Vol.2017-HPC-160 No.15
2017/7/27

階層型行列法ライブラリ$\mathcal{H}$ACApKを用いたアプリケーションのメニーコア向け最適化

星野 哲也[1]　伊田 明弘[1]　塙 敏博[1]　中島 研吾[1]

概要：階層型行列法 (Hierarchical matrices, $\mathcal{H}$-matrices) は，科学技術計算に現れる密行列の近似手法として注目されている．密行列の部分行列を低ランク行列を用いて近似し，大きな密行列全体を多数の小密行列と低ランク近似行列の集合として表すことで，計算量と必要メモリ量のオーダーを$\mathcal{O}(N^2)$から$\mathcal{O}(N\log N)$へと減らすことができる．本手法は，相対的にメモリ容量の小さいメニーコアプロセッサと相性が良いと考えられるが，メニーコアプロッセサを用いての評価は未だ十分でなく，メニーコアプロセッサ向けの最適化をするにあたってどのような課題があるか明らかではない．本研究では，階層型行列法ライブラリである$\mathcal{H}$ACApKを用いたアプリケーションを対象として，メニーコアプロセッサ向けの最適化を行なった．

1. はじめに

多数の演算コアを備えたメニーコアプロセッサを搭載した計算環境は一般的になりつつある．半導体プロセスの微細化に基づく動作周波数の向上による，計算コア単体の性能向上がリーク電流の増大等の物理的制約により難しくなり，周波数の低い多数のコアを用いて全体としての性能を高める手法が主流となったためである．スパコンのLinpak性能を競うTOP500ベンチマーク（2017年6月版 [1]）において国内最高位となったOakforest-PACS [2] に搭載されているIntel Xeon Phi Knights Landing [3]（以下，KNL）や，Linpakの電力効率を競うGreen500ベンチマーク（2017年6月版 [4]）において世界一位となったTSUBAME3.0 [5] に搭載されているNVIDIA Tesla P100 [6]（以下，P100）も，多数の演算コアを搭載したプロセッサである．

メニーコアプロセッサの登場により，演算性能が向上を続ける一方で，メモリ容量の向上は比較的緩やかである．さらに，メニーコアプロセッサの備えるメモリ容量は汎用CPU向けのメモリの容量と比較して小さい．例えば，P100を前世代のTesla K40と比較すると，演算性能が1.43TFlopsから5.3TFlopsへ3.7倍となった一方，メモリ容量は12GBから16GBと1.33倍の増加率である．また，本稿で計算環境として利用するReedbush-Hスーパーコンピュータシステム [7] の計算ノードは，汎用CPUとしてBroadwell世代のIntel Xeonプロセッサ2台とP100を2台備え，汎用CPUのメモリ容量が1プロセッサあたり128GBであるのに対し，P100 1枚のメモリ容量は16GBである．システム全体で比較すると，メモリ容量の差はより顕著である．メニーコアプロセッサであるKNLを用いたOakforest-PACSと汎用CPUから構成される京コンピュータ [8] を比較すると，Linpack性能ではOakforest-PACSが13.5PFlops（世界七位）であるのに対し京コンピュータが10.5PFlops（世界八位）とOakforest-PACSがやや上回る．一方でメモリ容量を比較すると，Oakforest-PACSのKNLが内蔵する高バンド幅メモリであるMCDRAMの総容量が131TB（DDR4と合算すると919TB）であるのに対し，京コンピュータのメモリ総容量は1260TBと，約10倍の差が生じている．

このような理由から，メニーコアプロセッサの利用に際してはそのメモリ容量が問題になりがちである．メモリ容量不足の問題を回避するために，メニーコアプロセッサが持つ高速・小容量のメモリに加え，汎用CPUの持つメモリやNVRAMなどのより低速・大容量の記憶媒体を階層的に用い，データの局所性を高めることによりなるべく計算速度を落とさない手法や，アプリケーションが使うメモリ量自体を削減する手法の開発などが求められている．

本研究で対象とする階層型行列法 [9] は後者の手法である．階層型行列 (Hieralchical matrix, $\mathcal{H}$-matrix) とは，一つの巨大な密行列を多数の小密行列と低ランク近似行列の集合で近似したものである．階層型行列を用いる場合，密行列をそのまま用いる場合と比較して，計算量・必要メモリ量は共に$\mathcal{O}(N^2)$から$\mathcal{O}(N\log N)$へと減少する．ここでNは行列のサイズ（行数または列数）を表す．我々は現在までに，階層型行列とベクトルの積を対象とし，KNLの前

[1] 東京大学情報基盤センター
Information Technology Center, The University of Tokyo

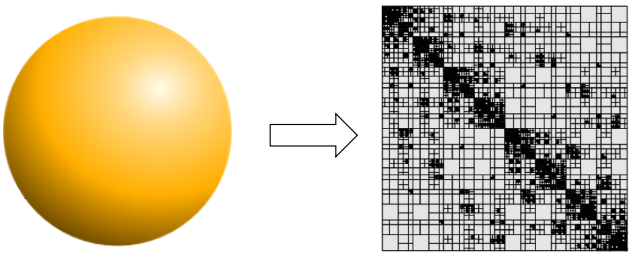

図 1　静電場解析対象と $\mathcal{H}$ACApK により生成される行列分割構造の例

世代である Intel Xeon Phi Knights Corner (KNC) 向けの最適化 [10] や，FPGA を用いての評価 [11] などを行なってきた．

本稿では，階層型行列法ライブラリである $\mathcal{H}$ACApK [12][13] を用いたアプリケーションを対象とし，階層型行列の生成及び階層型行列-ベクトル積計算を含む BiCGSTAB 法の，P100・KNL 向けの最適化を行う．特に階層型行列の生成部に関するメニーコアプロセッサでの評価・最適化は今までになされておらず，メニーコアプロセッサ向けに最適化を行う上での課題の洗い出しを目標とする．

