JAIST Repository
https://dspace.jaist.ac.jp/

Title	ヘテロジニアス環境における効率的なキャッシュコヒーレンス機構
Author(s)	三崎, 豊弘
Citation	
Issue Date	2026-03
Type	Thesis or Dissertation
Text version	author
URL	https://hdl.handle.net/10119/20526
Rights	
Description	Supervisor:田中　清史, 先端科学技術研究科, 修士(情報科学)

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修士論文

ヘテロジニアス環境における効率的なキャッシュコヒーレンス機構

三崎　豊弘

主指導教員　田中　清史

北陸先端科学技術大学院大学
先端科学技術専攻
（情報科学）

令和８年３月

Abstract

GPU is one of the primary technologies to support the remarkable progress of AI in recent years. GPU has been known as fast parallel computing units for AI models and its improvement of processing power has contributed to that of AI. The improvement of GPU can be divided into performance enhancement as a computing unit and architectural enhancement as a heterogeneous computing platform. Then, looking closer at the architectural aspect, while data had to be copied from CPU memory to GPU memory in advance of GPU processing in earlier architectures, the enhancement has allowed users to share the memory between CPU and GPU at present. This resolves the performance bottleneck in transferring data between memories and makes the heterogeneous environment achieve higher performance. On the other hand, this improvement has brought new issues such as cache coherence by sharing memory between CPU and GPU.

A survey article on this issue mainly explains that there are roughly two research areas: coherence and memory management, but the research papers and the articles did not consider memory consistency models and employed a conventional way of requesting coherence in their proposed methods. This study analyzed user programs for CPU-GPU heterogeneous environments and extracted two important characteristics. We exploit them and present two proposed approaches.

The first proposed approach is to leverage Release Consistency Model in which Acquire and Release work as sync operation since the relaxed memory consistency model is the best suit for the heterogeneous environments. In addition, the proposed approach implements these sync functions and applies them to user programs.

The second proposed approach is to exploit the tendency in memory management through which the shared data are likely to be assigned consecutive memory addresses. If the system finds that there is a range of consecutive memory addresses at sync operation, a directory controller sends a single coherence request to CPU/GPU cache controllers for invalidation. This coherence request includes multiple addresses for invalidation and the corresponding lines will be invalidated at each cache controller until the last memory address in the address range. The proposed approach implements these mechanisms in the directory controller and each cache controller. This study proposes these approaches in the form of hardware mechanisms.

This study employs gem5 simulator for evaluation as there is no hardware with proposed functions at present and AMD provides its heterogeneous configurations to gem5 platform as sample codes. We customized this default configuration and implemented our proposed approaches. We also lever-

aged sample user programs provided for gem5 heterogeneous environments by AMD and implemented sync functions on Release Consistency model in these programs. We evaluated three user programs in the customized heterogeneous environment and analyzed the difference between our proposed approach and the default approach. In these evaluations, we focus on differences in the number of invalidation requests (probes) and execution time (Ticks) spent processing these invalidation requests. Comparing these results, our proposed approach reduces the number of probes and the total number of ticks for the execution of the probes for all scenarios.

概要

近年著しい発展を遂げている AI を支える主要な技術として GPU がある．GPU は様々な AI モデルを高速に並列実行する計算機であり，この GPU の処理性能の発展が AI の発展に大きく貢献している．その発展は，GPU の計算機としての性能改善によるものと AI モデルの実行基盤である CPU と GPU のヘテロジニアス環境というアーキテクチャの改善によるものがある．特にヘテロジニアス環境のアーキテクチャ面ではかつて CPU と GPU が共有するデータは，それぞれのプロセッサが管理するメモリへコピーして実行させる必要があったが，現在ではメモリを共有できる構成も取れるように発展した．この発展によりオーバヘッドの大きいメモリ間コピーの問題が解決され性能が向上した．しかし一方で，CPU と GPU がメモリを共有することで新たにキャッシュコヒーレンスの問題が出てきた．

この問題に対してサーベイ論文が示す研究領域は，大きく分けるとコヒーレンス処理とメモリ管理に分類されるが，これらのいずれの領域においてもこれまでの研究や文献ではヘテロジニアス環境のメモリコンシステンシモデルまで考慮した提案になっておらず，またコヒーレンス処理も従来の 1 つのメモリアドレスに対して 1 つの処理要求を出すといった逐次的な実行方式を改善する提案にはなっていない．そこで本研究では CPU と GPU からなるヘテロジニアス環境で使用されるユーザプログラムの分析を行い，そこから抽出した 2 つの特徴を用いて最適なメモリコンシステンシモデルと効率的なコヒーレンス処理の提案を行う．

1 つ目の特徴は，メモリコンシステンシモデルとして緩和されたメモリコンシステンシモデルが最適であることから，同期操作である Acquire と Release を使用する Release Consistency Model を採用し，これらの同期処理の手続きをユーザプログラムに対して実装した．2 つ目の特徴は，CPU と GPU で共有されるメモリ領域は連続したアドレス帯が獲得される傾向があることから，同期ポイントにおいてコヒーレンス対象のアドレスが連続している場合，アドレス帯を一括して無効化できるようにコヒーレンス処理を改善し，グローバルディレクトリコントローラと各プロセッサのキャッシュコントローラに対してこの処理機構を実装した．これらの実装により同期ポイントにおいて，無効化対象のメモリアドレスが連続したアドレスの場合，ディレクトリコントローラから CPU/GPU のキャッシュコントローラに対して 1 回だけ無効化処理要求を発行し，それを受領した CPU/GPU キャッシュコントローラ側で指定されたアドレスの範囲に含まれる全てのキャッシュラインを無効化させることで無効化処理全体に関わる実行時間の低減を行った．本研究ではこのような提案をグローバルディレクトリコントローラと各プロセッサのキャッシュコントローラに対するハードウェア機構として提案する．

本研究の提案内容を実装したハードウェアは現在存在しないため，評

価環境のベースとして gem5 シミュレータを用いたが，AMD 社からヘテロジニアス環境用に提供されているサンプル構成に対して提案手法を追加実装したものを使用した．また評価対象のユーザプログラムも AMD 社からサンプル提供されているものに Release Consistency Model に基づく同期処理の手続きを実装したものを用いた．この実装が加えられた 3 つのユーザプログラムを用いて，同期ポイントにおいて無効化処理の要求数 (Probe) と無効化処理の実行時間 (Tick) について Default 設定と提案手法との比較検証を行った．結果，無効化処理の要求数と無効化処理の実行時間に関して，いずれのユーザプログラムにおいても提案手法の方が Default 設定よりも有効な手法であることが示せた．

目次

Abstract **I**

概要 **III**

目次 **V**

図目次 **VII**

表目次 **IX**

1 はじめに **1**
1.1 導入 1
1.2 論文構成 3

2 関連研究 **4**
2.1 非対称マルチコアプロセッサに関するサーベイ論文 4
2.2 ヘテロジニアス環境のマイクロアーキテクチャ差分の吸収に関する研究 6
2.2.1 Spandex 6
2.2.2 DeNovo 9
2.3 ヘテロジニアス環境のメモリコンシステンシモデルに関する研究 12
2.3.1 Heterogeneous-race-free Memory Models 12
2.3.2 A Primer on Memory Consistency and Cache Coherence 2nd Edition 16

3 提案手法 **23**
3.1 メモリコンシステンシモデルとコヒーレンスについて . . . 23
3.1.1 メモリコンシステンシモデル 23
3.1.2 コヒーレンス 27
3.2 本研究で採用するメモリコンシステンシモデルについて . . 27

3.2.1 ヘテロジニアス環境で使用されるユーザプログラムの分析 28
3.2.2 本研究で提案するメモリコンシステンシモデル . . . 29
3.3 本研究で提案するコヒーレンス処理について 32
3.3.1 本研究で提案するコヒーレンス処理について 32
3.3.2 本研究で提案するハードウェア実装について 34

4 実験・評価 43
4.1 実験環境 43
4.1.1 シミュレータ選定 43
4.1.2 gem5 シミュレータについて 43
4.1.3 実験結果 48

5 おわりに 55

図目次

1.1 従来手法と CXL を用いた手法の概略 2

2.1 Spandex コヒーレンス処理例 7
2.2 DeNovo コヒーレンス処理実装例 11
2.3 HRF-direct 違反のサンプルコード処理 14
2.4 HRF-direct 違反の処理フロー図 16
2.5 ヘテロジニアス環境の定義図 17
2.6 CPU-GPU 環境でのコヒーレンス処理コード 1 18
2.7 CPU-GPU 環境での CPU 側のコヒーレンス処理例 1 18
2.8 CPU-GPU 環境での GPU 側のコヒーレンス処理例 1 19
2.9 CPU-GPU 環境でのコヒーレンス処理コード 2 20
2.10 CPU-GPU 環境での GPU 側のコヒーレンス処理例 2 20
2.11 CPU-GPU 環境での CPU 側のコヒーレンス処理例 2 21

3.1 Eager Release Consistency (2 プロセッサの例) 26
3.2 Eager Release Consistency (4 プロセッサの例) 26
3.3 Lazy Release Consistency (4 プロセッサの例) 27
3.4 CPU-GPU ヘテロジニアス提案環境 32
3.5 CPU_ACQUIRE の提案実装 35
3.6 CPU_RELEASE の提案実装 (前半部分) 36
3.7 CPU_RELEASE の提案実装 (後半部分) 37
3.8 GPU_ACQUIRE の提案実装 38
3.9 GPU_RELEASE の提案実装 (前半部分) 39
3.10 GPU_RELEASE の提案実装 (前半処理のつづき) 40
3.11 GPU_RELEASE の提案実装 (後半処理) 41

4.1 gem5 Discrete event simulation 例 46
4.2 gem5 Default Tick Config 例 47
4.3 gem5 Probe 数 square.cpp 49
4.4 gem5 Probe 処理に関わる Tick 数 square.cpp 50
4.5 gem5 Probe 数 2dshfl.cpp 51
4.6 gem5 Probe 処理に関わる Tick 数 2dshfl.cpp 52

4.7 gem5 Probe 数 MatrixTranspose.cpp 53
4.8 gem5 Probe 処理に関わる Tick 数 MatrixTranspose.cpp 54

表目次

2.1 Spandex メモリアクセス要求の集約表 6
2.2 DeNovo Core i L1 のコヒーレンス状態 10
2.3 DeNovo L2 のコヒーレンス状態 10

4.1 HSA Hardware Building Blocks 44
4.2 HSA Software Building Blocks 45
4.3 gem5 Probe 数 square.cpp 49
4.4 gem5 Probe 処理に関わる Tick 数 square.cpp 50
4.5 gem5 Probe 数 2dshfl.cpp 51
4.6 gem5 Probe 処理に関わる Tick 数 2dshfl.cpp 52
4.7 gem5 Probe 数 MatrixTranspose.cpp 53
4.8 gem5 Probe 処理に関わる Tick 数 MatrixTranspose.cpp 54

第 1 章
はじめに

1.1　導入

近年, 社会における AI の活用が進み，社会の様々な場面で AI が用いられるようになった．特に生成 AI の登場は，このトレンドをさらに推し進める大きな要因となっている．この動きに合わせて AI を支える技術もまた同様に発展しており，それはハードウェア，ソフトウェアを含む多くの分野に及んでいる [1].

このように AI を取り巻く様々な技術的進化が認められる中で，AI アプリケーションの計算基盤である CPU と GPU からなるヘテロジニアス環境もまた進歩を繰り返している分野となる．かつて CPU と GPU で構成されたヘテロジニアス環境では，ユーザプログラムを実行させるために，まず CPU 側でデータを用意し，それを CPU のメモリから GPU のメモリへバス経由でコピーする必要があった．また，GPU 側で実行した計算結果についても同様に，CPU 側で確認するために GPU 側のメモリから CPU 側のメモリに対してコピーする必要があった．この互いにデータを転送しあう実行方式では，処理のオーバヘッドが大きく効率が悪かったが，この分野における技術的な進歩により，現在では CPU と GPU がお互いにメモリを共有できる構成も取れるようになった．図 1.1[2] は，こうしたメモリ共有の技術的な例として，CPU と GPU が CXL というバスによりお互いのメモリにアクセスできるようになった図を示している．

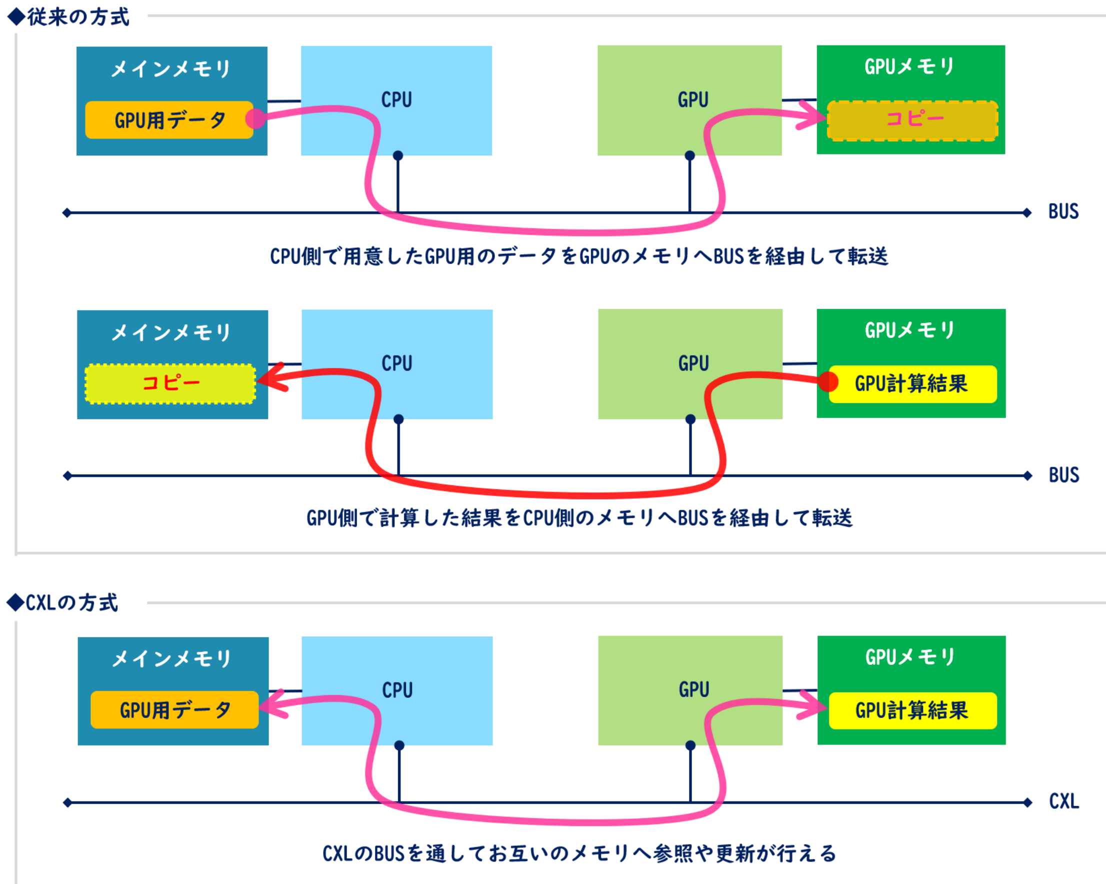

図 1.1: 従来手法と CXL を用いた手法の概略

CPU と GPU からなるヘテロジニアス環境において，CPU と GPU がメモリを共有できるようになることでデータ転送というオーバヘッドを回避できるようになった反面，プロセッサ間でメモリを共有することで新たにコヒーレンスの問題が出てきた．また，前述したようにヘテロジニアス環境では CPU と GPU の処理の目的が異なるため，CPU のマルチプロセッサ環境で取られている従来のコヒーレンス手法をそのまま GPU 側へ適用させる手法ではうまくいかず，新たなアプローチが必要になっている．現状，この問題に関する先行研究や文献はあるが，それらはヘテロジニアス環境でコヒーレンス処理を行う際の考慮点を説明しているものであったり，ヘテロジニアス環境において共有メモリにアクセスするために効率の良いメモリアクセス要求の整理を行うものであったりとヘテロジニアス環境におけるメモリコンシステンシモデルまで踏み込んだ研究はない．そこで本研究では，CPU と GPU からなるヘテロジニアス環境において，最適なメモリコンシステンシモデルとして緩和されたメ

モリコンシステンシモデルを採用し，それに従う効率の良いコヒーレンス手法を提案する．コヒーレンス手法については，従来の 1 つのキャッシュラインアドレスに対して 1 つの無効化処理を要求する方式から連続するキャッシュラインのアドレス帯が存在する場合，これらのアドレス群に対して無効化処理を 1 回だけ要求する方式を提案する．なお本提案の評価については，これらの提案を実現するハードウェアコヒーレンス機構を gem5 シミュレータに実装し評価を行った.

1.2 論文構成

本稿の構成は以下となる．第 1 章では本研究の背景や概要について述べた．第 2 章では関連研究について述べる．第 3 章では本研究の提案手法について述べ，第 4 章で提案手法を用いた実験の説明と評価結果を述べる．第 5 章では本研究のまとめ，および今後の課題を示す.

第 2 章

関連研究

2.1 非対称マルチコアプロセッサに関するサーベイ論文

関連研究を紹介するにあたり，まず本研究がヘテロジニアス環境を対象とした研究において，どのような分野として研究されているのかを示すために非対称マルチコアプロセッサのサーベイ論文 [3] で提示された研究の分類と内容について述べたい．まずヘテロジニアス環境に関する各研究の位置づけについてこのサーベイ論文では，非対称マルチコアプロセッサと呼ばれるヘテロジニアスアーキテクチャを大きく 2 つに分類している．それは対象のアーキテクチャが Reconfigurable かどうかであり，この分類をもとに Static と形容される通常のヘテロジニアスアーキテクチャと Reconfigurable なヘテロジニアスアーキテクチャについて, それぞれのデザインや管理手法に関する研究を各セクションに分けて紹介している．本研究は Static なヘテロジニアスアーキテクチャに分類されるが, この研究領域はデザインや管理手法のセクションに最適化手法のセクションを加えた大きく 3 つに分類される研究対象を構成している．この 3 つに分類された研究対象において，本研究と関連する内容はセクション 5 のヘテロジニアスアーキテクチャのデザインと管理手法に該当するが, 特にコヒーレンス関連は 5.10 Architecting heterogeneous ISA AMPs で紹介された研究となる．

セクション 5.10 では，全体で 7 つの研究について紹介しているが研究対象と内容から見ると大きく 3 つに分類できる．1 つ目の分類は，ヘテロジニアス環境においてマイクロアーキテクチャが異なるため，この差分を吸収させようとする研究である．これは例えば，ARM と MIPS のような異なるアーキテクチャを持つプロセッサだとそれぞれの ISA の実行状態はそれぞれの ISA に依存するが，この状態をなるべく同じ状態にしておくことで，異なるアーキテクチャ間において処理のマイグレーショ

ンを容易にさせようとする研究である．これらの研究については，異なるプロセッサのマイクロアーキテクチャの差分を吸収させる提案ではあるものの，その対象に CPU と GPU のヘテロジニアス環境は含まれていない．またプロセッサ間のマイグレーションを通した性能改善が提案内容となっているため，CPU と GPU のヘテロジニアス環境についてメモリコンシステンシモデルまで考慮したコヒーレンス手法の提案には至っていない．

2 つ目の分類は，ヘテロジニアス環境におけるスケジューリングに着目した研究である．この研究では，ソフトウェアもしくはハードウェア化されたスケジューラが異なるプロセッサアーキテクチャ間の処理をモニタしており，特定の条件に該当した際，当該タスクの割り当てを動的に変更させることで性能を改善させる提案となっている．例として，ソフトウェアがあるタスクを任意のコアに割り当てて実行させるが，未サポートのインストラクションによるフォルトが発生すると，ソフトウェアがそれを検知して異なるプロセッサへ処理をマイグレーションさせる手法となる．これらの研究についても，ヘテロジニアス環境を対象としているものの，メモリコンシステンシモデルの考慮とそれに従うコヒーレンス手法を扱う提案になっていない．

