演算加速機構を持つ並列システム向け
PGAS 言語コンパイラの研究

2018 年 3 月

田渕 晶大

演算加速機構を持つ並列システム向け PGAS 言語コンパイラの研究

田渕 晶大

システム情報工学研究科
筑波大学

2018 年 3 月

概要

ハイパフォーマンスコンピューティングの分野では，性能および消費電力あたりの性能を向上させるために演算加速機構 (アクセラレータ) を搭載した並列システムが増加している．本研究ではアクセラレータを搭載した並列システム向けのプログラムの記述を簡易にして生産性を向上すると共に高性能を達成することを目的とする．アクセラレータは多数の演算器と高バンド幅のメモリにより高性能と省電力を実現しているが，CUDA や OpenCL を用いてアクセラレータ向け並列カーネルコードの記述やホストと独立したメモリの利用が必要でプログラミングが複雑である．そこで簡易な記述として指示文によるプログラミングモデルが提案されており，特に OpenACC は最初の標準仕様である．他方で，並列システムの分散メモリ環境におけるプログラミングには従来 MPI が用いられており，ユーザがデータや処理の分散と MPI による通信・同期をすべて記述しなければならないため，プログラミングが複雑である．そこでプログラミングを簡易にするため分散メモリ環境で仮想的な大域名前空間を提供する Partitioned Global Address Space (PGAS) モデルが提案されている．特に XcalableMP(XMP) では指示文や coarray による簡易な記述が可能である．アクセラレータを搭載した並列システムにおいては，従来のプログラミングではオフロードと分散並列をすべて直接記述するため非常に複雑なプログラミングが必要となる．そのうえ主な演算がアクセラレータ上で実行されるためアクセラレータメモリ上のデータ通信が必要であり，それを高速に行うためのハードウェア機能を利用することが重要となる．

研究目的を達成するため，筑波大と理研の共同で OpenACC と XMP を統合した PGAS 言語 XcalableACC (XACC) を提案した．XACC では新たにアクセラレータ間通信の記述に対応することで XMP と OpenACC を単に組み合わる場合よりも高速な通信を可能としており，本研究では特に coarray による片側通信を提案した．XACC の生産性や性能を評価するため，NVIDIA GPU と PEZY-SC クラスタにむけてコンパイラを設計・実装した．XACC で利用する OpenACC に関して，2012 年当初はオープンソースのコンパイラ研究基盤がなく，商用コンパイラも GPU 等の一般的なアクセラレータのみの対応でかつ最適化が十分でなかったうえ，PEZY-SC には対応していなかった．そこで性能を向上させるため NVIDIA GPU 向けに，また対応コンパイラがなかったため PEZY-SC 向けに OpenACC コンパイラを設計・実装し最適化を行った．ベンチマークによる評価において NVIDIA GPU では既存の OpenACC コンパイラの最大 2.8 倍の性能を，PEZY-SC では直接記述する場合の 90% 以上の性能を達成した．また XACC のコンパイラを設計・実装し，実装した OpenACC コンパイラをバックエンドに用いてベンチマークやアプリケーションで評価を行なった．グローバルビューモデルでは逐次コードに指示文を加える形で簡易に記述でき，ローカルビューモデルは coarray による配列代入文で簡易に柔軟な通信を記述できた．NVIDIA GPU クラスタでは，グリッド分散と袖交換通信を行う Himeno benchmark はグローバルビュー

モデルによる記述で MPI+OpenACC による記述の 97% 以上の性能を達成した．複雑な通信を必要とする NPB-CG は，ローカルビューモデルにより MPI+OpenACC の 97% 以上の性能が得られた．PEZY-SC クラスタでは，Himeno benchmark はグローバルビューにより MPI+PZCL の 96% 以上の性能を達成した．NVIDIA GPU クラスタにおいて，MPI+CUDA 実装と比べて XACC グローバルビューモデルによる記述で流体力学ミニアプリケーション CloverLeaf は 87% 以上，格子 QCD ミニアプリケーションは 95% 以上の性能が得られ，簡易な記述でありながら従来の記述に近い性能が得られた．

目次

概要 i

第 1 章 序論 1
1.1 研究背景 1
1.2 研究目的 3
1.3 構成 4

第 2 章 研究の背景 5
2.1 NVIDIA GPU 5
2.2 CUDA 6
2.2.1 ホストコード 6
2.2.2 デバイスコード 7
2.3 PEZY-SC 8
2.4 PZCL 9
2.4.1 ホストコード 9
2.4.2 デバイスコード 10
2.5 OpenACC 11
2.5.1 実行モデル 11
2.5.2 メモリモデル 13
2.5.3 プログラム例 13
2.6 XcalableMP 13
2.6.1 グローバルビューモデル 14
2.6.2 ローカルビューモデル 15

第 3 章 演算加速機構を持つ並列クラスタ向け PGAS 言語 XcalableACC の提案 19
3.1 概要 19
3.2 グローバルビューモデル 19
3.3 ローカルビューモデル 21
3.4 関連研究 22

第 4 章 OpenACC コンパイラの設計・実装 25
4.1 NVIDIA GPU 向け OpenACC コンパイラ 25
4.1.1 設計 25
4.1.2 data 構文の実装 26
4.1.3 parallel 構文の実装 27
4.1.4 loop 構文の実装 28
4.1.5 評価 30
4.2 PEZY-SC 向け OpenACC コンパイラ 37
4.2.1 NVIDIA GPU 向け実装からの変更点 37
4.2.2 最適化 38
4.2.3 評価 39
4.3 関連研究 44

第 5 章 XcalableACC コンパイラの設計・実装 47
5.1 設計 47
5.2 実装 48
5.2.1 reflect 指示文 48
5.2.2 reduction 指示文 48
5.2.3 gmove 指示文 49
5.2.4 coarray 機能 50
5.3 NVIDIA GPU クラスタにおけるグローバルビューモデルの性能評価 53
5.3.1 Himeno Benchmark 53
5.3.2 NAS Parallel Benchmarks CG 57
5.4 NVIDIA GPU クラスタにおけるローカルビューモデルの性能評価 62
5.4.1 Himeno benchmark 65
5.4.2 NAS Parallel Benchmarks CG 68
5.5 PEZY-SC クラスタにおけるグローバルビューモデルの性能評価 69
5.6 生産性評価 73
5.6.1 グローバルビューモデル 73
5.6.2 ローカルビューモデル 75
5.7 考察 76
5.8 関連研究 77

第 6 章 アプリケーションを用いた XcalableACC の性能と生産性の評価 79
6.1 流体力学ミニアプリケーション CloverLeaf 79
6.1.1 XACC による実装 80
6.1.2 評価 84
6.1.3 考察 93

6.2 格子 QCD ミニアプリケーション 94
6.2.1 性能評価 95
6.3 関連研究 96

第 7 章 結論 99
7.1 まとめ 99
7.2 今後の課題 100

謝辞 103

参考文献 105

付録 A 公表論文リスト 111

図目次

2.1 Kepler アーキテクチャ (NVIDIA Kepler GK110 Architecture Whitepaper[1] より引用) . . 6
2.2 PEZY-SC プロセッサの構成 8
2.3 PEZY-SC プロセッサのスレッドの動作モデル 9
2.4 PEZY-SC プロセッサにおける明示的スレッド切り替えの有無による違い 12
2.5 XMP グローバルビューモデルによるプログラム例 14
2.6 shadow と reflect 指示文の例 15
2.7 gmove 指示文の例 16
2.8 XMP/C における coarray の文法 16
2.9 post/wait 指示文の構文 17
2.10 XMP ローカルビューモデルによるプログラム例 18

3.1 デバイス間通信の記述方法 20
3.2 XACC グローバルビューモデルのコード例 21
3.3 XACC ローカルビューモデルのコード例 22

4.1 Omni OpenACC compiler のコンパイルの流れ 26
4.2 OpenACC の並列性の CUDA へのマッピング 27
4.3 shuffle 命令を用いたバタフライリダクション 29
4.4 2^{20} 要素のリダクションの実行時間 32
4.5 最適化による 2^{20} 要素のリダクション時間の変化 33
4.6 CG の性能 34
4.7 EP の性能 35
4.8 IS の性能 36
4.9 FT の性能 37
4.10 MG の性能 38
4.11 yield, sync, flush 指示文の構文 39
4.12 PEZY-SC における N-body ベンチマークの性能 40
4.13 PEZY-SC における NPB-CG ベンチマークの性能 41
4.14 PEZY-SC における NPB-CG ベンチマークの性能 (OpenACC に最適化を追加) 42
4.15 K20X と PEZY-SC における OpenACC 版 N-body ベンチマークの性能 43

4.16 K20X と PEZY-SC における OpenACC 版 NPB-CG ベンチマークの性能 43

5.1 Omni XACC compiler におけるコンパイル処理の流れ 47
5.2 NPB CG にて用いられる gmove (4 ノード) 49
5.3 NPB-CG の gmove の改善前 (8 ノード) 50
5.4 NPB-CG の gmove の改善後 (8 ノード) 50
5.5 Himeno Benchmark の性能 55
5.6 Himeno Benchmark の 1 反復の実行時間の内訳 56
5.7 Himeno Benchmark の袖通信 1 回の実行時間 58
5.8 NPB CG の性能 60
5.9 conj_grad 関数の 1 回の実行時間の内訳 60
5.10 Allreduce のレイテンシ (2 プロセス) 61
5.11 Allreduce のレイテンシ (4 プロセス) 62
5.12 Allreduce のレイテンシ (8 プロセス) 63
5.13 MPI_Send/Recv() と coarray put/get の Ping-pong ベンチマーク性能 64
5.14 Himeno benchmark の性能と時間内訳 (size M) 66
5.15 Himeno benchmark の性能と時間内訳 (size L) 67
5.16 NPB CG (Class C) の性能と時間内訳 70
5.17 NPB CG (Class D) の性能と時間内訳 71
5.18 PEZY-SC クラスタにおける Himeno benchmark (Size L) の性能と時間内訳 72
5.19 PEZY-SC クラスタにおける Himeno benchmark (Size L) の通信時間内訳 73
5.20 nonblock, wait_nonblock 指示文のシンタックス 77

6.1 CloverLeaf におけるスタッガード格子配置 79
6.2 CloverLeaf の XACC 実装におけるセル中心の配列の分散 81
6.3 CloverLeaf の XACC 実装におけるセル頂点の配列の分散 82
6.4 960^2 セルのストロングスケーリングにおける実行時間 85
6.5 3840^2 セルのストロングスケーリングにおける実行時間 86
6.6 ストロングスケーリングにおける各ノードのタイムステップ当たりの袖交換通信量 . . . 88
6.7 960^2 セルのウィークスケーリングにおける実行時間 90
6.8 3840^2 セルのウィークスケーリングにおける実行時間 91
6.9 ウィークスケーリングにおける各ノードのタイムステップ当たりの袖交換通信量 92
6.10 if 節の構文 93
6.11 edged-block 分散の構文 94
6.12 reflect 指示文の offset 節の構文 94
6.13 格子 QCD の CG 法 1 反復当たりの実行時間 95
6.14 格子 QCD の CG 法 1 反復当たりの実行時間の内訳 96
6.15 格子 QCD の CG 法 1 反復当たりの袖交換時間 97

表目次

4.1 NVIDIA GPU 向け OpenACC コンパイラの評価環境 31
4.2 青睡蓮のノード構成とソフトウェア 40
4.3 青睡蓮のノード構成とソフトウェア 41
4.4 HA-PACS/TCA のノード構成とソフトウェア 42
4.5 N-Body と NPB-CG の SLOC. (内数は指示文の行数) 44

5.1 HA-PACS/TCA におけるグローバルビューモデルの性能評価に用いたソフトウェアと環境変数 53
5.2 Himeno Benchmark のカーネルによる占有率や性能の違い 56
5.3 Himeno Benchmark の袖通信の要素数 57
5.4 CG の Class D における配列 w の長さ 62
5.5 HA-PACS/TCA におけるローカルビューモデルの性能評価に用いたソフトウェア 63
5.6 青睡蓮におけるグローバルビューモデルの性能評価に用いたソフトウェア 69
5.7 Himeno Benchmark のコード行数 (内数は通信関連の行数) 74
5.8 Himeno Benchmark の逐次コードへの変更行数 (指示文は除く) 74
5.9 NPB CG のコード行数 (内数は通信関連の行数) 75
5.10 NPB CG の逐次コードへの変更行数 (指示文は除く) 75
5.11 Himeno benchmark と NPB CG の SLOC 76

6.1 HA-PACS/TCA における CloverLeaf の性能評価に用いたソフトウェア 84
6.2 CloverLeaf の各コードの SLOC と DSLOC 89
6.3 格子 QCD の評価に使用したソフトウェア 95

ソースコード目次

2.1 CUDA によるホストコード例 . . . 7
2.2 CUDA によるデバイスコード例 . . . 7
2.3 PZCL カーネルの例 . . . 10
2.4 OpenACC のコード例 . . . 13
4.1 data 構文の例 . . . 26
4.2 data 構文の変換例 . . . 26
4.3 parallel 構文の例 . . . 28
4.4 parallel 構文の変換例 . . . 28
4.5 loop 構文の例 . . . 30
4.6 loop 構文の変換例 . . . 30
5.1 アクセラレータ上の coarray 宣言 . . . 51
5.2 アクセラレータ上の coarray 宣言を変換したコード . . . 51
5.3 coarray によるアクセラレータメモリ間の put 通信 . . . 52
5.4 coarray によるアクセラレータメモリ間の put 通信を変換したコード . . . 52
5.5 XACC で記述した Himeno Benchmark . . . 54
5.6 XACC による NPB CG のコード . . . 58
5.7 XACC ローカルビューモデルにおける Himeno benchmark の袖交換 (ik 平面) . . . 65
5.8 NPB CG の XACC ローカルビューモデルによる通信 . . . 68
6.1 CloverLeaf の XACC 実装における宣言・確保 . . . 80
6.2 CloverLeaf の XACC 実装における計算 . . . 82
6.3 ソースコード 6.2 の XACC コードを Omni XACC compiler が変換したコード . . . 84

第 1 章

序論

1.1　研究背景

ハイパフォーマンスコンピューティング (HPC) の分野において，スーパーコンピュータの性能は年々向上しており，すでにペタスケールが達成され，次のマイルストーンであるエクサスケールの達成に向けて世界各国で研究・開発が進められている．その際に性能の向上に加えて，スーパーコンピュータに対する現実的な電力供給量には限りがあるため消費電力あたりの性能も向上させる必要がある．現在，この目標を達成するために演算加速機構 (アクセラレータ) の活用が盛んに行われている．CPU は少数の性能の高いコアを持つ一方で，アクセラレータは数十から数千個の性能の低いコアと高バンド幅のメモリを持つことで，より高い性能と省電力を実現している．スーパーコンピュータの性能ランキングである Top500[2] をみると 2017 年 11 月では，第 2 位の Tianhe-2 はカード型の Intel Xeon Phi，第 3 位の Piz Daint と第 5 位の Titan は NVIDIA Tesla，第 4 位の Gyoukou は PEZY-SC をアクセラレータとして搭載している．Top500 のシステムのうち約 20% の 102 システムにおいてアクセラレータが用いられており，NVIDIA GPU を搭載したシステムがそのうちの 85% と高いシェアを占める．またスーパーコンピュータの省電力ランキングである Green500 においては，上位 10 システムのうち 4 システムが PEZY-SC，残り 6 システムが NVIDIA Tesla を搭載している．特に上位 3 システムは全て PEZY-SC を搭載しており，高い電力性能比を達成している．2018 年に稼働予定の産業技術総合研究所の人工知能処理向け大規模・省電力クラウド基盤 (AI Bridging Cloud Infrastructure, ABCI) やオークリッジ国立研究所の Summit もアクセラレータとして NVIDIA Tesla を搭載すると発表されており，今後も同様のシステムは増加する見込みである．

アクセラレータのプログラミング方法を見てみると，NVIDIA GPU に対しては同社が提供する専用の開発環境 Compute Unified Device Architecture(CUDA)[3] が広く普及している．NVIDIA GPU のアーキテクチャに合わせた記述と成熟したコンパイラにより，高性能なプログラム開発が可能である．またアクセラレータを含むヘテロジニアス環境向けの並列計算フレームワークである Open Computing Language(OpenCL)[4] もその汎用性から多様なアクセラレータで用いられている．これら 2 つのプログラミング方法では，どちらもアクセラレータを用いて計算をするのに必要なアクセラレータ上のメモリの確保やデータ転送などの事前・事後処理を API 関数で記述し，実際にアクセラレータ上の多数のコアで行う並列処理を拡張された C 言語で記述する必要があり，プログラミングが煩雑である．加えて，CUDA の記述は NVIDIA GPU でしか動作せず，また OpenCL も各アクセラレータに適切な記述が異なるため実

際の可搬性は高くない．近年でも PEZY-SC が新たなアクセラレータとして登場しており今後も様々なアクセラレータが登場することを考えると，アクセラレータごとに大幅にプログラムを書き換えなければならないこれらの記述は可搬性の低さの面で問題がある．そこで，これらの問題の解決のために指示文ベースのプログラミングモデルがいくつか提案されている．研究では OpenMPC や hiCUDA，商用コンパイラでは CAPS の HMPP，PGI の PGI Accelerator Programming Model などがある．逐次コードに指示文を追加する形でアクセラレータへの処理のオフロードを記述するため，逐次コードをベースに簡易に記述できるうえ，高いレベルでの記述であるため特定のアクセラレータへの依存が低く可搬性が高いという特徴がある．その中でも 2011 年に CAPS・Cray・NVIDIA・PGI が策定した OpenACC[5] は，アクセラレータ向け指示文としては初の標準仕様であり，CAPS・Cray・PGI コンパイラがこれをサポートしたことによって HPC 分野において広く認知された．しかしながら，OpenACC の登場当初はオープンソースのコンパイラ研究基盤がなく，商用コンパイラでも GPU のような一般的なアクセラレータのみの対応で最適化も不十分であった．なお，OpenACC だけでなく 2013 年には OpenMP[6] でも仕様 4.0 でアクセラレータプログラミングが導入されたため，指示文によるアクセラレータプログラミングは今後より一般的になると考えられる．

スーパーコンピュータやクラスタのような複数ノードを用いる分散メモリ環境におけるプログラミングでは，従来 Message Passing Interface (MPI) が通信や同期に用いられている．逐次コードと比較して，データや処理の分散と通信が必要であるため非常に多くの処理を記述しなければならない．そこで MPI に代わるプログラミングモデルとして Partitioned Global Address Space (PGAS) モデルが提案されている．PGAS モデルは分散メモリ環境において仮想的な大域名前空間をユーザに提供するモデルで，データに統一的にアクセスできるため記述が容易である．プログラミング言語で PGAS モデルを採用している代表例として既存言語の拡張である UPC, X10, Coarray Fortran, XcalableMP[7]，独自言語の Chapel がある．特に XMP では逐次コードへ指示文を追加するモデルと，coarray を用いて通信を記述するモデルの 2 種類の記述が可能である．またライブラリによる実装としては OpenSHMEM が代表的である．

演算加速機構を持つ並列システム向けのアプリケーション開発では，分散メモリプログラミングに加えてアクセラレータプログラミングも行う必要がある．従来の CUDA もしくは OpenCL と MPI の組み合わせによるプログラミング方法では複雑さがさらに増し，開発期間の増加やメンテナンス性の低下といった課題がある．加えて，主な演算をアクセラレータで実行するので通信するデータがアクセラレータ上にあることが多く，アクセラレータ間の通信が重要となる．例えば NVIDIA GPU では GPUDirect RDMA(GDR)[8] により PCI Express 経由で直接 NIC にデータを転送することで低レイテンシの通信が可能である．そのようなアクセラレータ独自の高速な通信機能の利用が大規模な並列プログラムにおけるスケーラビリティの点で重要となるが，特別な通信ライブラリの使用が必要な場合がありプログラミングはさらに煩雑になる．より簡易で可搬性の高い記述として，分散メモリプログラミングに PGAS を，アクセラレータプログラミングに指示文を用いることが考えられる．しかしながら，先に挙げた XMP と OpenACC により記述を行う場合にはアクセラレータ間の通信を記述する方法がなく，ハードウェアの機能を活かせず性能が低下する可能性がある．

1.2　研究目的

本研究の目的は演算加速機構を持つ並列システムを活用するために，プログラミングを容易にして生産性を向上させるとともに高性能を達成できるプログラミング環境を提案することである．この目的の達成のため，筑波大学と理化学研究所の共同で PGAS 言語 XcalableACC (XACC) [9] を提案する．XACC は XMP と OpenACC の統合であり，XMP と OpenACC によって高い生産性と可搬性を実現すると同時に，新たにアクセラレータ間の通信の記述を提供することで XMP と OpenACC を単に組み合わる場合よりも高速な通信を可能とする．XACC の生産性や性能を評価するため，対象システムとして最も普及している NVIDIA GPU クラスタと高電力効率の PEZY-SC クラスタを用いる．XACC で利用する OpenACC のコンパイラに関して，NVIDIA GPU において高い性能を達成するため OpenACC コンパイラの設計・実装および最適化を行い，ベンチマークを用いて既存の OpenACC コンパイラと性能を比較する．また PEZY-SC 向けには OpenACC コンパイラが存在しなかったため，コンパイラを設計・実装および最適化を行い，ベンチマークを用いて PEZY-SC 向けに直接記述する場合と性能・生産性を比較する．そして XACC のコンパイラを設計・実装し，ベンチマークやアプリケーションを用いて従来の記述と比較して性能・生産性を評価する．本研究の貢献を以下に示す．

- オープンソースの source-to-source OpenACC コンパイラを設計・実装した．NVIDIA GPU 向け実装ではループ並列化以外は元のコードを維持して変換することで CUDA コンパイラによる最適化が十分に働き，さらにメモリプールの利用や Kepler アーキテクチャ向けにリードオンリーキャッシュやリダクションに shuffle や atomic 命令を利用することで，商用コンパイラの最大 2.8 倍の性能を達成した．PEZY-SC 向け実装では kernels 指示文を単一カーネルに変換することでカーネル起動コストを削減でき，またスレッド切り替えや同期やフラッシュ用の拡張指示文を追加することでユーザによるチューニングを可能とし，PZCL による記述と同等の性能を達成した．加えて PEZY-SC 向けの初の OpenACC コンパイラであり，既存 OpenACC プログラムの利用や PZCL よりも簡易な記述を可能にしたことにより生産性の向上に寄与した．
- XACC ローカルビューモデルにおけるアクセラレータメモリ上の coarray の宣言とそれに対する通信の記述方法，および同期削減の指示文を提案した．そして XACC グローバルビューモデルを NVIDIA GPU と PEZY-SC クラスタ向けに，ローカルビューモデルを NVIDIA GPU クラスタ向けに実装した．Himeno benchmark と NAS Parallel Benchmarks (NPB) CG による評価から，グローバルビューモデルは逐次コードに指示文の追加，ローカルビューモデルは coarray の配列代入文形式の通信により簡易に記述可能であり，MPI+OpenACC と同等の性能が得られることを示した．また，Himeno benchmark においては NVIDIA GPU と PEZY-SC クラスタ向けに XACC でほぼ同様に記述できることを示した．
- 流体力学と格子 QCD のミニアプリケーションを用いた評価により XACC グローバルビューモデルが実アプリケーションにおいても逐次コードをベースに簡易に記述でき，NVIDIA GPU クラスタにおいて従来の MPI+CUDA や MPI+OpenACC による記述と近い性能を得られることを示した．

1.3　構成

本稿の続きは次に示す構成となっている．2 章では研究の背景として対象とするアクセラレータのアーキテクチャや開発環境，提案する PGAS 言語 XACC のベースとなる OpenACC および XMP について紹介する．3 章では演算加速機構を持つ並列システム向けに提案する PGAS 言語 XACC の説明をする．4 章では XACC のベースとして用いる OpenACC のコンパイラの設計と実装を解説し，ベンチマークを用いて評価を行う．5 章では XACC のコンパイラの設計と実装を解説し，ベンチマークによる性能・生産性の評価を行う．6 章ではアプリケーションを用いて XACC の生産性や性能の評価を行う．7 章では結論と今後の課題を述べる．

第 2 章

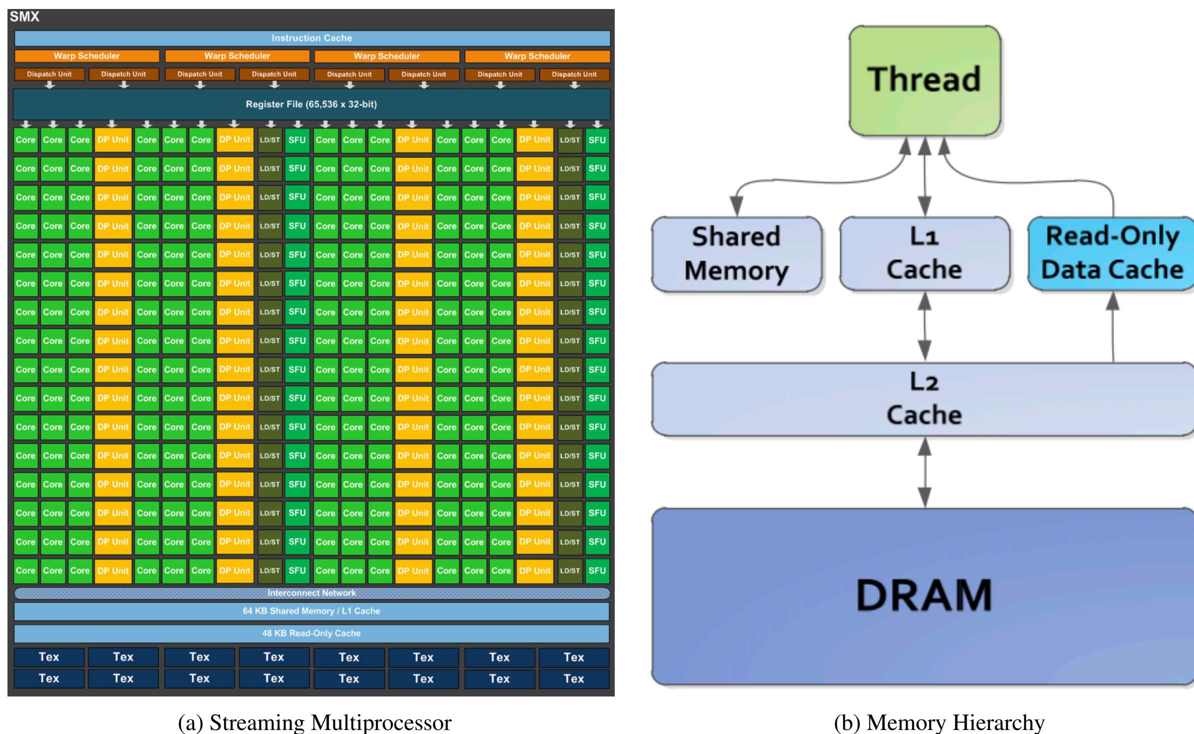
研究の背景

本章では，本研究で開発や評価の対象としたアクセラレータである NVIDIA GPU と PEZY-SC のアーキテクチャ，およびそれらの開発環境である CUDA と PZCL によるプログラミング方法を説明し，PGAS 言語 XACC のベースとなる OpenACC および XMP のモデルや記述について解説する．

2.1 NVIDIA GPU

NVIDIA GPU の HPC 向け製品である Tesla シリーズのアーキテクチャは Tesla・Fermi・Kepler・Maxwell・Pascal そして最新の Volta へと変遷している．ここでは本研究で実装・評価に用いた Kepler アーキテクチャについて解説する．GPU チップ内は複数のストリーミング・マルチプロセッサ (SMX) により構成されている．SMX の数は同世代でも製品によって異なるが Kepler では最大 15 個の SMX が搭載されている．Kepler アーキテクチャでは図 2.1a に示すように SMX 内に 192 個の CUDA コアがあり，各 CUDA コアには整数演算ユニット (ALU) がある．また同様に SMX 内には 64 個の浮動小数点演算ユニット (FPU) と 32 個の特殊関数ユニット (SFU) がある．Kepler では単精度と倍精度の両方で Fused Multiply-Add (FMA) 命令に対応している．FMA とは浮動小数数の乗算と加算 ($a + b \times c$) を一度に行う命令であり，丸め処理が一度であるため精度が高くかつ高速に演算が可能である．スレッドの実行は 32 スレッドをまとめたワープという単位で実行され，ワープ内の 32 スレッドは同時に同じ命令を実行する．Kepler では SMX 内にワープスケジューラが 4 つあるため，同時に 4 つのワープを実行できる．また各スケジューラには 2 つの命令ディスパッチユニットがあり，1 ワープの命令を最大で 2 つ同時に実行可能である．

図 2.1b に Kepler アーキテクチャにおけるメモリ階層を示す．GPU ではキャッシュや Shared メモリを利用することでデータアクセスのレイテンシ，スループットを向上させることができる．Shared メモリは SMX 上にあるオンチップメモリで SMX 内のスレッドが共有して使えるメモリである．Shared メモリと L1 キャッシュは各 SMX に合わせて 64 KB あり，16 KB / 48 KB，32 KB / 32 KB，48 KB / 16 KB の 3 通りの設定がある．Shared メモリを多用するカーネルでは Shared メモリを多く，そうでない場合は L1 キャッシュを増やしてキャッシュヒット率を高めるというようにカーネルによって設定を変えることでリソースを有効利用できる．また Kepler では新たに各 SMX に 48 KB の Read-Only Data Cache が導入された．Fermi 世代ではテクスチャデータのキャッシュとしてテクスチャユニットからしかアクセスできな



(a) Streaming Multiprocessor (b) Memory Hierarchy

図 2.1: Kepler アーキテクチャ (NVIDIA Kepler GK110 Architecture Whitepaper[1] より引用)

かったが，Kepler では通常のデータでも利用できるようになったため，以前よりも利用が容易になった．L1 キャッシュや Read-Only Data Cache と DRAM の間には 1.5 MB の L2 キャッシュも用意されている．

2.2 CUDA

CUDA は NVIDIA が提供している NVIDIA GPU 用の開発環境である．CUDA による GPU プログラミングでは主にホスト側から GPU を操作するためのホストコードと GPU 側で実行するデバイスコードの 2 つを記述する必要がある．

2.2.1 ホストコード

CUDA によるホストコード例をソースコード 2.1 に示す．GPU は独自に高バンド幅のメモリ (GDDR や HBM) を持っているため，基本的には GPU メモリ上のデータに対して処理を行う．そのために GPU で処理を行う準備として GPU メモリの確保と必要データの転送を行う．その後 GPU の処理を記述したカーネルを起動し，その処理の完了を待機する．最後に，処理結果を GPU メモリからホストメモリに転送し，GPU メモリを解放する．これらのメモリ操作は CUDA が提供する Runtime API を通して行うことができ，GPU メモリの確保・解放には cudaMalloc() と cudaFree() を，メモリコピーには cudaMemcpy() を用いる．GPU による処理は後述するデバイスコードで記述しておき，任意のスレッド数を指定して起動すると GPU 上で実行される．CUDA では多数のスレッドを管理するために 3 次元のグリッドやブロッ

クによる階層モデルが用意されている．グリッドはブロックの集まりで最大の大きさは Kepler アーキテクチャでは $(2^{31}-1)\times 65535\times 65535$ である．ブロックはスレッドの集まりで Kepler アーキテクチャでは最大 1024 スレッドまでのスレッドブロックが実行できる．各スレッドからは属するグリッドやブロックのインデックスを取得可能で，カーネル関数ではそれらを用いてスレッドごとに異なる処理を実行するよう記述する．カーネルを起動するには “*func-name* <<< *gridDim*, *blockDim* >>> (*arg0*, *arg1*, ...)” として関数呼び出しにグリッドサイズとブロックサイズ指定を加えて呼び出す．ソースコード 2.1 では 12 行目で $128\times 1\times 1$ の大きさのスレッドブロックを総スレッド数が配列サイズ以上となるグリッド数でカーネルを起動している．

2.2.2　デバイスコード

CUDA では GPU 上で実行するプログラムをカーネルと呼び，ソースコード 2.2 に示すように関数として記述する．修飾子__global__により通常の関数とは区別される．このカーネルをホストスレッドから起動することで，デバイス上で多数のスレッドで実行される．カーネルではスレッドブロック中のスレッド番号 (threadIdx)，グリッド中のブロック番号 (gridIdx)，スレッドブロックのサイズ (blockDim) からそのスレッドで処理すべきインデックスを求めて配列に値を代入する．

ソースコード 2.1: CUDA によるホストコード例

```
1 void func(int *host_a, int n)
2 {
3   int *device_a;
4   // allocate device memory
5   cudaMalloc((void**)&device_a, n * sizeof(int));
6   // copy host memory to device memory
7   cudaMemcpy(device_a, host_a, n * sizeof(int), cudaMemcpyHostToDevice);
8 
9   int num_threads = 128;
10  dim3 grid((n-1)/num_threads+1, 1, 1), block(num_threads, 1, 1);
11  // launch kernel
12  kernel<<<grid, block>>>(device_a, n);
13 
14  // copy device memory to host memory
15  cudaMemcpy(host_a, device_a, n * sizeof(int), cudaMemcpyDeviceToHost);
16  // free device memory
17  cudaFree(device_a);
18 }
```

ソースコード 2.2: CUDA によるデバイスコード例

```
1 __global__
2 void kernel(int *a, int n)
3 {
4   int i = blockDim.x * blockIdx.x + threadIdx.x;
5   if(i < n){
6     a[i] = i * i;
```

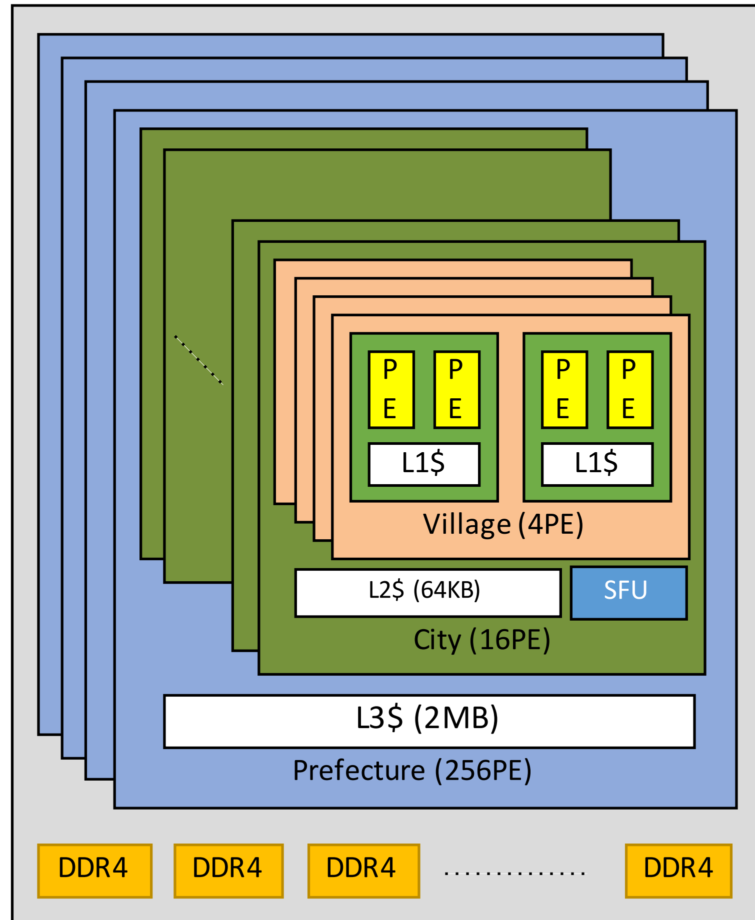


図 2.2: PEZY-SC プロセッサの構成

```
7     }
8 }
```

2.3 PEZY-SC

PEZY-SC は PEZY Computing が開発したメニーコアプロセッサである．2017 年には第 2 世代の PEZY-SC2 が登場しているが，ここでは本研究で使用した第 1 世代の PEZY-SC について解説する．図 2.2 にプロセッサの構成を示す． 1 つのプロセッサには 1024 個の Processing Element (PE) があり，1PE では 8 スレッドを実行することができる．したがって，1 プロセッサで最大 8192 スレッドを同時に実行することが可能である．また GPU とは異なり各スレッドは MIMD で動作するという特徴があり，GPU では各スレッドで条件分岐先が異なると性能低下が起きるが PEZY-SC では問題がない．

マルチプロセッサの構成は階層的な構造になっており，小さい順に PE, Village, City, Prefecture と呼ばれる．1 つの PE には 8 スレッド分のレジスタ，16 KB のローカルメモリ，2 つの ALU，2 つの FPU 等が含まれる．Village は 4 つの PE および 2 PE 毎の 2 KB L1 キャッシュで構成される．City は 4 つの Village と 1 つの SFU，1 つの 64 KB L2 キャッシュで構成され，プログラム実行時には City の単位でス

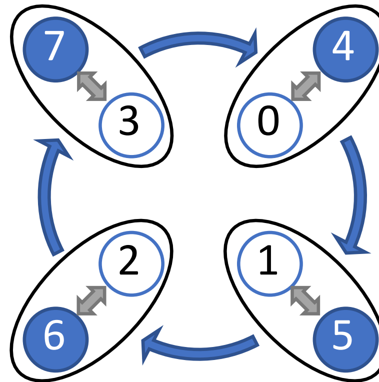


図 2.3: PEZY-SC プロセッサのスレッドの動作モデル

レッドを実行する必要がある．Prefecture は 16 個の City と 2 MB L3 キャッシュからなる．1 つのプロセッサには 4 つの Prefecture があり，全体で 1024 個の PE が含まれる．スレッドは全体同期もしくは各階層における細粒度の同期が可能である．また L1–L3 キャッシュではコヒーレンシがとられないため，あるスレッドが書き込んだデータを他のスレッドが読む場合には適切な階層レベルでキャッシュされたデータをフラッシュする必要がある．またチップ上に 8 チャネルの DDR4 メモリが接続されている．

1 つの PE で実行される 8 スレッドは図 2.3 に示すように，実際には 4 組のスレッドペアとなっている．はじめは各ペアのどちらか片側のスレッドだけがラウンドロビンで実行され，同期命令もしくはスレッド切り替え命令によって反対側のスレッドが実行されるようになる．

2.4　PZCL

PZCL は PEZY Computing が提供している PEZY-SC プロセッサ用のプログラミング環境である．OpenCL の方言であり，ホストコードは通常の OpenCL と同様であるが，デバイスコードは OpenCL C でなく通常の C++ が用いられる．

2.4.1　ホストコード

ホストコードは OpenCL と同様に C 言語の API により記述する．現在の PZCL では OpenCL 1.1 相当の機能が提供されており，既存の OpenCL コードの大部分はそのまま使用することができる．通常の OpenCL コードと異なる点について以下に述べる．多くの OpenCL コードでは実行時にカーネルコードをコンパイルするオンラインコンパイルを用いるが，PZCL では事前にカーネルコードをコンパイルしておくオフラインコンパイルのみに対応しているため，カーネルファイルの読み込みの箇所の書き換えが必要になる場合がある．次にカーネル起動時にいくつかの制限がある．実行する全スレッド数である global_work_size は 128 の倍数でなければならず，これは City の単位でスレッドを実行するためである．また OpenCL の work-group に相当するのが PE であるため，local_work_size は各 PE 内のスレッド数である 8 で固定である．加えて，OpenCL では work item の形状は少なくとも 3 次元まで指定可能であるが，PZCL では 1 次元しか指定できない．最後に，global_work_size の上限は特に決められていないが，

全スレッド数の 8192 を超えて指定した場合にはカーネルが複数回起動することになるため，8192 以下にすることが推奨されている．

2.4.2 デバイスコード

デバイスコードは C/C++ を用いて，OpenCL C とは異なる方法で記述する．ソースコード 2.3 に PZCL によりベクトル加算を行うコードの例を示す．

ソースコード 2.3: PZCL カーネルの例

```
1 void pzc_vector_add(int n, int *a, int *b, int *c)
2 {
3   int g_work_size = get_maxpid() * get_maxtid();
4   int g_work_id = get_pid() * get_maxtid() + get_tid();
5
6   for(int i = g_work_size; i < n; i += g_work_id){
7     c[i] = a[i] + b[i];
8     chgthread();
9   }
10
11  flush();
12 }
```

ホストから起動するカーネルは OpenCL や CUDA と同様に関数として定義する．関数名のプレフィックスを "pzc_" とすることでカーネル関数であることを指定する．例ではカーネル関数名を "pzc_vector_add" としている．OpenCL ではカーネルパラメータに `__global` や `__local` の指定が必要になるが，PZCL では必要ない．パラメータのポインタはグローバルメモリのポインタとして扱われる．例では，*a,b,c* はグローバルメモリのポインタで，*n* は値として渡される．また work-group 内で共有するメモリ (OpenCL で言うところのローカルメモリ) を定義するインタフェースはない．

カーネル関数内では，PE やスレッドの ID を用いて処理を並列に行う．PE の ID は get_pid() により，PE 内のスレッドの ID は get_tid() により取得できる．これらは OpenCL における get_group_id(0) や get_local_id(0) に相当する．また PE 数は get_maxpid() により，PE 内のスレッド数は get_maxtid() により取得できる．これらは OpenCL における get_num_groups(0) や get_local_size(0) に相当する．

PZCL では PEZY-SC 用にいくつかの組み込み関数を提供している．まず，chgthread() は実行をスレッドのペアの逆側に切り替える命令である．sync() はプロセッサ内のすべてのスレッドで同期する．また，細粒度な同期として sync_L1(), sync_L2(), sync_L3() がそれぞれ Village, City, Prefecture 内の同期として提供されている．"L1"–"L3" はそのキャッシュを共有するレベルで同期するという意味である．さらに，flush() は全スレッド同期と全キャッシュフラッシュを行う．こちらも同様に細粒度のフラッシュとして，flush_L1(), flush_L2(), flush_L3() が提供されている．コード例ではカーネルの最後に flush() を実行することで，キャッシュされている配列 *c* の値をメモリにフラッシュする．これを行わないとカーネル実行後にデバイスメモリ上に結果が残らず，正しく結果が得られない．PZCL においてはこれらの組み込み関数によって高速化することが重要になる．ここでは本研究で用いた 2 つの最適化について説明する．

カーネルマージ

CUDA のスレッドブロック間もしくは OpenCL の work-group 間ではカーネル実行中に同期を取ることができない．したがって全体同期が必要な部分ではカーネルを分割してカーネルを一度終了させることで全体同期を実現するが，カーネルを起動するオーバーヘッドの増加が問題となる．PZCL では起動する global_work_size を 8192 以下にした場合には sync() 命令によりスレッド全体での同期が可能であるためカーネルを分割しないで済む．

明示的スレッド切り替え

コード例のようにループ中で chgthread() により明示的に実行を逆側のスレッドに切り替えることにより 2 つの効果がある．

1. メモリアクセスなど時間のかかる命令のレイテンシ隠蔽．逆側のスレッドに実行を移すことでストール時間を短くする．
2. キャッシュヒット率の向上．ペアのスレッドはメモリ上の近い位置をアクセスすることが多いため，両側のスレッドが同じように動作することでキャッシュの利用率を高めることができる．

図 2.4 にスレッド切り替えの有無による動作の違いを示す．図の上側が明示的なスレッド切り替えなし，下側がありの場合を示している．図中の丸付き数字は実行される順番を示している．明示的なスレッド切り替えなしの場合では，最初はスレッド 0–3 のみがループの本体を実行する．その後，ループを抜けて flush() に到達した際に，暗黙的にスレッド切り替えが発生する．そしてスレッド 4–7 がループを実行して flush() に到達して処理が終了する．この場合メモリアクセスは double 型 4 要素，つまり 32byte ごとに飛び飛びのアクセスになり，L1 キャッシュは読み込んでもその半分しか利用されない．次に切り替え命令ありの場合では，はじめにスレッド 0–3 が配列 *a[]* にアクセスするがその後にスレッドが切り替わり，スレッド 4–7 が配列にアクセスする．これをループが終わるまで繰り返して，全スレッドがほぼ同時に flush() に到達する．表と裏のスレッドがほぼ同時に処理が進むことでキャッシュが有効に利用できるうえ，メモリロード後にスレッド切り替えを入れることでメモリアクセスによるストールが軽減できる．

2.5 OpenACC

OpenACC は C・C++・Fortran コードの一部をアクセラレータにオフロードするための指示文ベースのプログラミングモデルである．2011 年 11 月に CAPS・Cray・NVIDIA・PGI により策定され，最新の仕様は 2017 年 11 月に策定された version 2.6 である．データや演算のオフロードを指示文で簡易に記述できるため習得が容易で，既存の逐次コードをベースに開発できるため生産性が高い．またアクセラレータの種類に依存しない記述であるため，コードの可搬性も維持される．

2.5.1 実行モデル

OpenACC ではホストから GPU のようなホストに接続されたアクセラレータに処理を行わせることが可能である．コード中の負荷の高い部分を `parallel` または `kernels` 指示文で指定すると，その

