特別研究報告書

メモリアクセスの局所性を最適化するループ再構成法

指導教官　　富田 眞治 教授

京都大学工学部情報工学科

津田 健

平成１１年２月１２日

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

内容梗概

近年、プロセッサの演算性能は年々向上されている。しかし、メモリのアクセススピードはそれほど向上されてはおらず、メモリのアクセススピードはプロセッサの演算速度に比べて相対的に遅くなってきている。

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て、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り、プロセッサの性能を100%に近く引き出すことができるなら、ハードウェアのコストを大きく削減することができる。

そこで、キャッシュを効率的に利用する最適化技法が重要である。特に、問題サイズが大きくなった場合、プロセッサの演算性能を生かすためには、1次キャッシュのヒット率向上だけにとどまらず、階層化された各記憶階層のヒット率を向上させ、全体としてのパフォーマンス向上をはかることが重要となる。

プログラムの大部分を占めるループの最適化技法として、タイリングが知られている。タイリングは、各階層ごとに適用可能である。本稿では、各記憶階層の容量に合わせて、多重にタイリングを施すことにより、各記憶階層でのヒット率向上をはかり、その際に各タイルのサイズや処理順序を決定する手法について述べる。しかし、ある階層における最適なタイル・サイズと処理順序は、各階層間で互いに独立ではなく、相反するものである。したがって、全体としての最適な解をもとめることは、非常に難しい。

本手法では、タイルサイズはアクセス時間が大きい記憶階層のみから決定してもかまわない。アクセススピードが速い階層のヒット率をわずかに上げることによるメリットよりも、アクセススピードが遅い階層のキャッシュのヒット率を大幅に向上させるメリットの方が大きくなるからである。

また、階層の容量を有効的に利用する処理順序を考え、ループの順序の決定

方法について述べる。

行列積に本手法を用いて、レジスタレベル、1 次キャッシュレベル、2 次キャッシュレベルまで最適化を施し、性能評価を行った。1 次キャッシュレベルの最適化および 2 次キャッシュレベルの最適化においては、レジスタレベルの最適化と比較して、それぞれ、31%、40%の性能向上が見られた。各レベルごとのキャッシュの最適化による性能向上が見られた。また、問題のサイズが大きくなることによる性能低下は見られなかった。

LU 分解に本手法を用いて性能評価を行った。LU 分解に 1 次キャッシュレベル、2 次キャッシュレベルまで本手法を用いて最適化をした。従来の最適化手法である外積ブロックガウス法と評価比較をおこなった。

LU 分解に本手法を適用すると、内積形式ガウス法を最適化したものになる。従来の最適化手法である外積ブロックガウス法は、外積形式ガウス法を最適化したものである。外積ブロックガウス法と比較して、本手法を用いた LU 分解の 1 次キャッシュレベルの最適化および 2 次キャッシュレベルの最適化では、約 12%、26%の性能向上が見られた。

本手法を用いると従来の最適化手法よりもさらに最適化が可能であることがわかり、良い結果が得られた。

A Loop restructuring technique to optimize memory access locality

Takeshi Tsuda

Abstract

Recently, the operation performance of processors is improved very much every year. But, the speed of the memory is not improved as well and the memory access speed is relatively lower than the processor speed.

Among computers which are targeted for large-scale numerical transaction, it is usual to use the main memory which has ennough through put for processor operation without depending on the cache.

But, when the processor speed is faster than the memory speed, such system is very costy. On the contrary, if nearly 100% performance of processor could be used, hardware cost could be cut sharply.

Then it is important to optimize technique to make efficient use of the cache. Especially, when the problem size is bigger, it is important for not only improvement of the cache hit ratio of first cache but also improvement of the cache hit ratio of each class of memory hierarchy and improvement of total performance.

Loop execution occupies large amount of program execution. Tiling is known for the loop optimization technique. Tiling can be applied to each class of memory hierarchy.

This paper discusses the tile size and the order of loop execution to improve the hit ratio of each class of the memory hierarchy. But, the most optimum tile size of a certain level of memory hierarchy and order of transaction is not independent from that of the next one and is opposite. So, it is very difficult to find best solution to optimize total performance.

In this technique tile size may be decided by the size of slower memory hierarchy level. The merit of big improvement of the cache hit ratio of lower class of cache is greater than that of little improvement of the cache hit ratio.

Then, I evaluate and compare the time of program execution which is optimized for register level , first cache level, and second cache level. The optimization program at first level of cache show 31% better result than that the program at register level of cache. And the program at second level of cache show 40% better result than the program at the register level.

Then I find the improvement of performance by optimization of each level of cache. And when size of problem is bigger, I do not find the decrease of performance.

I apply this technique to LU and evaluate and compare the time. I optimize for fisrt cache level, and second cache level. And I measure the time of the outer product block which is former optimization technique for LU. The optimization of inner product is applied to LU in this technique.

The former optimization, outer block product is the optimization of outer product. When I compare with outer block product, I get about 12% better result for the optimized program at the fisrt level of cache, and about 26% better result for the optimized program at the second level of cache.

I get good result and find that more optimization can be done than former technique.

メモリアクセスの局所性を
最適化するループ再構成法

目次

1　はじめに

近年, プロセッサの周波数は年々高速化されているが, メモリのアクセススピードはそれほど向上していない. 現在のRISC型のプロセッサがもつ高い演算性能は(1次)キャッシュに強く依存している. 問題のサイズが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な数値処理演算においては, その性能は著しく低下する. そのような計算においては, キャッシュのヒット率をいかに向上させるかが非常に重要な課題となる. プログラムの大部分はループと呼ばれる繰り返しの部分であり, ループにおけるキャッシュのヒット率向上というのは非常に重要な課題である.

また、ますます大きくなるプロセッサとメモリとの速度差を埋めるためには、(1次)キャッシュのヒット率だけではなく、各記憶階層において、ヒット率向上をはかるのが重要となる。本稿では、各記憶階層に合わせて多重にタイリングを施すことにより、各記憶階層のヒット率向上をはかる際に、各階層に対応するタイルのサイズや処理順序を決定する手法について述べる。

行列積を用いて本手法を説明し、行列積を用いて各レベルの記憶階層のおける最適化をみるために、評価比較を行う。また、LU分解に本手法を用いて、従来の最適化技法である外積ブロックガウス法と評価比較を行う。

2　記憶階層とタイリング

2.1　準備

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，以降の説明においては，実行環境のパラメータとしては，プロセッサはUltraSPARC-I/II、OSはSolaris 2.6のものを用いる。また，言語はCを用いる。

メモリ階層としては特に，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．

2.1.1　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TLBエントリ数を超える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．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，表に示すようにかなり特殊である．

このようにどの階層にあたるとも言い難いので，例外的に扱わざるを得ない．

2.1.2　レジスタ

本手法では，レジスタを，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．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．

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，キャッシュ，主記憶を含む各メモリ階層のことを，単に階層と呼ぶことにする．

2.2　実行環境のパラメータ

Sun Ultra2 model 2300 を例に取り、実行環境のパラメータを以下の表にまとめる。

ここで、ブロックサイズおよび容量の単位は倍精度浮動小数点 (double 型) である。

2.3　タイリング

ループの最適化技法として知られているタイリングについて述べる.

