# メモリーアクセスのパフォーマンス・ボトルネックの検出

## インテル® VTune™ Amplifier XE の新しいメモリーアクセス解析により アプリケーションのパフォーマンスを素早く簡単に向上

**Kevin O'Leary インテル コーポレーション ソフトウェア・テクニカル・コンサルティング・エンジニア**
**Dmitry Ryabtsev インテル コーポレーション ソフトウェア開発エンジニア**
**Alexey Budankov インテル コーポレーション ソフトウェア開発エンジニア**

アプリケーションのメモリーアクセスは、パフォーマンスに大きく影響します。スレッド化と**ベクトル化**によりアプリケーションを並列化するだけでは十分ではありません。メモリー帯域幅も重要ですが、多くの場合、ソフトウェア開発者はそのことをよく理解していません。メモリー・レイテンシーを最小限に抑え、帯域幅の向上を支援するツールを利用することで、パフォーマンス・ボトルネックをピンポイントで特定し、その原因を診断することができます。

最近のプロセッサーは、多様なメモリーアクセスを行います。例えば、L1 キャッシュヒットのレイテンシーは、すべてのキャッシュをミスして DRAM にアクセスしなければならない場合のレイテンシーとは大きく異なります。不均等メモリーアクセス (NUMA) アーキテクチャーではさらに複雑さが増します。

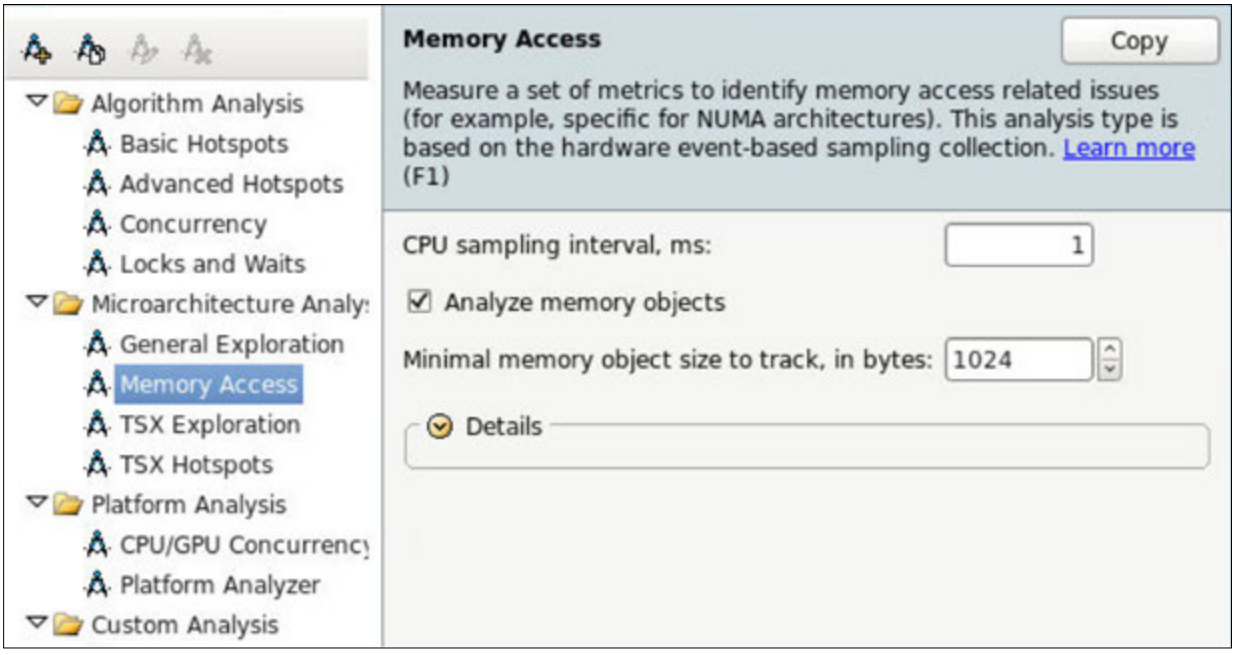
インテル® VTune™ Amplifier XE は、メモリーアクセスの解析を支援するさまざまな機能を備えたパフォーマンス・プロファイラーです。新しいメモリーアクセス解析タイプに含まれる各種機能を利用して、次のことが可能です。

- メモリー階層で**パフォーマンスの問題を検出** (L1-、L2-、LLC-、DRAM-バウンドなど)。
- **メモリー・オブジェクトを追跡**し、そのレイテンシーを適切なコードとデータ構造に関連付けます。
- **帯域幅により制限されたアクセスを解析**し (DRAM とインテル® QuickPath インターコネクト [インテル® QPI] の帯域幅を含む)、帯域幅をプログラムのタイムラインで表示した DRAM およびインテル® QPI のグラフとヒストグラムを素早く確認できます。
- パフォーマンスに影響する **NUMA 関連の問題**を特定できます。

この記事では、新しいメモリーアクセス機能の概要と、この機能によりいくつかの困難なメモリーの問題を解決し、アプリケーションのパフォーマンスを大幅に向上する方法を説明します。

## はじめに

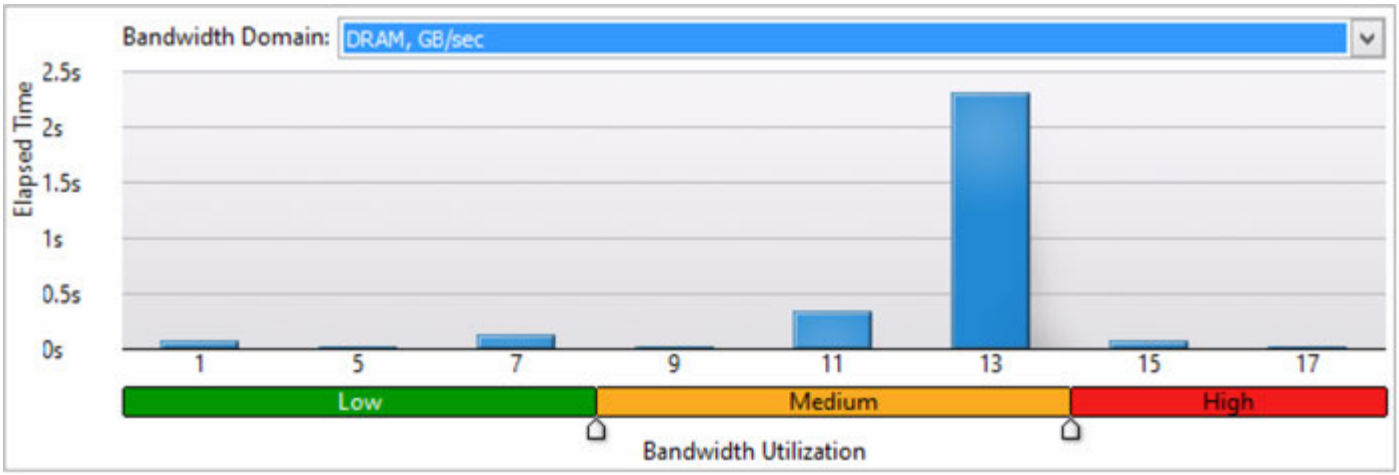
インテル® VTune™ Amplifier XE のメモリーアクセス機能を使用するには、新しい [Memory Access (メモリーアクセス)] 解析タイプをクリックして、[Start (開始)] をクリックします (**図 1**)。




1 メモリーアクセス機能を利用する

### 帯域幅使用率を表示する

DRAM およびインテル® QPI の帯域幅の使用状況を確認できます。帯域幅の使用率が高い箇所に注目します。この問題を修正するには、帯域幅に影響しているコード領域を見つけます (**図 2**)。




