修士論文

ループネストを多次元的にベクトル化するコンパイラ最適化

A Compiler Optimization Technique to Multidimentionally Vectorize Loop Nests

平成27年2月5日提出

指導教員　田浦 健次朗 准教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-136454 早水 光

概 要

科学技術計算のようなアプリケーションでは実行時間のほんとんどがループネストで消費される傾向にある．また，近年の多くのプロセッサは SIMD 命令セットを持ち，SIMD レジスタの長さも増加傾向にある．このような背景では，ループネスト中で SIMD 命令を活用することの重要性が増してくることとなる．ループで SIMD を活用する方法にループの SIMD 化と呼ばれるループ変換が知られている．プログラマが自分で変換しなくとも，既存の多くのコンパイラはループネストの最内ループに対して自動で行なってくれることがある．
本研究ではコンパイラによるループネストに対して多次元的に SIMD 化を行うような，新しいループ変換技術を提案する．このような変換をループの多次元 SIMD 化と呼ぶこととする．多次元 SIMD 化は既存の SIMD 化と違い，メモリアクセス回数と計算に必要なレジスタ数の削減が期待でき，性能向上が見込まれる場合がある．また，多次元 SIMD 化を LLVM 上で実装し自動で変換できるようにした．評価において，３つのアプリケーションで実験を行い，行列積のアプリケーションで LLVM の最適化以上の性能向上を確かめることが出来た．

目 次

第 1 章 序論 **1**
1.1 ループに対するコンパイラ最適化 1
1.2 目的 1
1.3 本稿の構成 2

第 2 章 LLVM **3**
2.1 LLVM のアーキテクチャ 3

第 3 章 ループ変換 **5**
3.1 コンパイラによるループ変換 5
3.1.1 ループ変換の正当性 5
3.1.2 ループ変換の有益性 7
3.2 ループの SIMD 化 7
3.2.1 概要 7
3.2.2 SIMD 7
3.2.3 変換例 8
3.2.4 正当性 8
3.3 ループタイリング 9
3.3.1 概要 9
3.3.2 変換例 9

第 4 章 関連研究 **11**
4.1 ループの SIMD 化 11
4.2 Polyhedral Model 11
4.3 多次元 SIMD 化 12

第 5 章 多次元ベクトル化 **13**
5.1 多次元 SIMD 化 13
5.2 具体例: 3 次元ベクトル化 14

第 6 章 実装 **20**
6.1 LLVM のループ SIMD 化 20
6.1.1 ベクトル命令生成 21
6.2 多次元 SIMD 化 22
6.2.1 Polly 22

6.2.2 CFG 23
6.2.3 イテレーションのインスタンスの実行順序 23
6.2.4 ベクトル命令生成 24

第 7 章 評価 **27**
7.1 評価環境 27
7.2 実験 27

第 8 章 結論 **31**
8.1 まとめ 31
8.2 今後の課題 31
謝辞 31
発表文献 33
参考文献 33

図 目 次

2.1 LLVM のアーキテクチャ 4

3.1 ループネスト 6
3.2 図 3.1 のイテレーションスペース 6
3.3 スカラー演算と SIMD 演算 8
3.4 SIMD 化前 8
3.5 SIMD 化後 9
3.6 The iteration space 10
3.7 行列積 : ループタイリング前 10
3.8 行列積 : ループタイリング後 10

5.1 スカラーループ 14
5.2 １次元 SIMD 化ループ (VF=4) 15
5.3 ２次元 SIMD 化ループ (VF=4) 15
5.4 Iteration space (1D SIMDization) 16
5.5 Iteration space (2D SIMDization) 17
5.6 One iteration of 2D SIMDized code ($VF = 4$) 17
5.7 行列積 18
5.8 ３次元 SIMD 化ループ (VF=4) 19

6.1 LLVM のループ SIMD 化 20
6.2 LLVM により SIMD 化された後の制御フローグラフ 21
6.3 Polly によるタイリング 23
6.4 N 次元 SIMD 化されたコードの制御フローグラフ 24
6.5 タイル内での計算実行順序 25
6.6 N 次元 SIMD 化 26

7.1 Multiply and Add 28
7.2 性能比較 (MulAdd) 28
7.3 N 体シミュレーション 29
7.4 性能比較 (N 体) 29
7.5 行列積 30
7.6 性能比較 (行列積) 30

表 目 次

第1章　序論

1.1　ループに対するコンパイラ最適化

プログラム中に現れるループ，特にループネスト中の命令は実行される回数が多く，性能に与える影響も大きいことが多いことから，ループネストを効率良く実行することは特に科学技術計算において強く要求されてきた．プログラマ自身がループネストが効率良く実行されるように記述することは可能ではあるが，実行する CPU についての正確な知識や経験などが要求され、容易なことではない．プログラマビリティや生産性の観点から，コンパイラによる自動的なループへの最適化は重要であると考えられ，長年様々な研究がされてきた [12, 1, 2]. さらに，近年の多くのプロセッサは「SIMD(Single Instruction Multiple Data)」という機能を持っている．SIMD レジスタという複数のデータを保持できるレジスタに対する SIMD 命令を実行することができる．近年，プロセッサの理論性能の向上はクロック周波数の向上によるものというより，SIMD レジスタの大きさの増大による傾向になっている．また，大きな SIMD レジスタを持つ Xeon Phi などのアクセラレータも市場に出回っている．このような状況から，ループを SIMD 命令を使って最適化することの重要性が高まっている．そのような最適化の典型的なものにループの SIMD 化というものがある．
ループの SIMD 化とは簡単に説明すると，ループのイテレーションを SIMD レジスタの大きさだけ並列に処理するようにループを変換するようなもので，単純な場合には，数倍の性能向上が見込める．GCC，ICC や LLVM などのコンパイラはループの SIMD 化が実装されており，プログラムの性能向上に貢献している．これらの既存のコンパイラでは典型的にはループネストの最内のループに対して SIMD 化を行っている．

1.2　目的

本研究ではコンパイラによるループネストに対して多次元的に SIMD 化を行うような，新しいループ変換技術を提案する．このような変換をループの多次元 SIMD 化と呼ぶこととする．多次元 SIMD 化は既存の SIMD 化と違い，メモリアクセス回数と計算に必要なレジスタ数の削減が期待でき，性能向上が見込まれる場合がある．
多次元 SIMD 化を行う最適化パスを LLVM に実装し，実験を行った．LLVM は言語やアーキテクチャから独立したコンパイラ基盤であり，種々の最適化がモジュール化されているので，自作の最適化を組み込むことが比較的容易であることから採用した．我々の研究室で手動で N 体問題の直接解法を多次元 SIMD 化し評価した結果，性能が向上したことも確認できた [13].

1.3　本稿の構成

3 章で本研究と関連の強いループ変換である Loop Vectorization と Loop Tiling のそれぞれについて説明する．4 章では本研究と関連する研究を紹介する．5 章では本研究の提案である，多次元 SIMD 化について説明する．6 章では実装の詳細を述べる．7 章で評価を行い，8 章で結論を述べる．

第2章　LLVM

本章では，本研究の実装のプラットフォームである，LLVM について説明する．

2.1　LLVM のアーキテクチャ

従来のコンパイラの典型的なアーキテクチャは以下の通りである．

フロントエンド プログラムに対して字句解析，構文解析を行い，抽象構文木 (Abstract Syntactic Tree, AST) などの独自の中間表現コードを生成する．

ミドルエンド 中間表現に対し最適化を行い新たな中間表現のコードを生成する．

バックエンド 中間表現コードを特定の命令セットの機械語を出力する．

このようなアーキテクチャのため，異なるプログラミング言語のために異なるコンパイラが必要であり，異なるマシンに対しても異なるコンパイラを用意する必要があった．このような理由から，コンパイラが古い技術のまま進化してしまったり，モジュールの再利用性がないなどの問題があった．そこで，LLVM は共通の中間表現 **LLVM-IR** を定義し，プログラミング言語を LLVM-IR に変換するフロントエンドさえ作れば，共通のミドルエンド，バックエンドによりコード生成を行うことを可能にした．例えば，clang というコンパイラは LLVM の C/C++, Objective-C/C++ のフロントエンドと LLVM という構成のコンパイラである．また，新しい命令セットに対しても，LLVM のバックエンドに手を入れることでコード生成が可能となる．また，LLVM は再利用性がたかくなるように，上手くモジュール化されており，LLVM-IR に対する解析や最適化はそれぞれ個別の独立した **Pass** として実装されている．このような設計により，最適化を好きに組み合わせて施したり，独自の最適化の実装もやりやすくなっている．

