Vol.2022-HPC-183 No.5
2022/3/17

Fortran 標準規格 do concurrent を用いた GPU オフローディング手法の評価

星野　哲也[1,a)]　河合　直聡[1]　三木　洋平[1]　塙　敏博[1]　中島　研吾[1]

概要：Fortran 2008 の標準規格として導入された `do concurrent` 構文は，適用ループが並列実行可能であることを意味する．NVIDIA 社の nvfortran コンパイラは，`do concurrent` で記述されたループを NVIDIA GPU でオフロード実行する機能を提供しており，従って専用言語や指示文を用いない標準規格の Fortran プログラムの GPU 実行が可能となっている．本稿では，`do concurrent` による GPU オフローディングの評価を目的とし，ICCG ソルバ，$\mathcal{H}$-行列・ベクトル積，3 次元拡散方程式の `do concurrent` 実装を行った．OpenACC や OpenMP 5.x と比較を行い，NVIDIA Tesla A100 上で評価を行った．その結果，`do concurrent` の制約を受けないケースでは OpenACC や OpenMP 5.x と遜色ない性能が得られたものの，`do concurrent` の制約である縮約演算が必要なケースや明示的に CPU-GPU 間のデータ移動を行いたいケースにおいて，大きな性能ギャップが生じることや，制約を避けるための実装コストが大きいことがわかった．

1. はじめに

消費電力性能に優れる GPU を搭載したスパコンは，国内外で広く運用されている．GPU を利用するためには種々のプログラミング手法があり，CUDA, HIP, OpenCL, OpenACC, OpenMP(>4.0) が代表的である．これらに加えて新たに，C++や Fortran の標準言語規格を用いた GPU オフローディング手法が NVIDIA 社より提供されている．本稿の評価対象である `do concurrent` 構文は Fortran 2008 で導入され，`do concurrent` で記述されたループが並列実行可能であることを意味し，NVIDIA 社が提供する nvfortran コンパイラは当該ループを自動的に GPU にオフロードする [1]．標準言語規格を用いることの利点は，一部のコンパイラでしかサポートされていない CUDA, HIP, OpenCL, OpenACC, OpenMP 4.x とは対照的に，CPU・GPU 問わず多くの計算環境での長期サポートが期待できることである．しかし，標準言語規格を用いた GPU オフローディングに関する研究はほとんどなく，既存のプログラムからの移植コストや得られる性能は明らかではない．特に，`do concurrent` 構文は既存の並列化手法と比べて多くの制約 (pure function しか呼び出せない，atomic やリダクションなどの縮約演算が出来ない，CPU-GPU 間の明示的なデータ移動ができないなど) を持ち，それらの制約が移植コストや性能にどの程度の影響をもたらすのか評価する必要がある．

本稿では `do concurrent` 構文を用いて，不完全コレスキー分解前処理付き共役勾配法 (ICCG)，$\mathcal{H}$-行列・ベクトル積 (HiMV)，3 次元拡散方程式 (Stencil) の 3 つのアプリケーションを GPU 向けに移植する．ICCG はリダクション演算を，HiMV はリダクション及び atomic 演算を，Stencil は MPI 通信の際に CPU-GPU 間データ移動をそれぞれ必要とし，`do concurrent` を使う上で性能上の懸念がある．性能や移植コストを比較するために，OpenACC と OpenMP 5.x による GPU 向けの実装を用意し，NVIDIA A100 GPU において評価する．

2. Fortran do concurrent による GPU オフローディング

2.1 Fortran do concurrent

Fortran 2008 で導入された `do concurrent` 構文は，適用されたループに相互依存性がなく，並列実行可能であることを意味する．**図 1** は，`do concurrent` 構文の利用例である．1 行目のループが `do concurrent` 構文により並列実行可能であることが明示されているため，コンパイラによりスレッド・SIMD 並列化が適用されることが期待される．

ただし `do concurrent` にはいくつか制限があり，特に関数呼び出しに関する制限 (pure function 以外の関数呼び

[1] 東京大学 情報基盤センター
Information Technology Center, Kashiwa 277-0882, Japan
[a)] hoshino@cc.u-tokyo.ac.jp

```
1 DO CONCURRENT (i = 1:n)
2    y(i) = y(i) + a * x(i)
3 END DO
```

図 1 do concurrent により記述した AXPY コード．変数 i, n は整数，a は実数であり，x, y は実数の配列である．

出しはできない) は，アプリケーションプログラムの実装コストを増加させる要因となり得る．また do concurrent 構文における縮約演算は最新の規格である Fortran 2018 においてサポートされておらず，Fortran 202X での導入が検討されている段階であり，縮約演算を必要とするアプリケーションでは性能への影響が懸念される．

2.2 GPU オフローディング

2.2.1 do concurrent

do concurrent 構文の GPU オフローディングは NVIDIA 社の nvfortran コンパイラにおいて提供されており，NVIDIA 社製の GPU でのみ利用できる．GPU 実行時に必要なデータの CPU-GPU 間転送を明示的に行うことはできず，Unified Memory [2] 機能を前提として暗黙的に行われる．図 1 の場合，2 行目の配列変数 x, y へのアクセスが GPU から試みられた際に，x, y が GPU のメモリ上に無い，または CPU のメモリ上の x, y より古い場合，CPU から GPU へのデータ転送が発生する．4 行目以降，x, y が CPU 上で更新されない限りは CPU から GPU へのデータ転送は生じず，CPU から x, y へのアクセスが試みられた際に GPU から CPU へのデータ転送が発生する．スカラ変数 n, a は Unified Memory 機能を用いずに GPU カーネル起動時に予め転送されるが，暗黙的にデータ転送が行われることに関しては配列変数と同様である．なお，Unified Memory 機能による CPU-GPU 間のデータマイグレーションはページ単位で行われ，かつページサイズの変更はできないため，本来転送すべきデータより大きなデータ転送が発生する可能性がある．