タイリングの中には、ループインターチェンジングとストリップマイニング

表 2.1: 実行環境のパラメータ

階層	エントリ数	ブロックサイズ (dw)	容量 (dw)	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

がある。

2.3.1　ループインターチェンジング

ループ交換は、2 つのループを入れ替えることによってストライドを小さくしてメモリ参照性を良くしようとする手法である. ループ交換可能な条件は、交換する外側のループによって生じるデータの依存性がないことと、内側ループにおいて負の方向に依存性がないことである.

```
do I = 1, N
  do J = 1, N
    C += A(I,J)*B(I,J)
  enddo
enddo
```

のループがあったときに, I と J のループを入れ替える.

```
do J=1,N
  do I=1,N
    C += A(I,J)*B(I,J)
  enddo
enddo
```

2.3.2　ストリップマイニング

1 つのループを 2 つのループの分割する. 内側のループの変化幅、ストリップサイズはキャッシュの容量に合せて最適化を行う. ストリップサイズは、ループにおけるデータの依存関係に影響を与えるが常に適用可能である.

以下のようなループがあったとする.

```
do I=1,N
  do J=1,N
```

```
    C += A(I)*B(I)
  enddo
enddo
```

IのループのストリップサイズをΙ32にして、外側にもう1重ループを付け加え、Jのループと入れ替える。

```
do IS = 1, N, 3
  do J=1,N
    do I = IS, MIN(N,IS+31)
      C += A(I)*B(I)
    enddo
  enddo
enddo
```

となる。

2.3.3 フットプリント

タイリングでは、多重化されたループに対して、それぞれのループに対してストリップマイニングを適用し、その結果生まれた外側ループをさらに外側のループと交換することにより、外側のレベルまで移動させる. 1つのループの中で参照されるそれぞれの配列の部分をフットプリントと呼び、多重化されたループのそれぞれのレベルにおいて、ループにおけるそれぞれの配列の添字の変化量からフットプリントのサイズを知ることができる.

```
do IT = 1, N, 32
  do JT = 1, M, 32
    do I = IT,MIN(N,IT+31)
      do J = JT,MIN(M,IT+31)
        A(I,J) = B(J)*C(I)
      enddo
    enddo
  enddo
enddo
```

例えば、上記のタイリングされたプログラムにおいて、フットプリントのサイズはB、Cとも32である. BとCの要素がキャッシュに収まるのであれば、BとCは32回使用されるが、ロードは1回ですむ.

また、フットプリントを決める際には、別な問題としてキャッシュの1ラインのサイズを考慮に入れなければならない.

2.3.4　タイリング

タイリングの最適化の順序としては,

まず、それぞれのループに対してストリップマイニングを施し、その結果生じた外側ループをさらに外側のループと交換することにより外側のレベルまで移動させる.

つぎに、キャッシュメモリの局所性を考慮して, ストリップマイニングされた内側のループおよび外側のループ同士を交換し、それぞれの要素に対してストリップサイズを最適化する.

2.3.5　多重タイリング

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。

本手法では、各記憶階層の局所性に応じてタイルサイズの決定及びタイル内のサブタイルの処理順序を決定し、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、各記憶階層における最適なタイルサイズとタイル内のサブタイルの処理順序は、各記憶階層間で関係があり、全体の性能向上をはかることは非常に難しい。本手法では、各階層間の関係も考慮にいれて、各階層に対するタイルサイズとサブタイルの処理順序を決定する。

3　マルチレベルタイリング

3.1　タイルの形状

行列積を例にとって説明する。図3.1を参照する。

行列積C(i,j)+=A(i,k)*B(k,j)の`i`、`j`、`k`のループに対してストリップマイニングを施したときに、ストリップサイズをそれぞれs_i、s_j、s_kとする。

タイル内においては、`i`、`j`、`k`のループの順番において6通りの処理順序が考えられる。ここでは、図3.2に示すように、外側から`kij`型について述べる。

図3.2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合計サイズS_{all}に対して、ずっと小さくてもかまわない。セットアソシアティブによるミスを考慮せずに、置き換えはLRUによってのみ起こされるものとすると、図3.2の網掛け部分が階層に残っていれば、キャッシュのヒッ

```
for(k1=0;k1<N;k1+=sk)
  for(i1=0;i1<N;i1+=si)
    for(j1=0;j1<N;j1+=sj)
      for(k=k1;k<k1+ki;k++)
        for(i=i1;i<i1+si;i++)
          for(j=j1;j<j1+sj;j++)
             C[i][j]+=A[i][k]*B[k][j]
```

図 3.1: Program1

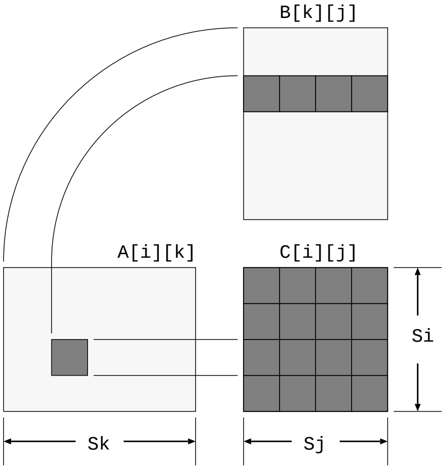


図 3.2: タイルのフットプリント

ト率は変わらない。

したがって、網掛けの部分の面積 S_cは、

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

を満たせば良い。

キャッシュのミスヒット率を計算すると、C(i,j) の 1 要素につき、A(i,k) と B(k,j) の要素を s_k個参照して計算する。A(i,j) の要素は、$s_i s_j$個あるので、アクセス回数は、$s_i s_j s_k$回である。また、キャッシュにミスヒットするのは初回のミスのみであるのでタイルのフットプリントの面積 S_{all}に等しい。よって、ミス

ヒット率 M は以下のように計算される。

$$M = S_{all}/s_i s_j s_k$$

$$= (s_i s_j + s_j s_k + s_k s_i)/s_i s_j s_k$$

$$= 1/s_i + 1/s_j + 1/s_k ... (2)$$

となる。よって、キャッシュのミスヒット率を最小にするためには、s_i、s_j、s_k をできるだけ大きくすれば良いことがわかる。ミスヒット率を最小にするためには、直感的には分子の S_{all} を最小化すれば良いように思えるがそうではない。そのとき、実際 M は $s = s_i = s_j = s_k$ のときに最小値 $3/s$ を取るが、その場合においては関係式 (1) 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．

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 の値は次の章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。

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。もちろん，すべてのレジスタを使えるわけがない。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 0.5 回/FLOP まで削減される．

3.2 多重化 (1)

さて，レジスタの次の階層、つまり実際には 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 3.1 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、基本的には前章で述べた内容と同じことを繰り返す。

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，`k` 方向に長い直方体であり，配列参照 A(i,k)，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、つまり正方形であった。

前階層でのミス率を低減するために、フットプリント方向の長さ 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。