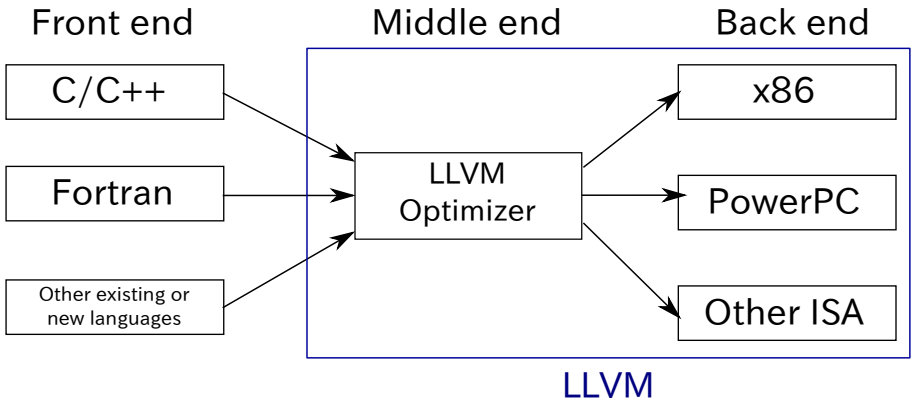

図 2.1. LLVM のアーキテクチャ

第3章　ループ変換

コンパイラによって施されるループに対する最適化はループ変換と呼ばれ，様々なループ変換が研究され，実装されている．中でも本研究に関係の強いループの SIMD 化 (ベクトル化) とループタイリングについて，この章で説明する．

3.1　コンパイラによるループ変換

ループ変換は概して，ループのイテレーションの実行順序を入れ替えることに相当し，その入れ替え方で分類される．コンパイラはコンパイラ内部での中間表現上でループ変換を行う．ループ変換を行う際，コンパイラはその変換の正当性・有益性を検証した上で実施する．両方とも，#pragma などによるソースコード上のアノテーションや，コンパイラオプションによる指定などにより，検証が無視されることもある．

3.1.1　ループ変換の正当性

ループ変換の正当性とは変換の前後でプログラムの計算結果がかわらないことを意味する．イテレーション間に同じメモリ領域へのアクセスがあるとき，ループ内に依存関係があるという．依存関係にあるイテレーションの相対的な順序関係が保存されるとき，プログラムの意味が変わらないとして，ループ変換の正当性を検証する．問題となる依存関係は以下の３つのパターンである．

フロー依存 あるイテレーションが読みこんだ後のメモリ領域に別のあるイテレーション が書きこんだときの依存関係

逆依存 あるイテレーションが書きこんだ後のメモリ領域に別のあるイテレーション が読みこんだときの依存関係

出力依存 あるイテレーションが書きこんだ後のメモリ領域に別のあるイテレーション が書きこんだときの依存関係

ループ変換はこれらの依存関係を保存するように変換しなければならない．さらに，ループ内の依存を表現する用語をここで定義しておく．

イテレーションベクトルとイテレーションスペース

あるループネスト中の文について，イテレーションを特定するベクトル．最外から最内のループに順番にイテレーションナンバーを並べたベクトルである．イテレーションナンバーとは１から始まて対応するループのイテレーションが実行された順に番号を振ったもので，図 3.1 ではあ

るイテレーションのイテレーションベクトルは $\mathbf{i} = (i, j, k)$ と表現できる．このように定義すると，イテレーションの実行順序はイテレーションベクトルの辞書順であることがわかる．例えば，$\mathbf{i}_0 = (i, j, k+1)$ は $\mathbf{i}_1 = (i, j+1, k)$ は $\mathbf{i}_0$ の方が先に実行される．
あるループネストのイテレーションベクトルが取りうる値の集合をイテレーションスペースと呼ぶ．例えば，図 3.1 で $L = M = N = 2$ の時，イテレーションスペースは

$$\{(1,1,1), (1,1,2), (1,2,1), (1,2,2), (2,1,1), (2,1,2), (2,2,1), (2,2,2)\}$$

となる．深さ D のループネストの場合，D 次元の空間上にイテレーションベクトルを格子点として表現することができて，イテレーションスペースは D 次元空間上の格子点の集合と捉えることもできる．

```
1 for(i=1;i<=L;i++)
2   for(j=1;i<=M;j++)
3     for(k=1;k<=N;k++)
4       A[i+1][j][k−1] = A[i][j][k] + 10;
```

図 3.1. ループネスト

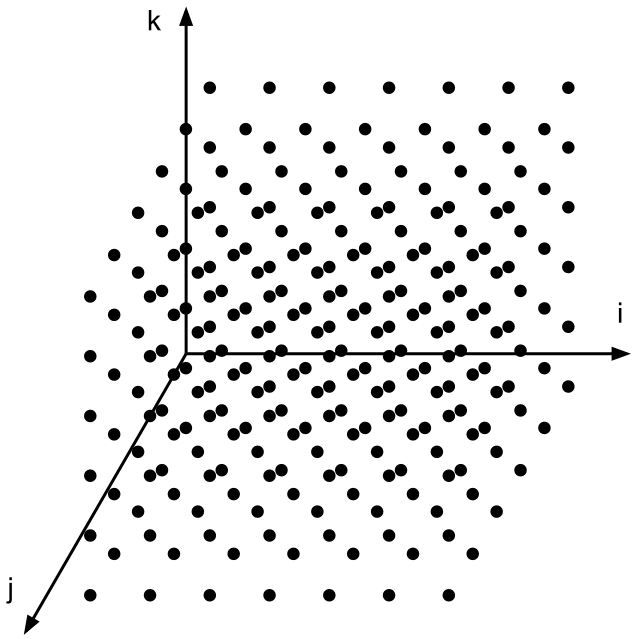

図 3.2. 図 3.1 のイテレーションスペース

ディスタンスベクトルとディレクションベクトル

イテレーション間の依存を表現する方法として，ディスタンスベクトルがある．依存元のイテレーション $\mathbf{i}$ と依存先のイテレーション $\mathbf{j}$ の距離を表すもので，図 3.1 の例では，$\mathbf{d}(i,j) = \mathbf{j} - \mathbf{i} = (i+1, j, k-1) - (i, j, k) = (1, 0, -1)$ となる．ディレクションベクトルは依存元から依存先への向きを表現するもので，ディレクションベクトルの符号に対応して $(<, =, >)$ や，$(+, 0, -)$ と表現されたりする．ディレクションベクトルの性質として，最も左の位置の $=$ でない要素は $<$ でなければならない．イテレーションベクトルが辞書順に実行されるので，もし最も左の位置の $=$ でない要素が $>$ だと依存元のイテレーションが依存先のイテレーションより先に実行されることになり定義とずれる．さらに，ループ変換が正当であるとき，変換後のディレクションベクトルの最も左の位置の $=$ でない要素が $>$ であるものは存在しない．

3.1.2　ループ変換の有益性

ループ変換の有益性とは変換後に性能が向上することである．確実に有益性を保証するためには実行をシミュレートする方法があるが，コンパイラにとってコンパイル時間が短いことも求められるため，大抵は変換前後のコストを大雑把に見積もって比較する方法を取る．

3.2　ループの SIMD 化

3.2.1　概要

ループの SIMD 化はあるループを SIMD レジスタの大きさだけイテレーションをまとめて並列実行するように変換する最適化である．まず，SIMD の機能について詳しく説明し，変換例を示す．

3.2.2　SIMD

SIMD とは Single Instruction Multiple Data の略であり，コンピュータアーキテクチャ上の演算の並列化の分類の１つである．１つの命令が複数のデータに対して適用されるような並列化を SIMD と呼ぶ．具体的に説明すると，逐次の演算が１つのデータに対して１つの命令を適用していくのに対して，SIMD の演算は同じ型の複数のデータを格納できるレジスタに対して１つの命令が適用される．このような複数のデータを格納できるレジスタを **SIMD レジスタ**や**ベクタレジスタ**と呼び，SIMD 用の命令を **SIMD 命令**や**ベクトル命令**と呼ぶ．また，ベクタレジスタがあるデータ型を格納できる個数を **SIMD 幅**，**ベクトル幅**と呼ぶ．
SIMD は同じ演算を大量のデータに対して同じような計算を行うような場面で性能を発揮しやすく，音声・画像・映像のようなマルチメディア処理や，科学技術計算によく利用される．
SIMD 命令はよく拡張命令セットとしてアーキテクチャに追加され，最近の多くのプロセッサに採用されている．例えば，x86 の SSE/AVX や，PowerPC の AltiVec，ARM の NEON などがある．SIMD レジスタの長さや SIMD 命令の種類はアーキテクチャによって異なっており，統一されていない．また，同じアーキテクチャでも命令セットによって SIMD レジスタの長さが増えたり，新たな命令が追加されたりする．例えば，x86 の SSE 命令セットでは 128bit の SIMD レジス

