インテル® VTune™ Profiler

エクセルソフト株式会社

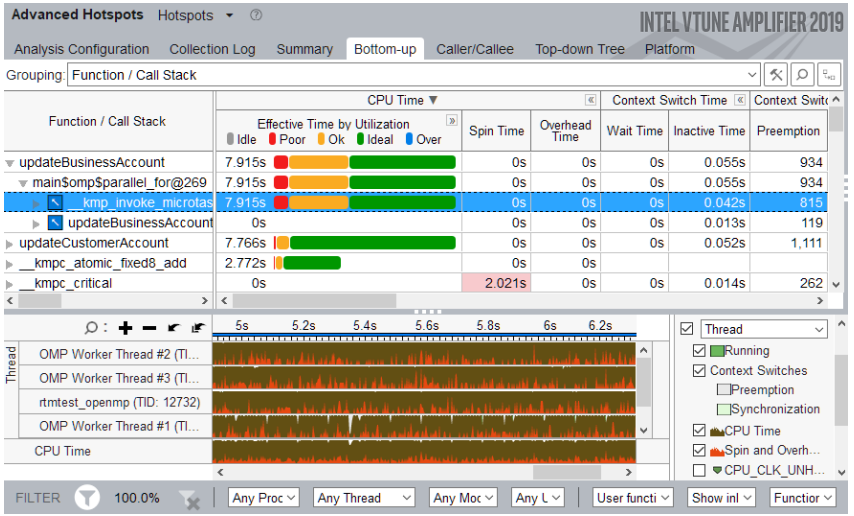
インテル® VTune™ Profiler

インテル® プロセッサーで動作するプログラムを最適化するためのプロファイラー

CPU の使用率や、メモリーの利用状況を数値化、グラフ化して表示

パフォーマンス問題のある個所を自動的に検出

修正の効果があると見込めるところのみを表示



プロファイラーとは

効率良く最適化を行うために、プログラムの
実行状況を確認するためのツール

どこに時間を費やしているか
関数ごとの時間や呼び出し経路

Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Source File
grid_intersect	3.844s	grid.cpp
sphere_intersect	2.780s	sphere.cpp
grid_bounds_intersect	0.291s	grid.cpp
shader	0.162s	shade.cpp

長い待機があるか
待機した時間と理由、スレッドの並行動作状況

Wait Time by Thread Concurrency (Idle / Poor / Ok / Ideal / Over)	Wait Count	Spin Time
4.036s	10	0s
4.014s	288	0.016s
0.714s	49	0s
0.649s	2	0.016s

効率の悪い実行をしているか
キャッシュミスや頻繁なメモリーアクセスなどイベントとソースコードの対応

Source	CPI Rate	Back-End Bo			
		Memory Bound			
		DRAM Bound	L3 Bound		
		LLC Miss	Conte...	Data ...	LLC Hit
for(int k=0; k<n; k++)	11.385	0.000	0.000	0.000	0.000
for(int j=0; j<p; j++)	3.618	0.000	0.000	0.000	0.000
c[i][j] = c[i][j] + a[i][k] * b[k][j];	4.436	0.342	0.014	0.094	0.261

バージョン 2019 の新機能
インテル® VTune™ Profiler

インテル® Optane™ DC パーシステント・メモリーと第2世代インテル® Xeon® スケーラブル・プロセッサーのサポートを追加

プラットフォーム・プロファイラーの追加

- ハードウェア構成の問題点を検出
- 適切にチューニングされていないアプリケーションを特定

MPI プリケーションの解析を改善

- MPI_PControl API によるデータ収集をサポート

さまざまなシナリオに対応できるプロファイル機能

シングルスレッド

シングルスレッド・
パフォーマンスを最適化

マルチスレッド

利用可能なすべてのコアを
効率的に利用

システム

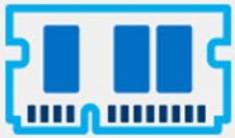
システムレベルのアプリケー
ション・パフォーマンスを表示

メディア/OpenCL* アプリケーション

ハイパフォーマンスな画像/
ビデオ処理パイプラインを実現

HPC/クラウド

HPC/クラウド・
コンピューティングに特化
した詳細な解析にアクセス



メモリーとストレージ管理

メモリー、ストレージ、データ
プレーンのボトルネックを診断

データの解析とフィルター

必要なデータを
マイニング

環境

各自の環境と
ワークフローに適合

プログラム言語、ターゲットのサポート

C/C++、Fortran、.NET*、Java*、Python*、Go*

- ネイティブコード、マネージドコードに対応

CPU と内蔵 GPUの両方を使用したヘテロジニアス構成

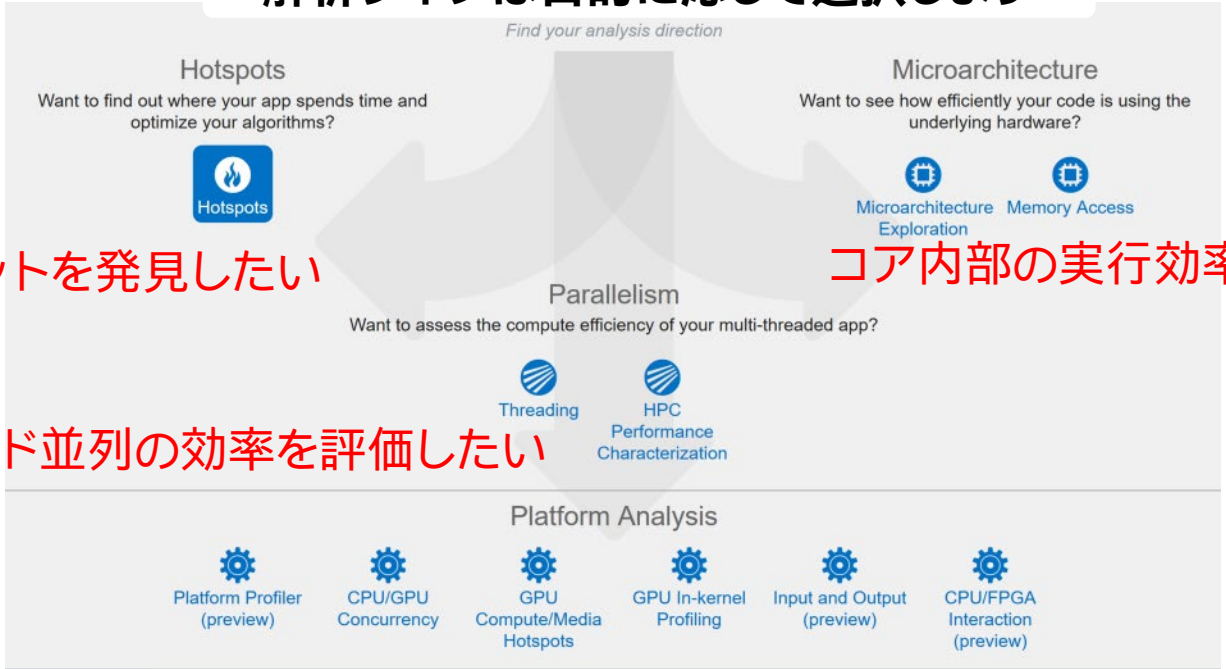
- OpenCL* と C/C++ など

コンテナー上で実行されるC/C++、Fortran および、Java* で記述されたアプリケーション (Linux* ホスト)

- LXC, Docker*, Mesos*, Singularity*

提供される解析タイプ

解析タイプは目的に応じて選択します



ホットスポットを発見したい

コア内部の実行効率を評価したい

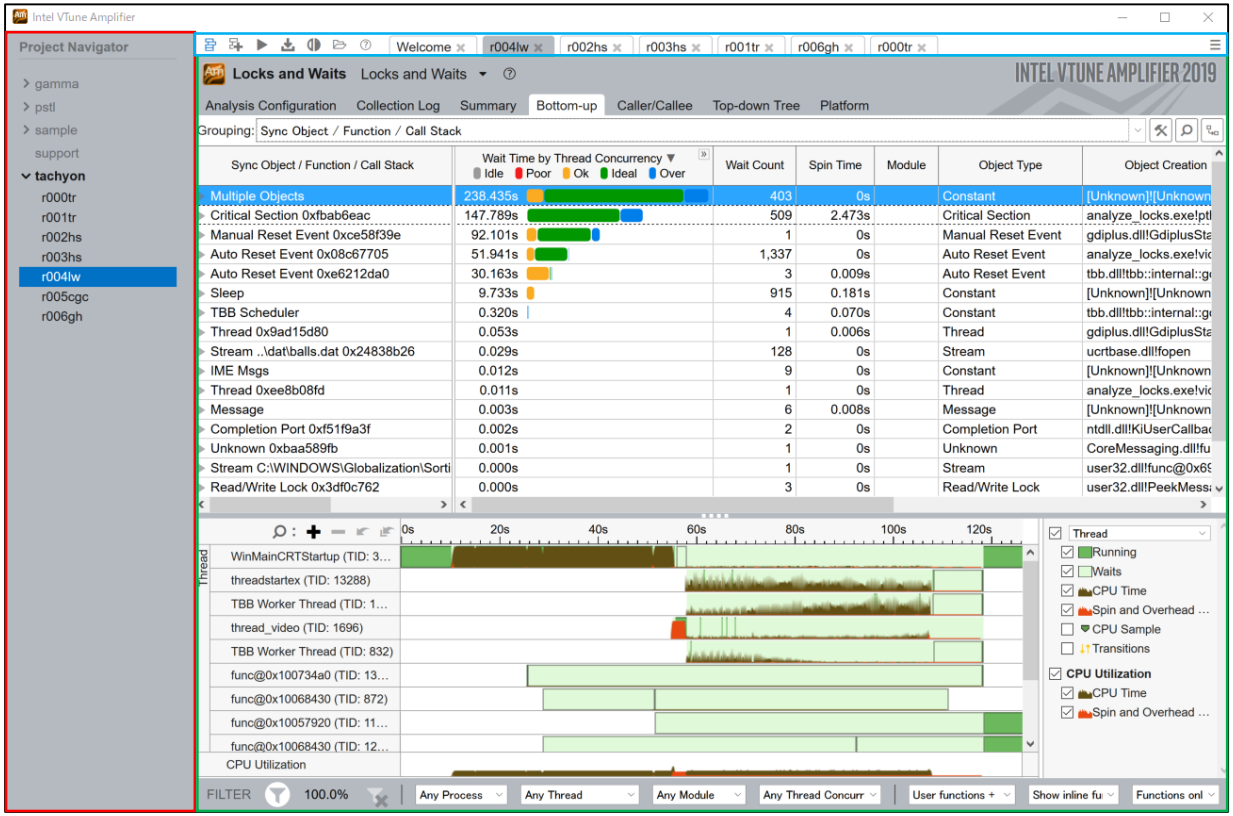
スレッド並列の効率を評価したい

出典: https://software.intel.com/en-us/VTune-Profiler-help-analysis-ty

インテル® VTune™ Profiler 画面構成

プロジェクト・ナビゲーター

※スタンドアロンのみ



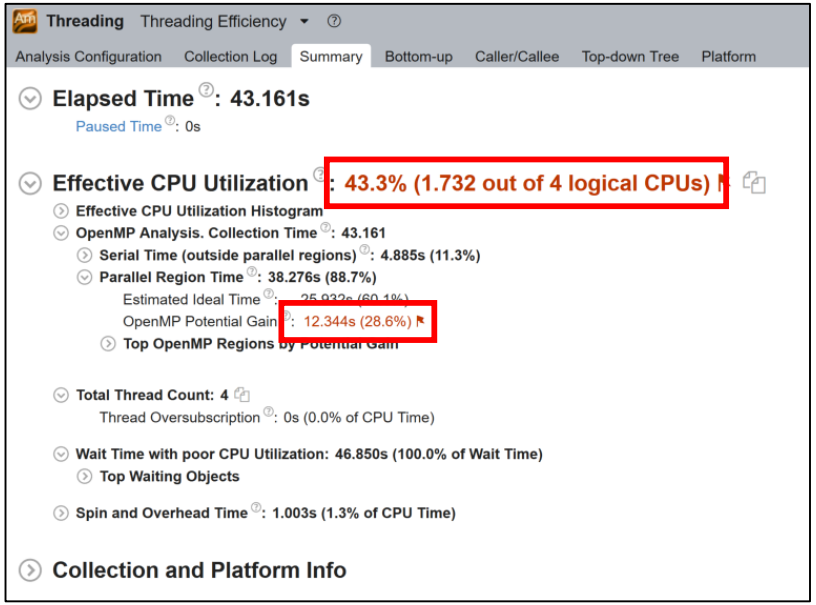
メニューバーとタブ

解析結果の表示

パフォーマンス向上の可能性を表示

性能問題となりえそうな項目を赤く表示します

- 注目すべき個所を指摘

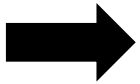


注目すべき処理のソースコードを確認する

対象のオブジェクトをダブルクリックすることで、
ソースコード･レベルに情報を掘り下げます

Grouping: Function / Call Stack

Function / Call Stack	Serial CPU Time	Effective Time by Utilization: Idle / Poor
multiply3ompparallel_for@	0s	38.334s
omp_get_max_threads	2.214s	0s
allrem	0s	0.360s
__kmp_fork_barrier	0s	0s
alldiv	0s	0.194s
__kmp_join_call	0s	0s
init_arr	0.090s	0.090s
free	0.054s	0.054s
exit	0.014s	0.014s
[Unknown]	0s	