また GPU が起動するスレッド数は指定できず，コンパイラにより暗黙的に決定される．なお nvfortran は独自拡張として，規格上はサポートされていないリダクションをサポートしており，リダクションが必要なループに do concurrent を適用することで GPU 上で並列実行可能である．

2.2.2 OpenACC

OpenACC は GPU を含む演算アクセラレータ向けの指示文ベースのプログラミング規格であり，C/C++, Fortran プログラムに!$acc で始まる指示行を追加することで，指定されたループをアクセラレータ上でオフロード実行することができる．主なオフロード対象は演算アクセラレータであるが，CPU でも実行可能な仕様となっている．**図 2** は図 1 の OpenACC 実装版であり，同様の動作が期待される．配列変数 x, y の CPU-GPU 間データ転送は，1, 8 行目の data 指示文によって明示的に行われる．ただし，nvfortran により NVIDIA 社製の GPU を利用する際には，コンパイラのフラグを切り替えることで Unified Memory 機能を有効化することもでき，その場合には 1, 8 行目の data 指示文は無視されるため，記述は必須ではない．

```
1 !$acc data copyin(x), copy(y)
2 !$acc kernels
3 !$acc loop independent
4 do i = 1, n
5    y(i) = y(i) + a * x(i)
6 end do
7 !$acc end kernels
8 !$acc end data
```

図 2 図 1 とおよそ等価な AXPY の OpenACC 実装．

GPU が起動するスレッド数を 3 行目の loop 指示文にヒントとして与え，最適化することが可能である．縮約演算は loop 指示文に reduction 節を付与することで実現できる．2 行目の kernels 指示文に async 節を付与することで，別の kernels タスクとのタスク並列実行も可能である．

2.2.3 OpenMP

従来の OpenMP は共有メモリ環境のマルチコア CPU を対象とした指示文ベースのプログラミング規格であったが，OpenMP 4.x における拡張で追加された target 指示文により，アクセラレータを対象とできるようになった．また，OpenMP 5.x で追加された loop 指示文は CPU, GPU の両者を対象とすることを念頭にした指示文であり，do concurrent 構文に近い理念を持つ．

図 3 は図 1 を OpenMP target, loop 指示文を利用して実装しなおしたものであり，同様の動作が期待される．配列変数 x, y の CPU-GPU 間データ転送は，1, 8 行目の map 指示文によって明示的に行われる．また OpenACC 同様，nvfortran により NVIDIA 社製の GPU を利用する際には，コンパイラのフラグを切り替えることで Unified Memory 機能を有効化することができ，その場合には map 指示文の記述は必須ではない．

OpenACC と異なる点として，loop 指示文が付与されたループの並列化粒度 (teams, parallel, thread レベル) の指定はできるものの，GPU が起動するスレッド数の最適化はできず，これを行う場合には loop 指示文の代わりに teams distribute, parallel for, simd などの指示文を用いる必要がある．また target タスクのタスク並列実行を行う場合は，OpenMP のタスク並列指示文と組み合わせて使う必要がある．縮約演算は OpenACC と同様に loop 指示文に reduction 節を付与することで実現できる．

```
!$omp target data map(to:x) map(tofrom:y)
!$omp target
!$omp loop
do i = 1, n
   y(i) = y(i) + a * x(i)
end do
!$omp end target
!$omp end target data
```

図 3 図 1 とおよそ等価な AXPY の OpenMP 実装.

3. 評価手法

3.1 実装手法

本稿では do concurrent によるアプリケーションの移植を，簡便さと性能の観点から評価する．比較評価のために，以下の実装を用意する．

- **stdpar1**: 規格に準拠した do concurrent 実装.
- **stdpar2**: do concurrent をリダクションが必要なループにも適用した実装（規格準拠ではない）.
- **acc1**: GPU における並列実行のために必要な最低限の指示文を適用したナイーブな OpenACC 実装．CPU-GPU 間のデータ転送は Unified Memory により暗黙的に行う.
- **acc2**: スレッド数調整や async 節の利用により指示文を最適化した OpenACC 実装．CPU-GPU 間のデータ転送は data 指示文により明示的に行う.
- **ompgpu1**: target 及び loop 指示文を用いたナイーブな OpenMP 実装．CPU-GPU 間のデータ転送は Unified Memory により暗黙的に行う.
- **ompgpu2**: 指示文を最適化した OpenMP target, loop による実装．CPU-GPU 間のデータ転送は target map 指示文により明示的に行う.

規格に準拠している stdpar1 は多くの環境で実行可能であるが，リダクションを CPU 1 コアで実行することとなり，特に GPU を利用する場合は CPU-GPU 間のデータ移動を伴うために性能への影響が大きいと考えられる．これに対し stdpar2 は規格準拠ではないが，Fortran 202X においてリダクションがサポートされた際の性能を推定する上で必要な実装である.