2. 対象アプリケーション

本稿では，階層型行列を用いたアプリケーションとして，ppOpen-APPL/BEM ver.0.5.0 [14] 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．ppOpen-APPL/BEM とは，JST CREST「自動チューニング機構を有するアプリケーション開発・実行環境: ppOpen-HPC」[13] の構成要素の一つであり，境界要素法 (Boundary Element Method, BEM) 用のソフトウェアフレームワークである．本ソフトウェアでは，境界要素法において係数行列として出現する密行列を $\mathcal{H}$ACApK ライブラリにより近似するリファレンス実装が提供されており，本稿ではこれをベースラインの実装とし，最適化を行う．なお，ppOpen-APPL/BEM ver.0.5.0 は [13] よりダウンロードできる．

図 **1** は，本稿で静電場解析の対象とする構造物と $\mathcal{H}$ACApK により生成される行列分割構造の例を示したものである．図 1 中で黒く塗りつぶされた領域は小密行列で表現され，それ以外は低ランク近似行列で表現される．このような階層型行列を係数行列として BiCGSTAB などの線形ソルバーを解くことで，解ベクトルを求める一連の計算手順について最適化を試みる．

3. 階層型行列計算ライブラリ $\mathcal{H}$ACApK の実装概要

以下では，$\mathcal{H}$ACApK ライブラリの実装の概要について述べる．$\mathcal{H}$ACApK は主に，

(1) 階層型行列の生成
(2) 階層型行列を係数行列に持つ線形方程式のための線形ソルバ

の二つのパートから構成され，MPI+OpenMP のハイブリッドプログラミングモデルにより並列化されている．

3.1 階層型行列の生成

本稿では階層型行列生成手法そのものには手を加えないため，メニーコアプロセッサ向けに最適化する上で必要な部分のみの説明にとどめる．詳細は [12] を参照していただきたい．$\mathcal{H}$ACApK の階層型行列の生成は，以下の 3 ステップにより行われる．

(1) 幾何情報に基づくクラスタリング（図 **2**：左）．
(2) 階層型行列構造の作成（図 2：中央）．この時点では部分行列のフレームを作成するだけで，行列要素は計算されない．
(3) 部分行列の計算（図 2：右）．低ランク近似可能な部分行列については ACA(Adaptive Cross Approximation [15], 図 **3**) を用いて近似部分行列を生成し，近似不可能と判定された部分は密行列として計算される．

このうち，ステップ (1)・(2) に関しては全 MPI プロセスで冗長計算を行い，ステップ (3) では，各部分行列の計算に依存関係がないため，部分行列を単位として並列に計算を行う．この計算は MPI+OpenMP を用いて行われるが，プロセス・スレッド間の通信は不要である．本稿で行うメニーコア向けの最適化はステップ (3) を対象とし，ステップ (1)・(2) については見送る．ステップ (3) の疑似コードを図 **4** に示す．1 行目の部分行列のループを MPI+OpenMP により並列化している．部分行列の計算開始時点では近似に必要なベクトルの本数（ランク数）k が未知であるため，5 行目では作業用に十分大きな配列を確保している．作業用配列はランク k の確定後に改めて確保した低ランク近似用の領域に値をコピーした後解放することで，メモリの無駄遣いを防ぐ実装となっている．

また，7-8 行目及び 20 行目の列・行ベクトルと小密行列の生成において，要素の計算方法はユーザーが定義するという手法を採用していることが $\mathcal{H}$ACApK の特徴的なところである．図 **5** は，行列要素値を返す擬似関数である．この関数は，$\mathcal{H}$ACApK を使用するためにユーザーが実装する必要があり，列・行ベクトルと小密行列の生成の際に繰り返し呼ばれる．この際，行列要素値の計算に必要な情報を格納しておくための構造体が 3 行目の st_HACApK_calc_entry であり，あらかじめ値を入力しておく必要がある．図 5 に示した関数は行列要素数回，つまり $\mathcal{O}(N\log N)$ 回呼ばれることとなるため，行列生成部の実行時間に対して支配的となる．

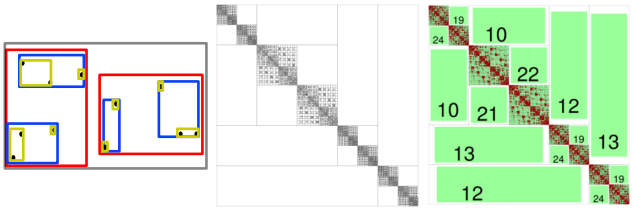

図 2　左：Cluster tree の作成，中央：階層型行列構造の作成，右：部分行列の計算（数字は部分行列の低ランク近似後のランク）

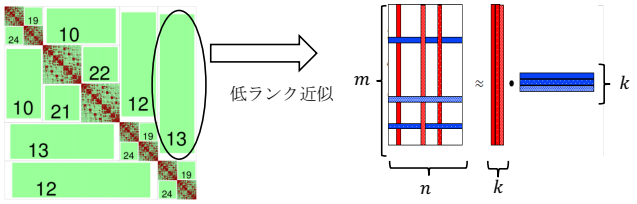

図 3　ACA による低ランク近似．$m \times n$ 行列を k 本の列ベクトルと k 本の行ベクトルの直積により近似する．k が大きくなるほど近似誤差が小さくなることが期待され，k の値により使用メモリ量と近似精度を制御する．

```
do i=1, NUM_SUBMTX  !部分行列のループ
   m = leaf(i)%m !部分行列の列の長さ
   n = leaf(i)%n !部分行列の行の長さ
   if leaf(i)%isLowRank then !低ランク近似可能の時
      allocate(mk_tmp(m,KMAX),kn_tmp(n,KMAX))
         do k=1, KMAX
            mk_tmp(:,k) = 列ベクトルの選択・生成
            kn_tmp(:,k) = 行ベクトルの選択・生成
            zeps = 近似誤差の計算
            if zeps < EPS exit
         enddo
         leaf(i)%k = k
         allocate(leaf(i)%mk(m,k))
         allocate(leaf(i)%kn(n,k))
         leaf(i)%mk(:,1:k) = mk_tmp(:,1:k)
         leaf(i)%kn(:,1:k) = kn_tmp(:,1:k)
      deallocate(mk_tmp,kn_tmp)
   else !小密行列の時
      allocate(leaf(i)%mn(m,n))
      leaf(i)%mn(:,:) = 小密行列の生成
   endif
enddo
```