3 つ目の分類は，分散型共有メモリを対象にした研究である．これは例えば，モバイルシステムのように中央のプロセッサと周辺のプロセッサという処理関係を持つヘテロジニアスアーキテクチャに対して，中央プロセッサの処理の一部を周辺プロセッサが特殊なモジュールとして実行できるように中央プロセッサの処理をカプセル化したものを一定のタイミングで周辺プロセッサへ送付し，周辺プロセッサ側で実行させる方式である．これらのタイミング制御はソフトウェアで実行され，こうしたヘテロジニアス環境においてもデータの共有が行えるようにしている．また分散型共有メモリの研究は他にも各ヘテロジニアス環境において実行可能な ISA バイナリを事前に複数用意しておき，任意の処理に対してある時点で最適と判断されるアーキテクチャドメインで実行させる手法を取る．この方法により，割り当てられた処理はそのアーキテクチャドメイン内で完結するため処理の途中でデータを共有することはない．分散型共有メモリの研究例としてはこうした手法が提案されているが，ある一定のタイミングで異なるプロセッサ間のコヒーレンス同期を取るという点が CPU と GPU からなるヘテロジニアス環境のメモリコンシステンシモデルに近いものの，それを踏まえたコヒーレンス手法の研究とはなっていない．

本研究においてヘテロジニアス環境に対する研究の分類と内容を分析するためにサーベイ論文を調査したが，ヘテロジニアス環境の研究に対する方向性として本研究の類似性は見られたものの，本研究で提案するようなメモリコンシステンシモデルを考慮した上でコヒーレンス手法の効率化を提案するような関連研究はなかった．

2.2 ヘテロジニアス環境のマイクロアーキテクチャ差分の吸収に関する研究

2.2.1 Spandex

サーベイ論文で紹介された研究には含まれなかったものの，そこで分類されたマイクロアーキテクチャの差分吸収に関する研究分野についてさらにキャッシュコヒーレンスまで考慮して提案している研究について紹介したい．Spandex[4] は，CPU と GPU，もしくはアクセラレータからなるヘテロジニアス環境がメモリを共有する構成を対象とした研究である．この研究では，CPU や GPU など各プロセッサアーキテクチャのメモリアクセス要求の特徴を分類した上でラストレベルキャッシュ (LLC) から見て各プロセッサからのメモリアクセス要求が整理，集約された形で実行されるような実装提案を行っている．表 2.1 は，Spandex が提案する各プロセッサのメモリアクセス要求を整理した表となる．

コヒーレンスプロトコル	デバイス要求	Spandex 要求	アクセス粒度
GPU コヒーレンス	Read	ReqV	Line 単位
	Write	ReqWT	Word 単位
	RMW	ReqWT+data	Word 単位
DeNovo	Read	ReqV	Word/Line 単位
	Write	ReqO	Word 単位
	RMW	ReqO+data	Word 単位
	Owned Repl	ReqWB	Word 単位
MESI	Read	ReqS	Line 単位
	Write	ReqO+data	Line 単位
	RMW	ReqO+data	Line 単位
	Owned Repl	ReqWB	Line 単位

表 2.1: Spandex メモリアクセス要求の集約表

表 2.1 では，ヘテロジニアス環境において各プロセッサがそのコヒーレンスプロトコル内で実行する処理をデバイス要求の欄で記載し，それぞれが Spandex ではどのような要求に調和変更されているかを Spandex 要求の欄で表している．こうしてデバイスの種類により要求する処理は違えど，処理を受け付ける LLC 側からはできる限り共通化された要求へ

と整理統合することでヘテロジニアス環境の共有メモリに対するコヒーレンス処理の効率化を提案している．

次に，Spandex が提案するヘテロジニアス環境におけるコヒーレンス処理について具体的な例を説明する．図 2.1 は実装例として 4 つのケースについて説明したものである．

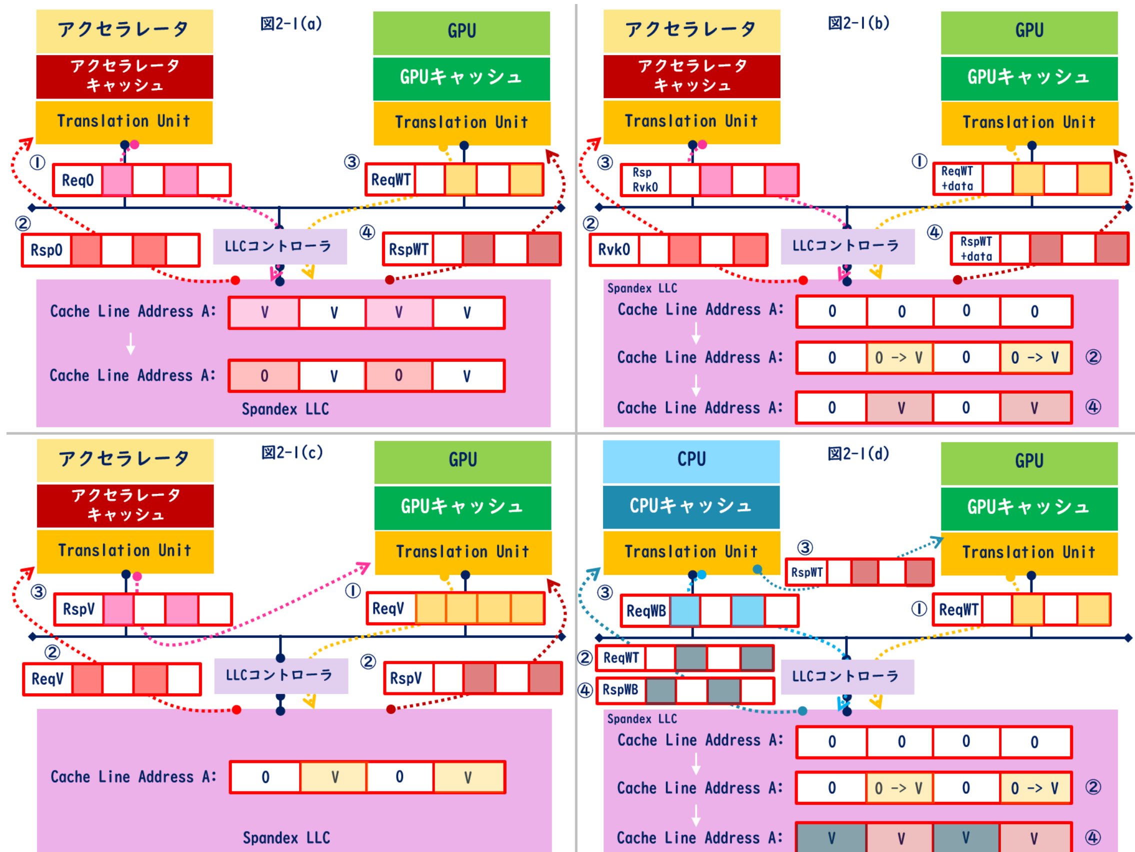

図 2.1: Spandex コヒーレンス処理例

図 2.1(a) では，キャッシュラインに対してワードレベルのオーナー要求を出す際のコヒーレンス処理について説明している．①でアクセラレータからアドレス A のキャッシュラインに対してワード単位のオーナー要求 (ReqO) を出す．この要求を受けた Spandex は②において LLC 上でワード単位のオーナー要求に対するレスポンス (RspO) を返す．このとき③で GPU から同じアドレス A のキャッシュラインに対して異なるワードへのライトスルー (ReqWT) 要求が実行される．④で Spandex はキャッシュライン全体に対する操作は行わず，ワード単位での要求に応じた処理を行い，GPU 側にレスポンス (RspWT) を返す．このように要求を

してくる主体の特徴に応じて，処理する内容を LLC 側で柔軟に対応させている．

図 2.1(b) では，図 2.1(a) と同様にアドレス A のキャッシュラインに対してワード単位のメモリアクセス要求を行えるプロセッサ間の振る舞いについて，書き込み時の処理例を示している．①で GPU からアクセラレータがオーナー (O) であるアドレス A のキャッシュラインに対して，ワード単位の書き込み要求 (ReqWT+data) を実行する．②でこの要求を受けた Spandex は，キャッシュライン全体に対する操作は行わず，当該ワードに対してだけオーナーの放棄要求 (RvkO) をアクセラレータへ送る．③で要求を受けたアクセラレータは，当該ワードに対してオーナーを放棄したレスポンス (RspRvkO) を返す．④でアドレス A のキャッシュラインに対して書き込まれたワードだけ，そのキャッシュ状態が最新であることを示す V(Valid) に変更され，GPU にレスポンス (RspWT+data) を返す．

図 2.1(c) では，アクセラレータと GPU 間においてキャッシュライン単位の読み込み要求が発生した場合の処理例を説明している．①で LLC にあるアドレス A のキャッシュラインに対して GPU から当該キャッシュライン全体の読み込み要求 (ReqV) が実行される．②で Spandex は要求を受けた当該キャッシュラインの一部のワードのオーナー (O) であるアクセラレータに対して，オーナーとなっているワードに対する読み込み要求 (ReqV) を実行する．一方で Spandex は当該キャッシュラインにおいてオーナーとなっていないワードについては，GPU へ読み込み (ReqV) に対するレスポンス (RspV) を返す．③でオーナーであるワードに対する読み込み要求を受けたアクセラレータは，要求されたワードに対するデータを GPU に対して返信 (RspV) する．

図 2.1(d) では，キャッシュライン単位の要求を行う CPU とワード単位での要求が行える GPU 間のコヒーレンス処理について説明している．①で GPU から LLC にある CPU がオーナー (O) となっているアドレス A のキャッシュラインに対してワード単位の書き込み要求 (ReqWT) が実行される．②でこの要求を受けた Spandex は，該当するワードに対するキャッシュのステータスをオーナーから最新の状態 (V) へと変更する．そして当該キャッシュラインのオーナーである CPU に対して GPU から受けたワード単位の書き込み要求 (ReqWT) を転送する．③でこの要求を受けた CPU は，要求を受けたキャッシュライン全体のオーナーを放棄し，書き込み要求 (ReqWT) を受けたワード以外のワードのデータを LLC に対してライトバック (ReqWB) する．また同時に CPU から GPU のワード単位の ReqWT に対するレスポンス (RspWT) を GPU へ返す．④で Spandex は，CPU がオーナー (O) であったワードのライトバックを受け，アドレス A のキャッシュラインに対して全ての状態を最新の状態 (V) にし，CPU から受けたワード単位のライトバック (ReqWB) に対してレスポンス (RspWB) を返す．このようにして，Spandex はキャッシュライン

に対して粒度の異なる要求を出すプロセッサ間の処理を LLC で調和しながら制御している．なお各プロセッサのキャッシュコントローラに対して TU と呼ばれるトランスレートユニットという層が付け加えられているが，この層は Spandex と各プロセッサの処理の境界としてメモリアクセス要求の変換処理を担っている．

マイクロアーキテクチャの差分を吸収するコヒーレンス手法として Spandex が提案されているが，この提案ではヘテロジニアス環境での共有メモリアクセス手法について深く検討されているものの，メモリコンシステンシモデルについての考慮はない．またコヒーレンス処理についても，従来通り 1 つのキャッシュラインアドレスに対して 1 つの要求を実行する方式に従っており，コヒーレンス処理自体の効率化の提案にはなっていない．

2.2.2 DeNovo

マイクロアーキテクチャの差分を吸収する研究手法のもう 1 つの関連研究として，Spandex の論文でも比較対象として紹介された DeNove[5] がある．DeNovo は複数のプロセッサがメモリを共有する環境において効率の良いコヒーレンス手法を提案するコヒーレンスプロトコルである．このため CPU のマルチプロセッサ環境でも，CPU と GPU のヘテロジニアス環境でも使用できるようになっており，既存の MESI などコヒーレンスプロトコルの新たな代替え手法として提案されている．その DeNovo の特長はコヒーレンスプロトコルのシンプルさにある．メモリアクセスに対する要求はできる限りシンプルに行い，それに加えてキャッシュラインに対してワード単位での読み書きを可能にし，キャッシュメモリ間の直接なデータのやり取りを許容する設計を取ることで従来の MESI などのプロトコルに見られたオーバヘッドを小さくしている．表 2.2 と表 2.3 は，それぞれ DeNove が定義するレベル 1 キャッシュ (L1) とレベル 2 キャッシュ (L2) のコヒーレンスの状態を表す．

L1 ステート	$Read_i$	$Write_i$	$Read_k$	$Register_k$	$ResponseforRead_i$	Writeback
Invalid	Update tag; Read miss to L2; Writeback(if needed);	Go to Registered; Reply to core i; Register request to L2; Write data; Writeback(if needed);	Nack to core k	Reply to core k	If tag match, go to Valid and load data; Reply to core i;	Ignore
Valid	Reply to core i	Go to Registered; Reply to core i; Register request to L2;	Send data to core k	Go to Invalid; Reply to core k;	Reply to core i	Ignore
Registered	Reply to core i	Reply to core i	Reply to core k	Go to Invalid; Reply to core k'	Reply to core i	Go to Valid; Writeback;

$Read_i$: Core i からの Read, $Read_k$: core k からの Read

表 2.2: DeNovo Core i L1 のコヒーレンス状態

L2 ステート	Read miss from core i	Register request from core i	Read response from memory for core i	Writeback from core i
Invalid	Update tag; Read miss to memory; Writeback(if needed);	Go to $Registered_i$; Reply to core i; Writeback(if needed);	If tag match, go to Valid and load data; Send data to core i;	Reply to core i; Generate reply for pending writeback to core i;
Valid	Data to core i	Go to $Registered_i$; Reply to core i;	X	X
$Registered_j$	Forward to core j; Done;	Forward to core j; Done;	X	if i==j, go to Valid and load data; Reply to core i; Cancel any pending writeback to core i;

表 2.3: DeNovo L2 のコヒーレンス状態

表 2.2 と表 2.3 から見ても分かるように DeNovo ではキャッシュの状態が少なく，L1，L2 キャッシュともに Invalid，Valid，Registered の 3 つの状態のみで構成される．ヘテロジニアス環境では，各プロセッサからキャッシュメモリに対してプロセッサ固有の様々なメモリアクセス要求が発生するが，それらに対するキャッシュの状態を少なく定義することでヘテロジニアス環境のコヒーレンス処理の複雑さを緩和している．図 2.2 は，このような特長を持つ DeNovo についてコヒーレンス処理の実装例を示す．

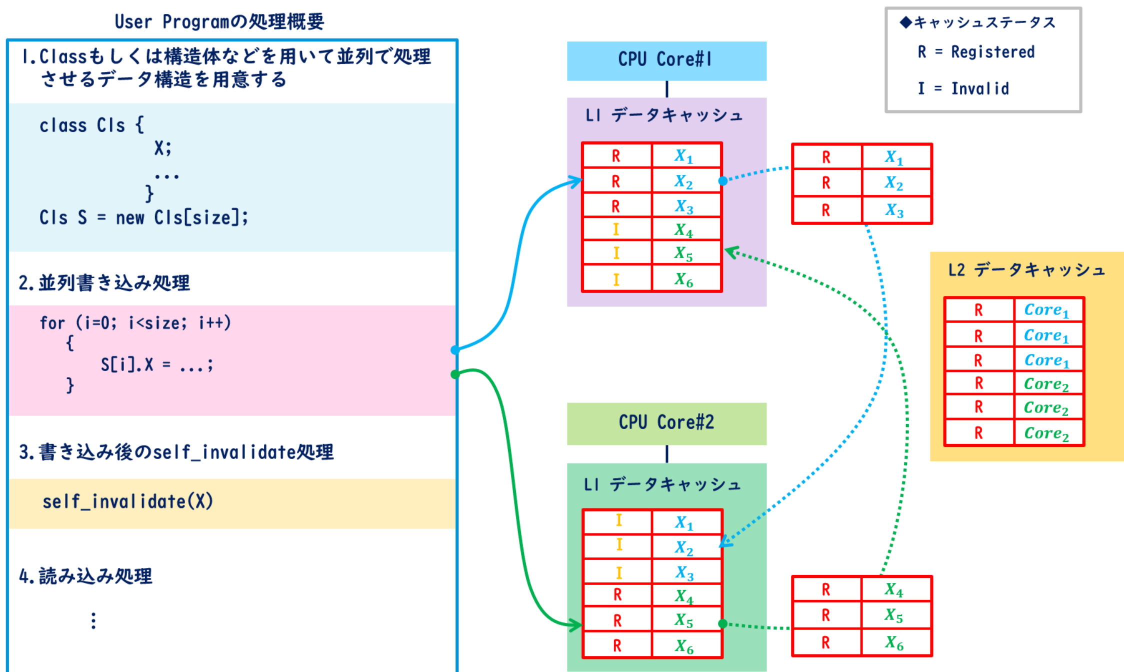

図 2.2: DeNovo コヒーレンス処理実装例

図 2.2 では図の左に示されているユーザプログラムをマルチコアプロセッサで実行した際，DeNovo が実装された L1, L2 キャッシュメモリがどのような処理と状態を取るのかを説明している．初期状態では，L1 と L2 キャッシュのラインは全て Valid 状態からスタートしている．まずユーザプログラムの 2. 並列書き込み処理で，S[i].X = ... という書き込み処理が for ループ内で繰り返し実行されると S[i].X に対して Core#1 と Core#2 による並列の書き込みが実行される．図 2.2 の右側には，このときの各キャッシュの状態が示されており Core#1 が S[1].X，S[2].X，S[3].X に対して X_1，X_2，X_3 というデータの書き込みを行い，Core#2 が S[4].X, S[5].X, S[6].X に対して X_4，X_5，X_6 というデータの書き込みを行う．この結果は，L2 キャッシュにそれぞれ R(Registered) という状態で通知され，L2 キャッシュでは S[1].X，S[2].X，S[3].X への書き込みデータである X_1，X_2，X_3 に対応するキャッシュライン上のワードが Core#1 による Registered を意味する R と $Core_1$ という状態に遷移している．同様に，S[4].X，S[5].X, S[6].X への書き込みデータである X_4，X_5，X_6 に対応するキャッシュライン上のワードが Core#2 による Registered を意味する R と $Core_2$ という状態に遷移している．次に，3. 書き込み後の self_invalidate 処理に移り self_invalidate(X)の処理を実行すると，Core#1 と Core#2 の L1 キャッシュラインにおいて，Registered 状態以外のキャッシュライン上のワードに対して Invalidate 処理が行われコヒーレンスが取られる．その後，ユー

ザプログラムは 4. 読み込み処理に入り 2. 並列書き込み処理で書き込んだ配列に対して読み込みを実行すると，このとき Invalid となっている状態のワードに対してのみ読み込みの要求をかける．こうしたコヒーレンス手法を採用することで，マルチプロセッサ環境やヘテロジニアス環境において頻繁に発生しうるキャッシュライン全体を単位として同期処理を行う False Sharing を避け性能向上を実現している．

以上が DeNovo に関する特長となるが，DeNovo も Spandex と同様にヘテロジニアス環境において各プロセッサのマイクロアーキテクチャの違いを調和させるコヒーレンス手法に特化した提案であり，メモリコンシステンシモデルまで考慮はされていない．またコヒーレンス処理自体も 1 つのキャッシュラインアドレスに対して，1 つのコヒーレンス処理要求が発生する点でコヒーレンス処理に対する提案はない．

2.3 ヘテロジニアス環境のメモリコンシステンシモデルに関する研究

2.3.1 Heterogeneous-race-free Memory Models

ヘテロジニアス環境のコヒーレンス研究について, 2.1 節のサーベイ論文で記載したようにマイクロアーキテクチャの差分を吸収する研究以外にメモリ関連の技術に着目した研究もある．ここではそうした研究の中でもヘテロジニアス環境のメモリコンシステンシモデルを対象とした研究 [6] を紹介する．この研究ではヘテロジニアス環境において Data Race Free(DRF) を確保するためにスコープという概念を用いて同期を行う範囲や順番を定義している．この定義に従いユーザプログラムを記述すれば，ヘテロジニアス環境においてもデータの競合が発生しないメモリコンシステンシモデルを構築できる点を紹介している．

この研究では，そのようなメモリコンシステンシモデルを構築するために 2 つのモデルを提案している．それらは Heterogeneous-Race-Free-Direct(HRF-direct) と Heterogeneous-Race-Free-indirect(HRF-indirect) であるが，下記 (1) と (2) にそれぞれの定義 (2.1) と (2.2) を記載する．

(1) HRF-direct の公式な定義

コンフリクトの定義

- Ordinary Conflict: 2 つのオペレーション op1 と op2 において，どちらも同じアドレスに対する操作で少なくとも一方が書き込み，かつ，少なくとももう一方が通常のデータ操作を行っている場合，このコンフリクトが発生する．