指示文の最適化を行わないナイーブな実装である acc1, ompgpu1 においては，リダクションを並列実行できる点が stdpar1 との主な差である．GPU 実行時の性能は acc1, ompgpu1, stdpar2 の三者でおよそ同程度になることが期待される．また，指示文の最適化により生じる性能差を評価する狙いで，acc2, ompgpu2 の実装を用意し比較する.

上記の実装手法の一部，または全てを以下のアプリケーションに対して適用する．各アプリケーションの詳細については後述する.

- **ICCG**: 不完全コレスキー分解前処理付き共役勾配法 (ICCG 法) により連立一次方程式を解くソルバー．自明な並列性を持つループとリダクション演算を必要とするループにより構成され，並列化対象ループ内からの関数呼び出しはない．MPI による並列化は行わない.
- **HiMV**: 行列近似法のひとつである階層型行列 ($\mathcal{H}$-行列) 法において生成される $\mathcal{H}$-行列とベクトルの積を計算するカーネル．解ベクトルへの縮約演算にリダクションに加え atomic 演算を必要とする．並列化対象ループ内からの関数呼び出しはない．$\mathcal{H}$-行列は大きさがまちまちの低ランク行列と小密行列からなるため，高速化にはスレッド数の調整が必須となる．MPI により並列化されているが，1 プロセスにより評価する.
- **Stencil**: 三次元拡散方程式のカーネル．自明な並列性を持つ三重ループにより構成される．並列化対象ループ内からの関数呼び出しはない．MPI により並列化され，通信タスクと演算タスクのタスク並列実行により，袖領域の通信時間を隠蔽することが可能である.

3.2 評価環境

本稿では評価環境として，東京大学情報基盤センターが運用する Wisteria/BDEC-01 [3] の Aquarius ノードを用いる．Aquarius ノードには Intel IceLake 2 ソケットと NVIDIA A100 GPU 8 基を搭載している．ノードの詳細を**表 1** に示す．またコンパイラには，NVIDIA 社が提供する nvfortran コンパイラを用いる．コンパイラオプションには，共通して-O3, -gpu=cc80 を付与し，stdpar1, stdpar2 には-stdpar=gpu，ompgpu1 には-mp=gpu -gpu=managed，ompgpu2 には-mp=gpu，acc1 には-acc=gpu -gpu=managed，acc2 には-acc=gpu をそれぞれ付与した.

ICCG と HiMV は A100 1 基により評価を行い，Stencil は最大で Aquarius の 2 ノード 16 GPU を用いて評価する．Aquarius ノード内の GPU-GPU 間は高い帯域幅を持つ NVLink によって接続されているが，CPU-GPU 間は PCI Express Gen4 により接続されている．ノード間ネットワークは InfiniBand HDR により接続されている．MPI ライブラリには CUDA aware MPI に対応した OpenMPI 4.1.1 を用いる.

3.3 do concurrent の評価手法

do concurrent の評価は，移植コストと性能の 2 つの観点から行う．移植コストの評価は，(1) 指示文及び do concurrent の挿入行数による定量的な比較と (2) 指示文または do concurrent を使う上で必要なプログラムの書き換え内容の 2 項目を考える.

指示文の行数の数え方については，利用した指示子の数

表 1 評価環境

ノード群	Aquarius	
	CPU	GPU
プロセッサ	Intel Xeon Platinum 8360Y (Ice Lake) ×2	NVIDIA A100 ×8
コア数 (単体)	36	108 SMs
ピーク性能 (単体)	2.7648 TFlops	19.5 TFlops
メモリ帯域幅 (単体)	204.8 GB/sec	1,555 GB/sec

だけ行数としてカウントする．例えば図 2 は，単純に挿入した指示行の行数を数えると 5 行であるが，1 行目は 3 行に分けて!$acc data (改行) !$acc& copyin(x) (改行) !$acc& copy(y) と書くこともでき，プログラマの作業コストとして数えるのであれば，1 行目は 3 行分と見なすのが妥当である．この要領で指示行を数えなおすと，1 行目が 3 行分，3 行目が 2 行分となり，図 2 は 8 行分の移植コストがあると見なせる．本稿ではこの規則に基づき (1) の挿入行数の評価を行う．(2) については，例えばアルゴリズムの変更による変更行数と，単に関数をインライン展開する場合の変更行数は作業コストとして等価と見なせないため，どのような変更をしなくてはならなかったかの記述に留める．

また性能評価は Flops 値の比較によって行う．15 回実行した際の中央値をその実装の性能と見なす．

4. 性能評価

4.1 不完全コレスキー分解前処理付き共役勾配法 (ICCG)

本稿では前処理手法として対称行列向けに広く使用されている不完全コレスキー分解 (Incomplete Cholesky Factorization, IC) を使用した，前処理付き共役勾配法 (Preconditioned Conjugate Gradient Method) である，ICCG 法を用いる．ここでは，P3D アプリケーション [4] において使用されている ICCG 法を取り上げる．ICCG 法のアルゴリズムを Algorithm 1 に示す．ICCG 法の計算は主に IC 前処理，疎行列・ベクトル積，ベクトルの内積によって構成される．IC 前処理のデータの依存性は Reverse Cuthill-Mckee と Cyclic-Multicoloring(CM-RCM 法) によって解消されており，`do concurrent` によって並列化可能な依存性のないループにより構成される．反復ループ中にリダクションを必要とする内積計算が 3 度出現するため，`do concurrent` の仕様に準拠するとこの部分は並列化できないこととなる．ICCG 法は疎行列を扱う前処理と疎行列・ベクトル積が支配的となるため，内積計算の性能への影響は小さいが，stdpar1 の実装ではリダクションを伴うループを並列化できず CPU 1 コアでの実行となり，特に GPU 利用時には GPU から CPU へのベクトルの転送が発生し，性能に大きく影響すると考えられる．なお，前処理及び疎行列・ベクトル積も行列からベクトルへの縮約演算であるが，今回対象とする問題は疎行列の各行の非ゼロ要素数が 7 以下でありループ長が短いため，行方向の並列化は行わず逐次実行し，列方向ループのみ並列化する．