図 4　部分行列の計算の疑似コード（NUM_SUBMTX：部分行列の数，leaf(i)：i 番目の部分行列，KMAX：ベクトル本数 k の打ち切り値，EPS：必要とする近似精度，leaf(i)%k：i 番目の部分行列の近似に必要であったベクトル本数，leaf(i)%mk/kn：i 番目の部分行列を近似するための k 本の列/行ベクトル，leaf(i)%mn：i 番目の部分行列=小密行列）

3.2　階層型行列を用いた線形ソルバ

$\mathcal{H}$ACApK では，線形ソルバとして BiCGSTAB 法と GCR(m) 法が提供されているが，本稿では BiCGSTAB 法を対象とする．階層型行列を用いた BiCGSTAB 法においては，階層型行列-ベクトル積が実行時間に対して支配的となる．階層型行列 A とベクトル b の積 $x = Ab$ の疑似コードを図 **6** に示す．

```
real*8 function HACApK_entry_ij(i,j,st_bemv)
   integer :: i,j
   type(st_HACApK_calc_entry) :: st_bemv
         !計算内容はユーザーが実装
   return HACApK_entry_ij
end function
```

図 5　密行列上の要素番号 i, j が与えられた時，行列要素値を返す関数．ユーザーが実装する必要がある．

```
do i=1, NUM_SUBMTX  !部分行列のループ
   if leaf(i)%isLowRank then !低ランク近似行列の時
      tmp(1:leaf(i)%k) = 0.0d0
      do k = 1, leaf(i)%k
      do n = 1, leaf(i)%n
         ii = n + leaf(i)%nstr !密行列上での要素番号
         tmp(k)=tmp(k)+leaf(i)%kn(n,k)*b(ii)
      enddo
      enddo
      do k = 1, leaf(i)%k
      do m = 1, leaf(i)%m
         jj = m + leaf(i)%mstr !密行列上での要素番号
         x(jj)=x(jj)+leaf(i)%mk(m,k)*tmp(k)
      enddo
      enddo
   else !小密行列の時
      do n = 1, leaf(i)%n
      do m = 1, leaf(i)%m
         ii = n + leaf(i)%nstr !密行列上での要素番号
         jj = m + leaf(i)%mstr !密行列上での要素番号
         x(jj)=x(jj)+leaf(i)%mn(m,n)*b(ii)
      enddo
      enddo
   endif
enddo
```

図 6　図 4 で生成された階層型行列を A とした時，階層型行列-ベクトル積 $x = Ab$ の疑似コード（b：右辺ベクトル，x：解ベクトル，leaf(i)%mstr/nstr：i 番目の部分行列の元の密行列上での開始位置．開始位置情報を用いて b, x のインデックスを計算する．）

並列化は階層型行列の生成と同様に部分行列を単位として行うが，部分行列-ベクトル積の結果の解ベクトル x への書き込みが各部分行列で独立ではないため，MPI プロセスによる通信や OpenMP スレッド間での atomic 演算などが必要である．各 MPI プロセスは冗長に解ベクトル x 全体を持ち，各 OpenMP スレッドはローカルに解ベクトル x_t を持つ．各 OpenMP スレッドは図 6 の計算でまず x_t に書き込みを行い，その後 x に atomic 演算により x_t の足し込みを行う．その後，各 MPI プロセスは他プロセスと通信を行い，自身の持つ解ベクトル x を更新する．

3.3　$\mathcal{H}$ACApK における並列化方針

前述の通り，$\mathcal{H}$ACApK では MPI+OpenMP によるハイブリッド並列実装が採用されている．MPI・OpenMP どちらの並列化でも，静的に実行領域を決定しており，階層型

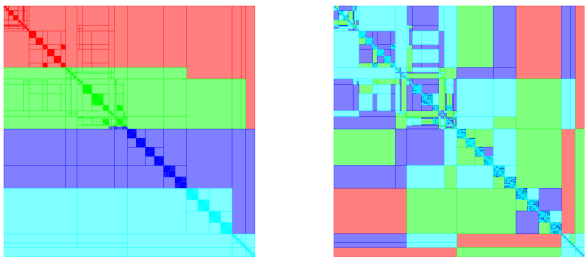

図 7　左：MPI の領域分割方針，右：OpenMP の領域分割方針（実際には左の MPI による分割を行なった上で，各 MPI プロセスの担当領域を右の方針で OpenMP スレッド分割することになる）．

表 1　評価対象プロセッサ

略称	P100	KNL	BDW
名称	NVIDIA Tesla P100 SXM2	Intel Xeon Phi 7250 (Knights Landing)	Intel Xeon E5-2695 v4(Broad well-EP)
動作周波数	1.328 GHz	1.40 GHz	2.10 GHz
コア数	3,584	68	18
ピーク性能	4,759 GFlops	3,046 GFlops	604.8 GFlops
主記憶容量	16GB	16GB	128GB
メモリバンド幅性能	534 GB/sec	490 GB/sec	65.5 GB/sec

行列生成部・線形ソルバ部どちらについても同じ分割方法を適用している．ただし，MPI と OpenMP では実行領域割り当て戦略が異なる．OpenMP では単にスレッド間の計算負荷不均衡を最小化するよう領域分割する (図 **7**：右) が，MPI は計算負荷の不均衡に加え，解ベクトルの通信量が少なくなるよう考慮する (図 7：左)．本稿ではこの並列化方針をそのまま用いているが，メニーコアプロセッサにおいてこの方針が最適であるかどうかは明らかではない．

4. 計算環境

4.1 評価対象プロセッサ

本研究において評価対象としたプロセッサを表 **1** に示す．表 1 中の P100，KNL は前述の NVIDIA Tesla P100 と Intel Xeon Phi Knights Landing であり，BDW は Intel Xeon シリーズの最新世代 (Broadwell-EP) である．表 1 中のメモリバンド幅性能は，Stream Triad の実測値を示している．KNL の主記憶容量・メモリバンド幅性能は MCDRAM の数値を示しており，この他に DDR4（主記憶容量 96GB，メモリバンド幅性能 84.5GB/s）を利用することができる．また，KNL のメモリモード・サブ NUMA クラスタリングモードは，それぞれ Flat・Quadrant モードを使用している．なお，KNL と BDW では Linux カーネルが常駐するため，プログラム全体を単体で実行できるが，P100 には OS が存在しないため，プログラムの実行にはホストプロセッサが必要となる．P100 のホストプロセッサとしては BDW を使用している．

