシは性能に大きく影響する．

“ARTED 相対性能” は，SKL プロセッサ 1 台の性能を基準とした値を算出している．SKL を採用す

表 7.1 各アーキテクチャの性能比較

	Skylake-SP	Knights Landing	Pascal	Volta
プロダクト名	Xeon Gold 6148	Xeon Phi 7250	Tesla P100	Tesla V100
理論性能 [GFLOPS]	1,024	3,046	4,800	7,000
HPL 性能 [GFLOPS]	992.1	1,993.3	3,333.6	4,053.0
理論メモリバンド幅 [GB/s]	127.8	none	732	900
実効メモリバンド幅 [GB/s]	98.5	490.4	551.3	839.1
Byte/FLOP	0.096	0.160	0.114	0.119
TDP [W]	150	215	250	250
GFLOPS/W	6.82	14.16	19.2	28
ステンシル計算性能 [GFLOPS]	419.8	758.4	864.0	2,226.0
理論性能比	41.0%	24.9%	18.0%	31.8%
HPL 性能比	42.3%	38.0%	25.9%	54.9%
ARTED 相対性能	1.0	1.6	1.6	3.3

表 7.2 想定される計算ノード構成

	Skylake-SP node	Pascal node	Volta node
CPU	Xeon Gold 6148 ×2	Xeon Gold 6148 ×2	Xeon Gold 6148 ×2
GPU	–	Tesla P100 ×2	Tesla V100 ×2
総理論性能 [GFLOPS]	2,048	11,648	16,048
総実効メモリバンド幅 [GB/s]	197	1,299.6	1,875.2
総 TDP [W]	300	800	800
GFLOPS/W	6.82	14.56	20.06

る CPU クラスタの場合，計算ノードは 2 ソケットで構成されることが一般的で，SKL の計算ノードと比較すると各プロセッサの相対性能はこの半分である．すなわち本研究の実装によって，SKL プロセッサを唯一の計算リソースとして持つ計算ノードは実効性能だけで見れば KNL と Pascal を凌いでいる．実アプリケーションの実装は KNL と SKL で同一であり，この結果は電力効率を犠牲にした結果とも言える．また，KNL は第 5 章で述べた通り多くの最適化を費やしたにも関わらず，実行効率は SKL よりも低い結果となっていると言わざるを得ない．

SKL は KNL と異なり，アプリケーションの逐次性が高い場合でもある程度の性能が期待できる．プロセッサあたり最大で 28 個の演算コアを持ち，Hyper Threading により各コアは 2 スレッドを処理できることから，SKL もメニーコア CPU とみなせる．また，5.6.2 節に示したように本研究で行ってきた最適化も利用できる．理論性能比や実装の可搬性を考えれば，たしかに SKL などの従来 CPU で構成された CPU クラスタは，最も利便性に優れている．その一方で，GPU は並列化モデルの変更による最適化に加え，GPU 間のデータハンドリングをしなければならず実装は容易ではない．

GPU はどちらも 250 W と比較的高い TDP を持つが，電力効率は KNL に比べて高い．対して SKL の電力効率は最も低く，動作周波数が演算コア数の増加と反比例せずに維持されていることが要因と考えられる．各プロセッサは単独で用いられることは少なく，複数台で計算ノードが構築される．表 7.2 に，同水準の CPU を GPU のホストプロセッサとした場合の，現実的な CPU クラスタと GPU クラスタに

表 7.3 GPU に要した追加実装 (SLOC: source lines of code)

	SLOC	関数定義数	OpenACC ディレクティブ行数
OpenACC/Fortran の追加実装	844	17	152
+ CUDA/C の追加実装	+ 406	+ 15	+ 2

おける計算ノードの性能と電力効率について示す．Xeon Phi ノードは Oakforest-PACS や Cori などがプロセッサ 1 台で計算ノードを構成しているため，値は表 7.1 と同一である．同表では，SKL プロセッサが 2 台で，GPU は InfiniBand といった通信デバイスとの共存 (PCIe レーン数の制限) から各 CPU に 1 台接続される構成を想定した．SKL の電力効率の低さが影響し，P100 や V100 の電力効率はプロセッサ単体時と比べて 25% 以上の低下が見られるが，理論上の電力効率は 15 GFLOPS/W 以上と，依然として KNL を凌ぐ値を示している．

HPL 性能と比べれば，全てのプロセッサでステンシル計算の性能は最低でも 25% を超え，Volta では約 55% の実行効率を達成している．理論性能と実性能の乖離が最も激しい KNL においても，HPL 比で約 38% の効率を達成していることから，各アーキテクチャで十分な実性能を達成できたと考えられる．

7.1.2 実アプリケーションの協調開発

本節では実装方法に関して，主に定性的評価を行う．4.3 節に従い，メニーコア CPU に対するコデザインでは，従来 CPU でも実行できるように並列化構造を変えずに実装と最適化を行った．したがって，Xeon Phi や Xeon CPU の性能評価では，すべて同じ OpenMP+MPI 実装が使われている．ステンシル計算では，AVX や AVX-512 など SIMD 命令の Intrinsics を用いた最適化を行い，各 SIMD 命令の手動ベクトル計算を実装した．本研究の遂行中，同計算が含まれるハミルトニアン計算のアルゴリズムの変更は一度も行われず，ユーザはブラックボックス的に最適化されたステンシル計算実装を利用している．また，アプリケーションの開発過程で最適化したステンシル計算に近い計算が必要となったが，リファレンス実装として提供していたステンシル計算のコンパイラベクトル化向け実装を用いて，計算に沿った形で修正した実装がユーザの手で行われた．言い換えれば，最適化によって実アプリケーション開発を妨げないだけでなく，開発を支援したと言える．

実アプリケーションの GPU への実装について考えると，大規模実アプリケーションはそのことごとくが OpenMP+MPI によって実装されているはずで，ソフトウェア資産が一切ない状態，ゼロから開発することは非常に稀である．結局 “GPU 実装” は，既存の CPU 実装に対し，GPU への最適化と制御コードを追加して GPU に対応することを意味し，ソフトウェア規模を増加させる．OpenMP 4.0 のアクセラレータ拡張による実装も考えられるが，OpenACC と同様に並列化モデルに合わせた最適化が必要となるため，ソフトウェア規模の増加を解決できない．本研究でコデザインのターゲットとした ARTED も，もとより OpenMP+MPI で実装され，本研究遂行の過程で OpenMP+MPI 実装 (CPU)，OpenMP+MPI+OpenACC/CUDA 実装 (GPU) をひとつのソフトウェアパッケージとして提供した．本研究で最適化した OpenMP+MPI 実装に対して，GPU 実装によるソフトウェア規模の増加を評価することで，GPU 実装コストを数値化する一助となる．

表 7.3 に GPU に要した追加実装について，SLOC (source lines of codes) と関数定義数を示す．ここ

で，関数は Fortran の subroutine/function，C/C++ 言語における関数の総数である．SLOC によるソフトウェア規模の数値化が真に有効であるか，本研究で議論するにはあまりにも材料が不足しているが，ひとつの基準として記載する．ARTED では，主たる時間発展計算を GPU (OpenACC/Fortran) で計算するために 844 行が追加で必要となる．実アプリケーション全体の行数はおよそ 2.2 万行で，単純な行数では高々 4% しか増えていない．追加したコードの大半は並列化方法を GPU へ適した形にするため，アクセスや計算順序，ループ方向などが変更されており，OpenMP 実装と完全に異なるが，同じ計算結果を返す関数が複製されている．

