Knights Landing† プロセッサーにおける 高帯域幅メモリー (HBM) としての MCDRAM: 開発者ガイド

Ryo Asai
Colfax International

2016 年 5 月 11 日

抄録

この文献は、第 2 世代インテル® Xeon Phi™ プロセッサー開発コード名 Knights Landing (KNL) の新機能に注目した開発者ガイドの一部です。ここでは、マルチチャネル・ダイナミック・アクセス・メモリー (MCDRAM) テクノロジー・ベースのオンパッケージの高帯域幅メモリー (HBM) について説明します。

- HBM の 3 つのモード: フラットモード、キャッシュモード、ハイブリッド・モード。
- アドレス指定可能なメモリーとして HBM を使用する 2 つの手法: numactl ツールによるアフィニティー・ポリシーの設定と memkind ライブラリーの特殊アロケーターの使用。
- ブート可能な Knights Landing† 上で実行するアプリケーション向けの最適な使用モデルを決定するためのガイドライン

この文献と KNL† プロセッサーに関するほかのホワイトペーパーは、Colfax Research の Web サイト (colfaxresearch.com/knl-guide (英語)) から入手できます。

目次

Colfax International は、ハイパフォーマンス・コンピューティング・ソリューションと並列コンピューティングのエキスパートレベルの教育プログラムのリーディング・プロバイダーです。すぐ使える Colfax システムには、ワークステーション、サーバー、クラスター、ストレージ、およびパーソナル・スーパーコンピューティング・ソリューションがあります。Colfax が提供する教育プログラムは、ハードウェア革新と計算分野における進化を利用して、ソフトウェア開発者が最先端の計算プラットフォームで最高のパフォーマンスを実現できるようにします。Colfax が提供する包括的なサービスは、価格/パフォーマンスにおいて大幅な利点をもたらし、IT アジリティーを高め、業績を向上し、新たな発見へと導きます。Colfax International の広範な顧客ベースには、Fortune 1000 企業、教育機関、政府機関が含まれます。1987 年設立の Colfax International は、カリフォルニア州サニーベールにある、株式非公開企業です。

†開発コード名

1. はじめに

コンピューティング・システムのメモリー帯域幅は、計算アプリケーションにおける一般的なパフォーマンス・ボトルネックの 1 つです。帯域幅によって制限されるアプリケーションは、アルゴリズムにおいてメモリーアクセスごとの浮動小数点演算数が少ない (計算強度が低い) という特徴を持ちます。そのようなアルゴリズムには、BLAS レベル 1 およびレベル 2 ルーチン (ベクトルドット積や行列-ベクトル乗算など)、高速フーリエ変換 (FFT)、およびステンシル操作が含まれます。アプリケーションの計算強度が低いため、プロセッサーの浮動小数点処理能力は一般に重要ではありませんが、メモリー帯域幅によってアプリケーションのパフォーマンスが制限されます ([1] を参照)。

そのようなアプリケーションにおけるメモリー帯域幅の要求に対応するため、Knights Landing† (KNL†) アーキテクチャー・ベースの第 2 世代インテル® Xeon Phi™ プロセッサーは、マルチチャネル・ダイナミック・アクセス・メモリー (MCDRAM) テクノロジー・ベースのオンパッケージの高帯域幅メモリー (HBM) を搭載しています。このメモリーは、同じプラットフォーム上の DD4 メモリー (≥ 90GB/秒) と比較して、最大で約 5 倍のパフォーマンス (≥ 400GB/秒) を提供します。HBM を最大限に利用することが、帯域幅依存のアプリケーションで素晴らしいパフォーマンスを達成する鍵です。

HBM の利用可能な使用モデルは、Knights Landing† プロセッサーの 2 つのフォームファクター (セルフブート可能なプロセッサー・バージョンと PCIe* アドインのコプロセッサー・カード・バージョン) では異なります。ここでは、セルフブート可能なプロセッサー・バージョンにおける HBM の使用について述べます。メモリー構成を図 1 に示します。

