ハイパフォーマンスコンピューティング(3)

ループアンローリング（ループの展開）

- ループを展開して、１回のループで複数回分の処理をすると、ループのオーバーヘッド（条件判定など）の影響が小さくなり、処理速度が向上する場合がある。
- 展開の数を増やしていくと性能は向上していくが、上限はある。
- １回のループでループカウンタ（i, jなど）が展開の数だけ飛ぶので、端数処理をする必要がある。

```
/* アンローリングなし */
s = 0.0;
for (i = 0; i < n; i++) {
    s += a[i];
}
```

```
/* アンローリングあり（2回） */
s1 = 0.0;  s2 = 0.0;
for (i = 0; i <= n - 2; i += 2) {
    s1 += a[i];  s2 += a[i+1];
}
s = s1 + s2;
for ( ; i < n; i++) { /* 端数処理 */
    s += a[i];
}
```

【実習】アンローリングの例

```
#include <stdio.h>

int main(void)
{
  int i, n;

  printf("n = \n");
  scanf("%d", &n);

  printf("unrolling = 2\n");
  for (i = 0; i <= n - 2; i += 2) {
    printf("  i = %d\n", i);
    printf("i+1 = %d\n", i + 1);
  }
  printf("reminder\n");
  for ( ; i < n; i++) {
    printf("  i = %d\n", i);
  }
  /* 右へつづく */
```

```
  /* つづき */

  printf("unrolling = 4\n");
  for (i = 0; i <= n - 4; i += 4) {
    printf("  i = %d\n", i);
    printf("i+1 = %d\n", i + 1);
    printf("i+2 = %d\n", i + 2);
    printf("i+3 = %d\n", i + 3);
  }
  printf("reminder\n");
  for ( ; i < n; i++) {
    printf("  i = %d\n", i);
  }

  return 0;
}
```

【実習】行列積プログラムへの適用(2)

- 既に作成した行列積のプログラムにループアンローリングを適用してみよう。
- ６パターンすべてに適用するのは大変なので、６パターンの内、一番高速だったものについて適用しよう。
- 最適化オプションによっては自動的にアンローリングを行うものもあるので、はじめは最適化オプションは付けないで試してみよう。
- 展開の数を、2, 4, 8のように増やしていき、性能向上の上限を探そう。
- 最適化オプションをつけたときにどうなるかも見てみよう。

ループアンローリングの行列積への適用例

・行列積のループパターンのうち、一番高速だった(i,k,j)にループアンローリングを適用

■ jのループ2段展開

```
for(i = 0; i < m; i++){
        for(k = 0; k < p; k++){
                /* アンローリングあり（2回） */
                for(j = 0; j <= n-2; j+=2){
                        C[i][j] += A[i][k] * B[k][j];
                        C[i][j+1] += A[i][k] * B[k][j+1];
                }
                /* 端数処理 */
                for( ; j < n; j++){
                        C[i][j] += A[i][k] * B[k][j];
                }
        }
}
```

■ jのループ4段展開

```
for(i = 0; i < m; i++){
        for(k = 0; k < p; k++){
                /* アンローリングあり（4回） */
                for(j = 0; j <= n-4; j+=4){
                        C[i][j]   += A[i][k] * B[k][j];
                        C[i][j+1] += A[i][k] * B[k][j+1];
                        C[i][j+2] += A[i][k] * B[k][j+2];
                        C[i][j+3] += A[i][k] * B[k][j+3];
                }
                /* 端数処理 */
                for( ; j < n; j++){
                    C[i][j] += A[i][k] * B[k][j];
                }
        }
}
```

以降では、すべて最適化オプションの –O2 を付けることを仮定する。

キャッシュの有効利用

- メモリから取ってきたデータは、キャッシュに一時保存される。
- キャッシュ上のデータは、高速に演算可能。
- メモリからキャッシュへのデータ転送は少し時間が掛かる。
- 演算に必要なデータがキャッシュ内にある場合を「キャッシュヒット」、ない場合を「キャッシュミス」と呼ぶ。
- キャッシュミスが起きた場合は、メモリにデータを取りにいく。
- キャッシュの大きさは限られているため、それを超えるとキャッシュ内のデータが置き換わる（古いデータが消える）。

