メモリ・アクセスの局所性を最適化するループ再構成法

津田　健[†]　山本孝伸[†]　田中利彦[†]
五島正裕[†]　森　眞一郎[†]　富田眞治[†]

現在のRISCプロセッサが持つ高い演算性能は（1次）キャッシュに強く依存しており、キャッシュに収まらないような巨大なデータを扱う大規模数値処理などの応用ではその性能は著しく悪化する。本稿では、Phasingと呼ぶループ再構成手法について述べる。Phasingでは、$n(n \geq 3)$重ループの内の最内ループにストリップ・マイニングを施し、その結果得られる2重ループの外側のループを全体の最外に移動する。Phasingを適用すると、キャッシュ・ヒット率は、問題のサイズやキャッシュのブロック・サイズに依らず、キャッシュの容量だけによって決まる、ある十分に高い値となる。例えば内積形式LU分解では、キャッシュ容量が16KBのとき、そのヒット率は99.6%となる。キャッシュ容量が増加すれば、ヒット率は更に向上する。

A loop restructuring technique to optimize memory access locality

Takeshi Tsuda,[†] Takanobu Yamamoto,[†] Toshihiko Tanaka,[†]
Masahiro Goshima,[†] Shin-ichiro Mori[†] and Shinji Tomita[†]

Recently the high performance of RISC processor highly depends on (1st) cache hit ratio. The performance is highly decreased when we do calculation in large size of numerical data which is much bigger than the size of cache.

In this paper, I will speak about the loop restructuring technique which is called Phasing. When applying Phasing technique, I do strip mining for the most nested loop in more than 3-level nested loop and move outer loop which is stripped to the most level.

When the program is applied to phasing, cache hit ratio depends on only cache size, and independence from the size of subject or cache block size and is very high ratio.

For example in lu decomposition the cache ratio is 99.6% when cache size is 16kb. If cache size is larger, hit ratio is more increased.

1. はじめに

近年のプロセッサの示す高い演算性能は，キャッシュに強く依存している．例えば，平均2サイクルに1回のロード/ストア命令があり，キャッシュ・ミス時の平均レイテンシが20サイクルであるとしよう．ピーク性能の90%を引き出すには，99.9%もの高いキャッシュ・ヒット率が必要になる．しかも，メモリの速度は，プロセッサのそれに比べて相対的に遅くなりつつあり，キャッシュに対する依存度はますます高くなっていく．

一方で，大規模数値処理を主なターゲットとする計算機では，キャッシュに頼ることなく，プロセッサの演算性能に見合うスループットを持つ主記憶を構成することも多い．

しかし，プロセッサとメモリの速度差が大きくなるにつれ，そのようなシステムは非常に高価なものとなる．逆に，キャッシュを効果的に利用することによって，プロセッサの性能を100%近く引き出すことができるなら，ハードウェア・コストを大きく削減することができる．

そのためには，キャッシュを効果的に利用する最適化技術が不可欠である．特に，ますます大きくなるプロセッサとメモリの速度差を埋めるためには，1次キャッシュだけではなく，メモリの各階層において，ヒット率を向上させることが重要となる．本稿では，各メモリ階層に合わせて多重にタイリングを施すことによって各階層でのヒット率の向上を図るのだが，その際に，各タイルのサイズと処理順序を決定する手法について述べる．

[†] 京都大学 大学院 情報学研究科 通信情報システム専攻
Division of Communications and Computer Engineering, Graduate School of Infomatics, kyoto University

2. メモリ階層とタイリング

まず本章では，次章で手法の具体的説明をする前に，メモリ階層とタイリングについてまとめる．

なお実行環境としては，Sun，Ultra 2 model 2300 1),2)，OS は Solaris 2.6 を用いた．実行環境のパラメータを，表 1 にまとめる．以降では，この値に基づいて説明を行う．また，言語としては C を用いるが，Column-major な言語に対しては，行と列を読み替えるなどすればよい．

2.1 メモリ階層

本節ではメモリ階層として，特に，TLB とレジスタについて述べる．本稿では，レジスタは，第一のメモリ階層として他のメモリ階層と同様に扱うが，TLB は例外的に扱う．

