インテル・テクノロジー講座
ソフトウェア最適化方法論(1)

インテル株式会社

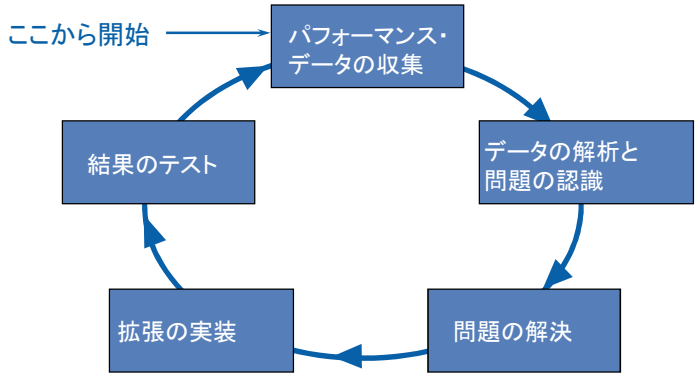
2017 年度HAL版

コースの内容

- 性能最適化サイクルの概要
- 性能最適化作業の内容
- 性能測定の方法
  - インテル® VTune™ Amplifier XE の紹介

概要

性能最適化サイクル

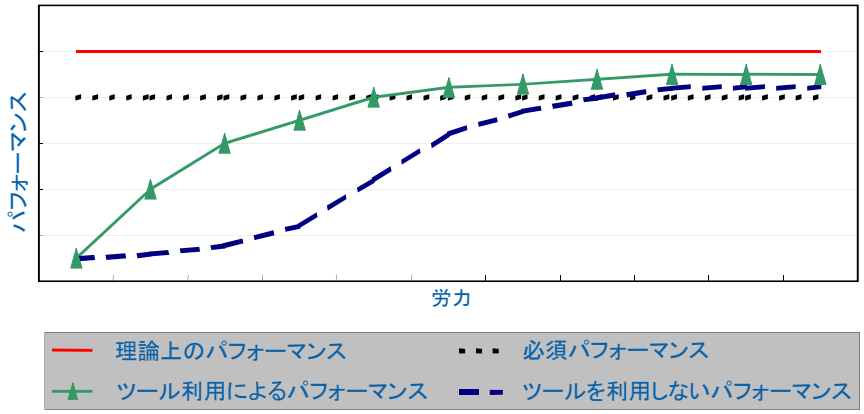


概要

開始時期

- ユーザーの要求
- ソフトウェア・ベンダーの要求
- 関連するドキュメントへパフォーマンス要求を盛り込む
- 製品ライフサイクルの各段階（要求事項確認、設計、およびテスト）でパフォーマンスを考慮
- 例外: アプリケーションのデバッグが完了し、ソースコードの整理が完了するまで “コード・チューニング” は行わない

概要
労力 vs. パフォーマンス



概要
終了時期

アーキテクチャーの利用効率は最大か?

理論的な最大値の計算方法を知っておく必要がある

パフォーマンス要件は満たされているか?

完了まで多岐にわたる最適化が徐々に行われる

チューニングのゴールを決定することは、すべてのチューニングの出発点です。すべてのチューニング戦略は、ターゲットとなるプロセッサー上で最短時間で最良のスピードアップを達成するというゴールに進むためであり、非常に重要なことです。

概要

広い視野での再考察

「個々の細かな効率のことは忘れて、全体の効率について考えるべきです。中途半端な最適化はよい結果を生みません」

Donald Knuth

質の高いコードとは:

- 可搬性がある
- 判読可能である
- メンテナンスが容易である
- 信頼性が高い

パフォーマンスと品質の最適なバランス

コースの内容

- 性能最適化サイクルの概要
- 性能最適化作業の内容
- 性能測定の方法
  - インテル® VTune™ Amplifier XE の紹介

パフォーマンス・サイクルの詳細

パフォーマンス・データの収集（サイクル1）

タイマー

- 実際の利用時間を取得
- 正確で、オーバーヘッドが少ない

ツールを利用（インテル® VTune™ Amplifier XE等）

- プロファイラー: コードの使用率に関する情報を収集
- パフォーマンス・モニター: システムリソースの使用率に関する情報を収集

パフォーマンス・データの収集

タイマー

- パフォーマンスに影響しやすいコードの識別
- 便利なタイマー・プロパティー
  - 正確な時間情報の取得
  - アクセス・オーバーヘッドが少ない
  - リセットがほとんど発生しない
- インテル® コンパイラーのプラグマを使用してプロセッサーの内部クロックカウンターにアクセス

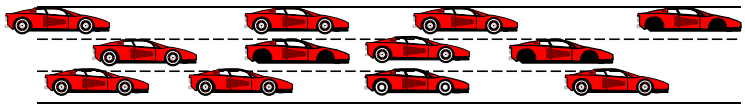
パフォーマンス・サイクルの詳細

適切なワークロード

良いワークロードには、次のような特徴がある

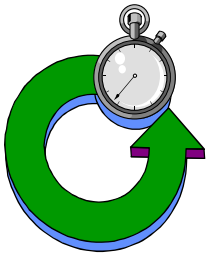
- 測定可能
  （性能計測が定量化できること）
- 再現可能
  （異なる実行で同じ結果をもたらす）
- 静的
  （仕事量が時間によって変化しない）
- 典型的
  （実行される作業は正常なシステム状況下におけるストレスを示す）

スループット



- 時間単位あたりにシステムが完了できる作業量
- 通常は、上記の値がスループットの上限になる
- 帯域幅は、スループットの 1 つの測定単位である

レイテンシー



- アクションを完了するために必要な合計時間のこと
- レイテンシーは、サブタスクのいくつかのレイテンシーの累計として表示される
- レイテンシーは、応答時間（レスポンス時間）とも呼ばれる

ベンチマーク

- Linpack* ベンチマーク
- NAS
- STREAM
- SPEC* (Standard Performance Evaluation Corporation)
- TPC*

MIPS/FLOPS/CPI

MIPS
- 1 秒あたりの命令数（百万単位）

FLOPS
- 1 秒あたりの浮動小数点演算数

CPI
- 命令あたりのサイクル数

パフォーマンス・サイクルの詳細

スピードアップ

2 つの基本式

$$\text{スピードアップ} = \frac{\text{基準となる時間}}{\text{最適化された時間}}$$

$$\text{スピードアップ} = \frac{\text{最適化されたスループット}}{\text{基準となるスループット}}$$

パフォーマンス・サイクルの詳細

データの解析と問題の認識（サイクル2）

1. 現在のパフォーマンスの基準
2. hotspot の調査 （処理時間が費やされる場所）
3. ボトルネックの識別 （性能低下の原因）
4. 潜在的な最大パフォーマンスの計算

パフォーマンス・サイクルの詳細

hotspot の調査

パレートの法則（80/20 の法則）

- 重要でない 80% ではなく、重要な 20% に集中する

hotspot: アクティビティーが大量に含まれている
アプリケーションまたはシステム内の場所

通常はループからなる

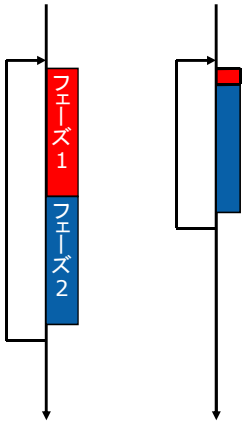
hotspot のないアプリケーションでは、次の項目を検査する

- メモリーレイアウト
- 例外の発生
- コンパイラーの有効な活用

パフォーマンス・サイクルの詳細

アムダールの法則:
図説

実行時間の半分のみが最適化された場合、2 倍のスピードアップは不可能である



アムダールの法則

アムダールの法則の式

$$\text{スピードアップ} = \frac{1}{(1 - \text{並列化率}) + \frac{\text{並列化率}}{\text{並列化による スピードアップ}}}$$

