筑波大学計算科学研究センター
CCS HPCサマーセミナー
「最適化Ⅰ」

高橋大介
daisuke@cs.tsukuba.ac.jp
筑波大学計算科学研究センター

講義内容

- 並列計算機システムの計算ノード単体における プログラムの最適化手法
  - レジスタブロッキング
  - キャッシュブロッキング
  - ストリーミングSIMD命令の活用

性能チューニング

- ソフトウェア・アプリケーションにおけるパフォーマンスの重要性については誰もが認識している.
- しかし，パフォーマンスのチューニングに関していえば，ソフトウェア開発サイクルの中で後回しになりがちで，まったく考慮されない場合すらある.
- このような状況に陥っている要因として，
  - コード生成ツールやコンパイラだけでアプリケーションを最適化できるという認識
  - 単に最新のプロセッサを使えばアプリケーション実行時に最高のパフォーマンスが得られるという過剰な期待

  が挙げられる.

性能チューニングの意義

- しかし，実行に数ヶ月以上かかるような計算において，最適化を行うことにより，月のオーダーで実行時間を削減できるような場合．
- 数値計算ライブラリのように，多くの人に使われるプログラムであれば，チューニングを行う価値は十分にある．
- チューニングによってパフォーマンスが仮に3割向上したとすれば，それは，3割性能の高いマシンを使用しているのと同じことになる．

最適化

- 最適化の対象はいろいろある.
  - コード量の削減
  - データ量の削減
  - 実行時間の削減
- 今回は，実行時間を削減するためにプログラムを書き換えることを「最適化」と呼ぶことにする.

最適化の利点

- 最適化を行って実行時間を削減することにより，
  - 計算機の有効活用
  - 電気代（または課金）の削減
  - 同じ時間でより多くの計算ができる
- プログラムを書く時間＋実行時間の観点から考えると，長時間実行されるプログラムであるほど，最適化のメリットを享受できる．
  - 最適化によって性能が仮に3割向上したとすれば，それは，3割性能の高いマシンを使用しているのと同じことになる．
- 1回しか実行されず，かつ実行時間の短いプログラムは，最適化してもあまり意味がない．

最適化を行う前に

- そもそも，最適化を行う必要があるか？
- 現在用いているアルゴリズムは最適か？
- 効率の悪いアルゴリズムを最適化しても，意味がない．
  - バブルソートのプログラムを最適化しても，クイックソートよりは速くならない．
- 最適なアルゴリズムは
  - 解くべき問題の性質
  - 使おうとする計算機のアーキテクチャ，メモリ量など

に大きく依存する

最適化の方針

- ベンダー提供の高速なライブラリが使える場合には，できるだけ使うようにする．
  - BLAS，LAPACKなど
- 最近のコンパイラの最適化能力は非常に高くなっている．
- コンパイラでもできる最適化は，ユーザー側では行わない．
  - 手間が掛かるだけ．
  - プログラムが複雑になりバグが入り込む余地が出てくる．
  - コンパイラの最適化能力を過信しない．
- 人間はアルゴリズムの改良に専念する．
- アセンブラはやむを得ない場合を除き，使わない．

最適化の第一歩

- まず，自分のプログラムでどの位の演算性能が出ているかを調べる.
- 演算性能の指標として，FLOPS（Floating Operations Per Second）がある.
  - 1秒間に実行可能な浮動小数点演算の回数を表す単位
  - MFLOPS（10^6），GFLOPS（10^9)，TFLOPS（10^12），PFLOPS（10^16）
- プログラム全体（または一部）の実行時間と，演算回数から，FLOPS値を算出し，プロセッサの理論ピーク性能と比較する.
  - 最新のIntel Core i7では1コア当たりクロックの32倍のFLOPS値

時間計測

- 時間計測を行う対象として
  - 経過時間（elapsed time）
  - CPU時間（CPU time）

  がある.
- 対象とするプログラムの実行時間が短い場合，タイマーの精度が足りない場合がある.
  - 何回か外側にループを回して測定する.
- この場合，コンパイラの最適化により，ループが回っていないことになる場合があるので注意する.
  - ダミールーチンを入れるか，測定対象をサブルーチンにして，分割コンパイルする.

ホットスポット

- 計算時間の大半を占有する部分を「ホットスポット」という.
- まず，どこがホットスポットかを調べる.
- 便利なツールとして，プロファイラがある.
  - Linuxではgprofコマンドが使える.
    - 「gcc -pg foo.c」のように，コンパイラオプションに「-pg」を付けることにより，gprofによって使用されるプロファイル情報を書き込む特別なコードが生成される.
  - a.outを実行し，その後にgprof a.outとすることで，ホットスポットを特定することができる.

gprofの出力例

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
48.90	2.90	2.90	2	1.45	2.83	zfft1d0_
32.38	4.82	1.92	49152	0.00	0.00	fft8b_
14.17	5.66	0.84	16384	0.00	0.00	fft8a_
4.55	5.93	0.27	1	0.27	5.93	MAIN__
0.00	5.93	0.00	16384	0.00	0.00	fft235_
0.00	5.93	0.00	4	0.00	0.00	factor_
0.00	5.93	0.00	3	0.00	1.89	zfft1d_
0.00	5.93	0.00	2	0.00	0.00	settbl_
0.00	5.93	0.00	1	0.00	0.00	settbls_

gprofの結果から分かること

- ホットスポットは
  - zfft1d0_
  - fft8b_
  - fft8a_

  の3つであり，この3つで全実行時間の95%以上を消費している.
- これらのホットスポットのみに着目して最適化すればよい.
- プログラムを記述する際にはホットスポットが集中するように配慮する.
- ホットスポットが多くあると，コードの改良に手間が掛かる.
  - 最初からコードを書き直した方がましな場合もある.