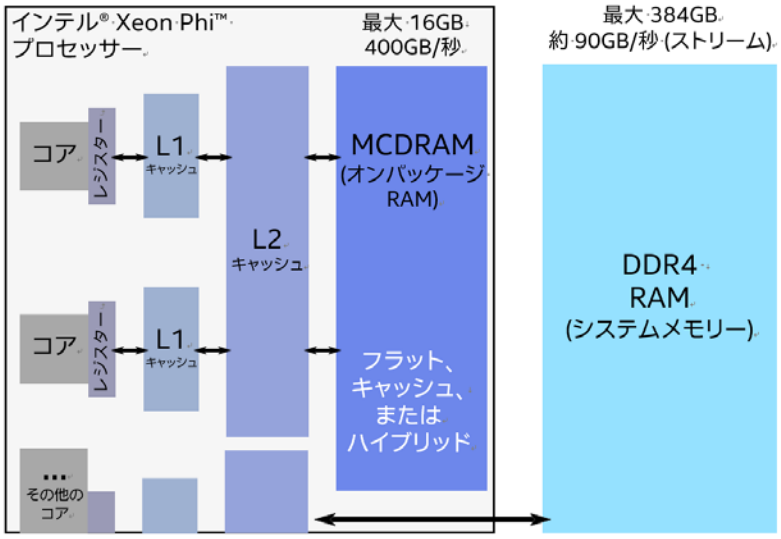

図 1: ブート可能な第 2 世代インテル® Xeon Phi™ プロセッサー (Knights Landing†) の物理メモリー構成

オンパッケージの HBM (図では "MCDRAM" と表記) は、プロセシング・コアの隣の CPU チップ上にあります。従来のメモリーモジュールとは異なり、MCDRAM は取り外したり、交換することができません。モデルに応じて、Knights Landing† プロセッサーには最大 16GB の HBM が搭載されています。オンパッケージ・メモリーは、従来の DDR4 メモリーモジュールとして取り付けられるオンプラットフォーム・メモリーとは異なります。メモリーモジュールのサイズに応じて、オンプラットフォーム RAM は 384 GB になることもあります。

†開発コード名

2. HBM モード

Knights Landing† プロセッサーの HBM は、ラストレベル・キャッシュまたはアドレス指定可能なメモリーとして利用できます。この設定は、ブート時に BIOS 設定で 3 つの MCDRAM モード (フラットモード、キャッシュモード、ハイブリッド・モード) からいずれかを選択することで決定されます。図 2 は、3 つのモードを図解したものです。

