インテル oneAPI ツールキット
汎用CPUノード 高速化技法の基礎

エクセルソフト株式会社

はじめに

- この資料は2024年7月9日時点の情報をもとにしています
- インテル® コンパイラーの利用方法は SQUID のガイドをご参考ください
- 記載しているコンパイラー・オプションは
  Linux、Windows それぞれで異なる場合があります
  - ✓ オプション名の先頭は Linux だと - 、Windows は / です

    Linux 例: -help

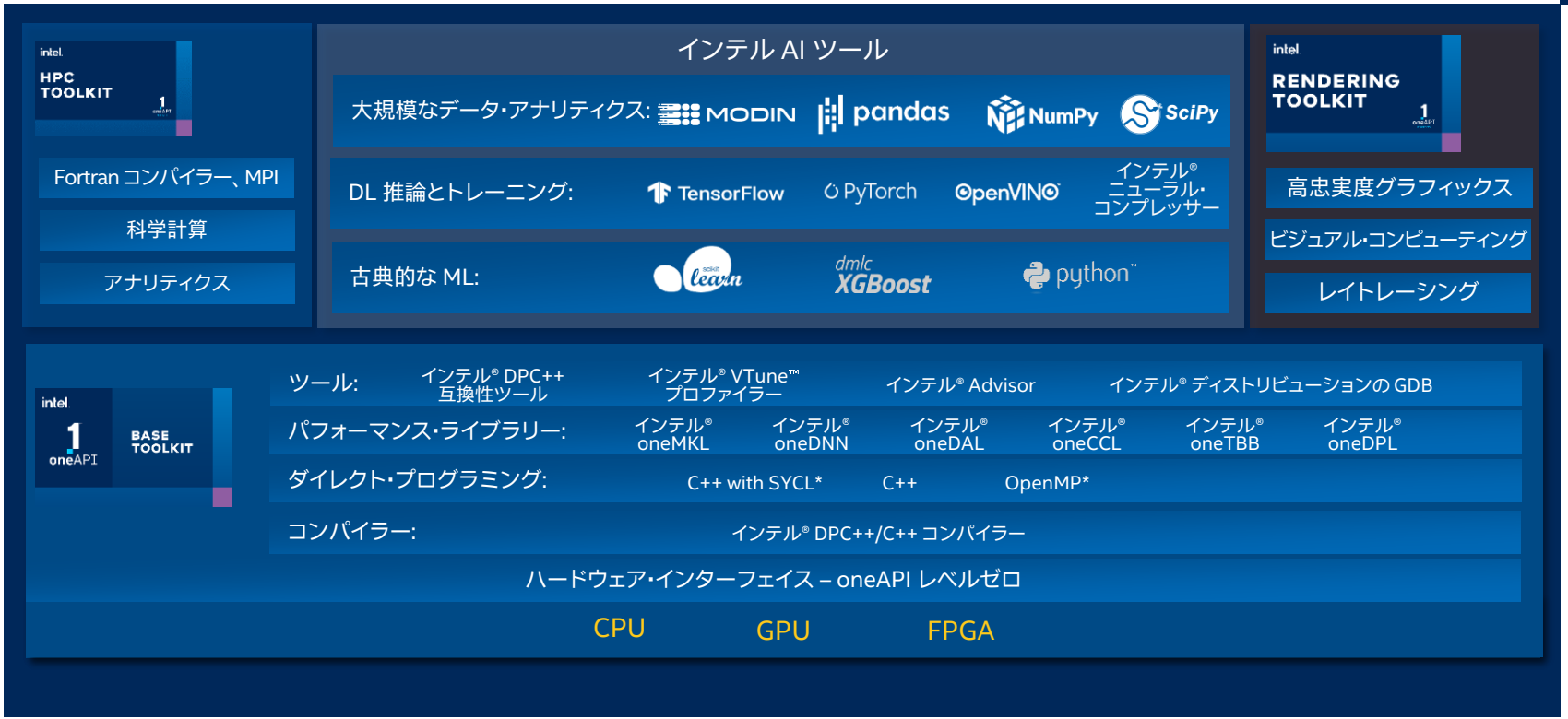
    Windows 例: /help

    資料中のコンパイラー オプションは Linux 向けで表記しています
- 資料中に示すソースコードは C/C++, もしくは Fortran です

ご紹介内容

- インテル® ソフトウェア開発ツールの概要
  - ✓ インテル® oneAPI ベース & HPC ツールキット
- 提供されるコンパイラーについて
- 主要なコンパイラーオプションの適用
  - ✓ 特にパフォーマンスへ影響しやすいもの
- インテル® VTune™ プロファイラーのご利用
  - ✓ アプリケーションの解析と調査

インテル® ソフトウェア開発ツール
powered by oneAPI



インテル® oneAPI ベース & HPC ツールキット

コンパイラーや実行環境

- インテル® Fortran コンパイラー・クラシック
- インテル® Fortran コンパイラー
- インテル® oneAPI DPC++/C++ コンパイラー
- インテル® ディストリビューションの Python*

ツール/ライブラリー

- インテル® MPI ライブラリー
- インテル® oneAPI スレッディング・ビルディング・ブロック (インテル® oneTBB)
- インテル® oneAPI DPC++ ライブラリー (インテル® oneDPL)
- インテル® oneAPI コレクティブ・コミュニケーション・ライブラリー (インテル® oneCCL)
- インテル® oneAPI マス・カーネル・ライブラリー (インテル® oneMKL)
- インテル® oneAPI ディープ・ニューラル・ネットワーク・ライブラリー (インテル® oneDNN)
- インテル® oneAPI データ・アナリティクス・ライブラリー (インテル® oneDAL)
- インテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP)
- インテル® DPC++ 互換性ツール
- oneAPI ベース・ツールキット用 インテル® FPGA アドオン

解析/デバッグツール

- インテル® Inspector
- インテル® Trace Analyzer & Collector
- インテル® VTune™ プロファイラー
- インテル® Advisor
- インテル® ディストリビューションの GDB

個別ダウンロードが必要なコンポーネントを含みます

インテル® **HPC** ツールキット

インテル® oneAPI **ベース**・ツールキット

インテル® oneAPI ベース & HPC ツールキット

- インテル® Parallel Studio XE の後継製品
  - ✓ コンパイラーやライブラリーなどのコンポーネントを引き続き提供
- インテル® アーキテクチャ上で実行するアプリケーションの最適化を支援
  - ✓ インテル® コンパイラー
    - 後述します
  - ✓ インテル® oneMKL
    - インテル® MKL の後継
    - BLAS、LAPACK、FFT などの計算処理を実装した MPI 対応の数値演算ライブラリー
    - プロセッサーの ISA を認識して最適な関数を実装するディスパッチ機能
  - ✓ インテル® MPI ライブラリー
    - オープンソースの MPICH の仕様を実装した MPI ライブラリー
    - 時間を消費しやすい集合操作などをチューニングするためのユーティリティを提供

インテル® コンパイラー

- CPU、GPU を含むインテル製ハードウェア向けに
  最適化されたアプリケーションの開発できるように設計
  - ✓ 最新のインテル® アーキテクチャ向けに継続してアップデートされています
    例: Granite Rapids, Sierra Forest, Lunar Lake, Arrow Lake
- 2020年に oneAPI ツールキットがリリースしたタイミングにて
  新しいコンパイラーが提供されるようになりました
  - ✓ インテル® oneAPI DPC++/C++ コンパイラー
  - ✓ インテル® Fortran コンパイラー
- インテル Parallel Studio XE に含まれる
  コンパイラーは名称が変更されています
  - ✓ インテル® C++ コンパイラー・クラシック
  - ✓ インテル® Fortran コンパイラー・クラシック

コンパイル コマンド

C プログラムをコンパイルするシンプルなコマンド例

```
> icx ./simple.c -o muapp.out
```

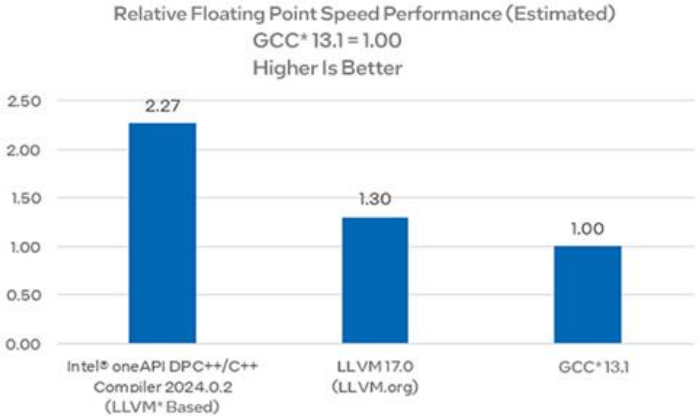
- それぞれのコンパイラーに対応した複数のコンパイラードライバーを実装しています
- icx/icpx
  - ✓ インテル® oneAPI DPC++/C++ コンパイラー
- ifx
  - ✓ インテル® Fortran コンパイラー
- icc/icpc
  - ✓ インテル® C++ コンパイラー・クラシック
- ifort
  - ✓ インテル® Fortran コンパイラー・クラシック

インテル® oneAPI DPC++/C++ コンパイラー (icx/icpx)

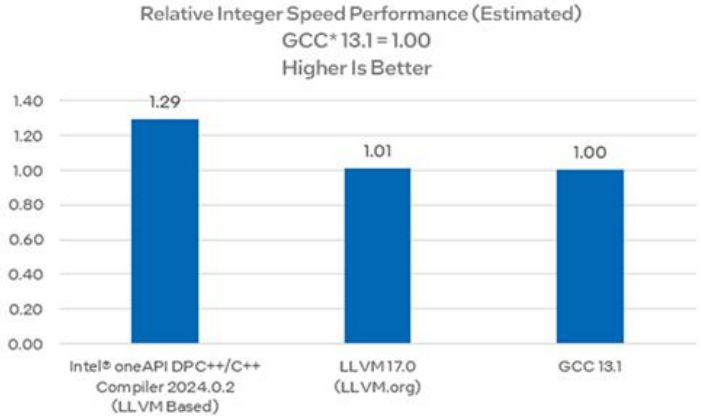
- clang フロントエンドおよび LLVM バックエンドを採用
  - ✓ + 独自のオプティマイザーとコード生成を実装
  - ✓ clang のコンパイルオプションを認識しつつ、インテル® コンパイラー拡張のオプションを使用できます
- SYCL* 2020 によるヘテロジニアス プログラミングをサポート
  - ✓ バージョン 2024 にて SYCL* 2020 に準拠した初めてのコンパイラーとして認定されました

Intel® Compilers Boost C++ Application Performance on Linux*

Performance Advantage Relative to Other Compilers on Intel® Xeon® Platinum 8580 Processor



Estimated: internal measurement of the geometric mean of the C/C++ workloads from the SPECspeed* 2017 Floating Point suite (baseline)



Estimated: internal measurement of the geometric mean of the C/C++ workloads from the SPECspeed* 2017 Integer suite (baseline)

Testing Date: Performance results are **based on testing by Intel as of December 13, 2023** and may not reflect all publicly available security updates.

Configuration Details and Workload Setup: Intel® Xeon® Platinum 8580 CPU, QS-A1 stepping, at 2.00 GHz, 2 socket, Intel® Hyper-Threading Technology on, turbo on, 32 G x16 DDR5 4800 (1DPC). Ubuntu* 22.04 LTS, 6.2.0-33-generic. Software: Intel® oneAPI DPC++/C++ Compiler for applications running on Intel® 64 v2024.0.2 build 20231213. GCC* v13.1.0, Clang/LLVM* v17.0.0. SPECint®_speed_base_2017 compiler switches: Intel oneAPI DPC++/C++ Compiler: -xsapphirerapids -O3 -ffast-math -flto -mfpmath=sse -funroll-loops -qopt-mem-layout-trans=4 -fiopenmp -ljemalloc. GCC: -march=sapphirerapids -mfpmath=sse -Ofast -funroll-loops -flto -fopenmp -ljemalloc. Clang/LLVM: -march=sapphirerapids -mfpmath=sse -Ofast -funroll-loops -flto -fopenmp -ljemalloc. qkmalloc used for Intel compilers. jemalloc 5.0.1 used for GCC and LLVM. SPECfp®_speed_base_2017 compiler switches: Intel oneAPI DPC++/C++ Compiler: -xsapphirerapids -Ofast -ffast-math -flto -mfpmath=sse -funroll-loops -qopt-mem-layout-trans=4 -fiopenmp -ljemalloc. GCC: -march=sapphirerapids -mfpmath=sse -Ofast -funroll-loops -flto -fopenmp -ljemalloc. Clang/LLVM: -march=sapphirerapids -mfpmath=sse -Ofast -funroll-loops -flto -fopenmp -ljemalloc. jemalloc 5.0.1 used for Intel compilers, GCC and LLVM.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

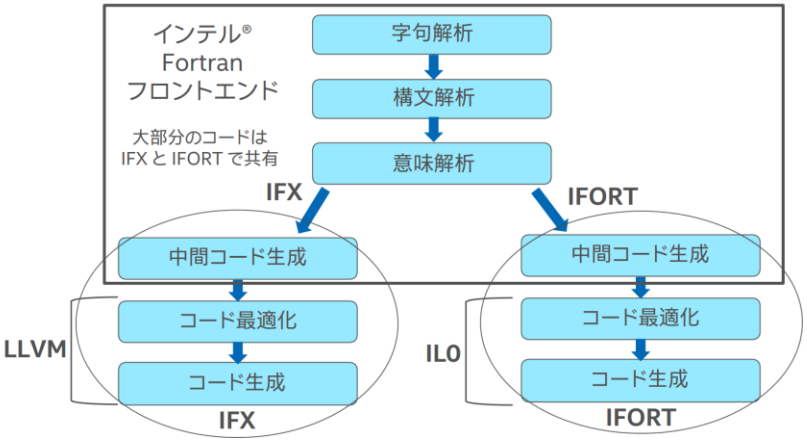
Performance varies by use, configuration, and other factors. Learn more at www.Intel.com/PerformanceIndex. Your costs and results may vary. *Other names and brands may be claimed as the property of others.

ベンチマーク：出典

インテル® Fortran コンパイラー (ifx)

- 独自の ifort フロントエンド と LLVM バックエンド を採用
  - ✓ 最適化やコード生成における基盤が異なります

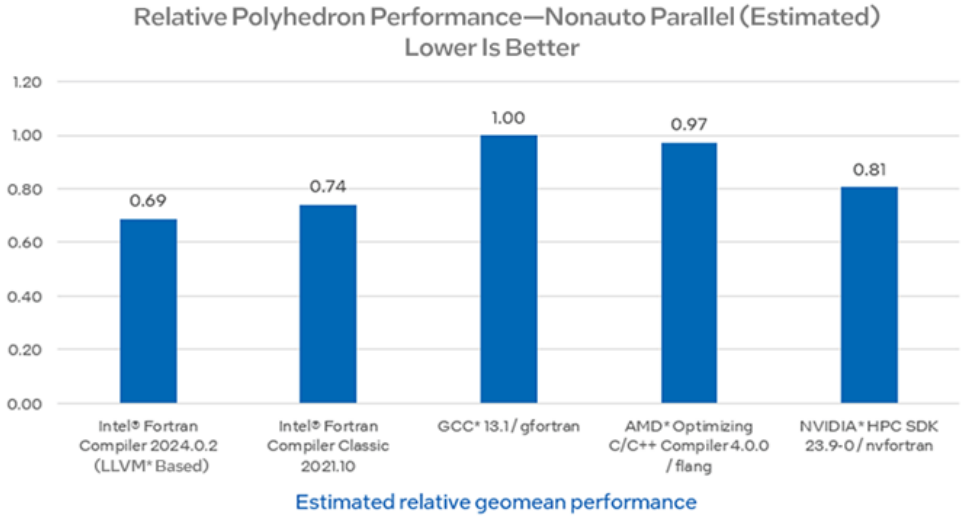
自動ベクトル化
インライン展開
ループ アンロール
プロシージャー間の最適化
プロファイルに基づく最適化
浮動小数点演算の制御
最適化レポート
etc...



- インテル® oneAPI ツールキット 2021 よりベータ版が提供開始されバージョン 2023 に正式リリースされました

Intel® Fortran Compiler Boosts Application Performance on Linux*

Performance Advantage Measured by the Polyhedron Fortran Benchmark on an Intel® Xeon® Platinum 8580 Processor



Testing Date: Performance results are **based on testing by Intel as of December 13, 2023** and may not reflect all publicly available security updates.

Configuration Details and Workload Setup: Intel® Xeon® Platinum 8580 CPU at 2.00 GHz, 2 socket, Intel® Hyper-Threading Technology on, turbo on, 32 G x16 DDR5 4800 (IDPC). Ubuntu* 22.04 LTS, 6.2.0-33-generic. Software: Intel® Fortran Compiler for applications running on Intel® 64 v2024.0.0 build 20231017, Intel Fortran Compiler Classic for applications running on Intel 64 v2021.10.0 build 20230609_000000, GCC* v13.1.0 / gfortran, AMD* Optimized C/C++ Compiler 4.0.0 / flang – AMD for Clang v14.0.6 (CLANG: AOCC_4.0.0-Build#434 2022_10_28, based on llvm-mirror v14.0.6), NVIDIA* HPC SDK 23.9-0 / 23.9-0 64-bit target on x86-64 Linux*. Nonauto parallel compiler switches: Intel Fortran Compiler: -Ofast -XCORE-AVX512 -flto -nostandard-realloc-lhs -qopt-dynamic-align. Intel Fortran Compiler Classic: -O3 -XCORE-AVX512 -ipo -nostandard-realloc-lhs -no-prec-div. GCC / gofrotran: -march=native -funroll-loops -O3. AMD Optimized C/C++ Compiler / flang: compile: -O3 -flto -march=znver2 -funroll-loops -ffast-math -mllvm -disable-indvar-simplify -mllvm -unroll-aggressive -mllvm -unroll-threshhold=150. NVIDIA HPC SDK / nvfortran: -fast -Mipa=fast, inline -Mallocatable=03 -Mfprelaxed -Mstack_arrays.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

Performance varies by use, configuration, and other factors. Learn more at www.Intel.com/PerformanceIndex. Your costs and results may vary. *Other names and brands may be claimed as the property of others.