コンパイルオプション

- コンパイルオプションの指定の仕方によって，性能が大きく変化する.
- コンパイラのマニュアルを参考に，いろんなコンパイルオプションを試してみる.
  - 「-fast」，「-O3」，「-O2」，など
  - Intel Compilerでは「-xCORE-AVX512」（最新のIntel Core i7向け）
- 必ずしも最適化レベルを高くしたからといって，速いコードを出力するとは限らない.
  - コンパイラが余計な最適化を行う可能性があるため.
  - 計算結果が合わない場合もあるので注意する.

コンパイラディレクティブ

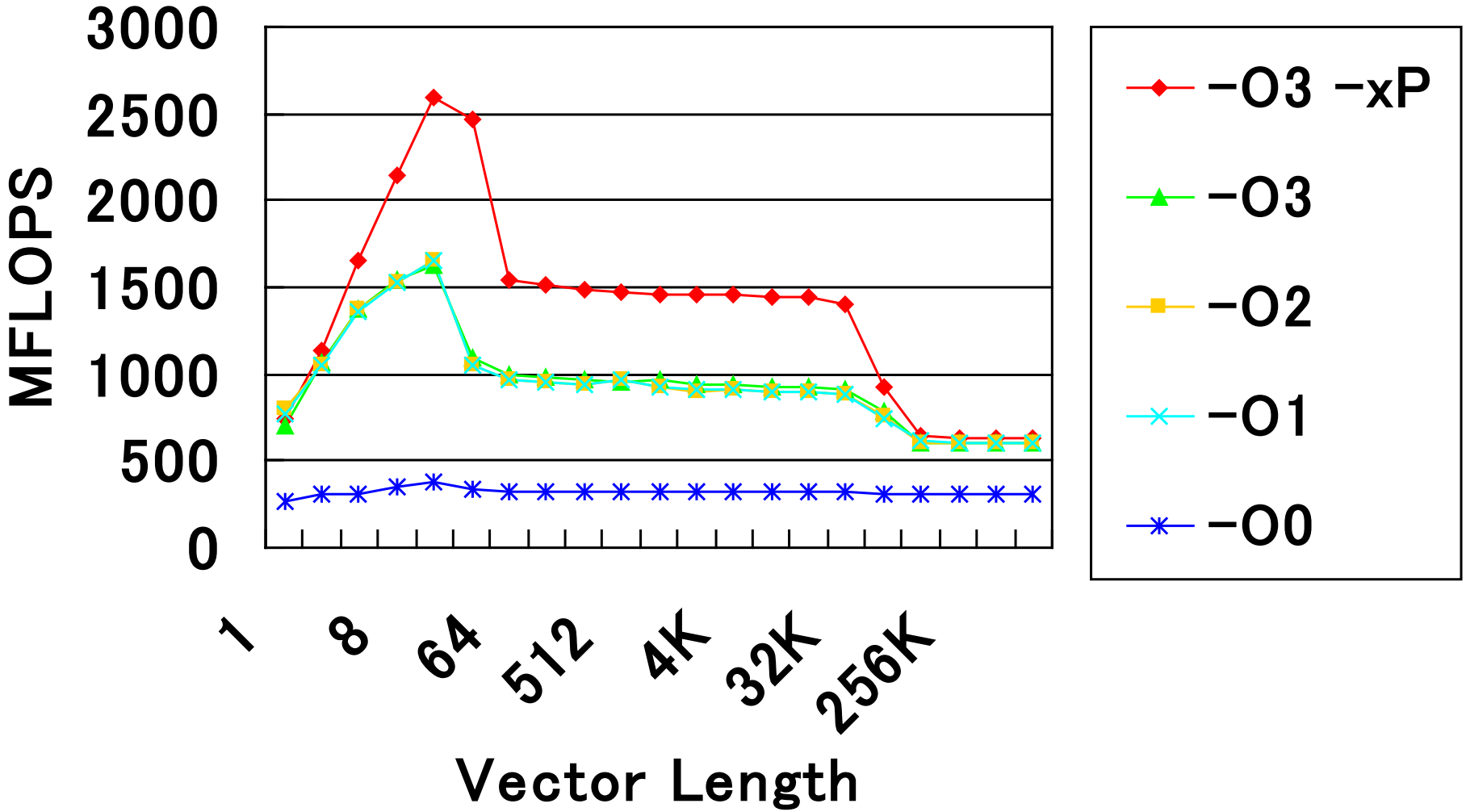
- コンパイラディレクティブ（指示行）は，コンパイラにプログラマの意図を伝え，最適化を支援する.
  - コンパイルオプションと違い，ループ単位で最適化をコントロールできる.
- ディレクティブの例
  - ベクトル化を行う際に，ループの依存性がないことをコンパイラに指示する.
  - ベクトル化の抑止
- C言語では「#pragma」，Fortranでは「!dir$」や「cpgi$l」などで記述することが多い.
  （コンパイラによって違うことがあるので注意）

