小ポイントFFTの マルチコア上での自動並列化手法

古山 祐樹[1]　見神 広紀[1]　木村 啓二[1]　笠原 博徳[1]

概要：高速フーリエ変換 (FFT) は，ディジタル信号処理や画像圧縮など様々な分野で使用される非常に応用性の高い計算アルゴリズムである．その中でも，LTE 等のベースバンド処理で用いられる小ポイントの FFT プログラムは，データ転送や制御のオーバーヘッドを伴う専用ハードウェアを使用しにくく，マルチコア上での並列化の要求が高まっている．本稿では，そのような小ポイントの FFT プログラムに対しコンパイラによる自動並列化及び，false sharing 回避を目的としたキャッシュ最適化を適用し，データキャッシュを持つ種々の共有メモリ型マルチコアアーキテクチャに向けて低オーバーヘッドな並列化コードを生成する自動並列化手法を提案する．提案手法を OSCAR 自動並列化コンパイラに実装し，32 ポイントから 256 ポイントまでの小ポイント FFT を並列化し，8 つの SH4A コアを集積した情報家電用マルチコアプロセッサ RP2 上で性能評価を行ったところ，256 ポイントの FFT プログラムで，逐次プログラムに対し 2 コア並列化で 1.97 倍，4 コア並列化で 3.9 倍というスケーラブルな速度向上を得ることが出来た．また，FFT と同様にバタフライ演算を行う高速アダマール変換のプログラムにも同手法を適用し評価を行い，256 ポイントのプログラムで 2 コア並列化で 1.91 倍，4 コア並列化で 3.32 倍という高い速度向上が得られ，提案手法の有用性が確認された．

Automatic Parallelization of Small Point FFT on Multicore Processor

Yuuki Furuyama[1]　Hiroki Mikami[1]　Keiji Kimura[1]　Hironori Kasahara[1]

Abstract: Fast Fourier Transorm (FFT) is one of the most frequently used algorihtms in many applications including digital signal processing and image processing to compute Descrite Fourier Transform (DFT). Although small size FFT programs must be used in baseband signal processing such as LTE and so on, it's difficult to use special hardwares like DSPs for computing such a small problem because of their relatively large data transfer and control overhead. This paper proposes an automatic parallelization method to generate parallelized programs with low overhead for small size FFTs suited for shared memory multicore processor by applying cache optimization to avoide false sharing between cores. The proposed method has been implemented in OSCAR automatic parallelizing compiler, parallelized small point FFT programs from 32 points to 256 points and evaluated them on RP2 multicore processor having 8 SH-4A cores. It achieved 1.97 times speedup on 2 SH-4A cores and 3.9 times speedup on 4 SH-4A cores in a 256 points FFT program. In addition to the FFT programs, the proposed approach is applied to Fast Hadamard Transform (FHT) which has similar computation to the FFT. The results are 1.91 times speedup on 2 SH-4A cores and 3.32 times speedup on 4 SH-4A cores. It shows effectiveness of the proposed method and easiness of applying the method to many kinds of programs.

[1] 早稲田大学
Waseda University

1. はじめに

高速フーリエ変換 (FFT) は，ディジタル信号処理や画像圧縮など様々な分野で使用される非常に応用性の高い計算アルゴリズムである．その中でも，LTE 等のベースバンド処理 [1] で用いられる小ポイントの FFT プログラムは，実行サイクル数が小さいため, データ形式変換や入出力データ転送など処理オーバーヘッドを伴う DSP などの専用ハードウェアの使用は難しく，マルチコアプロセッサを用いた並列化の要求が高まっている．マルチコアによる並列化は特定ハードウェア開発が不要なため，短期間，低コストでの実行速度向上の実現が可能である．

FFT の高速化はそのアルゴリズムの重要性から数多くの研究がなされており，並列化アプローチも様々なものが提案されている．Airoldi, R らは 9 コア集積のメッシュ型 NoC(Network on Chip) を用い，コア間のデータフローを工夫し転送オーバーヘッドを抑える手法で 64 ポイント FFT を高速化している [2]．また Long Chen らは，手動でのレジスタ・リネーミングや命令スケジューリングを行い，IBM Cyclops-64 プロセッサに特化した並列 FFT の構成を提案している [3]．しかしこのような特定のアーキテクチャに向けた手動での最適化には，FFT アルゴリズムの性質や，アーキテクチャの深い理解が必要不可欠であり，さらにプログラムのポータビリティが失われてしまうという問題も生じる．一方，FFT の自動チューニングという観点では，Franchetti, F らにより，行列として表した DFT 式に線形変換を繰り返し行い，マルチコアに最適な FFT を自動生成する手法が提案されているが [4][5]，生成できるプログラムがあらかじめ決められた信号処理の計算式に限られるため，多様なバリエーションが存在する FFT では生成可能なコードの柔軟性に欠けてしまう．