TLB

`double a[][1024]` では，`a[i][j]` と `a[i+1][j]` は，既に異なるページにある．したがって，同時にたかだか 64 に近い数の行にアクセスするようなパタンは，TLB のあふれを引き起こす．TLB ミスのペナルティは，2 次キャッシュのそれより大きいため，無視できない．

この問題はページ・テーブルのキャッシングや Large Page を使えば緩和/解消されるが，標準的な OS にそのようなことは期待できない．

メモリは，TLB にアドレスが載っている領域と載っていない領域に階層化されていると考える必要がある．

しかし TLB にアドレスが載っている主記憶の領域は，他のメモリ階層と比べると，表 1 に示すようにかなり特殊である．キャッシュのエントリの数は十分大きいため，列方向アクセスでもその容量をある程度使いきることができる．実際 後述する例では，キャッシュの容量のみを考慮すればよく，エントリ数やブロック・サイズは無視しても構わない．しかし，TLB のエントリ数は非常に少ないので，そういうわけにはいかない．

本稿では，TLB を例外的に扱うことにする．

レジスタ

一方 レジスタは，第 1 のメモリ階層として，他の階層と区別せず統一的に扱う．すなわち，計算の主体は演算ユニットであり，これらがレジスタに対して参照と定義を実行すると考える．このように考えると，レジスタ上のデータに再びアクセスすることは，「レジスタにヒットした」とできる．演算ユニットによるアクセスが「レジスタにミス」したとき，ロード/ストアを実行することになる．

表 1　実行環境のパラメータ

階層	エントリ数	ブロック☆ サイズ	容　量☆	レイ☆☆ テンシ
レジスタ	32	1	32	—
1 次キャッシュ	512	2	2K	8
2 次キャッシュ	16K	8	256K	62
TLB	64	1K	64K	100

☆倍精度浮動小数点数 単位

☆☆ミス時のレイテンシ (cycles)

```
for (i1=0; i1<N; i1+=si)
  for (i=i1; i<min(i+si, N); i++)
    s += a[i];
```

図 1　Program 0

レジスタは，他のメモリ階層とは，どのデータが載っているかを完全に制御できる点と，書き込みに対する動作だけが異なる．

機能ユニットからの書き込みに関して，レジスタは Writeback 型のキャッシュに近い動作をする．ただしキャッシュの writeback が暗示的に実行されるのに対し，レジスタからの writeback はストア命令によって明示的に実行され，命令スロットを消費する．

以降では，レジスタ，1 次/2 次キャッシュ，主記憶の各メモリ階層のことを，単に階層と呼ぶことにする．

2.2 タイリング

次に，マルチレベル・タイリングにおいて基本的な処理であるタイリングについて簡単に説明する 3)．

1 重のループを 2 重のループに分解する処理は，ストリップ・マイニングと呼ばれる．図 1 に，ストリップ・マイニングされたループの例を示す．ここで，外側の `i1` ループをストリップ・ループ，内側の `i` ループを要素ループ，そして，`si` をストリップ・サイズという．

タイリングとは，多重ループの各ループにストリップ・マイニングを施し，生じるストリップ・ループを外側へ，要素ループを最内側へと交換する処理である．次ページの図 2 に，タイリングを施した行列積プログラムを示す．

タイリング処理後には，要素ループを互いに交換することにより，タイル内の処理順序が；ストリップ・ループを互いに交換することにより，タイル間の処理順序が，それぞれ独立に制御される．

多重ループは，各ループに対応する次元を持つ直行空間を定義する．この空間をそのループの反復空間 (iteration space) という．タイリングは，反復空間を複数のタイルに分割する．

`a[i][k]` のような配列参照によって参照される配列の部分を，その配列参照のフットプリントと呼ぶ．タイリング後は，タイルのフットプリントの面積を縮小することによって，参照の局所性の向上を図ることができる．

3. マルチレベル・タイリング

タイリングは，多重に行うことができる．多重タイリングでは，タイルを階層的にサブタイルに分割する．

原理的には，各メモリ階層に対応するタイルのサイズとタイル内サブタイルの処理順序を階層の容量に合わせて決めることによって，各階層に対するヒット率を向上させることができる．

