BETTER ABSTRACTION FOR EFFICIENT CODE IN HPC PROGRAMS

YungYu Zhuang and Shigeru Chiba（荘　永裕、千葉　滋）
The University of Tokyo（東京大学）

ますます複雑になってきたHPCプログラム

- 特定な環境で専用コンパイラ
  - 富士通Cコンパイラ(fcc)
  - インテルCコンパイラ(icc)
  - :
- いろいろなチューニング手法
  - ダブルバッファーリング
  - メモリアライメント
  - 時間ブロッキング
  - 空間ブロッキング
  - :

細かい差で性能が大きく変わる

- コードパターンが合うとかなり速い
  - コンパイラの最適化が効くから
  - 特に専用コンパイラはそういう傾向
- 実行環境毎にコードパターンが変わる
  - チューニングは大変

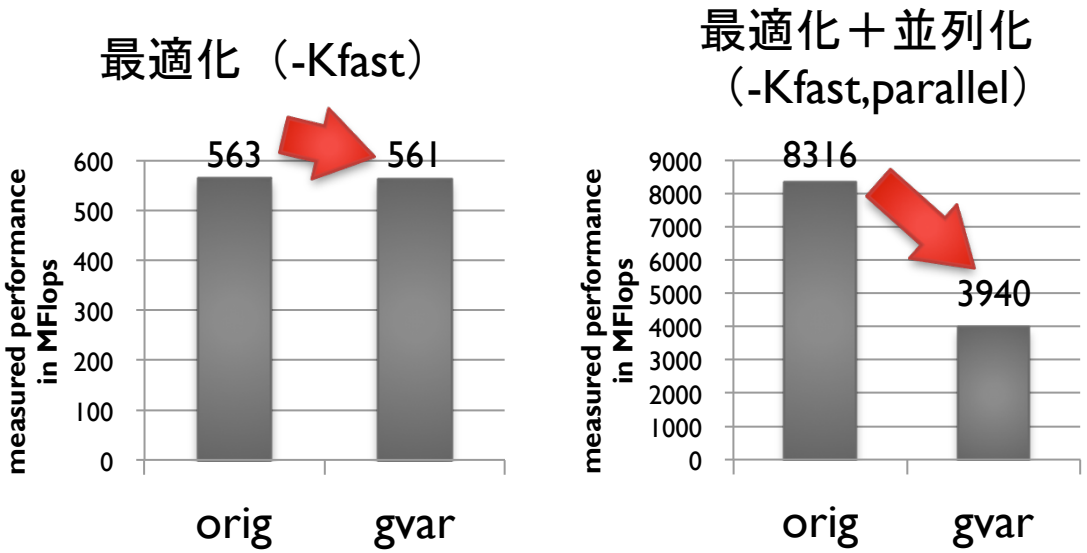
→ 今回の実行環境

- 東大のFX10の１ノード（fcc）
  SPARC64 IXfx 1.848 GHz processor with 16 cores
- 研究室のマシン（icc）
  Dual Intel Xeon E5-2687W processors
  (Sandy Bridge EP, 3.1GHz, 8 cores)

例１：
リダクションにグローバル変数を使えば

- fccの最適化と並列化を妨害する
  - FX10で姫野ベンチを例として実験してみた

```
static float gosa;

float jacobi(int nn) {
  float gosa;

  for(n=0 ; n<nn ; ++n) {
    gosa = 0.0;

    for(i=1; i<imax-1; i++)
      for(j=1; j<jmax-1; j++)
        for(k=1 ; k<kmax-1; k++) {
              :
          gosa+= ss*ss;
              :
        }
         :
  }
  return(gosa);
}
```