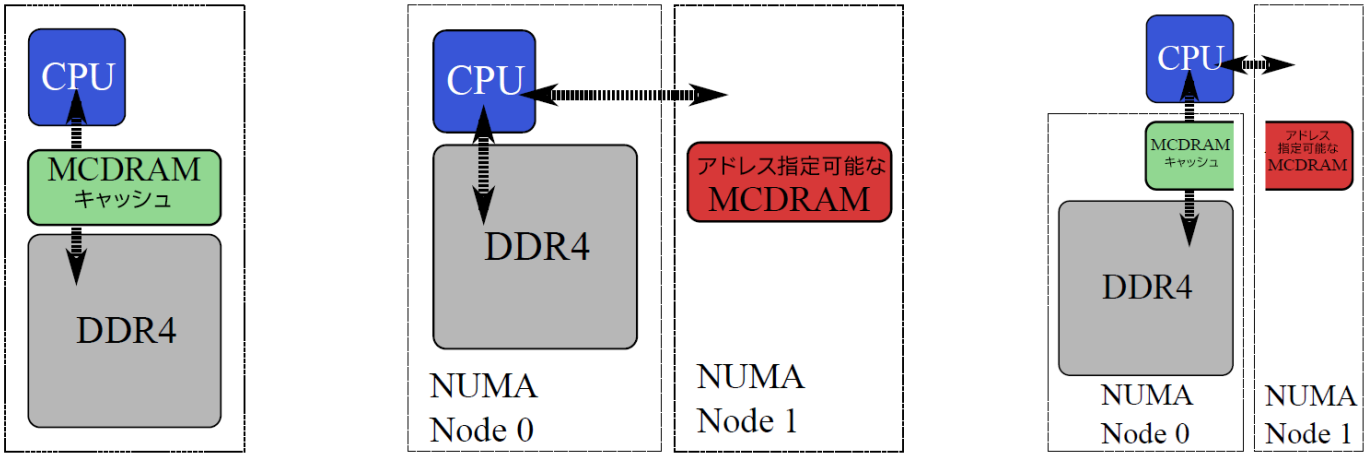

図 2: ブート可能な Knights Landing† プロセッサーの HBM の使用モード

フラットモードは、HBM 全体をアドレス指定可能なメモリーとして使用し、**キャッシュモード**は、HBM 全体をキャッシュとして使用します。**ハイブリッド・モード**は、HBM の一部をアドレス指定可能なメモリーとして使用し、残りをキャッシュとして使用します。アドレス指定可能なメモリーは、オブジェクトを明示的に割り当てるのに利用できます。HBM をキャッシュとして使用する場合、オペレーティング・システム (OS) には HBM が見えないため、L2 キャッシュとオンプラットフォームの DDR4 メモリーの間のラストレベル・キャッシュとして "裏で" 動作します。

3 つのモードの長所と短所は次のとおりです。

- **キャッシュモード** – 使用するために追加の作業は必要ありません。HBM キャッシュでミスが頻繁に発生する場合、フラットモードよりもパフォーマンスが悪くなります。
- **フラットモード** – キャッシュモードよりも優れたパフォーマンスを達成できる可能性がありますが、コードや実行環境の変更が必要になります。
- **ハイブリッド・モード** – **フラットモード**と**キャッシュモード**の利点が得られますが、それぞれのサイズが小さくなります。

最適なモードは、アプリケーションに依存します。特定のアプリケーションでモードを選択するためのガイドラインについては、セクション 4 で詳しく述べます。このセクションの後半では、3 つの MCDRAM モードの技術的な情報を掘り下げていきます。

†開発コード名

2.1. キャッシュモード

Knights Landing† プロセッサーを**キャッシュモード**または**ハイブリッド・モード**でブートすると、HBM の全体または一部がキャッシュとして使用されます。HBM キャッシュは、Knights Landing† プロセッサーのメモリー階層において、L2 キャッシュとアドレス指定可能なメモリーの間のラストレベル・キャッシュ (LLC) として扱われます。HBM キャッシュは 物理アドレス空間全体を格納し、HBM 自体は L2 キャッシュに格納されます。

HBM をキャッシュとして使用する利点は、プラットフォームで管理でき、ソフトウェアから見えることです。開発者は、何もしなくてもアプリケーションで HBM を使用することができます。そのため、このモードは、メモリー・パフォーマンスのチューニングに馴染みのない開発者やチューニングが困難なアプリケーションに役立ちます。

このモードの欠点は、オンプラットフォーム・メモリー (DDR4) へのアクセス・レイテンシーが増加する可能性があることです。図 3 は、コアが L1 または L2 キャッシュに格納されていないメモリーアドレスからデータを要求し、そのアドレスが HBM キャッシュにも格納されていない場合を示します。

