並列アプリケーションのスケーラビリティーの問題を特定する

インテル® VTune™ Amplifier XE のメモリー解析を使用してインテル® Xeon® プロセッサーおよびインテル® Xeon Phi™ プロセッサー向けにスケーラビリティーを向上する方法

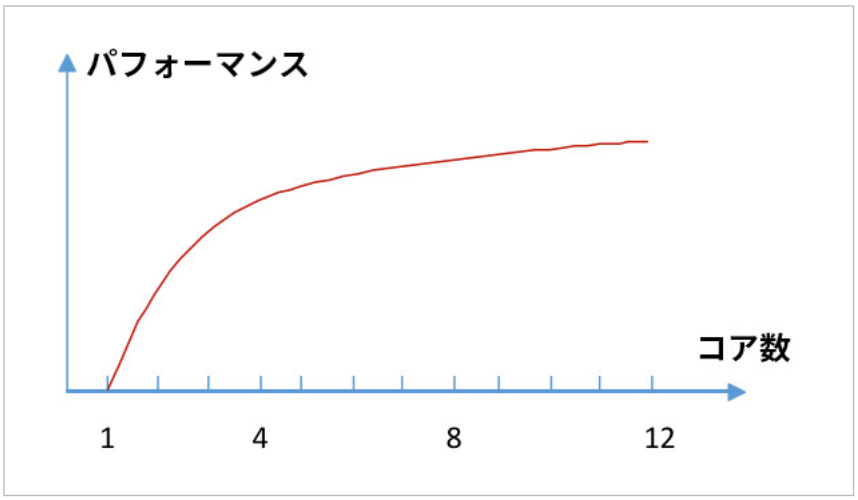
Vladimir Tsymbal インテル コーポレーション ソフトウェア・テクニカル・コンサルティング・エンジニア

システムの計算コア数が増え続ける中、効率良く並列化されたソフトウェアでは、コア数の増加に伴ってパフォーマンスが直線的に向上することが望まれます。しかし、いくつかの要因により、マルチコアシステムの並列性とスケーラビリティーが妨げられることが分かっています。この記事では、ほとんどのケースで原因となる、並列処理の実装について取り上げます。

- **ロードインバランス**により、スレッドと CPU コアがアイドル状態になります。
- **過度の同期**により、スピン待機やその他の非生産的なワークに CPU 時間が費やされます。
- **並列ランタイム・ライブラリーのオーバーヘッド**が、適切でないライブラリー API の使用により発生します。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

これらの要因を排除することで、並列処理の効率が大幅に向上し、すべての CPU コアが有用なワークを実行するためビジー状態になります。STREAM や LINPACK のような最適なチューニングが行われているベンチマークでは、ほぼ直線的なスピードアップを達成できます。しかし、システムのコア数の増加に伴い (または、コア数の多い新しいシステムでコードを実行した場合に)、アプリケーションのパフォーマンスが直線的に向上しなかったり、並列パフォーマンスが伸び悩むことがあります (**図 1**)。



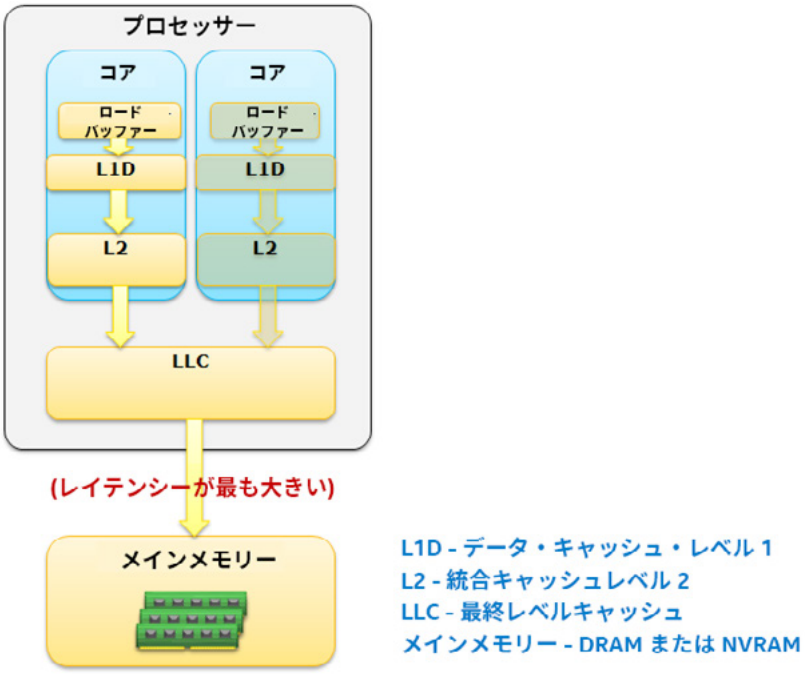
1 コア数とパフォーマンスの変化

トップダウンのパフォーマンス解析アプローチ [1] では、最初にほかのコンポーネントによりパフォーマンスが妨げられていないかチェックすべきです。次のことを確認します。

- **ほかのプロセスによってシステムが常にビジー状態でない**: リソースを消費するほかのアプリケーションやサービスに計算時間が費やされていることがあります。
- **アプリケーションがシステムの I/O に依存していない**: 例えば、ディスク、ほかのファイルシステム、ネットワーク・システムの処理が完了するのを待っていないか確認します。
- **システムに十分な物理メモリー容量がある**: ハードディスク・ドライブとのスワップが頻発するのを防ぎます。

一般に、ハードウェアが適切に構成され、メモリー・サブシステムが想定するパフォーマンス特性を提供しているかどうか把握しておくと良いでしょう。例えば、すべてのメモリースロットにマザーボードの特性 (チャネル数、メモリー速度など) に対応した DIMM が搭載されている場合、一般的なベンチマークを使用してハードウェアのパフォーマンスを簡単にチェックできます。問題の修正は、ソフトウェアを最適化するよりもハードウェアで行ったほうが容易なため、このようなチェックを行うことが重要です。

これらのチェックが終わったら、並列処理がスケーリングしない主な原因の 1 つであるメモリー・レイテンシーを確認します。x86 システム・アーキテクチャーでは、CPU はキャッシュ・サブシステムからデータを読み取ります。命令で必要なときに、CPU に最も近いキャッシュ (L1 データキャッシュ) にデータがあることが理想的です (**図 2**)。要求されたデータが CPU から離れた場所にある場合、データが CPU コアの実行ユニットに到達するのに長い時間がかかります。CPU ハードウェア・プリフェッチャーは、迅速にデータを取得できるように支援しますが、常に適切に動作するとは限りません。データの遅延はよくあることで、CPU のストールを引き起こします。[2]



基本的に、データが遅延する理由は 2 つあります。

1. CPU の実行ユニットで実行される命令がデータを要求し、プリフェッチャーが適切に動作しない場合、メインメモリーやほかのキャッシュから CPU の L1D にデータが読み込まれます。これにより、メモリー・レイテンシーの問題が生じます。
2. プリフェッチャーが適切に動作し、データが事前に要求されても、転送インフラストラクチャーの処理能力が原因で CPU への転送が遅れることがあります。この場合、メモリー帯域幅の問題が生じます。

2 メモリー・サブシステムからのデータの読み取り

複数のソースから複数の要求があった場合、両方の問題が発生する可能性があります。これらの問題を回避するには、データを上手く利用することが重要です。メモリー・レイテンシーの問題を解決するには、アドレスを使用してインクリメンタルにデータにアクセスします。シーケンシャル・データ・アクセス (または一定の短い距離のユニットストライド) を使用すると、プリフェッチャーによって適切に処理され、迅速にデータにアクセスできます。メモリー帯域幅の問題を解決するには、データを再利用し、できる限りキャッシュに保持するようにします。そのためには、データ・アクセス・パターンの見直しや、場合によってはアルゴリズム全体の実装の見直しが必要になります。

アプリケーションのスケーラビリティーを妨げる要因

CPU でコードが効率良く実行されず、ほとんどのストールがメモリー依存であることが分かったら、問題の具体的な原因を明らかにする必要があります。問題への対応方法は、原因がメモリー・レイテンシーによるものか、メモリー帯域幅によるものかに応じて異なります。ここでは、**インテル® VTune™ Amplifier XE** の組込みメモリーアクセス解析を使用して、メモリーの問題を詳しく調査します。

