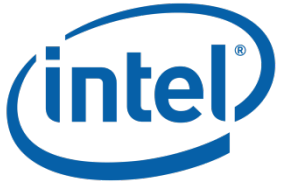


インテル® マイクロアーキテクチャー Nehalem† ソフトウェア最適化

† 「Nehalem」の開発コード名で呼ばれていた新しいインテル® マイクロアーキテクチャー

インテル株式会社
ソフトウェア＆サービス統括部

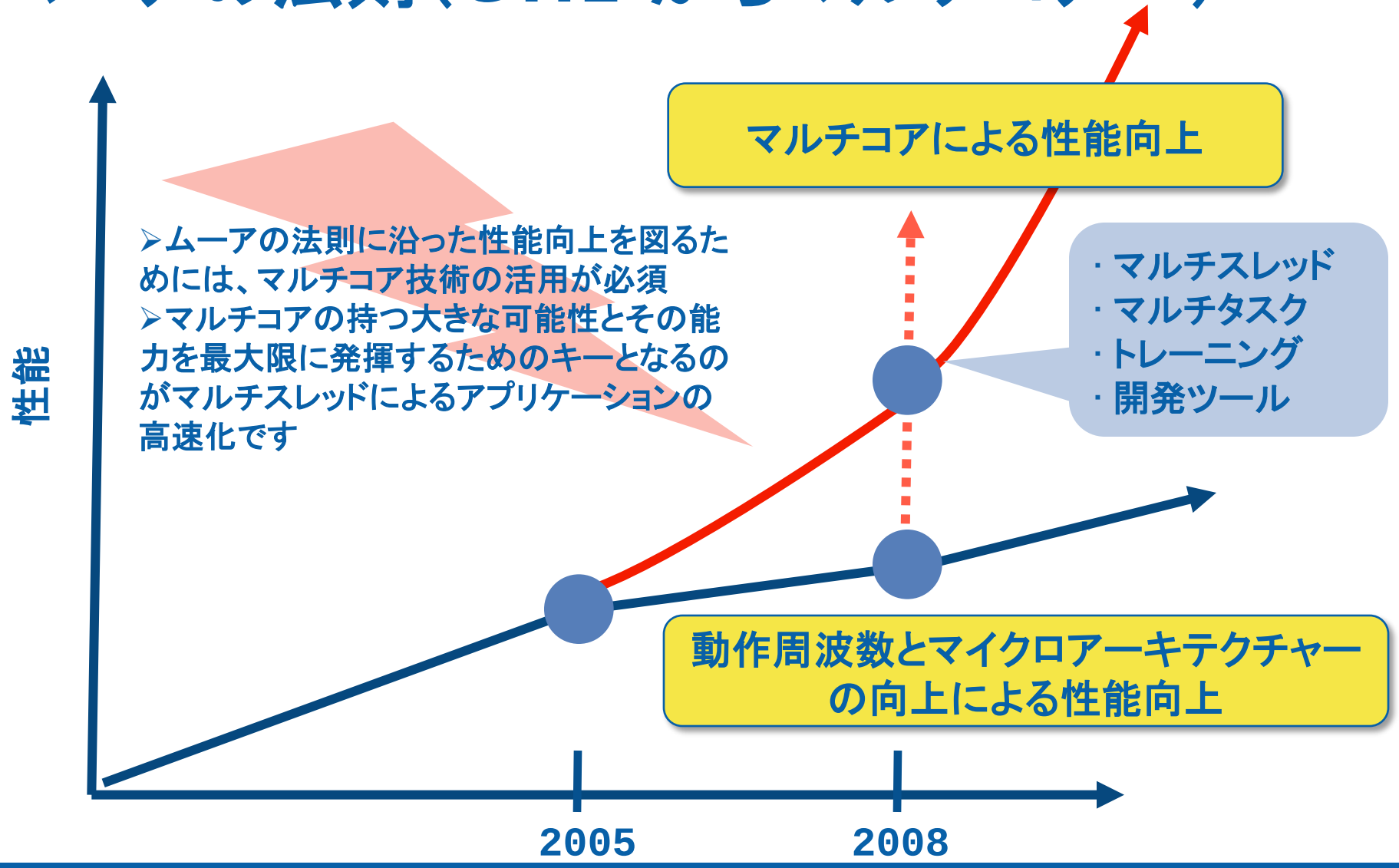
内容

- ソフトウェア最適化と並列化のおさらい
- インテル® マイクロアーキテクチャー Nehalem の特徴と最適化
- 今後のプログラミング環境と開発ツール

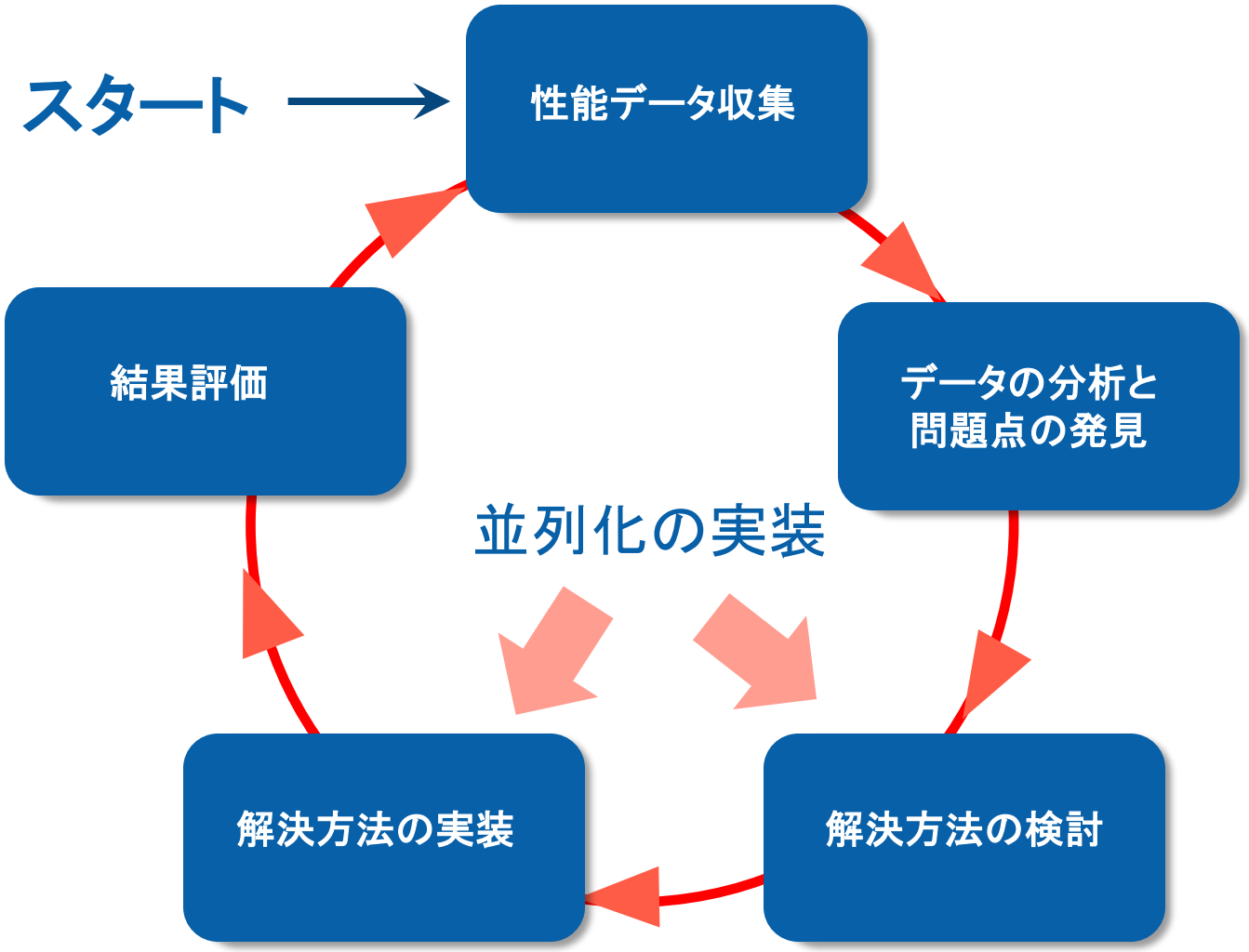
ソフトウェア最適化のおさらい

- 性能最適化
  - 命令レベルの並列性（**ILP**）
  - 並列計算（プロセスとスレッド）

ムーアの法則（GHz からマルチコアへ）



性能最適化手法



従来の最適化

SSE/SSE2/SSE3を使用して命令レベルの並列化(ILP)を高める

SSE3は水平方向のSIMD化が可能

HADDPS: Packed Single-FP Horizontal Add (Continued)

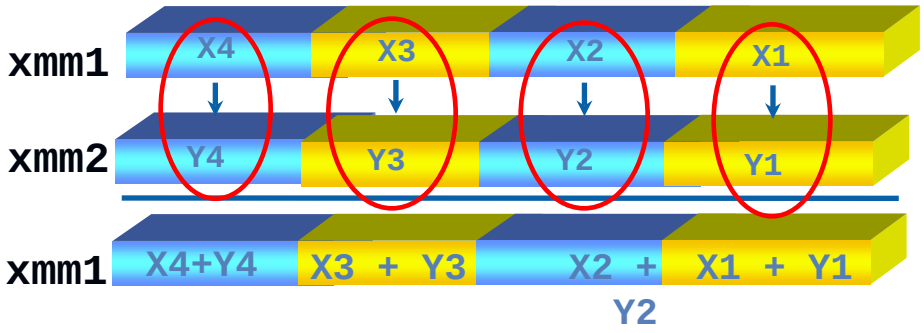
Operation

```
xmm1[31-0] = xmm1[31-0] + xmm1[63-32];
xmm1[63-32] = xmm1[95-64] + xmm1[127-96];
xmm1[95-64] = xmm2/m128[31-0] + xmm2/m128[63-32];
xmm1[127-96] = xmm2/m128[95-64] + xmm2/m128[127-96];
```

Intel C/C++ Compiler Intrinsic Equivalent

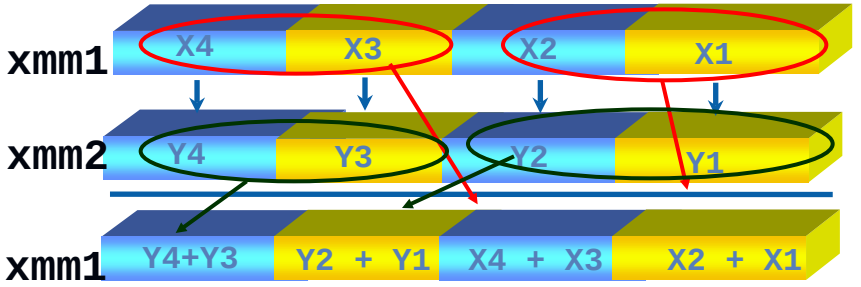
```
HADDPS __m128 _mm_hadd_ps(__m128 a, __m128 b)
```

例： ADDPS xmm1, xmm2



ADDSD: __m128 _mm_add_ps(__m128 a, __m128 b)

例：SSE3 HADDPS xmm1, xmm2



HADDPS __m128 _mm_hadd_ps(__m128 a, __m128 b)

デフォルトで SSE2 によるベクトル化が有効に

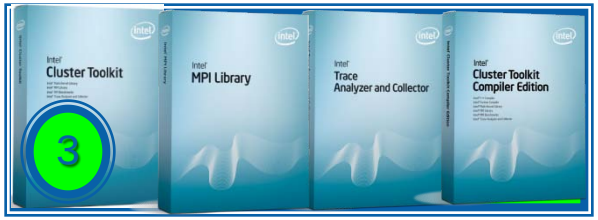
インテル® コンパイラー 11.0 プロフェッショナル・エディション

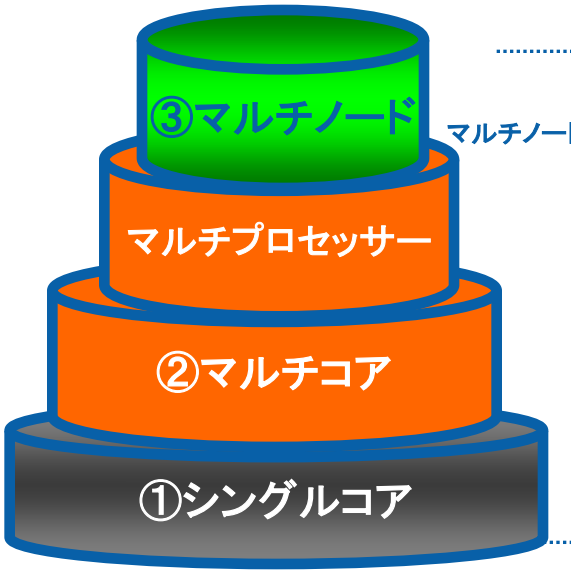
–msse2 （Windows* では /arch:sse2）スイッチがデフォルトに