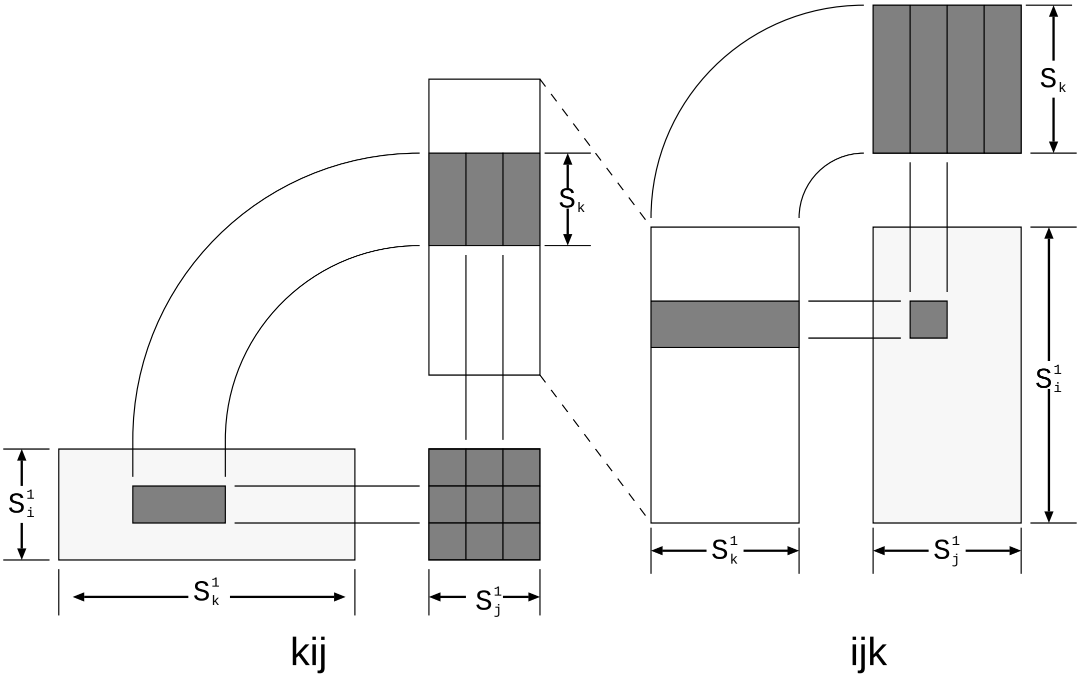


図 3.3: program2

図 3.4に，`kji` 型と `ijk` 型のフットプリントを示す。図では，網掛けで示したキャッシュにのせる部分と、それぞれの自由軸の長さである 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 型の束縛軸の長さ s_i^1、s_j^1 が、jik 型の束縛軸の長さ 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。したがって，`kij` 型のミス率 M 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．

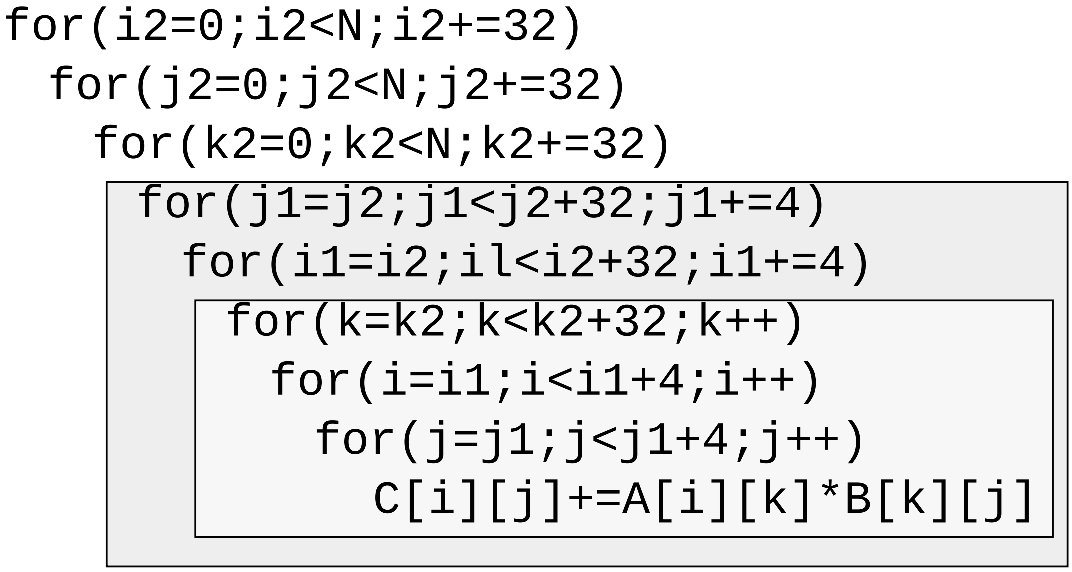
`kij`，`ikj` 型でのミス率の悪化は，前階層のフットプリント `k` の方向に大きく

キャッシュの容量を取られてしまうことによる。それに対して`ijk`型では，`k`方向の大きさと，キャッシュ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`ループが隣り合って、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)

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，後でまとめる。ここでは，`ijk`型を選択し、自由軸はi1とする。



```
for(i2=0;i2<N;i2+=32)
  for(j2=0;j2<N;j2+=32)
    for(k2=0;k2<N;k2+=32)
      for(j1=j2;j1<j2+32;j1+=4)
        for(i1=i2;il<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]*B[k][j]
```

図 3.4: program2

次に，この階層の容量と，式 (2) から，ストリップサイズ s_j^1 と s_k^1 を決定する．前階層の自由軸ののストリップサイズ s_k は，$s_k = s_k^1$ であるから，この時点で決定される。

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，$s_k = s_k^1 = s_j^1 = 32$ が得られる．この場合，1次キャッシュ・ヒット率は最大約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，前階層の自由軸の長さ（今の場合 $s_k = s_k^1$）は，この階層のミス率のみから決定して構わない．

前階層における束縛軸の長さは前階層の容量から決まるのに対して，自由軸の長さはこの階層の容量から決まる．したがって，自由軸は束縛軸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の長さであり，自由軸の長さの影響はずっと小さいものとなる．一方前階層の自由軸の長さはこの階層では束縛軸の長さになるので，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 3.4 に，得られる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．

以下のようにして、最終的に配列のすべてがキャッシュにのるように、各階層に対して再帰的に繰り返せば良い。

3.4　自由軸の選択

これから、自由軸の選択基準について述べる。

自由軸を選択する基準は、以下の 2 つがある。

- 書き込みの回数
- アクセス方向

3.4.1　書き込みの回数

レジスタレベルの最適化において、自由軸を `k` としたので、`k` を含まない `c[i][j]` に対するアクセスは，他の配列参照に対して，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，書き込みの量を減らす効果がある．書き込みは，そのタイルの処理の最後に 1 回だけ，小さなフットプリントに対して行われることになる．更にキャッシュが Writeback 方式であれば，ヒットと相まって，下位のキャッシュに対する書き込みの回数を非常に少なくすることができる．

レジスタ・レベルでは，ストアを減らすことは重要である．

ある配列参照のフットプリントは1階層おきに小さくすることができる．例えば上記の例では，まずレジスタに対して小さくしたが，すると1次キャッシュに対しては小さくできず，次には2次キャッシュに対して小さくできることになる．計算機の構成によっては，1次キャッシュ，主記憶に対して小さくした方がよい場合もあるだろう．

3.4.2　アクセス方向

敢えてストアの対象となる要素数を増やしても，アクセスの方向を連続アドレス方向にする最適化がよく知られている．しかし今回の手法では，結果としてキャッシュ・ヒット率が十分高くなるので，そのような意味でのアクセス方向はほとんど影響がない．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．

3.4.3　TLB

また，TLBには注意する必要がある．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．実際本章の例では，レジスタ・レベルのタイリングによって既に1ブロック程度の幅をとってしまうので，以降では容量のみを考慮すればよい．しかし，TLBのエントリの数は非常に少ないので，そういうわけにはいかない．

2次元配列の64行がもう64ページに渡るため，条件は非常に厳しい．同時には64に近い数のページにはアクセスしないよう注意する必要がある．

TLBを，1つのキャッシュとして統一的に扱うことは非常に困難である．本章の例題では，TLBの実効容量は，1次キャッシュと同程度となったので，まとめて1回のタイリングで対処することにした．