単純な行列乗算ベンチマークを向上するいくつかの反復について考えてみます。この例は単純ですが、アルゴリズムの実装方法に応じてメモリーの問題が発生する可能性を効果的に示しています。パフォーマンスの測定には、**インテル® Xeon® プロセッサー E5-2697 v4** (開発コード名 Broadwell、36 コア) ベースのシステムを使用します。このシステムの理論メモリー帯域幅は 76.8GB/ 秒で、1 秒あたりの倍精度 (DP) 浮動小数点演算の回数 (FLOPS) は 662GFLOPS/ 秒です。

行列乗算アルゴリズムのナイーブ実装

ナイーブ行列乗算実装 (**図 3** の multiply1) は、CPU コア数が多い場合は直線的にスケーリングしません。しかし、パフォーマンスが悪い原因の特定方法を示す良い例です。ここでは、**ベクトル化**が行われるように、–no-alias コンパイラー・オプションを追加します。そうしないと、スカラー実装のパフォーマンスは約 1/10 に低下します。サイズ 9216 x 9216 でベクトル化されているベンチマーク (multiply1) を実行した結果が**表 1** です。最高のパフォーマンスが理論最大 FLOPS を大きく下回っていることが分かります。

```
void multiply2(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE c[][NUM], TYPE t[]
[NUM])
{
  int i,j,k;

// ループ交換
  for(i=tidx; i<msize; i=i+numt) {
    for(k=0; k<msize; k++) {
#pragma ivdep
        for(j=0; j<msize; j++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }}}}
```

3 行列乗算アルゴリズムの最適化された実装 (multiply2)

スレッド数	実行時間 (秒)	DP FLOPS (GFLOPS/ 秒)
4	208	7.7
8	102	15.1
16	59	26.8
36	42	37.8
72 HT	24	66.1

表 1. ナイーブ行列実装のパフォーマンスとスケーリング (36 コア、インテル® Xeon® プロセッサー E5-2697 v4、2 ソケット @ 2300MHz)

表 1 に示すように、並列ベンチマークは、スレッド数の増加に伴ってほぼ直線的にスケーリングしています。そして、30 コアに達すると横ばいになります。**表 1** のデータは、multiply1 ベンチマークのパフォーマンスとスケーリングについて誤った認識を招く恐れがあります。ベンチマークがマシンの計算能力をどれぐらい使用しているか理解することは非常に重要です。この例では、(ベンチマークで測定された) FLOPS がマシンの理論値を大きく下回っています (約 1/10)。並列処理のスケーラビリティーではなく、シリアル・パフォーマンスが制限されています。インテル® VTune™ Amplifier XE は、ループ内のコード実行が効率的でないことを示しています (**図 4**)。リタイアした命令数が低く、CPI レートが高いことから、実際の制限からどれぐらいかけ離れているのか予測できます。

S. Li.	Source	Cloc...	Inst... Ret...	CPI Rate	Front-... Bound	Bad Specu...	Back-... Bound	Retiring
66	void multiply1(int msize, int tidx, int numt, TYPE a[][NUM]							
67	{							
68	int i,j,k;							
69								
70	// Naive implementation							
71	for(i=tidx; i<msize; i=i+numt) {							
72	for(j=0; j<msize; j++) {	38,488...	35,...	1.077	1.1%	2.2%	76.6%	20.1%
73	for(k=0; k<msize; k++) {	71,300...	69,...	1.033	0.0%	0.0%	100.0%	14.7%
74	c[i][j] = c[i][j] + a[i][k] * b[k][j];	1,867,...	903...	2.067	0.7%	0.0%	87.1%	12.1%
75	}							
76	}							
77	}							
78	}							

4 ナイーブ並列行列乗算ベンチマークのパフォーマンス

次に、行列乗算アルゴリズムの最適化された実装 (**図 3** の multiply2) について見てみます。アルゴリズムが単純で、コンパイラーが理解できれば、インデックス・ストライドの効率が悪いことを認識し、ループ交換したバージョンを自動的に生成します (あるいは手動で生成できます)。

スレッド数	実行時間 (秒)	DP FLOPS (GFLOPS/ 秒)
4	208.8	7.8
8	103.3	15.1
16	58.8	26.4
36	38.4	40.5
72 HT	24.7	63.0

表 2. 最適化された行列乗算のパフォーマンスとスケーリング (36 コア、インテル® Xeon® プロセッサー E5-2697 v4、2 ソケット @ 2300MHz)

表 2 から、絶対数はやや向上しましたが、まだ理想からはかけ離れていることが分かります。

パフォーマンスとスケーラビリティーを妨げている要因について考えてみましょう。General Exploration (全般解析) プロファイルの結果 (**図 5**) は、CPU マイクロアーキテクチャー向けの別のトップダウン解析アプローチです。[3] いくつかの興味深いことが分かります。

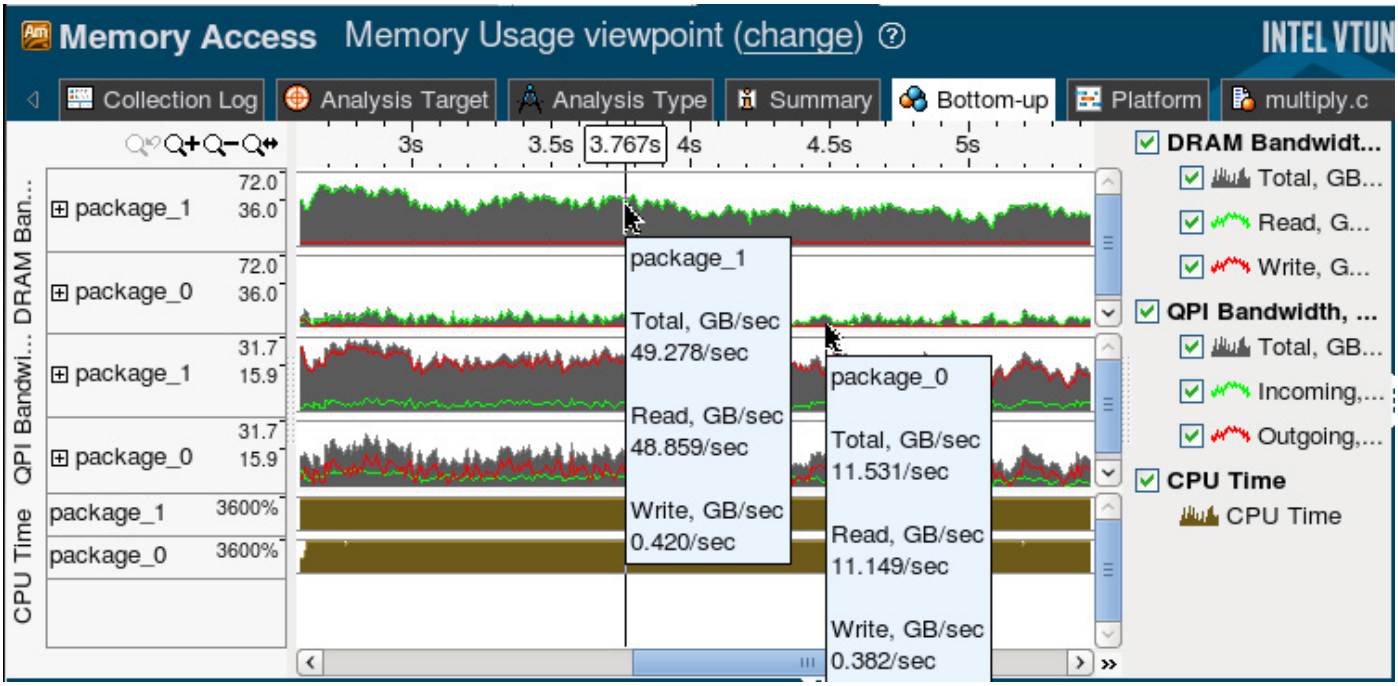
Function / Call Stack	Clockticks ▼	Instructions Retired	Front-End Bound	Bad Specul...	Back-End Bound	
					Memory Bound	Core Bound
multiply2	3,536,756,00...	1,160,060,2...	5.6%	0.1%	71.8%	14.3%