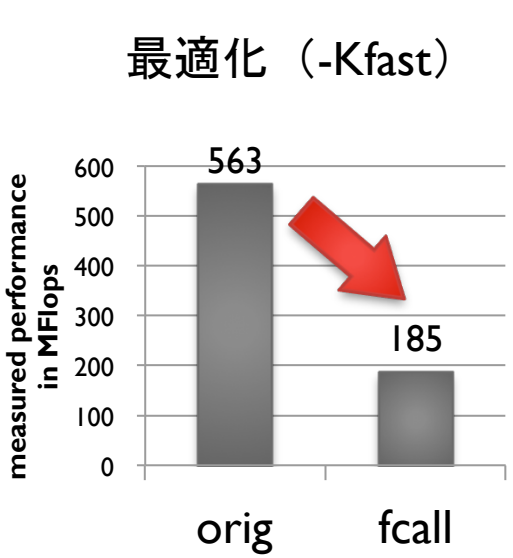
iccの場合、並列化しなければ最適化に影響しないが、
並列化すると最適化に影響

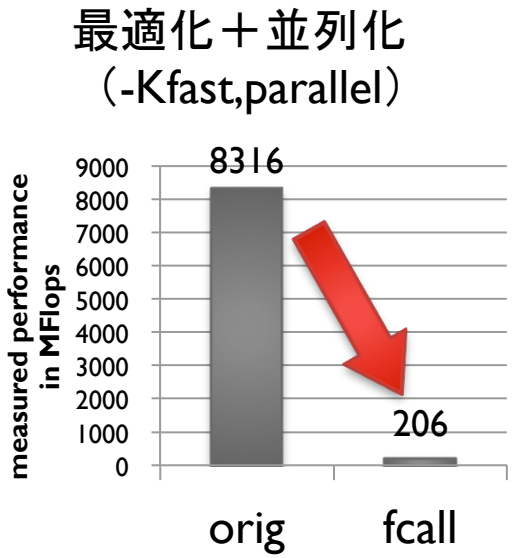
-fast: 7054 MFlops → 7062 MFlops; -fast -parallel: 6808 MFlops → 487 MFlops

例２：
ループ条件に関数の呼び出しがあれば

- fccのループ最適化が効かない
  - 同じFX10で姫野ベンチを例として

```
float jacobi(int nn) {
  float gosa;

  for(n=0 ; n<nn ; ++n) {
    gosa = 0.0;

    for(i=1; i<getimax()-1; i++)
      for(j=1; j<getjmax()-1; j++)
        for(k=1; k<getkmax()-1; k++) {
              :
        }
          :
  }
  return(gosa);
}

int getimax() { return imax; }
int getjmax() { return jmax; }
int getkmax() { return kmax; }
```





iccの場合は普通にインライン展開された
-fast: 7054 MFlops → 7055 MFlops
-fast -parallel: 6808 MFlops → 6357 MFlops

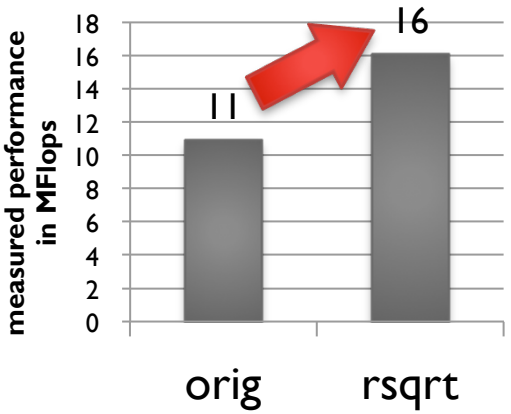
例３：sqrtで割るよりは1.0f / sqrtをかける？

- iccならrsqrtの最適化があるので速い
  - N-bodyシミュレーションの例をインテルマシンで

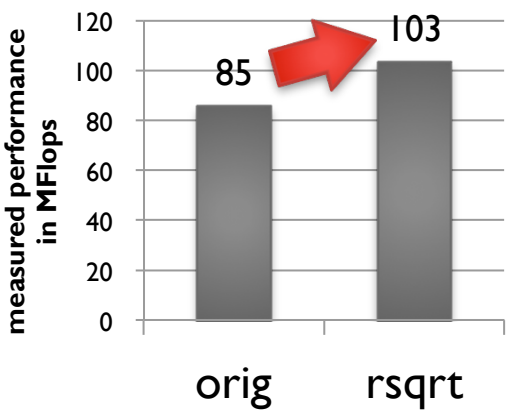
```
float a = sqrtf(rx * rx + ry * ry + rz * rz + 0.01f);
float s = pos1[i].w / (a * a * a);
```

```
float a = 1.0f / sqrtf(rx * rx + ry * ry + rz * rz + 0.01f);
float s = pos1[i].w * (a * a * a);
```