- Synchronization Conflict: 2 つの同期操作 op1 と op2 において，どちらも同じロケーションに対する操作で，少なくとも一方が書き込み (もしくは Read-Modify-Write)，かつ，異なるスコープに関して操作を行っている場合，このコンフリクトが発生する．

Sequentially Consistent Candidate Execution の定義

- プログラムオーダー ($\overrightarrow{po}$): op1 $\overrightarrow{po}$ op2 は，どちらも同じワークアイテムもしくはスレッドの処理を実行しており，かつ，op1 が op2 よりも先に完了する場合を示す．
- スコープ同期オーダー ($\overrightarrow{so_s}$): $\overrightarrow{so_s}$ において，どちらもスコープ S に対して実行され，かつ，Release rel1 が Acquire acq2 よりも先に完了する場合，rel1 は acq2 よりも先に示される．
- Heterogeneous-happens-before-direct($\overrightarrow{hhb.d}$): プログラムオーダーに従う全スコープ同期オーダーは，非反射推移閉包の和集合となり式 (2.1) として定義される．

$$\bigcup_{\forall S \in \mathbb{S}} (\overrightarrow{po} \cup \overrightarrow{so_s})^+ \tag{2.1}$$

  式 (2.1) では，$\mathbb{S}$ は実行する処理における全てのスコープの集合を表す．なお，閉包は内側の和集合にのみ適用される．
- Heterogeneous Race: Ordinary もしくは同期処理である op1 と op2 のペアが $\overrightarrow{hhb.d}$ において順番通りに実行されない場合，ヘテロジニアスの競合が発生する．
- Heterogeneous-race-free Execution: ヘテロジニアスの競合がない状態で処理を実行すれば，それはヘテロジニアスの競合フリーの実行となる．

(2) HRF-indirect の公式な定義

HRF-indirect では，HRF-direct の定義と同一の構造を持つが異なる happens-before 関係を持つ．

Heterogeneous-happens-before-indirect($\overrightarrow{hhb.i}$): プログラムオーダーに従う全てのスコープ同期オーダーは，非反射推移閉包となり式 (2.2) として定義される．

$$\left(\overrightarrow{po} \cup \bigcup_{\forall S \in \mathbb{S}} \overrightarrow{so_s}\right)^+ \tag{2.2}$$

式 (2.2) では，$\mathbb{S}$ は実行する処理における全てのスコープの集合を表す．

HRF-direct の定義では，プログラムの実行順序は同じワークアイテムもしくはスレッドが対象の場合，op1 が op2 よりも先に完了するとき $op1 \ \vec{po} \ op2$ と表記される．またスコープ S に対して rel1 と acq2 が定義されていた場合，rel1 が acq2 よりも先に処理されるとき $rel1 \ \vec{so}_s \ acq2$ と表記される．これらのプログラム内の実行順序やスコープに対する同期順序の定義を推移閉包と和集合を用いて組み合わせたものを HRF-direct として定義しているが，この定義に基づいた順番で処理が実行される場合，ヘテロジニアス環境においても競合フリーの状態になると説明している．

一方，HRF-indirect の定義では，HRF-direct と同様のプログラム実行順序とスコープ間の同期順序の定義を持ち，それらの関係が推移閉包を用いて定義されているが，HRF-direct とは異なりこの関係を和集合としたものとは定義していない．

式 (2.1) と (2.2) の定義について実際に HRF-direct 違反の例を図 2.3 のプログラム処理図と図 2.4 の処理フロー図を用いて説明する．

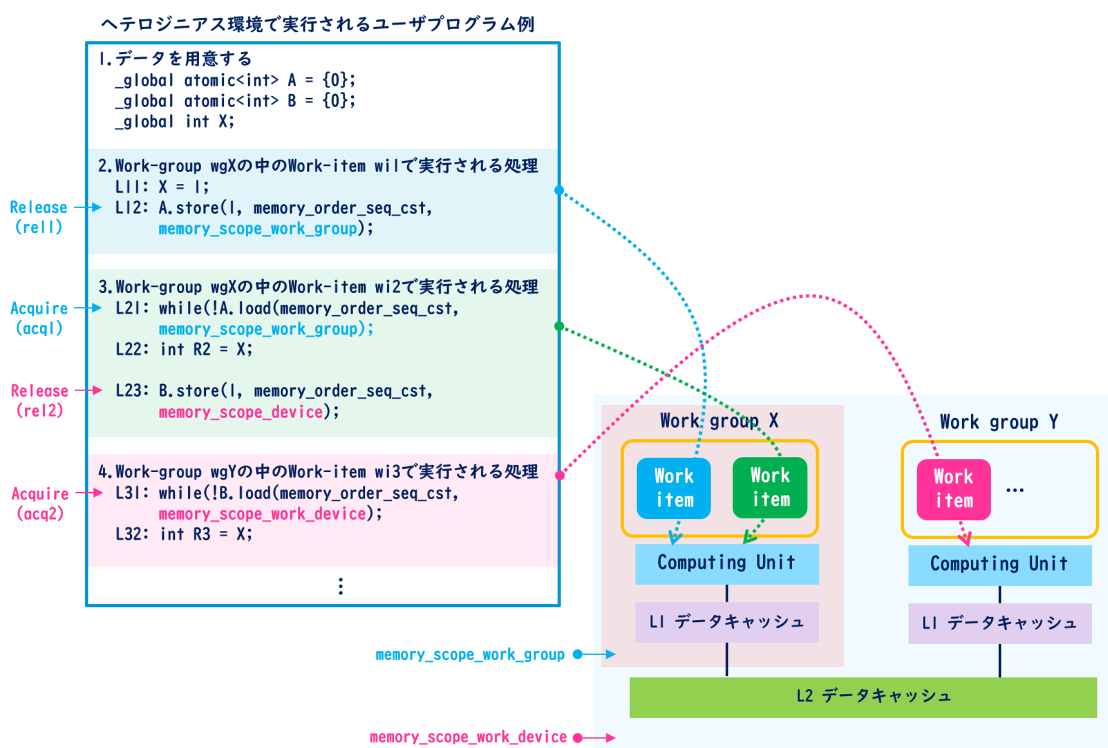

図 2.3: HRF-direct 違反のサンプルコード処理

図 2.3 に記載されたコードは HRF-direct 違反を例示するためのサンプルコードであるが，コードの左端の数字はプログラムの実行順序を示しており，色付きの文字で書かれた処理はそれぞれがどのスコープで同

期されるのかを示している．簡単な流れを説明すると，実行順番 11 行 (L11) において，X = 1 がワークグループ X のワークアイテム 1(wi1) にて実行される．実行順番 12 行 (L12) においてワークグループ X のスコープで 1 を A に書き込む．実行順番 21 行 (L21) にて同じワークグループのスコープで Release(rel1) が Acquire(acq1) よりも先に定義されているため，HRF-direct の定義により A への書き込みが先に実行される．これにより同じワークグループに属する全てのスレッドは，Acquire 時にワークアイテム 1(wi1) で書き込まれた値について最新のものを参照できる．その後，実行順番 23 行 (L23) において，今度はワークグループ X のワークアイテム 2(wi2) によりデバイスのスコープで B に対して 1 の書き込みが行われる．最後に，実行順序 31 行 (L31) でワークグループ Y のワークアイテム 3(wi3) よりデバイススコープの Acquire(acq2) が実行されることで，デバイススコープ内の Release(rel2) と Acquire(acq2) の実行関係が保証される．しかしながら実行順番 32 行 (L32) の R3 = X を実行するにあたり，ワークグループのスコープで閉じた処理とデバイスのスコープで閉じた処理では，それぞれのスコープ内の処理に関する同期は保証されているが，ワークグループのスコープとデバイスのスコープというスコープの関係で見ると同期の順番が保証されておらず HRF-direct の定義に反するコンフリクトが発生している．なおこの HRF-direct の実行順番に違反が発生したフローについて別の処理表現で示したものを図 2.4 で補足する．

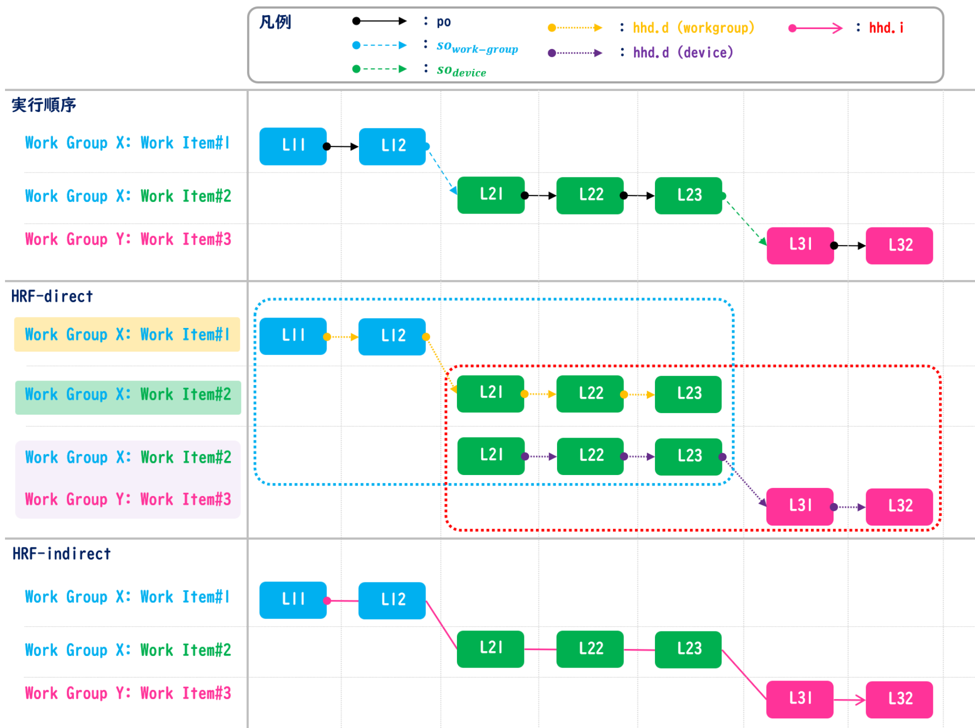

図 2.4: HRF-direct 違反の処理フロー図

この関連研究のようにヘテロジニアス環境においてメモリ管理に注目してメモリコンシステンシモデルを提案した研究もあるが，主にワークグループとデバイスのスコープを用いる GPU 側のメモリコンシステンシモデルを意識した提案になっているため，CPU と GPU からなるヘテロジニアス環境まで含めたメモリコンシステンシモデルとはなっていない．

2.3.2 A Primer on Memory Consistency and Cache Coherence 2nd Edition

ヘテロジニアス環境においてメモリ関連の技術に注目した研究として，最後に文献 [7] を紹介する．本文献では，キャッシュのコヒーレンスからメモリコンシステンシモデルまで幅広いトピックについて扱われているが，その中でもヘテロジニアス環境について述べている第 10 章の Consistency and Coherence for Heterogeneous Systems について紹介したい．当該章ではいくつかのヘテロジニアス環境のユースケースをもとにヘテロジニアス環境でのメモリコンシステンシモデルとコヒーレンス手法について説明している．図 2.5 はそれらのユースケースのベースとなるヘテロジニアス環境の構成である．

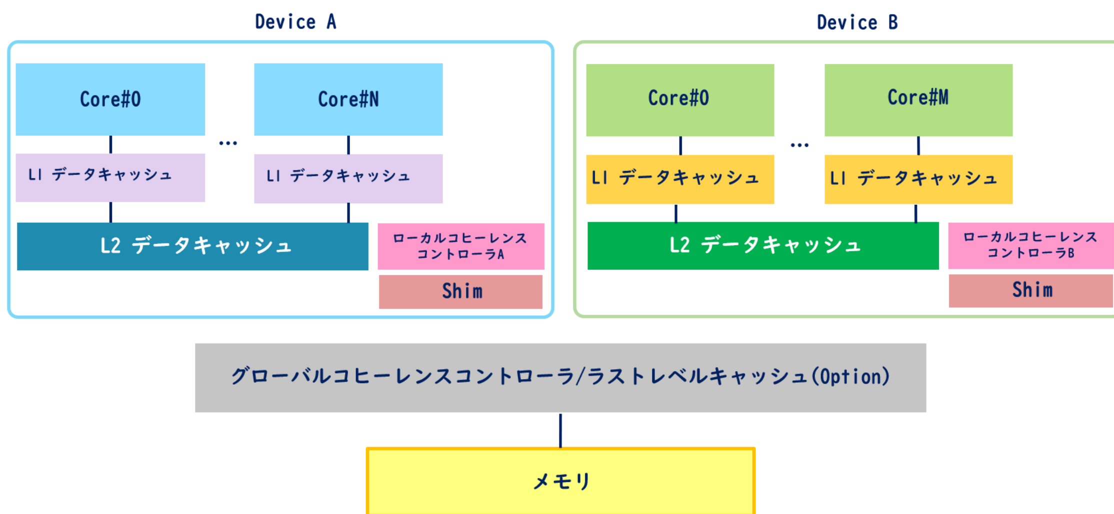

図 2.5: ヘテロジニアス環境の定義図

図 2.5 の構成の概要として，Device A，Device B と表記された各プロセッサはマルチコアで構成されており，それぞれのプロセッサ内で独自の L1 と L2 のキャッシュおよびキャッシュコントローラを持つ．また，各プロセッサ内では独自のコヒーレンスプロトコルが動作しており，それらはグローバルコヒーレンスコントローラを介してメモリを共有している．この構成をヘテロジニアス環境の前提として，各プロセッサが特定のメモリコンシステンシモデルとそれに従うコヒーレンスプロトコルを採用した場合，グローバルコヒーレンスコントローラが果たすべき役割についてユースケースを使って紹介している．それらのユースケースについて関連研究として 2 つのケースを取り上げる．1 つ目は，CPU が Sequential Consistency(SC) という強いメモリコンシステンシモデルを採用し，コヒーレンスプロトコルとして MESI を動作させる．一方で，GPU は Lazy Release Consistency(LRC) という弱いメモリコンシステンシモデルを採用し，コヒーレンスプロトコルは Self-Invalidate を実行する一般的な GPU のコヒーレンスプロトコルを採用する．このモデルにおいて図 2.6 のプログラムを実行した際，ヘテロジニアス環境としてどのような考慮が必要になるか図 2.7 で CPU 側の考慮点を，図 2.8 で GPU 側の考慮点をそれぞれ示している．

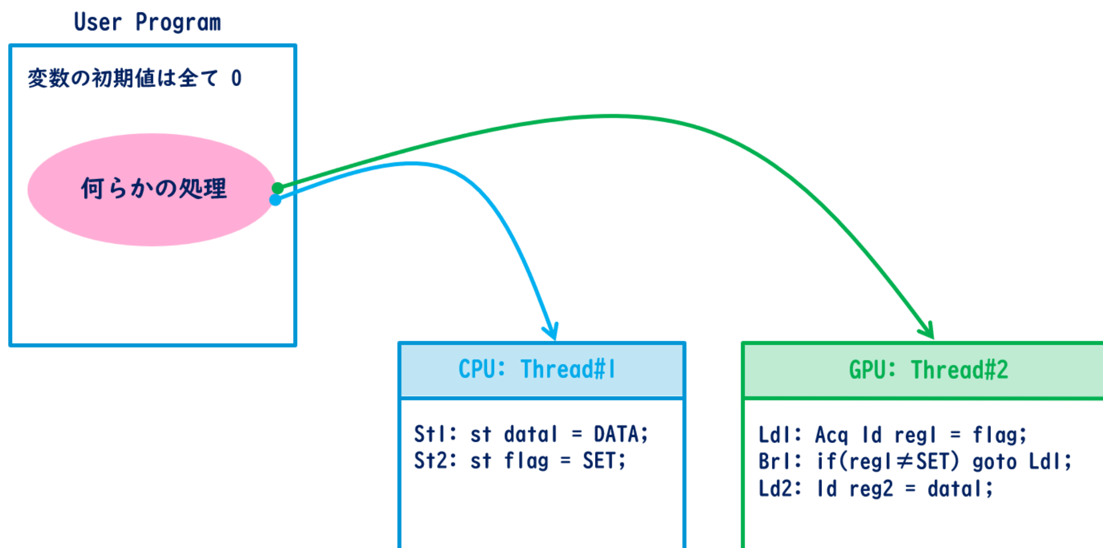

図 2.6: CPU-GPU 環境でのコヒーレンス処理コード 1

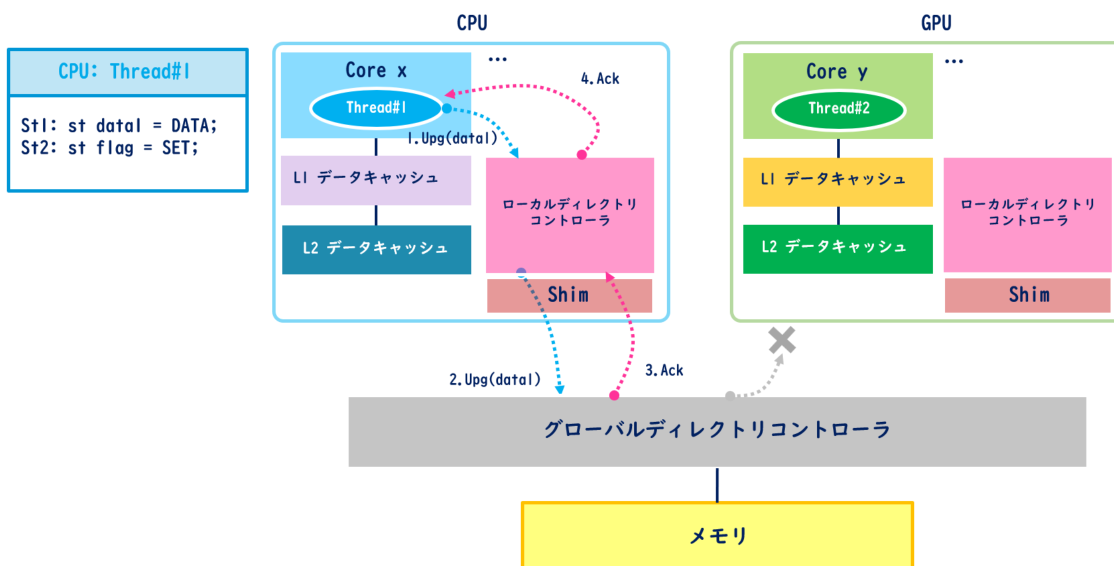

図 2.7: CPU-GPU 環境での CPU 側のコヒーレンス処理例 1

図 2.7 で実行される CPU の処理では CPU が St1 と St2 で，それぞれの変数にデータをストアするが，この処理は CPU のローカルキャッシュコントローラに対して該当するキャッシュアドレスのデータへの更新 (Upg(data1)) を発生させる．この更新要求を受けて CPU のローカルキャッシュコントローラは，グローバルディレクトリコントローラに対して

も同様にキャッシュアドレスのデータ更新を要求するが，グローバルディレクトリコントローラは GPU が LRC のメモリコンシステンシモデルのため，このストアにともなう Invalid を GPU 側に対して実行しない．代わりにグローバルディレクトリコントローラは，CPU のローカルキャッシュコントローラに対して更新要求に対する Ack を返す．