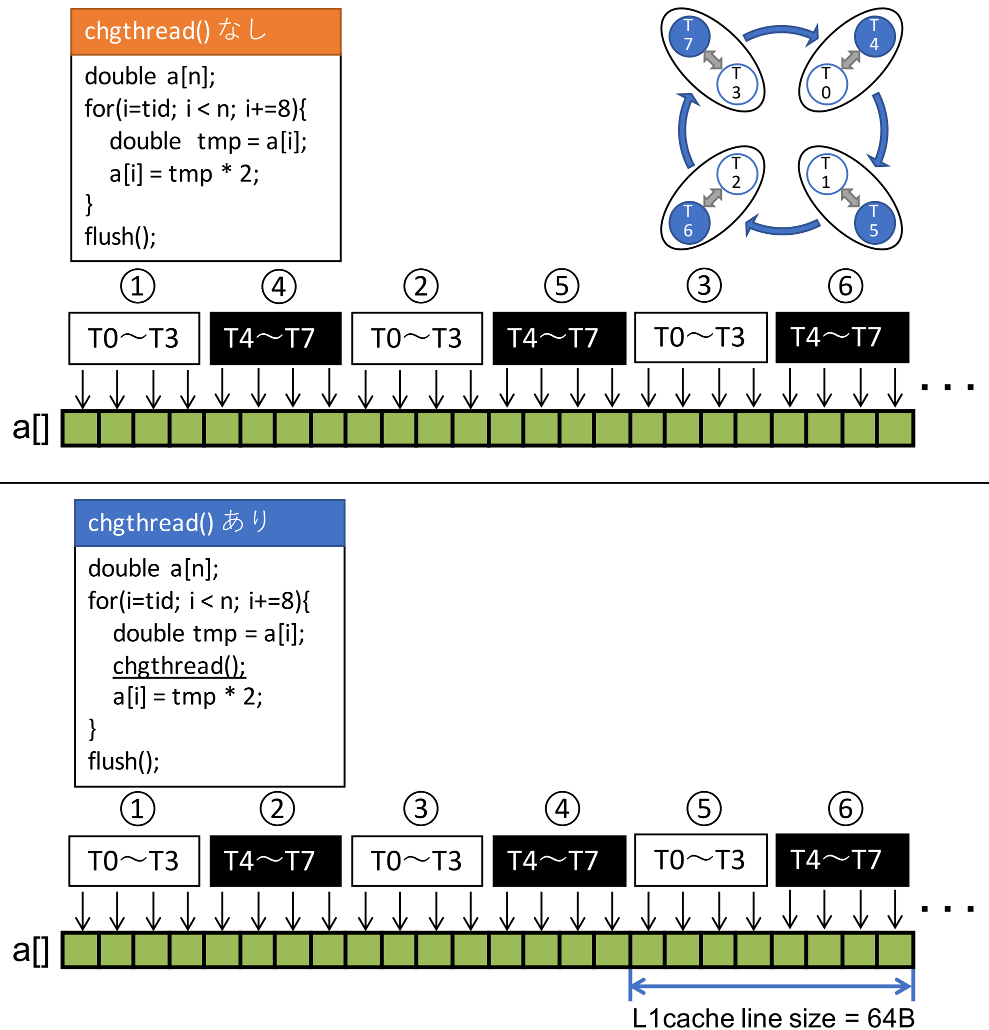


図 2.4: PEZY-SC プロセッサにおける明示的スレッド切り替えの有無による違い．上側が明示的スレッド切り替え無し，下側が有りの場合を示す．丸付き数字は実行順序を示す．

処理はアクセラレータでカーネルとして実行される．OpenACC では上位から gang, worker, vector の 3 段階の並列性が存在する．gang は粗粒度の並列性で完全に並列に動作し gang 間での同期は基本的にできない．worker は細粒度の並列性である．各 gang は 1 つ以上の worker を持ち，各 worker では vector を用いてベクトル演算を行うことができる．これら 3 つの並列性が実際のアクセラレータの演算器とどのように対応付けられるかはコンパイラ依存である．カーネル領域のプログラムは何も指定がなければ gang, worker, vector はそれぞれ gang-redundant，worker-single，vector-single モードで実行される．gang-redundant モードではすべての gang が同じ処理を重複して行う．worker-single 及び vector-single モードでは 1 つの worker，1 つの vector レーンのみが処理を実行する．並列化可能なループに対して `loop` 指示文で並列化を指定した際には，gang-partitioned, worker-partitioned, vector-partitioned モードに移行し，並列実行を行う．

2.5.2　メモリモデル

現在多くのアクセラレータでは物理的または仮想的にホストメモリとアクセラレータメモリが分かれている．OpenACC のモデルでは，ホストとアクセラレータのメモリ間のコピーはコンパイラによって暗黙に管理される．しかしながら，指示文によりメモリの管理をすることも可能で，data 指示文などによってデータの生存期間を明示することで，不要なメモリ確保やデータ転送を削減することができる．アクセラレータが実際にホストメモリと別のメモリを持つ場合には，指定されたようにデータ確保を行い，アクセラレータとホストがメモリを共有している場合には確保や移動は行わない．

2.5.3　プログラム例

OpenACC 指示文を追加した C 言語のコードをソースコード 2.4 に示す．4 行目では data 指示文を使用して 5–10 行目においてアクセラレータメモリに配列 *a[]* と *b[]* を確保するよう明示する．配列 *a[]* の値はデバイスで参照されるが更新はされないため，copyin 節を用いて領域の最初にホストメモリからアクセラレータメモリへコピーするよう指定している．他方で配列 *b[]* の値はデバイスで更新されるが参照はされないため，copyout 節を用いて領域の最後にアクセラレータメモリからホストメモリへコピーするよう指定している．6 行目では parallel loop 指示文を用いて変数 *i* に関するループをアクセラレータで並列実行するよう指定している．以上の指示文からコンパイラはこのループをアクセラレータで実行するよう適切に変換を行う．

ソースコード 2.4: OpenACC のコード例

```
 1 #define N 1024
 2 int main(){
 3   int a[N], b[N];
 4 #pragma acc data copyin(a) copyout(b)
 5   {
 6 #pragma acc parallel loop
 7     for(int i = 0; i < N; i++){
 8       b[i] = a[i] + 1;
 9     }
10   }
11 }
```

2.6　XcalableMP

XcalableMP (XMP) は分散メモリシステム向けに C と Fortran を拡張した PGAS 言語であり，PC クラスタコンソーシアム [10] の XMP 規格部会により仕様が策定されている．XMP は 2 つのプログラミングモデルを提供しており，一つは逐次コードに指示文を追加することによりノード全体の動作を記述するグローバルビューモデルで，もう一つは MPI プログラミングのように各ノードの動作を記述するローカルビューモデルである．

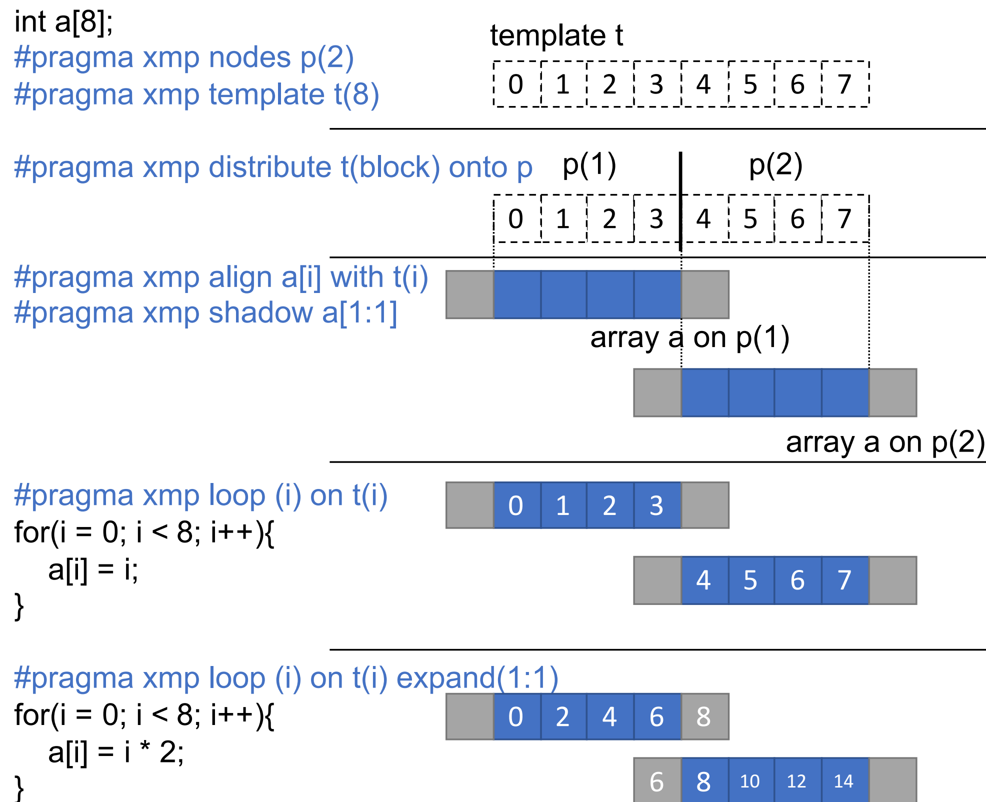


図 2.5: XMP グローバルビューモデルによるプログラム例

2.6.1 グローバルビューモデル

グローバルビューモデルは High Performance Fortran (HPF)[11] の影響を受けた指示文ベースのプログラミングモデルである．逐次コードに XMP 指示文を追加することで分散メモリプログラムを記述する．HPF ではユーザは指示文によってデータ分散の指定のみ行い，コンパイラが処理の並列化や通信を自動で行うのに対して，XMP はユーザがデータ分散・処理のマッピング・通信・同期を指示文で明示する必要があるため，HPF に比べて挙動が明確で性能チューニングがしやすいという特徴がある．

グローバルビューモデルのプログラミングを図 2.5 に示す．XMP における実行単位はノードであり，MPI におけるプロセスに対応するものである．はじめに，`nodes` 指示文によりノード集合を定義する．XMP ではインデックス空間を表す仮想的な配列であるテンプレートを用いて，データと処理のマッピングを行う．`template` 指示文はこのテンプレートを定義する．なおテンプレートを未定義サイズで指定しておき，後から `template_fix` 指示文によりサイズを指定することも可能である．`distribute` 指示文はテンプレートをノード集合にどの方法で分散するかを指定する．分散方法としては `block`，`cyclic`，`gblock`，もしくは*(分散しない) がある．`gblock` 分散は “general block” 分散の略であり “`gblock`(*blocksize-array*)” と記述することで，各ノードのブロックサイズを *blocksize-array* で指定したブロックサイズにすることが可能である．`align` 指示文は分散されたテンプレートに沿って配列を

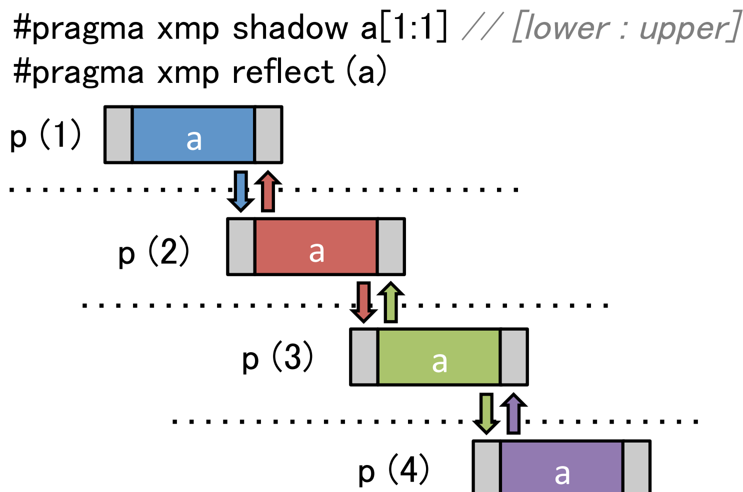


図 2.6: shadow と reflect 指示文の例

分散する．また shadow 指示文はシャドウと呼ばれるいわゆる袖領域を分散配列に指定するもので，“(*lower-width* : *upper-width*)” という形式で下側・上側の袖幅を指定できる．loop 指示文は直後のループをテンプレートに沿って分散する指示文である．通常は割り当てられたループ範囲が自分のノードに割り当てられたテンプレートの範囲を超えることはないが，XMP 1.3 の仕様で追加された expand 節を用いると，ローカルのループ範囲を拡張することが可能である．図では最後のループに expand 節を指定したことで，*p(1)* ではループ範囲が 0–4 に，*p(2)* では 3–7 に拡大されている．次に XMP の通信指示文について解説する．最も特徴的なのはステンシル計算において袖交換を行う reflect 指示文である．図 2.6 は reflect 指示文の例である．配列 *a[]* は 4 ノードに分散されており，幅が 1 要素のシャドウ領域を持っている．reflect 指示文により隣接ノードが持っている実際の領域の値でシャドウ領域を更新する．width 節を用いると更新する幅を指定でき，もし指定がなければシャドウ領域すべてが更新される．reduction 指示文は変数や配列の値をノード間で集計する指示文である．演算子には OpenMP と同様に +, *, min, max 等を指定できる．gmove 指示文は変数や分散配列に対する様々な代入処理を行うことが可能な指示文である．代表的な 3 種類の通信を図 2.7 に示す．(a) は分散配列から分散配列への代入でこの例では send-recv の通信となる．(b) は分散配列からローカル配列への代入で，この例では broadcast の通信となる．(c) は分散配列から分散配列への転置で，all-to-all の通信となる．

2.6.2　ローカルビューモデル

ローカルビューモデルは coarray という他のノードからもアクセスが可能な配列を通信に用いるプログラミングモデルである．coarray は Coarray Fortran [12] として提案され，Fortran 2008 から正式に導入された機能である．XMP Fortran (XMP/F) は Fortran 2008 との互換性があり，XMP C (XMP/C) は C の文法を拡張することで coarray に対応している．データや処理の分散は MPI と同様にユーザが行う必要があるが，coarray では配列の次元のようにノードを扱えるため，通信を配列代入文で簡易に記述できる利点がある．

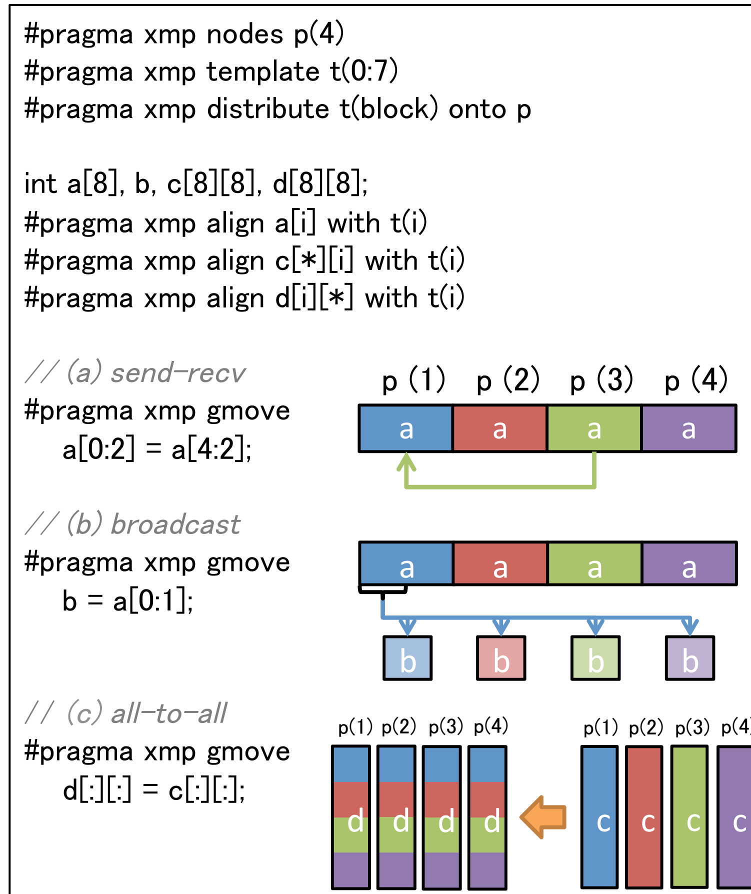
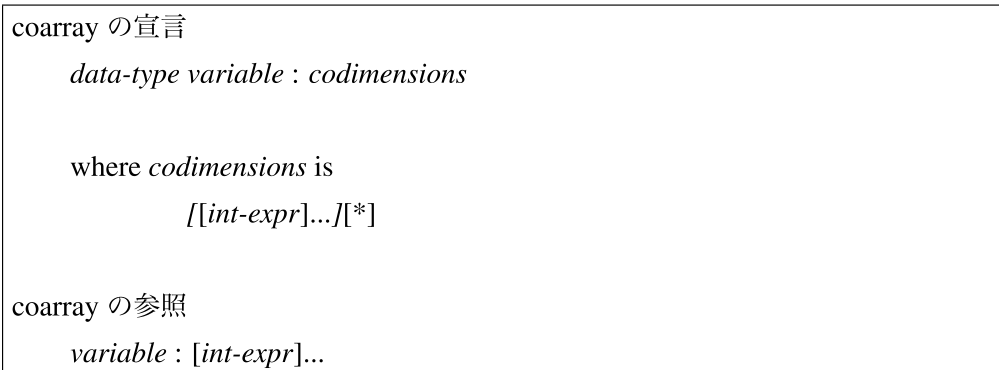


図 2.7: gmove 指示文の例



coarray の宣言

data-type variable : *codimensions*

where *codimensions* is

[[*int-expr*]...*]*[*]

coarray の参照

variable : [*int-expr*]...

図 2.8: XMP/C における coarray の文法

図 2.8 は XMP/C における coarray の文法である．coarray では実行単位をイメージと呼び，これは XMP グローバルビューモデルにおけるノードに相当する．coarray は通常の変数宣言の最後に *codimensions* を追加することで宣言する．イメージ集合の形状は多次元で指定することが可能であり，データを多次元分散する際に役立つ．他のイメージの coarray にアクセスする場合には，通常の変数参照に加えて *co-indices* を指定する．もし co-indices が指定されなければ，通常の変数と同じ扱いになるため自分のイメージで

```
#pragma xmp post ( nodes-ref, tag )
#pragma xmp wait [( nodes-ref [, tag] )]
```

図 2.9: post/wait 指示文の構文

のローカルアクセスとなる．リモートの coarray に対しての代入は PUT 通信，参照は GET 通信となる．Fortran では配列アクセスで範囲を指定可能であるが，C では要素ごとのアクセスしか対応していない．XMP/C では言語拡張で配列アクセスにおける範囲記述に対応しており，“[*lower[*:*length[*:*stride]]*]” の形式で記述できる．coarray を用いた通信では，要素ごとのアクセスはオーバーヘッドが大きいため，配列の範囲を用いて記述することで効率良く通信することが可能である．

次に coarray における同期方法について説明する．XMP/F においては Fortran 2008 と同様に， sync memory, sync image, sync images, sync all が文として定義されており，XMP/C においては関数として xmp_sync_memory(), xmp_sync_image(), xmp_sync_images(), xmp_sync_all() が定義されている．sync memory はこれより前の coarray 操作がローカルで完了するまで待機する．sync image / sync images は自身と指定した image 間での coarray 操作が完了するまで待機する．sync all はすべてのイメージ間での coarray 操作が完了するまで待機する．

XMP/C では追加機能として，同期のためのシグナルを送受信する `post` と `wait` 指示文を提供している．図 2.9 は `post/wait` 指示文の構文である．*nodes-ref* はグローバルビューモデルにおける XMP ノード集合の参照であるが，実質ローカルビューモデルにおけるイメージ番号と同じである．`post` 指示文では *tag* を指定して同じノードからの複数のシグナルを区別することができ，`wait` 指示文では *tag* を指定した場合は対応するシグナルを待つが指定がなければ任意のタグのシグナルを待つ．xmp_sync_memory() と `post/wait` 指示文を組み合わせることにより xmp_sync_image() と同等，もしくはより柔軟な同期を記述することが可能である．

図 2.10 は XMP ローカルビューモデルによるコード例である．*a[4]:[*]* は codimension が指定されているため coarray である．各イメージはそれぞれ coarray *a*[] を重複して保持する．イメージ 1 は配列の範囲表記を用いて自イメージの *a*[2–3] をイメージ 2 の *a*[0–1] に代入 (put) する．xmp_sync_all() の呼び出しにより全イメージでの coarray 通信の完了を待機する．その後，イメージ 2 では *a*[0–1] に書き込まれた値を読むことが可能になる．

```
int a[4]:[*];   // coarray宣言
int b, st;


if( xmp_this_image() == 1 ){
   for(i = 0; i < 4; i++) a[i] = i * 2;

   a[0:2]:[2]  = a[2:2]; // put
}

xmp_sync_all(&st);

if( xmp_this_image() == 2 ){
   b = a[0] + a[1];
}
```

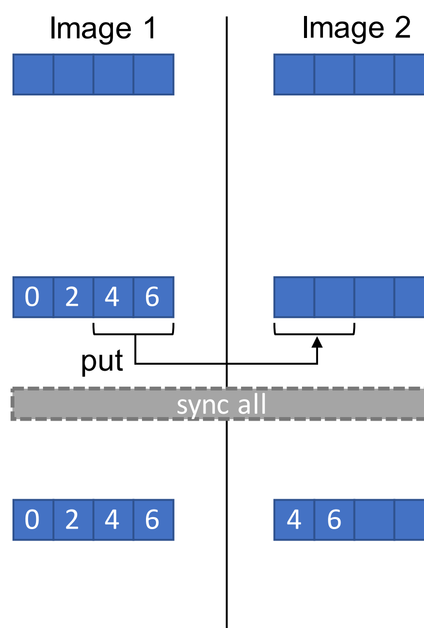


図 2.10: XMP ローカルビューモデルによるプログラム例

第 3 章

演算加速機構を持つ並列クラスタ向け PGAS 言語 XcalableACC の提案

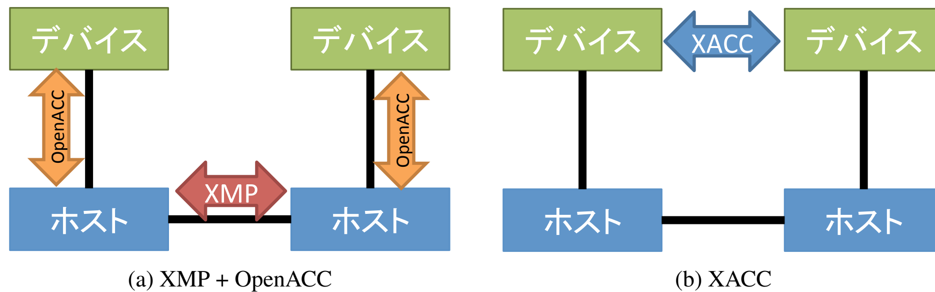
本章では演算加速機構を持つ並列システム向けの PGAS 言語 XcalableACC のプログラミングモデルや記述方法について述べる．XACC は筑波大学と理研が共同で提案している PGAS 言語で，既存の PGAS 言語 XMP とアクセラレータプログラミングモデル OpenACC を統合した言語である．現在，仕様の version 1.0 を公開している [9].

3.1　概要

XACC は XMP と OpenACC を統合したプログラミングモデルであり，分散メモリ並列化を XMP で記述し，アクセラレータへのオフロードを OpenACC で記述する．XACC の特徴の一つはアクセラレータメモリ上のデータに対するノード間通信である．図 3.1 はアクセラレータ上のデータをノード間で通信する際の記述イメージを XMP と OpenACC の組み合わせと XACC とで比較したものである．単に XMP と OpenACC を組み合わせた際には，アクセラレータ間の通信を行う指示文が存在しないため図 3.1a のように OpenACC によるホストとアクセラレータ間の通信と XMP によるホスト間の通信が必要である．XACC では既存の XMP の通信指示文や coarray を拡張することで，図 3.1b のようにアクセラレータ間の通信を直に記述可能である．この機能により，(1) プログラマの意図することを簡潔かつ直接的に記述できる，(2) コンパイラ・ランタイムが実行環境ごとに適切な通信を行える，という利点がある．(2) の例として，NVIDIA GPU において GDR が利用可能な環境であればそれを利用して高速に通信をすることができる．XMP グローバルビューモデルをベースにした XACC グローバルビューモデルが中尾らにより提案されている [13]．本研究では XMP ローカルビューモデルをベースとした XACC ローカルビューモデルを新たに提案する．

3.2　グローバルビューモデル

XACC グローバルビューモデルでは基本的に XMP グローバルビューモデルにおける指示文ベースの分散や通信の記述と OpenACC 指示文の組み合わせにより記述する．XMP 指示文により分散された配



(a) XMP + OpenACC　(b) XACC

図 3.1: デバイス間通信の記述方法

列およびループに対して OpenACC 指示文を指定する方法やアクセラレータ間の通信の記述方法とその動作について説明する．分散配列をアクセラレータ上に確保およびコピーするには通常の変数と同様に OpenACC `data`, `enter data`, `declare` 指示文を用いる．指定された分散配列の各ノードが保持する部分のみがそのノードが保持するアクセラレータに確保やコピーされる． また `update` 指示文を用いてホスト側とアクセラレータ側で分散配列の更新が可能である．

次に，XMP `loop` 指示文で分散されたループに対しては OpenACC `parallel loop`，`kernels loop`，`loop` 指示文が指定できる．OpenACC `parallel loop`，`kernels loop` 指示文では各ノードが担当するループ範囲をそのノードに接続されているアクセラレータで実行する．ただし OpenACC `loop` 指示文の場合は，その XMP `loop` 指示文のループの分散やループ範囲がカーネル領域の内で不変でなければならない，という制約がある．分散ループに `loop` 指示文のみ指定する場合は，すなわちそのループが `parallel` か `kernels` 指示文でアクセラレータでの実行対象として指定されているということである．実装の観点から，アクセラレータカーネル内でループの分散を行うのは非効率であるため，カーネルに入る前に分散の処理をできるよう制約を設けている．

アクセラレータ上のデータに対する通信は，XMP 通信指示文の最後に `acc` 節を加えることで指定する．この場合，アクセラレータ上のデータのみ対象となるためホスト上のデータは変化しない．また，XMP `gmove` 指示文においては “`acc`*[(variable)]*” として対象の変数名が指定できるようになっており，その場合には gmove の左辺もしくは右辺の指定された変数のみでアクセラレータ側のデータが用いられる．

図 3.2 は XACC グローバルビューモデルのプログラム例である．左側のコードは XMP コードをベースにした際に XACC で追加・変更した部分を青色で強調してある．この例ではまず，XMP 指示文により 2 ノードに配列 *a[]* をブロック分散してかつ上下端に袖領域を追加する．次に，分散配列 *a[]* をアクセラレータで処理するために OpenACC `data` 指示文によりアクセラレータメモリ上に確保・コピーを行う．この際，各ノードがローカルに持っている部分のみがアクセラレータに配置される．XMP で分散されたループにさらに OpenACC `kernels loop` 指示文を追加することで，そのループをアクセラレータで処理するよう指定する．XMP `reflect` 指示文の最後に `acc` 節を追加することにより，アクセラレータ上の分散配列 *a[]* に対して袖通信が実行される．この際，ホストメモリ上にある分散配列 *a[]* は変更されない．

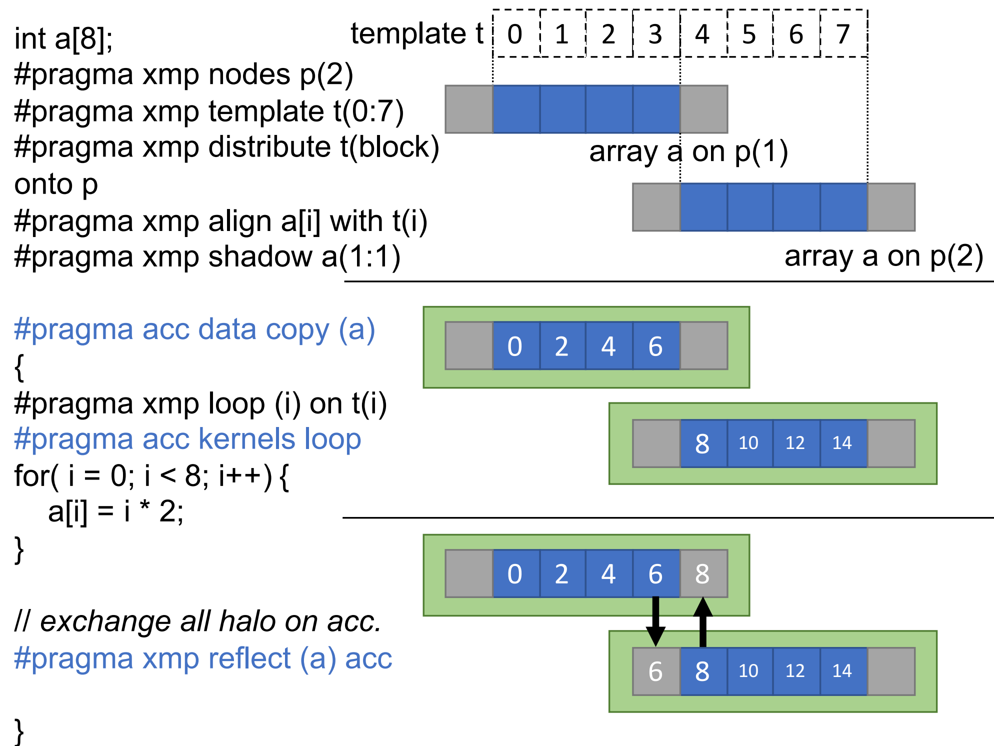


図 3.2: XACC グローバルビューモデルのコード例

3.3　ローカルビューモデル

XMP ローカルビューモデルの coarray は Fortran2008 からの正式な仕様であるが，現在最新の OpenACC 2.6 においては coarray に対して OpenACC がどのように動作するかは定義されていない．そこで XACC では coarray をアクセラレータ間の通信でも利用できるように，アクセラレータ上に coarray を宣言しそれを参照するための記述方法を提案する．coarray をアクセラレータ上に確保するには OpenACC `declare` 指示文を用いる．`declare` 指示文は指定した変数をそのスコープにおいてアクセラレータ上にも確保する指示文である．coarray に対する `declare` 指示文の利用時にはコピーを伴わない `create` 節のみ指定できる．通常のホストメモリ上の coarray 宣言の後に `declare create` でその coarray を指定すると，アクセラレータメモリ上にも同じ coarray が宣言される．なお，OpenACC にはアクセラレータ上にデータを確保する指示文として他に `data` と `enter data` が存在するが，いずれもアクセラレータにおけるデータの生存期間が動的であったり部分配列が指定可能であったりして自由度が高く，coarray の宣言には適さないので許可していない．

アクセラレータ上にも coarray を宣言した場合でも特に指定がない場合はホストメモリ上の coarray に対するアクセスになる．アクセラレータ上の coarray をアクセスするには OpenACC `host_data` 指示文を用いる．この指示文の `use_device` 節にはアクセラレータ上にデータが確保されている変数を指定でき，その変数は直後のコード領域においてホストコードにおける変数アドレスがアクセラレータメモリ

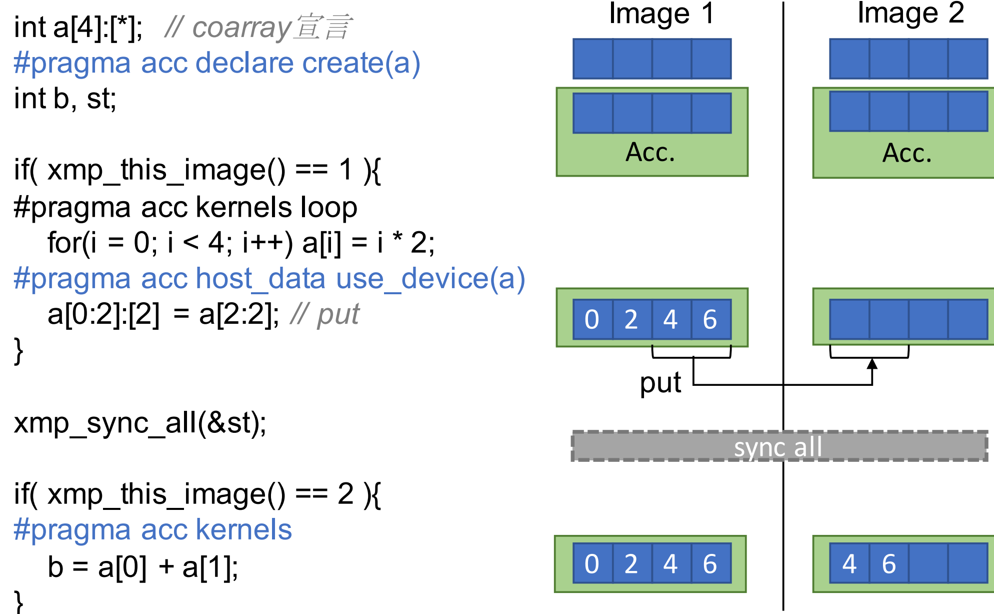


図 3.3: XACC ローカルビューモデルのコード例

側のアドレスに置きかわる．この指示文は主にアクセラレータ向けに書かれた外部ルーチンにアクセラレータメモリ側のアドレスを渡すために使用される．`use_device` に，coarray を指定した場合にはその coarray のアクセスはアクセラレータ側に対して行われる．なお，リモートの coarray のみ `use_device` で指定し，ローカルの coarray もしくは通常の変数には何も指定しなかった場合は，リモートのアクセラレータメモリとローカルのホストメモリでの通信が行われる．アクセラレータ上の coarray を用いた通信の同期はホスト側の coarray と同様に，sync all や sync image などで行う．アクセラレータ上の coarray は通常の変数と同じように `parallel`, `kernels` 領域中でアクセスしたり，`update` 指示文でホストとアクセラレータ間でコピーしたりすることが可能である．

XACC ローカルビューモデルのコード例を図 3.3 に示す．ホスト上に通常の coarray *a[]* を宣言した後に，OpenACC `declare` 指示文で *a[]* をアクセラレータメモリ上にも宣言する．まずイメージ 1 において，coarray *a[]* に値を代入し，その後 `use_device` で coarray *a* を指定することでアクセラレータ上の *a[2:2]* の値をイメージ 2 の *a[0:2]* にリモートライトする．xmp_sync_all() で同期を行った後，イメージ 2 は書き込まれた *a[0:2]* のデータを用いて処理を行う．

3.4　関連研究

演算加速機構を持つ並列システム向けに XMP を拡張した XcalableMP acceleration device extension (XMP-dev) が李らにより提案されており，XMP-dev から CUDA や OpenCL に変換するコンパイラが実装されている [14, 15, 16]．XMP にアクセラレータ上のデータ宣言，ホスト・アクセラレータ間のデータ転送，アクセラレータでの並列ループ実行，およびアクセラレータ間の通信を指定する指示文を追加することにより，演算加速機構を持つ並列システムで利用できる記述能力と性能を有することを明らか

にした．特にノード間のアクセラレータメモリ上のデータ通信が性能向上に大きく貢献することを示した．XMP-dev はオフロードに独自の指示文を用いていたが，XACC は OpenACC が標準規格として提案されたことを受けて，XMP-dev を発展させ，オフロードを OpenACC に沿った指示文にすることにより OpenACC ユーザが分かりやすくなるようにした．また特にデータの管理に関して，XMP-dev ではアクセラレータ上にデータが有るか無いかに関わらずデータを確保することしかできなかったが，XACC ではオフロードについて OpenACC を取り入れることにより，すでにある場合には確保しないなど，十分な操作ができるようになった．さらに XMP-dev ではグローバルビューの通信にのみ対応していたが，XACC ではローカルビューモデルにも対応することでより柔軟な通信の記述を可能としている．性能の点では，XMP-dev は CUDA もしくは OpenCL に変換し，XACC は OpenACC に変換するという違いがあるが，OpenACC はさらに CUDA や OpenCL に変換（コンパイル）することができるので，同等な記述に関しては同等な性能が得られるはずである．ただし，OpenACC は CUDA 等よりも高レベルな記述であるため新規のプラットフォームに対応しやすいが，XMP-dev では開発当時のアクセラレータのアーキテクチャに合わせた CUDA コードや OpenCL コードになっており，現在主流のアーキテクチャには適さない可能性がある．

第 4 章

OpenACC コンパイラの設計・実装

本章では，XACC のベースとなるアクセラレータ向け指示文ベースプログラミングモデル OpenACC のコンパイラを設計・実装し，ベンチマークを用いて評価を行う．OpenACC は初の標準仕様であったが，当初はオープンソースのコンパイラがなく，商用コンパイラにおいても NVIDIA GPU のような一般的なアクセラレータのみの対応で，最適化が不十分であった．そこで，OpenACC から source-to-source 変換を行うコンパイラを設計し，NVIDIA GPU と PEZY-SC に向けて実装および最適化を行う．そしてベンチマークプログラムを用いて，NVIDIA GPU 向けにおいては既存の商用コンパイラとの性能比較を，PEZY-SC 向けにおいては PZCL コードとの性能・生産性の比較を行う．

4.1　NVIDIA GPU 向け OpenACC コンパイラ

NVIDIA GPU を対象に実装した Omni OpenACC compiler についてその設計と実装について述べる．

4.1.1　設計

OpenACC は C・C++・Fortran に対応しているが，このコンパイラでは C 言語のみを対象とする．NVIDIA GPU のプログラミングには言語・コンパイラともに成熟している CUDA を用いる．本実装では CUDA のカーネルコードにおいてループ並列化以外はできる限り元のソースコードを維持することにより，NVIDIA CUDA compiler (NVCC) による最適化が十分に行われ，より高速な実行ファイルが生成されるように変換を行う．これを実現するためにコードの解析や元のコードを維持した変換が可能である Omni compiler infrastructure[17] を実装に用いる．これは C と Fortran95 をソースからソースへ変換するコンパイラ用のプログラムセットである．またホストコードにおいては多くの処理を実行時ライブラリの呼び出しに変換することでプログラムの可搬性を向上させる．コンパイルの流れを図 4.1 に示す．OpenACC 指示文の書かれた C 言語のコードをコンパイルする際には，まず Omni frontend で XML 形式の中間コードである XcodeML に変換する．それを OpenACC translator で読み込んでコード変換し，ホストコードと CUDA コードを出力する．その際 GPU で実行する部分はカーネル関数として分離しループの並列化を行うが，ループの内部は変更しない．また Kepler アーキテクチャ向けの最適化として，読み込みのみの配列に対して Read-only data cache が使われるよう指定したり，リダクションにおいて shuffle

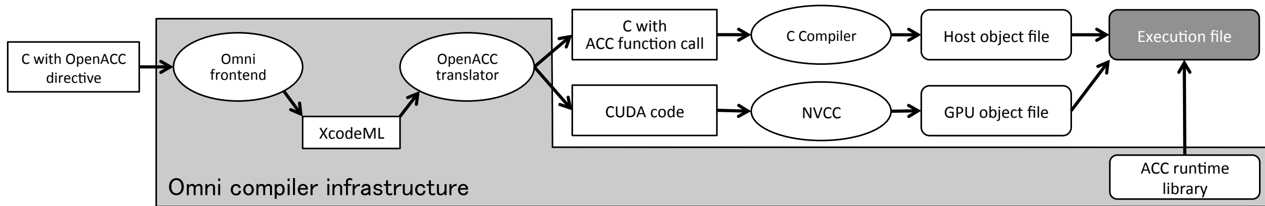


図 4.1: Omni OpenACC compiler のコンパイルの流れ

命令や atomic 命令を用いたりする．一方でホストコードでは指示文を実行時ライブラリ呼び出しに置き換える．ホストコードは一般的な C コンパイラでコンパイルし，CUDA コードは NVCC でコンパイルする．最後にコンパイルされた 2 つのオブジェクトファイルを OpenACC runtime library とリンクして実行可能バイナリを生成する．

4.1.2 data 構文の実装

データ指示文が指定された領域では指示節に基づいて，データ領域の最初にデバイスメモリの確保とホストメモリからデバイスメモリへの転送，データ領域の最後にデバイスメモリからホストメモリへの転送とデバイスメモリの解放を行う．これらの操作はすべてライブラリ関数の呼び出しに変換する．

ソースコード 4.1 に `data` 構文例を，ソースコード 4.2 にその変換例を示す．この `data` 構文では領域内で配列 *a[]* と変数 *b* のためのデバイスメモリを確保する．また copy 節内で指定されている配列 *a[]* は領域の最初と最後で転送し，copyout 節内で指定されている変数 *b* は領域の最後でのみ転送する．デバイスメモリの確保にはライブラリ関数_ACC_init_data() を用いる．この際，配列の場合には各次元の lower と length を指定する．DEV_ADDR_*name* はホスト上の *name* に対応するデバイスメモリのポインタである．DESC_*name* は *name* のホストアドレス，デバイスアドレス，配列形状，要素サイズ等が格納された構造体へのポインタである．ホストメモリとデバイスメモリ間の転送はライブラリ関数_ACC_copy_data() で行う．ライブラリ関数_ACC_finalize_data() により最後にデバイスメモリの解放を行う．

ソースコード 4.1: data 構文の例

```
1 int a[100], b;
2 #pragma acc data copy(a) copyout(b)
3 {
4   /* some codes using a and b */
5 }
```

ソースコード 4.2: data 構文の変換例

```
1 int a[100], b;
2 {
3   void *DESC_a,*DEV_ADDR_a,*DESC_b,*DEV_ADDR_b;
4   unsigned long long _lower[] = {0};
5   unsigned long long _length[] = {100};
6   _ACC_init_data(&(DESC_a),&(DEV_ADDR_a),a,sizeof(int),1,_lower,length);
7
```

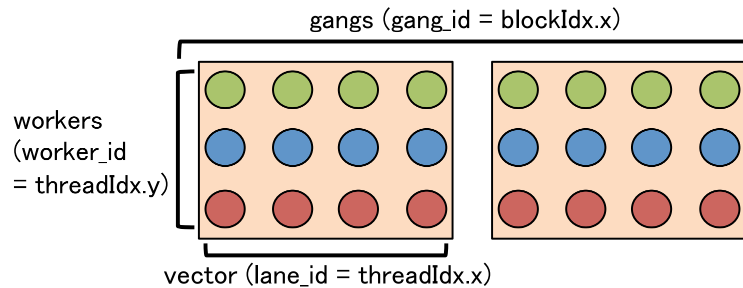


図 4.2: OpenACC の並列性の CUDA へのマッピング

```
 8   _ACC_init_data(&(DESC_b),&(DEV_ADDR_b),&(b),sizeof(int),0,NULL, NULL);
 9   _ACC_copy_data(DESC_a,_ACC_HOST_TO_DEVICE,_ACC_ASYNC_SYNC);
10   {
11     /* some codes using a and b */
12   }
13   _ACC_copy_data(DESC_a,_ACC_DEVICE_TO_HOST,_ACC_ASYNC_SYNC);
14   _ACC_copy_data(DESC_b,_ACC_DEVICE_TO_HOST,_ACC_ASYNC_SYNC);
15   _ACC_finalize_data(DESC_a);
16   _ACC_finalize_data(DESC_b);
17 }
```

4.1.3　parallel 構文の実装

parallel 指示文で指定されたコード領域を GPU で実行するには，GPU 上での処理を記述したカーネル関数を作成し，それをホストから起動する必要がある．CUDA ではカーネルは 3 次元のグリッドとスレッドブロックで実行され，各スレッドはそのインデックスを用いて並列に処理を行う．本実装における OpenACC の並列性の CUDA へのマッピングを図 4.2 に示す．gang, worker, vector をそれぞれ CUDA のグリッドの X 次元，スレッドブロックの Y 次元，スレッドブロックの X 次元に対応させ，その他の次元は 1 として利用しない．それぞれ CUDA の組み込み変数 blockIdx.x，threadIdx.y，threadIdx.x により識別できる．なお現在の実装では worker 数，すなわちスレッドブロックの Y 次元は必ず 1 としている．カーネル関数を起動する際の vector 長はデフォルトで 256 とし，gang 数は gang で並列化するループ長から決定する．ループを gang のみで並列化する場合はそのループ長に，gang と vector で並列化する場合はループ長を vector 長で割った数にする．加えて，カーネルが起動できるグリッドの X 次元のサイズは Fermi アーキテクチャでは 65535，Kepler アーキテクチャでは $2^{31}-1$ という制限があるため gang 数の調整が必要となる．Kepler アーキテクチャにおいてもスレッドブロック数が増えるとオーバーヘッドが増加するため，Fermi アーキテクチャの制限に合わせて gang 数が 65535 個以下となるように $numGangs' = \lceil numGangs/\lceil numGangs/65535\rceil\rceil$ とする．gang 数の整数倍がおよそループ長になるため gang 間の負荷が均等になりやすい．

カーネル関数を起動する際には参照するデータのポインタ，もしくは値を直接渡す必要がある．コンパイラはカーネル領域内で使用されるデータを調べ，それらに対する GPU メモリのポインタをカーネル

関数の引数として渡す．ただし firstprivate 節で指定された変数に対しては値渡しにすることでその変数の初期化を実現している．さらに Kepler アーキテクチャ用にコンパイルする際にはリードのみのデータに対して Read-Only Data Cache を利用するようコードを生成する．カーネル関数の仮引数でのポインタの宣言に “const __restrict__” 修飾子を付けることで Read-Only Data Cache を介してロードするよう CUDA コンパイラにヒントを与える．GPU メモリの確保等については 4.1.2 節で述べる．

ソースコード 4.3 に parallel 構文例を，ソースコード 4.4 にその変換例を示す．関数_ACC_GPU_FUNC0_DEVICE() がカーネル関数である．gang 数が 4，vector 長が 64 なので 4 スレッドブロック，1 スレッドブロックあたり 64 スレッドとする．関数_ACC_GPU_ADJUST_GRID() でブロック数を 65535 以下になるように調整したのち，カーネル関数に変数 a の GPU メモリのポインタを渡して起動する．その後カーネル関数の終了を_ACC_wait() で待機する．

ソースコード 4.3: parallel 構文の例

```
1 #pragma acc parallel present(a) num_gangs(4) vector_length(64)
2 {
3   /* codes in parallel region */
4 }
```

ソースコード 4.4: parallel 構文の変換例

```
1  __global__ static
2  void _ACC_GPU_FUNC_0_DEVICE(void *a...)
3  {
4    /* codes in parallel region */
5  }
6  extern "C"
7  void _ACC_GPU_FUNC_0(void *DEV_ADDR_a, ...)
8  {
9    int block_x=(4), block_y=(1), block_z=(1);
10   int thread_x=(128), thread_y=(1), thread_z=(1);
11   _ACC_GPU_ADJUST_GRID(&block_x,&block_y,&block_z,65535);
12   dim3 DIM3_block(block_x, block_y, block_z);
13   dim3 DIM3_thread(thread_x, thread_y, thread_z);
14
15   _ACC_GPU_FUNC_0_DEVICE<<<DIM3_block,DIM3_thread,0,
16     _ACC_gpu_get_stream(_ACC_ASYNC_SYNC)>>>(DEV_ADDR_a, ...);
17   _ACC_wait(_ACC_ASYNC_SYNC);
18 }
```