ベンチマーク：出典

コンパイラーの選択

- CPU 上での実行を対象としたプログラミングにおいて
  どのコンパイラーでも利用いただけます
  - ✓ icc/icpc, ifort および icx/icpx, ifx
  - ✓ インテル® GPU 向けプログラミングは新しいコンパイラーのみ対応
- 本資料では icc/icpc, ifort で利用可能なオプションを説明します
  - ✓ 新しいコンパイラーで利用できない、もしくは変更された
    オプションを説明する際には必要に応じて補足します

主要なコンパイラーオプション

- パフォーマンスに影響しやすいオプション
  - ✓ 一般的な最適化
  - ✓ プロセッサー固有の最適化
  - ✓ 自動並列化
  - ✓ プロシージャー間の最適化 (IPO)
  - ✓ プロファイルに基づく最適化 (PGO)
- 演算結果の再現性に関するオプション
  - ✓ 浮動小数点演算の制御

一般的な最適化オプション

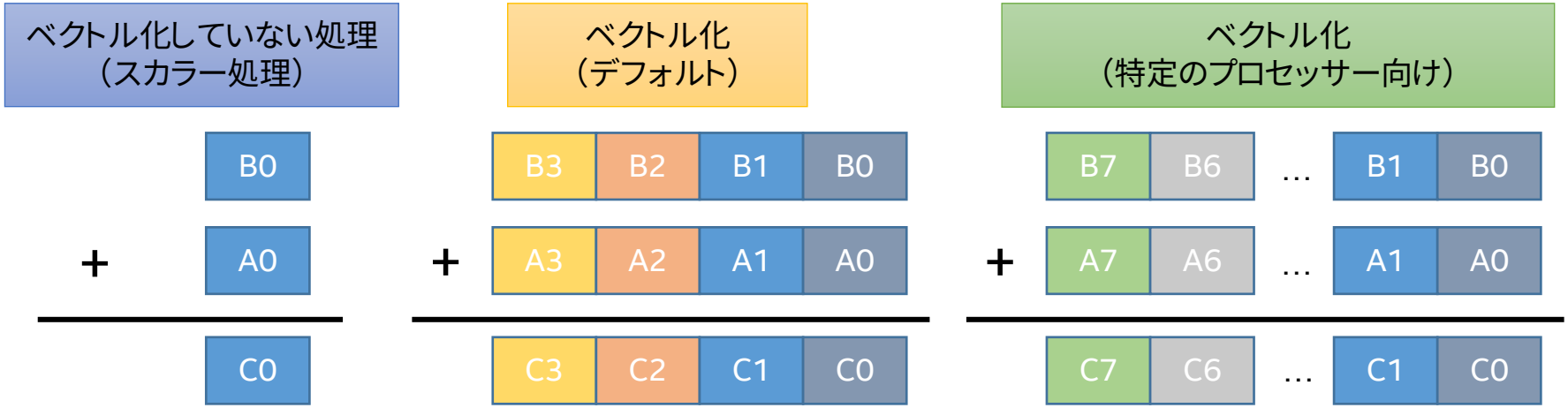
- -O0
  - ✓ 最適化を行いません
    -g オプションにより自動的に指定されます
- -O1
  - ✓ コードサイズを増やさない限定的な最適化を行います
- -O2
  - ✓ 自動ベクトル化を含む、多くの最適化を有効にします
    デフォルトの最適化レベルです
- -O3
  - ✓ -O2 レベルの最適化に加えて、ループ処理と
    メモリーアクセスについてより積極的に最適化を行います
  - ✓ O2 以下と比較してコンパイルに時間がかかるようになります

自動ベクトル化

```
for (int i = 0; i < count; i++)
{
    C[i] = A[i] + B[i];
}
```

- ループ処理や関数についてハードウェアでサポートされる SIMD 操作を用いるコードを生成して実行効率を高めます

SIMD : Single Instruction Multiple Data



SIMD 操作のサポート

[SSE] float 4 要素を一度に演算
addps %xmm1, %xmm2

[AVX] float 8 要素を一度に演算
vaddps %ymm1, %ymm2, %ymm3

[AVX512] float 16 要素を一度に演算
vaddps %zmm1, %zmm2, %zmm3

- インテル® ストリーミング SIMD 拡張
  - ✓ インテル® SSE2,インテル® SSE3,インテル® SSE4
  - ✓ 128bit 長のレジスタ (xmm)を使用した SIMD 操作を実行します
- インテル® Advanced Vector Extensions
  - ✓ インテル® AVX およびインテル® AVX2
  - ✓ 256bit 長のレジスタ (ymm)を利用した SSE の拡張命令セット
- インテル® Advanced Vector Extensions 512
  - ✓ インテル® AVX-512
  - ✓ 512bit 長のレジスタ (zmm)を利用して AVX の2倍、
    SSE の4倍の要素を処理できるように拡張された命令セット

プロセッサー固有の最適化オプション

<target> 例
CASCADELAKE
COFFEELAKE
COOPERLAKE
HASWELL
SKYLAKE
SKYLAKE-AVX512

COMMON-AVX512
CORE-AVX512
CORE-AVX2
…

- SIMD 操作を含む特定のプロセッサーに対応したコードの生成を指示します
- -x<target>
  - ✓ <Target> で指定した命令セットをサポートするインテル® プロセッサー向けの専用コードを生成します
  - ✓ 実行可能ファイルは非インテル® プロセッサーや、指定した<Target> の命令をサポートしていないインテル® プロセッサーで動作しません

```
> icc -O3 -xcore-avx2 ./myapp.c -o ./myapp.out
```

- -xhost
  - ✓ ローカルマシンをターゲットにした専用コードを生成します

```
> icc -O3 -xhost ./myapp.c -o ./myapp.out
```

プロセッサー固有の最適化オプション 続き

- -ax<target>,<target>…
  - ✓ <target> で指定されたプロセッサー向けの専用コードと x86 プロセッサー向けの汎用コードを生成します
  - ✓ 複数の <target> を指定できます
  - ✓ 専用コードを実行できないシステムでは SSE2 の汎用コードを実行します

```
> icc -O3 -axcore-avx512,core-avx2 ./myapp.c -o ./myapp.out
```

- x86 プロセッサー向けの汎用オプション -march を指定できます

```
> icc -O3 -march=native ./myapp.c -o ./myapp.out
```

プロセッサー固有の最適化オプション 続き

- デフォルトは SSE および SSE2 を生成します (-O2 以上)
  - ✓ -x もしくは -ax を利用して、より上位の命令セットを生成するように指示します
  - ✓ -xicelake-server の指定をお試しください
- -x, -ax, -march はいずれかのみ指定できます
  - ✓ 複数指定した場合、コマンドの最後に記述されたものが適用されます

プロセッサー固有の最適化オプション 続き

- -qopt-zmm-usage=[low] もしくは [high]
  - ✓ zmm レジスターの使用を指示します
    low では明示的に指定のない限り xmm, ymm レジスタによって実装されます
  - ✓ -x, -ax の指定により変化しますが、多くの場合 low が設定されます
  - ✓ 積極的に zmm レジスターの使用したコード生成を指示するには high を指定します
  - ✓ 例: -qopt-zmm-usage=high, -qopt-zmm-usage=low の比較

```
vmovupd   b(%r11,%rbx,8), %zmm0
vmovupd   64+b(%r11,%rbx,8), %zmm1
vfmadd213pd c(%rax,%rbx,8), %zmm2, %zmm0
vfmadd213pd 64+c(%rax,%rbx,8), %zmm2, %zmm1
vmovupd   %zmm0, c(%rax,%rbx,8)
vmovupd   %zmm1, 64+c(%rax,%rbx,8)
```

```
vmovupd   b(%rsi,%r8,8), %ymm1
vmovupd   32+b(%rsi,%r8,8), %ymm2
vmovupd   64+b(%rsi,%r8,8), %ymm3
vmovupd   96+b(%rsi,%r8,8), %ymm4
vfmadd213pd c(%rax,%r8,8), %ymm0, %ymm1
vfmadd213pd 32+c(%rax,%r8,8), %ymm0, %ymm2
vfmadd213pd 64+c(%rax,%r8,8), %ymm0, %ymm3
vfmadd213pd 96+c(%rax,%r8,8), %ymm0, %ymm4
vmovupd   %ymm1, c(%rax,%r8,8)
vmovupd   %ymm2, 32+c(%rax,%r8,8)
vmovupd   %ymm3, 64+c(%rax,%r8,8)
vmovupd   %ymm4, 96+c(%rax,%r8,8)
```

自動並列化

- 安全にスレッド並列で実行できる構造のループ処理を検出して OpenMP* によるマルチスレッド化したコードを生成します
- -parallel
  - ✓ 自動並列化を有効にします (デフォルトは無効)
  - ✓ デフォルトでは実行時のスレッド数は論理コア数と同数となります
  - ✓ スレッド数の調整は
    環境変数: OMP_NUM_THREADS=N
    またはコンパイラーオプション: -par-num-threads=N
    を使用できます

    コンパイル時に par-num-threads を記述した場合、実行時に OMP_NUM_THREADS による指定を上書きします

自動並列化による仕事量の分割

```
for (int i = 0; i < count; i++)
{
  /* code */
}
```

- コンパイラーはループ処理の仕事量 (0, 1, 2, … count-1) をスレッドへ分配できるように分割します
  - ✓ 分割された個々の仕事をタスクとして各スレッドが計算します
- 割り当てスケジュールと各タスクの仕事量は -par-schedule-<keyword> オプションで指示できます
  - ✓ デフォルトは -par-schedule-static-balanced
    仕事量を均等に分割してラウンドロビン方式でスレッド番号順に割り振られます
  - ✓ 例えば - par-schedule-dynamic=100 を指定すると、
    0~99, 100~199 で分割されたタスクを生成
    プログラムの実行中に、各スレッドはタスクが完了次第、
    次のタスクが割り当てられて計算します

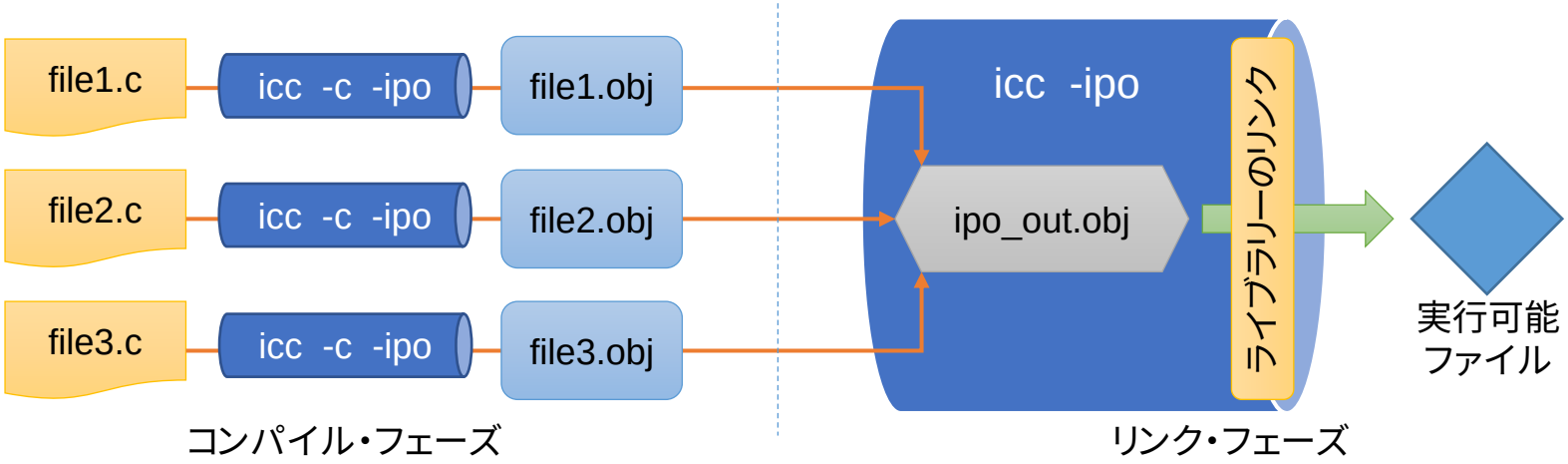
プロシージャー間の最適化
IPO - Interprocedural Optimization

- コンパイラーがアプリケーションを構成する
  個々のソースファイルを分析します
  - ✓ ソースファイル間で利用されるグローバル変数の操作や
    関数/サブルーチンの呼び出し関係を識別できるようになり
    より多くの最適化が適用されやすくなります
    - 関数 / サブルーチンのインライン展開
    - ループ展開や定数伝播
    - 不要と判定された処理や変数、条件分岐の削除 など…
- -ipo
  - ✓ IPO を有効にします (デフォルトでは無効)

```
> icc -ipo ./myapp.c -o ./myapp.out
```

IPO によるコンパイルとリンク

- コンパイル・フェーズ:
  個々のソースファイルから中間表現を含むオブジェクトを生成
- リンク・フェーズ:
  オブジェクトファイルを集約し、追加の最適化を含むコンパイルを実施



IPO の例：関数のインライン展開

- コンパイラーはデフォルトで単一ファイル内の関係を識別します
- シンプルな例としてループ内で異なるファイルに定義された関数の呼び出しはループやベクトル化に関する最適化が抑制されやすいです
  - ✓ IPO によって関数内の処理がインライン展開されベクトル化して問題ないと評価されると、ループ処理は自動ベクトル化機能によってベクトル化されます

```
for (int i = 1; i < nx; i++)
{
  x = x0 + i * h;
  sumx = sumx + func(x, y, xp, yp);
}
```

異なるソースファイルに記述された func 関数の実装

```
float func(float x, float y, float xp, float yp)
{
  float denom;

  denom = (x - xp) * (x - xp) + (y - yp) * (y - yp);
  denom = 1.0f / sqrtf(denom);

  return denom;
}
```

プロファイルに基づく最適化 (PGO)

- 実行時のプロファイル情報を生成して、
  その結果をもとに再度最適化します
- -prof-gen と -prof-use
  - ✓ プロファイルの生成とプロファイルの利用
- 以下などの最適化により、CPU の命令キャッシング、
  メモリー・ページング、分岐予測を支援します
  - ✓ 適切なレジスター割り当て
  - ✓ 入れ子の If 文や、Switch 文の分岐最適化
  - ✓ 関数のインライン展開、自動ベクトル化、自動並列化による性能評価
  - ✓ 基本ブロックや関数の並び替え　など…

プロファイルに基づく最適化の適用

1. プロファイルを取得するスクリプトを埋め込んだ実行ファイルを生成

```
> icc -O3 -xhost -prof-gen ./myapp.c -o ./myapp.out
```

2. プログラムを実行

```
> ./myapp.out data.txt
```

- ✓ プロファイル情報が *.dyn ファイルとして保存されます
- ✓ 条件により実行されるコードパスが異なる場合は、それぞれのパスを通るように複数回実施します (実行の度に *.dyn ファイルを生成します)

■ プロファイル情報 (*.dyn)をもとにフィードバック・コンパイル

```
> icc -O3 -xhost -ipo -prof-use ./myapp.c -o ./myapp_pgo.out
```

- ✓ フォードバック・コンパイル時に IPO も付与

必須ではありませんが相性の良い最適化オプションです

PGO の例: Switch 文の最適化

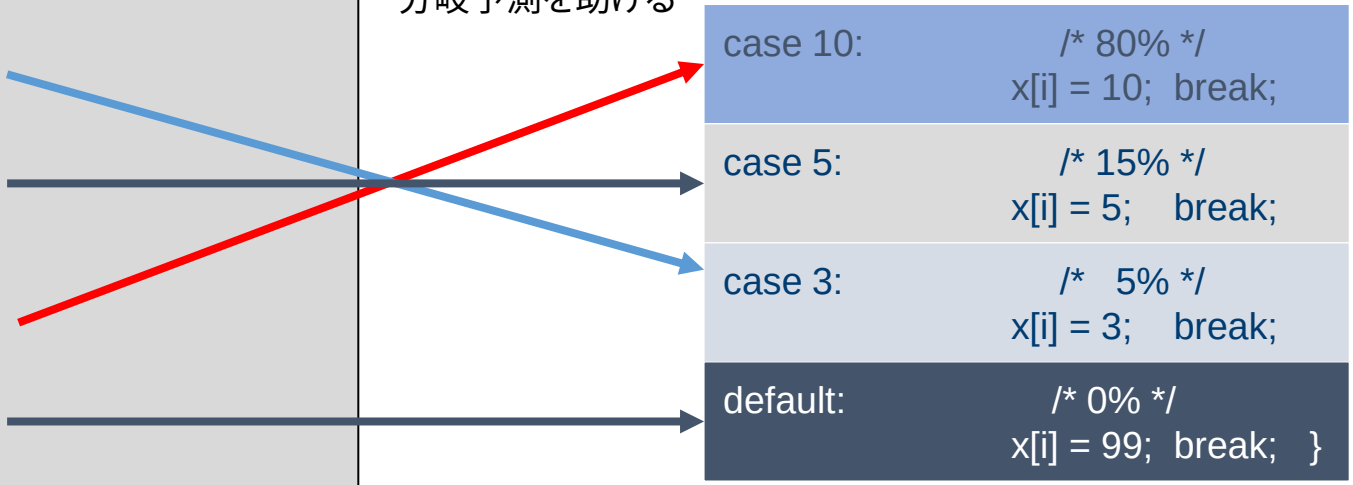
```
for (int i = 0; i < NUM_BLOCKS; i++)
{
  switch (check(i) )
  {
  case 3:
    x[i] = 3;
    break;

  case 5:
    x[i] = 5;
    break;

  case 10:
    x[i] = 10;
    break;

  default:
    break;
  }
}
```