Back-End Bound							
Memory Bound							
L1 Bound	L2 Bound	L3 Bound	DRAM Bound				Store Bound
			Memory Bandwidth	Memory Latency			
				Local DRAM	Remote DRAM	Remote Cache	
7.9%	0.0%	33.3%	82.2%	29.0%	41.9%	0.0%	0.1%

5 全般解析プロファイルの結果

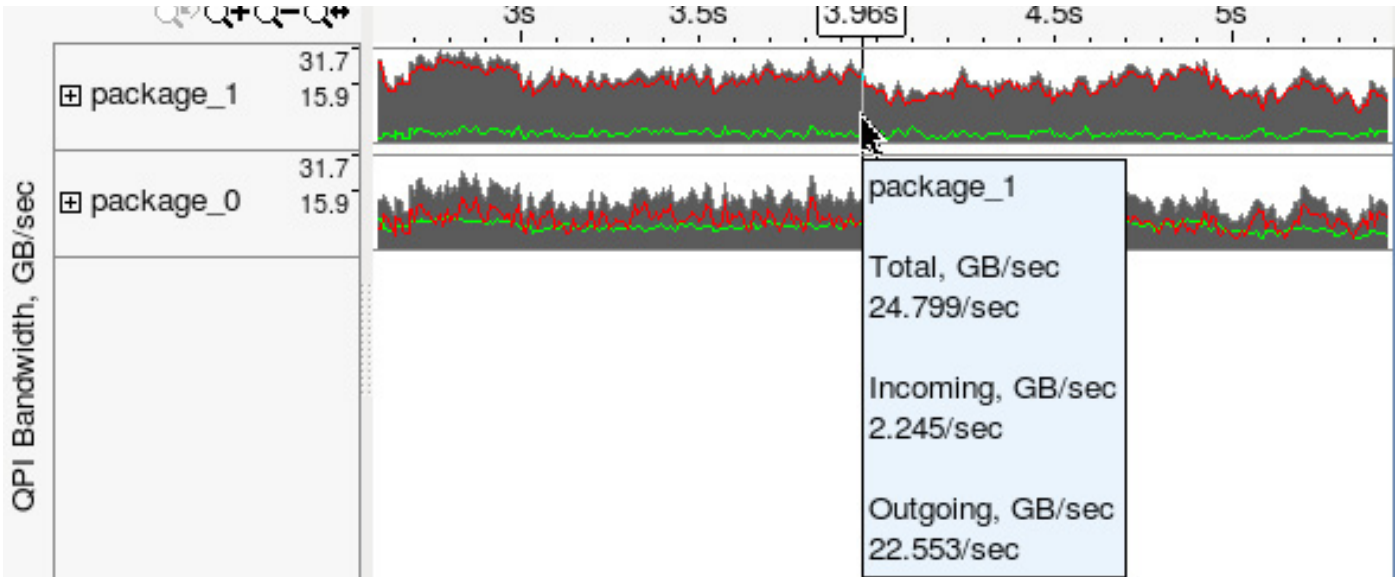
最初に、DRAM から CPU へのデータの読み取りメモリー・レイテンシーが減っています。アルゴリズムで連続するアドレスへのアクセスを実装しているため、これは想定内です。しかし、メモリー帯域幅メトリックが非常に高くなっているため、DRAM コントローラーと**インテル® QuickPath インターコネクト (インテル® QPI)** (英語) がボトルネックになっていないか、メイン・データ・パスの帯域幅を確認すべきです。2 つ目に、データアクセスが連続したパターンにもかかわらず、L3 レイテンシーも大きくなっていることが分かります。これは調査する必要があります。L3 レイテンシーが大きいということは、L2 ミスが頻繁に発生しているということです。連続したアクセスで DRAM レイテンシーが減っていないことから、ハードウェア L2 プリフェッチャーは適切に動作しているはずであり、これは奇妙です。3 つ目に、リモート DRAM レイテンシーが大きいことが分かります。これは、不均等メモリーアクセス (NUMA) による影響で、各ノードに対してデータの一部がリモート DRAM からフェッチされることを示しています。データ転送の全体像を掴むには、ソケット間の DRAM メモリー・コントローラーとインテル® QPI バスのデータ・トラフィックを測定する必要があります。これには、インテル® VTune™ Amplifier XE のメモリー・アクセス・プロファイルを使用します。

図 6 は、72 スレッドでプロファイルした結果です。1 つの DRAM コントローラー (package_1) のみでデータがロードされ、平均データ転送速度は約 50GB/ 秒、つまり最大帯域幅の約 2/3 であることが分かります。package_0 のメモリー・コントローラー上にはほとんどトラフィックがありません。



6 72 スレッドで multiply2 のメモリー・アクセス・プロファイルを収集

同じ期間のインテル® QPI 出力レーンのデータ・トラフィックの半分は package_1 が占めています。これで、package_1 DRAM から package_0 CPU コアへのデータの流れの説明がつきます (**図 7**)。このインテル® QPI のクロストラフィックが原因で、プリフェッチャーによってリモート DRAM からフェッチされるデータと CPU コアによってリモート LLC からフェッチされるデータには、追加のレイテンシーがかかります。ベンチマークでは、データが適切に構造化されており、スレッド間で均等に分配されるため、NUMA による影響を簡単に排除できます。スレッド・アフィニティーを CPU コアに変更し、各スレッドで **a**、**b**、**c** 行列を初期化するだけです。ただし、各スレッドでデータを割り当てることで NUMA による影響がすべて排除されるとは限りません。



7 インテル® QPI のクロストラフィック

例えば、**図 8** はパフォーマンスが向上しない例です。インテル® VTune™ Amplifier XE でこの問題を検出できます。ベンチマークのソースコードに、列挙された CPU にスレッドをピニングする関数を追加します。**図 8** は、コードの一部です。

```
CreateThreadPool( … )
{
  pthread_t ht[NTHREADS];
  pthread_attr_t attr;
  cpu_set_t cpus;
  pthread_attr_init(&attr);

  for (tidx=0; tidx<NTHREADS; tidx++) {
       CPU_ZERO(&cpus);
       CPU_SET(tidx, &cpus);
       pthread_attr_setaffinity_np(&attr, sizeof(cpu_set_t), &cpus);
    pthread_create( &ht[tidx], &attr, (void*)start_routine, (void*) &par[tidx]);

  }
  for (tidx=0; tidx<NTHREADS; tidx++)
    pthread_join(ht[tidx], (void **)&status);
}
```

8 列挙された CPU にスレッドをピニング

データ初期化関数では、配列は乗算関数で乗算される順序でスレッド間に分配されるべきです。**図 9** は、NUMA 対応を容易にするため、コードを変更した後の関数です。初期化関数では、データ配列は行列のサイズをスレッド数で割った、サイズ **`msize/numt`** のチャンクに分割されます。**図 10** に示すように、乗算関数でも同様の処理を行います。驚くべきことに、ベンチマークの実行時間は、NUMA 対応でないバージョンとあまり変わりがありません。インテル® VTune™ Amplifier XE のメモリー・アクセス・プロファイルで解析してみましょう (**図 11**)。

```
InitMatrixArrays (int msize, int tidx, int numt,  … )
{
    int i,j,k,ibeg,ibound,istep;
    istep = msize / numt;
    ibeg = tidx * istep;
    ibound = ibeg + istep;

    for(i=ibeg; i<ibound; i++) {
      for (j=0; j<msize;j++) {
            a[i][j] = 1.0*i+2.0*j+3.0;
            b[i][j] = 2.0*i+1.0*j+3.0;
            c[i][j] = 0.0;
        }
    }
}
```