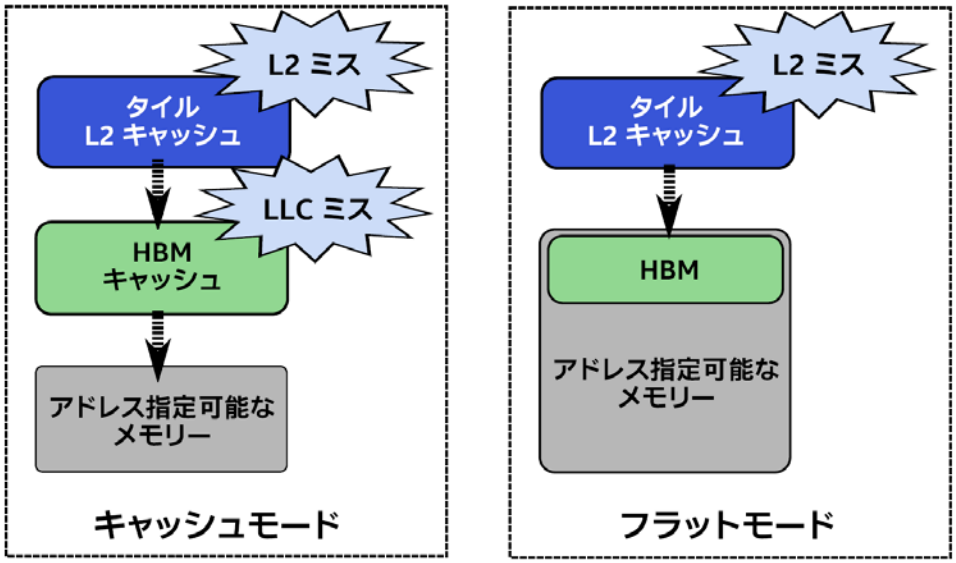

図 3: フラットモードと**キャッシュモード**のキャッシュ・ミス・レイテンシー

キャッシュモードでは、コアは HBM キャッシュを照会してからでないと、オンプラットフォーム・メモリー・コントローラーに要求を送信することができません。一方、アドレス指定可能なメモリーの一部として HBM を**フラットモード**で使用する場合、L2 キャッシュをミスしたコアは直接オンプラットフォーム・メモリーにアクセスできます。HBM キャッシュでは、L2 および HBM キャッシュミスがあった場合、追加のステップが発生するため、レイテンシーも増えます。

2.2. フラットモード

Knights Landing† プロセッサーをフラットモードでブートすると、HBM 全体がアドレス指定可能なメモリーとして使用されます。HBM をアドレス指定可能なメモリーとして使用する場合、物理アドレス空間を DDR4 と共有し、L2 キャッシュに格納されます。NUMA (Non Uniform Memory Access) アーキテクチャーでは、HBM のアドレス

†開発コード名

指定可能メモリーはコアを持たない個別の NUMA ノードとして認識され、ほかの NUMA ノードにすべてのコアと DDR4 が含まれます。

`numactl` (Linux* ディストリビューションの一部) は、HBM をフラットモードで使用する場合に便利なコマンドライン・ツールです。HBM に関連付けられている NUMA ノードを確認するには、`--hardware` または `-H` オプションを指定して `numactl` コマンドを実行し、コアのないノードを探します。

```
user@knl% numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16  ... rest of the cores ...
node 0 size: 98207 MB
node 0 free: 94141 MB
node 1 cpus:
node 1 size: 16384 MB
node 1 free: 15923 MB
```

リスト 1: `numactl -H` を使用して HBM のサイズと NUMA ノードを確認

リスト 1 のシステム (製品化前の第 2 世代インテル® Xeon Phi™ コプロセッサー 16GB MCDRAM 搭載) では、NUMA ノード 0 がすべてのコアが関連付けられたオンプラットフォーム DDR4 メモリーで、NUMA ノード 1 がコアなしのオンパッケージ MCDRAM (つまり HBM) です。

HBM をアドレス指定可能なメモリーとして使用する利点は、HBM をキャッシュとして使用する場合と比べて、開発者が HBM をより細かく制御できることです。適切なチューニングにより、16GB を超えるメモリーを使用するアプリケーションは、HBM をキャッシュとして使用するよりも、アドレス指定可能なメモリーとして使用するほうがパフォーマンスを向上できることがあります。フラットモードでは、キャッシュモードと異なり、オンプラットフォーム・メモリーにあるオブジェクトにアクセスする場合、オンパッケージ HBM を照会する必要がないからです。

しかし、HBM をアドレス指定可能なメモリーとして使用する場合、その使用については完全に開発者に任されるため、HBM をキャッシュとして使用する場合よりも多くのアプリケーション開発作業が必要になります。アプリケーション・メモリーはデフォルトで DDR4 に割り当てられるため、開発者はコマンドライン・ツールや特殊アロケーター (セクション 3 で説明) を使用して HBM への割り当てを指定する必要があります。パフォーマンス・チューニングは、通常、HBM をアドレス指定可能なメモリーとして使用する場合のほうが難しく、特に大規模なアプリケーションや複雑なアプリケーションではより困難です。

2.3. ハイブリッド・モード

Knights Landing† プロセッサーをハイブリッド・モードでブートすると、HBM の一部はアドレス指定可能なメモリーとして使用され、残りはキャッシュとして使用されます。2 つの比率はブート時に選択されます。アドレス指定可能なメモリー領域はフラットモードと同様に動作し、キャッシュ領域はキャッシュモードと同様に動作します。ハイブリッド・モードは、すべてのアプリケーションが HBM をアドレス指定可能なメモリーとして使用することに対応していない大規模なマルチユーザー・クラスターで便利です。