各 case の実行頻度についてプロファイル情報を収集
頻度の高い順番に並べ替えることで、プロセッサーの
分岐予測を助ける



(疑似コード)

浮動小数点数値演算の再現性

- 浮動小数点を含む演算は近似値を求めることとなり
  演算結果には誤差を含みます
  - ✓ これはシステムや OS 、コンパイラーなどによって変化する可能性があります
- "演算結果の再現性"と"計算時間の短縮"はトレードオフの関係にあり
  最適化を適用することで異なる演算結果を出力することがあります
  - ✓ インテル® コンパイラーの浮動小数点演算における結果の一貫性 | iSUS
- 一貫した同じ結果を求める場合、インテル® コンパイラーでは
  -fp-model オプションにより制御します

-fp-model

type	説明
fast[=1|2]	計算結果に影響がある最適化を許可します fast (fast=1) がインテル® コンパイラーにおけるデフォルトです fast=2 が指定されると、さらにいくつかの追加の近似が許可されます
precise	計算結果に影響しない最適化のみ有効にします
consistent	異なるプロセッサーや最適化レベルでも 一貫した再現性のある結果が得られるようにコードを生成します
strict	precise を指定した場合の効果に加えて、FMA (Fused Multiply-Add) 命令の使用を 抑止し、FPU 環境へのアクセスを許可します また厳密な浮動小数点例外セマンティクス (except の指定) を有効にします

リダクション処理による演算結果が変わる可能性

- 部分和 sum1, sum2 … sumN の計算に分割し、後から足し合わせるリダクションの実装は演算結果が変わる可能性があります

```
for ( i = 0 ;  i < N ;  i++ )
{
    sum  =  sum  +  …
}
```

```
sum3 = 0.0 ;
for ( i = begin ;  i < end ;  i++ )
{
    sum3  =  sum3  +  …
}
```

sum = sum1 + sum2 + sum3 + ⋯ sumx;

分割数や計算順序などは
最適化や実行時の条件により異なる場合がある

コンパイラーオプションまとめ

- 複数のコンパイラーオプションを組み合わせることで
  より性能を向上しやすくなります
- -O2 もしくは -O3 をお試しください
- 対象のプロセッサーを指定するとより最適化できます
  - ✓ -xicelake-server をお試しください
- 自動並列化はマルチコアを利用できる可能性があります
  - ✓ -parallel で簡単に有効にできます
- IPO と PGO の組み合わせをお試しください
- 演算結果の一貫性は -fp-model を指示ください
  - ✓ ただし性能とトレードオフとなります

最適化レポートオプション

- コンパイラーの最適化および変更をテキストファイルで出力します
- -qopt-report=n
  - ✓ 最適化レポートを生成します
  - ✓ レポートは .optrpt 拡張子を持つファイルに出力されます
  - ✓ n には、0 ~ 5 のレベルを指定します (デフォルトは 2 です)

ループの最適化レポート例

- 最適化レポートはソースファイル毎に生成されます
  - ✓ IPO を適用した場合、ソースファイル毎に生成しません
    ipo_out.optrpt の単一ファイルが生成されます
- レポートにはループ処理にかかる最適化や自動ベクトル化、自動並列化、OpenMP* の適用など、多くのコンパイラーが実施した最適化の状況をソース行とあわせて出力します

```
> icc -O3 -xhost -qopt-report ./matmul.c -o myapp.out
```

```
for(int i=0; i < m; i++)
  for (int j = 0; j < p; j++)
    for (int k = 0; k < n; k++)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

```
LOOP BEGIN at ./matmul.c(23,3)
   remark #25444: Loopnest Interchanged: ( 1 2 3 ) --> ( 1 3 2 )
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at ./matmul.c(23,3)
      remark #15542: loop was not vectorized: inner loop was already vectorized

      LOOP BEGIN at ./matmul.c(23,3)
         remark #15542: loop was not vectorized: inner loop was already vectorized

         LOOP BEGIN at ./matmul.c(23,3)
            remark #25442: blocked by 128   (pre-vector)
            remark #25440: unrolled and jammed by 4   (pre-vector)
            remark #15542: loop was not vectorized: inner loop was already vectorized

            LOOP BEGIN at ./matmul.c(25,5)
               remark #25442: blocked by 128   (pre-vector)
               remark #25440: unrolled and jammed by 4   (pre-vector)
               remark #15542: loop was not vectorized: inner loop was already vectorized

               LOOP BEGIN at ./matmul.c(24,3)
                  remark #25442: blocked by 128   (pre-vector)
                  remark #15301: PERMUTED LOOP WAS VECTORIZED
                  remark #26013: Compiler has chosen to target XMM/YMM vector. Try using -qopt-zmm-usage=high to override
                  remark #25456: Number of Array Refs Scalar Replaced In Loop: 24
               LOOP END
            LOOP END
…
```

-qopt-report-phase=name1,name2,…

- 最適化フェーズ name1、name2 固有の最適化レポートを生成しま
- name 引数には、以下のキーワードを指定できます
  - ✓ all　すべてのフェーズのすべての最適化レポート (デフォルト)
  - ✓ loop　ループの入れ子とメモリーの最適化
  - ✓ vec　自動ベクトル化と明示的なベクトル・プログラミング
  - ✓ par　自動並列化
  - ✓ openmp　OpenMP* によるスレッド化
  - ✓ cg　コード生成
  - ✓ ipo　インライン展開を含むプロシージャー間の最適化
  - ✓ pgo　プロファイルに基づく最適化

```
> icc -qopt-report=3 -qopt-report-phase=vec,par sample.c -o myapp.out
```

```
LOOP BEGIN at C:¥work¥fortran¥pi¥pi_calc.f90(23,3) inlined into
C:¥work¥fortran¥pi¥pi_calc.f90(42,8)
   remark #17109: LOOP WAS AUTO-PARALLELIZED
   remark #17101: parallel loop shared={ } private={ } firstprivate={ X I } lastprivate={ }
firstlastprivate={ } reduction={ SUM }
   remark #15305: vectorization support: vector length 8
   remark #15399: vectorization support: unroll factor set to 4
   remark #15309: vectorization support: normalized vectorization overhead 0.138
   remark #15355: vectorization support: SUM is double type reduction
[ C:¥work¥fortran¥pi¥pi_calc.f90(21,3) ]
   remark #15300: LOOP WAS VECTORIZED
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 41
   remark #15477: vector cost: 6.120
   remark #15478: estimated potential speedup: 6.690
   remark #15486: divides: 1
   remark #15487: type converts: 1
   remark #15488: --- end vector cost summary ---
   remark #25015: Estimate of max trip count of loop=93750
```

自動ベクトル化されない
もしくは性能を低下させる要因

- 最適化レポートはベクトル化の適用について
  より詳細な情報を入手するための手段として活用いただけます
- ループ中の計算処理によって適用されない場合があります
  - ✓ 計算順序に依存関係がある
  - ✓ 複数の終了条件を持つ
  - ✓ ライブラリー関数やユーザーが作成した関数、サブルーチンを呼び出している
    数学ライブラリーを除く
- またメモリーアクセスによってベクトル化の性能が大きく変化します
  - ✓ ユニットストライドアクセス
  - ✓ メモリーアライメント

ループ処理をベクトル化しない依存関係の例

- 条件分岐による終了条件を含む
- 配列データへのアクセスに依存関係を持つ
  - ✓ X = 1 とすると I = 2 となり A[2] の値を計算するためにA[1] の計算結果を要求しています
  - ✓ X = 2 以上と仮定された場合はベクトル化される可能性があります
- 関数やサブルーチンの呼び出しを含む
  - ✓ インライン展開等のコンパイラーの最適化によりベクトル化できる場合があります

```
do I=1, N
  C(I) = A(I) + B(I)
  if (A(I) < 0.0) exit
enddo
```

```
do I=X + 1, N
  A(I) += A(I) + A(I-X)
enddo
```

```
  do I=1, N
    A(I) = myfunc(B(I))
  enddo
```

連続したメモリーアクセスと間接メモリーアクセス

- 連続しないメモリーアクセスはベクトル化による性能を低下させる要因となります
  - ✓ ループ内にて連続したメモリーアクセスを含む場合最適化レポートでは unit stride と表現されます

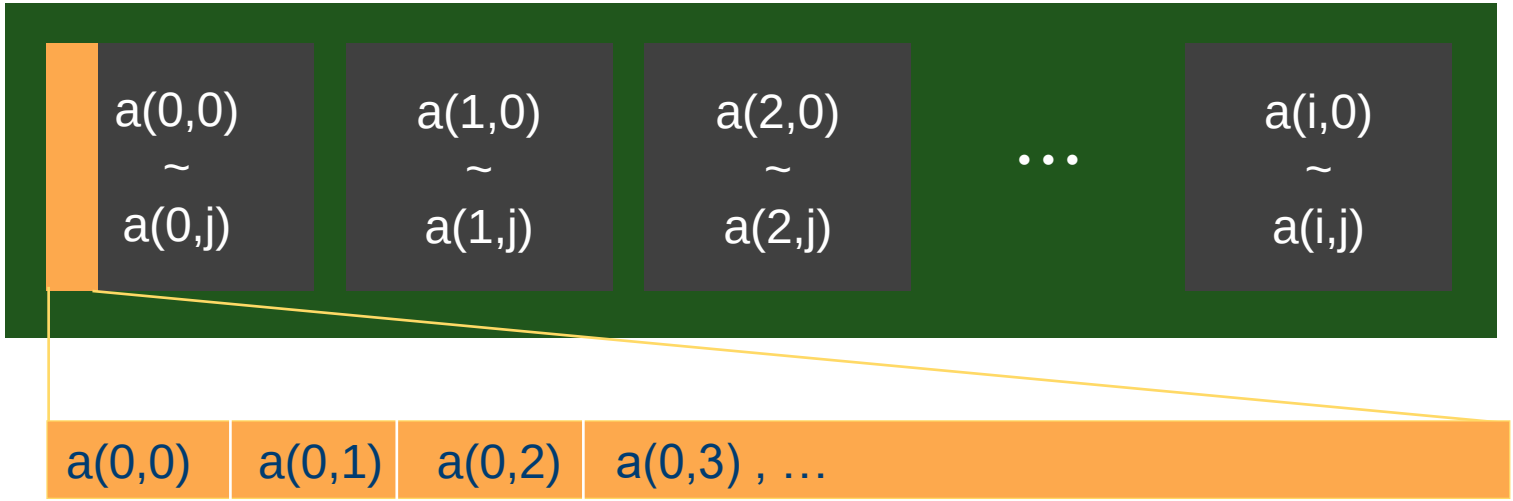
```
remark #15448: unmasked aligned unit stride loads: 2
remark #15449: unmasked aligned unit stride stores: 1
```

- ループ内のメモリーアクセスが non-unit stride の場合、SIMD 操作する要素のロード/ストアにかかるコストが増加します
- コンパイラーはスカラー処理と比較して効率が悪いと判断したループをベクトル化しません
  - ✓ インテル® AVX-512 などの新しい SIMD 操作ではマスク処理やzmmレジスタの活用によりベクトル化できる場合があります

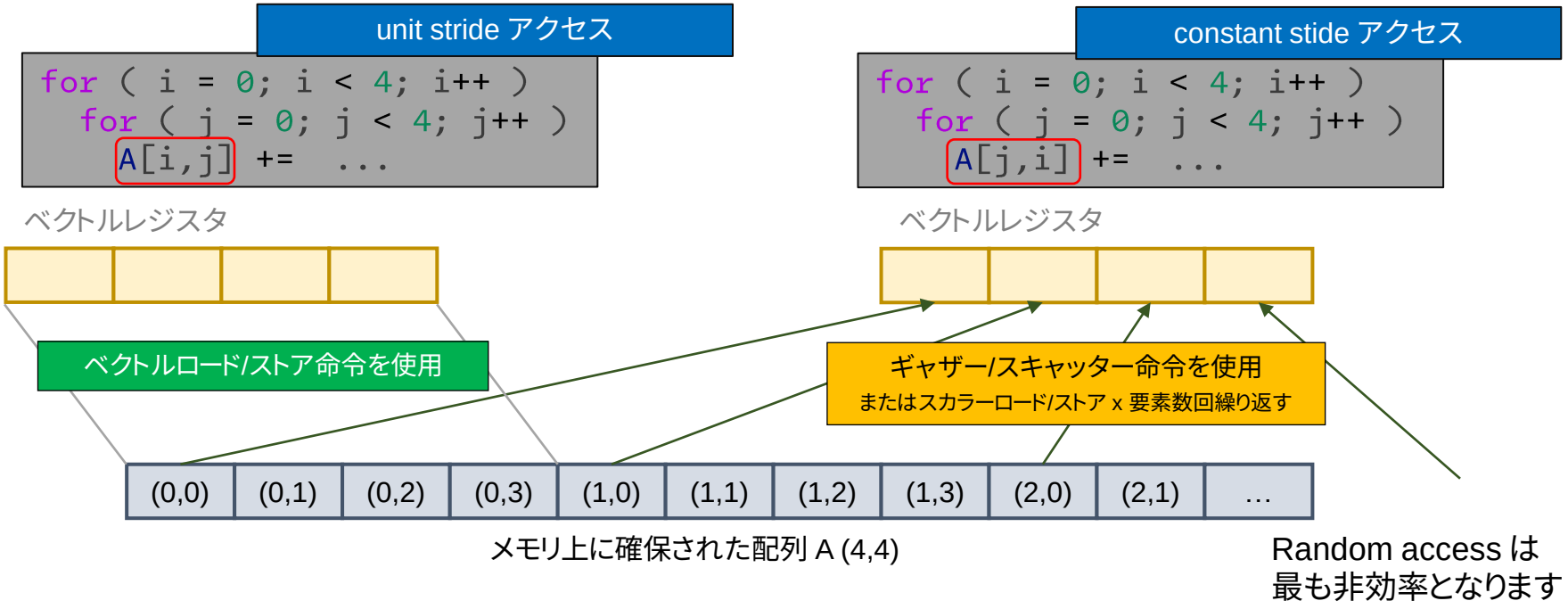
配列のメモリーアクセス

- ベクトル化された処理はデータのアクセス順序に影響を受けます
  - ✓ 配列の値は行優先にメモリに配置される (Fortran は列優先で配置します)

```
float a[i][j];
```



ストライド メモリーアクセスのイメージ



Random access は
最も非効率となります

アラインメント (Alignment)

- データ・アライメントは特定のバイト境界上の
  メモリーにデータ・オブジェクトを生成します
- 基本的にコンパイラーによって各変数はアライメントされるように
  動作しますが、ユーザーが明示的にコンパイラーへ
  指示もしくはコードに記述することもできます
- アラインメントされているデータはロード/ストアの効率を高めます
  - ✓ コンパイラーはアライメントされたデータによる処理と
    アラインメントされてないデータを処理する複数のコードを生成する場合があります

```
remark #15388: vectorization support: reference a[i][j] has aligned access
remark #15388: vectorization support: reference b[i][j] has aligned access
remark #15388: vectorization support: reference c[i][j] has aligned access
```

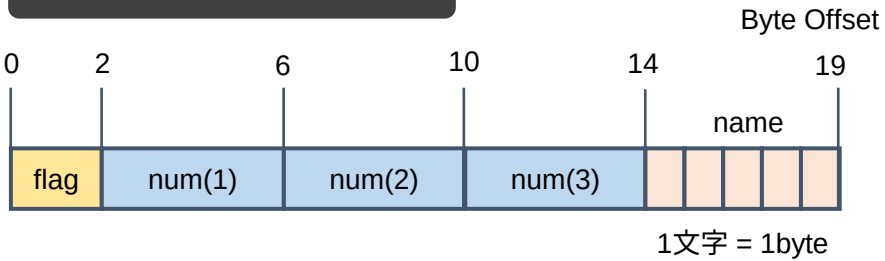
アライメント例

```
logical(2) flag
integer num(3)
character(5) name
common /ts/ flag, num(3), name
```

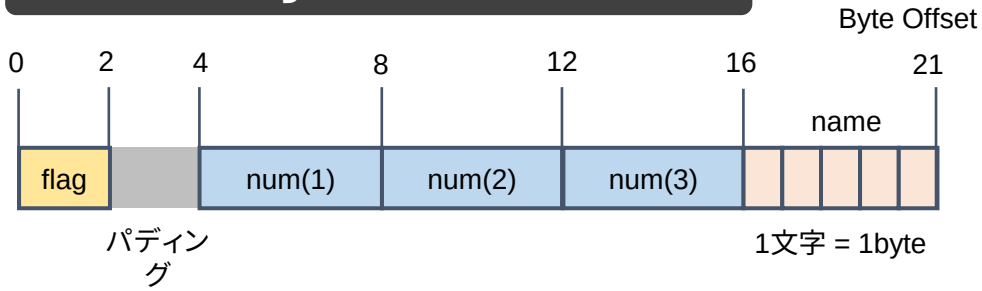
- 一部のデータ型は
  アライメントされません
  - ✓ 構造体やCOMMON文の
    アライメントは明示的もしくは
    コンパイラー・オプションを
    利用します
    もしくは大きなデータ型から
    小さなデータ型へ並べ替えます
  - ✓ ここでは 4byte 境界にデータを
    配置するためのパディングします

COMMON 文のアライメント例