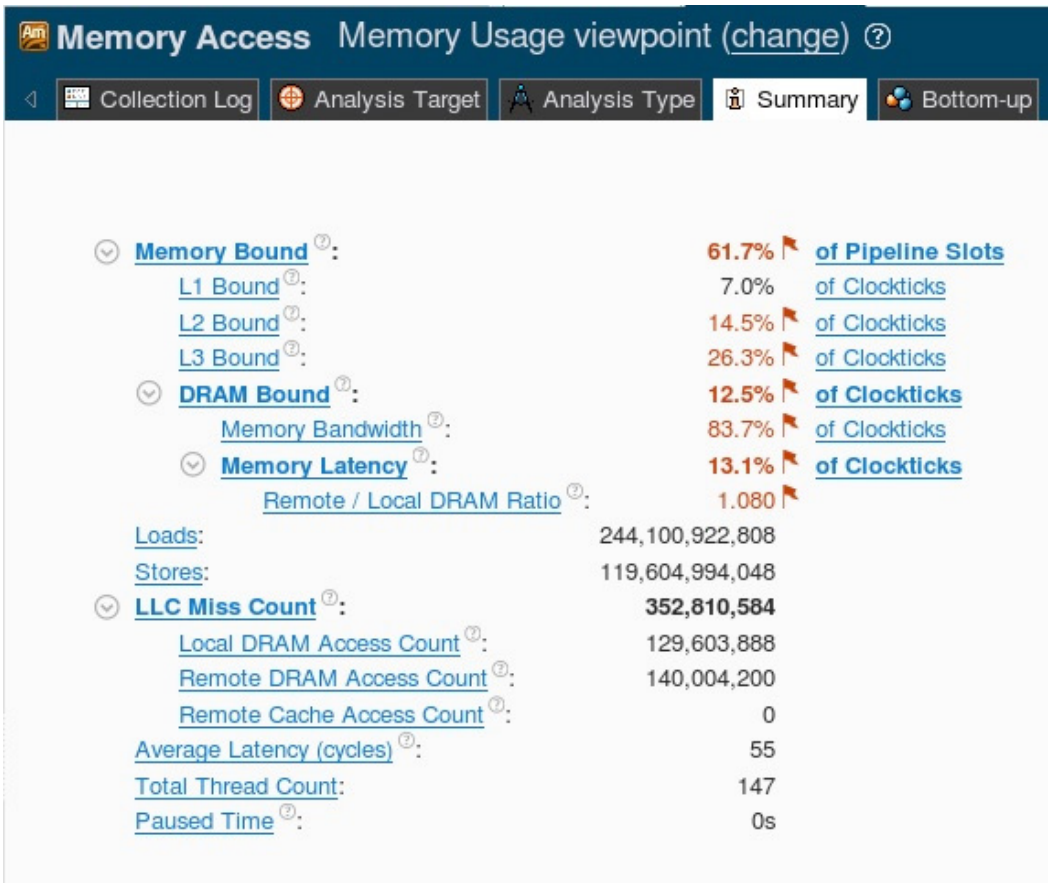
9 NUMA 対応の単純化

```
multiply2(int msize, int tidx, int numt,  ··· )
{
    int i,j,k,ibeg,ibound,istep;
    istep = msize / numt;
    ibeg = tidx * istep;
    ibound = ibeg + istep;

        for(i=ibeg; i<ibound; i++) {
                for(k=0; k<msize; k++) {
                        for(j=0; j<msize; j++) {
                                c[i][j] = c[i][j] + a[i][k] * b[k][j];
}}}}
```

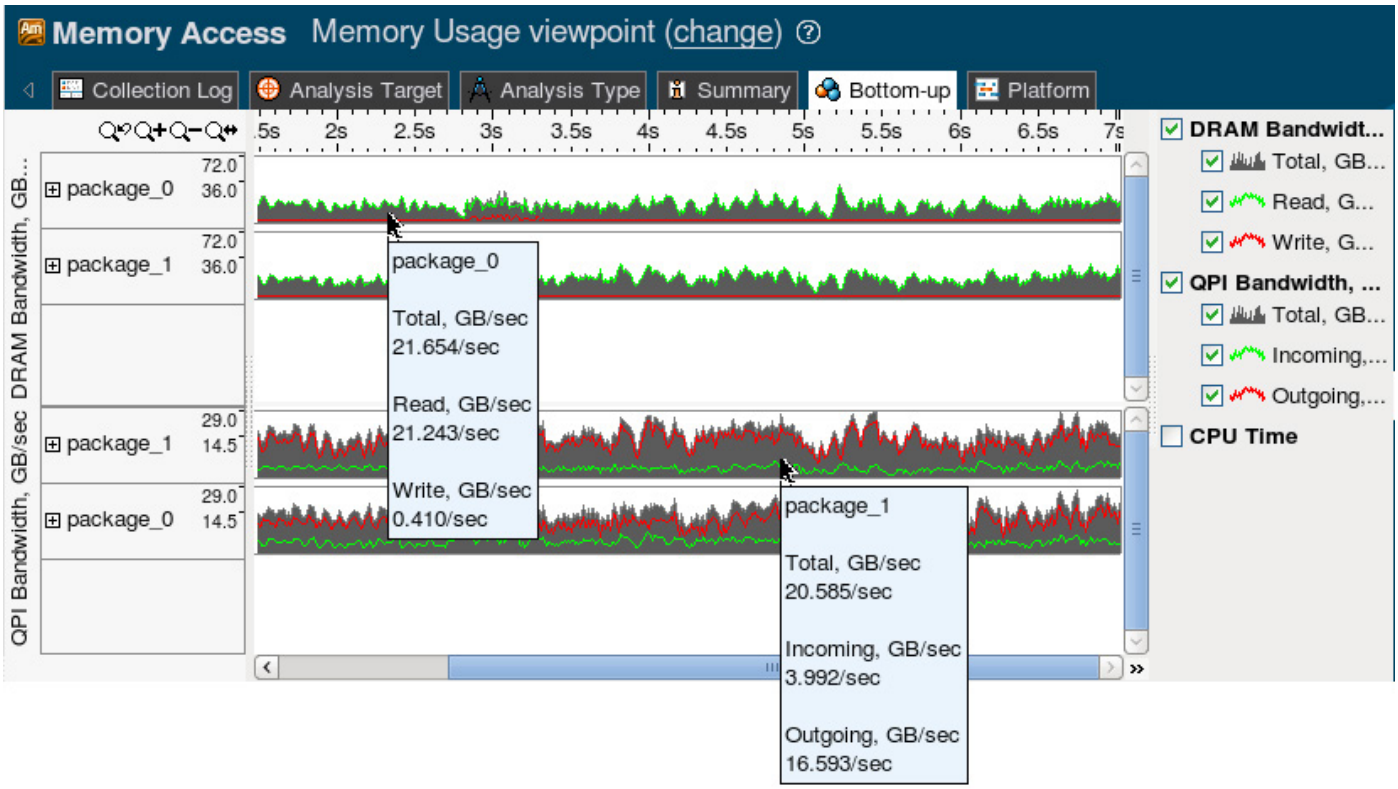
```
Threads #: 72 Pthreads
Matrix size: 9216
Using multiply kernel: multiply2
Freq = 2.30100 GHz
Execution time = 20.162 seconds
MFLOPS: 72826.877 mflops
```

10 乗算関数



11 メモリー・アクセス・プロファイル

[Summary] ページは、アプリケーションがまだメモリー依存であることを示しています (メモリーからのデータ・レイテンシーとデータ・トラフィックによるストールがあります)。ただし、レイテンシーのほとんどは LLC によるもので、DRAM によるものはわずかです。ローカルアクセスとリモートアクセスの比率が非常に高いことも分かります。これは、NUMA 対応のアプローチが適していないことを意味します。DRAM コントローラーとインテル® QPI のトラフィックのタイムライン (**図 12**) を確認すると、DRAM からのデータストリームがピーク帯域幅の 30% に達することはほとんどなく、一方でインテル® QPI の帯域幅は各方向で約 90% に達しており、飽和状態といえます (このシステムにおけるインテル® QPI の実際的な制限は 29.2GB/ 秒です)。



12 DRAM コントロールとインテル® QPI のトラフィックをタイムラインでチェック

リモートアクセス (DRAM または LLC) は、メモリーブロック読み取りレイテンシーと CPU ストールを増加させます。インテル® VTune™ Amplifier XE のメモリーアクセスでこれらのレイテンシーを測定することで、非効率的なリモートアクセスのデータ (行列) を特定できます。メモリー解析サマリー (**図 13**) から、レイテンシーの大きいメモリー・オブジェクトが分かります。

Top Memory Objects by Latency

This section lists memory objects that introduced the highest latency to the overall application execution.

