モダンコードと インテル® アーキテクチャー

パート 2/3

Colfax International — *colfaxresearch.com*

2016 年 11 月

法務上の注意書き

本トレーニングの準備には最善を尽くしていますが、Colfax International は、内容の正確さや完全性について、いかなる表明または保証もいたしません。また、いかなる責任も負いません。特に、商品適格性または特定目的への適合性の黙示的保証はいたしません。本資料に含まれる情報またはプログラムにより、直接的または間接的に生じた一切の損失、間接的または結果的損害、あるいは損失の申し立てについて、本資料の発行元は一切責任を負いません。販売担当者または販売促進資料は、一切の保証またはその追加・延長を行うものではありません。

コース・ロードマップ

- **パート 1**: マルチスレッドの手法
  - 同期を最小限に抑える
  - フォルス・シェアリングを回避する
  - 並列処理を行う
- **パート 2**: ベクトル化のチューニング
  - コンパイラーと開発者の役割
  - ディレクティブを利用したチューニング
  - データコンテナーの最適化
  - 言語拡張
- **パート 3**: メモリー・トラフィックの制御
  - キャッシュ使用率の最大化
  - メモリー帯域幅の最適化
  - インテル® Xeon Phi™ プロセッサー: 高帯域メモリー

§2. インテル®
アーキテクチャー

計算プラットフォーム

計算プラットフォーム

インテル® Xeon® プロセッサー

現在: Broadwell†
次世代: Skylake†

インテル® Xeon Phi™ コプロセッサー (第 1 世代)

Knights Corner (KNC)†

インテル® Xeon Phi™ プロセッサー (第 2 世代)*

*ソケットとコプロセッサー

Knights Landing (KNL)†

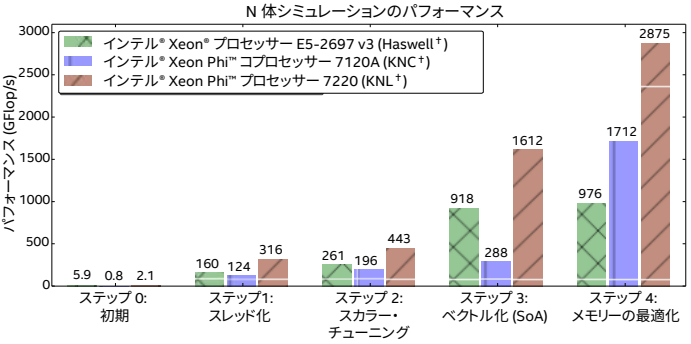
マルチコア・
アーキテクチャー

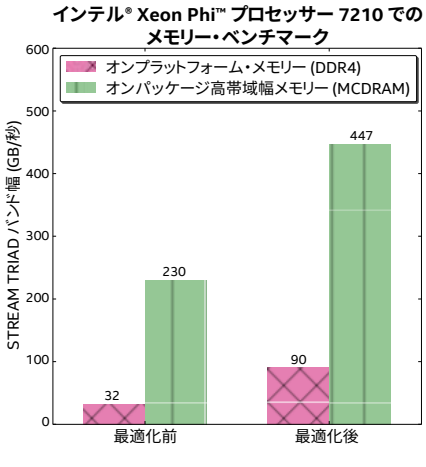
インテル® メニー・インテグレーテッド・コア
(インテル® MIC) アーキテクチャー