3.5　性能評価

行列積を用いて性能評価を行った。

行列積をレジスタレベル、1次キャッシュレベル、2次キャッシュレベルまで、最適化を施し、行列のサイズを変化させてその効果を調べた。

図3.5を参照する。行列のサイズが$N = 64$を越えると、レジスタレベルのみ最適化されている行列積は、タイルのフットプリントのサイズが1次キャッシュ

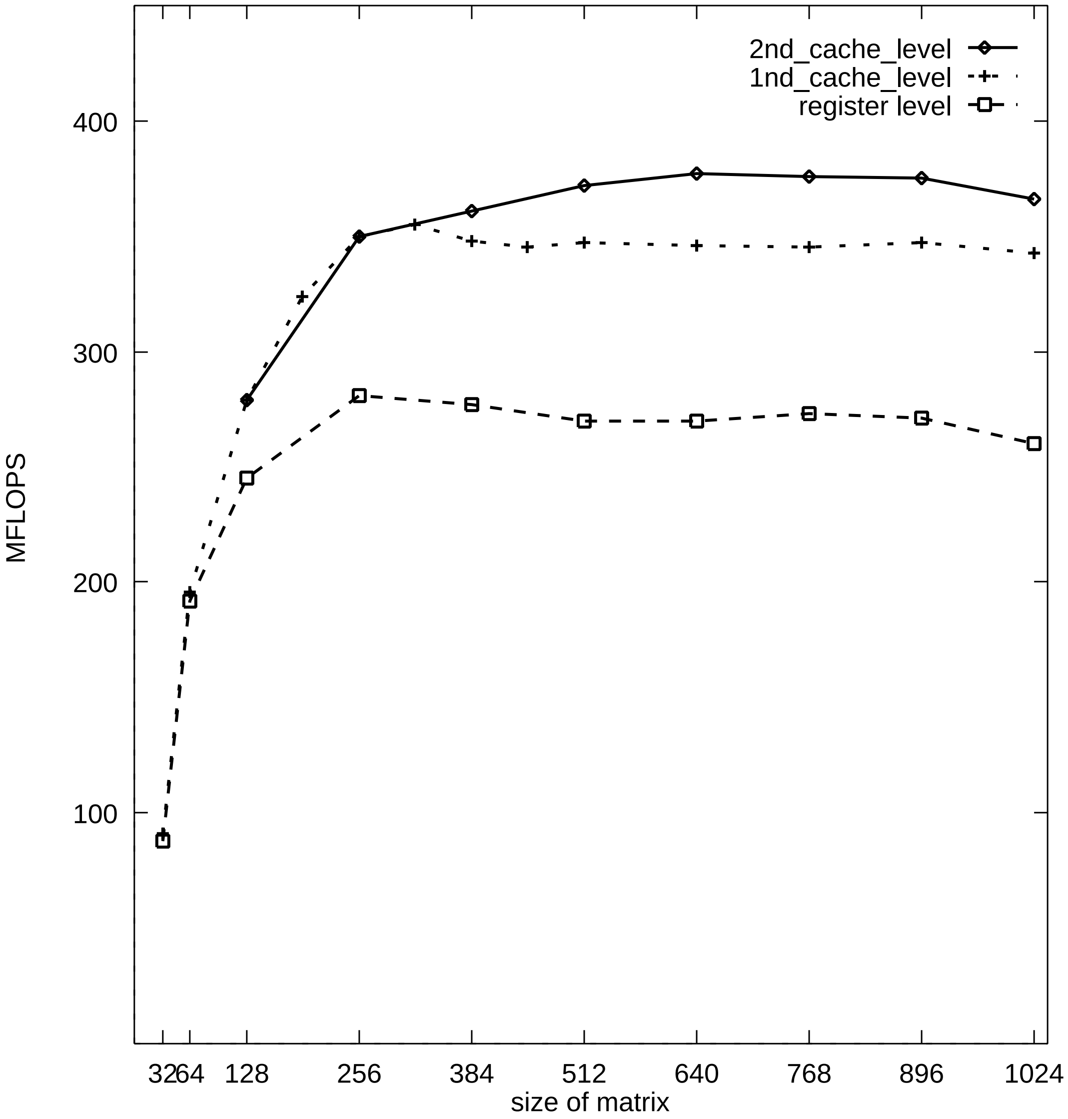


図 3.5: 行列積の性能評価グラフ

の容量を越えるので、性能低下が見られる。

$N = 320$ を越えると、問題のサイズが 2 次キャッシュを越えるので、1 次キャッシュレベルまで最適化をしているプログラムでは、性能低下が見られる。2 次キャッシュレベルまで最適化しているプログラムでは、性能低下は見られず、2 次キャッシュレベルまでの最適化が効果を発揮していることがわかる。

以上の結果から、行列のサイズ N が 64、320 のときがそれぞれ、1 次キャッシュレベルの最適化、2 次キャッシュレベルの最適化をするべきサイズであることがわかる。

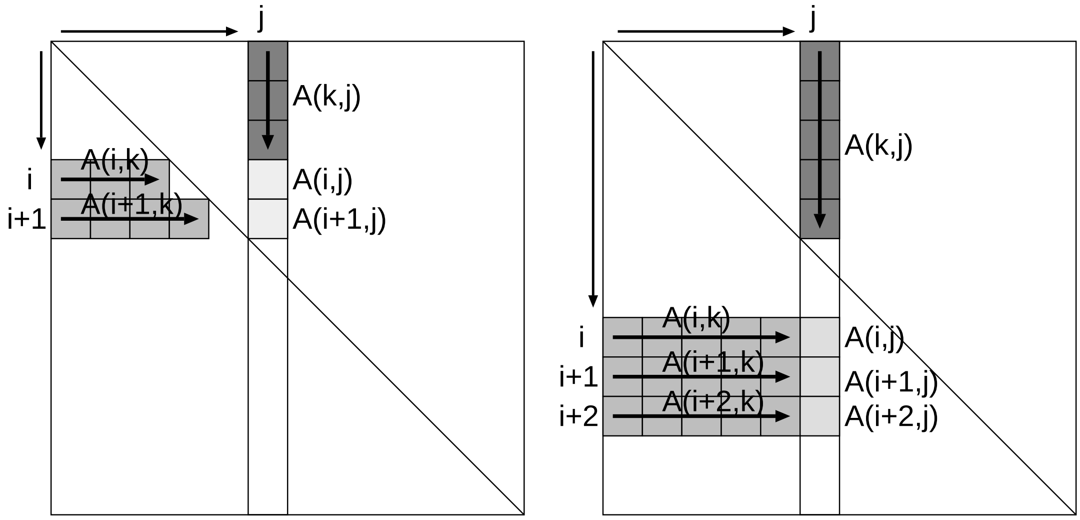


図 4.1: 内積ガウス法

4　LU 分解の最適化

4.1　LU 分解

LU 分解とは、$Ax = b$ の解を求めるときにおいて、行列 A を下三角行列 L と上三角行列 U の積に変換し、前進代入と後進代入により解 x をもとめる計算である.

今回のプログラムにおいては, LU 分解のみを実装した. 前進代入と後進代入の部分に関しては未実装である。

4.1.1　内積ガウス法

内積ガウス法は、図 4.1に示すような方法である.

1 つのデータ A(i,j) を計算するときに、右横のデータ A(i,k) と上のデータ A(k,j) を $k(= min(i,j))$ 個参照して集中的に内積計算を行い、計算結果を A(i,j) にストアした後に、次のデータの計算に移るという計算方式である。

4.1.2　外積ガウス法

外積ガウス法は図 4.2に示すような方法である.

第 k列と第 k行の要素を用いて, 右下の小行列に対して 1 回の乗算と加算を行った後に、結果をストアして、すぐに次の要素の演算に移るという計算を繰り返す。これを $k = 1$ から $k = N$まで実行する.