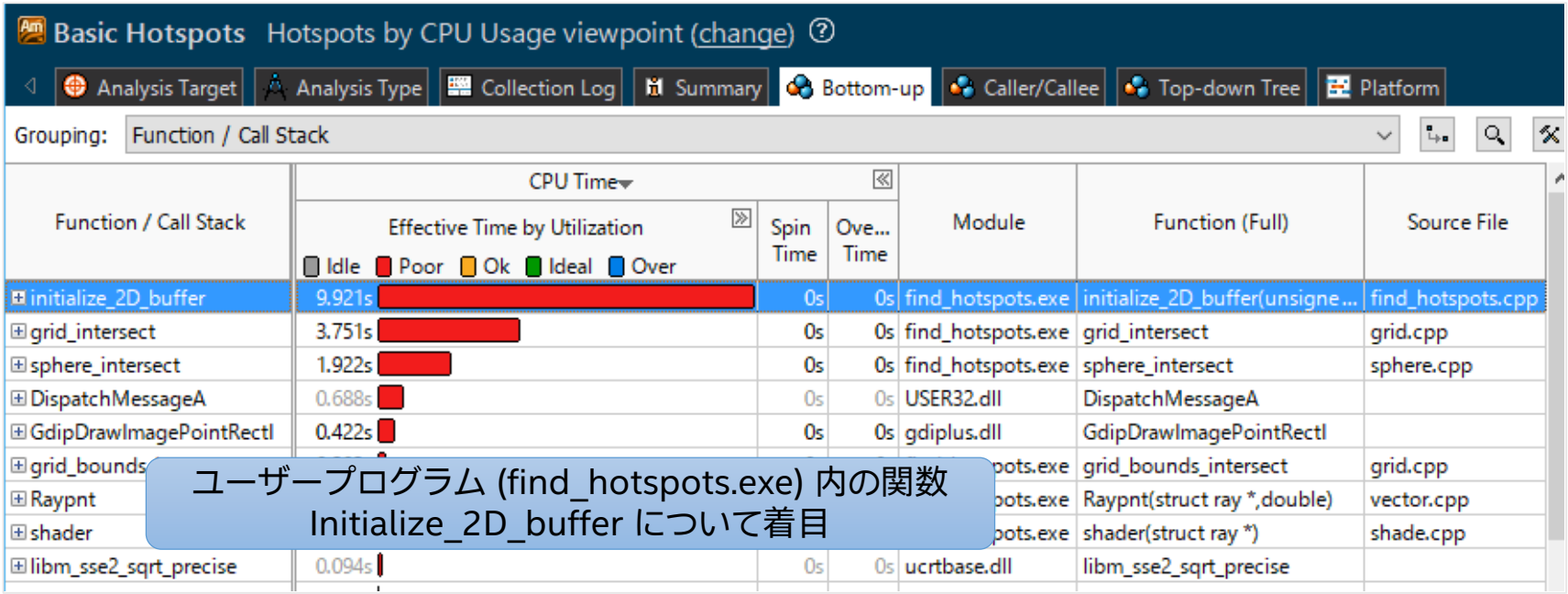


Source | Assembly

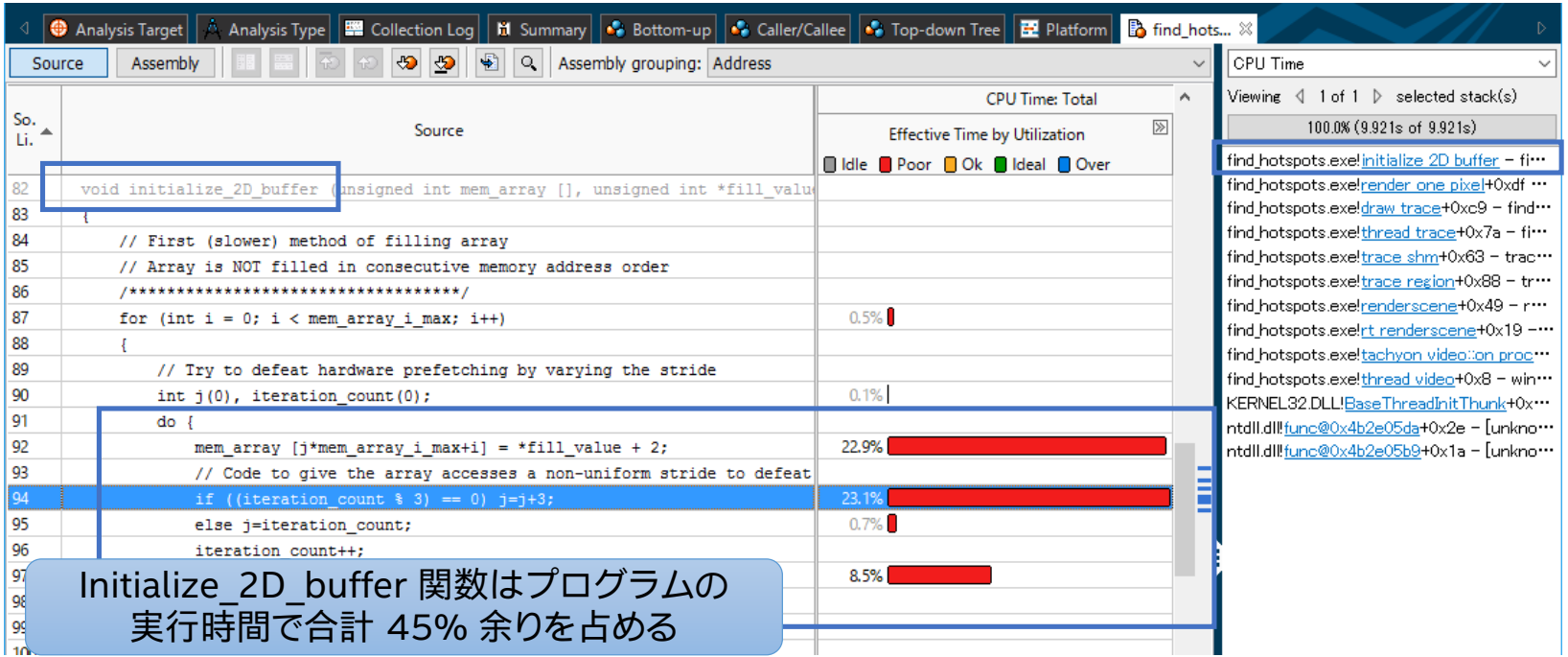
Sou ▲	Source	CPU Time: Total — Effective Time by Utilization (Idle / Poor / Ok / Ideal / Over)	Spin Time
202	}		
203	void multiply3(int msize, int tidx, int numt, TYPE a[][N		
204	{		
205	int i,j,k;		
206			
207	#pragma omp parallel for collapse (2)	91.9%	0.0%
208	for(i=0; i<msize; i++) {	0.6%	0.0%
209	for(k=0; k<msize; k++) {	0.1%	0.0%
210	#pragma ivdep		
211	for(j=0; j<msize; j++) {	2.6%	0.0%
212	c[i][j] = c[i][j] + a[i][k] * b[k][j];	88.6%	0.0%
213	}		
214	}		
215	}		

Hotspots 解析結果例

Bottom-up (関数/項目ごとの表示)



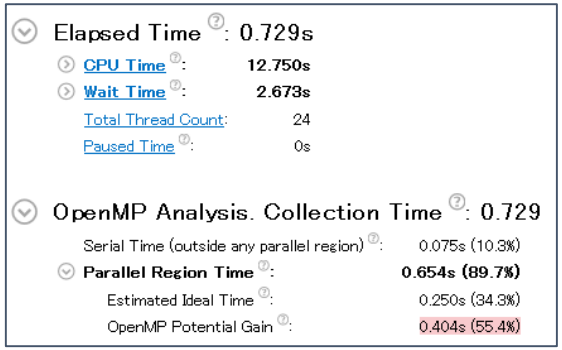
Hotspots 解析結果例



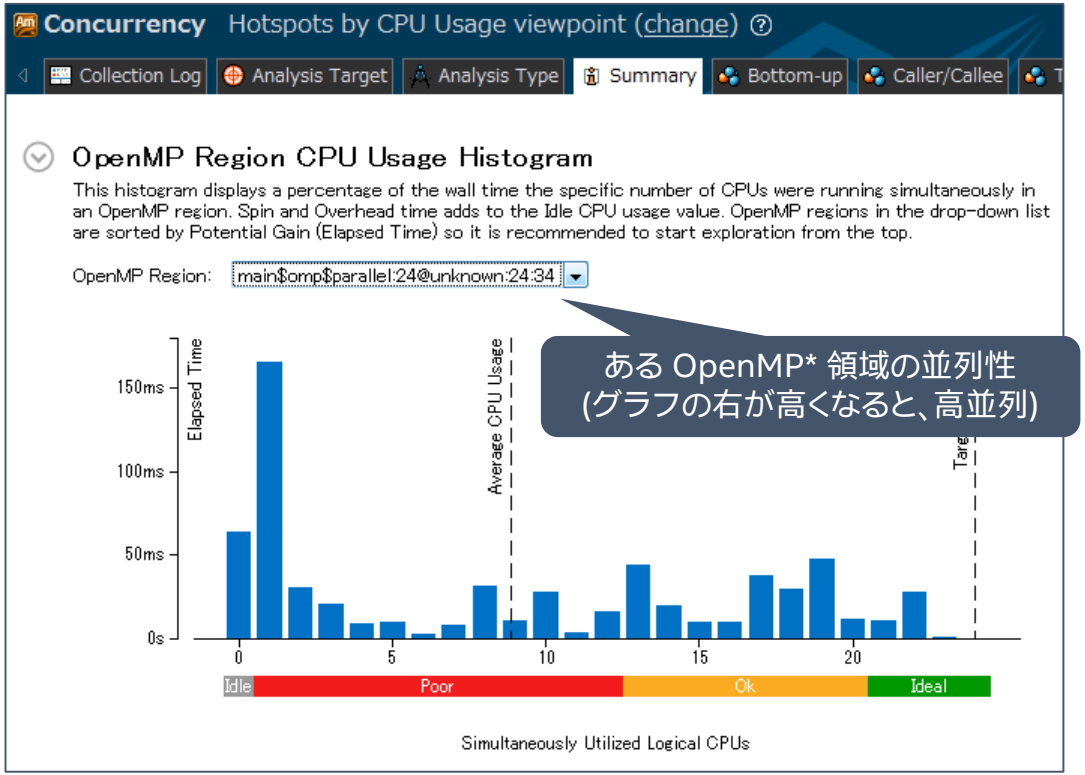
Threading 解析

各スレッドの動作状況を中心に解析

特に OpenMP* の利用について
有益な情報を表示する



シリアル/OpenMP* 領域での実行時間
(Elapsed Time) の割合と、OpenMP* 領域で
効率化できる割合 (Potential Gain)



ある OpenMP* 領域の並列性
(グラフの右が高くなると、高並列)

OpenMP* アプリケーションの パフォーマンス向上のヒント

OpenMP* 並列領域内の潜在的なパフォーマンス向上の可能性を評価します

- ロード･インバランス、スレッド生成、スケジューリング、ロックによるオーバーヘッド

Grouping を [OpenMP* Region > Function > Call Stack] などに設定

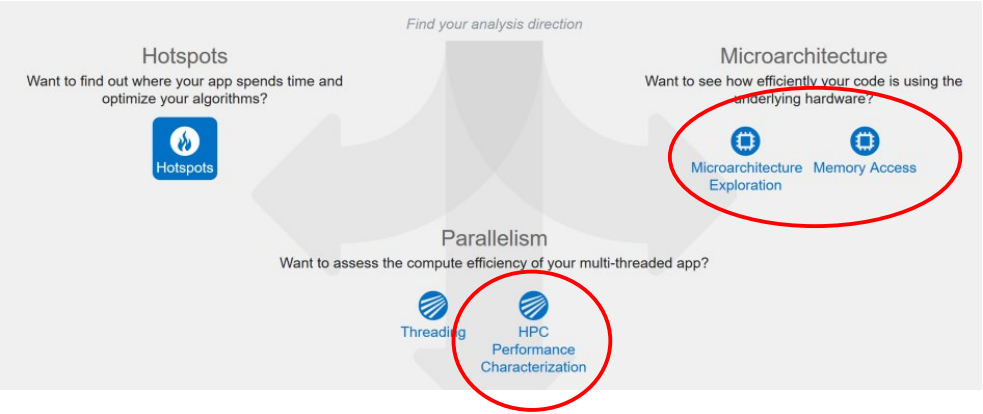
Grouping: OpenMP Region / Function / Call Stack

OpenMP Region / Function / Call Stack	OpenMP Potential Gain ▼						Elapsed Time	Number of OpenMP threads	Instance Count	Idl
	Imbalance	Lock Contention	Creation	Scheduling	Reduction	Atomics				
primesompparallel:64@/home/xlsoftkk/workspaces/takeda/ips/omp/prime.cpp:62:62	8.914s	0s	0s	0s	0s	0s	18.873s	64	1	637.1
matmulompparallel:64@/home/xlsoftkk/workspaces/takeda/ips/omp/matmul.cpp:25:34	0.026s	0s	0s	0s	0s	0s	0.055s	64	1	0.4
[Serial - outside parallel regions]							2.451s			1.4
matmulompparallel:64@/home/xlsoftkk/workspaces/takeda/ips/omp/matmul.cpp:33:39							0.368s	64	1	22.4

パフォーマンス問題

マイクロアーキテクチャーの最適化

- Microarchitecture Exploration
- memory-access
- HPC Performance Characterization



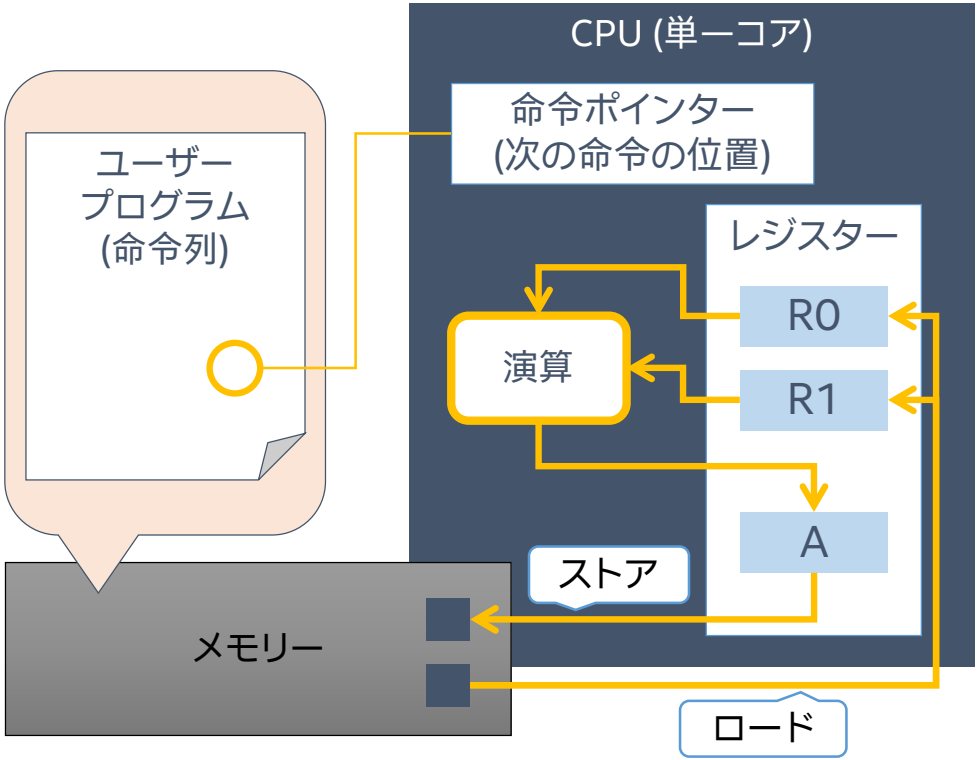
これらの解析タイプは、CPU、メモリーの利用状況に深く結びついた解析結果を表示します

プロセッサー上で実行されるプログラム

プロセッサーは命令を実行する

主に実行される命令

- ロード/ストア
  - データをメモリーからレジスターへ読み込み
  - データをレジスターからメモリーへ書き込み
- レジスターを使用して演算
  - 四則演算、ビット操作、比較
- ジャンプ
  - 次に読み込むプログラムの位置を変更
  - 関数呼び出し/復帰や条件分岐を実現