2 帯域幅ヒストグラム

### 帯域幅に影響するメモリー・オブジェクトを確認する

帯域幅に影響しているソースコードとメモリー・オブジェクトを特定します。帯域幅ドメインでグループ化すると、メモリー帯域幅に最も影響しているメモリー・オブジェクトを特定できます (**図 3**)。DRAM やインテル® QPI 関連の問題を含むコード領域を確認できます。

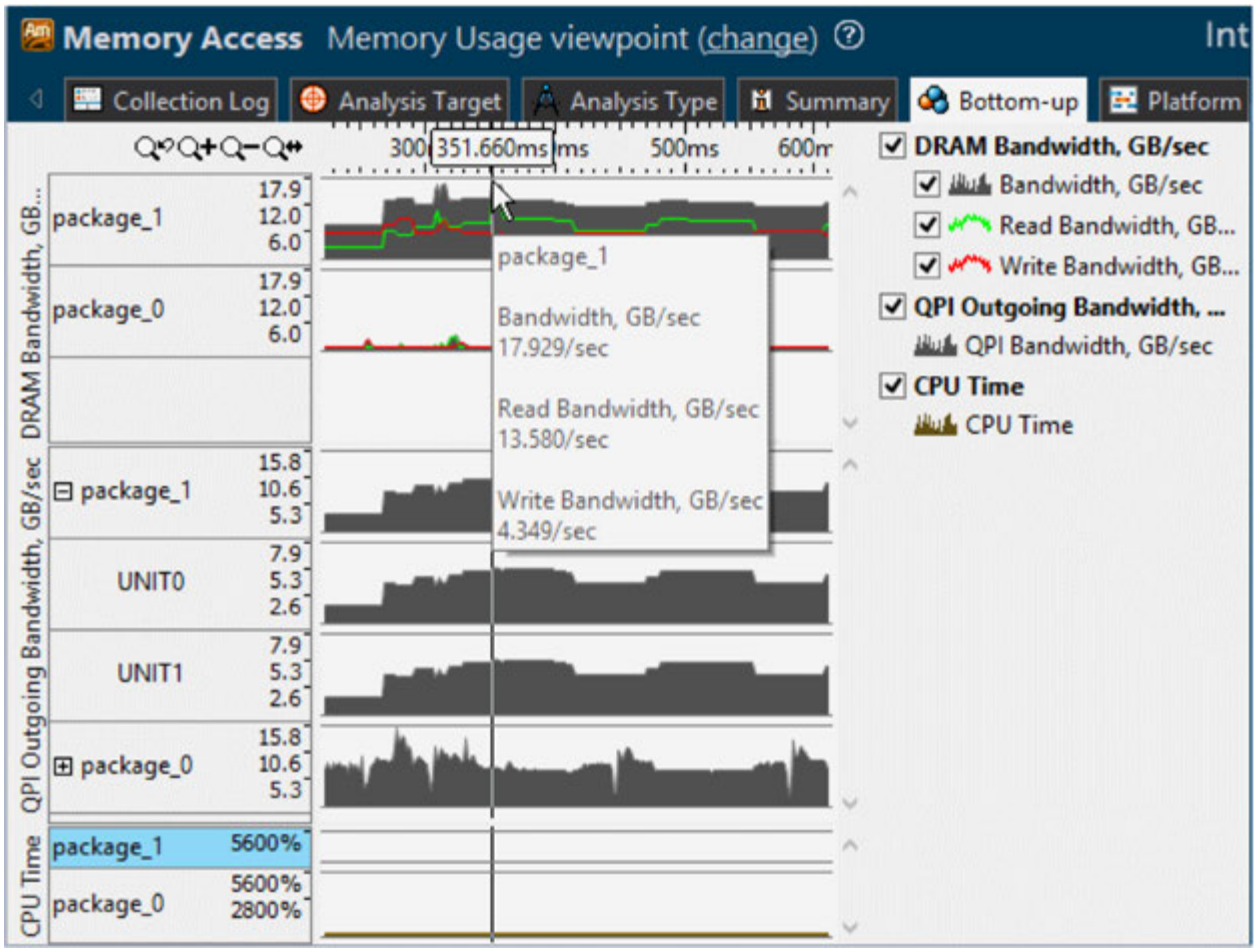
Grouping: Bandwidth Domain / Bandwidth Utilization Type / Memory Object / Allocation Stack

| Bandwidth Domain / Bandwidth Utilization Type / Memory Object / Allocation Stack | CPU Time | Memory Bound | Loads | Stores | LLC Miss Count | Average Latency (cycles) |
|---|---|---|---|---|---|---|
| DRAM, GB/sec | 2.873s | 0.818 | 1,764,005,292 | 846,012,690 | 37,202,232 | 195 |
| High | 2.291s | 0.924 | 1,176,003,528 | 601,209,018 | 33,602,016 | 196 |
| lin_stream.cpp:99 (152 MI | | | 300,000,900 | 140,402,106 | 12,600,756 | 316 |
| lin_stream.cpp:100 (152 M | | | 426,001,278 | 261,603,924 | 12,000,720 | 199 |
| lin_stream.cpp:98 (152 MI | | | 444,001,332 | 193,202,898 | 9,000,540 | 73 |

3 メモリー帯域幅

### メモリー帯域幅をアプリケーションのタイムラインで表示する

メモリー帯域幅は、一般にプログラムの実行とともに変動します。読み取り/書き込みの帯域幅を GB/秒で表示したグラフ (**図 4**) から、アプリケーションのメモリー使用量が増加した場所を確認し、その領域に注目します。タイムラインでメモリー使用量が増加した場所を選択してフィルターし、その時間に動作していたコードを確認します。




4 メモリー使用状況

メモリー帯域幅に影響しているアプリケーションのコード領域を追跡する機能は非常に役立ちます。平均レイテンシーはメモリーアクセスをチューニングする際の重要なメトリックです。タイムライン・グラフで帯域幅を表示すると、アプリケーション実行時のメモリー使用量を簡単に区別することができます。インテル® VTune™ Amplifier XE の最新バージョンでは、帯域幅グラフがプラットフォームで可能な最大値に対して相対的に表示されるため、パフォーマンス向上の可能性を簡単に把握できます。

# メモリーの問題を解決する

### 困難な問題 1: フォルス・シェアリング

最初に、用語を説明しましょう。

**シェアリング (共有):** 複数のスレッドが同じメモリー位置にアクセスする場合、それらのスレッドはメモリーを「シェア」していると言います。最近のコンピューターの構造上、シェアリングはさまざまなパフォーマンス・ペナルティーを引き起こします。すべてのスレッド/コアがメモリーアドレスに格納されるものについてコミットし、競合に関連するすべてのキャッシュを同期しなければならないため、これらのパフォーマンス・ペナルティーが生じます。

**フォルス・シェアリング:** 2 つのスレッドが同じキャッシュライン上のメモリー位置にアクセスする場合に発生します。実際に同じメモリー位置をシェアしているわけではありませんが、メモリー参照が互いに近い位置にあるため、同じキャッシュライン上に格納されています。複数のスレッドでフォルス・シェアリングが発生すると、実際に同じメモリー位置をシェアしている場合と同様のパフォーマンス・ペナルティーが生じます (このパフォーマンス・ペナルティーは完全に不要なものと言えます)。