なお、この計算方式は1 回の乗算と加算を行うごとにストアするという方式なので内積ガウス法と比べて非常にロードとストアの回数が大きくなるという

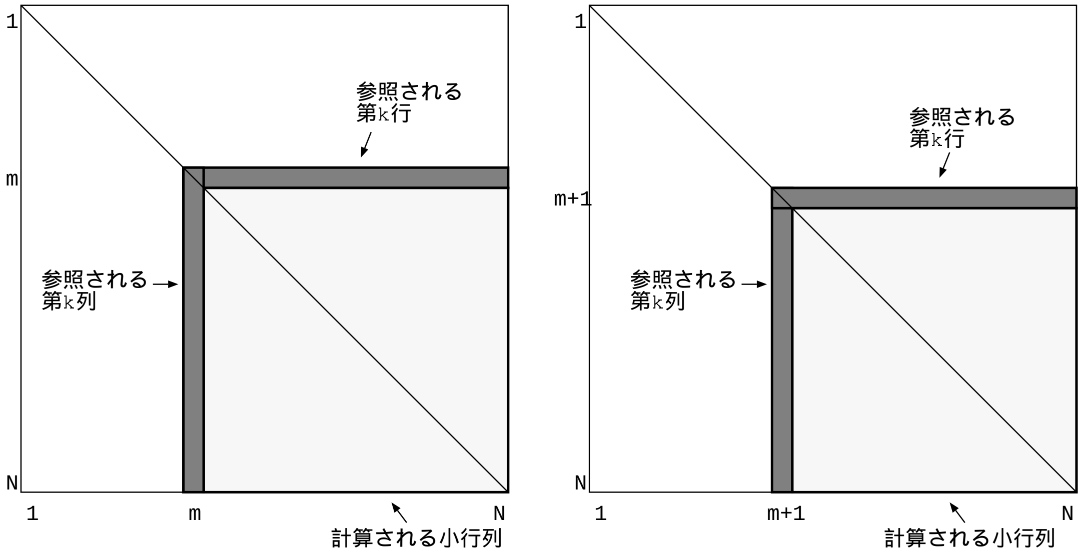


図 4.2: 外積ガウス法

特徴がある.

4.1.3 浮動小数点演算数

LU 分解に関して, 浮動小数点演算数を計算する。

外積ガウス法で計算をする。内積ガウス法とは、ロードストアの回数は異なり計算順序が違うのみで、1 つの要素に対する計算は同じである。

図 4.2を参考にする。$m-1$ 列までの消去が終わり、これから m 列と m 行を参照して、右下の小行列の計算に移るとする。

(1) まず、m 列の計算を行う。ピボットの逆数と $i=m+1$ から $i=N$まで掛算する。

$$A(i,m) = A(i,m) * \frac{1}{A(m,m)}$$

(2) 右下の小行列に対する演算。$i=m+1$ から $i=N$、$j=m+1$ から $j=N$ までの計算

$$A(i,j) -= A(i,m) * A(m,j)$$

$N-m=l$とすると、(1) では l回の除算が必要である。$m=1$ から $m=N-1$ まで計算が行われるのでよって、計算回数は

$$\sum_{m=1}^{N-1} l = \sum_{m=1}^{N-1} N-m = \frac{N(N-1)}{2}$$

(2) では、l^2回の乗算と減算が行われるので、

$$\sum_{m=1}^{N-1} l^2 = \sum_{m=1}^{N-1} m^2 = \frac{N(N-1)(2N-1)}{6}$$

となる。ゆえに、計算回数の合計は、

$$\frac{N(N-1)}{2} + \frac{N(N-1)(2N-1)}{6} = \frac{(N-1)N(N+1)}{3}$$

となるので、十分大きい N に対して、LU 分解をするときの演算コストは、$\frac{n^3}{3}$ の乗算と $\frac{n^3}{3}$ の減算であることがわかる。

4.2　LU 分解への応用

LU 分解においては、そのほとんどが内積計算の繰り返しである。ピボットと呼ばれる対角線成分によって、計算の範囲が区切られてはいるが、行列積と同じく 3 重ループで各ループに対してストリップマイニングを適用した場合、ループ交換が可能であるので本手法が適用可能である。

LU 分解の最適化手法として、外積ガウス法を最適化した外積ブロックガウス法が一般的に知られている。内積ガウス法を本手法を用いて最適化を施した。

4.3　1 次キャッシュレベルの最適化

内積ガウス法を LU 分解に本手法を応用し、レジスタレベル、1 次キャッシュレベルの最適化を行う。

実際のアクセスパターンを図 4.3で説明する。まず、phasing を用いてレジスタレベル、1 次キャッシュレベルでの最適化を行ったとする。まず、1kj のブロックの要素を左の 1s の三角形のブロックを参照して計算する。上の 1kj のブロック (A(j,k)) を毎回ループで参照し (1 次キャッシュにヒットさせながら)、左の 1ik(A(i,k)) のブロックを参照して 1ij(A(i,j)) の計算を行う。

次に、下の 2kj のブロックの要素を左の 2s の三角形を参照して計算する。2kj のブロック (A(k,j)) を毎回ループで参照しながら,2ik のブロック (A(i,k)) を参照して 2ik(A(i,j)) の計算を行う。下の 3kj のブロックの要素を左の 3s の三角形を参照して計算する。3kj のブロック (A(k,j)) を毎回ループで参照しながら,3ik のブロック (A(i,k)) を参照して 3ik(A(i,j)) の計算を行う。