コア内部の実行の流れ

コア内部で実行される命令は、複数の
独立した工程 (ステージ) を通して処理します

フェッチ (Instruction Fetch)

- メモリーから命令を取り込む

デコード (Instruction Decode)

- 取り込まれた命令を実行ユニットが読解可能な
  マイクロオペレーション (uOP) に変換する

実行 (Instruction Execution)

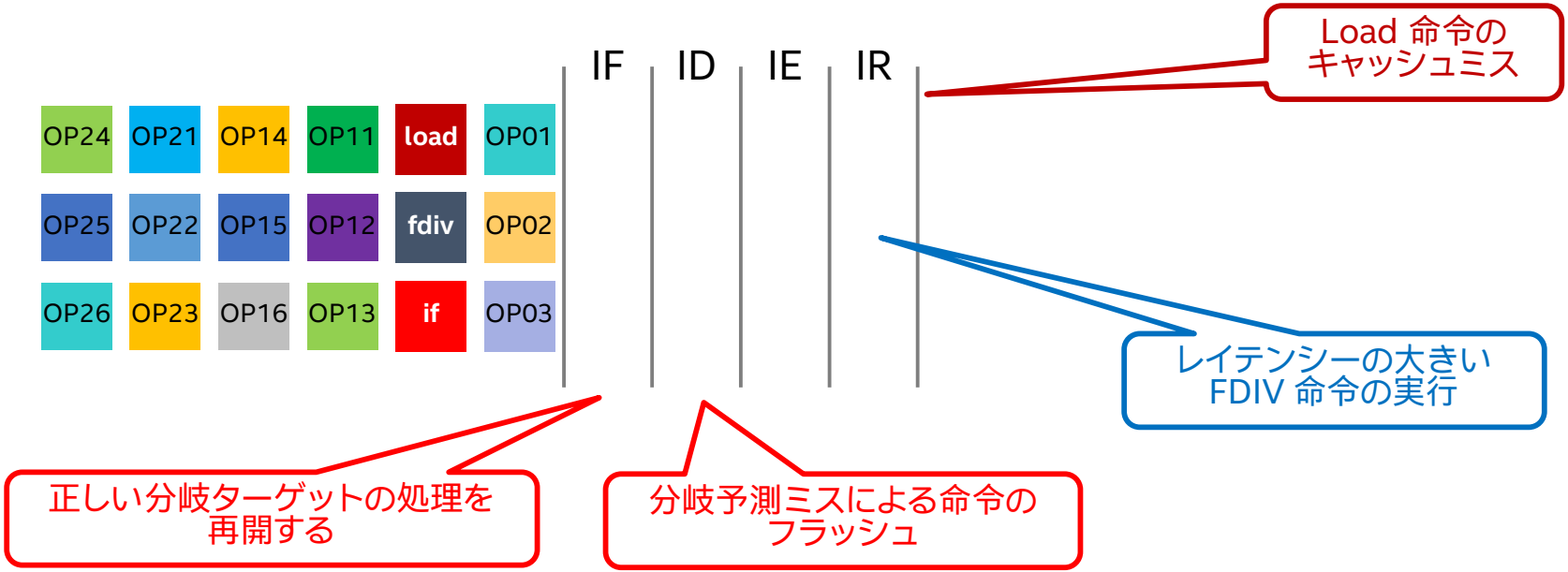
- デコードされたマイクロオペレーションを実行する

リタイア (Instruction Retirement)

- 実行結果をメモリーにアップデートして命令の実行を完了する

CPU サイクルは止まらないが 命令の実行は止まる

パイプラインのスムーズな処理の流れを止める現象



パイプラインの最適化

無駄になったサイクルは実行が最適化されていない

- 無駄になったサイクルを発見することが重要

命令あたりのサイクル数 (CPI) を確認して、
ストールしている個所に目安をつける

✓ **CPI 値が高くなる主な要因**

- 実行コストの大きな命令
  - 例:浮動小数点数の除算
- キャッシュミスを起こしたメモリーアクセスが多い
- 頻繁な分岐命令の実行
  - 分岐命令に 売く命令は、分岐方向が定まるまで未確定

メモリー階層

✓ **キャッシュメモリーの存在**

メモリーアクセスの待ちを低減するための高速メモリー

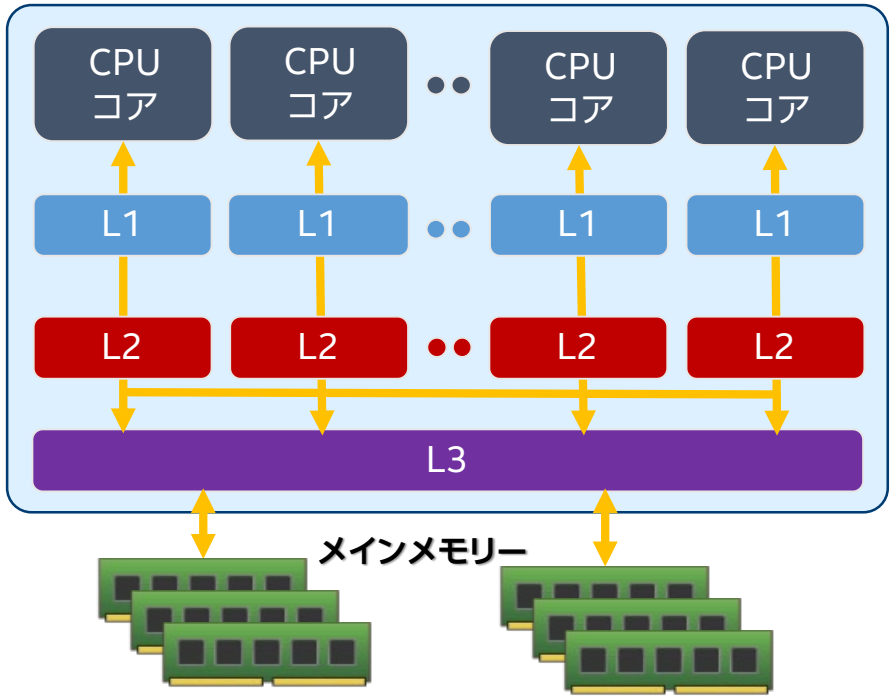
一度アクセスしたデータを保持する

同じデータが再利用されるときに有効

主要なインテル® プロセッサー は L3 (LLC)、L2、L1 レベルのキャッシュ領域を持っている

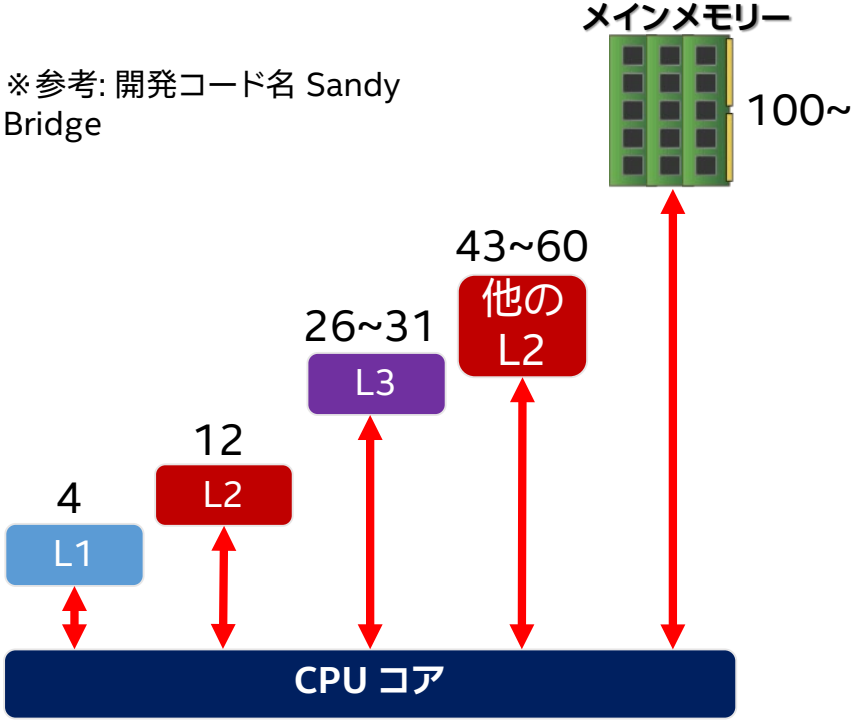
L2、L1 はコアごとに持ち、L3 はコア共有の領域として扱う

主要なインテル® プロセッサーのメモリー構造

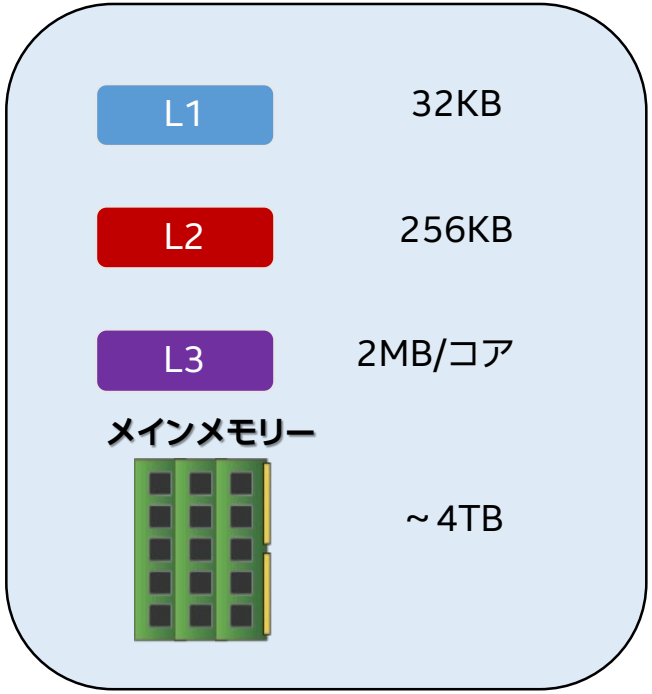


コアへデータ転送に要する時間と容量

※参考: 開発コード名 Sandy Bridge



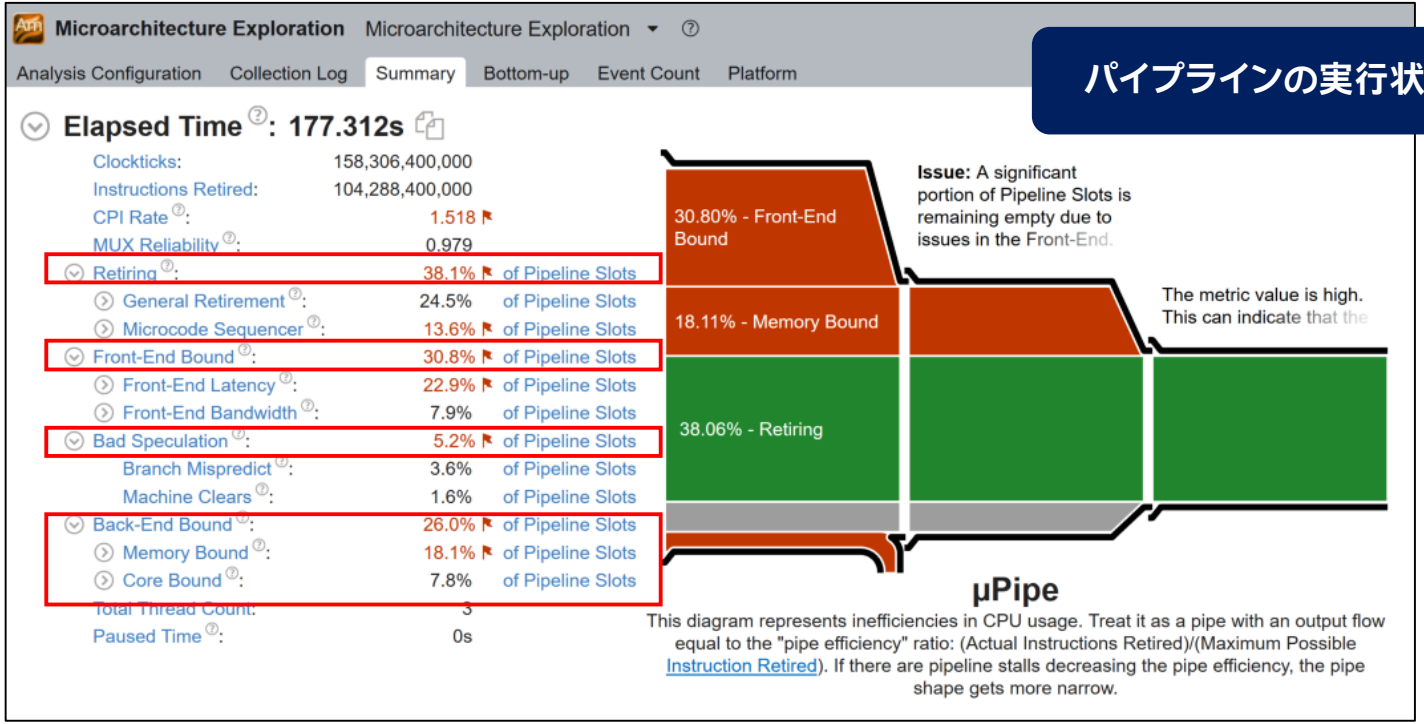
転送時間 (単位は CPU サイクル)