そのような背景を踏まえ，本稿では小ポイント FFT プログラムに対しコンパイラによる自動並列化及びキャッシュ最適化を適用し，データキャッシュを持つ種々の共有メモリ型マルチコアアーキテクチャに向けて低オーバーヘッドな並列化コードを生成する自動並列化手法を提案する．本手法の特徴は，FFT のメモリアクセスパターンを考慮したキャッシュ最適化にあり，FFT の並列実行時に生じる false sharing を回避するためのバッファ変数を自動で挿入し，それと同時に FFT のバタフライ演算で使用する配列のインデックスを適切に変換した並列化コードを生成する点である．

本稿ではこの提案手法を説明するために，2 章で FFT の概要を，3 章で FFT の特徴を考慮した並列化手法について述べる．4 章では 3 章で述べた並列化手法を OSCAR コンパイラで行う方法について述べ，5 章で提案手法の性能評価を述べる．最後に 6 章で本稿のまとめを述べる．

2. 高速フーリエ変換

本章では高速フーリエ変換の概要と，バタフライ演算における特徴的なメモリアクセスの性質について述べる．

2.1 高速フーリエ変換の概要

高速フーリエ変換 (FFT) は, $O(N)$ の計算量である離散フーリエ変換 (DFT) を，$O(NlogN)$ のオーダーで計算する高速なアルゴリズムである．N 個の複素入力データ系列 $a_{k(0\leq k\leq N-1)}$ から出力データ系列 $c_{k(0\leq k\leq N-1)}$ を得るための DFT の定義を式 (1) に示す.

$$c_k = \sum_{j=0}^{N-1} a_j \exp(-2\pi ijk/N) \qquad (1)$$

式 (1) は 2 のべき乗の入力データをもつ場合，その入力データを偶数列・奇数列に分解し，さらに k の前半と後半で式を分けると，式 (2), 式 (3) のように変形することができる．尚，$N' = N/2$，$e_j = a_{2j}$，$o_j = a_{2j+1}$，$\omega_{jk} = \exp(-2\pi ijk/N')$ とする．

$$c_k = \sum_{j=0}^{N/2-1} e_j\omega_{jk} + \exp(-2\pi ik/N) \sum_{j=0}^{N/2-1} o_j\omega_{jk} \qquad (2)$$

$$c_{k+N'} = \sum_{j=0}^{N/2-1} e_j\omega_{jk} - \exp(-2\pi ik/N) \sum_{j=0}^{N/2-1} o_j\omega_{jk} \qquad (3)$$

式 (2), 式 (3) は, 式 (1) の N 点の DFT が, 偶数列の $2/N$ 点の DFT と，奇数列の $2/N$ 点の DFT に回転因子と呼ばれる複素指数関数 $\exp(-2\pi ik/N)$ を掛けたものの足し合わせで表現できることを表している．上述した一連の DFT の分解を更に再帰的に行っていき，DFT のポイント数が 1 になるまで繰り返し行うアルゴリズムが Cooley-Tukey の FFT のアルゴリズムである [6]．ポイント数 8 の FFT の演算をシグナルフローグラフとして表すと**図 1** のようになり，バタフライ演算と呼ばれるたすき掛け部分の演算が，再帰的な処理として複数のステージに分割されて処理されていく流れがわかる．図 1 からわかるように，Cooley-Tukey FFT は入力データの配列用領域を中間のステージの演算結果の格納領域としても利用することができる．このような FFT の実装は in-place 型の FFT と呼ばれる [7]．

上記の FFT は各ステージで 2 点のデータのバタフライ演算を行なう基数 2 の FFT であるが，一度に 4 点のバタフライ演算を行い回転因子乗算回数を減らす基数 4 の FFT や，基数 4 の FFT と基数 2 の FFT を組み合わせた混合基数 FFT も存在する [8]．

2.2 バタフライ演算におけるメモリアクセスの性質

図 1 で見たように，各ステージで実行されるバタフライ

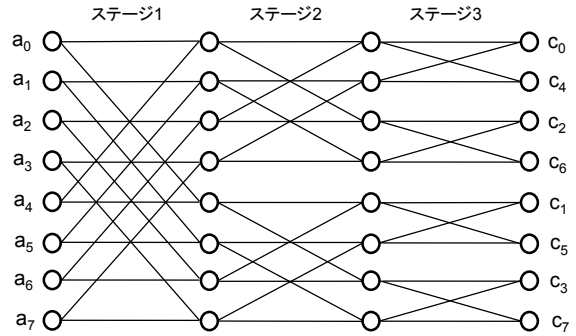

図 1 ポイント数 8 の FFT のシグナルフローグラフ

演算のメモリアクセスのストライドは，FFT のステージが進む度に狭まっていく特徴があり，N ポイントの基数 r-FFT の，ステージ $k(k = 1, 2, ... \log_2 N)$ でのメモリアクセスのストライド s_k は $s_k = N/r^k$ と表される．ここで，128 ポイント FFT のステージ間でストライドが変化する様子を**図 2** に，またその擬似コードを**図 3** にそれぞれ示す．