Memory Object	Total Latency	Loads	Stores	LLC Miss Count
_mm_malloc (648 MB)	63.5%	112,553,776,512	25,600,384	339,210,176
_mm_malloc (648 MB)	36.2%	129,952,698,464	118,683,380,224	8,000,240
[Unknown]	0.2%	1,149,634,488	606,409,096	5,600,168
[Stack]	0.0%	120,803,624	201,603,024	0
_mm_malloc (648 MB)	0.0%	276,808,304	88,001,320	0
[Others]	0.0%	47,201,416	0	0

13 レイテンシー別上位メモリー・オブジェクト

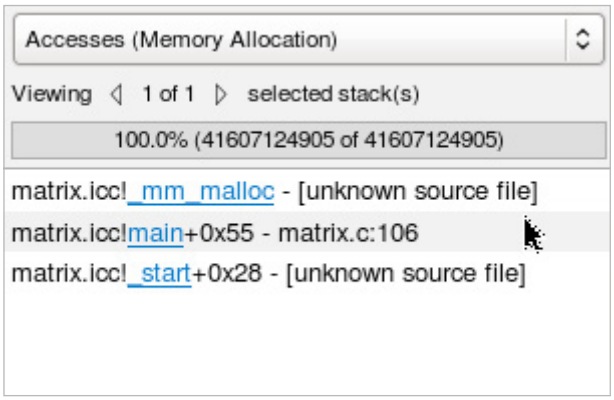
上位 3 つのメモリー・オブジェクト (割り当て関数によって表現される) のうち、1 つのレイテンシーが最も大きく、多くのロード操作を行っていることが明らかです (**図 14**)。平均レイテンシーから、LLC のリモート DRAM からデータを読み取っているのは、1 つのオブジェクトのみであることが分かります。これは、[Remote DRAM Access] 列の値によって裏付けることができます。

Grouping: Memory Object / Function / Allocation Stack

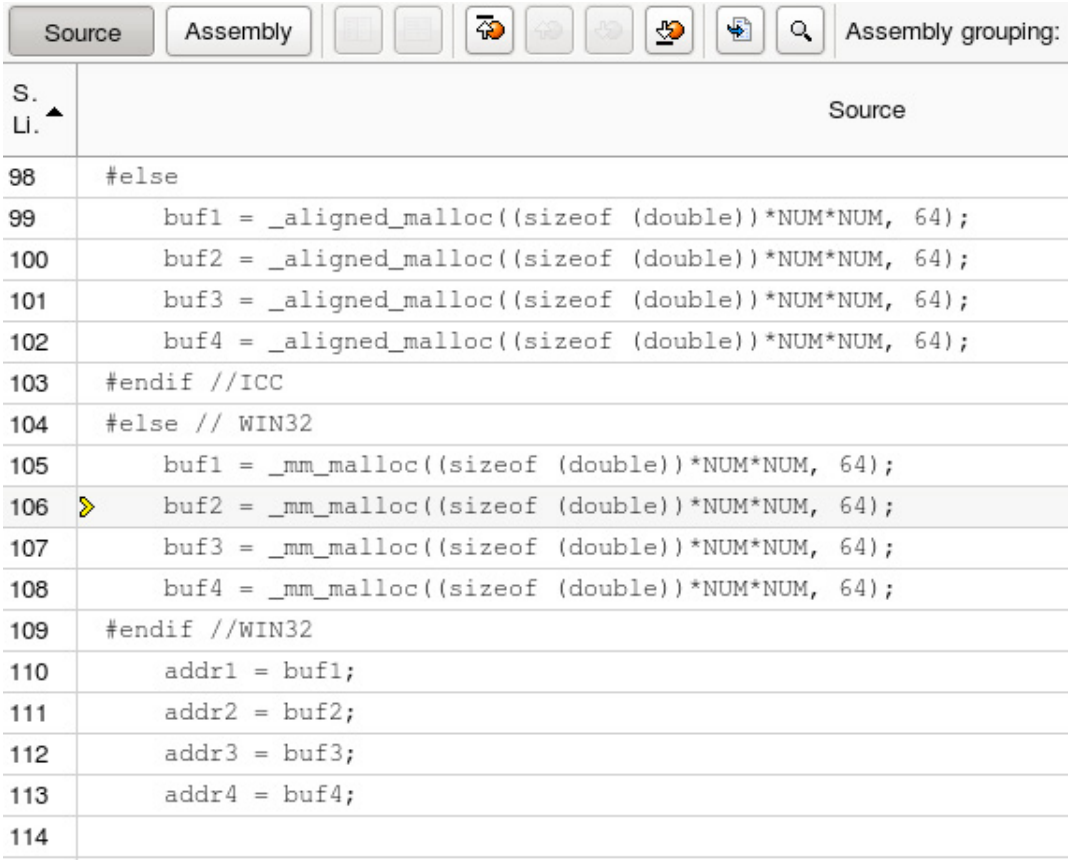
Memory Object / Function / Allocation Stack	Loads	Stores	LLC Miss Count	
			Local DRAM Acces...	Remote DRAM Access Count
[Unknown]	1,149,634,488	606,409,096	3,200,096	3,200,096
_mm_malloc (648 MB)	129,952,698,464	118,683,380,224	8,800,264	0
_mm_malloc (648 MB)	112,553,776,512	25,600,384	117,603,528	136,804,104
_mm_malloc (648 MB)	276,808,304	88,001,320	0	0

14 割り当て関数別メモリー・オブジェクト

この 3 つのオブジェクトが **a**、**b**、**c** 行列であることは明らかです。[Stores] の値が最も大きいのは行列 **c** です。レイテンシーが大きい行列データを特定するには、インテル® VTune™ Amplifier XE の [Stack] ペイン (**図 15**) でメモリー・オブジェクトのスタックを確認する必要があります。ユーザーコードのスタックを確認して、インテル® VTune™ Amplifier XE の [Source] ビューでデータ割り当てのソース行にドリルダウンできます (**図 16**)。この例では、レイテンシーとロード操作の増加の原因は行列 **b** のデータにあります。ピニングしたスレッド内でデータ配列の割り当てと初期化を行っているにもかかわらず、これらが増加している原因を理解する必要があります。