最適化（-fast）



最適化＋並列化
（-fast -parallel）

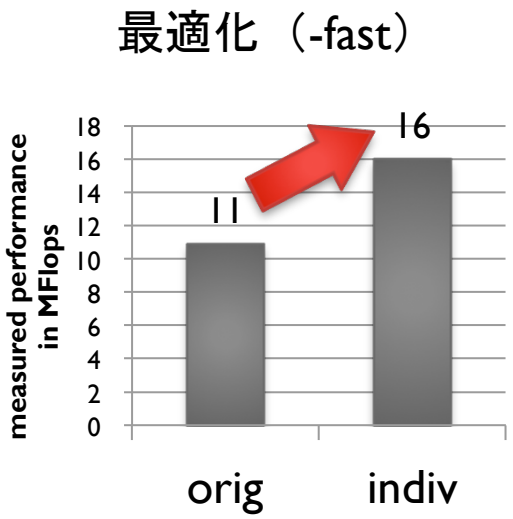


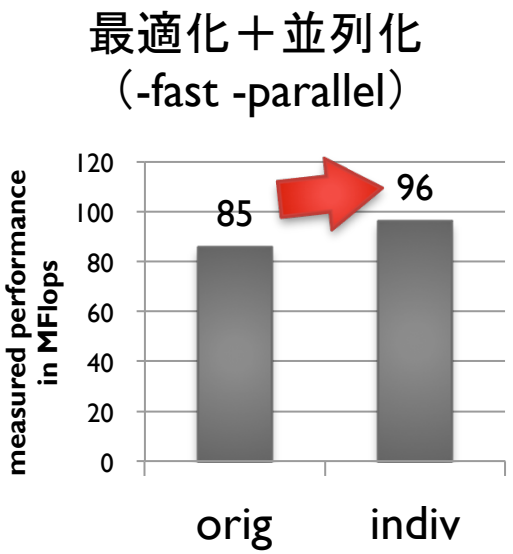
fccの場合は少しだけ速くなる
-Kfast: 6.2 GFlops → 6.5 GFlops; -Kfast,parallel: 73 GFlops → 77 GFlops

例４：
構造体の配列よりは単純な配列が速い？

- iccの場合は速くなる

```
typedef struct {
  float x, y, z, w;
} Vec4;
static Vec4 pos[N];

static float posx[N];
static float posy[N];
static float posz[N];
static float posw[N];
```





x86とsparcの違い？

fccならあまり差が出ない

-Kfast: 6.2 GFlops → 6.3 GFlops; -Kfast,parallel: 73 GFlops → 73 GFlops

提案：抽象度をあげましょう

→ 性能に関するコードを抽象化して
ほかのロジックから分離する

→ 関心事の分離

- チューニングしやすいため
  - 細かい差で性能が大きく変わるので
    コードを簡単に切り替えたい
  - 様々なコンパイラに対応したい時に

- 性能に影響しないという前提で
  - 遅くなると意味がない

具体的にどんな利点がある？

- 可読性の向上
  - 例えばvectorを使ってまとめて操作する
    - x, y, zを別々三回やる必要がないし、紙上の計算式に近い

```
pos[i].x *= 0.016f;
pos[i].y *= 0.016f;
pos[i].z *= 0.016f;
```

```
vec3_scale(&pos[i], 0.016f);
```

C++ではさらに
pos[i] *= 0.016f;
も可能です

- 性能に関するコードの切り替え
  - 例えばfcc/iccによってメモリ配置を切り替える
    - 計算ロジックを触らずに実行環境によるチューニング

```
typedef float Float4Array[4][MIMAX][MJMAX][MKMAX];
#define FLOAT4ARRAY(a, n, i, j, k) (a[n][i][j][k])
```

ほかのコードは
そのまま

```
typedef float Float4Array[MIMAX][MJMAX][MKMAX][4];
#define FLOAT4ARRAY(a, n, i, j, k) (a[i][j][k][n])
```