パフォーマンス・サイクルの詳細
その他の考慮事項

- Big O 記法 （O（n）、O(n²)、O（n log n））
- 使用率、効率、スループット、レイテンシー
- ボトルネック
  - I/O、メモリー、CPU
- MIPS/FLOPS/CPI
- 同時性、並列性
- スケーラビリティー
- 計算ごとのロード/ストア

（机上での推定）
演習 1:

```
float *loop(float *data1, float *data2) {
        float Pi = 3.14, *ans;
        ans = (float *) malloc(MAX * sizeof(float));
        for ( i = 1, i < MAX, i++) {
                ans[i] = data1[i] + Pi * data2[i]; }
        return ans;
}
```

計算: Big O 記法 O(n)
合計メモリー使用量 12MAX ＝ MAX＊float（4バイト）＊3つのデータ配列
ストア vs. ロード vs. 浮動小数点演算 1ストア＋3ロード＋2演算

演習 2:

MAX=4,000,000 (4百万)として想定する

では、第 3 世代インテル® Core™ i7 3370 プロセッサー(3.40GHz Memory DDR3 1600 x 2 ポート)でシングルスレッドが計算を完了するための予想時間は?

4,000,000 * 4 byte * 4 load / store = 64,000,000 (64MB)
0.064 GB / 25.6 GB = 0.0025 sec

問題の解決(サイクル3)
最適化デザインレベル

全般的なチューニング方法論は、システムレベルから始まり、マイクロアーキテクチャー・レベルまで網羅する。特定のチューニング・ゴールに関係なく、高いレベルから低いレベルまで、順に沿って、レベルごとに解析を行なう

- システムレベル
- アプリケーション・レベル
- マイクロアーキテクチャー・レベル

最短時間で最良のスピードアップが得られるようにするには、これらのステップに沿うことが重要

問題の解決

チューニング・ゴールと調査する領域

順序	チューニング・レベル	ゴール	調査する主な領域	スピードアップの可能性
1	高: システムレベル	アプリケーションとシステムの相互作用を改善することにより、アプリケーションをスピードアップする	•ネットワークの問題 •ディスクのパフォーマンス •メモリーの使用量	3 倍
2	中: アプリケーション・レベル	アプリケーションのアルゴリズムを改善することにより、アプリケーションをスピードアップする	•ロック •ヒープの競合 •スレッド化アルゴリズム •API の使用量	2 倍
3	低: マイクロアーキテクチャー・レベル	プロセッサー上でのアプリケーションの実行を改善することにより、アプリケーションをスピードアップする	•アーキテクチャーのコーディング上の問題点 •データ/コードの局所性 (キャッシュ) •データのアラインメント	1.1 ～ 1.5 倍

インテル® VTune™ Amplifier XEを使用して、これらのチューニング方法論を実装し、目標とするプラットフォームで最短時間で最良のスピードアップを実現する

問題の解決

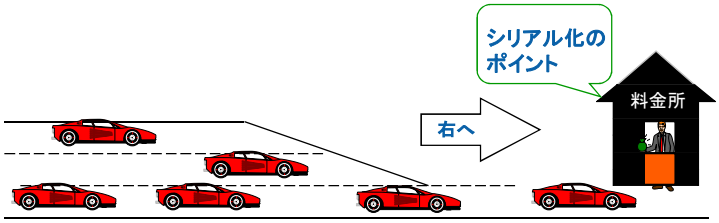
システム・レベル・チューニング

システムリソースの利用方法を最適化し、システムとの相互作用を改善することでアプリケーションのスピードアップを目指す。システム・レベル・チューニングではたいていの場合、最少の作業で最大のスピードアップが可能

1. カウンターモニター・データ・コレクターで OS のパフォーマンス・カウンターを監視
2. システムレベルのパフォーマンス・ボトルネックとシステムの相互作用に注目
3. 改善の余地がありそうな範囲にズームし、チューニング・サマリーに注目