（同等のオプション –xSSE2 / –xW と /QxSSE2 / QxW）

- **インテル® 64 対応の 10.0 コンパイラーではすでにデフォルト**
  - **すべてのインテル® 64 システムは SSE2 命令が実行可能であるため**
- **IA-32 において –mia32（Linux*）、/arch:IA32（Windows*）オプションで SSE コードの生成を抑止できる**
- **注：-no-vec（Linux）と /Qvec-（Windows）オプションはベクトル化（パックド SSE）を無効にするが、スカラー SSE コードは生成される**
- **Mac OS* X では IA-32 で –xsse3が、インテル® 64 では –xssse3 がデフォルト（同等オプションは –xP と –xT）**

プラットフォームとプログラミング階層

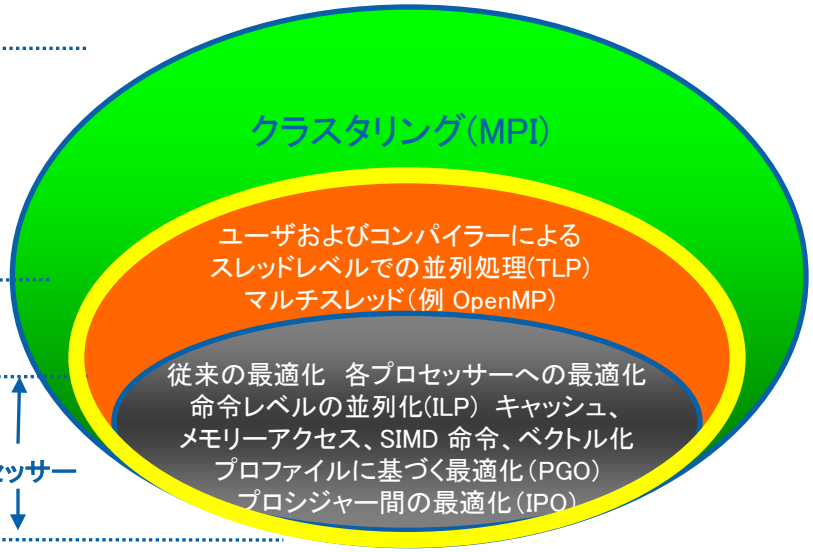


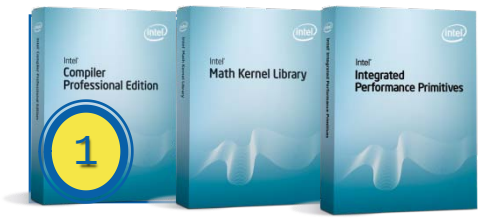


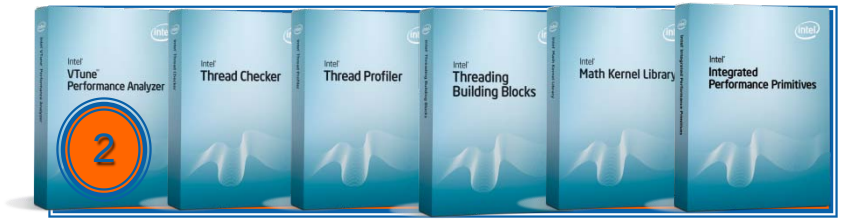
マルチノード、 クラスター

シングルノード （SMP）

シングルプロセッサー







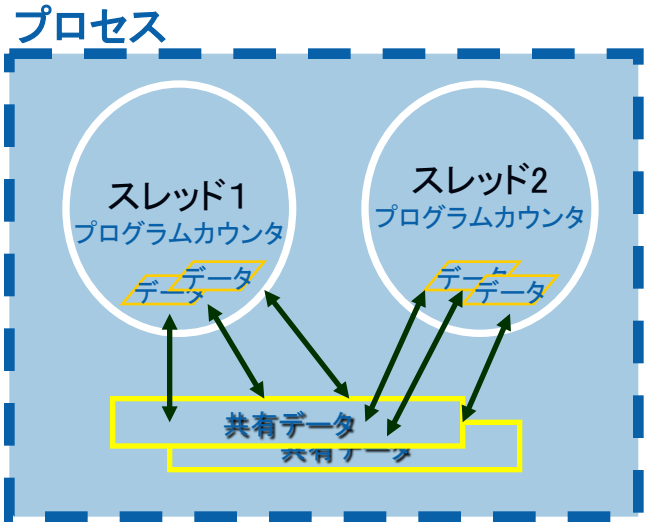
並列計算

- 特定の問題に対して、複数のプロセッサーを利用して高速に処理する並列処理
- 対象となる問題を複数のコア、プロセッサーを同時に利用して短時間で解く
- アプリケーションを作成する際に、プログラム中で複数のスレッドを用いてアルゴリズムを作成する手法をここではマルチスレッド・プログラミングと呼ぶ
- Windows や他の OS の API を用いたプログラミングや OpenMP などの抽象化技術の概念とインテル開発ツール

スレッド　vs. プロセス

スレッド

- スレッド間では仮想アドレス空間を共有
- レジスターやスタックなどは独立
- スレッド生成のオーバーヘッドは軽いが共有メモリーの書き込みに同期が必要
- SMP計算機CPUのイメージ

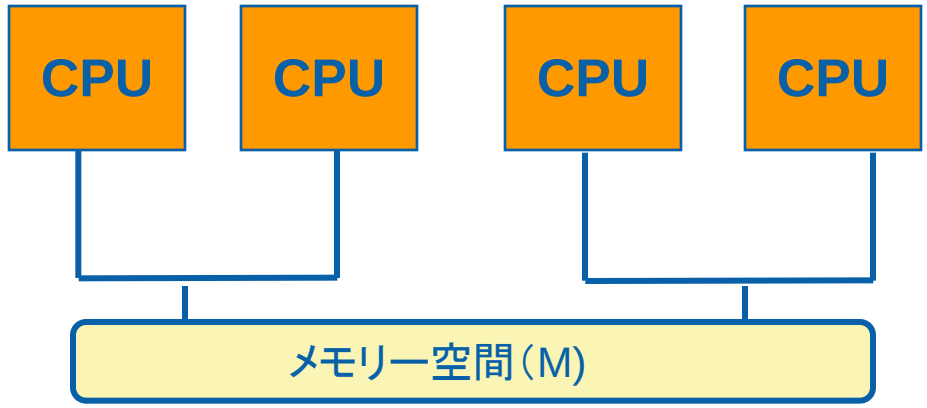


プロセス

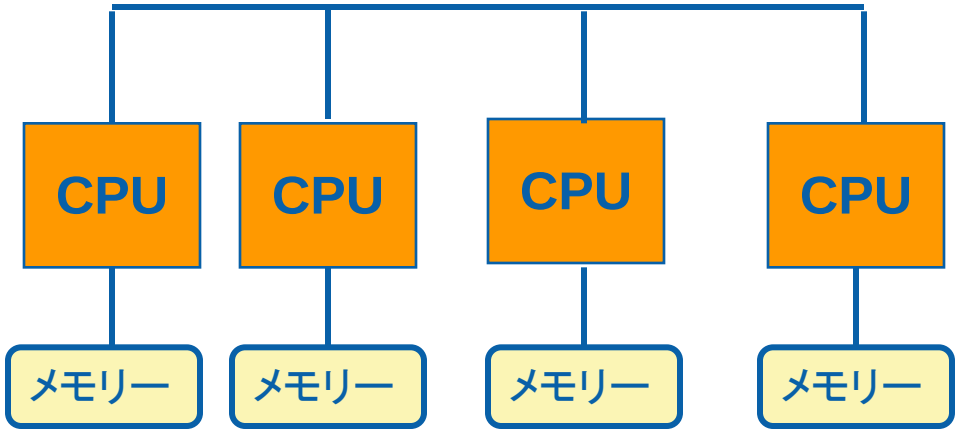
- 各プロセスは独立した仮想アドレス空間とスタックをもち、1つ以上のスレッドから構成される
- プロセス間での同期や共有にはプロセス間通信を行なう
- 保護が容易だがオーバーヘッドが重い
- HPCクラスターの計算機のイメージ

スレッドとプロセスのイメージ

スレッド

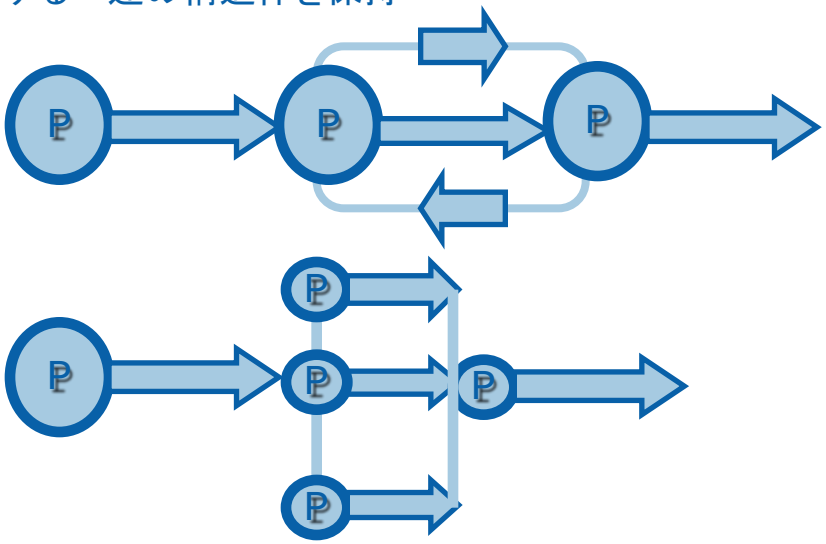


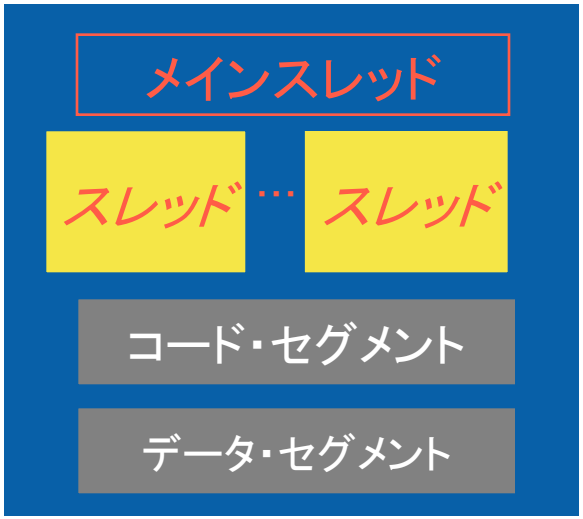
プロセス



マルチスレッド・プログラミングについて

- プログラム中で複数のスレッドを用いてアルゴリズムを構成
  - OSはプロセス単位でプログラムをロード
  - プロセスはエントリーポイントをメインスレッドとして実行
  - スレッドはプロセス内で他の(子)スレッドを生成できる
  - プロセス内のすべてのスレッドはコード、データを共有
  - スレッドは2つのスケジューラをもつ (active, inactive)
  - OSは論理プロセッサー上で現在実行しているスレッドのスケジュール管理
  - * スレッドとはOSがプロセッサー時間を割り当てる対象の基本単位。各スレッドは例外ハンドラ、スケジューリングの優先順位、およびコンテキストをスケジュールされるまでの間保存するために使用する一連の構造体を保持