キャッシュ、メモリーの容量

コアの実行効率やメモリーアクセスの問題を調査する

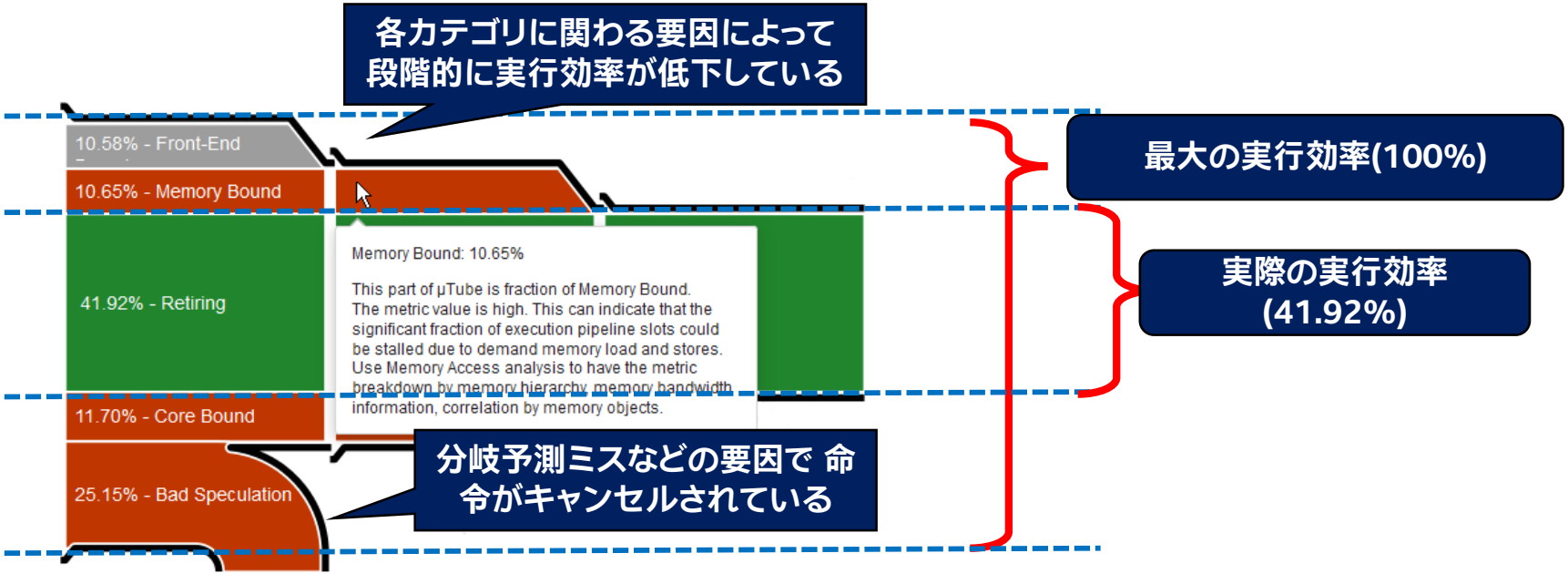
Microarchitecture Exploration 解析



パイプラインの実行状況を視覚化

パイプライン内部の実行を視覚化

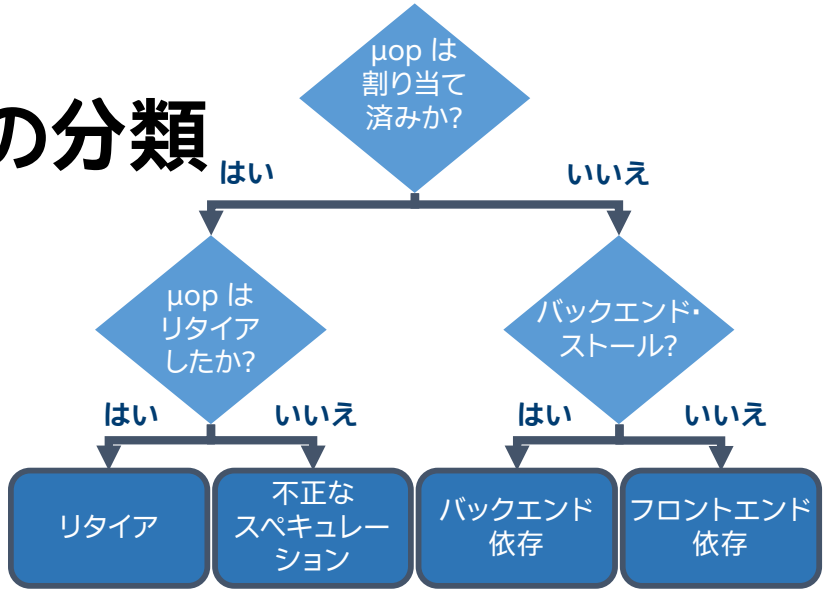
実行効率が悪い状態では、カテゴリされている要因に
制限されてパイプは狭くなる



パイプライン・スロットの分類

パイプライン・スロットは µop に対してどのような処理を行うかに基づいてサイクルごとに 4 つのカテゴリーに分類される

- リタイア
- 不正なスペキュレーション
- バックエンド依存
- フロントエンド依存

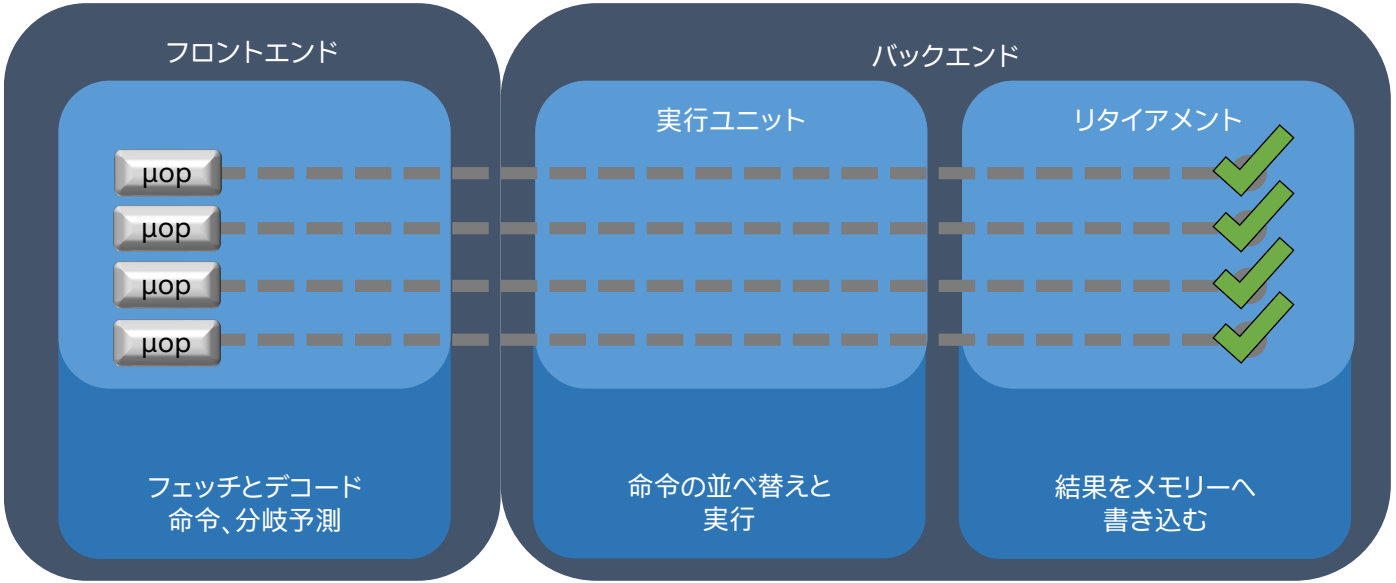


各カテゴリーには適切にチューニングされたアプリケーションにおいて想定される値の範囲がある

カテゴリー ＼ アプリケーション・タイプ	クライアント/デスクトップ	サーバー/データベース/分散	ハイパフォーマンス・コンピューティング
リタイア	20-50%	10-30%	30-70%
不正なスペキュレーション	5-10%	5-10%	1-5%
フロントエンド依存	5-10%	10-25%	5-10%
バックエンド依存	20-40%	20-60%	20-40%

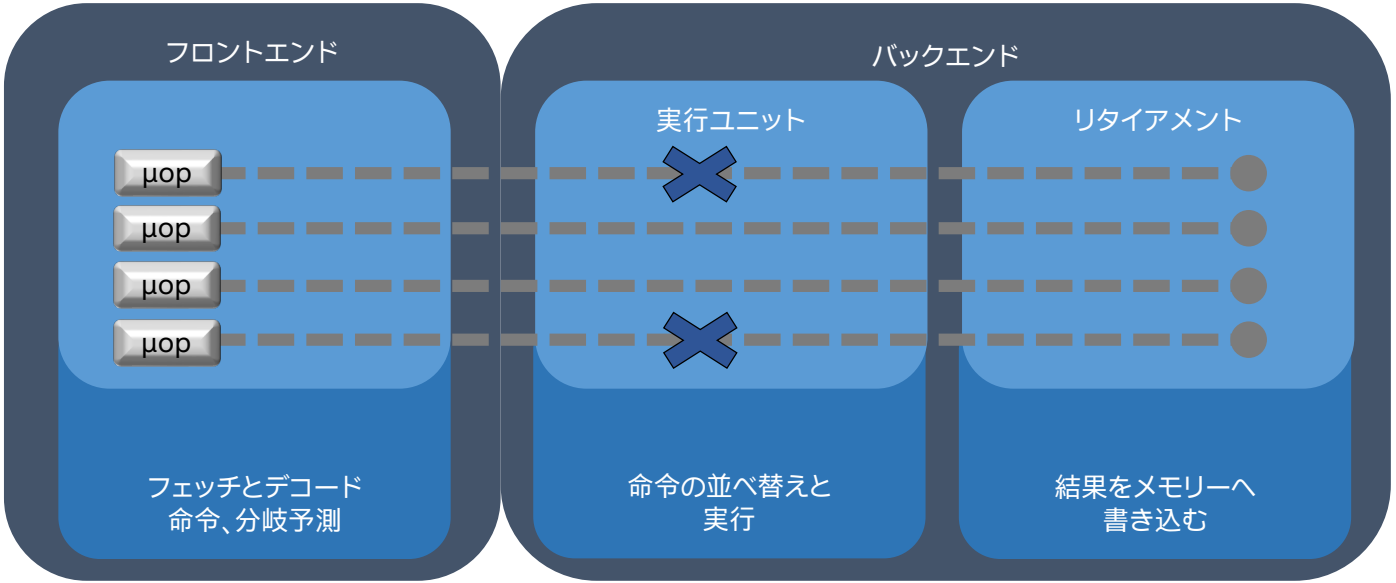
リタイア

できるだけ多くのスロットがこのカテゴリーになるようにする
(ただし、このカテゴリーであってもさらに最適化できることがある)



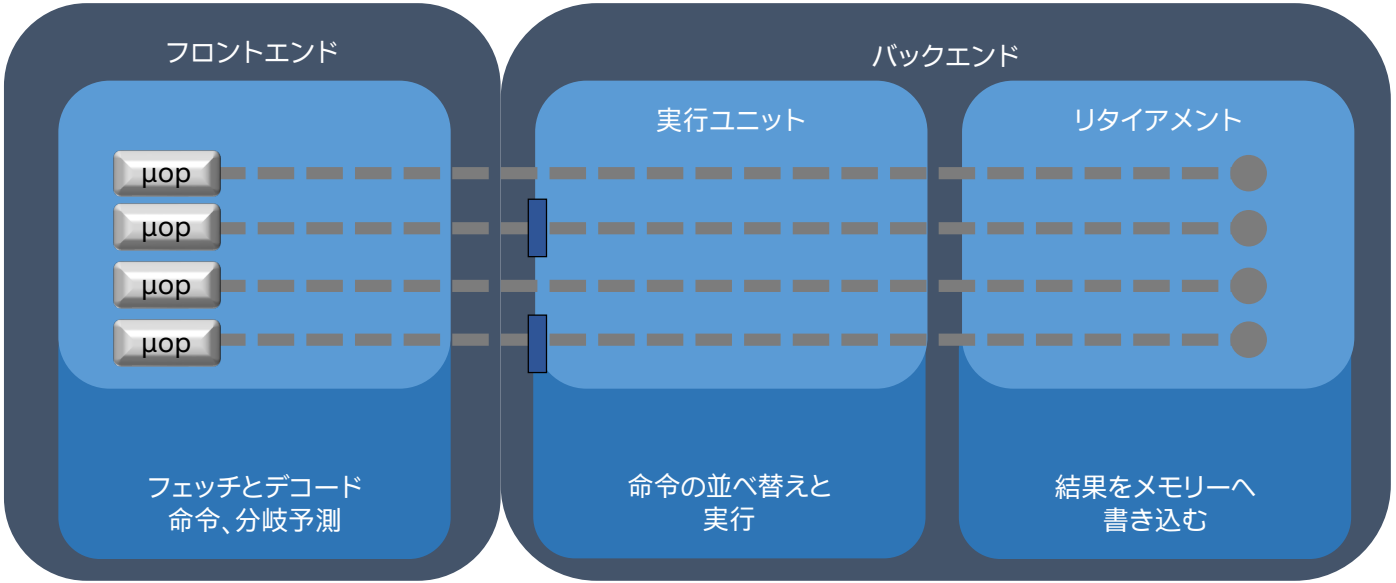
不正なスペキュレーション

分岐予測ミスによるキャンセルなどにより、命令がリタイアせずにバックエンドから削除 (キャンセル) された場合



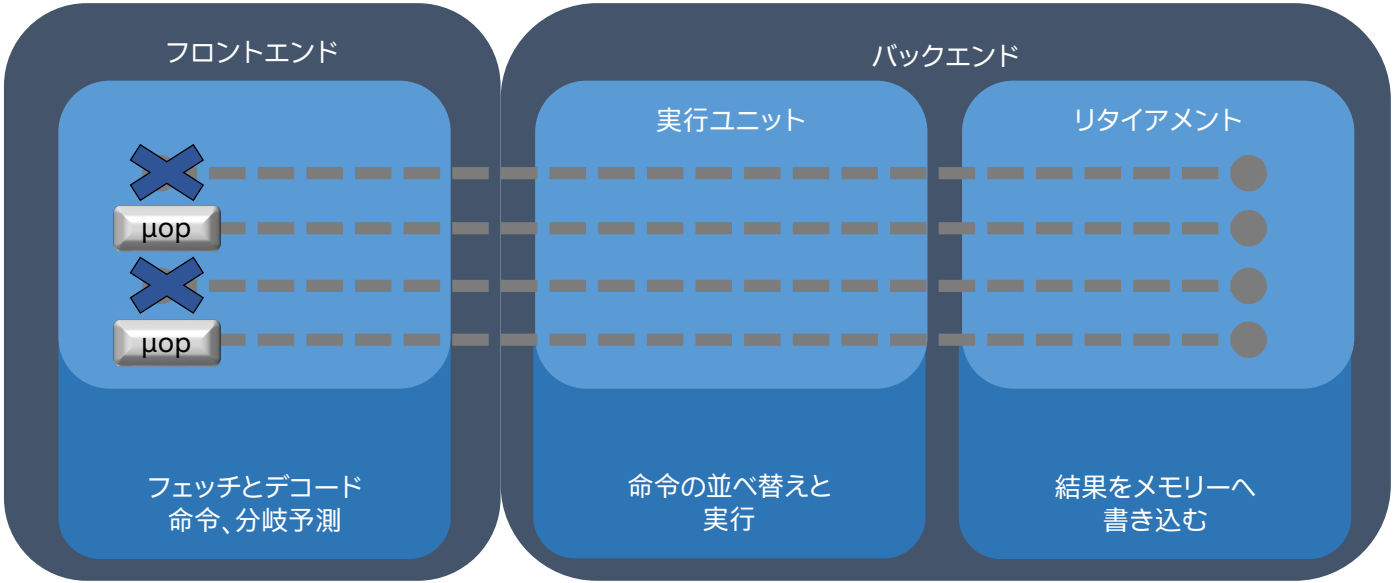
バックエンド依存

バックエンドにデータや長い実行を待機中の命令が含まれているため、フロントエンドが命令を供給しても、バックエンドがそれを受け取ることができない場合



フロントエンド依存

バックエンドは μop を受け取る準備ができているが、コードのフェッチや命令のデコードの遅延によりフロントエンドが μop を供給できない場合



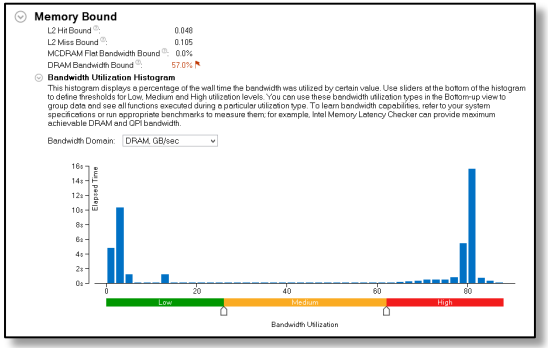
HPC アプリケーションに有益な情報を提供

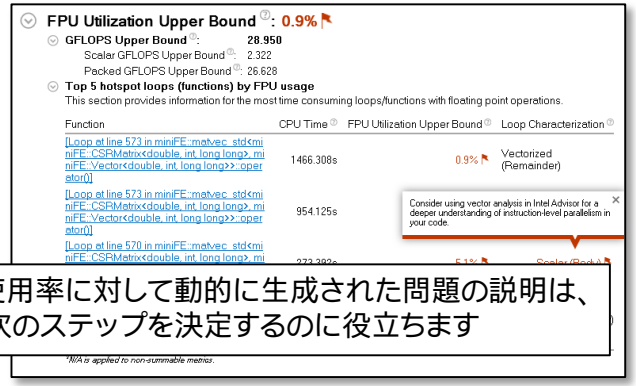
HPC Performance Characterization 解析

HPC アプリケーションの最適化に重要な情報を提示します

- CPU 利用状況 、メモリー帯域、FP ユニットの活用...