```
> ifort common.f90
```




```
> ifort -align common common.f90
```



データをアライメントする

- コンパイラー・オプションによりデータのアライメントを指定できます

```
> ifort -align array64byte common.f90
```

- コード中にアラインメントするように記述します
  - ✓ いくつかの手法があります

```
__attribute__((aligned(64))) a[n]
```

C++17 だと aligned_alloc を利用できます

Fortran

```
double precision, allocatable :: a(:,:), b(:,:), c(:,:)
!!DIR$ attributes align:64 :: a,b,c
```

```
subroutine calc(size, c, a, b)
  implicit none
  integer :: size
  double precision :: a(size,size), b(size,size), c(size,size)
  !DIR$ ASSUME_ALIGNED a:64, b:64, c:64
```

アプリケーションの解析

プロファイラーの利用

- CPU、メモリーなどのシステムリソースの利用状況と
  プログラムの動作を紐付けして表示できます
- インテル® VTune™ プロファイラーの解析により得られる情報を
  参考にしつつプログラムの動作状況を確認します
- プログラムにデバッグ情報を追加することを推奨します
  - ✓ 最適化オプションはあわせて指示します

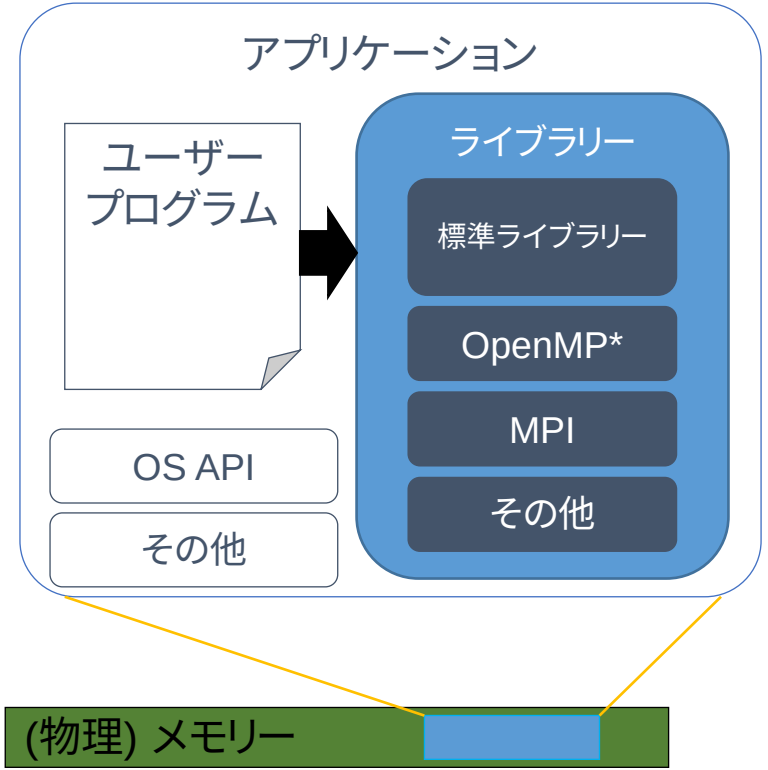
```
> ifort -g -O2 sample.f90
```

パフォーマンス・データの収集

- プログラムの動作状況を確認します
  - ✓ システムリソースの使用率
    CPU 使用率、メモリー帯域幅
- 最適化に影響するプログラムの動作を把握する
  - ✓ プロセス/スレッドの発生
  - ✓ 関数およびループ単位の分析
  - ✓ 入出力 (ネットワークまたはファイル) や待機 (排他処理、同期) の検出

最適化の候補

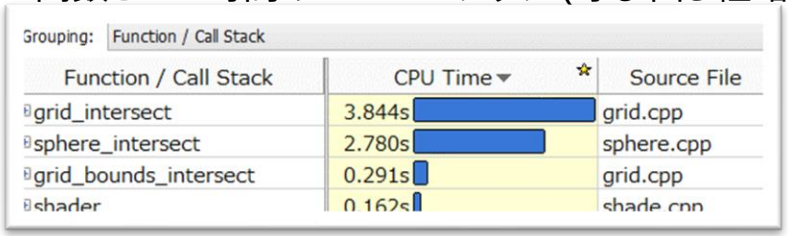
- ユーザープログラム中の処理
  - ✓ ほとんどはループ処理
- 一方でユーザープログラム以外の動作が影響している可能性を考慮する必要があるかもしれません
  - ✓ 主にライブラリーの利用
    - 標準関数を提供するコンパイラーのライブラリー
    - OpenMP* や MPI
    - サードパーティ製のライブラリー



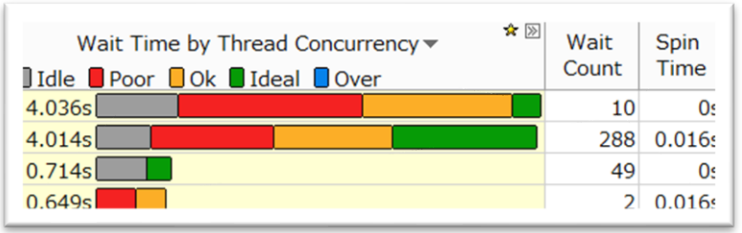
インテル® VTune™ プロファイラー

- インテル® プロセッサー上で実行されるプログラムを解析するためのプロファイラー

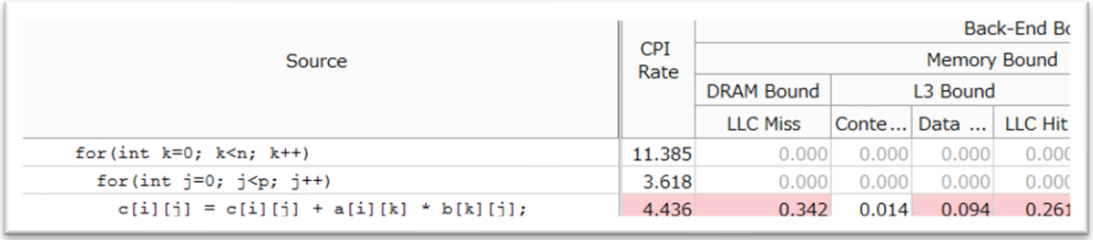
✓ 実行時、どこに時間を費やしているか
関数ごとの時間やコールスタック (呼び出し経路)



✓ 実行時に長い待機があるか
待機した時間と理由、スレッドの並行動作状況



✓ 効率の悪い命令を実行をしていないか
キャッシュミスなどのハードウェア・イベントとソースコードの対応



解析コマンド

- vtune コマンドで解析を実行する

vtune -collect <解析タイプ> [追加オプション] -- <解析対象> <引数>

```
> vtune -collect hotspots -- a.out input.txt
```

✓ インテル® MPI ライブラリーを利用した MPI プログラムの解析例

```
> mpirun -np 4 -gtool "vtune -collect hotspots -result-dir my_result:0" a.out
```