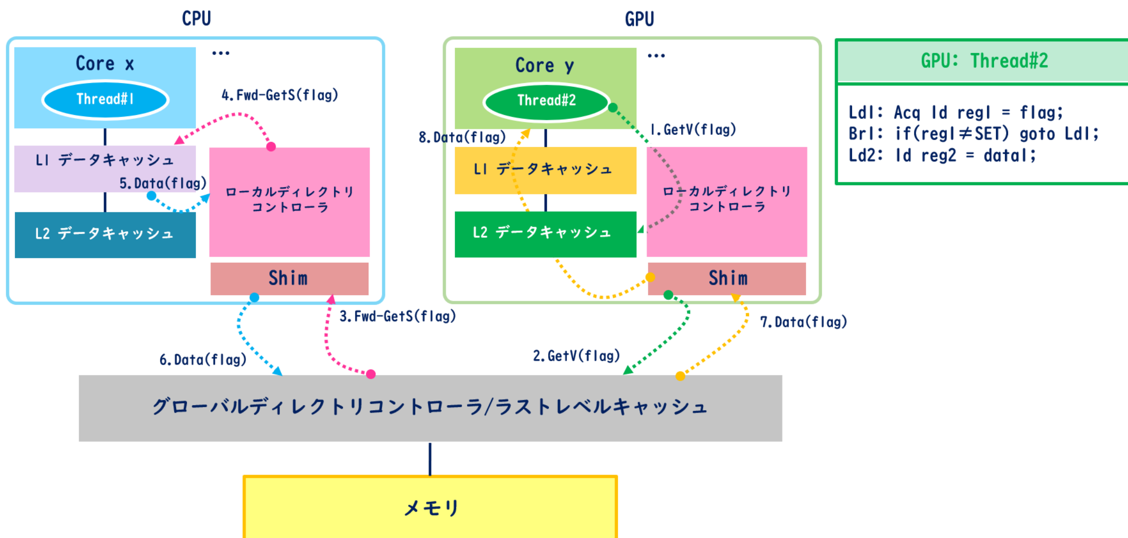

図 2.8: CPU-GPU 環境での GPU 側のコヒーレンス処理例 1

次に図 2.8 で示された Ld1 から始まる GPU の処理を見ると，GPU が Ld1 において flag のデータを読み込む際，GPU は自身の L2 キャッシュにデータ読み込み要求 (GetV(flag)) を伝える．GPU の L2 キャッシュは，この要求を受けた時点で当該データを持っていないため LLC に GetV(flag) を要求する．この要求を受けた LLC では，ディレクトリ情報を検索し当該データは CPU にて更新されたものであると確認した後，GetV(flag) 要求を GetS(flag) のデータ共有要求へ変更して CPU のキャッシュコントローラへ転送する．この要求を受けた CPU のキャッシュコントローラは，さらに CPU の L1 キャッシュに対して GetS(flag) 要求を伝達する．最終的に最新のデータを持つ CPU の L1 キャッシュからデータが GPU まで転送されて，GPU の Ld1 が完了する．

図 2.6 で示したコードの処理について，今度は GPU 側で Release 処理を加えたストアを行い，それを CPU で読み込む場合について検討する．図 2.9 は，その内容を反映したコード処理図であり，図 2.9 をもとに図 2.10 で GPU 側，図 2.11 で CPU 側のコヒーレンス処理における考慮点を示す．

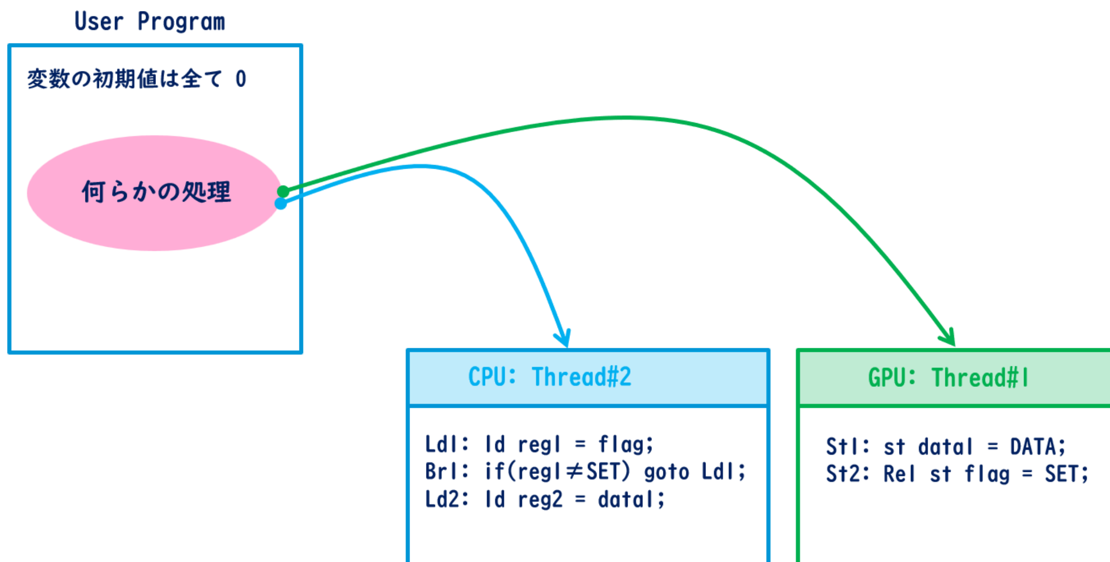

図 2.9: CPU-GPU 環境でのコヒーレンス処理コード 2

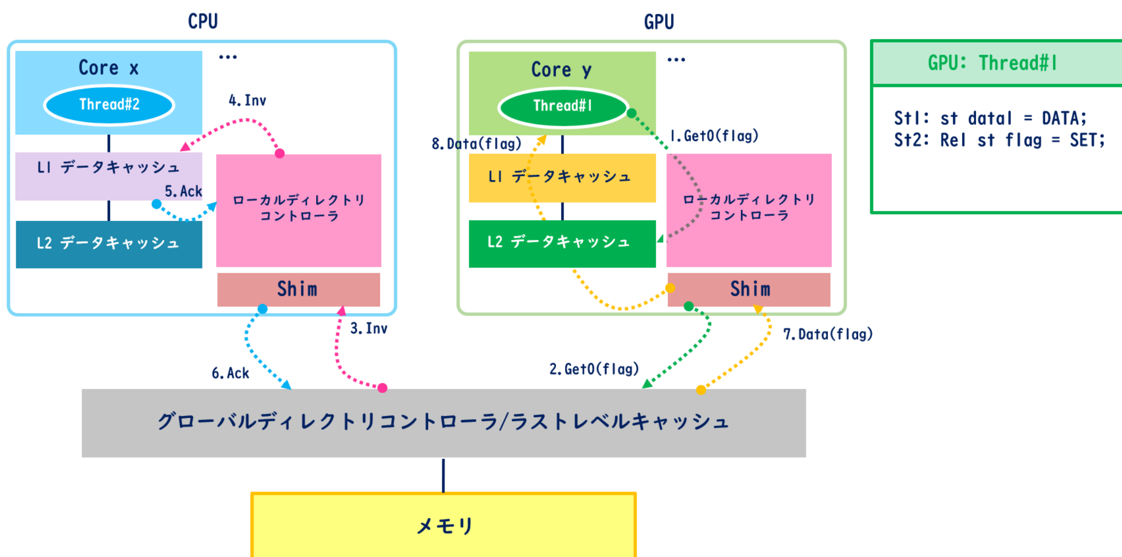

図 2.10: CPU-GPU 環境での GPU 側のコヒーレンス処理例 2

図 2.9 のコード処理に基づき図 2.10 ではまず GPU 側でストア (St1, St2) を実行するが，St2 は Release(Rel) に従い実行されるため，この処理に続く GPU 側での同期処理 (Self-Invalidate) に備えてローカルの L2 キャッシュにオーナー要求 (GetO(flag)) を実行する．この要求を受けて GPU の L2 キャッシュではグローバルディレクトリコントローラに対して同

様に GetO(flag) を要求する．グローバルディレクトリコントローラでは，この要求を受けるとコヒーレンスの必要から CPU 側のキャッシュに残っている当該データを無効にするため，CPU のキャッシュコントローラに対して Invalid(Inv) を実行する．CPU 側では Inv 要求が伝達されると自身のキャッシュにある当該アドレスのエントリを無効化しグローバルディレクトリコントローラに対して Ack を返す．この Ack の受領によりグローバルディレクトリコントローラでは当該アドレスのデータに対するコヒーレンスが確保されたと判断し，GPU 側から GetO(flag) 要求を受けていたデータを送付する．

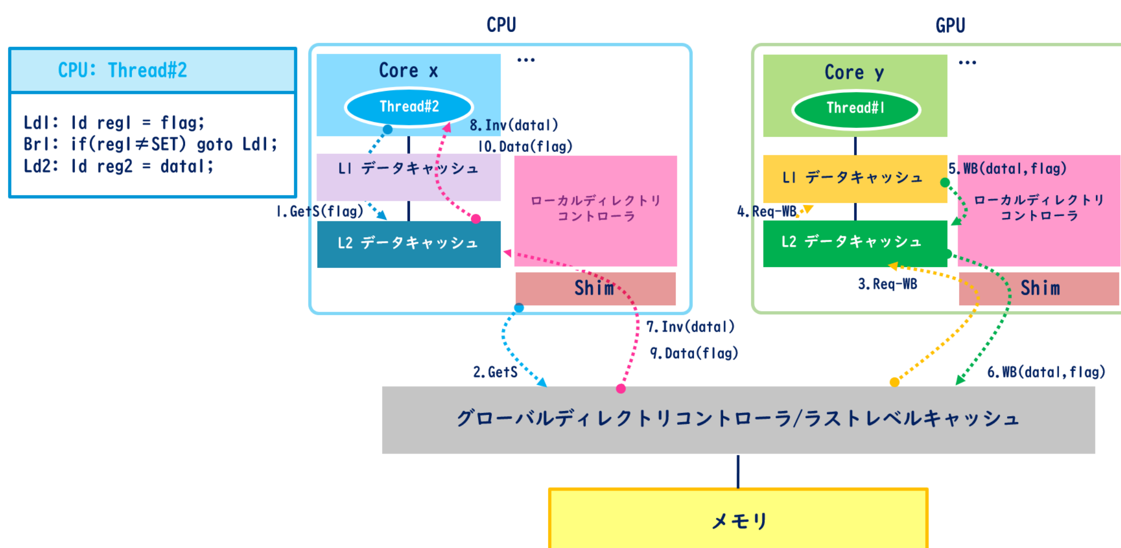

図 2.11: CPU-GPU 環境での CPU 側のコヒーレンス処理例 2

図 2.11 では図 2.10 の GPU 側のストア (GetO(flag)) 後の CPU 処理を表すが，GPU 側での更新にともない CPU 側で無効化された flag データを読み込むために読み込み要求 (GetS(flag)) をローカルの L2 キャッシュに対して実行する．この要求を受けて L2 キャッシュコントローラは，グローバルディレクトリコントローラに対して GetS を要求する．ここでグローバルディレクトリコントローラは，ディレクトリ情報を検索し flag データのオーナーである GPU に対してライトバック要求 (Req-WB) を実行する．このライトバックされたデータがグローバルディレクトリコントローラを経由して CPU まで伝達されることでコヒーレンスが実現される．また，GPU 側からのライトバック処理により GPU のストア処理 (St1) で書き込まれたデータ (data1) もライトバックされることから，グローバルディレクトリコントローラはライトバックされた data1 に対しても CPU 側に Invalid を実行することで別のコヒーレンスも実現している．

以上のユースケースを通して本文献ではヘテロジニアス環境で考慮が必要なコヒーレンス手法を実例を通して説明しているが，これらの説明はヘテロジニアス環境でコヒーレンス手法を検討する上で重要ではあるものの概要レベルの内容に留まっており，実装面での考慮やコヒーレンス処理自体を効率化させる方法については記載されていない．

本研究の関連研究として，まずサーベイ論文からヘテロジニアス環境のコヒーレンス研究について，その対象や種類を調査した．そしてそこで示されたコヒーレンス研究分野の 2 つの方向性に関して追加でそれぞれ 2 つの論文を調査した．しかしながら，いずれも CPU と GPU からなるヘテロジニアス環境において最適なメモリコンシステンシモデルの選定に関する提案や，それを踏まえた効率的なコヒーレンス手法の提案はなかった．

第 3 章

提案手法

3.1 メモリコンシステンシモデルとコヒーレンスについて

本研究の提案にあたり，ヘテロジニアス環境に適したメモリコンシステンシモデルの検討と，そのメモリコンシステンシモデルに従う効率の良いコヒーレンス処理の検討という 2 つの検討事項について考慮する必要があるが，まずこれらの提案手法の前提としてメモリコンシステンシモデルとコヒーレンスについて説明をしたい．

3.1.1 メモリコンシステンシモデル

文献 [7] で記載された定義を踏まえると，メモリコンシステンシモデルとは，コンシステンシ，メモリコンシステンシ，またはメモリモデルと呼ばれ，共有メモリに対する読み書きのルールを正しく定義するものである．特にマルチスレッドやマルチコア，またはヘテロジニアス環境などにおいて共有メモリを使用する際，それぞれの処理を実行する主体が共有メモリに対する読み書きの操作において許容される振る舞いを定義する．例えば，あるスレッド A があるデータ X に対して書き込みを行ったとき，そのデータ X を読み込むスレッド B は最新のデータが読み込まれる保証がなされている必要がある．ただし，それをどの時点で保証するかはアプリケーションや実行環境に委ねられた問題であり，必ずしもスレッド A がデータ X を書き込んだ直後に誰から見てもデータ X の値が最新であることが分かるようにする必要はない．スレッド B が最新のデータ X を読み込めるという保証があるならば，それがいつの時点で最新の値に更新されるべきかは最適と考えられる任意のタイミングに委ねることができる．こうした共有メモリに対する読み書き操作の振る舞いを定義したものをメモリコンシステンシモデルと呼ぶ．

このように定義されるメモリコンシステンシモデルには，複数の種類が存在しており代表的なものをいくつか紹介すると，プログラムの実行順序どおりに読み書きが実行されることを保証するモデルが Sequential Consistency(SC) と呼ばれ，式 (3.1) から (3.5) で定義される．

(1) SC の定義：

$$\text{If } L(a) < p\ L(b) \Rightarrow L(a) < m\ L(b) \tag{3.1}$$

$$\text{If } L(a) < p\ S(b) \Rightarrow L(a) < m\ S(b) \tag{3.2}$$

$$\text{If } S(a) < p\ S(b) \Rightarrow S(a) < m\ S(b) \tag{3.3}$$

$$\text{If } S(a) < p\ L(b) \Rightarrow S(a) < m\ L(b) \tag{3.4}$$

$$\text{Value of } L(a) = \text{Value of } \mathrm{MAX}_{<m}\{S(a) \mid S(a) < m\ L(a)\} \tag{3.5}$$

L(x) は，x をロードする処理を意味し，S(x) は，x をストアする処理を意味する．また X<pY は，プログラムの実行順序を表し，X が Y よりも先に実行されることを表す．同様に X<mY は，メモリアクセスの実行順序を表し，X が Y よりも先にメモリにアクセスすることを表す．式 (3.1) から (3.5) で定義されたとおり，SC ではプログラムの実行順序どおりにメモリアクセスが行われることを保証するメモリコンシステンシモデルとなる．なお式 (3.5) は，同じアドレスに対するロードは，直近のストアの値を読み込むことを定義している．

SC 以外のメモリコンシステンシモデルとして，基本的に SC の定義式に従うが式 (3.4) について，ロードとストアが異なるメモリアドレスの場合メモリアクセス順序を定義しない Total Store Order(TSO) と呼ばれるメモリコンシステンシモデルや FENCE を使うことでより読み書きの順序の制約を緩めた Relaxed Consistency(XC)，Acquire と Release という手続きを用いてさらに読み書き順序の制約を緩めた Release Consistency(RC) などがある．なおそれぞれの定義について，XC は式 (3.6) から (3.14) で，RC は式 (3.15) から (3.20) で示す．

(2) XC の定義：

$$\text{If } L(a) < p \text{ FENCE} \Rightarrow L(a) < m \text{ FENCE} \tag{3.6}$$
$$\text{If } S(a) < p \text{ FENCE} \Rightarrow S(a) < m \text{ FENCE} \tag{3.7}$$
$$\text{If FENCE} < p \text{ FENCE} \Rightarrow \text{FENCE} < m \text{ FENCE} \tag{3.8}$$
$$\text{If FENCE} < p\ L(a) \Rightarrow \text{FENCE} < m\ L(a) \tag{3.9}$$
$$\text{If FENCE} < p\ S(a) \Rightarrow \text{FENCE} < m\ S(a) \tag{3.10}$$
$$\text{If } L(a) < p\ L\text{'}(a) \Rightarrow L(a) < m\ L\text{'}(a) \tag{3.11}$$
$$\text{If } L(a) < p\ S(a) \Rightarrow L(a) < m\ S(a) \tag{3.12}$$
$$\text{If } S(a) < p\ S\text{'}(a) \Rightarrow S(a) < m\ S\text{'}(a) \tag{3.13}$$
$$\text{Value of } L(a) = \text{Value of MAX}_{<m} \{S(a) \mid S(a) < m\ L(a) \text{ or } S(a) < p\ L(a)\} \tag{3.14}$$

(3) RC の定義 (この定義での矢印はメモリアクセスの順番を示す)：

$$ACQUIRE \rightarrow Load, Store \tag{3.15}$$
$$Load, Store \rightarrow RELEASE \tag{3.16}$$

SC ordering of ACQUIREs and RELEASEs:

$$ACQUIRE \rightarrow ACQUIRE \tag{3.17}$$
$$ACQUIRE \rightarrow RELEASE \tag{3.18}$$
$$RELEASE \rightarrow ACQUIRE \tag{3.19}$$
$$RELEASE \rightarrow RELEASE \tag{3.20}$$

このようにプログラムの実行順序と実際のメモリへのアクセス順序について制約の強度に応じて強いメモリコンシステンシモデル，または弱いメモリコンシステンシモデルと形容するが，ここで緩和された (弱い) メモリコンシステンシモデルの 1 つである Release Consistency についてさらに説明を加えたい．

Release Consistency は，Acquire と Release という手続きを使ってコヒーレンスのタイミングを制御する手法を取るが，このタイミングについても 2 つの手法が存在する．1 つは，Release の手続きのタイミングでコヒーレンスを実行する Eager Release Consistency と呼ばれる手法で，もう 1 つは Acquire の手続きのタイミングでコヒーレンスを実行する Lazy Release Consistency と呼ばれる手法となる．これらの手法の違いを端的に表す図 [8] を図 3.1 から図 3.3 に示す．

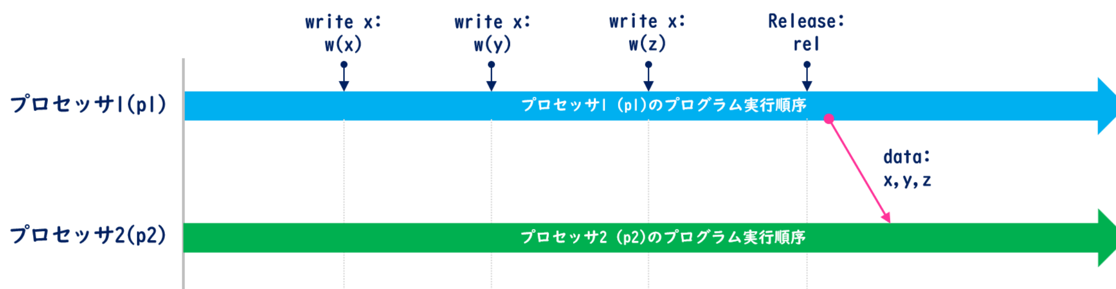

図 3.1: Eager Release Consistency (2 プロセッサの例)

図 3.1 では p_i はプロセッサを表しており，p_1 では，x，y，z という 3 つのデータに対して w() を用いて書き込みを順番に行っている．この書き込まれたデータを Release(rel) のタイミングで p_2 に伝えるのが Eager Release Consistency となる．さらにプロセッサ数が増えた場合には図 3.2 のように各プロセッサの Release(rel) タイミングで，都度各プロセッサが書き込んだデータ x に対してコヒーレンス処理が実施される．

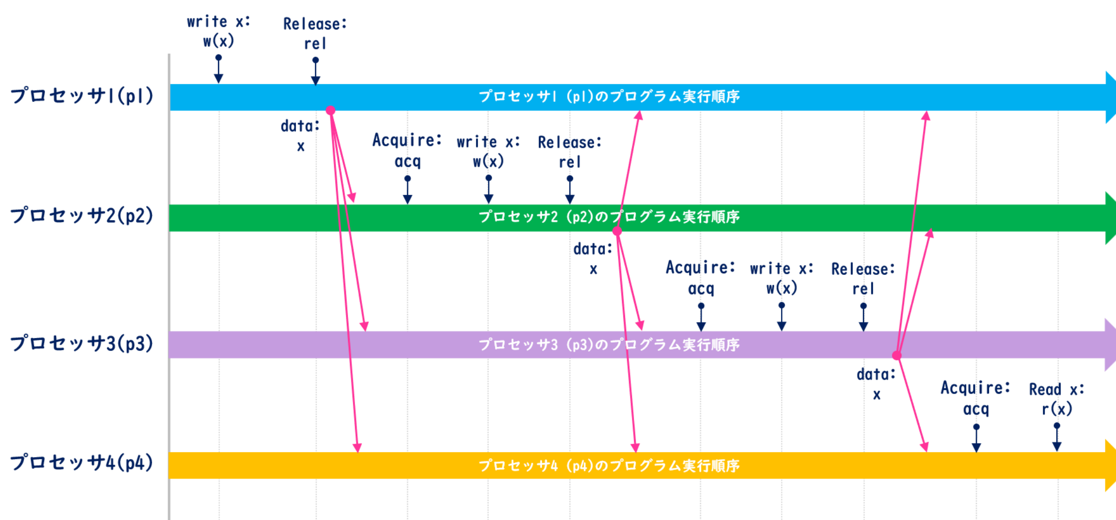

図 3.2: Eager Release Consistency (4 プロセッサの例)

