2024年度

筑波大学情報学群情報科学類

卒業研究論文

題目

GPU・CPU一体型モジュールにおける Unified Memory使用時の性能評価

主専攻　情報システム主専攻

著者　吉田 智

指導教員　朴 泰祐, 藤田 典久

要　旨

高性能計算分野において、演算加速装置としての GPU の利用が拡大している。GPU は高い演算性能と電力効率を実現している一方、プログラミング時に GPU メモリの管理や CPU-GPU 間データ転送制御を要求するため、プログラミングの生産性を低下させる要因となっている。NVIDIA 社が提供する同社製 GPU 向けの開発環境である CUDA では、Unified Memory (UM) と呼ばれる機能が提供されている。GPU, CPU 双方からアクセス可能な統一メモリ空間を提供し、簡潔なプログラミングを可能にすることでプログラミングの生産性を向上させることが期待されている。しかし GPU-CPU 間のオーバヘッドの大きい転送処理を発生させるため、プログラム全体の性能を落とす要因となっている。NVIDIA GH200 Grace Hopper Superchip は GPU・CPU を一体化させたモジュールである。GH200 では System-Allocated Memory (SAM) と呼ばれる新しい UM を提供しており、ハードウェアによるサポートによって転送性能を向上させている。GH200 を活用することで生産性と性能を両立したプログラミングが可能になると期待できる。そこで本研究では、GH200 の SAM がプログラム内のメモリアクセスに与える影響について分析した。また GH200 上で SAM を使用したプログラムと既存のシステム上の UM を使用したプログラムを実行して、性能やプログラムの生産性の比較を行った。その結果、SAM は GPU-CPU 間メモリアクセスにおいて、場合によっては 2 つを接続するバスの理論性能に匹敵する性能を実現しているほか、GPU からのアクセスが集中したデータを GPU メモリに自動的に移動させることで、プログラマの負担なく性能を改善することがわかった。また従来の UM でもメモリ管理や転送制御が必要となるプログラムにおいても、SAM ではそれらを不要にしてプログラムの生産性を改善させることがわかった。研究によって GH200 は性能とプログラムの生産性と性能の両立において有効性が確認された。

目 次

図目次

表 目 次

第1章　序論

高性能計算 (HPC: High Perfomance Computing) 分野において、演算加速装置としての GPU (Graphics Processing Unit) の利用が拡大している。GPU は高い演算性能と電力効率を実現していることから、電力消費を抑えつつ高い演算性能を確保することが要求されている近年の HPC 分野での重要性が増している。

高い演算性能を実現するために、GPU は独自に広バンド幅のメモリを持っており、プログラミングでは CPU メモリの管理に加えて GPU メモリの管理が求められる。また GPU と CPU はどちらももう一方の持つメモリに直接アクセスできないため、GPU-CPU 間のデータのやり取りにはデータ転送を行う必要があり、プログラミングではこの転送の制御も求められる。その結果、GPU はプログラミングを複雑化させてその生産性を低下させる要因となっている。また GPU-CPU 間の転送性能は演算性能と比較して低く、性能のボトルネックになる場合が多い。

GPU プログラムを開発するための環境として、NVIDIA 社が提供する同社製 GPU 向け開発環境の CUDA がある。CUDA 6.0 より Unified Memory (UM) という機能が導入されている。UM は CPU と GPU の両方からアクセス可能な統一メモリ空間を提供し、CPU-GPU 間のデータ転送を自動化することで、GPU メモリの管理や転送制御が不要な簡潔なプログラミングが可能にしている。UM を使用することで、プログラミングの生産性を向上させることが期待できる。しかし UM では GPU-CPU 間アクセスを行う度に、ページフォールトを伴うオーバーヘッドの大きな転送処理を頻発させる。そのため使用した場合にアクセス速度を低下させて、プログラム全体の性能が低下する可能性があるというデメリットがある。

NVIDIA GH200 Grace Hopper Superchip は同社製の Hopper GPU と Grace CPU を、NVlink-Chip-2-Chip (C2C) と呼ばれる広帯域で低レイテンシなバスを介して密結合させた GPU・CPU 一体型モジュールである。既存のシステムでは一般的に GPU は独立した部品としてマザーボード上の PCIe などのバスによって CPU と接続されるが、GH200 は一枚の小さな基板上に CPU と GPU のチップをはめ込んだ構造を持つ。GH200 では System-Allocated Memory (SAM) と呼ばれる新しい UM を提供している。SAM では転送処理を伴わない高速な GPU-CPU 間アクセスを実現しており、アクセス性能の改善が期待できる。GH200 の SAM を活用することで性能とプログラミングの生産性の両立が可能になると期待できる。

本研究では、GH200 の SAM がプログラム内のメモリアクセスに与える影響について分析し、どのようなプログラムが SAM の恩恵を受けることができるか示す。また同じプログラムにおいて、既存のシステムで UM を使用した場合と GH200 の SAM を使用した場合を、性能やプログラミングの生産性の観点から比較する。分析や比較を通じて、GH200 が生産性と性

能を両立する手段としての有効性を検証する。

本論文では、まず 2 章で研究の背景や目的、関連研究について述べる。3 章で GH200 のアーキテクチャや構成要素、メモリシステムの詳細について述べる。4 章で評価に用いるプログラムやベンチマークについて述べる。5 章で性能評価について述べる。最後に、6 章で結論を述べる。

第2章　研究背景

この章では研究の背景、研究の目的、関連研究について述べる。

2.1　GPU

GPU は大規模な並列処理に特化した演算用プロセッサである。その名の通り、元々は画像や動画などのグラフィック処理に用いられていたが、近年では GPGPU (General Purpose computing on GPU) と呼ばれる汎用計算への利用も行われている。GPU は非常に多数のコアと大量のデータへの同時アクセスを可能にする独自の広バンド幅のメモリを用いて、大規模な並列処理を実行する。GPU では多数のコアを一括制御するため、並列処理を実行する場合に演算あたりの電力消費を抑えることができる。それゆえ、GPU では大規模な並列処理を実行することで、高い演算性能と電力効率を実現している。

先述の通り GPU は独自のメモリを持つため、プログラミングでは CPU, GPU それぞれが保持する 2 つのメモリ空間の管理が必要である。また GPU と CPU は互いにもう一方のメモリに直接アクセスできないため、GPU-CPU 間でデータのやり取りにはデータ転送の制御が必要である。これらはプログラミングを複雑化させて、プログラミングの生産性を落とす要因となっている。また多くの場合 GPU-CPU 間の通信性能は演算性能と比較して低いため、性能のボトルネックになる場合が多く、性能を引き出すためにデータアクセスや通信の最適化が必要となる。しかし最適化によってプログラムが複雑化することで、より生産性が落ちる場合が多い。

2.2　CUDA

CUDA は NVIDIA 社が提供している、同社製 GPU 向けの開発環境である。GPU プログラムを開発するための API やライブラリ、コンパイラを含んでおり、プログラマはこれを使用することで簡単に GPU プログラムの開発が可能である。

CUDA 6.0 より、Unified Memory (UM) と呼ばれる機能が導入されている。双方からアクセス可能な統一メモリ空間を提供して、簡潔なプログラミングを可能にする機能である。使用するメモリ空間が 1 つとなるため、メモリ管理が容易となる。データは物理的には CPU、GPU メモリのどちらか一方に格納される。GPU-CPU 間メモリアクセスの場合には、アクセスした側のメモリに物理的にページ単位でデータ転送を行うことでこれを実現している。このデー

タ転送はアクセス時に自動的に行われるため、プログラマによる転送制御は不要となる。この機能によってプログラミングの複雑さが解消され生産性が向上する。

しかし、UM では GPU-CPU 間アクセスの際にページフォールトを伴う転送処理が発生する。ページフォールトの処理のオーバヘッドは大きく、この転送処理のオーバーヘッドも大きくなっている。また不要な転送を行う可能性もある。そのため UM を使用することで、転送性能が悪化してプログラム全体の性能が低下してしまうことが多い。

2.3 GH200

NVIDIA GH200 Grace Hopper Superchip は NVIDIA が開発した GPU・CPU 一体型モジュールである。同社製の Grace CPU と Hopper GPU を NVlink-C2C によって 1 つの基板上で密結合させた構造を持つ。NVlink-C2C は広帯域、低レイテンシ、キャッシュコヒーレントなバスである。GH200 ではハードウェアによってサポートされた新しい UM である SAM を使用することができる。SAM 上での GPU-CPU 間アクセスは従来の UM のように転送処理を発生させることはなく、高速化されている。またアクセスはキャッシュ経由で行われるため、キャッシュの利用によってアクセスの回数を減らすことが可能である。

詳細については 3 章で述べる

2.4 研究の目的

本研究の目的は、GH200 の SAM がプログラム上のメモリアクセスに与える影響を明らかにし、どのようなプログラムに対して特に恩恵があるか示すこと、既存システムと比較して GH200 の有効性を確かめることである。そこで SAM によるメモリアクセスの性能の測定しその影響を分析する。同じプログラムを UM で GPU 化し既存システム上で実行した場合と、SAM を使用して GH200 システム上で実行した場合を、性能やプログラミングの観点から比較する。