```
> mpiexec.hydra -n 16 -genv I_MPI_GTOOL=
"vtune -collect hotspots -result-dir my_result:0" a.out
```

MPI プロセスに対する解析は1、2 プロセスあたりを推奨します

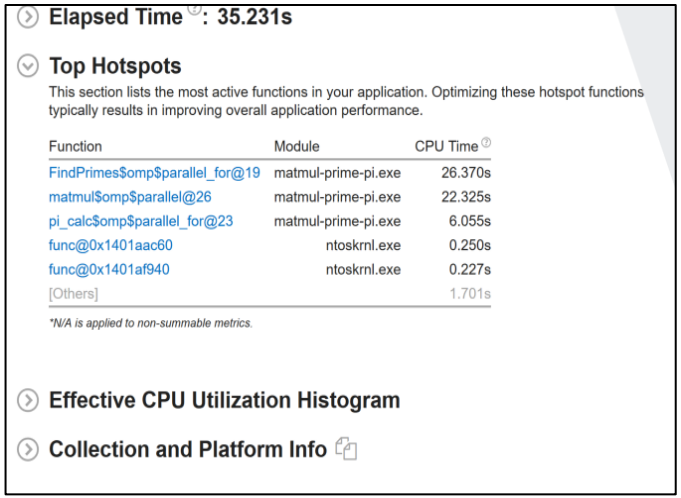
主な解析タイプ

解析タイプオプション名	表示情報
hotspots	実行したプログラムが使用した CPU 時間を主に示します 低オーバーヘッドで動作します
threading	マルチスレッド化されたプログラムの効率を主に示します OpenMP* 並列領域を識別して潜在的な性能問題を評価します ※ 旧バージョンの Concurrency 解析タイプと Locks and Waits 解析タイプに相当します
memory-access	キャッシュ効率およびメモリー帯域幅利用率および リモートメモリーアクセスを主に示します
uarch-exploration	マイクロアーキテクチャー全般の性能情報を主に示します
hpc-performance	マルチコアを含む CPU 使用率、ベクトル演算/FPU 使用率、メモリー帯域幅利用率の 3 つの観点に対する性能情報を主に示します -knob analyze-openmp=true オプションで OpenMP* 領域を識別します

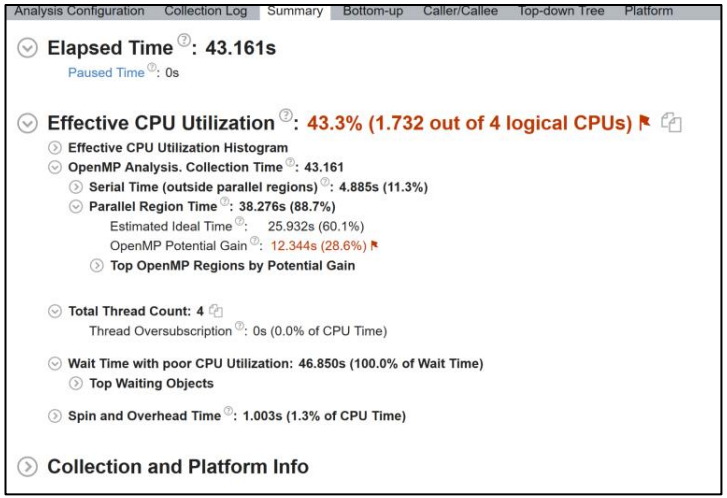
表示される情報

- 解析タイプにより収集される情報と表示内容が異なります
  - ✓ 解析タイプごとに確認すべき情報をある程度切り分ける
    取得する情報が多いほどオーバーヘッドが高くなる

Hotspots 解析

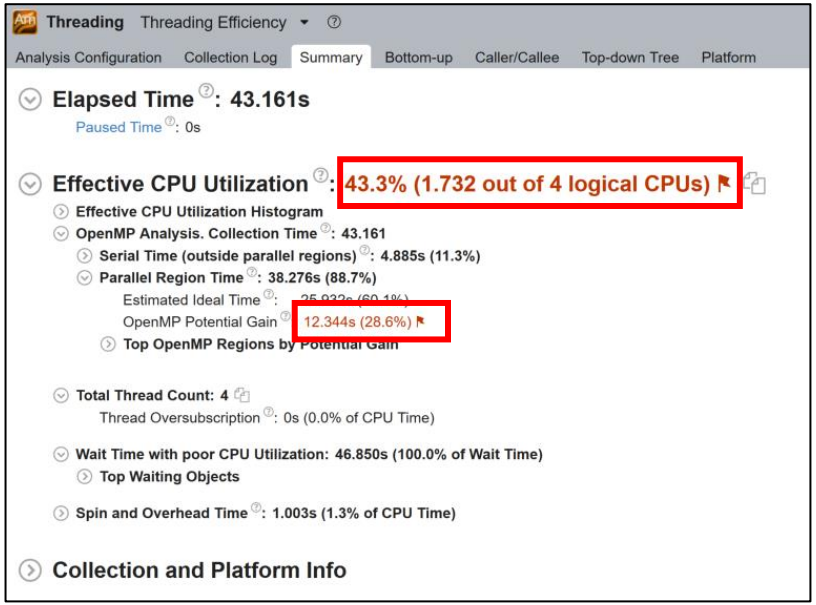


Threading 解析



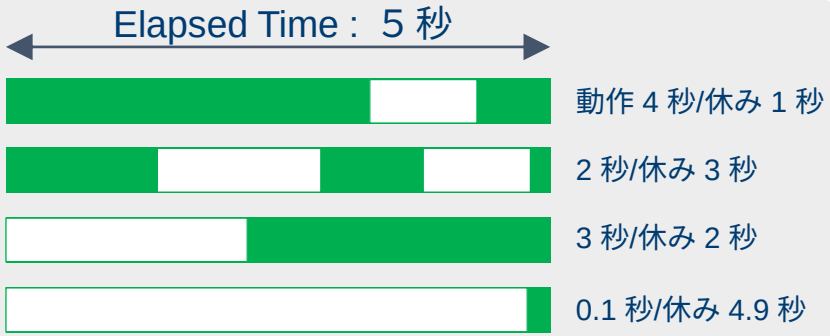
パフォーマンス向上の可能性を表示

- 潜在的な性能問題を抱えていそうな項目を赤く表示します
  - ✓ プロファイラーの評価式に基づいて表示されます

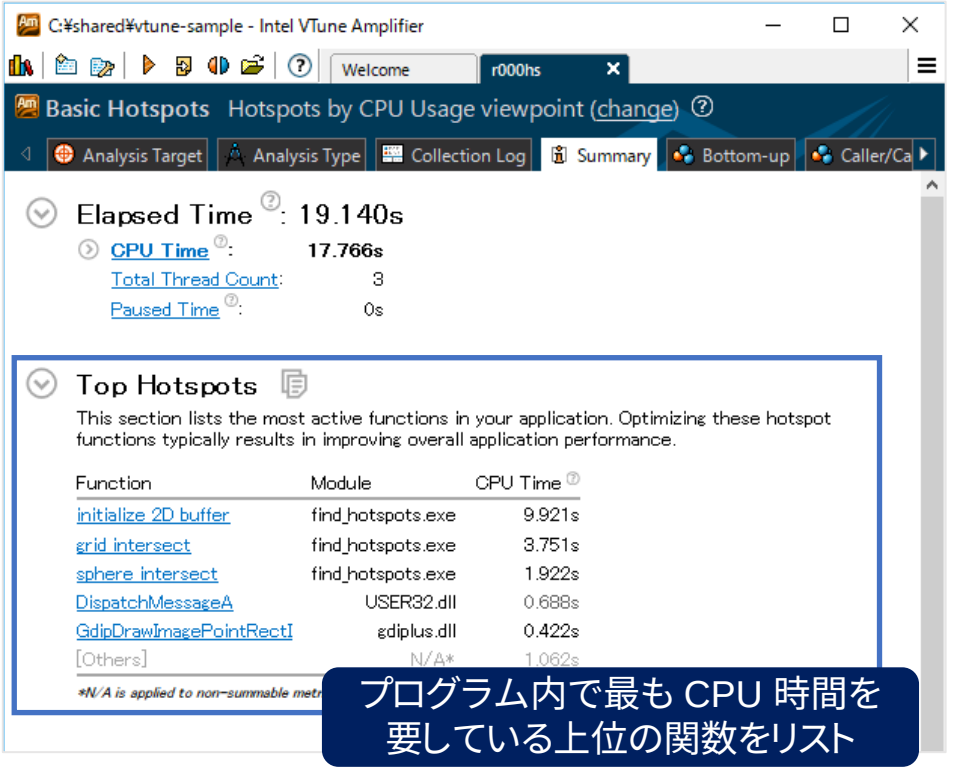


Hotspots 解析結果例

- Summary (概要) タブを参照
  - ✓ Elapsed Time: 経過時間 (実時間)
  - ✓ CPU Time: CPU の合計稼働時間



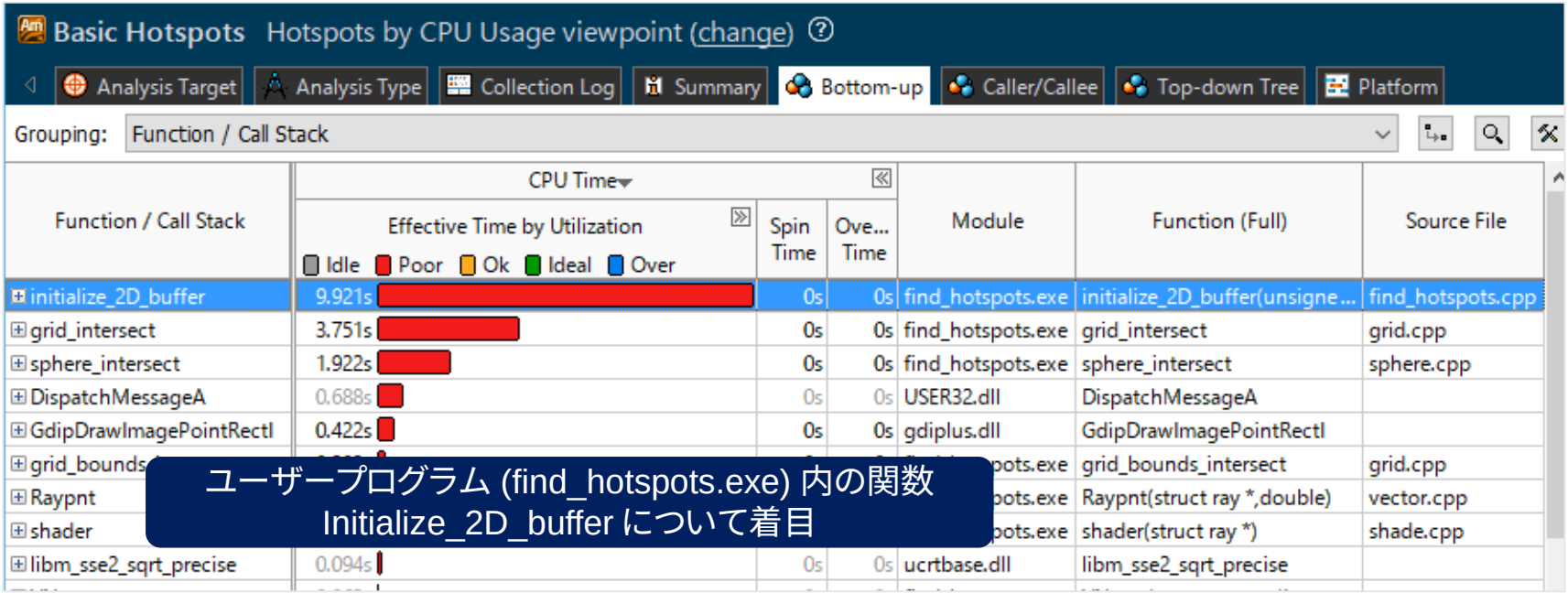
4コア CPU での一例



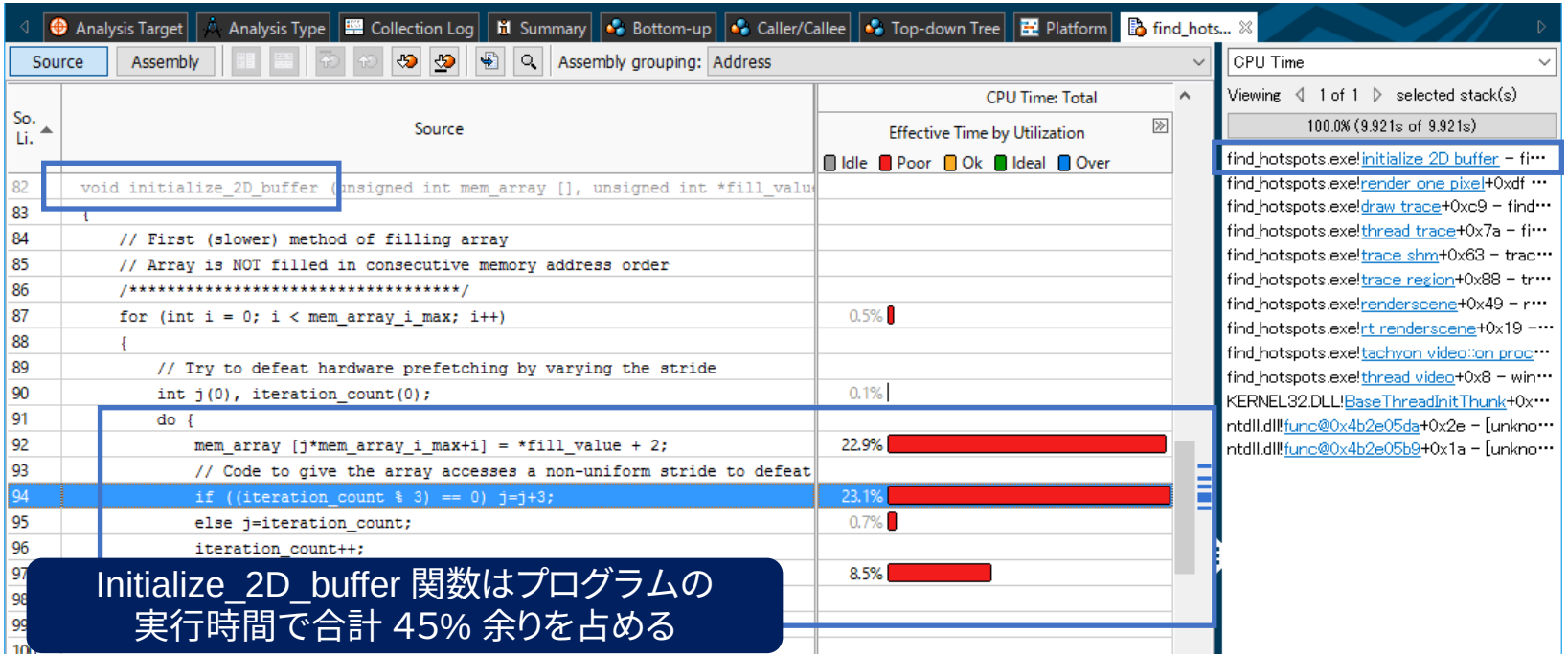
プログラム内で最も CPU 時間を要している上位の関数をリスト

Hotspots 解析結果例

Bottom-up (関数/項目ごとの表示) タブ

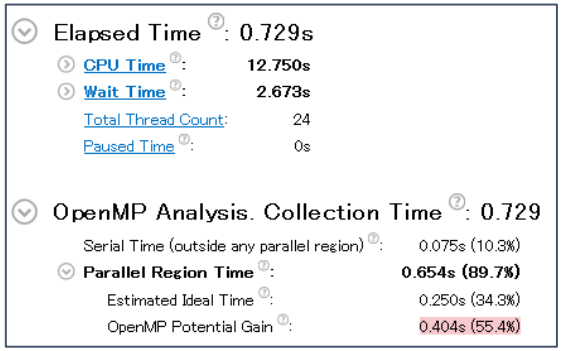


Hotspots 解析結果例

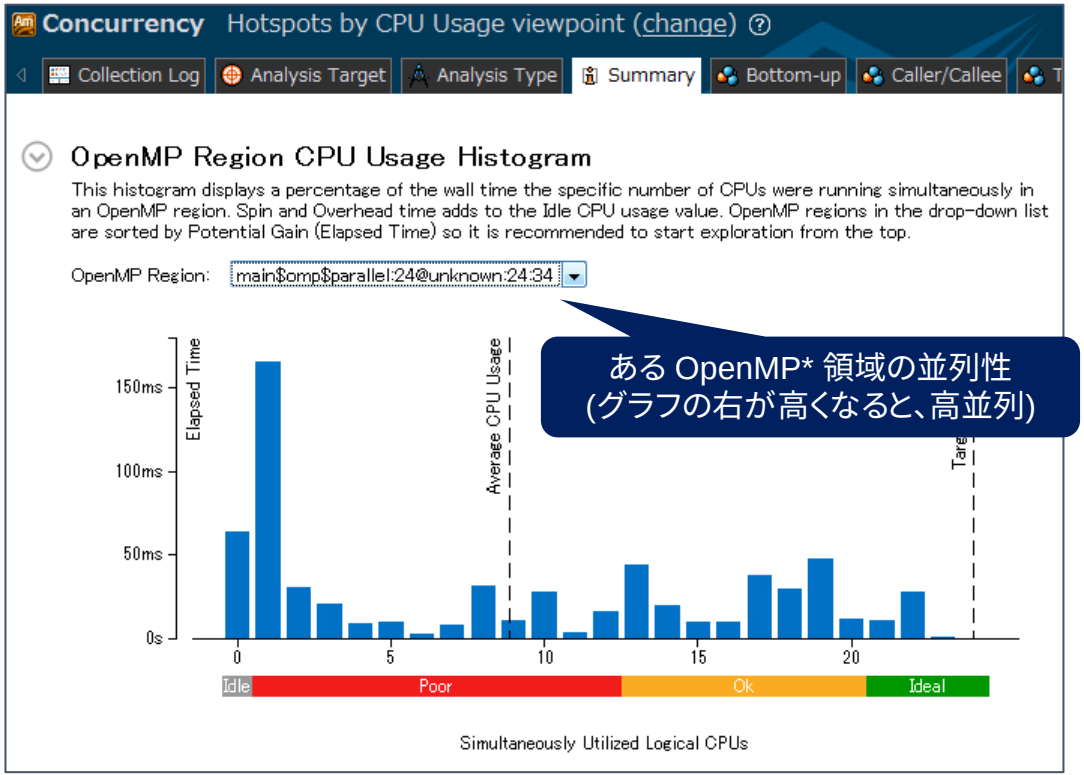


Threading 解析

スレッドが効果的に使用されているか
確認するための解析タイプ
特に OpenMP* の利用について注目します



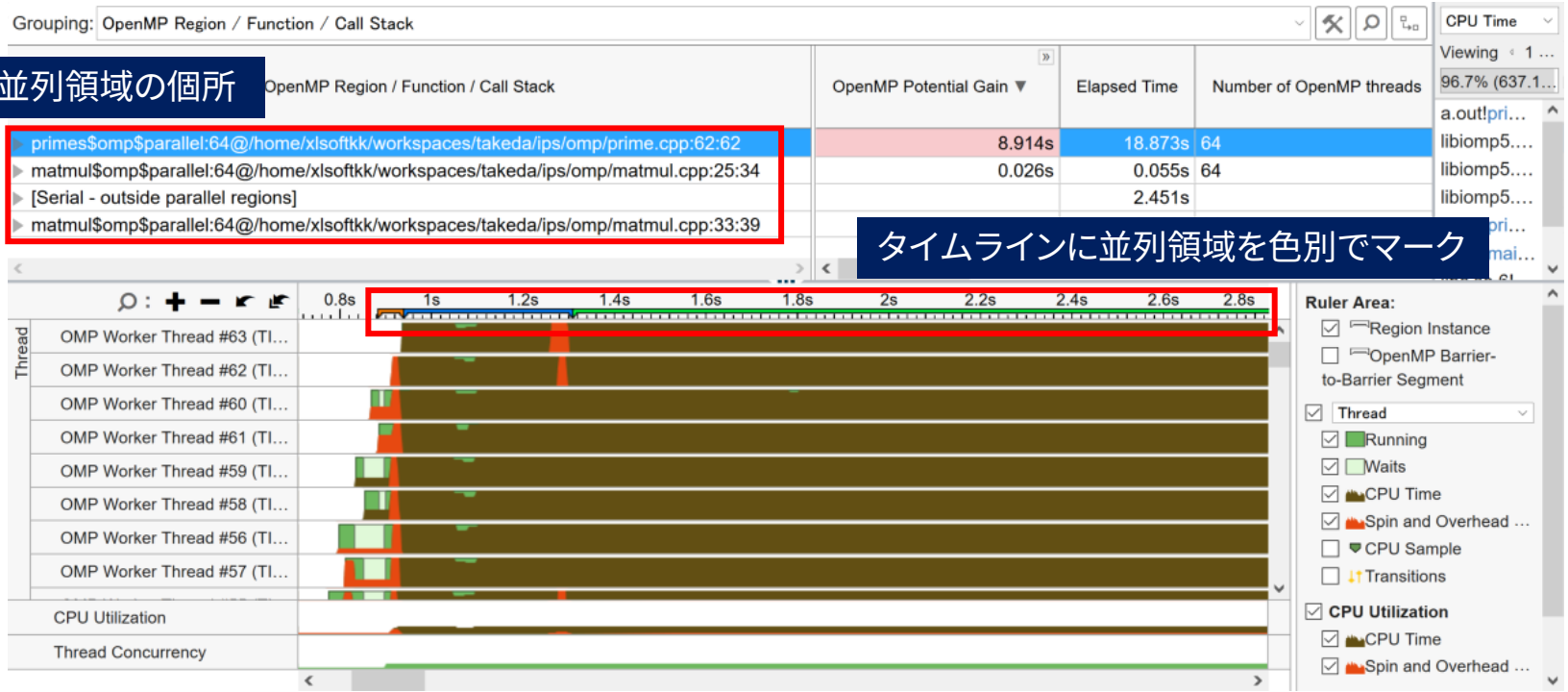
シリアル/OpenMP* 領域での実行時間
(Elapsed Time) の割合と、OpenMP* 領域で
効率化できる割合 (Potential Gain)



ある OpenMP* 領域の並列性
(グラフの右が高くなると、高並列)

OpenMP* 領域の解析

並列化の効率を評価



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示します
  - ✓ 性能に影響する項目をリストします

    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など
  - ✓ 実行ファイルに -parallel-source-info=2 オプションを付けると 行情報や関数/ルーチン名、ファイルパスやファイル名の情報を追加できます