Fortranで記述したZAXPY

```
      subroutine zaxpy(n,a,x,y)
      complex*16 a,x(*),y(*)
!dir$ vector aligned
      do i=1,n
         y(i)=y(i)+a*x(i)
      end do
      return
      end
```

ZAXPYの性能

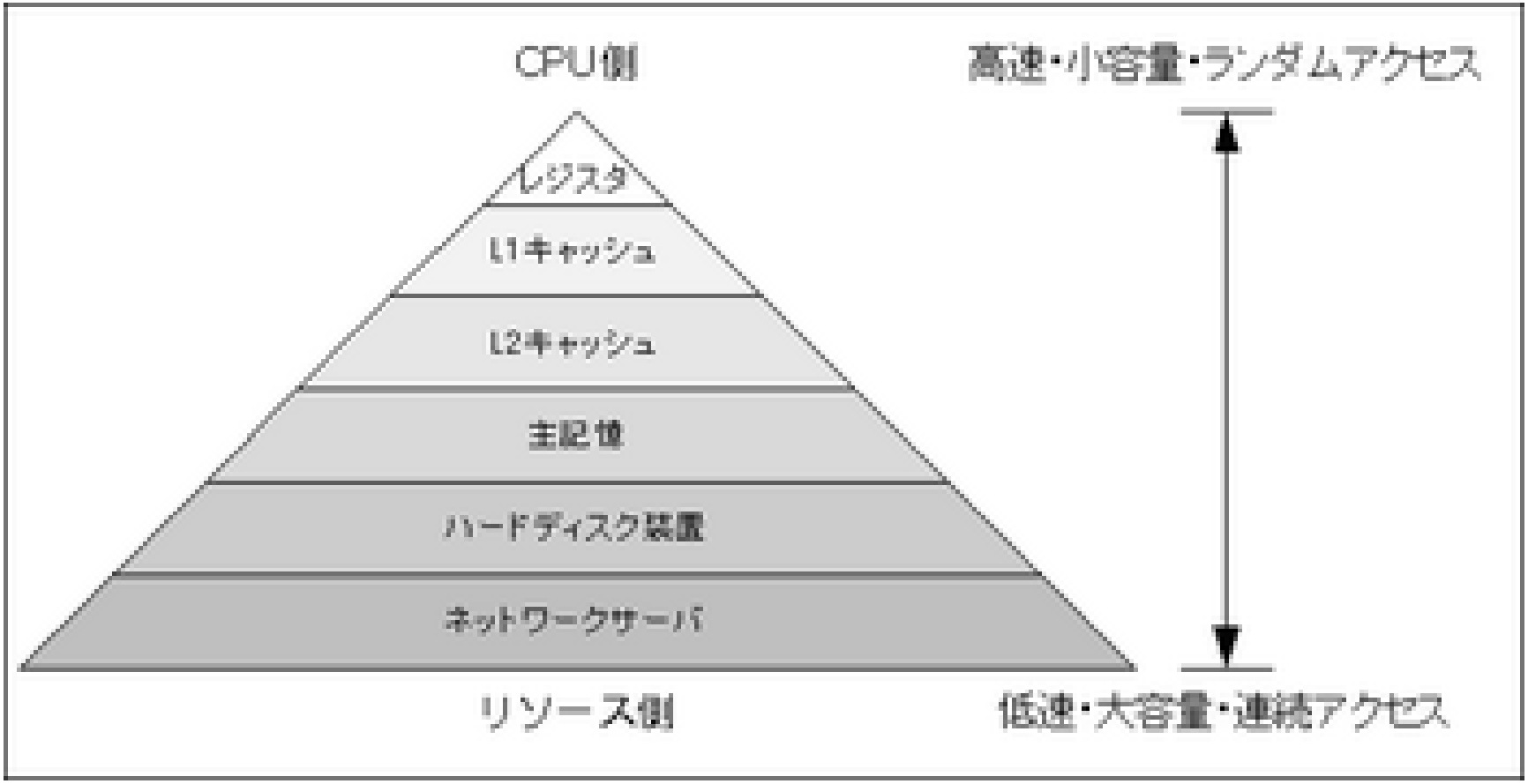
(Xeon 2.8GHz, 1CPU, Intel Fortran)



プログラムを記述する際の注意点

- CやFortranの文法をきちんと守る.
  - コンパイラによっては，warningが出るだけのこともあるが，多くの場合バグの原因になる.
- コンパイラに依存する拡張機能は，やむを得ない場合（例えばディレクティブなど）を除き，できるだけ使わないようにする.
  - g77における自動割付配列
    - real*8 a(n)で，a(n)が仮引数でなく，かつnが変数のような場合
  - プログラムの移植性が悪くなる.
  - 思わぬエラーの原因となる.
- あまり使われていない（と思われる）関数や機能はなるべく使わないようにする.
  - コンパイラのバグが取りきれていない可能性がある.