システムレベルのパフォーマンスの問題点をすべて解決した場合、または重大な問題点がないと判断した場合は、チューニングの次のレベルである、アプリケーション・レベル・チューニングに進む

ボトルネックの識別



- ボトルネックは、システムで最も低速な部分である
- ある処理が他の処理が完了するまで待たなければならない場合、シリアル化のポイントが存在する
- ボトルネックは、システムが単位時間あたりに処理できる作業量を最終的に決定する

使用率

リクエストに応じるためにデバイスが稼動していた時間の割合

$$\text{使用率} = \frac{\text{デバイス稼動時間}}{\text{合計時間}} * 100$$

時間の残りの割合は、*アイドル時間*

アイドル時間 = (100 - 使用率)

典型的なボトルネック

メモリー
- 合計メモリー使用率を計算する
- スループットを計算する

I/O
- ディスク上のデータを計算する
- スループットを計算する

CPU

問題の解決

アプリケーション・レベル・チューニング

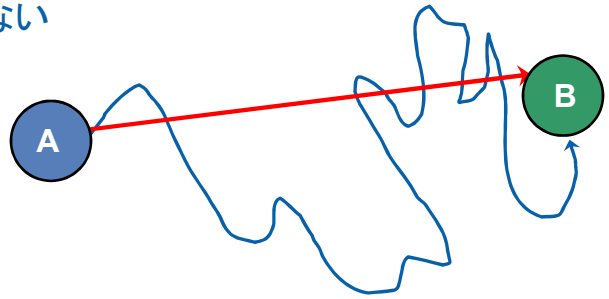
アプリケーション・レベル・チューニングのゴールは、アプリケーションのアルゴリズムの改善、実装のスレッド化、および API やプリミティブの使用によって、アプリケーションをスピードアップすること

アプリケーション・レベル・チューニングでは、2 つの主要なチューニング戦略を選択できる

- スレッドモデルの向上
- 演算の効率性の向上

効率

- 必要な作業を行うために実際に費やされた時間の割合のこと
- リクエストに応じている間に、多くのオーバーヘッドやその他の非効率な処理が発生する場合がある
- 使用率が高いことは、必ずしも効率が良いことを意味しない



問題の解決

スレッドモデルの向上

効率的なスレッドモデルを使用することは、パフォーマンスのスケールを向上するためには非常に重要

シングルスレッド・アプリケーションの場合、パフォーマンスを向上するための最初のステップは、アプリケーションにマルチスレッド機能を追加すること

既にアプリケーションがマルチスレッド化されている場合は、マルチプロセッサー・システム（およびハイパー・スレッディング・テクノロジー搭載の単一プロセッサー・システム）上でパフォーマンスおよびスケールを向上させる

問題の解決

演算の効率性の向上

サンプリングを使用し、アプリケーションのパフォーマンスに与える影響が高いコード領域を識別する

- ホットスポットの使用
- コンカレンシーの使用
- ロック＆ウェイトの使用
- EBS（イベント・ベース・サンプリング）の使用

アプリケーション・レベルのパフォーマンスの問題点をすべて解決した場合、または重大な問題点がないと判断した場合は、チューニングの次のレベルである、マイクロアーキテクチャー・レベル・チューニングに進む

問題の解決

マイクロアーキテクチャー・レベル・チューニング

- 特定のプロセッサー上でのアプリケーションの実行を改善し、アプリケーションのパフォーマンスのスピードアップを目指す
- このチューニング・タイプは、特にプロセッサー集中型アプリケーションに適切
- 開発しているアプリケーションが、プロセッサー集中型ではない場合、マイクロアーキテクチャー・レベル・チューニングを行う前に、システムレベルおよびアプリケーション・レベル ・チューニングを行う