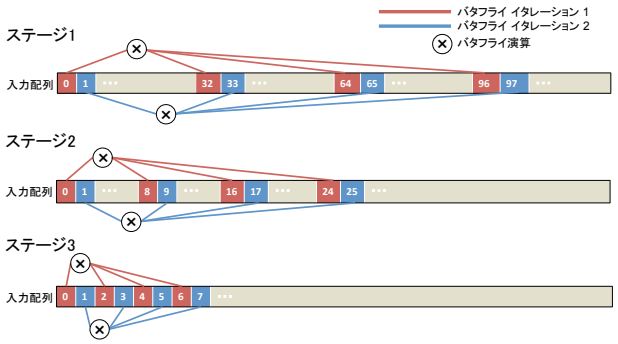

図 2 128 ポイントの基数 4FFT のメモリアクセスの性質

```
/* ステージ 1 */
for (i = 0; i < 1; i++)
  for (j = 0; j < 32; j++)
    butterfly(a[j+0],a[j+32],a[j+64],a[j+96]);

/* ステージ 2 */
for (i = 0; i < 4; i++)
  for (j = 0; j < 8; j++)
    butterfly(a[i*8+j+0],a[i*8+j+8],a[i*8+j+16],a[i*8+j+24]);

/* ステージ 3 */
for (i = 0; i < 16; i++)
  for (j = 0; j < 2; j++)
    butterfly(a[i*2+j+0],a[i*2+j+2],a[i*2+j+4],a[i*2+j+6]);
```

図 3 128 ポイントの基数 4FFT の各ステージの擬似コード

図 2 の FFT では，ステージ 1 ではストライド 32，ステージ 2 ではストライド 8，ステージ 3 ではストライド 2 と，バタフライ演算で使用するデータのメモリアクセスがステージ毎に狭まっていく様子がわかる．さらに図 3 のコードから，各ステージのバタフライ演算のループは，ループイタレーション間で使用するデータに重なりはなく，全てのステージで DOALL ループとなっていることがわかる．

3. 小ポイント FFT の並列化手法

本章では，FFT の並列化時の各コアへのデータ分割，並列化時に生じる false sharing の問題点，及びそれを解決するためのキャッシュ最適化手法について述べる．

3.1 FFT の並列化時のデータ分割

2.2 節で述べた通り，FFT の各ステージのループは DOALL ループとなっているが，この DOALL ループをキャッシュミス率を低減させつつ各プロセッサにスケジューリングするためのデータの分割方法として，ブロックサイクリック-ブロック混合分割手法を提案する．

一般的にブロック分割は，入力データ配列の先頭からプロセッサコア数分に均等に分割し，各コアはそのデータに対してバタフライ演算を行う．より大きなブロックでデータを分割することにより，メモリアクセスの空間的局所性が高まり，キャッシュミス回数を低く抑えることが可能になる．しかし 128 ポイント FFT のステージ 1 は 32 のストライドであり，先頭から 32 要素毎のブロックに分けることはできない．

そこで，提案手法では先頭から 8 要素のブロック分割をプロセッサ毎に繰り返すブロック・サイクリックな分割とする．なお，このブロック・サイクリックな分割は 128 ポイントの FFT に限らず，FFT の 1 番目のステージではそのような分割が有効である．この混合ブロック分割の利点は，FFT の基数とプロセッサコア数が同じ場合には，2 番目のステージ以降は各プロセッサコアのキャッシュ内でのみ計算が行われ，他のプロセッサとの通信はステージ 1-2 間の一度しか行われない点である．**図 4** に，ブロックサイクリック-ブロック混合分割した場合の 128 ポイント FFT の 4 プロセッサコアへのデータの割り当て方を示す．

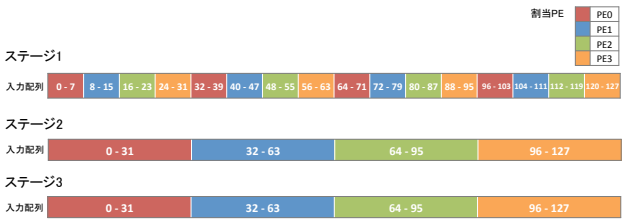

図 4 128 ポイント FFT の各ステージをブロックサイクリック-ブロック混合分割で分割した場合

この場合のプロセッサコア間のデータ転送数 n は，FFT のポイント数を N，プロセッサコア台数を p とした場合，1 つのコアあたり式 (4) で表される．

$$n = \frac{(p-1)N}{p^2} \tag{4}$$

3.2 in-place 型 FFT の並列化時に生じる false sharing

Cooley-Tukey の in-place 型 FFT では，入力用の配列領域とバタフライ演算結果を格納する出力用配列領域が同一であるため，並列実行時に false sharing による性能劣化が生じる可能性がある．すなわち，複数プロセッサコアがそれぞれ別々のデータを書き換えているのにも関わらず，お互いのデータが同一キャッシュラインに存在するためにキャッシュのハードウェアコヒーレンシが働き，コア毎の invalidate 要求とそれに伴うライトバックが発生し，さらに再度データにアクセスした際に本来は無用なキャッシュミスが生じる．ブロック分割で並列化した FFT で false sharing が起きない条件は，分割された一つのブロックサイズがキャッシュラインサイズの倍数となることである．従って，ポイント数 N，基数 r の FFT で，ステージ $k(k = 1, 2, ... \log_2 N)$ でのストライドを N/r^k，FFT の入力データの要素当たりのバイト数を b，プロセッサ台数を p，プロセッサのキャッシュラインサイズを c とすると，false sharing が発生しないステージの条件は，ステージ 1 では式 (5)，ステージ 2 以降では式 (6) が成り立つ場合である．