記憶階層(1/4)



出典:Wikipedia

記憶階層(2/4)

- データを保持する記憶装置のコストバランス
  - 「小容量×高速≒大容量×低速」が成り立つ
- 「小容量×高速」記憶装置
  - レジスタ
- 「大容量×低速」記憶装置
  - ハードディスクや磁気テープ
- 「大容量×高速」はコストパフォーマンスが悪く実現困難

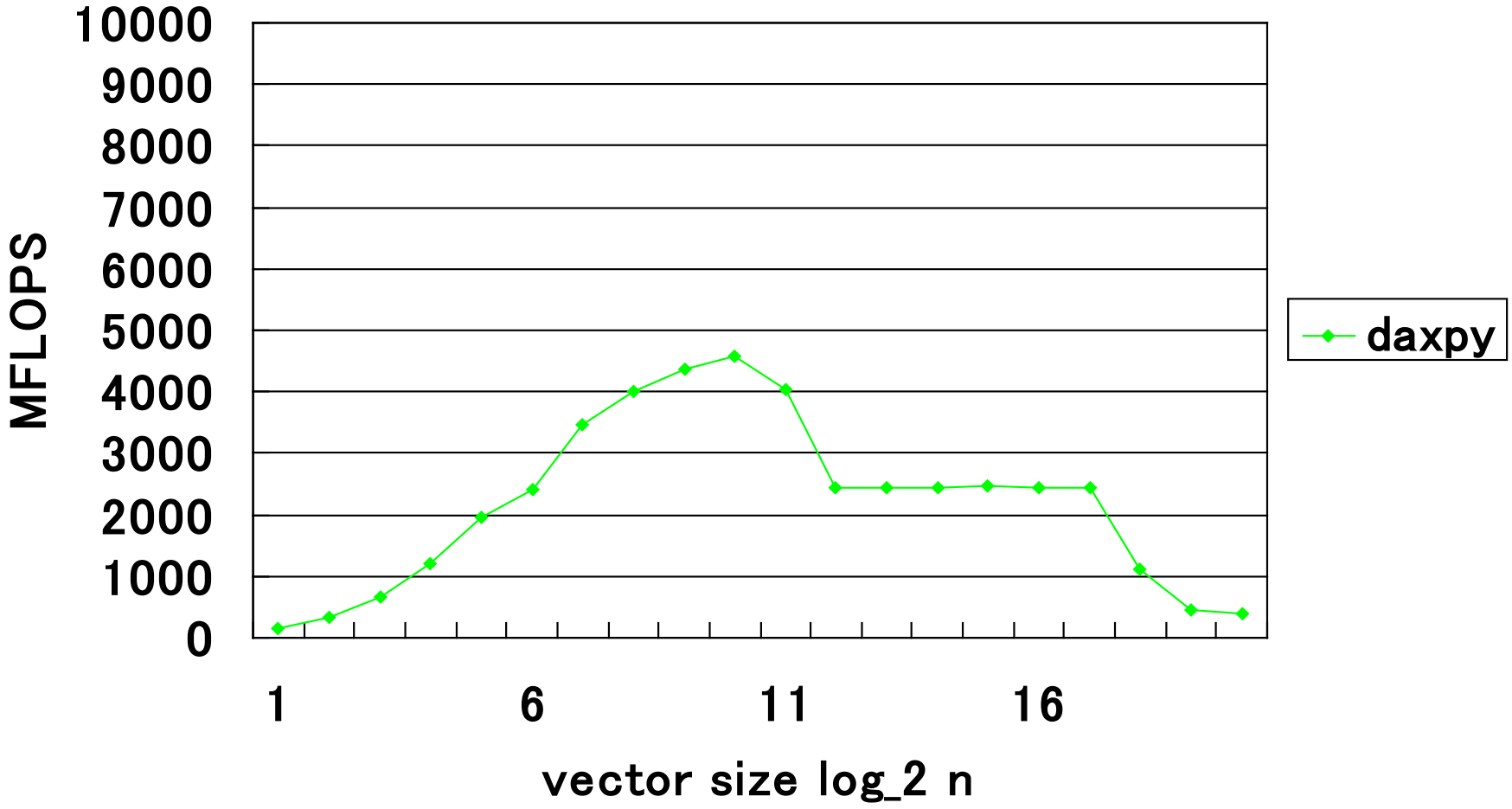
記憶階層(3/4)

- 記憶階層は記憶域に対するアクセスパターンの局所性(locality)を前提に設計されている.
- 局所性には
  - 時間的局所性
    - ある一定のアドレスに対するアクセスは，比較的近い時間内に再発するという性質
  - 空間的局所性
    - ある一定時間内にアクセスされるデータは，比較的近いアドレスに分布するという性質