†開発コード名

3. アドレス指定可能なメモリーとしての HBM

フラットモード (または**ハイブリッド・モード**) で使用する場合、開発者は手動でアプリケーションが HBM をアドレス指定可能なメモリーとして使用するように指示しなければなりません。これは、numactl ツールを使用するか、memkind ライブラリーを使用して行うことができます。推奨される方法は、HBM のアドレス指定可能なメモリー領域のサイズとアプリケーションのメモリー・フットプリントに依存します。`numactl` ツールを使用して、HBM のアドレス指定可能なメモリーのサイズを確認できます (リスト 1 を参照)。メモリー・フットプリントの確認方法は付録 A で説明します。アプリケーションのメモリー・フットプリントが HBM のアドレス指定可能なメモリーのサイズよりも**小さい**場合、numactl の使用を推奨します。そうでない場合は、memkind ライブラリーを使用します。

注: セクション 3 の以降の説明では、HBM は HBM をアドレス指定可能なメモリーとして使用することを指します。

3.1. numactl

アプリケーションのメモリー要件が利用可能な HBM よりも小さい場合、HBM の利点を得るため、デフォルトですべての割り当てを HBM で行うようにします。これは、`--membind` または `-m` オプションを指定して `numactl` を実行し、アプリケーションを特定の NUMA ノードにバインドすることで達成できます。アプリケーションを NUMA ノードにバインドすると、アプリケーション内のすべての割り当てはデフォルトで指定した NUMA ノードのメモリーで行われます。`numactl` は実行時に使用されるコマンドなので、コードの変更や再コンパイルは不要です。

ここでは NUMA ノード 1 が HBM であるため (リスト 1 を参照)、次のコマンドで実行ファイル `run-app` のメモリー割り当てを HBM にバインドすることができます。

```
user@knl% numactl --membind 1 ./run-app
```

HBM に関連付けられる NUMA ノードは、クラスタリング・モードのブート時の設定オプションに応じて異なります (クラスタリング・モードの詳細は [2] を参照)。そのため、`numactl -m` でアプリケーションを HBM の NUMA ノードにバインドする前に、`numactl -H` で Knights Landing† プロセッサーの NUMA 構成を必ず確認したほうが良いでしょう。

SNC-4/SNC-2 モード (上記のクラスターモード) で HBM NUMA ノードを使用するには、`numactl --membind` でカンマ区切りのリストを使用します。

```
user@knl% numactl -m 1,3,5,7 ./run-app
```

この方法は、使用されているプログラミング言語に関係なく、メモリー・フットプリントが 16GB 未満の任意の実行ファイルで利用できます (コードが NUMA ポリシーを変更せず、明示的にオンプラットフォーム・メモリーへの割り当てを行っていない場合)。

†開発コード名

3.2. memkind ライブラリー

memkind ライブラリーは、汎用メモリー割り当て関数の C ライブラリーである jemalloc をベースに開発された、ユーザーが拡張可能なヒープマネージャーです。任意の NUMA アーキテクチャーに一般化することができますが、Knights Landing† プロセッサーでは、主に C/C++ 向けの特殊アロケーターを使用して手動で HBM に割り当てる際に使用されます。memkind ライブラリーは、Fortran も限定的にサポートしています。詳細はセクション 3.3 で述べます。

memkind ライブラリーのオープンソース版は [3] からダウンロードできます。memkind ライブラリーには、*hbwmalloc* と *memkind* の 2 つのインターフェイスがあります。どちらのインターフェイスも同じバックエンドを使用します。実際、高帯域幅メモリー・アロケーターである hbwmalloc は、内部で memkind インターフェイスを呼び出します。つまり、*memkind* インターフェイスは、*hbwmalloc* インターフェイスのすべての機能を備えています。さらに、*hbwmalloc* インターフェイスには含まれないいくつかの試験的な機能も備えています。ただし、この論文の執筆時点では、*hbwmalloc* インターフェイスは安定していますが、*memkind* インターフェイスは完全に安定していません。そのため、ここでは *hbwmalloc* インターフェイスについて説明します。memkind ライブラリーに興味のある方は、次のコマンドを実行して、*memkind* インターフェイスに関する Linux* マニュアルページを確認できます。

```
user@knl% man memkind
// ... Memkind manual ... //
```