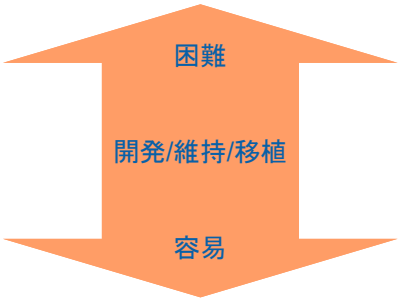
マイクロアーキテクチャー・レベル・チューニングの一般的な方法論は次のとおり

1. パフォーマンスに与える影響が大きい、最も時間を費やしているコード領域を見つけます。
2. インテル® アーキテクチャー上でそれらのコード領域の実行を解析します。
3. マイクロアーキテクチャー・レベルでのパフォーマンスの問題点を識別します。
4. パフォーマンスを向上するために問題点を回避する方法を決定します。

問題の解決

コード・チューニングのレベル

アセンブラー
組み込み関数
C++ ベクトルクラス
マルチスレッド
ループ変換
インテル® コンパイラー
インテル® パフォーマンス・ライブラリー



問題の解決

コード・チューニング

並列処理の場合
- –クラスター別に処理を分散(分散メモリー)
- –単一ノードの最適化
- –プロセッサー(コア)別に処理を分散(SMP)

拡張の実装（サイクル4）

- インテル® パフォーマンス・ライブラリーを使用する
- 各種コンパイラー・オプションを使用する
- 実装を行う前に、コンパイラーやハードウェアが拡張を自動的に実装するかどうかを調査
- ソースを修正する
  （ループ変換、キャッシュ最適化、SIMD、OpenMP*、組み込み関数、アセンブリー）

テスト（サイクル5）

- アプリケーションが拡張後も正しく実行することを確認する（リグレッション・テスト）
- 拡張が実際にパフォーマンスを向上させていることを確認する
- スピードアップを計算する
- 最適化を終了するかどうかを決定する

コースの内容

- 性能最適化サイクルの概要
- 性能最適化作業の内容
- 性能測定の方法
  - インテル® VTune™ Amplifier XE の紹介

性能計測の方法

チューニングの第一歩は現状を知ることから始まるが、時間計測はほとんどのソフトウェアに対して有効

もっとも簡単な方法は、画面を見ながら時計を使用しておよその時間を計測することであるが、コード中に計測コードを埋め込んでより正確な時間を得ることができる

- C/C++ 言語がサポートするタイマー機能 : clock() 関数
- CPU が持つタイマー機能 : RDTSC 命令、RDTSCP 命令

clock() 関数

clock_t clock(void);

clock() 関数は Windows や Linux 環境で提供される C ランタイムライブラリーに含まれる

clock() 関数は呼び出したプロセスが起動してからのウォールクロック経過時間を返す。利用されるタイマーの精度は、1/CLOCKS_PER_SEC 秒。取得した時間を表示する場合、時間/CLOCKS_PER_SEC で秒単位に変換する

clock() 関数の利用例

```
clock_t start_time, stop_time;

start_time = clock();
// 時間計測する処理をここで行う
stop_time = clock();

printf("経過時間%lf 秒\n",
((double)(stop_time - start_time)/CLOCKS_PER_SEC));
```

Sample code: clock.c

プロセッサーのリアルタイム・タイマーを利用する

Pentium 以降の IA-32 プロセッサーには、64 ビット幅のリアルタイム・クロックカウンターが搭載されており、このタイマーはプロセッサーの電源が投入されると、クロック・カウントごとにインクリメントされる。IA-32 命令セットにはこのタイマーを読みだす命令が用意されており、プロセッサー・クロックの精度で時間計測ができる

```
RDTSC       // 引数なしで実行

EAX   レジスター   64ビットカウンターの下位32ビット
EDX   レジスター   64ビットカウンターの上位32ビット
```


RDTSC 命令の利用例

```
_asm {
        rdtsc                   // 64 ビットのタイマーカウンターを取得
        mov    high1, EDX       // EDX <- 上位 32 ビット
        mov    low1, EAX        // EAX <- 下位 32  ビット
}
```