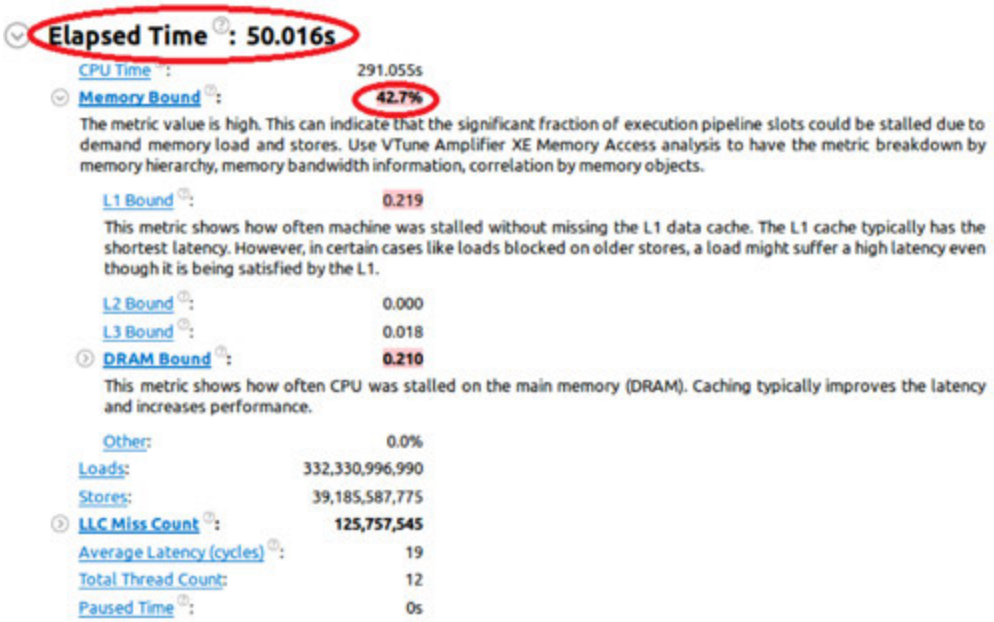
ここでは、Phoenix System* の linear_regression アプリケーションを検証します (**csl.stanford.edu/~christos/sw/phoenix**)。

#### ステップ 1: メモリーアクセス解析を実行して潜在的なメモリーの問題を明らかにする

次の設定で、メモリーアクセス解析を実行します。

- [Memory Objects Analysis (メモリー・オブジェクト解析)] を選択します。
- すべてのメモリー割り当てをキャプチャーするように、[Object Size Threshold (オブジェクト・サイズのしきい値)] を 1 に設定します。

[Summary (サマリー)] ビューに主要メトリックが表示されます (**図 5**)。

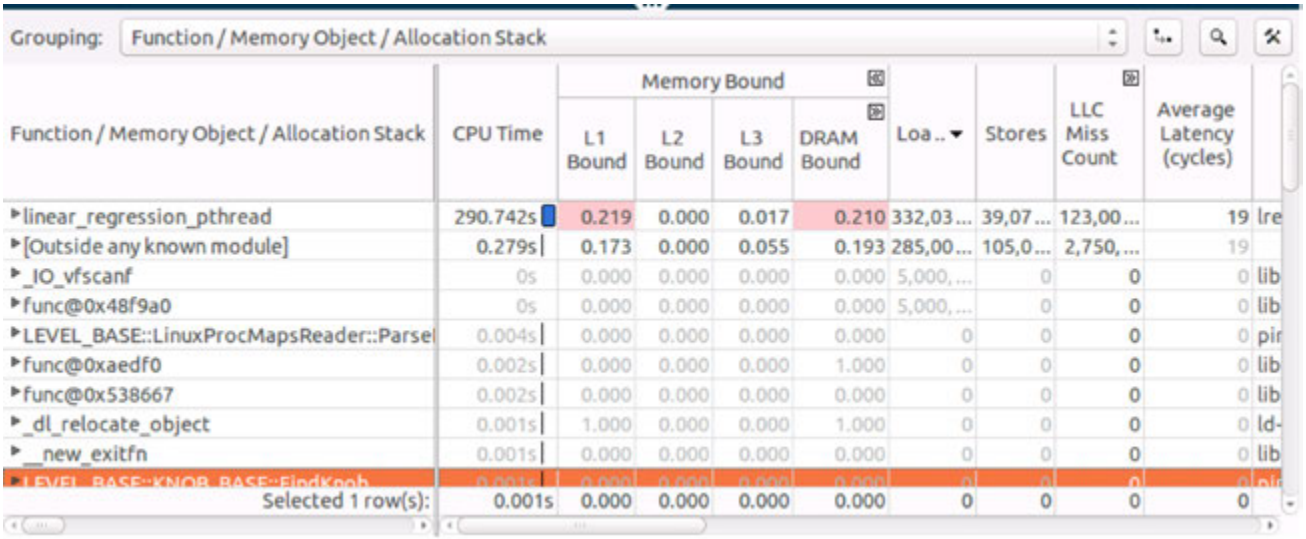



5 主要メトリック

最初の実行の経過時間は 50 秒です。また、アプリケーションが「メモリーバウンド (依存)」であり、メモリー操作の完了を待機するため、CPU の 42% 以上が無駄になっていることが分かります。Memory Bound (メモリーバウンド) メトリックはピンク色で表示されており、対応が必要なパフォーマンスの問題が存在する可能性を示しています。

## ステップ 2: 見つかったメモリーの問題を調査する

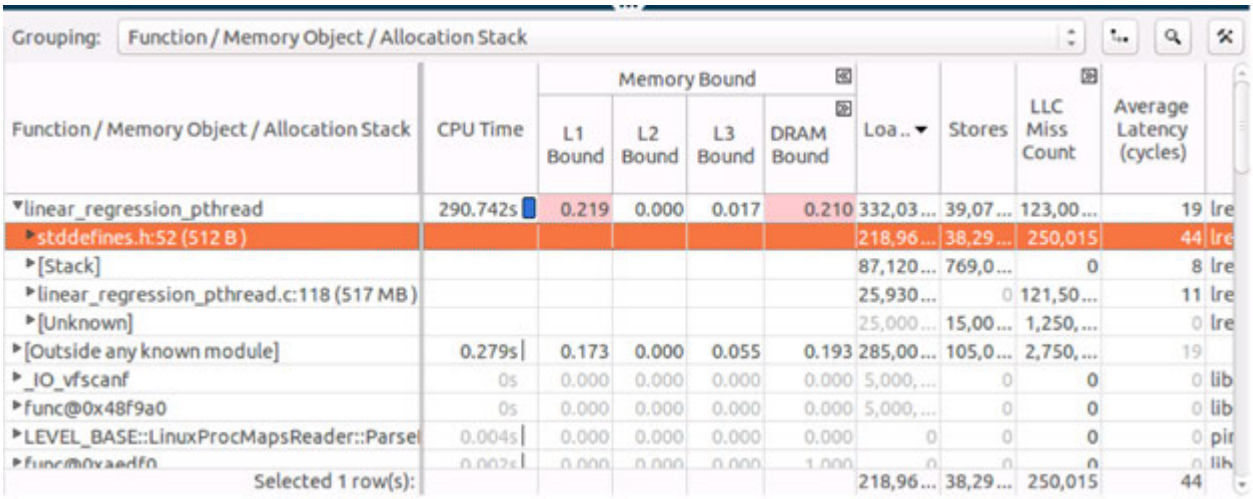
[Bottom-Up (ボトムアップ)] タブに切り替えて (**図 6**) 詳細を確認します。




6 [Bottom-Up (ボトムアップ)] タブ

実行時間のほとんどが、1 つの関数 linear_regression_pthread で費やされていることが分かります。この関数は、L1 と DRAM に依存しています。

linear_regression_pthread 関数の行を展開すると、アクセスしたメモリー・オブジェクトがロード数順に表示されます (**図 7**)。




**7** 行を展開してロード数順に表示

最も時間がかかっているオブジェクト—stddefines.h:52 (512B)—はたった 512 バイトで、L1 キャッシュに収めることができるはずですが、Average Latency (平均レイテンシー) メトリックは 44 サイクルになっています。これは、通常の L1 アクセス・レイテンシーである 4 サイクルをはるかに超えています。このような状況では、多くの場合、シェアリングまたはフォルス・シェアリングによる競合が発生している可能性があります。

"stddefines.h:52 (512B)" オブジェクトの割り当てスタックを確認すると (**図 8**)、オブジェクトの割り当てが行われたソースの位置が分かります。

```
127     num_threads = num_procs;
128
129     printf("Linear Regression P-Threads: Running...\n");
130
131
132     POINT_T *points = (POINT_T*)fdata;
133     long long n = (long long) finfo.st_size / sizeof(POINT_T);
134
135     req_units = n / num_threads;
136     tid_args = (lreg_args *)CALLOC(sizeof(lreg_args), num_procs);
```