✓ FPU 負荷の比率 (%) (FPU に完全に負荷がかかっている場合は 100%、しきい値は 50%)
✓ FPU 使用率による上位 5 つのループ/関数
スカラーとパックドに GFLOPs を細分化 (ベクトル化されているかどうか)





低い FPU 使用率に対して動的に生成された問題の説明は、その原因と次のステップを決定するのに役立ちます

ベクトル化の最適化にはインテル® Advisor

スレッドのプロトタイプ生成およびベクトル化の最適化を支援するツール

ベクトル化アドバイザーによる情報収集が効果的

ループがベクトル化されているか?
ベクトル化されたループ
ベクトル化されなかったループ

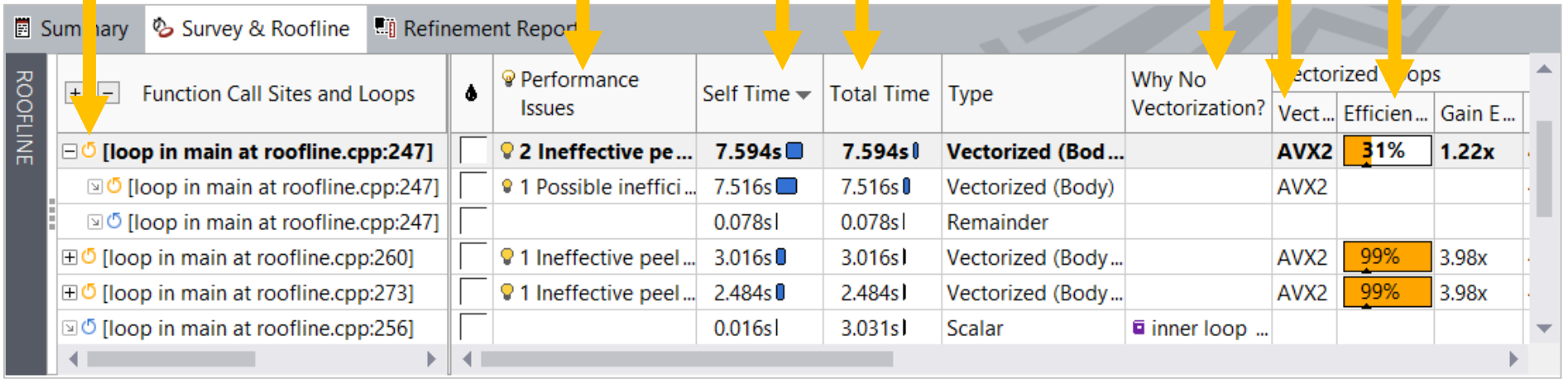
パフォーマンスを妨げているものは?
次にすべきことは?

ループに費やされた時間

ベクトル化を妨げているものは?

最新の命令セットを利用しているか?

ベクトル化の効率



Function Call Sites and Loops	Performance Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops: Vect...	Efficien...	Gain E...
[loop in main at roofline.cpp:247]	**2 Ineffective pe...**	**7.594s**	**7.594s**	**Vectorized (Bod...**		**AVX2**	**31%**	**1.22x**
[loop in main at roofline.cpp:247]	1 Possible ineffici...	7.516s	7.516s	Vectorized (Body)		AVX2		
[loop in main at roofline.cpp:247]		0.078s	0.078s	Remainder				
[loop in main at roofline.cpp:260]	1 Ineffective peel...	3.016s	3.016s	Vectorized (Body...		AVX2	99%	3.98x
[loop in main at roofline.cpp:273]	1 Ineffective peel...	2.484s	2.484s	Vectorized (Body...		AVX2	99%	3.98x
[loop in main at roofline.cpp:256]		0.016s	3.031s	Scalar	inner loop ...			

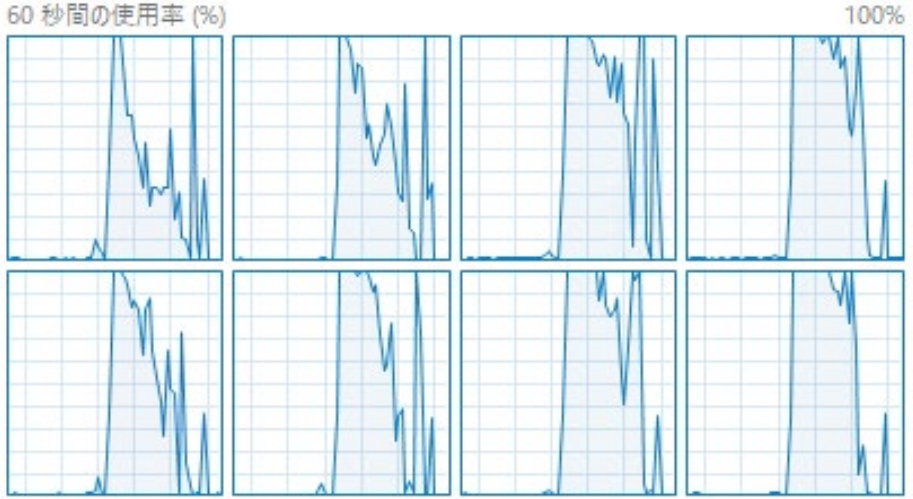
インテル® VTune Profiler を活用する

例えば、CPU 利用率にバラつきがある

全ての CPU コアが利用されているが、各コアの使用率にバラつきがある

スレッド毎に異なる処理を実行していることもあるので、意図した動作なのか見る

全スレッドを使用した計算主体の処理である場合、大きく最適化できる可能性



最適化の余地を調査する

Hotspots 解析からホットスポットを検索する

- ユーザーコードにフィルタリング

シリアル処理の場合、マルチスレッド化による複数コアの利用が可能か検討する

マルチスレッド化されている場合は、Timeline の内容に注目

- CPU 利用率が少ない場合、効率よく複数のスレッドによる処理が行われていないことを示しています

Threading 解析から制限する要因を確認します

- 複数のスレッドを扱う処理に注目

Hotspots 解析の実行

33 秒程度の実行時間

CPU Time は 162秒程度

CPU 利用率にバラつきがあることが伺える

Elapsed Time: 33.375s

- **CPU Time**: 162.001s
- Total Thread Count: 8
- Paused Time: 0s

8 スレッドで実行されている

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

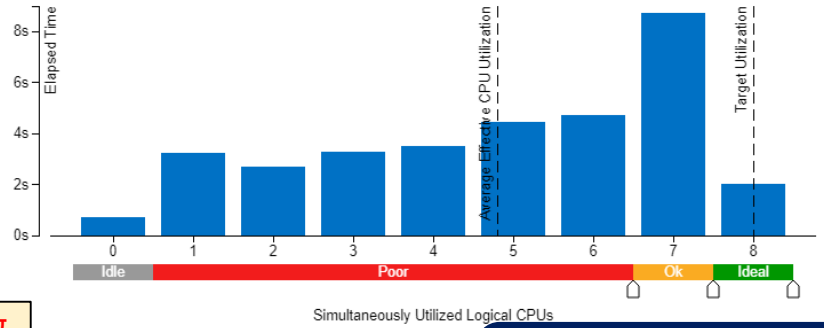
Function	Module	CPU Time
FindPrimesomp1	matmul-prime-pi.exe	146.047s
pi_calcomp1	matmul-prime-pi.exe	7.515s
matmulomp1	matmul-prime-pi.exe	6.878s
NtYieldExecution	ntdll.dll	0.825s
vcomp_barrier	VCOMP140.DLL	0.565s
[Others]		

*N/A is applied to non-su

ホットスポットは FindPrimes 関数
(OpenMP で並列化されている)

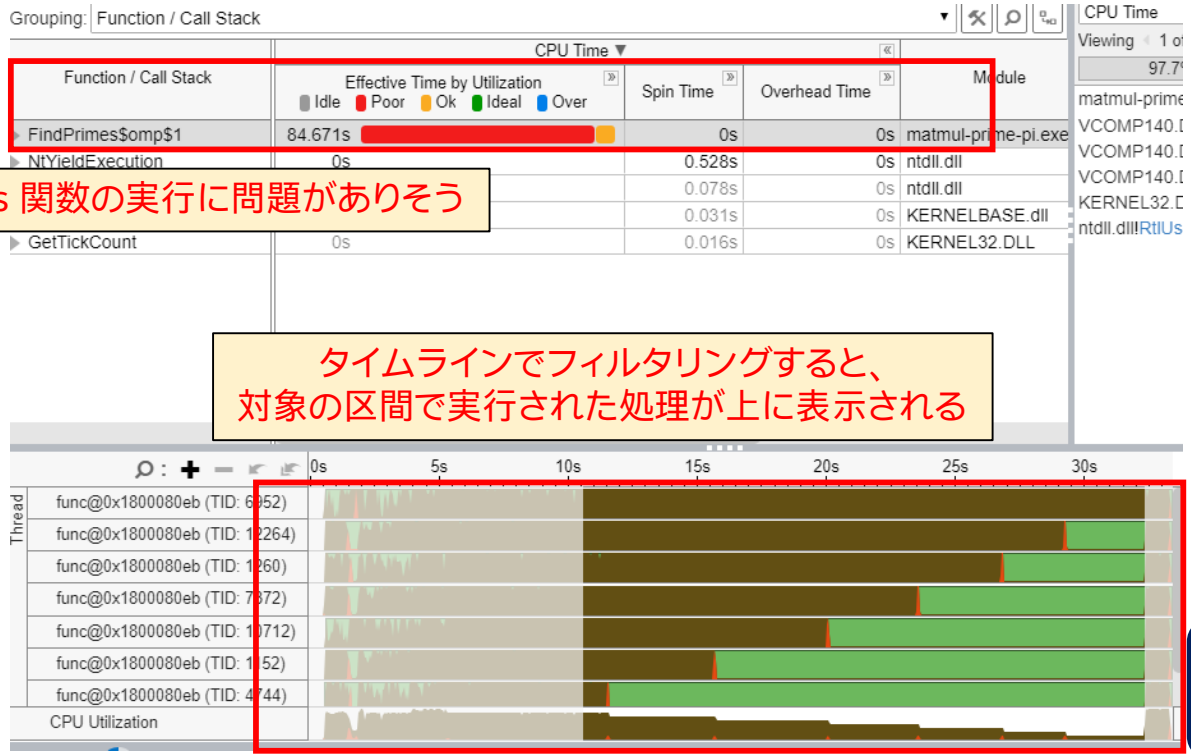
Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