CPU と GPU の各アーキテクチャ実装を 1 つのソフトウェアパッケージとして提供する場合，各アーキテクチャに最適化された複数の実装を維持する必要があり，保守・拡張性を下げてしまう．既存実装に修正や拡張を行う場合，多くのアプリケーションユーザはまず CPU (OpenMP+MPI) で動作するコードだけを実装してしまう可能性が十分に考えられる．普段の開発サイクルの中で，GPU のために別実装を考え，手間をかけるという可能性は非常に低いと考えられる．メニーコア CPU でも手動ベクトル化実装といった高度な最適化により同じ問題が発生する可能性はあるが，本研究においては将来に渡るコード変更の可能性が低いステンシル計算に限っており，CUDA を用いた GPU への最適化も同じである．以上の問題は CPU と GPU で並列化モデルが異なることに起因しており，性能とのトレードオフで現在のアーキテクチャでは避けることが困難である．最初から GPU 利用を前提で開発を行えば，CPU と GPU の両実装を共存する必要はないかもしれないが，既に述べたように GPU のサポートは既存のアプリケーションの拡張であることがほとんどで，現実に即していない．

メニーコア CPU におけるコデザインは開発との同時進行に影響を与えず遂行できているが，GPU は性能と保守・拡張性に明らかなトレードオフがある．しかしながら，V100 は OpenACC でも HPL 比で約 46% の実行効率を達成しており，アーキテクチャの進化によるトレードオフの緩和が期待される．

7.1.3 次世代システムへの適用可能性

本節では各アーキテクチャに対し実施した最適化について，次世代システムへどのように適用できるかを考える．

メニーコア CPU に対する最適化は，アプリケーション全体を考えれば，デファクトスタンダードである OpenMP+MPI による並列化モデルは今後のメニーコア CPU システムにおいても有効と考えられる．特殊な例として，Sunway TaihuLight のように制御コアと演算コアが別に存在する場合，GPU のようにアクセラレータ向けのプログラミングモデルが必要となる可能性はある．より一般化すると，Xeon CPU と Xeon Phi が何らかの NOC (Network on Chip) で接続され，アプリケーションの制御を Xeon CPU，計算を Xeon Phi で行うといったケースである．本研究は GPU との実装が共存しており，もし次世代システムがそのような構成を取るようになっても，OpenMP+MPI の実装はそのまま利用できることが期待される．

ステンシル計算は SIMD 命令による最適化を行ってきたが，実行するプロセッサによって，命令セットごとの実装と最適化が必要となり，可搬性は著しく低い．しかしながら，高性能計算システムでは概ね Intel が提供する AVX，AVX2 などに準拠した命令セットが提供されることが多く，実装はある程度の移植性が期待できる．重要なのは SIMD 命令のフォーマットではなく，ハードウェアが提供する SIMD 長

に応じた計算の最適化であるため，どの命令セットにおいても最適化で考慮するべき内容はほぼ同一と考えられる．2020 年以降に実現される予定のポスト「京」コンピュータでは，512-bit 長の SIMD 命令として ARM SVE に HPC 向け拡張命令を追加した ARM プロセッサを実装する．SVE がプリミティブな命令セットしか持たないことは，5.6.2 節で述べたとおりである．たしかに HPC-ACE に近いものが拡張命令セットとして提供されることが望ましいが，そうでなければ複数の SVE 命令を組み合わせて実装しなければならない．SIMD 最適化については，SIMD 長と命令セットに応じた複数の実装が必要となることから，可搬性に問題があり，システムに応じて考える必要がある．

コンパイラに頼った最適化を行えば，高い可搬性をもって実アプリケーションを次世代システムへ移植できることが期待されるが，本研究で述べたようにボトルネックとなる計算において，コンパイラによる最適化は不十分であると言える．コンパイラの成熟により手動ベクトル化実装との差は縮まりつつあるため，将来的には逆転する可能性もあるが，成熟した頃には SIMD 長が増加し陳腐化している可能性もある．結局は性能と可搬性のトレードオフで，選択は開発者に委ねられる．

NVIDIA GPU と CUDA プラットフォームは，高性能計算を対象とする GPU システムのデファクトスタンダードである．したがって，GPU への最適化はアーキテクチャによる違いはあってもプラットフォームは今後も同じソフトウェアが提供され，可搬性は高いと考えられる．開発当初よりデファクトスタンダードであったことを考えれば，NVIDIA GPU がその座から降りることは考えづらい．また OpenACC 実装と CUDA 実装の差が縮まりつつあることから，ディレクティブベース言語による開発のデメリットが解消されてきており，最適化をコンパイラに頼りやすくなっている．しかし最適化以前の問題として，ディレクティブベース言語を実用レベルでサポートしているコンパイラが非常に限られている．Omni compiler や OpenARC，OpenUH など研究用途の実験的な複数のコンパイラにおいて実装されているものの，OpenACC/C しかサポートしていない．GCC は OpenACC と OpenMP 4.0 のアクセラレータ拡張のどちらもサポートしているが，第 3 世代にあたる Kepler 世代に準拠した PTX コードしか出力されず，ユーザが指定する手段もなく，Volta 世代が第 6 世代にあたることからも性能評価に利用できるものではない．最適化が期待される商用コンパイラでは，NVIDIA 傘下の PGI コンパイラと，Cray コンパイラしかサポートしていない [75]．また，OpenMP 4.0 のアクセラレータ拡張は商用コンパイラである IBM XL コンパイラが精力的にサポートするにとどまっている．スレッド並列化 API のデファクトスタンダードである OpenMP がより広範にサポートされていることを考えれば，ディレクティブベース言語によるアクセラレータプログラミングは可用性に欠ける．

7.1.4　総括

以上より，本研究におけるコデザインの達成状況について表 7.4 に示す．本表では前節までの考察について，アプリケーションの実性能，アプリケーションユーザとの協調開発の可否，最適化されたアプリケーションの可搬性についてまとめている．

実性能は両アーキテクチャ共に HPL 比で約 40–50% のステンシル計算効率が得られており，理論性能と実性能の乖離を考慮すれば比較的高く，どちらも達成できている．メニーコア CPU においては，最大規模の KNL クラスタ Oakforest-PACS 上での全系実行を達成し，大規模システムにおける高いスケーラビリティを示した．今後の課題として，大規模 GPU システムにおける性能評価が必要と考えられる．

表 7.4 コデザインの達成状況

	Many-core CPU	GPU Accelerator	評価方法
実性能	◯	◯	HPL 性能より
協調開発	◎	△	保守性と拡張性より
可搬性	△	◯	将来のシステム予測より
達成可否	◯	△	協調開発を優先

協調開発は，メニーコア CPU は OpenMP+MPI の並列化構造や最適化の度合いについて明確にした結果，アプリケーションユーザによる実装や修正を妨げることなく，最適化を実現できている．GPU は性能と保守性にトレードオフがあり，メニーコア CPU よりも高い性能ながら普段の開発サイクルにおける GPU 実装の重要度を考慮すると，保守性は非常に低いと言わざるを得ない．これは並列化モデルの違いにより発生する問題であり，CPU 実装と共存して GPU を適用する上では避けられない問題と言える．