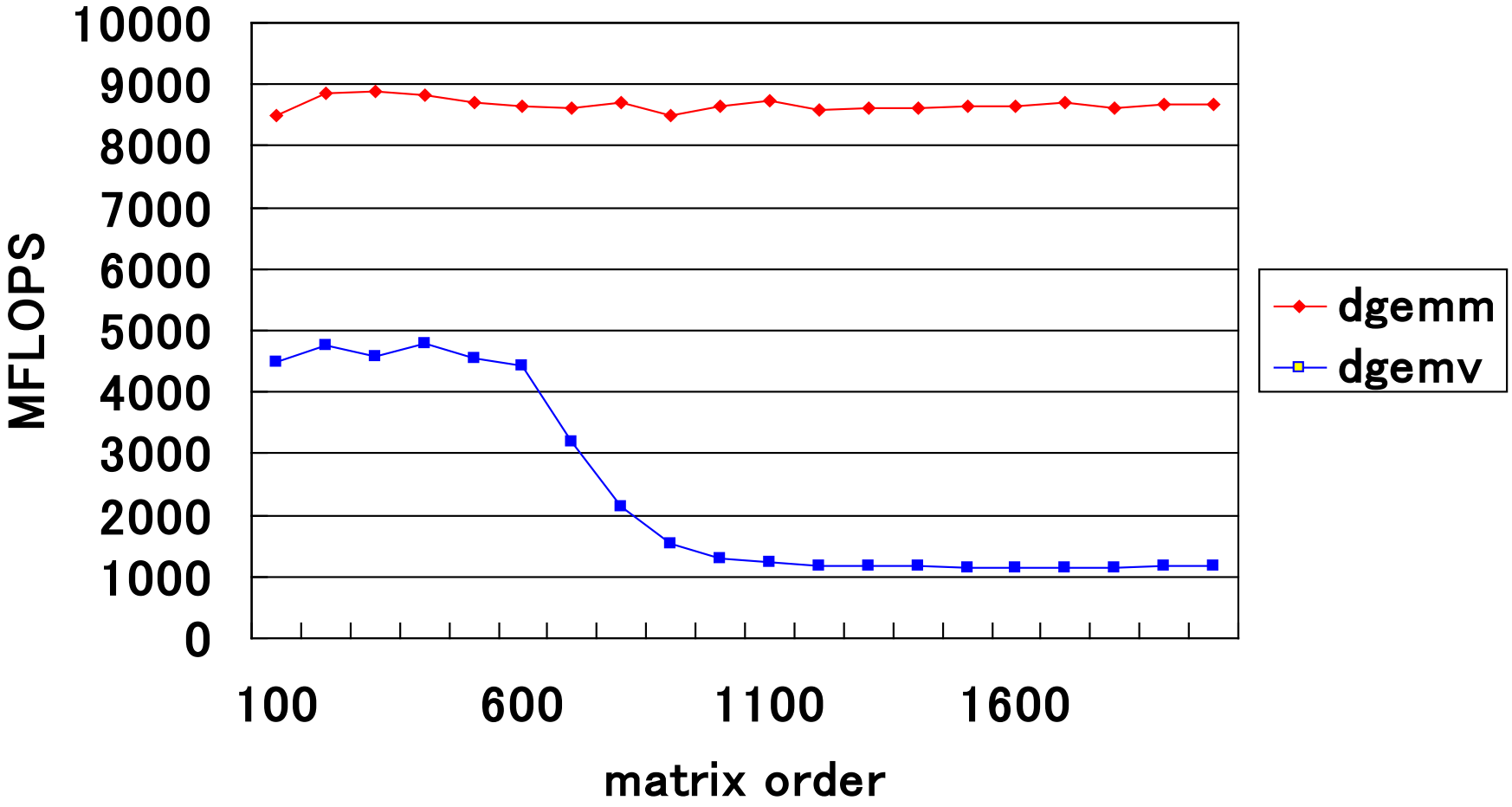
記憶階層(4/4)

- これらの傾向は，事務計算などの非数値計算には当てはまることが多いが，数値計算プログラムでは一般的ではない．
- 特に大規模な科学技術計算においては，データ参照に時間的局所性がないことが多い．
- これが，科学技術計算でベクトル型スーパーコンピュータが有利であった大きな理由．

BLASの性能（Woodcrest 2.4GHz 4MB L2 cache，Intel MKL 9.1）



BLASの性能（Woodcrest 2.4GHz 4MB L2 cache, Intel MKL 9.1）



BLASの演算回数

BLAS	ロード回数 ＋ ストア回数	浮動小数点演算回数	比 $n = m = k$
Level 1 DAXPY $y = y + \alpha x$	$3n$	$2n$	$3:2$
Level 2 DGEMV $y = \beta y + \alpha Ax$	$mn + n + 2m$	$2mn$	$1:2$
Level 3 DGEMM $C = \beta C + \alpha AB$	$2mn + mk + kn$	$2mnk$	$2:n$

Byte/Flopの概念（1/2）

- 1回の浮動小数点演算を行う際に必要なメモリアクセス量をByte/Flopで定義することができる.

```
subroutine daxpy(n, a, x, y)
real*8 a, x(*), y(*)
do i = 1, n
    y(i) = y(i) + a * x(i)
end do
```

- daxpyでは，1回のiterationにつき，2回の倍精度浮動小数点演算に対して3回の倍精度実数データ（合計24Byte）のload/storeが必要.
  - この場合，24Byte/2Flop = 12Byte/Flopとなる.
- Byte/Flop値は，小さいほど良い.