2.5 関連研究

[1] では統一メモリシステムが GPU アプリケーションに与える影響を定量化している。6 つの異なるアクセスパターンを持つアプリケーションにおいて、従来の UM を使用したバージョンと GH200 の SAM を使用したバージョンを比較し、アプリケーションに対する SAM の影響を評価している。Schieffer らは、SAM の影響の出方や度合いはアプリケーションの性質によって異なり、GH200 の SAM はアプリケーションによっては従来の UM よりも高い性能を引き出すことが明らかにしている。本研究では SAM 上での様々なアクセスパターンの性能に重点を置いて評価し、どのようなアクセスパターンが高速化されるかを把握し、どのようなアプリケーションであれば高い性能を引き出せるかを示す。また既存システム上で従来の

UM を使用した場合と GH200 上で SAM を使用した場合を比較して、性能や生産性の観点からその有効性を明らかにする点に本研究の独自性がある。

第3章　GH200のアーキテクチャ

この章では GH200 のアーキテクチャやメモリシステムの詳細について述べる。

3.1　アーキテクチャ

GH200 は Grace CPU を Hopper GPU と NVlink-C2C によって密結合させた一体型モジュールである。それぞれのチップを内装したソケットやカードをマザーボード上の PCIe で接続する既存のシステムと異なり、1 枚の小さな基板上にチップをはめ込み、NVlink-C2C で接続する構造を持つ。図 3.1 に GH200 のアーキテクチャの概要を示す。

GH200 を構成する要素について説明する。Grace CPU は NVIDIA が開発した初のデータセンター向け CPU である。72 個の Arm Neoverce V2 コアと、Scalable Coherency Fabric (SCF) を呼ばれる分散キャッシュとそれらをコアと接続するメッシュ構造を持つ [3]。SCF はコア、L3 キャッシュ、メモリ、インターコネクトを総バンド幅 3.2TB/s で相互接続しデータのルーティングも行う。Hopper GPU は同社が開発した第 9 世代のデータセンター向け GPU である。通常の演算コアの他、Tensor コアと呼ばれる AI に特化したプロセッサを持つ。NVlink-C2C は広バンド幅、低レイテンシでありキャッシュコヒーレンシーを提供する専用のバスである。双方向最大 900GB/s の帯域幅を持ち、これは一般的なバスである PCIe Gen 5 の約 7 倍である。

CPU は最大 480GB の LPDDR5X を、GPU は 96GB の HBM3 または 144GB の HBM3e をメモリとして持つ。LPDDR5X は容量によって異なるが最大 512GB/s、HBM3 は 4TB/s、HBM3e は 4.9TB/s の帯域幅を持つ。

3.2　メモリシステム

GH200 では System-Allocated Memory (SAM) と呼ばれる新しい UM を提供している。以降、2.2 で説明した従来型の UM を UM と呼び、System-Allocated Memory を SAM を呼ぶ。UM と異なり、GPU-CPU 間のメモリアクセスの際に転送処理を発生させないことで高速なメモリアクセスを実現している。この機能は Address Translation Service (ATS) と呼ばれる仕組みによって提供されている。ATS は GPU,CPU 両方からのメモリアクセスに対して、両方の物理メモリへのアドレス変換を行う仕組みである。ATS によって、両方のプロセッサが 1 つのメモリテーブルを共有して双方のメモリへ直接アクセスすることが可能になっている。また NVlink-C2C が提供するキャッシュコヒーレンシーによって、キャッシュを介したアクセスが可能である。

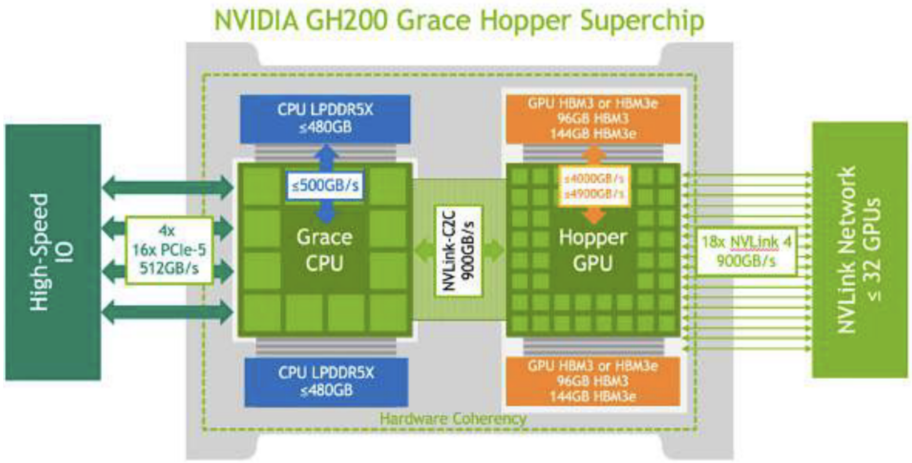

図 3.1: GH200 の構造 [2]

SAM は C 言語の malloc() などの標準的な動的確保の方法で割り当てられる、SAM では malloc() が呼び出された時、ページテーブルエントリのみを作成する。その後最初にアクセスしたプロセッサの持つ物理メモリに領域を確保してデータを格納する。この動作は First Touch と呼ばれる。また SAM には Migration と呼ばれる機能がある。これは GPU から頻繁にアクセスされた CPU の物理メモリのデータを、ページ単位で GPU の物理メモリへ移動させる機能である。GPU のメモリアクセスを監視するハードウェアのカウンタによって、ページを Migration させるか決定する。この機能によって頻繁にアクセスされるデータに対してのアクセスの性能を向上させる。First touch と Migration によってデータを最適な物理メモリに格納することで不要なデータ転送を削減し、プログラムの性能を向上させることが可能である。

3.3 UM を使用したプログラミング

UM を使用した際のプログラミングについて、通常の GPU プログラミングと比較して述べる。

最初に、CUDA のプログラミングの概要について説明する。CUDA では GPU をホスト、GPU をデバイスと呼ぶ。デバイス側で実行したい処理をカーネルと呼ばれる関数によって記述し、それを呼び出すことでデバイス側で計算を行う。

リスト 4.8 に通常のプログラムの例を、リスト 4.9 に UM を使用したプログラムの例を挙げる。1000 個の要素を持つ float 型の配列 A と B の和を C に格納するプログラムである。リストにはメモリや転送に関する部分のみ記述している。

通常の CUDA プログラムについて説明する。図 3.3 に通常時のプログラムにおけるメモリアクセスの概要を示す。棒線矢印が自身の側のメモリへのアクセス、破線矢印がもう一方のメモリへのアクセスを示す。図に示す通り、通常のプログラムではホストはホストメモリ、デ

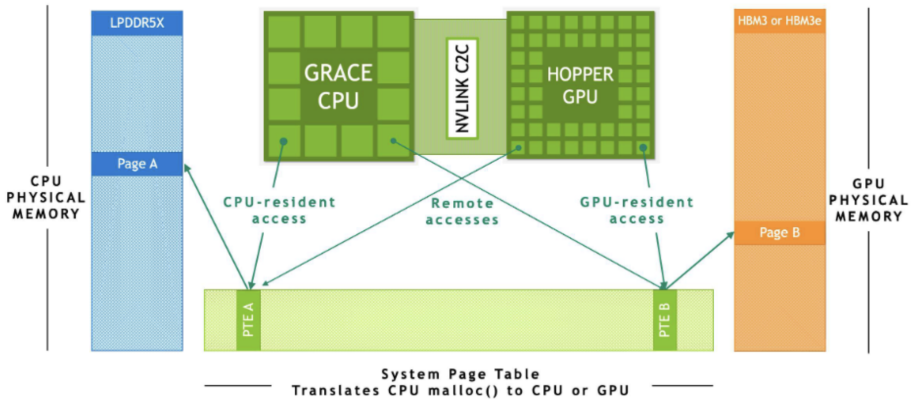

図 3.2: ATS による GPU-CPU 間アクセス [2]

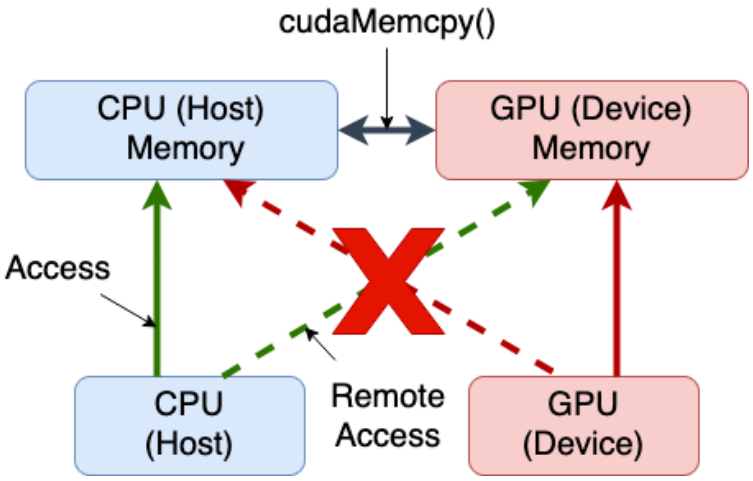

図 3.3: 通常のメモリアクセス

バイスはデバイスメモリのみにアクセスすることができる。もう一方のメモリへのアクセスは不可能であり、アクセスした場合は実行時にエラーが返される。そのためデバイスで計算を実行する場合は、CPU 側で初期化したホストメモリのデータのデバイスメモリで確保した領域への転送をプログラム内で制御する必要がある。その後、カーネルを呼び出し計算を実行した後、計算結果をデバイスからホストに転送する際も明示する必要がある。リスト 4.8 では、配列 A,B,C の領域を両方のメモリで確保している。混同を避けるために、ホスト側には h_をデバイス側には d_という接頭辞をつけている。宣言の後、リストの 6〜8 行目でホストメモリに領域を確保している。通常の C 同様 malloc() で確保する。リスト 11〜13 行目でデバイスメモリに領域を確保している。デバイスメモリは cudaMalloc() で確保する。h_A, h_B を初期化した後、16,17 行目の cudaMemcpy() で d_A, d_B にそれぞれデータを転送している。その後、20 行目でカーネルを呼び出す。このとき引数として確保したデバイス領域を指すポイン