$$\frac{N}{r^k \times p} \times b = c \times j \ \ (j : \mathbb{Z}) \tag{5}$$

$$\frac{N}{r^{k-1}} \times b = c \times j \ \ (j : \mathbb{Z}) \tag{6}$$

ステージ 1 とステージ 2 以降で別々の条件式が必要になる理由は，3.1 節で述べた通り，ブロック分割の FFT の 1 番目のステージではブロック・サイクリックな分割になるためである．式 (5) と式 (6) を，入力データの要素サイズ 2Byte の 32/64/128/256 のポイント数の FFT に適用すると，各 FFT の並列化時に false sharing が発生するか否かは**表 1** のように表される．ここでキャッシュラインサイズ 32Byte を想定する．

表 1 FFT の並列化時に false sharing が発生するか否か

ポイント数	ステージ	2PE 並列化	4PE 並列化
32 ポイント	ステージ 1	× (false sharing あり)	×
	ステージ 2	○ (false sharing なし)	×
	ステージ 3	○	×
64 ポイント	ステージ 1	×	×
	ステージ 2	○	○
	ステージ 3	○	○
128 ポイント	ステージ 1	○	×
	ステージ 2	○	○
	ステージ 3	○	○
	ステージ 4	○	○
256 ポイント	ステージ 1	○	○
	ステージ 2	○	○
	ステージ 3	○	○
	ステージ 4	○	○

表 1 から，上記の条件では 128 ポイントまでの FFT の 4 プロセッサ並列化で必ず false sharing が発生し，一方 256 ポイントの FFT ではどのステージでも false sharing は発生しないことがわかる．

3.3 false sharing 回避のためのバッファ変数挿入

3.2 節で示した通り，キャッシュラインサイズや並列化するプロセッサ台数によっては並列化時に false sharing が発生するステージが生じる．そこでその false sharing を回避するために，false sharing が発生するステージへの中間バッファ変数を挿入する．バッファ変数はプロセッサ毎に用意し，false sharing が発生するステージでバタフライ演算結果をそのバッファ変数に格納する．格納する順番は要素番号が連番に並ぶような形にする．**図 5** に，64 ポイントの基数 4-FFT を 4PE で並列化した場合のバッファ変数挿入の例を示す．図 5 の場合，ステージ 1 のバタフライ演算結果の出力時に false sharing が発生するため，ステージ 1 の出力用にプロセッサ毎にバッファ変数を挿入し，そのバッファ変数に対して出力を行う．そしてステージ 2 では，そのバッファ変数を入力配列として使用し，出力は元のオリジナルの配列に書き戻す．

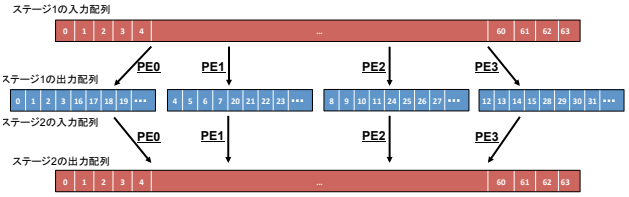

図 5 false sharing 回避のためのバッファ変数挿入

バッファ変数を挿入するにあたって，FFT のバタフライ演算の出力配列の添字を書き換える必要がある．ポイント数 N，基数 r の FFT で，ステージ $k(k = 1, 2, ... \log_2 N)$ でのストライドを N/r^k，バタフライ演算のイタレーションを i，プロセッサ台数を p とした時，ステージ k で false sharing が発生した場合のステージ k の出力配列の添字変換は式 (7) として表される．式 (7) は矢印を挟んで左辺がステージ k の入力配列の添字を，右辺が変換されるステージ k の出力配列の添字を表す．

$$\begin{aligned}
i &\Rightarrow (i \bmod \frac{N}{r^k \times p}) \\
i + \frac{N}{r^k} &\Rightarrow (i \bmod \frac{N}{r^k \times p}) + \frac{N}{r^k \times p} \\
i + \frac{2N}{r^k} &\Rightarrow (i \bmod \frac{N}{r^k \times p}) + \frac{2N}{r^k \times p} \\
&\cdots \\
i + \frac{(r-1)N}{r^k} &\Rightarrow (i \bmod \frac{N}{r^k \times p}) + \frac{(r-1)N}{r^k \times p}
\end{aligned} \tag{7}$$