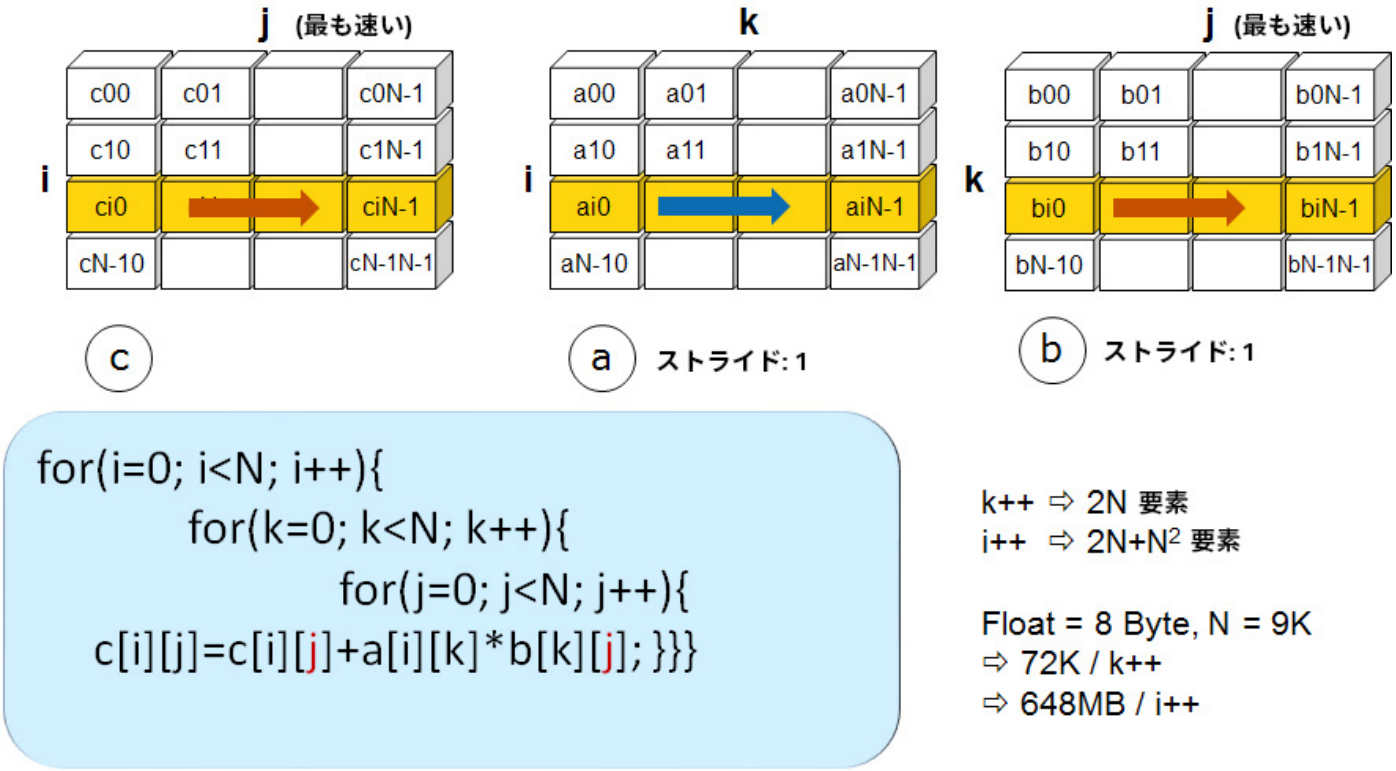


15 [Stack] ペインのメモリー・オブジェクト



16 インテル® VTune™ Amplifier XE の [Source] ビュー

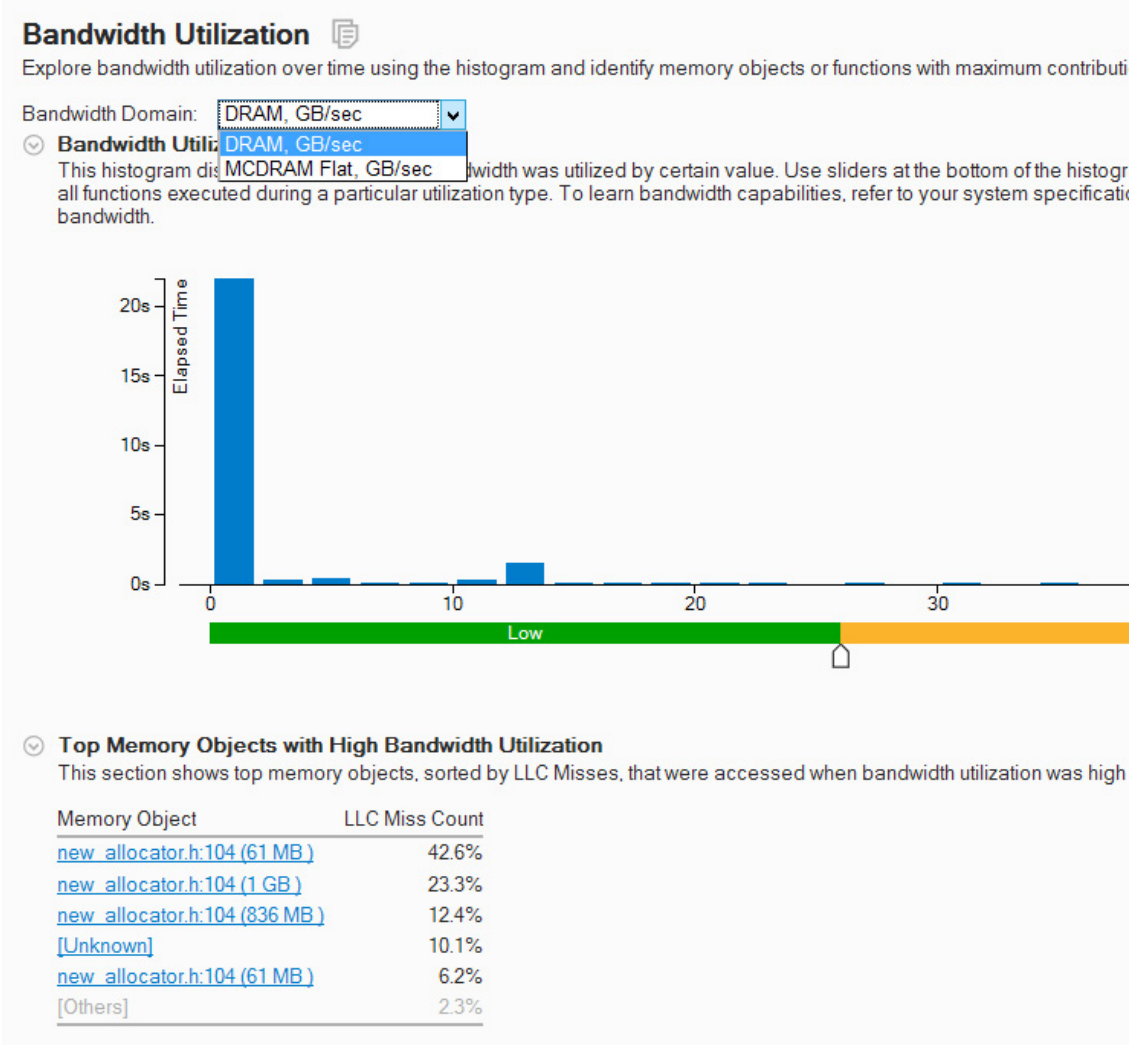
転置行列のアルゴリズムを調査したところ、データ・アクセス・パターンの根本的な問題が見つかりました (**図 17**)。行列 **a** の各行において、行列 **b** 全体をメモリーから読み取らなければなりません。



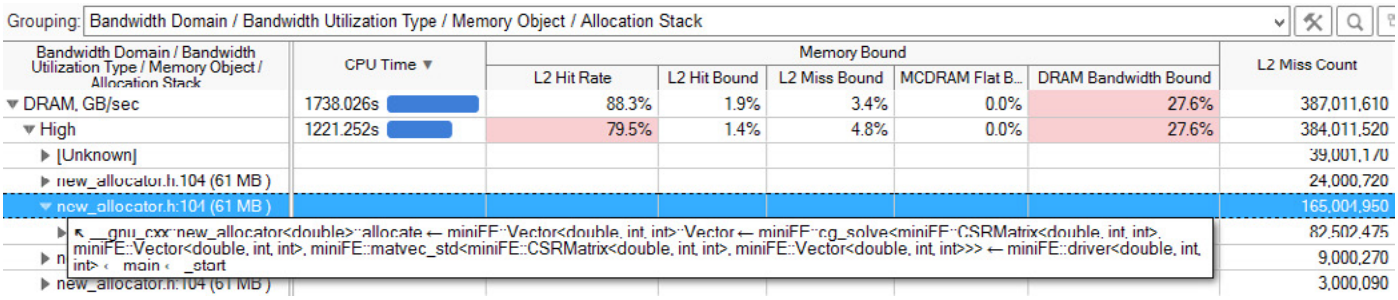
17 転置行列を使用するアルゴリズム

行列の列 / 行には、約 9,000 要素が含まれています。そのため、行列全体のメモリーブロックが CPU キャッシュに収まらず、キャッシュデータの退避と DRAM からのリロードが繰り返されます。行列 **c** と **a** の分散された行は、それらが割り当てられた CPU コアのスレッドによってアクセスされますが、行列 **b** は違います。アルゴリズムのこの実装では、行列 **b** のデータの半分はスレッドによってリモートソケットから読み取られます。さらに、行列 **a** の各行で行列 **b** 全体を読み取ることで、冗長なデータのロード操作が (N 回も余分に) 発生し、リモートデータにアクセスするためインテル® QPI トラフィックが増加します。

同様に、インテル® Xeon Phi™ プロセッサー・ベースのシステムでは、DRAM や MCDRAM のトラフィックを増加させているデータ・オブジェクトを特定できます。解析するメモリー・ドメイン・トラフィックを選択するだけで、オブジェクトへの参照と割り当てスタックの情報が得られます (**図 18**)。また、帯域幅ドメインと帯域幅使用率のタイプでグループ化すると、L2 ミスのカウントが最も大きいオブジェクトを特定できます (**図 19**)。