しかし，最適なタイル・サイズとサブタイルの処理順序は，各階層間で互いに独立ではなく，全体として最適な解を求めることは難しい．

本章では，行列積を例に，各階層に対するタイル・サイズと処理順序を決定する手法について述べる．

3.1 タイルの形状

図 2 に，`i`，`j`，`k` の各ループをそれぞれストリップ・サイズ s_i，s_j，s_k でタイリングした行列積プログラムを示す．タイルは，同図中では四角の内部で表される．ここでは，タイルの形状と階層のヒット/ミス率について考える．

タイル内では，`i`，`j`，`k` の各ループの交換によって 6 通りの処理順序が考えられるが，それぞれ階層のヒット/ミスに関しては対称である．以降では，図 2 に示したとおり，外側から `kij` 型のものについて述べる．

図 3 にタイルのフットプリントを示す．

階層のサイズ C は，タイルのフットプリントの面積の合計 S_{all} に対して，ずっと小さくて構わない．競合性ミスを考慮せず，置き換えは LRU によって起こるものとすると，同図中の網掛けの部分が階層に載りさえすれば，これ以上大きくてもヒット率は変らない．この場合，データの再利用率は 100%となり，ミスは初期ミスとのみとなる．したがって，網掛けの部分の面積 S_c は；

$$S_c = s_i s_j + s_j + 1 < C \qquad (1)$$

とする．

タイル内の階層のミスは初期ミスのみであるので，ミス回数は S_{all} に等しい．ミス率 M は，タイル内の配列アクセスの回数を A として；

$$\begin{aligned} M &= S_{all}/A \\ &= (s_i s_j + s_j s_k + s_k s_i)/s_i s_j s_k \\ &= 1/s_i + 1/s_j + 1/s_k \qquad (2) \end{aligned}$$

となる．ただし，階層のブロック・サイズは 1 とする．また実際には，`c[i][j]` に対しては，参照と定義の 2 回のアクセスが生じるが，ここでは合わせて 1 回と数えている．

前述したように，M は `i`，`j`，`k` の順序によらない．

式 (2) から，M を最小化するためには，s_i，s_j，s_k をできるだけ大きくすればよいことが分かる．

ミス率 M を最小化するためには，直感的には，フットプリントの面積 S_{all} を最小化するのがよいように思えるが，そうではない．実際 M は，$s \equiv s_i = s_j = s_k$ のとき最小値 $3/s$ をとるが，その場合，式 (1) の条件から，s 自体が非常に小さい値 $(s < C' \simeq \sqrt{C})$ になってしまう．

```
for (k1=0; k1<N; k1+= s_k)
  for (i1=0; i1<N; i1+= s_i)
    for (j1=0; j1<N; j1+= s_j)
      for (k=k1; k< s_k; k++)
        for (i=i1; i< s_i; i++)
          for (j=j1; j< s_j; j++)
            c[i][j]+=a[i][k]*a[k][j];
```

図 2 Program 1

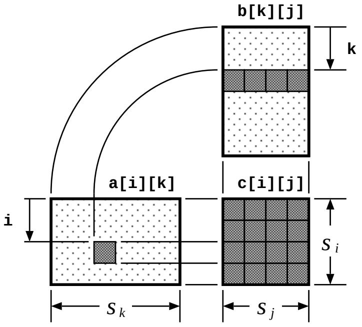

図 3 タイルのフットプリント

結局 M を最小にするには，$s_k \gg s_i$,s_j とすればよい．つまり，s_i と s_j の値は式 1 から上限が決まるのに対して，s_k にはこの段階では制限が無く，好きなだけ大きな値にすればよい．s_k の値は後述する次の段階で決定することにする．

タイル内の最外側になるこの `k` をタイルの自由軸と呼び，内側にある `i`，`j` を束縛軸と呼ぶことにする．タイルは自由軸 `k` 方向に長い直方体となる．

実際にはこの最初の階層の処理は，レジスタに対して行われる．$C = 32$ とすると，式 (1) から $s_i = s_j = 4$ が得られる．これは，4 行 4 列同時処理によるロード/ストアの削減を意味する．これによりロード/ストアは，2 回/FLOP から 最小 0.5 回/FLOP まで削減される．