一方，Lazy Release Consistency は，各プロセッサの Acquire(acq) のタイミングで各プロセッサが必要とするデータに対してのみコヒーレンス処理を実施する手法となり，図 3.3 のような処理フローをたどる．Eager Release Consistency との違いとして，Release(rel) 時にコヒーレンスを実施しておらず，各プロセッサの Acquire(acq) の実行タイミングに応じて各プロセッサが必要とするデータに対してコヒーレンスを実施している．

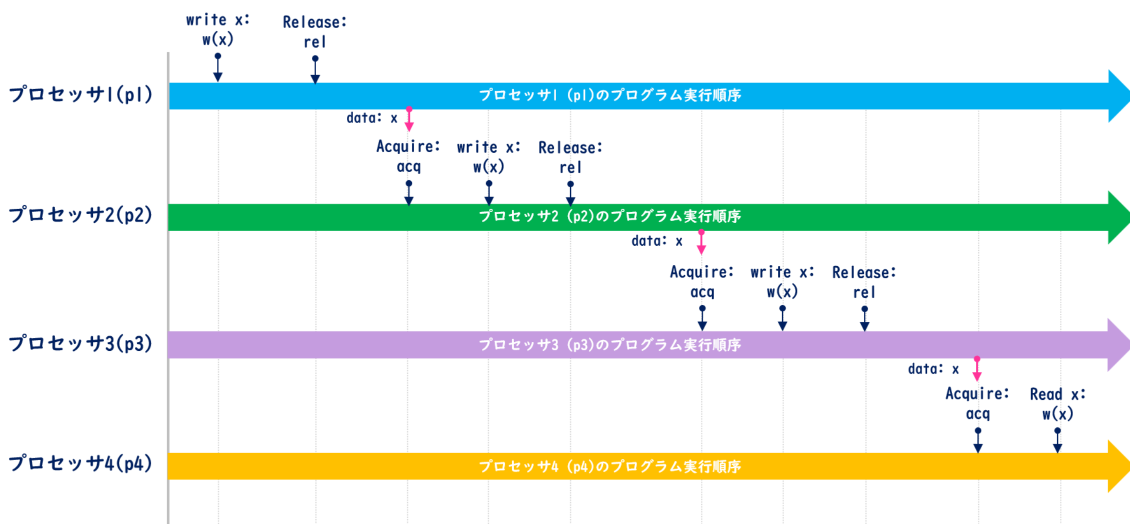

図 3.3: Lazy Release Consistency (4 プロセッサの例)

3.1.2 コヒーレンス

共有メモリやキャッシュのデータには，読み込みと書き込みのアクセスが頻繁に発生しうるが，読み込み時には常に最新の値が提供される必要がある．これを保証する仕組みをコヒーレンスと呼ぶ．コヒーレンスは，これを実現するために複数の状態とアクションの組み合わせで定義されたプロトコルの体系を持ち，MESI，MOESI などが有名である．これ以外にも，自身の持つデータを自身のタイミングで無効化する Self-Invalidate を行う GPU のコヒーレンスプロトコルなどもある．スレッドやコア，プロセッサなど複数の処理の主体がメモリを共有する構成において，こうしたデータの一貫性を保証する仕組みは重要であり，それを支える技術がコヒーレンスとなる．

3.2 本研究で採用するメモリコンシステンシモデルについて

本研究は，CPU と GPU で構成されたヘテロジニアス環境を対象としている．そこでこの環境に最適なメモリコンシステンシモデルを提案するにあたり，こうした環境で使用されるユーザプログラムの分析を行ない，そこで抽出した特徴をもとに提案内容の説明を行う．

3.2.1 ヘテロジニアス環境で使用されるユーザプログラムの分析

ヘテロジニアス環境で最適なメモリコンシステンシモデルを検討するにあたり，AMD 社がヘテロジニアス環境のために提供したサンプルコード [9] を分析した．Algorithm 1 がそのサンプルコードとなる．

Algorithm 1 AMD APU MI300A のサンプルコード

```
 1: procedure MAIN
 2:     N ← WIDTH × HEIGHT
 3:     A, B, C ← AlignedAllocate(N × sizeof(double), 128)
 4:
 5:     for i = 0 to N − 1 do
 6:         A[i] ← 1.1
 7:         B[i] ← i/1.3
 8:     end for
 9:
10:     AddKernel(grid, block, A, B, C, WIDTH, HEIGHT)
11:     hipDeviceSynchronize()
12:
13:     Free(A, B, C)
14:
15: end procedure
```

Algorithm 1 の処理は，5–8 行目で CPU にて配列データの初期値を設定し，それらを使った GPU カーネルが 10 行目に起動する．その後，GPU での計算が終了すると，11 行目で GPU と後続の CPU 処理の待ち合わせ同期処理が入り，13 行目から再び CPU 処理が始まる．13 行目以降は CPU 側でメモリを解放して処理が終了するという流れになる．これはサンプルコードではあるが，CPU と GPU で構成されたヘテロジニアス環境でのユーザプログラムの処理方式についての特徴をよく表している．それは，まず CPU 側で GPU が処理するためのデータを準備し，それを使って GPU が計算を行う．そして最後に再び CPU 側に処理が戻るという基本的な処理の流れである．ここで注目する点は，このヘテロジニアス環境において CPU と GPU がある時点において同時にデータにアクセスすることはなく，CPU と GPU の共有データへの読み書きのアクセスは常に一方向に限定されているという点である．この点を考慮すると，このヘテロジニアス環境において CPU と GPU が共有データに対して即時にコヒーレンスを取る必要はなく，CPU と GPU の処理境界においてコヒーレンスを取れば良いことが分かる．Algorithm 1 のサンプルコードを例

にすれば，CPU で 5–8 行目に行った共有データへの書き込みに対するコヒーレンスは，5–8 行目の処理時に都度取る必要はなく，10 行目の GPU 処理が始まる直前でコヒーレンスを取ればデータの一貫性は保証できる．同様に，GPU カーネルの処理が終了し，GPU と後続の CPU 処理との待ち合わせ同期処理が 11 行目取られた後，13 行目から再び CPU 側の処理に戻るが，11 行目の待ち合わせ同期処理後のタイミングでコヒーレンス処理を実行すれば 13 行目 CPU 側の処理に対してデータの一貫性は保証できる．

以上のような CPU と GPU のヘテロジニアス環境におけるユーザプログラム実行上の特徴を踏まえると，CPU 側もしくは GPU 側で書き込みが発生するたびに，もう一方へコヒーレンスを取りに行く強いメモリコンシステンシモデルよりも，CPU と GPU の処理の境界をコヒーレンスのタイミングとしてコヒーレンスを取りに行く緩和されたメモリコンシステンシモデルの方が最適である．したがって本研究では，緩和されたメモリコンシステンシモデルである Release Consistency を採用し，これを踏まえた上で効率の良いコヒーレンス処理を提案する．

3.2.2 本研究で提案するメモリコンシステンシモデル

Algorithm 2 が, 本研究における Release Consistency の提案内容となる．

Algorithm 2 提案するメモリコンシステンシモデル

```
1:  procedure MAIN
2:      CPU_ACQUIRE()                          ▷ CPU Acquire 手続き開始
3:      CPU の読み書き処理
4:      CPU_RELEASE()                          ▷ CPU Release 手続き開始

5:      HIPLAUNCHKERNELGGL(GPU_ACQUIRE)
6:                                             ▷ GPU Acquire Kernel 起動
7:      hipDeviceSynchronize()     ▷ GPU と後続処理の待ち合わせ同期

8:      HIPLAUNCHKERNELGGL(GPU_KERNEL)         ▷ GPU Kernel 起動
9:      hipDeviceSynchronize()     ▷ GPU と後続処理の待ち合わせ同期

10:     HIPLAUNCHKERNELGGL(GPU_RELEASE)
11:                                            ▷ GPU Release Kernel 起動
12:     hipDeviceSynchronize       ▷ GPU と後続処理の待ち合わせ同期

13:     CPU_ACQUIRE()                          ▷ CPU Acquire 手続き開始
14:     CPU の読み書き処理
15:     CPU_RELEASE()                          ▷ CPU Release 手続き開始
16: end procedure

17: procedure CPU_ACQUIRE()
18:     CPU 側の Acquire 処理
19: end procedure

20: procedure CPU_RELEASE()
21:     GPU に対してコヒーレンス処理を実施
22: end procedure

23: procedure GPU_ACQUIRE()
24:     GPU 側の Acquire 処理
25: end procedure

26: procedure GPU_KERNEL()
27:     GPU の計算処理を実施
28: end procedure

29: procedure GPU_RELEASE()
30:     CPU に対してコヒーレンス処理を実施
31: end procedure
```

提案する Algorithm 2 に関して，コヒーレンスのタイミング設計，コヒーレンスの処理設計，および提案の留意事項の 3 点について説明する．まず本提案におけるコヒーレンスのタイミング設計だが，前述したようにRelease Consistency を採用しており，ユーザプログラムにおいて CPU と GPU の処理の切り替えポイントをコヒーレンスのタイミングとしている．提案アルゴリズムの中では，それは 3 行目に CPU の読み書き処理が実行され 5 行目から GPU の処理が起動されることから，4 行目が CPU から GPU への処理の境界となるため，ここを CPU から GPU に対するコヒーレンスのタイミングとし，コヒーレンス処理の手続きを設定した．また 8 行目にて起動された GPU の読み書き処理は，9 行目にて後続処理との待ち合わせ同期が行われた後，13 行目から再び CPU 処理が開始されるため，10 行目から 12 行目が GPU から CPU への処理の境界となる．そこで，この間を GPU から CPU に対するコヒーレンスのタイミングとし，コヒーレンス処理の手続きを設定した．

本提案のコヒーレンス処理設計については，Release 手続きの実行時にコヒーレンスを取るように設計しているため，Release Consistency の中でも Eager Release Consistency を採用した．この理由として CPU と GPU の双方からアクセスされるデータは共有であるがゆえ，どちらからも共有と見なしている領域へのアクセスは全てコヒーレンスの対象となる．このため Eager Release Consistency もしくは Lazy Release Consistency のいずれを選んだとしてもコヒーレンスの実行タイミングが変わるだけで，コヒーレンスの対象とその処理数は変わらない．そこで本研究では実装面を考慮し，より実装が行いやすいモデルを選択した．

本提案の留意事項として，GPU 側の Acquire と Release の手続きの実装において，提案アルゴリズムの 5 行目と 10 行目に記載しているように，これらの手続きごとに個別の GPU カーネルを起動してそれぞれの手続きを実行させている．これは並列処理を行う GPU に対して Acquire や Release の手続きといった並列ではない処理を実施させるための手法となる．GPU はその内部に多くの物理的な計算ユニットを持ち，それらの計算ユニットに数千のスレッドを処理単位に分割して割り当てることで大規模な並列処理を行っているが，Acquire や Release などの手続きは並列処理が不要で，かつ，そのカーネル内の最初もしくは最後といった特定のタイミングで実行する必要がある．この条件を満たす処理を 1 つの GPU カーネル内で実行する場合，GPU 内の他の全てのスレッドを待ち合わせさせる必要があるが，1 つの GPU カーネル内で制約なくこれを実現する手法は現時点で GPU では提供されていない．この点について例を挙げて補足すると，NVIDIA の CUDA Programming Guide Release 13.1 (2025 年 12 月 4 日リリース) [10] では，従来の GPU カーネルの同期は __syncthreads() によるスレッドブロック内の全スレッドの同期処理というシンプルな機能の提供に限られていたが，Cooperative Groups という機能を新たに提供することで grid 全体の同期を行えるようになったと記

載している．しかしながら，この機能を使用する場合，起動する GPU カーネルのブロック数を調整する必要があり，本来 GPU のリソースを最大に使用して並列計算を行わせることが GPU の目的であるにもかかわらず，本機能の使用によりそうした目的に制約が発生してしまう．こうした点から，現時点では 1 つの GPU カーネル内において制約なしで GPU の全ブロックの全スレッドを同期させる手法は提供されていないと言える．そこでこれらの手続きを実行するにはカーネルを分割して実行させる方法が現実的な解決法となった．

3.3　本研究で提案するコヒーレンス処理について

3.2 節では提案するメモリコンシステンシモデルとして Release Consistency を採用することについて述べたが，本節では効率の良いコヒーレンス処理についての提案内容を述べる．

3.3.1　本研究で提案するコヒーレンス処理について

本研究の提案対象となる CPU と GPU からなるヘテロジニアス環境は図 3.4 の環境を前提する．

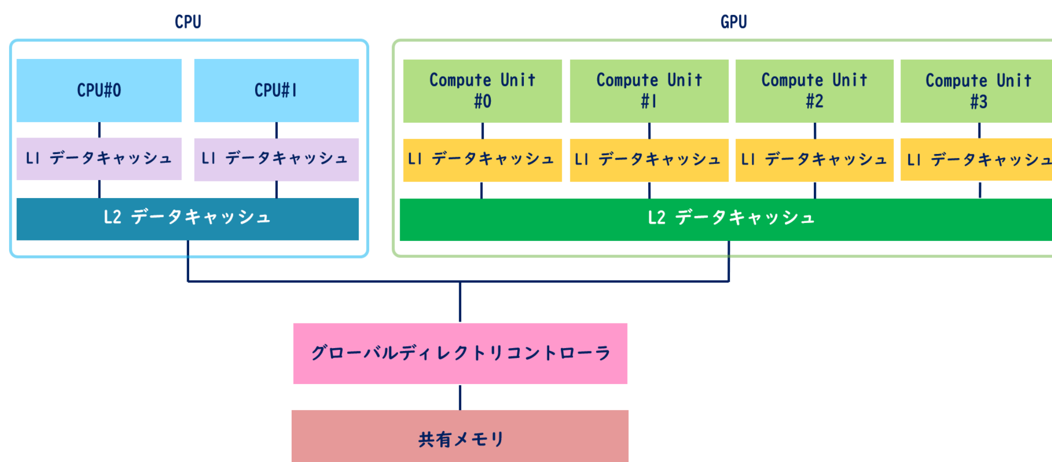

図 3.4: CPU-GPU ヘテロジニアス提案環境

図 3.4 の環境では，CPU は L1 と L2 のキャッシュを持ち，CPU のキャッシュコントローラでそれらのキャッシュの制御が行われる．GPU は計算ユニットである Computing Unit(CU) に対して TCP と呼ばれる L1 キャッシュと TCC と呼ばれる全ての CU で共有される L2 キャッシュからなり，GPU のキャッシュコントローラがこれらのキャッシュを制御す

る．この環境において全てのプロセッサは，グローバルディレクトリコントローラを介してメインメモリを共有する．このためヘテロジニアス環境でのコヒーレンス処理は，グローバルディレクトリコントローラによる機能を中心に提供される．

次にこの環境における効率の良いコヒーレンス処理を考えるにあたり，ヘテロジニアス環境で使用されるユーザプログラムを再び分析する．Algorithm 3[11] は，AMD 社がヘテロジニアス環境用のシミュレータにサンプル提供しているユーザプログラムの抜粋である．

Algorithm 3 AMD 社により提供されたサンプルコードの抜粋

```
1:  Constant: WIDTH ← 32
2:  Constant: NUM ← WIDTH × WIDTH
3:
4:  procedure MAIN
5:      float gpuTransposeMatrix[2]
6:      width ← WIDTH
7:      hipHostMalloc(gpuTransposeMatrix[0], NUM × sizeof(float))
8:      hipHostMalloc(gpuTransposeMatrix[1], NUM × sizeof(float))
            ▷ CPU と GPU の共有メモリ確保
9:      MultipleStream(data, randArray, gpuTransposeMatrix, width)
            ▷ GPU Kernel 起動用の手続き
10:     hipDeviceSynchronize()
            ▷ GPU と後続処理の待ち合わせ同期
11:
12: end procedure
```

Algorithm 3 はサンプルではあるが, CPU と GPU のヘテロジニアス環境で使用するユーザプログラムの共有メモリの扱いについて基本的な特徴を示している. それは, こうした環境で用意される共有データは 7-8 行目で見られるような配列構造を取ることである. 具体的には, Algorithm 3 では 7-8 行目で gpuTransposeMatrix[0] と gpuTransposeMatrix[1] という名前の各配列に対して, 1-2 行目で定義したパラメータを用いて hipHostMalloc 関数によりメモリ領域を確保している．ユーザプログラム上ではこの領域を行列を構成するデータとして扱って計算しているが，こうして取得されるメモリは連続したアドレスで取得される性質を持つと言える．この点に注目すると，CPU と GPU で共有されるメモリはその構造上連続してアクセスされる可能性が高く，連続領域に対する読み込みや書き込みによるコヒーレンス処理は，都度逐次的に実行するよりも連続した領域を一括して実行した方が効率が良い．

そこで本研究では，この特徴を有効に活用すべく CPU と GPU で共有するメモリへのヒーレンス処理に対して，その対象のアドレスが連続

している場合は，コヒーレンス処理をアドレスの範囲で指定して実行する方式を提案する．この場合，コヒーレンス処理を要求するのはグローバルディレクトリコントローラとなるが，グローバルディレクトリコントローラはコヒーレンス処理の対象アドレスが連続している場合は，その範囲を指定したコヒーレンス処理を Release Consistency に基づき設定した同期ポイントにおいて，CPU もしくは GPU のキャッシュコントローラに対して実行する．この要求を受けた各プロセッサのキャッシュコントローラは，連続したアドレスの範囲分，該当するアドレスがキャッシュに存在するかキャッシュタグの検索を行う．もし該当するアドレスがキャッシュタグに存在している場合は，そのキャッシュラインを無効化する．存在しない場合は，次のアドレスを用いてキャッシュタグの検索を行う．この処理を指定されたアドレス範囲に対して最後まで続ける．本研究では，この処理をハードウェア機構の実装方式で提案する．

3.3.2 本研究で提案するハードウェア実装について

本節では，前節で述べた提案内容に従うハードウェア実装方式を説明する．この説明にあたり，ユーザプログラムを開始から終了までいくつかの段階に分け，それぞれの段階において本提案をどのように実装しているかを Algorithm 2 の内容に沿って説明していく．図 3.5 は，Algorithm 2 の CPU_ACQUIRE() の処理に該当し，ユーザプログラムが開始され CPU 側で Acquire の手続きを実行する処理までを表したものである．