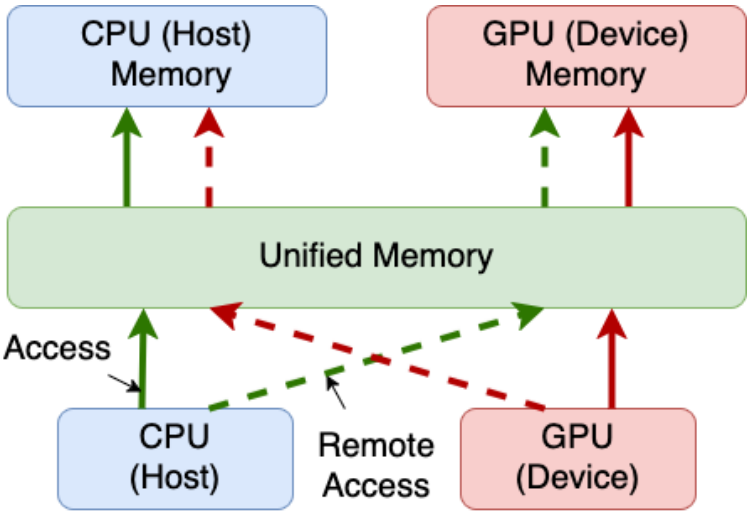

図 3.4: UM 上でのメモリアクセス

タを渡す。22 行目の cudaDeviceSynchronize() でデバイスの計算が終了するまで同期を取る。カーネルの呼び出しは非同期であり、ホストは呼び出し後すぐに次の処理を開始するため、計算結果を利用する場合には同期が必要である。同期をとった後、結果をホストが利用する場合には、結果を転送する必要がある。そのため、23 行目の cudaMemcpy() で d_C から h_C にデータを転送する。これが通常の CUDA プログラムの流れである。

UM を使用したプログラムについて説明する。図 3.4 に UM 使用時のメモリアクセスの概要を示す。図に示す通り、ホストとデバイス両方からアクセス可能な統一されたメモリ空間が提供される。この空間を介して両方のメモリへのアクセスが可能となる。UM を使用することで、GPU を使用しない C に近いプログラミングが可能となる。リスト 4.9 では UM に A,B,C の領域を確保している。宣言の後, リストの 5〜8 行目で領域を確保している。このとき従来の UM を使用するには cudaMallocManage()、GH200 において SAM を使用するには malloc() で確保する。初期化した後、リスト 11 行目でそのまま領域を指すアドレスを渡してカーネルを呼び出して計算を実行させている。計算後同期をとった後、ホストが結果を利用する場合にはそのまま結果が格納されている C にアクセスすれば良い。これが UM を使用した場合の流れである。通常のプログラムと比較して、使用する空間が 1 つになることでメモリ確保の回数や管理すべきポインタの数が減っている。ホスト側の処理もデバイス側の処理も同じメモリ領域にアクセスすることができ、メモリアクセスが正しいかどうか考慮する必要もなくなっている。またデータ転送の制御も不要となっており、GPU を使用しない C プログラムに近い記述になっている。このため、C に慣れているプログラマならば、プログラミングがよりやりやすくなっている。また GPU を使用することによって増える作業は必要なくなることで、プログラムの改善に集中することも可能になる。それゆえ、UM を使用することで生産性を向上させることができるといえる。

リスト 3.1: 通常の GPU プログラミング

```
float *h_A, *h_B, *h_C;
float *d_A, *d_B, *d_C;
size_t n_size = 1000 * sizeof(float)

// ホストメモリに領域を確保
h_A = (float*)malloc(n_size);
h_B = (float*)malloc(n_size);
h_C = (float*)malloc(n_size);

// デバイスメモリに領域を確保
cudaMalloc((void**)&d_A, n_size);
cudaMalloc((void**)&d_B, n_size);
cudaMalloc((void**)&d_C, n_size);

// A, B を初期化後にデバイスメモリ上の領域にデータ転送
cudaMemcpy(d_A, h_A, n_size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, n_size, cudaMemcpyHostToDevice);

// カーネルを呼び出し, GPU で C=A+B を計算
GPUkernel<<<grid, block>>>(d_C, d_A, d_B);

// ホストとデバイスの間で同期をとる
cudaDeviceSynchronize();

// 計算結果 C をホストに転送
cudaMemcpy(d_C, h_C, n_size, cudaMemcpyDeviceToHost);

// 領域を解放
cudaFree(d_A);
free(h_A);
```

リスト 3.2: UM を使用したプログラミング

```
float *A, *B, *C;
size_t nSize = 1000 * sizeof(float)

// UM に領域を確保
A = (float*)malloc(nSize) or cudaMallocManaged((void**)&A, nSize)
    ;
B = (float*)malloc(nSize) or cudaMallocManaged((void**)&B, nSize)
    ;
C = (float*)malloc(nSize) or cudaMallocManaged((void**)&C, nSize)
    ;

// A, B を初期化後にカーネルを呼び出しデバイスで C=A+B を計算
```

```
10 GPUkernel<<<grid, block>>>(C, A, B);
11
12 // ホストとデバイスの間で同期をとる
13 cudaDeviceSynchronize();
14
15 // 領域を解放
16 free(A) or cudaFree();
```

第4章　評価方法

この章では実験に使用するプログラムはベンチマークについて述べる。

4.1　予備評価

通常のメモリ領域におけるメモリ性能およびインターコネクトの性能を測定する。

メモリ性能の測定では、double 型配列 A,B において、A[i]=B[i] を実行し、その時の性能を測定する。リスト 4.1 に CPU メモリの性能測定のコードを示す。リストでは重要な要素のみを示す。配列の領域の確保は、CPU メモリ固定の領域を確保する 9, 10 行目の cudaMallocHost() で行う。10〜14 行目でそれぞれの配列を CPU にて初期化する。19, 20 行目の A[i] = B[i] を for ループで繰り返すことで配列のコピーを行う。18 行目の#pragma omp の指示分を挿入して for ループの OpepnMP による並列化を行なっている。ただしコンパイラによる最適化は行なっていない。その部分を現在の時刻を返す 1 行目に定義した cpuSecond() で挟み、その差分を取ることで実行時間を測る。

リスト 4.2 に GPU メモリの性能測定のコードを示す。2 行目から定義した initFromGPU() は GPU での配列の初期化, 10 行目から定義した memTest() は配列のコピー処理を行うカーネルである。メモリ確保は GPU メモリ固定の領域を確保する cudaMalloc() で行う。配列を初期化したのち、24 行目で memTest() を CPU から呼び出すことで GPU にて処理を行う。処理が終了したのち cudaDeviceSynchronize() によって GPU と同期を取る。この部分の実行時間を CPU の場合と同様に測定する。

インターコネクトの測定では、CUDA の転送処理を指示する cudaMemcpy() による、double 型配列のデータ転送を行い、その実行時間から性能を測定する。リスト 4.3 に NVlink-C2C の性能測定のコードを示す。hM は cudaMallocHost()、dM は cudaMalloc() で確保する。CPU メモリにある hM と GPU メモリにある dM をそれぞれメモリ性能測定と同様に初期化する。3〜6 行目で cudaMemcpy() によって hM から dM へのデータ転送を行い、CPU から GPU へのデータ転送の実行時間を nTime 回測定する。11〜15 行目で dM から hM へのデータ転送を行い、GPU から CPU へのデータ転送の実行時間を同じく nTime 回測定する。

それぞれの処理を 200 回連続で行い、1 つ 1 つの処理の実行時間を測定する。これを 1 つの実験として 10 回行う。また配列のサイズが 4GB と 8GB の 2 つの場合で実験を行う。

リスト 4.1: CPU メモリの性能測定

```
1 double cpuSecond() {
2    struct timeval tm;
```

```
3     gettimeofday(&tm,NULL);
4     return (double)(tm.tv_sec) + ((double)(tm.tv_usec)) / 1.0e6;
5  }
6
7  int main(int argc, char **argv) {
8  // 領域の確保
9     cudaMallocHost((void**)&A, nSize);
10    cudaMallocHost((void**)&B, nSize);
11 // 配列の初期化
12 #pragma omp parallel for
13    for (i = 0; i < nElem; ++i) {
14       A[i] = 1;
15       B[i] = (i + j) % 10;
16    }
17
18 // 性能測定
19    start = cpuSecond();
20 #pragma omp parallel for
21    for (i = 0; i < nElem; ++i)
22       A[i] = B[i];
23    end = cpuSecond();
24 ...
25 }
```