†開発コード名

パフォーマンスの最適化

優れたソフトウェアはパフォーマンスを引き出す

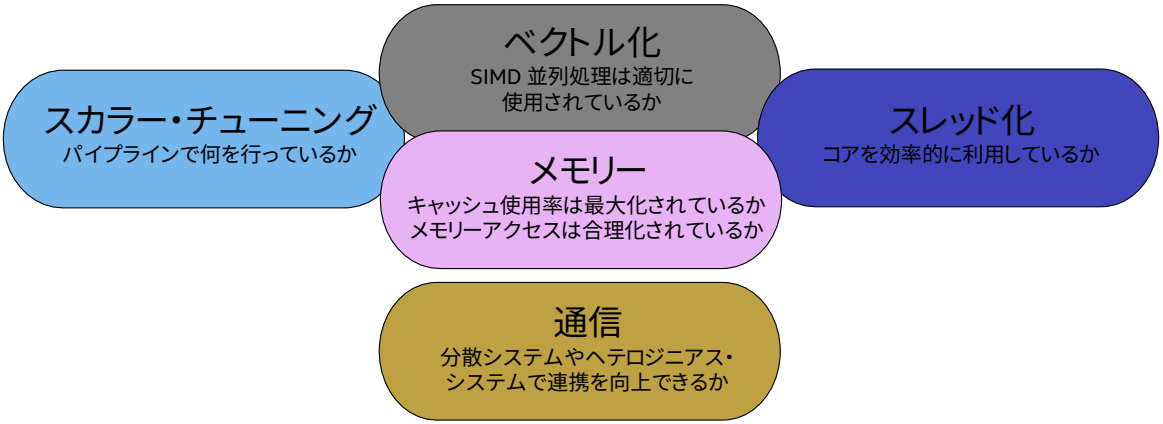




詳細は[この書籍]の第 23 章 N 体シミュレーションを参照

† 開発コード名

最適化領域

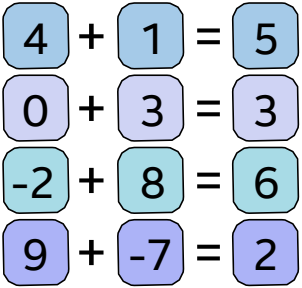


§3. ベクトル化

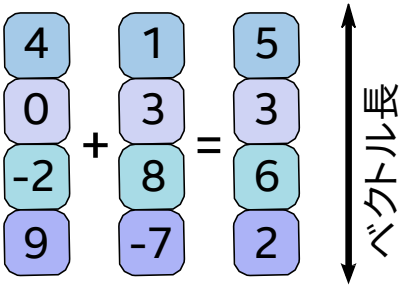
ショートベクトルのサポート

ベクトル命令 – SIMD (Single Instruction Multiple Data) 並列処理の実装の 1 つ

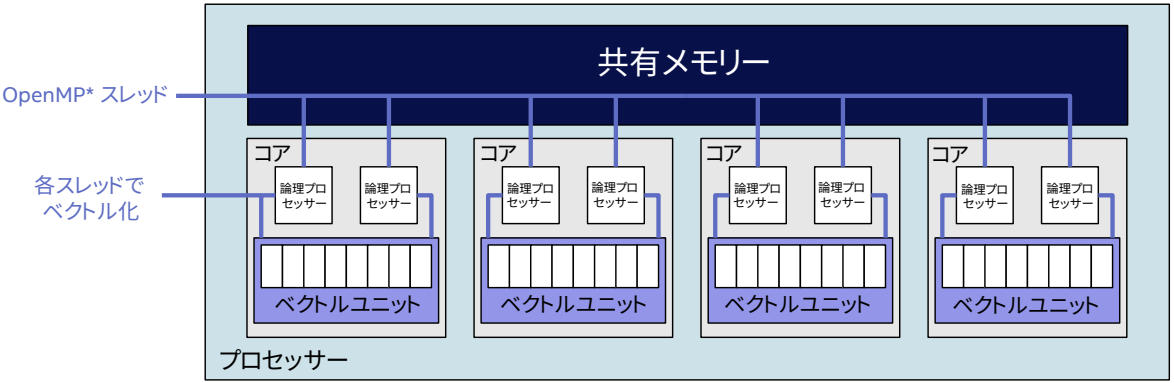
スカラー命令



ベクトル命令

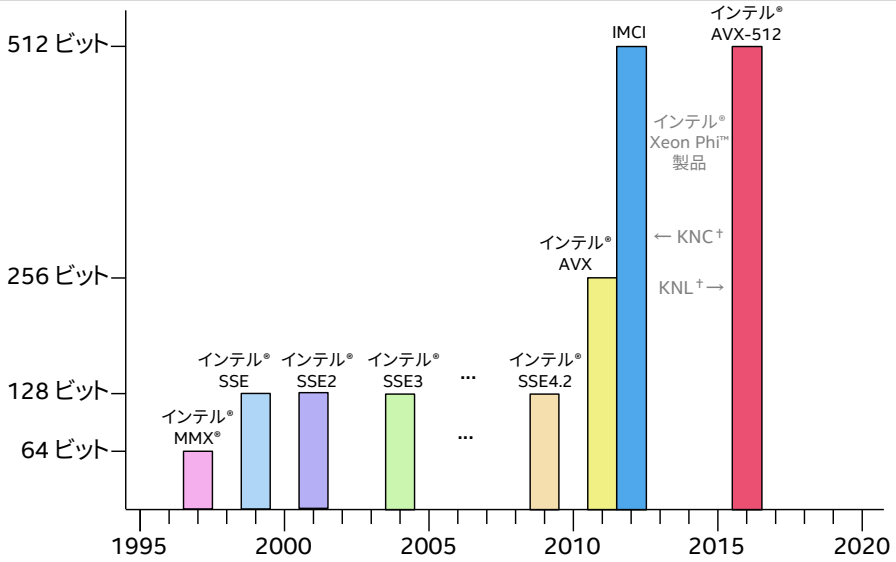


ベクトルとの共存



コアの利用: 複数のスレッド/プロセスを実行 (MIMD)
ベクトルの利用: 各スレッド (プロセス) がベクトル命令を発行 (SIMD)

インテル® アーキテクチャーの命令セット

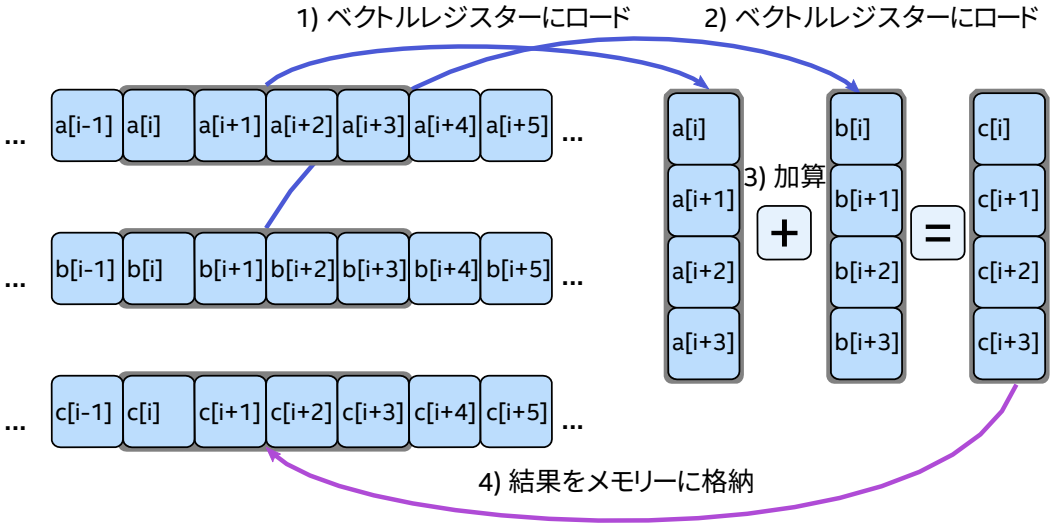


†開発コード名

ベクトル化

自動ベクトル化と
明示的なベクトル化