並列化プログラミングモデル

データ分割（Data Parallelism）

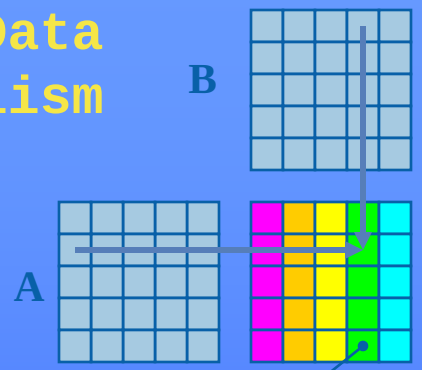
- 科学技術計算、マルチメディアアプリケーション等演算する領域を分割してそれぞれのスレッドに割り当てる
- 各スレッドが自分の割り当て分を処理する

機能分割（Function Parallelism）

- 処理を同程度の時間を要する複数の小さなサブ処理に分割してそれぞれのスレッドに割り当てる
- データがスレッド間を流れながらパイプライン処理される

並列化プログラミングモデル

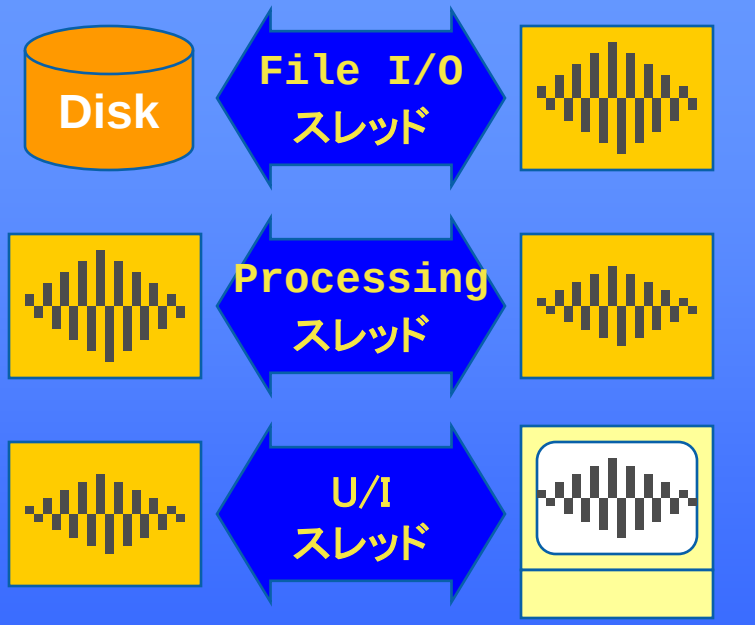
Data Parallelism



異なるデータを独立して同時に演算

```
for (i=0; i<M; i++)
   for (k=0; k<L; k++)
      for (j=0; j<N; j++)
         C(i,j) +=
            A(i,k)*B(k,j);
```

Functional Parallelism

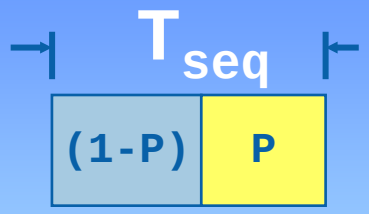


異なる機能を独立そて同時に実行

アプリケーションに応じた適切なスレッド化が重要

並列化計算の問題点: アムダールの法則

並列化のスピードアップ(スケーリング)の上限を説明　オーバーヘッドの影響を考慮する際に役立つ

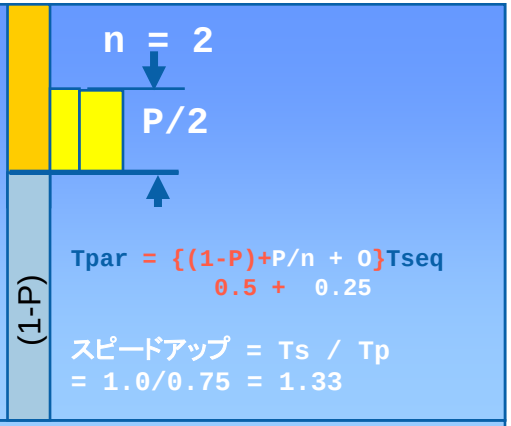


$T_{par} = ((1-P)+P/n+O)\ T_{seq}$

P = 並列化可能な領域の割合

n = プロセッサ数

O = オーバーヘッド



並列化効率 = 1/ (P + n(1-P))

スピードアップ = 並列化効率 * n = 1 / ((1-P) + P/n)

並列化率(p) = 並列化可能部分のシングルスレッドでの実行時間 / シングルスレッドでの実行時間

SMP 上でパフォーマンスを得るためには並列化の領域を増やすことが重要

- 80% 並列化できれば性能は 1.66 倍に (n=2)

オーバーヘッドにも注意

- スレッドの生成、スレッドの同期 - 適切な同期方法と位置を考える

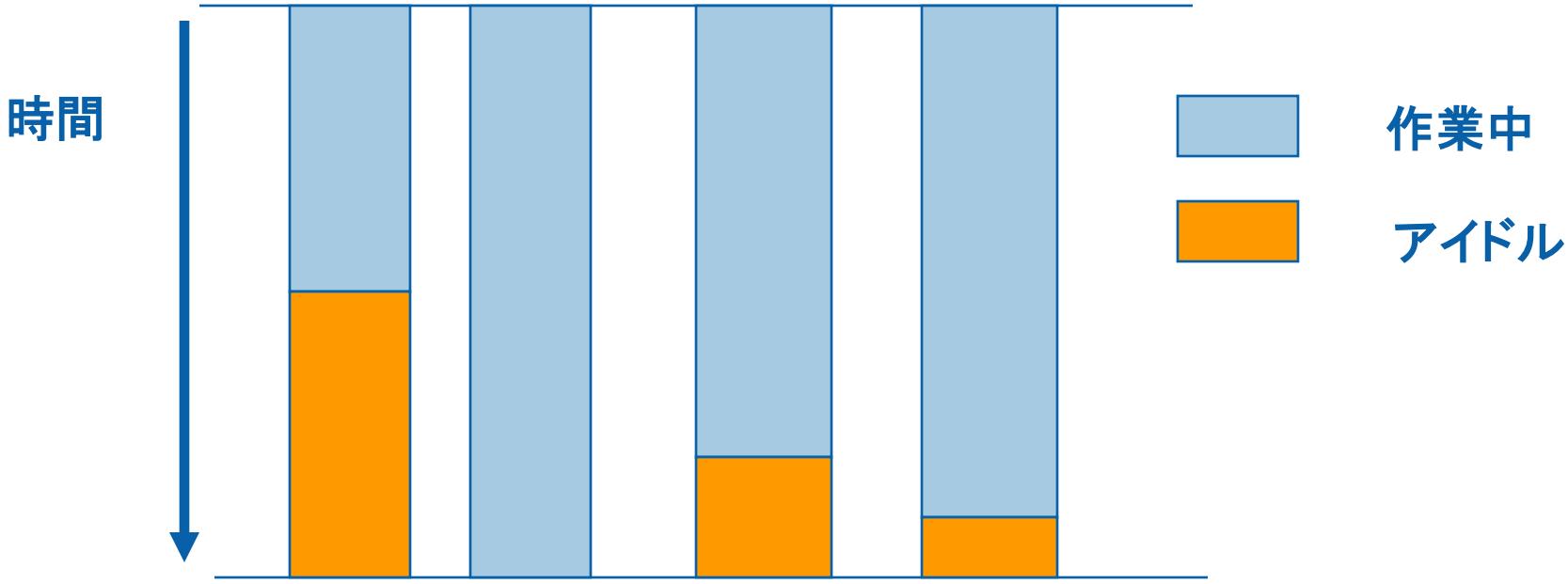
並列計算の問題点　ロード・バランシング

負荷の不均衡

ロード・バランシング：　並列コンピューターのプロセッサー（コア）間の作業の分配

最も時間のかかる作業が終わるまで全体の作業は完了しない

仕事量が等しくないとアイドル・スレッドが生じて実行時間が無駄になる



パフォーマンスを達成するにはスレッドすべて均等に負荷を与えることが重要

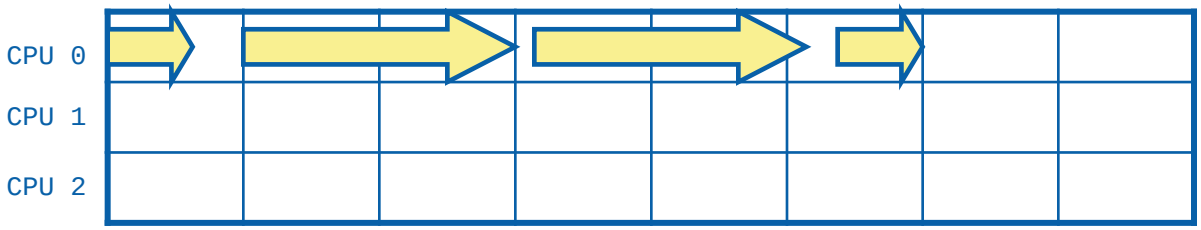
並列計算の問題点　粒度

細粒度と疎粒度

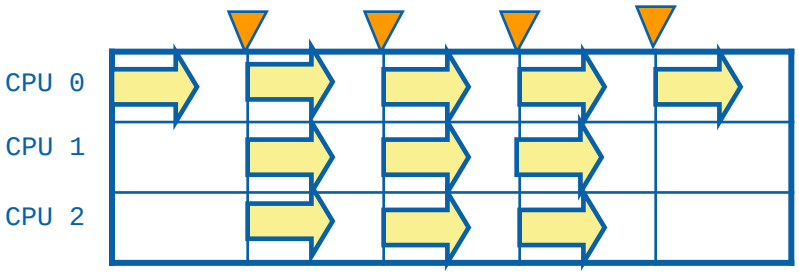
粒度（Granularity）とは1つのスレッド並列実行を行う**仕事の分割単位**

仕事（タスク）をどのように分散するかまた同期処理も重要

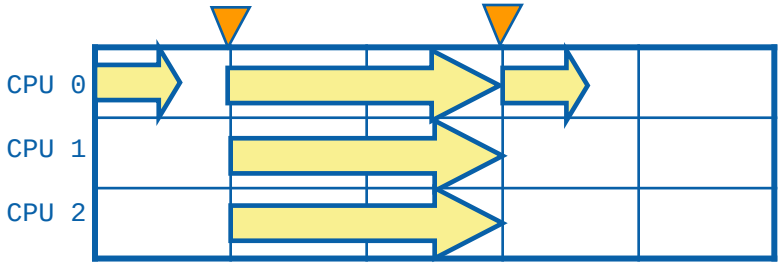
遂次処理



並列処理: 細粒度での並列処理



並列処理: 疎粒度での並列処理



同期(シンクロナイゼーション)について

- スレッディングの検討段階でできるだけ独立性(＝並列性)を高め、同期は最小限にする
- 可能な限り高速な同期機能を使う
- スレッドでのプログラムの同期: タスク B の開始前にタスク A の処理が終わることを保障する

同期処理機構

- バリアー　:スレッドはすべてのスレッドがバリアーに進むまで休止
- Windowsの同期オブジェクト

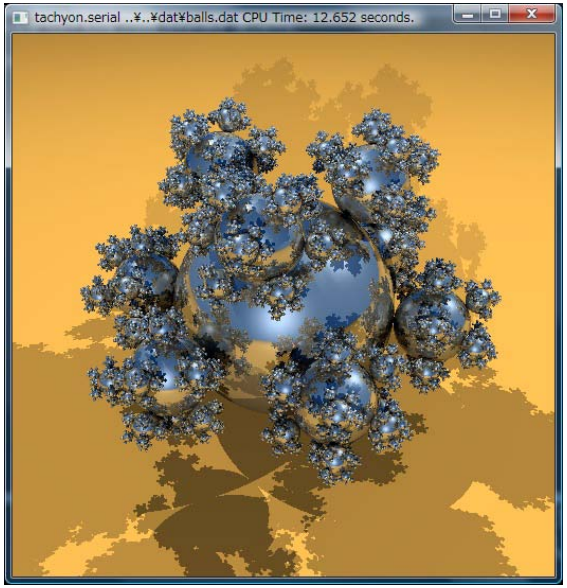
名前	速度	プロセス間	リソース カウント 機能	性質
Critical Section	高速	なし	なし	ユーザーモード
Mutex	低速	あり	なし	カーネルオブジェクト
Semaphore	低速	あり	あり	カーネルオブジェクト
Event	やや低速	あり	名前付きを使えば可能	カーネルオブジェクト