ピボットまで達すると、4kj の上三角のブロックを参照して、ピボット以下の計算を行う。計算が終了すると、右横の列の計算に移る。

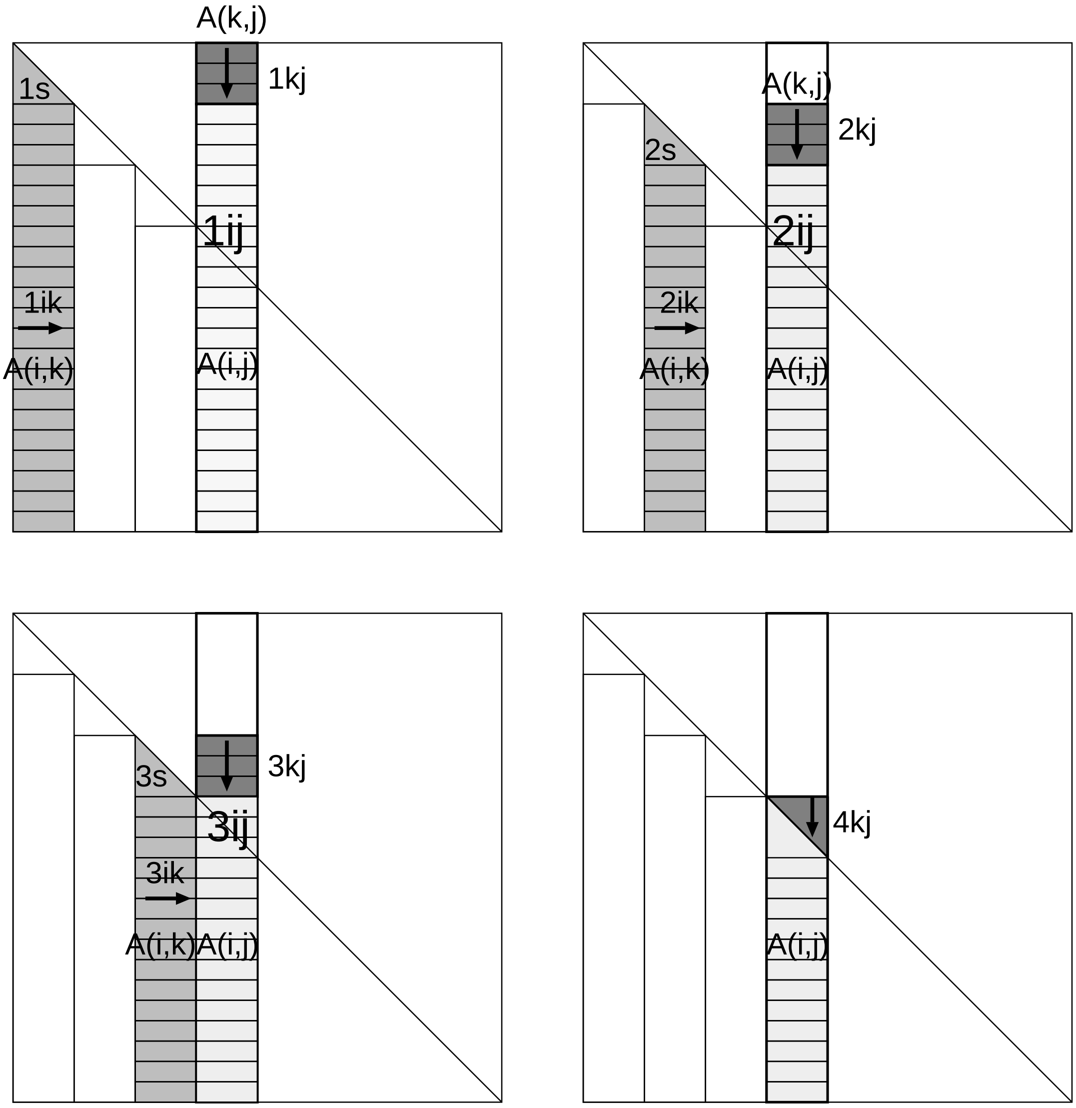


図 4.3: LU 分解の 1 次キャッシュレベルの最適化

4.4 2 次キャッシュレベルの最適化

2 次キャッシュレベルまで、内積ガウス法を本手法を用いて最適化を行う。

図 4.4で説明する。1 次キャッシュまでの最適化のときは、計算結果がストアされる A(i,j) の I 方向のループは $i = N$まで計算を行うが、2 次キャッシュまでの最適化をするときは、i 方向のループのストリップサイズを少なくし、その分 j 方向のループのストリップサイズを大きくし、キャッシュにミスヒットする左の要素 A(i,k) のミスヒットを減らす。

まず、1ij の小行列を 1s の下三角小行列を使って計算する。次に、2ij の小行列を 1ij と 1ik の小行列を使って計算した後、2s の下三角小行列を使って計算す

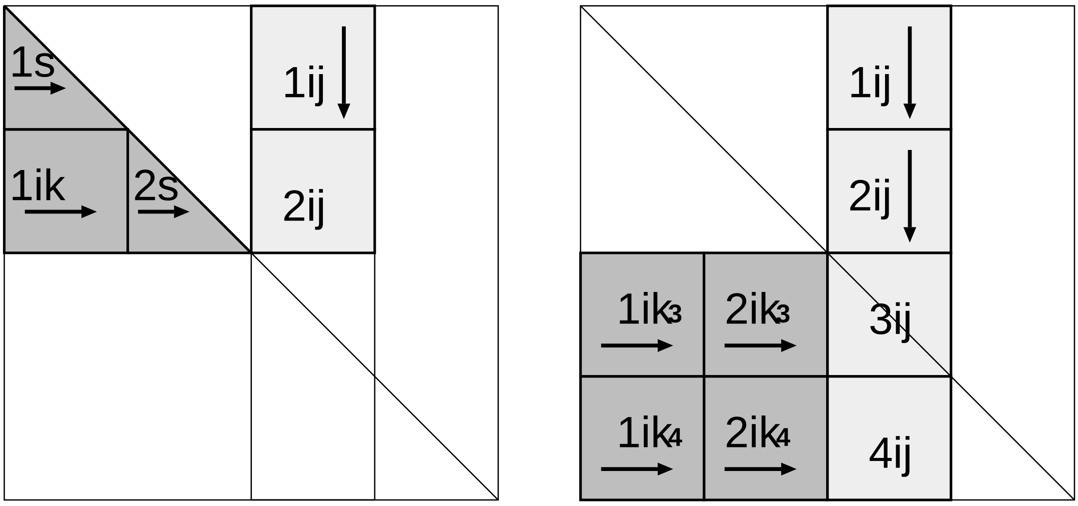


図 4.4: 2 次キャッシュまでの LU 分解の最適化

る。同じように、3ij と 4ij の計算を行う。これを繰り返す。

4.5　従来の最適化手法

外積ガウス法に、1 次キャッシュレベルまでの最適化を行うと、一般的に良く知られている LU 分解の最適化手法、外積ブロックガウス法になる。

図 4.5で説明する。まず 1kj の行と 1ik の列の計算を行い、1kj の行と 1ik の列を参照して、その右下の小行列 1ij の計算を行う。つぎに、2kj の行と 2ik の列を計算し、2kj の行と 2ik の列を参照して、その右下の小行列 2ij の計算を行う。3kj の行と 3ik の列を計算し、3kj の行と 3ik の列を参照して、その右下の小行列 3ij の計算を行う。

これを繰り返す。

4.6　本手法と従来手法との比較

A(i,j) -=A(i,k)*A(k,j) の計算におけるデータのロードとストアについて、比較を行う。

A(i,j) の計算をする前に、A(i,k) と A(k,j) のブロックの計算を行っている。両方のブロックの計算において、1 次キャッシュレベルおよび 2 次キャッシュレベルで最適化した LU 分解では、2 次キャッシュからロードしてくるのに対して、外積ブロックガウス法では、メモリからロードしてくるので、本手法の方が 2 次キャッシュのヒット率は高い。