4.2 ソフトウェア環境

本研究において使用したコンパイラ，実行時の環境変数などについて述べる．特に KNL では計算コアへの MPI プロセス・OpenMP スレッドの割り当て方を環境変数で設定するため，注意する必要がある．

P100 向けのコンパイラとしては，pgfortran 17.3 を用い，オプションには-acc -ta=tesla:cc60 -O3 -Mcuda -mcmodel=medium を指定した．MPI のライブラリには MVAPICH2.2 を用いた．特別な環境変数は特に用いていない．

KNL 向けのコンパイラとしては，ifort 17.0.4 を用い，オプションには-align array64byte -xMIC-AVX512 -qopenmp -O3 -ipo を指定した．MPI のライブラリとしては，Intel MPI 2017 update3 を用いている．環境変数としては，OMP_STACKSIZE=1G, ulimit -s 1000000, OMP_NUM_THREADS=66/132/198/264, KMP_HW_SUBSET=1T/2T/3T/4T, KMP_AFFINITY=scatter,verbose, I_MPI_PIN_PROCESSOR_EXCLUDE_LIST =0,1,68,69,136,137,204,205, I_MPI_PIN_DOMAIN=264 を指定している．各環境変数に関する詳細説明は割愛する．また，numactl –membind=1 を使用することで，MCDRAM を利用している．

BDW 向けのコンパイラとしては，ifort 17.0.2 を用い，オプションには-align array64byte -xHOST -qopenmp -O3 -ipo を指定した．環境変数には，OMP_STACKSIZE=1G, ulimit -s 1000000, OMP_NUM_THREADS=18, KMP_AFFINITY=compact を指定している．

5. OpenACC+CUDA による $\mathcal{H}$ACApK の実装

前述の通り，$\mathcal{H}$ACApK ライブラリは OpenMP+MPI のハイブリッド並列が実装されているが，P100 を含む GPU 向けの実装は含まれていない．本研究では GPU 向けの実装として，OpenACC と CUDA による実装を施した．基本的実装方針としては，OpenMP 並列を行なっている部分に OpenACC 実装を追加し，そのうち性能上重要性の高い部分のみ CUDA による実装を行う．この際，MPI 並列実装はそのまま用いることとする．

5.1 階層型行列の生成

3.1 節で記載した通り，階層型行列の生成は三つのステップからなるが，ステップ 3 の部分行列の計算について OpenACC+CUDA 実装を行う．図 4 に示した部分行列の計算の OpenACC+CUDA 実装を行う上で考えるべき

点は，並列化の方針，メモリ管理，ユーザー関数の扱いである．

5.1.1 並列化の方針

GPU では，CPU と比較してはるかに多くのスレッド（数万〜数百万）を生成することができ，またハードウェアの構成上，多くのスレッドを生成した方が効率が良いことが知られている．このため，CPU と同じ並列化方針が最適とは限らない．また GPU ではこの多数のスレッドを 2 階層で管理しており，CUDA ではスレッドとスレッドブロック（スレッドの集合），OpenACC では vector と gang（vector の集合）と呼ぶ．同一スレッドブロック内のスレッド間では同期をとりかつ低コストで通信することができる．一方異なるスレッドブロックにあるスレッドと同期を取るためには，必要なデータをメモリに書き込み，スレッドを破棄し，ホストプロセッサに処理を戻す必要があるため，大きなオーバーヘッドとなる（以下では一連の処理を，カーネルを閉じると呼称する）．以上を踏まえて GPU での並列化の方針を決める必要がある．経験的には，CPU のスレッド並列と GPU のスレッドブロック並列，CPU の SIMD 並列と GPU のスレッド並列が，それぞれ同一のループに適用されることで良好な性能が得られることが多い．そこで本稿での実装においても OpenMP の並列化に習い，部分行列を単位としてスレッドブロック並列を行う．ただし，図 7 : 右のような，計算負荷のバランスは行わない．部分行列と同じ数のスレッドブロックを生成し，計算負荷のバランスは GPU の動的なスケジューリングに任せる．スレッド並列は，図 4 の 7-8 行目及び 19 行目に対して適用する．そのため，スレッドは主に図 5 の行列要素値を計算する関数を単位として並列に計算する．なお，この部分に関しての実装は CUDA を用いて行う．

5.1.2 メモリ管理

上記並列化方針において問題となるのが，作業用領域によるメモリ圧迫である．図 4 中 5 行目では，部分行列の計算毎に独立した作業用領域を確保する実装となっている．しかし GPU では，論理的には全てのスレッドブロックが並列に実行されるため，最終的に階層型行列のために必要なメモリ容量の何倍ものメモリ量が作業用領域のために必要となってしまい，本末転倒である．そこで本実装では，決め打ちで大きめな一次元配列を作業用領域として用意しておき，用意した作業用領域で扱えるだけの量の部分行列の計算を並列に実行する．一度カーネルを閉じた後，作業用領域の値をコピーし，また扱える量だけの部分行列の計算を行う．この一連の処理を，全ての部分行列の計算が終わるまで繰り返す．

本実装では同時に，図 4 中 13-14 行目の leaf(:)%mk/kn の配列も，それぞれコンパイル時に決め打ちのサイズの一次元配列（P100 の場合，メモリ容量を踏まえ，それぞれ 3GB 程度の大きさ）で代用してしまっている．19 行目の leaf(:)%mn はあらかじめサイズが求まるため，サイズを計算した上で，一次元配列化して確保している．つまり leaf(:)%mk の一次元化配列，mk_1D は，leaf(1)%mk の全要素，leaf(2)%mk の全要素,..., leaf(NUM_SUBMTX)%mk の全要素の順に格納されている．この時，leaf(i) が小行列の場合には，leaf(i)%mk の要素数を 0 とみなす．また，leaf(i)%mk の先頭要素のインデックスを格納した，mk_idx を別途用意している．GPU のメモリの確保に相対的に時間がかかること，構造体へのアクセスに時間がかかること，作業用領域に利用できるメモリ量の見積もり易さ，などを考慮してこのような実装にしているが，この実装では利用できるメモリ量を自ら制限してしまっている．利用可能なメモリ量制限の緩和は今後の課題である．