タを持つが，x86 の AVX 命令セットでは 256bit の SIMD レジスタを持つ．SIMD 命令セットによって違いは存在するものの，SIMD レジスタにメモリからデータをロードする際，メモリアドレスは SIMD レジスタの長さにアラインされていること，データはメモリ上で連続していること，といった制約はおおよそ共通している．また，GPU のようなグラフィック処理に特化したアクセラレータも SIMD 型の処理を行っており，SIMT(Single Instruction Multiple Thread) と呼ばれるような，より特殊な実行形式をとっているが，本稿では特に CPU 上の SIMD についてのみ考慮に入れて議論を進めていく．

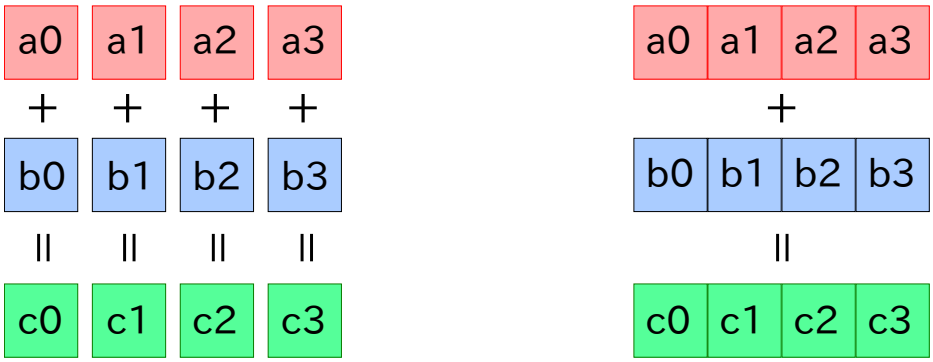

図 3.3. スカラー演算と SIMD 演算

3.2.3　変換例

ここで，ループの SIMD 化の変換例を示す．図 3.4，3.5 に C 言語ライクな擬似コードを示した．図 3.5 において，VF は SIMD レジスタに格納できるデータ数を表している．図 3.5 の SIMD 化されたループはループの回数が約 N/VF 回に削減されており実行される命令数も約 $1/VF$ に削減されており，性能向上されてることが期待できる．ここでは，ソースコードレベルの変換例を示したが，コンパイラは内部の中間表現について行う．

```
1 int i;
2 float *a, *b, *c;
3 for(i=0;i<N;i++){
4   c[i] += a[i] + b[i];
5 }
```

図 3.4. SIMD 化前

3.2.4　正当性

ループのイテレーションが並列に実行できるのは異なるイテレーション間の依存関係が無いときであると知られている．SIMD 化されたループは VF 毎に並列に実行される．つまり，ディス

```
1  for(i=0;i<N;i+=VF){
2    va = SIMD_LOAD(&a[i]);
3    vb = SIMD_LOAD(&b[i]);
4    vc = SIMD_LOAD(&c[i]);
5    vc += va + vb;
6    SIMD_STORE(&c[i], vc);
7  }
8  for(;i<N;i++){
9    c[i] += a[i] + b[i];
10 }
```

図 3.5. SIMD 化後

タンスベクトルの SIMD 化するループと対応する要素が VF 以上であるとき，SIMD 化を行なってもプログラムの意味が変わらないと判断できる．

3.3　ループタイリング

3.3.1　概要

ループタイリングはメモリ階層を有効に使うループ最適化としてよく知られているものである．大抵, L1 キャッシュやレジスタなどのメモリアクセスの速い階層のサイズに合わせて行われる．一度アクセスされたデータをアクセスの速いメモリ階層に保持されている間に再利用することで, キャッシュミスやロード/ストアによるペナルティを削減することが, 狙いである．

3.3.2　変換例

図 3.7 の行列積のコードを例にループタイリングした例を図 3.8 に示す．なお, N が 32 の倍数であると仮定して, タイルのサイズは $32 \times 32 \times 32$ としてタイリングを行ったものである．

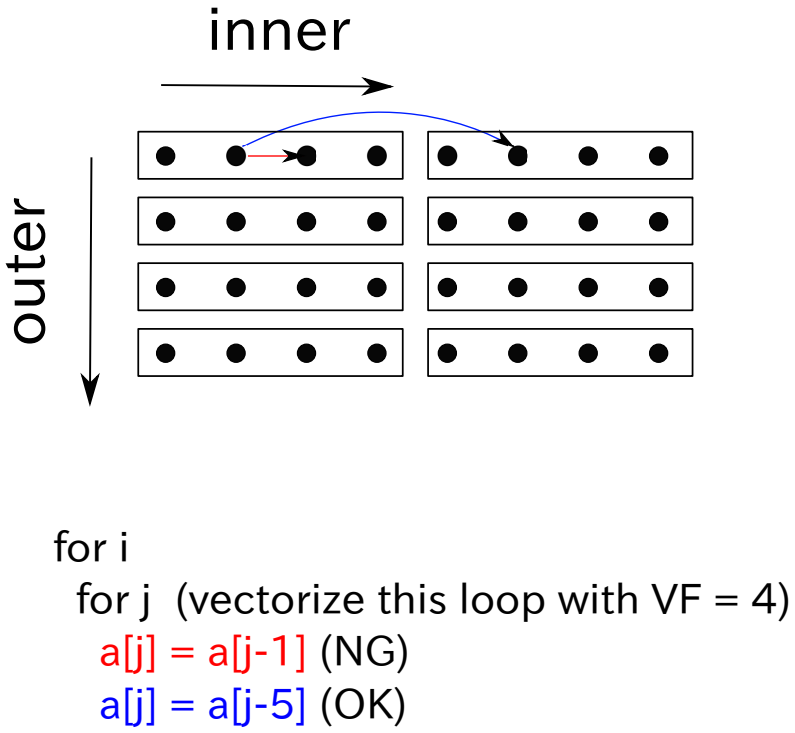

図 3.6. The iteration space

```
1  for (i=0; i<N; i++)
2    for (j=0; j<N; j++)
3      for (k=0; k<N; k++)
4        C[i][j] += A[i][k] * B[k][j];
```

図 3.7. 行列積：ループタイリング前

```
1  for (ii=0; ii<N; ii+=32)
2    for (jj=0; jj<N; jj+=32)
3      for (kk=0; kk<N; kk+=32)
4        for (i=ii; i<ii+32; i++)
5          for (j=jj; j<jj+32; j++)
6            for (k=kk; k<kk+32; k++)
7              C[i][j] += A[i][k] * B[k][j];
```

図 3.8. 行列積：ループタイリング後

第4章　関連研究

4.1　ループのSIMD化

3章で説明したようなループのSIMD化は比較的最近のSIMD演算を備えたプロセッサの能力を引き出すようなループ変換であるが，古くからベクトル型のマシン向けにループの自動ベクトル化 (Loop Vectorization) として研究されてきた [1]．ループのSIMD化の問題に，SIMD命令特有のアラインメント制約や，アーキテクチャ間のSIMD命令セットの特徴の違いなどがある．Eichenbergerら [3] はコード生成時にアラインメントされていないメモリアクセスを避ける手法を提案した．また，Nuzmanら [8] は異なるアーキテクチャ間の可搬性を考慮したGCC内のLoop SIMDizationの実装を提案し，GCCにマージされてリリースされた．さらに，最内のループについて同時実行可能で無い場合に，外側のループについてSIMD化する手法 [9] もGCCにマージされている．ループネストに対してSIMD化を行う際，ループインターチェンジというループネスト内のループの入れ替えを行ってから最内ループにSIMD化を行うという選択肢もあったが，SIMD化が最内以外にもできるようになり，どうループを入れ替え，どのループをSIMD化するという探索空間の増大という問題が出てきた．そこで，ループのSIMD化に関するコスト関数を定義し，GCC内のPolyhedral Modelframeworkに組み込むことで，SIMD化の探索空間から最適なものを選択できるような技術も開発された [11]．Polyhedral Modelについては次節で説明する．

4.2　Polyhedral Model

ループ変換というレベルでコンパイラ最適化技術に目を向けると，Polyhedral Model[7, 4, 10, 2]という非常に強力なフレームワークがある．ループ変換はLoop interchange, unroll and jam, fusion, tilingなど [12], 様々な種類のものがよく知られているが，従来のコンパイラではそれぞれの変換毎に別の最適化として扱われていたが，Polyhedral Modelではそのようなループ変換の組み合わせを統一的に数学的な操作で表現できるモデルを与えてくれる．3.1.1節で説明したようなイテレーションスペースはイテレーションベクトルとループの下限と上限の値から整数連立不等式として表すことができて．これは空間上の超多面体をなす．3章で説明したようなループ変換の依存関係という制約も連立不等式を用いて多面体として表現できる．Polyhedral Modelでのループ変換は多面体を新たな多面体への変換とも捉えることができる．ある目的を持って, ループ変換を行いたいとき, コスト関数を導入することで, 整数線形計画問題に帰着し, 変換を決定することができる．このようなPolyhedral Modelを用いたLLVMでの実装にPolly[5]がある．Pollyはソースコードレベルの変換ツールであるPLuTo[2]をLLVM上に移植したようなものである．PLuToはデータ局所性と並列性を重視したループ変換を行うもので，具体的にはC言語のループネストに対してLoop TilingとOpenMPのコードの挿入を行う．Pollyではさらに，SIMD化も