なお，空間局所性を利用するため，この時にキャッシュ・ブロックのサイズを考慮することには意味がある．

3.2 多重化 (1) 処理順序の決定

さて，次の階層 —— 実際には 1 次キャッシュに対するヒット率を高めるために，前階層で求めたタイルをサブタイルとして，再び前階層と同様の処理を繰り返す．図 2 で言えば，図中の四角内の処理がサブタイルにあたるので，外側の `i1`，`j1`，`k1` ループを，再びタイリングする．この時のストリップ・サイズをそれぞれ，s_i^1，s_j^1，s_k^1 とする．

この階層でのヒット率を高めるために，基本的には前階層と同じ議論を繰り返せばよい．

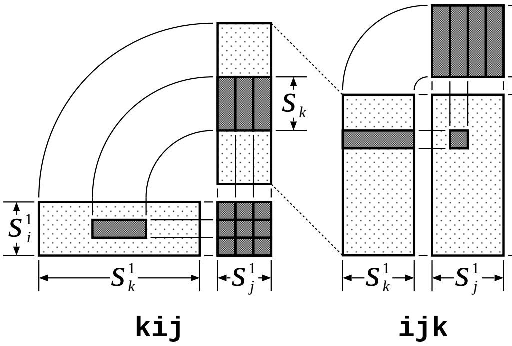

図 4　Program 1

ただし前階層とは，サブタイルの形状が異なる．前項で求めたこの階層でのサブタイルの形状は，自由軸 `k` 方向に長い直方体であり，配列参照 `a[i][j]`，`b[k][j]` のフットプリントは，`k` 方向に長い長方形である．一方前階層では，文 `c[i][j]+=a[i][k]*b[k][j];` がサブタイルにあたり，各配列参照のフットプリントは 1 配列要素 ——— いわば「正方形」であった．

前階層でのミス率を低減するために，自由軸方向の長さ s_k は十分に大きくする必要がある．しかしあまりに大きくし過ぎると，今度はこの階層でのミス率が悪化してしまう．前階層とこの階層のミス率をうまくバランスさせる必要がある．

図 4 に，`kji` 型と `ijk` 型のフットプリントを示す．図では，① 網掛けで示した階層にのせるとよい部分の面積と ② それぞれでの自由軸長である s_k^1 と s_i^1 が，双方の型で同じになるようにしてある．図から，`kij` 型の束縛軸長 s_i^1，s_j^1 が，`jik` 型の束縛軸長 s_j^1，s_k^1 より小さくなっていることが分かる．

ミス率は，やはり式 (2) で与えられるので，`kij` 型のミス率は，`ijk` 型のそれより悪くなる．また，s_k も `kij` 型の方が短く，前階層でのミス率も悪くなる．図では示さなかったが，`ikj` 型でも同じことが言える．

`kij`，`ikj` 型でのミス率の悪化は，前階層の自由軸 `k` の方向に無駄に階層の容量を取られることによる．

それに対して `ijk` 型では，`k` 方向の長さと，階層の容量が競合しなくなっている．`jik` 型でも同じことが言える．それは，`k` が最内側になっていることによる．

`k` ループを最内側にすると，前階層の自由軸 `k` ループが融合して 1 段のループになる．結局 前階層で自由軸に選んだループには，この階層ではストリップ・マイニングを施さないことになる．

3.3　多重化 (2) タイル・サイズの決定

前階層の自由軸にストリップ・マイニングを施さないので，この階層の自由軸は，前階層の自由軸以外から選ぶことになる．自由軸の選択基準については，3.4 でまとめる．ここでは，`ijk` 型を選択する．

次に，この階層の容量と，式 (1) から，ストリップ・サイズ s_j^1 と s_k^1 を決定する．前階層の自由軸長 s_k は，$s_k = s_k^1$ であるから，この時点で決定される．

```
for (i2=0; i2<N; i2+=32)
  for (j2=0; j2<N; js+=32)
    for (k2=0; k2<N; k2+=32)
      for (j1=j2; j1<j2+32; j1+=4)
        for (i1=i2; i1<i2+32; i1+=4)
          for (k=k2; k<k2+32; k++)
            for (i=i1; i<i1+4; i++)
              for (j=j1; j<j1+4; j++)
                c[i][j]+=a[i][k]*a[k][j];
```