※ Interlocked API – increment, decrement, ExchangePointer　　高速

Demo

2-D レイトレーシング/レンダー アプリケーション

インテル® スレッディング・ビルディング・ブロック
に含まれるサンプルプログラム



1. コンパイラーオプションによる最適化
2. OpenMP を使用した並列化
3. インテル® TBB を使用した並列化

並列計算の問題点：
並列化における性能劣化の原因

並列化により、逆に性能が劣化した場合の原因

- スレッド化によるオーバヘッド（短いループやコンパイラーが追加するコード）
- データ依存関係
- 並列ループに対する「従来の最適化」の影響
- 不均等な負荷
- メモリーアクセス（メモリーのアクセス、キャッシュのアクセスの方法が変わる）
- 共有アドレスへの同時アクセス
- フォールス・シェアリング

アプリケーションをスレッド化するには設計、
デバッグ、およびパフォーマンスのチューニングの繰り返しが必要

キャッシュ・コヒーレンシー

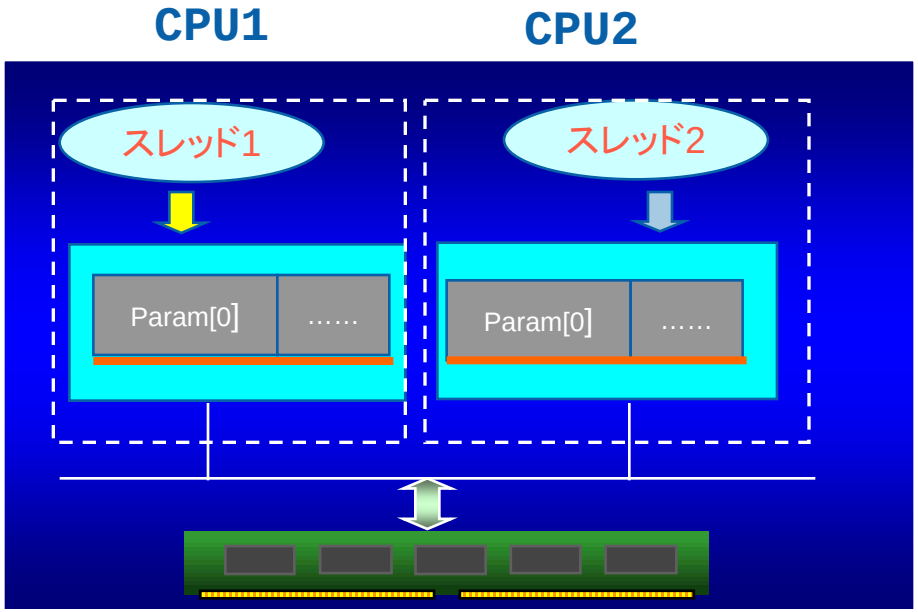
Cache Coherency　データが失われたり、上書きされないようにキャッシュを管理

2 つのスレッド（プロセッサー）はメモリーを共有する

- 変数はメモリー内、または両方のプロセッサー内のキャッシュ内に存在することがある
- キャッシュラインのすべてのコピーは同じデータでなくてはならない…
  - Modified- Exclusive –Shared – Invalid（MESI プロトコル）

Coherence キャッシュ問題

- 各々のスレッドが同一のキャッシュ・ラインにアクセスしたとき



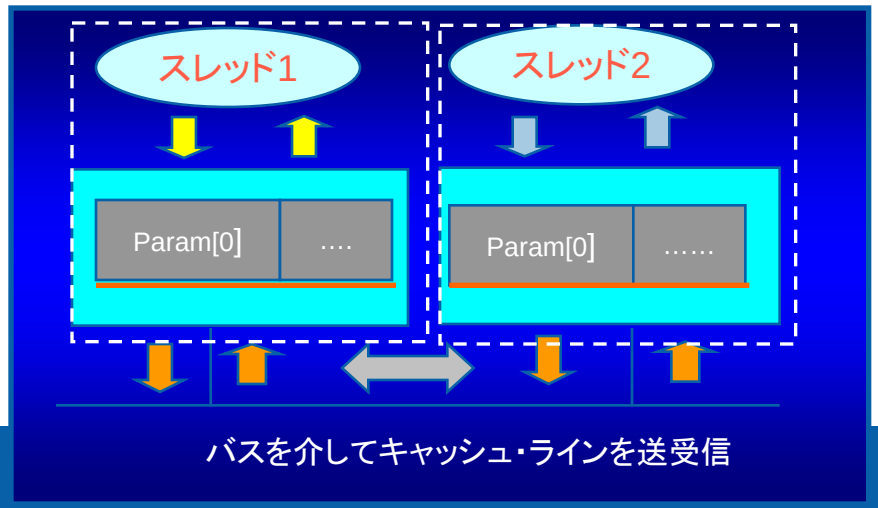
フォールス・シェアリングについて

マルチプロセッサー上の問題

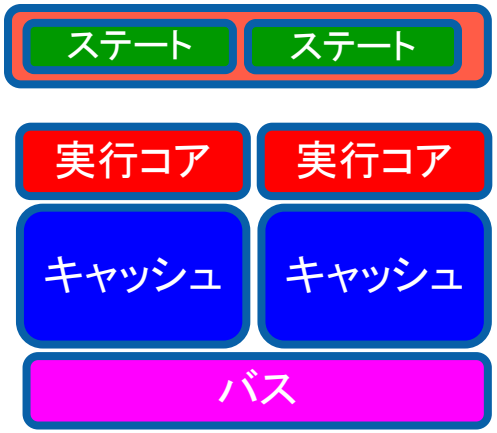
- クアッドコア、デュアルコア、ハイパースレッド上でも同様に問題

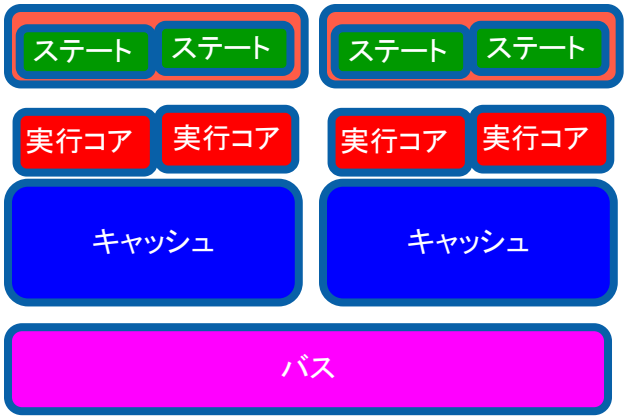
対策

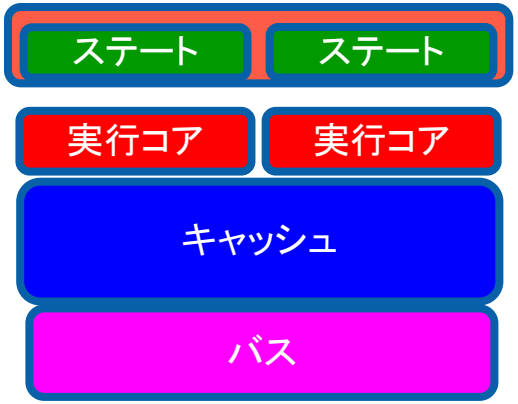
- VTune™ パフォーマンス・アナライザーのイベントを参照する
- デュアルコア　ハイパースレッド上で数百倍のイベントが検出された場合は疑いが強い
- 関数レベルで6-7倍、アプリケーションレベルで2倍のパフォーマンスの低下をもたらすことがある