式 (7) の添字変換は，false sharing が発生したステージ

k の出力配列の添字と，ステージ $k+1$ の入力配列の添字の両者に適用する必要がある．

4. OSCAR コンパイラによる FFT プログラムの自動並列化

本章では，上記で述べた FFT の並列化を実際に行う OSCAR コンパイラによる自動並列化手法の流れについて述べる．OSCAR 自動並列化コンパイラは並列性を抽出するパス内において，粗粒度並列処理，中粒度並列処理，近細粒度並列処理の三つの粒度の並列性を同時に抽出するマルチグレイン並列処理を特徴としたコンパイラである [9]．

また，今回並列化の対象とする基数 4・基数 2 の混合基数 FFT プログラムの擬似コードを**図 6** に示す．

```
void fft(Int16* fftin_i, Int16* fftin_q) {
  /* 基数 4 FFT */
  for(i=0;i<num_rx4;i++){
    for(j=0;j<num_rot;j++){
      for(k=0;k<num_btf;k++){
        butterfly(fftin_i[a], fftin_i[b], fftin_i[c], fftin_i[d]);
      }
    }
    num_rot<<=2;
    num_btf>>=2;
  }

  /* 基数 2 FFT */
  for(i=0;i<num_rx2;i++){
    for(j=0;j<num_btf;j++){
      butterfly(fftin_i[a], fftin_i[b]);
    }
  }
}
```

図 6 基数 4・基数 2 の混合基数 FFT のプログラム例

4.1 FFT プログラムのステージへの分解

対象とする図 6 の FFT プログラムは，各ステージがループのイタレーションとして表現されているため，バタフライ演算を表すループの回転数がイタレーション毎に変化する等，ループの解析が困難なものとなっている．また，false sharing 回避のためのバッファ変数を挿入しようにも，全てのステージが 1 つのループで書かれている以上，そのような最適化が難しい．そこで，各ステージ毎のループが別々のものになるよう，入力プログラムのリストラクチャリングを行う．具体的には，FFT のポイント数をパラメータとして渡し，その値を元に FFT を表すループの階層でループアンローリングを行う．アンローリング後，パラメータとして渡された FFT のポイント数を元に，定数伝搬・定数畳み込みを行うことで，ステージで使用する各パラメータの具体値を決定させる．各ステージへの分解とステージ毎のパラメータの決定が行われることで，その後のループ毎の解析・最適化が行い易くなる．

4.2 各ステージにおける false sharing の検知

false sharing 検知は式 (5) と式 (6) の条件式を用いて，各ステージ毎に false sharing の判定を行う．条件式の通り，false sharing の判定材料となる要素は，FFT のポイント数・FFT の入力データの要素当たりのバイト数・プロセッサコア数・プロセッサのキャッシュラインサイズである．

4.3 false sharing 回避のためのバッファ変数挿入・配列インデックス書き換え

前述のフローで false sharing 発生と判定されたステージに対しては，false sharing 回避のためのバッファ変数の挿入が行われる．バッファ変数の挿入は false sharing が発生するステージ k の出力配列と，ステージ $k+1$ の入力配列に対して行われ，同時に式 (7) に示した添字の変換式を用いて，バッファ配列の添字が変換される．

OSCAR コンパイラにより，バッファ変数挿入と配列インデックスの書き換えが行われたプログラム例を**図 7** に示す．図 7 のプログラム中の NEWA00001，NEWA00002 配列が挿入されたバッファ変数である．

```
/* 基数 4FFT ステージ 1 */
for (j = 0; j < 32; j++) {
  /* 入力配列からロード */
  x0re=fftin_i[a]+b2re;
  x0im=fftin_q[a]+b2im;

  butterfly(x0re, x0im, ...);

  /* 出力配列にストア */
  NEWA00001[k_NO76 % 8 + k_NO76 / 8 * 32]=(Int16)(b0re>>1);
  NEWA00002[k_NO76 % 8 + k_NO76 / 8 * 32]=(Int16)(b0im>>1);
}
/* 基数 4FFT ステージ 2 */
for (j = 0; j < 4; j++) {
  for (k = 0; k < 8; k++) {
    /* 入力配列からロード */
    x0re=NEWA00001[8 * j_NO77 + k_NO78]+b2re;
    x0im=NEWA00002[8 * j_NO77 + k_NO78]+b2im;

    butterfly(x0re, x0im, ...);

    /* 出力配列にストア */
    ps2_fftin_i[index_a]=(Int16)(b0re>>1);
    ps2_fftin_q[index_a]=(Int16)(b0im>>1);
  }
}
....
```