行なってくれるが，1 次元的な SIMD 化であるし，完全な SIMD 化ではなく Strip Mining を行うだけのものであった．Kong ら [6] の手法では PLuTo と同じような変換を行い，さらにタイル内を SIMD 化するというものであったが，こちらも 1 次元的な SIMD 化にとどまっていた．
本研究の多次元 SIMD 化は，私の知る限り，自動生成するツールあるいはコンパイラは存在せず，新規性のある提案であると考えている．

4.3　多次元 SIMD 化

本稿の提案である多次元 SIMD 化と同じような変換を手動で N 体シミュレーションに適用した研究が本研究室で研究されていた [13]．しかし，私の知る限り，多次元 SIMD 化されたコードを自動生成するツールあるいはコンパイラは存在せず，新規性のある研究であると考えている．

第5章　多次元ベクトル化

5.1　多次元SIMD化

一般的なループSIMD化は，１つのレベルのループに沿ってデータをSIMDレジスタに格納し，SIMD演算を行うようにする変換であったが，それを多次元的に行うようなSIMD化を**多次元SIMD化**として提案する．多次元SIMD化を定義すると従来のループのSIMD化は多次元SIMD化の１次元の場合と捉えることもできるようになる．
多次元SIMD化のコンセプトを説明するために，再度，１次元SIMD化について考える．図5.2を例に，イテレーションスペースで表現すると図5.4のようになる．図5.4の長方形が１次元SIMD化されたときにまとめられたイテレーションに対応し，格子点を$VF \times 1$個含む．図5.2の最内ループでは，図5.1の以下のイテレーションベクトルの組み合わせ$(VF = 4)$が実行される．

1. $(i, j), (i, j + 1), (i, j + 2), (i, j + 3)$

多次元SIMD化は１次元SIMD化のより一般的なループ変換であり，イテレーションスペースを一方向に沿ってSIMD化するのではなく，多次元方向にSIMD化するような変換である．例えば，図5.1を多次元SIMD化するとしたら，図5.5のようになる．これは２次元のイテレーションスペースの２次元SIMD化を表現している．図5.5内の各矩形は$VF \times VF$回分のイテレーションを含むとすると，元の演算をベクトル幅VFのベクトル命令に変換してVF回行うことで達成できるはずである．$VF = 4$のときの図5.5の矩形を１つ拡大したのが図5.6である．図5.6の各セルがスカラーのイテレーションに相当し，各セルの数字が同じものはベクトル命令により同時に実行され，数字が実行順になっている．つまり，以下のループ変数の組み合わせが以下の順番に実行される．

1. $(i, j), (i + 1, j + 1), (i + 2, j + 2), (i + 3, j + 3)$
2. $(i, j + 1), (i + 1, j + 2), (i + 2, j + 3), (i + 3, j)$
3. $(i, j + 2), (i + 1, j + 3), (i + 2, j), (i + 3, j + 1)$
4. $(i, j + 3), (i + 1, j), (i + 2, j + 1), (i + 3, j + 2)$

各組み合わせに対応したベクタレジスタを用意して上記の組み合わせのベクトル命令を実行すればいいのだが，図5.1だと，iに沿ったデータが２つにjに沿ったデータが１つある．iに沿ったデータは使いまわせるとして，jに沿ったデータは違う順番でデータが格納されている．つまり，$2 + 1 * 4 = 6$個のベクタレジスタが必要になる．今回の例だとSSE/AVXベクタレジスタの本数である16本以内に収まっているので影響は少ないが，レジスタ多く消費することは余計なメモリアクセスの増加につながるので好ましくないときがある．しかし，多くのSIMD命令セットには**SHUFFLE命令**や**PERMUTE命令**といった種類の命令が存在する．これらの命令を使えば，ソースレジスタの好きな位置の要素をデスティネーションレジスタの好きな位置に格納する

といったことが可能になる．つまり，ベクタレジスタの各要素の位置の入れ替えといったことが可能になる．これを利用すると，j に沿ったデータは 1 つのレジスタで位置を入れ替えつつ演算を進めることでレジスタ数の削減ができる．つまり，この例では，使用するレジスタ数を増やす事なく 2 次元 SIMD 化が出来る．そのようにして，2 次元 SIMD 化した例が，図 5.3 である．この変換では加算や乗算などの総実行数は 1 次元 SIMD 化と比べても変わらず，むしろ，レジスタ内の要素を入れ替える命令が増えているため実行命令数が増える可能性があるが，プロセッサのスーパースカラ実行によりたいていそのような命令は加算などの実行と命令レベルで並列に実行される．よって実行時間がメモリのバンド幅で律速されるようなループより，計算で律速されるようなループに効果が期待できる．
図 5.3 は多次元 SIMD 化の概念の中でも特殊な変換に相当する．というのも，図 5.3 は 4×4 の大きさで SIMD 化されているが，AVX 命令セットでは 8×8 や 8×4 で SIMD 化することもできる．それに，ベクトル命令で同時に計算する (i, j) の組み合わせも図 5.3 と同じような組み合わせをとる必要もなく，例えば，配列 a へのアクセスが $a[5 * i]$ というようなとき，一回のロード命令では必要な要素を全て持ってくることはできず，必要な要素をベクタレジスタに集めてくる際，違う組み合わせの方が効率がよい可能性もある．特殊な例を用いて説明してきたが，多次元 SIMD 化を一般的に説明すると，ループネストの多次元方向に沿って演算に必要な要素をベクタレジスタに集めてベクトル命令を行うような変形である．さらに，違う言い方をしてみると，ベクタレジスタに収まるようなサイズでループタイリングを行い，タイル内ではデータをレジスタに保存したまま，SIMD 命令によってタイル内の演算を行うような変換とも言える．この変換はループの SIMD 化とループタイリングの両方の利点を狙えるものであり，SIMD 命令によって演算数を減らすだけでなく，レジスタの再利用性が高く，メモリアクセスの削減さえもできる．

```
1 int i,j;
2 float *a, *b, *c;
3 for(i=0;i<N;i++){
4   for(j=0;j<N;j++){
5     c[i] += a[i] * b[j];
6   }
7 }
```

図 5.1. スカラーループ

5.2　具体例: 3 次元ベクトル化

さらに，理解を深めるために 3 次元ベクトル化した例も示す．図 5.7 のような行列積コードを 3 次元 SIMD 化することを考える．5.1 節での例のように内側のループから順にずらしていくような組み合わせで $VF = 4$ として 3 次元 SIMD 化してみる．つまり，以下のような組み合わせに対応する要素を SIMD 演算していく．

1. $(i, j, k), (i + 1, j + 1, k + 1), (i + 2, j + 2, k + 2), (i + 3, j + 3, k + 3)$
2. $(i, j, k + 1), (i + 1, j + 1, k + 2), (i + 2, j + 2, k + 3), (i + 3, j + 3, k)$

```
1  for(i=0;i<N;i++){
2   va = {a[i], a[i], a[i], a[i]};
3   vc = {c[i], c[i], c[i], c[i]};
4   for(j=0;j<N;j+=4){
5    vb = SIMD_LOAD(&b[j]);
6    vc += va * vb;
7   }
8   SIMD_STORE(&c[j], vc);
9  }
```

図 5.2.　1 次元 SIMD 化ループ (VF=4)

```
1  for(i=0;i<N;i+=4){
2   va = SIMD_LOAD(&a[j]);
3   vc = SIMD_LOAD(&c[j]);
4   for(j=0;j<N;j+=4){
5    vb = SIMD_LOAD(&b[j]);
6    vc += va * vb;
7    vb = {b[j+1], b[j+2], b[j+3], b[j]};
8    vc += va * vb;
9    vb = {b[j+2], b[j+3], b[j], b[j+1]};
10   vc += va * vb;
11   vb = {b[j+3], b[j], b[j+1], b[j+2]};
12   vc += va * vb;
13  }
14  SIMD_STORE(&c[j], vc);
15 }
```

図 5.3.　2 次元 SIMD 化ループ (VF=4)