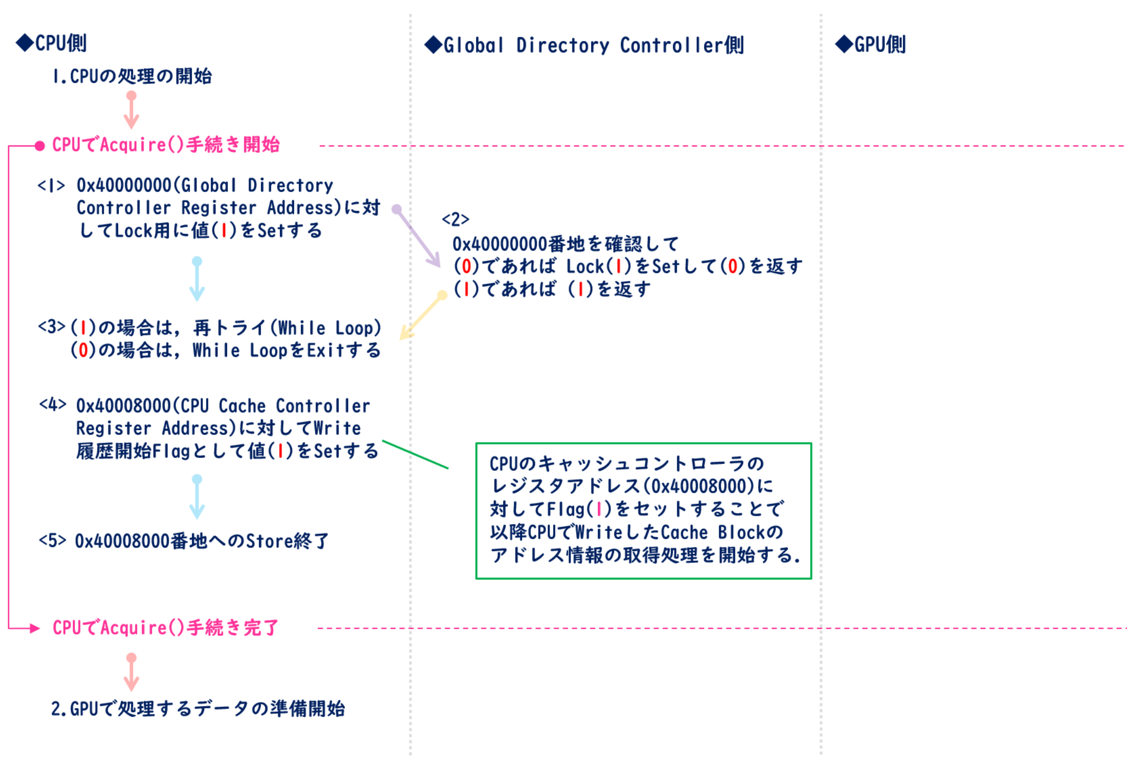

図 3.5: CPU_ACQUIRE の提案実装

CPU 側で Algorithm 2 の CPU_ACQUIRE() に該当する Acquire の手続きに入ると，ロックを取得するためにグローバルディレクトリコントローラに対して 0x40000000 のレジスタアドレスにロック用のフラグをセットする書き込みを行う (図 3.5 <1>)．この要求を受けたグローバルディレクトリコントローラは，ロックされていなければフラグをセットして CPU 側へ応答を返す．ロック獲得処理が終われば，CPU は次に CPU のキャッシュコントローラに対して 0x40008000 のレジスタアドレスにフラグをセットする書き込みを行う (図 3.5 <4>)．この要求を受けた CPU のキャッシュコントローラはフラグをセットし，以降 CPU で書き込みを行ったキャッシュラインに対しては，そのアドレス情報をキャッシュコントローラ内のメモリに保管する．CPU 側の Acquire の手続きはこの処理で完了し，CPU 側で GPU 用のデータを用意する読み書き処理が始まる．

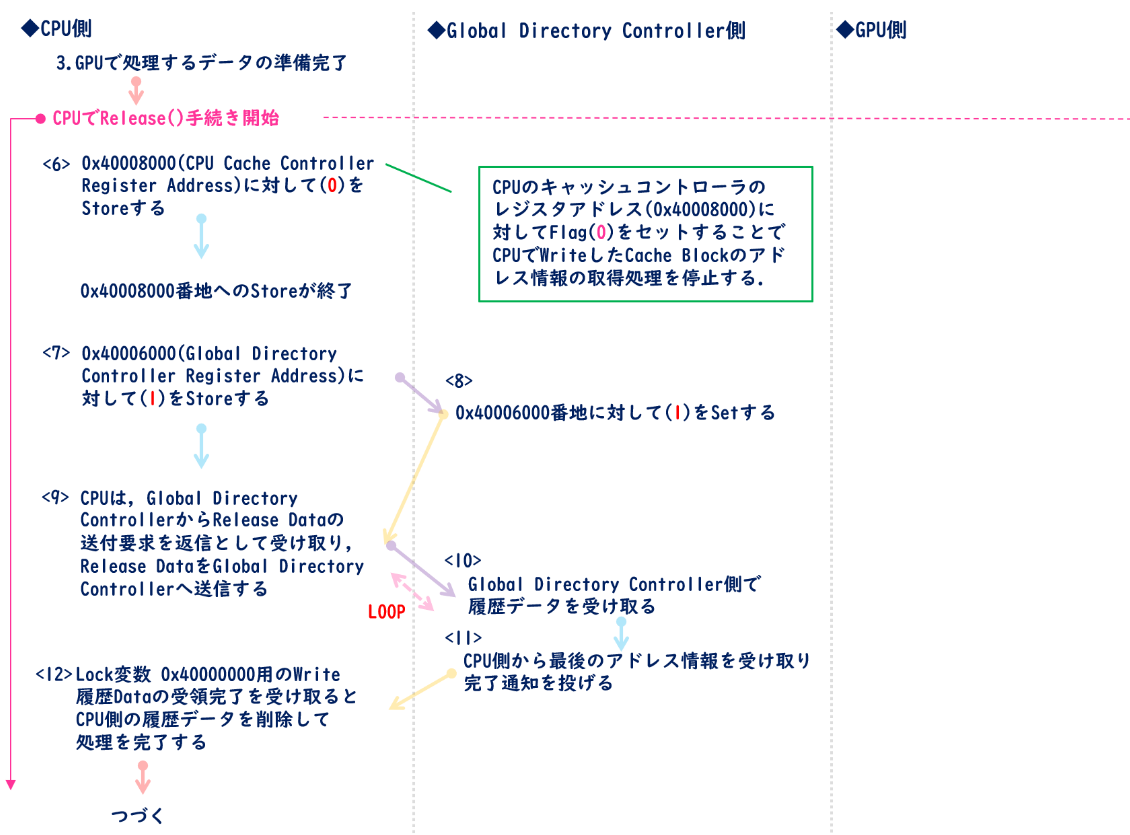

図 3.6: CPU_RELEASE の提案実装 (前半部分)

CPU 側で GPU 用のデータの用意が完了すると，CPU 側で Algorithm 2 の CPU_RELEASE() に該当する Release の手続きを実行する．この Release 処理は大きく 2 つの処理を行っており，図 3.6 では前半の処理を説明する．まず CPU 側の Acquire 処理で CPU のキャッシュコントローラにある 0x40008000 のレジスタアドレスに対して設定したフラグを解除する書き込みを行う (図 3.6 <6>)．この処理により CPU 側で Acquire 手続き以降 CPU で書き込みのあったキャッシュラインに対する履歴情報の取得処理を停止させる．次に，グローバルディレクトリコントローラに対して 0x40006000 のレジスタアドレスに書き込みを実施する (図 3.6 <7>)．このレジスタアドレスに対して 1 の値が書き込まれると，グローバルディレクトリコントローラ側で CPU で書き込みのあったキャッシュラインのアドレス履歴情報の受領処理が起動し，CPU のキャッシュコントローラに対してデータ送信を開始させるための Ack を返す (図 3.6 <8>)．グローバルディレクトリコントローラからの Ack を受信すると，CPU キャッシュコントローラは自身の内部で保管した書き込みのあったキャッシュラインのアドレス情報をグローバルディレクトリコントローラに対し

て送付する (図 3.6 <9>). 送付するアドレス情報が多い場合，複数回の送付処理が発生するが，全てのアドレス情報をグローバルディレクトリコントローラが受け取った時点で，グローバルディレクトリコントローラから CPU のキャッシュコントローラに対して受領完了通知が送られる (図 3.6 <11>). この通知を受け取ると，CPU のキャッシュコントローラは送付済みのアドレス情報を全て削除する (図 3.6 <12>).

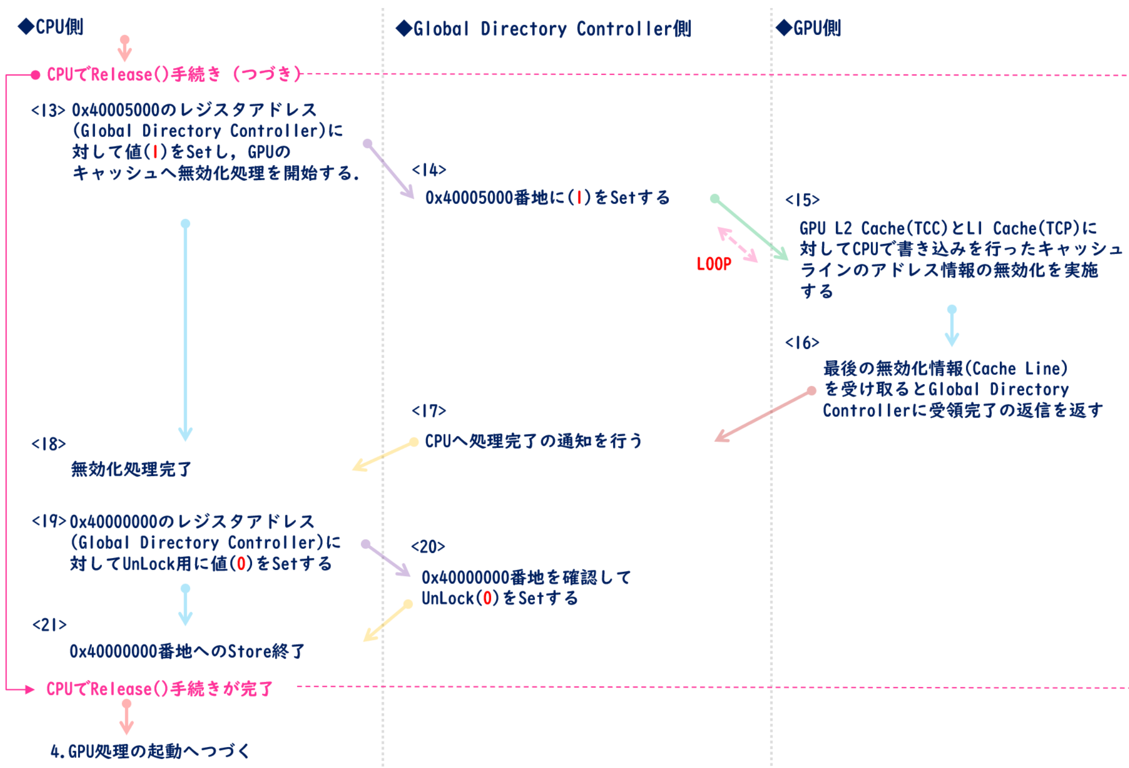

図 3.7: CPU_RELEASE の提案実装 (後半部分)

図 3.7 に記載した CPU 側の Release 処理の後半の処理として，グローバルディレクトリコントローラのレジスタアドレス 0x40005000 に対して 1 の値の書き込みを行う. この処理によりグローバルディレクトリコントローラ側で GPU に対するハードウェア化されたコヒーレンス処理が起動する (図 3.7 <14>). グローバルディレクトリコントローラは，前半の処理で受領した CPU 側で書き込みのあったアドレス情報をもとに GPU 側に対してハードウェア化されたキャッシュラインの無効化処理を開始する. なおこの処理において無効化するキャッシュラインのアドレス情報が連続している場合，キャッシュラインのアドレス情報を範囲で指定して無効化処理の依頼を要求する (図 3.7 <15>). 無効化する処理が複数回に及ぶ場合，無効化処理を繰り返し，GPU 側で最後の無効化処理が完

了すると，グローバルディレクトリコントローラに対して無効化処理完了の通知を行う (図 3.7 <16>)．この通知を受けたグローバルディレクトリコントローラは，CPU 側に無効化処理完了の通知を行う (図 3.7 <17>)．CPU 側ではこの通知の受領を受けて無効化処理を完了し Acqure の手続きで実施したロック変数の解放を行う (図 3.7 <19>)．グローバルディレクトリコントローラのレジスタアドレス 0x40000000 に対して 0 の値の書き込みが完了すると，CPU の Release 処理が完了する (図 3.7 <21>)．

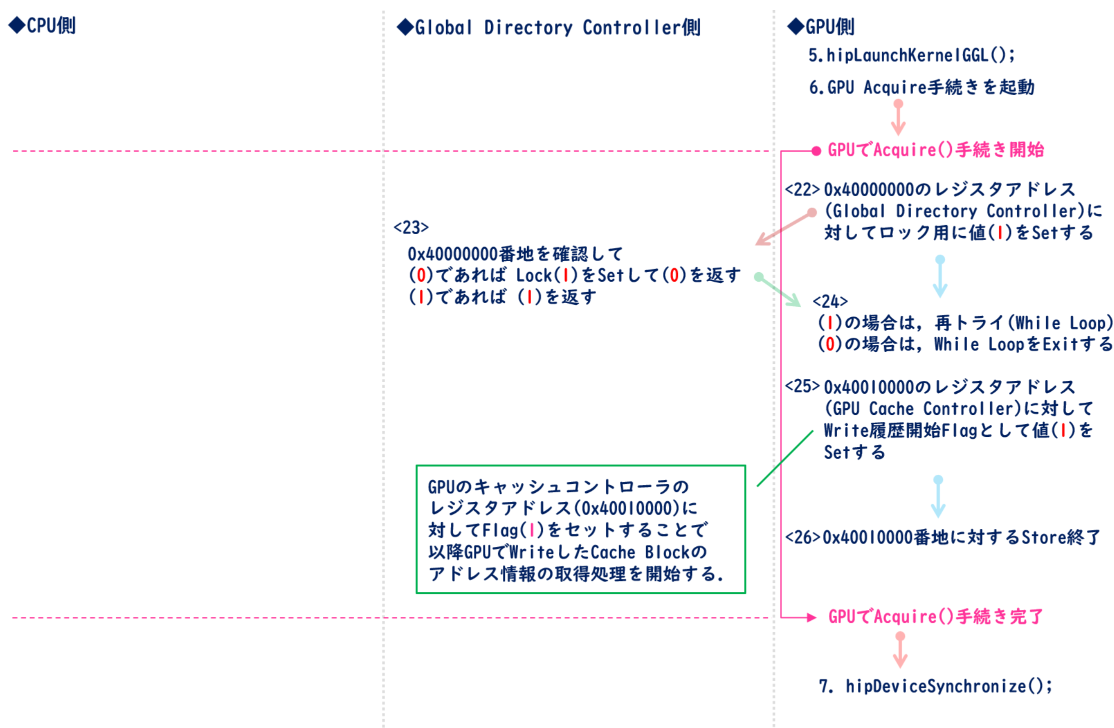

図 3.8: GPU_ACQUIRE の提案実装

CPU 側で GPU 処理用データの用意と GPU キャッシュの無効化処理を終えると，Algorithm 2 の GPU_ACQUIRE() に該当する処理に移行する (図 3.8 内の 5 および 6)．GPU Acquire の手続きは，CPU 側と同様にまずグローバルディレクトリコントローラのレジスタアドレス 0x40000000 に対してロックを獲得するために 1 の値を書き込む (図 3.8 <22>)．ロックが解放されていれば 1 の値を書き込みロックを獲得する (図 3.8 <23>)．ロックを獲得後，GPU のキャッシュコントローラのレジスタアドレス 0x40010000 に対して 1 の値を書き込む (図 3.8 <25>)．この値の書き込みにより，GPU 側で以降の書き込み処理されたキャッシュラインのアドレス情報の履歴が取得されるようになる．この処理の完了後，GPU の Acquire 手続きは完了する．GPU の Acquire 手続き用の GPU カーネル

処理が終わると最後に GPU と後続処理の待ち合わせ同期を行うために hipDeviceSynchronize 関数を実行する (図 3.8 内の 7).

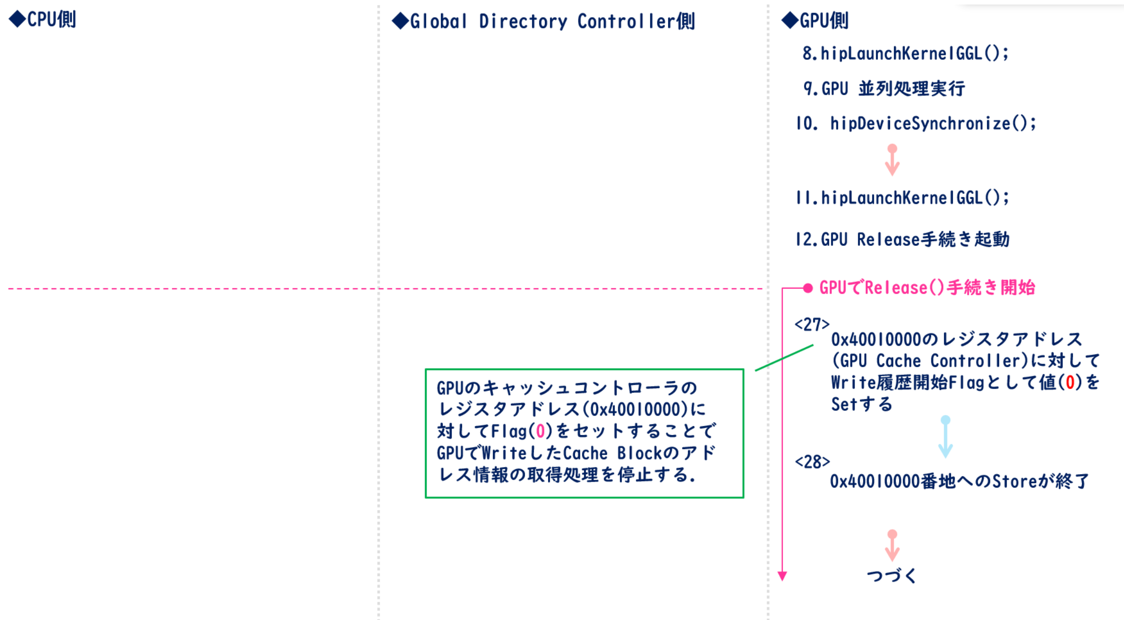

図 3.9: GPU_RELEASE の提案実装 (前半部分)

GPU の Acquire 手続きが完了後，Algorithm 2 の GPU_KERNEL() に該当する処理に移る．図 3.9 の 8 と 9 に示すように CPU から GPU 計算用の GPU カーネルを hipLaunchKernelGGL 関数を用いて起動させ, GPU 側で並列計算処理を行わせる．GPU の計算が終了後，hipDeviceSynchronize 関数を用いて GPU と後続処理の待ち合わせ同期をとる (図 3.9 内の 10).

GPU での計算処理が終了後，Algorithm 2 の GPU_RELEASE() に該当する処理に移る．この処理では，CPU 側から hipLaunchKernelGGL 関数を使用して GPU の Release 手続きを起動する (図 3.9 内の 11)．GPU の Release 手続きも CPU の Release 手続きと同様に 2 つの処理に分かれる．前半処理では，GPU のキャッシュコントローラのレジスタアドレス 0x40010000 に対して 0 の値を書き込む．この書き込みにより GPU 側でキャッシュラインに書き込みが発生するたび取得していたアドレス情報の取得を停止する (図 3.9 <27>).