メニーコア CPU における最適化は，ベクトル並列化は SIMD 命令セットの違いよりアーキテクチャに依存した最適化となってしまい，次世代システムへの適用は性能と可搬性のトレードオフ関係にある．しかしながら，SIMD 最適化で取るべき戦略は命令セットに依らないと考えられること，本研究では将来に渡って変更が想定されない成熟したステンシル計算だけを SIMD 命令セットで実装したことから，深刻な問題とは言えない．GPU はプラットフォームがデファクトスタンダード化し可搬性は高いと言えるが，コンパイラの問題によって可用性が妨げられる可能性がある．

以上の評価より，コデザインが達成されているかを議論する．ソフトウェアのエコシステムが重要視される中，実アプリケーションも持続的に開発が行われるものと考えると，コデザインにおいて，協調開発の可否は非常に重要であると言える．両アーキテクチャ共に最適化が重要であることは本研究より明白だが，可搬性や保守性のトレードオフがあり，最高性能を得るためにはこれらを犠牲にしなければならない．高性能計算において，最高性能を達成するのはもちろんひとつの目的であるし非常に重要だが，協調開発を掲げるコデザインにおいて，アプリケーションユーザを無視し保守性を犠牲にすることは果たして最適解なのか，大いに疑問である．最後の可搬性も当然重要だが，SIMD 命令を用いた手動ベクトル化計算は対象範囲が限られており，たしかに最適化の可搬性は低いが，アプリケーションの可搬性は決して低くない．重要度が高い順に協調開発，実性能，可搬性と並べると，メニーコア CPU はある程度を達成できていると考えられる．一方，GPU は協調開発に問題を抱えているため達成できているとは言い難い．

7.2 その他の議論

7.2.1 アーキテクチャ間評価

前節での議論や評価を踏まえ，各アーキテクチャの評価を表 7.5 に示す．逐次処理性能や並列処理性能は述べたとおりだが，最適化を踏まえると，従来 CPU は電力効率を犠牲にしながらも高い実行効率を持っていると言える．アプリケーション開発コストは，メニーコア CPU は従来 CPU での実装を踏襲できるが，積極的に並列化や最適化が必要と言える．GPU は並列化の違いから実装の複製が必要となり，保守・拡張性を下げてしまう．システムの開発コストは，GPU は制御用のホスト CPU が必要な分，コ

表 7.5　アーキテクチャ間評価

	Traditional CPU	Many-core CPU	GPU Accelerator
逐次処理性能	◎	△	×
並列処理性能	◯	◎	◎
理論性能	◯	◎	◎
実行効率	◎	◯	◯
電力効率	×	◯	◎
アプリケーション 開発コスト	OpenMP+MPI 従来実装を維持可能	OpenMP+MPI 積極的な並列化が必要	OpenMP+MPI+OpenACC 並列構造の変更が必要
システム 開発コスト	理論性能と電力効率の トレードオフ	理論性能と電力効率の のトレードオフ	別途ホスト CPU が必要
メリット	電力効率を考慮しなければ 高い実行効率を達成可能	最適化ができれば 電力効率と性能を両立可能	開発コストを考慮しなければ 性能と電力効率を両立可能

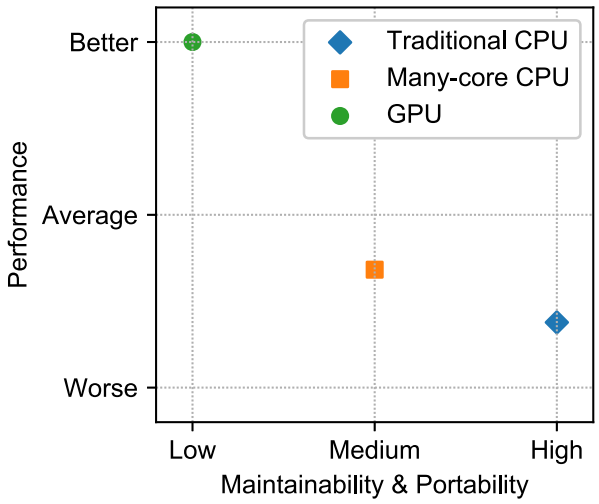

図 7.1　Performance vs. Maintainability & Portability

ストは増加傾向にある．

以上をまとめると，各アーキテクチャを採用するメリットは次の通りである．

Traditional CPU (Xeon)

電力効率を考慮しなければ，アプリケーション開発コストは低い状態で，高い実行効率を達成可能

メニーコア CPU

アプリケーション開発コストは高いが，最適化が可能であれば，電力効率と実行効率を両立可能

GPU (Accelerator)

開発コストを考慮せず，最適化が可能であれば，電力効率と実行効率を両立可能

性能と保守・可搬性について，指標を図 7.1 に示す．ここで，“Performance” は本研究で得られる実性能，“Maintainability & Portability” は，保守・可搬性の議論より決定している．“計算科学とのコデザ

表 7.6 本研究で取り上げたうち条件を要する最適化

最適化	適用条件	補足
非アラインメモリアクセス	連続方向次元数がベクトル長の倍数	倍精度複素数以外でも適用可能
近傍点のインデックス計算	多数のインデックスを求める場合	複雑な計算が必要な場合に効果が期待
周期境界領域の拡張	周期境界条件を持つ	付随コストについて検証が必要

イン” では，性能と保守・可搬性などのトレードオフの議論により，メニーコア CPU や GPU の利用有無を決定するべきである．しかし，これらの意思決定はスーパーコンピュータを用いた大規模シミュレーションを求める計算科学側にあるはずで，高性能計算はこれらのトレードオフについての選択肢を提示することによって，利用有無の議論の材料としなければならない．

性能や可搬性のみならず，スーパーコンピュータを提供する多くの研究機関，大学などにとって費用は大きな問題である．計算ノードを 1 台構築するために必要な金額から，購入に充てられる費用から設置可能なノード数が決定され，ネットワークスイッチの台数なども大きく影響する．GPU クラスタの場合，GPU だけでなく制御用のホスト CPU を要するため，計算ノードの単価は CPU クラスタに比べて圧倒的に高い．東京工業大学の TSUBAME 3.0 や筑波大学の HA-PACS など，ノードあたり GPU 4 台などの高集積クラスタが多く実装され，ノードあたりの性能と価格が上昇傾向にある．例えば 2018 年 6 月の TOP500 で最高性能を持つ ORNL の Summit は，計算ノード 1 台に 6 台の Volta GPU が接続され，計算ノードは合計 4,608 台と，非常に高密度設計である．設置スペースの問題，冷却システム，ストレージ容量なども，計算性能には直接影響しないが費用面では重要な要素になりうる．最後に，運用コストとして電力効率は極めて重要で，4 MW でも年間数億円単位の電気料金がかかることは，第 2 章で述べた通りである．本研究ではシステムの導入や運用保守のコストについては強く述べないが，最先端アーキテクチャ，プロセッサをどのように導入していくかという話においては，避けられない問題である．

7.2.2 他アプリケーションへの最適化の適用

本研究では，電子動力学シミュレーション ARTED に対し，メニーコア CPU と GPU のそれぞれの特徴から最適化を行ってきたが，これらを他のアプリケーションに対しどう適用できるかについて議論を行う．特に ARTED が計算する特殊なパターンのステンシル計算について特に最適化を行ったが，ARTED のステンシル計算は各 OpenMP スレッドがメモリ空間上も独立したドメインについて独立に計算を行い，各実空間をタスクとして扱っている．実施した最適化によって，メニーコア CPU および GPU どちらにおいても高い計算性能を達成している．