対象問題は $N_x \times N_y \times N_z$ のサイズの三次元領域に対して以下のポアソン方程式を有限体積法に基づき解くものである：

$$\Delta\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} + \frac{\partial^2\phi}{\partial z^2} = f \tag{1}$$

$$\phi = 0 \,@\, z = z_{\max} \tag{2}$$

MPI による並列化は行わず，全てシングルプロセスにより評価を行う．評価対象問題のサイズは $N_x = N_y = N_z = 256$ とし，計算精度は倍精度である．IC 前処理は CM-RCM 法により 10 色にカラーリングし，相対残差が 10^{-8} を下回った時点で反復計算を停止する．行列の格納形式としては Sell-C-σ [5], [6] を採用している．

表 2 に，各 ICCG 実装の移植コストを示す．最も行数が少ないのは 50 行の stdpar1 であるが，これに加えて並列領域からリダクションを追い出すためのコード変更が必要となっている．次点で，規格準拠ではない stdpar2 の行数が少ない．移植の簡便さは CPU-GPU 間のデータ転送が Unified Memory により暗黙的に行われることによる影響が大きく，ompgpu1, acc1 もそれぞれ ompgpu2, acc2 と比較すると必要行数が少なくなっている．

A100 による評価結果を**図 4** に示す．規格に準拠した stdpar1 は，他の実装と比較して性能が著しく劣っている．これはリダクションを CPU で実行するための CPU-GPU 間データ転送が原因であり，やはりリダクションに関する制約の影響は大きいことがわかった．一方で，規格準拠ではないものの，nvfortran コンパイラの拡張によりリダクションを GPU 実行する stdpar2 は，ompgpu1, acc1 と比較して遜色ない性能が得られている．最速であったのは acc2 であるが，この性能向上は async 節によってもたらされたものである．素朴な OpenACC 実装では，CPU は起動した GPU カーネルの終了を待つため，次の GPU カーネルを起動するまでに空き時間ができる．これは `do concurrent` や OpenMP でも同様であるが，OpenACC の場合は async 節を用いることで空き時間を削減することができる．また nvfortran では普通，GPU のスレッドブロック当たりのスレッド数として 128 が採用されるが，リダクション計算部 (Algorithm 1 の 6, 14, 17 行目) ではより大きいスレッド数を採用した方が有利であったため，OpenACC でのみスレッド数の変更を行っている．これらの機能の有無が，性能ギャップを引き起こす原因となっている．

4.2 $\mathcal{H}$-行列・ベクトル積 (HiMV)

本稿では，$\mathcal{H}$-行列向けのライブラリとして提供されている $\mathcal{H}$ACApK [7] を用いて評価を行う．$\mathcal{H}$-行列法は境界要素法などに現れる係数の密行列を近似する手法であり，$\mathcal{H}$ACApK は対象問題の係数行列要素を計算するユーザ関

Algorithm 1 ICCG 法のアルゴリズム．ここで A, M は疎行列，r, b, x, z, p, q はベクトル，k, err, ϵ, ρ, α, β はスカラーである．

```
1: Compute r^0 := b − Ax^0
2: k := 0; err := ∞
3: while err > ε do
4:     k := k + 1
5:     solve Mz^{k−1} := r^{k−1} #IC 前処理
6:     ρ_{k−1} := r^{i−1} z^{i−1} #リダクション
7:     if k = 1 then
8:         p^1 = z^0
9:     else
10:        β_{k−1} := ρ_{k−1}/ρ_{k−2}
11:        p^k := z^{k−1} + β_{i−1} p^{i−1}
12:    end if
13:    q^k := Ap^k #疎行列・ベクトル積
14:    α_k := ρ_{k−1}/(p^k q^k) #リダクション
15:    x^k := x^{k−1} + α_k p^k
16:    r^k := r^{k−1} − α_k q^k
17:    err := sqrt(||r^k||^2/||b||^2) #リダクション
18: end while
```

表 2 各 ICCG 実装の移植コスト

	指示文等行数	その他のコード変更内容
stdpar1	50	並列領域外へのリダクションの追い出し
stdpar2	54	特になし
ompgpu1	82	
ompgpu2	97	
acc1	98	
acc2	137	