リスト 4.2: GPU メモリの性能測定

```
1  // GPU 上での初期化
2  __global__ void initFromGPU(double *A, double *B) {
3     int i = threadIdx.x + blockIdx.x * blockDim.x;
4
5     A[i] = 1;
6     B[i] = i % 10;
7  }
8
9  // コピー処理の実行
10 __global__ void memTest(double *A, double *B) {
11    int i = threadIdx.x + blockIdx.x * blockDim.x;
12
13    A[i] = B[i];
14 }
15
16 int main(int argc, char **argv) {
17 // メモリ領域の確保
18    cudaMalloc((void**)&A, nSize);
19    cudaMalloc((void**)&B, nSize);
20 // 配列の初期化
```

```
21      initFromGPU<<<grid, block>>>(A, B);
22      cudaDeviceSynchronize();
23
24  // メモリ性能測定
25      start = cpuSecond();
26      memTest<<<grid, block>>>(A, B);
27      cudaDeviceSynchronize();
28      end = cpuSecond();
29  ...
30  }
```

リスト 4.3: NVlink-C2C の性能測定

```
1   int main(int argc, char **argv) {
2   // 領域の確保
3       cudaMallocHost((void**)&hM, nSize)
4       cudaMalloc((void**)&dM, nSize)
5   // HtoD の性能測定
6       start = cpuSecond();
7       cudaMemcpy(dM, hM, nSize, cudaMemcpyHostToDevice);
8       end = cpuSecond();
9   ...
10  // DtoH の性能測定
11      start = cpuSecond();
12      cudaMemcpy(hM, dM, nSize, cudaMemcpyDeviceToHost);
13      end = cpuSecond();
14  }
```

4.2 SAM 上でのメモリ性能測定

SAM において、さまざまなパターンのメモリアクセスの性能を測定する。以下に行う実験を列挙する。

4.2.1 8 パターンのメモリアクセス

SAM 上で 8 パターンのメモリアクセスの性能を測定する。SAM を使用したプログラムではどこから読み出すか、どこに書くこむか、その処理をどちらが実行するか決定することで 8 パターンのメモリアクセスが想定される。図 4.1 に 8 つのアクセスパターンを示す。それぞれのパターンに (1)〜(8) の番号をつけている。配列 Src と Dst を First Touch を用いてそれぞれのメモリに配置し、Dst[i]=Src[i] の処理を GPU または CPU で実行して 8 パターンの性能を測定する。測定は連続して 200 回行い、それを 1 つの実験として計 10 回行う。それぞれの配列の大きさが 4GB および 8GB の 2 つの場合で実験を行う。

リスト 4.4 に CPU 側の初期化のコードを、リスト 4.5 に GPU 側の初期化のコードを示す。First Touch では、領域を確保する malloc() を呼び出した後、一番最初にアクセスしたプロセッサ側の物理メモリの領域を割り当てる。CPU 側に割り当てる場合は、malloc() の後、リスト 4.4 で初期化、GPU 側に割り当てる場合は、リスト 4.5 で初期化することで割り当てることができる。

CPU で処理を行う場合のコードをリスト 4.6、GPU で処理を行う場合をリスト 4.7 に示す。CPU の場合では、7, 8 行目の for ループで Dst[j]=Src[j] を実行している。予備評価同様、性能の上限に達するために 6 行目の指示分を入れて OpenMP を使用している。GPU の場合では、2〜5 行目に定義されたカーネルによって処理を実行する。14 行目でカーネルを呼び出して実行し、実行時間を測定する。以上の 2 つのコードを組み合わせることで、計 8 パターンの測定を行う。

この測定によって、起こりうるアクセスパターンにおける性能を測定しメモリアクセスが頻発することによる Migration の挙動を把握する。

4.2.2 Migration の挙動

First Touch を用いて、Src を CPU メモリに Dst を GPU メモリに割り当て、上記の処理を GPU で 100 回実行したのち CPU で 100 回実行し再度 GPU で 100 回実行する。これを 1 つの実験として 10 回行う。配列のサイズが 4GB と 8GB の 2 つの場合で実験を行う。

この測定によって、Migration の挙動をより詳細に把握する。

リスト 4.4: CPU 側での初期化

```
1 int main() {
2     // CPU 上での初期化
3     #pragma omp parallel for
4     for (i = 0; i < nElem; ++i)
5         Src[i] = i % 10;
6         Dst[i] = 0;
7 }
```

リスト 4.5: GPU 側での初期化

```
1 // GPU 上での初期化
2 __global__ void initArrayOnDevice(double *A, double *B, const int
      nElem) {
3     int i = threadIdx.x + blockIdx.x * blockDim.x;
4
5     if (i < nElem) {
6         A[i] = i % 10;
7         B[i] = 0;
8     }
9 }
```

```
10
11 int main() {
12     ...
13     initArrayOnDevice<<<grid, block>>>(Dst, Src, nElem);
14     ...
15 }
```

リスト 4.6: CPU で処理を実行する場合

```
1  int main() {
2      ...
3      // メモリ性能実行
4      for (i = 0; i < nTime; ++i) {
5          // CPU で処理を実行
6          start = cpuSecond();
7      #pragma omp parallel for
8          for (j = 0; j < nElem; ++j)
9              Dst[j] = Src[j];
10         end = cpuSecond();
11     }
12     ...
13 }
```

リスト 4.7: GPU で処理を実行する場合

```
1  // GPU での処理を実行
2  __global__ void CopyArrayOnDevice(double *Dst, double *Src, const
       int nElem) {
3      int i = threadIdx.x + blockIdx.x * blockDim.x;
4
5      if (i < nElem) Dst[i] = Src[i];
6  }
7
8  int main() {
9      ...
10     // メモリ性能測定
11     for (i = 0; i < nTime; ++i) {
12         start = cpuSecond();
13         CopyArrayOnDevice<<<grid, block>>>(Dst, Src, nElem);
14         cudaDeviceSynchronize();
15         end = cpuSecond();
16     }
17     ...
18 }
```

4.3 姫野ベンチマーク

GH200 のシステムと既存のシステムを比較するためのベンチマークとして、姫野ベンチマークを使用する。

姫野ベンチマークは、ポアソン方程式の解をヤコビの反復法で求める処理を行うベンチマークである [4]。3 次元の空間を格子点に分割し、全ての点に対して計算を行なって更新を繰り返すことで方程式を解く。図 4.2 に姫野ベンチマークにおける空間の 1 点の計算の概要を示す。姫野ベンチマークでは、空間上の 1 点 1 点を、自身とその周りの 18 点のデータを用いて更新していくステンシル計算を行う。1 点の計算には合計 19 点のデータへのアクセスが必要となり、それが空間の全点に対して実行されるため、実行中に非常に多量のデータアクセスが行われる。そのため実行するシステムのメモリ性能に結果が大きく依存する。

姫野ベンチマークには C と Fortran の 2 つのバージョンのコードが存在する。本研究では C バージョンのコードを CUDA によって GPU 化した。また測定するシステムの規模に応じて 4 つの問題サイズが選択可能である。本研究では最も大きい 512x512x1024 を選択した。

姫野ベンチマークはメモリを確保する newMat() や解放する freeMat()、繰り返す回数 nn を受け取りヤコビの反復法を実行する jacobi() などの関数で構成される。

姫野ベンチマークの処理の流れを説明する。

1. 実行コマンドの引数から、問題サイズを決定する。
2. 必要な配列のためのメモリ領域を問題サイズに応じて newMat() で確保する。
3. 配列を初期化する。
4. ヤコビ法の繰り返しの数を 3 回とし、jacobi() を実行、その実行時間を測定する。これをリハーサル測定と呼ぶ
5. 測定の結果の MFLOPS を表示し、60 秒間計算するための繰り返しの回数を決定する。
6. jacobi() を実行、実行時間を測定し、結果の MFLOPS、Pentimu III 600 MHz の性能に対する比を表示する。

GPU 化するにあたり、リハーサル測定を削除している。リハーサル測定によって UM や SAM において、データ転送が発生することでその後の測定が正確に行えなくなることを防ぐためである。またヤコビ法の繰り返しの回数は 3000 回とした。

GPU 化においては、求解処理をする関数 jacobi() を GPU 化した。またメモリの確保方法が異なる 3 つのバージョンのコードを作成した。個別のメモリ管理と転送制御が必要な通常の CUDA バージョン、UM を使用したバージョン、SAM を使用したバージョンの 3 つである。

リスト 4.8 に通常バージョン、リスト 4.9 に UM バージョン、リスト 4.10 に SAM バージョンのコードの jacobi() 部分を示す。リストには重要な部分を示している。またメモリ管理やデータ転送の部分については 1 つのデータ領域に対するもののみを示しており、実際はこれらの関数の呼び出しがデータ領域の個数分存在する。リスト 4.8 の 1〜10 行目に示す通り、姫

野ベンチマークでは、静的に確保された Matrix 型の構造体のメンバに、計算で使用される動的に確保された配列へのポインタを格納するデータ構造を持つ。そのため通常のバージョンでは、リスト 4.8 の 19 行目で構造体の領域、21 行目で配列の領域をそれぞれ GPU メモリに確保している。データ転送もそれぞれの領域に対して必要であり、24 行目で構造体の 26 行目で配列のデータ転送をしている。また 28 行目に示す GPU メモリ上の構造体と配列を結びつける処理も必要である。従来の UM は配列部分にのみ適用可能であり、配列の領域の管理とデータ転送は不要であるが、4.9 に示す通り 6 行目の構造体自体の領域の管理と 9 行目のデータ転送は必要である。一方で SAM では、GPU は静的に確保されたメモリにもアクセス可能であるため、リスト 4.10 に示す通りメモリ管理や転送制御を記述することなく GPU での計算が可能である。