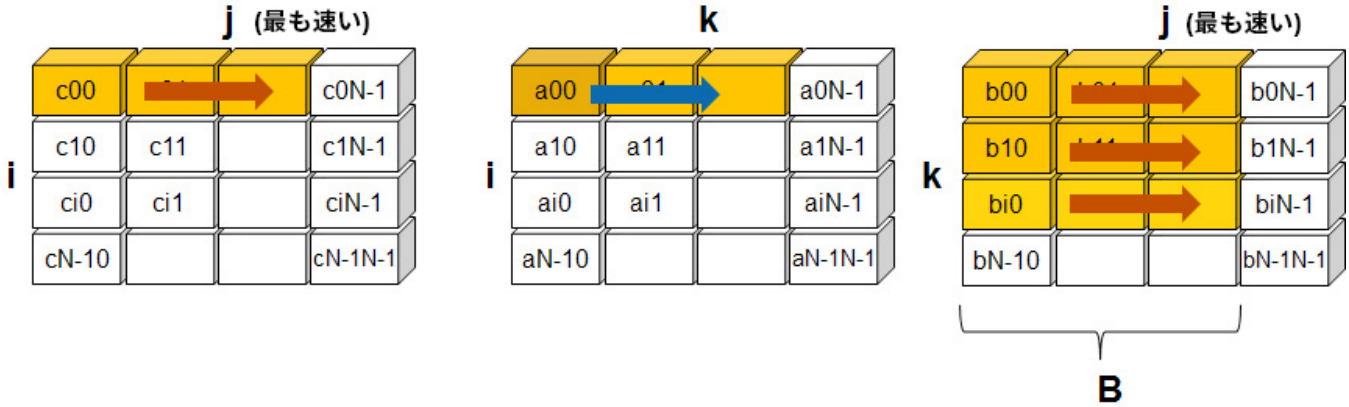
18 メモリー・ドメイン・トラフィックの解析



19 帯域幅ドメイン

データ・ブロッキング

乗算アルゴリズムをさらに変更して、CPU ストールを排除することで、データ・レイテンシーを減らすことができます。ローカルソケットで実行中のスレッドが 3 つの行列のすべてのデータにアクセスできるようにします。一般的に良く使用される手法の 1 つはデータ・ブロッキングです (**図 20**)。各配列の小さなブロックを処理することで、キャッシュにデータを保持し、CPU がそれらを再利用できるようにします (CPU キャッシュサイズのブロックを最適化することで、パフォーマンスがさらに向上する可能性があります)。また、スレッド間のブロックの配布が容易になり、過度のリモートアクセスとリロードを防ぐことができます。




```
for (i = ii; i < ii + B; i++) {
    for (k = kk; k < kk + B; k++) {
        for (j = jj; j < jj + B; j++) {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Float = 8 Byte
B = 64 要素
- L1D キャッシュに収まる

i++ ⇨ 2N + N^2 要素
N = 4K ⇨ 256K / i++
- L2 キャッシュに収まる

20 行列データ・ブロッキング

キャッシュブロック変更後の結果から (**図 21**)、NUMA による影響を修正しなくても、メモリー・レイテンシーが大幅に減り、高速に実行されるようになったことが分かります。

```
./matrix.icc
Threads #: 72 Pthreads
Matrix size: 9216
Using multiply kernel: multiply3
Freq = 2.3 GHz
Execution time = 12.08 seconds
MFLOPS: 128710.367 mflops
```

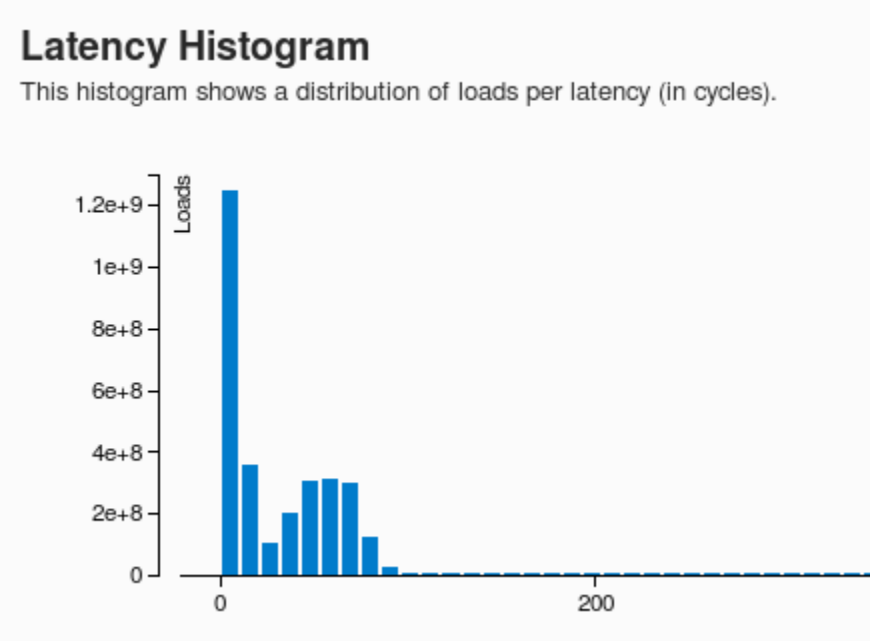
21 キャッシュ・ブロッキング変更後 (multiply3) のパフォーマンス

General Exploration (全般解析) プロファイル (**図 22**) は、リタイアしたパイプライン・スロットが最大 20% 増加し、残りの CPU ストールがメモリー依存とコア依存の実行で共有されることを示しています。

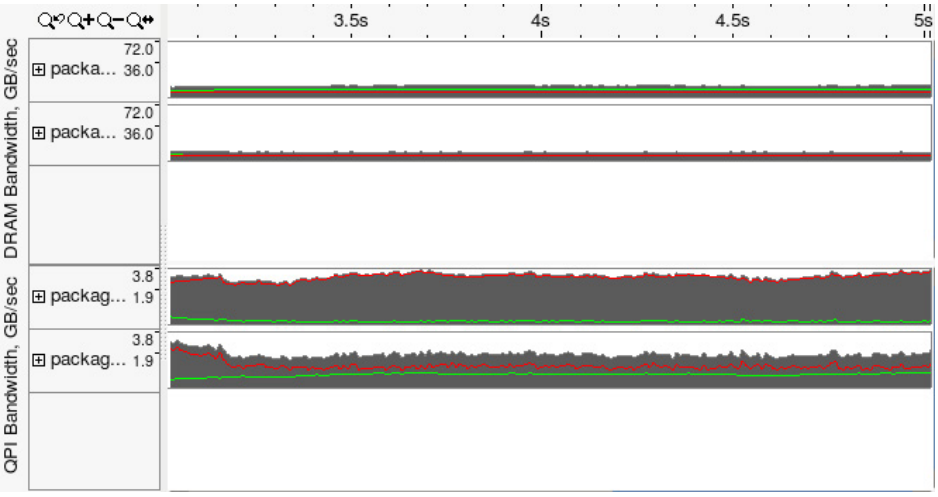
Clockticks:	2,268,184,100,000	
Instructions Retired:	1,851,470,100,000	
CPI Rate:	1.225	
MUX Reliability:	0.994	
Front-End Bound:	**9.4%**	**of Pipeline Slots**
Bad Speculation:	**0.0%**	**of Pipeline Slots**
Back-End Bound:	**70.9%**	**of Pipeline Slots**
Memory Bound:	**45.6%**	**of Pipeline Slots**
L1 Bound:	**0.7%**	**of Clockticks**
L2 Bound:	5.6%	of Clockticks
L3 Bound:	**31.9%**	**of Clockticks**
Contested Accesses:	0.0%	of Clockticks
Data Sharing:	0.0%	of Clockticks
L3 Latency:	100.0%	of Clockticks
SQ Full:	0.0%	of Clockticks
DRAM Bound:	**3.0%**	**of Clockticks**
Memory Bandwidth:	26.9%	of Clockticks
Memory Latency:	**69.5%**	**of Clockticks**
Local DRAM:	36.6%	of Clockticks
Remote DRAM:	6.5%	of Clockticks
Remote Cache:	0.0%	of Clockticks
Store Bound:	**1.2%**	**of Clockticks**
Core Bound:	**25.3%**	**of Pipeline Slots**
Retiring:	**19.6%**	**of Pipeline Slots**
Total Thread Count:	73	
Paused Time:	0s	

22 全般解析プロファイル (multiply3)

レイテンシー・ヒストグラム (**図 23**) は、レイテンシーのほとんどが L2 アクセスに関連するもので、残りは 50 ～ 100 サイクルの LLC ヒット・レイテンシーであることを示しています。帯域幅タイムライン・ダイアグラム (**図 24**) では、データのほとんどがローカル DRAM から読み取られ、インテル® QPI トラフィックがわずかに増えたことが分かります。これでもまだ、**インテル® マス・カーネル・ライブラリー (インテル® MKL)** の倍精度行列乗算の実装 (dgemm) のパフォーマンスには及びませんが、この行列サイズでは、同程度のパフォーマンスが得られるでしょう (**図 25**)。このほかに最適化としてできることは、アルゴリズムがブロックされるようにし、完全に NUMA 対応にすることです。最終的なパフォーマンスを**表 3** と**図 26** に示します。



23 レイテンシー・ヒストグラム (multiply3)



24 帯域幅タイムライン・ダイアグラム (multiply3)