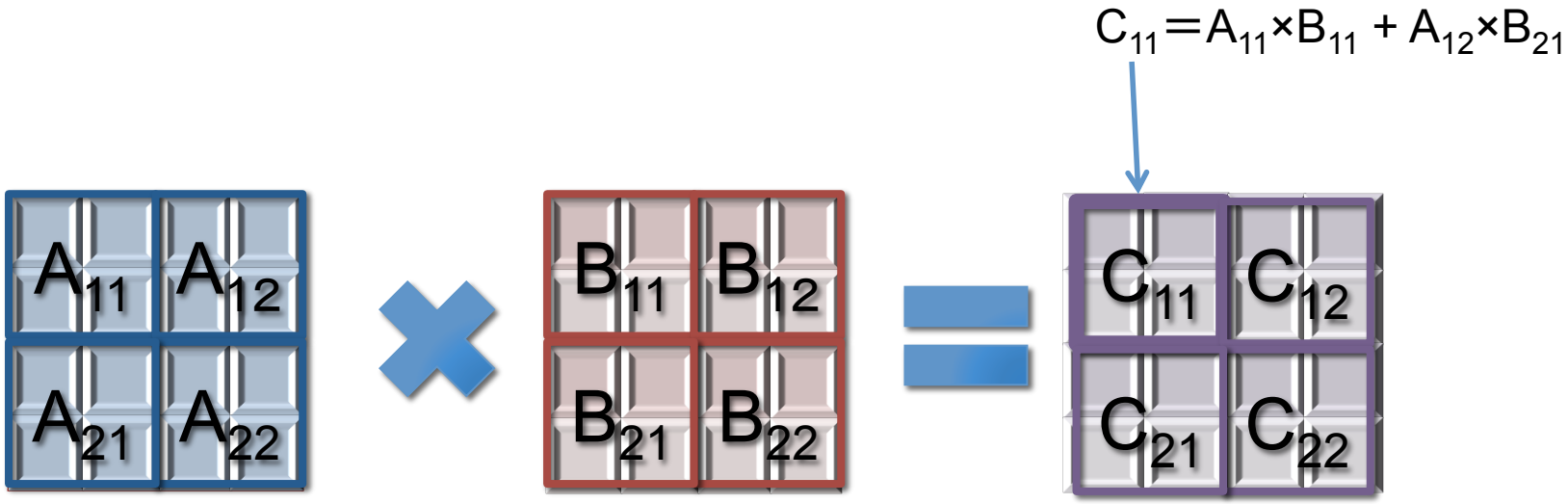
キャッシュヒット率（データの再利用性）を向上させる必要がある。

ブロック化によるアクセス局所化

- 配列データの格納方式に従ってメモリに連続アクセスをしても、キャッシュミスが多発すると性能が悪くなる。

⇒ ブロック化によって、キャッシュヒット率が高くなる（キャッシュにあるデータの再利用性が向上する）。

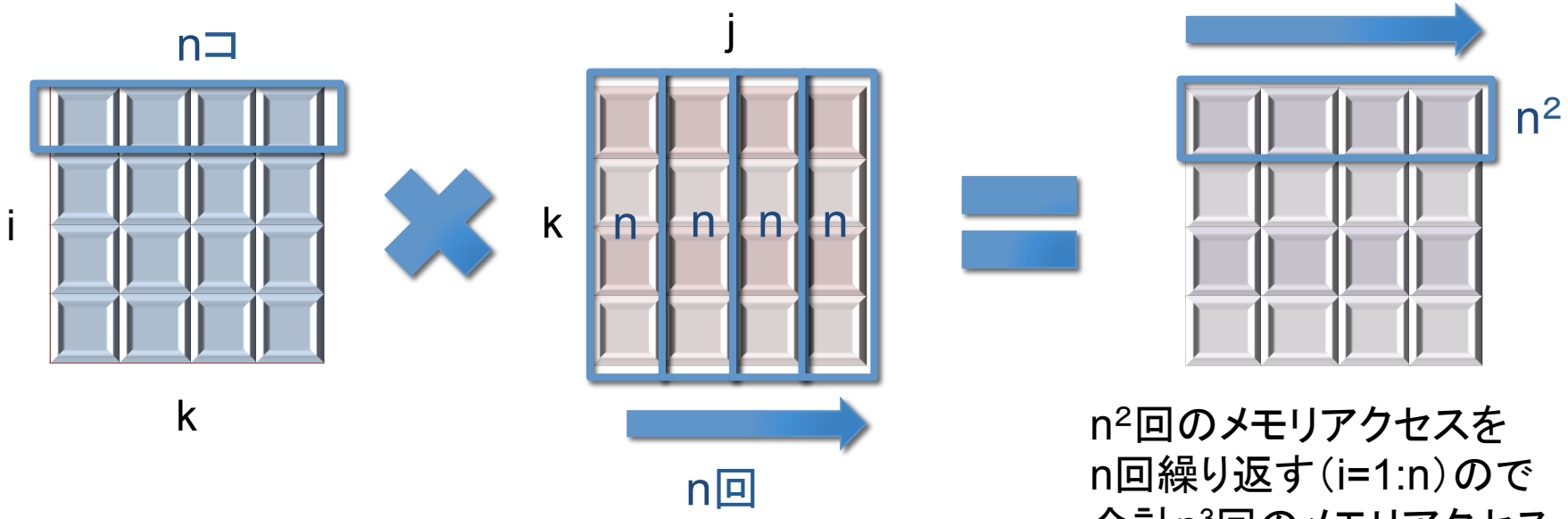
⇒ 演算が高速化される。



ブロック化によるメモリアクセスの削減

【ブロック化しない場合】
n×nの行列積の場合、n^2個のデータがキャッシュに収まらないと、
n^3回メモリアクセスが必要。

＜(i,j,k)での例＞



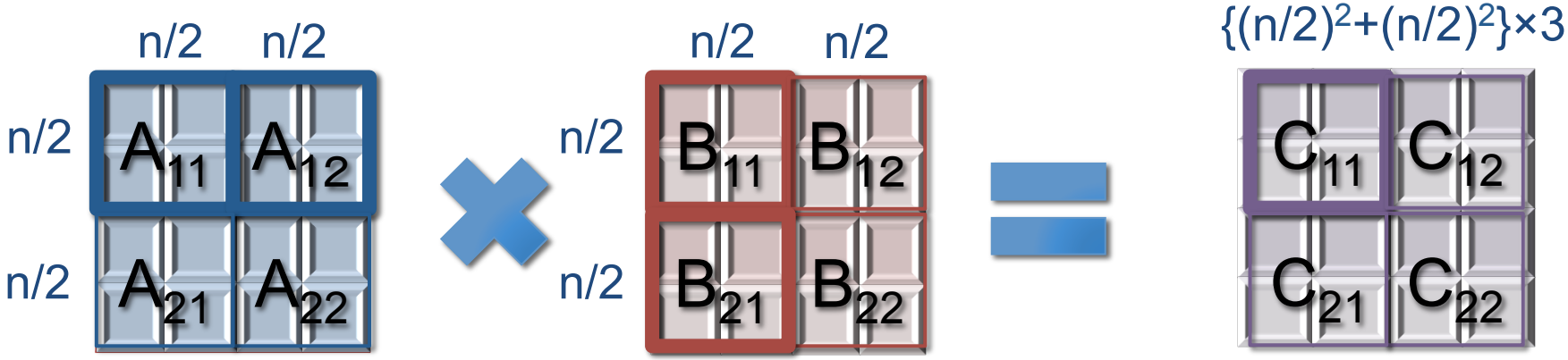
n^2回のメモリアクセスを
n回繰り返す(i=1:n)ので
合計n^3回のメモリアクセス

ブロック化によるメモリアクセスの削減

【2×2ブロックに分割した場合】

- ブロック3つ分はキャッシュに収まるとする。
- たとえば、$C_{11}=A_{11}\times B_{11} + A_{12}\times B_{21}$を計算すると、$(5/4)n^2$回のメモリアクセスが必要。
  - $A_{11}\times B_{11}$ については、A_{11}とB_{11}にそれぞれ $(n/2)^2$回、C_{11}に $(n/2)^2$回