5.1.3 ユーザー関数の扱い

$\mathcal{H}$ACApK では，図 5 の行列要素値の計算を行う関数をユーザーが実装する必要があるが，CPU 向けに書かれた関数は CUDA で書かれた関数からは呼ぶことができないため，新たに CUDA で書かれた実装を用意する必要がある．ユーザーに CUDA を書かせることになるが，行列要素値の計算を行う関数の実装自体はさほど難しくない．本実装では行列要素値の計算は 1 スレッドが行うため，関数内部での並列化を意識する必要がないためである．基本的には，図 5 の 1 行目行頭と，この関数から呼び出される関数に，`attributes(device)` を付与するだけである．ただし，ユーザーは CUDA のデバイス関数の制限を守る必要があり，例えば save 属性の変数などは利用できない．

むしろ問題となるのは，図 5 で使用されている構造体，st_bemv の扱いである．この構造体の中には，行列要素値の計算を行うために，あらかじめユーザーが用意した配列などが登録されている．CPU のメモリと GPU のメモリは物理的に分かれているため，GPU で実行される関数内で扱うデータは，CPU のメモリからあらかじめ GPU のメモリ上にコピーしておく必要がある．st_bemv も GPU のメモリ上にコピーしておく必要があるが，ここでディープコピーの問題が発生する．構造体のメンバに allocatable な配列を含む場合，構造体が持つのはあくまでその配列のアドレス情報であり，配列が実際に確保されるのはメモリ上では構造体と別の領域である．そのため CPU メモリ上の構造体を GPU メモリ上にコピーしても，配列の実体はコピーされない．尚且つ構造体が持つアドレス情報は CPU のメモリ上ものである．正しくコピーするためには，構造体とは別に配列の実体のコピーを行い，構造体の持つアドレス情報を更新する必要がある．これがディープコピーの問題である．つまり st_bemv を GPU のメモリ上に正しくコピーするためには，st_bemv をコピーするだけではうまくいかず，構造体メンバもコピーする必要がある．しかし構造体のメンバはユーザーが用意するため，メンバのコピーはユーザーが書く必要がある．

```
1 subroutine HACApK_copy_st_bemv(st_bemv)
2    type(st_HACApK_calc_entry) :: st_bemv
3       !$acc enter data &
4       !$acc copyin(st_bemv) &
5       !$acc copyin(st_bemv% メンバ1) &
6       !$acc copyin(st_bemv% メンバ2) &
7       ...
8       !$acc copyin(st_bemv% メンバN)
9 end subroutine
```

図 8　st_bemv をコピーするための関数．ユーザーは OpenACC の指示文を用い，構造体のメンバのコピーを正しく実装しなくてはならない．

この問題を解決するために，st_bemv をコピーするための関数図 **8** を用意した．この関数は $\mathcal{H}$ACApK 内の適切なタイミングで呼ばれる．ユーザーは 5 行目以降の OpenACC の指示文を正しく実装する必要がある．本実装方針はユーザーの負担を増やすことになるが，ユーザーの負担軽減方法の検討については今後の課題である．

5.2　階層型行列を用いた線形ソルバ

前述の通り，本稿では BiCGSTAB を対象として OpenACC+CUDA 化を行う．実行時間に対して支配的な図 6 相当部のみを CUDA により実装し，残りを OpenACC により実装する．BiCGSTAB の実装方針は行列生成部と同様，部分行列を単位としてスレッドブロック並列を行い，部分行列内部の計算をスレッド並列する．ここで，図 6 の 7, 13, 21 行目 leaf(:)%mk/kn/mn は，行列生成時に一次元配列化されているため，ここでも同様に一次元配列として扱う．

CUDA による階層型行列-ベクトル積の実装において特徴的なのは，atomic 演算の方針である．OpenMP による実装では，3.2 節で説明した通り，スレッドローカルな解ベクトル x_t を持ち，図 6 の計算の後に atomic 演算により解ベクトル x に縮約した．一方 CUDA では，図 6 の 13, 21 行目において直接 x に対して atomic 演算を用いて足し込んでいる．

6.　最適化

オリジナルの MPI+OpenMP 実装と，今回実装した MPI+OpenACC+CUDA 実装をベースラインとして，P100，KNL 向けに以下の最適化を加えた．

6.1　P100 向け最適化

6.1.1　ノルム計算の並列化

図 4 の 9 行目で行われている近似誤差の計算は，7, 8 行目で生成されたベクトルのノルムを用いている．このノルムの計算では，桁の違いすぎる数を足し合わせることによる桁落ちを防ぐために，演算の順番を工夫している．これによりループ伝搬依存が生じ，並列化しにくいノルム計算となっている．このノルム計算を，単なる自乗和のルートによるノルム計算へと置き換えることで，並列化可能とした．この変更は，解の精度に影響を与える可能性がある．今回対象とした問題では有意と思われる影響は無かったが，この変更による影響の評価を今後行う必要がある．

6.1.2　部分行列のソートによる負荷不均衡の解消

5.1.2 節で説明した通り，階層型行列生成部における部分行列の計算は，決まったサイズの作業用配列で扱えるだけの部分行列を対象として計算する．この際一つのスレッドブロックは一つの部分行列を担当し，一つのスレッドブロックは P100 の 56 個の SM（64 個のコアを内包するコア群）のうちの一つの SM にスケジューリングされる．部分行列のサイズはまちまちであるため，負荷の不均衡が生じる．この不均衡を解消するため，部分行列を行列の大きさ順にソートする．部分行列あたりの計算量は，部分行列のサイズのみでなく，収束時のランク k にも依存する．しかし同程度のサイズを持つ部分行列は同程度のランク k で近似される傾向があるため，サイズによるソートのみである程度の負荷バランスが取れると期待できる．