4.1.4 loop 構文の実装

loop 指示文に続く for ループは GPU 上で並列に実行するように変換する．並列化に用いる並列性 (gang, vector) が指定されていればそれを用い，指定がない場合はコンパイラが自動で使用する並列性を決定する．ループ分割方法は gang の場合には空間局所性を高めるために block 分割を用い，vector の場合は隣接するスレッドが配列の隣接する要素にアクセスするコアレスドアクセスになりやすくするために

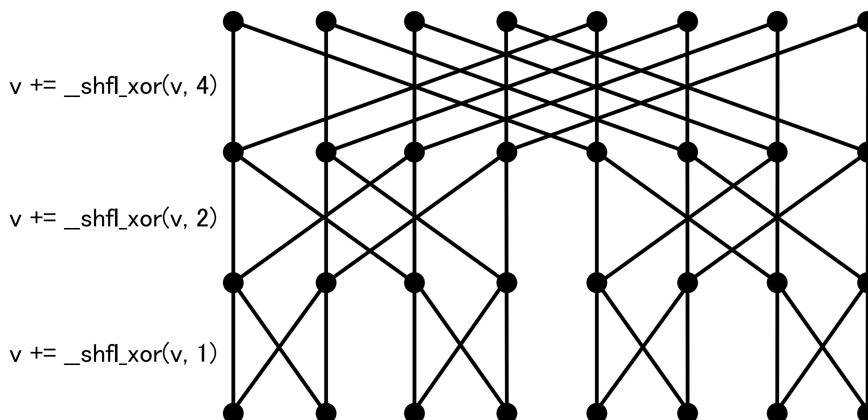


図 4.3: shuffle 命令を用いたバタフライリダクション（この図は 8 スレッド間で変数 v の和を求める場合）

cyclic 分割を用いる．さらに `loop` 指示文に `reduction` 節が付記されている場合は指定された変数用にプライベート変数を用意し，ループの終わりにリダクションして元の変数に書き込むように変換する．

vector reduction

通常は Shared メモリのみを用いてスレッド間で値を共有し，並列にリダクションを行う．一方，Kepler アーキテクチャ向けにコンパイルする際には shuffle 命令も利用する．shuffle 命令により同一ワープ内のスレッド間で値の受け渡しが可能であり，そのうち__shfl_xor() を使うことで図 4.3 に示すようなワープ内のリダクションが実現できる [18]．なお Kepler アーキテクチャでは __shfl_xor() は int と float 型しか用意されてないため，double 型では上位 32bit と下位 32bit に分割して交換する関数を用意した．スレッド間でのリダクションの流れは，まず shuffle 命令を用いて各ワープ内でリダクションし，部分結果を Shared メモリに書き込む．その後 1 ワープが部分結果を shuffle 命令でリダクションして全体の結果を求める．Kepler アーキテクチャでは 1 スレッドブロック当たりのワープ数は最大 32 個であるため部分結果のリダクションには 1 ワープ 32 スレッドで十分である．通常の Shared メモリのみを用いた方法に比べ，Shared メモリの読み書きとそれに伴うスレッド間同期が削減できるため，より高速なリダクションが可能となる．

gang reduction

CUDA ではスレッドブロック間で同期を取ることができないため，通常はブロックごとの部分結果をバッファに格納しておき，最後に別のカーネルでその結果をリダクションする．しかしながらバッファの確保が必要，リダクション用のカーネル起動のコストがかかる，最後のリダクション用カーネルは 1 ブロックで実行するため部分結果の数が多い場合は性能が落ちるなど，多くの問題を抱えている．Kepler アーキテクチャの GPU では同じグローバルメモリアドレスに対する atomic 演算が 1 クロック当たり 1 演算実行可能となり，Fermi アーキテクチャと比べて 9 倍も高速化されている [1]．そこで Kepler アーキテクチャ用にコンパイルする際には atomic 演算が提供されている int 型と float 型の一部のリダクション演算 (+, min, max 等) は，各ブロックの部分結果を atomic 演算でリダクション変数に直接リダクションするように実装した．

また atomic 命令を利用できないリダクションにおいてバッファ確保・解放のオーバーヘッドを削減するために，メモリプールを用いている．このメモリプールは OpenACC の `async` 節などで指定するキュー番号ごとに確保されている．カーネル起動の際にはそのカーネルを実行するキューに対応付けられたメモリプールの先頭から順にメモリを利用してゆき，カーネル実行後は何もする必要はない．キュー番号ごとに別のメモリプールを用いることで，同一のメモリプールを利用するカーネルは必ず 1 つだけとなるため，複雑な管理が不要となっている．またこのメモリプールはリダクション変数自体のデバイスメモリの確保にも利用できる．このメモリプールの大きさはデフォルトで 1 MB としている．スレッドブロック数は 65535 以下になるよう調整しているため double 型のリダクション変数 1 つあたり最大で，8 Byte × (1+65535)=512 KB 使用する．したがって double 型であれば最大 2 つのリダクション変数とそのバッファをメモリプールから確保できる．

ソースコード 4.5 に `loop` 構文例を，ソースコード 4.6 にその変換例を示す．_ACC_calc_niter() でループ長を求め，_ACC_init_thread_iter() で，そのスレッドが担当するインデックスの範囲を求める．その後ループ内では_ACC_calc_idx() により実際のループ変数の値を求め，ループ本体を実行する．またリダクション部分に関しては，始めに_ACC_init_reduction_var() によりスレッドローカル変数を初期化し，ループ実行後に_ACC_reduction_thread() によりスレッド間でローカル変数の値をリダクションする．

ソースコード 4.5: loop 構文の例

```
1 /* inside parallel region */
2 #pragma acc loop vector reduction(+:sum)
3 for(i = 0; i < N; i++){
4   a[i]++;
5   sum += a[i];
6 }
```

ソースコード 4.6: loop 構文の変換例

```
1  /* inside gpu kernel function */
2  int _niter_i, _idx, _init, _cond, _step, _red_sum;
3  _ACC_init_reduction_var(&_red_sum,0);
4  _ACC_calc_niter(&_niter_i, 0, N, 1);
5  _ACC_init_thread_iter(&_init,&_cond,&_step,_niter_i);
6  for(_idx = _init; _idx < _cond; _idx += _step){
7    int i;
8    _ACC_calc_idx(_idx, &i, 0, N, 1);
9    a[i]++;
10   _red_sum += a[i];
11 }
12 _ACC_reduction_thread(sum,_red_sum, 0);
```

4.1.5 評価

NVIDIA GPU 向け OpenACC コンパイラの基礎評価としてリダクションのマイクロベンチマーク，および NAS Parallel Benchmarks (NPB) のうち 5 つのカーネルベンチマーク CG, EP, IS, FT, MG で評

表 4.1: NVIDIA GPU 向け OpenACC コンパイラの評価環境

	K20-Opteron (Cray XK6)	K20-Xeon
CPU	AMD Opteron 6272 2.1 GHz (16 Core)	Intel Xeon E5-2609 2.4 GHz (4 Core)
Memory	DDR3-1600 32 GB	DDR3-1066 32 GB
GPU	Tesla K20 (GDDR5 5 GB, ECC on)	Tesla K20 (GDDR5 5 GB, ECC on)
OS	CNL (Compute Node Linux)	CentOS 6.4
Compiler	CCE 8.3.0, CUDA 5.5	GCC 4.4.7, PGI compiler 14.10, CUDA 5.5

価を行った．NPB の C 言語 OpenACC 版はソウル大学校が公開している SNU NPB Suite[19] に含まれる NPB3.3-SER をベースに実装し，さらに高速化のためにコードの一部を変更した．実装した Omni OpenACC compiler と Cray compiler 8.3.0，PGI compiler 14.10，OpenUH-OpenACC[20](2014 年 5 月 5 日時点) の性能を比較した．なお OpenUH-OpenACC は `parallel` 指示文の `async` 節など実装されていない機能があり，同一条件で比較できなかったためリダクションの性能のみを載せる．また GPU カーネルのデフォルトのスレッドブロック数がコンパイラごとに異なるため，`vector_length` 節で指定した．

評価環境を表 4.1 に示す．“K20-Opteron” は Cray XK6 であり Cray compiler が利用可能である．また “K20-Xeon” は通常の GPU クラスタで PGI compiler が利用可能である．本実装のコンパイルオプションには “-acc -O3” を指定し，nvcc に渡すオプションは “-O3 -arch=sm_35” とした．また K20-Opteron ではホストコードのコンパイルに Cray compiler を，K20-Xeon では gcc 4.4.7 を使用した．Cray compiler を用いた OpenACC のコンパイルのオプションには “-O3 -h acc_model=auto_async_none” を指定した．auto_async_none を指定することで非同期カーネル実行が自動で行われるのを抑止し，その際のオーバーヘッドを削減した．PGI compiler のコンパイルオプションには “-O3 -acc -ta=tesla:cc3+,cuda5.5,nordc -Msafeptr” を指定した．-Msafeptr によりポインタのエイリアスがないことをコンパイラに伝えている．OpenUH-OpenACC のコンパイラオプションには “-fopenacc” を用い，nvcc に渡すオプションは “-arch=sm_35” とした．最適化オプションは使用するとコンパイラが異常終了するので指定できなかった．また本実装と異なり CUDA ソースファイルにはデバイスコードしか含まれないため nvcc のオプションに “-O3” は付けなくても問題は無い．

リダクションベンチマーク

このリダクションのベンチマークでは 2^{20} 要素の int，float，double 型配列の値の加算を行う時間を計測する．配列要素のリダクションを行う 1 重ループに `parallel loop` 指示文を記述し，さらに各コンパイラでスレッド数などを同じにするため `num_gangs(4096) vector_length(256)` を指定した．配列はあらかじめデバイスに確保し初期化しておき，リダクション変数の確保・解放とメモリコピーを含んだ処理時間を計測した．図 4.4 に 2^{20} 要素の配列のリダクションの実行時間を示す．K20-Opteron において，Cray compiler と比較して 1.42–1.68 倍の性能が得られた．Cray compiler における性能低下の原因として，`parallel` 領域をリダクションも含めて 1 カーネルで実行している点があげられる．Cray

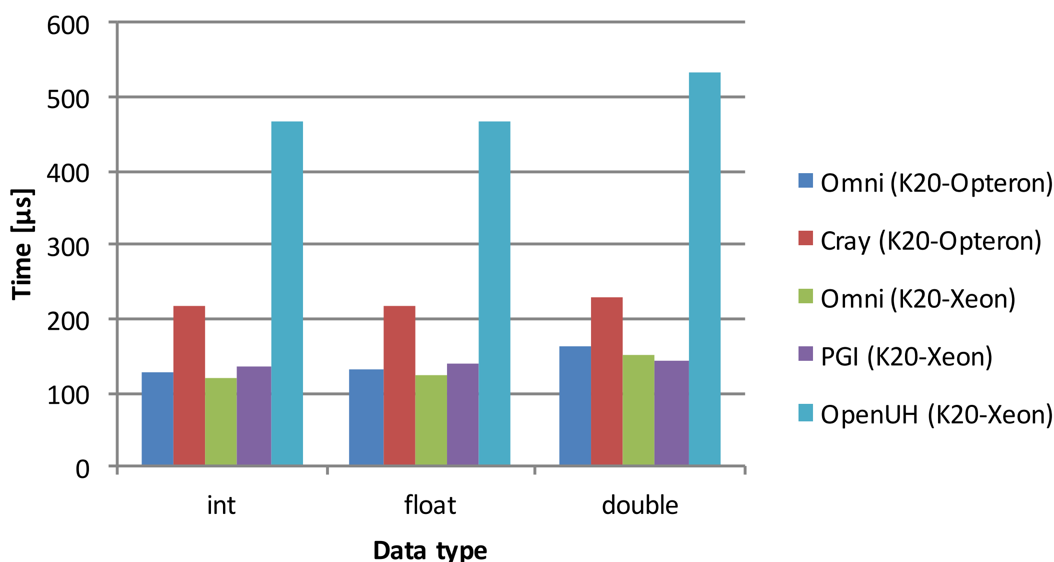


図 4.4: 2^{20} 要素のリダクションの実行時間

compiler におけるリダクションの方法は，(1) 各スレッドブロックは部分結果を一時配列の自分のスレッドブロックインデックスに書き込み，メモリフェンスにより書き込みの完了を保証する．(2) 事前に 0 クリアされたカウンタを atomic 演算でインクリメントし，インクリメント後の値が全スレッドブロック数よりも小さければ最後のスレッドブロックではないので何もせず，等しければ最後のスレッドブロックなので一時配列の部分結果をまとめて全体の結果を書き込む，となっていた．書き込みを完了してからカウンタを増やすためにフェンスによる待機や atomic 演算が必要になるため，スレッドブロック数に比例して実行時間が増加する．なお，リダクション変数やバッファの確保のためのメモリ確保は最初のカーネル実行でのみ行われていた．一方, Omni compiler においては部分結果のリダクションの部分は別のカーネルとして実行する．カーネル起動オーバーヘッドがあるが，リダクション以外のメインのカーネルの実行時間が長ければそのオーバーヘッドは隠蔽可能であり，定数時間であるためこちらのほうが実行時間が短くなりやすい．K20-Xeon において，PGI compiler と比較し int, float 型では 1.14 倍，double 型では 0.94 倍の性能が得られた．PGI compiler では Cray compiler と同じく一時結果を保持するためのメモリはキャッシュをしており，毎回確保はされていなかった．Shared メモリを用いた一般的な処理を行っており，shuffle 命令や atomic 演算は使用していなかったため，本実装の方がより高い性能を得られた．なお 2018 年 1 月時点で最新の PGI 17.10 においても同様に shuffle や atomic は利用されていなかった．OpenUH-OpenACC は PGI compiler と同じく Shared メモリを用いたリダクションを行っていたが，カーネル実行前後にバッファの確保・解放を行っていたため，時間が非常に増加している．

次に実装で行った最適化の効果を確認するために，K20-Xeon 上で最適化を順に適用した際の性能を図 4.5 に示す．“cache” は Read-Only Data Cache の利用，“mempool” はメモリプールの利用，“shuffle” は shuffle 命令の利用，“atomic” は atomic 演算の利用による最適化を表す．リダクションでは Read-Only Data Cache を使用すると逆に性能が低下した．リダクションでは同じデータは 1 度しか読まれずかつコ

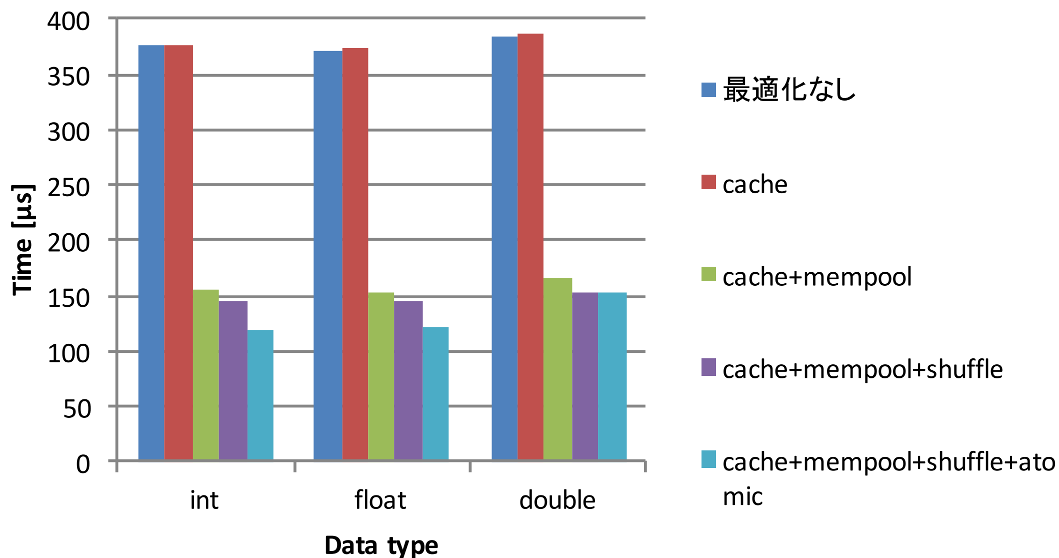


図 4.5: 最適化による 2^{20} 要素のリダクション時間の変化

アレスアクセスになるため，キャッシュする利点がなく逆にキャッシュするオーバーヘッドが増加したからである．次いでメモリプールの利用によりリダクション変数とバッファの確保時間が大幅に減少した．さらに shuffle 命令の利用ではすべての型で性能向上が確認できた．atomic 演算は int と float 型でのみ利用するようにしており，それぞれ性能が向上した．

CG

CG ベンチマークの性能を図 4.6 に示す．まず本実装での Kepler アーキテクチャ向け最適化の有無により 1.38–2.11 倍の性能向上が確認できた．最適化ありの本実装の性能は Cray compiler と比較して 0.97–1.04 倍でほぼ同等であった．クラス B にて Cray compiler より高い性能が出た理由は Cray compiler ではスレッドブロックで並列化するループ長が 65535 を超えたときスレッドブロック数を 65535 個に設定していたのに対して，本実装では 65535 以下の数になるようにループ長を割った数にしており，それによりブロック数が減少しスレッドブロックを実行するオーバーヘッドが減少したからである．

K20-Xeon においては PGI compiler と比較して 1.34–2.80 倍の性能となった．CG ベンチマークで最も時間のかかる疎行列ベクトル積のカーネルについて PGI compiler に “keep” オプションを付けて CUDA コードを出力したところ，const __restrict__修飾子や__ldg() によるロードはなく，Read-Only Data Cache が使用されていないことが分かった．また使用レジスタ数も使用レジスタ数を調べたところ本実装では 24 個のところ PGI compiler では 35 個と多く，1SMX で実行できるブロック数が少ないため SMX の使用率が低下していた．加えて gang で並列化するループ長が 65535 以上のとき，PGI compiler でも Cray compiler と同様にスレッドブロック数を 65535 個に設定していた．加えて Kepler アーキテクチャ向け最適化なしの状態でも PGI compiler より高い性能が得られていることから，本実装の方が NVCC による最適化の効果が大きいと言える．

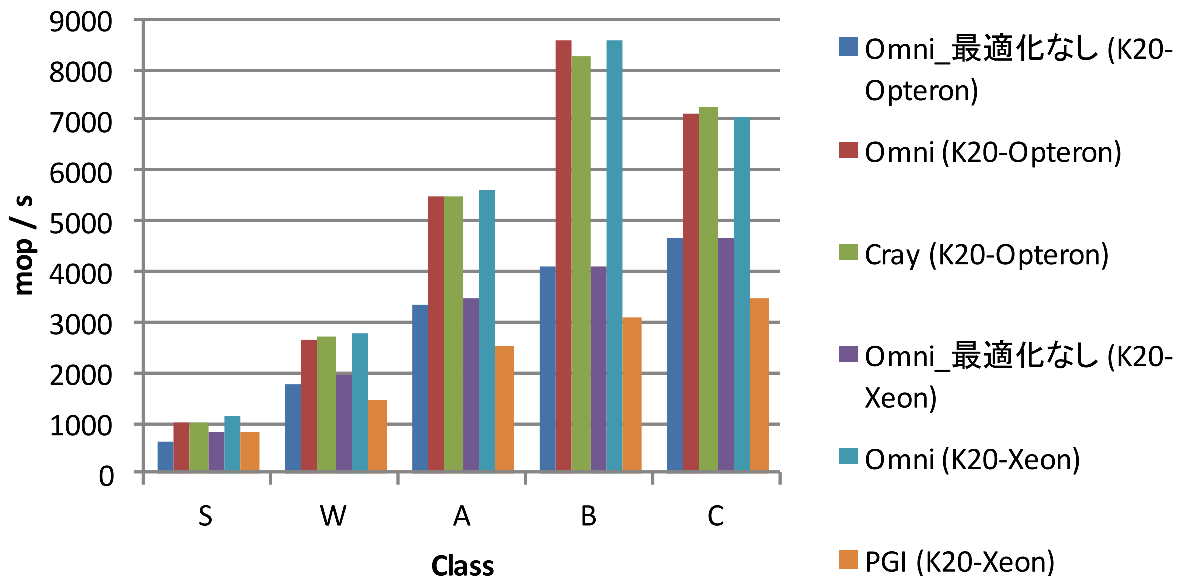


図 4.6: CG の性能

EP

EP ベンチマークの性能を図 4.7 に示す．本実装は Cray compiler と比較して 1.26–2.23 倍の性能が得られ，PGI compiler と比較すると 1.14–1.54 倍の性能が得られた．本実装ではすでにクラス B で性能が頭打ちになっているが，Cray compiler と PGI compiler ではクラス C まで性能が向上している．

Cray compiler での性能低下の原因を調査したところ，プロファイラの情報から Cray compiler でコンパイルしたバイナリでは倍精度演算数が 1.78 倍になっており，アセンブリをみると最内ループにおいて計算結果をレジスタに保持せず，同じ計算を 2 度実行している部分が多くあった．この差が起きた理由としては，本実装でコンパイルに用いる NVCC は内部ではコードを最適化した後，PTX と呼ばれる疑似アセンブリ言語に変換しそれを PTXAS (PTX optimizing assembler) で最終的に機械語に変換するのに対して，Cray compiler はコンパイラで最適化を行い直接 PTX コードを出力し，それを PTXAS で機械語に変換するため，PTX を出力する前までの最適化に違いがあるからである．この場合は CUDA コンパイラの方がよい最適化を行ったと言える．

また PGI compiler の場合は，実行された倍精度演算の命令数に大差はなかったが，プロファイラで調べると instruction per clock (IPC) が低下しており，クラス S では本実装の 0.69 倍，クラス B では 0.87 倍となっていた．本実装と PGI compiler の性能比はおおよそ IPC の比と近似することから IPC の差が性能に影響していると言える．特に命令間の依存によるストールの発生率が例えばクラス B で本実装は 49% なのに対して PGI compiler は 74% となっていた．PGI compiler は本実装と同じく CUDA コードへの変換を行い NVCC によるコンパイルを行っているが，最適化により元のコードから大きく変更されており NVCC が最適化を行いにくくなったと考えられる．

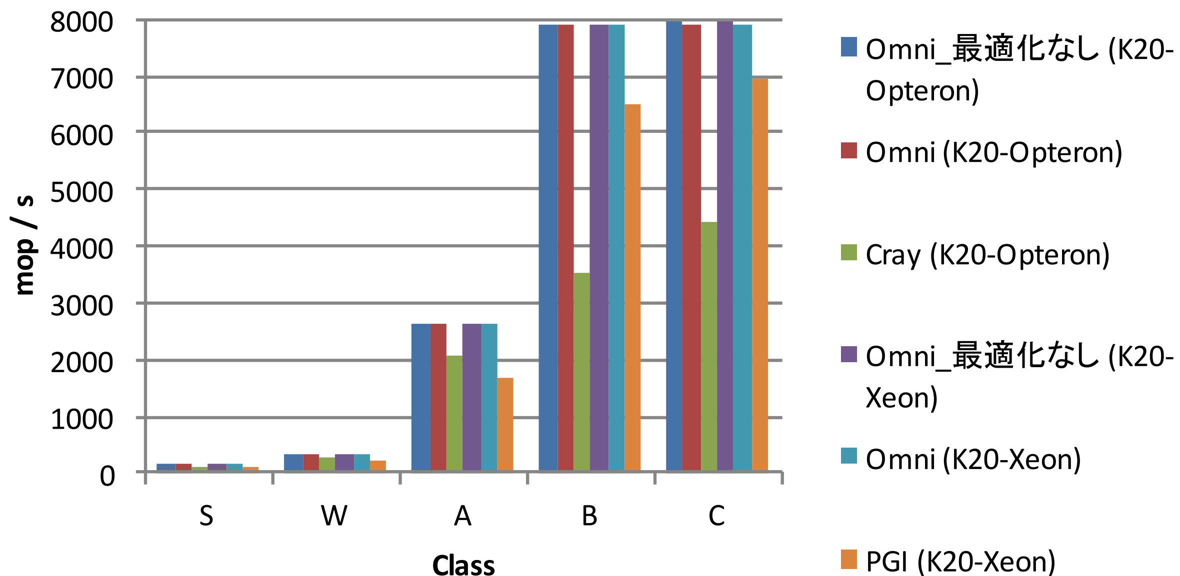


図 4.7: EP の性能

IS

IS ベンチマークの性能を図 4.8 に示す．評価時点では OpenACC 2.0 の機能である `atomic` 指示文に対応していなかったためヒストグラム計算は CPU で実行し，prefix sum のみオフロードするコードを用意した．そのため実行時間の 90% 以上が CPU での計算となり，全体的に他のベンチマークより性能が低下した．K20-Opteron においては，Cray compiler と比較しクラス S では 1.05 倍の性能となったがそれ以外では同等の性能が得られた．K20-Xeon においては，PGI compiler と比較し 0.90–0.97 倍の性能となった．PGI compiler より性能が低下していた原因としてホストコードのコンパイラの違いが考えられる．本実装は K20-Opteron 上ではホストコードのコンパイラに同じ Cray compiler を用いたが，K20-Xeon 上では GCC を用いたことでホストコードの性能差が生まれ，それが直接性能に表れている．

FT

FT ベンチマークの性能を図 4.9 に示す．K20-Opteron においては，全てのクラスで Cray compiler の 0.66–0.86 倍であり低い性能となった．K20-Xeon においては，PGI compiler はクラス S のみ正しく動作し，それ以外はメモリアクセスエラーによりプログラムが終了した．こちらも同様に PGI compiler の 0.78 倍の性能しか得られなかった．FT における性能低下の原因は gang private 配列用のメモリ確保・解放と非効率なカーネルである．問題サイズが小さい時は前者が，問題サイズが大きい時は後者が主な原因となっていた．FT では FFT のカーネルに各 gang でプライベートに持つよう指定された配列がある．配列サイズが小さければ Shared メモリを利用することができるが，Shared メモリを超える大きさの場合にはグローバルメモリを代わりに使うことになる．FT の場合，クラス S においても各 gang に 67584 byte 必要になるため Shared メモリでは足りずグローバルメモリを使わなければいけない．Cray compiler では

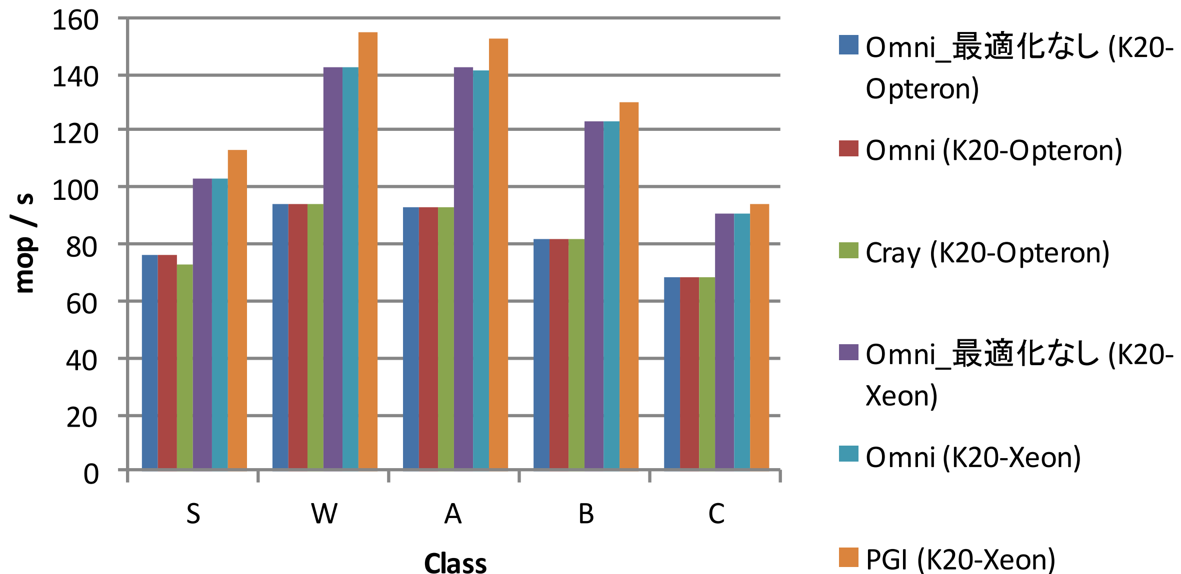


図 4.8: IS の性能

各カーネルの最初の実行時に確保したグローバルメモリをその後の実行で再利用していたため頻繁な確保・解放は行われていなかった．一方本実装はカーネル実行毎に確保・解放を行っていたため，そのオーバーヘッドが発生した．

次に非効率なカーネルとなった原因であるが，CG，EP，IS ではループのネストは単純な 1 重，2 重ループであったのに対して，FFT のカーネルでは gang で並列化するループの内に複数の vector で並列化するループがありカーネル内で vector-single モードと vector-partitioned モードが切り替わる．本実装では vector-single モードをモデルと同じく 1 スレッドで実行するよう変換を行う．例えば vector-single モードにて変数が宣言されて値が代入される場合，Shared メモリ上に変数を確保し 1 スレッドのみがその変数へ書き込みを行う．そのため Shared メモリメモリへのアクセスが増え，モードの切り替え時にはスレッド間の同期が必要になる．一方で，Cray compiler では変数は各スレッドのローカルなレジスタとして保持しておき，vector-single モードにも関わらずすべてのスレッドで読み書きを行う．処理結果はこちらでも正しくなるため問題はなく，Shared メモリへのアクセスやスレッド間での同期が減少し性能が向上していると考えられる．

MG

MG ベンチマークの性能を図 4.10 に示す．K20-Opteron においては Cray compiler と比較し，0.68–0.90 倍の性能しか得られなかった．K20-Xeon においては，PGI compiler では結果不正が起きたため，評価できなかった．MG での性能低下の主な原因は gang private 配列の確保・解放にある．Cray compiler ではクラス S では Shared メモリを，それ以外ではグローバルメモリを使用していたが各カーネルの最初の実行時にのみ確保していた．そのため非同期カーネル実行を連続して行うことができた．一方本実装では毎回カーネル実行前後に確保・解放したため，GPU メモリを解放する cudaFree 関数で GPU との同期が発

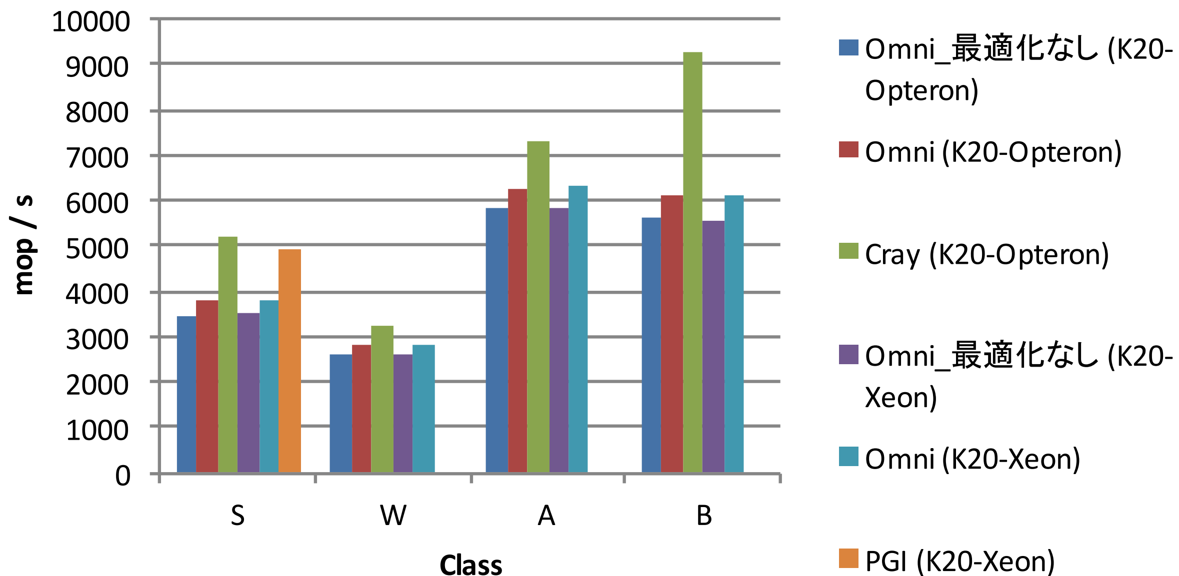


図 4.9: FT の性能

生し，効果的にカーネルを起動できなかった．

4.2 PEZY-SC 向け OpenACC コンパイラ

NVIDIA GPU 向けの Omni OpenACC compiler を拡張し，アクセラレータとして PEZY-SC を利用可能とした．PEZY-SC のプログラミングには PZCL を用いるため，トランスレータで出力するカーネルコードを PZCL 向けに変更し，ランタイムを PZCL で実装した．コンパイル処理は NVIDIA GPU 版 (図 4.1) における CUDA を PZCL で置きかえたものとなる．

4.2.1 NVIDIA GPU 向け実装からの変更点

ホストコード

基本的にはホストコードから呼び出される Omni OpenACC runtime において CUDA で記述していた部分を OpenCL(PZCL) で記述することで対応した．またその際に，ランタイム内部で CUDA や OpenCL に依存しない部分を切り分けることで最小限の実装で CUDA と OpenCL のどちらでも対応可能としている．PEZY-SC に向けて特に変更したのはカーネルの起動部分で，PEZY-SC では global_work_size を 128 の倍数かつ 8192 以下になるよう調整し，local_work_size は 8 で固定としている．

デバイスコード

OpenACC の並列性と PEZY-SC の並列性の対応関係は，現在の実装では gang を PE に，worker を 1 で固定し，vector を PE 内のスレッドに割り当てている．このマッピングは NVIDIA GPU 向けの実装を

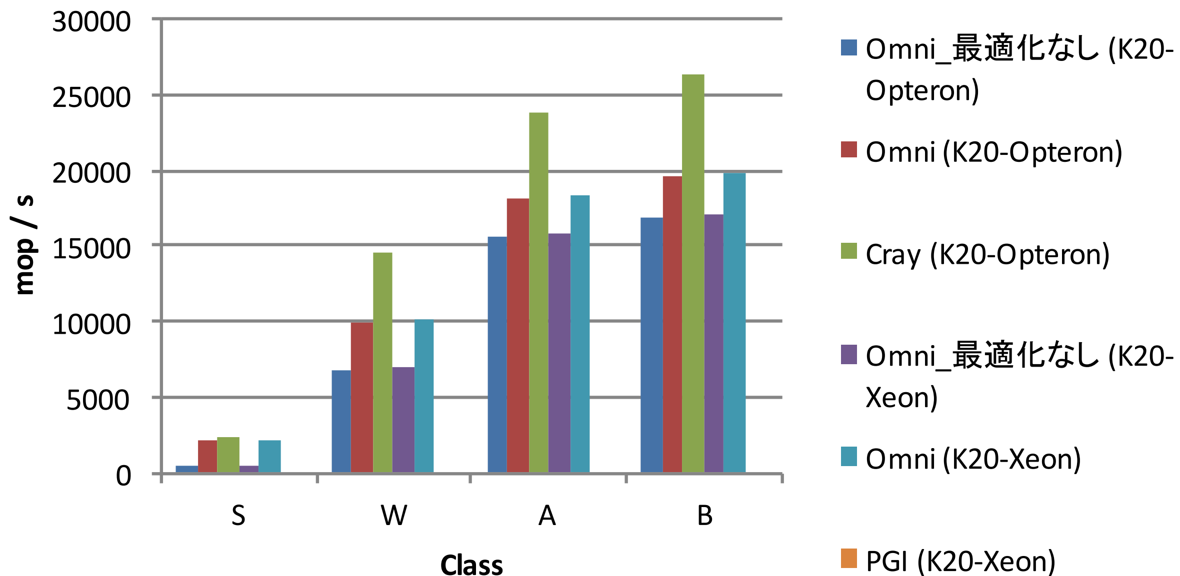


図 4.10: MG の性能

ほぼそのまま使用できるという理由で用いているが，最適なマッピングについては検討の余地がある．PEZY-SC のプログラミングを行う上で NVIDIA GPU と最も異なる点が，階層的なキャッシュにコヒーレンシがなく，プログラマが適宜フラッシュを行わなければならない部分である．ループのワークシェアリングに関して考えると，gang すなわち PE での並列化を行った後に各 PE が書き込んだデータをすべての PE で同じように読み出すためにはすべての階層のキャッシュをフラッシュしなければならないため，flush() を実行する (flush() を実行すると同期も行われるため sync() は必要ない)．vector すなわちスレッドでの並列化のあとに，PE 内の各スレッドが書き込んだデータを PE 内の 8 つすべてのスレッドで同じように読み出せるようにする必要があるが，1 PE 内の全スレッドは同じキャッシュを用いるためコヒーレンシに関して特に何もする必要はなく，スレッド間の同期のみで良い．またカーネル内で呼び出されるループのワークシェアリングする際の担当範囲計算やリダクションのための関数を PZCL の組み込み関数を用いるように書き換えている．

4.2.2 最適化

PEZY-SC 向けに最適化するためコード生成の改善や拡張機能の実装を行った．1 つめはカーネルマージである．これは OpenACC `kernels` 指示文における最適化である．`kernels` 指示文は対象領域のコードをコンパイラが適宜分割していくつかのカーネルとして実行するものである．NVIDIA GPU 向けの実装ではカーネル実行中に全スレッドでの同期を取ることができないため，`kernels` のコードは gang すなわちスレッドブロック間での並列ループの単位で分割して複数のカーネルとして実行するようにしている．一方で，PEZY-SC ではプロセッサの全スレッドで同期をとる命令 (sync()) があるため，`kernels` の対象領域を 1 つのカーネルとして変換することが可能である．これによりカーネル起動のオーバーヘッ

```
#pragma acc yield
#pragma acc sync [( int-expr )]
#pragma acc flush [( int-expr )]
```

図 4.11: yield, sync, flush 指示文の構文

ドを最小限にすることができる．

2 つめは明示的なスレッド切り替え，同期，フラッシュのための指示文の追加である．追加した `yield`,`sync`,`flush` 指示文の構文を図 4.11 に示す．PEZY-SC では 1PE 内のスレッドは実際には 4 つの表裏のスレッドペアであり，通常は表裏のどちらかしか実行されず，同期命令やスレッド切り替え命令に到達した際に反対側が実行される．明示的にスレッドの表裏を切り替えることにより，キャッシュ利用率向上やメモリアクセスレイテンシの隠蔽が可能である．`yield` 指示文によりユーザがスレッドを切り替えるタイミングを指定できるようにした．また MIMD ではあるが同期をとって動作したほうが効率よくキャッシュを利用できることがあるのでユーザが任意のタイミングで同期を取れるように `sync`, `flush` 指示文を追加している．`sync` と `flush` には引数として整数値を指定可能で，同期・フラッシュ時のキャッシュレベルを指定するものである．例として “#pragma acc sync(2)” と指定した場合は L2 キャッシュがある City レベルでの同期 sync_L2() が実行される．引数を指定しない場合は全体での同期・フラッシュとなる．

4.2.3 評価

評価には N-body と NPB-CG ベンチマークを用いた．N-body は相互作用する粒子の動きをシミュレーションするもので，単精度浮動小数点を用いて全粒子ペアの相互作用を計算するナイーブな実装である．NPB-CG は正値対称な大規模疎行列の最小固有値を共役勾配法で求めるベンチマークである．PZCL 版と OpenACC 版を実装し，性能と生産性について比較する．

性能評価

性能評価のために高エネルギー加速器研究機構 (KEK) の青睡蓮を利用した．そのノード構成とソフトウェアを表 4.2 に示す．PEZY-SC における N-body の性能を図 4.12 に示す．凡例の “PZCL (base)” は特に最適化をしていない PZCL コード，“PZCL (opt. 1)” はカーネルマージ (opt. 1) を加えた PZCL コード，“PZCL (opt.1+2)” はさらに明示的スレッド切り替え (opt. 2) を加えたコード，“OpenACC (base)” は特に最適化を加えていない OpenACC コードである．性能の Flops は 1 相互作用計算に 38 FLOP を要すると仮定して算出している [21]．OpenACC 版の性能は PZCL 版の 97.8–100.0% であった．このベンチマークにおいては相互作用の計算時間が支配的であり，カーネルマージと明示的スレッド切り替えの効果はほぼなかった．

次に PEZY-SC における NPB-CG ベンチマークの性能を図 4.13 に示す．OpenACC 版の性能は PZCL(base) 版の 91.9% 以上であった．OpenACC 版ではループ分散やランタイムのオーバーヘッドおよび不必要なデータ転送があり特に Class A で性能が低下している．OpenACC ではリダクションの対

表 4.2: 青睡蓮のノード構成とソフトウェア

	青睡蓮 (Suiren Blue)
CPU	Intel Xeon-E5 2618Lv3 2.3 GHz
Memory	DDR4 64 GB, 1866 MHz
Accelerator	PEZY-SC×4
Peak perf.	SP: 3 TFlops, DP: 1.5 TFlops
Memory	DDR4 16 GB, 1333 MHz, 85.3 GB/s
Compiler	ICC 14.0.2, PZSDK 2.1,
	Omni OpenACC compiler 0.9.3 for PEZY-SC

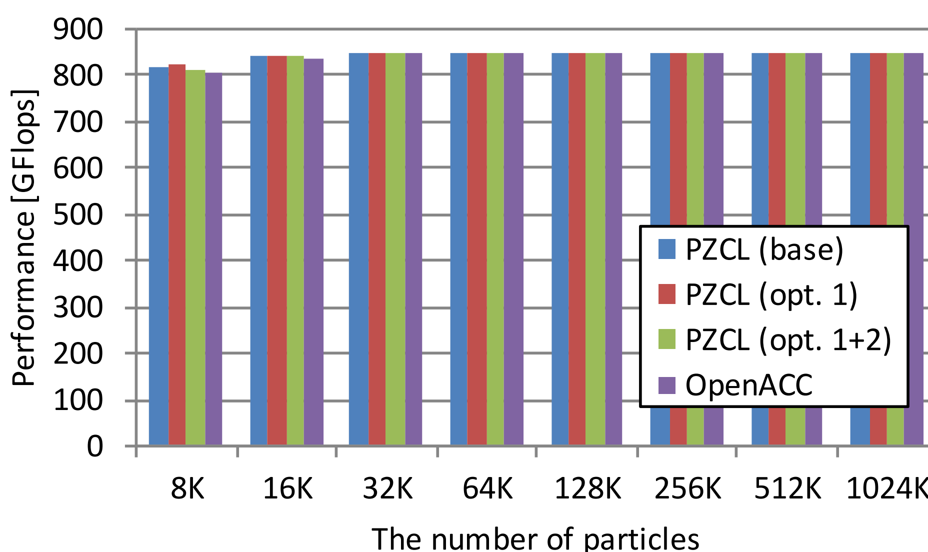


図 4.12: PEZY-SC における N-body ベンチマークの性能

象の変数は最初にホストからデバイスに転送されるが，初期値がゼロの場合には実際には不必要であるためその部分でオーバーヘッドが生じている．カーネルマージによる最適化はカーネルオーバーヘッドの割合が大きくなる疎行列サイズが小さな時に特に効果が大きかった．明示的スレッド切り替えによる最適化は逆に行列サイズが大きな時に効果が大きかった．これはスレッドを切り替えることでキャッシュが追い出される前に逆側のスレッドがキャッシュを利用できたからである．最終的に PZCL(opt.1+2) 版と比べて OpenACC 版は 61.6–87.5% の性能であった．

その後 OpenACC に `kernels` 指示文においてカーネルを 1 つにまとめる最適化 (opt.1 に相当)，および拡張の `yield` 指示文 (opt.2 に相当) を導入して NPB-CG の評価を再度行った．なお青睡蓮の構成が変わったため，再度そのノード構成とソフトウェアを表 4.3 に示す．図 4.14 に再度評価した NPB-CG の性能を示す．凡例の “OpenACC (opt. 1)” が OpenACC コードにおいてコンパイラでカーネルマージを行ったもの，“OpenACC (opt. 1+2)” が OpenACC コードにおいてコンパイラでカーネルマージを行いか

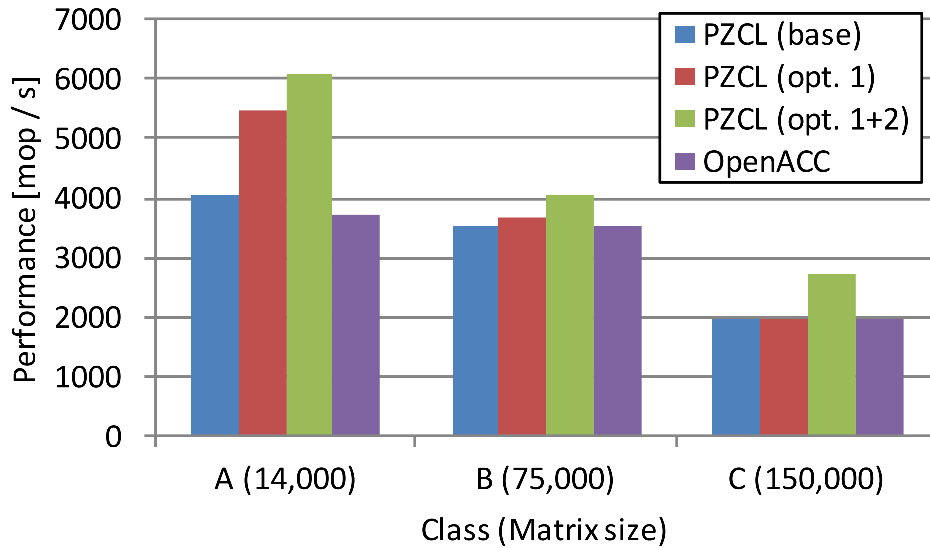


図 4.13: PEZY-SC における NPB-CG ベンチマークの性能

表 4.3: 青睡蓮のノード構成とソフトウェア

	青睡蓮 (Suiren Blue)
CPU	Intel Xeon-E5 2618Lv3 2.3 GHz
Memory	DDR4 64 GB, 1866 MHz
Accelerator	PEZY-SCnp×4
Peak perf.	SP: 3 TFlops, DP: 1.5 TFlops
Memory	DDR4 16 GB, 1333 MHz, 85.3 GB/s
Interconnect	InfiniBand: Mellanox Connect-X3 FDR
Compiler	ICC 14.0.2, PZSDK 3.0, Omni OpenACC compiler 1.1.2 for PEZY-SC