他分野で用いられる一般的なステンシル計算は，巨大な 1 個の実空間を MPI で分散，OpenMP で並列計算する．このとき，OpenMP のスレッド並列化によって，各スレッドが計算する実空間はいくつかのメモリセグメントに分割され，キャッシュブロッキングや適切な分割数などメモリアクセスコストの削減が必要となる．ARTED のステンシル計算は 1 個の小規模実空間を 1 スレッドで計算し，かつ各実空間は L2 キャッシュにほぼ収まるサイズのため，メモリアクセス上で有利な条件となっている．しかし，本研究で実施した最適化の多くは SIMD レベルの最適化であり，実空間分割の有無に依らずすべてのステンシル計算に応用可能である．

したがって，格子点のサイズや問題に依存するメニーコア CPU への最適化について整理し，表 7.6 に適用条件を持つ最適化について示す．手動ベクトル化では，連続方向の Z 次元のサイズが 4 の倍数，すなわち倍精度複素数ベクトル長の倍数であることを唯一の条件として実装および最適化を行った．非アラインメモリアクセス最適化のみが同条件を前提として，load 命令および alignr 命令による近傍点データの整形を行っている．多くのステンシル計算では，格子サイズを増加させることは問題の求解精度を向上させることと等しいため，格子サイズを増やしベクトル長の倍数となるように調整することは可能である．また，同最適化で 8×4 の正方行列を生成したが差分の深さが 4 でない場合，深さ N とすると，$2N \times N$ の行列を生成するように修正し，深さ 4 以外のステンシル計算に対応可能である．

周期境界条件を含む複雑な近傍点インデックス計算のベクトル化は，非常に問題依存の最適化で，各次元で正負方向計 8 個の近傍点を必要とすることを踏まえ，非連続方向の Y および X 次元のインデックス計 16 点分をベクトル演算でまとめて求める．本研究で取り上げた計算のように，非常に多数の近傍点が必要な場合や，周期境界条件などインデックス計算が複雑な場合，インデックス計算のベクトル化は効果が期待できる．

最後に，周期境界の拡張だが，同最適化は周期境界条件を持つ問題全てに適用可能である．特に，自動ベクトル化実装では高い効果が期待されるが，KNC では拡張した領域へのデータコピーコストは無視できないため，シミュレーション全体で性能向上が得られるかは留意が必要となる．

7.3　エクサスケールコンピューティングとその先に向けて

コデザインの観点から，2020 年前半を目処に実現予定のエクサスケールやその先のシステムにおいて，どのような技術やアーキテクチャの収束が必要となるかを議論する．なお，von Neumann 型コンピュータの終焉と次世代コンピュータに関する議論が始まって久しいが，この議論は他の研究に譲る．

実アプリケーションのコデザインにおいて，各 CPU と GPU 実装の共存と保守性，拡張性を両立させるためには，各ハードウェアによって異なる並列化モデルの統一が必要と考えられる．協調開発の議論から，OpenMP+MPI の構造維持ができるメニーコア CPU は，コデザインにおける実アプリケーション開発において有効な選択肢と言える．しかしながら，ORNL の Summit が HPL 性能で 100 PFLOPS を超え，GPU の活用はスーパーコンピュータ開発と大規模計算による成果還元という観点から極めて重要となっている．GPU における実アプリケーションのコデザインは，CPU 実装との共存を考えたときに保守・拡張性に乏しいと述べたが，突き詰めればハードウェアに最適な並列化モデルの違いによるところが大きい．

GPU システムは圧倒的に絶対数が少なく，GPU のみを対象とした実アプリケーションの開発はエコシステムを考えたときに現実的でない．ディレクティブベース言語は元のアプリケーション構造を維持可能なように規格化されているが，並列化構造はアーキテクチャに依存し，アプリケーションの構造を維持した最適化は非常に困難である．しかし，メニーコア CPU も各演算コアがフラットなネットワークで接続される訳ではなく，PEZY-SC プロセッサのように階層化された複雑なネットワークが実装されることもある [82]．ホストプロセッサとして動作するメニーコア CPU もさらに省力化，コア数の増加によって階層化されたアーキテクチャとなる可能性は大いに考えられ，もしそうなれば OpenACC や CUDA の階層型並列化モデルが一般的になる．しかし，階層型並列化はハードウェアに完全依存し最適な階層化が各

ハードウェアで異なる可能性が高く，並列化モデルの一般化だけでなく，ハードウェア非依存での性能向上が求められるだろう．

電力効率の観点からメニーコア化と，各演算コアの省力化は避けられないが，演算性能の向上のためにbit 幅が増え続けている SIMD 命令に対応しなければならない．現在のところ，実装されている SIMD 演算機の中で 512-bit が最長だが，SIMD 長のさらなる増加により，ハードウェアの提供する SIMD 長よりも短い単位での演算が増加する可能性が高い．ARM が提案する SVE は，128-bit から最大 2048-bit で 128-bit 単位での SIMD 長をサポートし，SIMD 長が異なるアーキテクチャでも同一のバイナリを実行可能とする．AVX 命令と同じように演算のマスクが可能で，predicate レジスタと Agnostic 演算をサポートするための predicate 生成命令によって，SIMD 長とデータ数に依存しない計算が実現され，非常にスケーラブルと言える．ARM が提案する Vector Length Agnostic プログラミングは，本章で議論してきた SIMD 長が異なるアーキテクチャに対する複数の実装を解決するひとつの手段として，可搬性の向上が期待される [83]．SVE は RISC アーキテクチャ上の命令セットであって，ステンシル計算や疎行列ベクトル積など，真に最適化が必要とされる計算カーネルを効率的に実装できるかという点には議論の余地がある．本研究では，コンパイラの成熟度が不足しているとして SIMD 命令を直接使用した手動のベクトル化により実行効率をさらに向上させたが，SIMD 長や命令セットに依存した最適化はやはり可搬性に乏しいと言わざるを得ない．これらへの依存により，ハードウェア刷新の度に実装と最適化を行わなければならず，多大なコストを費やしてしまっている現状は時間の浪費に近い．SVE のように SIMD 長に依存しない柔軟な命令セットは，コンパイラによる最適化の幅を広げ，より効率的な最適化の提供と，それに伴うコンパイラを活用した可搬性の高いアプリケーション開発が期待される．

Turbo Boost のような動的なクロック調整は，計算性能と供給可能電力のバランスという観点から非常に重要である．特にメニーコア CPU や GPU では，回路規模が縮小されたとはいえ通常の CPU の数倍のコアを搭載するため，TDP 要求はより厳しいものとなる．しかしながら，本研究で実施した大規模全系実行により，同機能がアプリケーションのアルゴリズムとは一切関係無い計算性能のずれを引き起こす可能性を示唆しており，実アプリケーション開発をさらに困難にしてしまうことが危惧される．

第 8 章

結論

8.1　まとめ

本研究ではメニーコア CPU や GPU といった各最先端アーキテクチャおよびそれらを採用するシステムにおいて，電子動力学シミュレーションのコデザインによる最適化と性能評価を行い，各アーキテクチャで得られる性能や記述性，可搬性などの議論から，実アプリケーションのコデザイン手法の模索と，電子動力学への寄与を目的とした．