既存の抽象化手法を使えば？

- 関数、構造体、マクロなど
  - C言語でもよく使われる
  - ある程度のコード変換ができる
    - 可読性の良いコードを性能の良いコードに

→ 本当にうまく性能の良いコードを抽象化できるか？
  - 可能な場合、オーバーヘッドはどのぐらい？

可読性のために抽象化しよう

例：N-bodyシミュレーション

- Step 1: 似ているコードを一つの関数に
  - map1, map2: 1ラウンドの全てのbody移動
  - sum1, sum2: 一つのbodyに対する力を合算
  - 1と2の違いは対象配列だけ

```
typedef struct {
  float x, y, z, w;
} Vec4;
typedef struct {
  float x, y, z;
} Vec3;
static Vec4 pos1[N];
static Vec4 pos2[N];
```

```
void map1() {
  for (int i = 0; i < N; i++) {
    Vec3 p;
    p.x = pos1[i].x;
    p.y = pos1[i].y;
    p.z = pos1[i].z;
      :
    pos2[i].x = p.x + x * 0.016f;
    pos2[i].y = p.y + y * 0.016f;
    pos2[i].z = p.z + z * 0.016f;
  }
}
```

```
void map2() {
  for (int i = 0; i < N; i++) {
    Vec3 p;
    p.x = pos2[i].x;
    p.y = pos2[i].y;
    p.z = pos2[i].z;
      :
    pos1[i].x = p.x + x * 0.016f;
    pos1[i].y = p.y + y * 0.016f;
    pos1[i].z = p.z + z * 0.016f;
  }
}
```

```
void map(Vec4* pin,Vec4* pout) {
  for (int i = 0; i < N; i++) {
    Vec3 p;
    p.x = pin[i].x;
    p.y = pin[i].y;
    p.z = pin[i].z;
      :
    pout[i].x = p.x + x * 0.016f;
    pout[i].y = p.y + y * 0.016f;
    pout[i].z = p.z + z * 0.016f;
  }
}

map(pos1, pos2);
map(pos2, pos1);
```

さらに抽象化してみよう

- Step 2: vector構造体とそのための関数を使う
  - x, y, z三回の操作を関数にまとめる
  - 関数vec3_addやvec3_scaleなどを用意する

```
void map(Vec4* pin,Vec4* pout) {
  for (int i = 0; i < N; i++) {
    Vec3 p;
    p.x = pin[i].x;
    p.y = pin[i].y;
    p.z = pin[i].z;
      :
    pout[i].x = p.x + x * 0.016f;
    pout[i].y = p.y + y * 0.016f;
    pout[i].z = p.z + z * 0.016f;
  }
}
```

```
Vec3* vec3_set(Vec3* obj,Vec3* val) {
  obj->x = val->x;
  obj->y = val->y;
  obj->z = val->z;
  return obj;
}

Vec3* vec3_add(Vec3* obj,Vec3* val) {
  obj->x += val->x;
  obj->y += val->y;
  obj->z += val->z;
  return obj;
}
```

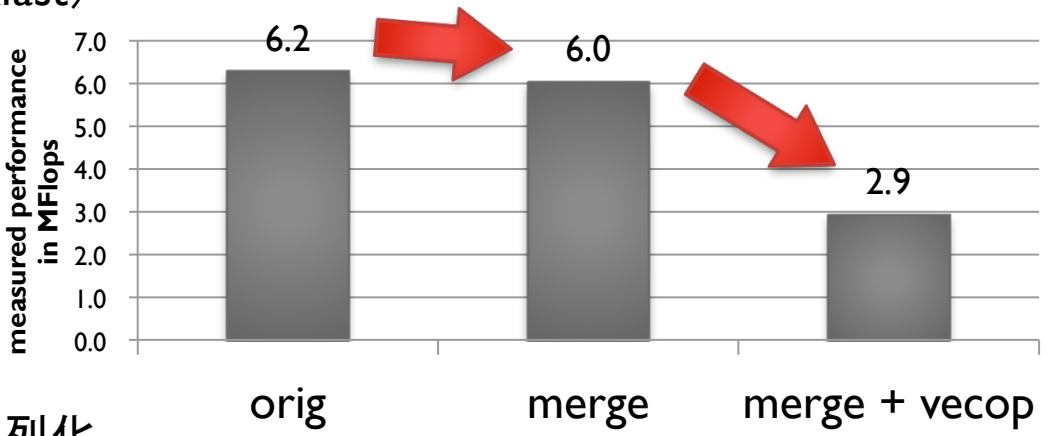
```
Vec3* vec3_scale(Vec3* obj, float s) {
  obj->x *= s;
  obj->y *= s;
  obj->z *= s;
  return obj;
}

void map(Vec3* pin,Vec3* pout) {
  for (int i = 0; i < N; i++) {
    Vec3 p;
    vec3_set(&p, &pin[i]);
      :
    vec3_set(&pout[i],
             vec3_add(&p,
                      vec3_scale(&v,
                                 0.016f)));
  }
}
```