ベクトル計算のワークフロー



例: 数値積分

$$I(a,b) = \int_a^b \frac{1}{\sqrt{x}} \mathrm{d}x$$

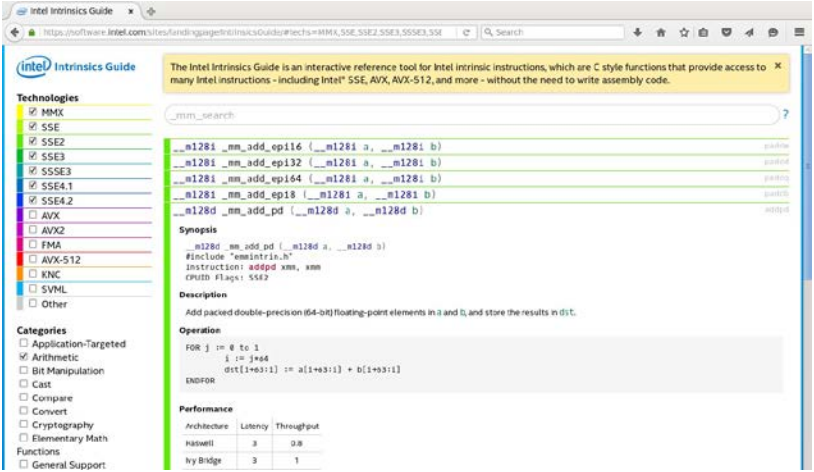
矩形メソッド:

$$\begin{aligned} \Delta x &= \frac{b-a}{n}, \\ x_i &= (i+1)\Delta x, \\ I(a,b) &= \sum_{i=0}^{n-1} \frac{1}{\sqrt{x_i}} \Delta x + O(\Delta x). \end{aligned}$$

```
1  float Integrate(const float a,
2                  const float b,
3                  const int N) {
4    const float dx = (b-a)/float(n);
5    float S = 0.0f;
6    for (int i = 0; i < n; i++) {
7      const float xi = dx*float(i+1);
8      S += 1.0f/sqrtf(xi) * dx;
9    }
10   return S;
11 }
```