  - $A_{12}\times B_{21}$ については、A_{12}とB_{21}にそれぞれ $(n/2)^2$回（C_{11}はすでにキャッシュにある）

⇒ 合計、$5n^2$回のメモリアクセスに削減できる。



【実習】行列積のブロック化

- nがブロック幅Mで割り切れるとき、以下のような6重ループになる。
- 既に作成した行列積のプログラムに適用し、計算時間の変化を確認しよう。
- ブロック幅Mを20, 40, 60, 80のように変化させ、最適なMを探そう。

```
 /* ブロック幅Mでブロック化 */
for ( ii=0; ii<n; ii+=M ) {
   for ( jj=0; jj<n; jj+=M ) {
      for ( kk=0; kk<n; kk+=M ) {
         /* 行列積演算 */
          for ( i=ii; i<ii+M; i++ ) {
               for ( k=kk; k<kk+M; k++ ) {
                  for ( j=jj;  j<jj+M; j++ ) {
                        C[i][j]  +=  A[i][k]  *  B[k][j];
                  }  }  }  }  }  }
```

【実習】行列積のブロック化のアンローリング

- 行列積のブロック化をしたプログラムにループアンローリングを適用してみよう。

- ブロック幅Mと展開の数（2, 4, 8のように増やす）を変更して色々試し、性能向上の上限を探そう。