リスト 2 は *hbwmalloc* インターフェイスを使用して基本的なメモリーの割り当て/解放を行う方法を示し、リスト 3 はコンパイル手順を示します。

```
#include <hbwmalloc.h>   // hbwmalloc interface
// ... //

const int n = 1<<10;
double* A = (double*) hbw_malloc(sizeof(double)*n); // Allocation to HBM
// ... //

hbw_free(A); // Deallocate with hbw_free
```

リスト 2: *hbwmalloc* を使用した基本的な割り当て/解放

```
user@knl% icpc foo.cc -lmemkind -o run-app-intel
user@knl% g++ foo.cc -lmemkind -o run-app-gcc
```

リスト 3: *memkind* および *hbwmalloc* とのリンク

`hbw_malloc()` は、標準の `malloc()` と同様に動作します。HBM にブロックを割り当て、ブロックの開始位置へのポインターを返します。`jemalloc` ライブラリーの 2 つのヒープ・アロケーター `calloc()` と `realloc()` に対応する HBM バージョンは `hbw_calloc()` と `hbw_realloc()` です。

†開発コード名

メモリー割り当て関数は、HBM への割り当てを試みるだけであることに注意してください。HBM に十分な空き容量がない場合、警告なしで DDR4 メモリーに割り当てられます。この動作は、`hbw_set_policy()` で変更できます。利用可能なポリシーの説明は、次のコマンドを実行して、*hbwmalloc* に関する Linux* マニュアルページを参照してください。

```
user@knl% man hbwmalloc
// ... hbwmalloc manual ... //
```

リスト 4: *hbwmalloc* マニュアル

`hbw_check_available()` 関数を使用して、HBM が利用可能かどうか確認することもできます。ただし、利用可能かどうかは分かりますが、空き容量は返されません。

hbwmalloc には、特定の最適化に必要な制御を提供する高度なアロケーターがあります。例えば、場合によっては、割り当て時にメモリー・アライメントを指定したいことがあるでしょう ([4] を参照)。*hbwmalloc* でアライメントされたメモリーブロックを割り当てるには、`hbw_posix_memalign()` を使用します。リスト 5 は、HBM でアライメントされた割り当てを行います。

```
1 double* A;
2 int ret = hbw_posix_memalign((void*) A, 64, sizeof(double)*n);
3 // ..... //
4 hbw_free(A);
```

リスト 5: HBM でのアライメントされた割り当て

`hbw_posix_memalign_psize()` を使用することで、メモリー・ページ・サイズを指定することもできます。利用可能なページサイズについては、*hbwmalloc* に関する Linux* マニュアルページ (リスト 4) を参照してください。

3.3. Fortran

memkind ライブラリーは、Fortran を限定的にサポートしています。Fortran では、割付け配列のみ明示的に MBM に配置できます。FASTMEM 属性は、割付け配列を HBM に配置するようにランタイムシステムに指示します。

† 開発コード名

```
1  REAL, ALLOCATABLE :: A(:), B(:)
2
3  ! FASTMEM attribute
4  !DEC$ ATTRIBUTES FASTMEM :: A
5
6  ! A is allocated in HBM
7  ALLOCATE (A(1:1024))
8
9  ! B is allocated in DDR4
10 ALLOCATE (B(1:1024))
```

リスト 6: Fortran での手動による HBM へのメモリー割り当て

同時に、前述のとおり、Fortran アプリケーションが HBM (最大 16GB) に完全に収まる場合、numactl ツールを使用して、Fortran アプリケーションを HBM から実行することができます。

†開発コード名

4. メモリーとプログラミング･モデルの選択

4.1. HBM を利用するプログラミング

異なる HBM 使用モデル向けの 3 つの MCDRAM モードと (セクション 2 を参照)、HBM をアドレス指定可能なメモリーとして使用する 2 つの方法 (セクション 3 を参照) が分かったところで、HBM を利用する最適な方法を選択するにはどうした良いのでしょうか? 最初に、図 4 のフローチャートを使用して、HBM の使用モデルを決定することを推奨します。