抽象化するほど遅くなる…

- merge: 似ているコードを関数でマージする
- vecop: vectorとその関数を使う

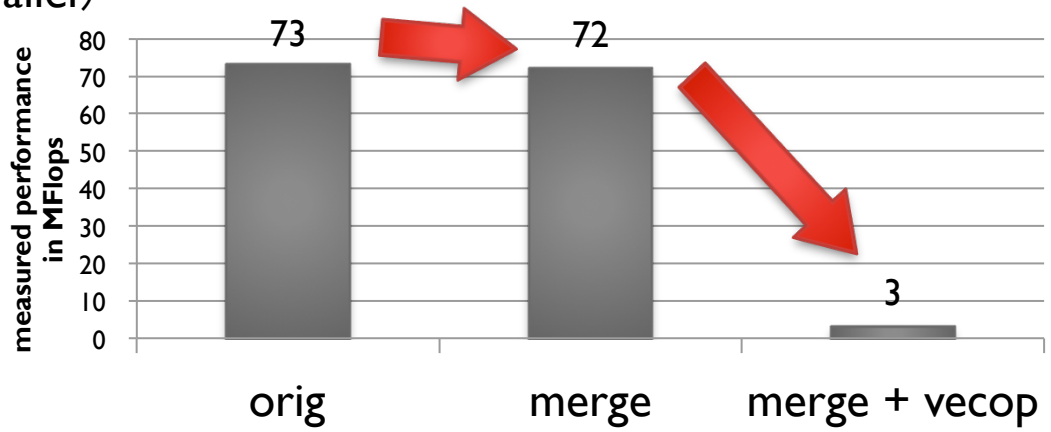
最適化（-Kfast）

FX10/fccで



最適化＋並列化
（-Kfast,parallel）

FX10/fccで



iccでは
-fast:
11 GFlops
→ 12 GFlops
→ 9 GFlops

-fast -parallel:
85 GFlops
→ 87 GFlops
→ 15 GFlops

関数の代わりにマクロを使えば

- 同じように書くのは難しい
  - 複数の行がある
  - 返り値は使えない
    （gcc拡張ならできるがインライン関数になる）

```
void map(Vec3* pin,Vec3* pout) {
  for (int i = 0; i < N; i++) {
    Vec3 p;
    vec3_set(&p, &pin[i]);
      :
    vec3_set(&pout[i],
                vec3_add(&p,
                            vec3_scale(&v,
                                          0.016f)));
  }
}
```

```
#define map(pin, pout) do {\
  for (int i = 0; i < N; i++) {\
    Vec3 p;\
    vec3_set(p, pin[i]);\
      :
    Vec3 t = {0, 0, 0};\
    vec3_scale(t, vel[i], 0.016f);\
    vec3_add(pout[i], p, t);\
  }\
} while (0)
```

```
Vec3* vec3_add(Vec3* obj,Vec3* val) {
  obj->x += val->x;
  obj->y += val->y;
  obj->z += val->z;
  return obj;
}
```