MASM フォーマットでの記述例

このタイマーで取得できる精度は、1 / プロセッサー動作クロックでカウントされるクロック・カウント。たとえば 2GHz のプロセッサーの場合、1 クロックは 0.5ns（ナノ秒 ）となる

Sample code: rdtsc.c

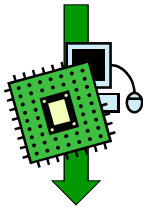
コンパイラーがサポートする RDTSC 命令の利用例

```
#pragma intrinsics(__rdtsc)
unsigned __int64 count1, count2

count1 = __rdtsc();
….
ccount2 = __rdtsc();
```

Visual C++ では組み込み関数がサポートされており、
前述の記述を簡単に次のように表現できます。

プロセッサー時間と待機時間



プロセッサー 時間は、論理プロセッサーで
スレッドの実行に費やした時間

待機時間は、スレッドが同期待機や I/O 待
機などが完了するまで待機に要した時間

時間計測の問題

タイマー計測を埋め込んだ時間計測は簡単で、複雑な構造のソフトウェアでも容易に時間計測が行えるが、いくつかの問題もある

1. 計測範囲内の内訳が解らない
2. ソフトウェア内には複数のホットスポットが存在する
3. ホットスポットの原因が解らない
4. プロセッサー時間と待機時間が区別できない

ツールを利用した性能計測

より高度なそして簡単な解析を行うには、性能解析をサポートするツールが必須
Visual Studio や GCC にもロードテストやプロファイル機能が備わっているが、ここではインテル® VTune™ Amplifier XE を利用したパフォーマンス解析の活用例を紹介する

```
start_time = clock();
for (i=0; i<num_steps; i++){
        x = (i + .5)*step;
        sum = sum + 4.0/(1.+ x*x);
}
pi = sum*step;
stop_time = clock();
```

Sample code: pi.c

性能解析ツールに対する要請

- 性能測定が対象ソフトウェアの性能に影響してはならない
- プログラマはシステム全体の性能を知る必要がある
- 性能解析を行う通常状態でソースコードを参照できなくてはならない

VTune Amplifier

- ソフトウェアの性能に影響するボトルネックを見つけ出し、影響を少なくするのを手伝う
  - プロセッサ構造及びプラットフォーム上の解析
  - 低オーバヘッドのサンプリングでホットスポットを特定
  - コールグラフで呼び出し回数を測定
  - システム全体の解析結果からソース・コードまで追跡
  - アプリケーション、DLL、デバイスドライバ、等を解析

VTune Amplifier

- VTune アンプリファイヤ を構成するモジュール
  - モジュール表示／ソース表示
  - サンプリング
    - タイムベース
    - イベントベース
  - コールグラフ
  - C/C++, Fortran, アセンブラ, Javaをサポート
  - リモート・エージェント (Windows*／Linux*)

サンプリング

アプリケーションの実行情報を取得するため、時々プロセッサに割り込みをかける

- 「時々」とは、「定期的」の意味である。たとえば、1000サンプル／秒は次のもので示される:
  - OS タイマー・サービス(NMI or RTC)
  - プロセッサのN個のClocktickイベント毎
- プロセッサが持つイベント・カウンタ・レジスタはサンプリングをトリガすることができる(interrupts). これをEvent Based Sampling (EBS)と呼ぶ.
  - これらのイベントはプロセッサ特有である: L2 Cache Misses, Branch Mispredictions, Floating-point instructions retired, etc.