3. $(i, j, k+2), (i+1, j+1, k+3), (i+2, j+2, k), (i+3, j+3, k+1)$
4. $(i, j, k+3), (i+1, j+1, k), (i+2, j+2, k+1), (i+3, j+3, k+2)$
5. $(i, j+1, k), (i+1, j+2, k+1), (i+2, j+3, k+2), (i+3, j, k+3)$
6. $(i, j+1, k+1), (i+1, j+2, k+2), (i+2, j+3, k+3), (i+3, j, k)$
7. $(i, j+1, k+2), (i+1, j+2, k+3), (i+2, j+3, k), (i+3, j, k+1)$
8. $(i, j+1, k+3), (i+1, j+2, k), (i+2, j+3, k+1), (i+3, j, k+2)$
9. $(i, j+2, k), (i+1, j+3, k+1), (i+2, j, k+2), (i+3, j+1, k+3)$
10. $(i, j+2, k+1), (i+1, j+3, k+2), (i+2, j, k+3), (i+3, j+1, k)$
11. $(i, j+2, k+2), (i+1, j+3, k+3), (i+2, j, k), (i+3, j+1, k+1)$
12. $(i, j+2, k+3), (i+1, j+3, k), (i+2, j, k+1), (i+3, j+1, k+2)$
13. $(i, j+3, k), (i+1, j, k+1), (i+2, j+1, k+2), (i+3, j+2, k+3)$
14. $(i, j+3, k+1), (i+1, j, k+2), (i+2, j+1, k+3), (i+3, j+2, k)$
15. $(i, j+3, k+2), (i+1, j, k+3), (i+2, j+1, k), (i+3, j+2, k+1)$
16. $(i, j+3, k+3), (i+1, j, k), (i+2, j+1, k+1), (i+3, j+2, k+2)$

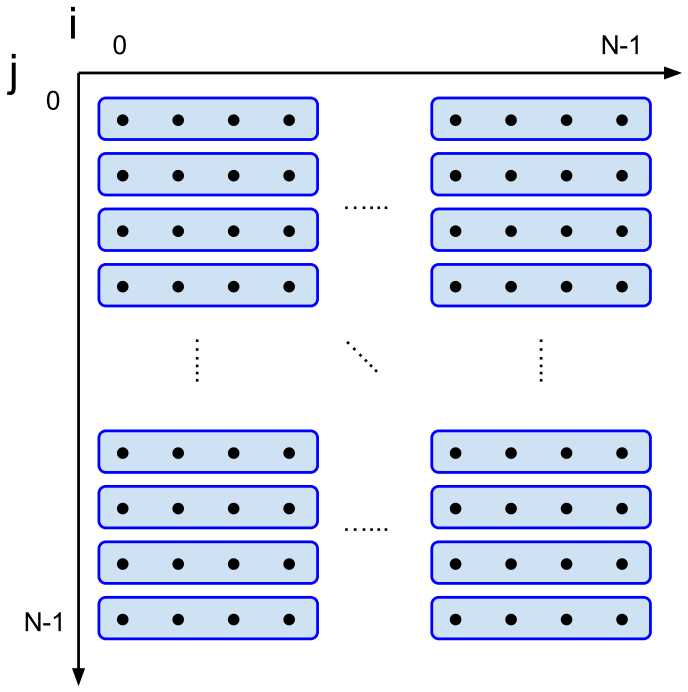

図 5.4. Iteration space (1D SIMDization)

実に $VF \times VF \times VF/VF = 16$ もの組み合わせの演算をしていく．結果的に，5.8 のようなコードになる．図 5.7 の a,b,c はそれぞれ，２つのループインデックスに対応しているので，5.1 節の例のように各ステップで必要な要素が連続に配置されていないので，ギャザー命令のような特殊な命令がない場合，スカラーのロード命令を４つ実行して，SIMD レジスタに挿入していくというような操作が必要になっている．さらに，a,b については前後のステップ間で使いまわせることができないので，結局各組み合わせの演算を行うたびにスカラーロードを実行することになる．c について，４つのステップのたびにスカラーロードが必要となる．このことは，ステップの実行順序を入れ替えれば，a が４つのステップの間は使いまわせたり，b を使いまわせるようになったりする．このような変換はスカラーロードや SIMD レジスタへの挿入などのオーバーヘッドが計算に対して大きいので，性能向上は期待できない．

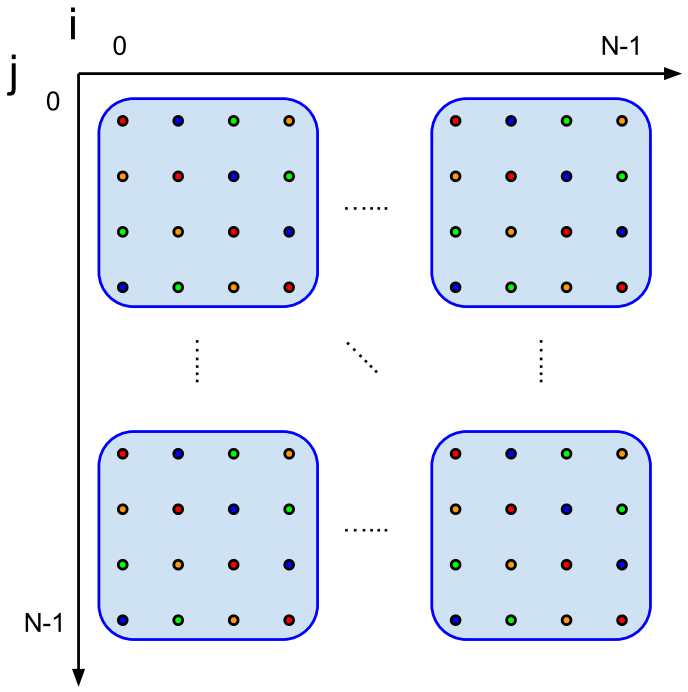

図 5.5. Iteration space (2D SIMDization)

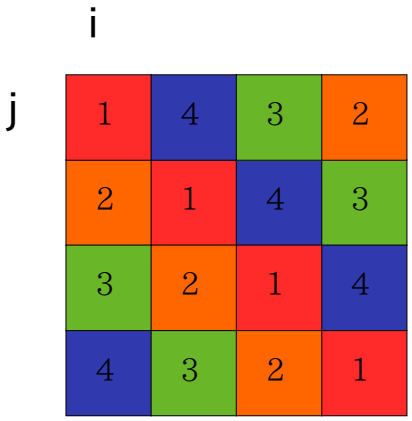

図 5.6. One iteration of 2D SIMDized code ($VF = 4$)

```
1 int i,j;
2 float *a, *b, *c;
3 for(i=0;i<N;i++){
4   for(j=0;j<N;j++){
5     for(k=0;k<N;k++){
6       c[i][j] += a[i][k] * b[k][j];
7     }
8   }
9 }
```

図 5.7. 行列積

```
1  for(i=0;i<N;i+=4){
2   for(j=0;j<N;j+=4){
3    vc = {c[i][j], c[i+1][j+1], c[i+2][j+2], c[i+3][j+3]};
4    for(k=0;k<N;k+=4){
5     va = {a[i][k], a[i+1][k+1], a[i+2][k+2], a[i+3][k+3]};
6     vb = {b[k][j], b[k+1][j+1], b[k+2][j+2], b[k+3][j+3]};
7     vc += va * vb;
8     va = {a[i][k+1], a[i+1][k+2], a[i+2][k+3], a[i+3][k]};
9     vb = {b[k+1][j], b[k+2][j+1], b[k+3][j+2], b[k][j+3]};
10    vc += va * vb;
11    va = {a[i][k+2], a[i+1][k+3], a[i+2][k], a[i+3][k+1]};
12    vb = {b[k+2][j], b[k+3][j+1], b[k][j+2], b[k+1][j+3]};
13    vc += va * vb;
14    va = {a[i][k+3], a[i+1][k], a[i+2][k+1], a[i+3][k+2]};
15    vb = {b[k+3][j], b[k][j+1], b[k+1][j+2], b[k+2][j+3]};
16    vc += va * vb;
17    va = {a[i][k], a[i+1][k+1], a[i+2][k+2], a[i+3][k+3]};
18    vb = {b[k][j+1], b[k+1][j+2], b[k+2][j+3], b[k+3][j]};
19
20    vc = {c[i][j+1], c[i+1][j+2], c[i+2][j+3], c[i+3][j]};
21
22    vc += va * vb;
23    va = {a[i][k+1], a[i+1][k+2], a[i+2][k+3], a[i+3][k]};
24    vb = {b[k+1][j+1], b[k+2][j+2], b[k+3][j+3], b[k][j]};
25    vc += va * vb;
26    va = {a[i][k+2], a[i+1][k+3], a[i+2][k], a[i+3][k+1]};
27    vb = {b[k+2][j+1], b[k+3][j+2], b[k][j+3], b[k+1][j]};
28    vc += va * vb;
29    va = {a[i][k+3], a[i+1][k], a[i+2][k+1], a[i+3][k+2]};
30    vb = {b[k+3][j+1], b[k][j+2], b[k+1][j+3], b[k+2][j]};
31    vc += va * vb;
32    va = {a[i][k], a[i+1][k+1], a[i+2][k+2], a[i+3][k+3]};
33    vb = {b[k][j+2], b[k+1][j+3], b[k+2][j], b[k+3][j+1]};
34
35    vc = {c[i][j+2], c[i+1][j+3], c[i+2][j], c[i+3][j+1]};
36
37    vc += va * vb;
38    va = {a[i][k+1], a[i+1][k+2], a[i+2][k+3], a[i+3][k]};
39    vb = {b[k+1][j+2], b[k+2][j+3], b[k+3][j], b[k][j+1]};
40    vc += va * vb;
41    va = {a[i][k+2], a[i+1][k+3], a[i+2][k], a[i+3][k+1]};
42    vb = {b[k+2][j+2], b[k+3][j+3], b[k][j], b[k+1][j+1]};
43    vc += va * vb;
44    va = {a[i][k+3], a[i+1][k], a[i+2][k+1], a[i+3][k+2]};
45    vb = {b[k+3][j+2], b[k][j+3], b[k+1][j], b[k+2][j+1]};
46    vc += va * vb;
47    va = {a[i][k], a[i+1][k+1], a[i+2][k+2], a[i+3][k+3]};
48    vb = {b[k][j+3], b[k+1][j], b[k+2][j+1], b[k+3][j+2]};
49
50    vc = {c[i][j+3], c[i+1][j], c[i+2][j+1], c[i+3][j+2]};
51
52    vc += va * vb;
53    va = {a[i][k+1], a[i+1][k+2], a[i+2][k+3], a[i+3][k]};
54    vb = {b[k+1][j+3], b[k+2][j], b[k+3][j+1], b[k][j+2]};
55    vc += va * vb;
56    va = {a[i][k+2], a[i+1][k+3], a[i+2][k], a[i+3][k+1]};
57    vb = {b[k+2][j+3], b[k+3][j], b[k][j+1], b[k+1][j+2]};
58    vc += va * vb;
59    va = {a[i][k+3], a[i+1][k], a[i+2][k+1], a[i+3][k+2]};
60    vb = {b[k+3][j+3], b[k][j], b[k+1][j+1], b[k+2][j+2]};
61    vc += va * vb;
62
63   }
64   vc = {c[i][j], c[i+1][j+1], c[i+2][j+2], c[i+3][j+3]};
65   c[i][j] = vc[0];
66   c[i+1][j+1] = vc[1];
67   c[i+2][j+2] = vc[2];
68   c[i+3][j+3] = vc[3];
69  }
70 }
```