要求される高い電力効率から，回路規模を縮小し省力化した演算コアを大量に接続，高い並列演算性能により計算量を引き上げるメニーコア CPU と GPU は，エクサスケールシステム実現のためのキーテクノロジーである．理論性能と実性能の乖離を初めとして，高い並列性を活かすための最適化や新たなプログラミングモデルの導入による開発コストの増加など，実アプリケーションへの要求が極めて高く，コデザインが極めて重要である．本研究では実アプリケーションのコデザインについて可搬性，協調開発，最適化の 3 つを定義し，これらを踏まえて電子動力学シミュレーション ARTED を各アーキテクチャとシステムへの最適化，性能評価を行った．

メニーコア CPU では，OpenMP+MPI で開発されている実アプリケーションの構造を維持し，真に最適化が要求かつ将来に渡るコード変更の可能性が低いステンシル計算について，IMCI と AVX-512 命令を用いた 512-bit 長の SIMD 計算の手動実装を行った．この実装により，Intel の最新アーキテクチャである Knights Landing において，HPL 性能比でおよそ 38% の実行効率を獲得した．メニースレッドへの最適化や，ヘテロジニアス環境におけるロードバランスなどの最適化を行い，Ivy-Bridge + Knights Corner クラスタと Knights Landing クラスタの性能を引き出した．世界最大規模の Knights Landing クラスタである Oakforest-PACS では，核となるハミルトニアン計算で 4 PFLOPS，HPL 比で約 30% の実行効率を達成し，高いスケーリング性能を示した．

メニーコア CPU と共存する形で GPU 実装を行い，NVIDIA との共同研究で実施した OpenACC 実装とステンシル計算の CUDA を用いた最適化について述べた．OpenACC で実装することで，GPU アプリケーションの開発コストは非常に小さくなったと考えられるが，やはりメニーコア CPU における SIMD 命令と同じく，ネイティブな CUDA プラットフォームを用いた最適化が要求されている．また，最新の Volta アーキテクチャでは OpenACC と CUDA の性能差が前世代の Pascal に比べ大幅に縮まっており，アーキテクチャの進化によるアプリケーション開発コストの削減に期待がかかる．

各アーキテクチャのステンシル計算性能は HPL 比で 40–50% の性能を達成し，本研究で実施した最適化は各アーキテクチャの性能をよく引き出せていると言える．今日のシステムは理論性能と実性能の乖離が激しく，動的なクロック調整や AVX base clock といった TDP を維持するための対電力効率技術により，各システムが語る理論ピーク演算性能は無意味なカタログスペックと化している．HPL 性能が真に達成可能な最高性能である，という考えは理にかなっており，今後の高性能計算における新たな評価指標として期待される．

実アプリケーションの開発と，アーキテクチャやシステムへの最適化は同時並行かつ持続的に行われ，コデザインは全ての開発者が容易にメンテナンスを行えることが最重要と考えられる．その中で，最適化はメンテナンス性を著しく阻害してはならず，計算の重要性だけではなく実装の修正頻度から最適化水準は決定されるべきである．システムの大規模化に伴い，実アプリケーションの規模はさらに増加する可能性があり，可搬性をどれだけ担保できるかは議論の対象となり，性能とメンテナンス性のトレードオフを十分に検討しなければならないだろう．

最後に，本研究で得られた成果についてまとめる．

- 計算科学との協調により，性能とメンテナンス性をバランスする “実アプリケーションのコデザイン” の実例を提示
- 複数の最先端アーキテクチャのサポートを 1 つのパッケージで実現する希少なオープンソースソフトウェアとして提供
- 各プロセッサとシステムの高い演算性能を享受可能な実アプリケーションを実現
- 世界最大規模のメニーコア CPU クラスタ Oakforest-PACS の全系実行において高い計算性能を示し，アプリケーションによる電子動力学の発展が期待される

8.2 今後の課題

大規模システムにおいて，対電力効率技術によりプロセッサ間の処理性能に差が発生し，意図しない性能のインバランスによるサチュレーションが発生する恐れがある．不均衡を解決するために，動的負荷分散機能の検討が必要と考えられるが，実アプリケーション全体に渡る動的負荷分散の実装は非常に困難である．これらの機能について指摘した問題は，考察に基づく推論であって機能をオフにした状態での性能評価は実施できておらず，追加の実験が必要である．

大規模メニーコア CPU クラスタにおいて高い演算性能を獲得したが，GPU では数ノードの小規模クラスタの評価にとどまる．大規模 GPU クラスタにおいても，メニーコア CPU のように解決が困難な問題が発生しうる可能性が大いにあり，東京工業大学の TSUBAME 3.0 や産業技術総合研究所の ABCI システムなど，大規模 GPU システムでの性能評価が必要である．

現在は，ARTED をベースとした光科学アプリケーション SALMON (Scalable Ab-initio Light-Matter simulator for Optics and Nanoscience) の開発がオープンソースソフトウェアとして進められており，本研究の成果はほぼ全て SALMON に取り込まれている [84, 85]．本研究の延長として，SALMON でのコデザインの継続による電子動力学への貢献と，次世代スーパーコンピュータの活用を目指す．

参考文献

[1] TOP500.
https://www.top500.org/

[2] Green500.
https://www.top500.org/green500/

[3] S. A. Sato and K. Yabana. Maxwell + TDDFT multi-scale simulation for laser-matter interactions. *J. Adv. Simulat. Sci. Eng.*, Vol. 1, No. 1, pp. 98–110, 2014.

[4] E. W. Draeger, X. Andrade, and J. A. Gunnels amd *et al.* Massively parallel first-principles simulation of electron dynamics in materials. *Journal of Parallel and Distributed Computing*, Vol. 106, pp. 205 – 214, 2017.

[5] M. Noda, K. Ishimura, K. Nobusada, and *et al.* Massively-parallel electron dynamics calculations in real-time and real-space: Toward applications to nanostructures of more than ten-nanometers in size. *Journal of Computational Physics*, Vol. 265, No. 14, pp. 145–155, 2014.

[6] SUMMIT: Oak Ridge National Laboratory's next High Performance Supercomputer.
https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

[7] 理化学研究所計算科学研究センター. フラッグシップ 2020 プロジェクト.
http://www.r-ccs.riken.jp/fs2020p/

[8] Haohuan Fu, Conghui He, Bingwei Chen, and *et al.* 18.9Pflops Nonlinear Earthquake Simulation on Sunway TaihuLight: Enabling Depiction of 18-Hz and 8-meter Scenarios. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pp. 2:1–2:12, New York, NY, USA, 2017. ACM.

[9] C. Yang, W. Xue, H. Fu, and *et al.* 10M–Core Scalable Fully-Implicit Solver for Nonhydrostatic Atmospheric Dynamics. In *Proceedings of SC16*, 2016.

[10] Intel Xeon Phi X100 Family Coprocessor - the Architecture.
https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

[11] Intel Math Kernel Library.
https://software.intel.com/en-us/mkl

[12] A. Sodani. Knights landing (KNL): 2nd Generation Intel Xeon Phi processor. In *Proceedings of Hot Chips 27 Symposium*, 2015.

[13] C. Rosales, J. Cazes, K. Milfeld, and *et al.* A Comparative Study of Application Performance and Scalability on the Intel Knights Landing Processor. In *ISC '16 Proceedings*, pp. 307–318, 2016.