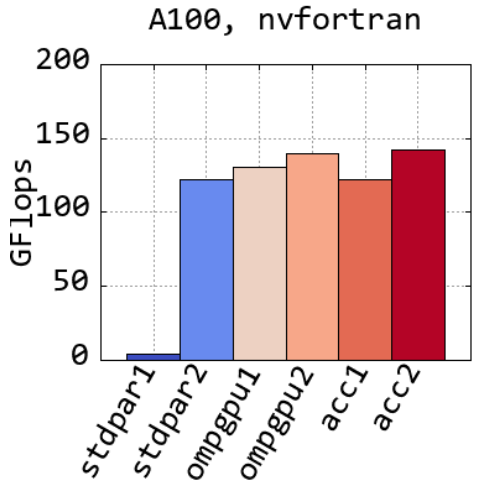

図 4 ICCG の GPU 実行結果．

数を元に，MPI+OpenMP によって並列化された $\mathcal{H}$-行列生成ルーチン及び $\mathcal{H}$-行列・ベクトル積 (HiMV) ルーチンを提供する．

$\mathcal{H}$-行列 A の構造と，Adaptive Cross Approximation (ACA) [8] によって近似された部分行列構造を**図 5** に示す．また $\mathcal{H}$-行列 A を用いて $y = Ax$ を計算する HiMV のアルゴリズムを Algorithm 2 に，HiMV の概念図を**図 6** に示す．ここで y および x はベクトルである．Algorithm 2 に示される通り，4 行目の s のループは並列化可能であり，すなわち各部分行列の計算は並列実行可能である．ただし，図 6 からわかるように，解ベクトル y に各部分行列の計算結果を縮約する際には書き込み競合が起こり，素朴には atomic_add が必要となる．また，GPU での並列実行を見据えた場合，十分な並列性を確保するために部分行列内の計算も並列化すべきであるが，部分行列内の計算は密行列・ベクトル積 (dgemv) であり，リダクションを必要とする．故に，stdpar1 の実装においては，部分行列内の並列化は行わず，atomic 演算を回避するような実装を行った．規格準拠ではない stdpar2 では部分行列内のリダクションが必要なループも `do concurrent` ループとした．OpenMP 実装である ompgpu1, ompgpu2 では atomic 指示文により atomic 演算の利用を試みたが，結果が不正となってしまったため，stdpar1, stdpar2 同様，atomic 演算を回避する実装を行った．OpenACC 実装である acc1, acc2 では部分行列内のリダクションが必要なループの並列化を行った上で，atomic 指示文により atomic 演算を行う，加えて acc2 では，スレッド数の調整を行っている．また OpenMP の target data 指示文，OpenACC の data 指示文による構造体の扱いは煩雑であり，構造体の扱い自体を回避してしまった方がメリットが多いため，Unified Memory に頼らない ompgpu2 と acc2 では構造体を回避している．構造体を難なく扱える点でも Unified Memory は簡便さの向上に貢献している．

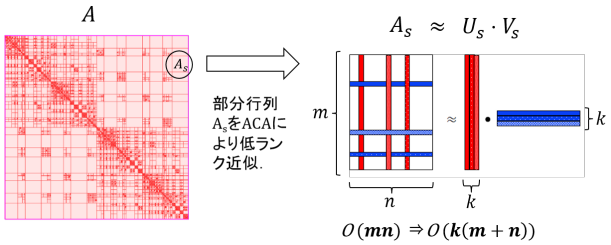

図 5 $\mathcal{H}$-行列 A の構造と部分行列 A_s の ACA による近似．$\mathcal{H}$-行列構造において，薄紅に塗られた部分行列は低ランク近似行列，濃紅に塗られた部分行列が密行列である．$m \times n$ の部分行列 A_s は選出された k 本のベクトルにより代表され，$m \times k$ の密行列 U_s と $k \times n$ の密行列 V_s の積として近似される．

評価には，境界要素法 (Boundary Element Method, BEM) 用のソフトウェアである ppOpen-APPL/BEM ver.0.5.0 [9], [10] に含まれる $\mathcal{H}$ACApK ライブラリ利用版のリファレンス実装を用いる．図 5 の $\mathcal{H}$-行列構造は，リファレンス実装における静電場解析で得られる例であり，人型の構造体 9 体を 3×3 の配置で並べた際のものである．本稿の HiMV の評価ではこの $\mathcal{H}$-行列を用いる．計算精度は倍精度であり，行列サイズは $176,976^2$ である．$\mathcal{H}$-行列

Algorithm 2 $y = Ax$ を計算する HiMV の OpenMP による並列アルゴリズム．ここで l はスレッド番号，y_l はスレッドローカルな解ベクトル，s は部分行列の添え字，S は添え字集合，m_s, n_s, k_s はそれぞれ部分行列 A_s における縦，横のサイズ，近似時のベクトル本数を表す．

```
1: !$omp parallel private(l, y_l, t, s)
2: l := omp_get_thread_num()
3: y_l := 0
4: for all s ∈ S_l do
5:    if A_s is dense sub-matrix then
6:       y_l|_{m_s} := y_l|_{m_s} + A_s|_{m_s×n_s} · x|_{n_s}  #リダクション
7:    else
8:       t|_{k_s} := V_s|_{k_s×n_s} · x|_{n_s}  #リダクション
9:       y_l|_{m_s} := y_l|_{m_s} + U_s|_{m_s×k_s} · t|_{k_s}  #短いリダクション
10:   end if
11: end for
12: nt := omp_get_num_threads()
13: y := y + Σ_l^{nt} y_l  #atomic 演算
14: !$omp end parallel
```