サンプリングを用いたより詳細な解析

- タイムベース・サンプリング（TBS）
  - システム上の全てのソフトウェアについてサンプリングを行う
  - OS，デバイスドライバ、アプリケーションソフトウェア、JITでコンパイルしたJavaアプリケーション
  - 定期的な割り込みにより、ドライバが1ms（デフォルト）毎にサンプリング
- イベントベース・サンプリング（EBS）
  - キャッシュ・ミスや分岐予測ミス等のプロセッサ・イベントによるパフォーマンスの問題点を明らかにする
  - 一般的なプロセッサイベント
  - データキャッシュアクセスミスの回数／予測ミスした分岐の回数…
  - 同時に複数のイベントに関するサンプリングが可能

サンプリングにおける3つの注意点

- コードの変更する必要がない
  - No Instrumentation Required
  - BUT do compile with symbols & line numbers
  - BUT do make Release builds with optimizations.
- システム全体のサンプリング
  - Not Just YOUR Application
  - Can even sample drivers in ring zero
- オーバーヘッドが少ない＝サンプリングのために必要とする命令数が少ない
  - Minimal Intrusion Means High Validity
  - Overhead can be reduced further by turning off progress meters in the User Interface.

ソース表示機能

- ソース表示によって、コードの構造をわかりやすくする
- 3つのソースコードの表示方法
  - ファイル → “Open Static Module Viewer”
  - コールグラフ → ドラッグ・ドロップメニューから“View Source” を選択
  - サンプリング → ホットスポットをダブルクリックすることで、該当するプログラムが表示される
- ソースコードは、3つのモードで表示
  - ソースコードのみ
  - ソースコードとアセンブラコードのMIX表示
  - MIX表示にも Source/Combination/Execution のモードがある
  - アセンブラコードのみ
- ソースコードがない場合には、逆アセンブリ表示される

clock() 計測との違い

clock() 関数で取得したデータでは、時間計測を行っているコード範囲の処理時間しかわからない

ホットスポットは？ - (1)

```
start_time = clock();
for (i=0; i<num_steps; i++){
        x = (i + .5)*step;
        sum = sum + 4.0/(1.+ x*x);
}
pi = sum*step;
stop_time = clock();
```

19	for (i=0; i<num_steps; i++)		
20	{		
21	x = (i + .5)*step;	0.062s	0s
22	sum = sum + 4.0/(1.+ x*x);	3.913s	0s
23	}		
24	pi = sum*step;		

ボトルネックは？ - (2)

Elapsed Time: 4.084s

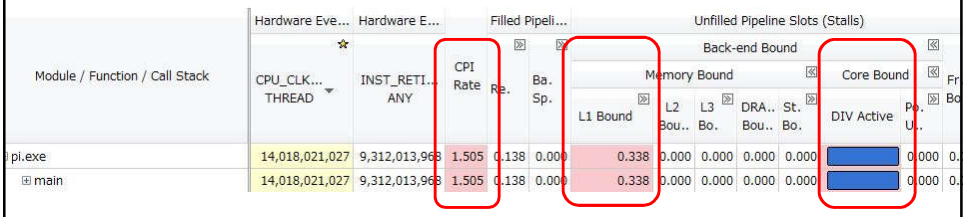
CPI Rate: 1.508
- Clockticks: 14,066,021,099
- Instructions Retired: 9,330,013,995

Filled Pipeline Slots
- **Retiring:** 0.138
- **Bad Speculation:** 0.000

Unfilled Pipeline Slots (Stalls)
- **Back-end Bound:** 0.860
  - DIV Active: 1.000
    The DIV unit is active for a significant portion of execution time. Locate the hot long-latency operation(s) and try to eliminate them. For example, if dividing by a constant, consider replacing the divide by a product of the inverse of the constant. If dividing an integer, see whether it is possible to...

DIV 実行ユニットが非常に長時間アクティブである。

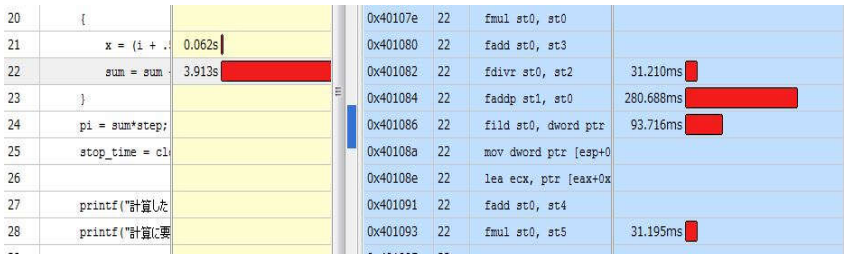
ボトルネックは？ - (3)