[14] NVIDIA Volta Architecture.
`https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/`

[15] NVIDIA CUDA.
`https://developer.nvidia.com/cuda-toolkit`

[16] HPCG.
`http://www.hpcg-benchmark.org/`

[17] D. Sugimoto, Y. Chikada, J. Makino, and *et al.* A Special-Purpose Computer for Gravitational Many-Body Problems. *Nature*, Vol. 345, No. 6270, pp. 33–35, 1990.

[18] GRAPE: a special-purpose computer for N-body problems. J. Makino and T. Ito and T. Ebisuzaki and D. Sugimoto. In *Proceedings of the International Conference on Application Specific Array Processors*, 1990.

[19] J. Makino, M. Taiji, T. Ebisuzaki, and D. Sugimoto. GRAPE-4: a one-Tflops special-purpose computer for astrophysical N-body problem. In *Proceedings of Supercomputing '94*, 1994.

[20] J. Makino and M. Taiji. Astrophysical N-body simulations on GRAPE-4 special-purpose computer. In *Proceeding of Supercomputing '95*, 1995.

[21] T. Fukushige and J. Makino. N-body simulation of galaxy formation on GRAPE-4 special-purpose computer. In *Proceeding of Supercomputing '96*, 1996.

[22] J. Makino and H. Daisaka. GRAPE-8 – An Accelerator for Gravitational N-body Simulation with 20.5Gflops/W Performance. In *Proceedings of SC12*, 2012.

[23] I. Ohmura, G. Morimoto, Y. Ohno, and *et al.* MDGRAPE-4: a special-purpose computer system for molecular dynamics simulations. *Philos Trans A Math Phys Eng Sci.*, Vol. 372, No. 2021, 2014.

[24] 塙 敏博, 児玉 祐悦, 朴 泰祐, 佐藤 三久. Tightly Coupled Accelerators アーキテクチャに基づく GPU クラスタの構築と性能予備評価. 情報処理学会論文誌コンピューティングシステム, Vol. 6, No. 4, pp. 14–25, 2013.

[25] 金田 隆大, 鶴田 千晴, 塙 敏博, 天野 英晴. Performance Evaluation of PEACH3: Field-Programmable Gate Array Switch for Tightly Coupled Accelerators. In *Proceeding of Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*, 2017.

[26] T. Kaneda, T. Kuhara, T. Mitsuishi, and *et al.* Parallel processing of Breadth First Search by Tightly Coupled Accelerators. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2015.

[27] S. Lee, J. Kim, and J. S. Vetter. OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing. In *IEEE International Conference of Parallel and Distributed Processing Symposium*, 2016.

[28] H. R. Zohouri, N. Maruyama, A. Smith, and *et al.* Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *Proceedings of SC16*, 2016.

[29] H. Noda, R. Sakai, T. Miyajima, and *et al.* Acceleration of the aggregation process in a Hall-thruster simulation using Intel FPGA SDK for OpenCL. In *Proceedings of International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*, 2017.

[30] Y. Hasegawa, J. -I. Iwata, M. Tsuji, and *et al.* First-principles Calculations of Electron States of a Silicon Nanowire with 100,000 Atoms on the K Computer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pp. 1:1–1:11, New York, NY, USA, 2011. ACM.

[31] S. Matsuoka. FLAGSHIP 2020 Project: Development of "Post-K" and Arm SVE. In *GoingArm for HPC*, 2018.

[32] B. Gerofi, R. Riesen, R. W. Wisniewkie, and Y. Ishikawa. Toward Full Specialization of the HPC Software Stack: Reconciling Application Containers and Lightweight Multi-kernels. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers 2017*, pp. 7:1–7:8, 2017.

[33] XcalableMP website.
`http://xcalablemp.org/`

[34] M. Nakao, H. Murai, H. Iwashita, and *et. al.* Implementing Lattice QCD Application with XcalableACC Language on Accelerated Cluster. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 429–438, Sept 2017.

[35] Exascale Computing Project.
`https://www.exascaleproject.org/`

[36] S. Seo, A. Amer, P. Balaji, and *et al.* Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 29, No. 3, pp. 512–526, 2018.

[37] Y. Zheng, A. Kamil, M. B. Driscoll, and *et al.* UPC++: A PGAS Extension for C++. *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 1105–1114, 2014.

[38] Flang.
`https://github.com/flang-compiler/flang`

[39] M. Schultze, K. Ramasesha, C. Pemmaraju, and *et al.* Attosecond band-gap dynamics in Silicon. *Science 12 Dec 2014*, Vol. 346, No. 6215, pp. 1348–1352, 2014.

[40] M. Lucchini, S. A. Sato, A. Ludwig, and *et al.* Attosecond dynamical Franz-Keldysh effect in polycrystalline diamond. *Science 26 Aug 2016*, Vol. 353, No. 6302, pp. 916–919, 2016.

[41] M. Malinauskas et.al. Ultrafast laser processing of materials: from science to industry. *Light: Science and Applications*, Vol. 5, p. e16133, 2016.

[42] Y. Hasegawa, J. Iwata, M. Tsuji, and *et al.* First-principles Calculations of Electron States of a Silicon Nanowire with 100,000 Atoms on the K Computer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. ACM,

2011.

[43] N. Troullier and J. L. Martins. Efficient pseudopotentials for plane-wave calculations. *Physical Review B*, Vol. 43, , 1991.

[44] OCTOPUS.
`http://octopus-code.org`

[45] X. Andrade and *et al.* Time-dependent density-functional theory in massively parallel computer architectures: the OCTOPUS project. *Journal of Physics: Condensed Matter*, Vol. 24, p. 233202, 2012.

[46] X. Andrade, J. Alberdi-Rodriguez, D. A. Strubbe, and *et al.* Time-dependent density-functional theory in massively parallel computer architectures: the octopus project. *Journal of Physics: Condensed Matter*, Vol. 24, No. 23, 2012.

[47] T. Barnes, B. Cook, J. Deslippe, and *et al.* Evaluating and optimizing the NERSC workload on Knights Landing. In *Proceedings of the 7th International Workshop on PMBS '16*, pp. 43–53, 2016.

[48] B. Joó, D. D. Kalamkar, T. Kurth, K. Vaidyanathan, and A. Walden. Optimizing Wilson-Dirac Operator and Linear Solvers for Intel KNL. *High Performance Computing: ISC High Performance 2016 International Workshops, Revised Selected Papers*, pp. 415–427, 2016.

[49] C. Yount and A. Duran. Effective use of large high-bandwidth memory caches in HPC stencil computation via temporal wave-front tiling. In *Proceedings of the 7th International Workshop on PMBS'16*, pp. 65–75, 2016.

[50] A. Nguyen, N. Satish, J. Chhugani, and *et al.* 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *Proceedings of SC10*, 2010.

[51] J. D. Garvey and T. S. Abdelrahman. Automatic Performance Tuning of Stencil Computations on GPUs. In *44th International Conference on Parallel Processing (ICPP)*, 2015.

[52] Y. Zhang and F. Mueller. Autogeneration and Autotuning of 3D Stencil Codes on Homogeneous and Heterogeneous GPU Clusters. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 24, pp. 417–427, 2013.

[53] T. Shimokawabe, T. Endo, N. Onodera, and T. Aoki. A Stencil Framework to Realize Large-Scale Computations Beyond Device Memory Capacity on GPU Supercomputers. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017.