```
> ifort -g -O2 -parallel-source-info=2 source.f90
```

Grouping: OpenMP Region / Function / Call Stack

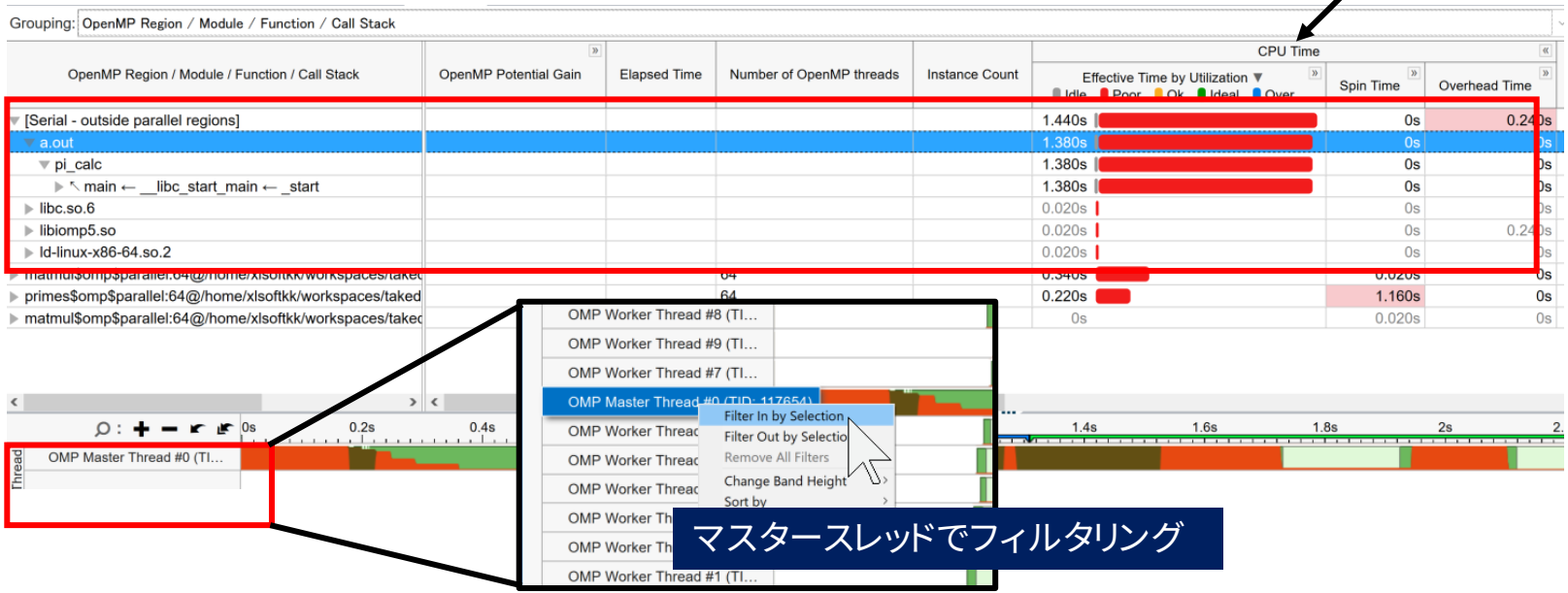
OpenMP Region / Function / Call Stack	OpenMP Potential Gain ▼						Elapsed Time	Number of OpenMP threads	Instance Count	Idl
	Imbalance	Lock Contention	Creation	Scheduling	Reduction	Atomics				
primesompparallel:64@/home/xlsoftkk/workspaces/takeda/ips/omp/prime.cpp:62:62	8.914s	0s	0s	0s	0s	0s	18.873s	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

OpenMP* 領域外の処理

[Serial - outside parallel regions] は OpenMP* 領域外で実行された処理を含みます
→展開すると、シリアル実行されたモジュールや関数を確認できます

CPU 時間でソート

カラムをクリック



マスタースレッドでフィルタリング

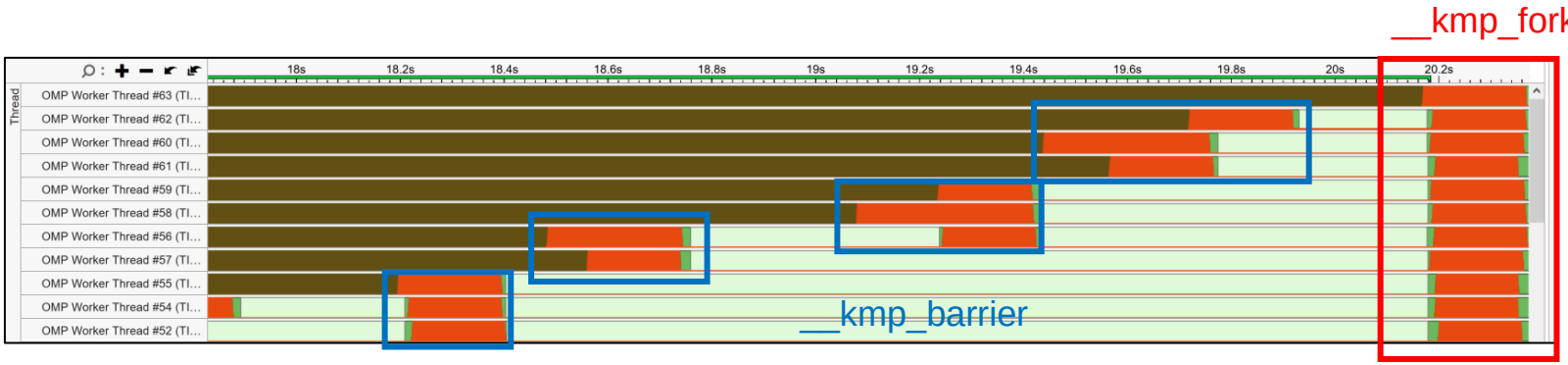
OpenMP* 利用時のオーバーヘッド

- インテル® コンパイラーが提供する OpenMP* ランタイムでは Spin Time に計上されるバリアセグメントの種類を確認することができます
  - ✓ 並列実行の終了後、次の入力まで待機している状態など
  - ✓ ユーザー指定のバリアや、暗黙的なバリアを含む

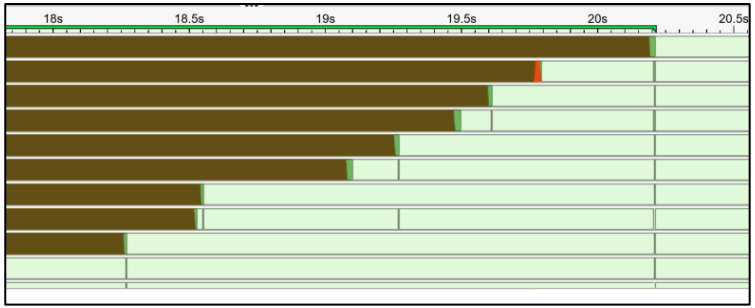
Grouping: Module / Function / Call Stack

Module / Function / Call Stack	CPU Time ▼			
	Effective Time by Utilization (Idle Poor Ok Ideal Over)	Spin Time	Overhead Time	Wait Time by Utilization (Idle Poor Ok Ideal Over)
a.out	661.440s	0s	0s	
libiomp5.so	0.020s	36.373s	0.361s	649.794s
__kmp_barrier	0s	21.758s	0.041s	548.617s
__kmp_fork_barrier	0s	14.395s	0s	101.164s
__kmp_get_global_thread_id_reg	0s	0s	0.180s	0.013s
__kmp_join_barrier	0s	0.160s	0s	
_INTERNAL_26_____src_z_Linux_util_cpp_d7ee2e5e	0s	0s	0.060s	
__kmp_fork_call	0s	0s	0.040s	
__kmp_join_call	0s	0.040s	0s	
__kmp_finish_implicit_task	0s	0s	0.020s	
__kmp_yield	0s	0.020s	0s	

例: バリアセグメントを含むタイムライン



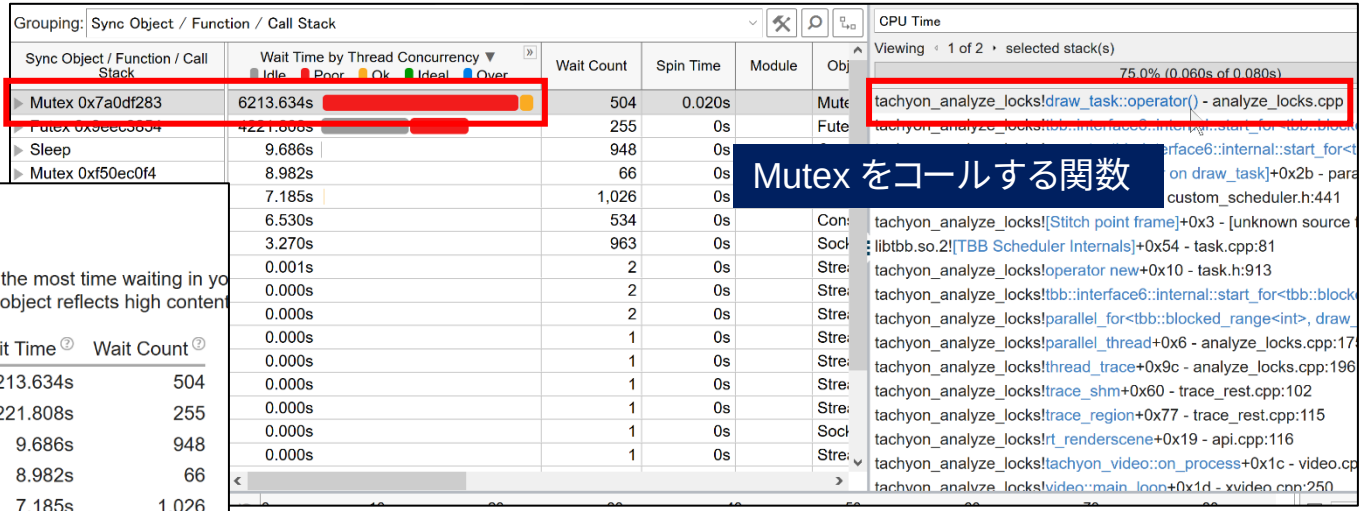
KMP_BLOCKTIME 環境変数を使用してスリープ状態になるまでの待機時間を調整することができます
デフォルトでは 200msec の待機が実行されます



参考：KMP_BLOCKTIME=0 にした状態
待機時間をなくしたため、Spin Time はなくなっている
実行性能が改善された状態ではないので注意

ロック・オブジェクトの検出

スレッド並列実行時のクリティカル処理による待機、
スレッドの同期待ちに要した時間を表示します



Mutex をコールする関数

Top Waiting Objects

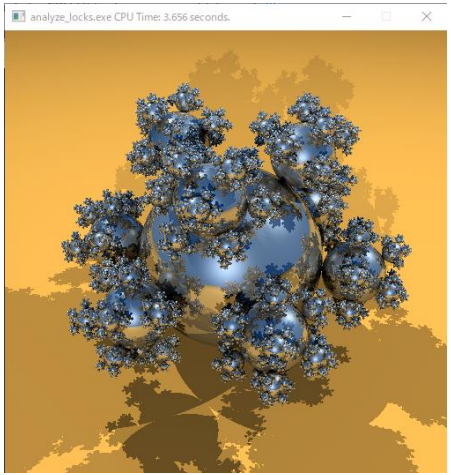
This section lists the objects that spent the most time waiting in yo
time associated with a synchronization object reflects high content

Sync Object	Wait Time	Wait Count
Mutex 0x7a0df283	6213.634s	504
Futex 0x9eec3854	4221.808s	255
Sleep	9.686s	948
Mutex 0xf50ec0f4	8.982s	66
Stream /proc/cpuinfo 0xcc250cd1	7.185s	1,026

時間を多く消費した待機オブジェクトを表示

解析例
スレッド間の同期にかかる時間を特定

- tachyon_analyze_locks サンプルプログラムを使用
- インテル® VTune™ プロファイラー を使用して
  サンプルプログラムに潜むボトルネックを調査します
- このサンプルでは、不必要な同期処理により
  実行性能が低下しています
  - ✓ 古いサンプルのため、現在は利用できません
    解析手順の参考としてご紹介します



ここでは Windows* 環境で実施しています

最適化の余地を調査する

- (Performance snapshot 解析を実行する)