図 7 OSCAR コンパイラによる FFT へのバッファ変数挿入例

5. 性能評価

本稿で提案する FFT 並列化方式を OSCAR 自動並列化コンパイラに実装し，情報家電用マルチコアプロセッサ RP2 上にて性能評価を行った．

5.1 情報家電用マルチコアプロセッサ RP2

情報家電用マルチコア RP2 は，SH-4A コアを 2 つのクラスタに 4 コアずつ，計 8 コアを 1 つのチップ内に集積したホモジニアスマルチコアプロセッサである．RP2 プロセッサのブロック図を**図 8** に示す．キャッシュのコヒーレンシはクラスタ内のみハードウェアコヒーレンシが機能し，クラスタ間のコヒーレンシはソフトウェアによって担保する必要がある．各プロセッサコアはラインサイズ 32Byte のデータキャッシュ 16KB と，命令キャッシュ 16KB を持つ．また RP2 プロセッサの各コアの動作周波数は動的に制御することが可能となっており，75，150，300，600MHz で動作可能である．尚，評価では全てのコアを 600MHz で駆動させ評価を行った．

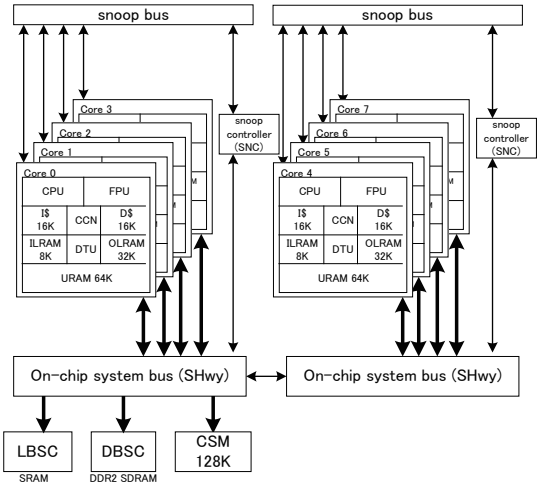

図 8 情報家電用マルチコア RP2 のブロック図

5.2 評価対象プログラム

評価対象の FFT プログラムは富士通 (株) からご提供戴いた基数 4・基数 2 の混合基数 FFT プログラムを 32 ポイントから 256 ポイントまで用意したものである．本プログラム例は図 6 に示した．ただし，32 ポイントの FFT は逐次プログラムのサイクル数がとても小さいため，2 コアまでの並列化となっている．

また，FFT と同様にバタフライ演算を行う高速アダマール変換 (FHT) に対しても，64 ポイントから 256 ポイントまでのプログラムを用意し性能評価を行った．高速アダマール変換のプログラム例を**図 9** に示す．

5.3 高速フーリエ変換の性能評価結果

32 ポイントから 256 ポイントまでの FFT プログラムの RP2 上での並列実行時の速度向上率を**図 10** に示す．図 10 より，128 ポイント FFT では 2 コア並列化で 1.96 倍，4 コア並列化で 3.22 倍，256 ポイント FFT では 2 コア並列化で 1.97 倍，4 コア並列化で 3.9 倍というスケーラブルな速度向上が得られている．256 ポイント FFT では 2 コア，4

```
void fht(Int16* CQI_MtrFHT, Int32 u4_num_stg) {
  /* 基数 4 FHT */
  for (i = 0; i < num_rx4; i++) {
    for (j = 0; j < num_rot; j++) {
      for (k = 0; k < num_btf; k++) {
        index_a = a+k+(num_btf+(btf_stp2>>1))*j;
        index_b = b+k+(num_btf+(btf_stp2>>1))*j;
        index_c = c+k+(num_btf+(btf_stp2>>1))*j;
        index_d = d+k+(num_btf+(btf_stp2>>1))*j;
        x0 = CQI_MtrFHT[index_a] + CQI_MtrFHT[index_c];
        x2 = CQI_MtrFHT[index_a] - CQI_MtrFHT[index_c];
        x1 = CQI_MtrFHT[index_b] + CQI_MtrFHT[index_d];
        x3 = CQI_MtrFHT[index_b] - CQI_MtrFHT[index_d];
        CQI_MtrFHT[index_a] = (Int16)(x0 + x1);
        CQI_MtrFHT[index_b] = (Int16)(x0 - x1);
        CQI_MtrFHT[index_c] = (Int16)(x2 + x3);
        CQI_MtrFHT[index_d] = (Int16)(x2 - x3);
      }
    }
  }
  /* 基数 2 FHT */
  for (i = 0; i < num_rx2; i++) {
    for (j = 0; j < num_btf; j++) {
      x0 = CQI_MtrFHT[a + 2 * j] + CQI_MtrFHT[b + 2 * j];
      x1 = CQI_MtrFHT[a + 2 * j] - CQI_MtrFHT[b + 2 * j];
      CQI_MtrFHT[a + 2 * j] = (Int16)x0;
      CQI_MtrFHT[b + 2 * j] = (Int16)x1;
    }
  }
}
```