[54] 筑波大学計算科学研究センター.
`http://www.ccs.tsukuba.ac.jp/`

[55] 小田嶋 哲哉, 塙 敏博, 児玉 祐悦, 朴 泰祐, 村井 均, 中尾 昌広, 佐藤 三久. HA-PACS/TCA における TCA および InfiniBand ハイブリッド通信. 情報処理学会研究報告, Vol. 2014-HPC-147, No. 32, 2014.

[56] 最先端共同 HPC 基盤施設.
`http://jcahpc.jp/`

[57] M. S. Birrittella, M. Debbage, R. Huggahalli, and *et al.* Enabling Scalable High-Performance

Systems with the Intel Omni-Path Architecture. In *in IEEE Micro*, Vol. 36, pp. 38–47, 2016.

[58] 塙 敏博, 中島 研吾, 大島 聡史, 星野 哲也, 伊田明弘. パイプライン型共役勾配法の性能評価. 情報処理学会研究報告, Vol. 2016-HPC-157, No. 6, 2016.

[59] T. Maruyama. SPARC64(TM) VIIIfx: Fujitsu's New Generation Octo Core Processor for PETA Scale Computing. In *HotChips 21*, 2009.

[60] 松田 元彦, 丸山 直也, 滝沢 真一郎. Xeon Phi (Knights Corner) の性能特性とステンシル計算の評価. 情報処理学会研究報告, Vol. 2014-HPC-143, No. 32, 2014.

[61] 伊奈 拓也, 朝比 祐一, 井戸村 泰宏. テラフロップス級メニーコアアーキテクチャにおけるステンシル計算の最適化手法の開発. 情報処理学会研究報告, Vol. 2015-HPC-152, No. 10, 2015.

[62] R. Krishnaiyer, E. Kultursay, P. Chawla, S. Preis, A. Zvezdin, and H. Saito. Compiler-Based Data Prefetching and Streaming Non-temporal Store Generation for the Intel(R) Xeon Phi(TM) Coprocessor. In *IPDPSW 2013*, pp. 1575–1586, 2013.

[63] Intel VTune Amplifier XE.
`https://software.intel.com/en-us/intel-vtune-amplifier-xe`

[64] D. Takahashi. Implementation and Evaluation of Parallel FFT Using SIMD Instructions on Multi-core Processors. In *IWIA 2007*, pp. 53–59, 2007.

[65] J. Hofmann, J. Treibig, G. Hager, and G. Wellein. Comparing the Performance of Different x86 SIMD Instruction Sets for a Medical Imaging Application on Modern Multi- and Manycore Chips. In *WPMVP '14*, pp. 57–64, 2014.

[66] C. Andreolli. Eight Optimizations for 3-Dimensional Finite Difference (3DFD) Code with an Isotropic (ISO).
`https://software.intel.com/en-us/articles/eight-optimizations-for-3-dimensional-finite-difference-3dfd-code-with-an-isotropic-iso`

[67] Intel Intrinsics Guide.
`https://software.intel.com/sites/landingpage/IntrinsicsGuide/`

[68] Intel MPI Library.
`https://software.intel.com/en-us/intel-mpi-library`

[69] R. Rabenseifner. Optimization of Collective Reduction Operations. In *Computational Science - ICCS 2004*, 2004.

[70] G. E. Blelloch. Prefix Sums and Their Applications. *School of Computer Science, Carnegie Mellon University*, Vol. CMU-CS-90-190, , 1990.

[71] P. J. Martin, L. F. Ayuso, R. Torres, and A. Gavilanes. Algorithmic strategies for optimizing the parallel reduction primitive in CUDA. In *2012 International Conference on High Performance Computing & Simulation*, pp. 511–519, 2012.

[72] 東京大学物性研究所.
`http://www.issp.u-tokyo.ac.jp/`

[73] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith. GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming

models. *P3̂MA Workshop at ISC High Performance*, 2016.

[74] PGI Compilers & Tools.
`https://www.pgroup.com/`

[75] OpenACC.
`https://www.openacc.org/`

[76] 星野 哲也, 松岡 聡. 圧縮性流体解析プログラムの OpenACC による高速化. 第 153 回 HPC 研究会, Vol. 2016-HPC-153, No. 4, 2016.

[77] NVIDIA CUDA Programming Guide.
`http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

[78] 中島 研吾, 星野 哲也, 成瀬 彰, 塙 敏博, 三木 洋平. 有限要素法における係数行列生成部のマルチコア・メニィコア向け最適化. 情報処理学会研究報告, Vol. 2018-HPC-163, No. 28, pp. 1–8, 2018.

[79] 佐藤 駿一, 高橋 大介. GPU における SELL 形式疎行列ベクトル積の実装と性能評価. 情報処理学会研究報告, Vol. 2018-HPC-164, No. 3, pp. 1–6, 2018.

[80] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. Technical report, arXiv.org, 2018.

[81] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers.
`https://www.cs.virginia.edu/stream/`

[82] A. Tabuchi, Y. Kimura, S. Torii, and *et al.* Design and Preliminary Evaluation of Omni OpenACC Compiler for Massive MIMD Processor PEZY-SC. In *International Workshop on OpenMP, IWOMP2016*, 2016.

[83] 小田嶋 哲哉, 児玉 祐悦, 松田 元彦, 他. ベクトル長を可変とする SVE アーキテクチャの評価. 情報処理学会研究報告, Vol. 2017-HPC-160, No. 11, 2017.

[84] M. Noda, Shunsuke A. Sato, Y. Hirokawa, and *et al.* SALMON: Scalable Ab-initio Light-Matter simulator for Optics and Nanoscience. *arXiv: Computational Physics*, arXiv:1804.01404, 2018.

[85] SALMON (Scalable Ab-initio Light-Matter simulator for Optics and Nanoscience).
`http://salmon-tddft.jp/`

謝辞

筑波大学計算科学研究センター教授 朴 泰祐先生には，本研究の遂行にあたり，厳しくも的確な御指摘と，深い知識と広範な議論により至らぬ著者をご指導いただきました．博士前期課程から約 4 年半に渡り研究活動に従事し多くの成果を挙げられたのは朴先生の熱心な御指導の賜物です．ここに深く感謝申し上げます．

筑波大学計算科学研究センター教授 矢花 一浩先生，高橋 大介先生，筑波大学システム情報系教授 櫻井 鉄也先生，東京大学情報基盤センター准教授 塙 敏博先生には，審査にあたり，本研究の副査をお引き受けいただきました．御多忙の中，至らぬ著者に対し多くの御指摘と深い議論を賜り，本研究の質と価値の向上につながりました．ここに深く感謝申し上げます．

筑波大学計算科学研究センター教授 矢花 一浩先生，マックス・プランク研究所 佐藤 駿丞様，筑波大学計算科学研究センター 植本 光治様には，電子動力学シミュレーションコード ARTED および SALMON の研究開発について，約 4 年間に渡る共同研究を，現在に至るまで実施させていただいております．本研究は，皆様方との共同研究がなければ遂行すら困難でした．ここに深く感謝申し上げます．

東京工科大学コンピュータサイエンス学部教授 生野 壮一郎先生，富士通株式会社 川口 優樹様には，東京工科大学コンピュータサイエンス学部在学当時，右も左もわからぬ著者に高性能計算技術を御指導賜りました．御二方との出逢いがなければ著者は研究者の道を目指すことはなく，著者の約 27 年の人生の中で最も重要で貴重なものとなりました．ここに深く感謝申し上げます．

筑波大学計算科学研究センター 藤田 典久 博士，富士通研究所 田渕 晶大 博士，津金 佳祐 博士には，在学中，至らぬ著者に対し近い視点での多くの御指導，御指摘，議論を賜りました．ここに深く感謝申し上げます．

最後に，筑波大学ハイパフォーマンスコンピューティングシステム研究室の皆様と，約 27 年に渡り著者を支えてくれた家族に心から感謝申し上げます．

付録 A

研究業績

査読付き論文誌

1. 廣川 祐太, 朴 泰祐, 佐藤 駿丞, 矢花 一浩: “電子動力学シミュレーションのステンシル計算最適化とメニーコアプロセッサへの実装”, 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 9, No. 4, 1882-7829, P. 1-14, 2016.