次のステップ→
Bottom-up を確認

タイムラインからスレッドの利用効率が低い処理を特定

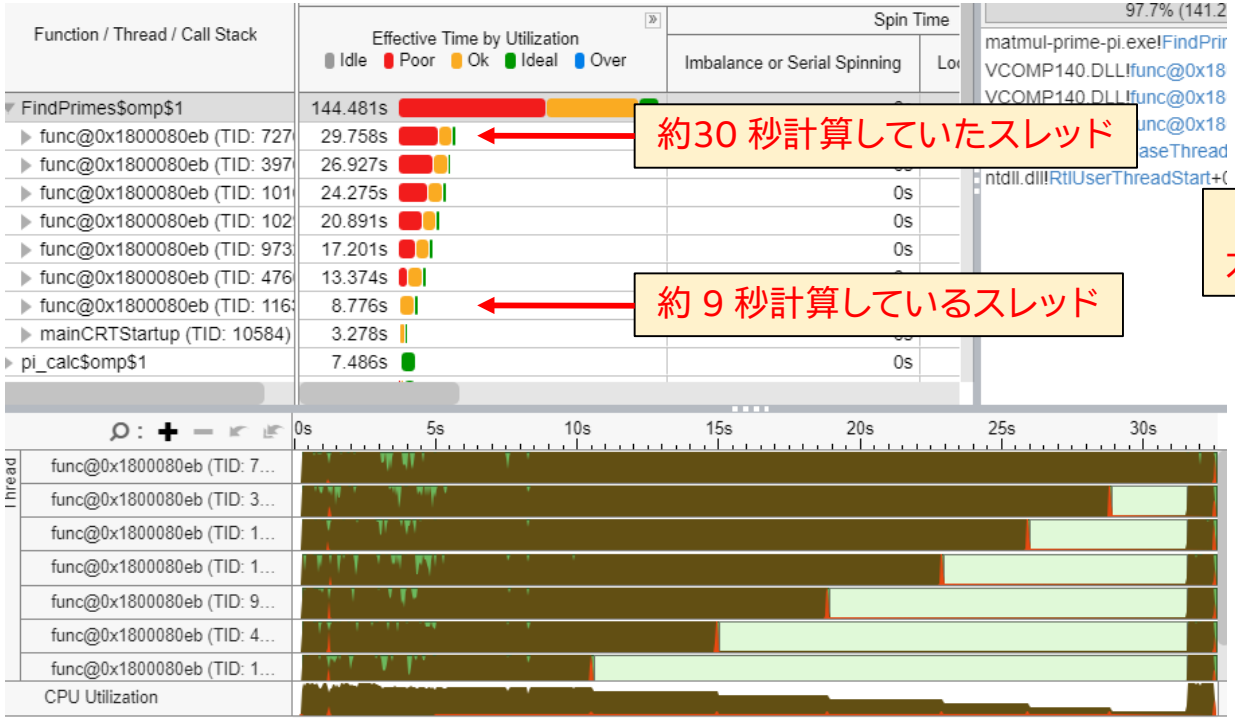


FindPrimes 関数の実行に問題がありそう

タイムラインでフィルタリングすると、
対象の区間で実行された処理が上に表示される

次のステップ→
Threading 解析を実行

Threading 解析から FindPrime に注目

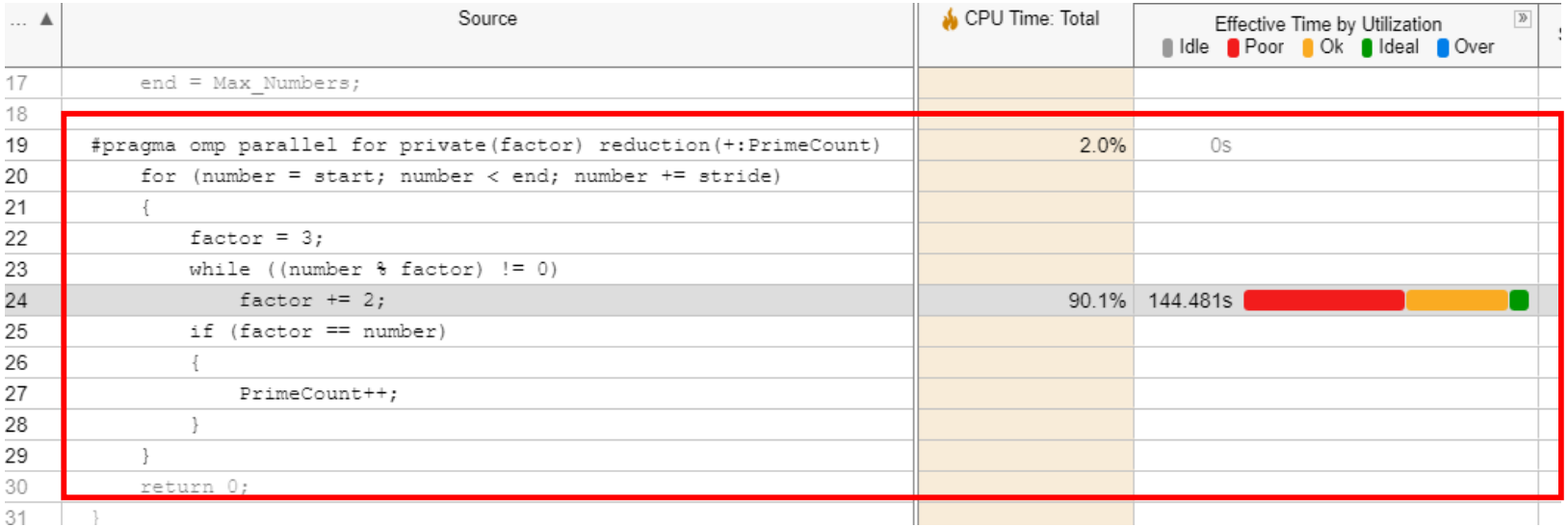


スレッド毎の実行時間に
大きな差異が発生している

次のステップ→
FindPrime のコードを確認

コードの確認

並列領域内のループ処理で、仕事量のバランスが悪い状態



...	Source	CPU Time: Total	Effective Time by Utilization (Idle Poor Ok Ideal Over)
17	`end = Max_Numbers;`		
18			
19	`#pragma omp parallel for private(factor) reduction(+:PrimeCount)`	2.0%	0s
20	`for (number = start; number < end; number += stride)`		
21	`{`		
22	`factor = 3;`		
23	`while ((number % factor) != 0)`		
24	`factor += 2;`	90.1%	144.481s
25	`if (factor == number)`		
26	`{`		
27	`PrimeCount++;`		
28	`}`		
29	`}`		
30	`return 0;`		
31	`}`		

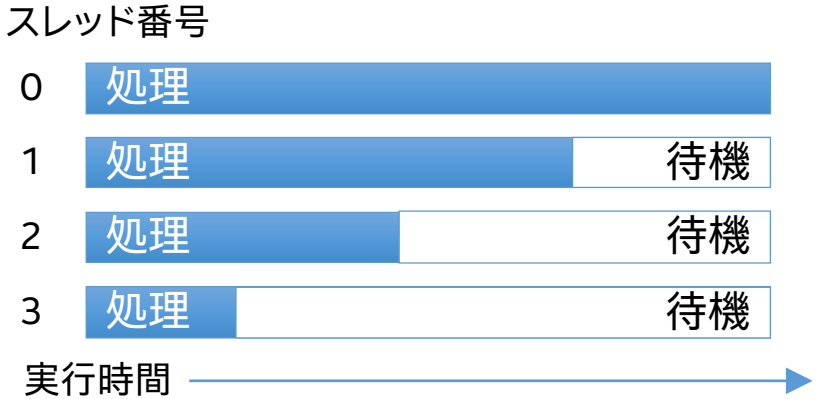
ロードバランス (処理の均一性) の維持

各スレッドの処理量を均一化する

OpenMP* では右図のような
状況が発生しやすい

```
#pragma omp parallel for
for (int i=0; i<10000; i++){
        for(int j=0; j<max; j++){
                if( … )  break;
        }
}
```