図 5.8.　3 次元 SIMD 化ループ (VF=4)

第6章　実装

多次元 SIMD 化の実装を説明する．その前に，LLVM のループの SIMD 化のコードを元にしているので，まず，LLVM のループの SIMD 化の実装を説明する．

6.1　LLVM のループ SIMD 化

LLVM(ver.3.4.2) のループ SIMD 化の流れは以下のようになっている．

1. 最内ループの判定.
2. 変換の正当性検査.
3. コスト見積りにより，SIMD 幅，アンロール幅の決定．LLVM のループ SIMD 化器はループアンローリングも行う.
4. CFG の変換.
5. スカラー命令をベクトル命令に変換する.
6. コード変換に伴う各解析モジュールの情報アップデート.

```
1 void vectorize(LoopVectorizationLegality *Legal) {
2   // CFG の変更
3   createEmptyLoop(Legal);
4   // 元のループをスキャンしながらベクトル命令生成
5   // Legal はインダクション変数，リダクション変数の情報を持つ
6   vectorizeLoop(Legal);
7   // コード変更に伴う解析情報更新
8   updateAnalysis();
9 }
```

図 6.1. LLVM のループ SIMD 化

LLVM のループ SIMD 化は最内ループのみに対して行われるので，まず，最内ループの検出が行われる．そして，正当性，有益性の検証により VF 及びアンロール幅 (UF) の決定がされる．4 のステップでは元のループを図 6.2 のように変形する．LLVM-IR は静的単一代入形式 (Static Single Assignment, SSA) で記述されており，コードは基本ブロック単位で記述される．基本ブロックとは制御が分岐がない最大の長さのコードブロックのことで，LLVM-IR から容易に制御フローグラフ (Control Flow Graph, CFG) を記述できるし，LLVM-IR を記述するときには CFG を意識しなければならない．図 6.2 の黄色ブロックは元のコードから存在するブロックで，青いブロックは新たに作られたブロックである．また，赤い矢印は新規に作られた，あるいは，変更

された制御の流れである．このステップでは，基本ブロックの作成が主であるが，新たに作られた SIMD ループのインダクション変数を更新するコードやブランチ命令は挿入される．

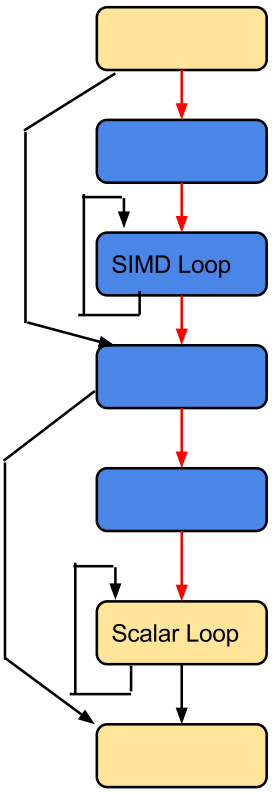

図 6.2. LLVM により SIMD 化された後の制御フローグラフ

6.1.1　ベクトル命令生成

CFG の変更の後，対象のループ内の命令を上からスキャンしながら，対応するベクトル命令を挿入していく．その際，スカラー命令からベクトル命令に対応させる **WidenMap** という連想配列に対して，参照と更新しつつ行う．この WidenMap のおかげで，あるスカラー命令をベクトル化する時，スカラー命令の引数の値に対応するベクトル値は WidenMap を参照することで得られる．5 のとき，ほとんどの命令は単純に型を変換させるような簡単な変換で済むのだが，特別な扱いが必要な場合がある．特にインダクション変数，リダクション変数とメモリアクセス命令がそれに該当する．

インダクション変数のベクトル化

LLVM はインダクション変数が 1 ずつインクリメントする整数及びポインタを対象に SIMD 化を行う．つまり，インダクション変数に遭遇したとき，インダクションの初期値から始まる連続す

る値を含むベクトルをつくればよい．また，インダクション変数に依存しない変数，つまり，ループ不変な変数は同じ値を持つベクトルをつくる．

リダクション変数のベクトル化

リダクション変数が例えば，加算命令によるものなら，[リダクション変数の初期値, 0, ,,,, 0] となるようなベクトルを作り，ベクトル化されたループを抜けた時，ベクトル内の値を足しあわせた変数を作る．その際，ループの外でリダクション変数を参照する命令がある場合は適切に参照先を修正する．

メモリアクセス命令のベクトル化

アラインメントが合っていなかったり，ポインタがループ不変だったり，不連続なアクセスをする場合，１つずつアクセスして，結果を１つのベクトルに集めるというようなことをする命令を生成する (このような変換を **Scalarize** と呼ぶこととする)．そうでない時は，ベクトル命令によるメモリアクセスを行うコードを生成する．

6.2　多次元 SIMD 化

従来の SIMD 化がイテレーションのインスタンスを最内ループに沿って複数まとめて実行するが，多次元 SIMD 化はイテレーションのインスタンスを多次元的に捉えて処理する．その際，どのような組み合わせをどのような順番でやるかの選択肢が多いが，一定のパターンについて実装した．多次元 SIMD 化の前段階として，Polly というモジュールにループタイリングを行い，そのタイルにたいして多次元 SIMD 化するという方法をとった．

6.2.1　Polly

4 章で説明した通り，Polly は LLVM-IR にたいして Polyhedral Model を利用してタイリングと並列化を行う．Polly によるループタイリングの例を C 言語レベルで示すと図 6.3 のようになる．なお，実際は LLVM-IR 上で同様の変換がされている．

多次元 SIMD 化にあたって，内側のタイルを完全に潰したかったので，タイルの一辺が VF になるようにタイリングを行う必要があった．そのため，コンパイル時にオプションでタイルサイズを指定できるように Polly を拡張した．図 6.3 では $VF = 8$ として，$VF \times VF$ のタイリングが行われているが，内側のタイルの上限値が min() で決まるので，タイルの形は常に $VF \times VF$ の形をしているわけではない．つまり，直接タイル内のコードをそのまま変換することができない．

```
1  // Before transformation
2  for(i = 0; i < N; i++){
3    for(j = 0; j < M; j++){
4      f[i] += x[j] - x[i];
5    }
6  }
7
8  // Trasformed code by Polly
9  if ((M >= 1) && (N >= 1)) {
10   for (c1=0;c1<=N-1;c1+=8) {
11     for (c2=0;c2<=M-1;c2+=8) {
12       for (c3=c1;c3<=min(N-1,c1+7);c3++) {
13         for (c4=c2;c4<=min(M-1,c2+7);c4++) {
14           Stmt_for_body6(c3,c4);
15         }
16       }
17     }
18   }
19 }
```