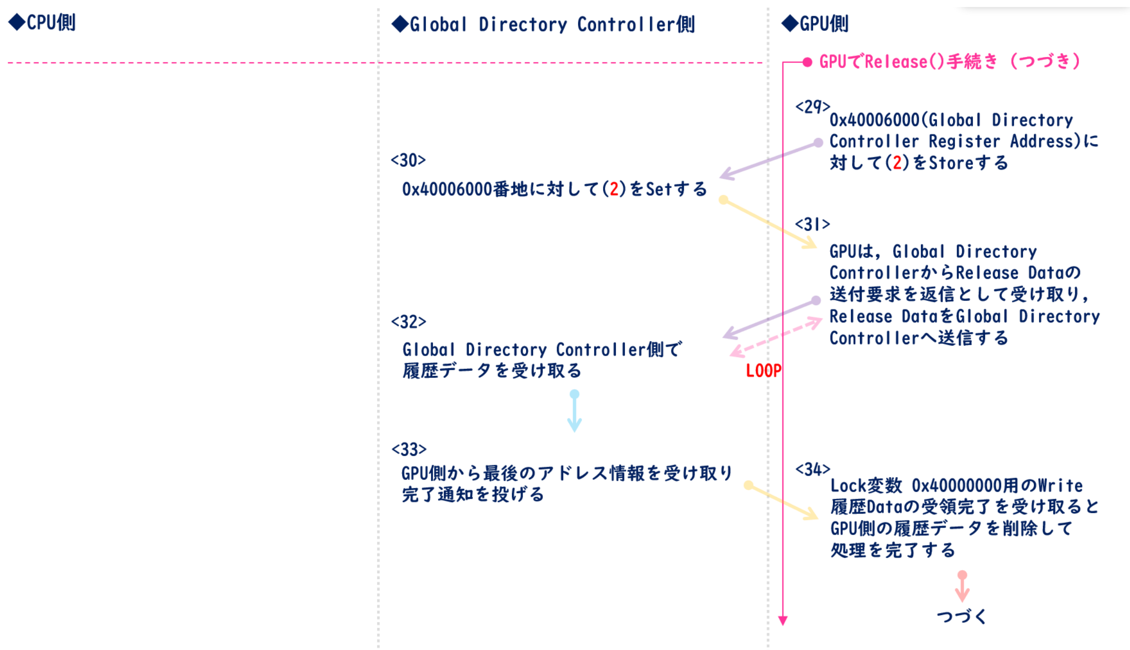

図 3.10: GPU_RELEASE の提案実装 (前半処理のつづき)

図 3.10 で GPU_RELEASE() 内の前半処理の続きに移るが，GPU はグローバルディレクトリコントローラのレジスタアドレス 0x40006000 に対して 2 の値を書き込む．この書き込みを受けたグローバルディレクトリコントローラでは，GPU から送付される書き込みのあったキャッシュラインのアドレス情報を受け取る処理を起動し，GPU 側にアドレス情報の送信を開始するように Ack を返す (図 3.10 <31>)．Ack を受けとった GPU のキャッシュコントローラは，グローバルディレクトリコントローラに対して書き込みのあったキャッシュラインのアドレス情報を送付する．送付するアドレス情報が多い場合は，この処理を複数回繰り返す (図 3.10 <32>)．GPU 側で最後のアドレス情報の受領返信を受け取ると，GPU のキャッシュコントローラで履歴情報を削除して処理を完了させる (図 3.10 <34>)．

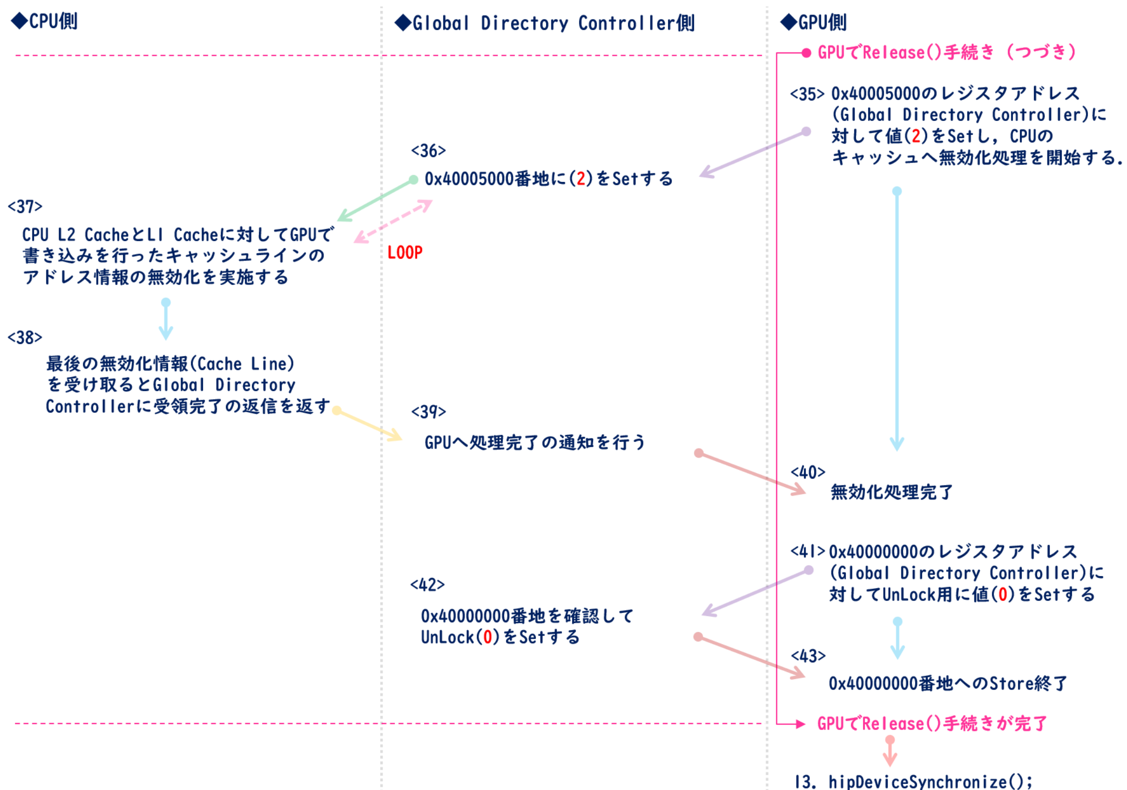

図 3.11: GPU_RELEASE の提案実装 (後半処理)

GPU Release 手続きの後半処理では，グローバルディレクトリコントローラのレジスタアドレス 0x40005000 に 2 の値を書き込む (図 3.11 <36>)．この書き込みを受けると前半の処理によりグローバルディレクトリコントローラ側で保管している GPU で書き込みを行ったキャッシュラインのアドレス情報を使い，CPU のキャッシュコントローラに対してハードウェア化された無効化処理を開始する．CPU の Release 処理時と同様に無効化処理の対象アドレスが連続している場合は，アドレスを範囲で指定して CPU のキャッシュコントローラに対して無効化を要求する．この処理を全ての書き込みアドレス情報に対して繰り返し実施する (図 3.11 <37>)．CPU 側で最後の無効化処理が完了すると，CPU のキャッシュコントローラからグローバルディレクトリコントローラに対して完了通知を行い (図 3.11 <39>)，さらにグローバルディレクトリコントローラから GPU へ無効化処理完了通知を行うことで，GPU 側の無効化要求処理が終了する (図 3.11 <40>)．

CPU 側のキャッシュの無効化処理が終わると，GPU のキャッシュコントローラからグローバルディレクトリコントローラのレジスタアドレス 0x40000000 に対して 0 の値を書き込む (図 3.11 <41>)．この書き込

みを受けてグローバルディレクトリコントローラ側でレジスタアドレス 0x40000000 に対するロックを解放する．ロック解放後，GPU の Release 手続きは完了し，最後に GPU と後続処理の待ち合わせ同期を行うために hipDeviceSynchronize 関数を実行する (図 3.11 内の 13)．

Algorithm 2 の GPU_RELEASE() に該当する処理の後，ユーザプログラムではもう一度 CPU 処理に遷移するが，ここで実行される Acquire と Release の手続きはユーザプログラムの開始時に CPU 側で実行した処理と同様の処理を実行する．本研究ではこのような Release Consistency に従い一連のコヒーレンス処理を実行するハードウェア機構を提案する．

第 4 章

実験・評価

4.1　実験環境

本研究で提案する機構を実装したハードウェアは現時点で存在していないため，本提案ではシミュレータを用いて提案内容の評価を行った．そこで本章ではまずシミュレータの選定について記載し，次に選定したシミュレータの特徴を説明した上で，評価結果を述べる．

4.1.1　シミュレータ選定

シミュレータの選定にあたりシミュレータのサーベイ論文 [12] を参照し，ヘテロジニアス環境のシミュレータとして紹介された論文 [13] で扱われていたシミュレータを実験環境の候補とした．それらは gem5-gpu, gem5，Multi2Sim の 3 つのヘテロジニアス環境のシミュレータであったが，gem5-gpu は 2017 年からシミュレータとしてのプロジェクトが停止しており，メインテナンスが行われていないため現在シミュレータとしてのサポートは行われていない．また Multi2Sim は分散型メモリ環境をサポートするヘテロジニアス環境シミュレータであったため本研究の要件と合わず，結果として gem5 を選定した．

4.1.2　gem5 シミュレータについて

本提案の実験・評価では gem5 シミュレータを用いるが，gem5 を用いて評価を行うにあたり gem5 の特徴を踏まえた補足事項を 4 点述べる．1 つ目はカスタマイズについて，CPU と AMD GPU からなる gem5 シミュレータのヘテロジニアス構成は，gem5 に標準実装されており利用可能ではあるものの提案手法で述べたハードウェア機構は標準構成では存在しない．そこで本提案では評価にあたり，これらの機構の追加実装を行った．

2つ目はgem5のヘテロジニアス環境に対するサポートについて，現在提供されている実装でヘテロジニアス環境のシミュレーションは行えるものの，GPUの機能は完全にシミュレータ化されてはいない．このため例えばユーザプログラム上でGPUの処理を記述しても実際に動作しない処理も存在する．この点について表4.1と表4.2でハードウェア機能とソフトウェア機能の実装状況を示す[14].

表4.1: HSA Hardware Building Blocks

Category	**Function Details**
Supported	• Shared virtual memory – Single address space – Coherent – Fast Access from all components – Can share pointers • Platform atomics • Defined memory model – Acquire and Release semantics as implemented by the compiler
Work-in-progress	• Architected user-level queues – Via architected queuing language (AQL) • Signal • Defined memory model – Merging functional and timing models
Long Term Work	• Shared virtual memory – Pageable • Context switching

表 4.2: HSA Software Building Blocks

Category	**Function Details**
Supported	• Radeon Open Compute platform (ROCm) – Allocate memory • Heterogeneous Compute Compiler (HCC) – CLANG
Work-in-progress	• Radeon Open Compute platform (ROCm) – Create queues – Device discovery – AQL support • Machine ISA – GCN3
Long Term Work	• Radeon Open Compute platform (ROCm) – AMD's implementation of HSA principles • Heterogeneous Compute Compiler (HCC) – LLVM - direct to GCN3 ISA – C++, C++ AMP, HIP, OpenMP, OpenACC, Python

3 つ目は gem5 の動作仕様について，gem5 のシミュレーションは，イベントとして定義されるアクションを離散的に実行していくことで実現される．その際，各イベントにはレイテンシと呼ばれる事前定義された Tick 数が付与されており，各イベントの実行とともにこのレイテンシが逐次加算されシミュレーション実行時間が算出される．図 4.1 はこの仕組みを説明した資料 [15] である．

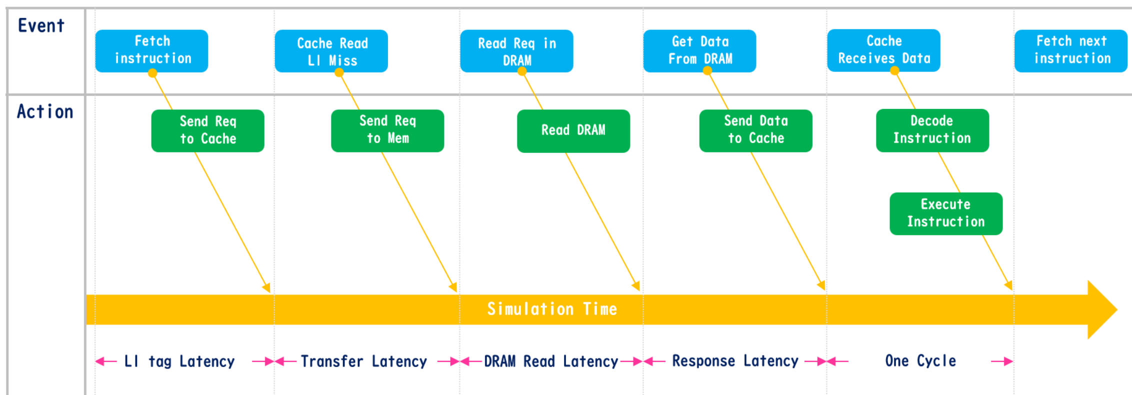

図 4.1: gem5 Discrete event simulation 例

図 4.1 ではインストラクションのフェッチからスタートし，L1 キャッシュの Read ミス発生により，DRAM からデータをキャッシュに読み込むまでの各処理について，対応するイベントとレイテンシの関係を表している．このような離散的なイベントが多重並列的に実行され，それぞれのレイテンシの総和でシミュレーション実行時間が決定される．

4 つ目はシミュレーションの実行時間の単位 (Tick) について，gem5 ではグローバルな Tick(Defaut 1THz) がコンフィグで定義されており，CPU やメモリなど動作速度の異なるコンポーネントの実行時間は，各コンポーネントで事前定義されたレイテンシの値をグローバルな Tick へ換算して算出される．図 4.2 は gem5 のコンフィグ [11] から抜粋した Default の Tick 値と各コンポーネントのレイテンシのデータ例となる．

```
(a)
 apu_se.py
   m5.ticks.setGlobalFrequency(”1THz”)

(b)
 config_ini
  1:[system.ruby.clk_domain]
      type=SrcClockDomain
      clock=500

  2:[system.ruby.cp_cntrl0]
      type=CorePair_Controller
      clk_domain=system.ruby.clk_domain

  3:[system.ruby.tcc_cntrl0]
      type=SrcClockDomain
      clock=1000

  4:[system.ruby.tcp_cntrl0.clk_domain]
      type=SrcClockDomain
      clock=1000
```

```
(c)
 config_ini
  1:[system.ruby.cp_cntrl0.L1D0cache]
    [system.ruby.cp_cntrl0.L1D1cache]
      tagAccessLatency=1

  2:[system.ruby.cp_cntrl0.L2cache]
      tagAccessLatency=1

  3:[system.ruby.tcp_cntrl0.L1cache]
    [system.ruby.tcp_cntrl1.L1cache]
    [system.ruby.tcp_cntrl2.L1cache]
    [system.ruby.tcp_cntrl3.L1cache]
      tagAccessLatency=4

  4:[system.ruby.tcc_cntrl0.L2cache]
      tagAccessLatency=2
```

図 4.2: gem5 Default Tick Config 例

gem5 の Default のグローバル Tick は図 4.2 (a) に 1THz と定義されており，各コンポーネントのシミュレーション上の Tick は，各コンポーネントに事前付与された Tick に対して，この値を基準に換算された値が用いられる．各コンポーネントに付与された Tick については，図 4.2 (b) の config_ini ファイルに定義されており，CPU のキャッシュコントローラ (cp_cntrl0) の場合 (図 4.2 (b) 内の 2), clk_domain=sytem.ruby.clk_domain と定義されているため clock=500(図 4.2 (b) 内の 1) となる．これは 1THz のグローバル Tick 500Tick 分が 1 周期になることを定義しており，2GHz のクロックと換算される．一方で，GPU の tcc_cntrl0 と tcp_cntrl0 の場合 (図 4.2 (b) 内の 3 および 4), clock=1000 とあるため, 1000Tick 分が 1 周期になることから 1GHz のクロックと換算される. こうした各コンポーネントの Tick の定義を踏まえて, 実際に CPU のキャッシュコントローラが L1 キャッシュのタグにアクセスするのに必要な Tick 数は，図 4.2 (c) の config_ini の 1 に tagAccessLatency=1 と定義されているので，cp_cntrl0 の 1 周期分のクロックが必要となり，500Ticks として算出される．同様に以下それぞれのキャッシュコントローラの tag Access Latency(図 4.2 (c) config_ini 2,3,4) を見ると，CPU L2 の場合 500Ticks，GPU の L1 の場合 4000Ticks，GPU の L2 の場合 2000Ticks となる．

gem5 は以上のようなイベントのレイテンシの定義をもとに，それらのレイテンシをシミュレーションの基準となる Tick に変換した値を用い

てシミュレーション実行時間を算出しているが，本研究の評価にあたって 2 点の留意点がある．1 点目は，本研究の提案手法であるキャッシュラインのアドレスを範囲指定して無効化させる処理において，Default の gem5 の定義では，どれだけ重い負荷の処理を実行しても同じ Tick でしか実行時間が見積もられない点である．これにより広大なアドレス範囲を指定して無効化した処理と 1 つのキャッシュラインのアドレスを無効化した処理が同じ実行時間としてシミュレーション上計算されてしまうため評価が正しく行われない．そこでキャッシュラインのアドレスを範囲指定して無効化した処理を実行した場合，各キャッシュコントローラ内でタグアクセスした回数分の Tick を評価結果に加算することで実験結果を正しく補正した．2 点目は，gem5 のシミュレーションは正常にシミュレーションが終了すると全てのイベントの Tick の総和をシミュレーション実行時間として算出する．これによりシミュレーション結果が本研究で提案した無効化処理の Tick とそれ以外の全ての処理の Tick の合計値となってしまうため提案手法に対する効果の判定が難しい．そこで curTick 関数を用いて無効化処理 (probe) に関わる実行時間だけを算出できるように実装を加えた．これによりシミュレーションにおいて計測したい無効化処理の開始と終了時の Tick を取得し，その差を取ることで無効化処理に関わる実行時間の抽出を行った．なおこの手法は，gem5 の HP にある Creating a simple cache object[11] にて紹介された手法を参考にした．

4.1.3 実験結果

前節で説明した gem5 シミュレータの特徴を踏まえて 3 つのユーザプログラムを評価した結果を説明する．1 つ目のユーザプログラムは，square.cpp と呼ばれるプログラムで CPU 側で配列を用意し，各要素に初期値を設定した後，GPU 側で各要素の二乗計算を行い，最後に CPU 側で GPU の計算結果のチェックを行う．このプログラムで使用する配列の要素数を，200，2000，20000，40000, 100000，200000，300000 と変更し，各要素数の場合の計算に要するコヒーレンス処理数および実行時間について，Default 設定と提案手法で比較評価した．結果は図 4.3 と図 4.4 となる．要素数が小さい場合，その差が読み取りづらいため表の結果も表 4.3 と表 4.4 として併記する．

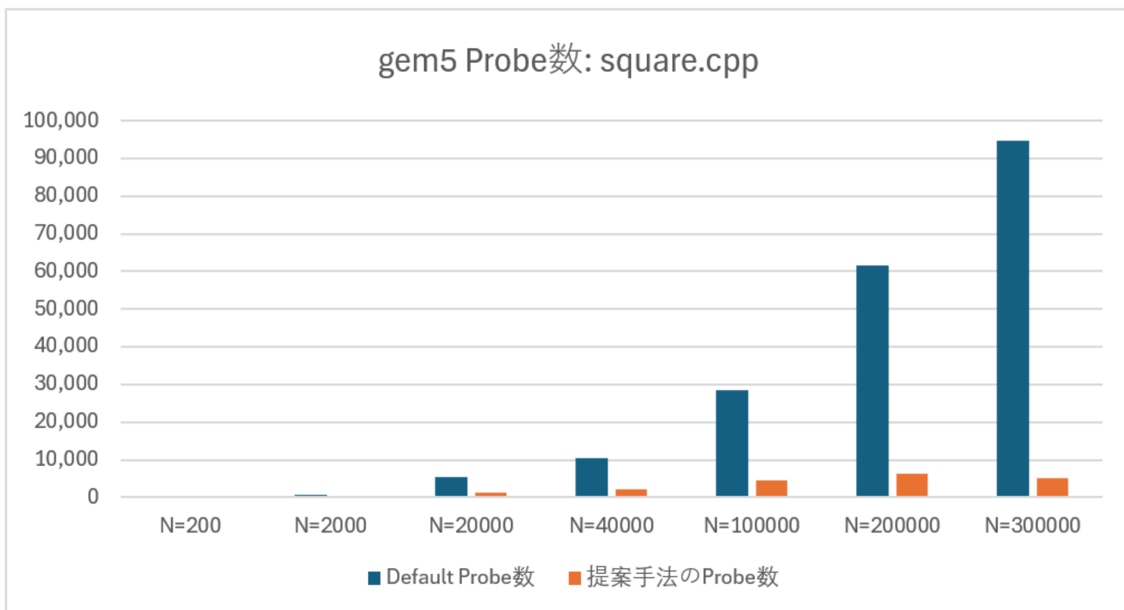

図 4.3: gem5 Probe 数 square.cpp

配列要素数	Default Probe数	提案手法のProbe数
N=200	67	21
N=2000	581	149
N=20000	5,301	1,153
N=40000	10,572	2,189
N=100000	28,627	4,384
N=200000	61,612	6,242
N=300000	94,774	5,111

表 4.3: gem5 Probe 数 square.cpp

図 4.3 と表 4.3 にて，配列として用意した共有データに対してグローバルディレクトリコントローラから各キャッシュコントローラへキャッシュラインを無効化する要求 (Probe) を送信した数を表す．N=200 のとき約 70%，N=300000 のとき約 95% の無効化要求数の低減となった．なお N=200000 のときと比べて N=300000 のときの提案手法の Probe 数が減少しているが，提案手法では無効化するアドレスが連続する場合，それらを 1 回の無効化要求で処理する仕様となっている．この点を踏まえると，N=300000 のときの共有メモリの確保の状況が N=200000 のときよりもより連続的な範囲となっている事が原因となっており，これは gem5 もしくは gem5 のヘテロジニアス環境の仕様に依存している．