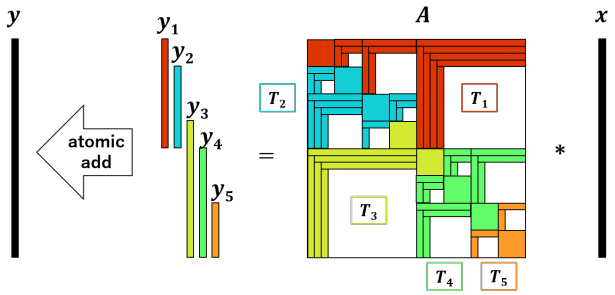

図 6 $y = Ax$ を計算する HiMV の OpenMP 並列化 (5 スレッド) の概念図．T_l は OpenMP のスレッド番号を表す．解ベクトル y への縮約演算時に書き込み競合が発生するため，素朴に実装すると atomic_add が必要となる．

自体の近似精度は 10^{-8} とし，その際の部分行列の組成が低ランク行列 $\times 139,588$，密行列 $\times 288,876$ である．シングルプロセス実行であり，評価区間内に CPU-GPU 間のデータ転送時間は含まれない．

表 3 に，各 HiMV 実装の移植コストを示す．表のとおり，指示文等の挿入行数は stdpar1, stdpar2 が少なく優れているが，atomic 演算をサポートしていないために，これを回避するようアルゴリズムとデータ構造を工夫する必要があった．具体的には，各部分行列から生じる行列・ベクトル積の結果ベクトル全てを，疎行列計算でよく使われる CRS のようなフォーマットで束ね，書き込み競合を回避しつつ並列に縮約演算ができるようにした．このようなアルゴリズムの変更は移植コストを著しく増大させるため，好ましくない．

性能評価結果を**図 7** に示す．リダクションを行えない stdpar1 は，ICCG 同様に他の実装と比較して性能が大きく劣っているが，リダクション可能な stdpar2 は，バグの影響で atomic を利用できなかった ompgpu1 とほとんど同程度の性能であった．Atomic 演算を利用した acc1 は，簡便な実装でありながら，A100 GPU が atomic_add をハードウェアサポートしている恩恵もあり，ompgpu1 より高性能であった．ompgpu2 と acc2 で性能が向上しているのは，構造体を回避したことによる恩恵が大きい．$\mathcal{H}-$ 行列の構成上，acc2 におけるスレッド数の調整は効果があると見込んでいたが，スレッド数 (32, 64, 128, 256) に対して性能値が (88.8, 124.6, 125.1, 75.1) GFlops となったため，たまたまデフォルトの 128 スレッドが最適値であり，この点では性能差が生じなかった．

表 3 各 HiMV 実装の移植コスト

	指示文等行数	その他のコード変更内容
stdpar1	12	atomic 回避のためのアルゴリズム変更
stdpar2	18	
ompgpu1	26	atomic 使用時のバグ回避のためのアルゴリズム変更
ompgpu2	39	同上．加えて構造体回避
acc1	28	特になし
acc2	38	構造体回避

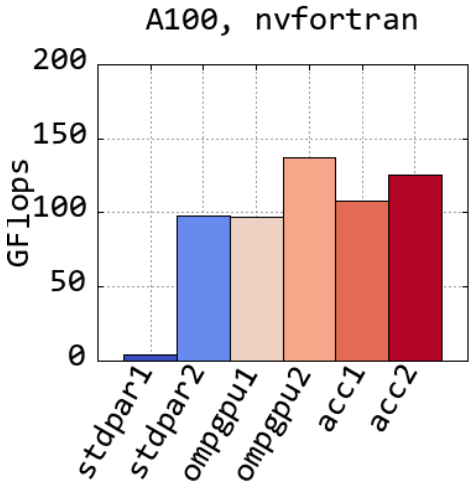

図 7 HiMV の GPU 実行結果．

なお既存研究 [11] において，部分行列のソートや密行列の合体により GPU での高速化が見込まれることがわかっているが，今回の `do concurrent` の評価という趣旨と外れるため，素朴な実装により比較を行った．

4.3 3 次元拡散方程式 (Stencil)

Stencil は科学技術計算において頻出するステンシル計算の典型的なパターンであり，高性能計算分野においてベンチマークプログラムとしてよく利用される [12]．**図 8** に 3 次元拡散方程式の `do concurrent` による実装を示す．i, j, k の三重ループはいずれも並列化可能であり，かつ一重化 (collapse) することができるため，図 8 のように

do concurrent により置き換えることができ，この場合は i ループが最内側，k ループが最外ループとなる．図 8 のカーネルは時間ステップの進行に伴い繰り返し実行されるが，MPI により並列化を行う場合には隣接プロセスの持つデータの一部を毎ステップ通信する必要がある．通信量は MPI による領域分割手法により異なるが，本稿では**図 9** のように，最も単純な z 軸 (k ループ) の一次元分割を用い，MPI_Isend, MPI_Irecv により袖領域の交換を行う．また MPI により交換すべき袖領域部分の計算と内点の計算を分離し別カーネルとし，袖領域計算 → 袖領域通信 → 内点の計算の順で実行することで，袖領域通信時間の隠蔽を行う．

表 4 に各実装で掛かった移植コストを示す．Stencil は自明な並列性を持つループのみからなるため，必要な変更行数は小さく，特別なコード改変も必要ない．ompgpu2, acc2 で必要行数が増えているのは，データ転送関連の指示文によるものである．Stencil においても，Unified Memory は移植コストを大きく減らしていると言え，特に stdpar1 ではわずかなコード改変で済んでいる．