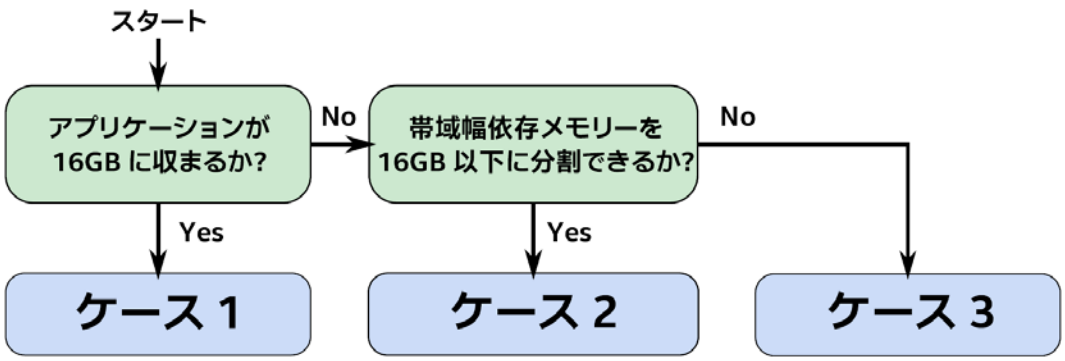

図 4: HBM 使用ケースのフローチャート

- **ケース 1: アプリケーション全体が HBM に収まる場合**

  これは最高のシナリオです。アプリケーションが HBM に収まる場合、フラットモードに設定して、セクション 3.1 の `numactl` の手順に従います。この使用モデルは、コード変更が不要で、ほかの場所へ割り当てを行う特殊なアロケーターを使用していない限り、任意の言語で記述された任意のアプリケーションで動作します。ソースコードを変更する必要はありませんが、一般的なメモリーの最適化により利点が得られる可能性があります。メモリー･トラフィックの最適化については、[5] などのオンライン資料を参照してください。

  `numactl` を使用できない場合、キャッシュモードを使用します。問題が HBM キャッシュに収まるため、HBM キャッシュミスはわずかです。HBM キャッシュミスは、HBM をアドレス指定可能なメモリーとして使用する場合とキャッシュとして使用する場合にパフォーマンスの違いが生じる主な要因です。そのため、キャッシュモードを使用することで、フラットモードのパフォーマンスと同じか、それに近くなります。しかし、HBM をキャッシュとして使用する場合に固有のオーバーヘッドがあるため、`numactl` の使用を推奨します。

- **ケース 2: アプリケーションの帯域幅依存領域が分割可能で HBM に収まる場合**

  帯域幅依存のデータセット領域が HBM の収まる場合、**フラットモード**と memkind ライブラリーの使用を推奨します (セクション 3.2 を参照)。この使用モードでは、若干のコード変更が必要になります。また、データセットのどのコンポーネントが**帯域幅依存**か把握する必要があります。その方法は、付録 B を参照してください。

  何らかの理由で memkind ライブラリーを利用できない場合 (例えば、アプリケーションの言語でサポートされていない場合) であっても、外部ライブラリーを呼び出すことでこのモデルの利点を得ることができま

[†] 開発コード名

す。例えば、インテル® マス・カーネル・ライブラリー (インテル® MKL) は、Python* や R を含むさまざまな言語から呼び出すことができます。もちろん、memkind ライブラリーを使用する独自の C/C++ モジュールを作成することもできます。

- **ケース 3: アプリケーションが大きく分割が困難な場合**

  残念ながら、これは最悪のシナリオです。デフォルトで**キャッシュモード**を使用することを推奨します。これにより、アプリケーションがわずかであっても HBM の利点を得ることができます。

  また、フラットモードで `numactl` を使用して DDR4 への割り当てを試してみるのも良いでしょう。一部のアプリケーションでは、HBM キャッシュのキャッシュ・ミス・レイテンシー (セクション 2.1 を参照) により HBM の利点が得られないことがあります。

  どちらの場合も、付録 B に示す手法を試し、帯域幅依存のデータ領域を特定して分離可能かどうか調査することを推奨します。分離可能であれば、ケース 2 の手法を使用できます。