図 5　Program 1

この議論を UltraSPARC-I/II の 1 次キャッシュに対して適用すると，$s_k = s_k^1 = s_j^1 = 32$ が得られ，1 次キャッシュ・ヒット率は最大 96.875%になる．

さて，前階層の自由軸にストリップ・マイニングを施さないことによって緩和されるとはいえ，前階層とこの階層のミス率が，前階層の自由軸長（今の場合 $s_k = s_k^1$）を介して，ある程度背反することは避け得ない．しかし，以下の理由により，前階層の自由軸長は，この階層のミス率のみから決定して構わない：前階層においては，束縛軸長は前階層の容量から決まるのに対して，自由軸長はこの階層の容量から決まるので，自由軸長は束縛軸長に比べてずっと大きい（$s_k \gg s_i$，s_j）．したがって，前階層でのミス率を支配するのは前階層の束縛軸長であり，自由軸長の影響はずっと小さいものとなる．一方前階層の自由軸長はこの階層では束縛軸長になるので，この階層のミス率に大きな影響を与える．例えば本章の例では，この階層のミス率のみを考慮して求めた $s_k = s_k^1$ の最適値は 32 である．ここで例えば，$s_k = s_k^1 = 64$ としても，前階層でのミス率は 3%しか改善されなにのに対して，この階層でのミス率は 20%も悪化してしまう．

図 5 に，ここまでで得られたループを示しておく．外側と内側の四角の間の部分が，この階層におけるタイリングの結果得られる部分である．`k` 方向にはストリップ・マイニングしていないことに注意．

以下，本節で述べた手続きを，配列のすべてがある階層に載るようになるまで，階層的に繰り返せばよい．

図 6 に，2 次元反復空間におけるマルチレベル・タイリングの模式図を示す．図中，黒丸がループ本体の 1 回の実行を表す．本手法ではまず，内側から細長いタイルを作る．短軸が束縛軸，長軸が自由軸である．束縛軸長はこの時決め，自由軸長は決めずに次のステップに進む．次のステップでは，このサブタイルを束縛軸方向にまとめ，その方向に細長いタイルを作る．ここでこのタイルの束縛軸長を決めるのだが，これによってサブタイルの自由軸長が決まる．これを，階層的に繰り返す．

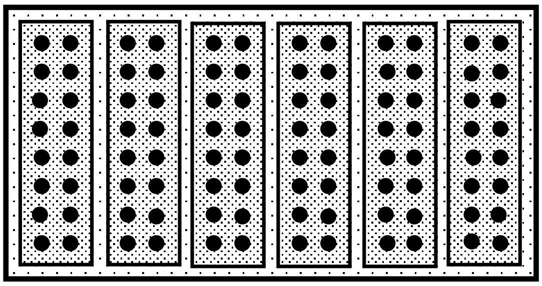

図 **6**　マルチレベル・タイリングの模式図

3.4　自由軸の選択

さて，今までの議論では，各階層での自由軸の選び方には任意性があった．本項では，自由軸の選択基準について述べる．自由軸は，書き込みの回数によって選択する．

3.1 の議論において，自由軸を `k` としたので，`k` を含まない配列参照である `c[i][j]` に対するアクセスは，他の配列参照に対して，そのフットプリントが非常に小さくなることが分かる．

式の左辺に表われる配列参照のフットプリントを小さくすることには，writeback 方式の階層に対して，dirty コピーの数を減らし，汚染を予防する効果がある．3.1 で `k` を自由軸に選んだのは，この理由による．この効果は，レジスタ・レベルでは，ストアの回数の減少にあたるので特に重要である．

この効果は，1 階層おきに適用することができる．例えば上記の例では，まずストアの回数を少なくしたが，すると次には 2 次キャッシュから主記憶への writeback を減らすことができ，都合がよい．