GH200 搭載システムでは全てのバージョンを、既存システムは通常の CUDA と Managed Memory のバージョンを実行する。得られた性能やプログラムの生産性から GH200 の SAM を評価する。

リスト 4.8: 通常バージョンの jacobi()

```
1  struct Mat {
2    float* m;
3    int mnums;
4    int mrows;
5    int mcols;
6    int mdeps;
7  };
8
9  /* prototypes */
10 typedef struct Mat Matrix;
11
12
13 float jacobi(int nn, Matrix* a,Matrix* b,Matrix* c,
14     Matrix* p, Matrix* bnd, Matrix* wrk1,Matrix* wrk2)
15 {
16 Matrix *da, *db, *dc, *dp, *dbnd, *dwrk1, *dwrk2;
17 float *dam, *dbm, *dcm, *dpm, *dbndm, *dwrk1m, *dwrk2m, *dgosa;
18
19 cudaMalloc((void**)&da, mSize);
20 ...
21 cudaMalloc((void**)&dam, a->mnums * a->mrows * a->mcols * a->
       mdeps * sizeof(float));
22 ...
23
24 cudaMemcpy(da, a, sizeof(Matrix), cudaMemcpyHostToDevice);
25 ...
26 cudaMemcpy(dam, a->m, a->mnums * a->mrows * a->mcols * a->mdeps
       * sizeof(float), cudaMemcpyHostToDevice);
```

```
...
cudaMemcpy(&(da->m), &dam, sizeof(float*), cudaMemcpyHostToDevice
    );
...

for(n=0 ; n<nn ; n++){
 jacobi_step<<<grid, block>>>(da, db, dc, dp, dbnd, dwrk1, dwrk2,
   imax, jmax, kmax);
 cudaDeviceSynchronize();

 jacobi_update<<<grid, block>>>(dp, dwrk2, imax, jmax, kmax);
 cudaDeviceSynchronize();
} /* end n loop */

cudaFree(dam);
...

cudaFree(da);
...
cudaDeviceReset();
}
```

リスト 4.9: UM バージョンの jacobi()

```
float jacobi(int nn, Matrix* a,Matrix* b,Matrix* c,
      Matrix* p,Matrix* bnd,Matrix* wrk1, Matrix* wrk2)
{
  Matrix *da, *db, *dc, *dp, *dbnd, *dwrk1, *dwrk2;

  cudaMalloc((void**)&da, mSize);
  ...

  cudaMemcpy(da, a, mSize, cudaMemcpyHostToDevice);
  ...

  for(n=0 ; n<nn ; n++){
    jacobi_step<<<grid, block>>>(da, db, dc, dp, dbnd, dwrk1,
        dwrk2,
      omega, imax, jmax, kmax);
    cudaDeviceSynchronize();
    jacobi_update<<<grid, block>>>(p, wrk2, imax, jmax, kmax);
    cudaDeviceSynchronize();
  } /* end n loop */

  cudaFree(da);
  ...
```

```
22 }
```

リスト 4.10: SAM の jacobi()

```
1  float
2  jacobi(int nn, Matrix* a,Matrix* b,Matrix* c,
3       Matrix* p,Matrix* bnd,Matrix* wrk1,Matrix* wrk2)
4  {
5    ...
6    for(n=0 ; n<nn ; n++){
7      jacobi_step<<<grid, block>>>(a, b, c, p, bnd, wrk1, wrk2,
8        omega, imax, jmax, kmax);
9      cudaDeviceSynchronize();
10     jacobi_update<<<grid, block>>>(p, wrk2, imax, jmax, kmax);
11     cudaDeviceSynchronize();
12
13   } /* end n loop */
14 }
```

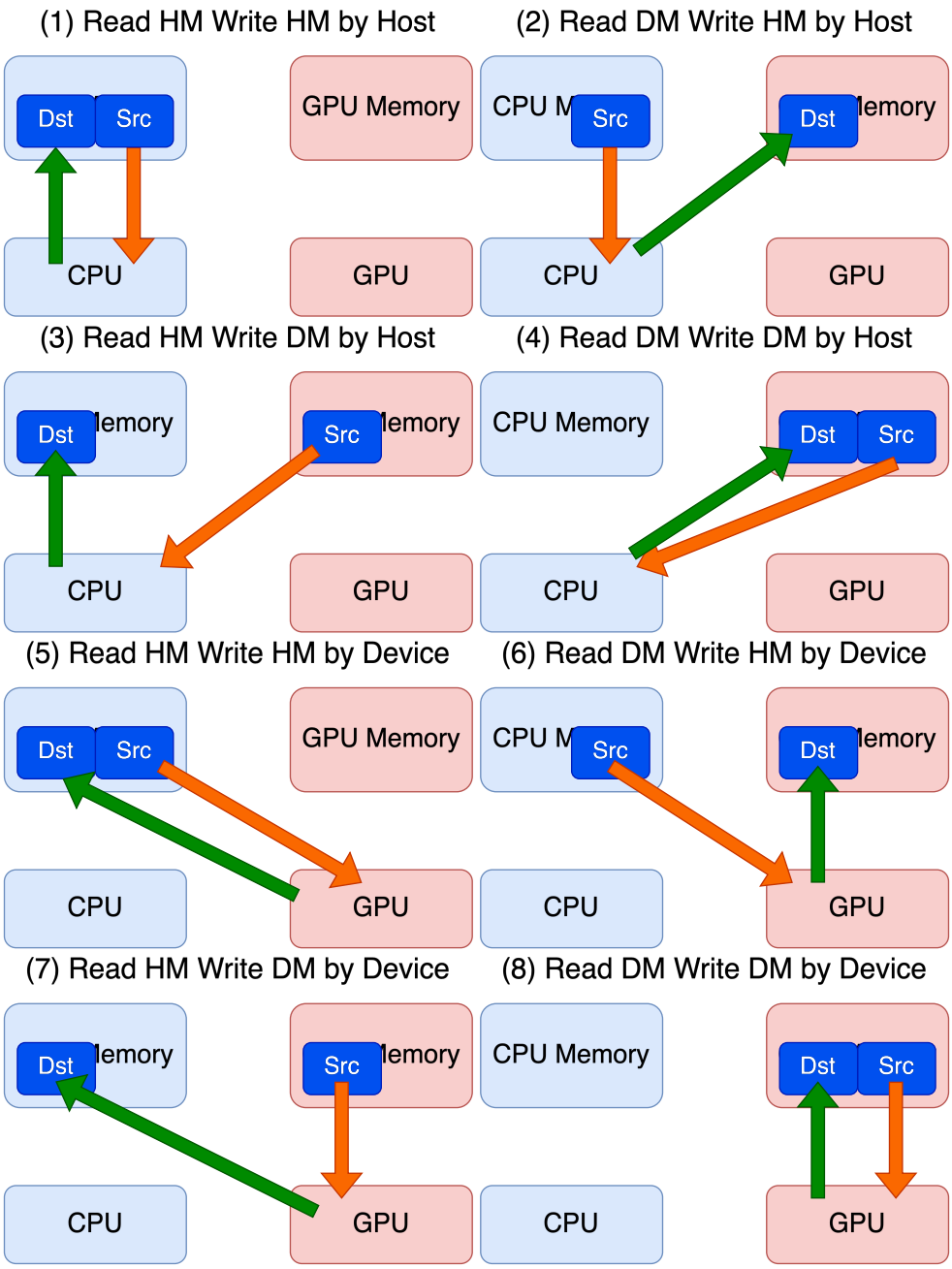

図 4.1: 8 パターンのアクセス

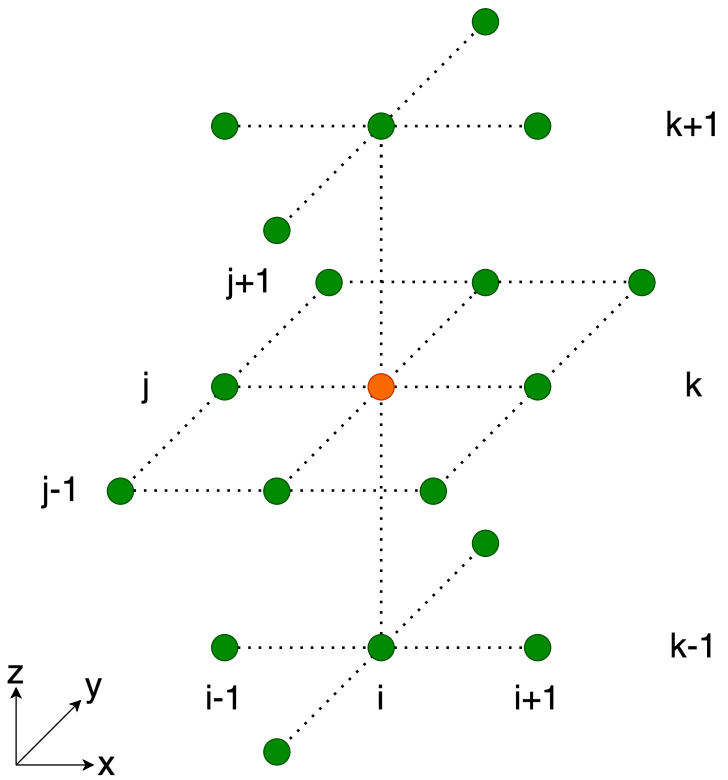

図 4.2: 姫野ベンチマークのステンシル計算

第5章　性能評価

5.1　実験環境

ここでは本研究の実験で使用した計算機システムの環境について述べる。

5.1.1　Miyabi

GH200 を搭載したシステムとして、本研究では Miyabi で実験を行った。Miyabi は東京大学情報基盤センターと筑波大学計算科学研究センターが共同運営する最先端共同 HPC 基盤施設 (JCAHPC: Joint Center for Advanced High Performance Computing) が運用するスーパーコンピュータシステムである [5]。GH200 を搭載したノード群 Miyabi-G と Intel 社製の CPU を搭載したノード群 Miyabi-C で構成されている。GH200 を搭載した国内初の汎用大規模システムである。本研究では Miyabi-G 内の 1 ノード使用した。表 5.1 に Miyabi-G の実験環境を示す。また表 5.2 に GH200 のメモリや NVlink-C2C の理論性能を示す。