Module / Function / Call Stack	CPU_CLK... THREAD	INST_RETI... ANY	CPI Rate	Re.	Ba. Sp.	Back-end Bound / Memory Bound / L1 Bound	L2 Bou..	L3 Bo.	DRA.. Bou..	St. Bo.	Core Bound / DIV Active	Po. U..	Fr Bo
pi.exe	14,018,021,027	9,312,013,963	1.505	0.138	0.000	0.338	0.000	0.000	0.000	0.000		0.000	0.
main	14,018,021,027	9,312,013,963	1.505	0.138	0.000	0.338	0.000	0.000	0.000	0.000		0.000	0.

- CPI レートが高い
- L1 Bound が発生している
- DIV Active がレポートされている

ボトルネックは？ - (4)



ソースコードの 21 行目と 22 行目を VTune Amplifier XE でアセンブリー表示してみる。この 2 つの行のアセンブリー表示を見ると、まず浮動小数点演算にSIMD 命令が利用されておらず、旧式の x87 命令が利用されていることがわかる。fadd や fmul など f で始まる命令は、x87 命令

性能評価の指標

CPI 値は、基本的なパフォーマンス評価基準
（Clock per Instruction Retired - リタイアした命令ごとのサイクル数）

1 命令の完了に何クロック要したか

最新のスーパースカラー・プロセッサーは、1 サイクルあたり最大 4 命令を発行およびリタイアできるため、理論上最高の CPI は 0.25

アーキテクチャー	計算式
Coreマイクロアーキテクチャー	CPU_CLK_UNHALTED.CORE / INSTRUCTION_RETIRED.ANY
Nehalem SandyBridge IvyBridge	CPU_CLK_UNHALTED.THREAD / INSTRUCTION_RETIRED.ANY

CPI 値の算出

		CPU_CLK_UNHALTED.THREAD	INSTRUCTION_RETIRED.ANY
20	{		
21	x = (i + .5)*step;	228,000,342	76,000,114
22	sum = sum + 4.0/(1.+ x*x);	13,800,020,700	9,224,013,836
23	}		

sum = sum + 4.0/(1. + x*x) の行にかかる
「CPU_CLK_UNHALTED.THREAD」= 13,800,020,700 を
「INSTRUCTION_RETIRED.ANY」= 9,224,013,836 で割ると、
1 つの命令にかかる平均クロック数（CPI）が解ります。
ここでは CPI は、およそ 1.49

ツールの入手方法

http://software.intel.com/en-us/articles/intel-software-evaluation-center/

Intel VTune Amplifier XE (Windows , Linux)
Intel C++ Composer XE (Windows, Linux)

登録後 1 か月利用可能

まとめ

性能最適化サイクル:

- パフォーマンス・データの収集
- データの解析と問題の認識
- 問題の解決
- 拡張の実装
- 結果のテスト

インテル® ソフトウェア開発ツール

最適化に関する注意事項

インテル® コンパイラーは、互換マイクロプロセッサー向けには、インテル製マイクロプロセッサー向けと同等レベルの最適化が行われない可能性があります。これには、インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3)、ストリーミング SIMD 拡張命令 3 補足命令 (SSSE3) 命令セットに関連する最適化およびその他の最適化が含まれます。インテルでは、インテル製ではないマイクロプロセッサーに対して、最適化の提供、機能、効果を保証していません。本製品のマイクロプロセッサー固有の最適化は、インテル製マイクロプロセッサーでの使用を目的としています。インテル® マイクロアーキテクチャーに非固有の特定の最適化は、インテル製マイクロプロセッサー向けに予約されています。この注意事項の適用対象である特定の命令セットの詳細は、該当する製品のユーザー・リファレンス・ガイドを参照してください。

改訂 #20110804