- 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、スレッド化による複数コアの利用が可能か検討する
- スレッド化されている場合は、Timeline の内容に注目
  - ✓ CPU 利用率 (茶色のバー) が少ない場合、効果的なスレッド並列による処理が行われていないことを示しています
- Threading 解析からスレッド実行を制限する要因を確認します

Hotspots 解析を実行

20 秒程度の実行時間

Elapsed Time: 19.138s
- CPU Time: 7.563s
  - Effective Time: 5.658s
  - Spin Time: 1.905s
  - Overhead Time: 0s
- Total Thread Count: 14

CPU Time は 7 秒程度

14 スレッドで実行されている

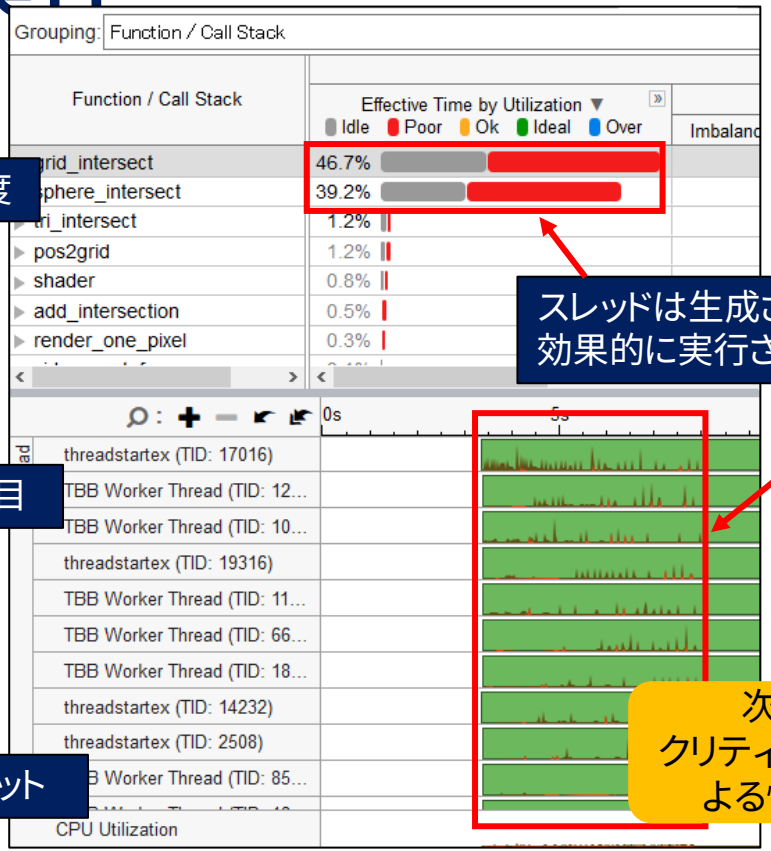
Top Hotspots

This section lists the m… performance.

Function	Module	CPU Time
grid_intersect	analyze_locks.exe	1.487s
func@0x1800164d0	USER32.dll	1.477s
RtlEnterCriticalSection	ntdll.dll	1.373s
sphere_intersect	analyze_locks.exe	1.248s
func@0x18000c540	USER32.dll	0.925s
[Others]		

フラグが立っている処理にも注目

grid_intersect 関数がホットスポット



スレッドは生成されているが
効果的に実行されていない

次のステップ→
クリティカル・セクションによる性能低下を確認

Threading 解析を実行

Wait Time が 200 秒

Elapsed Time : 19.305s

Wait Time	**200.227s**
Wait Count	2,928
Spin Time	2.114s
CPU Time	7.541s
Total Thread Count:	15
Paused Time	0s

Thread Concurrency Histogram

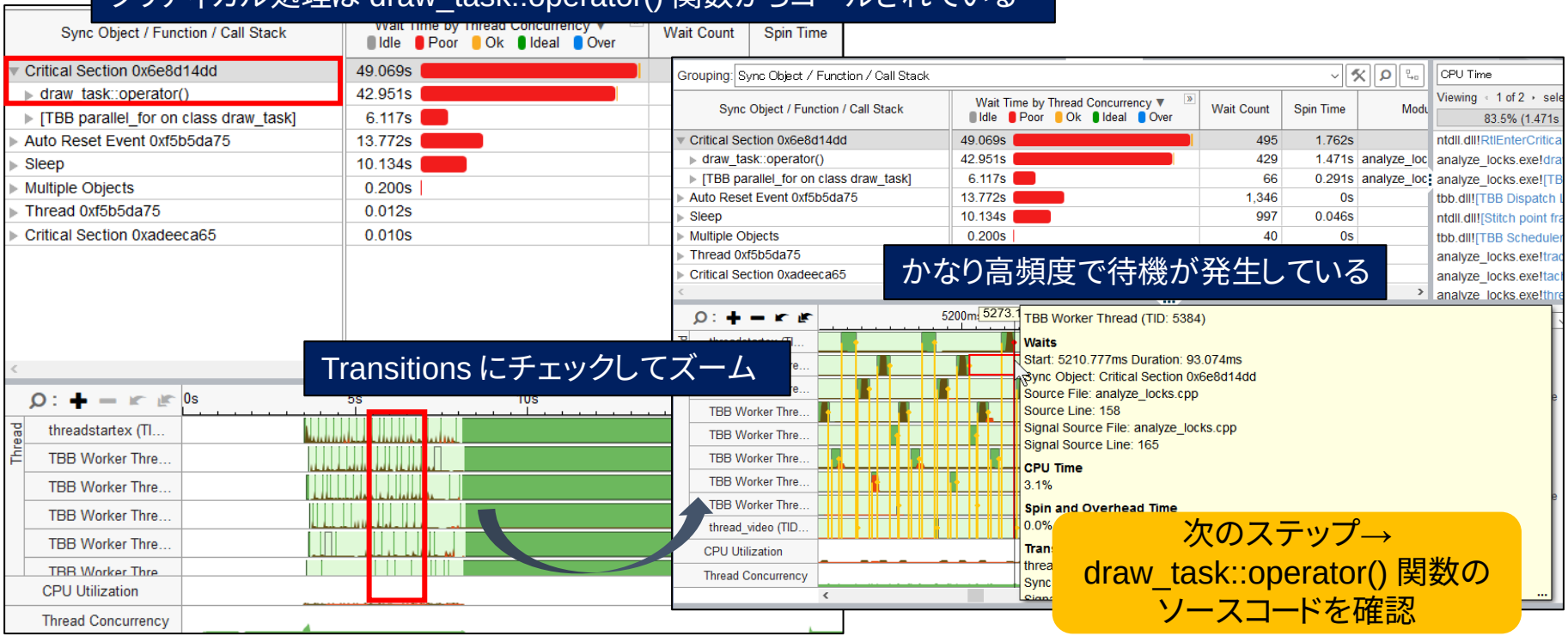
This histogram represents a breakdown of the Elapsed Time. I running simultaneously. Threads are considered running if they are eit Essentially, Thread Concurrency is a measurement of the number of threads that ads are in the runnable state and not consuming CPU time.

スレッドの同時実行時間が少ない

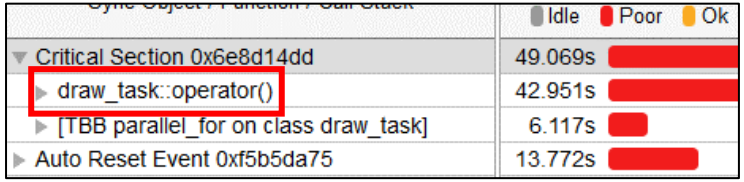
次のステップ→
Bottom-up を確認

Threading 解析を実行

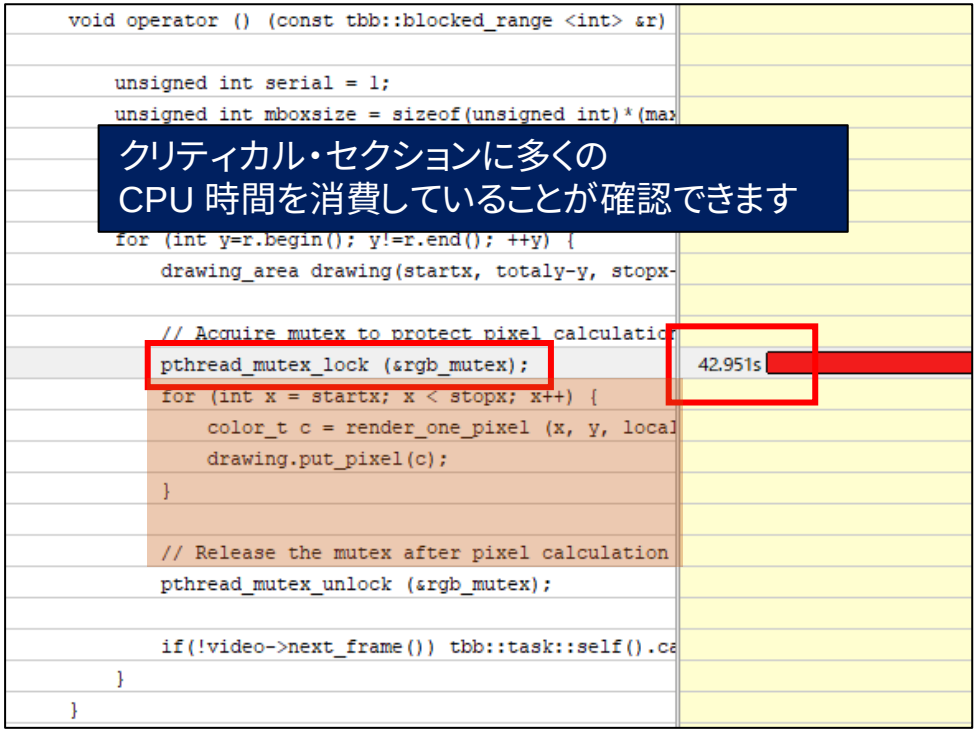
クリティカル処理は draw_task::operator() 関数からコールされている



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・セクションに多くの
CPU 時間を消費していることが確認できます

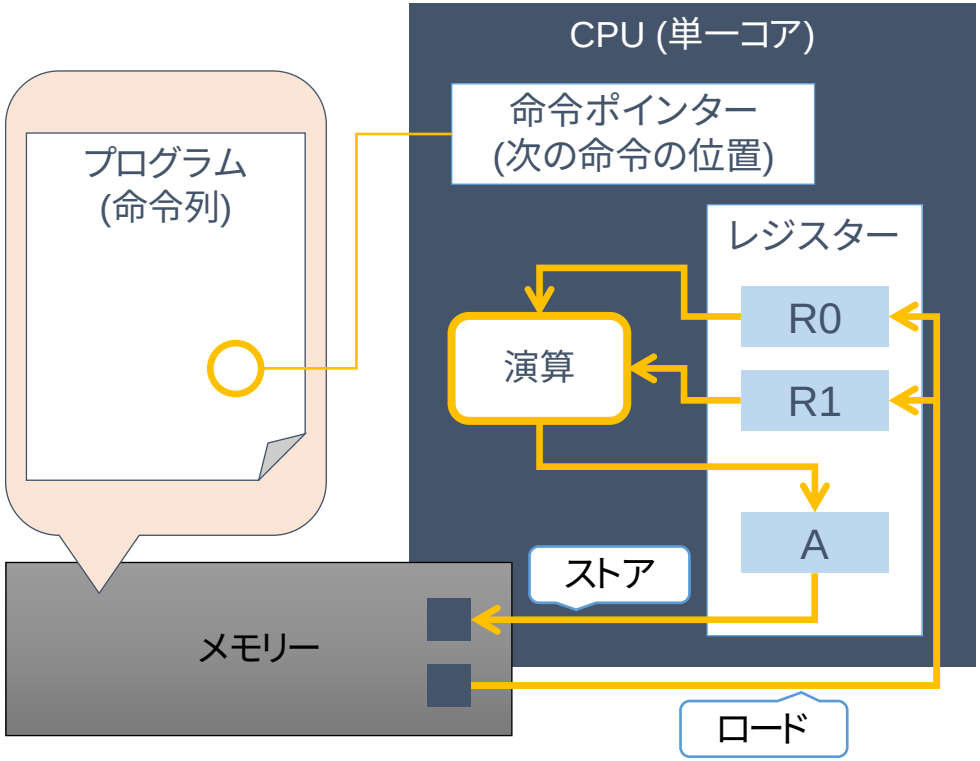
クリティカル・セクション内部の処理に注目

- 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

マイクロアーキテクチャーの分析

プロセッサーとプログラム

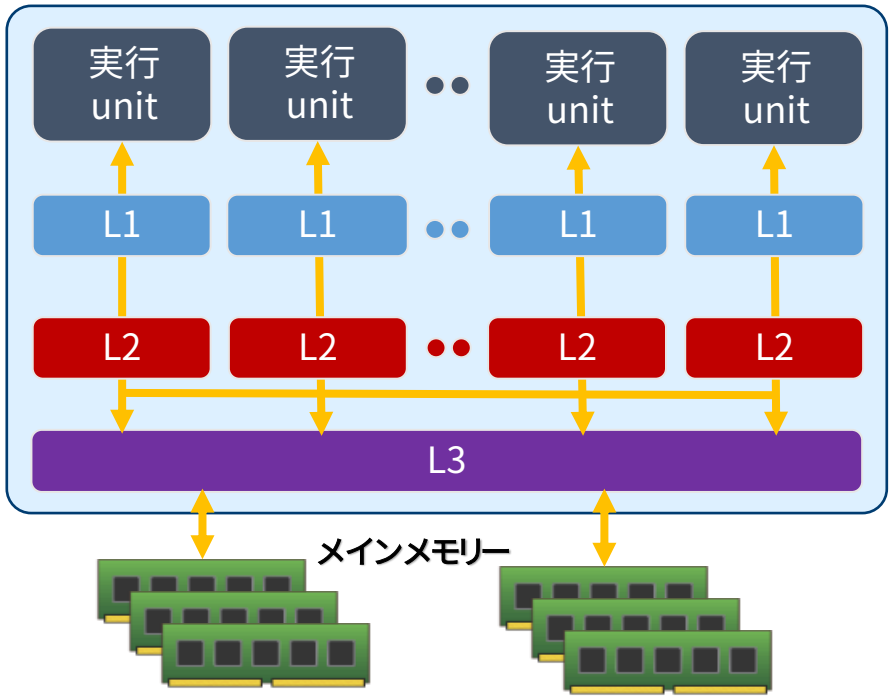
- プロセッサーは命令を実行します
- ロード/ストア
  - ✓ データをメモリーから レジスターへ読み込み
  - ✓ データをレジスターから メモリーへ書き込み
- レジスターを使用して演算
  - ✓ 四則演算、ビット操作、比較
- ジャンプ
  - ✓ 次に読み込むプログラムの 位置を変更
  - ✓ 関数呼び出し/復帰や 条件分岐を実現



ハードウェアのメモリー階層

- 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

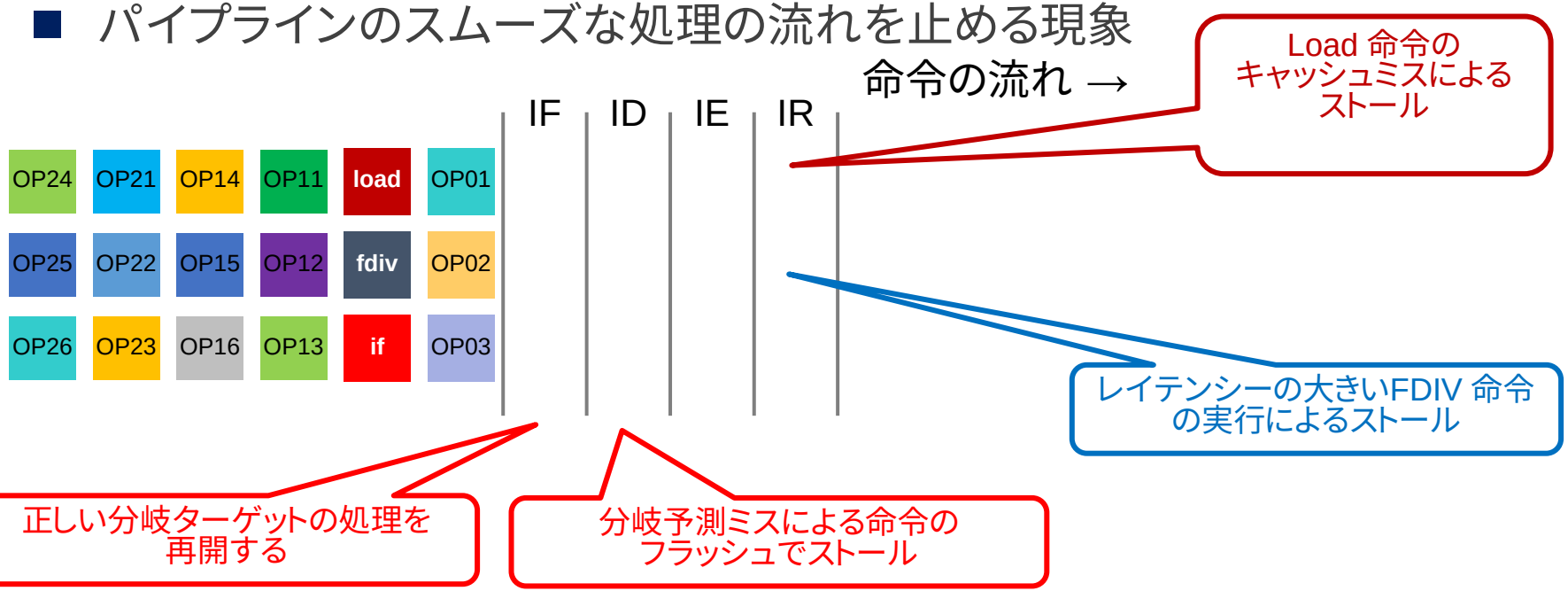


コアの内部: パイプラインの並列実行

- プロセッサー内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
- 命令フェッチ (Instruction Fetch)
  - ✓ メモリーから命令を取り込む
- 命令デコード (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
- 命令実行 (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
- 命令リタイア (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

パイプラインのストール (停止)

- 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