```
$./matrix.mkl
Threads #: 72 requested OpenMP threads
Matrix size: 9216
Using multiply kernel: multiply5
Freq = 2.799980 GHz
Execution time = 2.897 seconds
MFLOPS: 540032.936 mflops
```

25 インテル® MKL ベースの multiply5 のパフォーマンス

スレッド数	実行時間 (秒)	DP FLOPS (GFLOPS/ 秒)
4	104.8	14
8	60.1	25
16	31.3	49
36	17.85	87
72 HT	12.08	128

表 3. 最後の最適化を適用した matrix3 のパフォーマンス (36 コア、インテル® Xeon® プロセッサー E5-2697 v4、2 ソケット @ 2300MHz)

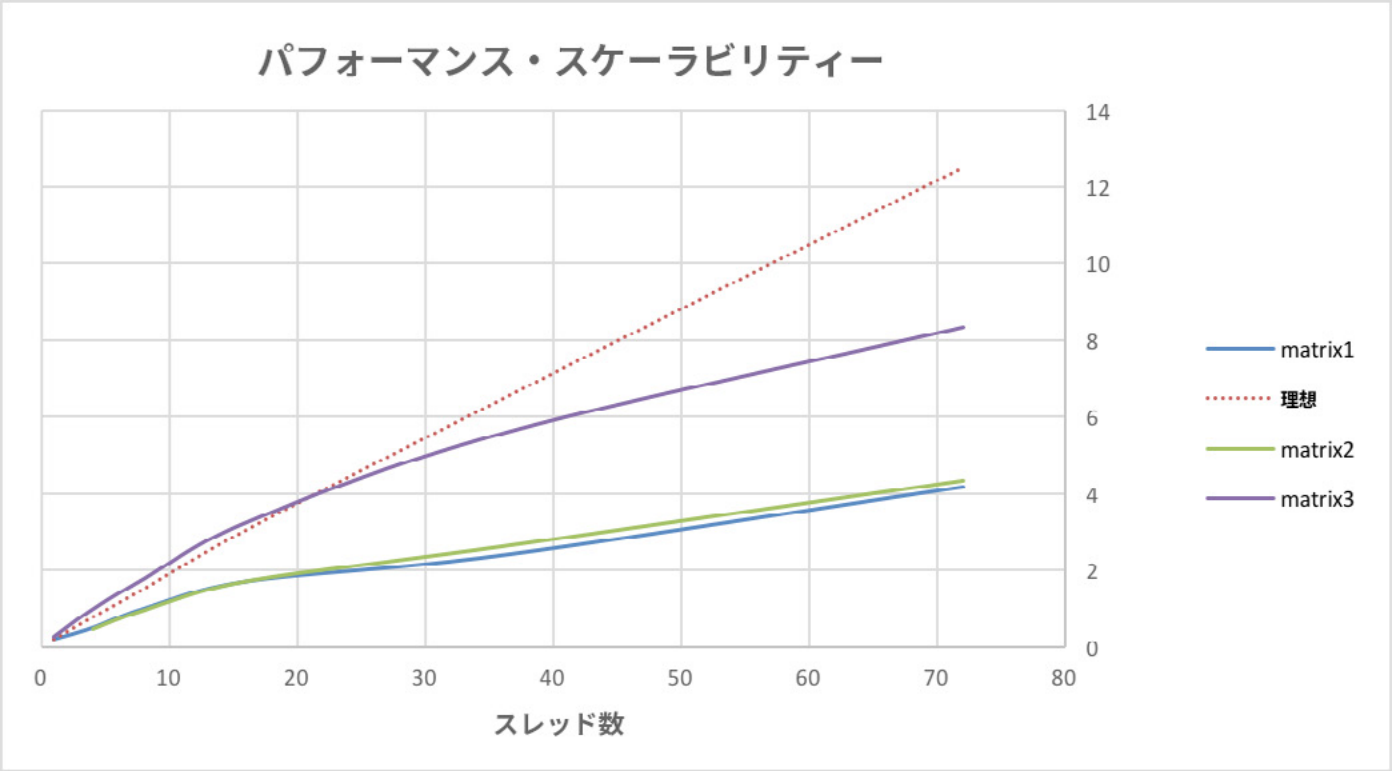
BLOG HIGHLIGHTS

AI 開発者としてはじめにすべきこと

NIVEN SINGH (INTEL) >

人工知能の可能性は、少なくとも 1950 年代から我々の想像力を掻き立ててきました。コンピューター科学者を新しくより複雑なテクノロジーの創造へと駆り立て、一般消費者の将来への関心を盛り上げました。身体的なリスクを負わずに海底を散歩したり、スマートウェイで無人自動車に乗ることができたらどうしますか？ AI とその可能性に対する理解はこの数十年の間に変わりましたが、ついに AI 時代が到来すると信じるに足る根拠が見つかりました。開発者として、我々は何をすべきでしょうか？ この記事では、AI の基本について説明し、役立つツールとリソースを紹介します。

この記事の続きはこちら (英語) でご覧になれます。 >



26 行列乗算ベンチマークの結果

スケーラビリティー・グラフに関する補足事項:

- matrix3 は、キャッシュ・ブロッキングにより理想的なスケーラビリティーを超えています。そのため、シングルスレッド・バージョンはナイーブ実装よりも高速に実行できます。
- スレッド数が物理コア数と等しくなるまで、matrix3 は理想的なスケーラビリティーとほぼ同じです。**ハイパースレッディング**を追加しても、スケーリングは向上しません。

まとめ

一部のメモリー・アクセス・パターンは、CPU マイクロアーキテクチャーの制限により、並列アプリケーションにおいてスケーラビリティーを妨げているように見えるかもしれません。これらの制限を回避するには、データの待機中に CPU ストールを引き起こすデータ配列を特定する必要があります。インテル® VTune™ Amplifier XE のメモリー・アクセス・プロファイルを利用することで、最も大きな遅延を引き起こすデータ・オブジェクト、遅延の長さ (CPU クロック数)、データが配置されているキャッシュ・サブシステムのレベル、データ・オブジェクトの割り当てと遅延アクセスのソースコードを識別することができます。この情報は、アルゴリズムを見直し、メモリー・アクセス・パターンを改善するのに役立つでしょう。

参考資料

1. Charlie Hewett. **Top Down Methodology for Software Performance Analysis (英語)**.

2. **Intel® 64 and IA-32 Architectures Optimization Reference Manual (英語)**.

3. Ahmad Yasin. *"A Top-Down Method for Performance Analysis and Counters Architecture."* IEEE Xplore: 26 June 2014. Electronic ISBN: 978-1-4799-3606-9.

4. **インテル® VTune™ Amplifier XE の General Exploration (一般解析) がどのように動作するかを理解する**

5. **マイクロアーキテクチャーのトップダウン解析法を使用してアプリケーションをチューニングする**

インテル® VTUNE™ AMPLIFIER XE を評価する >