**8** ソースの位置

この例では、num_procs はスレッド数で、割り当てられた構造体は lreg_args です。

```
typedef struct
{
    pthread_t tid;
    POINT_T *points;
    int num_elems;
    long long SX;
    long long SY;
    long long SXX;
    long long SYY;
    long long SXY;
} lreg_args;
```

スレッドは、次のように lreg_args 構造体にアクセスしています。

```
 // ADD UP RESULTS
for (i = 0; i < args->num_elems; i++)
{
   //Compute SX, SY, SYY, SXX, SXY
   args->SX  += args->points[i].x;
   args->SXX += args->points[i].x*args->points[i].x;
   args->SY  += args->points[i].y;
   args->SYY += args->points[i].y*args->points[i].y;
   args->SXY += args->points[i].x*args->points[i].y;
}
```

各スレッドは、配列の別々の要素にアクセスしているため、フォルス・シェアリングと考えられます。

### ステップ 3: フォルス・シェアリングを排除するようにコードを変更する

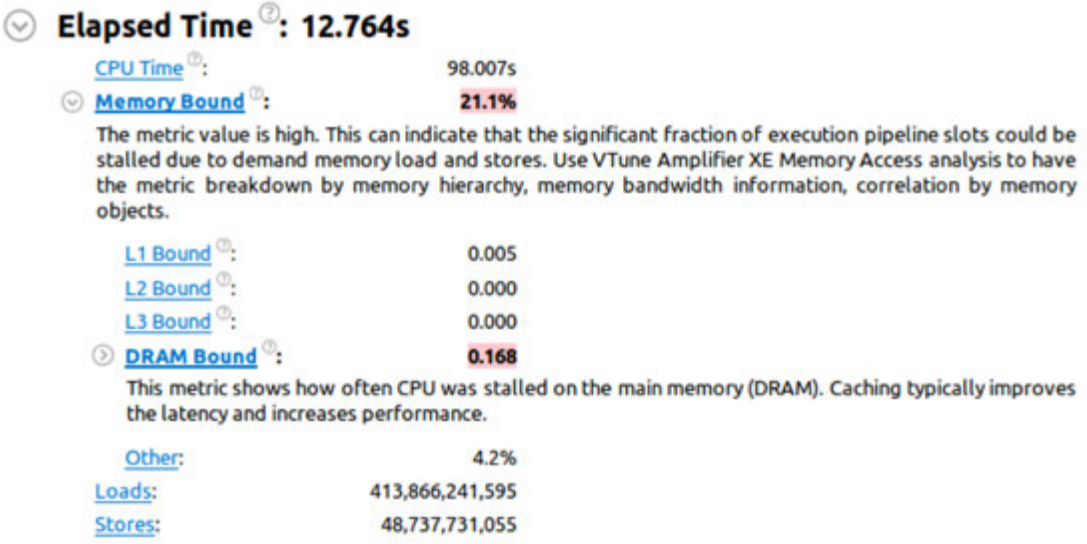
フォルス・シェアリングは通常、スレッドが異なるキャッシュラインにアクセスするように、パディングを追加することで簡単に回避できます。

次のように、lreg_args 構造体に **`char pad[80]`** フィールドを追加します。

```
typedef struct
{
    char pad[80];
    pthread_t tid;
    POINT_T *points;
    int num_elems;
    long long SX;
    long long SY;
    long long SXX;
    long long SYY;
    long long SXY;
} lreg_args;
```

### ステップ 4: メモリーアクセス解析を再度実行する

**図 9** は、新しい解析結果です。




9 経過時間

新しい経過時間は 12 秒です。構造体にパディングを追加する 1 行のコード変更で、アプリケーションのパフォーマンスを約 4 倍向上できたことになります。Memory Bound (メモリーバウンド) メトリックも小さくなり、L1 バウンドの問題も解決されました。

### 困難な問題 2: NUMA 問題

NUMA をサポートするプロセッサーでは、実行している CPU のキャッシュミスを把握するだけでは十分ではありません。NUMA アーキテクチャーでは、別の CPU のキャッシュと DRAM を参照することもできます。このようなアクセスのレイテンシーは、ローカルよりも大きくなります。これらのリモート・メモリー・アクセスを識別して最適化する必要があります。

ここでは、OpenMP* を使用して並列化された単純な Triad (積和演算) アプリケーションを、デュアルソケットのインテル® Xeon® プロセッサー上で実行して検証します。

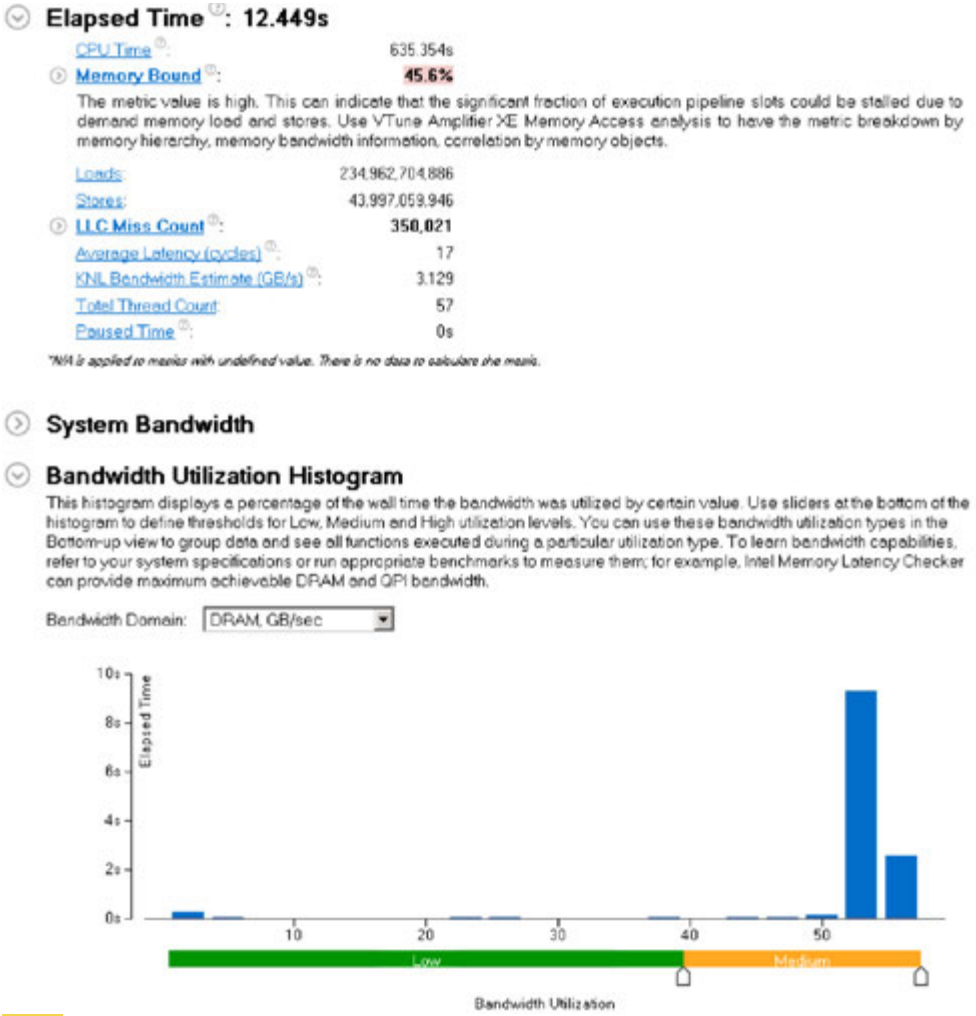
以下にコードを示します。

```
static double   a[N];
static double   b[N];
static double   c[N];

void doTriad(double x)
{
#pragma omp parallel for
    for (int i = 0; i < N; i++)
        a[i] = b[i] + x*c[i];
}

int main() {

    for (int i = 0; i < N; ++i)
    {
        a[i] = 1.0;
        b[i] = 2.0;
        c[i] = 0.0;
    }

    for (int n = 0; n < NTIMES; ++n)
    {
        doTriad(3.0);
    }

    return 0;
}
```