一方で，敢えてストアの回数を大幅に増やしても，主記憶アクセスを連続アドレス方向にする最適化がよく知られているが，そのような意味でアクセス方向を考慮する必要は全くない．今回の手法では，結果としてキャッシュ・ヒット率が十分高くなる．主記憶とは異なり，キャッシュは，ヒットしている限り，不連続アクセスに対しても性能が悪化しないからである．ただし，整合配列などを利用して，競合性のミスが多発しないようにする必要はある．

3.5　TLB

TLB を，1 つの階層として統一的に扱うことは難しい．前章の例題では，1 次キャッシュに載せる領域のサイズを，$s_i = s_j = 4$，$s_k = s_k^1 = s_j^1 = 32$ とした．これは 40 ページに相当し，TLB のエントリをほぼ使いきってしまう．すなわち，TLB（にアドレスが載っている領域）の実効容量は，1 次キャッシュと同程度であり，前述の議論は成り立たない．

したがって，1 次キャッシュと TLB のレベルを，まとめて 1 回のタイリングで対処した．実際，そうした場合が，2 回に分けて 2 重にタイリングを施した場合よりよい性能を示した．

4. 性能評価

本章では，前章で述べた手法を行列積の他，LU 分解に適用した場合の性能を示す．

なお，レジスタ・レベルでのタイリングを施した部分は，アセンブリ言語で記述した．4 行 4 列同時処理するプログラムに対し，ソフトウェア・パイプライニングを施した上でさらに 4 重のループ・アンローリングを施した．1 次キャッシュ・ヒット率を 100% した場合のこの部分の性能は 455MFLOPS=1.54FLOP/cycle であった．それ以外の部分は，C 言語で記述し，Sun WorkShop 5.0 を用いてコンパイルした．

4.1　行列積

図 7 に，前章で述べた行列積に対するタイリングを，レジスタ，1 次/2 次キャッシュ・レベルまで施したものの性能を示す．

レジスタ/1 次キャッシュ・レベルまでタイリングしたものでは，問題のサイズがそれぞれ 1 次/2 次キャッシュを超えるところで性能の低下が見られるが，2 次キャッシュ・レベルまでタイリングしたものでは，性能低下は見られない．なお，行列サイズが大きいところでの性能の低下は，OS の影響によるものと思われる．

参考までに，式 (2) と表 2 から求めたミス率とミス・ペナルティを，表 2 に示しておく．ミス・ペナルティは，プロセッサの性能を 1.54FLOP/cycle とした場合の，1cycle あたりのペナルティである．

表からは，ピークに対して，$100/(1 + 0.0898 + 0.115 + 0.0509) = 79.6\%$の性能となる．ただし実際には，実験に用いた UltraSPARC は，ロード/ストアが non-blocking であり，1 次キャッシュ・ミスのレイテンシ (8cycles) に対してなら，ある程度の隠蔽が可能である．実際 実測では，$455 \times 0.796 = 362.18$MFLOPS より若干高い性能を示している．

また表からは，以下のことが読み取れる：

- TLB ミスの影響が無視できない．もちろん，レイテンシの隠蔽は図れない．
- 2 次キャッシュ・ミス，すなわち主記憶の影響は，キャッシュの影響より小さい．主記憶のピーク・スループットの 5.09% しか使用せず，スループットは問題にならない．

4.2　LU 分解

図 8 に，外積ブロック・ガウス法と，本手法を 1 次/2 次キャッシュ・レベルまで適用したものの性能を

表 **2**　行列積のミス率とペナルティ

階層	ミス率 (%)	ペナルティ(%)
2 次キャッシュ	0.195	5.09
1 次キャッシュ	3.52	11.5
TLB	0.220	8.98
レジスタ	53.1	—