5.1.2　Pegasus

既存のシステムとして、本研究では Pegasus で実験を行った。Pegasus は筑波大学計算科学研究センターが運用するスーパーコンピュータシステムである。150 個の計算ノードは、CPU として Intel Xeon Platinum 8468、GPU として NVIDIA H100 を搭載している。H100 は Hopper GPU を搭載した GPU カードである。さらに Pegasus 独自の要素として不揮発性メモリを搭載している。本研究では 1 ノードを使用した。表 5.3 に、Pegasus の実験環境を示す。また表 5.2 に Pegasus のメモリや PCIe の理論性能を示す。

5.2　実験結果

5.2.1　予備評価

4.1 に示した通常のメモリ領域へのメモリアクセス、インターコネクト性能の実験を行った。図 5.1 に CPU メモリの結果を、図 5.2 に GPU メモリの結果を、図 5.3 にインターコネクトの HtoD における結果を、図 5.4 に DtoH の結果を示す。縦軸が性能であるバンド幅、横軸が 200 回のうちの何回目の測定であるかを示している。10 回の実験の結果のグラフを重ねて表示し

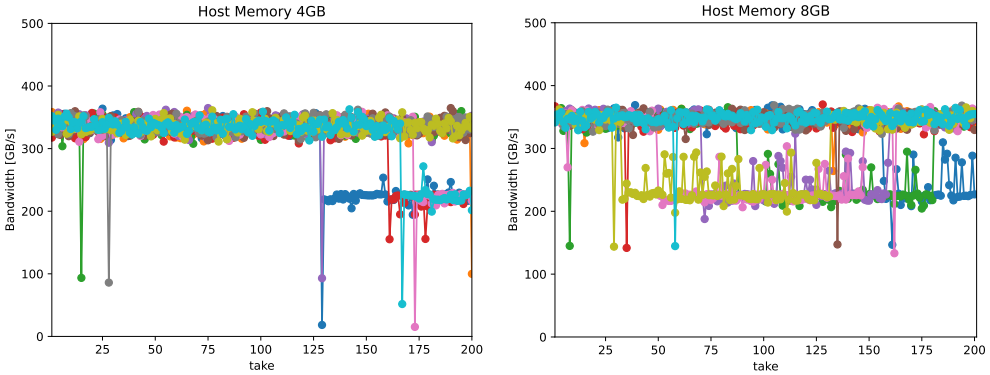

図 5.1: CPU (Host) メモリのアクセス性能

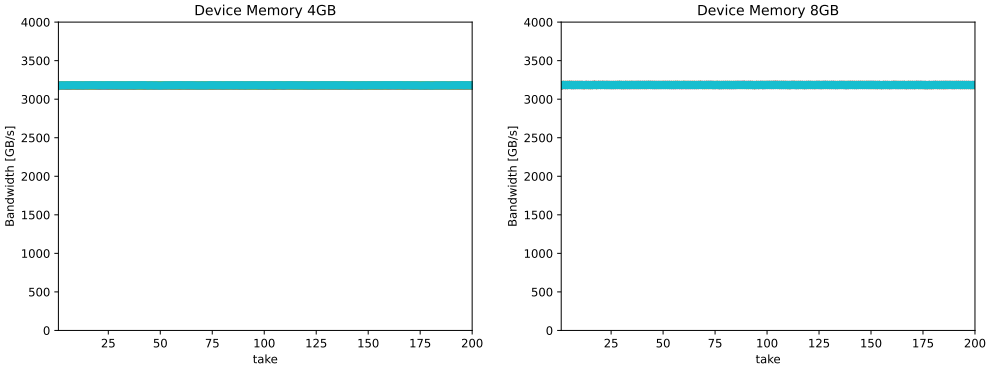

図 5.2: GPU (Device) メモリのアクセス性能

ている。グラフの左側が配列サイズが 4GB の場合、右側が配列サイズが 8GB の場合の結果である。

GPU メモリについては 3.2TB/s ほどのハンド幅が確認でき、変動は見られない。CPU メモリでは 310〜360GB/s のバンド幅が確認できるが、試行によっては 100GB/s ほどの大きな性能の変動が見られる。インターコネクトにおいては HtoD については、410GB/s ほどで安定している一方、DtoH については 300GB/s で安定している試行もあれば、60GB/s ほど性能が低下した後、その値で安定する現象が確認できる試行もある。配列のサイズによる性能の変化は確認されない。

5.2.2 SAM 上でのメモリ性能測定

8 パターンのメモリアクセス

GH200 の System-Allocated Memory において 4.2.1 の実験を行った。図 5.5 に配列のサイズが 4GB における 8 パターンの結果を、図 5.6 に配列のサイズが 8GB における結果を示す。それぞれのグラフの番号は 4.2 の 1 で説明したパターンの番号に対応している。ここで HM は

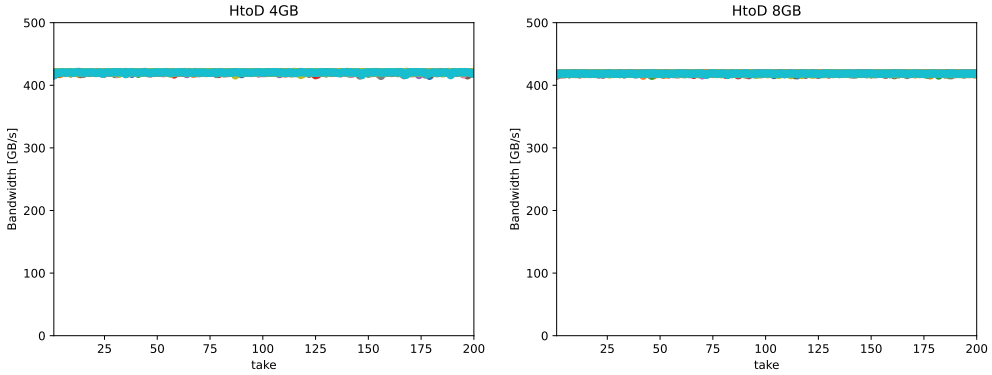

図 5.3: インターコネクト (HtoD) 性能

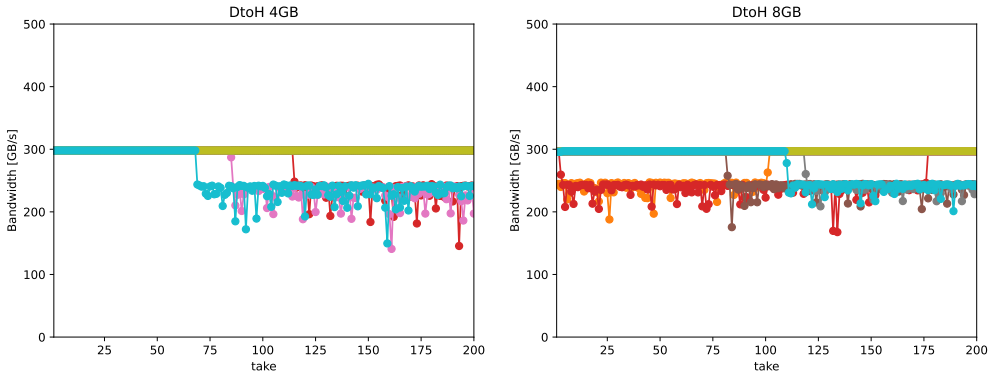

図 5.4: インターコネクト (DtoH) 性能

Host Memory つまり CPU メモリ、DM は Device Memory つまり GPU メモリのことを指す。Read がその次に示されるメモリからの読み込み、Write が次に示されるメモリへの書き込みを指す。by の後に処理を実行したプロセッサを示している。グラフの縦軸がバンド幅 [GB/s]、横軸が何回目の処理であるかを示している。10 回の実験の結果を重ねて表示している。

(1)〜(4) の CPU がアクセスする場合ではしばしば性能が大きく落ちる点が見られる。(5)〜(8) の GPU がアクセスする場合についてもまれに性能が大きく低下する点が確認できる。また試行によって性能の変動が発生する箇所が異なることも確認できる。

CPU が CPU メモリにアクセスする (1) では、5.1 の予備評価の CPU メモリのアクセス性能と同等以上の性能が確認できる。通常のアクセスとリモートアクセスを行う (2),(3) ではインターコネクトの理論性能に匹敵する性能が確認できる。(4) では、インターコネクトの DtoH の性能と同等の性能が確認できる。

(5)〜(7) の GPU がリモートアクセスを行う実験において、徐々に性能が上昇し、最終的には配列のサイズが 4GB の場合は 3TB/s、8GB の場合では 2.3TB/s や 2.5TB/s で安定することが確認できる。