図 6.3. Polly によるタイリング

6.2.2　CFG

図 6.3 のように変換されたコードに対して多次元 SIMD 化を行う．まず，基本ブロックを追加し CFG を変更する．N 次元 SIMD 化されたコードは対象のループの回る回数がそれぞれ VF であるときに実行されることとして実装を行った．結果的に N 次元に SIMD 化されたとき，CFG は Figure.6.4 のようになる，１つの箱が１つの基本ブロックを表し，点線の矢印では基本ブロックが省略されている．青い基本ブロックは多次元 SIMD 化により挿入された基本ブロックである．Test Rectangle と書かれたブロックで対象の N 次元ループが各次元についてループ回数が VF と一致するか調べる．イテレーション空間で考えると，対象ループが一辺 VF の超直方体であるかどうか調べている．超直方体であれば，N 次元 SIMD 化されたコードに分岐する．

6.2.3　イテレーションのインスタンスの実行順序

多次元 SIMD 化のタイル内の実行順序は多くの選択肢があるが，本研究の実装では図 6.5 のように SIMD レジスタに対応するイテレーションのデータを格納しながら実行を行う．i, j, k はループネストの各ループを示し，i の方が内側である．最内のイテレーションを循環シフトさせて一周したら，１つ外側を１つだけ循環シフトさせる．このように，内側のイテレーション変数が一周したら１つだけシフトするといようなことを繰り返す．なお，一番外側はシフトすることはない．これは，5.2 節で示した例の実行順序と同じような実行順序である．ここで，説明のため，N 次元 SIMD 化するときの内側 $N-1$ 次元のループを **ShiftingLoop** と呼ぶこととする．また，各ステップにおいてシフトさせるループを **CurrentShiftingLoop** と呼ぶ．さらに，対象の N 次元ループを **OriginalLoop** と呼ぶこととする．

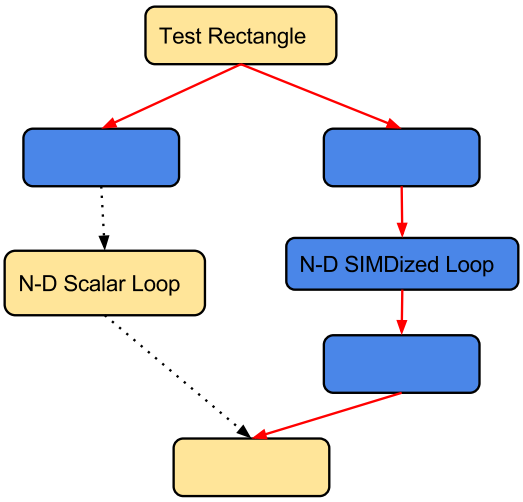

図 6.4. N 次元 SIMD 化されたコードの制御フローグラフ

6.2.4　ベクトル命令生成

LLVM と同様，元のスカラー命令をスキャンしながらベクトル命令を新たに作った基本ブロックに挿入していく．N 次元 SIMD 化は元のループの最内ループ内の演算を SIMD レジスタの要素を入れ替えながら，$\frac{VF^N}{VF} = VF^{N-1}$ ステップ行う．つまり，スカラー命令をスキャンしながらベクトル命令を挿入していく操作を，VF^{N-1} 回繰り返す．その際，図 6.5 の順番通りに，計算に必要な SIMD レジスタの要素を入れ替えながら行う．そのためには，インダクション変数，リダクション変数，メモリアクセス命令のベクトル化を工夫する．LLVM の元のループ SIMD 化モジュールも同様であるが，最終的に生成されるコードは非常に冗長なものになっている．これら冗長性の排除は他の最適化パスに頼っている．

インダクション変数のベクトル化

インダクション変数は 1 回目のスキャンの時は 1 次元の時と同様，連続な値を持つベクトルを作る．2 回目以降のスキャンのときは，CurrentShiftingLoop に依存しているとき，前回のベクトル化の結果を循環シフトさせるコードを生成する．そうでない場合は，1 回目のものを使う．

リダクション変数のベクトル化

1 回目のスキャンの時，リダクション変数の初期値が OriginalLoop に依存しないとき，1 次元の時と同じようなベクトルを作るが，そうでない時は，すでにベクトル化された命令が WidenMap にあるはずなので，それを用いる．2 回目以降はリダクション変数の初期値が CurrentShiftingLoop に依存する時は，循環シフトさせる．そうでない時は，リダクション演算のベクトル化の結果を使用する．

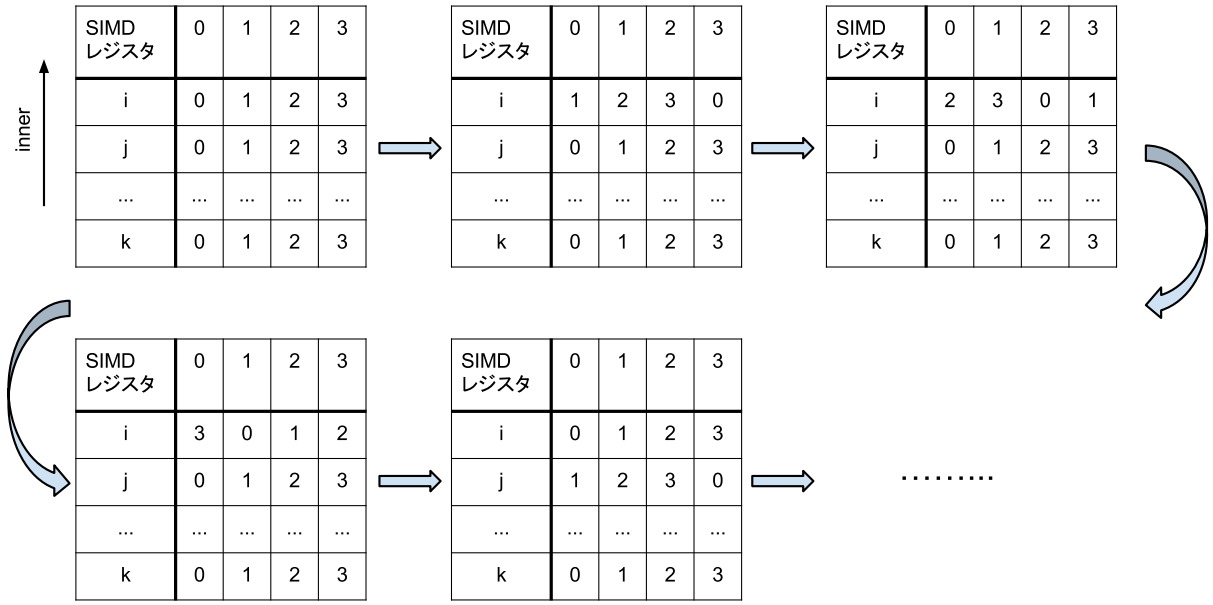

図 6.5. タイル内での計算実行順序

メモリアクセスのベクトル化

多次元 SIMD 化の際には，メモリアクセスのポインタが OriginalLoop 中の複数のループについて依存するとき，連続なアクセスではないとして，Scalarize する．１つにループについて依存するようなときも，不連続アクセスになるなら，Scalarize する．以下，連続アクセス出来る場合について，ロード・ストア命令それぞれについて説明する．

ロード　１回目はベクトルロードを生成する．２回目は CurrentShiftingLoop について依存するなら，１回目にロードしたものを循環シフトさせる．そうでない場合は，そのまま使用する．

ストア　ストアは特にその命令の結果の値というものもなく，連続なメモリにストアできるとき，ポインタがループに依存していても，結果的に同じ場所に保存する．ただベクトル化する．

```
1  void vectorizeLoopRec(LegalityMap* LegMap, unsigned ShiftingLevel){
2    if (ShiftingLevel == 1){
3      for (unsigned i=0; i < VF; i++) {
4        CurrentShiftingLoops.clear();
5        CurrentShiftingLoops.insert(CurrentShiftingLoops.begin(),
6                  ShiftingLoops.end()−ShiftingLevel, ShiftingLoops.end());
7        vectorizeLoop(LegMap);
8      }
9    }else{
10     for (unsigned i=0; i < VF; i++) {
11       CurrentShiftingLoops.clear();
12       CurrentShiftingLoops.insert(CurrentShiftingLoops.begin(),
13       ShiftingLoops.end()−ShiftingLevel, ShiftingLoops.end());
14       vectorizeLoopRec(LegMap, ShiftingLevel−1);
15     }
16   }
17 }
18
19 void vectorize(LegalityMap* LegMap) {
20   createEmptyLoop(LegMap);
21   // CurrentShiftingLoops を適切に設定しつつ，vectorizeLoop でベクトル命令
22    を生成する.
23   vectorizeLoopRec(LegMap, VectorizationDimention);
24   updateAnalysis();
25 }
```