インテルの組込み関数のガイド (英語)

https://software.intel.com/sites/landingpage/IntrinsicsGuide



インテル® SSE4.2 による実装

```
float Integrate(const float a,
                const float b, const int n) {
  __m128 dx = _mm_set1_ps((b - a)/float(n));
  __m128 S  = _mm_set1_ps(0.0f);
  for (int i = 0; i < n; i += 4) {
    __m128i ip1  =
              _mm_set_epi32(i+4, i+3, i+2, i+1);
    __m128  ip1f = _mm_cvtepi32_ps(ip1);
    __m128  xi = _mm_mul_ps(dx, ip1f);
    __m128  fi = _mm_rsqrt_ps(xi);
    __m128  dS = _mm_mul_ps(fi, dx);
    S  = _mm_add_ps(S, dS);
  }
  ConverterType c;
  c.v = S;
  return c.f[0] + c.f[1] + c.f[2] + c.f[3];
}
```

動作するが次の制限がある

- n は 4 の倍数であると仮定している
- インテル® SSE4.2 でのみ有効 (2011 年頃)
- メモリーアクセスを行う場合はピーリングが必要になることがある

ループの自動ベクトル化

```
1  #include <cstdio>
2
3  int main(){
4    const int n=8;
5    int i;
6    int A[n] __attribute__((aligned(64)));
7    int B[n] __attribute__((aligned(64)));
8
9    // 初期化
10   for (i=0; i<n; i++)
11     A[i]=B[i]=i;
12
13   // このループは自動ベクトル化される
14   for (i=0; i<n; i++)
15     A[i]+=B[i];
16
17   // 出力
18   for (i=0; i<n; i++)
19     printf("%2d %2d %2d¥n", i, A[i], B[i]);
20 }
```

```
vega@lyra% icpc autovec.cc -qopt-report
vega@lyra% cat autovec.optrpt
...
ループの開始 autovec.cc(14,3)
リマーク #15399: ベクトル化のサポート:
アンロールファクターが 2 に設定されます。
[autovec.cc(14,3)]
リマーク #15300: ループがベクトル化されました。
[autovec.cc(14,3)] ループの終了
...
vega@lyra% ./a.out
0  0  0
1  2  1
2  4  2
3  6  3
4  8  4
5 10  5
6 12  6
7 14  7
```

自動ベクトル化の制限

- ループに入る前に反復回数が判明していなければならない
- 最内ループのみ (オーバーライド可能)
- ベクトル依存関係があってはならない
- ベクトルループから呼び出される関数は SIMD 対応でなければならない

特定の命令セットを対象にする

`-x[code]` は、命令セットと最適化を含む、固有のプロセッサー機能を対象にするようにコンパイラーに指示する

コード	対象アーキテクチャー
MIC-AVX512	インテル® Xeon Phi™ プロセッサー (KNL†)
CORE-AVX512	将来のインテル® Xeon® プロセッサー
CORE-AVX2	インテル® Xeon® プロセッサー E3 v3 ファミリー
CORE-AVX-I	インテル® Xeon® プロセッサー E3/E5/E7 v2 ファミリー
AVX	インテル® Xeon® プロセッサー E3/E5 ファミリー
SSE4.2	インテル® Xeon® プロセッサー 55XX/56XX/75XX 製品ファミリーおよびインテル® Xeon® プロセッサー E7 ファミリー
host	コードがコンパイルされるアーキテクチャー

† 開発コード名

言語拡張

配列表記 (アレイ・ノーテーション) の拡張

配列表記は以下の項目を指定する手法

- 配列のスライス (開始、長さ)

```
A[0:16] += B[32:16]; // B[32]...B[47] を A[0]...A[15] に追加
```

- ストライド (開始、長さ、ストライド)

```
A[0:16:2] += B[32:16:4]; // B[32],B[36]...B[92] を A[0],A[2]...A[30] に追加
```