(5)〜(8) の GPU がアクセスを行う場合において、配列サイズが 4GB の場合では、5.1 の予備

評価の GPU メモリの性能に近い性能が出ているが、サイズが 8GB の場合では 500〜1000GB ほどの性能低下が見られる。

5.2.3 Migration の挙動

次に 4.2.2 の実験を行った。図 5.7 に結果を示す。図 5.5, 5.6 のグラフ同様、縦軸は性能であるバンド幅、横軸が何回目の処理であるかを示している。10 回の実験のグラフを重ねている。左側が配列サイズが 4GB、右側が 8GB の場合の結果である。

GPU が実行する最初の 100 回では、5.2.2.1 の (5)〜(7) 同様徐々に性能が上昇していく現象が確認できる。その次の CPU が実行する 100 回では、(4) と同等の性能が確認できる。GPU が実行する最後の 100 回では、最初の 100 回のような現象は確認できない。配列サイズが 8GB の場合では最後の 100 回において、試行によっては最初の 100 回から 200〜500GB/s ほどの変動が確認できる。

5.2.4 姫野ベンチマーク

4.4 で示した姫野ベンチマークを実行して性能を測定した。

図 5.8 に姫野ベンチマークの測定結果を示す。縦軸が性能を示すスループットである。青色の棒グラフが通常の GPU 化バージョン、オレンジ色が UM を使用したバージョン、緑色が SAM を使用したバージョンである。

Miyabi での実行において、SAM の結果は通常バージョンと比べて 100GFLOPS ほどの性能低下が見られる。また UM と比較しても 80GFLOPS ほどの性能低下が見られる。

5.3 考察

5.3.1 SAM 上のメモリ性能

リモートアクセスが発生する場合において、従来のデータ転送の性能を測定した予備評価に近い性能が確認できる。このことから GH200 では従来の UM の弱点を克服しているといえる。多くのパターンいおいてスパイクのような性能が低下する点が見られるが、それが発生する原因については調査中である。

GPU が実行する (5)〜(7) の G3 つのパターンにおいて、性能がある試行回数まで徐々に上昇しその後一定の値を取る現象が見られる。これは Migration によって CPU メモリから GPU メモリへデータが移動していることが要因であると言える。また徐々に上昇するという結果から、Migration では配列全体を一度に移動させるのではなく分割して少しずつ移動させていると言える。CPU が実行する場合は変動はあるが、一定の性能を示していることから、GPU から CPU への Migration は発生しないと考えられる。

5.2.2.3 に示した結果より CPU が頻繁にアクセスした後再度 GPU がアクセスしても、性能が下がって徐々に上がっていくことはなく最初の 100 回の最高性能以上の値が見られる。この結果からも GPU から CPU への Migration は発生しないことがわかる。

以上より GPU で初期化を行うプログラムでリモートアクセスが多いプログラムは特に System-Allocated Memory の恩恵を受けると言える。CPU で初期化するプログラムの場合は、GPU で演算を始める前に GPU からアクセスして Migration を起こすことで、プログラムの性能を改善することができると言える。またコードの最適化を行わなくても、Migration によって性能が確保できると言える。

しかし、GPU メモリにおいて使用するメモリ領域のサイズによっては、通常の場合と比較して性能が低下することが確認できる。このため GPU がリモートアクセスをそれほど行わないプログラムでは、性能が低下する可能性が考えられる。低下する原因については調査中である。

5.3.2 姫野ベンチマーク

SAM を使用したベンチマークにおいて、通常の GPU 化よりも 100GFLOPS ほどの性能低下が見られた。低下した原因については調査中であるが、ベンチマークにおけるメモリ領域のサイズでは SAM が通常の場合と比較して性能が低下している、最初の 50 60 回の処理において、Migration 前の低いメモリ性能によって性能が低下したことが考えられる。また姫野ベンチマークが、転送が 1 回だけ行われる転送の少ないベンチマークであることも原因として考えられる。

一方で、SAM を使用することで従来の UM でも必要となるメモリ管理や転送制御を不要にすることが確認できた。4.3 でも述べたが、姫野ベンチマークのデータ構造の関係で、リスト 4.9 に示す通り従来の UM でも構造体自体のメモリ管理や転送制御が必要であった。しかしリスト 4.10 に示す通り SAM ではそれら UM でも必要であった記述を削除することできた。このことからプログラムよっては、SAM を使用することで UM よりもプログラムを簡潔にすることができ、プログラミングの生産性をより向上させることが可能であると言える。

表 5.1: Miyabi-G の環境

CPU	NVIDIA Grace CPU Arm Neoverce V2 (72 cores) ×1
GPU	NVIDIA Hopper GPU ×1
Interconnect	NVlink-C2C
OS	Rocky Linux 9
コンパイラ	NVIDIA CUDA Toolkit
CUDA バージョン	12.6

表 5.2: GH200 の理論性能

	理論性能 [GB/s]
CPU Memory LPDDR5X 120GB	512
GPU Memory HBM3	4000
NVlink-C2C HtoD	450
NVlink-C2C DtoH	450

表 5.3: Pegasus の環境

CPU	Intel Xeon Platinum 8468 (48 cores) ×1
GPU	NVIDIA H100 ×1
Interconnect	PCIe Gen 5 x16
OS	Ubuntu 22.04
コンパイラ	NVIDIA CUDA Toolkit
CUDA バージョン	12.3

表 5.4: Pegasus の理論

	実測性能 [GB/s]	理論性能 [GB/s]
CPU Memory DDR5-4800	282	
GPU Memory HBM2E	2000	
PCIe Gen 5 HtoD	64	
PCIe Gen 5 DtoH	64	