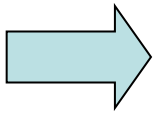
Byte/Flopの概念(2/2)

- Intel Core i7-7820X(Skylake 3.6GHz，8コア，DDR4-2666×4)では，
  - 理論ピーク性能は115.2GFlops×8コア=921.6GFlops
  - メモリバンド幅は最大85.3GB/s
  - Byte/Flop値は85.3/921.6=0.093
- daxpyでは，ワーキングセットがキャッシュの容量を超えた場合，メモリバンド幅(85.3GB/s)が律速となるので，85.3/12=7.1GFlops以上は出せない.
  - 理論ピーク性能のたった0.8%！
- メモリバンド幅が実効性能を左右することになる.

ループアンローリング（1/2）

- ループアンローリングとは，ループを展開することにより，
  - ループのオーバーヘッドを減らす
  - レジスタブロッキングを行う
- あまり展開し過ぎると，レジスタ不足や命令キャッシュミスを引き起こすので注意が必要．

```
double A[N], B[N], C;
for (i = 0; i < N; i++) {
  A[i] += B[i] * C;
}
```

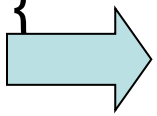


```
double A[N], B[N], C;
for (i = 0; i < N; i += 4) {
  A[i] += B[i] * C;
  A[i+1] += B[i+1] * C;
  A[i+2] += B[i+2] * C;
  A[i+3] += B[i+3] * C;
}
```

ループアンローリング（2/2）

```
double A[N][N], B[N][N],
          C[N][N], s;
for (j = 0; j < N; j++) {
  for (i = 0; i < N; i++) {
    s = 0.0;
    for (k = 0; k < N; k++) {
      s += A[i][k] * B[j][k];
    }
    C[j][i] = s;
  }
}
```

行列積の例



```
double A[N][N], B[N][N],
          C[N][N], s0, s1;
for (j = 0; j < N; j += 2)
  for (i = 0; i < N; i++) {
    s0 = 0.0;  s1 = 0.0;
    for (k = 0; k < N; k++) {
      s0 += A[i][k] * B[j][k];
      s1 += A[i][k] * B[j+1][k];
    }
    C[j][i] = s0;
    C[j+1][i] = s1;
  }
```

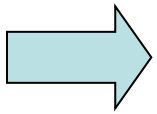
行列積を最適化した例

ループの入れ換え

- ループの入れ換えは，主にストライドの大きなメモリ参照による悪影響を軽減する手法.
- コンパイラが判断して入れ換えてくれることもある.

```
double A[N][N], B[N][N], C;
for (j = 0; j < N; j++) {
  for (k = 0; k < N; k++) {
    A[k][j] += B[k][j] * C;
  }
}
```

ループ入れ替え前



```
double A[N][N], B[N][N], C;
for (k = 0; k < N; k++) {
  for (j = 0; j < N; j++) {
    A[k][j] += B[k][j] * C;
  }
}
```

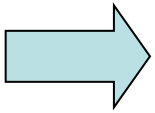
ループ入れ替え後

パディング

- 複数の配列がキャッシュの同じ位置にマッピングされてしまい，スラッシングが生じる場合に有効.
  - 特にサイズが2のべきとなる配列の場合
- 二次元配列の定義サイズを少し変えてみる.
- コンパイルオプションを指定すると行ってくれるものもある.

```
double A[N][N], B[N][N];
for (k = 0; k< N; k++) {
  for (j = 0; j < N; j++) {
    A[j][k] = B[k][j];
  }
}
```

パディングを行う前



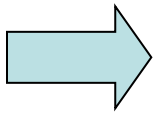
```
double A[N][N+1], B[N][N+1];
for (k = 0; k < N; k++) {
  for (j = 0; j < N; j++) {
      A[j][k] = B[k][j];
  }
}
```

パディングを行った後

ブロック化(1/2)

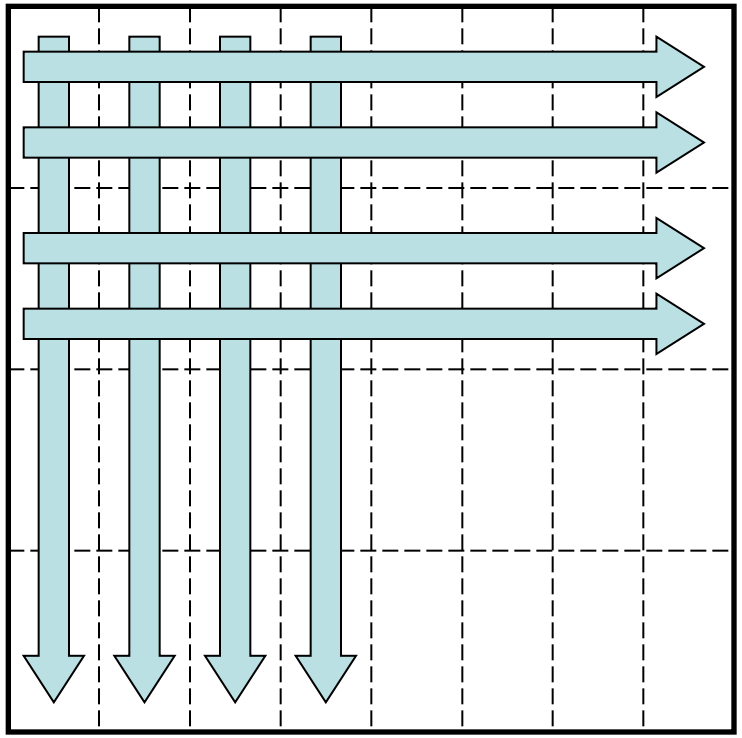
- メモリ参照を最適化するための有効な方法.
- キャッシュミスをできるだけ減らす.

```
double A[N][N], B[N][N], C;
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    A[i][j] += B[j][i] * C;
  }
}
```