- 多次元配列

```
A[:][:] += B[:][:]; // B を A に追加 - 配列は同じ形状
```

ストライド形式のループより優れている (この記事を参照)

配列表記を含む式は複雑

http://xeonphi.com/papers/efft の例

```
evenrek[:] = evens[kk  :kTILE:2];
evenimk[:] = evens[kk+1:kTILE:2];
oddrek [:] = odds [kk  :kTILE:2];
oddimk [:] = odds [kk+1:kTILE:2];

evens[kk  :kTILE:2] = evenrek[:] + coslist[:]*oddrek[:] - sinlist[:]*oddimk[:];
evens[kk+1:kTILE:2] = evenimk[:] + sinlist[:]*oddrek[:] + coslist[:]*oddimk[:];

oddmirrek[:] = odds[size-kk  :kTILE:-2];
oddmirimk[:] = odds[size-kk+1:kTILE:-2];

odds[size-kk  :kTILE:-2] =
                     evenrek[:] - coslist[:]*oddrek[:] + sinlist[:]*oddimk[:];
odds[size-kk+1:kTILE:-2] =
                    -evenimk[:] + sinlist[:]*oddrek[:] + coslist[:]*oddimk[:];
// ...
```

SIMD 対応関数

(旧称: 要素関数)
関数の実装が別のソースコード (例えば、ライブラリー関数) に含まれる場合

```
1 float my_simple_add(float x1, float x2){
2   return x1 + x2;
3 }
```

```
1 // 別のソースファイル:
2 for (int i = 0; i < N, ++i) {
3   output[i] = my_simple_add(inputa[i], inputb[i]);
4 }
```

このループは自動的にベクトル化されない

SIMD 対応関数は複雑な関数も対応可

http://xeonphi.com/papers/simd-lib の例

```
__attribute__((vector)) float MyErfElemental(const float inx){
    // 誤差関数の解析近似を計算
    const float x = fabsf(inx); // 絶対値を取得 (各ベクトルレーン)
    const float p = 0.3275911f; // ベクトルレーンの定数パラメーター
    const float t = 1.0f/(1.0f+p*x); // 各ベクトルレーンの式
    const float l2e = 1.442695040f; // log2f(expf(1.0f))
    const float e = exp2f(-x*x*l2e); // 各ベクトルレーンの超越数
    float res = -1.453152027f + 1.061405429f*t; // 多項式の計算
    res = 1.421413741f + t*res;                 // 各ベクトルレーン
    res =-0.284496736f + t*res;
    res = 0.254829592f + t*res;
    res *= e;
    res = 1.0f - t*res; // 各ベクトルレーンの解析近似
    return copysignf(res, inx); // 各ベクトルレーンの符号をコピー
}
```

ユニットストライド形式のアクセス

ユニットストライド形式のアクセス

ユニットストライド形式は最適:

```
1 for (int i = 0; i < n; i++)
2   A[i] += B[i];
```

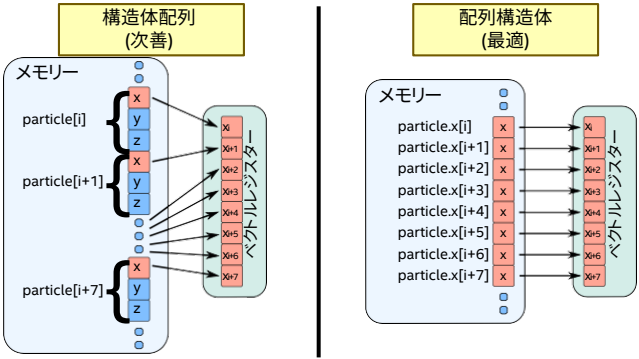
非ユニットストライド形式は遅い:

```
1 for (int i = 0; i < n; i++)
2   A[i*stride] += B[i];
```