図 9 高速アダマール変換のプログラム例

コア並列化共に false sharing は発生せず，ステージ間で中間バッファを介した計算にはなっていないためとても高い速度向上を得られた．

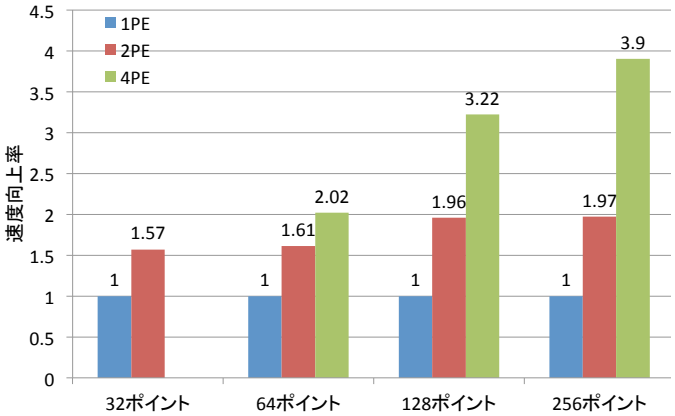

図 10 高速フーリエ変換の RP2 における並列処理性能

また 128 ポイント FFT の並列実行時の 1 コアあたりのサイクル数やキャッシュミス回数などの評価結果を**表 2** に示す．表 2 のキャッシュミス回数に注目すると，4 コア並列化時の load メモリアクセス (キャッシュミス) 回数は 14 回だが，これはステージ 1 での初期データ load 回数 $4 \times 2(re/im)$ + ステージ 2 での中間バッファからの load 回数 $2(re/im) \times 3(other\ 3\ cores) = 14$ と等しく，理論上

最小のメモリアクセス回数で済ませることが出来ている．

表 2 128 ポイント高速フーリエ変換の RP2 における並列処理性能

コア数	サイクル数	処理時間 (μ s)	$ミス回数 (load)	$ミス回数 (store)
1PE	19764	32.94	16	0
2PE	10080	16.8	11	8
4PE	6132	10.22	14	6

5.4 高速アダマール変換の性能評価結果

64 ポイントから 256 ポイントまでの高速アダマール変換 (FHT) プログラムの RP2 上での並列実行時の速度向上率を**図 11** に示す．また 128 ポイント FHT の並列実行時の 1 コアあたりのサイクル数やキャッシュミス回数などの評価結果を**表 3** に示す．

高速アダマール変換は基本的な加減算の演算のみで構成されており，表 3 に示す通り逐次ではとても小さな実行サイクル数にもかかわらず，128 ポイント FHT では 2 コア並列化で 1.61 倍，4 コア並列化で 2.01 倍，256 ポイント FHT では 2 コア並列化で 1.91 倍，4 コア並列化で 3.32 倍というとても高い速度向上が得られた．

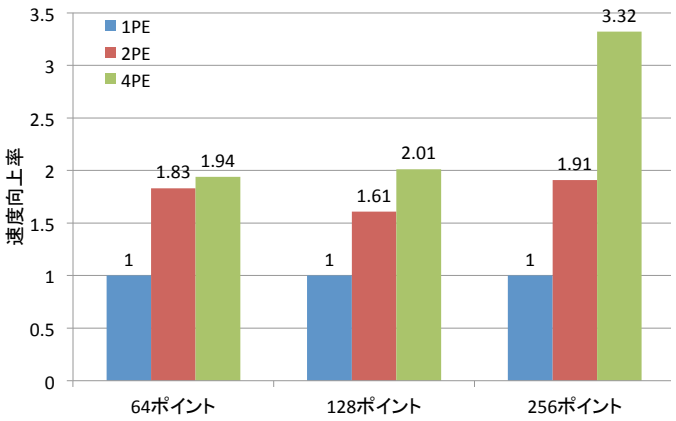

図 11 高速アダマール変換の RP2 における並列処理性能

表 3 128 ポイント高速アダマール変換の RP2 における並列処理性能

コア数	サイクル数	処理時間 (μ s)	$ミス回数 (load)	$ミス回数 (store)
1PE	4632	7.72	9	0
2PE	2880	4.8	11	0
4PE	2304	3.84	12	3

5.5 false sharing 回避バッファ変数挿入有無による性能比較

提案手法である，OSCAR コンパイラによる false sharing 回避用のバッファ変数を挿入する最適化を行った場合と，バッファ変数挿入の最適化はせず，OSCAR コンパイラによりループ並列化のみ行った場合の性能比較を行った．FFT と高速アダマール変換 (FHT) におけるバッファ変数挿入有無の性能向上率を**図 12** に示す．図 12 は，FFT と FHT の各 PE 数におけるバッファ変数挿入無しのときの性能を 1 としたときの，バッファ変数挿入有りのときの性能向上を表す．なお，false sharing が発生する 32 ポイント (2PE)，64 ポイント (2PE/4PE)，128 ポイント (4PE) の FFT と，64 ポイント (2PE/4PE)，128 ポイント (4PE) の FHT のみ示している (表 1 参照)．