最初に、配列を初期化し、"omp parallel for" を使用する Triad 関数を呼び出します。

### ステップ 1: メモリーアクセス解析を実行する

アプリケーションのメモリーアクセス解析を実行します。このアプリケーションは、DRAM 帯域幅に依存していて、システムの帯域幅を最大限に利用していないことが予想されます (**図 10**)。




10 メモリーアクセス解析

サマリーセクションには、非常に有益なメトリックが含まれています。経過時間は 12.449 秒で、Memory Bound (メモリーバウンド) メトリックは高く、予想どおりハイライトされています。Bandwidth Utilization (帯域幅使用率) ヒストグラムは、DRAM 帯域幅の使用率が 50 ～ 60GB/秒と中程度であることを示しており、調査する必要があります。

その他にも、次のメトリックが役立ちます。

- **Average Latency (平均レイテンシー)。**メモリーアクセスにかかった平均サイクル数。
  注: L1 メモリーアクセスは通常 4 サイクルですが、リモート DRAM アクセスには約 300 サイクルかかります。
- **KNL Bandwidth Estimate (KNL 帯域幅の推定値)。**次世代のインテル® Xeon Phi™ コプロセッサー (開発コード名 Knights Landing [KNL]) 上で実行した場合のコアごとの帯域幅の推定値。このプラットフォームへの移行を検討している場合、コードのメモリーアクセス部分の対応状況を確認するのに役立ちます。

### ステップ 2: 帯域幅使用率を調査する

[Bottom-Up (ボトムアップ)] タブに切り替えて (**図 11**) 詳細を確認します。

11 [Bottom-Up (ボトムアップ)] タブ

タイムライン・グラフから、DRAM 帯域幅が 1 つのソケット package_1 でのみ使用されており、QPI (intra-socket) トラフィックが最大 30GB/秒と高いことが分かります。これは、NUMA マシンにおける典型的な問題で、メモリーは 1 つのノードに割り当てられ、ワークは複数のノードで分割されています。この場合、一部のノードはローカル・メモリー・アクセスよりも非常に遅い QPI リンクを利用して、データをリモートでロードしなければなりません。

## BLOG HIGHLIGHTS

### Knights Landing (開発コード名) 上の MCDRAM (高帯域幅メモリー) の概要

MIKE PEARCE ›

次世代のインテル® Xeon Phi™ コプロセッサー x200 製品ファミリー (開発コード名 Knights Landing) には、従来の DDR4 に加えて、新しいメモリー・テクノロジーである MCDRAM (Multi-Channel DRAM) と呼ばれる高帯域幅のオンパッケージ・メモリーが搭載されています。MCDRAM は高帯域幅 (DDR4 の 4 倍以上高速)、低容量 (最大 16GB) のメモリーで、Knights Landing (開発コード名) シリコンに含まれています。MCDRAM は L3 キャッシュ (メモリー・サイド・キャッシュ)、個別の NUMA ノード (割り当て可能なメモリー)、あるいはその中間として構成することができます。異なるメモリーモードでシステムを起動できるため、ソフトウェア側でアプリケーションに最適なモードを理解することは困難です。

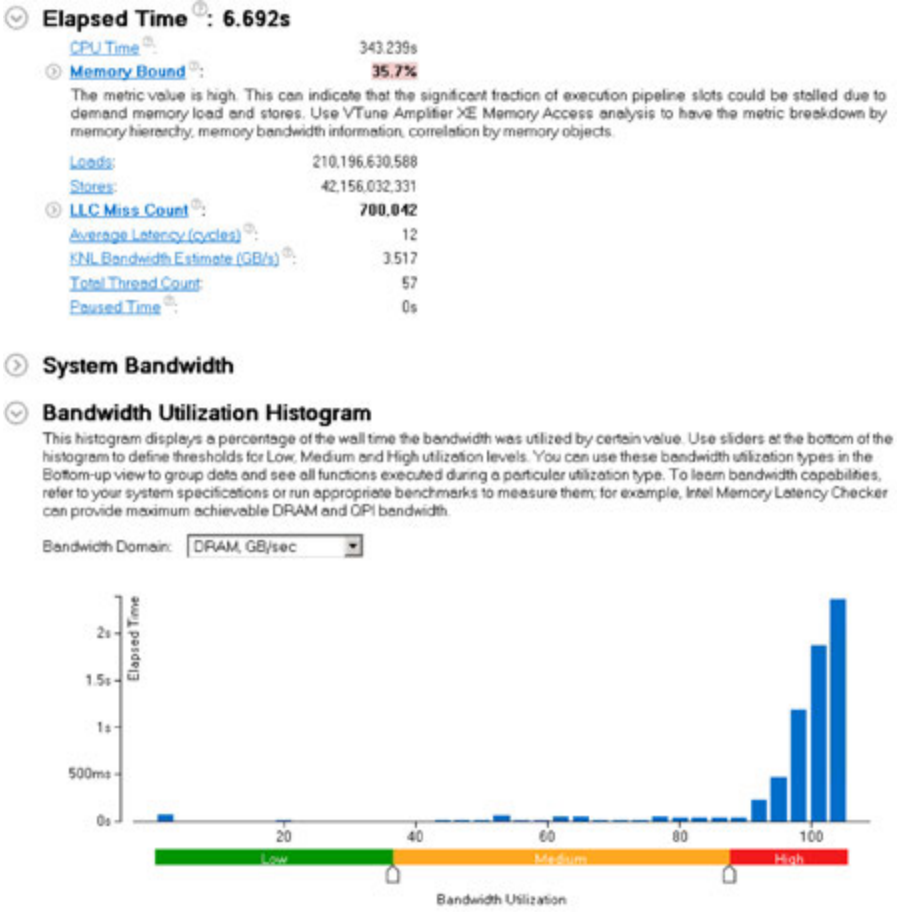
**この記事の続きはこちらでご覧になれます。 ›**

### ステップ 3: リモート・メモリー・アクセスを回避するようにコードを変更する

両方のソケットがローカルメモリーにのみアクセスするようにコードを変更して、リモート・ノード・アクセスを回避すれば、より高速に実行できるはずです。Linux* では、メモリーページは最初のアクセス時に割り当てられます。そのため、メモリーアクセスが発生するノードでメモリーを初期化することで、簡単に問題を解決できます。初期化ループに **`omp parallel for`** プラグマを追加します。

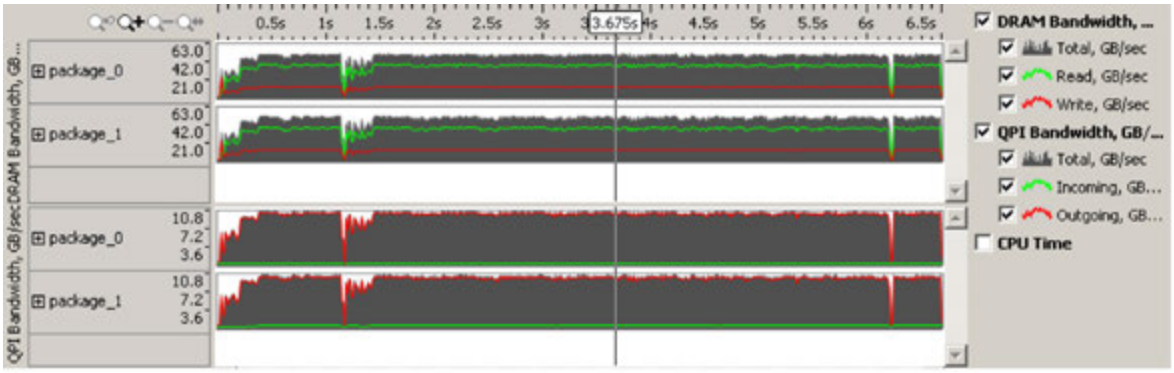
```
static double   a[N];
static double   b[N];
static double   c[N];

void doTriad(double x)
{
#pragma omp parallel for
    for (int i = 0; i < N; i++)
        a[i] = b[i] + x*c[i];
}

int main() {

#pragma omp parallel for
    for (int i = 0; i < N; ++i)
    {
        a[i] = 1.0;
        b[i] = 2.0;
        c[i] = 0.0;
    }

    for (int n = 0; n < NTIMES; ++n)
    {
        doTriad(3.0);
    }

    return 0;
}
```