つ `yield` 指示文でスレッド切り替えを指定したものである．Class A において，PZCL と OpenACC のそれぞれのバージョンを比較すると OpenACC 版は 88–91% の性能であった．どのバージョンも同じように性能が落ちており，OpenACC の変換によるオーバーヘッド，余計なデータ転送で遅くなっていると考えられる．しかしながら，Class C においては PZCL と OpenACC のどのバージョンも OpenACC 版で 99% 以上の性能が出ている．OpenACC においても PZCL と同様の最適化が適用されてかつ演算時間の比率が高くオーバーヘッドが見えなくなっているからである．

また比較として NVIDIA Tesla K20X における OpenACC 版の性能を評価した．評価は筑波大学計算科学研究センターの HA-PACS/TCA [22] で行った．表 4.4 にノード構成とソフトウェアを示す．N-body, NPB-CG の OpenACC コードの K20X と PEZY-SC における性能を図 4.15 と 4.16 に示す (PEZY-SC の性能は OpenACC(opt.1+2) を再掲)． K20X においては，PGI compiler と Omni OpenACC compiler の 2

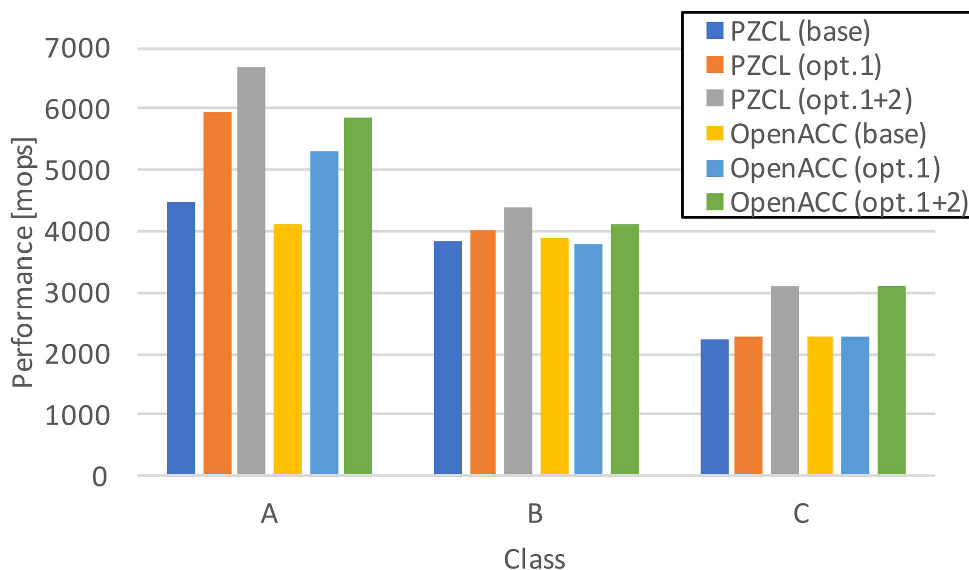


図 4.14: PEZY-SC における NPB-CG ベンチマークの性能 (OpenACC に最適化を追加)

表 4.4: HA-PACS/TCA のノード構成とソフトウェア

	HA-PACS/TCA
CPU	Intel Xeon-E5 2680v2 2.8GHz×2
Memory	DDR3 128GB, 1866MHz
Accelerator	Tesla K20X×4
Peak perf.	SP: 3.95 TFlops, DP: 1.31 TFlops
Memory	GDDR5 6GB, 250 GB/s
Interconnect	InfiniBand: Mellanox Connect-X3 FDR (PCIe Gen.2×8 接続)
Compiler	PGI 15.10, CUDA 7.5,
	Omni OpenACC compiler 0.9.3

つを用いて計測した．N-body ベンチマークにおいて，K20X では Omni は PGI の半分以下の性能しか得られなかった．これは粒子データ (float 型 4 要素から成る構造体) を PGI では 128bit 長でベクトルとしてロードしていたのに対して，Omni では 32bit 長でスカラとしてロードしていたためである．PGI を用いた K20X の性能と比較すると，Omni を用いた PEZY-SC の性能は 23–29% しか出ていない．N-Body では粒子間の距離から相互作用の力を求める際に平方根の逆数の計算が必要となるが，SFU が K20X では全体で 448 個 (32 units/SMX × 14 SMX) あるのに対して PEZY-SC では 64 個 (1 unit/city × 64 city) しかないため，その部分で性能が律速されていると言える．NPB-CG ベンチマークにおいては，K20X では Omni のほうが PGI よりも高い性能が出ており，これは CUDA コンパイラの最適化によるものである．Omni を用いた K20X の性能と比較すると，Omni を用いた PEZY-SC の性能は 26–64% であった．PEZY-SC はデバイスメモリのバンド幅が低いのが原因と考えられる．

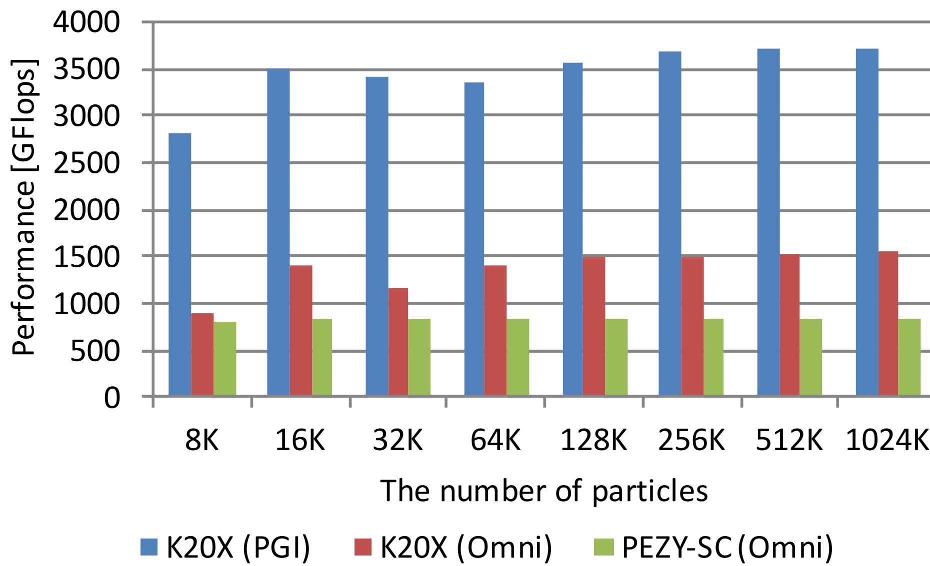


図 4.15: K20X と PEZY-SC における OpenACC 版 N-body ベンチマークの性能

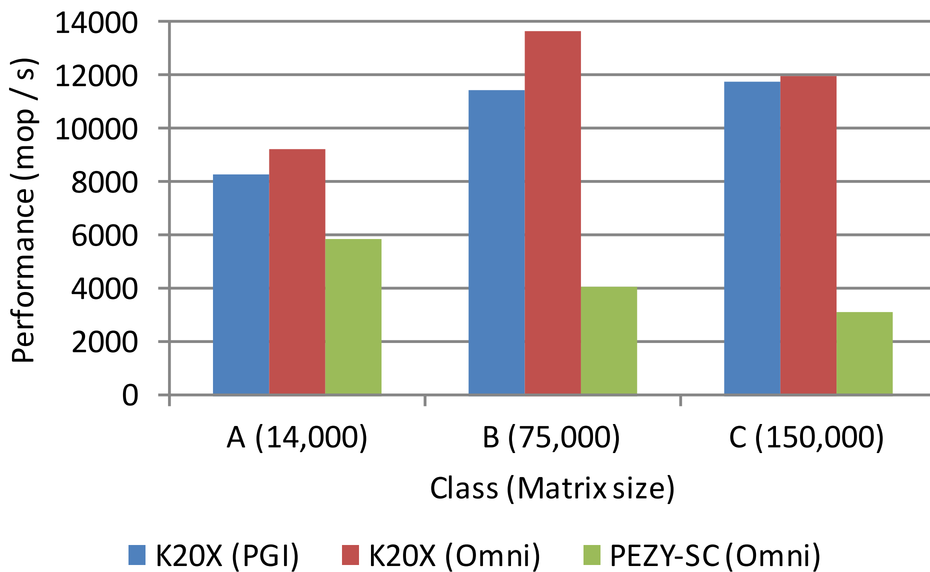


図 4.16: K20X と PEZY-SC における OpenACC 版 NPB-CG ベンチマークの性能

生産性評価

PZCL では OpenCL と同様にデバイスメモリの管理やカーネルの実行をプログラマが多くの複雑な API を用いて記述する必要があり，かつカーネルにおける並列処理は PZCL 独自の方法で書く必要がある．一方で，OpenACC ではプログラマは逐次コードに指示文を追加するのみで非常に簡易にアクセラレータにオフロードと並列化することを指定可能である．また OpenACC は標準仕様であり，対応するコンパイラ

表 4.5: N-Body と NPB-CG の SLOC. (内数は指示文の行数)

	N-Body	NPB CG
Serial	109	418
PZCL (opt.1+2)	240	1001
OpenACC	114 (5)	447 (25)

があれば他のアクセラレータでも実行可能である．

生産性を定量的に評価する指標の一つとして Source Lines Of Code (SLOC) を計測した．N-Body と NPB-CG の PZCL 版と OpenACC 版の SLOC を表 4.5 に示す．OpenACC 版の SLOC は PZCL 版に対して，N-Body で 48%，NPB-CG で 45% に抑えられており，かつ逐次版とほとんど差がない．逐次版からほぼ指示文の追加で記述できていることから OpenACC の生産性が PZCL よりも高いことは明らかである．

4.3 関連研究

CPU 向けの OpenMP から CUDA へ変換を行うコンパイラである OpenMPC[23] が提案されている．既存の OpenMP コードをそのまま利用して NVIDIA GPU 上で動作させることが可能である．さらに独自の指示文を使ったチューニングや，パラメータ探索によるブロックサイズなどのチューニングが可能である．また独自の指示文により CUDA をプログラミングできる hiCUDA[24] が提案されている．hiCUDA はループの分散方法 (スレッドブロックに分散するかスレッドに分散するか) や，スレッドブロック数やスレッド数などを明示しなければならない．そのため CUDA のプログラミングに近く CUDA と同等の性能を得やすいが，CUDA に依存しており NVIDIA GPU 以外での利用は難しい．

OpenACC コンパイラでは accULL[25], OpenUH-OpenACC[26], OpenARC[27], RoseACC[28] がある．また GCC でも OpenACC への対応が進められている [29]．accULL は最初のオープンソース OpenACC コンパイラであり，Python ベースの YaCF というソース・プログラム変換を行うコンパイラフレームワークを用いてコードを最適化し C を CUDA または OpenCL コードに変換する．現在は単純なループにしか対応しておらず，コンパイルできるプログラムには制限がある．OpenUH-OpenACC はコンパイラフレームワーク OpenUH を用いた OpenACC コンパイラで C から CUDA コードへの変換を行う．CUDA のブロック及びスレッドへの柔軟なループのマッピングや NVIDIA GPU 向けのリダクションの汎用な実装に関して研究されている [30]．多くのプログラムをコンパイル可能であるが，公開されているものは GPU カーネルをホストと非同期に実行するための “async” 節に対応しておらず，プログラムの高速化が難しい．OpenARC はアクセラレータ向けコンパイラフレームワークであり，オープンソースでは初めて OpenACC1.0 の全機能をサポートしている．OpenACC の C/C++ コードを CUDA や OpenCL に変換する．Cetus compiler infrastructure を元に作られており，様々なコード解析や変換およびオートチューニング機能が備わっているしかしながら 2018 年 1 月現在においてもクローズドベータであり一般にはコードが公開されていない．RoseACC はオープンソースコンパイラである Rose Compiler ベースの OpenACC コンパイラで，C から OpenCL コードへの変換を行う．GCC では C/C++/Fortran から NVIDIA GPU 向

けに PTX に変換する実装が行われている．これら OpenACC コンパイラは本研究とほぼ同時期もしくは本研究より後に行われている．

本実装で評価に用いる NAS Parallel Benchmarks については，Seo らにより OpenCL による実装や最適化が行われている [31]．OpenACC による実装が PathScale により行われたが，PGI compiler でコンパイルすると EP，SP 以外のベンチマークでコンパイル時エラーや実行時エラーが発生する [32]．また Xu らにより NPB のすべてのベンチマークが OpenACC で実装されている [33]．本研究では独自に OpenACC で実装して評価に用いた．

OpenMP は version 4.0 からアクセラレータへのオフロードを `target` 指示文によりサポートしている．OpenMP と OpenACC のオフロードの指示文はよく似ているように見えるが，並列化の指定の仕方が大きく異なる．OpenMP はどこで並列領域を作りどのループをワークシェアリングするかをプログラマが記述し，その通り並列化される．明示的であるためプログラマが制御しやすいが，アクセラレータごとに並列数や並列性の階層が異なるため可搬性には難がある．OpenACC はどのループを並列化できるかを指定し，コンパイラがアクセラレータに適した並列化を行う．当然 gang,worker,vector のどれで並列化するかを指定可能だが必須ではなく，そもそもそれらがハードウェアにどう対応付けられるかはコンパイラ依存である．したがってアクセラレータの並列性に依存しないため可搬性は高まるが，コンパイラの実装によっては性能の低下がありうる．

第 5 章

XcalableACC コンパイラの設計・実装

本章では，演算加速機構を持つ並列システム向け PGAS 言語 XACC のコンパイラを設計・実装し，ベンチマークにより評価する．グローバルビューモデルの通信指示文を NVIDIA GPU と PEZY-SC 向けに，ローカルビューモデルの coarray を NVIDIA GPU 向けに実装し，Himeno benchmark と NPB CG により性能と生産性を評価する．

5.1　設計

Omni XACC compiler は XACC コードから OpenACC コードへ変換する source-to-source コンパイラである．OpenACC コードに変換する理由は，アクセラレータコードの生成を OpenACC コンパイラに任せることで可搬性の向上と実装の簡略化が可能であるからである．Omni XACC compiler は理研 ACIS と筑波大学で開発している XMP のリファレンス実装である Omni XMP compiler[17] の拡張であり，現在はグローバルビューモデルの指示文を C と Fortran で，ローカルビューモデルの coarray を C のみで対応している．

図 5.1 にそのコンパイル処理の流れを示す．XACC トランスレータは入力の XACC コードを XACC ランタイム呼び出しが含まれた OpenACC コードに変換する．グローバルビューモデルに対する変換では指示文で分散が指定された配列やループを変換し，通信指示文を通信コードに置き換える．この際，複雑な処理の多くは XACC のランタイム呼び出しに変換する．ローカルビューモデルに対する変換では，coarray を通常の配列に変換して通信部分を XACC ランタイムライブラリ呼び出しに置き換える．グローバルビューとローカルビューモデルの変換は基本的に独立して行われる．XACC コードに含まれる OpenACC 指示文は，変換に伴う変数のリネームや coarray 確保部分で変化はあるものの，基本的には元の

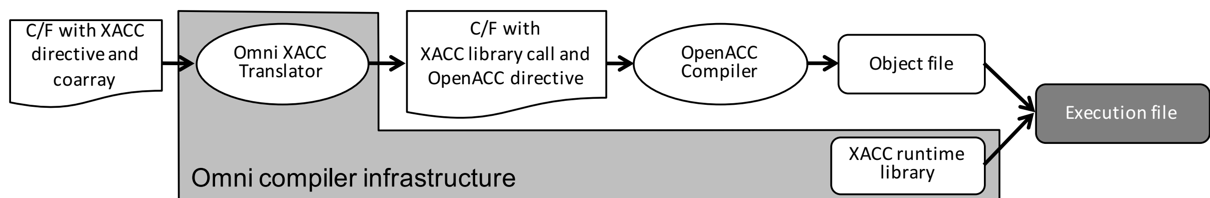


図 5.1: Omni XACC compiler におけるコンパイル処理の流れ

コードと同じ OpenACC 指示文が出力される．出力された OpenACC コードは OpenACC compiler (Omni OpenACC compiler, PGI compiler, Cray compiler, 等) でコンパイルし，Omni XACC ランタイムライブラリとリンクする．XACC ランタイムライブラリには XMP および XACC のランタイムルーチンが含まれる．XACC の通信ライブラリは NVIDIA GPU 向けと PEZY-SC 向けの 2 種類がある．NVIDIA GPU 向けの実装は MPI と CUDA により記述している．GPU メモリの転送には基本的には CUDA メモリの転送に対応した MPI の機能を利用している．CUDA に対応した MPI として MVAPICH2[34], OpenMPI[35] 等があり，これらの MPI 実装では MPI 関数の引数に GPU メモリアドレスを直接指定してホストメモリと同じように通信することが可能であり，内部的にホストバッファ経由もしくは GPUDirect RDMA によって直接通信を行う [36]．なお `reflect`，`reduction` 指示文など一部の通信指示文は CUDA に対応していない MPI でも利用できるようにホストメモリ経由で通信する実装もある．また不連続データの通信に対しては CUDA カーネルによるパックやアンパックをして転送するようにしている．さらに，coarray 機能の実装には MPI 3.0 から導入されたリモートメモリアクセス (RMA) 機能を用いている．現在多くの MPI 実装では MPI 3.0 に対応しており，MVAPICH2, OpenMPI は GPU メモリに対してもこの機能を利用可能である． PEZY-SC 向けの実装には MPI と PZCL を用いている．MPI には PEZY-SC のメモリ転送機能がないため，PZCL と MPI を組み合わせてホストメモリを経由して通信する．PEZY-SC 向けの実装は現在 `reflect` 指示文のみ対応している．

5.2　実装

5.2.1　reflect 指示文

XACC の `reflect` 指示文では，ある分散配列の袖交換時に通信バッファの確保や MPI の persistent 通信の登録等を行い，次回以降同じ分散配列に対して同じ設定で実行される `reflect` 指示文があった際には登録された設定を使い回すようにしている．XACC では加えて，通信の登録のみ行う `reflect_init` 指示文と，登録された袖通信を行う `reflect_do` 指示文に分けて記述することも可能であり，予期しない通信の再初期化や登録を防ぐことができる．村井らによって XMP における `reflect` 指示文の効率的な実装が行われており，不連続となる次元の通信においても，通信はブロックストライド (ブロック長・ストライド長・個数で表現可能な不連続領域) として表現できることが示されている [37]．本実装では，不連続な次元の送信時には XACC ランタイムライブラリ内の CUDA カーネルによりブロックストライドを GPU メモリ上でパックして連続領域を送受信するようにしている．通信は MPI_Send_init/Recv_init() により persistent 通信として登録を行い，MPI_Startall()/Waitall() により登録された通信を実行する．PEZY-SC 向けの実装においては PEZY-SC 上でのカーネルによるパック・アンパックに加えてホストメモリのバッファを介した通信するようにしている．

5.2.2　reduction 指示文

`reduction` 指示文の実装には XMP と同様に MPI_Allreduce() を用いた．XMP および XACC の `reduction` 指示文では，あるアドレスのデータをリダクションした結果を同じアドレスに保存しなければならないので，MPI_Allreduce() を呼び出す際には送信バッファに MPI_IN_PLACE を指定する．

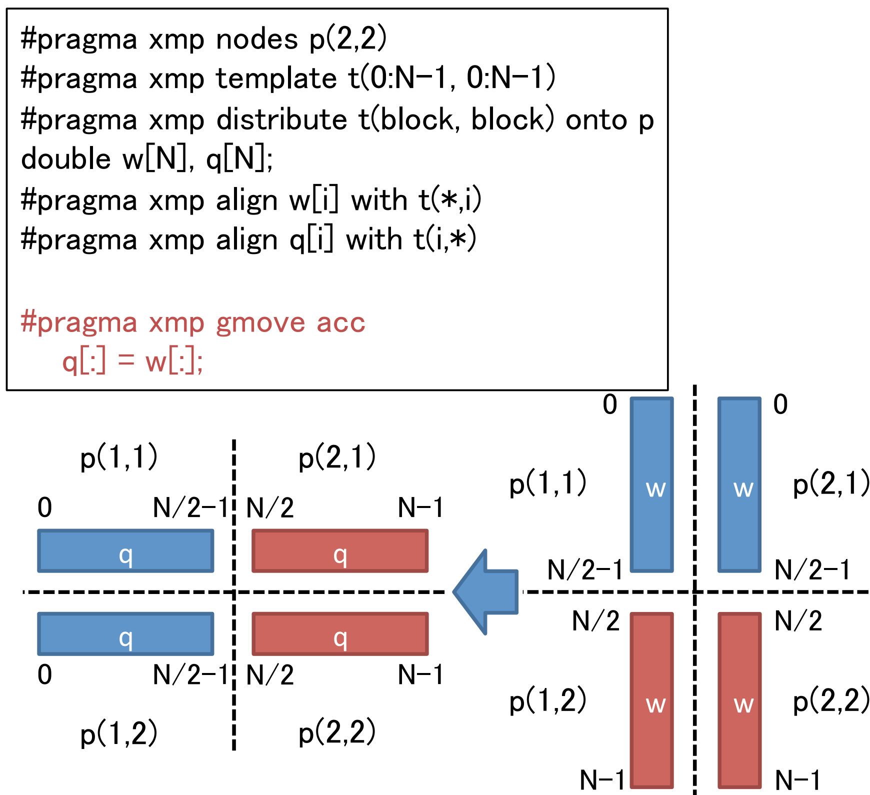


図 5.2: NPB CG にて用いられる gmove (4 ノード)

CUDA に対応した MPI を用いている場合には，既存の XMP のリダクション用ランタイムライブラリに OpenACC host_data 指示文により GPU のポインタを渡すように変更した．MPI が CUDA に対応していない場合には，アクセラレータのデータをホストに確保したメモリにコピーし，そのデータに MPI_Allreduce() を実行した後にアクセラレータへ書き戻すように実装した．

5.2.3 gmove 指示文

gmove 指示文は任意の通信を記述できるがすべてのパターンに対応するのは困難であるため，評価に用いる NPB CG で現れる 1 パターンについてのみ実装した．それは図 5.2 のように，2 次元テンプレートのある次元で分散された 1 次元配列を同一テンプレートの他の次元で分散された 1 次元配列に代入するパターンである．最初に XMP の gmove 指示文の実装の調査を行った．その結果，ノードの行数と列数が同じ場合には効率よく通信ができているが，ノードの行数と列数が異なる場合において非効率な通信が行われていた．例えば CG ではノードの列数が行数の 2 倍となることがあり，その際に図 5.3 に示す通信が行われていた．ノードの半分は 2 つのノードにデータを送る一方で，他の半分のノードはどこにもデータを送っていないためバランスが悪いうえに，同じノードで保持するデータも他のノードから送信しているので効率が悪い．そこで通信を図 5.4 に示すように全ノードが 1 つのノードに送るように改善した．さらに現在の実装では通信相手や送受信の範囲を求める計算部分が非常に複雑になっているため，通信相手と送受信の範囲をキャッシュすることで 2 度目以降は直ちに MPI 通信を開始できるようにしている．

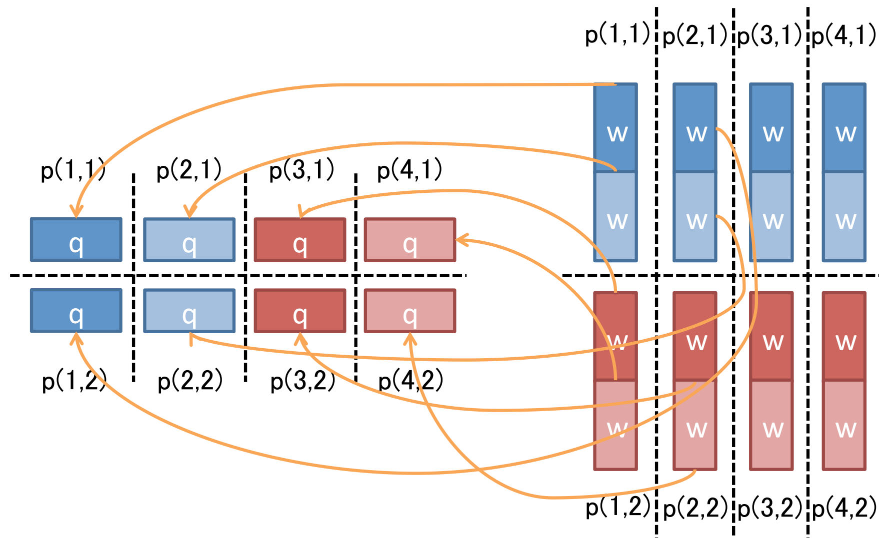


図 5.3: NPB-CG の gmove の改善前 (8 ノード)

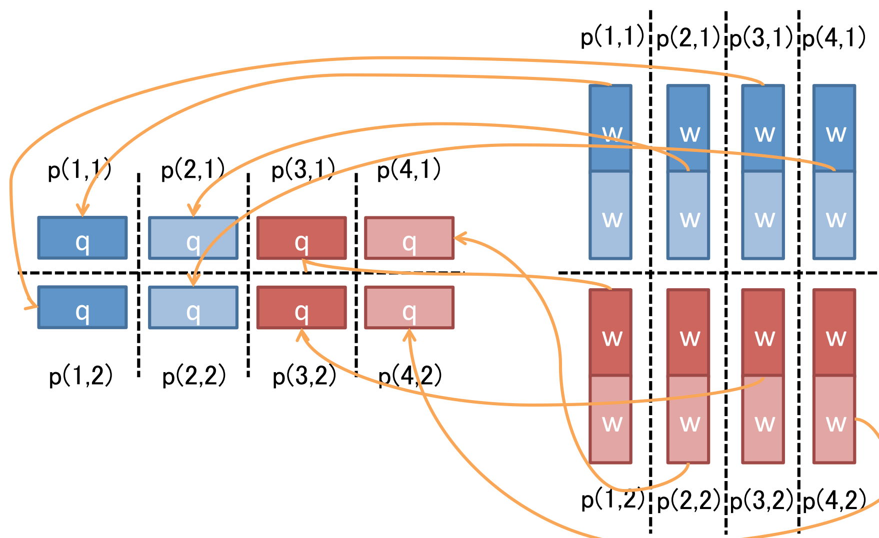


図 5.4: NPB-CG の gmove の改善後 (8 ノード)

5.2.4　coarray 機能

XACC における coarray の MPI による実装方法について解説する．なお，現在は NVIDIA GPU のみで対応している．MPI では，プロセスは *window* を通してリモートメモリにアクセスする．Omni compiler ではすべての coarray をホストメモリ用の window とアクセラレータメモリ用の window の 2 つで管理する．これにより coarray の同期をまとめて行うことが可能である．XACC は初期化時に coarray 用のヒープメモリをホスト用と GPU 用に確保し，それに関連付けた window をそれぞれ作成する．ヒープメモリサイズは環境変数 “XMP_ONESIDED_HEAP_SIZE” から指定が可能であり，すべての coarray のサイズの合計がこれを超えないように設定する必要がある．MPI の RMA は window のアクセスエポック内で

しか発行できない．そこで本実装では初期化の時に MPI_Win_lock_all() によりアクセスエポックを開始し，MPI_Win_unlock_all() により終了する．これによってユーザプログラム中では任意のタイミングで coarray による通信が可能となる．

ソースコード 5.1 とソースコード 5.2 はアクセラレータ上の coarray 宣言とそれを変換したコードである．

ソースコード 5.1: アクセラレータ上の coarray 宣言

```
1 int A[64]:[*];
2 #pragma acc declare create(A)
```

ソースコード 5.2: アクセラレータ上の coarray 宣言を変換したコード

```
1 void * _XMP_COARRAY_DESC_A;
2 int * _XMP_COARRAY_ADDR_A;
3 int * _XMP_COARRAY_ADDR_DEV_A;
4
5 extern void xmpc_traverse_init_file_sample_pp()
6 {
7  _XMP_coarray_malloc_info_1(0x40, sizeof(int));
8  _XMP_coarray_malloc_image_info_1();
9  _XMP_coarray_malloc_do(&(_XMP_COARRAY_DESC_A),&(_XMP_COARRAY_ADDR_A));
10  _XMP_coarray_malloc_do_acc(&(_XMP_COARRAY_DESC_A),&(_XMP_COARRAY_ADDR_DEV_A));
11  acc_map_data(_XMP_COARRAY_ADDR_A,
12               _XMP_COARRAY_ADDR_DEV_A,
13               _XMP_coarray_get_total_elmts(_XMP_COARRAY_DESC_A)*sizeof(int));
14 }
```

coarray *A[]* が宣言されていてかつ `declare` 指示文で指定されているので，この coarray はホストメモリとアクセラレータメモリの両方に確保される．*_XMP_COARRAY_DESC_A* は coarray *A[]* のホストアドレス・アクセラレータアドレス・形状等を保持した構造体へのポインタである．*_XMP_COARRAY_ADDR_A* は coarray のホストアドレス， *_XMP_COARRAY_ADDR_DEV_A* は coarray のアクセラレータアドレスでローカルの coarray アクセスに用いられる．この変換コードでは省略されているが，実際にはまず coarray 用のヒープ確保や window 作成が行われる．その後，このソースコードに示してある xmpc_traverse-_init_file...() が呼び出される．この関数はコンパイルの翻訳単位における初期化関数で，Omni compiler でリンクする際にはすべてのファイルの初期化関数を呼ぶルーチンが作成される．関数内では coarray *A[]* のサイズやイメージの情報を設定したのちに，_XMP_coarray_malloc_do() によりホスト側の coarray が，_XMP_coarray_malloc_do_acc() によりアクセラレータ側の coarray がヒープメモリから確保される．最後に，OpenACC の acc_map_data() を用いてホストメモリとアクセラレータメモリの対応付けを行うことで OpenACC から利用可能としている．

次に coarray による通信について述べる．coarray の put と get 操作は，それぞれ MPI_Put() や MPI_Get() により行われる．それらの関数ではリモートの window とそのオフセットに対して，ローカルのポインタのデータを put またはそのポインタの場所に get する．coarray の代入文におけるリモートの coarray が `use_device` 節で指定されているなら，アクセラレータ側の window を，そうでなければホスト側の

window を使用する．ローカルの変数や配列や coarray が use_device 節で指定されているなら，アクセラレータメモリのポインタを，そうでなければホストメモリのポインタを使用する．MPI_Put/Get() は非同期関数で通信の完了までは保証しない．リモートの coarray やローカルの配列に対して安全に読み書きを行うため，本実装では MPI_Put/Get() の直後に MPI_Win_flush() により通信の完了を待つようにしている．しかしながら，put/get ごとに完了待ちをすると依存のない通信を同時に発行できないという問題があるため，改善を検討している．現在の実装では，試験的に環境変数 "XMP_PUT_NB" や "XMP_GET_NB" を設定することで，通信をノンブロッキングにできるようにしている．なお，その際にはユーザ側が同期を行うまでの coarray 通信間で依存がないことを保証しなければならない．ソースコード 5.3 とソースコード 5.4 にアクセラレータ上の coarray 通信とその変換後のコードを示す．

ソースコード 5.3: coarray によるアクセラレータメモリ間の put 通信

```
1 //This statement puts array B on device to coarray A on device
2 #pragma acc host_data use_device(A, B)
3   A[:]:[2] = B[:];
```

ソースコード 5.4: coarray によるアクセラレータメモリ間の put 通信を変換したコード

```
1 unsigned long long _ACC_size_A =
2                            _XMP_coarray_get_total_elmts(_XMP_COARRAY_DESC_A);
3 #pragma acc host_data use_device(_XMP_COARRAY_ADDR_A[0:_ACC_size_A], B)
4 {
5   _XMP_coarray_rdma_coarray_set_1(0, 64, 1);
6   _XMP_coarray_rdma_array_set_1(0, 64, 1, 64, sizeof(int));
7   _XMP_coarray_rdma_image_set_1(2);
8   _XMP_coarray_rdma_do_acc(_XMP_PUT, _XMP_COARRAY_DESC_A, B, NULL, 1, 1);
9 }
```

この例では，coarray *A[]* と *B[]* はアクセラレータ上に確保されており，use_device 節によりアクセラレータ側のデータを通信の対象としている．すなわちこの通信では自ノードのアクセラレータ上の *B[]* をイメージ 2 のアクセラレータ上の coarray*A[]* に put する．変換後のコードでは，まず同様に use_device により配列 *A[],B[]* に対してアクセラレータアドレスを用いるようにする．_XMP_coarray_rdma_...() では通信する coarray サイズ, 配列サイズ, イメージ番号を設定したのち， リモート window にアクセラレータ用の window を用いる_XMP_coarray_rdma_do_acc() で通信を実行する．なお通信先がホストの coarray の場合には_XMP_coarray_rdma_do() を用いる．

同期に関しては，まず xmp_sync_memory() は MPI_Win_sync() によって window の同期を取りかつコンパイラ最適化などによる前後のコードの入れ替わりを防ぐ．もし put/get がノンブロッキングになるよう環境変数を指定していた場合は，MPI_Win_flush_all() によって自プロセスが発行したすべての RMA の完了を待機する．xmp_sync_image() や xmp_sync_images() は MPI_Send/Recv() による 1 対 1 同期の前後に xmp_sync_memory() を入れることにより実装しており，xmp_sync_all() は MPI_Barrier() による全体同期の前後に xmp_sync_memory() を入れることにより実装している．

表 5.1: HA-PACS/TCA におけるグローバルビューモデルの性能評価に用いたソフトウェアと環境変数

Compiler	GCC 4.4.7, CUDA 6.5 MVAPICH2-GDR 2.1a Omni OpenACC compiler 0.9.1
Environment variable	MV2_ENABLE_AFFINITY=0 MV2_NUM_PORTS=2 MV2_USE_CUDA=1, MV2_CUDA_IPC=0 MV2_USE_GPUDIRECT_GDRCOPY=1 MV2_USE_GPUDIRECT_RECEIVE_LIMIT=8192

5.3 NVIDIA GPU クラスタにおけるグローバルビューモデルの性能評価

評価には筑波大学計算科学研究センターの HA-PACS/TCA を利用した．1 ノードあたり 4 枚の GPU が搭載されているため，1 MPI プロセス当たり 1 GPU を割り当てて 1 ノードで 4 MPI プロセスを実行し，最大で 16 ノード上で 64 MPI プロセスを実行した．HA-PACS/TCA には GPU 間直接通信のための Tightly Coupled Accelerators (TCA) [38] が搭載されているが，この評価では InfiniBand を用いて一般的なクラスタにおける性能を評価する．また使用したソフトウェアと環境変数を表 5.1 に示す．MPI には CUDA に対応した MVAPICH2-GDR 2.1a を利用し，mpicc のコンパイルオプションには “-O3” を指定した．OpenACC コンパイラには本研究で実装した Omni OpenACC compiler を用い，バックエンドの CUDA コンパイラオプションには “-O3 -arch=sm_35” を指定した．

5.3.1 Himeno Benchmark

Himeno Benchmark は非圧縮流体解析コードの性能を評価するためのベンチマークである [39]．主な演算はポアッソン方程式解法をヤコビの反復法で解く 3 次元の 19 点ステンシルであり，主な通信は袖領域の交換である．複数の GPU を利用するため，問題のサイズは Large を用いた．Large では問題領域の大きさは $(i \times j \times k) = (256 \times 256 \times 512)$ である．ソースコード 5.5 に XACC で記述した Himeno Benchmark の分散の定義と反復処理部分を示す．k 次元の大きさは高々 512 であるうえ，分割すると袖領域がストライドになってしまうので i と j 次元の 2 次元分割とした．主計算であるステンシル計算は i,j,k の 3 重ループで構成されており，GPU で並列処理するために i,j ループをスレッドブロック（OpenACC における gang）で，k ループをスレッド（OpenACC における vector）で並列化した．比較として Himeno Benchmark の MPI 版に OpenACC 指示文を追加した “MPI+OpenACC” 版を用意した．オリジナルの MPI 版では袖領域の通信に MPI の派生型であるベクトル型を用いているが，jk 平面では padding も含めて連続型として通信するように改変した．また，XACC では ik 平面をパックして連続データとして通信するため，同様にパックするようにした “MPI+OpenACC (pack)” も比較対象に加えた．さらに，XACC

の機能を用いずに XMP と OpenACC で記述した “XMP+OpenACC” とも比較する．性能測定時の反復回数は 1000 回とし 10 回の計測のうち最良値を使用した．

ソースコード 5.5: XACC で記述した Himeno Benchmark

```
1 #pragma xmp template t(0:MKMAX-1, 0:MJMAX-1, 0:MIMAX-1)
2 #pragma xmp nodes n(1, NDY, NDX)
3 #pragma xmp distribute t(block, block, block) onto n
4 #pragma xmp align p[k][j][i] with t(i, j, k)
5 #pragma xmp shadow p[1:2][1:2][0:1]
6 // 省略した配列bnd,wrk1,wrk2のalignとshadowは配列pと同じ
7 #pragma xmp align a[*][k][j][i] with t(i, j, k)
8 #pragma xmp shadow a[0][1:2][1:2][0:1]
9 // 省略した配列b,cのalignとshadowは配列aと同じ
10 ...
11 #pragma acc data copy(p,bnd,wrk1,wrk2,a,b,c),create(gosa)
12 {
13 #pragma xmp reflect_init(p) width(1,1,0) acc
14     ...
15   for(n=0 ; n<nn ; ++n){
16     gosa = 0.0;
17 #pragma acc update device(gosa)
18
19 #pragma xmp loop (k,j,i) on t(k,j,i)
20 #pragma acc parallel loop firstprivate(omega) reduction(+:gosa) collapse(2)
       gang vector_length(64) async
21
22     for(i=1 ; i<imax-1 ; ++i)
23       for(j=1 ; j<jmax-1 ; ++j){
24 #pragma acc loop vector reduction(+:gosa) private(s0,ss)
25         for(k=1 ; k<kmax-1 ; ++k){
26           s0 = a[0][i][j][k]*p[i+1][j][k]+a[1][i][j][k] ...;
27           ss = ( s0 * a[3][i][j][k] - p[i][j][k] )
28              * bnd[i][j][k];
29           gosa += ss*ss;
30           wrk2[i][j][k] = p[i][j][k] + omega * ss;
31         }
32       }
33 #pragma xmp loop (k,j,i) on t(k,j,i)
34 #pragma acc parallel loop collapse(2) gang vector_length(64) async
35
36     for(i=1 ; i<imax-1 ; ++i)
37       for(j=1 ; j<jmax-1 ; ++j){
38 #pragma acc loop vector
39         for(k=1 ; k<kmax-1 ; ++k)
40           p[i][j][k] = wrk2[i][j][k];
41       }
42 #pragma acc wait
43 #pragma xmp reflect_do(p) acc
44 #pragma acc update host(gosa)
```

```
45 #pragma xmp reduction(+:gosa)
46   } /* end n loop */
47 ...
48 } //end of acc data
49 ...
```

Himeno Benchmark の性能を図 5.5 に，1 反復の実行時間の内訳を図 5.6 に示す．XACC における性能は MPI+OpenACC と比較して 93.7–102.8%，MPI+OpenACC (pack) と比較して 93.4–96.6% であった．一方，XMP+OpenACC では分割数が増えた際の性能向上率が悪く，8 × 8 × 1 分割では XACC の 8 割ほどの性能にとどまっている．実行時間の内訳から，ステンシル計算が全体の 4–9 割程を占めており，XACC や XMP+OpenACC では計算時間が MPI+OpenACC と比較して 6.5–13.5% 長いことが分かる．

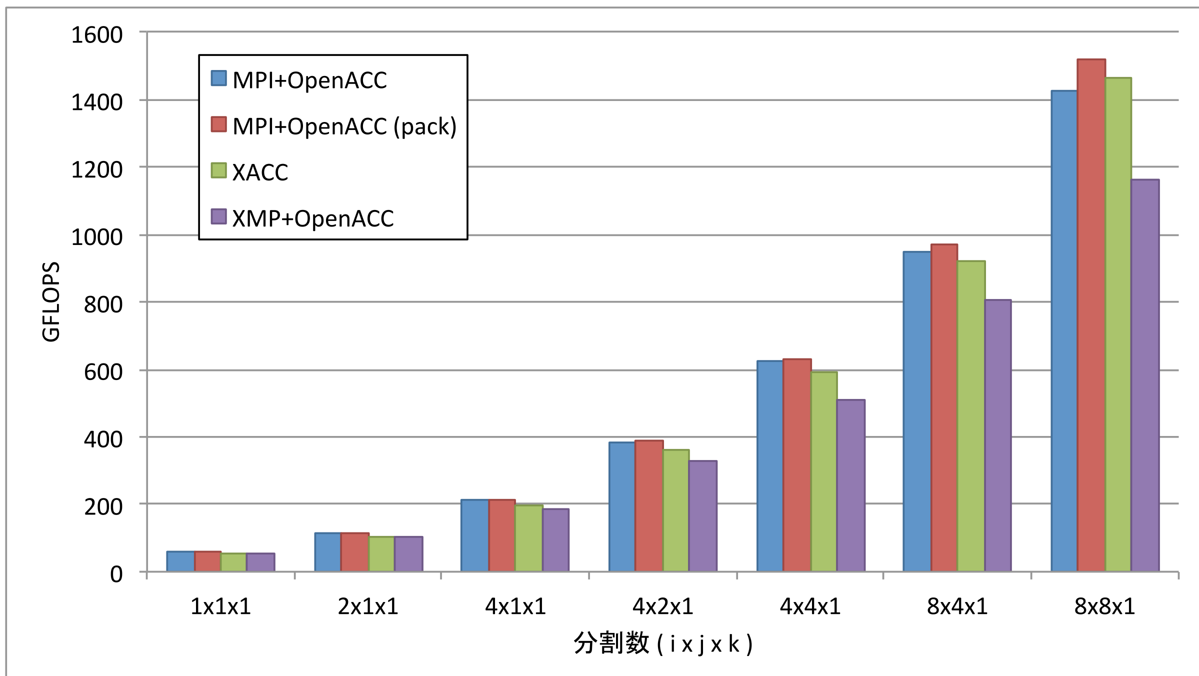


図 5.5: Himeno Benchmark の性能

ステンシル計算時間の差は配列へのアクセス方法の違いによる．MPI+OpenACC では，実行ノード数から配列サイズを事前に決定するため，静的配列へのアクセスとしてコンパイルされる．XACC や XMP+OpenACC では，分散配列は malloc で動的に確保されるよう変換されるため，ポインタへのアクセスとしてコンパイルされる．例として *p[i][j][k]* は **(p + acc0 * i + acc1 * j + k)* に変換される．次元の大きさを保持する *acc0* や *acc1* は分散配列ごとに用意される．この変換により，(1) インデックス計算の増加，(2) カーネルの実行率の低下，が生じる．(1) が生じるのは，分散配列 *p* 以外の分散配列は 3 次元配列なら *[i][j][k]*，4 次元配列なら *[0–3][i][j][k]* とほぼ同じインデックスでアクセスするにも関わらず，そのオフセット計算に必要な変数が分散配列ごとに異なるため，それぞれオフセット計算が必要になるからである．(2) が生じるのは，使用する変数の増加により GPU カーネルで必要なレジスタ数が増加し，同時実行可能なブロック数が減少するからである．これらの要因の影響を調査するため表 5.2 に示すようにカーネルを変えて 1 GPU での性能を測定した．占有率は GPU の SMX が実行可能なワープ数に対してど

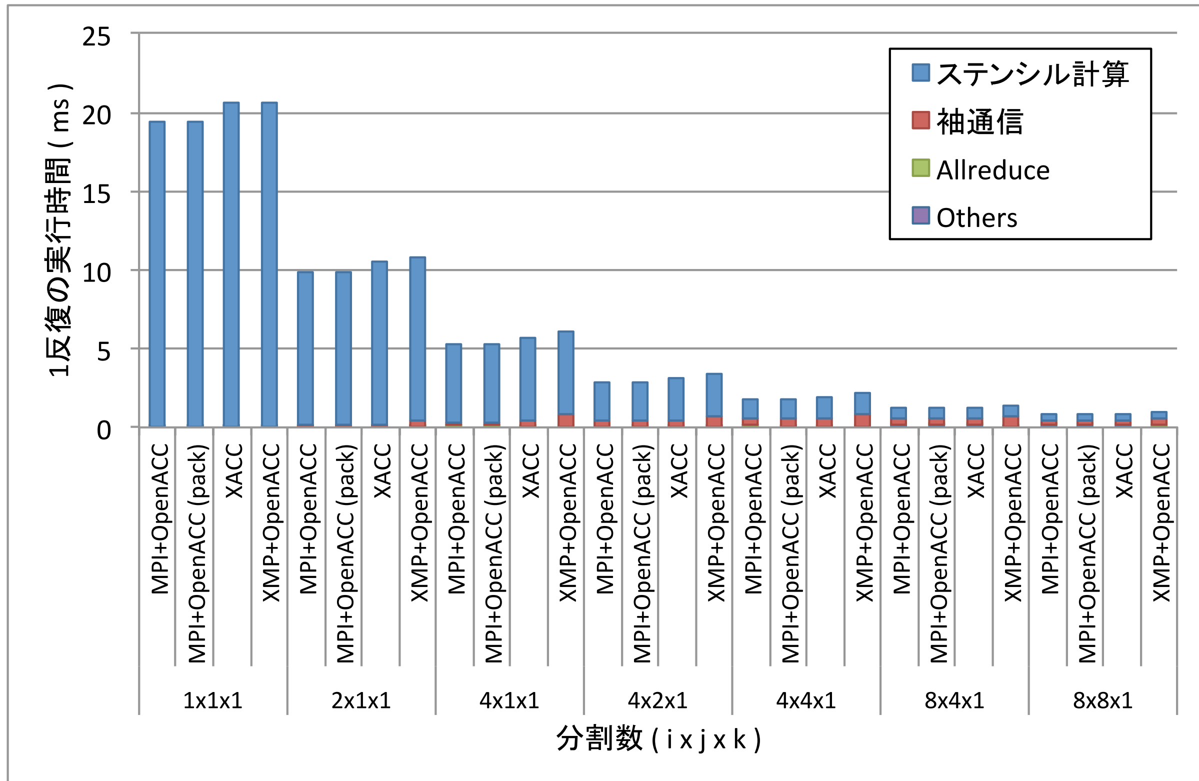


図 5.6: Himeno Benchmark の 1 反復の実行時間の内訳

れだけ実行しているかを表す割合であり，この割合が高いほど良い性能が出やすい．“XACC (reg=72)” および “XACC (reg=64)” はカーネルコンパイル時にオプションによって最大レジスタ数を制限したもので，72 や 64 は占有率が変わる境界値である．レジスタ数を制限することで若干性能は向上するが，“XACC (reg=64)” ではレジスタスピルによる性能低下が起こった．“XACC (shrink)” は分散配列ごとに別々に使用していたオフセット計算用の変数を共有して削減したもので，MPI+OpenACC に匹敵する性能が得られたことからインデックス計算の増加が性能低下の主要な原因であることが明らかになった．