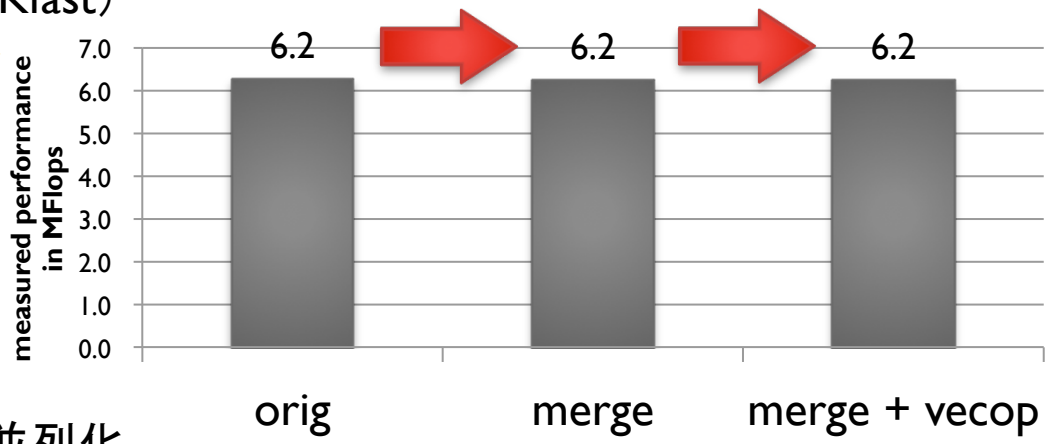
```
#define vec3_add(ret, obj, val) do {\
  ret.x = obj.x + val.x;\
  ret.y = obj.y + val.y;\
  ret.z = obj.z + val.z;\
} while (0)
```

関数の代わりにマクロを使えば

- 抽象化のオーバーヘッドを削減できる

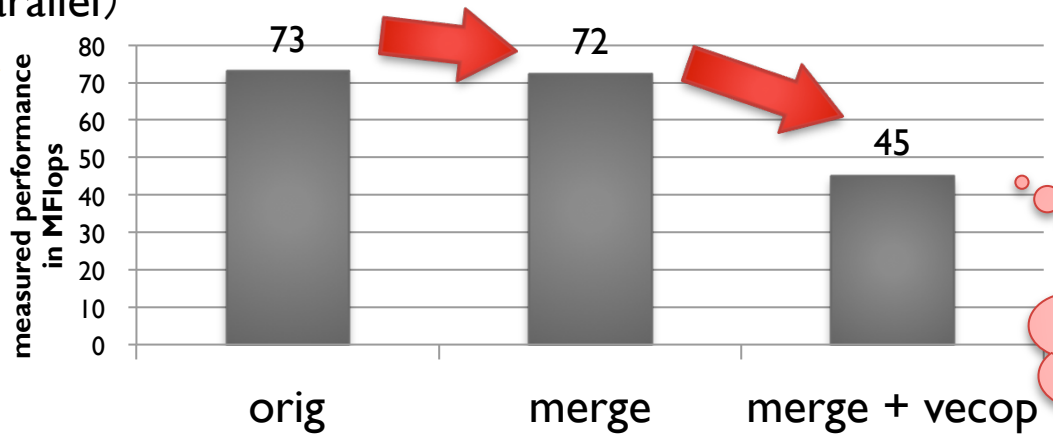
最適化（-Kfast）

FX10/fccで



最適化＋並列化
（-Kfast,parallel）

FX10/fccで



iccでは
-fast:
11 GFlops
→ 12 GFlops
→ 12 GFlops

-fast -parallel:
85 GFlops
→ 70 GFlops
→ 75 GFlops

やはり同じコードにならない

切り替えのために抽象化しよう

例：姫野ベンチ

- Step 1: 4次元配列をtypedefで定義する
  - 簡単にindex順をswapしてチューニング
  - 例：FX10では(n, i, j, k)より(i, j, k, n)が速い