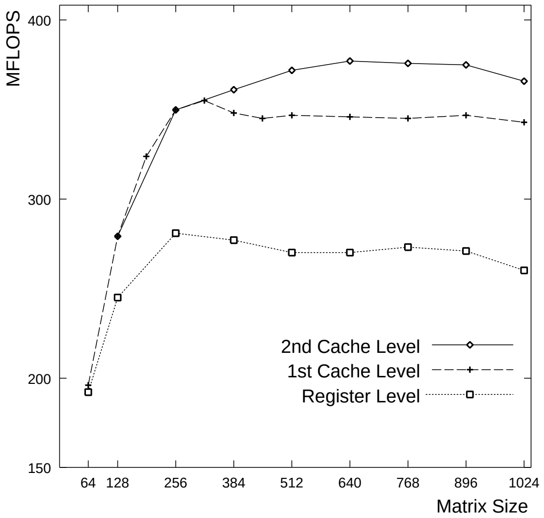

図 7　行列積

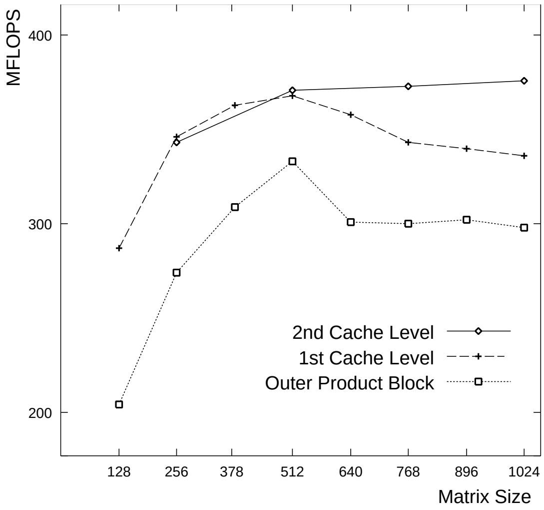

図 8　LU 分解

示す．

一般に外積ブロック・ガウス法として知られている手法は，外積形式ガウス法に対し 1 回のタイリングを施したものである$^{4)}$．一方，本稿で述べた手法では，内積形式と呼ばれる手法に近いものが得られる．1 次キャッシュ・レベルまで適用したものは，外積ブロック・ガウス法とは最外側ループの順序が異なり，2 次キャッシュの汚染が起こらない．外積ブロック・ガウス法の性能低下は，2 次キャッシュの汚染によるミス・レイテンシの増加が原因である．

外積ブロック・ガウス法と 1 次キャッシュ・レベルまでタイリングしたものとは異なり，2 次キャッシュ・レベルまでタイリングしたものには，やはり，キャッシュのあふれによる性能低下は見られない．

5. おわりに

本稿では，各メモリ階層に合わせて多重にタイリングを施す際に，タイル・サイズとループの順序を決定する手法について述べた．本手法を行列積と LU 分解に適用したところ，良好な結果が得られた．

ただし，今回の性能評価では，プログラムの変形を手で行っているため，定式化が不十分であることは否めない．また，手法の適用可能な条件等もよく分かっていない．定式化を進めるためにも，本手法を自動的に実行するコンパイラを作成することが，まず考えられる．

また，TLB に対する処置が不十分であり，より深い考察が必要である．

さて，タイリングは，ある要素の中間結果を保存しておいて別の要素の計算に移ることによって，局所性を抽出する訳である．数値処理などのプログラムに対して，コンパイラがこれを実行できるのは，加算や乗算が可換であるということを知っているからである．したがって，より一般的なプログラムに対しては，コンパイラによる自動的な変形は不可能である．

しかし，本稿で述べた手法の考え方自体はより広範な，非定型的なプログラムにも適用可能である．今後は，そのようなプログラムに対する本手法の効果を評価したい．また，アプリケーション・プログラマが本手法を適用する場合に，それをサポートするためのツール類の開発も，合わせて行いたい．

参考文献

1) Tremblay, M. and O'Connor, J. M.: UltraSPARC I: A Four-Issue Processor Supporting Multimedia, *IEEE Micro*, No. 4, pp. 42–49 (1996).
2) Sun Microsystems, inc.: *UltraSPARC User's Manual* (1997).
3) Wolfe, M.: More Iteration Space Tiling, *Proc. of Supercomputing '89*, pp. 655–664 (1989).
4) 山本有作, 大河内俊夫: ガウスの消去法の超並列機向け最適化, 並列処理シンポジウム JSPP '95, pp. 217–224 (1995).
5) 寒川光: 数値計算プログラムにおけるデータ移動制御のためのブロック化アルゴリズム, 情報処理学会論文誌, Vol. 38, No. 10, 情報処理学会, pp. 1183–1192 (1992).