表 5.2: Himeno Benchmark のカーネルによる占有率や性能の違い

	レジスタ数	占有率 (%)	GFLOPS
MPI+OpenACC	42	50.0	57.6
XACC	74	37.5	54.1
XACC (reg=72)	72	43.8	55.0
XACC (reg=64)	64	50.0	54.9
XACC (shrink)	66	43.8	57.3

次に袖通信の時間を図 5.7 に示す．XACC では MPI+OpenACC と比較して 93.4–114%, MPI+OpenACC (pack) と比較して 100–115% である．MPI+OpenACC(pack) や XACC では不連続な袖を pack することで通信時間が削減されている．しかしながら，XACC では $2 \times 1 \times 1$ や $4 \times 1 \times 1$ 分割において通信時間

が大きく増加している．これは通信量の違いによるものである．表 5.3 は袖通信で送られる要素数の比較である．MPI+OpenACC では分割しない次元には袖領域を確保せず，また分割する次元でも 2 プロセスでの分割の場合には片方だけしか袖領域を確保しない．一方 XACC では分割方法に関わらず `shadow` 指示文で定義された袖領域を確保する．それにより，例えば $2 \times 1 \times 1$ 分割では XACC の方が j 次元の大きさが 2 要素大きくなり，送る要素数としては 2×513 要素多くなる．このように分割が少ないときに XACC では MPI+OpenACC よりも多く通信することになり通信時間が増加した．XMP+OpenACC においては，通信量は XACC と同じであるがホストとデバイス間のデータコピーを OpenACC の `update` 指示文で記述したことにより通信時間が大幅に増加した．XACC の通信で用いている MVAPICH2 では，データがデバイスからホスト，ホストからホスト，ホストからデバイスへパイプライン転送されるが，XMP+OpenACC ではそれらを順次行うことになるからである．

表 5.3: Himeno Benchmark の袖通信の要素数

分割数 i×j×k	MPI+OpenACC		XACC	
	jk 平面	ik 平面	jk 平面	ik 平面
$2 \times 1 \times 1$	256×513	-	258×513	-
$4 \times 1 \times 1$	256×513	-	258×513	-
$4 \times 2 \times 1$	129×513	66×512	130×513	66×512
$4 \times 4 \times 1$	66×513	66×512	66×513	66×512
$8 \times 4 \times 1$	66×513	34×512	66×513	34×512
$8 \times 8 \times 1$	34×513	34×512	34×513	34×512

5.3.2 NAS Parallel Benchmarks CG

NPB CG は正値対称な大規模疎行列の最小固有値を共役勾配法によって解くベンチマークである．複数の GPU を用いた際にスケールするよう，行列サイズが $1{,}500{,}000 \times 1{,}500{,}000$ である Class D を用いた．XMP による CG の実装は文献 [40] で行われている．MPI 版と同じく 2 次元分割であり，XACC の実装でも同様の分割とした．ソースコード 5.6 に XACC による NPB CG のコードの一部を示す．主計算である疎行列ベクトル積は行と列の 2 重ループとなっており，GPU では行のループをスレッドブロックで，列のループをスレッドで並列化した．また主な通信は

1. 疎行列ベクトル積の結果を各行で足し合わせるための配列の Allreduce (27 行目)
2. (1) の結果を行分割から列分割にするための通信 (28,29 行目)

の 2 つであり，(1) は 27 行目の `reduction` 指示文で，(2) は 28 行目の `gmove` 指示文で記述した．元の MPI 版の CG は Fortran で記述されていたため，C で書き直してさらに OpenACC 指示文を追加した “MPI+OpenACC” を比較対象として実装した．また，Himeno Benchmark と同じく XMP と OpenACC の組み合わせで記述した “XMP+OpenACC” との比較も行う．なおこの問題サイズでは GPU のメモリ不足により 1 プロセスでの実行ができなかったため，2–64 プロセスでの評価のみ行った．

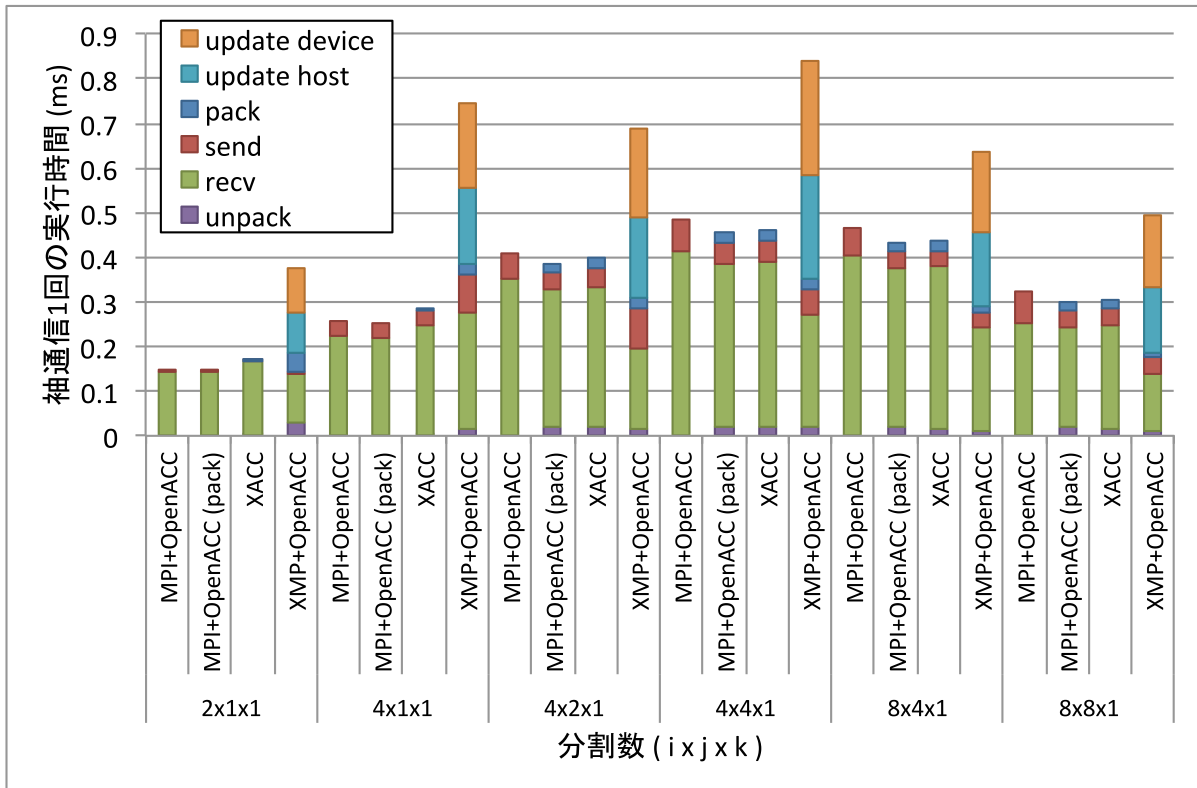


図 5.7: Himeno Benchmark の袖通信 1 回の実行時間

ソースコード 5.6: XACC による NPB CG のコード

```
1 #pragma xmp nodes p(NUM_PROC_COLS,NUM_PROC_ROWS)
2 #pragma xmp nodes sub_p(NUM_PROC_COLS)=p(:,*)
3 #pragma xmp template t(0:NA-1,0:NA-1)
4 #pragma xmp distribute t(block, block) onto p
5 #pragma xmp align w[i] with t(*,i)
6 #pragma xmp align q[i] with t(i,*)
7 //省略した配列r,p,x,zのalignは配列qと同じ
8 ...
9 #pragma acc data copy(p,q,r,x,z,w,rowstr[0:NA+1],a[0:NZ],colidx[0:NZ])
10 {
11   ...
12   for (cgit = 1; cgit <= cgitmax; cgit++) {
13     rho0 = rho;  d = 0.0; rho = 0.0;
14 #pragma xmp loop on t(*,j)
15 #pragma acc parallel loop gang
16     for(j=0; j < NA; j++){
17       double sum = 0.0;
18       int rowstr_j = rowstr[j];
19       int rowstr_j1 = rowstr[j+1];
20 #pragma acc loop vector reduction(+:sum)
21       for (k = rowstr_j; k < rowstr_j1; k++) {
22         sum = sum + a[k]*p[colidx[k]];
```

```
23       }
24       w[j] = sum;
25     }
26
27 #pragma xmp reduction(+:w) on sub_p(:) acc
28 #pragma xmp gmove acc
29     q[:] = w[:];
30
31 #pragma xmp loop on t(*,j)
32 #pragma acc parallel loop
33     for (j = 0; j < NA; j++)
34       w[j] = 0.0;
35 #pragma xmp loop on t(j,*)
36 #pragma acc parallel loop reduction(+:d)
37     for (j = 0; j < NA; j++)
38       d = d + p[j] * q[j];
39 #pragma xmp reduction(+:d) on sub_p(:)
40     alpha = rho0 / d;
41 #pragma xmp loop on t(j,*)
42 #pragma acc parallel loop
43     for (j = 0; j < NA; j++ ){
44       z[j] = z[j] + alpha*p[j];
45       r[j] = r[j] - alpha*q[j];
46     }
47 #pragma xmp loop on t(j,*)
48 #pragma acc parallel loop reduction(+:rho)
49     for (j = 0; j < NA; j++ )
50       rho = rho + r[j] * r[j];
51 #pragma xmp reduction(+:rho) on sub_p(:)
52     beta = rho / rho0;
53 #pragma xmp loop on t(j,*)
54 #pragma acc parallel loop
55     for (j = 0; j < NA; j++)
56       p[j] = r[j] + beta*p[j];
57   } /* end of do cgit=1,cgitmax */
58   ...
59 }//end of acc data
```

NPB CG の性能を図 5.8 に，また主関数である conj_grad() の 1 回の実行時間の内訳を図 5.9 に示す．XACC では MPI+OpenACC と比較して 73.5%–95.7% の性能が得られている．特に 4×8 分割のときの性能低下が大きい．また XMP+OpenACC では MPI+OpenACC と比較して 76.7%–92.6% の性能となっており，XACC と比べると 4×8 分割では性能が 4% 向上し，その他では 3–4% 低下している．

実行時間を比較すると XACC では SpMV や行分割から列分割にする通信の実行時間は MPI+OpenACC と同等であるが，配列リダクションの実行時間に大きな差があり 1.0–4.4 倍の時間がかかっている．XMP+OpenACC ではホストとデバイス間の通信を OpenACC の `update` 指示文で記述したことにより，行分割から列分割にする通信は XACC と比べて増加しているが，一方で配列リダクションは同じか減少している．これらの差は 2 つの要因から生じている．1 つ目はリダクションの実装の違いである．

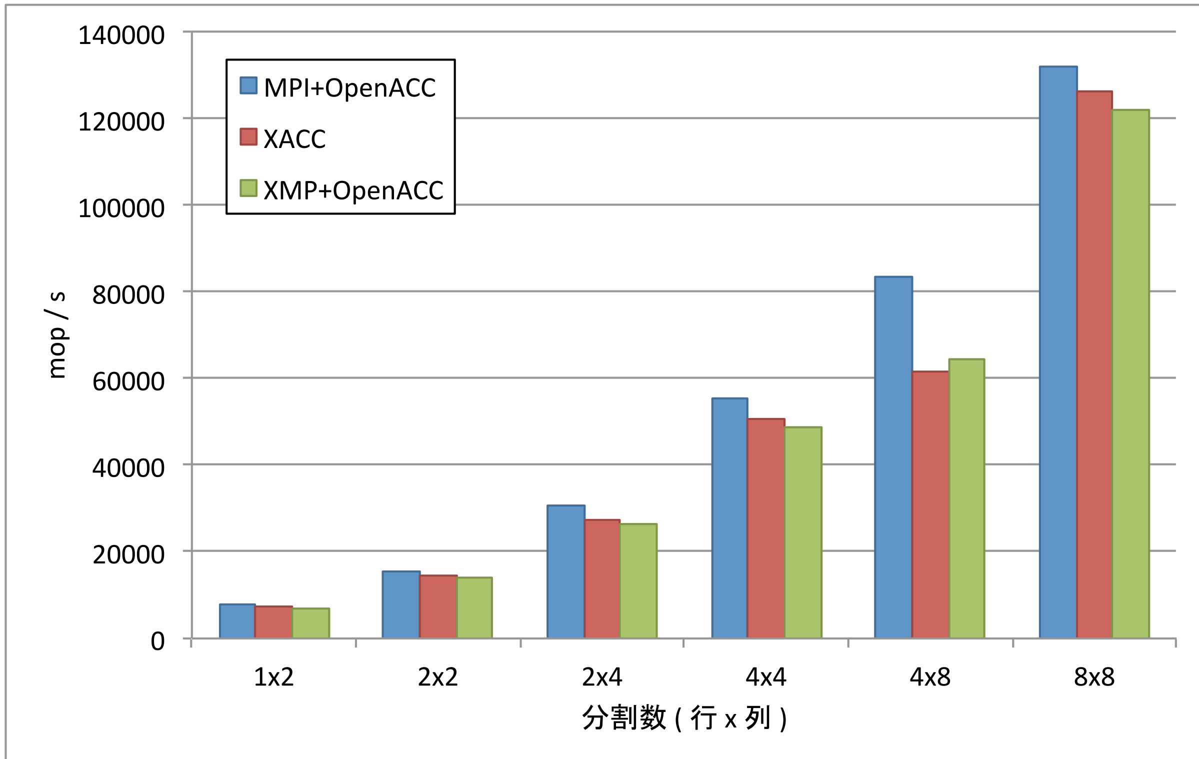


図 5.8: NPB CG の性能

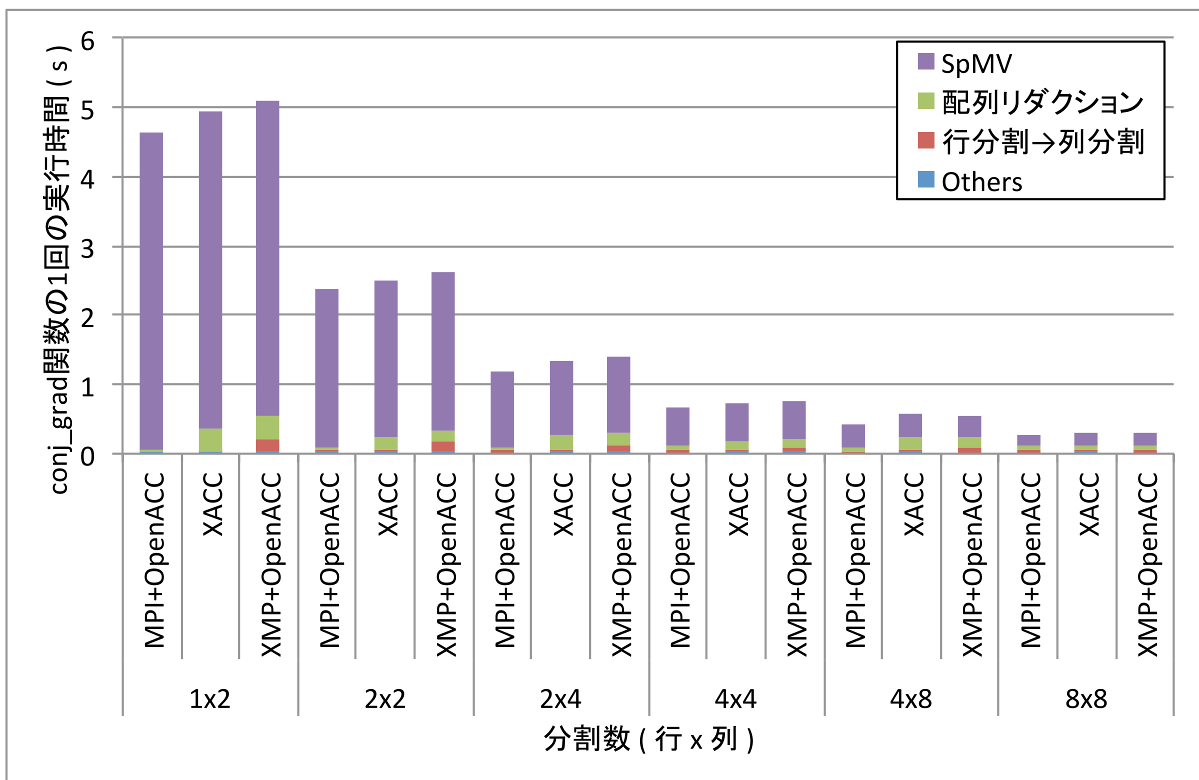


図 5.9: conj_grad 関数の 1 回の実行時間の内訳

MPI+OpenACC では，MPI_Isend/Recv() と配列を加算するループによって Recursive-Doubling 法によるリダクションを実現している．そのため，GPU メモリ間の通信と GPU での配列加算が行われる．一方，XACC では MPI_Allreduce() によりリダクションを行う．MVAPICH2 では，GPU メモリ上の Allreduce はホストメモリにコピーしてホスト上で Allreduce を行うように実装されている．すなわち，ホストメモリ間の通信と CPU での配列加算が行われる．XMP+OpenACC はホストメモリと GPU メモリ間のコピーを OpenACC で行う以外は XACC と同じである．CG で用いられるのと同じ GPU メモリ上の double 型配列の Allreduce のレイテンシを 2, 4, 8 プロセスで測定した結果をそれぞれ図 5.10，図 5.11，図 5.12 に示す．“MPI_Allreduce (GPU)” は MPI_Allreduce() に GPU メモリのポインタを渡す場合，“MPI_Isend/Recv” は allreduce を MPI_Isend/Recv() とカーネルで実現した場合，“MPI_Allreduce (Host)” は CUDA でホストにデータを送り，MPI_Allreduce() にホストメモリのポインタを渡す場合である．配列長が短い場合には通信レイテンシの低さから CPU で計算する MPI_Allreduce(GPU) の方が高速であるが，配列長が長い場合には計算の速さから GPU で計算する MPI_Isend/Recv() を用いた実装の方が高速である．プロセス数が増えると共に MPI_Isend/Recv() を用いた方が良くなるサイズが大きくなるのは，GPU 間通信やカーネル実行のオーバーヘッドが大きいからである．CG の各分割パターンでの配列 w の配列長を表 5.4 に示す．これらのデータから，8×8 の分割ではレイテンシに大きな差は無いが，その他の分割では MPI_Allreduce (GPU) よりも MPI_Isend/Recv() による実装の方が良いことが分かる．また同じ MPI_Allreduce() を使う場合でも 512 要素以下の場合には GPUDirect RDMA を用いる MPI_Allreduce (GPU) の方がレイテンシが短い．それ以上の要素数ではおおよそ同じではあるが，部分的に MPI_Allreduce (Host) のほうがレイテンシが短くなる．4×8 分割において XMP+OpenACC が XACC の性能を上回ったのはこれが原因である．

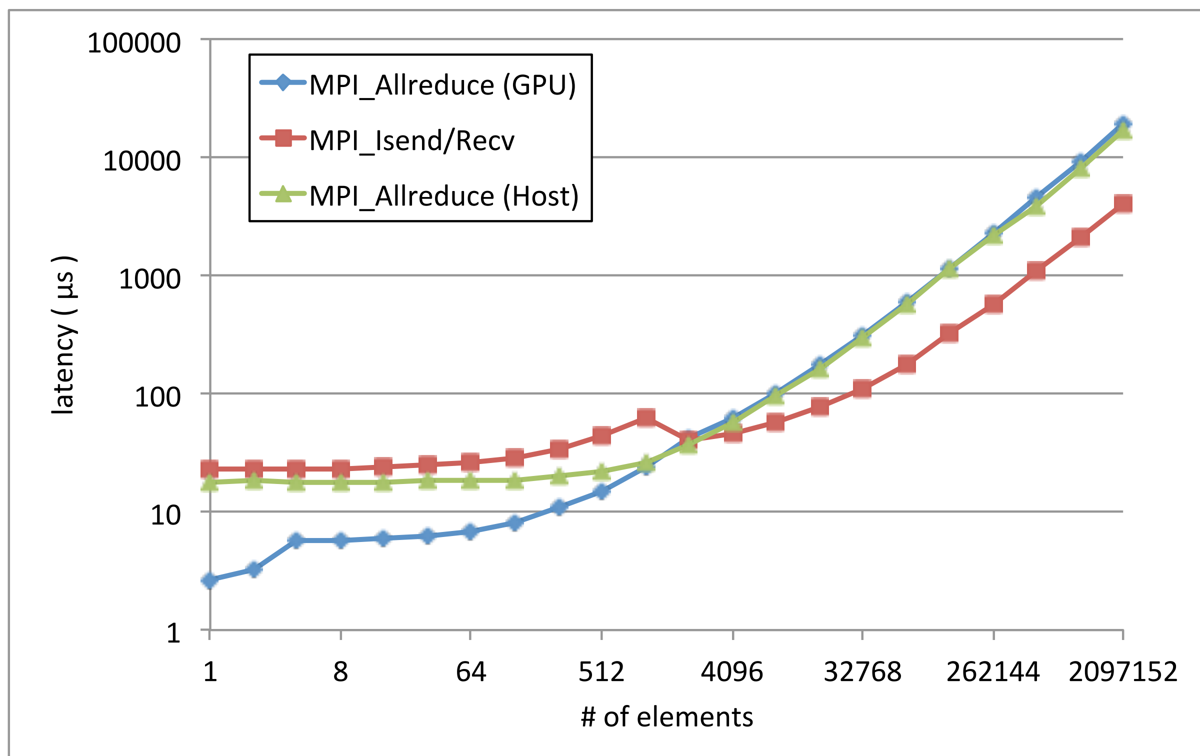


図 5.10: Allreduce のレイテンシ (2 プロセス)

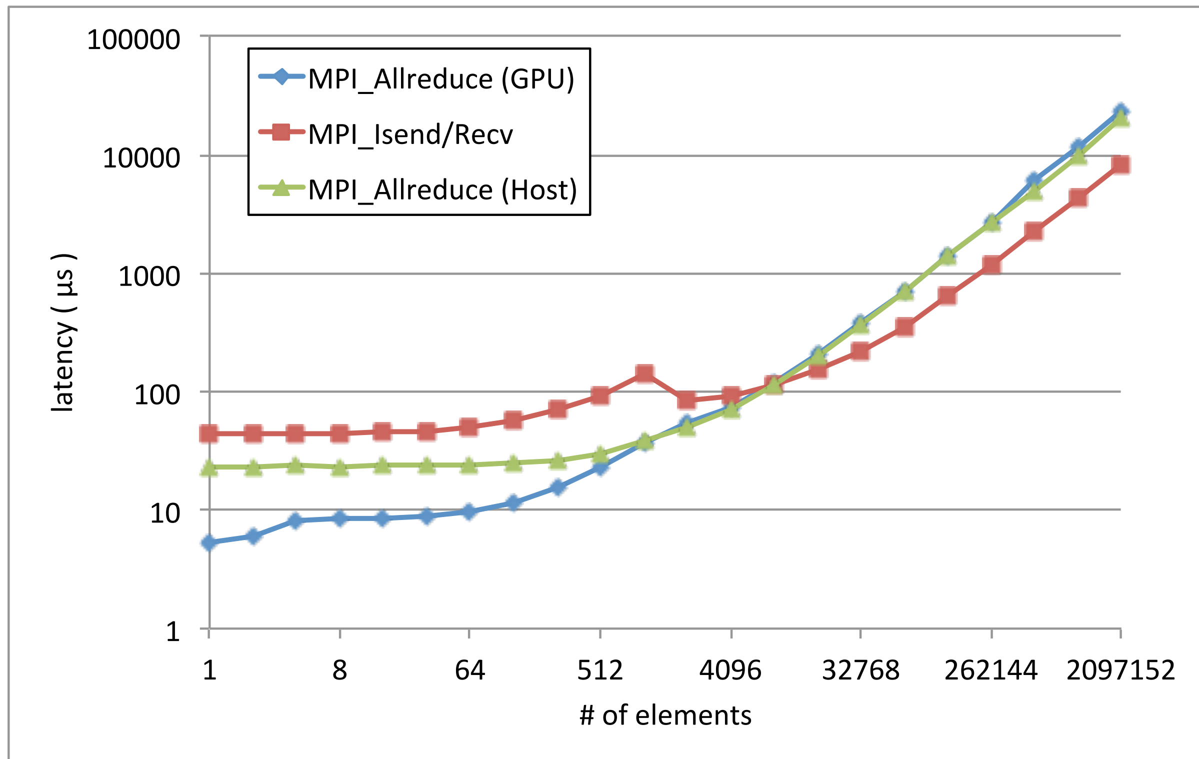


図 5.11: Allreduce のレイテンシ (4 プロセス)

表 5.4: CG の Class D における配列 w の長さ

分割数 (行 × 列)	1 × 1	1 × 2	2 × 2	2 × 4	4 × 4	4 × 8	8 × 8
配列長	1500000	1500000	750000	750000	375000	375000	187500

2 つ目の原因は処理する配列長の違いである．行の分割数と列の分割数が異なるとき図 5.4 のように行分割から列分割にするための通信に必要な配列要素は，偶数列のノードでは配列 w の前半分，奇数列のノードでは w の後半分のみである．そこで MPI+OpenACC ではすべての要素をリダクションせずに必要な部分のみリダクションを行うことで通信量を減らしている．一方，XACC や XMP+OpenACC は必ず配列 w の全要素をリダクションするため，行と列の分割数が異なる場合には MPI+OpenACC と比較して 2 倍の通信と演算が必要になる．それにより 2×4，4×8 分割にて配列リダクションの時間が大きく増加した．

5.4 NVIDIA GPU クラスタにおけるローカルビューモデルの性能評価

評価にはグローバルビューモデルと同じく Himeno benchmark と NPB CG を用いた．こちらでは，send/recv で通信する MPI+OpenACC，get で通信する MPI+OpenACC，XACC のグローバルビューモデル，XACC のローカルビューモデル，の 4 種類で比較を行う．ベンチマークはすべて C 言語で記述されている．評価環境には HA-PACS/TCA を用いた．計算ノードには CPU が 2 ソケット搭載されており，片

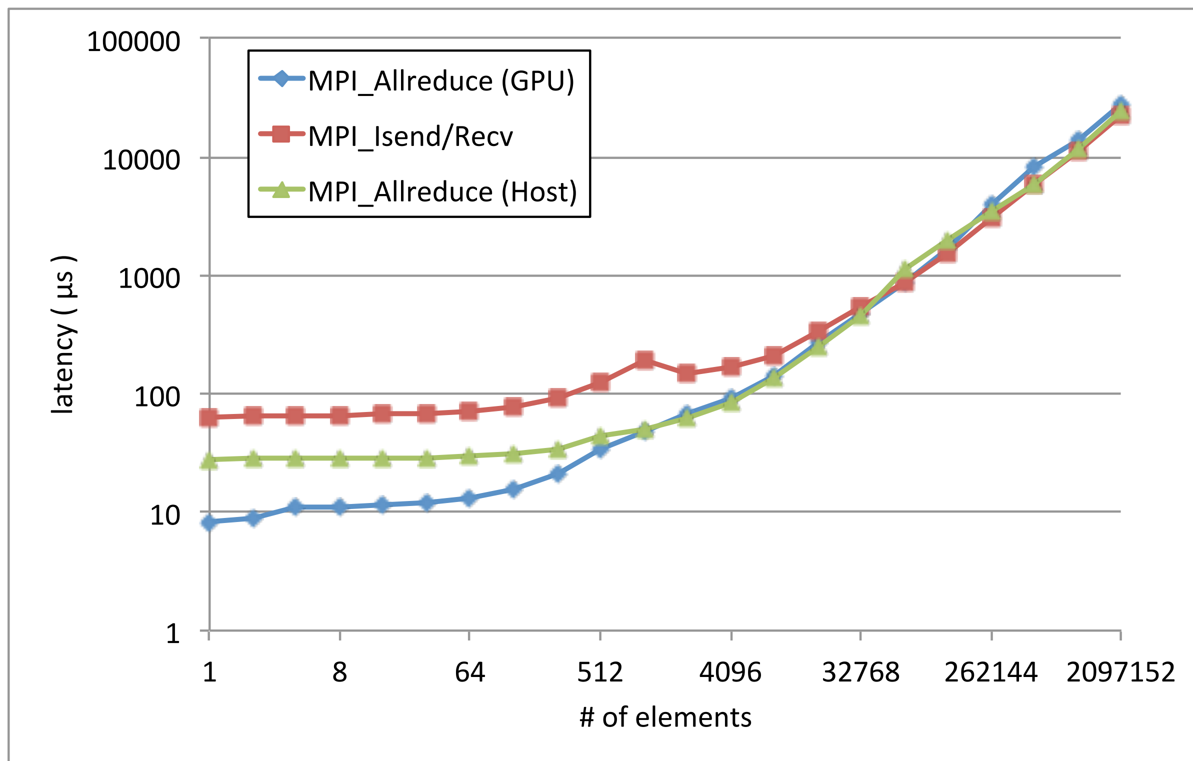


図 5.12: Allreduce のレイテンシ (8 プロセス)

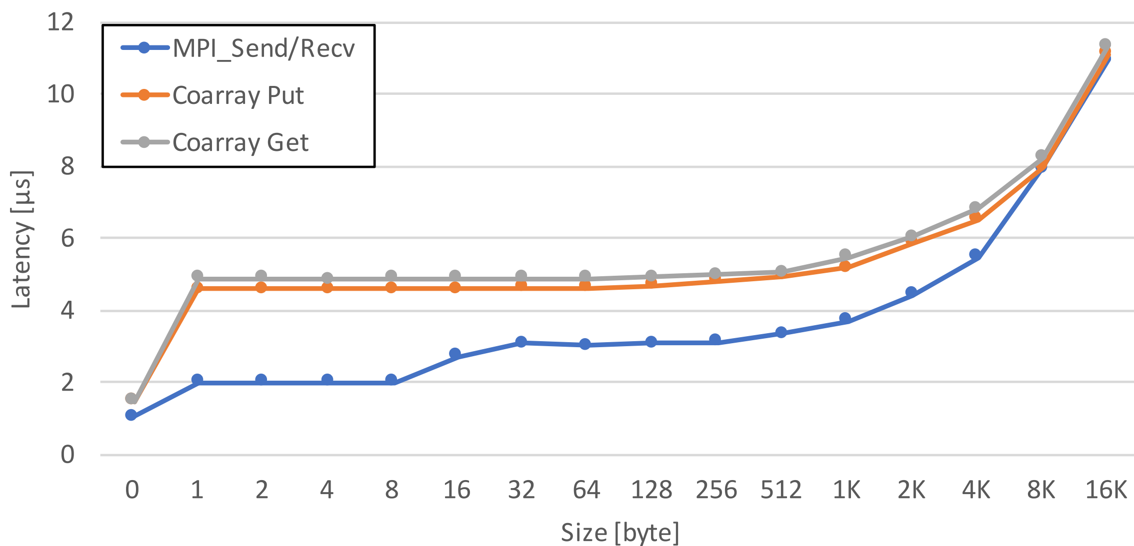
方の CPU 側に 2 枚の GPU と InfiniBand HCA が，もう片方の CPU 側に 2 枚の GPU と TCA が接続されており，CPU 間は QPI により接続されている．本評価では QPI を跨ぐ通信を避けるため，InfiniBand が接続されている側の 2 枚の GPU のみを用いる．また使用したソフトウェアの一覧を表 5.5 に示す．MPI

表 5.5: HA-PACS/TCA におけるローカルビューモデルの性能評価に用いたソフトウェア

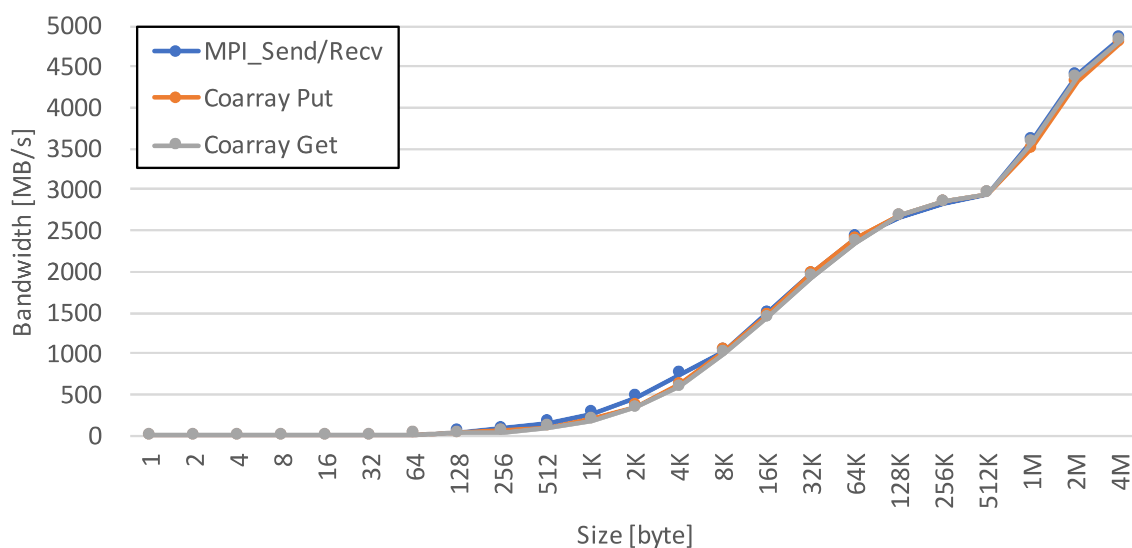
Compiler	GCC 4.4.7, CUDA 7.5
	MVAPICH2-GDR 2.2rc1
	Omni OpenACC/XACC compiler 1.0.3

には MVAPICH2-GDR を用い，GDR および gdrcopy[41] を有効化した．gdrcopy は NVIDIA が公開しているオープンソースのライブラリで，GDR を用いたホストと GPU 間の高速なメモリコピーが可能である．バックエンドの OpenACC コンパイラには本研究で実装した Omni OpenACC compiler を用いた．

はじめに，Ping-pong ベンチマークによる MPI_Send/Recv() と coarray の put と get の予備評価結果を図 5.13 に示す．coarray 版は通信の同期に xmp_sync_memory() と post/wait 指示文を用いている．データサイズが 8 KB より小さい時，put と get のレイテンシは MPI_Send/Recv() より大きいが，それ以外ではほぼ同じ性能が出ている．使用した MVAPICH2-GDR ではデフォルトで MPI_Send/Recv() の 8KB まで gdrcopy ライブラリを使って高速に通信を行うが，put と get においては用いられていないためレイテンシが増加している．しかしながら，この評価で用いる Himeno benchmark と NPB CG のサイズおよ



(a) レイテンシ



(b) バンド幅

図 5.13: MPI_Send/Recv() と coarray put/get の Ping-pong ベンチマーク性能

びプロセス数ではすべての RMA 通信が 4 KB 以上のサイズなので，この部分の影響は小さいと考えられる．なお特定のサイズの put 通信をすると実行時に MVAPICH2 ランタイムがエラーを起こす問題があったため，ベンチマークによる評価では片側通信に get を用いることにした．

5.4.1　Himeno benchmark

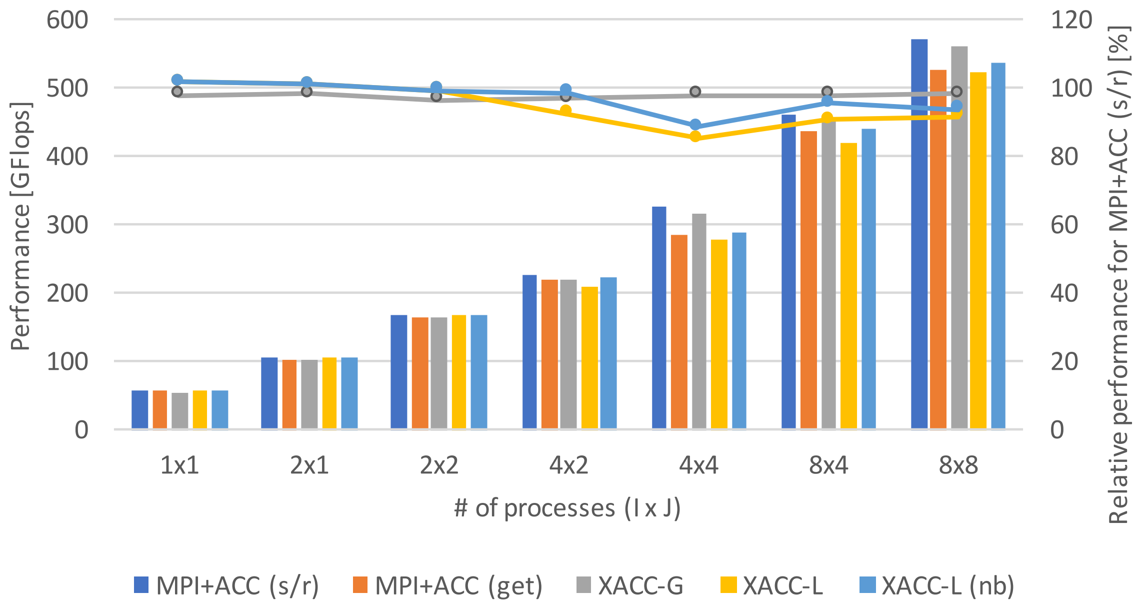
問題サイズとして Middle $(i \times j \times k) = (128 \times 128 \times 256)$ と Large $(256 \times 256 \times 512)$ を用いた．グローバルビューモデルの評価と同様に i, j 次元の 2 次元分割とした．比較の MPI+OpenACC 版は jk 平面にパディングを含めることで，ik 平面をパック・アンパックすることで連続領域を送るようにしている．XACC グローバルビュー版はソースコード 5.5 で示したように袖領域の更新は `reflect` 指示文で記述している．XACC ローカルビュー版は jk 平面はそのまま連続領域として通信し，ik 平面では OpenACC でパック・アンパックして，coarray の get により通信する．ik 平面における袖交換をソースコード 5.7 に示す．

ソースコード 5.7: XACC ローカルビューモデルにおける Himeno benchmark の袖交換 (ik 平面)

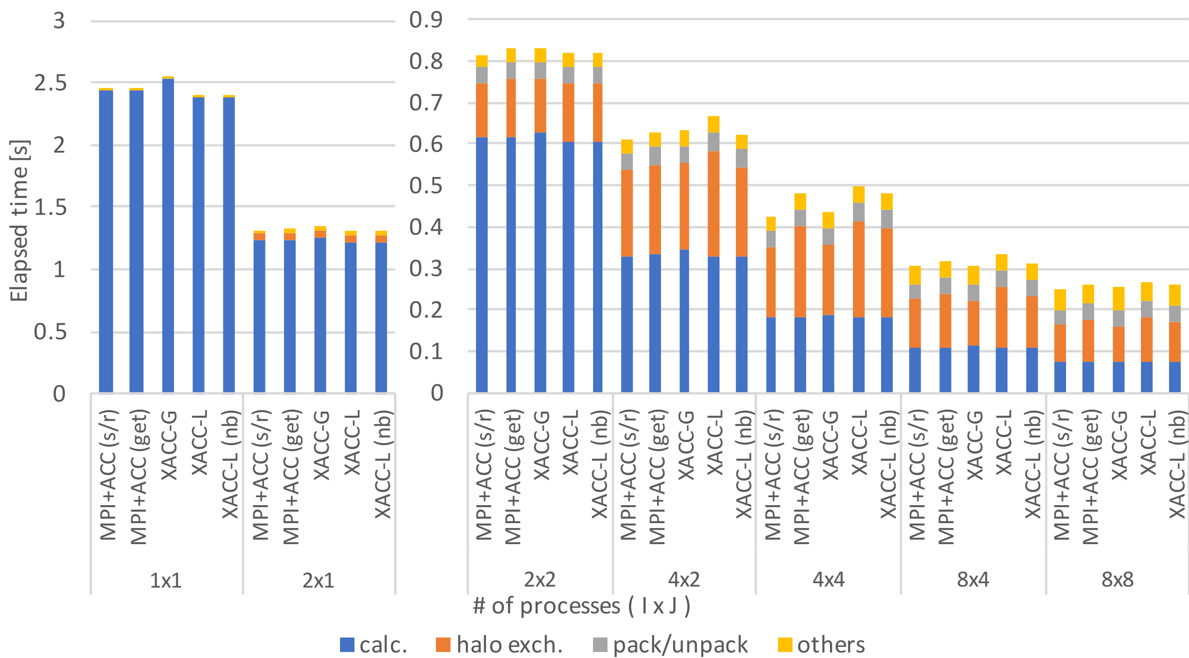
```
float p[MIMAX][MJMAX][MKMAX]:[NDZ0][NDY0][*];
float lo_sendbuf[MIMAX*MKMAX]:[NDZ0][NDY0][*];
float lo_recvbuf[MIMAX*MKMAX]:[NDZ0][NDY0][*];
float hi_sendbuf[MIMAX*MKMAX]:[NDZ0][NDY0][*];
float hi_recvbuf[MIMAX*MKMAX]:[NDZ0][NDY0][*];
#pragma acc declare create(p, lo_sendbuf, \
   lo_recvbuf,hi_sendbuf, hi_recvbuf)

/* ... */

sendp2_pack();

#pragma acc host_data use_device(lo_sendbuf,\
   lo_recvbuf, hi_sendbuf, hi_recvbuf)
{
  int len = imax*kmax;
  xmp_sync_images(num_npy, npy, NULL);

  if(mey > 1)
    lo_recvbuf[0:len]= lo_sendbuf[0:len]:[mez][mey-1][mex];

  if(mey < ndy)
    hi_recvbuf[0:len]= hi_sendbuf[0:len]:[mez][mey+1][mex];

  xmp_sync_images(num_npy, npy, NULL);
}

sendp2_unpack();
```

まず sendp2_pack() により不連続な袖領域を coarray の送信バッファにパックし，xmp_sync_images() により隣接イメージと同期をする．次に隣接イメージの送信バッファのデータを coarray get によって自イメージの受信バッファに転送する．その後 xmp_sync_images() で同期を取ったのちに，データをアンパックする．

図 5.14 と 5.15 に Himeno benchmark の性能および時間内訳を示す．凡例の “MPI+ACC (s/r)”, “MPI+ACC (get)”, “XACC-G”, “XACC-L”, “XACC-L (nb)” はそれぞれ MPI+OpenACC の send/recv 版, MPI+OpenACC の get 版, XACC グローバルビュー版, XACC ローカルビュー版, XACC ローカルビューのノンブロッキング通信版である． まず XACC グローバルビュー版は MPI+OpenACC send/recv