- ストールにより無駄になったサイクルを少なくする
  - ✓ ストールが頻発する処理を最適化の候補にします
    命令あたりのサイクル数 (CPI) を確認して、
    ストールしていそうな処理を見つける
- CPI 値が高い要因
  - ✓ レイテンシー (遅延) の大きい命令
    主に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
    分岐命令に続く命令は、分岐方向が定まるまで未確定

Hardware Event-based サンプリングをベースにしたHotspots 解析から CPI を確認

- CPU 負荷の高い関数、ループごとの CPI 値を表示します

CPI: Clockticks per Instructions Retired

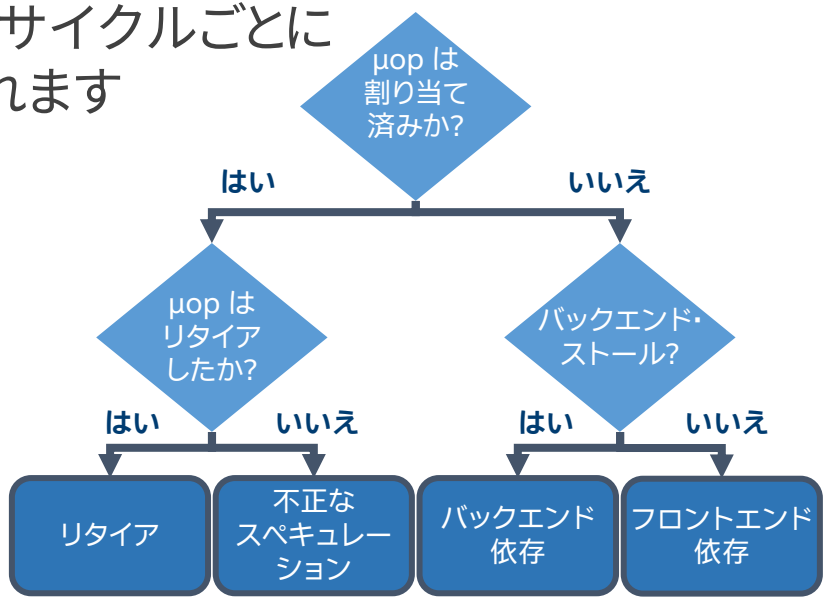
処理に対しての効率を表し、**少ない**ほど良い値とされます
理論値はプロセッサー・アーキテクチャーによって異なる

Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼	Instructions Retired	CPI Rate	CPU Frequency Ratio	
FindPrimesompparallel_for@19	9.748s	19,400,500,000	2.012	1.143	ma
matmulompparallel@26	9.327s	22,655,500,000	1.643	1.139	ma
pi_calcompparallel_for@23	2.155s	12,999,000,000	0.663	1.141	ma
_kmp_hyper_barrier_release	0.559s	724,500,000	3.106	1.148	lib
_kmp_hyper_barrier_gather	0.174s	161,000,000	4.239	1.121	lib

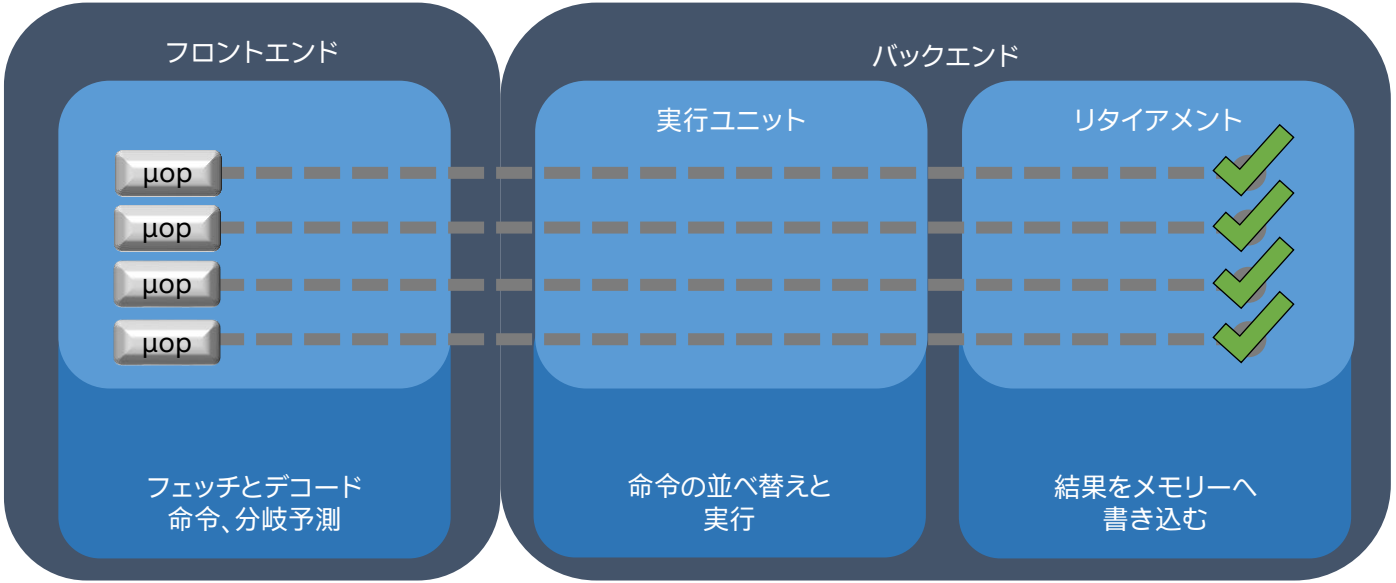
パイプライン・スロットの分類

- 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して考えられます
  - ✓ 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



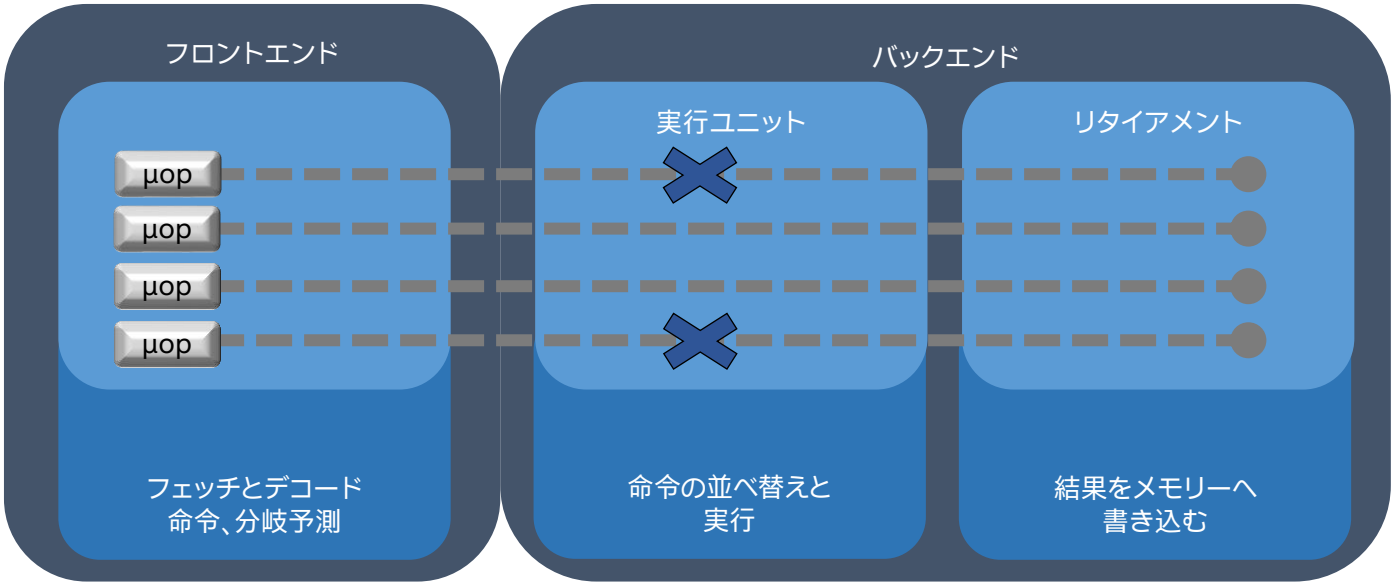
パイプライン・スロットの分類: リタイア

- ストールを起こさずに命令を完了する



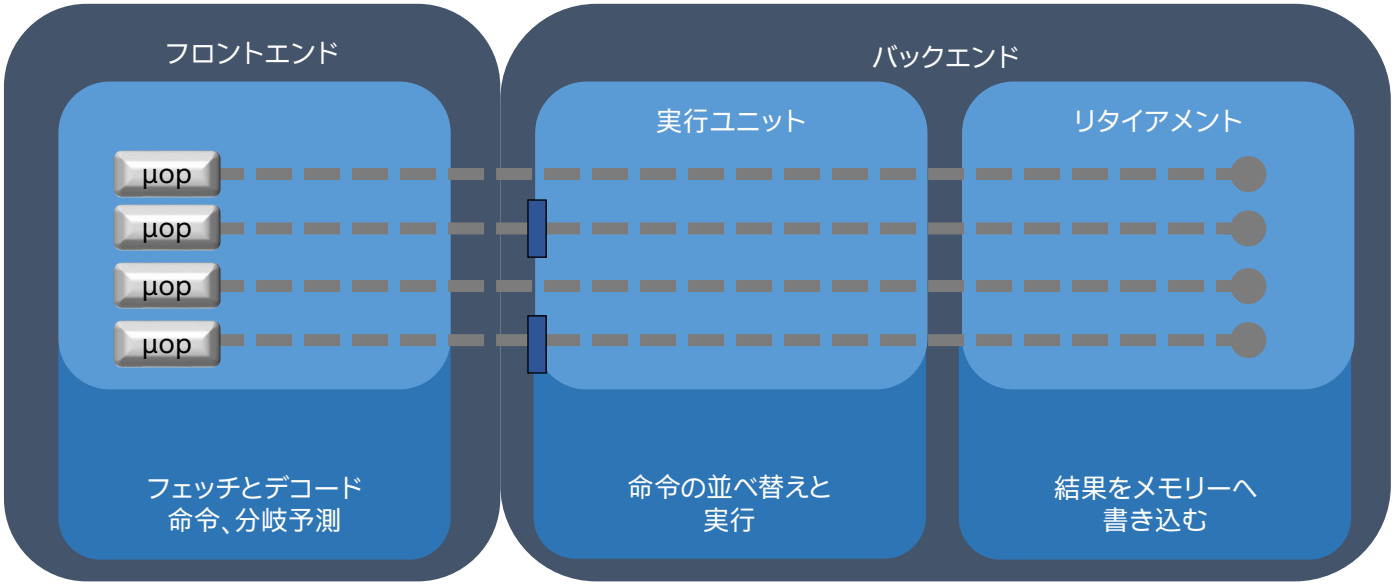
パイプライン・スロットの分類: 不正なスペキュレーション

- 分岐予測ミスによるキャンセルにより、μop が リタイアせずにバックエンドから削除 (キャンセル) される



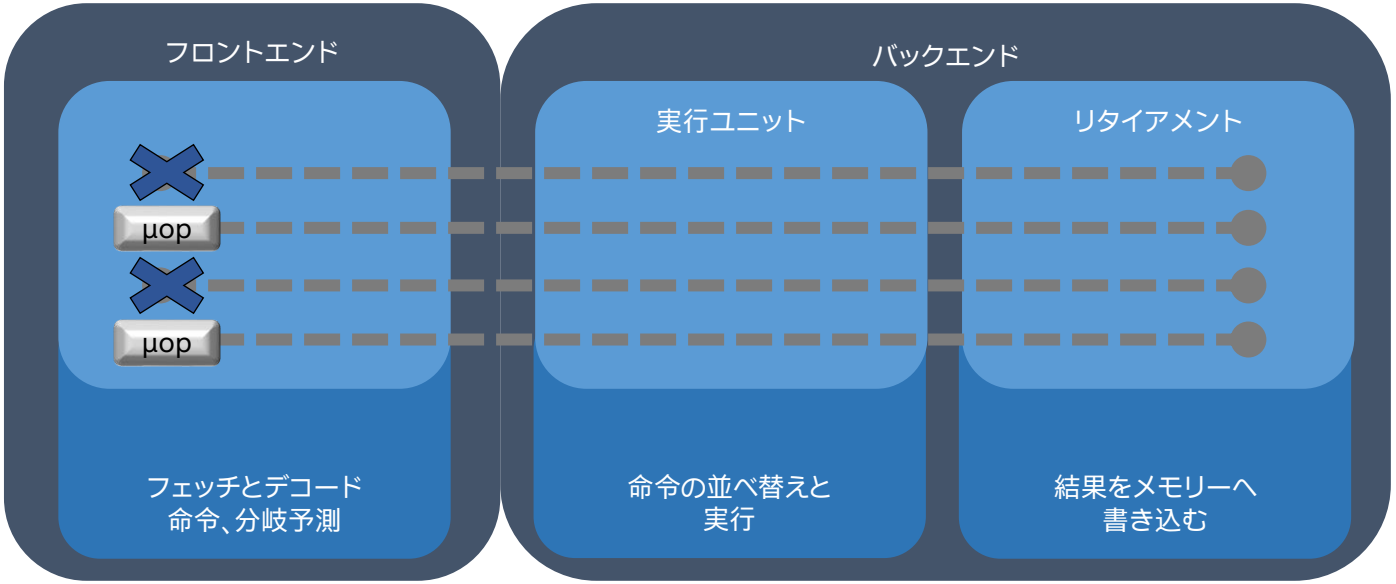
パイプライン・スロットの分類: バックエンド依存

- 実行に必要なデータを待っている、もしくは時間のかかる命令を実行しており、バックエンドが µop を受け取ることができない状態



パイプライン・スロットの分類: フロントエンド依存

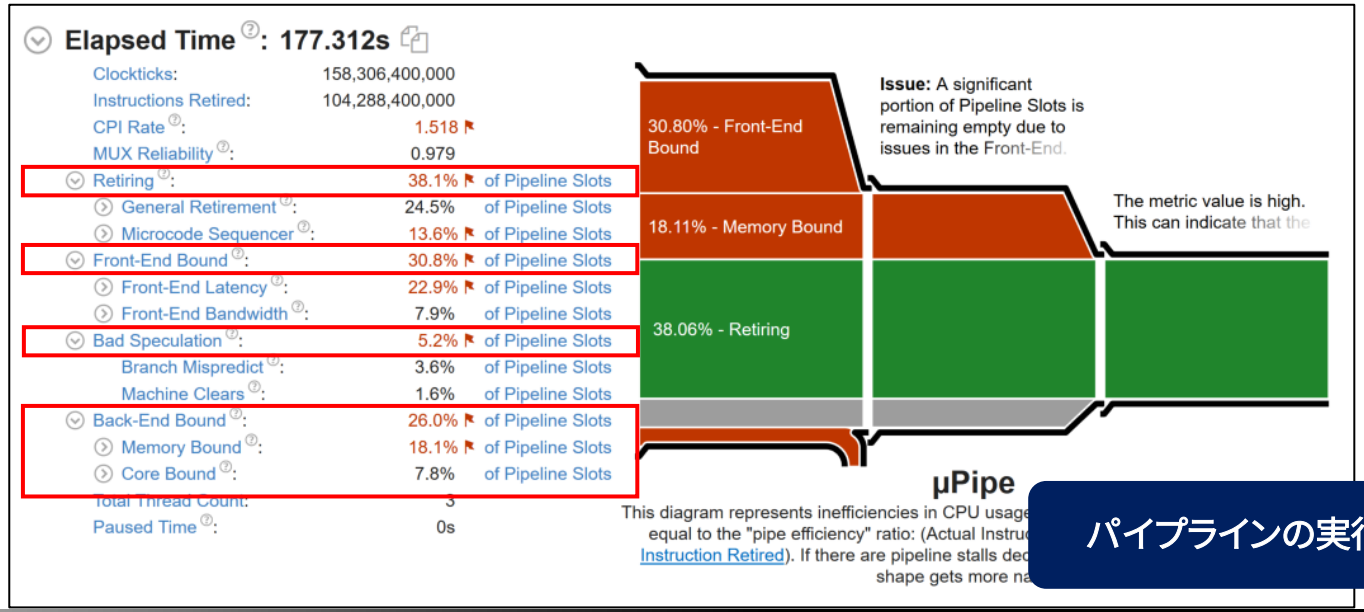
- バックエンドは μop を受け取る準備ができているが、コードのフェッチや命令のデコードの遅延によりフロントエンドが μop を供給できない場合



Microarchitecture Exploration 解析

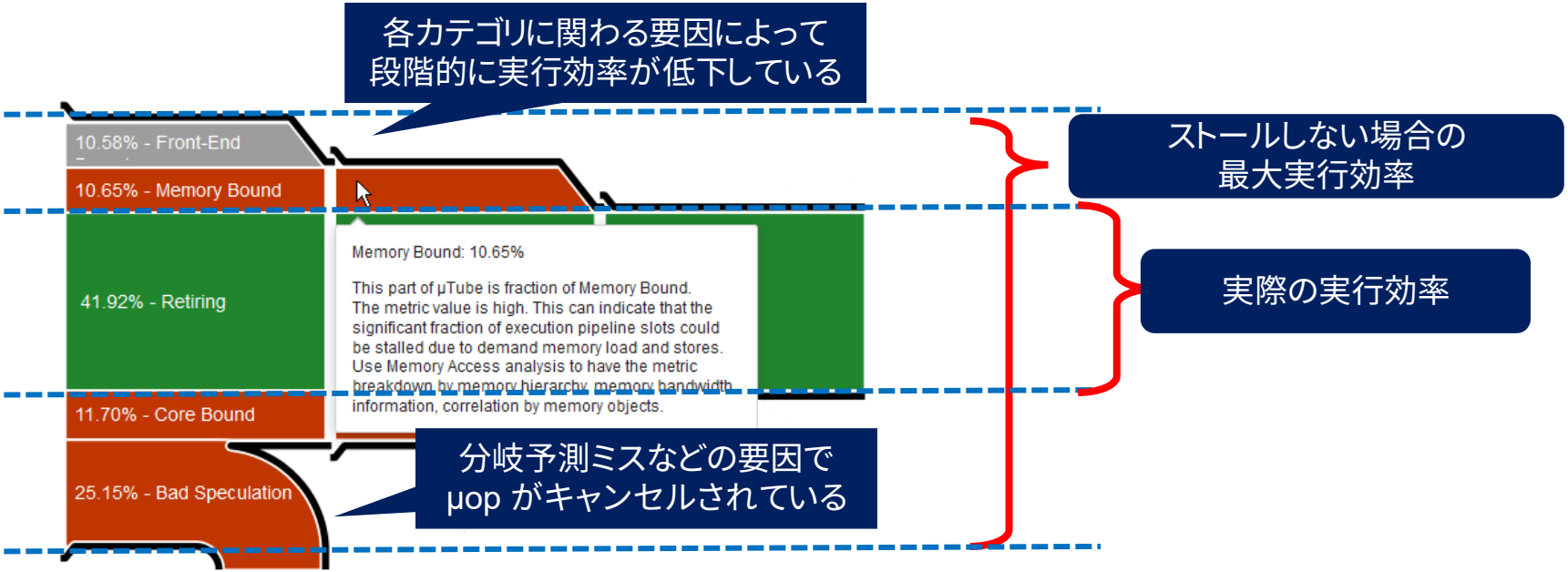
- ダイアグラムによるパイプラインの抽象化

  カテゴリされている要因に制限されてパイプは狭くなります



パイプラインの実行結果を視覚化

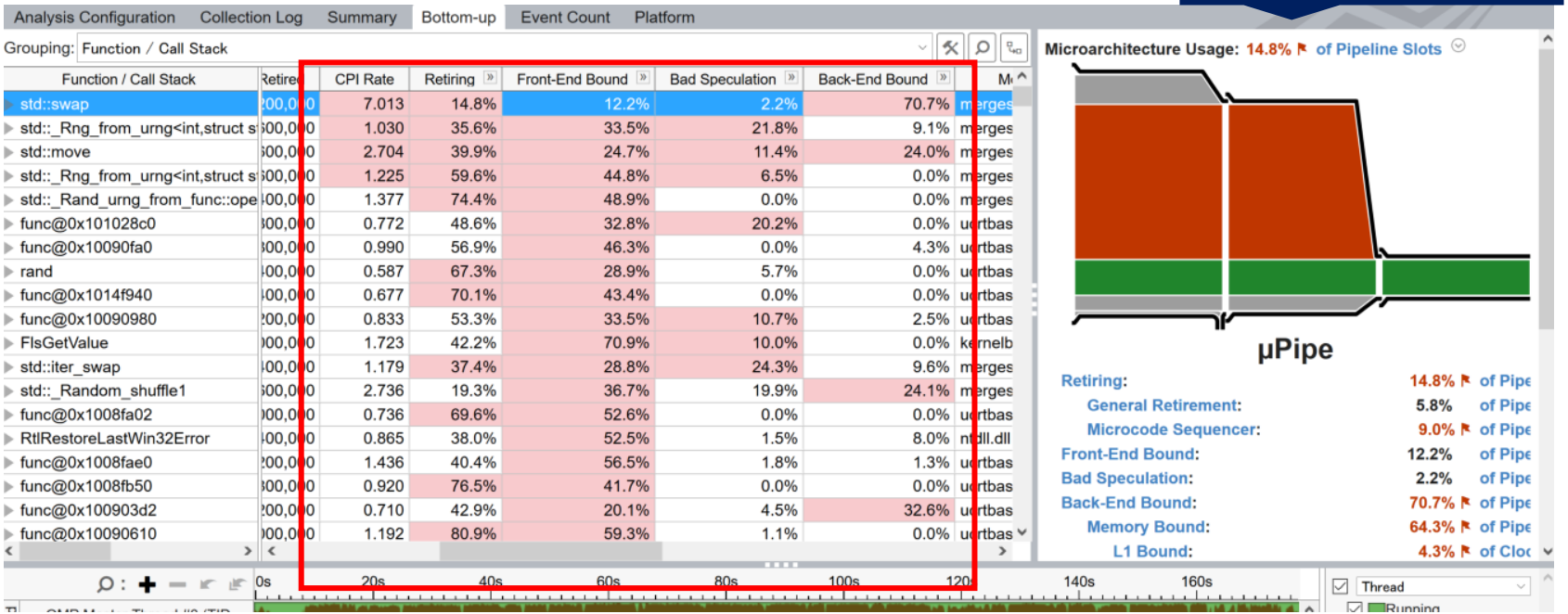
パイプライン内部の実行を視覚化



パイプライン内部の実行を視覚化

- 関数、ループごとの実行効率を確認できます

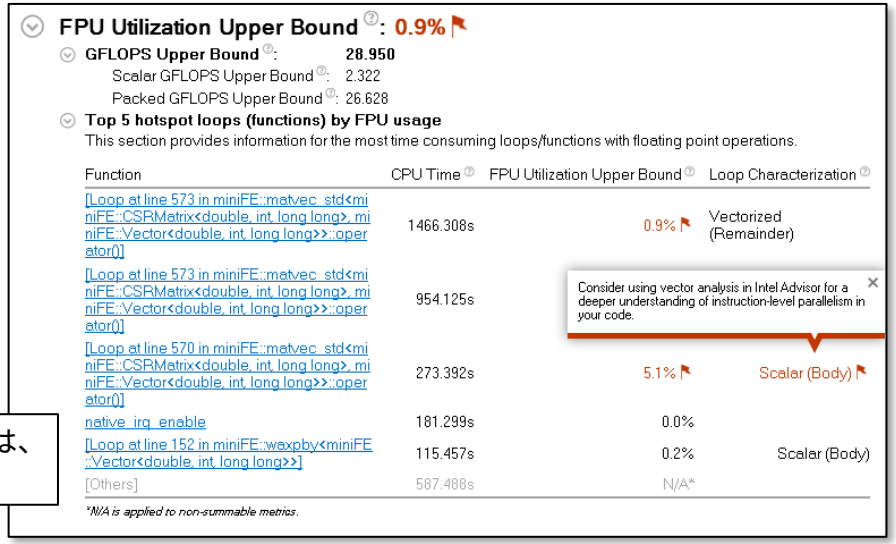
関数ごとのダイアグラム



HPC Performance Characterization 解析

- 計算集約型プログラム向けの性能情報を中心に表示します
  - ✓ CPU 利用状況 、メモリー帯域、FP ユニットの活用

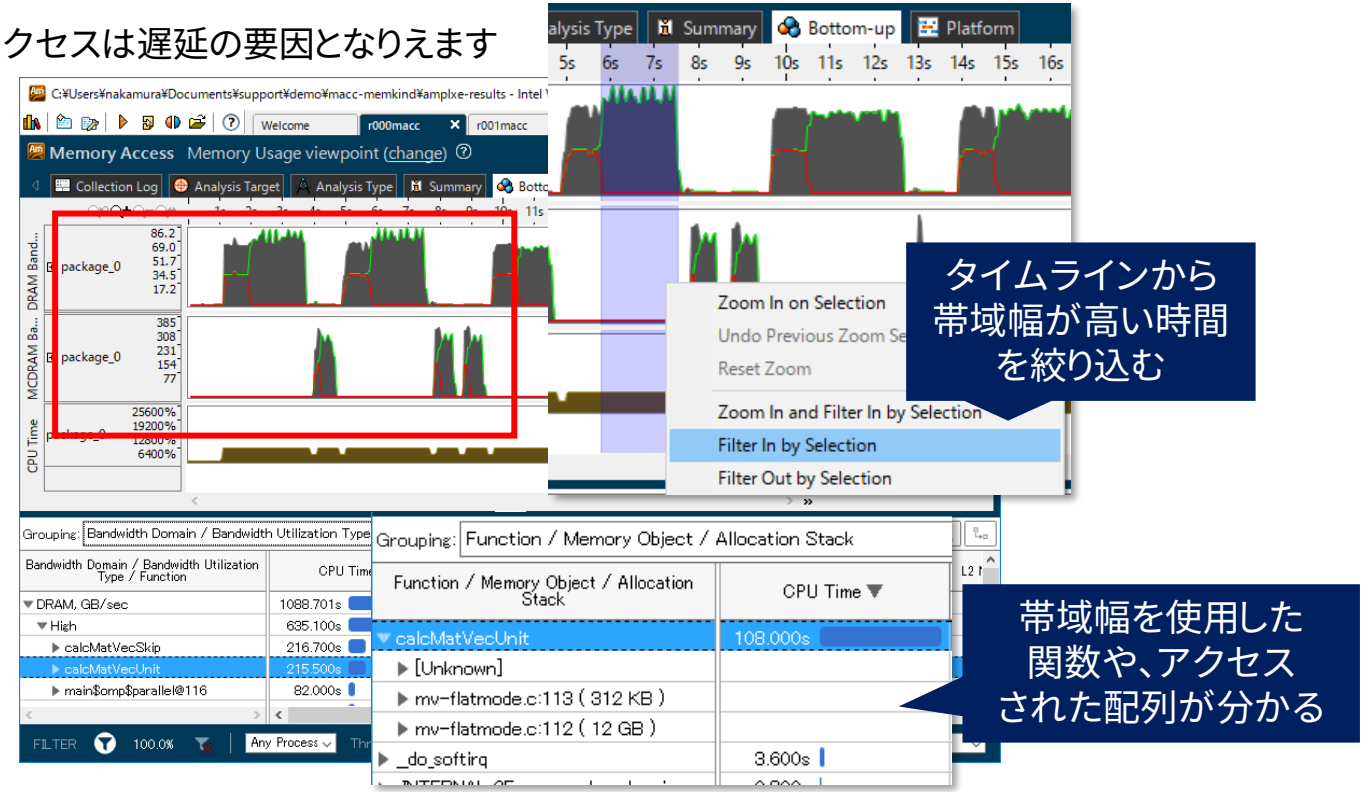
FPU 負荷の比率 (%)
FPU 使用率による上位 5 つのループ/関数
スカラーとパックドに GFLOPs を
細分化 (ベクトル化されているかどうか)



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

Memory Access 解析

UPI を経由したメモリーアクセスは遅延の要因となりえます



資料 まとめ

- インテル® コンパイラーの最適化オプションをご活用ください
  - ✓ 最適化レポートからコンパイラーの最適化状況を確認できます
- インテル® VTune™ プロファイラーの解析機能により
  アプリケーション内で時間を消費している処理を素早く発見できます
  - ✓ CPU 命令レベルで最適化に有益な情報を取得いただけます
- インテル® ソフトウェア開発ツールはインテルプロセッサー向けに
  サポートおよび調整された各種コンポーネントを提供しています

お問い合わせはこちらまで

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