## ステップ 4: KMP_AFFINITY 変数を指定してメモリー解析を再度実行する




**12** 帯域幅使用率

経過時間が 12.449 秒から 6.69 秒に減りました。これは、ほぼ 2 倍のスピードアップです。また、予想どおり DRAM と帯域幅の使用率が高くなりました (**図 12**)。




**13** [Bottom-Up (ボトムアップ)] タブ

帯域幅は 2 つのソケットで等分割され、QPI トラフィックは 1/3 になりました (**図 13**)。

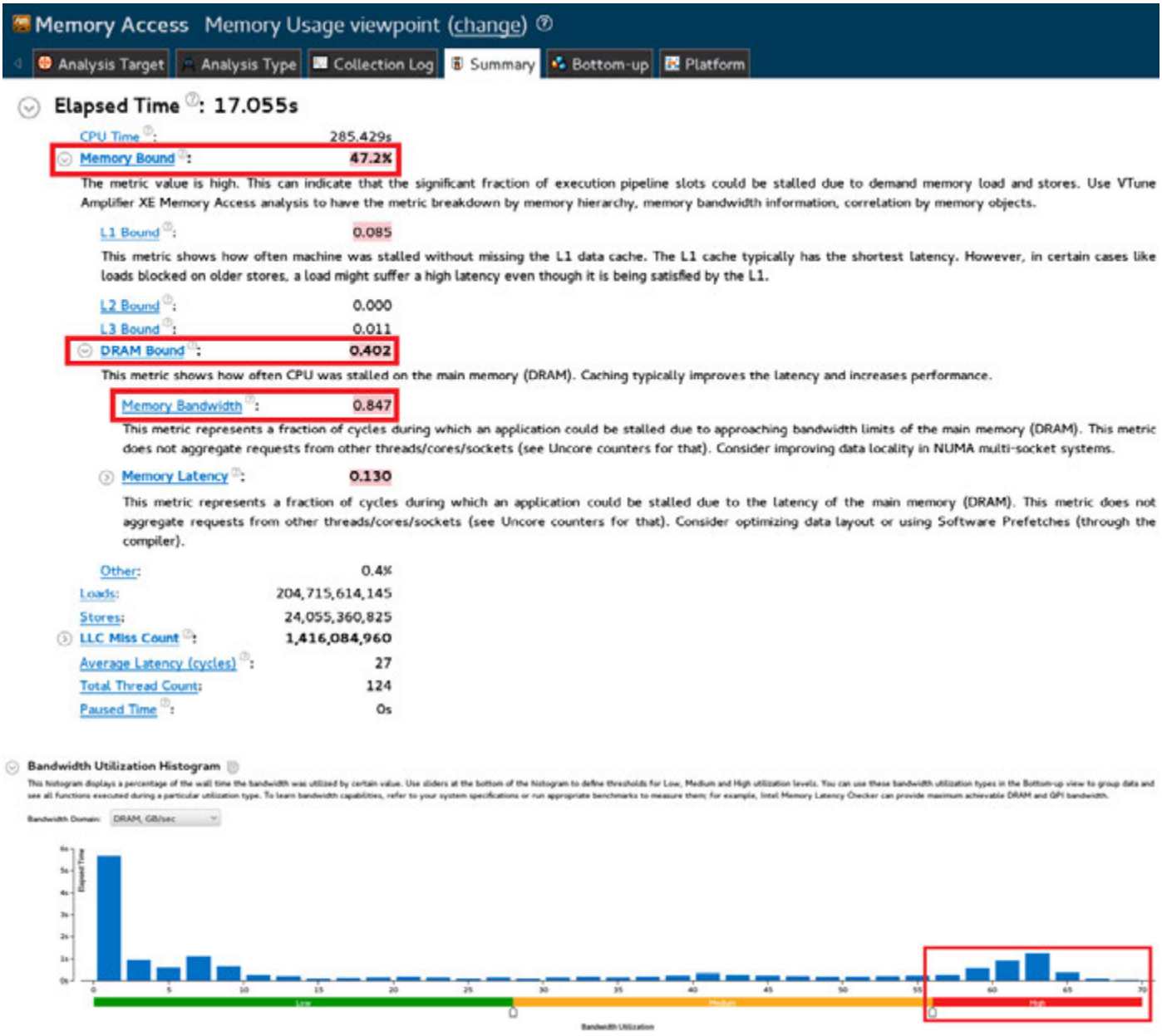
NUMA アーキテクチャーは複雑なため、メモリーアクセスに細心の注意が必要です。アプリケーションの最もレイテンシーが大きいメモリーアクセスを最適化すると、最もパフォーマンスを向上することができます。

### 困難な問題 3: 次世代のインテル® Xeon Phi™ コプロセッサー向けに最適化する

メモリー帯域幅によりアプリケーションのパフォーマンスが制限されることはよくあります。新しい世代のインテル® Xeon Phi™ コプロセッサーは、物理帯域幅の制限を緩和するため、特殊なオンパッケージの MCDRAM メモリーを搭載しています。MCDRAM メモリーは、高帯域幅を実現します。また、従来の DRAM (DDR4) メモリーも搭載されており、90GB/秒でアクセスできます。ただし、MCDRAM メモリーは容量が小さいため、その利点を最大限に得られるように配置するデータ・オブジェクトを決定することが重要です。

#### ステップ 1: メモリーアクセス解析を実行する

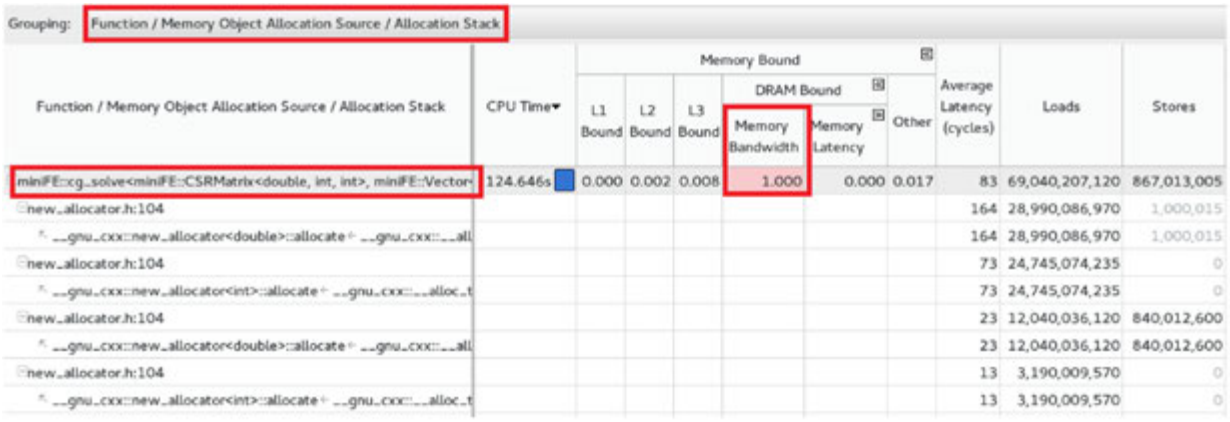
この例では、Mantevo Suite* の miniFE* ベンチマーク (**mantevo.org/**) を使用します。インテル® VTune™ Amplifier XE のメモリーアクセス解析機能を利用して、ベンチマークを変更せずにそのままプロファイルしたところ、ベンチマーク・コードはメモリー依存であることが分かりました。これは、従来のメモリーでは高トラフィックになり、MCDRAM の高帯域幅により利点が得られる可能性があることを意味します (**図 14**)。