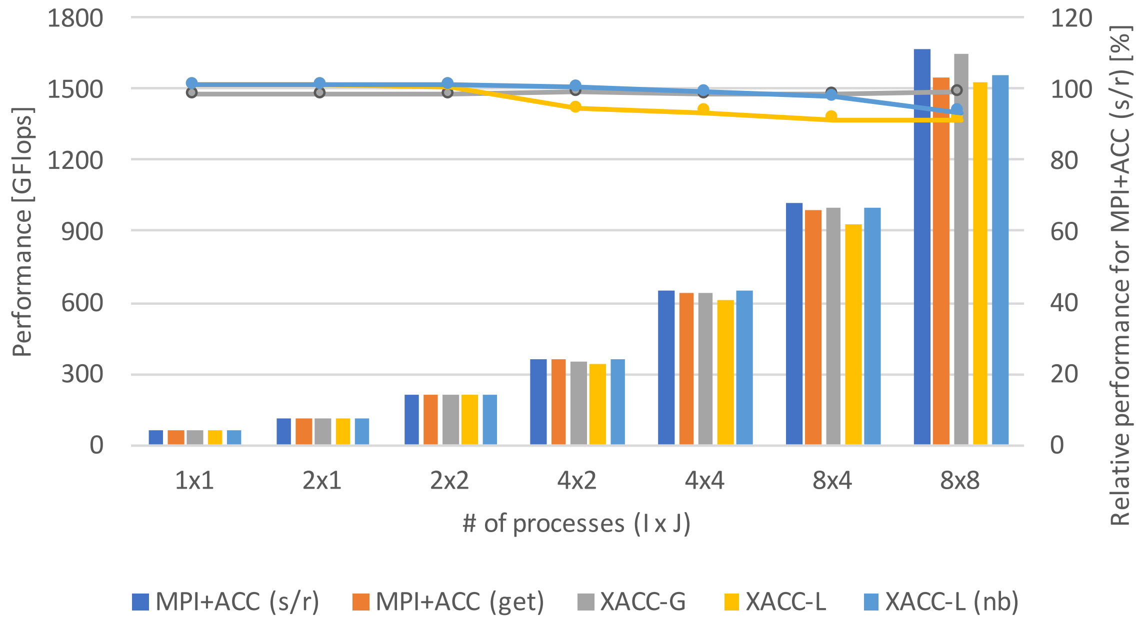


(a) 性能

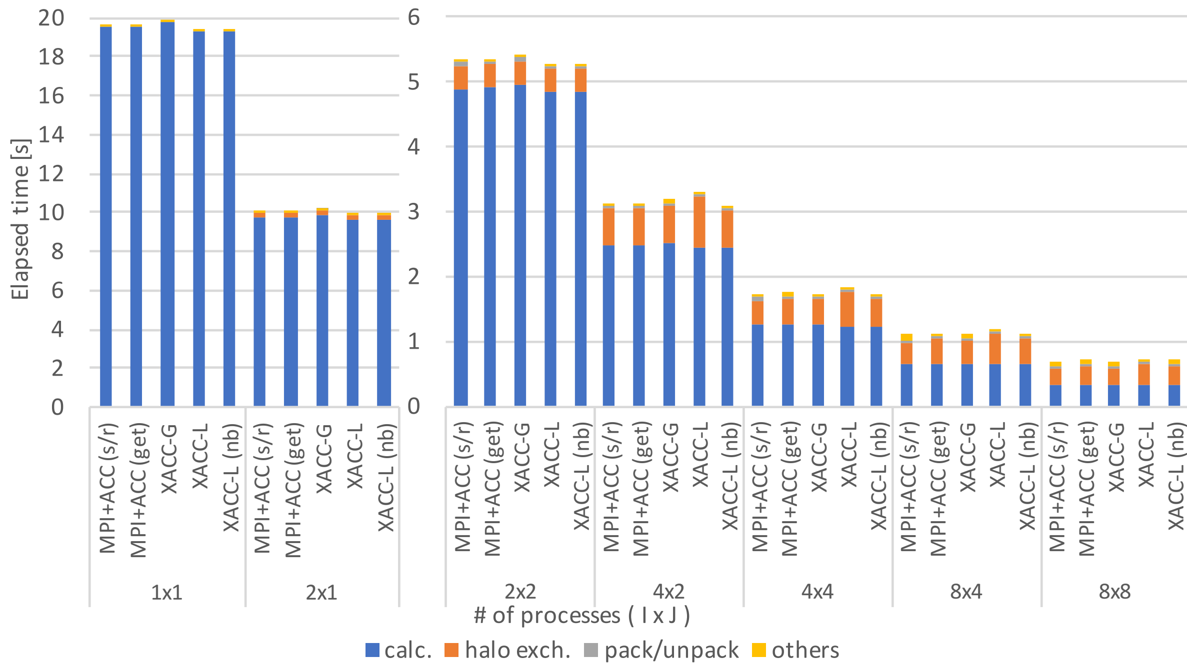


(b) 時間内訳

図 5.14: Himeno benchmark の性能と時間内訳 (size M)



(a) 性能



(b) 時間内訳

図 5.15: Himeno benchmark の性能と時間内訳 (size L)

版に対して計算時間が少し長くなるものの，サイズ M で 97% 以上，サイズ L で 98% 以上と安定して性能が出ている．次に XACC ローカルビュー版をみると，1×1 と 2×1 プロセスで MPI+OpenACC send/recv 版よりも 1–2% 性能が良くなっている．これは XACC コンパイラが多次元 coarray を 1 次元配列に変形することによって計算時間が短くなったためである．4×2–8×8 プロセスでは袖交換時間の増加によって性能が低下しており，サイズ M の 4×4 で 85%，サイズ L の 8×8 で 92% まで低下している．Himeno benchmark では各次元の袖交換を順に行うことで斜めの要素も更新されるようになっている．したがって同時にはある次元の隣接する 2 つのプロセスと通信することができる．しかしながら，5.2.4 項で述べたように現在の coarray 実装では各 coarray 通信はブロッキングとなるため，ローカルビュー版は同時に 1 つの隣接プロセスとしか通信できず性能が低下していた．coarray 通信がノンブロッキングであれば，袖交換の時間はより短くなるはずである．そこで，環境変数 “XMP_GET_NB” により coarray 通信をノンブロッキングにして計測をしたのが XACC ローカルビューのノンブロッキング通信版である．その結果，袖交換時間は MPI+OpenACC send/recv 版と同程度に短くなり，サイズ M で 89%，サイズ L で 93% まで改善した．しかしながら，ノンブロッキング版においてもサイズ M の 4×4 と 8×8 プロセスにおいては依然として性能が低い．ただ，MPI+OpenACC get 版も同様の性能であるため，send/recv と get の性能差によるものと言える．

5.4.2 NAS Parallel Benchmarks CG

問題サイズとして Class C(疎行列サイズ $750{,}000 \times 750{,}000$) と Class D (疎行列サイズ $1{,}500{,}000 \times 1{,}500{,}000$) を用いた．XACC グローバルビュー版ではソースコード 5.6 で示したように，通信に `reduction` や `gmove` 指示文を用いている．XACC ローカルビュー版の主な通信部分をソースコード 5.8 に示す．

ソースコード 5.8: NPB CG の XACC ローカルビューモデルによる通信

```
1 double w[na/num_proc_rows+2]:[*];
2 double q[na/num_proc_rows+2]:[*];
3 #pragma acc declare create(w,q)
4
5 /* ... */
6
7 #pragma acc host_data use_device(w, q)
8 {
9   /* array reduction */
10  for(i = l2npcols; i >= 1; i--){
11    int image = reduce_exch_proc[i-1] + 1;
12    int start = reduce_recv_starts[i-1] - 1;
13    int length = reduce_recv_lengths[i-1];
14    xmp_sync_image(image, NULL);
15    q[start:length] = w[start:length]:[image];
16    xmp_sync_image(image, NULL);
17
18 #pragma acc parallel loop
19    for(j = send_start-1; j < send_start+lengths-1; j++)
```

表 5.6: 青睡蓮におけるグローバルビューモデルの性能評価に用いたソフトウェア

Compiler	ICC 16.0.3, IntelMPI 5.1.3, PZSDK 3.0, Omni OpenACC compiler 1.2.1 for PEZY-SC

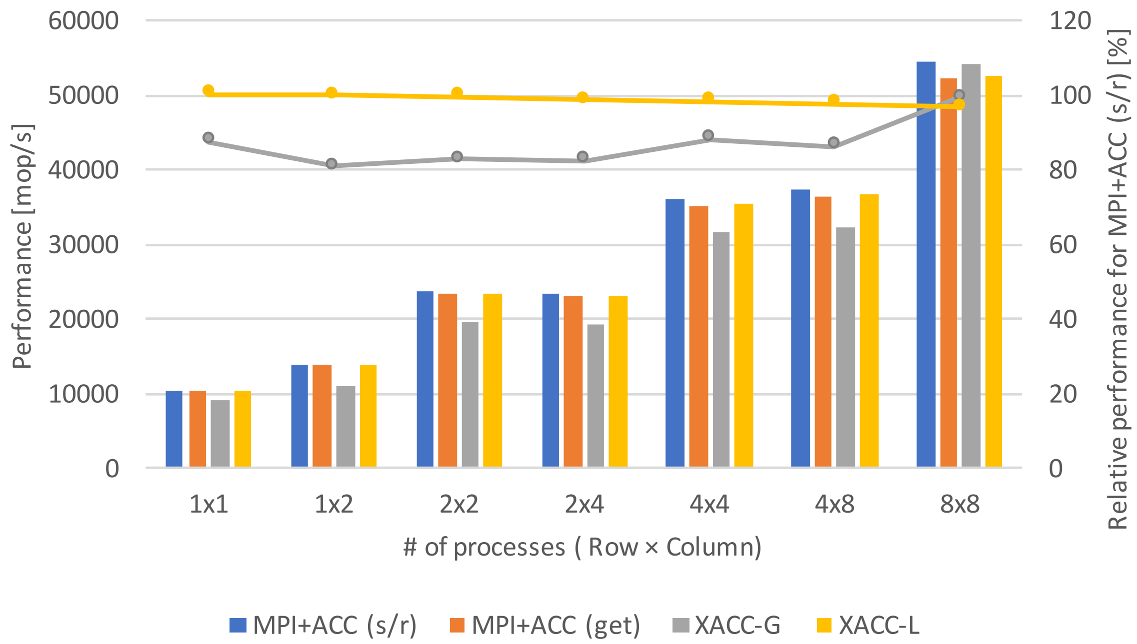
```
20       w[j] = w[j] + q[j];
21   }
22
23   /* column-distributed array <- row-distributed array */
24   if( l2npcols != 0 ) {
25     xmp_sync_image(exch_proc+1, NULL);
26     q[0:send_len] = w[send_start-1:send_len]:[exch_proc+1];
27     xmp_sync_image(exch_proc+1, NULL);
28   }else{
29 #pragma acc parallel loop
30     for(j=0; j < exch_recv_length; j++)
31       q[j] = w[j];
32   }
33 }
```

10–21 行目では配列リダクションを recursive-doubling 法を用いてアクセラレータ上での get 通信と計算で行う．25–27 行目では行分散の配列 *w[]* から列分散の配列 *q[]* への代入を get 通信と sync_image() によって行う．なお，このベンチマークでは一度に 1 プロセスとのみ通信して直後に同期を取るため XACC ローカルビュー版のノンブロッキング版の評価は行っていない．

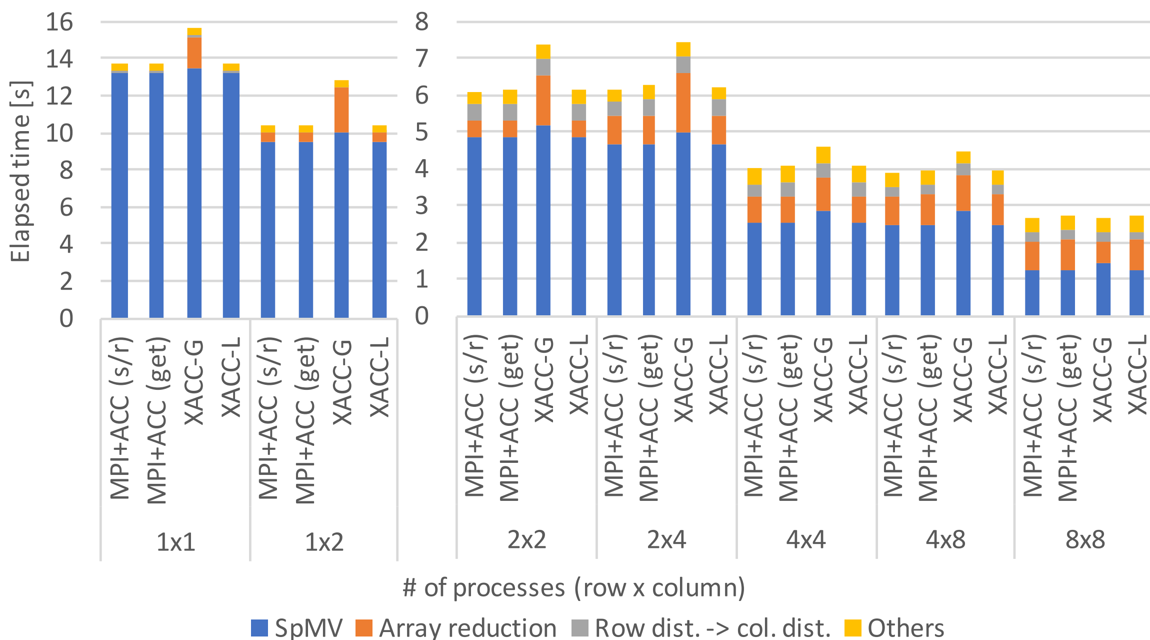
図 5.16 と 5.17 は NPB CG の Class C と D の性能と時間内訳である．XACC グローバルビュ版は大きく性能が低下している部分があるのに対して，XACC ローカルビュー版は MPI+OpenACC send/recv 版と比べて Class C で 97% 以上，Class D で 99% 以上の性能を達成した．Class C でプロセス数が多い時に性能がやや低下しているが，MPI+OpenACC get 版も同様に低下しているため，send/recv と get の性能差であると言える．NPB CG では通信は同時に 1 プロセスのみに対して行うので，通信がブロッキングであることによる Himeno benchmark で見られたような性能低下は起こらなかった．

5.5　PEZY-SC クラスタにおけるグローバルビューモデルの性能評価

評価環境には KEK の青睡蓮を用いた．その際のノード構成は表 4.3 に示した通りで，使用したソフトウェアを表 5.6 に示す．現在 PEZY-SC 向けには `reflect` 指示文のみ対応しているため，ベンチマークとして Himeno Benchmark のみを用いた．MPI+PZCLv1, MPI+PZCLv2, MPI+OpenACC, XACC の 4 種類のコードで比較を行う．MPI+PZCLv1 はオリジナルの Himeno benchmark と同じ配列配置を用いており，MPI+PZCLv2 は XACC で用いられる配列配置を用いている．XACC 版はソースコード 5.5 とほぼ同一であるが，オフロードする 3 重ループの並列化指定の変更と同期を行う拡張指示文の追加をした．問題サイズは Large で反復回数は 1000 回とした．図 5.18 に性能および 1 反復あたりの実行時間内訳を，図 5.19 に通信のみの時間内訳を示す．なお MPI+PZCLv1 については 4×4 までの評価である．まずはじめ

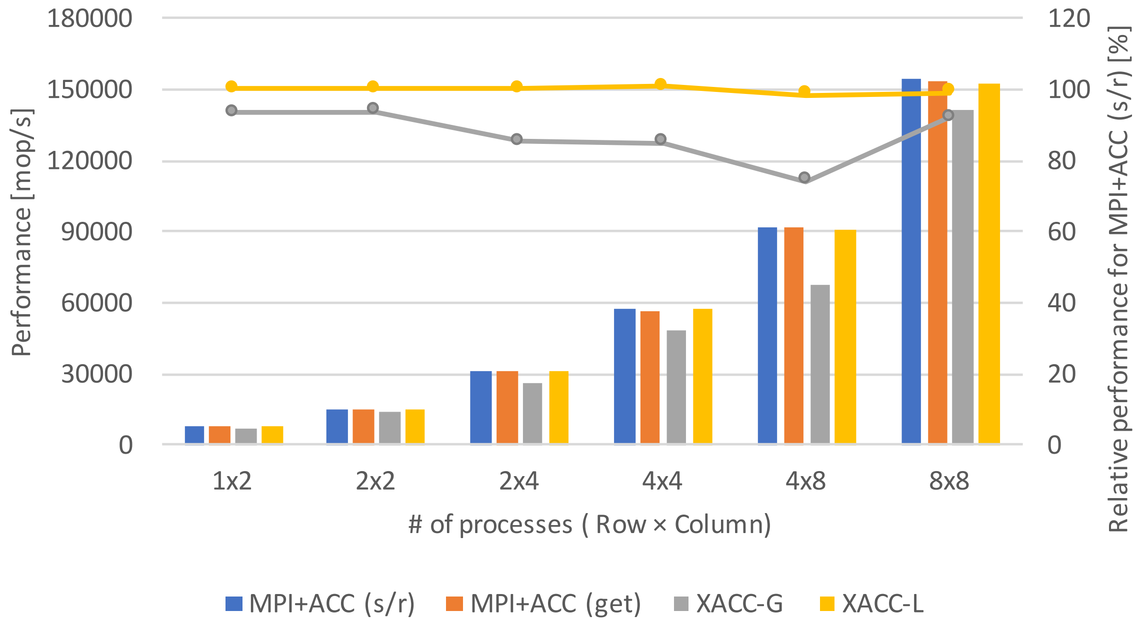


(a) 性能

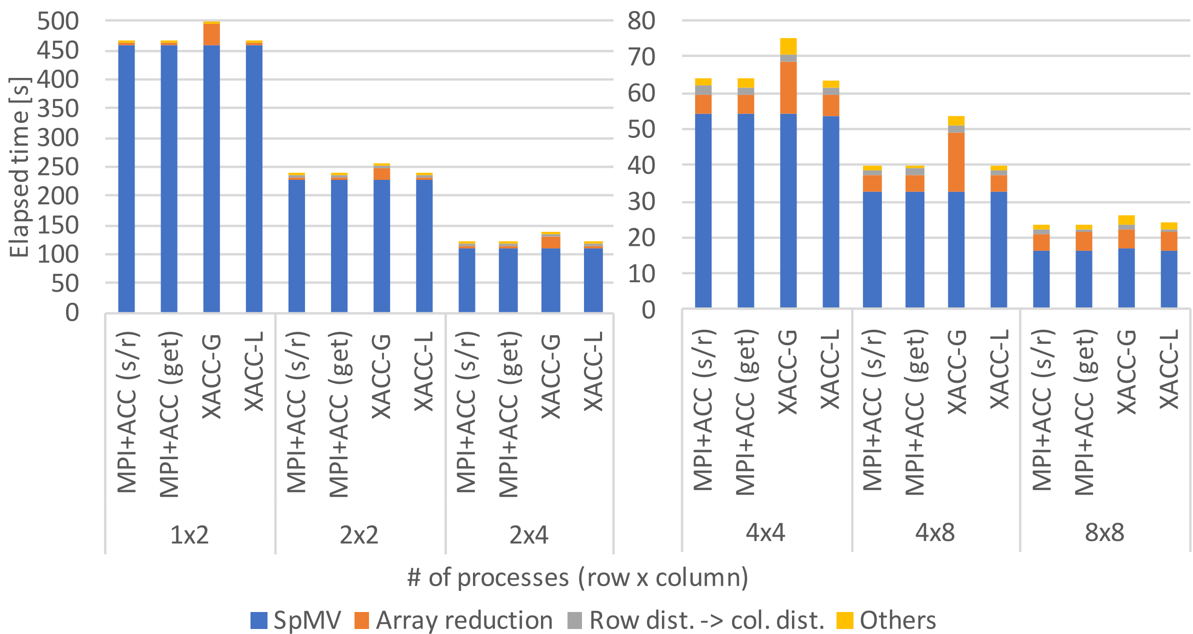


(b) 時間内訳

図 5.16: NPB CG (Class C) の性能と時間内訳

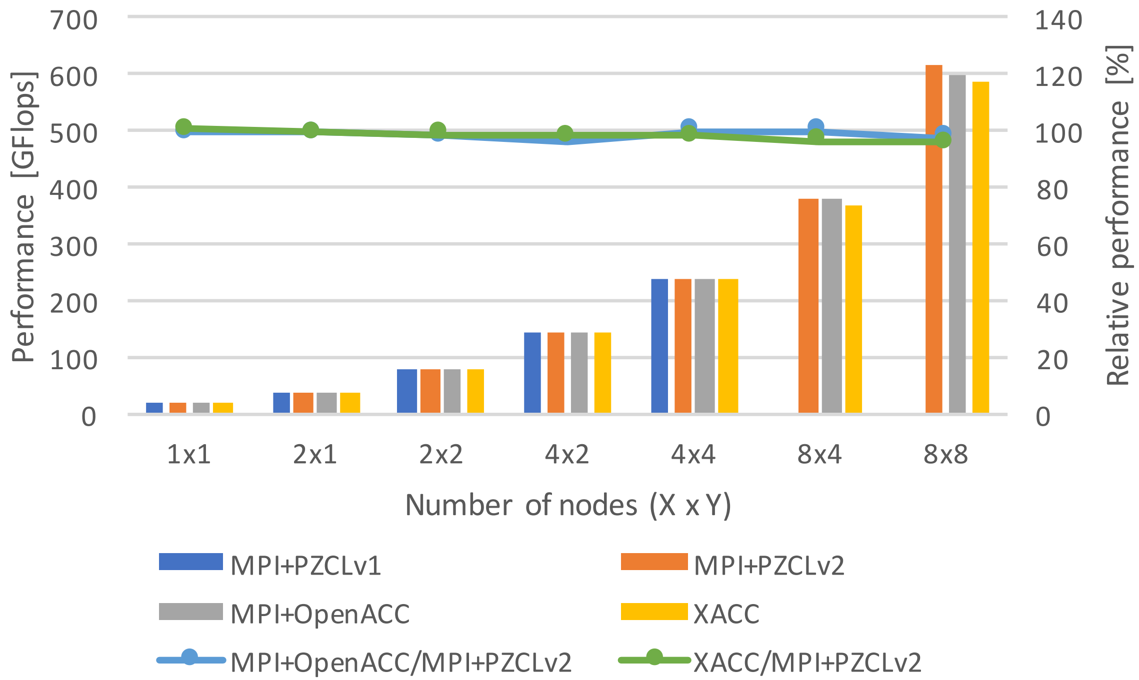


(a) 性能

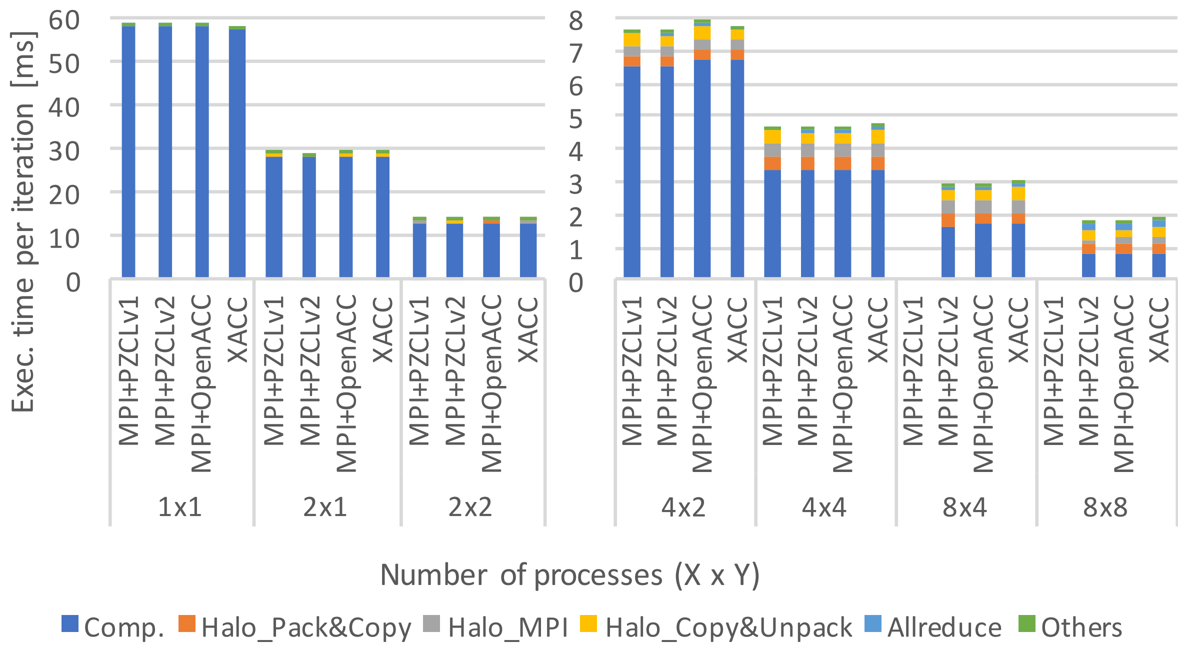


(b) 時間内訳

図 5.17: NPB CG (Class D) の性能と時間内訳



(a) 性能



(b) 時間内訳

図 5.18: PEZY-SC クラスタにおける Himeno benchmark (Size L) の性能と時間内訳

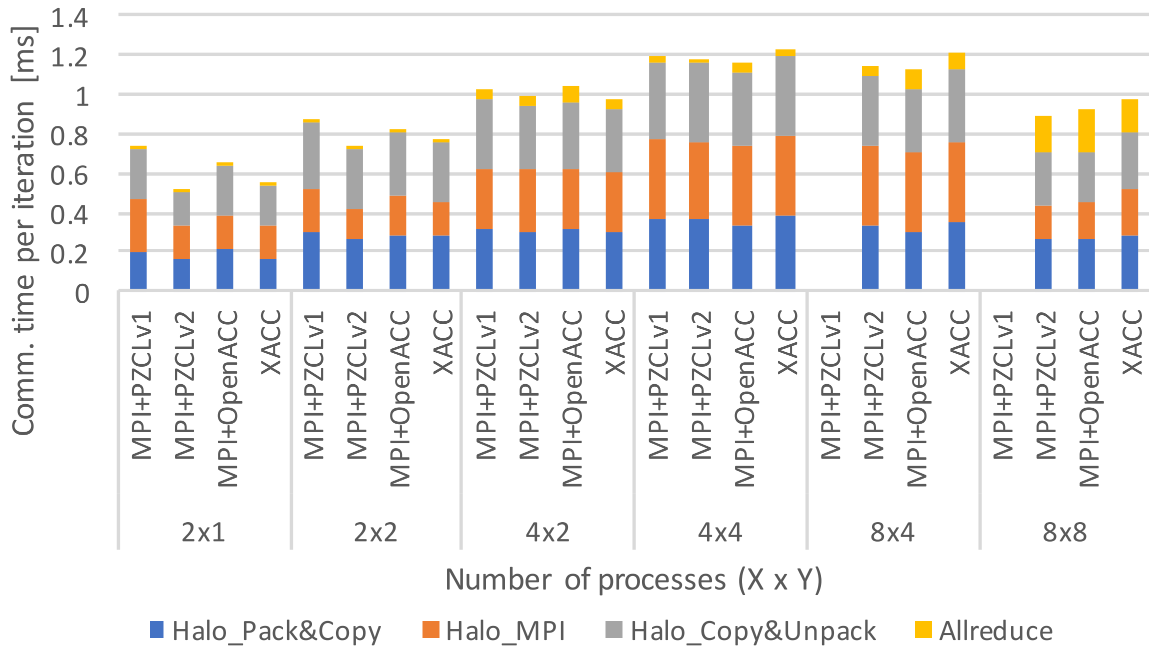


図 5.19: PEZY-SC クラスタにおける Himeno benchmark (Size L) の通信時間内訳

に MPI+PZCLv1 と v2 について比較すると袖交換で大きな差があり，2×1 では 1.43 倍も時間が増加している．両者は確保する配列サイズは同じで，使用する部分が異なるため袖領域部分の違いによって性能が低下していると考え調査をしたが，原因は不明であった．性能が出ている MPI+PZCLv2 をベースに他の実装と比較すると，MPI+OpenACC では 96-100% の性能が出ている．計算時間は OpenACC を使うことにより最大 3% ほど増加している．また通信時間は 95–126% とプロセス数によって大きく変化している．こちらも通信する際のメモリ位置の違いが原因であると考えられる．MPI+PZCL においては ij 平面の送受信バッファを別に確保しているが，MPI+OpenACC ではホスト側の配列 *p[]* を更新するようにしか記述ができない．XACC 版は MPI+PZCLv2 と比較して 96–100% の性能が得られた．最大 2% ほどの増加であり，袖通信時間は最大 14% 増加していた．プロセス数増加とともにやや性能が落ちるものの全体的には MPI+PZCL 版と同等の性能が得られた．

5.6　生産性評価

5.6.1　グローバルビューモデル

XACC グローバルビューモデルでは逐次コードへの指示文の追加のみで，複数ノードおよびデバイスへのデータと処理のオフロードが可能である．またノード間の分散は XMP で，デバイスへのオフロードは OpenACC で記述するため，分散とオフロードを区別してプログラムを書きやすい．XACC グローバルビューモデルの記述のしやすさを定量的に評価するためにベンチマークのコードの行数を数えた．Himeno Benchmark の行数を表 5.7 に，逐次コードからの変更行数を表 5.8 に示す．XACC グロー

バルビューの全体の行数は MPI+OpenACC の 60%，MPI+OpenACC (pack) の 51% である．XACC や XMP+OpenACC では MPI+OpenACC に比べて非常に少ない修正で記述が可能である．MPI+OpenACC ではデータ・処理分散のために約 60 行ほど必要であるのに対して，XACC では指示文によるデータ・処理分散の記述により 25 行で済んでいる．また通信に関しては，袖通信は通信相手と回数が多いため MPI+OpenACC は全体の 33%，pack を加えたコードでは 42% の行数を必要としているのに対して，XMP+OpenACC は reflect 指示文により簡易に記述可能であるため全体の 19% に抑えられている．さらに XMP+OpenACC では袖通信の際に必要となる部分のみをホスト・デバイス間でコピーするために逐次コードへの変更と OpenACC 指示文が必要であるが，XACC ではデバイス間の reflect 指示文 1 行で済むため全体の 5% の行数となり，さらに記述が簡易となった．XACC で指示文以外の逐次コードに加えた変更は時刻取得関数の変更，XMP 指示文の効果範囲を指定するための中カッコ，ヘッダのインクルードの追加等であり，ほぼ逐次コードをそのまま利用できた．

表 5.7: Himeno Benchmark のコード行数 (内数は通信関連の行数)

	XMP	OpenACC	指示文以外
逐次コード	-	-	146
MPI+OpenACC	-	13 (4)	315 (108)
MPI+OpenACC (pack)	-	21 (12)	369 (162)
XMP+OpenACC	33 (7)	21 (12)	181 (26)
XACC global-view	34 (9)	9 (0)	155 (0)

表 5.8: Himeno Benchmark の逐次コードへの変更行数 (指示文は除く)

	追加	修正	削除
MPI+OpenACC	180	12	11
MPI+OpenACC (pack)	234	12	11
XMP+OpenACC	41	5	6
XACC global-view	15	5	6

次に NPB CG の行数を表 5.9 に，逐次コードからの変更行数を表 5.10 に示す．MPI+OpenACC に対する XMP+OpenACC の全体の行数は 80%，XACC は 79% である．XMP+OpenACC と XACC との差は通信時にホストとデバイス間でコピーを行うための OpenACC update 指示文のみである．分散や通信のために逐次コードに対して加えた変更は，MPI+OpenACC と比べて XMP+OpenACC や XACC では 200 行程少ない．全体に対する通信の割合を見ると，配列リダクションや行分割から列分割への代入処理を send/recv で記述した MPI+OpenACC は 30% であるのに対して，reduction,gmove 指示文で記述した XMP+OpenACC は 5.3%，さらにデバイス間の通信とした XACC では 4.1% であった．XMP+OpenACC や XACC における指示文以外の変更のうち，約半分はプロセス数が 2 冪であるかのチェックで，残りは分散する配列をグローバル変数として宣言するための変更，疎行列の列インデックスのオフセットの修

正，2 つのノルムのリダクションを一度に行うための一時配列への代入と参照，ランク 0 のみが実行する部分の条件文，OpenACC 指示文の範囲を指定するための中カッコ等である．プログラムの実際の動作に大きく影響するのは疎行列の列インデックスのオフセットの修正のみであるため，記述のコストは低いと言える．2 つのベンチマークにおけるコードの比較から，XACC や XMP+OpenACC では指示文による

表 5.9: NPB CG のコード行数 (内数は通信関連の行数)

	XMP	OpenACC	指示文以外
逐次コード	-	-	444
MPI+OpenACC	-	24 (5)	748 (224)
XMP+OpenACC	48 (20)	28 (8)	541 (5)
XACC global-view	48 (20)	20 (0)	541 (5)

表 5.10: NPB CG の逐次コードへの変更行数 (指示文は除く)

	追加	修正	削除
MPI+OpenACC	304	43	0
XMP+OpenACC	109	16	12
XACC global-view	109	16	12

データ・分散の記述により各ノードのデータや処理の範囲を計算する処理が必要ないため，逐次コードへの変更が少なくなることや，袖通信のような典型的な通信パターンに対して，専用の指示文を用いることで非常に簡潔に記述可能であることが分かる．さらに XMP+OpenACC ではデバイス間の通信の際にホスト・デバイス間の通信を記述する必要があるが，XACC では必要がないため XMP+OpenACC よりも簡易に記述できる．以上のことから，XACC は MPI+OpenACC や XMP+OpenACC によるプログラミングよりも生産性が高いと言える．

5.6.2 ローカルビューモデル

XACC ローカルビューモデルは coarray を用いて，通信を配列代入文形式で記述できる．MPI で通信を行う場合は関数の引数に，バッファ，要素型，要素数，タグ，ステータス，ランクなど多くの情報を渡す必要があり，記述が複雑で間違えやすい．coarray で通信を行う場合には，coarray に，配列範囲，対象イメージ番号を指定するだけで良いため簡易で間違えにくい．表 5.11 に逐次，MPI+OpenACC，XACC グローバルビュー，XACC ローカルビュー版の Himeno benchmark と NPB CG の SLOC を示す．なお，コードの若干の変更により表 5.7 と表 5.9 の行数と差のある箇所がある．XACC グローバルビュー版では MPI+OpenACC よりも大幅に行数が減っているが，XACC ローカルビュー版ではほとんど差がない．ローカルビュー版は基本的には MPI 版と同様にユーザがデータ・処理分散を行なうため MPI 版と同様の行数となる．

表 5.11: Himeno benchmark と NPB CG の SLOC

	Himeno benchmark	NPB CG
逐次コード	146	444
MPI+OpenACC	398	769
XACC global-view model	198	609
XACC local-view model	395	768

5.7　考察

性能と生産性の評価からグローバルビューモデルはグリッド領域分割のプログラムを指示文により簡易に記述でき，かつ袖交換のような典型的な通信であれば十分な性能を達成できる．しかしながら，複雑な通信は記述ができず，NPB CG で見られたようなプログラム固有の最適化ができず，性能が低下する．一方で，ローカルビューモデルは複雑な分散と coarray による任意の通信を記述できるが，MPI と同様に分散と通信を記述するためコード量はグローバルビューモデルほど少なくない．したがって，グローバルビューモデルは典型的な分散・通信を行うプログラムを簡易に記述したい時に適しており，逆にローカルビューモデルは複雑な分散・通信や性能が特に重要となるプログラムに適している．

現在の実装では coarray 通信がブロッキングになるという問題がある．本来はコンパイラがコンパイル時もしくは実行時にローカルの読み書きと通信の依存を解析して，可能な部分をノンブロッキングにすることが望ましい．文献 [42] では UPC のランタイムにおいてリモートへの書き込み・読み込みを記録することで自動的に RMA をノンブロッキングにしている．この手法ではリモート側のデータの依存は解析可能であるが，ローカル側は解析できないので必ずローカルではブロッキングとなる．また特に C 言語ではポインタエイリアスがあるため根本的に解析が困難である．そこで，指示文を用いてコンパイラに最適化のヒントを与えることが考えられる．例えば，Cray compiler では次の fence 命令まで可能な限り同期を遅らせるよう指定する `defer_sync` 指示文が用意されている [43]．しかしながら次の fence が何を指すのかが分からず仕様が明確でなく，またローカル側のみブロッキングにしてリモート側はブロッキングしないという書き方ができない．そこで新たに提案する `nonblock`，`wait_nonblock` 指示文の構文を図 5.20 に示す．`nonblock` 指示文は直後の対象コード内の coarray 通信の完了を保証しなくても良いことをコンパイラに伝える．引数がない場合はローカルとリモート共に通信の完了を保証しなくてもよく，引数に `remote` が指定された場合はリモートのみ完了を保証しなくても良い．また `nonblock` 指示文が指定された通信は同期をまたがない限り他の文との順序が不定となる．完了保証がされなかった通信は，次に出会う同期 (sync memory, sync image(s), sync all) または `wait_nonblock` 指示文で完了を待つことができる．`wait_nonblock` 指示文は先行するノンブロッキングとなった通信の完了を保証する．引数がない場合はローカルとリモートの完了を保証し，引数に `local` が指定された場合はローカルの完了のみを保証する．

nonblock construct
```
 [F]   !$xmp nonblock( [remote])
          ...
       [!$xmp end nonblock]
 [C]   #pragma xmp nonblock ( [remote])
          structured-block
```

wait_nonblock directive
```
 [F]   !$xmp wait_nonblock( [local])
 [C]   #pragma xmp wait_nonblock ( [local])
```

図 5.20: nonblock, wait_nonblock 指示文のシンタックス

5.8 関連研究

TCA を用いた XACC の通信の実装と評価が行われている [13]．袖領域通信を TCA により実装し，MPI を用いた実装よりも高い性能を達成している．また，本研究の MPI 実装をベースに MPI と TCA のハイブリッドによる XACC の通信の実装も行われている [44]．しかしながら現在 TCA を利用できるクラスタは HA-PACS/TCA のみであるため，一般的な GPU クラスタでは利用することができない．本研究では一般的な GPU クラスタでも動作するよう GPU 間通信を MPI と CUDA により実装し，その場合でも XACC が有効であることを示した．

他の PGAS 言語においては Unified Parallel C (UPC) や OpenSHMEM で GPU のメモリへのアクセスや管理を行える拡張が行われており，共に GPU 間の通信に対応している [45, 46]．UPC は言語拡張，OpenSHMEM はライブラリベースの PGAS 言語であり，GPU のプログラミングに指示文ベースの OpenACC を用いる場合には，異なるプログラミング手法の組み合わせとなる．XACC では XMP が指示文ベースの PGAS 言語であるため，同じ指示文ベースの OpenACC との親和性が高い．そのため，既存の XMP のコードもしくは OpenACC のコードからシームレスに記述できるという利点がある．さらに UPC や X10 や Chapel では構文を拡張することにより GPU のプログラミングを可能にしている [47, 48, 49]．Chapel の拡張では 1 ノード上の GPU での実行にのみ対応している．UPC や X10 の拡張では複数ノード上の GPU の利用に対応しているが，ホスト・GPU 間の通信のみで GPU 間の通信については考慮されていない．XACC では複数ノード上のデバイスを利用可能であり，さらにデバイス間の通信にも対応することで，大規模システムにおいても高い性能向上が見込まれる．

CAFe はヘテロジニアスコンピューティング向けの Coarray Fortran 拡張である [50]．CAFe では image に属する *subimages* を導入している．タスクや並列ループ実行を subimages で実行することができ，それがアクセラレータでの実行となる．MPI+OpenACC のようなハイブリッドな手法と比べて CAFe はハイレベルな文法によってコンパイラ最適化が可能である．しかしながら，image は自分に属する subimage と coarray により通信ができるが，他の image が持つ subimage とは通信ができない．XACC ではアクセ

ラレータ間の通信が記述可能であるため，より高速なアクセラレータ間通信が可能であり，アクセラレータプログラミングを OpenACC にすることでプログラマがチューニングする余地を残している．

Coarray のアクセラレータへの拡張が提案されている [51]．この拡張ではホストとアクセラレータの両方からアクセス可能な coarray を提供しており，“accelerated” という変数属性で指定可能である．ホストとアクセラレータがデータを共有しているため，リモート image への coarray 通信を行えばリモートのアクセラレータからその値を読むことができる．実装では，CUDA の Unified Memory いうホストと GPU の両方からアクセス可能なメモリにより実現されている．SUMMA アルゴリズムを用いた行列積による評価では一般的な MPI+CUDA 実装よりも高い性能が達成されている．ユーザはホストとアクセラレータの両方から coarray にアクセスできるため記述が簡易である．しかしながらホストとアクセラレータ間のデータ転送がアクセスするタイミングで自動的に行われるため予期しない性能低下が起こり得る．XACC は OpenACC のメモリモデルと同じく，アクセラレータが別のメモリを持っている場合はホストとアクセラレータの coarray を区別して記述するようになっているため，プログラマが予期しないデータ転送は起きない．

Himeno benchmark が OpenACC と Fortran の coarray により実装されている [52]. その実装ではホストメモリ間の通信が coarray で記述されており，ホストとデバイスの通信は OpenACC `update` 指示文で記述されている．本研究のベンチマークではデバイスメモリ間での通信を coarray で記述している．

第 6 章

アプリケーションを用いた XcalableACC の性能と生産性の評価

本章では，PGAS 言語 XACC が実際にアプリケーション開発の生産性を高めかつ高い性能を得られるかどうかを確かめるため，流体力学と格子 QCD の 2 つのミニアプリケーションを用いて評価を行う．

6.1　流体力学ミニアプリケーション CloverLeaf

CloverLeaf は流体力学ミニアプリケーションであり Atomic Weapons Establishment と Warwick 大学により開発され，UK Mini-App Consortium において公開されている [53]．このアプリケーションは圧縮性オイラー方程式を 2 次元直交座標系において解いており，有限体積法を用いて二次精度で求めている．大きな特徴は，スタッガード格子を用いていることである．これは物理量によって異なる位置に配置する方法で，図 6.1 に示す様に 圧力 (pressure) 等はセルの中心に，速度 (velocity) 等はセルの頂点に配置される．すべての物理量を同じ位置に配置するよりも安定性が高いというメリットがある一方で，実装が難しくなるデメリットもある．時間発展では，以下の 2 つのステップを繰り返す．1 つ目は Lagrangian ステップで，予測子修正子法を用いてタイムステップを進める．2 つ目は移流ステップで，1 つ目のステップで流速により移動した物理量をセルに再配置 (remap) する．

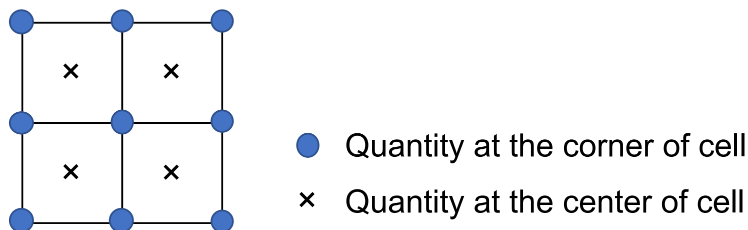


図 6.1: CloverLeaf におけるスタッガード格子配置

6.1.1　XACC による実装

CloverLeaf の MPI 実装は 2 次元領域を 2 次元ブロックで分散し，通信は主に袖領域の交換である．そこで XACC による実装ではそのような分割と通信を自然に記述可能なグローバルビューモデルを用いることにした．ソースコード 6.1 は XACC 版 CloverLeaf の配列定義と確保部分のコードである．

ソースコード 6.1: CloverLeaf の XACC 実装における宣言・確保

```
1 !$xmp nodes p(num_nodes_x,num_nodes_y)
2 !$xmp template t(:,:)
3 !$xmp distribute t(gblock(*),gblock(*)) onto p
4
5 REAL(KIND=8),ALLOCATABLE,DIMENSION(:,:) :: pressure, xvel0, ...
6
7 !$xmp align (i,j) with t(i,j) :: pressure, xvel0, ...
8 !$xmp shadow (2:2,2:2) :: pressure, ...
9 !$xmp shadow (2:3,2:3) :: xvel0, ...
10
11 INTEGER,ALLOCATABLE :: bsize_x(num_nodes_x), bsize_y(num_nodes_y)
12 ...
13
14 ! read input parameters (x_min, x_max, etc.)
15 ...
16
17 ! set blocksize array (bsize_x, bsize_y)
18 CALL set_blocksize(x_min, x_max, y_min, y_max)
19
20 !$xmp template_fix (gblock(bsize_x), gblock(bsize_y))&
21 !$xmp t(x_min-2:x_max+3, y_min-2:y_max+3)
22
23 ALLOCATE(pressure(x_min-2:x_max+2,y_min-2:y_max+2))
24 ALLOCATE(xvel0    (x_min-2:x_max+3,y_min-2:y_max+3))
25 ...
26
27 ! Allocate data on accelerator
28 !$acc enter data copyin(pressure,xvel0,...)
```

1 行目は 2 次元のノード集合 *p(num_nodes_x, num_nodes_y)* を定義する．処理するセルの数を入力ファイルから取得して決めるために，2 行目ではサイズ不定のテンプレート *t(:,:)* を定義し，3 行目ではそれをノード集合 *p* にブロックサイズ指定がない `gblock` 分散する．入力ファイル “clover.in” からパラメータ (*x_min, x_max, y_min, y_max*) を読み込んだ後，20–21 行目でテンプレート *t* のサイズおよび分散を指定する．そのサイズは *(x_min-2:x_max+3, y_min-2:y_max+3)* で，これは最も大きく確保する配列サイズと等しい (詳細は後述する)．処理の主となる部分は $(x_min : x_max, y_min : y_max)$ であるため，この部分を `gblock` 分散を用いて各ノードで均等になるよう分散する．`gblock` 分散用のブロックサイズ *bsize_x,bsize_y* は，最下端のノードが 2 多く，最上端のノードが 3 多くなるようにする．例として，*x_min = 1, x_max = 100* の場合には，*bsize_x = (/27, 25, 25, 28/)* となる．

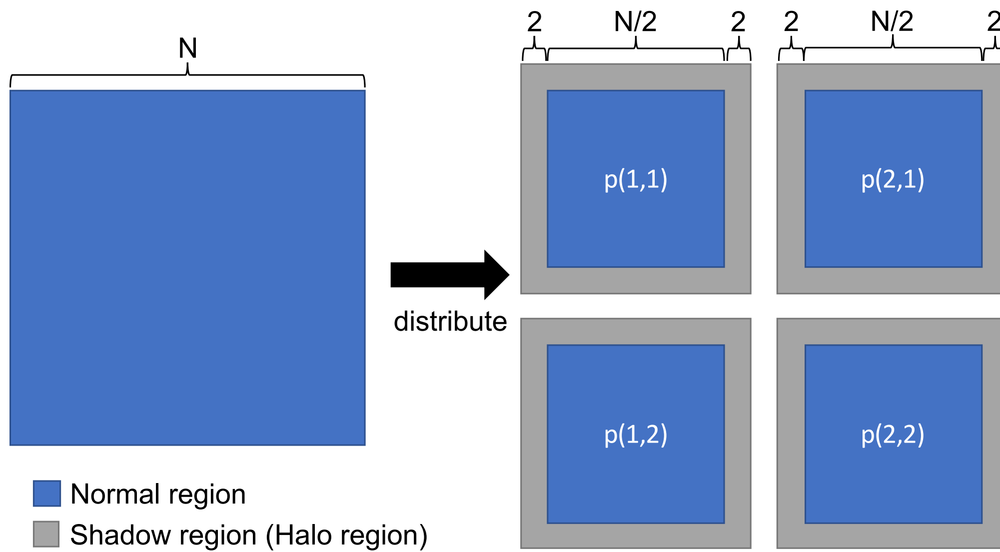


図 6.2: CloverLeaf の XACC 実装におけるセル中心の配列の分散

物理量を表す配列，例えば *pressure* や *xvel0* は 2 次元配列として定義され，パラメータを読み込んだ後にメモリに確保する．セルの中心に配置する物理量の配列サイズは *(x_min-2:x_max+2, y_min-2:y_max+2)* で，外側に境界条件を保持する幅が 2 の領域が追加されている．セルの頂点に配置する物理量の配列サイズは *(x_min-2:x_max+3, y_min-2:y_max+3)* で，セル中心の物理量の配列よりも各次元 1 要素多い．なぜなら，セルの数もしくはセルの中心の数が $(m \times n)$ の時にはセルの頂点の数は $((m+1) \times (n+1))$ になるからである．このセル頂点量の配列における拡張された部分を以後 “extra region” と呼ぶ．