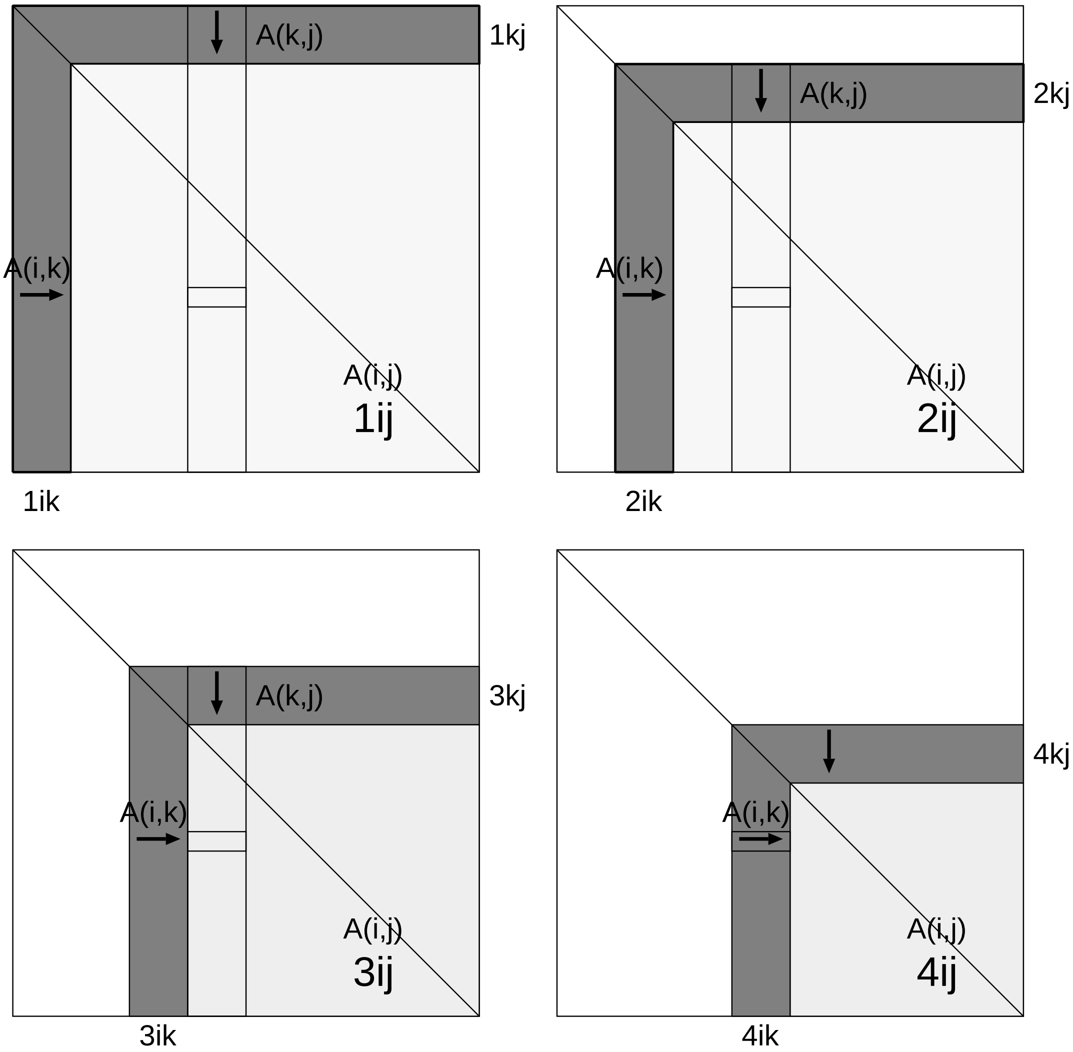


図 4.5: 従来の LU 分解の最適化手法

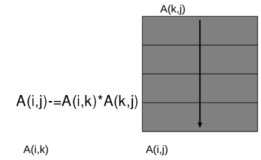


図 4.6: データの参照関係

次に,A(i,j) の計算をするときには、両手法とも A(k,j) の要素は毎回ループで参照され, キャッシュの容量に合わせてストリップマイニングされているので必ず 1 次キャッシュにヒットする。また両手法とも、それ以外の A(i,j) と A(i,k) の要素は 1 次キャッシュにミスヒットする。ゆえに両手法とも、1 次キャッシュ

のヒット率は同じである。

1次キャッシュレベルまで最適化したLU分解では、A(i,j)の要素は2次キャッシュからロードして、A(i,k)の要素はメモリからロードしてくることになる。2次キャッシュレベルまで最適化したLU分解では、A(i,j)の要素は2次キャッシュからロードして、A(i,k)の要素は2次キャッシュからロードしてくることになる。外積ガウス法においてはA(i,j)の要素はメモリからロードして、A(i,k)の要素は2次キャッシュからロードしてくる。

ところで、A(i,k)の要素はロードのみであるが、A(i,j)の要素はロードとストアを伴う。本手法では、ロードとストアを伴うデータが2次キャッシュにあるが、外積ブロックガウス法の場合にはメモリにある。

よって、本手法を適用したLU分解の方が有利である。

4.7　実際の実装

実際にphasingを応用したLU分解を実装するときであるが, 内積の計算

$$a(i,j) - = a(i,k) * a(k,j)$$

の計算を行うカーネルルーチンを用意して、LU分解を行うプログラムから関数コールするようにした.

カーネルルーチンの中身であるが、i方向に4、j方向に32、k方向に32のストリップサイズを持つルーチンを作った

なお、カーネルルーチンはアセンブリ言語で書いた.

カーネルルーチンを最適化することによって, LU分解を高速化することができる.

4.8　実行結果

LU分解で評価比較を行った。

4.8.1　行列のサイズによる比較

外積形式ガウス法を1次キャッシュのレベルまでタイリングを施したのは、外積ブロックガウス法として一般的に知られている。

本手法を用いて、内積ガウス法を1次キャッシュレベル、2次キャッシュのレベルまで最適化を行い評価比較を行った。1次キャッシュレベルまでの最適化をすると、外積ブロックガウス法と最外側のループだけがことなり、内側のルー