**14** メモリーアクセス

## ステップ 2: メモリー割り当て関連の帯域幅を調査する

[Bottom Up (ボトムアップ)] ビューで [Function/Memory Object/Allocation Stack (関数/メモリー・オブジェクト/割り当てスタック)] グループをメモリーアクセス解析結果に適用すると、ほとんどのメモリーアクセスは miniFE::CSRMatrix クラスと miniFE::Vector クラスのオブジェクトで発生していることが分かります (**図 15**)。

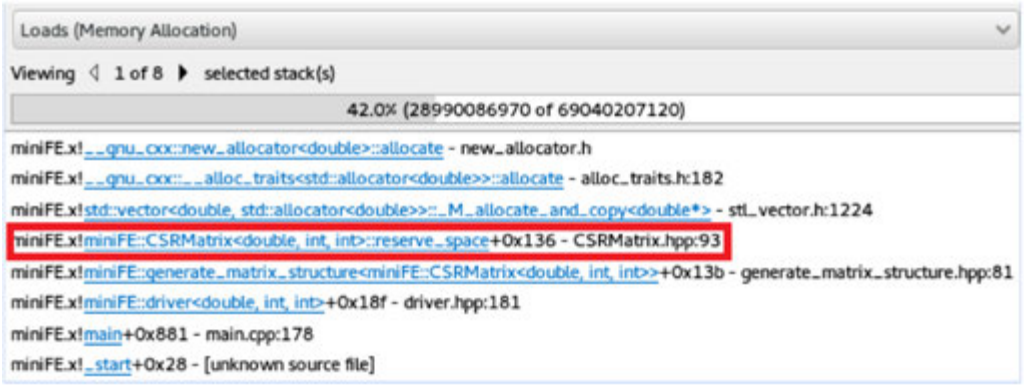


Grouping: Function / Memory Object Allocation Source / Allocation Stack

| Function / Memory Object Allocation Source / Allocation Stack | CPU Time | Memory Bound | | | | | | Average Latency (cycles) | Loads | Stores |
|---|---|---|---|---|---|---|---|---|---|---|
| | | L1 Bound | L2 Bound | L3 Bound | DRAM Bound: Memory Bandwidth | DRAM Bound: Memory Latency | Other | | | |
| miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Vector< | 124.646s | 0.000 | 0.002 | 0.008 | 1.000 | 0.000 | 0.017 | 83 | 69,040,207,120 | 867,013,005 |
| new_allocator.h:104 | | | | | | | | 164 | 28,990,086,970 | 1,000,015 |
| __gnu_cxx::new_allocator<double>::allocate | | | | | | | | 164 | 28,990,086,970 | 1,000,015 |
| new_allocator.h:104 | | | | | | | | 73 | 24,745,074,235 | 0 |
| __gnu_cxx::new_allocator<int>::allocate | | | | | | | | 73 | 24,745,074,235 | 0 |
| new_allocator.h:104 | | | | | | | | 23 | 12,040,036,120 | 840,012,600 |
| __gnu_cxx::new_allocator<double>::allocate | | | | | | | | 23 | 12,040,036,120 | 840,012,600 |
| new_allocator.h:104 | | | | | | | | 13 | 3,190,009,570 | 0 |
| __gnu_cxx::new_allocator<int>::allocate | | | | | | | | 13 | 3,190,009,570 | 0 |

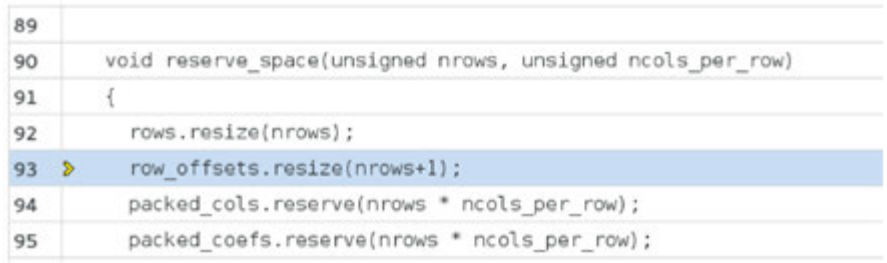
15 [Function/Memory Object/Allocation Stack (関数/メモリー・オブジェクト/割り当てスタック)] グループ

さらに、スタックペインにある miniFE::CSRMatrix クラスデータの完全なコールスタック割り当て情報から、割り当て操作のソースコードの位置を確認できます (**図 16**)。




16 スタックペイン

CSRMatrix.hpp:93 の行をクリックすると、ソースビューが開き CSRMatrix.hpp ファイルの行 93 にある割り当て操作が表示されます (**図 17**)。




17 ソースビュー

miniFE::CSRMatrix クラスデータは、格納されるベクトル要素にカスタム・メモリー・アロケーター・クラスを指定できる、柔軟性のある STL ベクトルコンテナーにより管理されます。

```
template<typename Scalar,
         typename LocalOrdinal,
         typename GlobalOrdinal,
         typename ComputeNode>
struct CSRMatrix {
…
  typedef Scalar         ScalarType;
  typedef LocalOrdinal   LocalOrdinalType;
  typedef GlobalOrdinal  GlobalOrdinalType;
  typedef ComputeNode    ComputeNodeType;

  bool                        has_local_indices;
  std::vector<GlobalOrdinal>  rows;
  std::vector<LocalOrdinal>   row_offsets;
  std::vector<LocalOrdinal>   row_offsets_external;
  std::vector<GlobalOrdinal>  packed_cols;
  std::vector<Scalar>         packed_coefs;
  LocalOrdinal                num_cols;
  ComputeNode&                compute_node;
…
}
```

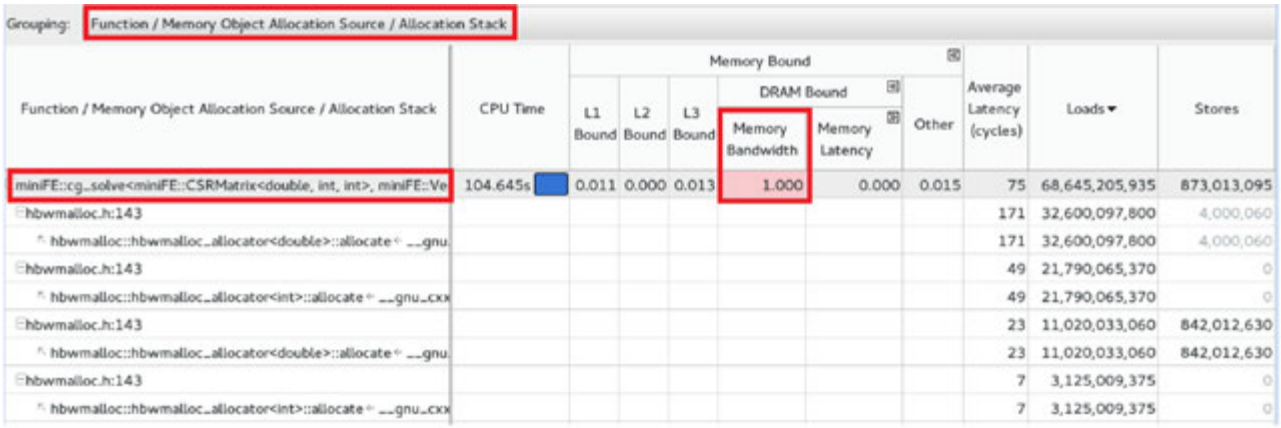
### ステップ 3: 高帯域幅メモリーを使用してオブジェクトを割り当てる

memkind ライブラリー API (**https://github.com/memkind/memkind**) を利用して、miniFE::CSRMatrix と miniFE::Vector データ・オブジェクトを MCDRAM メモリーに移動します。hbwmalloc.h ヘッダーファイルは、MCDRAM (高帯域幅) メモリーで STL ベクトルコンテナーのパラメーター化に使用される hbwmalloc::hbwmalloc_allocator クラスの実装を提供します。