図 10 に A100 GPU による評価結果を示す．計算精度は倍精度であり，問題サイズは $N_x = N_y = N_z = 256$ とし，1 GPU あたりの担当領域が $n_x = n_y = 256$, $n_z = 256$/GPU 数となるストロングスケーリングである．リダクションの必要はないため，stdpar2 の実装は用意しない．また，stdpar1 では MPI プロセス番号と使用する GPU デバイス番号をプログラム中で対応付ける手法が提供されていないため，環境変数 CUDA_VISIBLE_DEVICES 及び OMPI_COMM_WORLD_LOCAL_RANK を用いて，プログラム実行時にプロセス番号と GPU デバイス番号の対応付けを行う．OpenMP，OpenACC はプログラム中で対応付けを行うことができるが，行数評価の公平のため同様に環境変数を用いる．

図 10 の結果から，シングル GPU 実行時の性能は acc2 が 954.1 GFlops で若干高かったものの，acc1, ompgpu1, ompgpu2, stdpar1 はそれぞれ 864.5, 866.1, 881,3, 868.4 GFlops で大きな差はなかった．しかしマルチ GPU 実行において，stdpar1, ompgpu1, acc1 の性能が著しく低下していることがわかる．これらの実装はいずれも，CPU-GPU 間のデータ転送を Unified Memory 機能に頼ったものである．MPI_Isend, MPI_Irecv は CPU で実行されるため，その際に発生する CPU-GPU 間のデータマイグレーションのオーバヘッドが性能低下の要因となる．2.2 節で述べた通り，Unified Memory のデータ転送はページ単位で行われるため，本来転送すべき袖領域より大きなデータ転送が行われていると考えられる．また本稿の計算環境における CPU-GPU 間の接続は PCI Express Gen4 であり，高速な NVLink による接続ではないため，NVLink 環境における Unified Memory の利用を行った先行研究 [13]

```
1  DO CONCURRENT(i = 1:nx,j = 1:ny,k = 1:nz)
2     w = -1; e = 1; n = -1; s = 1; b = -1; t = 1;
3     if(i == 1)  w = 0; if(i == nx) e = 0
4     if(j == 1)  n = 0; if(j == ny) s = 0
5     if(k == 1)  b = 0; if(k == nz) t = 0
6     fn(i,j,k) = cc * f(i,j,k) &
7           + cw * f(i+w,j,k) + ce * f(i+e,j,k) &
8           + cs * f(i,j+s,k) + cn * f(i,j+n,k) &
9           + cb * f(i,j,k+b) + ct * f(i,j,k+t)
10 END DO
```

図 8 3 次元拡散方程式の do concurrent による実装

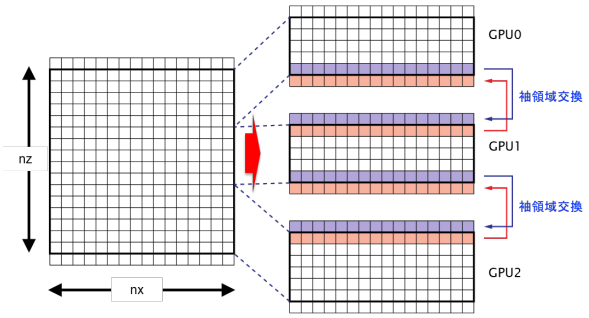

図 9 Stencil の MPI による領域分割.

と比較してより性能上の不利が大きい．

一方 ompgpu2 と acc2 はそれぞれ，omp target data use_device_ptr 指示文と acc host_data use_device 指示文により，MPI 関数に GPU 上のポインタを渡している．本稿で利用した MPI 環境は CUDA aware MPI [14] に対応しており，GPU 同士で直接通信する機能が提供されているため，ompgpu2, acc2 では CPU へのデータ転送を経ることなく通信を行っている．なお，Unified Memory を用いる場合には，CPU 上のポインタと GPU 上のポインタを同一視し区別がなくなるため，CUDA aware MPI の機能を使うことができない．16 GPU 利用時に性能が低下しているのは，16 GPU 以降でノード跨ぎの通信が発生するためである．また，ompgpu2 と acc2 の差は，OpenACC の async 節と wait 指示文によるタスク並列実行によるものである．これを用いると袖領域計算タスクと内点計算タスクを同時刻に実行でき，GPU の空きリソースを有効活用することができる．シングル GPU 実行時に acc2 の性能が高かったのもこのためであり，マルチ GPU 実行時には袖領域計算タスクが終了次第，袖領域通信タスクを発行できるため，通信タスクと内点計算タスクのオーバラップも可能となる．OpenMP の nowait 指示子ではタスクの実行順序を制御できないため，acc2 と同様の実装をするためにはタスク指示文を組み合わせる必要があり，ompgpu2 ではタスク指示文を用いていないために性能差が生じている．

5. おわりに

本稿では do concurrent による GPU 移植手法の評価

表 4　各 Stencil 実装の移植コスト

	指示文等行数	その他のコード変更内容
stdpar1	6	特になし
ompgpu1	12	
ompgpu2	21	
acc1	12	
acc2	25	

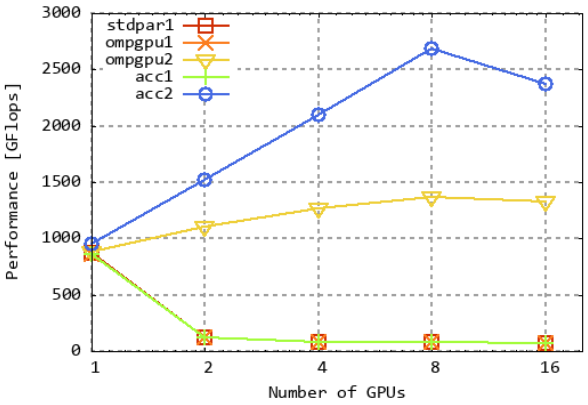

図 10　問題サイズ 256^3 に対するマルチ GPU によるストロングスケーリング.