6.1.3　ユーザー関数の最適化

3.1 節で説明した通り，図 5 で示したユーザー関数は，全体で $\mathcal{O}(N\log N)$ 回呼ばれ，行列生成部の実行時間に対して支配的である．それだけにユーザー関数の最適化は実行時間に大きな影響を及ぼす．ユーザー関数は無論ユーザーの実装次第であるため，ライブラリの提供者には本来手を出せない領域であるが，本稿ではユーザー関数の影響の程度の検証，新たなインターフェースの検討を行うために，最適化を行う．ユーザー関数のベースライン実装としては，ppOpen-APPL/BEM ver.0.5.0 のリファレンス実装を用いる．

ユーザー関数の最適化には，以下の 2 種類が考えられる．

(1) ユーザー関数内部で完結する最適化
(2) $\mathcal{H}$ACApK の変更を伴う最適化

ユーザー関数内部で完結する最適化としては，割り算の逆数の掛け算への置き換え，などが考えられる．本稿でもこれを実装した．$\mathcal{H}$ACApK の変更を伴う最適化としては，冗長なメモリアクセスや計算の削減，などが考えられる．図 5 の関数では，列要素 i と行要素 j から行列要素値を計算する必要があるため，列要素 i に依存する値と行要素 j に依存する値の双方を st_bemv から読み出す必要がある．しかし，例えば図 4 の 7 行目は列ベクトルの生成であるため，一連の行列要素生成過程において，行要素 j は一定である．従って j にのみ依存する値は使い回すことができるが，ループ構造は $\mathcal{H}$ACApK 側にあるため，ユーザーからは見えない．このような場合には，ユーザー関数と $\mathcal{H}$ACApK の双方を最適化する必要がある．

6.2 KNL・BDW 向け最適化

オリジナルの $\mathcal{H}$ACApK では，SIMD 最適化は行われていない．一方 KNL は 512bit の SIMD 幅を持ち，またこれを使いこなすことが性能向上の鍵であるため，主として SIMD 化を目的として最適化を行う．

6.2.1 ユーザー関数以外の SIMD 化

ユーザー関数に手を加えずできる範囲の SIMD 化を行うことを目的とし，以下を適用した．

(1) P100 での変更 (6.1.1 節) と同様，並列化可能なノルム計算への置き換え
(2) メモリアライメントの調整
(3) contiguous 属性の利用
(4) !$omp simd alined や，!DEC$ assume_aligned を用いたコンパイラへのアライメントに関するヒント

メモリアライメントの調整について，例えば図 4 の二次元配列 mk_tmp のアライメントを調整する場合，5 行目を `allocate(mk_tmp(m+mod(8-mod(m,8),8),KMAX)` のように変更する．`m+mod(8-mod(m,8),8)` は 8 の倍数であり，mk_tmp は倍精度であることから，mk_tmp(1,:) は 64byte 境界にアライメントされる．アライメントされていない場合，メモリロード命令が 2 回発行されてしまうが，アライメントすることにより 1 回のメモリロードで実行できる．

contiguous 属性は，ポインタ配列に付与することにより，ポインタの参照先がメモリ上に連続に並んでいることをコンパイラに教えるものである．例えば図 **9** のように，二次元配列の二次元目をサブルーチンの引数として渡し，サブルーチン内では一次元配列的に使用する場合，実際としては連続に並んでいないため，gather 命令によりメモリ読み込みを行う必要がある．コンパイラはこのような呼び出しがある可能性を考慮し，実際には図 9 のような呼び出しがなくても，効率の悪い gather 命令を発行する．図 9 のような呼び出しがない場合には，`real*8, pointer, contiguous :: array(:)` のように contiguous 属性を付与することにより，gather 命令の発行を抑制できる．

!$omp simd alined や，!DEC$ assume_aligned などの指示文は，配列がアライメントされていることをコンパイラに教えるためのものである．例えば図 **10** の場合，mk_tmp(1,k) が 64byte 境界にアライメントされていることをコンパイラに教えることができる．これを行わない場合，mk_tmp の一次元目のサイズが 8 の倍数ではない可能性を考慮し，ロード命令が二回発行されてしまう．また図 **11** のように，同一配列のうち複数箇所についてアライメントされていることを教えるためには，!DEC$ assume_aligned を利用する．

6.2.2 ユーザー関数の最適化

ユーザー関数の最適化については，6.1.3 節での P100 向けの最適化に加え，ユーザー関数の SIMD 化を施した．これは，$\mathcal{H}$ACApK の変更を伴う最適化である．ユーザー関数の SIMD 化の例を図 **12** 示す．スカラ変数のみを引数にもつ関数 user_func の中に演算を記述する．この関数を SIMD 並列で解くことを目標とする．計算に必要となる st_bemv の値を，2-4 行目・8-10 行目でそれぞれ，スカラ変数・長さ 8 の配列にあらかじめコピーしておく．12 行目から始まる長さ 8 のループの中で，13 行目のように引数を指定し，user_func を呼ぶ．user_func 側では，20 行目から始まる!$omp declare simd 指示文を指定することで，この関数が SIMD 並列されるべき関数であることをコンパイラに指示している．simdlen では SIMD 並列長を，linear(ref(ai)) は引数 ai の参照先が連続に並んでいること，uniform は定数であることをそれぞれ指示している．

```
1     ...
2     call foo(array2D(1,:))
3     ...
4  subroutine foo(array)
5     real*8, pointer, :: array(:)
6     ...
```

図 9 contiguous ではない例．サブルーチン foo の中では array は一次元配列に見えるが，実際には二次元配列の二次元目であり，ストライドアクセスが必要になる．

```
1  !$omp simd aligned(mk_tmp:64)
2  do m=1, mmax
3     mk_tmp(m,k)=mk_tmp(m,k)*xxx
4  enddo
```

図 10 !$omp simd aligned の利用例．mk_tmp(1,k) が 64byte 境界にアライメントされていることをコンパイラに教えつつ，SIMD 化する．

```
1  do it=1,k-1
2     !DIR$ assume_aligned mk_tmp(1,it):64
3     !DIR$ assume_aligned mk_tmp(1,k):64
4     !$omp simd
5     do m=1, mmax
6        mk_tmp(m,k)=mk_tmp(m,k)+xxx*mk_tmp(m,it)
7     enddo
8  enddo
```