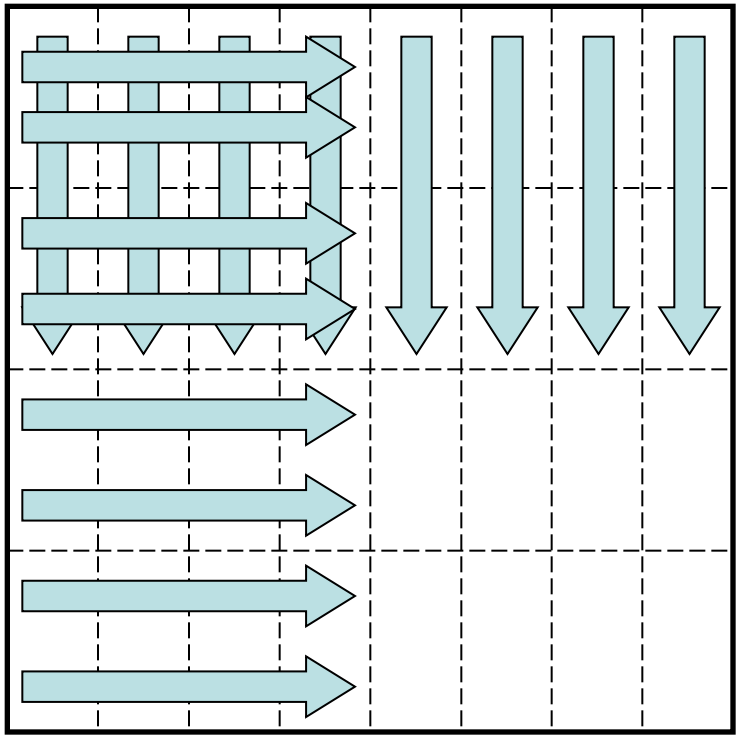


```
double A[N][N], B[N][N], C;
for (i = 0; i < N; i += 4) {
  for (j = 0; j < N; j += 4) {
    for (ii = i; ii < i + 4; ii++) {
      for (jj = j; jj < j + 4; jj++) {
        A[ii][jj] += B[jj][ii] * C;
      }
    }
  }
}
```

ブロック化(2/2)



ブロック化しない場合の
メモリアクセスパターン



ブロック化した場合の
メモリアクセスパターン

ストリーミングSIMD命令の活用

- 浮動小数点演算をより高速に処理するために，最近のプロセッサではストリーミングSIMD命令と呼ばれるものを搭載しているものが多い．
  - Intel Pentium4/XeonのSSE/SSE2/SSE3/SSE4/SSSE4/AVX/AVX2/AVX-512
  - AMD Athlonの3DNow!
  - Motorola PowerPCのAltiVec
- 最新のIntel Core i7では，AVX-512命令を活用することで，浮動小数点演算性能を32倍にすることができる．

SIMD命令の利用方法

- SIMD命令の利用方法としては，以下の方法が挙げられる.
  (1) コンパイラによりベクトル化する方法
  (2) SIMD組み込み関数を使用する方法
  (3) インラインアセンブラを使用する方法
  (4) アセンブラで「.s」ファイルを直接記述する方法
- (1)～(4)の順にコーディングが複雑になるが，性能という観点からは有利になる.

倍精度複素数の積和演算(a + b * c)をSSE3組み込み関数で記述した例

```
#include <pmmintrin.h>      /* SSE3命令を使う場合のヘッダファイル */

static __inline  __m128d ZMULADD(__m128d a, __m128d b, __m128d c)
{
  __m128d br, bi;                                  /* 128bitのデータ型で変数を定義 */

  br = _mm_movedup_pd(b);                          /* br = [b.r b.r]  実部のみを取り出す*/
  br = _mm_mul_pd(br, c);                          /* br = [b.r*c.r b.r*c.i] */
  a = _mm_add_pd(a, br);                           /* a = [a.r+b.r*c.r a.i+b.r*c.i] */
  bi = _mm_unpackhi_pd(b, b);                      /* bi = [b.i b.i]  虚部のみを取り出す */
  c = _mm_shuffle_pd(c, c, 1);                     /* c = [c.i c.r]  実部と虚部を入れ替える */
  bi = _mm_mul_pd(bi, c);                          /* bi = [-b.i*c.i b.i*c.r] */

  return _mm_addsub_pd(a, bi);                     /* [a.r+b.r*c.r-b.i*c.i a.i+b.r*c.i+b.i*c.r] */
}
```

Cで記述したZAXPY

```
typedef struct { double r, I; } doublecomplex;

void zaxpy(int n, doublecomplex a, doublecomplex *x, doublecomplex *y)
{
  int i;

  if (a.r == 0.0 && a.i == 0.0) return;

#pragma unroll(8)
#pragma vector aligned
  for (i = 0; i < n; i++) {
    y[i].r += a.r * x[i].r – a.i * x[i].i,
    y[i].i += a.r * x[i].i + a.i * x[i].r;
}
```