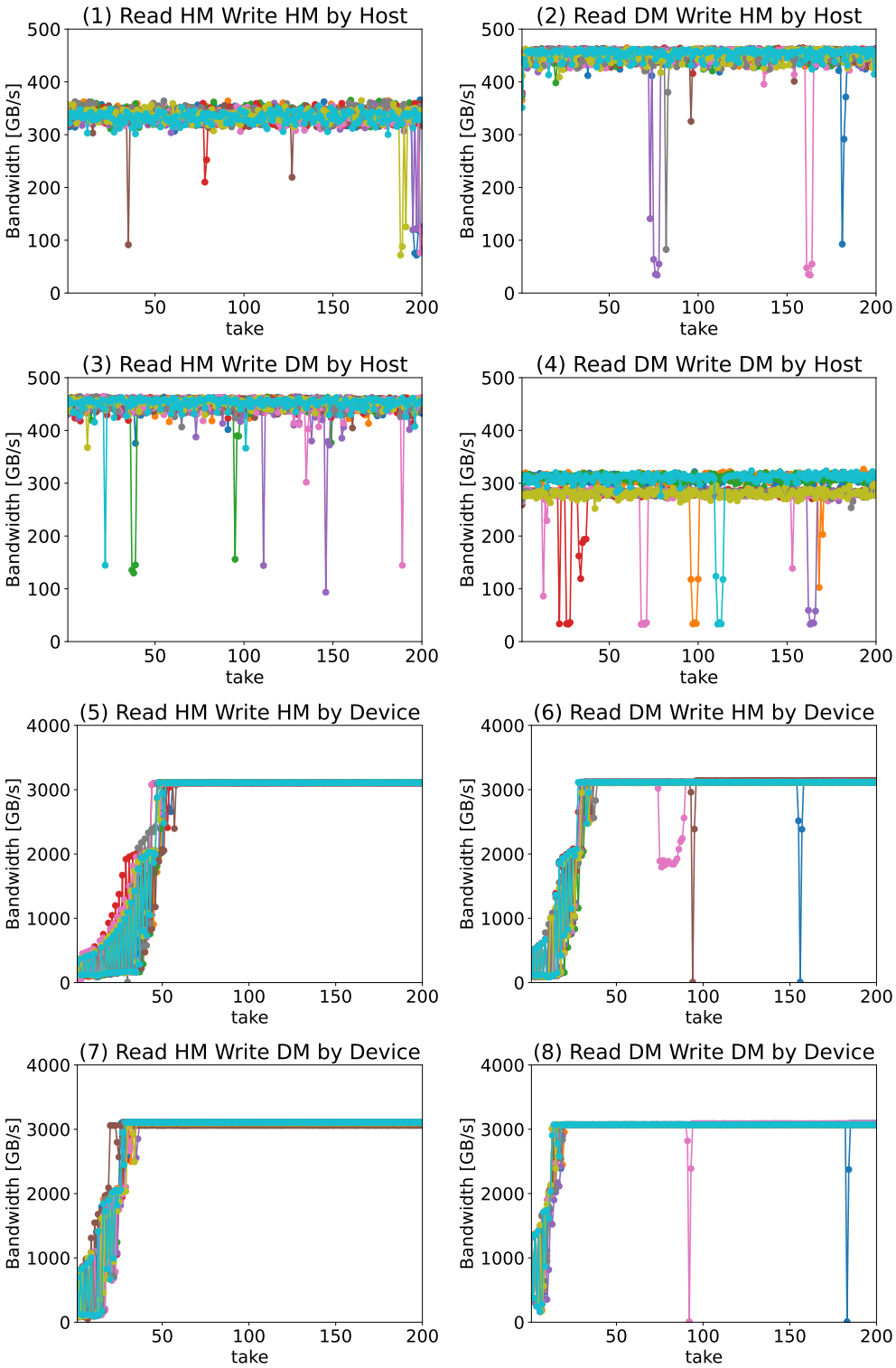

図 5.5: 配列サイズ 4GB における 8 パターンのアクセス性能

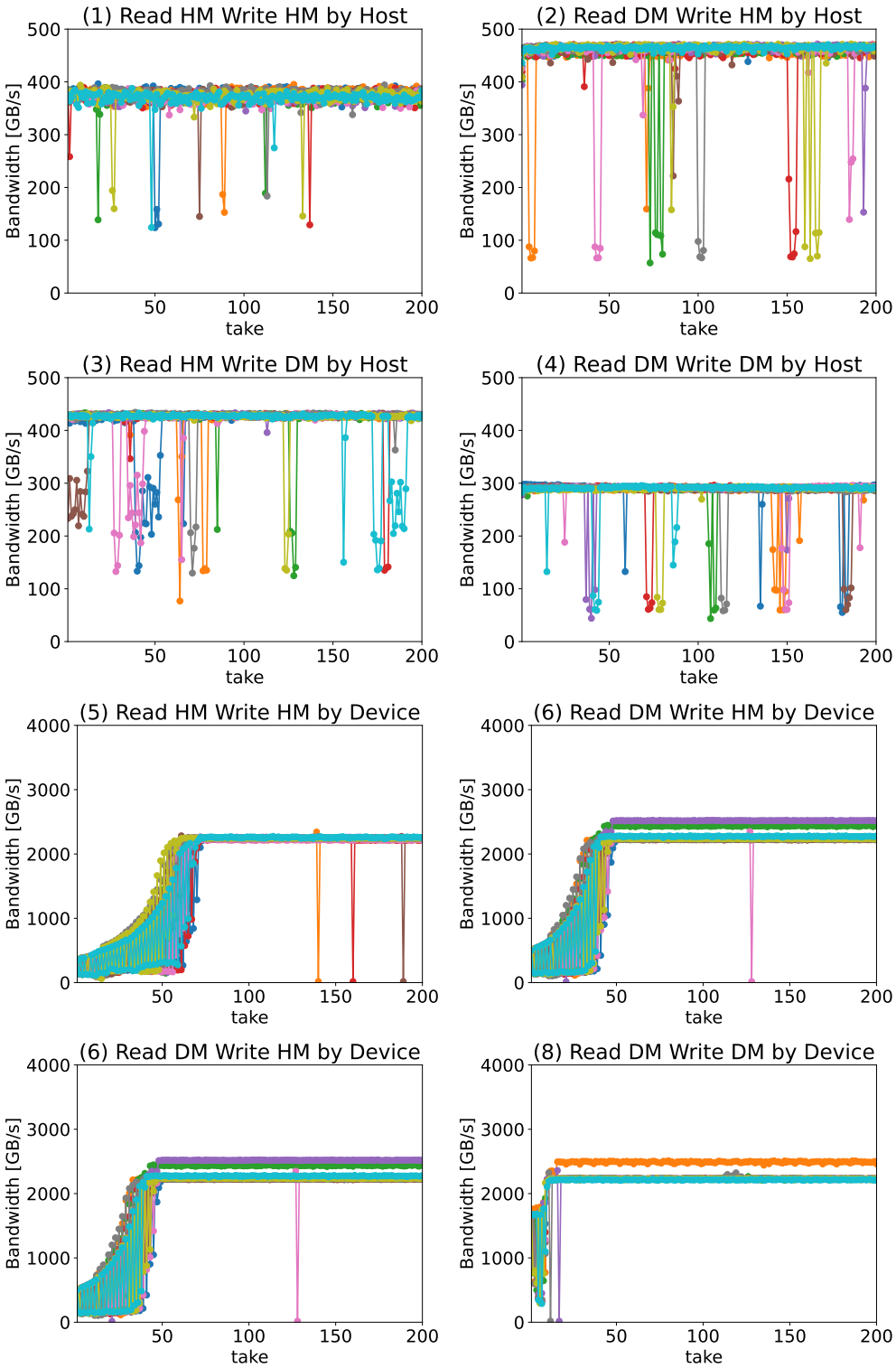

図 5.6: 配列サイズ 8GB における 8 パターンのアクセス性能

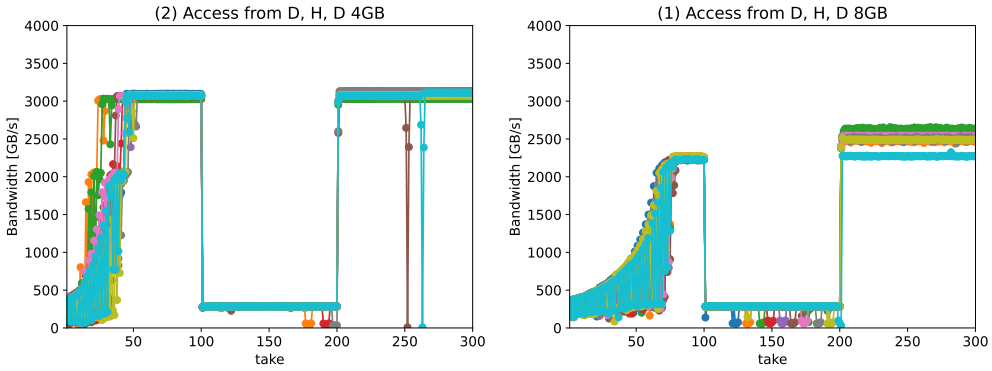

図 5.7: Device Host Device の順でメモリアクセスした場合の性能

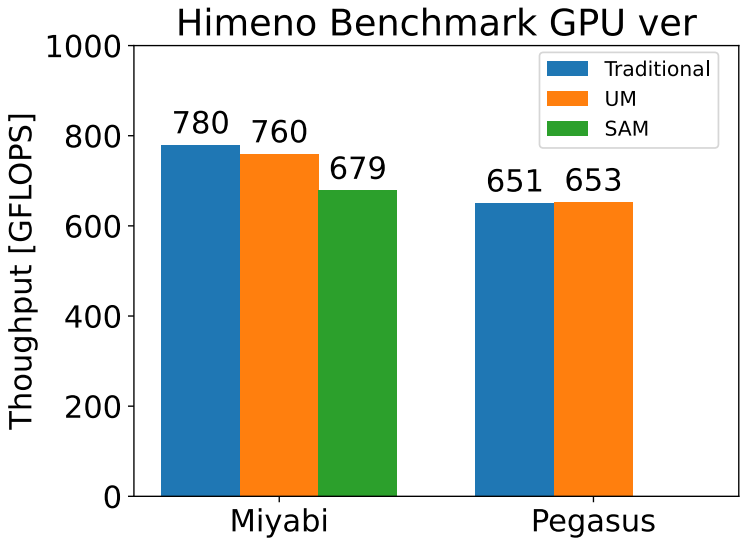

図 5.8: 姫野ベンチマークの結果

第6章　結論

本研究では、GH200 上で SAM を使用した場合のさまざまなメモリアクセスパターンに対する性能を測定し、どのようなプログラムであれば性能を引き出せるか、効果的な使用方法について考察した。結果として GH200 の SAM 上でのリモートアクセスは、従来のデータ転送に近い性能を実現しており、従来の UM の弱点を克服していることがわかった。また Migration は CPU から GPU への場合にのみ発生することを確認した。リモートアクセスが頻発する場合でも、最適な位置へデータを移動させてメモリアクセスがなるべく演算を妨げないように自動的に最適化しており、コードの最適化をしなくてもある程度の性能を確保することがわかった。さらに SAM を使用すると、従来の UM でも必要になるメモリ管理や転送制御を不要にすることが可能であり、プログラミングの生産性をより改善できることを確認した。以上より性能と生産性において GH200 の有効性が明らかになった。

今回は 1 ノードを用いて実験を行ったが、今後は複数ノードで GH200 の UM を使用した場合のプログラミングや性能についても評価していくことが課題として挙げられる。

謝辞

本研究にあたり、ご多忙の中丁寧にご指導いただきました筑波大学情報学群情報科学類教授 朴泰祐先生に感謝申し上げます。また、研究に関して助言をいただきました同助教 藤田典久先生にも感謝申し上げます。そして、さまざまな助言をいただきました東京科学大学総合研究院准教授 小林諒平先生にも感謝申し上げます。さらに日々の研究において助言や励ましをいただきました HPCS 研究室 Arch チームの先輩、同期の皆様にも感謝申し上げます。最後にお世話になっております HPCS 研究室の皆様に感謝申し上げます。

参考文献

[1] Gabin Schieffer, Jacob Wahlgren, Jie Ren, Jennifer Faj, Ivy Peng. 2024. Harnessing Integrated CPU-GPU System Memory for HPC: a first look into Grace Hopper. In ICPP '24: Proceedings of the 53rd International Conference on Parallel Processing. 199-209.

[2] NVIDIA. NVIDIA Grace Hopper Superchip Architecture whitepaper. https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper

[3] NVIDIA Docs. NVIDA Grace Performance Tuning Guide. https://docs.nvidia.com/grace-perf-tuning-guide/index.html

[4] 理化学研究所情報システム部. 姫野ベンチマーク. https://i.riken.jp/supercom/documents/himenobmt/

[5] JCAHPC. Miyabi スーパーコンピュータシステムについて - 最先端共同 HPC 基盤施設（JCAHPC）. https://www.jcahpc.jp/supercomputer/miyabi.html

[6] 筑波大学計算科学研究センター. Pegasus – Big memory supercomputers. https://www.ccs.tsukuba.ac.jp/wp-content/uploads/sites/14/Pegasus.pdf