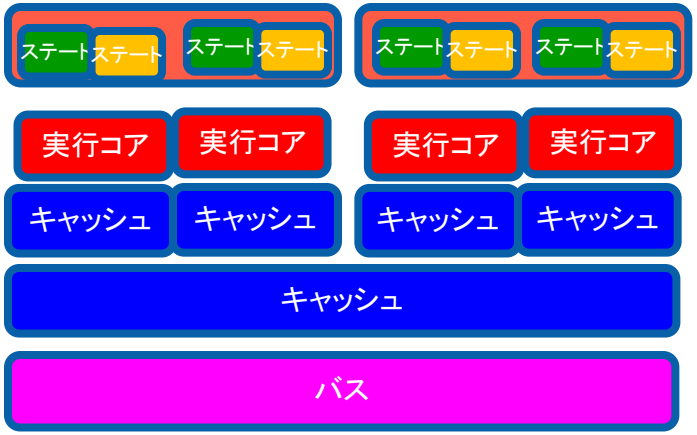


2005年以降のプロセッサー









内容

- ソフトウェア最適化と並列化のおさらい
- インテル® マイクロアーキテクチャー Nehalem の特徴と最適化
- 今後のプログラミング環境と開発ツール

インテル® Core™ マイクロアーキテクチャーの概要

ワイド・ダイナミック・エグゼキューション

- 最大4つまで同時実行　解析/リネーム/リタイア

アドバンスト・デジタル・メディア・ブースト

- 128ビットSIMD拡張命令(SSE)の実行

Intel® HD Boost

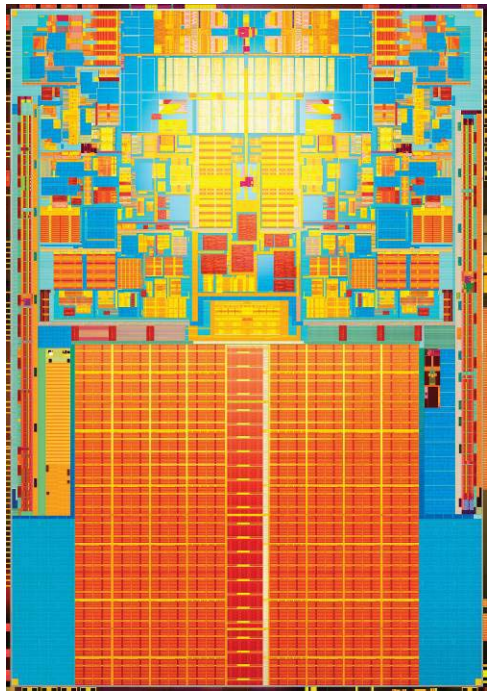
- 新たにSSE4.1命令セットから搭載

スマート・メモリー・アクセス

- メモリー・ディスアンビグエーション(メモリーの曖昧性解消)
- ハードウェア・プリフェッチ

アドバンスト・スマート・キャッシュ

- 2次キャッシュを共有することで待ち時間を短くし、キャッシュ領域を効率的に使用



Nehalem* はインテル® Core™ マイクロアーキテクチャーを基盤に設計

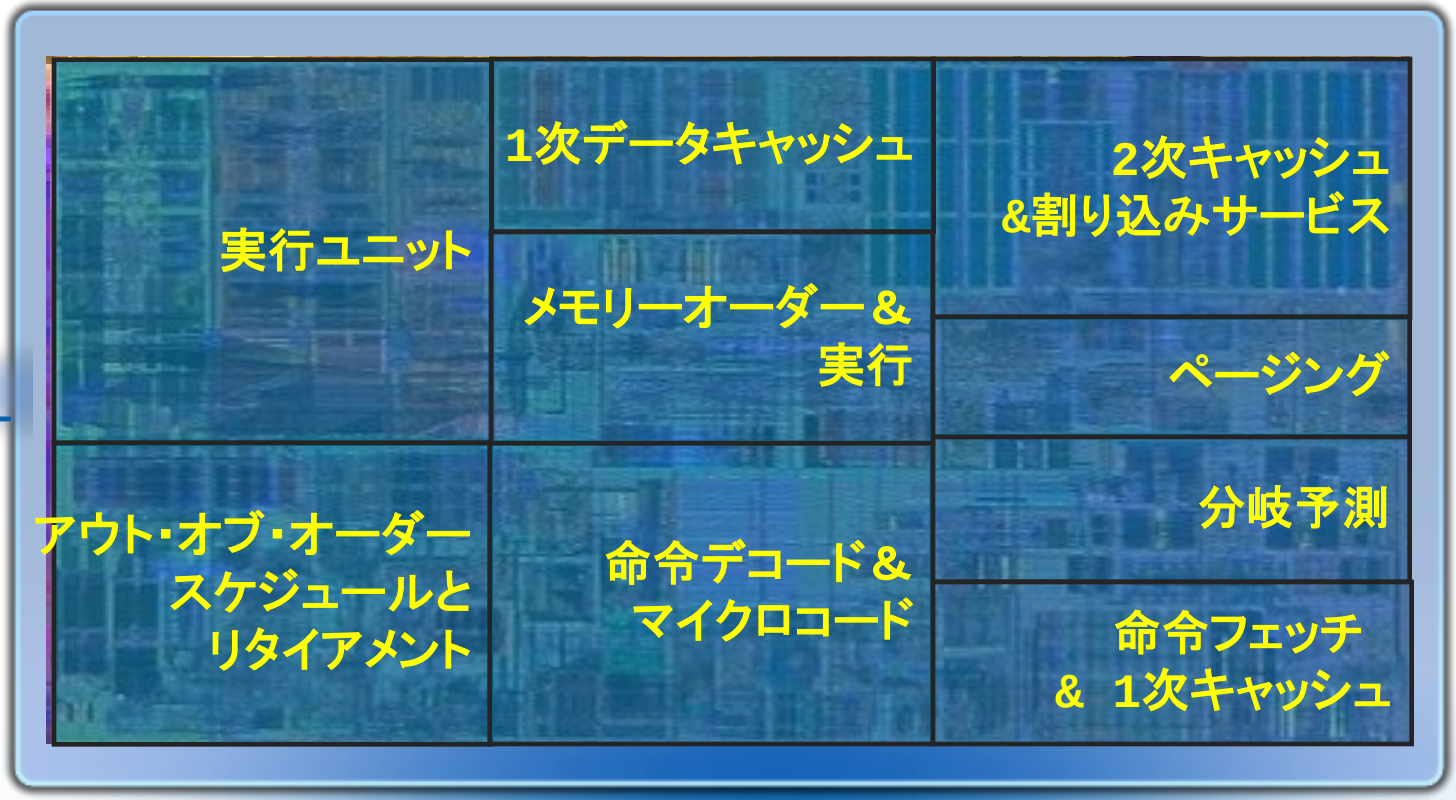
* 開発コード名

優れた性能を発揮する設計

新しいSSE4.2命令

ロック機能の改善

新たなキャッシュ階層



より深い
バッファー

ループ
ストリーム
の改善

インテル® ハイパー・
スレッディング・テクノロジー

仮想化の高速

より優れた分岐予測

インテル® ハイパー・スレッディング・テクノロジー

インテル® ハイパー・スレッディング・テクノロジー

- 各コアで 2 つのスレッドが同時に起動

最大4つまで可能な実行命令を最大限に活用

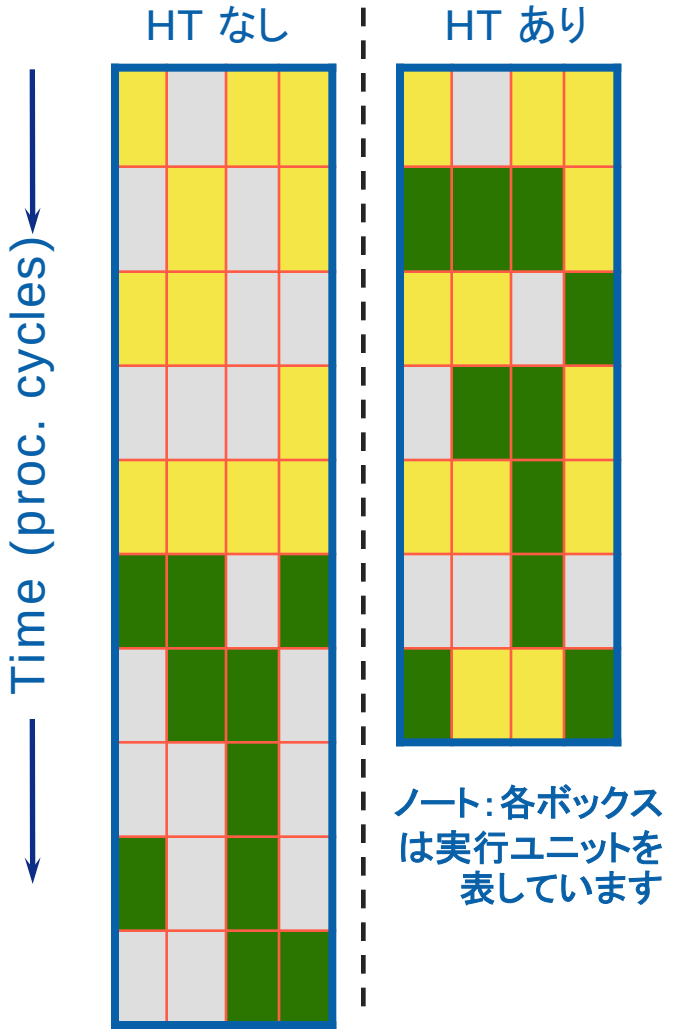
- 複数のスレッドを同時実行
- シングルスレッドでのレイテンシーを隠蔽

電力効率とパフォーマンスの向上

- ダイあたりのコストを抑える
- アプリケーションに応じて更なる性能を発揮
- コアを増やすより効率的

Nehalem* の利点

- より大容量のキャッシュ
- メモリー帯域幅の拡大



同時マルチスレッド機能は、パフォーマンスの向上とエネルギー効率向上に役立つ

* 開発コード名

ハイパースレッディングの強化

アーキテクチャー	インテル® アーキテクチャー Nehalem	Intel NetBurst® アーキテクチャー
CPUID	06_1AH	0F_02H,0F_03H,0F_04H, 0F_06H
発行ポートおよび実行ユニット	3 つの発行ポート(0、1、5)をALU、SIMD、FPの各演算向けに分散	ポートのバランスが取れていない。高速ALU、SIMD、FP で同じポート 1 を共有
バッファリング	適度なパイプラインの深さを確保しながら、ROB、RS、フィルバッファーなどのエントリーを増加	バッファーエントリーとパイプラインの深さとのバランスレベルが低い
分岐予測とアライメントの合っていないメモリーアクセス	スペキュレーティブ・エグゼキューションが強化され、予測ミスの直後に再利用が可能。キャッシュ分割を効率的に処理	マイクロアーキテクチャー・ハザードが多く発生し、両方のスレッドのパイプラインがクリアーされる
キャッシュ階層	大容量かつ効率的	回避すべきマイクロアーキテクチャー・ハザードが多い
メモリーと帯域幅	NUMA、DDR3 に対してソケットごとに3チャネル、ソケットごとに最大32GB秒	SMP、FSB またはデュアル FSB、FSB ごとに最大12.8GB秒

インテル・スマート・キャッシュ – コア・キャッシュ

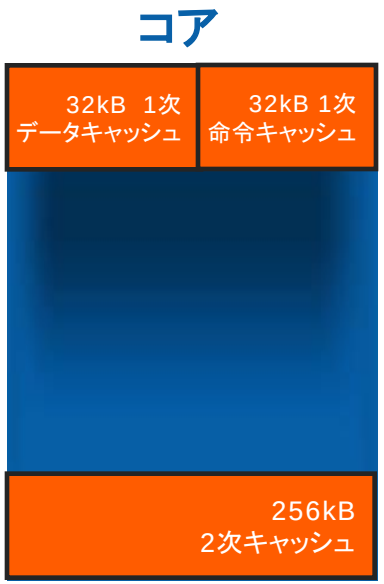
3段階に分かれたキャッシュ階層

1次キャッシュ (L1)

- 32kB 命令キャッシュ
- 32kB データキャッシュ
  - Core2 よりも 1 次キャッシュミスを最適化

2次キャッシュ (L2)

- 新たに Nehalem* から導入
- 統合キャッシュ（コードとデータ）
- コアごとに 256kB が割り当て
- 性能面: 待ち時間（=レイテンシー）を短く
- 拡張性: コア数が増えても、共有キャッシュへの負担を減らす（= アクセス数を軽減）



* 開発コード名

インテル・スマート・キャッシュ – 3次キャッシュ

3次キャッシュを新たに搭載 (L3)

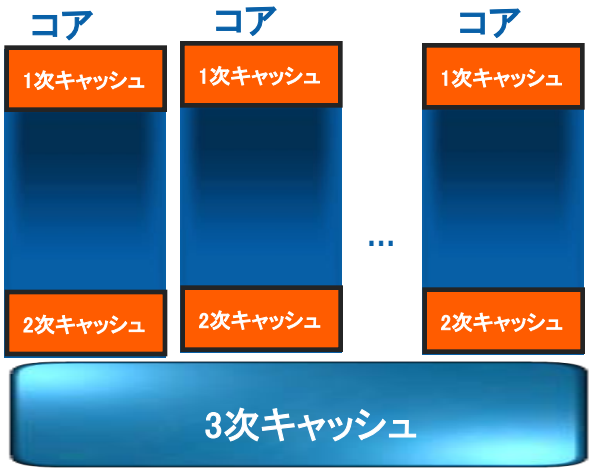
すべてのコアで共有

容量は CPU コア数によって増減

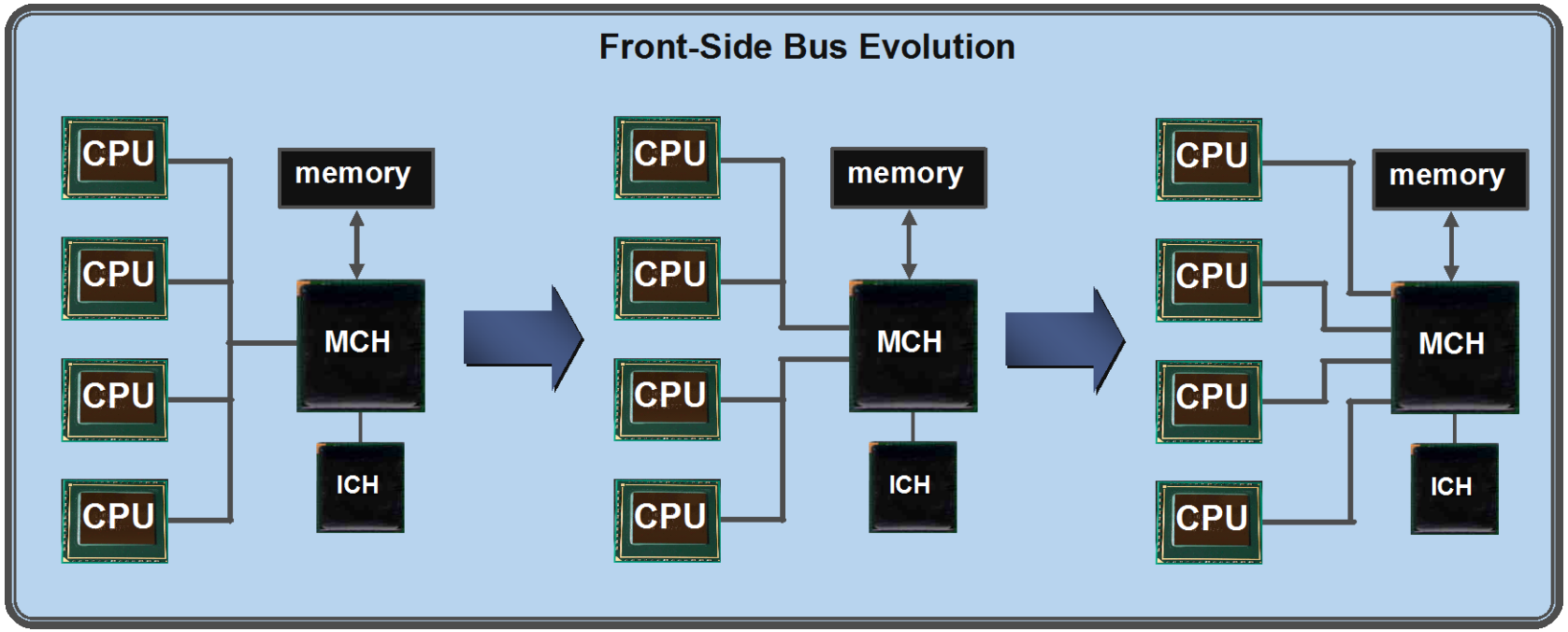
- クアッドコア: 最大 8 MB
- 拡張性:
  - さまざまな製品バリエーションに対応
  - 将来における拡張性