図 11 !DEC$ assume_aligned の利用例．mk_tmp(1,k) と mk_tmp(1,it) の両方が 64byte 境界にアライメントされていることをコンパイラに教えつつ，SIMD 化する．

7. 性能評価

上述した実装についての性能評価を行う．性能評価の対象とするプロセッサは表 1 である．実験は全て 1 プロセッサのみを用いて行い，MPI による並列化は行わない．また，入力データとしては表 **2** を用いる．いずれも境界要素法を用いた静電場解析において現れる行列である．

表 2 の 100ts を対象とし，最適化の効果を検証する．適用した最適化を表 **3** に示す．オリジナルの実装と，5 章で説

```
! 行番号に依存する変数の読み込み
a = st_bemv%a(j)
b = st_bemv%b(j)
...
do m=1,mmax,8
   ! 列番号に依存する変数の長さ 8 ベクトルへの読み込み
   i = m + str
   a8(1:8) = st_bemv%a(i:i+7)
   b8(1:8) = st_bemv%b(i:i+7)
   ...
   !$omp simd
   do l = 1, 8
      ans(l) = user_func(a8(l),b8(l),..., &
             ,a,b,...)
   enddo
   ...
enddo

real*8 function user_func(ai,bi,...,aj,bj,...)
!$omp declare simd(user_func) simdlen(8) &
!$omp linear(ref(ai,bi,...)) uniform(aj,bj,...)
   real*8 :: ai,bi,...,aj,bj,...
   ...
```

図 **12** !$omp declare simd を用いた関数の SIMD 化の例.

表 **2** 評価対象とする階層型行列

行列名	100ts	human1x1	216h
行数	101250	19664	21600
近似行列数	89534	16202	17002
小密行列数	132740	20416	33096
$\mathcal{H}$-行列使用時のメモリ量	3,307MB	472.0 MB	464.0MB
密行列に対するメモリ量	4.22%	16.0%	13.0%

明した OpenACC+CUDA 実装を (1)：ベースラインとし，6 章で説明した最適化を順次適用する．なお，(3) の負荷不均衡の解消についてであるが，3.3 で述べた通り，KNL・BDW ではベースライン時点で適用済みである．図 **13** に，行列生成部の実行時間を計測した結果を示す．この実行時間は，3.1 節で説明した，ステップ (3) の部分行列の計算の実行時間であり，ステップ (1), (2) は含んでいない．また P100 での実行においては，図 8 に示したデータ転送時間なども含まれている．P100 については，(3) のソートによる性能改善が大きく．(2) と比較して 3.13 倍の性能向上を達成している．KNL では，OpenMP のスレッド数を 66, 132, 198, 264 と 4 通りで実行し，その中の最速値を取っている．(1)~(5) では，264 スレッドが最速であったが，(6) では 132 スレッドが最速であった．スレッド数ごとの実行時間の変遷を図 **14** に示す．このような挙動を示した要因は現在調査中である．また，KNL と BDW 双方において，ユーザー関数の最適化，とりわけ SIMD 化が非常に効果的であった．(5) と比較して (6) では，KNL が 3.41 倍，BDW が 2.79 倍の性能向上を達成した．

また線形ソルバーについて，BiCGSTAB の 1 反復あたりの実行時間を図 **15** に示す．今回施した最適化は主に行列

表 **3** 最適化内容

	最適化内容
(1)	ベースライン実装
(2)	(1)+ノルム計算の並列化
(3)	(2)+ソートによる負荷不均衡の解消 (GPU のみ)
(4)	(3)+ユーザー関数以外の SIMD 化 (KNL・BDW)
(5)	(4)+ユーザー関数内で完結する，ユーザ関数の最適化
(6)	(5)+$\mathcal{H}$ACApK の変更を伴う，ユーザ関数の最適化

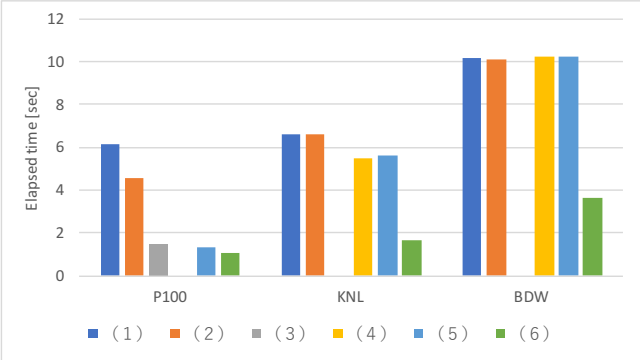

図 **13** 表 3 の最適化を適用した際の実行時間の変遷．入力データは表 2 の 100ts.

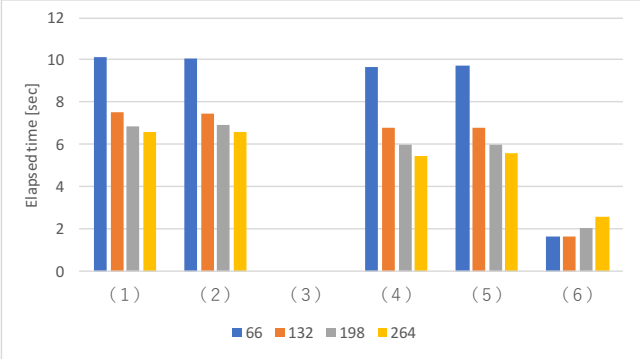

図 **14** 表 3 の最適化を適用した際の KNL の各スレッド数における実行時間の変遷．入力データは表 2 の 100ts.