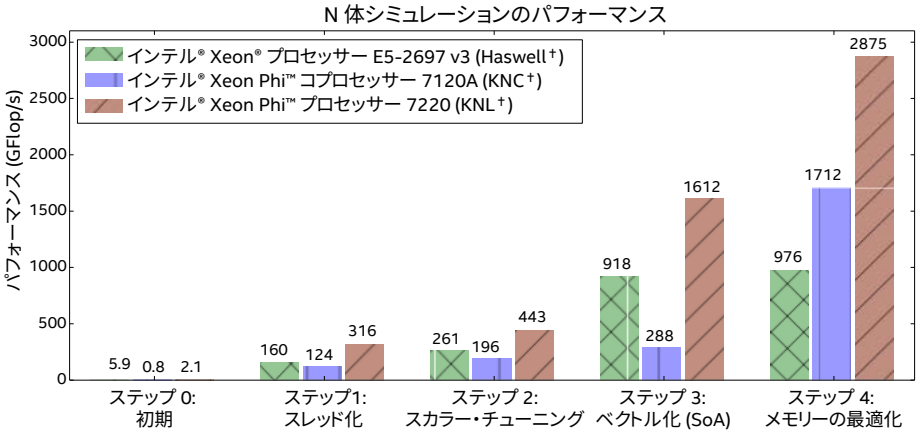
確率的アクセスはベクトル化されるが効率的ではない:

```
1 for (int i = 0; i < n; i++)
2   A[offset[i]] += B[i];
```

ループの入れ子の順序変更が問題になる場合、データ構造を修正する必要がある



ユニットストライド形式のアクセス



詳細は[この書籍]の第 23 章 N 体シミュレーションを参照

† 開発コード名

アライメントと
パディング

データ・アライメントの要件

`((size_t)p%n==0)` の場合、配列 `char* p` は n バイトにアライメントする

プロセッサー	演算	アライメント
インテル® Xeon® プロセッサー (Westmere† 以前)	インテル® SSE ロード、ストア	16 バイト
インテル® Xeon® プロセッサー (Sandy Bridge† 以降)	インテル® AVX ロード、ストア	32 バイト (relaxed)
インテル® Xeon Phi™ コプロセッサー (第 1 世代)	IMCI ロード、ストア	64 バイト (strict)
インテル® Xeon Phi™ コプロセッサー (第 1 世代)	オフロードで DMA 転送	4096 バイト (preferred)
インテル® Xeon Phi™ プロセッサー (第 2 世代)	インテル® AVX-512 ロード、ストア	64 バイト (relaxed)

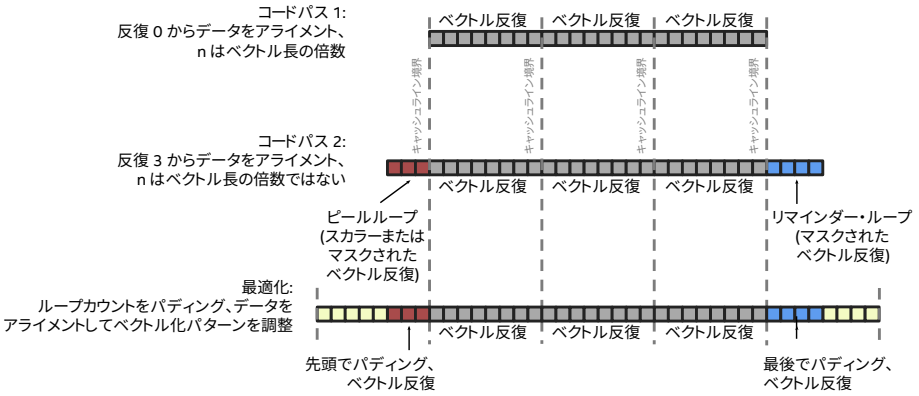
アライメントを行う理由: ベクトルロード/ストアのスピードアップ、
フォルス・シェアリングの回避 (セッション 7 を参照)、RDMA の高速化

† 開発コード名

アライメントを行わない場合

コンパイラーはピールループとリマインダー・ループを実装する:

```
for (i = 0; i < n; i++) A[i] = ...
```



アライメントされたデータコンテナーの作成

- スタックのデータ・アライメント

```
1 float A[n] __attribute__((aligned(64))); // 64 バイトでアライメント
```

- ヒープのデータ・アライメント

```
1 float *A = (float*) _mm_malloc(sizeof(float)*n, 64);
```

- A[0] は 64 バイト境界でアライメントされる
- アライメント値が非常に高いと仮想メモリーの浪費につながる
- Fortran: ディレクティブまたはコンパイラー引数 -align array64byte

多次元コンテナーのパディング

アライメントされた命令を使用するには、多次元配列の内側の次元を 16 (SP) または 8 (DP) 要素の倍数にパディングする必要がある

間違い:

```
// A - サイズ (n x n) の行列
// n は 16 の倍数ではない
float* A =
 _mm_malloc(sizeof(float)*n*n, 64);

for (int i = 0; i < n; i++)
  // A[i*n + 0] はアライメントされない
  for (int j = 0; j < n; j++)
    A[i*n + j] = ...
```