性能向上のためインクルーシブ・キャッシュ方式を採用

- L1/L2 内に存在するアドレスは 3 次キャッシュ内にも存在する



これまでのプラットフォーム・アーキテクチャー



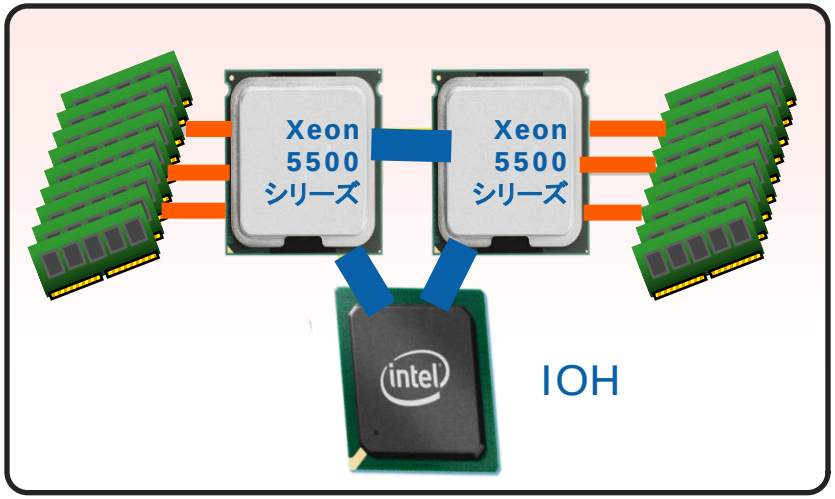
インテル® Xeon 5500 シリーズ プラットフォーム・アーキテクチャー

CPU メモリーコントローラーを内蔵

- ソケットあたり 3 チャンネル分の DDR3 メモリをサポート
- **メモリー帯域幅の増大**
- プロセッサー数に応じたメモリー帯域幅のスケーリング
- **メモリー・レイテンシーの削減**

インテル® QuickPath インターコネクト (インテル® QPI)

- ポイント・トゥー・ポイント接続
- ソケット間の接続
- ソケットとチップセットの接続
- 拡張性の高いソリューションを提供



新しいプラットフォームによる大幅な性能向上

* 開発コード名

共有メモリ型マルチプロセッサー
Non-Uniform Memory Access (NUMA)

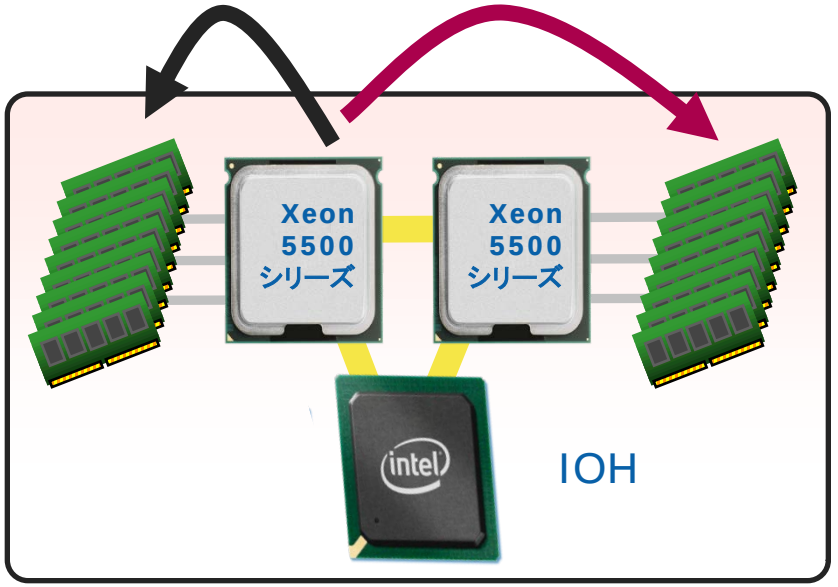
FSBアーキテクチャー
- 一箇所で全てのメモリーを制御

Xeon 5500 シリーズ世代
- 各プロセッサーが全ノードのメモリーを利用可能

メモリー・レイテンシーは、
各プロセッサーに左右される

「ローカル」メモリー
- 帯域幅がいちばん大きい
- 待ち時間（レイテンシー）が短い

「リモート」メモリー
- 待ち時間（レイテンシー）が長い



ソフトウェア対応が NUMA に適した性能を実現

パフォーマンス計測の複雑さ

- インテル® Core™ i7 プロセッサーにはパフォーマンス計測に影響する 2 つの機能がある:
  - インテル® ハイパースレッド・テクノロジー
  - インテル® ターボブースト・テクノロジー
- これらの機能が有効であると、パフォーマンス・データの計測と結果解釈はより複雑になる
  - 大部分のカウンターはスレッドごとにカウントされ、いくつかのイベントはコアにカウントされる
    - 各イベントの詳細については VTune のヘルプを参照
- パフォーマンス解析に際しこれらの機能を無効にし、最適化後有効にする方法がある

ソフトウェア最適化の手順

1. ホットスポットを識別する
   - ホットスポットの効率を測定する
     - 非効率であれば、プロセッサーのアーキテクチャー上の原因を特定する
2. 問題を解決する
3. 手順 1 に戻る

ステップ 1）ホットスポットを識別する

ホットスポットとは？： アプリケーションが多くの処理時間を費やしている場所

なぜ時間を費やしているか？： ボトルネックを含む可能性があり、最適化の対象となる

- アプリケーションの実行時間の多くを占めている原因はなにか？

どのように見つけるか？： VTune™ アナライザーのベーシック・サンプリング・ウィザードを利用する

- 通常ホットスポットは、CPU_CLK_UNHALTED.THREAD イベントに置き換えて定義される（通称：clockticks）

ステップ 2) 効率を測定する

次の 3 つの方法のいずれかを利用してホットスポットの効率を測定する:

- 実行ストールの割合
- CPI（clock per instruction retired）の変化
- コードの検証

効率測定 1: 実行ストールの割合（%）

何を示すのか？: アプリケーションがプロセッサーをどの程度効率よく利用しているか判断できる

どのように求める？: (UOPS_EXECUTED.CORE_STALL_CYCLES / (UOPS_EXECUTED.CORE_ACTIVE_CYCLES + UOPS_EXECUTED.CORE_STALL_CYCLES)) * 100

対処は？:

- < 10% 以下のストールサイクルは妥当
  コードの削減に注目する
- 10% - 50% のストールサイクルを持つクライアント・アプリケーション
  ストールの原因を調査して排除が必要
- 50% - 80% のストールサイクルを持つサーバー・アプリケーション
  ストールの原因を調査して排除が必要

効率測定 2: 命令がリタイアするサイクル(CPI)の変化

なぜ? : 2 度の実行を比較し他の効率を計測することは有効

- ワークロードの 1 つを実行した平均時間を知る

どのように求める? :
CPU_CLK_UNHALTED.THREAD / INST_RETIRED.ANY
(ホットスポット表示で自動的に計算される)

対処は? :

- CPI 値はアプリケーションとプラットフォーム全体に影響されるためばらつきがある
- コードサイズが不変であるならば、CPI 値を減らすことに集中する

効率測定 3: コードの調査

なぜ行うか？: 手順 1 と 2 では処理時間を計測したが、もう 1 つ注目する効率の測定方法に実行されている命令数に注目する方法がある

どのように行う？: VTune アナライザーの逆アセンブリー表示とソース表示を利用する

対処は？:

- より高度な命令を利用してコードサイズを減らす
- 次の 2 枚のスライドを参照

コードの問題を調査 1: 浮動小数点演算に SSE 命令を利用していない場合

なぜ: インテル® ストリーミング SIMD 拡張命令（SSE、SSE2、SSE3、SSSE3、SSE4.1、SSE4.2）のパラレル演算機能を利用すると、浮動小数点演算の性能を大幅に改善できる可能性がある。コンパイラーやライブラリーはデフォルトでは最新の命令セットは利用しない

どのように行う？: ソースコード表示にドリルダウンして、浮動小数点演算を行うループを探し、アセンブリー表示に切り替える

対処は？:

xmm レジスターを使用していても p（パックド）命令を利用していない場所を探す

適切なパックド SSE命令		遅い命令	
addps	xmm4,xmm5	faddp	st(0),st(1)
mulpd	xmm4,[rax]	fmul	[eax]
movups*	xmm4,[eax]	addss+	xmm4,xmm5

- この問題の解決方法は次のスライドと同じ

コードの問題を調査 2: インテル® SSE4.2 命令を利用していない

- なぜ: 新しいインテル® ストリーミング SIMD 拡張命令 (SSE4.2) は、XML構文解析、文字列解析/検索、パーサー処理、暗号化/復号化、CRCチェックサム、I-SCSI、RDMA、そしてビットパターン・アルゴリズムを最適化可能
- どのように行う: ソースコードや逆アセンブリーコードを参照し、SSE4.2 命令が適用可能であれば、次の命令セットに書き換える 

  PCMPISTRI、PCMPESTRI、PCMPISTRM、PCMPESTR
  CRC32、POPCNT、PCMPGTQ

- 対処は？: 最新の SSE4.2 命令セットを利用する。可能な方法として次のものがある:
  - ライブラリーを利用する: すでに SSE4.2 が実装されているインテル® インテグレーテッド・パフォーマンス・プリミティブやインテル® マスカーネル・ライブラリーを利用する
  - インテル・コンパイラー 11.0 を利用する: 最適化オプションに /QxSSE4.2（Linux では –xSSE4.2）を加える
    - 注意:11.0 コンパイラーでは SSE4.2 を利用するように、LIBM や CRT ライブラリー関数が強化されている（例:strlen など）
  - GCC (-mSSE4.2) や MS VC (/arch:SSE2) のスイッチ
  - アセンブリーや組み込み関数を書き換える

ステップ 3) アーキテクチャー上のボトルネックを調査

ステップ 1 と 2 で非効率な実行コード領域が見つかったら、次の順番で潜在する問題を調査する

- **キャッシュ・ミス**
- 分岐予測ミス
- フロントエンドのストール
- アドレス・エイリアシング
- 長いレイテンシーの命令と例外
- DTLB ミス

キャッシュミス

- なぜ？: キャッシュミスはアプリケーションの CPI を増加させる
  - > 長い遅延を伴う L2 と L3 キャッシュミスに注目する
- どのように？: つぎのイベントを収集

  MEM_LOAD_RETIRED.LLC_MISS、
  MEM_LOAD_RETIRED.LLC_UNSHARED_HIT、
  MEM_LOAD_RETIRED.OTHER_CORE_L2_HIT_HITM
- 対処方法は？:
  - > ホットスポット内で長い遅延のデータアクセスが何 % あるか見積もる:
    - L3 キャッシュミス: ((MEM_LOAD_RETIRED.LLC_MISS * 180) / CPU_CLK_UNHALTED.THREAD) * 100
    - L2 キャッシュミス: (((MEM_LOAD_RETIRED.LLC_UNSHARED_HIT * 35) + (MEM_LOAD_RETIRED.OTHER_CORE_L2_HIT_HITM * 74)) / CPU_CLK_UNHALTED.THREAD) * 100
  - > 上記の比率が 20% である場合ミスを減らすことを考える:
    ソフトウェア・プリフェッチ命令、データアクセスのブロック化、スレッドではローカル変数を利用、構造体をキャッシュラインに合わせるためパディング、データ利用を減らすためアルゴリズムを変更

分岐予測ミス

なぜ？：分岐予測ミスは無駄な処理や命令不足（分岐先の命令が取り込まれ実行可能なるのを待つ）により、パイプラインの効率を悪化させる

どのように？： 次のイベントを計測
UOPS_ISSUED.ANY、UOPS_RETIRED.ANY、
BR_INST_EXEC.ANY、BR_MISP_EXEC.ANY、
RESOURCE_STALLS.ANY

対処方法：

- ((UOPS_ISSUED.ANY/UOPS_RETIRED.ANY) – 1) は予測ミスにより浪費された実行の断片化を通知
- 命令不足 = (UOPS_ISSUED.CORE_STALL_CYCLES - RESOURCE_STALLS.ANY)/CPU_CLK_UNHALTED.THREAD
- 命令の浪費が 0.1 以上か命令不足が 0.1 以上の場合、分岐予測ミスの割合(%)を調査する ((BR_MISP_EXEC.ANY/BR_INST_EXEC.ANY) * 100)
- 分岐予測ミスの影響を軽減するため、より効率の良いコードを生成する（コンパイラーによるプロファイル・ガイド最適化、ハンドチューニング）