ここでは、次のようにコードを変更します。

```
…
#include “/opt/mk-0.3.0/include/hbwmalloc.h”
…
template<typename Scalar,
         typename LocalOrdinal,
         typename GlobalOrdinal,
         typename ComputeNode>
struct CSRMatrix {
…
  typedef Scalar         ScalarType;
  typedef LocalOrdinal   LocalOrdinalType;
  typedef GlobalOrdinal  GlobalOrdinalType;
  typedef ComputeNode    ComputeNodeType;

  bool                        has_local_indices;
  std::vector<GlobalOrdinal, hbwmalloc::hbwmalloc_allocator<GlobalOrdinal> > rows;
  std::vector<LocalOrdinal, hbwmalloc::hbwmalloc_allocator<GlobalOrdinal> >  row_offsets;
  std::vector<LocalOrdinal, hbwmalloc::hbwmalloc_allocator<GlobalOrdinal> >  row_offsets_
external;
  std::vector<GlobalOrdinal, hbwmalloc::hbwmalloc_allocator<GlobalOrdinal> > packed_cols;
  std::vector<Scalar, hbwmalloc::hbwmalloc_allocator<GlobalOrdinal> >        packed_coefs;
  LocalOrdinal                num_cols;
  ComputeNode&                compute_node;
…
}
```

変更したソースコードをリビルドして、メモリーアクセス解析を再度実行すると、miniFE::CSRMatrix と miniFE::Vector データ・オブジェクトが memkind ライブラリーの hbwmalloc::hbwmalloc_allocator クラスによって作成されていることが分かります (**図 18**)。



Grouping: Function / Memory Object Allocation Source / Allocation Stack

| Function / Memory Object Allocation Source / Allocation Stack | CPU Time | Memory Bound | | | | | | Average Latency (cycles) | Loads | Stores |
|---|---|---|---|---|---|---|---|---|---|---|
| | | L1 Bound | L2 Bound | L3 Bound | DRAM Bound: Memory Bandwidth | DRAM Bound: Memory Latency | Other | | | |
| miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Ve | 104.645s | 0.011 | 0.000 | 0.013 | 1.000 | 0.000 | 0.015 | 75 | 68,645,205,935 | 873,013,095 |
| hbwmalloc.h:143 | | | | | | | | 171 | 32,600,097,800 | 4,000,060 |
| hbwmalloc::hbwmalloc_allocator<double>::allocate __gnu | | | | | | | | 171 | 32,600,097,800 | 4,000,060 |
| hbwmalloc.h:143 | | | | | | | | 49 | 21,790,065,370 | 0 |
| hbwmalloc::hbwmalloc_allocator<int>::allocate __gnu_cxx | | | | | | | | 49 | 21,790,065,370 | 0 |
| hbwmalloc.h:143 | | | | | | | | 23 | 11,020,033,060 | 842,012,630 |
| hbwmalloc::hbwmalloc_allocator<double>::allocate __gnu | | | | | | | | 23 | 11,020,033,060 | 842,012,630 |
| hbwmalloc.h:143 | | | | | | | | 7 | 3,125,009,375 | 0 |
| hbwmalloc::hbwmalloc_allocator<int>::allocate __gnu_cxx | | | | | | | | 7 | 3,125,009,375 | 0 |

18 メモリーアクセス解析

## ステップ 4: ベンチマークを再度実行する

変更したバージョンのベンチマークをインテル® Xeon Phi™ コプロセッサー上の MCDRAM メモリーを利用して実行すると、従来の DRAM メモリーに利用頻度の高いデータを割り当てるオリジナルバージョンのほぼ 4 倍高速になりました。

```
[vtune@nntvtune46 src]$ /usr/bin/time /tmp/miniFE-2.0_openmp_ref_ORIG/src/miniFE.x.sh
MiniFE Mini-App, OpenMP Peer Implementation
Creating OpenMP Thread Pool...
Counted: 12 threads.
Running MiniFE Mini-App...
      creating/filling mesh...0.197327s, total time: 0.197329
generating matrix structure...13.5858s, total time: 13.7832
         assembling FE data...13.3513s, total time: 27.1345
      imposing Dirichlet BC...2.61192s, total time: 29.7464
      imposing Dirichlet BC...1.11535s, total time: 30.8617
making matrix indices local...1.19209e-06s, total time: 30.8617
Starting CG solver ...
Initial Residual = 201.001
Iteration = 20   Residual = 0.0609161
...
Iteration = 200   Residual = 0.00112011
Final Resid Norm: 0.00112011
2671.76user 9.53system 3:55.02elapsed 1140%CPU (0avgtext+0avgdata 1511308maxresident)
k0inputs+8outputs (0major+49614minor)pagefaults 0swaps

[vtune@nntvtune46 src]$ /usr/bin/time /tmp/miniFE-2.0_openmp_ref_KNL/src/miniFE.x.sh
MiniFE Mini-App, OpenMP Peer Implementation
Creating OpenMP Thread Pool...
Counted: 12 threads.
Running MiniFE Mini-App...
      creating/filling mesh...0.198685s, total time: 0.198686
```

```
generating matrix structure...13.9371s, total time: 14.1358
         assembling FE data...13.1823s, total time: 27.3181
      imposing Dirichlet BC...2.51502s, total time: 29.8331
      imposing Dirichlet BC...1.10896s, total time: 30.942
making matrix indices local...9.53674e-07s, total time: 30.942
Starting CG solver ...
Initial Residual = 201.001
Iteration = 20   Residual = 0.0609161
…
Iteration = 200   Residual = 0.00112011
Final Resid Norm: 0.00112011
475.87user 2.09system 0:52.25elapsed 914%CPU (0avgtext+0avgdata 2598752maxresident)
k0inputs+8outputs (0major+23143minor)pagefaults 0swaps
```

この例は、帯域幅依存のコードは、最新のインテル® Xeon Phi™ コプロセッサー・プラットフォームで利用可能な MCDRAM メモリーに利用頻度の高いデータを配置することで利点が得られ、memkind ライブラリーを利用することでその作業を簡単に行うことができることを示しています。

## まとめ

プログラムのメモリーアクセスを最適化することは極めて重要です。インテル® VTune™ Amplifier XE のようなツールを使って、プログラムがどのようにメモリーにアクセスしているか理解することは、ハードウェア機能を最大限に利用するのに役立ちます。

ここでは、インテル® VTune™ Amplifier XE のメモリーアクセス解析機能の概要を紹介し、この機能を使っていくつかの困難なメモリーの問題を解決する方法を示しました。

比較的小さなメモリー・オブジェクトで平均レイテンシーの値が高い場合にフォルス・シェアリング問題を検出する方法を紹介し、パディングを追加するたった 1 行のコード変更によりアプリケーションのパフォーマンスを 4 倍向上しました。

大量のリモート・メモリー・アクセスを含む NUMA 問題を検出する方法を説明し、それらのリモートアクセスを排除することでアプリケーションのパフォーマンスを 2 倍向上しました。

最後に、最新のインテル® Xeon Phi™ コプロセッサー・プラットフォームで利用可能なメモリー・テクノロジーにより利点が得られるコード領域を特定することで、ベンチマーク・アプリケーションを 4 倍スピードアップできることを示しました。