7 行目は物理量配列を分散するよう指定しており，8–9 行目でシャドウ領域を設定する．以後の説明では，シャドウ領域は shadow 指示文で確保する領域という意味で，袖領域は隣接するノードのデータを通信で複製し計算に用いるための領域という意味で，区別して用いる．図 6.2 はセル中心の物理量の配列分散を示している．図の左側が逐次コードにおける配列を表しており，X 次元のサイズは $N = (x_max + 2) - (x_min - 2) + 1$ である．図の右側が 2×2 ノードの場合の配列分散を表す．各ノードの分散配列は各次元の下端と上端に幅が 2 のシャドウ領域が追加されており，X 次元のサイズは $N/2 + 4$ となる．そのシャドウ領域のすべてが袖領域として利用される．

図 6.3 はセルの頂点の物理量の配列分散である．図の左側は逐次コードにおける配列を表しており，extra region が必要なため配列サイズは X 次元で $N + 1$ となる．図の右側はその配列を 2×2 ノードに分散した場合を表しており，これはやや複雑である．分散配列は下端に 2，上端に 3 の幅のシャドウ領域が追加されている．上端のシャドウの幅のみが 3 になっているのは，配列の分割辺上の頂点は重複して保持する必要があり，extra regions (図の濃灰色部分) が新たに必要となるためである．シャドウ領域のうち，extra region として用いられる部分は計算においては袖領域ではなく通常の領域として扱われるため，袖交換の際には更新の必要がない．各ノードにおける分散された配列の X 次元の大きさは *p(1,1), p(1,2)* では $N/2 + 5$，*p(2,1), p(2,2)* では $N/2 + 6$ である．

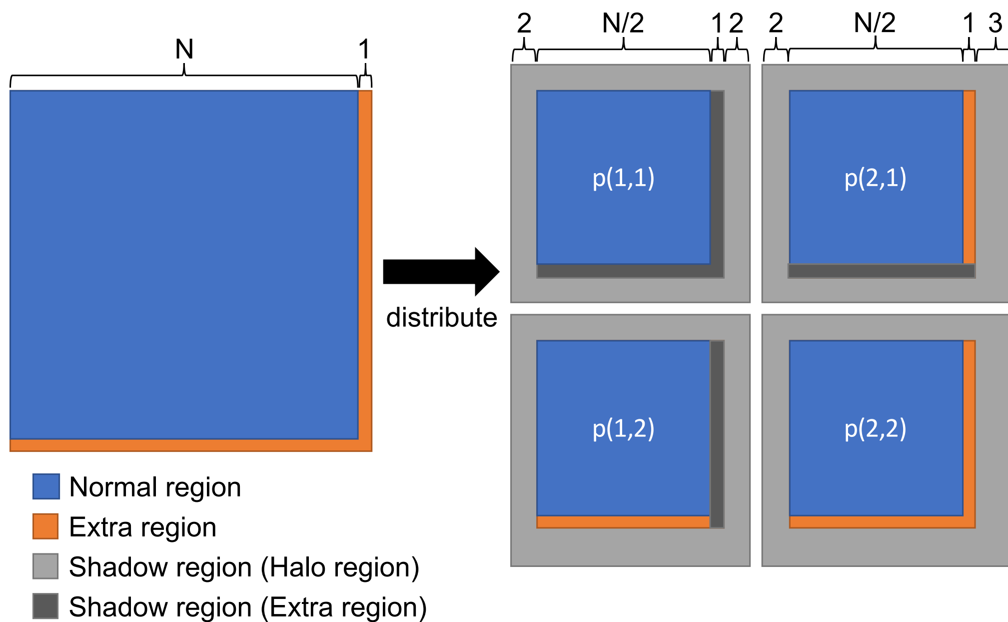


図 6.3: CloverLeaf の XACC 実装におけるセル頂点の配列の分散

最後に 28 行目で，OpenACC の `enter data` 指示文により配列はアクセラレータメモリ上にも確保される．事前に確保することで，カーネル実行毎にアクセラレータメモリに確保・転送する必要がなくなる．

次に，計算の分散とオフロード処理について説明する．データ分散と同じように，2 重ループを 2 次元ノード集合に分散し，各ノードのアクセラレータにオフロードするようにしている．アクセラレータ上でのループ並列は既存の MPI+OpenACC による CloverLeaf 実装で用いられているのと同じように指定している．ソースコード 6.2 は計算サブルーチンの一つである *accelerate_kernel()* の一部である．

ソースコード 6.2: CloverLeaf の XACC 実装における計算

```
1 SUBROUTINE accelerate_kernel(x_min,x_max,y_min,y_max,pressure,xvel0,...)
2
3   INTEGER :: x_min,x_max,y_min,y_max
4   REAL(KIND=8),DIMENSION(x_min:x_max,y_min:y_max) :: pressure, xvel0, ...
5
6 !$xmp template t(x_min-2:x_max+3,y_min-2:y_max+3)
7 !$xmp distribute t(gblock(bsize_x,bsize_y) onto p
8 !$xmp align (i,j) with t(i,j) :: pressure, xvel0, ...
9 !$xmp shadow (2:2,2:2) :: pressure, ...
10 !$xmp shadow (2:3,2:3) :: xvel0, ...
11 ...
12
13 !$acc data present(pressure,xvel0,...)
14
15 !$xmp loop (j,k) on t(j,k) expand(0:1,0:1)
16 !$acc kernels loop independent
```

```
17   DO k=y_min,y_max+1
18 !$acc loop independent
19     DO j=x_min,x_max+1
20       ...
21       xvel1(j,k)=xvel0(j,k)-...&
22         (pressure(j,k)-pressure(j-1,k))...
23       ...
24     ENDDO
25   ENDDO
26
27 !$acc end data
28 END SUBROUTINE
```

サブルーチンの引数の配列 (例:*pressure, xvel0*) は分散配列であるため，6–10 行目ではその分散に関して改めて指定する．13 行目では OpenACC `data present` 指示文ですでにその分散配列がアクセラレータメモリ上に存在すると明示している．CloverLeaf における主な計算は 2 重ループで構成されている．15 行目では，XMP `loop` 指示文により 2 重ループの分散を指定する．ループによっては，セル頂点の物理量のための extra region を処理するために，他よりローカルでのループ範囲を大きくする必要がある．そのような場合には，`loop` 指示文に `expand` 節を追加して，各ノードのループ範囲を拡大するよう指定する．16 行目では OpenACC `kernel loop` 指示文によりアクセラレータへのオフロードとループの並列化を指示し，18 行目では OpenACC `loop` 指示文によりループの並列化を指定している．CloverLeaf の MPI+OpenACC 実装では，複数のネストループを OpenACC `kernels` 指示文で囲うことによって，その部分をまとめてオフロードするように指定がされている．XACC で同様のことをするには，`kernels` で指定されたアクセラレータ領域内で XMP `loop` 指示文を指定することになる．XACC の仕様ではその記述を条件付きで認めており CloverLeaf で使用可能であるが，現在の Omni XACC compiler が対応していないため，各ネストループの最外ループで `kernels loop` として指定することでループごとにオフロードするようにしている．

CloverLeaf における主な通信は袖交換であり，袖領域を幅 1 か 2 で更新する (コード中では変数 *depth*)．MPI+OpenACC 版は袖のデータをアクセラレータ上でパックしたのちにホストメモリを経由して送信する．XACC 版では `reflect` 指示文に `acc` 節を追加してアクセラレータメモリでの袖交換を指定し，かつ `width` 節により更新幅を指定する．セル中心の物理量の配列に対しては，`width`(*depth:depth,depth:depth*) と指定することで MPI+OpenACC 版と同じように袖領域が更新される．しかしながら，セル頂点の物理量の配列に対しては少し異なる指定をしている．セル頂点の配列では extra region も含めてシャドウ領域を確保しており，実際に袖領域として更新が必要なのは内側の 1 要素を除いた部分である．ところが `width` 節ではシャドウ領域の内側から何要素更新するかの指定しかできない．そこで，やむなく extra region も含めて更新するように *width(depth:depth+1,depth:depth+1)* と指定している．そのため，MPI+OpenACC 版よりも通信の量が多くなっている．

表 6.1: HA-PACS/TCA における CloverLeaf の性能評価に用いたソフトウェア

Compiler	PGI 16.10, CUDA 8.0
	MVAPICH2 2.2
	Omni Compiler 1.2.1 + extension

6.1.2 評価

CloverLeaf の MPI+CUDA, MPI+OpenACC, XACC 版における性能と生産性を評価する．MPI+CUDA および MPI+OpenACC 版は MK-MAC が GitHub で公開しているコード[*1]をベースにいくつか修正を加えて使用した．MPI+CUDA 版では，性能向上のためにスレッドブロックサイズを 256 スレッドから 128 スレッドに変更し，また袖通信でのパック・アンパック・コピーの発行順序を変更したうえに，コードの整理を行った．MPI+OpenACC 版では袖通信の順序を MPI+CUDA と合わせ，こちらもコード整理を行った．

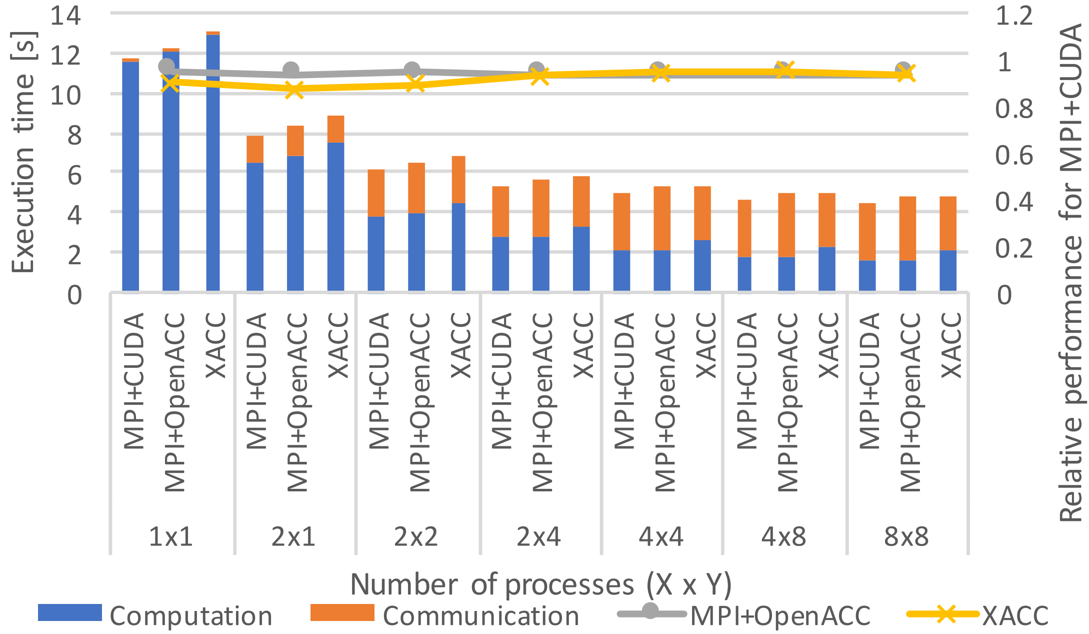
性能評価

筑波大学計算科学研究センターの HA-PACS/TCA を利用して性能評価を行った．使用したソフトウェアを表 6.1 に示す．Fortran の OpenACC コンパイラとして PGI compiler を，また MPI 実装に MVAPICH2 を用いる．XACC のコンパイラには実装した Omni XACC compiler を用いている．なお，評価の時点では XMP `loop` 指示文の `expand` 節の機能はまだマスターブランチにマージされていないため，“extension” と記載している．1 ノードあたり 4 枚の GPU が搭載されているため，1 ノードで 4 プロセス実行し，各プロセスが 1 枚の GPU を利用するようにした．最大で 16 ノード上の 64 プロセスで評価を行った．評価には 960^2 と 3840^2 セルの 2 種類を用い，タイムステップは 1000 回で固定したうえで，ストロングスケーリングとウィークスケーリングでの実行時間を測定した．

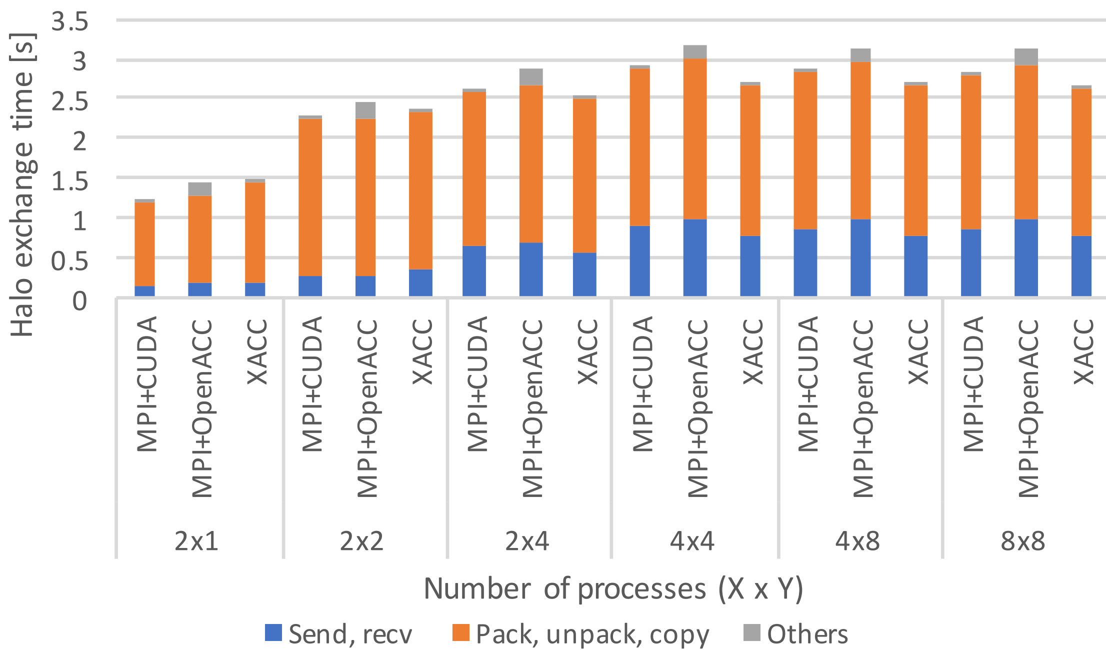
図 6.4 と 6.5 に 960^2 と 3840^2 セルのストロングスケーリング時の実行時間を示す．左側のグラフは全実行時間を，右側のグラフは袖交換のみの時間を示している． MPI+CUDA 版と比較して 960^2 セルにおいては MPI+OpenACC で 93–95%, XACC で 87–95%，3840^2 セルにおいては，MPI+OpenACC が 92–94%，XACC が 89–93% の性能であった．まず計算時間について見てみると，MPI+OpenACC 版は MPI+CUDA 版と比較すると 960^2 で最大 1.05 倍，3840^2 で最大 1.09 倍の計算時間となっている．これは CUDA と OpenACC の性能差によって生じている．XACC 版はオフロードに OpenACC を用いているので，MPI+OpenACC と同様の計算時間になるのが理想であるが，実際には MPI+OpenACC と比較して 960^2 セルで最大 1.2 倍，3840^2 セルで 1.12 倍となっている．この計算時間の増加の原因は XACC コンパイラによるコード変換にある．ソースコード 6.3 はソースコード 6.2 の XACC コードを変換した後のコードである．

ソースコード 6.3: ソースコード 6.2 の XACC コードを Omni XACC compiler が変換したコード

[*1] https://github.com/UK-MAC/CloverLeaf

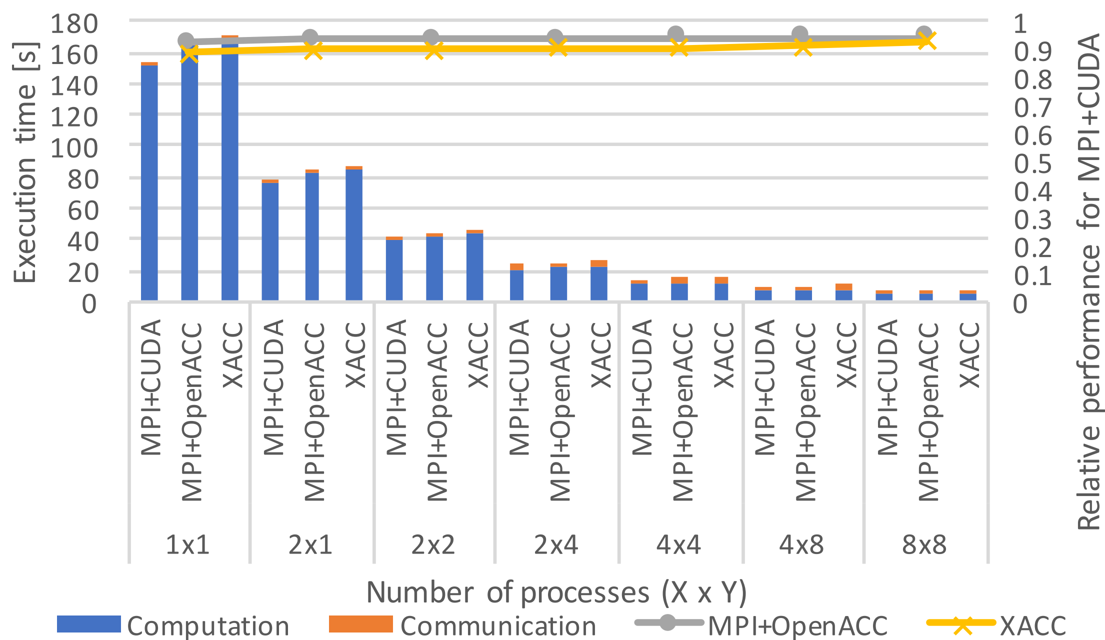


(a) 全実行時間

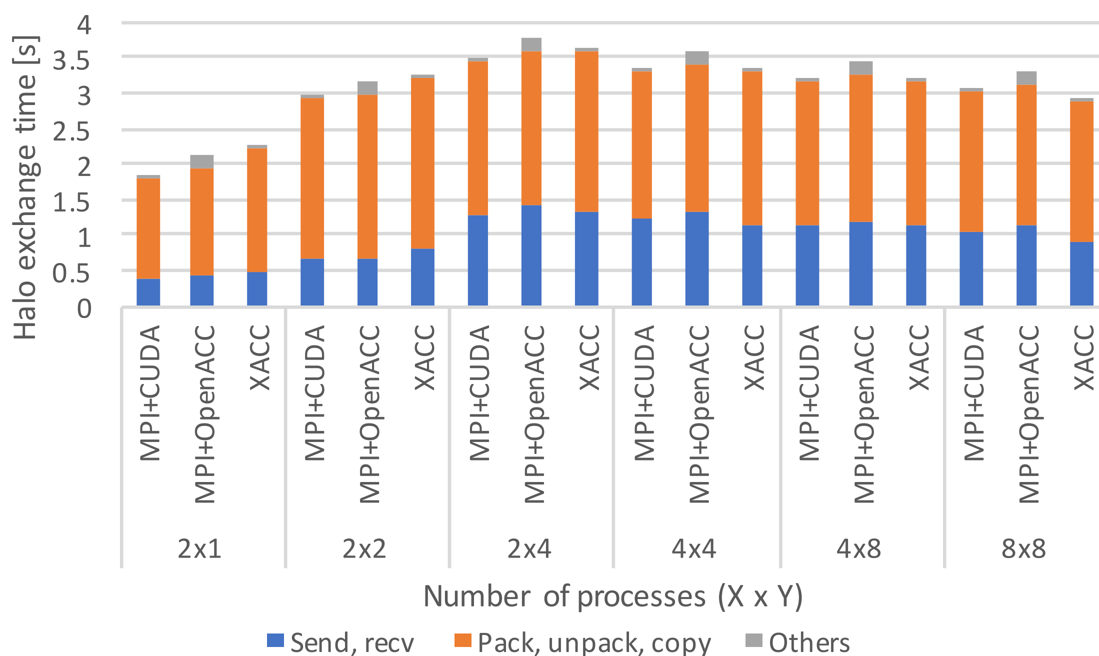


(b) 袖交換時間

図 6.4: 960^2 セルのストロングスケーリングにおける実行時間



(a) 全実行時間



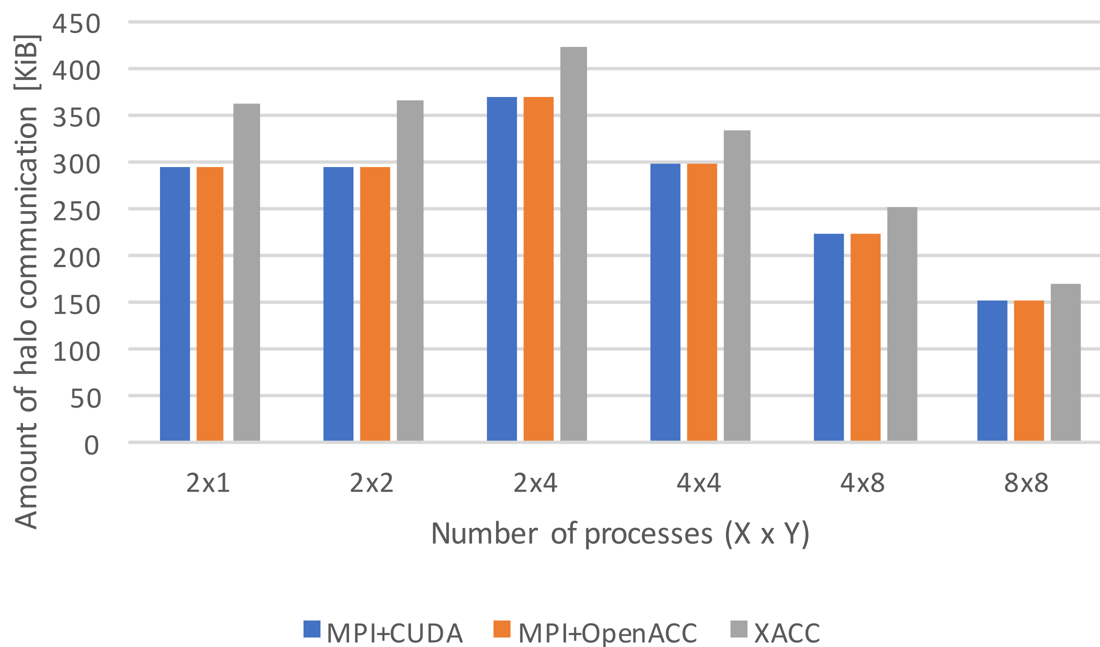
(b) 袖交換時間

図 6.5: 3840^2 セルのストロングスケーリングにおける実行時間

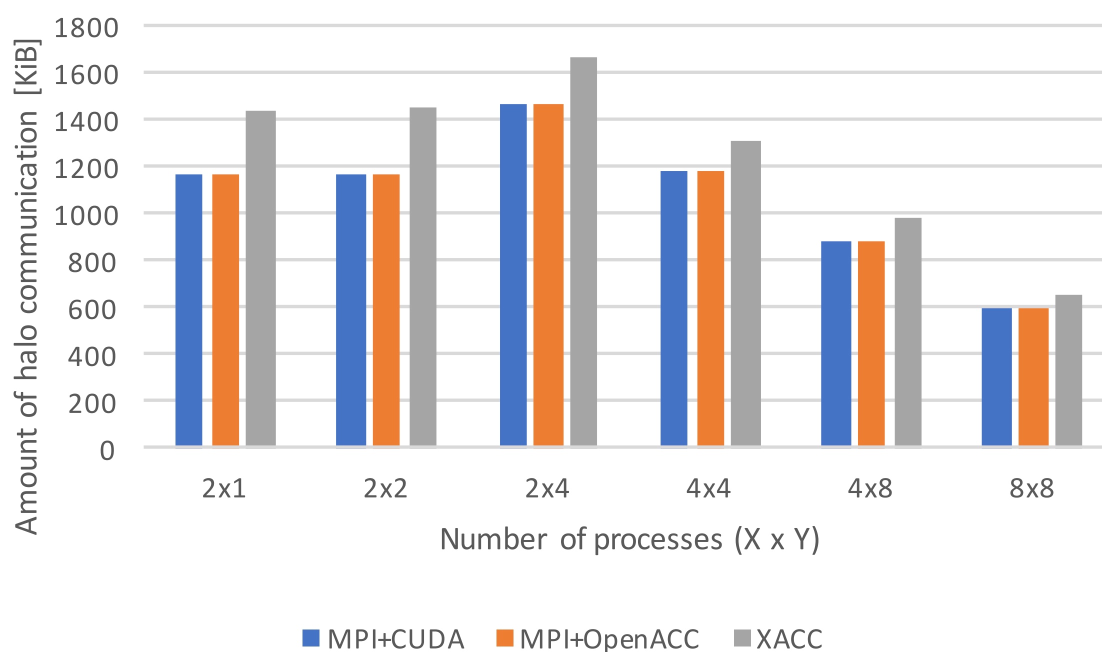
```
1 SUBROUTINE accelerate_kernel(x_min, x_max, y_min, y_max, ..., &
2                              XMP__pressure, XMP__xvel0,...)
3  INTEGER :: x_min, x_max, y_min, y_max
4
5  REAL(KIND=8) :: XMP__pressure ( 0 : , 0 : )
6  REAL(KIND=8) :: XMP__xvel0 ( 0 : , 0 : )
7
8  ...
9
10 !$acc data present ( XMP__pressure , XMP__xvel0, ...)
11
12  ...
13  CALL xmpf_loop_sched_ ( XMP_loop_lb3 , XMP_loop_ub4 , ...)
14  CALL xmpf_loop_sched_ ( XMP_loop_lb6 , XMP_loop_ub7 , ...)
15 !$acc kernels loop independent
16  DO k = XMP_loop_lb3 , XMP_loop_ub4 , 1
17 !$acc loop independent
18    DO j = XMP_loop_lb6 , XMP_loop_ub7 , 1
19      ...
20      XMP__xvel1(j+2,k+2) = XMP__xvel0(j+2,k+2) - ...&
21        XMP__pressure(j+2,k+2) - XMP__pressure(j+2-1,k+2))...
22      ...
23    ENDDO
24  ENDDO
25
26 !$acc end data
27 END SUBROUTINE
```

ソースコード 6.2 の XACC コードでは，サブルーチンの引数である配列 *pressure* などは explicit-shape array で `align` 指示文により分散配列として宣言されている．MPI+OpenACC 版でも XMP 指示文を除けば同様の記述となっている．MPI+OpenACC 版で，直接 PGI コンパイラでコンパイルする場合は，計算中のすべての配列サイズは変数 x_min, x_max, y_min, y_max から求められるため，配列アクセス時のインデックス計算が最適化できる．一方で，XACC で変換されたコードでは，Omni XACC compiler のランタイムが配列サイズを動的に決めるため，引数の配列は assumed-shape array として宣言される．この変換後のコードを PGI コンパイラでコンパイルすると，コンパイル時に配列サイズが不明であるためアクセスが最適化できず，GPU カーネルで使用する配列のサイズはすべて変数となるため，レジスタ使用数が増加し，同時実行できるスレッド数が減って性能が低下する．

MPI+CUDA と XACC 版における袖交換の時間はおおよそ同じであった．これは XACC における `reflect` 指示文が MPI+CUDA で記述されており，十分にチューニングがされているからである．図 6.6 はストロングスケーリングにおいて，各ノードの 1 タイムステップあたりの袖交換の通信量である．XACC 版ではセル頂点の物理量配列に対して余分に更新が必要であるため，他のコードよりも通信量がやや多くなっている．この差はプロセス数が少ない時に通信時間を増加させているが，プロセス数が多くなるとほとんど影響がなくなっている．MPI+OpenACC 版における袖交換時間は，他のコードよりもやや長くなっている．原因の一つは OpenACC で記述しているパックとアンパックが CUDA



(a) 960^2 セル



(b) 3840^2 セル

図 6.6: ストロングスケーリングにおける各ノードのタイムステップ当たりの袖交換通信量

で書くよりも遅いということである．またパック・アンパックでも MPI の通信でもないその他の時間が MPI+OpenACC では非常に目立っている．これは OpenACC `data present` 指示文によるオーバーヘッドである．MPI+OpenACC 版では更新対象の配列に加えて，8 つの通信バッファ (4 方向 ×send と recv) を `data present` で指定している．一方で XACC 版では通信バッファは XACC ランタイムが内部で管理しているため指定しなくて良く，更新対象の配列のみ指定しているためオーバーヘッドが少なく抑えられている．

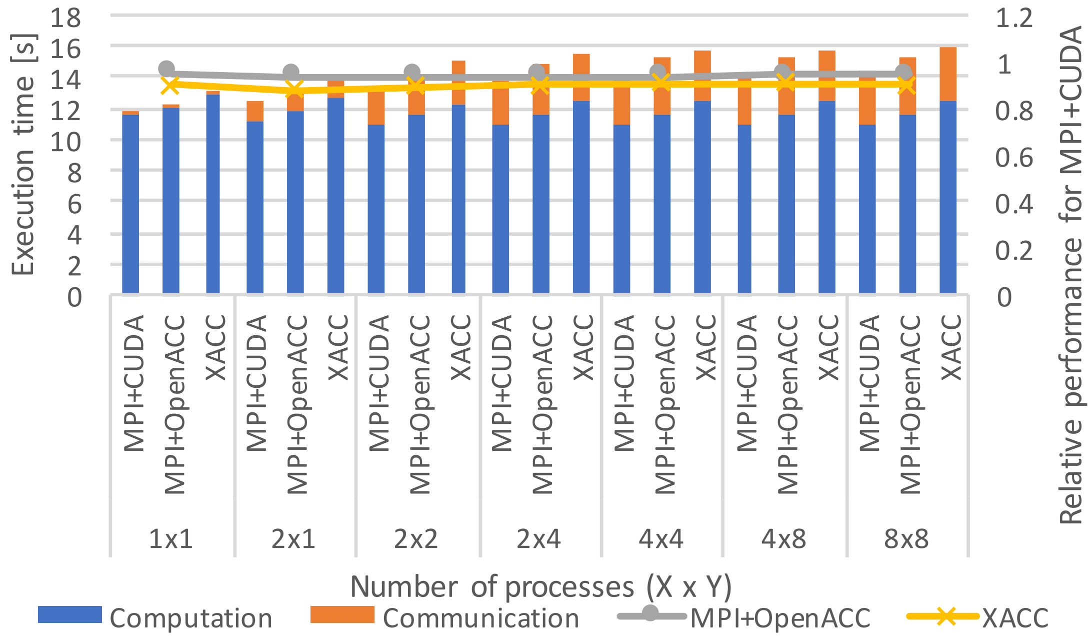
次にウィークスケーリングにおける実行時間を 960^2 の場合を図 6.7 に，3840^2 セルの場合を図 6.8 に示す． 960^2 セルでは MPI+CUDA 版に比べて MPI+OpenACC 版が 94–95% で，XACC 版が 88–91% の性能であった．また，3840^2 セルでは MPI+OpenACC 版が 92% で，XACC 版が 89% の性能であった．すべてのコードとセル数において，実行時間はプロセス数が増えてもほとんど増加していないため，CloverLeaf は良いウィークスケーラビリティを示している．計算時間の傾向はストロングスケーリング時と同じである．MPI+OpenACC では計算時間が MPI+CUDA と比べて最大 1.09 倍になっており，XACC 版では MPI+OpenACC 版と比べて最大 1.07 倍となっている．袖交換時間は 960^2 セルにおいてはストロングスケーリングと傾向は同じである．しかしながら，3840^2 セルでは，send/receive の時間が XACC 版ではすべてのプロセス数で他のコードよりも増加している．図 6.9 はウィークスケーリングにおける，各ノードの 1 タイムステップあたりの袖交換の通信量である．3840^2 セルでは 960^2 セルのときと比較して通信量が 4 倍になるため，通信時間への影響が大きくなっている．

生産性

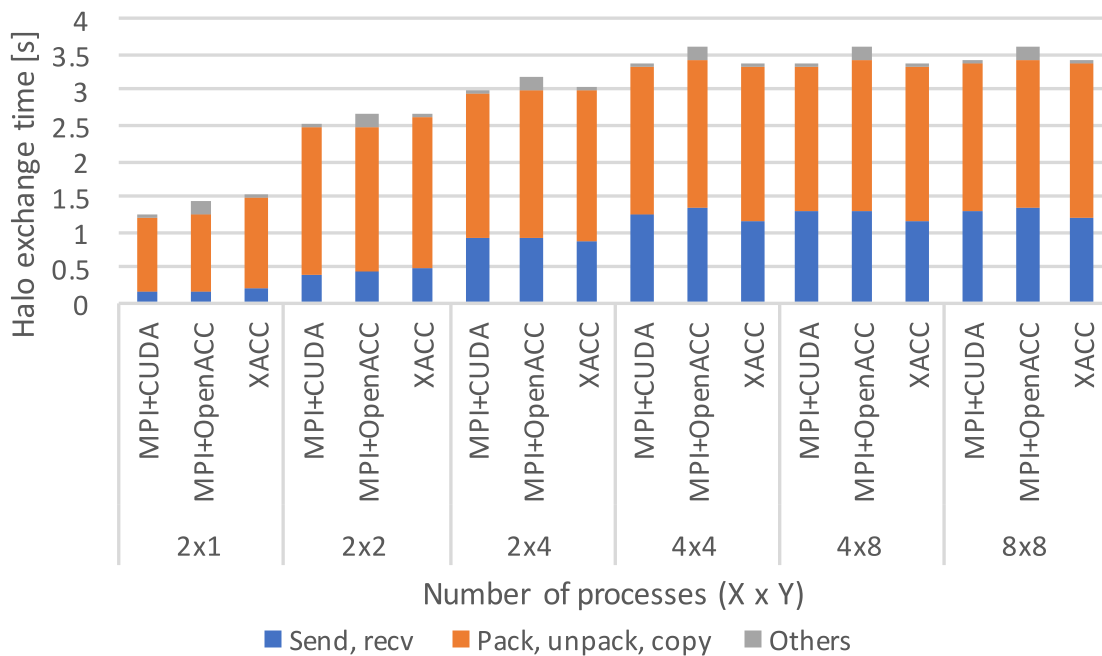
生産性の定量的評価として，source lines of code(SLOC) および逐次コードからの delta source lines of code (DSLOC) を計測した結果を表 6.2 に示す．DSLOC の表における “Add (code)” は指示文でないコードの追加行数を，“Add (directive)” は指示文の追加行数を表す．MPI+CUDA 版では Fortran と CUDA の言語の違いがあるので，SLOC のみ計測した．MPI+CUDA, MPI+OpenACC, XACC 版の SLOC は逐次コードと比較して，それぞれ 1.77, 1.33, 1.23 倍となっており，3 つの並列版のうち XACC 版が最も少ない行数で記述できたことがわかる．MPI+OpenACC と XACC 版の SLOC の差はそこまで大きくはないが，DSLOC でみると両者は大きく異なる．MPI+OpenACC 版では DSLOC の 70% が指示文でないコードの追加であるのに対して，XACC 版では DSLOC の 86% が指示文の追加である．したがって，指示文ベースの記述によって XACC は MPI+OpenACC よりも逐次コードのイメージを保てていると言える．しかしながら，XACC においても指示文追加以外の追加や変更が 100 行近くある．以下にその詳細を示す．

表 6.2: CloverLeaf の各コードの SLOC と DSLOC

Code version	SLOC	DSLOC			
Serial	3043	Modify	Add (code)	Add (directive)	Delete
MPI+CUDA	5397	–	–	–	–
MPI+OpenACC	4042	20	713	286	0
XACC	3746	14	84	619	0

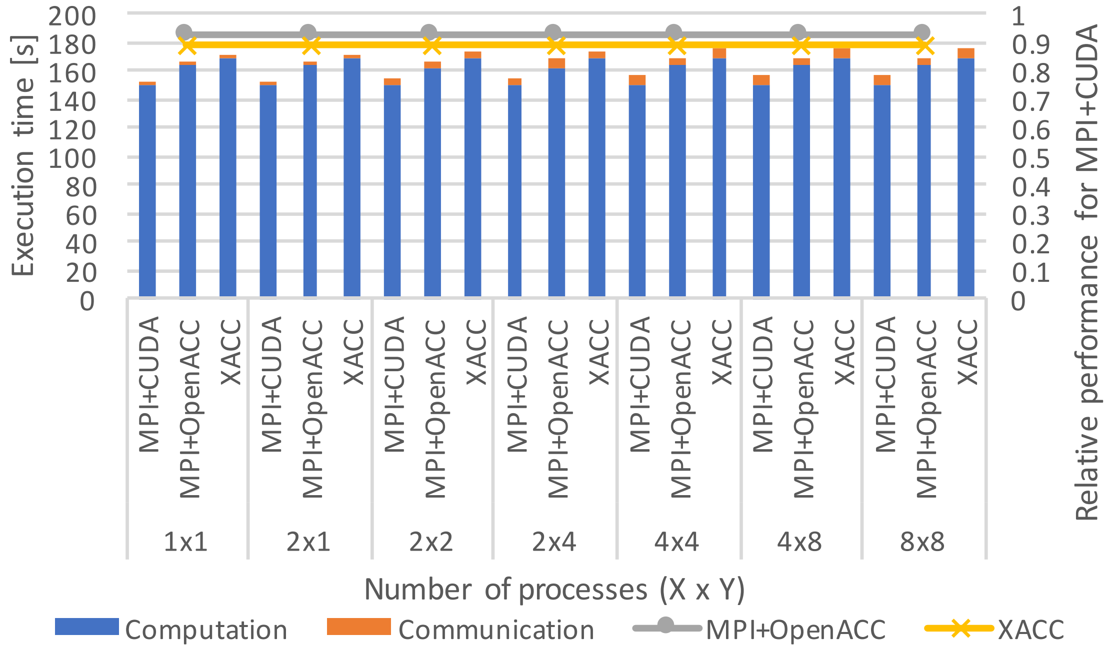


(a) 全実行時間

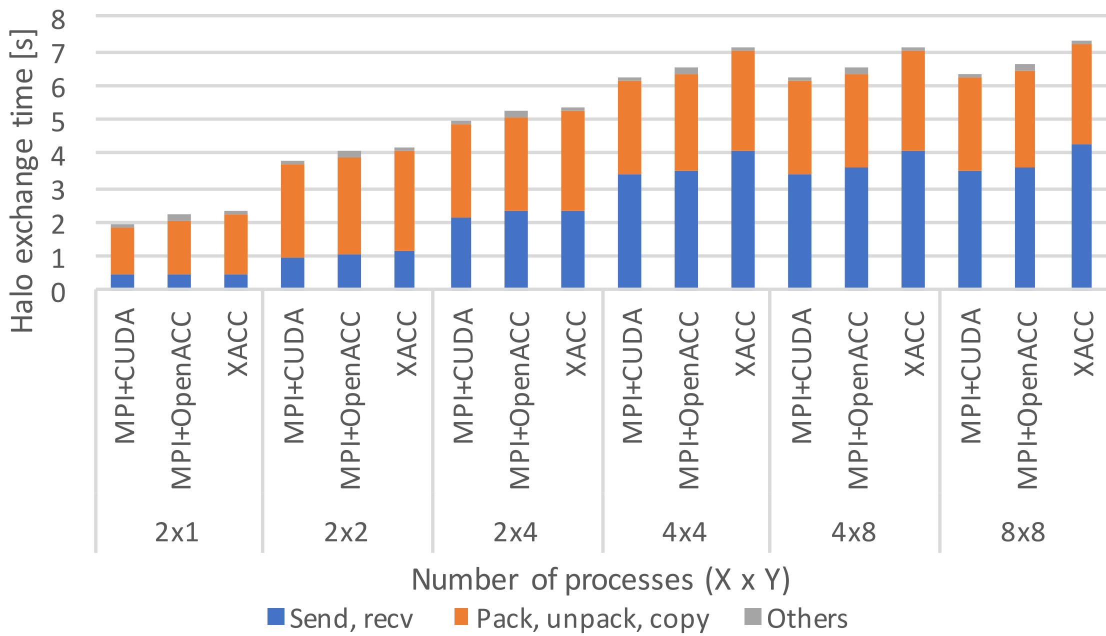


(b) 袖交換時間

図 6.7: 960^2 セルのウィークスケーリングにおける実行時間

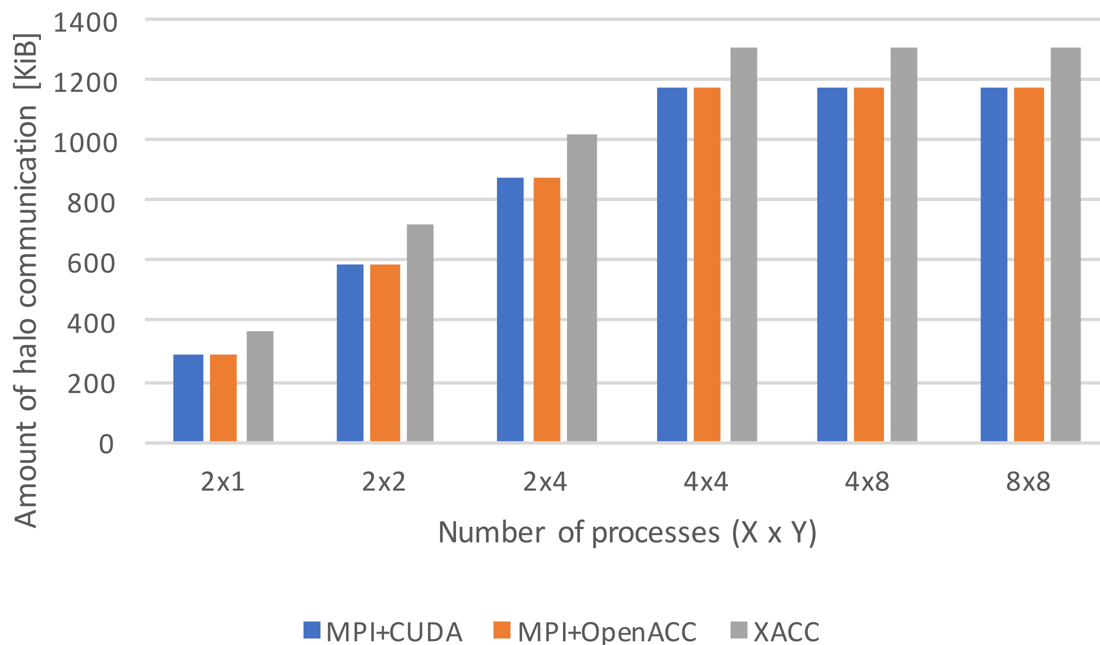


(a) 全実行時間

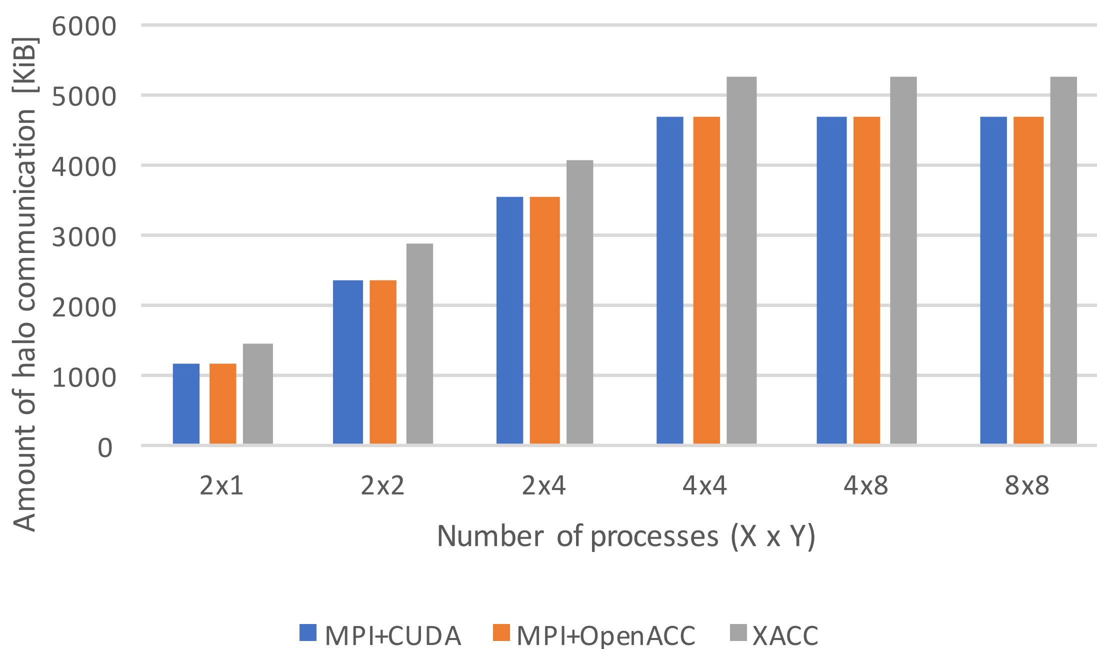


(b) 袖交換時間

図 6.8: 3840^2 セルのウィークスケーリングにおける実行時間



(a) 960^2 セル



(b) 3840^2 セル

図 6.9: ウィークスケーリングにおける各ノードのタイムステップ当たりの袖交換通信量

```
!$xmp {reflect|reduction|bcast} ... [ if ( condition ) ]

and the condition is logical-expr
```