査読付き国際会議論文

1. Yuta Hirokawa, Taisuke Boku, Mitsuharu Uemoto, Shunsuke A. Sato, and Kazuhiro Yabana: “Performance Optimization and Evaluation of Scalable Optoelectronics Application on Large Scale KNL Cluster”, ISC High Performance 2018 (ISC2018), Frankfurt, June 2018.
2. Yuta Hirokawa, Taisuke Boku, Shunsuke A. Sato, and Kazuhiro Yabana: “Performance Evaluation of Large Scale Electron Dynamics Simulation under Many-core Cluster based on Knights Landing”, The 1st International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia2018), Tokyo, Jan. 2018.
3. Yuta Hirokawa, Taisuke Boku, Shunsuke A. Sato, and Kazuhiro Yabana: “Electron Dynamics Simulation with Time-Dependent Density Functional Theory on Large Scale Symmetric Mode Xeon Phi Cluster”, The 17th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC2016), Chicago, May 2016.

査読付き国内会議論文

1. 廣川 祐太, 朴 泰祐, 佐藤 駿丞, 矢花 一浩: “電子動力学シミュレーションのステンシル計算に対するメニーコアプロセッサ向け最適化”, HPCS2016 ハイパフォーマンスコンピューティングと計算科学シンポジウム, 宮城, 2016 年 6 月.

査読付き国際会議ポスター発表

1. Yuta Hirokawa: “Electron Dynamics Simulation with Time-Dependent Density Functional Theory on Large Scale Many-Core Systems“, Supercomputing Conference (SC) 16, ACM Student Research Competition Poster, Salt Lake City, Nov. 2016.
2. Yuta Hirokawa, Taisuke Boku, Shunsuke A. Sato, and Kazuhiro Yabana: “Electron Dynamics Simulation with Time-Dependent Density Functional Theory on Large Scale Knights-Corner Cluster”, HPC in Asia Posters Session in International Supercomputing Conference (ISC) 2016, Frankfurt, Jun. 2016.

査読付き国内会議ポスター発表

1. 廣川 祐太, 朴 泰祐, 佐藤 駿丞, 矢花 一浩: “Xeon Phi クラスタにおける電子動力学シミュレーションの実装と性能評価”, HPCS2015 ハイパフォーマンスコンピューティングと計算科学シンポジウム, P2-4, 東京, 2015 年 5 月.

査読無し国内研究会原稿

1. 廣川 祐太, 朴 泰祐, 植本 光治, 佐藤 駿丞, 矢花 一浩: “電子動力学シミュレーションコードのメニーコアプロセッサと GPU における性能比較”, 第 163 回 HPC 研究会, Vol. 2018-HPC-163 No. 23, 愛媛, 2018 年 3 月.
2. 廣川 祐太, 朴 泰祐, 植本 光治, 佐藤 駿丞, 矢花 一浩: “電子動力学シミュレーション ARTED の KNL システム Oakforest-PACS での全系性能評価”, 第 160 回 HPC 研究会, Vol. 2017-HPC-160 No. 20, 秋田, 2017 年 7 月.
3. 廣川 祐太, 朴 泰祐, 佐藤 駿丞, 矢花 一浩: “電子動力学コード ARTED による Knights Landing プロセッサの性能評価”, 第 157 回 HPC 研究会, Vol. 2016-HPC-157 No. 8, 沖縄, 2016 年 12 月.
4. 廣川 祐太, 朴 泰祐, 佐藤 駿丞, 矢花 一浩: “電子動力学シミュレーションコードの Symmetric モードによる Xeon Phi クラスタへの実装と性能評価”, 第 153 回 HPC 研究会, Vol. 2016-HPC-153 No. 16, 愛媛, 2016 年 3 月.
5. 廣川 祐太, 朴 泰祐, 佐藤 駿丞, 矢花 一浩: “Xeon Phi クラスタにおける Symmetric 並列実行による電子動力学シミュレーションの性能評価”, 第 151 回 HPC 研究会, Vol. 2015-HPC-151 No. 18, 沖縄, 2015 年 9 月.
6. 廣川 祐太, 朴 泰祐, 佐藤 駿丞, 矢花 一浩: “実時間実空間密度汎関数理論による電子動力学シミュレーションの Xeon Phi クラスタ向け最適化”, 第 148 回 HPC 研究会, Vol. 2015-HPC-148 No. 19, 大分, 2015 年 3 月.

査読無し国内研究会ポスター発表

1. 廣川 祐太, 朴 泰祐, 植本 光治, 佐藤 駿丞, 矢花 一浩: “電子動力学シミュレーション ARTED の 25 PFLOPS メニーコアシステム Oakforest-PACS での全系性能評価”, 第 2 回ポスト「京」重点課題 (7) 研究会, 東京, 2017 年 7 月.
2. 廣川 祐太, 朴 泰祐, 佐藤 駿丞, 矢花 一浩: “電子動力学シミュレーションコードの大規模並列計算機への実装と最適化”, 第 1 回ポスト「京」重点課題 (7) 研究会 「次世代の産業を支える新機能デバイス・高性能材料の創成」, 東京, 2016 年 7 月.

招待講演

1. Yuta Hirokawa: “Performance evaluation of scalable optoelectronics application on large-scale Knights Landing cluster”, The 5th Accelerated Data Analytics and Computing Institute (ADAC) Workshop, Tokyo, Feb. 2018.
2. 廣川 祐太: “電子動力学シミュレーション ARTED の Oakforest-PACS での全系性能評価”, 第 5 回 JCAHPC セミナー, 千葉, 2017 年 10 月.
3. Yuta Hirokawa: “Performance optimization of electron dynamics simulation for large-scale many-core systems”, International Workshop on Massively Parallel Programming for Quantum Chemistry and Physics 2017, Hyogo, Jan. 2017.
4. 廣川 祐太: “第一原理電子動力学コードの最適化と性能評価: GPU と Xeon Phi”, GTC Japan 2016, 東京, 2016 年 10 月.

その他発表等

1. Yuta Hirokawa: “Performance Evaluation of Large Scale Electron Dynamics Simulation under Many-core Cluster based on Knights Landing”, JKHPC - Japan-Korea HPC Winter School 2018 mini-workshop, Tsukuba, Feb. 2018.
2. Yuta Hirokawa: “Performance Optimization of Electron Dynamics Simulation for Large-Scale Many-Core Systems”, KJHPC - Korea-Japan HPC Winter School 2017 mini-workshop, Seoul, Feb. 2017.