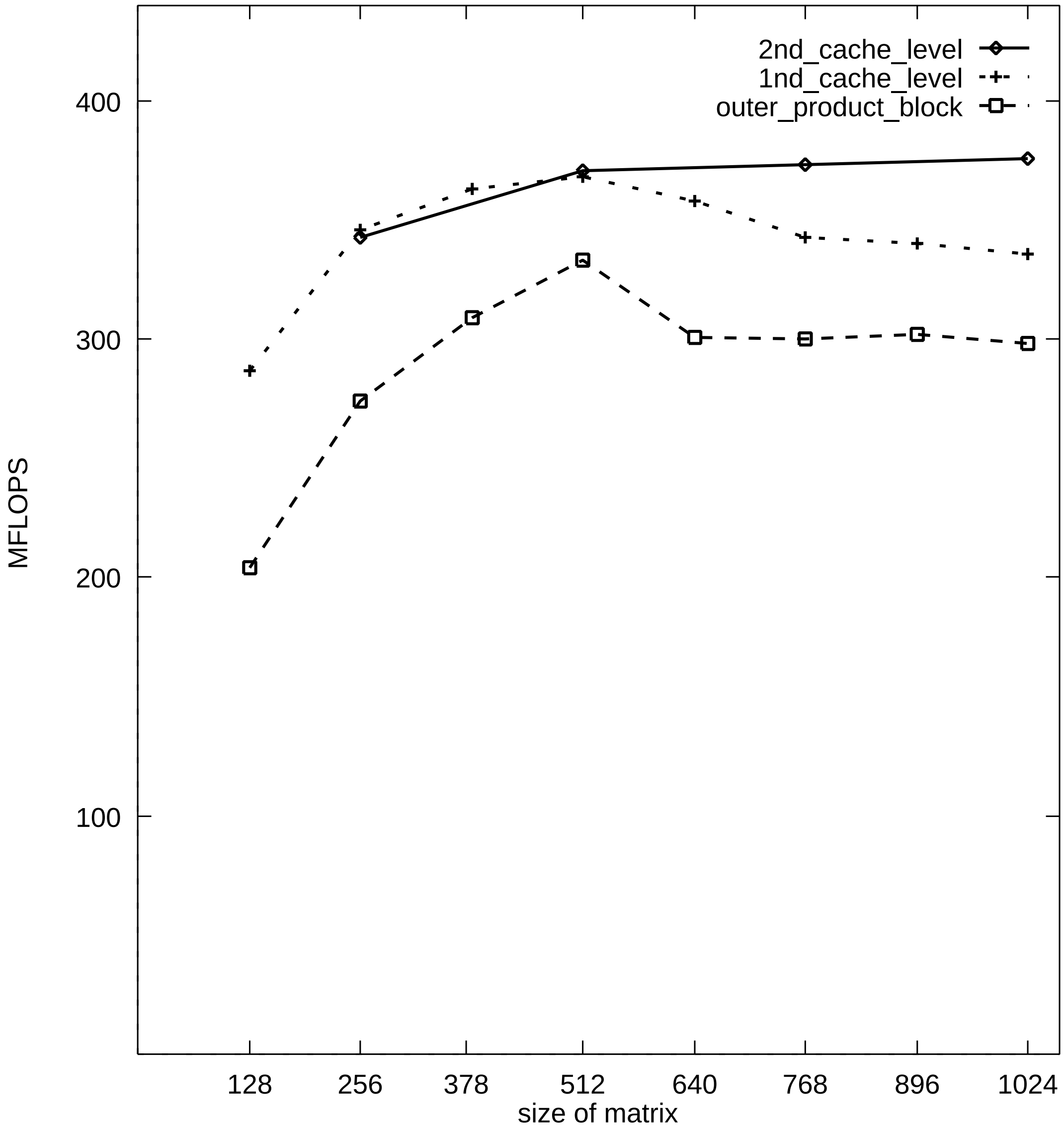


図 4.7: LU 分解の最適化

プはまったく同じである。評価に用いたマシンは UltraSparc/model 2300 であり、ピークパフォーマンスは 600Mflops である。図 4.7を参照する。外積ブロックよりも本手法が性能向上が見られるのは、前章の内容にあるように、式の左辺にあらわれる配列参照を優遇しているからである。行列のサイズ $N = 512$ を越えると、2 次キャッシュからあふれるので、外積ブロックと 1 次キャッシュまで最適化した LU 分解とも性能が落ちている。2 次キャッシュまで最適化した LU 分解には性能低下は見られない。2 次キャッシュまでの最適化の効果があらわれている。

4.8.2 カーネルルーチンによる比較

行列のサイズが $N = 1024$ のときにおいて計測を行った。カーネルルーチンの浮動小数点数による比較をする。

内積ガウス法をレジスタレベル、1 次キャッシュレベルの最適化をした LU 分解と、2 次キャッシュレベルまでの最適化をした LU 分解と外積ブロックガウス法を同じカーネルルーチンを用いて評価比較した。

アセンブリ言語で書いた内積計算をするカーネルルーチンとそれ以外の C 言語で書いたルーチンをわけて計測を行った。カーネルルーチンを 100 万回ループでまわし、1 次キャッシュのヒット率がほぼ 100%となるように、計測したときの浮動小数点演算数は 455mflops である。本手法のパフォーマンスの効果を厳密に調べるために、計測部分は LU 分解本体のみとし、オーダーが N^2 となる定量的な前進代入および後進代入は除いた。LU 分解本体の浮動小数点演算回数は $\frac{2}{3}N^2$ として計測を行った。

表 4.1: 実行時間

最適化手法	外積ブロック	1 次キャッシュ	2 次キャッシュ
LU 分解	2.2193s	1.9544s	1.8453s
カーネル	1.8455s	1.7211s	1.6090s
カーネル以外	0.3737s	0.2332s	0.2362s

表 4.2: 浮動小数点演算数

最適化手法	外積ブロック	1 次キャッシュ	2 次キャッシュ
LU 分解本体	322mflops	366mflops	387mflops
カーネル	378mflops	406mflops	434mflops

外積ブロックガウス法では、計算結果をストアする A(i,j) がメモリにあるのに対して、1 次キャッシュまで最適化した LU 分解では、行列のサイズが $N = 1024$ の場合においては、2 次キャッシュにのるので、パフォーマンスが向上する。また、カーネルルーチンで処理できないピボット付近の処理についても、本手法を用いた LU 分解のほうが、キャッシュのヒット率が高いので性能が向上して

いる。

また、1次キャッシュまで最適化したLU分解と2次キャッシュまで最適化したLU分解では、1次キャッシュのヒット率は、前者の方が良いが、2次キャッシュのヒット率は、後者の方が良い。1次キャッシュのヒット率の低下によるロスよりも2次キャッシュのヒット率の向上によるメリットが大きいので、全体としてのパフォーマンスが向上している。

5 おわりに

本稿においては, 各記憶階層の容量に合わせたループの再構成法を提案した。タイリングでは、各ループに対して、ループインターチェンジとストリップマイニングを適用することによりループの最適化を行う。また、各記憶階層のに対して多重にタイリングは適用可能である。

本手法では、各記憶階層に対して多重にタイリングを施すことによって(1次)キャッシュのヒット率向上だけではなく、各記憶階層のヒット率向上をさせる際に、、タイルサイズとループの順序を決定する方法である。

あるレベルの記憶階層のヒット率は、その上のレベルの記憶階層のヒット率とある程度相反するものとなるのは避けられない。しかし、タイルサイズはアクセス時間がより大きい記憶階層のみから決定しても良いことがわかった。また、各記憶階層の容量を有効利用するために、処理順序を決定する。

本手法を用いると、問題のサイズがどのように変化しようが、また、各記憶階層の容量や段数がどのように変化しようが、それに合わせた最適化が可能である。もちろん、その条件に最も最適といえるわけではないが、それに近い最適化が可能となる。

本稿では、行列積とLU分解において実装した。特に、LU分解では、行列のようにストリップサイズが一定ではなく、ピボット周辺で動的に変化するためプログラムの最適化にかなりの時間を費やした。

また、すべてのコードをC言語で書いたプログラムでは、コンパイラにとっては、ループのストリップサイズが一定であるため、外積ブロックガウス法の方が最適化しやすく、本手法を適用した内積ガウス法では、ストリップサイズが動的に変化するので最適化が難しい。しかし、プログラムの大部分をしめる内積計算をストリップサイズが一定となるカーネルルーチンにして切り出すことにより最適化がより容易となった。

最終的には、カーネルルーチンをアセンブリ言語で書くことによって、かなりのパフォーマンス向上がはかれた。同じカーネルルーチンを用いて、従来の外積ブロックガウス法と計算を行い、評価比較を行った結果、本手法を適用したLU分解が高い性能が得られた。

Linpack等のベンチマークで、LU分解をするプログラムの最高性能が発表されている。SunのUltraSparc2では、行列のサイズが$N = 1000$のときに、ピークパフォーマンスの約7割程度である。残念ながら、世界最高速とはいかなかったが、それに近い高い性能が得られた。

謝辞

本研究の機会を与えてくださり、適切な御指導を賜わった富田眞治教授に深甚な謝意を表します。

また、貴重なご助言をいただいた森眞一郎助教授、五島正裕助手に深く感謝致します。

さらに，共同研究者である田中利彦氏、山本考伸氏をはじめとして，日頃から御鞭撻下さった京都大学大学院情報学研究科通信情報システム専攻 富田研究室の諸兄に感謝致します。

参考文献

[1] 山本 大河内 ：「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.pp217-224.1995.

[2] M.Wolfe: 「More Iteration Space Tiling」Proc. of Supercomputing 1998.

[3] 津田孝夫 ：「数値処理プログラミング」岩波書店.1988.

[4] 建部 修見 ：「分散メモリ型並列計算機によるLU分解」第57回研究発表会(ハイパフォーマンスコンピューティング). pp55-60. 1995.

[5] 寒川 光 ：「LU分解のブロック化と計算順序」第43回研究発表会(数値解析).43-1.1992.

[6] 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 Oct.Vol38 No.10.pp1183-1192. 1992.

[7] 寒川光 ：「超高速RICSプログラミング」共立出版.1995.

[8] 小国 力 「行列計算ソフトウェア.WS、スーパーコン、並列計算機」丸善出版.1991.

[9] 坂根広史：「EM-X による密行列計算の細粒度並列処理」, 並列処理シンポジウム JSPP '97.pp29-36.1997.

[10] 富田眞治. 並列コンピュータ工学. 昭晃堂.1996.

[11] 富田眞治. コンピュータアーキテクチャ1. 丸善.1994