正しい:

```
// ...内側の次元をパディング
int lda=n + (16-n%16); // lda%16==0
float* A =
 _mm_malloc(sizeof(float)*n*lda, 64);

for (int i = 0; i < n; i++)
  // A[i*lda + 0] は i でアライメントされる
  for (int j = 0; j < n; j++)
    A[i*lda + j] = ...
```

コンパイラー・
ディレクティブ

ベクトル化のプラグマ、キーワード、コンパイラー引数

- `#pragma simd`
- `#pragma vector always`
- `#pragma vector aligned | unaligned`
- `__assume_aligned` キーワード
- `#pragma vector nontemporal | temporal`
- `#pragma novector`
- `#pragma ivdep`
- `restrict` 修飾子および `-restrict` コマンドライン引数
- `#pragma loop count`
- `-qopt-report -qopt-report-phase:vec`
- `-O[n]`
- `-x[code]`

§4. 関連情報

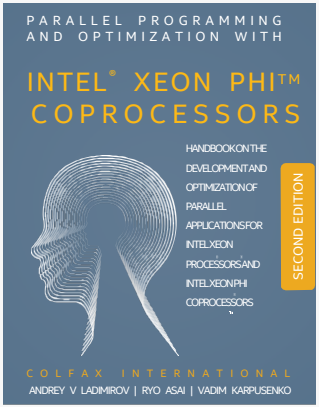

THE "HOW" SERIES TRAINING
DEEP DIVE
WITH CODE MODERNIZATION EXPERTS
It's free
→HowSeries.com
*10x 2-hour sessions | 24-hour 2-weeks remote access to a system

参考書

ISBN: 978-0-9885234-0-1 (508 ページ、電子版または印刷版)

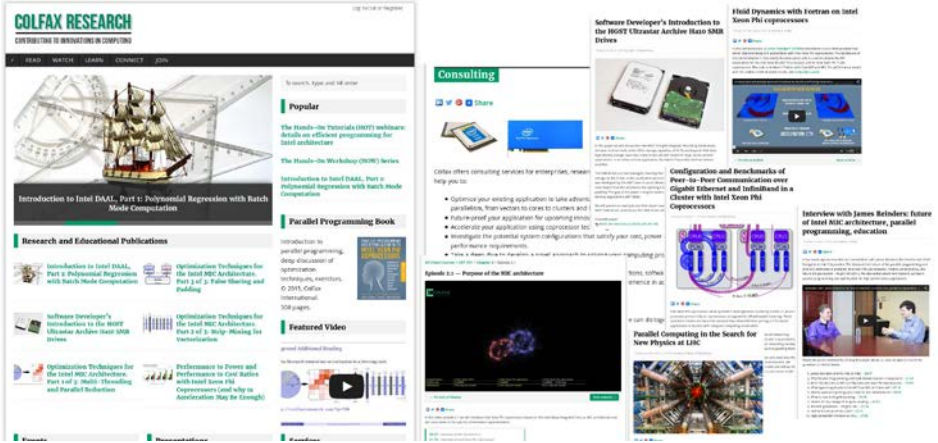
インテル® Xeon Phi™
コプロセッサーを利用した
並列プログラミングと最適化

インテル® Xeon® プロセッサーおよび
インテル® Xeon Phi™ コプロセッサー向け
並列アプリケーションの開発と
最適化のハンドブック



http://xeonphi.com/book

Colfax Research



http://colfaxresearch.com/

関連情報

Developer Access Program (DAP)

Knights Landing† 早期アクセスシステム受注中

詳細は、dap.xeonphi.com をご覧になるか、
dap@colfax-intl.com までお問い合わせください

†開発コード名

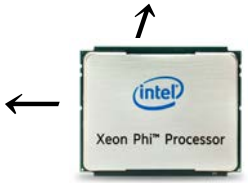
ブート可能なインテル® Xeon Phi™ プロセッサー

- ブート可能なホスト・プロセッサー
- RHEL/CentOS*/SUSE*/Windows®
- 64 コア × 4 HT、1.30GHz
- ≤ 384GiB DDR4、> 90GB/秒
- 16GiB HBM、> 400GB/秒
- ネットワーク用 PCIe* バス

dap.xeonphi.com

サーバー:

ワークステーション:



関連情報

ありがとうございました!