フロントエンドのストール

なぜ？: フロントエンドのストール（パイプラインの発行ステージでの）命令不足によることが多い。これによりパイプライン中の命令ステージが、実行する命令の不足によりストールする

どのように？: 次のイベントを計測
RESOURCE_STALLS.ANY、UOPS_ISSUED.CORE_STALL_CYCLES

対処方法:

- (UOPS_ISSUED.CORE_STALL_CYCLES - RESOURCE_STALLS.ANY)/CPU_CLK_UNHALTED.THREAD が 0.1 以上である場合、命令が不足している
- このカテゴリーのストールはコード生成の改善と再配置技術で回避できる:
  - インテル・コンパイラーで /QxSSE4.2 オプションを使用する
  - コンパイラーのプロファイルガイド最適化を利用する
  - リンカーのオーダー技術を使用する（Microsoft リンカーや gcc のリンクスクリプトでは /ORDER を使用）
  - コードサイズを減らすようなオプションを使用する（/O1 や /Os）

アドレスのエイリアシング

なぜ？: 大きなストライドで 2^N（たとえば 4K ）離れたデータをアクセスするといくつかの理由で効率が悪化する

- キャッシュの連想性
- 4K の誤ったストアフォワード

どのように？: PARTIAL_ADDRESS_ALIAS イベントを計測

対処方法:

- ((PARTIAL_ADDRESS_ALIAS) * 3) / CPU_CLK_UNHALTED.THREAD) * 100) は 4K フォルス・ストアフォワードにより浪費された実行の断片化を通知
- 上記の計測結果が 10% 以上である場合、他のキャッシュミスが発生する可能性があってもデータのレイアウトを変更する
- ソフトウェアが大きなストライドの 2^N でデータをスキップしないようにデータのレイアウトやアクセスパターンを変更する

長いレイテンシーの命令と例外

なぜ？：もしこのようなイベントや命令が性能低下の原因である場合、コードを変更するとパフォーマンスが向上する可能性がある

どのように？：次のイベントを計測する

UOPS_DECODED.MS、UOPS_RETIRED.ANY

対処方法？：

- (UOPS_DECODED.MS/UOPS_RETIRED.ANY) * 100 が 50% 以上であれば、過大な影響を受けている
- 例外や長いレイテンシーの命令の可能性を調査する
- 次の一般的な例外を検証：
  - 浮動小数点演算でデノーマルやオーバーフローが発生する可能性がないか調査する – もし SSE を使用しているなら FTZ や DAZ を有効にするか結果をスケーリングする
  - ゼロによる割り算を行っていないか確認する

DTLB ミス

なぜ？： ページをまたがったロードや、ロードごとに異なるページを読み込むと DTLB ミスが発生し、アプリケーションの性能に影響する

どのように見つける？： 次のイベントを監視する

DTLB_LOAD_MISSES.WALK_COMPLETED

対処方法？:

- TLB ミスの影響がどれ位あるか見積もる ≈ ((DTLB_LOAD_MISSES.WALK_COMPLETED * 30) / CPU_CLK_UNHALTED.THREAD) * 100
- もし 5 – 10% 以上であれば DTLB ミスを改善する
- 可能な最適化: データのアクセスを集中し TLB サイズに収まるようにする、仮想化システムで拡張ページテーブル（EPT）を使用する、ラージページを使用する（データベースやサーバーアプリケーションのみ）

検証

コード1:

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

コード2:

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

m = k = j = 1024 そして a, b, c は倍精度浮動小数点

実行結果

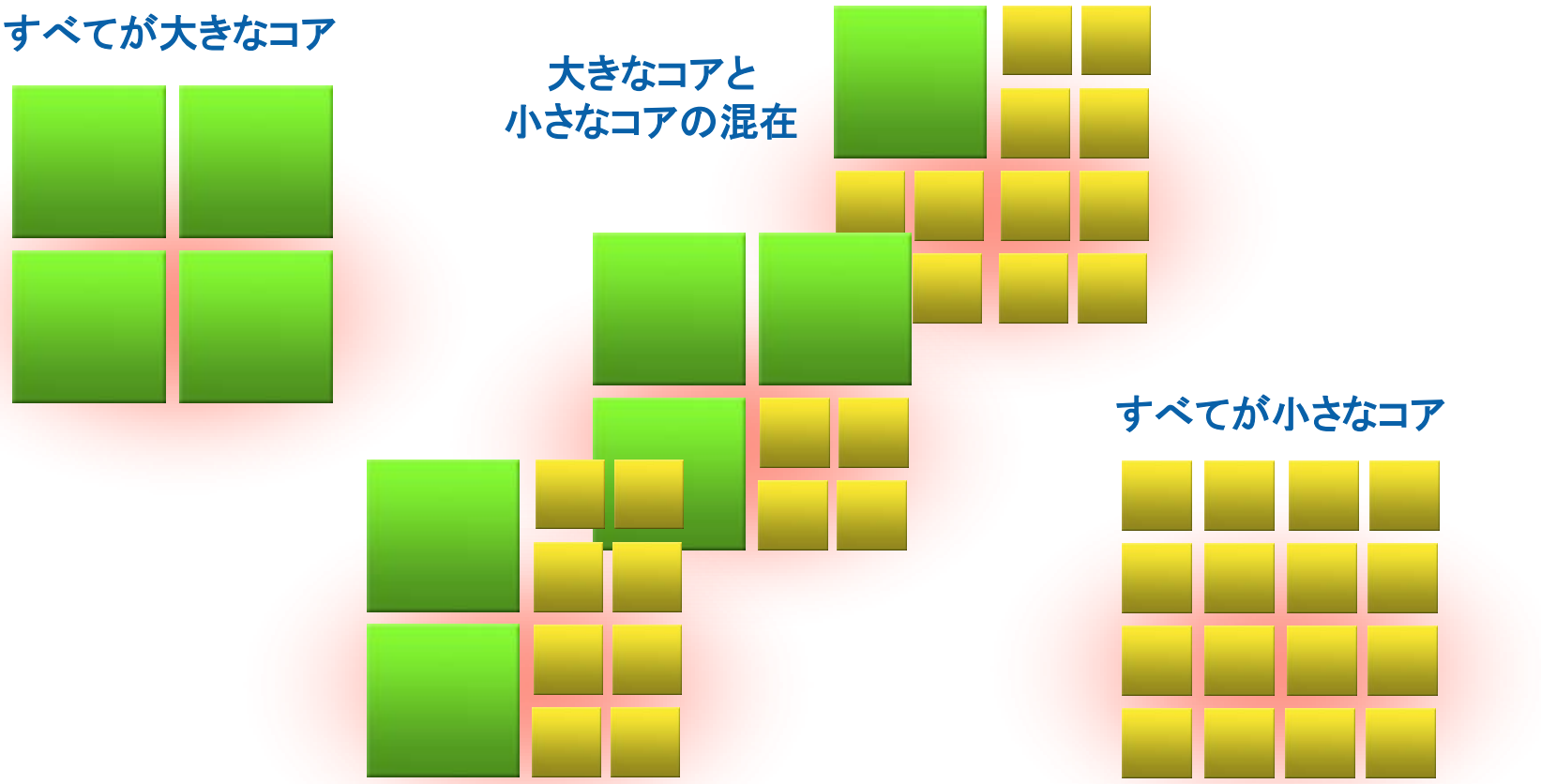
	インテル® Xeon X5570 x 2 (2.93GHz, 8 Core)	インテル® Xeon X5635 x 2 (3GHz, 8 Core)
コード1	34.7	37.5
コード2	110	253

	4 スレッド / 8 スレッド	4 スレッド / 8 スレッド
コード1	8.9 / 4.4	9.5 / 4.8
コード2	31 / 15	126 /109

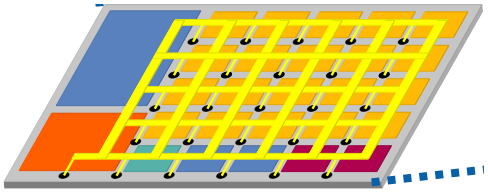
内容

- ソフトウェア最適化と並列化のおさらい
- インテル® マイクロアーキテクチャー Nehalemの特徴と最適化
- 今後のプログラミング環境と開発ツール

将来：マルチコアとメニーコア



メモリーバンクを接続、プロセッサーを接続、メモリーの一貫性モデル – すべてが現実となる、多様性の時代。



インテル・アーキテクチャーの命令セット
振り返ってみて

MMX™ テクノロジー →Pentium, 1995
- 57 個の命令セットを追加: グラフィックス、オーディオ/音声などDSPアプリを高速化

SSE（ストリーミング SIMD 拡張命令）→Pentium III, 1999
- 70 個の命令セットを追加: 浮動小数点演算を高速化

SSE2 → 第一世代 Pentium 4, 2001
- 144 個の命令セットを追加

SSE3 → 第三世代 Pentium 4, 2004
- 14 個の命令セットを追加

SSE4 → 45nm Core2 Duo (2007-08) と Core i7
- SSE2 以来の影響のある命令セット
- 47 個の命令セット
  - マルチメディア型のアプリケーションに貢献
  - 文字列操作型の命令をサポート

Sandy Bridge (2010): Intel® AVX!

SSE命令セットに 256 ビットのベクトル拡張

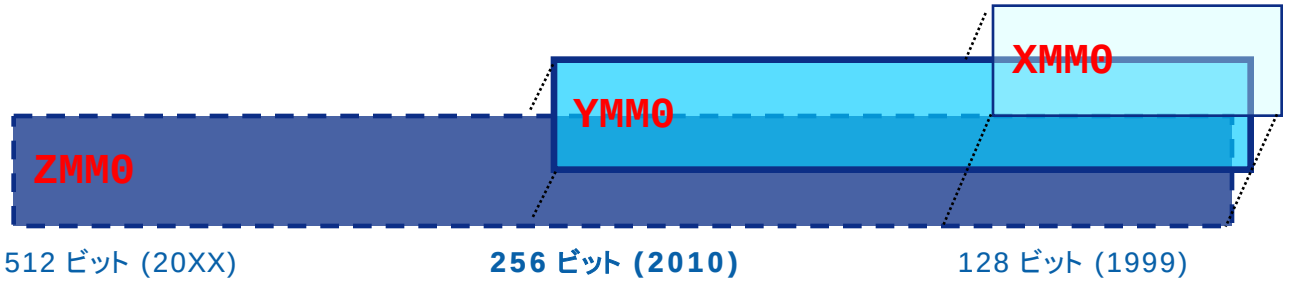
Intel® AVX は 16個の XMM レジスターを 256 ビットに拡張

Intel® AVX では下記のいづれかの操作が可能

- 256 ビット全体
- 下位 128 ビット（既存の SSE 命令のように）
  - 既存のスカラー及び 128 ビット SSE 命令を差し込みで置き換え可能

SSE を拡張/オーバーレイするため新たなステートを追加

YMM レジスターの下位（ビット 0-127）は XMM レジスターにマッピング



Intel® Advanced Vector Extensions (Intel® AVX)

浮動小数点演算ベースのアプリケーションに有益な SSE への 256 ビットベクトル拡張

おもな利点：

2倍以上の FLOP をもたらす広いベクトル

高度なデータ配置とアクセス向けの 256 のプリミティブ

3 オペランド、小さなコードサイズと並列操作のための非破壊シンタックス

Sandy Bridge ファミリー・プロセッサーで利用可能

詳細は http://software.intel.com/sites/avx

Intel® Software Emulator をダウンロードして始めてください

Intel AVX の利点

パフォーマンス: Intel AVX ではベクトル化可能な大きなデータセットを持つ既存のアプリケーションや新たしいアプリケーションのパフォーマンスを向上させる:

- より広いベクトルデータセットは 128 ビットデータセットの 2 倍のスループットを実現
- アプリケーションのパフォーマンスはハードウェア・スレッドとコア数にスケールできる
- アプリケーション・ドメインはインテル QPI などの高度なプラットフォーム相互接続ファブリックによってスケールアウトできる

電力効率: Intel AVX は電力効率が高い。命令が実行されるか、またはほとんど実行されない時の電力増加はわずかである。Intel AVX を多用するアプリケーションは高いパフォーマンスを提供し、より高いパフォーマンス・ワットを実現する電力効率を実現する

Intel AVX の利点（続き）

拡張性: Intel AVX はコードの改善に頼ることなく、強力な将来への拡張オプション提供する:

- OS のコンテキスト管理では 1 度だけ対応すればよい
- 256 ビットから 512 ビットの整数ベクトルをサポート予定
- 512 ビットと 1024 ビットの浮動小数点ベクトルをサポート予定

互換性: Intel AVX は SSE4 を含む以前の ISA 拡張への下位互換を提供:

- 既存の SSE アプリケーションから Intel AVX 128 ビットへの移行は簡単
- 既存の SSE アプリケーションから Intel AVX 256 ビットへの移行一貫性がある

広範囲: Intel AVX はサブノートからマルチプロセッサー・サーバーまで様々なインテル・プラットフォームで活用可能

サポート: **インテルの様々な開発ツールやオンラインサポートしついては** Intel® Software Network http://softwarecentral.intel.com を参照ください。Intel AVX の導入についての情報もあります

Larrabee

IA プログラミングの容易性と並列性を
高いスループットによる演算分野に適応

高い並列性

容易な IA アーキテクチャーのプログラミング

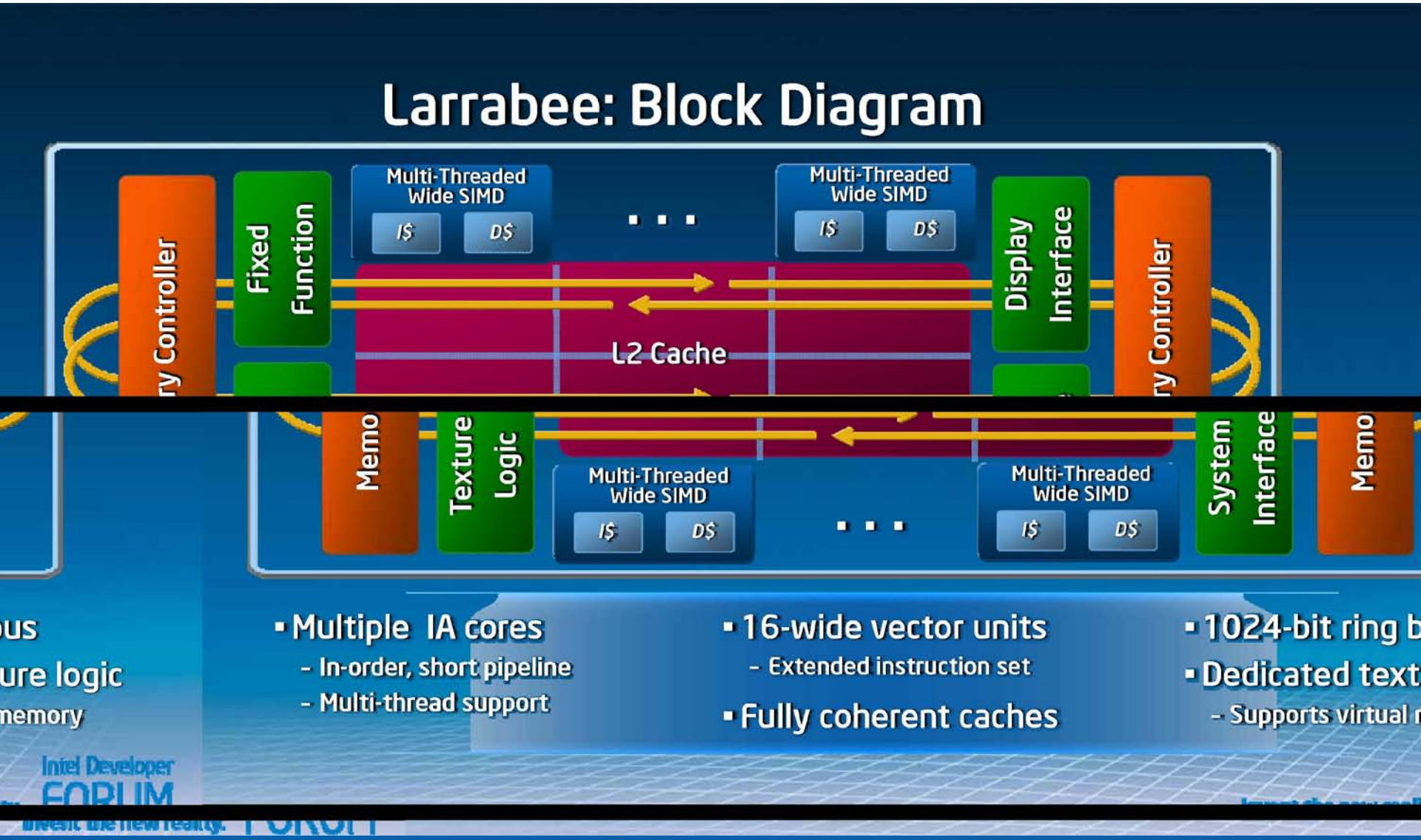
ソフトウェア・エコシステムに容易にスケール

拡張された IA コアのアレイ

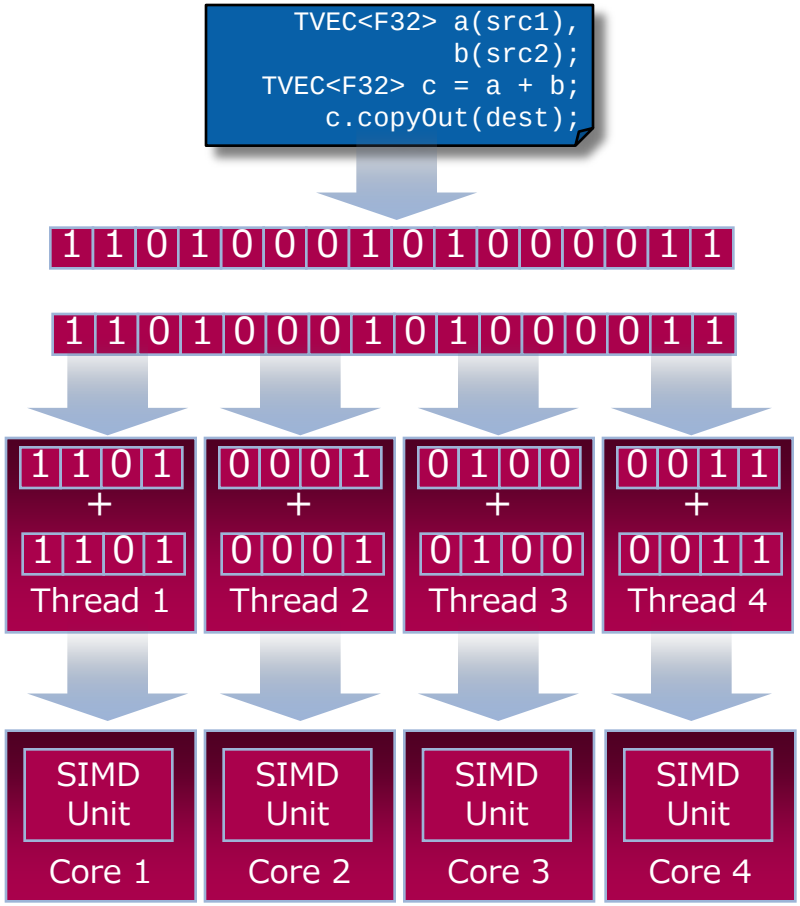
- 科学技術計算、RMS、ビジュアリゼーション、金融解析、ヘルスアプリケーション

テラフロップスの性能

Larrabee



Ct: スループット・プログラミング・モデル



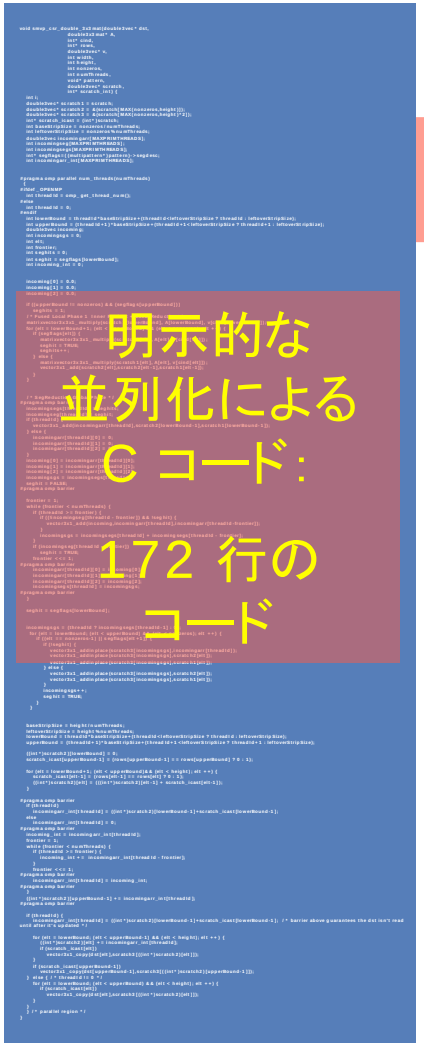
ユーザーはプロセッサー・コアに
依存しない　C++　コードを記述

Ct パラレル・ランタイム:
コアの増加に自動的にスケール

Ct JIT コンパイラー:
自動ベクトル化、SSE、
AVX、Larrabee

プログラマーはシリアルに考える; Ct が並列に展開

なぜ Ct を開発したか



Ct: 6 行以下のコード、
高速、スケーラブル

```
TVEC<F64> smvpCSC(TVEC<F64> A,
                  TVEC<I32> cols, TVEC<F64> v)
{
    TVEC<F64> expv, product, result;
    expv = v.distribute(cols);
    product = A*expv;
    return product.reduceSum();
}
```

LRB のビデオ・アルゴリズムを物理カーネルへ移植

- 剛体運動
- 広いおよび狭いフェーズ衝突
- ソルバー
- 布シュミレーション

“伝統的な”並列プログラミング・モデルでは困難が伴うことを認識

（ネストした）データ並列モデルはそれらを容易にする

Learn **Parallel**

並列化について学ぶ

Think **Parallel**

並列思考を行う

- 本資料に掲載されている情報は、インテル製品の概要説明を目的としたものです。製品に付属の売買契約書『Intel's Terms and conditions of Sales』に規定されている場合を除き、インテルはいかなる責を負うものではなく、またインテル製品の販売や使用に関する明示または黙示の保証 (特定目的への適合性、商品性に関する保証、第三者の特許権、著作権、その他、知的所有権を侵害していないことへの保証を含む) に関しても一切責任を負わないものとします。インテル製品は、医療、救命、延命措置などの目的への使用を前提としたものではありません。
- インテル製品は、予告なく仕様が変更される場合があります。本資料に記載されているすべての製品、日付、および数値は、現在の予想に基づくものであり、計画以外の目的ではご利用になれません。
- 機能や命令の中に「予約済み」または「未定義」と記されているものがありますが、その機能が存在しない状態や何らかの特性を設計の前提にしてはなりません。これらの項目は、インテルが将来のために予約しているものです。インテルが将来これらの項目を定義したことにより、衝突が生じたり互換性が失われたりしても、インテルは一切責任を負わないものとします。
- 本資料に掲載されているインテル製品は、エラッタと呼ばれる設計上の不具合が含まれている可能性があり、公開されている仕様とは異なる動作をする場合があります。現在までに判明しているエラッタの情報については、インテルまでお問い合わせください。
- 本資料には、設計段階にある製品の情報が含まれています。本資料で提供される情報は、予告なしに変更されることがあります。本資料をもとに設計を行わないでください。製品を注文する前に、販売代理店まで最新の仕様をお問い合わせください。
- Intel、インテル、Intel ロゴ、Intel Atom、Intel Core、Itanium、Xeon は、アメリカ合衆国およびその他の国における Intel Corporation の商標です。
- その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。
-