富士通からのアドバイス

- Step 2: kernel部分を抽出しよう
  - 全体をフレームワークとして再利用したい
  - 関数ポインターを使うしかない…

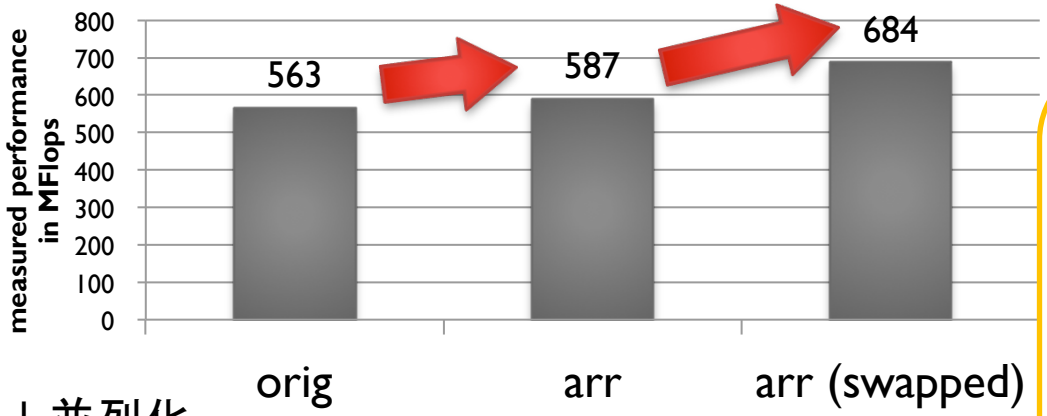
どれだけ遅い？と思って実験してみた

簡単に切り替えてチューニング

- arr: typedefで配列を定義する
- arr (swapped): typedef定義の中のindex順をswapした

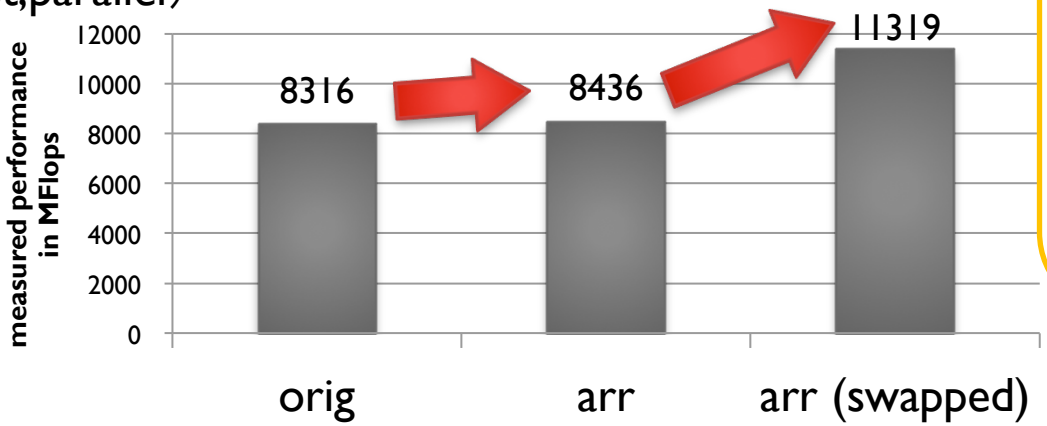
最適化（-Kfast）

FX10/fccで



最適化＋並列化（-Kfast,parallel）

FX10/fccで



x86とsparcの違い？

iccの場合swapしないほうが速い

-fast:
6313 MFlops
→ 6321 MFlops
→ 2951 MFlops

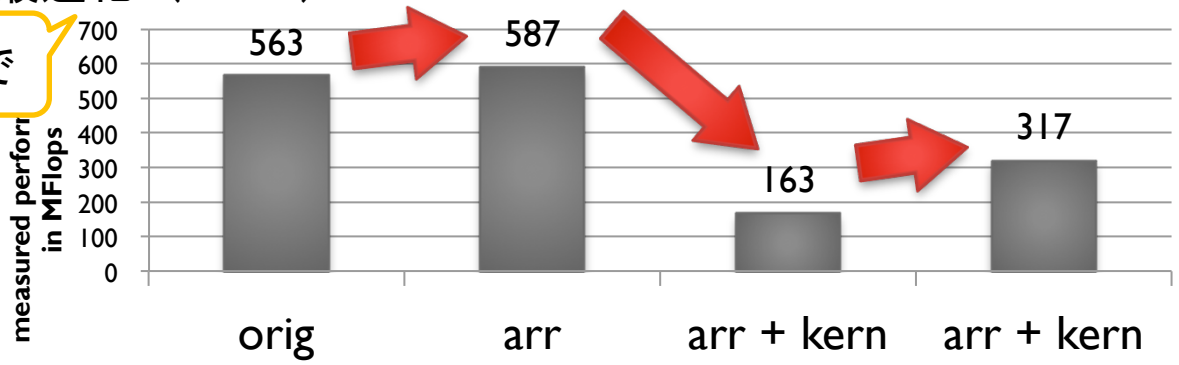
-O3 -parallel
-mcmodel=medium:
14785 MFlops
→ 15675 MFlops
→ 12062 MFlops