#pragma omp parallel for **schedule(static, 10)**



schedule 句: ループの分割の
大きさと割り当て方の方針を指示する

考えられる異なるケース

仕事量の差の他にも、同期処理によって待機が発生していることも

最適化の余地を調査する – もう一度掲載

Hotspots 解析からホットスポットを検索する

- ユーザーコードにフィルタリング

シリアル処理の場合、マルチスレッド化による複数コアの利用が可能か検討する

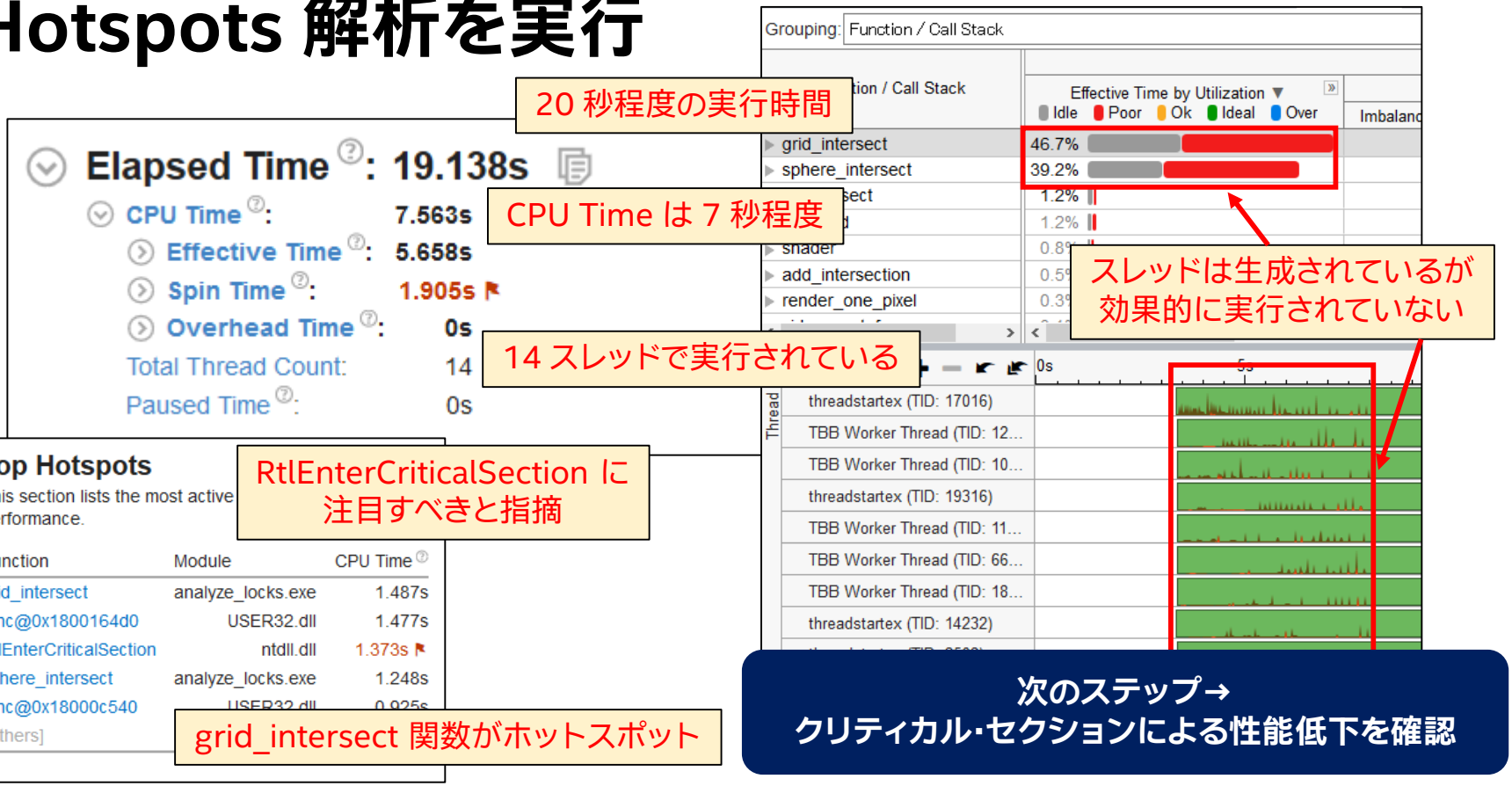
マルチスレッド化されている場合は、Timeline の内容に注目

- CPU 利用率が少ない場合、効率よく複数のスレッドによる処理が行われていないことを示しています

Threading 解析から制限する要因を確認します

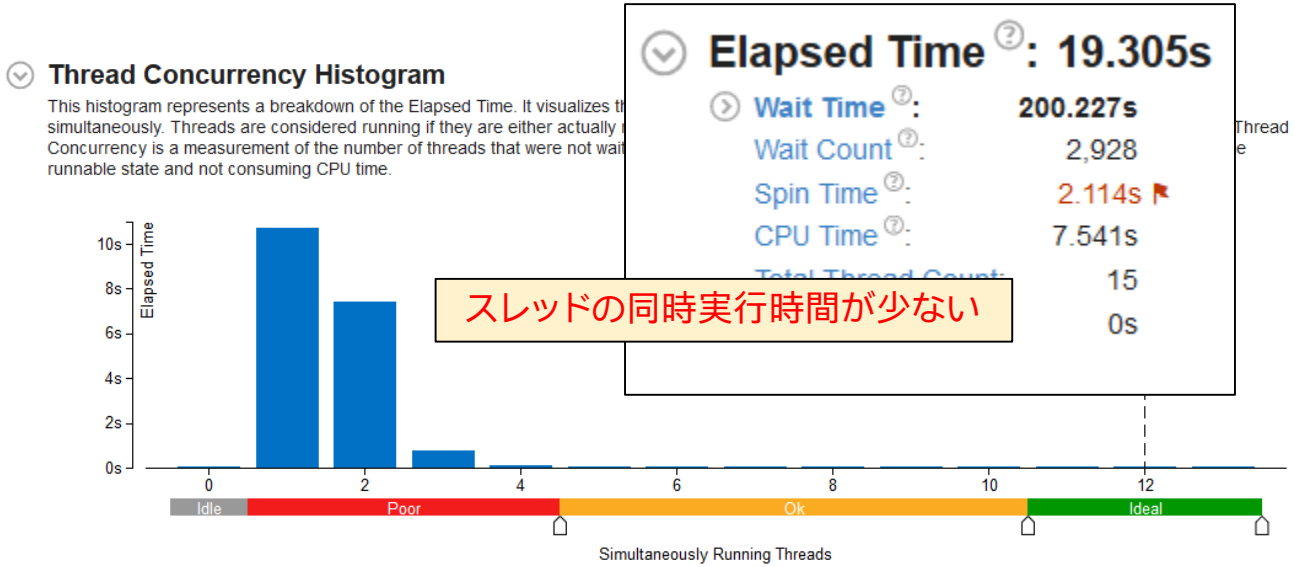
- 複数のスレッドを扱う処理に注目

Hotspots 解析を実行



Threading 解析を実行

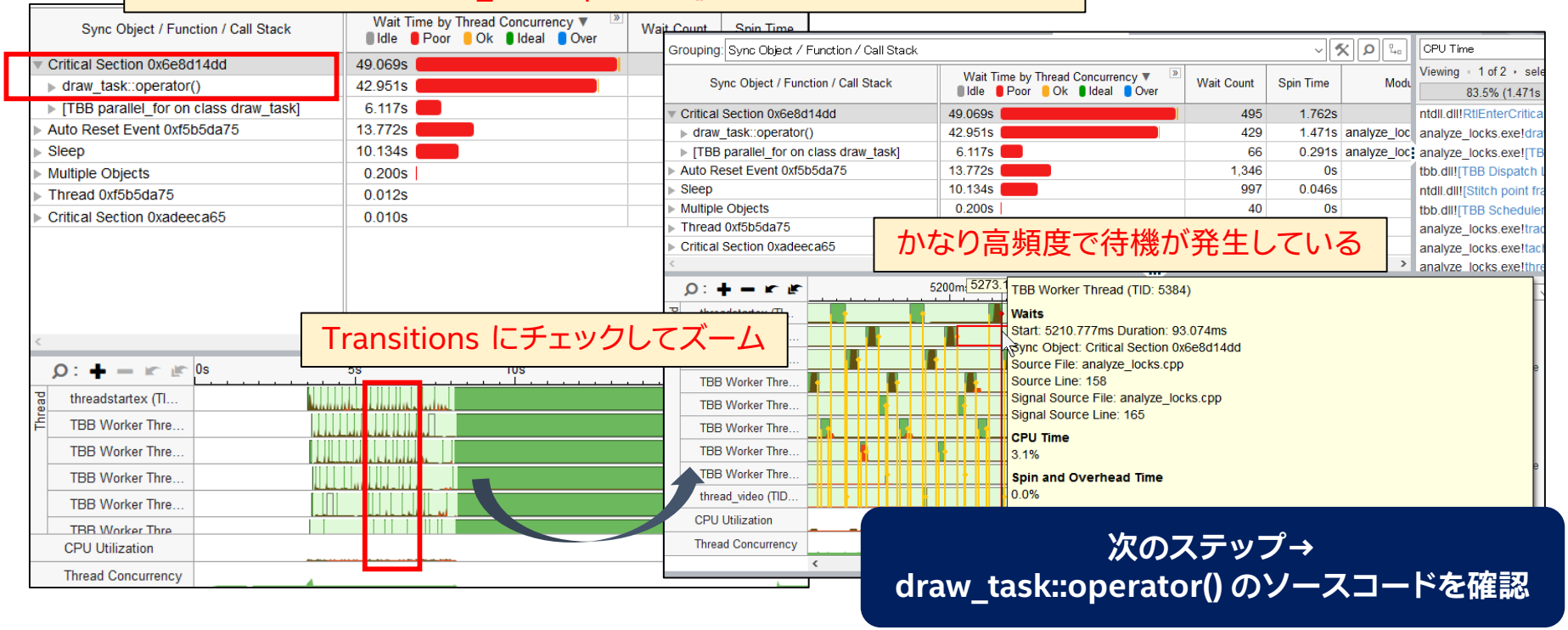
Wait Time が 200 秒



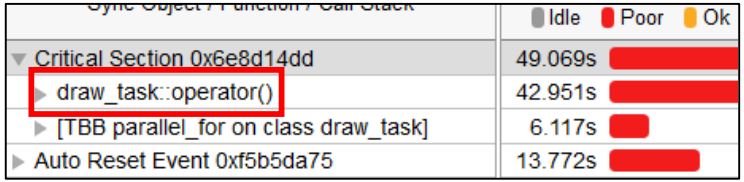
次のステップ→
Bottom-up を確認

ロック・オブジェクトに注目

クリティカル処理は draw_task::operator() 関数からコールされている

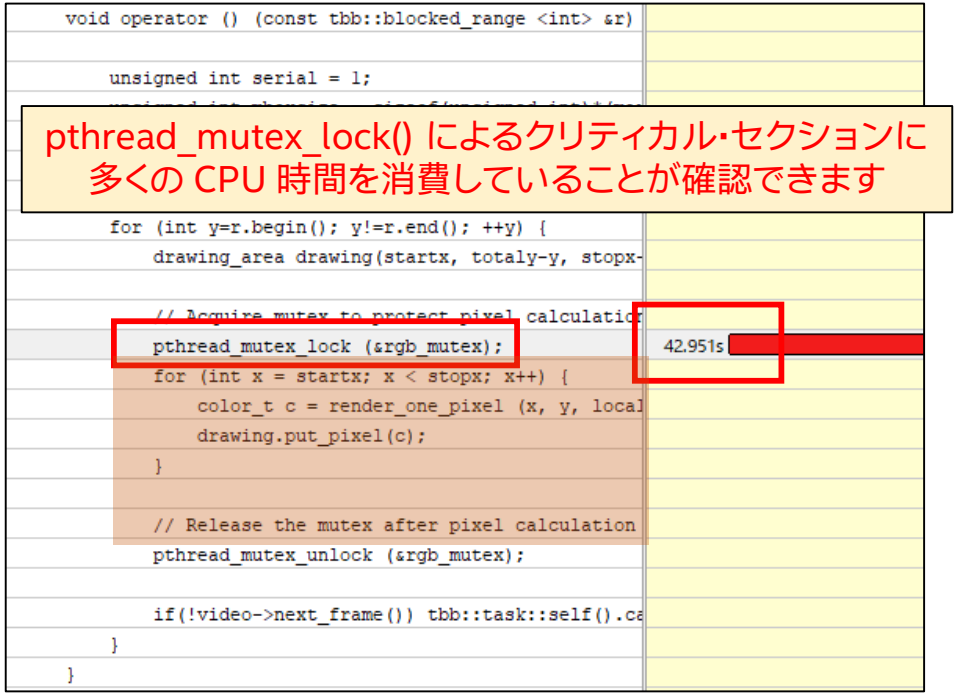


ソースコードを確認



draw_task::operator() 関数をダブルクリック

クリティカル・セクション内部の処理に注目
- 本当に必要なクリティカル処理か
- atomic 処理に置換できないか
- 並列領域外に出せないか

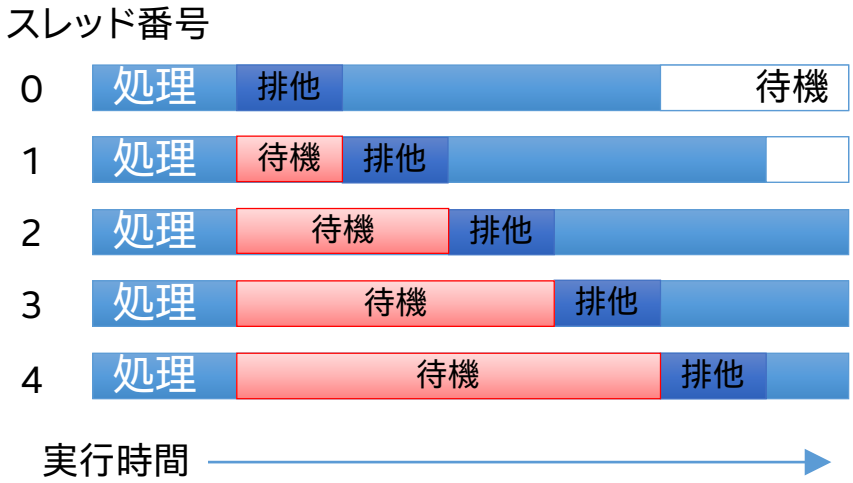


pthread_mutex_lock() によるクリティカル・セクションに多くの CPU 時間を消費していることが確認できます

排他制御による遅延

排他制御を最小限にする

```
#pragma omp parallel for
for (int i=0; i<10000; i++){
        for(int j=0; j<max; j++){
#pragma omp critical
                if( … ){
                    x = …;
                }
        }
}
```

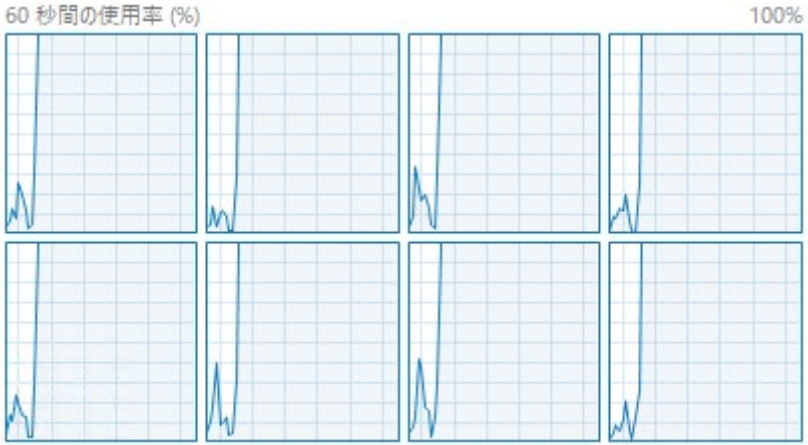


並列数が多いほど影響も大きい

同期処理自体の大きさ、実行回数と頻度に注意

CPU 使用率の高いプログラム

実行に時間は掛かっているが、CPU の各コアの使用率は高い
一見プログラムは十分にコアを活用しているように思われるが・・・・



最適化の余地を検証する

CPU コアの利用は正常にアプリケーションの計算/処理に充てられているのか?