を目的とし，ICCG, HiMV, Stencil の 3 つのアプリケーションを用いて OpenACC, OpenMP 実装との比較を行った．性能の観点では，Fortran の規格上リダクションがサポートされていないことによる影響が非常に大きく，リダクションが必要なプログラムではまともな性能を得られなかった．しかし，nvfortran の独自拡張によりリダクションが GPU 実行可能な実装においては，OpenACC, OpenMP の素朴な実装と遜色ない性能が得られた．次期 Fortran の規格でリダクションの導入は検討されており，今後に期待したい．また Stencil のマルチ GPU 実行においては，Unified Memory による CPU-GPU 間データ通信が `do concurrent` 実装の性能を著しく低下させることがわかった．性能低下は Unified Memory を使った OpenACC, OpenMP でも同様であったが，OpenACC, OpenMP には明示的にデータ転送を行う代替手段がある．CPU-GPU 間の通信が必要な部分のみ OpenACC や OpenMP で置き換える方法もあるが，その場合には `do concurrent` のメリットである標準性が失われるため悩ましい．

移植コストの観点では，atomic を使えないなど，制限を回避するためのコストが大きい．今回のアプリケーションプログラムは決して複雑なものではなく，並列領域内の関数呼び出しが無いため，関数呼び出しに関する制限には掛からなかったが，大規模なアプリケーションの移植を行う際には大きな問題となるだろう．

謝辞　本研究は JSPS 科研費 JP20H00580, JP21H03447 の助成を受けたものである．

参考文献

[1] Accelerating Fortran DO CONCURRENT with GPUs and the NVIDIA HPC SDK: `https://developer.nvidia.com/blog/accelerating-fortran-do-concurrent-with-gpus-and-the-nvidia-hpc-sdk/`.

[2] Unified Memory for CUDA Beginners: `https://developer.nvidia.com/blog/unified-memory-cuda-beginners/`.

[3] Wisteria/BDEC-01: `https://www.cc.u-tokyo.ac.jp/supercomputer/wisteria/service/`.

[4] Sakamoto, R., Kondo, M., Fujita, K., Ichimura, T. and Nakajima, K.: The Effectiveness of Low-Precision Floating Arithmetic on Numerical Codes: A Case Study on Power Consumption, HPCAsia2020, p. 199–206 (2020).

[5] Kreutzer, M., Hager, G., Wellein, G., Fehske, H. and Bishop, A. R.: A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units, *SIAM Journal on Scientific Computing*, Vol. 36, No. 5, pp. C401–C423 (2014).

[6] Kawai, M. and Nakajima, K.: Low/Adaptive Precision Computation in Preconditioned Iterative Solvers for Ill-Conditioned Problems, *International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia2022, Association for Computing Machinery, p. 30–40 (2022).

[7] Ida, A., Iwashita, T., Mifune, T. and Takahashi, Y.: Parallel Hierarchical Matrices with Adaptive Cross Approximation on Symmetric Multiprocessing Clusters, *Journal of Information Processing*, Vol. 22, No. 4, pp. 642–650 (online), DOI: 10.2197/ipsjjip.22.642 (2014).

[8] Kurz, S., Rain, O. and Rjasanow, S.: The adaptive cross-approximation technique for the 3D boundary-element method, *IEEE Transactions on Magnetics*, Vol. 38, No. 2, pp. 421–424 (online), DOI: 10.1109/20.996112 (2002).

[9] ppOpen-HPC: Open Source Infrastructure for Development and Execution of Large-Scale Scientific Applications on Post-Peta-Scale Supercomputers with Automatic Tuning (AT), `http://ppopenhpc.cc.u-tokyo.ac.jp/ppopenhpc/`.

[10] Iwashita, T., Ida, A., Mifune, T. and Takahashi, Y.: Software Framework for Parallel BEM Analyses with H-matrices Using MPI and OpenMP, *Procedia Computer Science*, Vol. 108, pp. 2200 – 2209 (online), DOI: https://doi.org/10.1016/j.procs.2017.05.263 (2017).

[11] Hoshino, T., Ida, A., Hanawa, T. and Nakajima, K.: Load-Balancing-Aware Parallel Algorithms of H-Matrices with Adaptive Cross Approximation for GPUs, *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 35–45 (online), DOI: 10.1109/CLUSTER.2018.00016 (2018).

[12] 星野哲也，塙　敏博：A64FX におけるテンポラルブロッキングの実装と性能評価，研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 2021-HPC-178, No. 17, pp. 1–8 (2021).

[13] 土井　淳：NVLink における Unified Memory と OpenACC によるプログラミングの性能評価，研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 2017-HPC-159, No. 5, pp. 1–7 (2017).

[14] An Introduction to CUDA-Aware MPI: `https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/`.