SSE3組み込み関数によるZAXPY

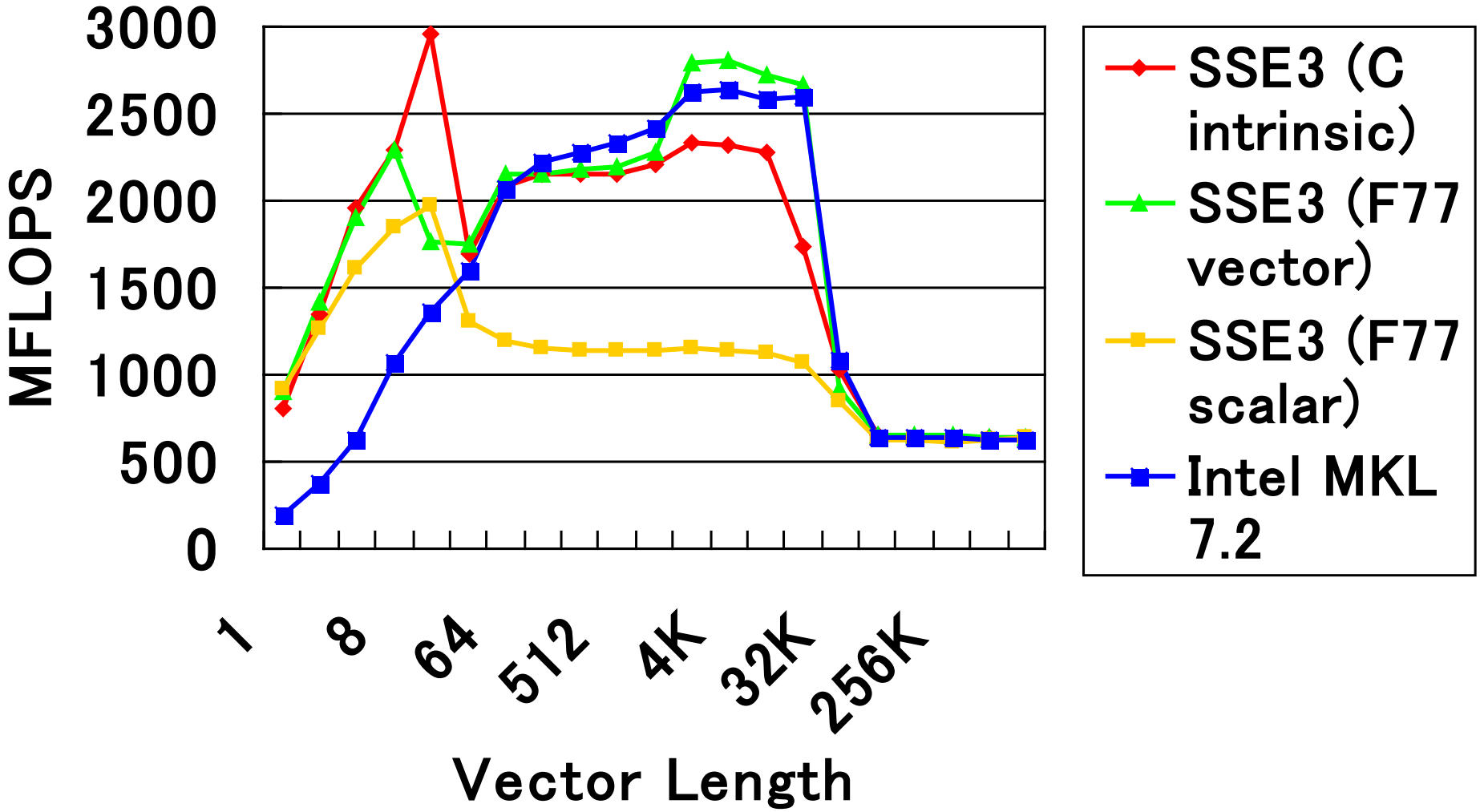
```
#include <pmmintrin.h>

typedef struct { double r, i; } doublecomplex;
__m128d ZMULADD(__m128d a, __m128d b, __m128d c);

void zaxpy(int n, doublecomplex a, doublecomplex *x, doublecomplex *y)
{
  int i;
  __m128d a0;

  if (a.r == 0.0 && a.i == 0.0) return;
  a0 = _mm_loadu_pd(&a);
#pragma unroll(8)
  for (i = 0; i < n; i++)
    _mm_store_pd(&y[i], ZMULADD(_mm_load_pd(&y[i]), a0, _mm_load_pd(&x[i])));
}
```

ZAXPYの性能
(Intel Xeon 3.4GHz, 1CPU)



課題

- 以下に示す行列積を行うプログラムを最適化し，最適化する前との実行時間を比較せよ．

```
#include <stdio.h>
#define N 1000

int main(void)
{
    static double a[N][N], b[N][N], c[N][N];
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            a[i][j] = rand(); b[i][j] = rand(); c[i][j] = rand();
        }
    }
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    return 0;
}
```