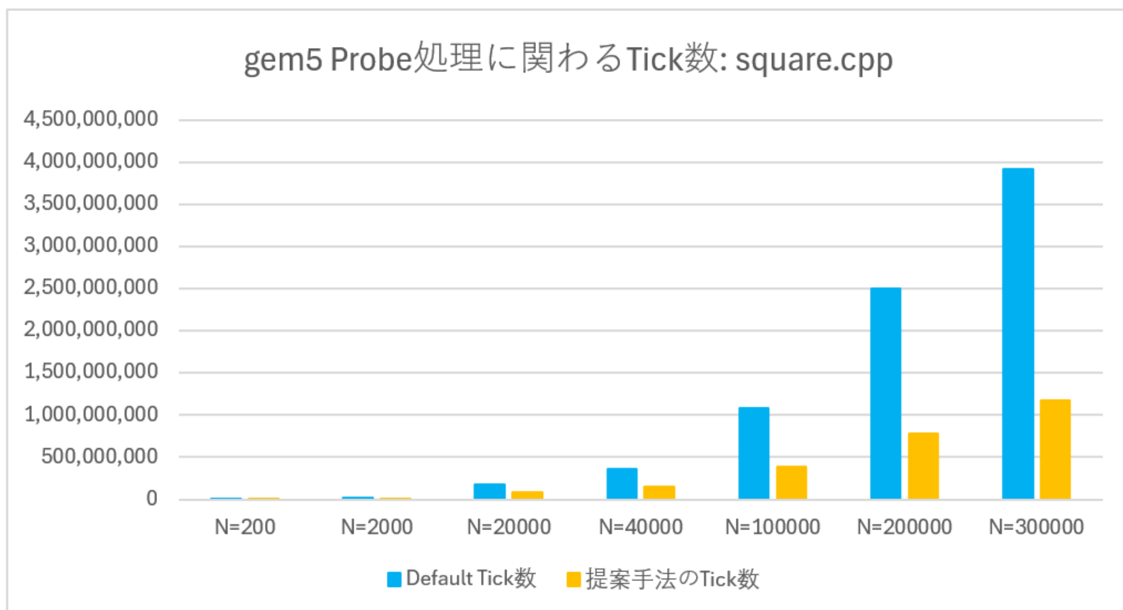

図 4.4: gem5 Probe 処理に関わる Tick 数 square.cpp

配列要素数	Default Tick数	提案手法のTick数
N=200	2,140,500	1,119,500
N=2000	19,285,500	8,843,500
N=20000	182,420,500	77,751,000
N=40000	364,226,000	154,594,000
N=100000	1,083,574,500	391,520,000
N=200000	2,497,267,000	780,212,500
N=300000	3,914,588,000	1,169,410,000

表 4.4: gem5 Probe 処理に関わる Tick 数 square.cpp

図 4.4 と表 4.4 にて，配列として用意した共有データに対してグローバルディレクトリコントローラから各キャッシュコントローラへ要求したキャッシュラインの無効化処理の実行時間 (Tick) を表す．N=200 のとき約 48%，N=300000 のとき約 70% の実行時間の低減となった．

2 つ目の実験は，2dshfl.cpp と呼ばれるユーザプログラムの実行結果となる．このプログラムでは行列の転置を実行するが，まず CPU 側で行列の初期値を用意し，次に GPU 側でその行列の転置を行う．最後に CPU 側で転置された値の確認を行うという処理の流れになる．このユーザプログラムで使用する行列のサイズを 4×4，8×8，16×16，20×20，30×30，40×40，50×50 と変更し，各行列サイズの場合の計算に要するコヒーレ

ンス処理数および実行時間について，Default 設定と提案手法で比較評価した．結果は図 4.5 と図 4.6，および表 4.5 と表 4.6 となる．

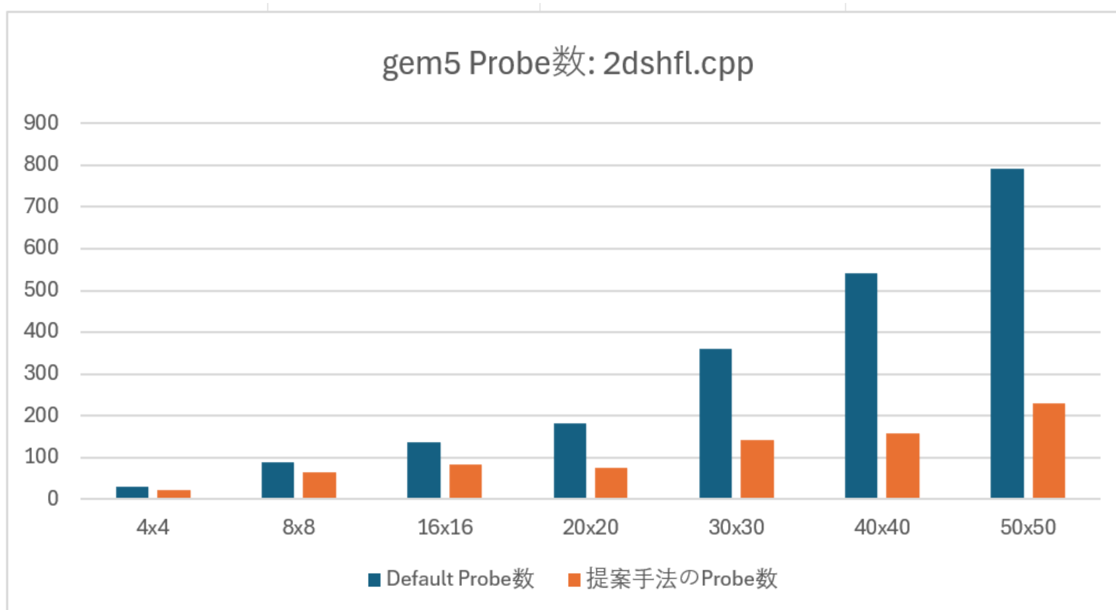

図 4.5: gem5 Probe 数 2dshfl.cpp

行列サイズ	Default Probe数	提案手法のProbe数
4x4	29	21
8x8	90	66
16x16	137	83
20x20	183	76
30x30	359	142
40x40	540	158
50x50	791	229

表 4.5: gem5 Probe 数 2dshfl.cpp

図 4.5 および表 4.5 は，行列として用意した共有データに対してグローバルディレクトリコントローラから各キャッシュコントローラへキャッシュラインを無効化する要求 (Probe) を送信した数を表す．4×4 のとき約 30% ，50×50 のとき約 70% の無効化要求数の低減となった．

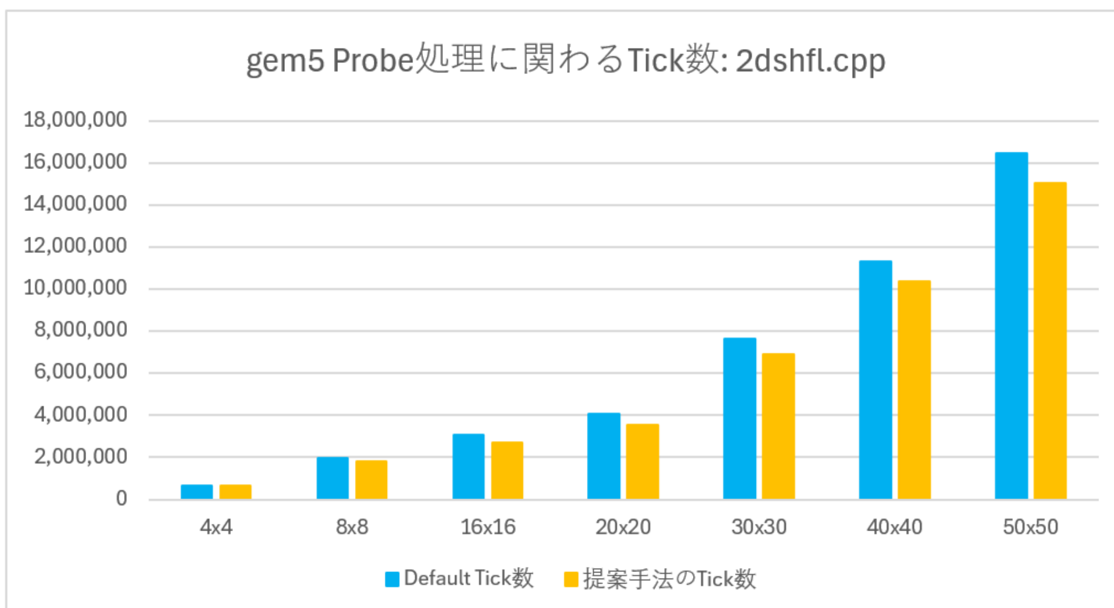

図 4.6: gem5 Probe 処理に関わる Tick 数 2dshfl.cpp

行列サイズ	Default Tick数	提案手法のTick数
4x4	653,500	633,500
8x8	1,963,000	1,829,000
16x16	3,044,500	2,683,000
20x20	4,046,500	3,534,500
30x30	7,654,500	6,885,500
40x40	11,306,000	10,351,000
50x50	16,451,500	15,046,500

表 4.6: gem5 Probe 処理に関わる Tick 数 2dshfl.cpp

図 4.6 および表 4.6 は，行列として用意した共有データに対してグローバルディレクトリコントローラから各キャッシュコントローラへ要求したキャッシュラインの無効化処理の実行時間 (Tick) を表す．ここでは行列サイズに関係なく約 5% から約 10% の実行時間の低減となった．この結果は，ユーザプログラム上でキャッシュを使わずにレジスタファイルで計算を行う GPU 専用の関数 (__shfl)[16] が用いられていることによるものである．表 4.5 でも示されているように，全ての場合において無効化処理の要求数が他の実験と比べて少ないことから，__shfl 関数の使用によりキャッシュの使用が減少することで無効化処理数も少なくなり，結果的に提案した手法の効果もまた生まれにくくなることを示している．

3 つ目の実験は，MatrixTranspose.cpp と呼ばれるユーザプログラムの実行結果となる．このプログラムも 2dshfl.cpp と同様に行列の転置を実行する．行列のサイズを 16×16，32×32，64×64，128×128，256×256，384×384，512×512 と変更し，各行列サイズの場合の計算に要するコヒーレンス処理数および実行時間について，Default 設定と提案手法で比較評価した．結果は図 4.7 と図 4.8 および，表 4.7 と表 4.8 となる．

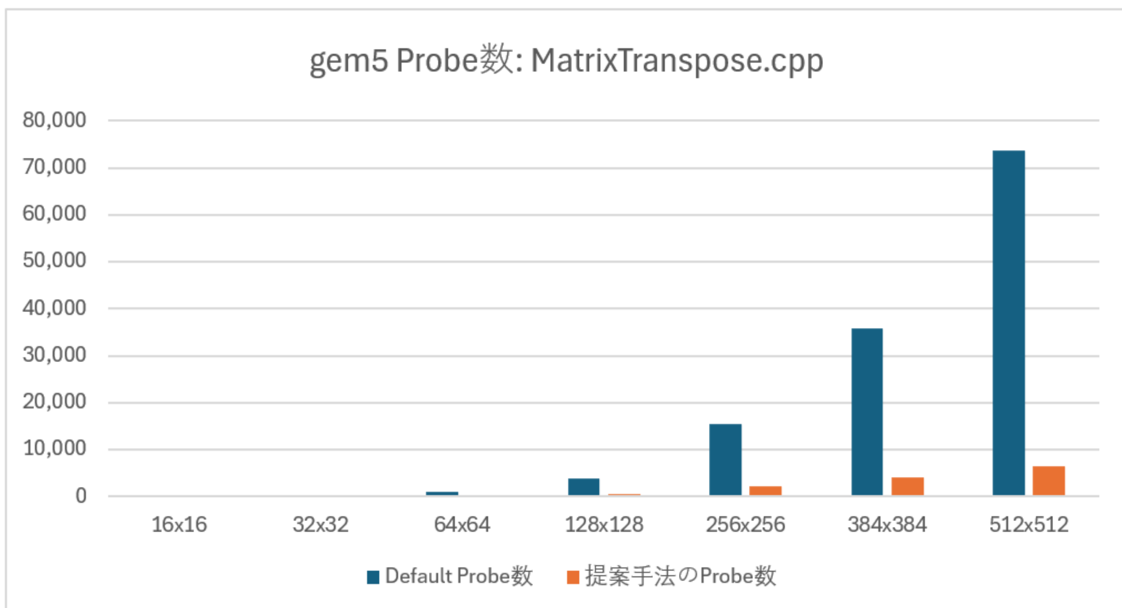

図 4.7: gem5 Probe 数 MatrixTranspose.cpp

行列サイズ	Default Probe数	提案手法のProbe数
16x16	117	58
32x32	220	28
64x64	944	137
128x128	3,886	583
256x256	15,433	2,287
384x384	35,715	4,144
512x512	73,762	6,502

表 4.7: gem5 Probe 数 MatrixTranspose.cpp

図 4.7 と表 4.7 にて，16×16 のとき約 50% ，512×512 のとき約 90% の無効化要求数の低減となった．

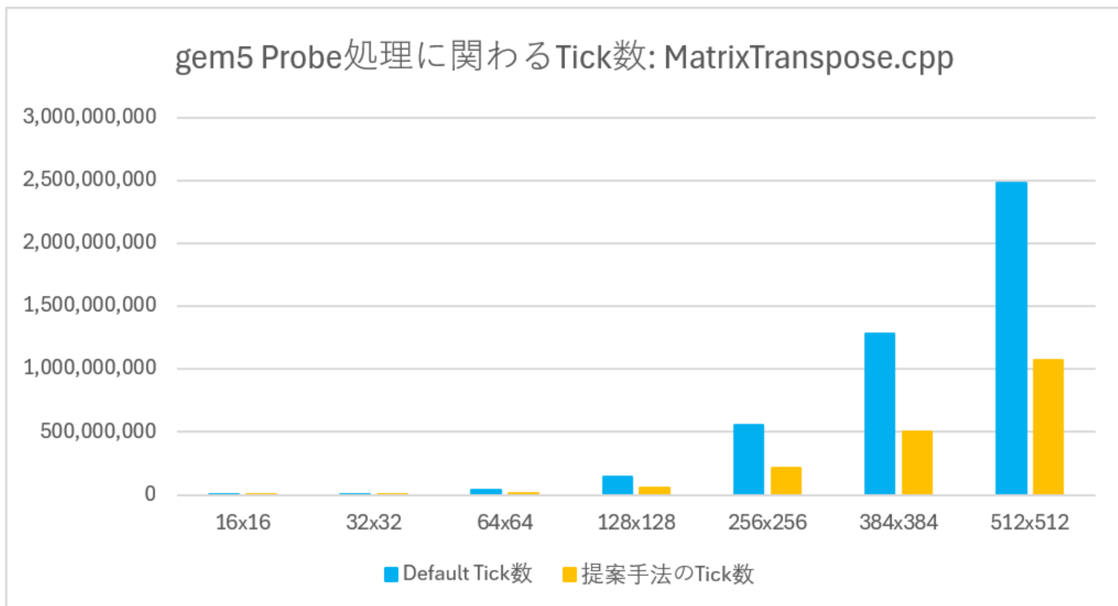

図 4.8: gem5 Probe 処理に関わる Tick 数 MatrixTranspose.cpp

行列サイズ	Default Tick数	提案手法のTick数
16x16	3,342,500	2,062,500
32x32	8,286,000	3,049,500
64x64	34,456,000	13,186,000
128x128	140,079,000	54,585,000
256x256	558,335,500	216,071,500
384x384	1,276,668,500	501,253,500
512x512	2,479,544,000	1,073,722,500

表 4.8: gem5 Probe 処理に関わる Tick 数 MatrixTranspose.cpp

同様に図 4.8 と表 4.8 にて，16×16 のとき約 40% の無効化処理に関わる実行時間の低減となり，その他のサイズにおいても約 60% の無効化処理の実行時間の低減となった．

第5章

おわりに

本研究では，CPU と GPU からなるヘテロジニアス環境においてメモリが共有された場合に生じるキャッシュコヒーレンスの問題を扱っており，こうした環境で使用されるユーザプログラムの分析から得た特徴に基づき，最適なメモリコンシステンシモデルと効率的なコヒーレンス処理を提案した．特にコヒーレンス処理について，CPU と GPU で共有されるメモリは，GPU の並列処理の特性上連続したアドレス領域が確保される性質に注目し，これらの領域への無効化処理を行う際，対象のアドレス情報が連続している場合，1 回の無効化要求でこれらのアドレス帯を一括して無効化処理させるハードウェア機構を提案した．

これらの提案を gem5 が提供するヘテロジニアスのシミュレーション環境にハードウェア機構として実装し Default 設定との比較評価を行った．結果，評価した 3 つのユーザプログラムの全てにおいて提案手法の方が Default 設定よりも無効化処理の要求数と無効化処理の実行時間において優れた性能を持つことを示した．

今後の課題や研究の展望について，下記に述べる．まず課題として，gem5 のヘテロジニアス環境構成は現時点で AMD 社から提供された構成のみとなっており，AMD 社の提供する構成に依存している．このため 4.1.2 節の表 4.1 と表 4.2 で示したように，AMD 社の GPU の全機能が gem5 に完全に実装されておらず，これがシミュレーション上の制約となっている．本研究でもユーザプログラム上でいくつかの標準サポートされている命令の実装を行ったが，gem5 側への未実装により評価を行えないことがあった．今後 gem5 への機能実装が改善されていくことで，こうした制約が解消され，現時点では行えなかった評価が可能になることが期待される．また今後の機能追加により，より精度の高い評価が将来的に行えるようになることも期待される．今後の展望については，本研究で提案したハードウェア処理機構は現在存在しない機構であるため，本研究ではシミュレータを用いて評価したが，今後はこの手法を実現するハードウェア機構の設計に取り組む．

参考文献

[1] https://www.soumu.go.jp/johotsusintokei/whitepaper/ja/r06/pdf/n1320000.pdf (参照 2026-01-19)

[2] https://www.fujitsu.com/jp/documents/products/software/os/linux/catalog/NEDIA_SNIA_CXL.pdf (参照 2026-01-19)

[3] S. Mittal, "A survey of techniques for architecting and managing asymmetric multicore processors,"ACM Computing Surveys, vol.48, no.3, pp.1–38, 2016.

[4] J. Alsop, M. D. Sinclair, and S. V. Adve, "Spandex: A Flexible Interface for Efficient Heterogeneous Coherence," Proc. International Symposium on Computer Architecture (ISCA 2018), pp.261-274, 2018.

[5] B. Choi et al., "DeNovo: Rethinking the memory hierarchy for disciplined parallelism,"Proc. International Conference on Parallel Architectures and Compilation Techniques, pp.155–166, 2011.

[6] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood, "Heterogeneous-race-free memory models," In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp.427-440, 2014.

[7] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood , "A Primer on Memory Consistency and Cache Coherence," Second Edition, Springer, 2020.

[8] P. Keleher et al., "Lazy release consistency for software distributed shared memory,"ACM SIGARCH Comput. Architecture News, vol.20, no.2, pp.13–21, 1992.

[9] https://rocm.blogs.amd.com/software-tools-optimization/mi300a-programming/README.html (参照 2026-01-19)

[10] https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cooperative-groups.html (参照 2026-01-19)

[11] https://www.gem5.org/ (参照 2026-01-19)

[12] H. Brais, R. Kalayappan, and P. R. Panda, "A survey of cache simulators," ACM Computing Surveys (CSUR), vol.53, no.1, pp.1-32, 2020.

[13] Shagufta, Muhammad Aleem, Muhammad Arshad Islam, and Muhammad Azhar Iqbal, "A Comparative Study of Heterogeneous Processor Simulators," International Journal of Computer Applications, vol.148, no.12 , 2016.

[14] https://old.gem5.org/wiki/images/1/19/AMD_gem5_APU_simulator_isca_2018_gem5_wiki.pdf (参照 2026-01-19)

[15] https://www.gem5.org/assets/files/hpca2023-tutorial/gem5-tutorial-hpca-2023.pdf (参照 2026-01-19)

[16] https://rocm.docs.amd.com/projects/HIP/en/latest/how-to/hip_cpp_language_extensions.html (参照 2026-02-01)