4.2. HBM を利用しないプログラミング

HBM は、メモリー帯域幅に依存するアプリケーションにのみ役立つということを思い出してください。アプリケーションの計算強度が高い場合 (つまり、キャッシュのデータ再利用率が高い場合)、またはメモリー・トラフィックがわずかな場合 (つまり、L2 キャッシュに収まる場合)、汎用 CPU と同様に、オンパッケージ・メモリーは考慮せずに、Knights Landing† の通常のオンプラットフォーム・メモリーを使用できます。

メモリー帯域幅に依存しないアプリケーションの例として、BLAS レベル 3 ルーチン (キャッシュ効率が最適化された一般的な行列-行列乗算) やメモリー・フットプリントの小さいモンテカルロ計算が挙げられます。計算負荷の高いアプリケーションでは、演算パフォーマンスのほうがメモリー帯域幅よりも重要であり、KNL† の演算処理能力に注目して開発が進められるべきです [6]。

参考文献 (英語)

1. Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an Insightful Visual Performance Model for Multicore Architectures. Communications of the ACM, 52(4):65–76, April 2009. http://dx.doi.org/doi:10.1145/1498765.1498785.
2. Clustering Modes in Knights Landing Processors: Developer's Guide. http://colfaxresearch.com/knl-numa/.
3. Open source memkind library on github. https://github.com/memkind/memkind.
4. Andrey Vladimirov. Fine-Tuning Vectorization and Memory Traffic on Intel Xeon Phi Coprocessors: LU Decomposition of Small Matrices. http://research.colfaxinternational.com/post/2015/01/27/LU.aspx.
5. Colfax Hands On Workshop (HOW) series. http://colfaxresearch.com/how-series.
6. Guide to Automatic Vectorization with Intel AVX-512 Instructions in Knights Landing Processors. http://colfaxresearch.com/knl-avx-512/.

†開発コード名

付録 A: アプリケーションのメモリー・フットプリント

HBM の使用モデルを決定するには、アプリケーションのメモリー・フットプリントを把握する必要があります。アプリケーションのメモリー使用量をチェックする方法はいくつかありますが、その 1 つが ps コマンドです。実行ファイル run-app のメモリー使用量を取得するには、アプリケーションを実行し、別のターミナルで次のコマンドを実行します。

```
user@knl% ps -C run-app u
USER   PID  %CPU  %MEM VSZ      RSS   TTY    STAT START TIME    COMMAND
user   6577 26330  0.0 17943576 78360 pts/2 Rl+  15:04 267:41 ./run-app
```

メモリー使用量は RSS の下に (KB 単位で) 表示されます。ps は、呼び出されたときにメモリー使用量を報告します。アプリケーションに多数の割り当て/解放が含まれる場合は、ps と一緒に watch を使用して使用量をモニターすると良いでしょう。

付録 B: 帯域幅依存データ

memkind ライブラリーを効率良く利用するには、アプリケーション・データの帯域幅に依存する部分を把握しておいたほうが良いでしょう。帯域幅依存メモリーとは、HBM に割り当てることで、アプリケーション全体に大きな利点をもたらすアプリケーション・データの領域です。帯域幅依存データは、次のような特徴を備えています。

- データの合計サイズが 30MB 以上である。

  これよりも小さい場合、L2 キャッシュに格納されます。小さなデータセットを HBM に配置しても利点が得られますが、大きなデータのほうが大きな利点が得られるため優先すべきです。

- データが複数回読み取られる。

  一度しか読み取られないデータは、HBM に最適なデータではありません。HBM に割り当てる場合、複数回読み取りまたは書き込みされるデータ構造を優先します。

- データ・アクセス・パターンが連続している。

  HBM は、データ・アクセス・パターンが連続している場合に最高のパフォーマンスを発揮し、ランダムな場合には最低のパフォーマンスとなります。HBM はランダムアクセスであっても DDR4 のパフォーマンスを上回ることがありますが、その差はわずかです。連続してアクセスされるデータ構造を優先的に HBM へ割り当てたほうが良いでしょう。

- データが帯域幅依存のセクションで使用されている。

  データ自体が**帯域幅依存**メモリーの特徴を備えていても、ワークロードのボトルネックがメモリーでない場合 (例えば、計算依存や I/O 依存の場合)、HBM による利点はあまり得られません。データポイントあたりの演算が少ない場所、またはデータポイントあたりの I/O 処理が少ない場所で使用されているデータを優先します。

[†]開発コード名