図 6.10: if 節の構文

1. `reflect` 指示文を条件的に行うための *IF* と *ENDIF* により 30 行の追加
2. XACC 版で用いる変数を入れている module を使用する際の *USE* 文が 15 行
3. `gblock` 分散に関連するコードが 9 行
4. 割り当てられたインデックス範囲の取得，ノード番号やノード数の取得，等

6.1.3 考察

MPI+CUDA と MPI+OpenACC の性能比較から，OpenACC は簡易なコード記述であることを考えると良い性能を得られていると言える．MPI+OpenACC と XACC を比較すると，XACC はコード変換による性能低下があるものの MPI+CUDA の 9 割程度であり，指示文による記述の簡易さを考えれば良い性能が出ていると言える．加えて，XACC では袖交換が MPI+CUDA によって実装されているため，いくつかのケースで MPI+OpenACC よりも通信時間が短くなっている．典型的な通信がランタイム側でチューニングされたものを用いることができる点で，性能と可搬性の点で優れている．

このように簡易なプログラミングとチューニングされた通信によって，CloverLeaf のような直交座標系で分散されており，主な通信が袖交換からなるアプリケーションにおいて高い有用性があると言える．またグローバルビューモデルでは複雑で記述できないような分散・通信を行うアプリケーションに対しては，XACC ローカルビューモデルの coarray 機能を用いれば簡易な記述で任意の通信が記述できる．加えて，XMP で提案されているグローバルビューモデルとローカルビューモデルを組み合わせたハイブリッドビューモデル [54] を XACC でも用いることで，両モデルの良い部分を使ったプログラミングが可能であると考えられる．

評価で述べたように，XACC 版では指示文追加以外のコード変更がまだあり，袖交換で余計な通信が起きている．それらの問題を解消するために，以下の改善を考えている．1 つ目は XMP と XACC の通信指示文 (`reflect`, `reduction`, など) への `if` 節の追加である．図 6.10 は提案する `if` 節の構文である．`if` 節が通信指示文に記述された場合，プログラムは条件的に通信を実行する．`if` 節内の *condition* が **.false.** と評価されたら通信を実行せず，*condition* が **.true.** と評価されたら `if` 節が存在しない時と同様に通信を実行する．この `if` 節による記述の方が *IF* や *ENDIF* を通信指示文に記述するよりも簡易である．また指示文をないものとして見た場合にステートメントのない *IF* は不自然に見えるため，節として指示文中に記述する方が自然である．

2 つ目は `distribute` と `template_fix` 指示文における `edged-block` 分散の追加である．XMP の `block` 分散はテンプレートをノード数で割り切れない場合に下側のノードにブロックサイズが偏るという問題があるため，`edged-block` は端の余りを考慮して分散するようにしたものである．`edged-block` 分散の構文を図 6.11 に示す．もし *edge-width* に 1 つの整数式が指定された場合は，下

```
edged-block ( edge-width )

and edge-width is int-expr [: int-expr ]
```

図 6.11: edged-block 分散の構文

```
!$xmp reflect ... [ offset ( reflect-offset [,reflect-offset]... ) ]

and reflect-offset is int-expr [:int-expr]
```

図 6.12: reflect 指示文の offset 節の構文

側・上側の両方の端の幅がその値になる．もし 2 つの整数式が指定されたら，前者が下側の後者が上側の端の幅となる．`edged-block` 分散でははじめに対象のテンプレート次元を上下に端がないものとして `block` 分散を行い，そのあとに最下端と最上端のノードに端を割り当てるようにして分散する．例えば，もしテンプレート *t(1:10)* をノード集合 *p(4)* に `edged-block` *(1)* で分散するなら，各ノードのブロックサイズは $(/3, 2, 2, 3/)$ となる．`edged-block` 分散を用いることで，ソースコード 6.1 の 20 行目は “!$xmp template_fix (edged-block(2:3), edged-block(2:3))&” と記述でき，ソースコード 6.2 の 7 行目は “!$xmp template t(edged-block(2:3), edged-block(2:3)) onto p” と記述できる．

3 つ目の改善案は `reflect` 指示文における `offset` 節の追加である．これはシャドウ領域を更新する際の内側からのオフセットを指定する節である．`offset` 節の構文を図 6.12 に示す．`offset` 節の構文は `width` 節と似ており，1 つの整数式が指定されたなら下側と上側のオフセットがその値となり，2 つの整数式が指定されたなら前者が下側，後者が上側のオフセットとなる．`reflect` 指示文は内側から `offset` で指定された要素を飛ばして `width` で指定された要素数を更新する．もし `offset` 節がない場合は，`offset`(0) と指定されたとみなされる．例として，ある分散配列が *shadow(4)* としてシャドウ領域が追加されていて，“`width`(2) `offset`(1)” と指定して袖の更新を行ったとすると，シャドウ領域の最内 1 要素と最外 1 要素を除く中間の 2 要素のみが更新される．`offset` 節を用いることで，セル頂点の物理量配列の袖更新に対して，“!$xmp reflect(*array-name*) width(depth:depth,depth:depth) offset(0:1,0:1)” と記述することができ，通信量を MPI+CUDA や MPI+OpenACC 版と同じにすることができる．

6.2　格子 QCD ミニアプリケーション

本章では XACC 記述された格子 QCD コードの性能評価を述べる．XACC による格子 QCD の実装は中尾 [55] が行った．本研究は評価の際の Omni XACC compiler の提供および性能解析の協力において貢献した．格子 QCD とは量子色力学 (quantum chromodynamics, QCD) を離散して定式化したものである．QCD では素粒子のグループの 1 種であるクォークと，強い相互作用を媒介する粒子であるグルーオンの間の相互作用を記述するもので，時間と空間の 4 次元の格子上に定式化されている．格子 QCD のコードで時間のかかる部分が CG 法により方程式を解く部分であり，反復処理の中で特に Wilson-Dirac 演算子の計算コストが高い．

表 6.3: 格子 QCD の評価に使用したソフトウェア

Compiler	Intel 16.0.2, CUDA 7.5.18
	MVAPICH2 2.1
	Omni Compiler 1.1

6.2.1　性能評価

格子 QCD コードの XACC 実装と，比較として MPI+CUDA と MPI+OpenACC による実装の性能評価を行った．評価には筑波大学計算科学研究センターの HA-PACS/TCA を用いた．表 6.3 に使用したソフトウェアを示す．計算ノードあたり 4 枚の GPU が搭載されているため，4 プロセスを各計算ノードに割り当てる．格子 QCD の計算領域の大きさを $(NT,NZ,NY,NX) = (32,32,32,32)$ と設定し，時間軸 NT と Z 軸 NZ において 2 次元分散を行い，ストロングスケーリングにおける性能を評価する．図 6.13 に格子 QCD の CG 法 1 反復当たりの実行時間を示す．MPI+CUDA 版と比較して，MPI+OpenACC は

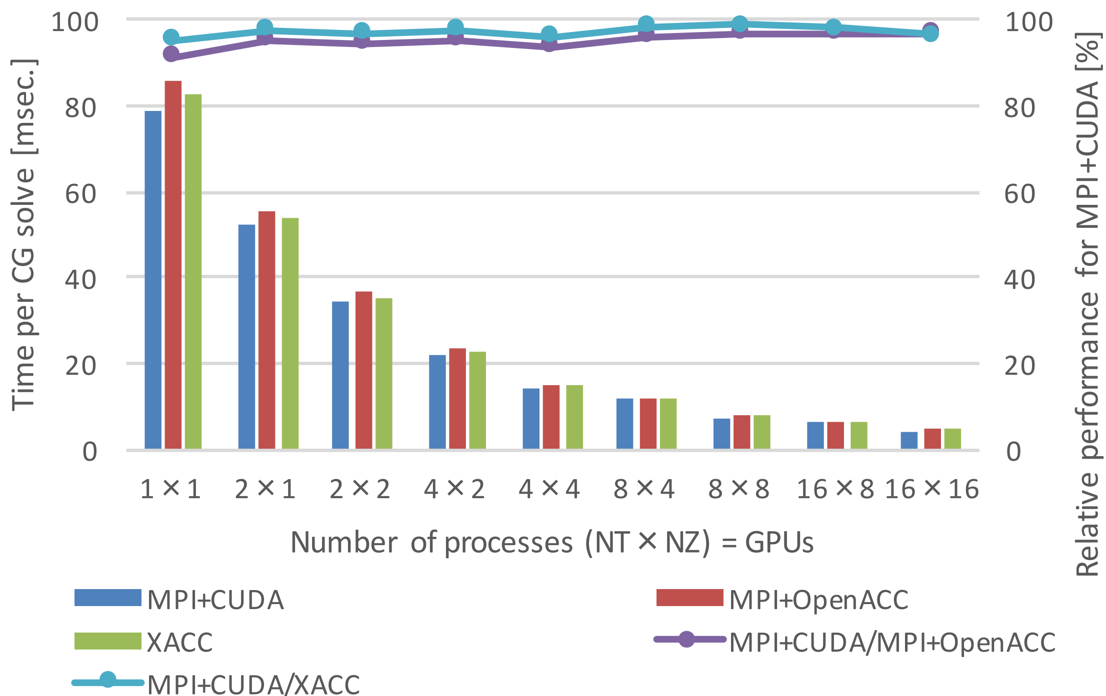


図 6.13: 格子 QCD の CG 法 1 反復当たりの実行時間

91–97%，XACC は 95–99% の性能を達成した．図 6.14 に実行時間の内訳を示す．計算時間を見てみると MPI+CUDA 版と比べて MPI+OpenACC 版では 3–13%, XACC 版では 3–9% 増加している．これは CUDA と OpenACC の性能差によるものである．なお MPI+OpenACC と XACC の間にも差が出る理由は XACC では XACC コンパイラによる配列の書き換えが起きるためである．図 6.15 に袖交換時間を示

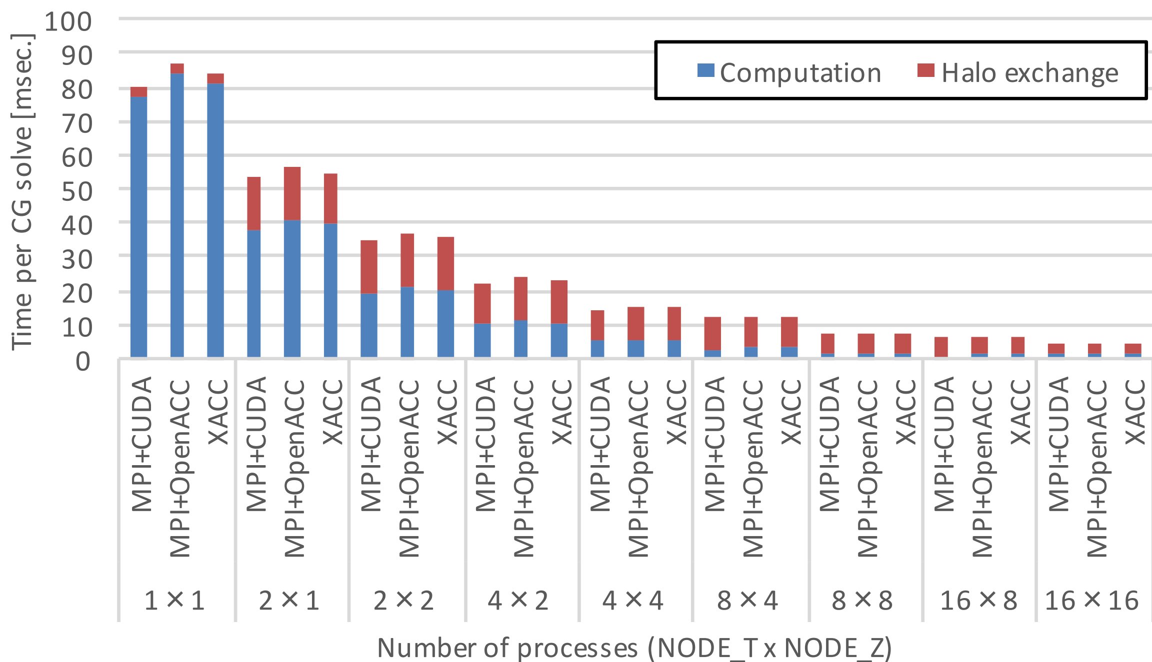


図 6.14: 格子 QCD の CG 法 1 反復当たりの実行時間の内訳

す．MPI による通信時間はどのコードでもほとんど差がないが，パック・アンパックで MPI+OpenACC 版では MPI+CUDA 版の最大 2.0 倍の時間がかかっていた．これはパック・アンパックも OpenACC で記述していることによる性能低下である．一方 XACC 版のパック・アンパックは CUDA で適切に記述されているため 1.1–1.2 倍程度でほぼ MPI+CUDA と同じであった．

6.3 関連研究

CloverLeaf はアクセラレータ向けに CUDA, OpenCL, OpenACC で実装されている [53, 56]．それらの研究では，各実装の詳細な説明と性能・生産性・可搬性の評価が行われた．OpenACC は生産性を向上させることが可能で，性能も部分的に OpenCL や CUDA よりも高いことがあった．また OpenMP 4.0 を用いた実装もされており，OpenACC と OpenMP でほぼ同様に性能が得られている [57]．

Mallinson らは CloverLeaf を PGAS モデルの Co-array Fortran や OpenSHMEM で実装し MPI と比較を行った [58]．彼らは CloverLeaf における袖交換を，不連続部分を直接送ったりパックしてから送ったりするなどの手法で実装した．Cray と SGI のシステムにおける評価では，ライブラリベースの OpenSHMEM は両システムで言語ベースの CAF よりも良い性能が得られている．本研究では PGAS 言語 XACC を用いるが，OpenSHMEM や CAF がローカルビューモデルであるのに対して，XACC ではグローバルビューモデルであるという違いがある．

XMP のグローバルビューモデルとローカルビューモデルを用いて格子 QCD が実装され，Blue Gene/Q で評価が行われた [59]．グローバルビューでは逐次コードに指示文を追加のみで簡易に記述でき，MPI による実装より性能が劣るものの十分な性能が得られた．ローカルビューモデルでは MPI とほぼ同じよ

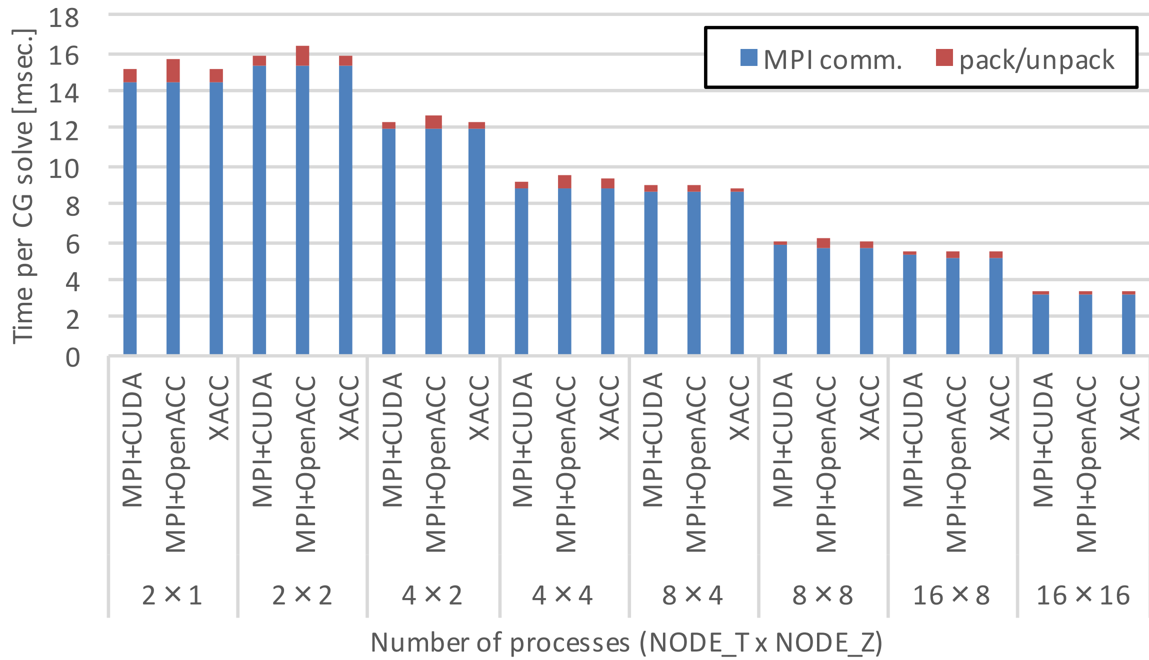


図 6.15: 格子 QCD の CG 法 1 反復当たりの袖交換時間

うに coarray で記述でき，その上 coarray の実装に用いられている片側通信ライブラリの性能が良く MPI よりも若干良い性能が得られている．本研究ではアクセラレータ並列システム向けに XACC のグローバルビューで記述して性能評価を行った．

XMP によるアプリケーションの評価が多数行われている．レーザー核融合の爆縮過程のシミュレーションを行うアプリケーション IMPACT-3D がグローバルビューとローカルビューにより実装されている [60]．また理研が開発している次世代スーパーコンピュータ実現のためのミニアプリ集である Fiber がローカルビューモデルにより実装されており，一部を除いてオリジナル版にほぼ近い性能が得られている [61]．さらに核融合シミュレーションコード GTC-P がグローバルビューとローカルビューを組み合わせたハイブリッドビューにより実装し評価が行われている [54]．PIC 法における場の計算を行う計算格子をグローバルビューで，粒子軌道演算をローカルビューで記述することにより生産性と性能の両立をしている．これらのアプリケーションにおいては XMP による実装をベースに XACC による実装も可能であると考えられる．

第 7 章

結論

7.1　まとめ

本研究ではアクセラレータを持つ並列システムを活用するために，プログラミングを容易にして生産性を向上させるとともに高性能を達成できるプログラミング環境を提案した．プログラミング言語として新たに PGAS 言語 XACC を理化学研究所と共同で提案した．XACC は OpenACC と XMP の垂直統合であり，さらにアクセラレータ間の通信の記述を提供する．XACC のグローバルビューモデルの提案は既に行われていたため，本研究では新たにローカルビューモデルを提案し，coarray に既存の OpenACC 指示文を指定することでアクセラレータ上に coarray を宣言しその coarray を用いてアクセラレータメモリにおける片側通信を可能とした．XACC はアクセラレータプログラミングに OpenACC を利用することで多様なアクセラレータに対応でき，指示文や coarray でアクセラレータ間通信を記述することで，多様な通信方法に対応可能である．XACC の生産性や性能を評価するため，NVIDIA GPU クラスタと PEZY-SC クラスタに向けて XACC に必要なコンパイラの実装を行った．

OpenACC のコンパイラは研究開始当初は商用コンパイラしかなく NVIDIA GPU など一般的なアクセラレータのみの対応で，かつ最適化が不十分であった．また PEZY-SC に対応した OpenACC コンパイラがなかった．そこで，NVIDIA GPU 向けの OpenACC の高速化のため，OpenACC から CUDA にプログラム変換を行うコンパイラを設計・実装した．ループ並列化以外は元のコードを維持して変換することで CUDA コンパイラがコード最適化を行いやすくした．さらにメモリ確保コスト削減のためにメモリプールの利用や，Kepler アーキテクチャ向けに読み込み専用キャッシュやリダクションで shuffle や atomic 命令を利用する最適化を行った．演算性能が重要となる NPB EP ベンチマークにおいて本実装では商用コンパイラの 1.1–2.2 倍の性能を達成し，リダクションやメモリアクセスが重要となる NPB CG ベンチマークにおいて，商用コンパイラの 1.0–2.8 倍の性能を達成した．これにより NVIDIA GPU 向けの OpenACC コンパイラの性能向上を達成した．

また PEZY-SC への対応のため OpenACC から PZCL へ変換を行うコンパイラを設計・実装した．こちらも同様の変換により PZCL コンパイラによるコード最適化が行われるようにした．また PEZY-SC 向けの最適化として kernels 指示文を単一カーネルに変換することでカーネル起動コストを削減し，スレッド切り替えや同期，フラッシュ用の拡張指示文を追加することでユーザがチューニング可能とした．演算が支配的である N-body ベンチマークにおいては直接記述した PZCL 版の 98% の性能が得られた．カーネ

ル起動コスト削減の最適化と拡張指示文による明示的スレッド切り替えによってメモリアクセスが支配的なNPB CGベンチマークにおいても直接記述したPZCLの88%以上の性能を達成した．OpenACC版とPZCL版のコードを比較すると，OpenACC版は逐次コードに指示文を加えるだけで簡易に実装できており，N-Bodyは48%，NPB CGは45%のコード行数となった．また本実装はPEZY-SC向けの初の指示文モデルコンパイラであり，簡易なプログラム開発と既存OpenACCコードの利用を可能にしたことで生産性向上に寄与した．これによりPEZY-SC向けプログラムの生産性の向上とPZCLと同等の性能を達成した．

さらにXACCコンパイラを設計・実装した．このXACCコンパイラはXACCからOpenACCとXACCランタイム呼び出しに変換するコンパイラであり，本研究で実装したものを含む通常のOpenACCコンパイラによりコンパイル可能である．XACCランタイムライブラリをNVIDIA GPUクラスタ向けにグローバル・ローカルビューモデルをMPI+CUDAで，PEZY-SCクラスタ向けにグローバルビューモデルをMPI+PZCLで実装した．NVIDIA GPU向け実装のベンチマークによる評価では，グリッド分散と袖交換通信を行うHimeno benchmarkはグローバルビューモデルの指示文で簡潔に記述できMPI+OpenACCによる記述の97%以上の性能を達成した．複雑な通信を必要とするNPB CGにおいては，ローカルビューモデルのcoarrayによりMPIと同様の通信パターンを記述できたことで，MPI+OpenACCの97%以上の性能が得られた．またPEZY-SC向けにグローバルビューモデルの袖通信の実装を行い，Himeno benchmarkを用いた評価においてMPI+PZCLの96%以上の性能を達成した．XACCによるアプリケーションの評価では，流体力学ミニアプリケーションCloverLeafをグローバルビューモデルにより指示文で簡易に実装することができ，性能はMPI+CUDAと比べて87%以上と十分に高い性能が得られた．また格子QCDミニアプリケーションによる評価でもMPI+CUDAの95%の性能を達成した．ベンチマークやアプリケーションによるXACCの評価から，典型的なグリッド分散と典型的な袖通信を行うようなプログラムはグローバルビューモデルにより逐次コードに指示文を追加するのみで簡易に記述ができるとともに高性能が得られることや，グローバルビューでは記述が困難な分散や通信を行うプログラムはローカルビューモデルのcoarrayを用いることで配列代入文形式によって簡易に柔軟な通信を記述でき高い性能を達成できることを示した．加えて高レベルな記述により多様なアクセラレータや通信に対応でき，Himeno benchmarkはほぼ同一の記述によりNVIDIA GPUとPEZY-SCクラスタで実行可能であった．

7.2　今後の課題

NVIDIA GPUおよびPEZY-SC向けOpenACCコンパイラにおいてはOpenACC 1.0相当の最低限の機能実装にとどまっているため，より詳細な並列化指定や使いやすさのために最新の仕様の実装が必要である．NVIDIA GPU向けの実装で判明したスレッド並列化しない部分のコード変換の改善やgang private配列を利用した時のメモリ確保と解放の削減などが必要となる．またPEZY-SC向け実装では，拡張指示文によるスレッド切り替えや同期の手段を提供したが，コンパイラによる自動追加が望ましい．加えてPEの階層構造に適したgang,worker,vectorへの並列レベルへの割り当てやコード変換も検討が必要である．XACCの実装では，NVIDIA GPU向け実装はgmove指示文がまだ一部の通信パターンにしか対応できておらず，Fortranにおけるcoarrayや CとFortranにおける提案した`nonblock`指示文が未実装である．PEZY-SC向け実装では袖交換の通信のみであるため，その他の通信の実装が必要である．アプリ

ケーションを用いた評価では，CloverLeaf で提案した指示文の実装により通信量を削減して性能が向上するかを確認する必要がある．またグローバルビューモデルでの評価しかできていないためローカルビューモデルを用いた評価も必要である．

XACC の可搬性に関しては現在 Himeno benchmark のみでしか複数の環境で実行が行われていないため，より多くのベンチマークやアプリケーションで評価を行う必要がある．今後はアクセラレータとホストのメモリが物理的もしくは仮想的に同一アドレス空間でアクセスできるようなシステムが増加すると考えられるため，アクセラレータのメモリモデルの違いにおける可搬性も考慮が必要である．例として NVIDIA GPU は CPU と GPU で同じようにアクセスできる Unified Memory を用いることができ，PEZY-SC2 には MIPS プロセッサが搭載されており，PE などと同一のメモリにアクセスすることが可能となる予定である．XACC ではアクセラレータ間の通信を記述しておくことで，従来のメモリが独立しているシステムでも，共有しているシステムでも適切に通信が可能であると考えられるため，メモリを共有しているシステムにおいても XACC の記述で効果的に利用できることの実証が必要である．

現在のアクセラレータは基本的にホストとなる CPU からの制御で通信を行うようになっているが，よりレイテンシを少なくするために今後はアクセラレータ側から直接通信を発行できるようになると考えられる．すでにアクセラレータのカーネル中からあらかじめ登録しておいた通信を起動する手法 [62, 63] や，アクセラレータカーネルからの細かな通信を CPU 側のプロキシが宛先ごとにまとめるなどの処理をして送る手法 [64] が提案されている．そのような通信方法にも対応させるためには XACC においても，OpenACC のアクセラレータ計算領域中に通信指示文や coarray を記述できるようにするのが望ましい．また並列システムの並列性とアクセラレータ内の並列数の増加により，単純なデータ並列だけではシステムの並列性を活かせなくなることが予想できる．その際に，XMP におけるタスク並列 [65] と OpenACC の組み合わせによるアプローチが有効であると考えられる．

HPC においても FPGA をアクセラレータとして用いることが広まると予想される．これまで Verilog-HDL を用いたハードウェアに近いプログラミングが必要であったが，OpenCL によるプログラミングが可能となり利用のハードルが下がりつつある．また OpenACC から FPGA 向けの OpenCL に変換する研究 [66] も行われているため，XACC で FPGA をアクセラレータとして利用することも十分に考えられる．また FPGA を通信および演算に用いる Accelerator in Switch(AiS) というコンセプトが提唱されている [67]．FPGA を主にはネットワークスイッチングハブとして利用し，回路の空いた部分を用いて通信前後の処理を行うというものである．特に GPU では不得意な計算を FPGA にオフロードすることで，高速化が期待できる．XACC において AiS を活用するとすれば，通信は指示文で用意されたパターンもしくは coarray による多次元配列の put/get のみであるため，FPGA を用いてリダクションやパック・アンパックを行うことは可能であると考えられる．しかしながら，それ以外のプログラム独自の処理を行うためには何らかの拡張，例えばユーザが通信や通信前後の処理を記述して新たな通信を定義できるようにすることが必要になると思われる．

謝辞

本研究を行うにあたり，多くのご指導ご鞭撻を賜りました筑波大学システム情報工学研究科教授（連携大学院）・佐藤三久先生に深く感謝いたします．アクセラレータを軸として 6 年間で様々な研究をさせていただき，多くの研究会や国際会議で発表を行うことができました．改めて御礼申し上げます．またご多忙の中，本論文の副査を引き受けて下さいました筑波大学計算科学研究センター教授・朴泰祐先生，筑波大学システム情報工学研究科教授・安永守利先生，同教授・和田耕一先生，および東京大学情報基盤センター准教授・塙敏博先生に心からお礼申し上げます．そして日頃の研究生活でお世話になりました筑波大学計算科学研究センター教授・高橋大介先生，同教授・建部修見先生，同准教授・川島英之先生，同助教・多田野寛人先生，同助教・小林諒平先生，筑波大学システム情報工学研究科准教授・山口佳樹先生，理化学研究所計算科学研究機構研究員・児玉祐悦博士に御礼申し上げます．

本研究を進めるにあたり OpenACC の実装や XACC の仕様・実装・評価に関して多大なるご協力をいただきました理化学研究所計算科学研究機構研究員・村井均博士，同研究員・中尾昌広博士，同特別研究員・Lee Jinpil 博士，同特別研究員・小田嶋哲哉博士，株式会社富士通・岩下英俊氏に感謝の意を表します．PEZY-SC 向け OpenACC および XACC の実装に際しまして，睡蓮および青睡蓮システムのご提供やご助言をいただきました高エネルギー加速器研究機構共通基盤研究施設計算科学センター准教授・石川正先生，同助教・松古栄夫先生，株式会社 ExaScaler・木村耕行氏，同・鳥居淳氏，同諸氏，および株式会社 PEZY Computing 諸氏に深く御礼申し上げます．インターンシップを受け入れてくださり，期間中研究・生活面で支えて下さいましたエジンバラパラレルコンピューティングセンター Adrian Jackson 氏，Giacomo Peru 氏，Mario Antonioletti 博士に感謝いたします．なお本研究の一部は JST-CREST 研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」・研究課題「ポストペタスケール時代に向けた演算加速機構・通信機構統合環境の研究開発」，ならびに筑波大学計算科学研究センター学際共同利用プログラムによるものです．

そして HPCS 研究室の皆様，とりわけ先輩である筑波大学計算科学研究センター・藤田典久博士，株式会社富士通研究所・大辻弘貴博士，ヤフー株式会社・鷹津冬将博士には数々の助言をいただき，海外出張でも非常にお世話になりました．厚く御礼申し上げます．同じ PA チームの同期の津金佳祐氏，後輩の宇川斉志氏，大川千聡氏，袁嘉希氏，杉山大輔氏，松村和朗氏，渡部裕氏には研究の相談・論文の校正などでお世話になりました．心より感謝申し上げます．

最後に 9 年間の長きにわたる大学生活を支えてくれた両親に深く感謝します．

参考文献

[1] NVIDIA Kepler GK110 Architecture Whitepaper. http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf.

[2] TOP500.org. TOP500 Supercomputer Sites. https://www.top500.org/, 2017.

[3] NVIDIA Corporation. CUDA Zone. https://developer.nvidia.com/cuda-zone, 2017.

[4] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/.

[5] OpenACC-standard.org. The OpenACC Application Programming Interface Version 2.6. https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf, 2017.

[6] OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 4.5 November 2015. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, 2015.

[7] XcalableMP Specification Working Group. XcalableMP WebSite. http://www.xcalablemp.org, 2017.

[8] NVIDIA. NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect.

[9] RIKEN AICS and University of Tsukuba. XcalableACC Language Specification Version 1.0. http://xcalablemp.org/download/XACC/xacc-spec-1.0.pdf, 2017.

[10] PC Cluster Consortium. PC Cluster Consortium. http://www.pccluster.org/en/, 2017.

[11] High Performance Fortran Forum. High Performance Fortran Language Specification. http://hpff.rice.edu/versions/hpf2/hpf-v20.pdf.

[12] R. W. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, Vol. 17, No. 2, pp. 1–31, August 1998.

[13] M. Nakao, H. Murai, T. Shimosaka, A. Tabuchi, T. Hanawa, Y. Kodama, T. Boku, and M. Sato. XcalableACC: Extension of XcalableMP PGAS Language Using OpenACC for Accelerator Clusters. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, pp. 27–36, 2014.

[14] J. Lee, M. Tran, T. Odajima, T. Boku, and M. Sato. An Extension of XcalableMP PGAS Lanaguage for Multi-node GPU Clusters. In *Euro-Par 2011: Parallel Processing Workshops*, Vol. 7155 of *Lecture Notes in Computer Science*, pp. 429–439. 2012.

[15] 李珍泌, チャントゥアンミン, 小田嶋哲哉, 朴泰祐, 佐藤三久. PGAS 並列プログラミング言語 XcalableMP における演算加速装置を持つクラスタ向け拡張仕様の提案と試作. 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 5, No. 2, pp. 33–50, Mar. 2012.

[16] T. Nomizu, D. Takahashi, J. Lee, T. Boku, and M. Sato. Implementation of XcalableMP Device Acceleration Extention with OpenCL. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pp. 2394–2403, May 2012.

[17] RIKEN AICS and University of Tsukuba. Omni Compiler Project. http://omni-compiler.org, 2017.

[18] Programming Guide :: CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[19] Center for Manycore Programming, Seoul National University. SNU NPB Suite. http://aces.snu.ac.kr/software/snu-npb/.

[20] OpenUH-OpenACC. https://github.com/pumpkin83/OpenUH-OpenACC.

[21] M. S. Warren, J. K. Salmon, D. J. Becker, M. P. Goda, T. Sterling, and W. Winckelmans. Pentium Pro Inside: I. A Treecode at 430 Gigaflops on ASCI Red, II. Price/Performance of $50/Mflop on Loki and Hyglac. In *Supercomputing, ACM/IEEE 1997 Conference*, pp. 61–61, Nov. 1997.

[22] Center for Computational Sciences, University of Tsukuba. Supercomputers - Center for Computational Sciences. https://www.ccs.tsukuba.ac.jp/eng/supercomputers/#HA-PACS, 2017.

[23] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–11, Nov. 2010.

[24] T. Han and T. Abdelrahman. hiCUDA: High-Level GPGPU Programming. *Parallel and Distributed Systems, IEEE Transactions on*, Vol. 22, No. 1, pp. 78–90, Jan. 2011.

[25] R. Reyes, I. López-Rodríguez, J. Fumero, and F. d. Sande. accULL: An OpenACC Implementation with CUDA and OpenCL Support. In *Euro-Par 2012 Parallel Processing*, Vol. 7484 of *Lecture Notes in Computer Science*, pp. 871–882. Springer Berlin Heidelberg, 2012.

[26] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman. Compiling a High-Level Directive-Based Programming Model for GPGPUs. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pp. 105–120. Springer International Publishing, 2014.

[27] S. Lee and J. S. Vetter. OpenARC: Open Accelerator Research Compiler for Directive-based, Efficient Heterogeneous Computing. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pp. 115–120, New York, NY, USA, 2014. ACM.

[28] RoseACC. http://roseacc.org/.

[29] OpenACC - GCC Wiki. https://gcc.gnu.org/wiki/OpenACC.

[30] R. Xu, X. Tian, Y. Yan, S. Chandrasekaran, and B. Chapman. Reduction Operations in Parallel Loops for GPGPUs. In *Proceedings of Programming Models and Applications on Multicores and Manycores*, PMAM'14, pp. 10–20, 2007.

[31] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pp. 137–148, Nov. 2011.

[32] PathScale/NPB2.3-OpenACC-C. https://github.com/pathscale/NPB2.3-OpenACC-C.

[33] R. Xu, X. Tian, S. Chandrasekaran, Y. Yan, and B. Chapman. NAS Parallel Benchmarks for GPGPUs using a Directive-based Programming Model. 2014.

[34] Network-Based Computing Laboratory. MVAPICH :: Home. http://mvapich.cse.ohio-state.edu/.

[35] The Open MPI Project. Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/.

[36] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda. Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *2013 42nd International Conference on Parallel Processing*, pp. 80–89, Oct. 2013.

[37] H. Murai and M. Sato. An Efficient Implementation of Stencil Communication for the XcalableMP PGAS Parallel Programming Language. In *7th International Conference on PGAS Programming Models*, p. 142, 2013.

[38] T. Hanawa, Y. Kodama, T. Boku, and M. Sato. Interconnection Network for Tightly Coupled Accelerators Architecture. In *High-Performance Interconnects (HOTI), 2013 IEEE 21st Annual Symposium on*, pp. 79–82, Aug. 2013.

[39] RIKEN Advanced Center for Computing and Communication. Himeno Benchmark. http://accc.riken.jp/en/supercom/himenobmt/.

[40] M. Nakao, J. Lee, T. Boku, and M. Sato. XcalableMP Implementation and Performance of NAS Parallel Benchmarks. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS ’10, pp. 11:1–11:10, New York, NY, USA, 2010. ACM.

[41] NVIDIA. NVIDIA/gdrcopy: A fast GPU memory copy library based on NVIDIA GPUDirect RDMA technology. https://github.com/NVIDIA/gdrcopy.

[42] W.-Y. Chen, D. Bonachea, C. Iancu, and K. Yelick. Automatic Nonblocking Communication for Partitioned Global Address Space Programs. In *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS ’07, pp. 158–167. ACM, 2007.

[43] Cray. Cray Fortran Reference Manual (8.5). http://docs.cray.com/PDF/Cray_Fortran_Reference_-Manual_85.pdf.

[44] 小田嶋哲哉, 朴泰祐, 塙敏博, 児玉祐悦, 村井均, 中尾昌広, 田渕晶大, 佐藤三久. アクセラレータ向け並列言語 XcalableACC における TCA/InfiniBand ハイブリッド通信. 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 8, No. 4, pp. 61–77, Nov. 2015.

[45] Y. Zheng, C. Iancu, P. H. Hargrove, S.-J. Min, and K. Yelick. Extending Unified Parallel C for GPU Computing. In *SIAM Conference on Parallel Processing for Scientific Computing (SIAMPP)*, 2010.

[46] S. Potluri, D. Bureddy, H. Wang, H. Subramoni, and D. Panda. Extending OpenSHMEM for GPU Computing. *Parallel and Distributed Processing Symposium, International*, pp. 1001–1012, 2013.

[47] L. Chen, L. Liu, S. Tang, L. Huang, Z. Jing, S. Xu, D. Zhang, and B. Shou. Unified Parallel C for GPU Clusters: Language Extensions and Compiler Implementation. In *Languages and Compilers for Parallel Computing*, Vol. 6548 of *Lecture Notes in Computer Science*, pp. 151–165. Springer Berlin Heidelberg, 2011.

[48] D. Cunningham, R. Bordawekar, and V. Saraswat. GPU Programming in a High Level Language: Compiling X10 to CUDA. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 ’11, pp. 1–10, New York, NY, USA, 2011. ACM.

[49] A. Sidelnik, B. L. Chamberlain, M. J. Garzaran, and D. Padua. Using the High Productivity Language Chapel to Target GPGPU Architectures. Technical report, Cray, 2011.

[50] C. Rasmussen, M. Sottile, S. Rasmussen, D. Nagle, and W. Dumas. CAFe: Coarray Fortran Extensions for Heterogeneous Computing. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 357–365, May 2016.

[51] V. Cardellini, A. Fanfarillo, S. Filippone, and D. Rouson. Hybrid coarrays: A PGAS feature for many-core architectures. In *International Conference on Parallel Computing (ParCo 2015)*, 2015.

[52] A. Hart, R. Ansaloni, and A. Gray. Porting and scaling OpenACC applications on massively-parallel, GPU-accelerated supercomputers. *The European Physical Journal Special Topics*, Vol. 210, No. 1, pp. 5–16, 2012.

[53] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. Accelerating Hydrocodes with OpenACC, OpenCL and CUDA. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pp. 465–471, Nov. 2012.

[54] K. Tsugane, T. Boku, H. Murai, M. Sato, W. Tang, and B. Wang. Hybrid-view Programming of Nuclear Fusion Simulation Code in the PGAS Parallel Programming Language XcalableMP. *Parallel Computing*, Vol. 57, pp. 37–51, 2016.

[55] M. Nakao, H. Murai, H. Iwashita, A. Tabuchi, T. Boku, and M. Sato. Implementing Lattice QCD Application with XcalableACC Language on Accelerated Cluster. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 429–438, Sep. 2017.

[56] A. C. Mallinson, D. A. Beckingsale, W. P. Gaudin, J. A. Herdman, and S. A. Jarvis. Towards Portable Performance for Explicit Hydrodynamics Codes. In *1st International Workshop on OpenCL*, IWOCL 13, May 2013.

[57] M. Martineau, S. McIntosh-Smith, and W. Gaudin. Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 338–347, May 2016.

[58] A. C. Mallinson, S. A. Jarvis, W. P. Gaudin, and J. A. Herdman. Experiences at Scale with PGAS Versions of a Hydrodynamics Application. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pp. 9:1–9:11, New York, NY, USA, 2014. ACM.

[59] 土井淳. XcalableMP による格子 QCD の並列化と Blue Gene/Q における性能評価. Technical Report 28, 日本アイ・ビー・エム株式会社東京基礎研究所, Dec. 2014.

[60] H. Sakagami. Performance Comparison between Two Programming Models of XcalableMP. http://wallaby.aics.riken.jp/lens/slides/xmp-6.sakagami.pdf.

[61] 村井均, 中尾昌広, 岩下英俊, 佐藤三久. PGAS 言語 XcalableMP による Fiber ミニアプリ集の実装と評価. Technical Report 27, 国立研究開発法人理化学研究所計算科学研究機構, Aug. 2016.

[62] E. Agostini, D. Rossetti, and S. Potluri. Offloading Communication Control Logic in GPU Accelerated Applications. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 248–257, May 2017.

[63] M. LeBeane, K. Hamidouche, B. Benton, M. Breternitz, S. K. Reinhardt, and L. K. John. GPU Triggered Networking for Intra-kernel Communications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pp. 22:1–22:12, New York, NY, USA, 2017. ACM.

[64] M. S. Orr, S. Che, B. M. Beckmann, M. Oskin, S. K. Reinhardt, and D. A. Wood. Gravel: Fine-grain GPU-initiated Network Messages. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pp. 23:1–23:12, New York, NY, USA, 2017. ACM.

[65] K. Tsugane, J. Lee, H. Murai, and M. Sato. Multi-tasking Execution in PGAS Language XcalableMP and Communication Optimization on Many-core Clusters. In *Proceedings of International Conference on High Performance Computing in Asia-Pacific Region*, 2018.

[66] S. Lee, J. Kim, and J. S. Vetter. OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 544–554, May 2016.

[67] T. Boku. Next Generation Interconnection For Accelerated Computing. http://nowlab.cse.ohio-state.edu/static/media/workshops/presentations/ExaComm16-Invited-Talk-7-Taisuke-Boku.pdf, June 2016.

付録 A

公表論文リスト

査読付き雑誌論文

1. 田渕晶大，中尾昌広，村井均，朴泰祐，佐藤三久．“演算加速機構を持つクラスタ向け PGAS 言語 XcalableACC の評価”，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.1，pp.17–29，2016 年 3 月．

査読付き国際会議論文

1. Akihiro Tabuchi, Masahiro Nakao, Mitsuhisa Sato. “A Source-to-Source OpenACC compiler for CUDA”, HeteroPar'2013, pp.178–187, Aachen, Germany, Aug. 2013.
2. Akihiro Tabuchi, Yasuyuki Kimura, Sunao Torii, Hideo Matsufuru, Tadashi Ishikawa, Taisuke Boku, Mitsuhisa Sato. “Design and Preliminary Evaluation of Omni OpenACC Compiler for Massive MIMD Processor PEZY-SC”, OpenMP: Memory, Devices, and Tasks: 12th International Workshop on OpenMP (IWOMP 2016), pp.293–305, Oct. 2016.
3. Akihiro Tabuchi, Masahiro Nakao, Hitoshi Murai, Taisuke Boku, Mitsuhisa Sato. “Implementation and Evaluation of One-sided PGAS Communication in XcalableACC for Accelerated Clusters”, 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp.625–634, May 2017.
4. Masahiro Nakao, Hitoshi Murai, Hidetoshi Iwashita, Akihiro Tabuchi, Taisuke Boku, Mitsuhisa Sato. “Implementing Lattice QCD Application with XcalableACC Language on Accelerated Cluster”, 2017 IEEE International Conference on Cluster Computing (CLUSTER), pp.429–438, HI, USA, Sep. 2017.
5. Akihiro Tabuchi, Masahiro Nakao, Hitoshi Murai, Taisuke Boku, Mitsuhisa Sato. “Performance Evaluation for a Hydrodynamics Application in XcalableACC PGAS Language for Accelerated Clusters”, Workshop on PGAS programming models: Experiences and Implementations (PGAS-EI), HPCAsia '18, pp.1–10, Jan. 2018.

査読なし論文 (研究会)

1. 田渕晶大，中尾昌広，佐藤三久．“Omni コンパイラによる OpenACC の試作”，第 136 回 HPC 研究発表会，Vol.2012-HPC-136 No.4，沖縄，2012 年 10 月
2. 田渕晶大，中尾昌広，佐藤三久．“NAS Parallel Benchmarks による Omni OpenACC コンパイラの評価”，第 199 回 ARC・第 142 回 HPC 合同研究発表会，Vol.2013-HPC-142 No.36，北海道，2013 年 12 月
3. 田渕晶大，村井均，朴泰祐，佐藤三久．“XcalableMP と OpenACC の統合による GPU クラスタ向け並列プログラミングモデル”，2014 年並列／分散／協調処理に関する『新潟』サマー・ワークショップ (SWoPP2014)，Vol.2014-HPC-145 No.39，新潟，2014 年 7 月
4. 田渕晶大，中尾昌広，村井均，朴泰祐，佐藤三久．“演算加速機構を持つ並列クラスタ向け PGAS 言語 XcalableACC の性能評価”，2015 年並列／分散／協調処理に関する『別府』サマー・ワークショップ (SWoPP2015)，Vol.2015-HPC-150 No.7，ビーコンプラザ 別府国際コンベンションセンター （大分），2015 年 8 月
5. 田渕晶大, 木村耕行, 鳥居淳, 松古栄夫, 石川正, 朴泰祐, 佐藤三久．”PEZY-SC 向け Omni OpenACC コンパイラの設計・試作”，第 154 回 HPC 研究発表会，Vol.2016-HPC-154 No.3，神奈川，2016 年 4 月
6. 田渕晶大，中尾昌広，村井均，朴泰祐，佐藤三久．“アクセラレータクラスタ向け PGAS 言語 XcalableACC の片側通信機能の実装と評価”，第 158 回 HPC 研究発表会，Vol.2017-HPC-158 No.26，静岡，2017 年 3 月

ポスター発表

1. Akihiro Tabuchi, Taisuke Boku, Hideo Matsufuru, Tadashi Ishikawa, Yasuyuki Kimura, Sunao Torii, Mitsuhisa Sato. “Preliminary Implementation of Omni OpenACC Compiler for PEZY-SC Processor”, The 6th AICS International Symposium, Feb. 2016.