さらにkernelを抽出したら…

- かなり遅くなる → もう性能の良いコードではない！

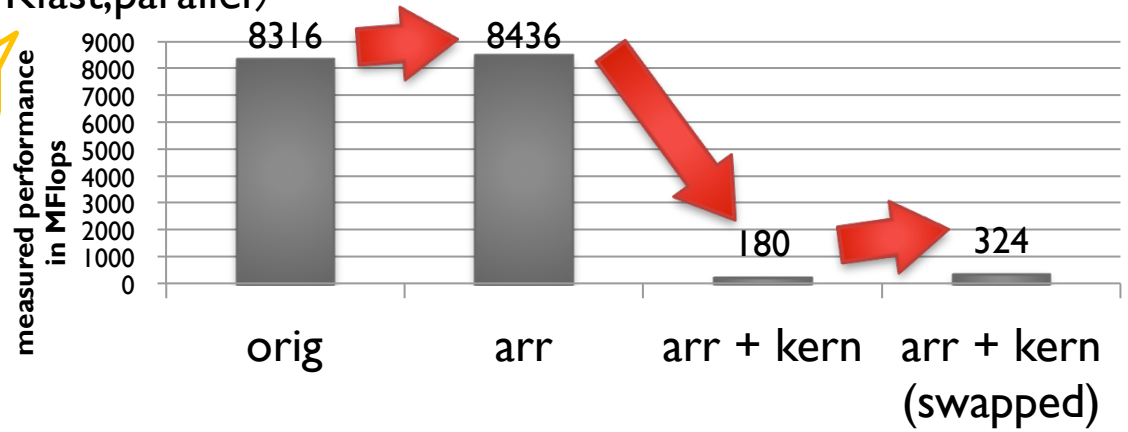
FX10/fccで

最適化（-Kfast）



FX10/fccで

最適化＋並列化（-Kfast,parallel）



iccの場合は影響が小さい

-fast:
6313 MFlops
→ 6321 MFlops
→ 6222 MFlops
→ 2951 MFlops

-O3 -parallel
-mcmodel=medium:
14785 MFlops
→ 15675 MFlops
→ 15263 MFlops
→ 12112 MFlops

既存手法は使えない

- 性能の良いコードを切り替えたいのに、抽象化すると性能がまったく出ない
  - マクロやtypedefは良いが、できることは限られる
- コードパターンにマッチしないとコンパイラの最適化が効かない
  - 受け入れられるパターンの制限が厳しい

→ より賢いコード変換が必要

メタプログラミング

- コードを生成するコードを書く
  - 例えばC++のテンプレート
    やLispのマクロなど
  - 様々強力なコード変換ができる
    - 抽象度の高いコードから性能の良いコードに
- しかしメタレベルで考えるのは大変

ドメイン特化言語（DSL）

- 特定の領域に特化した言語
  - より明確に記述できる
    - 表現できるものが限られる
  - 専用コンパイラでコードを変換する

- でも専用の開発環境の開発は面倒
  - プログラムのためにスペックを変更したら専用ツールを作り直す必要がある

内部DSLをメタプログラミングで

- 内部DSLなら使いやすい
  - ライブラリのような使い方
  - ホスト言語と一緒に使える
  - ホスト言語の開発環境も使える
- メタプログラミングで抽象度の高いコードを望ましいパターンに

→ Bytespresso
  - 我々が開発しているフレームワーク
  - 今後の課題：メタプログラミングをより簡単に

まとめ

- 性能の良いコードを抽象化すべし
  - チューニングのために
  - 細かい差で性能が大きく変わる
    - いくつかの例を挙げた（fccとicc）
- 既存の抽象化手法を使ってみた
  - 性能の良いコードは抽象化されたら
    もう同じものではない
- DSLとメタプログラミングを一緒に使えば
  - 任意のコード変換ができる？