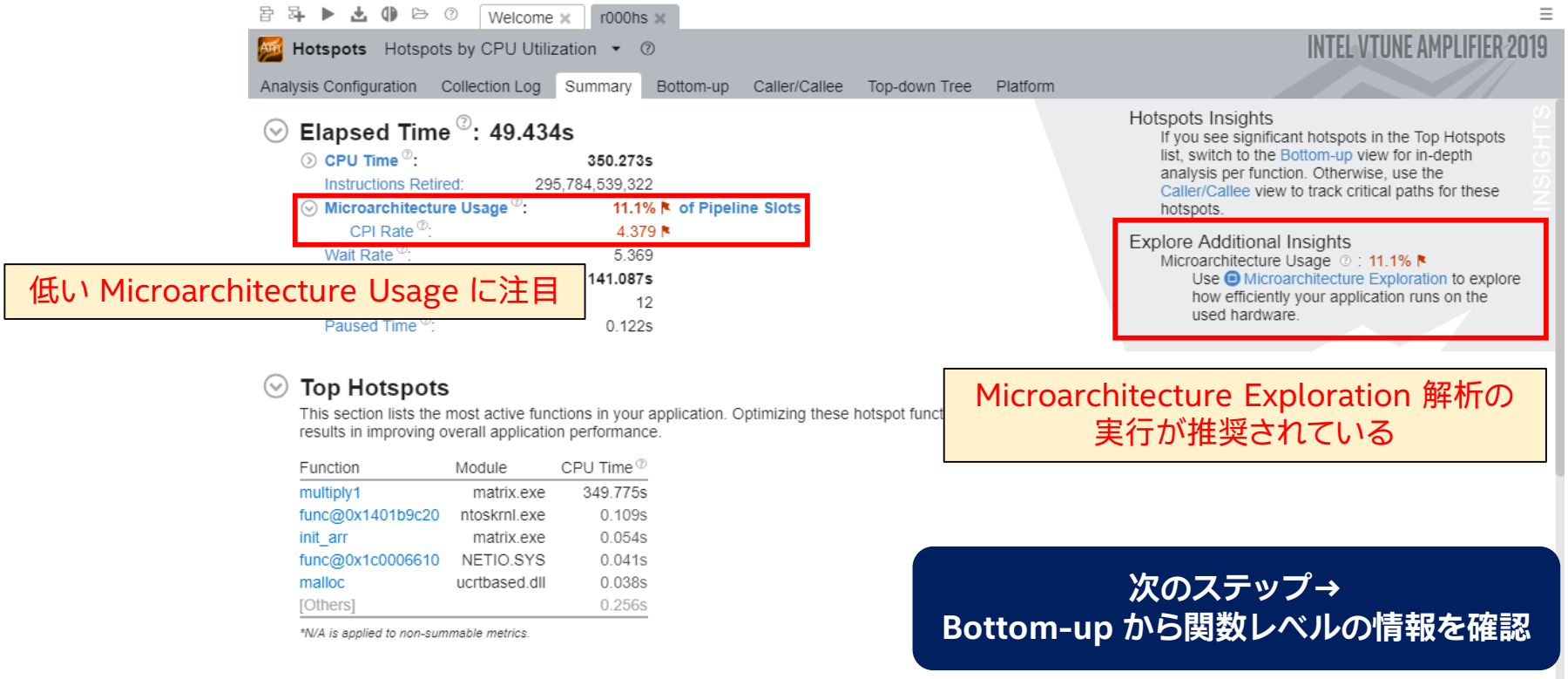
- 余分な処理に費やされていないか確認する

スレッドの待機によるものではない場合、Threading 解析では、特定が難しいことも

Hotspots 解析を実行して、パイプライン中の実行に問題が潜んでいないか確認

- パイプラインの実行効率が低い場合は、Microarchitecture Exploration を実行して、どの段階で実行が制限されているか確認

Hotspots 解析の実行

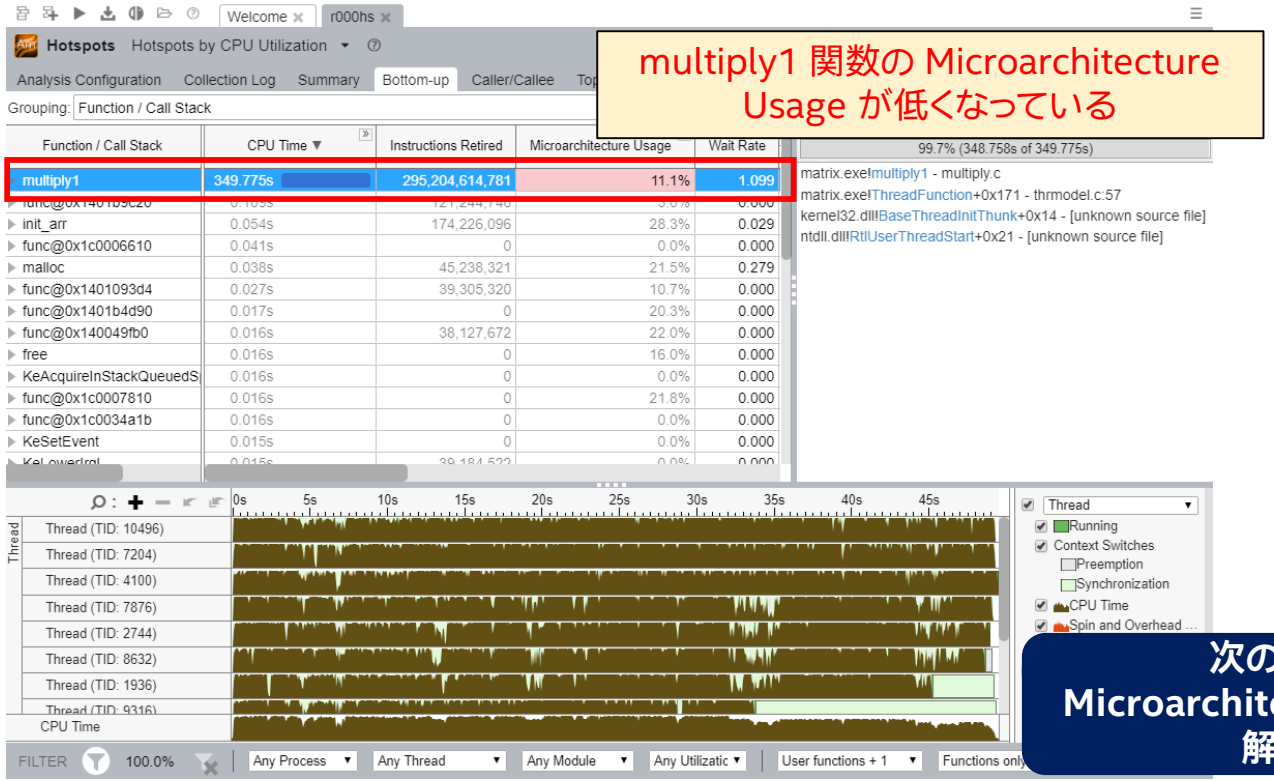


低い Microarchitecture Usage に注目

Microarchitecture Exploration 解析の実行が推奨されている

次のステップ→
Bottom-up から関数レベルの情報を確認

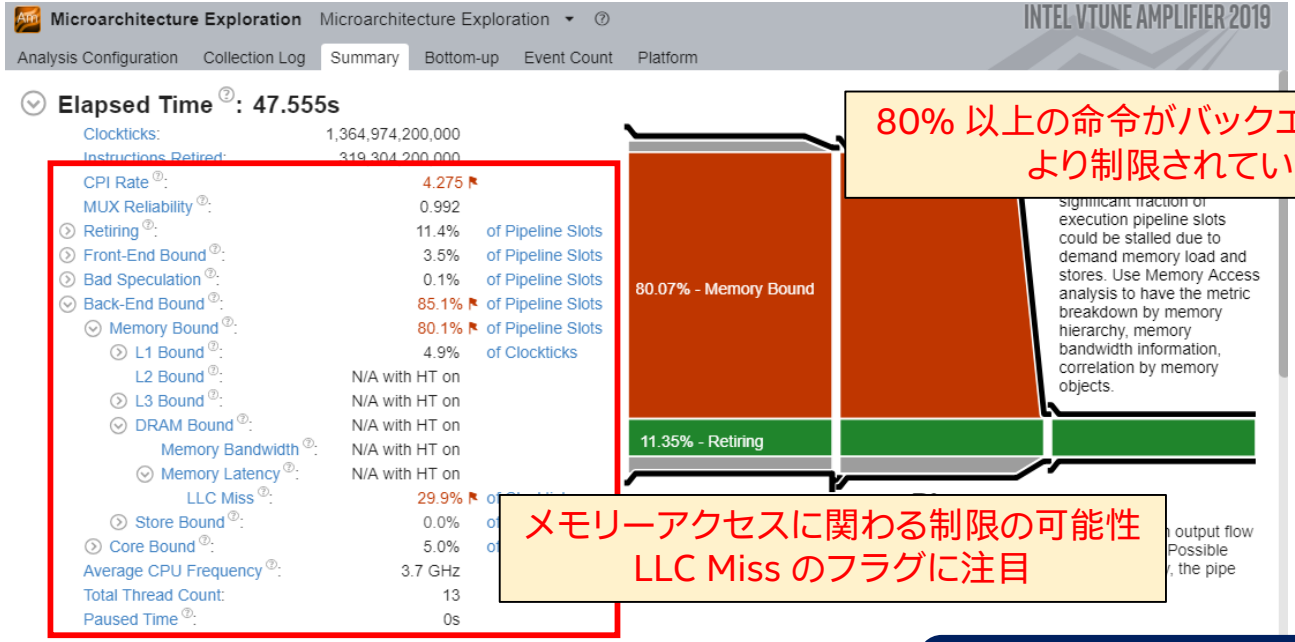
Microarchitecture Usage 項目を確認



multiply1 関数の Microarchitecture Usage が低くなっている

次のステップ→
Microarchitecture Exploration 解析の実行

Microarchitecture Exploration 解析の実行

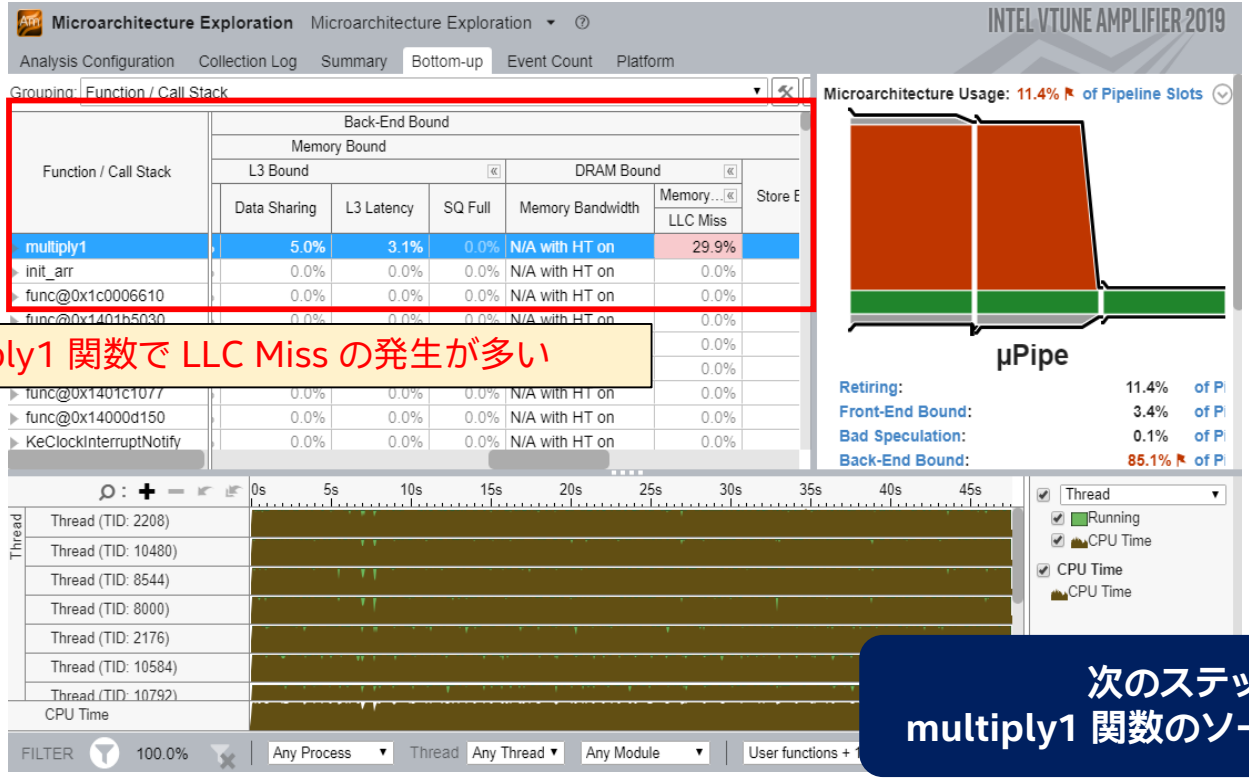


80% 以上の命令がバックエンド依存により制限されている

メモリーアクセスに関わる制限の可能性
LLC Miss のフラグに注目

次のステップ→
LLC Miss が発生している関数を確認

LLC のミスが多発している処理を特定



multiply1 関数で LLC Miss の発生が多い

次のステップ→
multiply1 関数のソースコードを確認

コードの確認


```
37                  c[i][j] = c[i][j] + a[i][k] * b[k][j];
38              }
39          }
40      }
41  }
42
43  void multiply1(int msize,
44  {
45      int i,j,k;
46
47  // Naive implementation
48      for(i=tidx; i<msize; i=i+numt) {
49          for(j=0; j<msize; j++) {
50              for(k=0; k<msize; k++) {
51                  c[i][j] = c[i][j] + a[i][k] * b[k][j];
52              }
53          }
54      }
55  }
56  void multiply2(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE c[][NUM],
```

	0.0%		
	0.0%	0.5%	0.1%
	0.1%	79.1%	4.8%
	0.0%	0.7%	0.0%

対策：メモリーへのアクセス頻度を落とす ≒ キャッシュのヒット率を上げる

- ここでは配列のアクセス順序の変更が効果的 → ループ処理の順序を入れ替え

他にも…

- **ループのアンロールを適用** ← コンパイラーオプションで調整
- **フォルスシェアリングが発生しているか** ← 解析結果の False Sharing 項目を確認
- **NUMA 構成の場合、アフィニティを決定してみる** ← Memory Access 解析も実行

まとめ

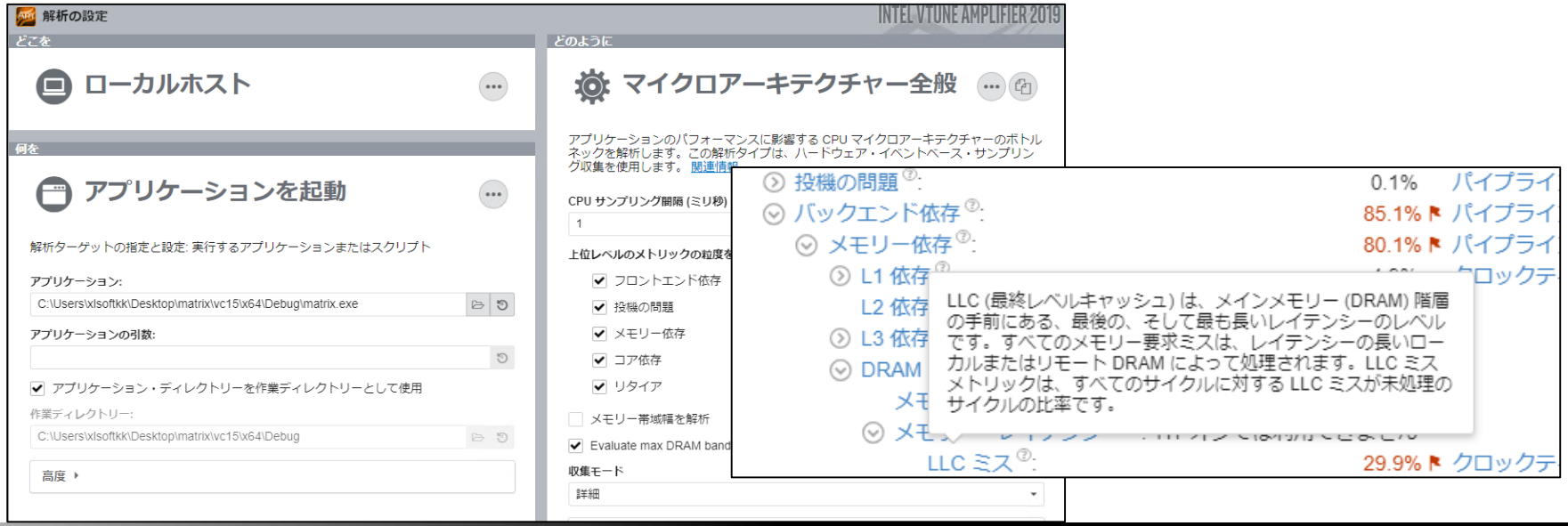
- ✓ 最適化にはホットスポットの特定と、ボトルネックの有無を調査することが重要です。
- ✓ インテル® VTune Profiler はさまざまなシナリオにおける、最適化作業で使用いただけます。
- ✓ 予め用意された解析タイプを使用すると、最適化を進めるために有益な情報を簡単に取得することができます。

インテル® VTune™ Profiler 30日間お試し版　https://www.xlsoft.com/jp/products/download/intelj.html

インテル® VTune™ Profiler 日本語化

iSUS にて日本語化パッケージが公開されています!

https://www.isus.jp/products/vtune/vtune_jp/



参考情報

インテル® VTune™ Profiler オンラインヘルプ (iSUS 翻訳版)
https://www.isus.jp/products/VTune™/help-top/
Intel 社 Web 版 (英語)
https://software.intel.com/en-us/vtune-Profiler-help

インテル® VTune™ Profiler チュートリアル (英語)
https://software.intel.com/en-us/articles/intel-vtune-Profiler-tutorials

次のステップ

プロセッサー世代別最適化ガイド (英語)
https://software.intel.com/en-us/articles/processor-specific-performance-analysis-papers

インテル® VTune™ Profiler パフォーマンス解析クックブック
https://jp.xlsoft.com/documents/intel/vtune/2019/Intel_VTune_Profiler_Performance_Analysis_Cookbook.pdf
Intel 社 Web 版 (英語)
https://software.intel.com/en-us/vtune-Profiler-cookbook

インテル® VTune™ Profiler XE の General Exploration (一般解析) がどのように動作するかを理解する - iSUS
https://www.isus.jp/products/vtune/understanding-how-general-exploration-works-in-vtune/

補足情報

サンプリング・ドライバーの動作確認

Windows* ターゲット

- amplxe-sepreg.exe -s の実行

```
C:¥Program Files (x86)¥IntelSWTools¥VTune Amplifier 2018¥bin64>amplxe-sepreg.exe -s
Checking sepdrv4_1 driver path...OK
Checking sepdrv4_1 service...
Driver status: the sepdrv4_1 service is running
Checking sepdal driver path...OK
Checking sepdal service...
Driver status: the sepdal service is running
Checking socperf2_0 driver path...OK
Checking socperf2_0 service...
Driver status: the socperf2_0 service is running
```

サンプリング・ドライバーがロードされているか確認

配置先：C:¥Program Files (x86)¥IntelSWTools¥VTune Profiler¥bin32

サンプリング・ドライバーの動作確認

Linux*ターゲット

./insmod-sep -q の実行

配置先:/opt/intel/VTune_Profiler/sepdk/src

```
[xlsoftkk@knl src]$ ./insmod-sep -q
pax driver is loaded and owned by group "vtune" with file permissions "666".
socperf2_0 driver is loaded and owned by group "vtune" with file permissions "666".
sep4_1 driver is loaded and owned by group "vtune" with file permissions "666".
vtsspp driver is loaded and owned by group "vtune" with file permissions "666".
```

サンプリング・ドライバーがロードされているか確認

バージョン 2018 Update1 以降では ./amplxe-self-checker.sh を実行

- 適切にドライバーがインストールされ、システムがパフォーマンス・データを収集できるか検証するセルフチェック・スクリプト

配置先: /opt/intel/VTune_Profiler/bin64

解析を始める前に

アプリケーションの準備

Visual C++*、GNU* gcc、およびその他の C++ コンパイラー、および Fortran コンパイラーで生成されたバイナリーを解析することができます

- インテル® コンパイラー以外でもOK

ソースコードと関連付けて解析を行うためシンボル情報を生成します

- デバッグ情報の付加
  - Windows*: /Zi
  - Linux*: -g
  - デフォルトの最適化 (/O2 および -O2) が無効になるため、明示的に指定し直す

デバッグ情報がないとループや関数のソースとの関連付けができません

インテル® コンパイラー利用時に設定したいオプション

追加のオプションを指定することで、解析結果により多くの関数、ループ、ソース情報を追加できます

関数のインライン展開用デバッグ情報の追加 (icc/ifort)

- Windows*: /debug:inline-debug-info , Linux*: -debug inline-debug-info

OpenMP* 利用時のソースコード情報 (icc/ifort)

- Windows*: /Qparallel-source-info=2 , Linux*: -parallel-source-info=2

インテル® スレッディング・ビルディング・ブロック (インテル® TBB) を使用している場合
Windows*: /DTBB_USE_THREADING_TOOLS
Linux*: -DTBB_USE_THREADING_TOOLS

データ収集の対象となる範囲を制御する

API によりソースコードレベルの解析を制御できます

➢Fortran、C/C++ コードが対象

主要な API 機能

- **void __itt_pause (void):** アプリケーションのデータ収集を一時停止します
- **void __itt_resume (void):** データ収集を再開します
- **void __itt_detach (void):** データ収集をデタッチ (終了) します

使用ヘッダーとリンク・ライブラリー

Windows*:

<install_dir>¥include¥ittnotify.h

<install_dir>¥lib32¥libittnotify.lib または <install_dir>¥lib64¥libittnotify.lib

Linux*:

<install_dir>/include/ittnotify.h

<install_dir>/lib32/libittnotify.a または <install_dir>/lib64/libittnotify.a

解析時にプロジェクト設定画面で API に使用を有効に

MPI アプリケーションの解析

コマンドベースで解析を実行する

```
mpirun -n 16 -ppn 4 -l amplxe-cl -collect hotspots -trace-mpi -result-dir my_result -- my_app.out
```

- 全ノードで解析を実行すると、各計算ノードごとの解析結果が表示されます
  - 上記だと 4 つの解析結果を生成します
- -trace-mpi
  インテル® MPI ライブラリー以外が提供する MPI ランチャーから実行するときに指定

インテル® MPI ライブラリー (バージョン5.0.2以降) を使用している場合

```
mpirun -gtool "amplxe-cl -collect hotspots -result-dir my_result:7,5" my_app.out
```

-gtool オプションは MPI アプリケーションの解析に必要な設定を簡易化します

お問い合わせはこちらまで

https://www.xlsoft.com/jp/qa