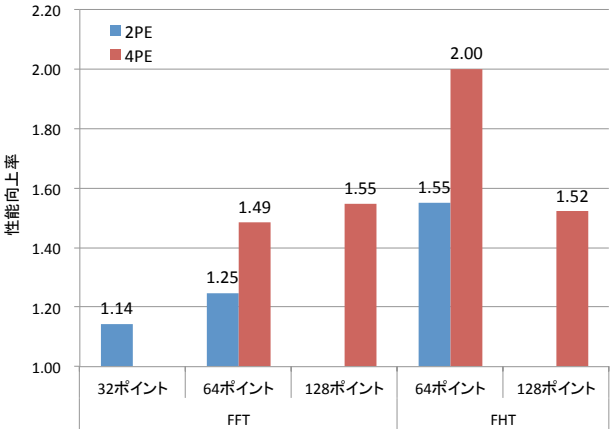

図 12 false sharing 回避用バッファ変数挿入の効果

高速フーリエ変換の方では，バッファ変数を挿入することで，挿入しない場合に対し，32 ポイントの 2 コア並列化では 14%の性能向上，64 ポイントの 2 コア並列化では 25%，4 コア並列化では 49%の性能向上，128 ポイントの 4 コア並列化では 55%の性能向上が見られた．また高速アダマール変換に関しては，バッファ変数挿入によって，64 ポイントの 2 コア並列化では 55%，4 コア並列化では 100%の性能向上，128 ポイントの 4 コア並列化では 52%の性能向上が見られ，両プログラムとも false sharing を回避するバッファ変数挿入の最適化により，高い性能向上が得られることがわかった．

6. まとめ

本稿では，SMP 上でキャッシュ効率を高め，低オーバーヘッドな並列化コードを生成する小ポイント FFT の自動並列化手法である，ブロックサイクリック-ブロック分割混合並列化手法を提案した．小ポイント FFT ではメモリアクセス回数を最小限に抑える必要があり，提案手法では自動で false sharing を回避し，SMP マルチコアに最適な並列化プログラムを生成する．提案手法を OSCAR 自動並列

化コンパイラに実装し，32 ポイントから 256 ポイントまでの FFT と，64 ポイントから 256 ポイントまでの高速アダマール変換 (FHT) に対し RP2 プロセッサ上で性能評価を行った結果，256 ポイントの FFT プログラムでは，2 コア並列化で 1.97 倍，4 コア並列化で 3.9 倍，そして 256 ポイントの FHT プログラムでは，2 コア並列化で 1.91 倍，4 コア並列化で 3.32 倍というスケーラブルな速度向上が得られ，同手法の有効性と応用性を確認することが出来た．

謝辞 本稿での評価プログラムである FFT プログラムや，様々な有益な情報をご提供戴いたアクセスネットワークテクノロジ株式会社 佐々木啓様と佐藤聡様に深く感謝致します．

参考文献

1) Dung-Rung Hsieh., De-Jhen Huang., Jen-Yuan Hsu., Chieh-Yu Kao., Ming-Che Lin., Chun-Nan Liu. and Pangan Ting.: Baseband design and software-defined-radio implementation for LTE femtocell, *Control Conference (ASCC), 2013 9th Asian*, pp. 1–6 (2013).
2) R. Airoldi., F. Garzia. and J. Nurmi.: Implementation of a 64-point FFT on a Multi-Processor System-on-Chip, *Research in Microelectronics and Electronics, 2009. PRIME 2009. Ph.D.*, pp. 20–23 (2009).
3) Long Chen., Ziang Hu., Junmin Lin. and G.R. Gao.: Optimizing the Fast Fourier Transform on a Multi-core Architecture, *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–8 (2007).
4) F. Franchetti., Y. Voronenko. and M. Puschel.: FFT Program Generation for Shared Memory: SMP and Multicore, *SC 2006 Conference, Proceedings of the ACM/IEEE*, pp. 51–51 (2006).
5) Lingchuan Meng., Yevgen Voronenko., Jeremy R. Johnson., Marc Moreno Maza., Franz Franchetti. and Yuzhen Xie.: Spiral-generated Modular FFT Algorithms, *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, New York, NY, USA, ACM, pp. 169–170 (2010).
6) 金田康生 (編): 並列数値処理 -高速化と性能向上のために-, コロナ社 (2010).
7) K.L. Heo., J.H. Baek., M.H. Sunwoo., B.G. Jo. and B.S. Son.: New in-place strategy for a mixed-radix FFT processor, *SOC Conference, 2003. Proceedings. IEEE International [Systems-on-Chip]*, pp. 81–84 (2003).
8) Eun Ji Kim. and Myung Hoon Sunwoo.: High speed eight-parallel mixed-radix FFT Processor for OFDM systems, *Circuits and Systems (ISCAS), 2011 IEEE International Symposium on*, pp. 1684–1687 (2011).
9) KEIJI KIMURA., TAKESHI KODAKA., MOTOKI OBATA. and HIRONORI KASAHARA.: Multigrain Parallel Processing on OSCAR Chip Multiprocessor, *IPSJ SIG Notes*, Vol. 2002, No. 112, pp. 29–34 (2002).