生成部のためのものであったため，(3) 以外では大きな影響はなかった．(3) のソートについては，P100 の実行時間に悪影響を及ぼし，図 15 の実行時間から 10%程度遅くなった．この原因は，解ベクトルへのアクセスの局所性が低下し，atomic 演算の効率が低下したためだと考えられる．部分行列の並び順は本来，column メジャーライクとなっているため，番号順に実行した場合には解ベクトルへの足し込みに局所性が生まれる．一方，部分行列のサイズ順にソートした場合，解ベクトルへのアクセスはランダムになり，局所性が損なわれる．また，P100 はハードウェアで倍精度の atomicAdd をサポートしている．L2 キャッシュレベルで atomic 演算を行うため，L2 キャッシュにヒットする限りは高速に atomic 演算を実行できる．しかしキャッシュから外れてしまうと一旦メモリに退避されてしまい，メモリに退避されたデータへの足し込みを行うためには，もう一度 L2 キャッシュ上にロードする必要がある．本実装で

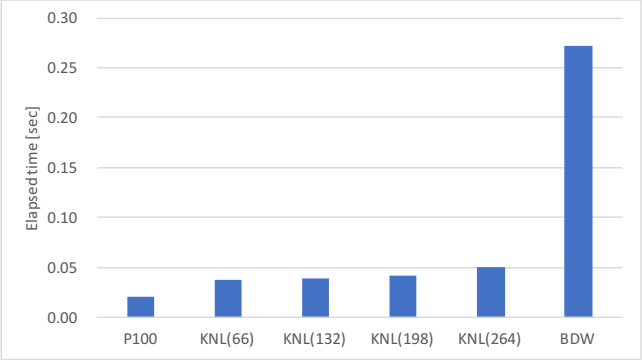

図 15　100ts における，各プロセッサの実行時間．ここで，KNL のカッコ内の数字はスレッド数を表す．

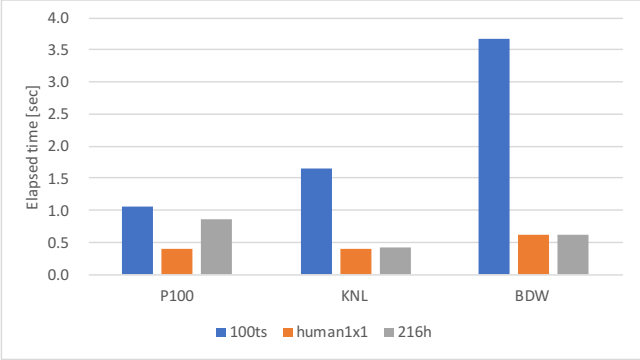

図 16　入力データの違いによる，階層型行列生成部における実行時間の変遷．

は L2 とデバイスメモリ間のメモリバンド幅を使い切っており，さらなる高速化を目指すためには atomic 演算の効率を考える必要がある．一方 KNL については，P100 と同程度のメモリバンド幅性能を持っているにも関わらず，1.8 倍程度低速であったため，メモリバンド幅を使い切れていない．この原因については今後調査が必要である．また，KNL のスレッド数は 66 の時が最適であり，264 スレッドの時が最も悪かった．行列生成部においては，ユーザー関数を SIMD 化できない場合，264 スレッドが最も高効率で，66 スレッドの効率が最も悪かったため，最適なスレッド数を一致させるためにも，ユーザー関数の SIMD 化を促進しなくてはならない．

最後に，入力データを変化させた際の影響を見るために，表 2 の階層型行列生成に掛かる時間を比較した．図 **16** に示したのは，表 3 の (6) の実装を用いた際の実行時間である．human1x1 と 216h は，100ts と比較して階層型行列のメモリ量が小さい．階層型行列生成部における計算量は，出来上がった階層型行列のメモリ量と比例の関係にあるため，human1x1 や 216h の方が演算量は 100ts と比較して一桁小さい．KNL・BDW では概ね演算量に比例した実行時間の低減が見られる一方で，P100 ではほとんど実行時間が減少していない．これは負荷の不均衡が主な原因であり，負荷バランスを均等にするような新たな実装方法を考える必要がある．

8. おわりに

本稿では，階層型行列法ライブラリである $\mathcal{H}$ACApK を用いたアプリケーションのメニーコア向けの最適化を目指し，P100・KNL 向けの実装・最適化を行なった．その結果，特に階層型行列生成部において，KNL・BDW ではユーザー関数の SIMD 化が重要であることが判明した．しかしユーザー関数を SIMD 化するためには，ユーザー関数と $\mathcal{H}$ACApK 双方に手を加える必要がある．このような負担をユーザーに強いるのは好ましくないため，新たな方法を考える必要がある．また P100 向けに，OpenACC+CUDA による実装を行なった．適用した最適化の中で最も効果的であったのは，計算負荷のバランスを調整するために行ったソートであり，また部分行列毎の演算量が比較的不均質な入力データに対して，あまり高速化効果が得られなかったことから，新たな負荷不均衡の解消方法を今後考えていく必要がある．

謝辞　本研究は JSPS 科研費 2611834 の助成を受けたものである．

参考文献

[1] The TOP 500 List: https://www.top500.org/lists/2017/06/.
[2] Oakforest-PACS スーパーコンピュータシステム: http://www.cc.u-tokyo.ac.jp/system/ofp/.
[3] Sodani, A., Gramunt, R., Corbal, J., Kim, H. S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R. and Liu, Y. C.: Knights Landing: Second-Generation Intel Xeon Phi Product, *IEEE Micro*, Vol. 36, No. 2, pp. 34–46 (2016).
[4] The Green 500 List: https://www.top500.org/green500/lists/2017/06/.
[5] TSUBAME3: http://www.gsic.titech.ac.jp/tsubame3.
[6] Pascal Architecture Whitepaper: http://www.nvidia.com/object/pascal-architecture-whitepaper.html.
[7] Reedbush スーパーコンピュータシステム: http://www.cc.u-tokyo.ac.jp/system/reedbush/.
[8] 京コンピュータ: http://www.aics.riken.jp/jp/k/.
[9] Börm, S., Grasedyck, L. and Hackbusch, W.: Hierarchical matrices, Technical report, Max Planck Institute for Mathematics in the Sciences (2003).
[10] 大島聡史, 伊田明弘, 河合直聡, 塙敏博: 階層型行列ベクトル積のメニーコア向け最適化, 研究報告ハイパフォーマンスコンピューティング（HPC）, Vol. 2016-HPC-155, No. 39, pp. 1–9 (2016).
[11] 塙敏博, 伊田明弘, 大島聡史, 河合直聡: FPGA を用いた階層型行列ベクトル積, 研究報告ハイパフォーマンスコンピューティング（HPC）, Vol. 2016-HPC-155, No. 40, pp. 1–6 (2016).
[12] 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 (2014).

[13] 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/`.
[14] 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 (2017).
[15] 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 (2002).