図 6.6. N 次元 SIMD 化

第7章　評価

7.1　評価環境

実装はLLVM 3.4.2のLoopVectorizer.cppをベースに新しいPassとして，C++で実装した．また，多次元SIMD化のパスはPolly 3.4.2によりタイリングされたループに対して実行するように設計されている．実行はSandy Bridge世代のIntel(R) Core(TM) i7-2600 3.40GHz 4コア 8スレッド，SIMD幅256bitのAVX命令セットのCPUとメモリ8GBのマシンで行った．実験結果は全て１スレッドで実行した結果である．このCPUの理論FLOPSは54.4GFLOPSである．実験では２重ループ内で加算と乗算 (MulAdd) を行う計算と，N体シミュレーションと行列積について２次元SIMD化を行った．

7.2　実験

各種アプリケーションの具体的なコードをそれぞれ図7.1,7.3,7.5に示した．MulAddとN体については以下の変換を比較した．

- SIMD化なし
- １次元のSIMD化．LLVMの本来のSIMD化．
- Pollyによるタイリングのみ
- ２次元のSIMD化．
- 手動による２次元SIMD化

結果は図7.2，7.4に示した．MulAddは１次元のSIMD化が最も良い結果となった．このMulAddのSIMD化はアンロール幅4のループアンローリングが最内ループに行われているためであると，考えられる．２次元SIMD化されたコードは２次元SIMD化されたコードを実行するための比較命令や条件分岐命令，SIMDレジスタの要素を入れ替える命令など，他にはない命令が増えている．これらは命令レベル並列性や分岐予測などで問題にならないが，MulAddの計算は加算・乗算が一回ずつしかなく，SIMDレジスタの要素を入れ替える命令がオーバーヘッドとなっている可能性がある．一方，タイリングのものは１次元SIMD化とアンロール幅4のループアンローリングも行われているのに，１次元SIMD化に負けている．つまり，この例では，タイリングが性能を低下させる要因となっている．１次元のSIMD化のものは連続アクセスの恩恵を受けていると考えられる．２次元SIMD化が最も速いわけではなかったが，Pollyに勝ることはできた．
N体シミュレーションも最も速くなることはできなかったが，Pollyに勝ることが確認できた．２次元SIMD化が１次元SIMD化に負ける要因として，レジスタが溢れてしまったことが原因であると考えられる．１つのSIMDレジスタの要素を入れ替える際，現状のLLVMの実装及び，AVX

命令セットでは 3 つもレジスタを必要とするようなコードが生成されていた．将来の命令セットの拡張で SIMD レジスタの要素の循環シフトが 1 命令 1 レジスタで行えるようになれば性能向上が期待できる．
行列積の結果は図 7.6 の通りである．LLVM のループ SIMD 化モジュールはコストを見積もった結果，図 7.5 を SIMD 化しなかった．そこで，オプションで強制的に 1 次元 SIMD 化を行った．図 7.5 のコードを SIMD 化すると Scalarize のコストが大きく，スカラーコードにも及ばなかった．しかし，2 次元 SIMD 化されたコードだと b について Scalarize のコストが生じるものの a, c の要素は SIMD ロードで取得することができ，そのまま SIMD レジスタを使いまわすことができる．この例は，上手く多次元 SIMD 化が働いた例だと言える．しかし，3 次元 SIMD 化の場合は Scalarize のコストが性能に響くと考えられる．

```
1  for(i = 0; i < N; i++){
2    for(j = 0; j < M; j++){
3      f[i] += x[j] * x[i];
4    }
5  }
```

図 7.1. Multiply and Add

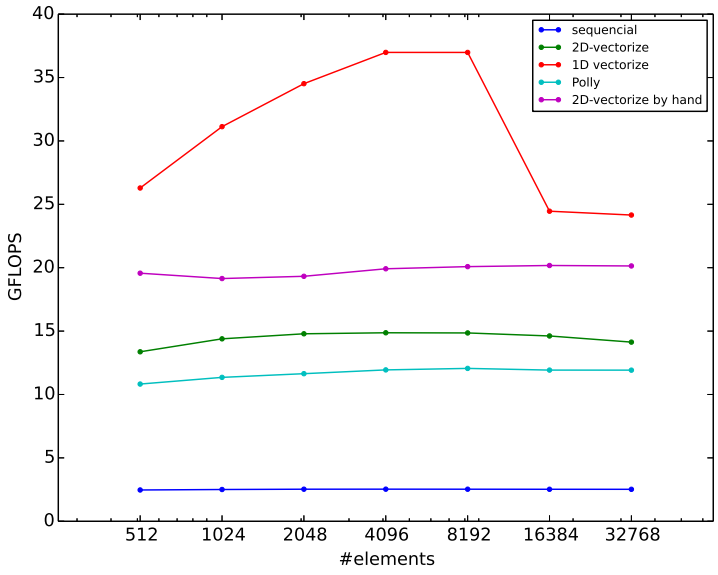

図 7.2. 性能比較 (MulAdd)

```
1   for(i = 0; i < N; i++){
2     for(j = 0; j < M; j++){
3       float Mg = m[i] * G;
4       float dx = x[j] - x[i];
5       float dy = y[j] - y[i];
6       float dz = z[j] - z[i];
7       float dr2 = dx*dx + dy*dy + dz*dz + FLT_EPSILON;
8       float dr = sqrtf(dr2);
9       float inv_dr3 = 1.f / (dr2 * dr);
10      float s = m[j] * Mg * inv_dr3;
11      fx[i] += dx*s;
12      fy[i] += dy*s;
13      fz[i] += dz*s;
14    }
15  }
```

図 7.3. N 体シミュレーション

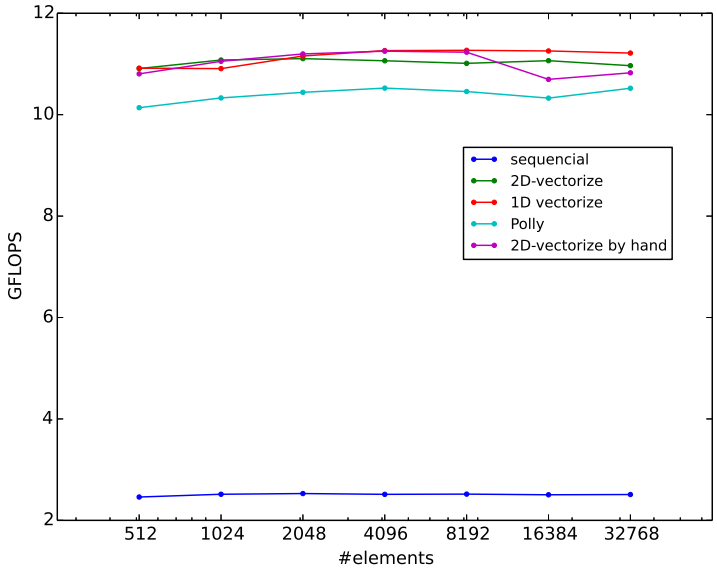

図 7.4. 性能比較 (N 体)

```
1  for (i=0; i<M; i++){
2    for (j=0; j<N; j++){
3      for (k=0; k<K; k++){
4        c[i*N+j] += a[i*K+k] * b[k*N+j];
5      }
6    }
7  }
```

図 7.5. 行列積

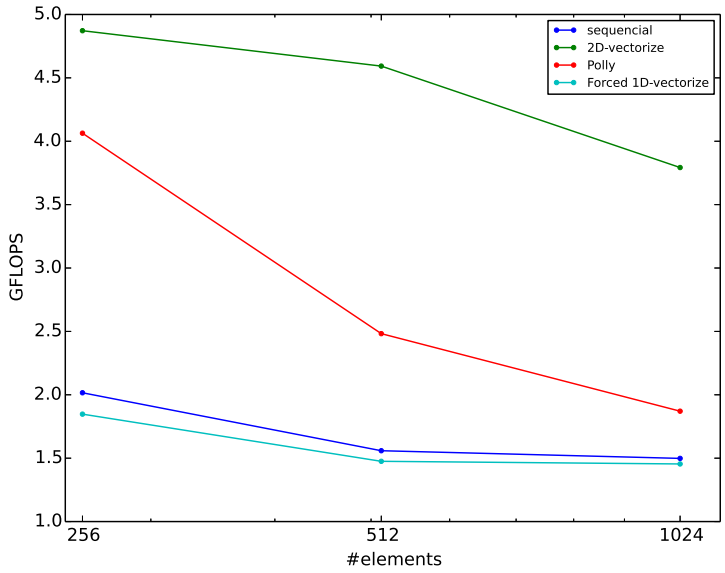

図 7.6. 性能比較 (行列積)

第8章　結論

8.1　まとめ

ループネストに対して性能向上のため，様々なループ変換が存在する中，新しいループ変換としてループネストの多次元 SIMD 化を提案した．ループのタイリングによるデータの再利用性と使用するレジスタ削減によるメモリアクセスの削減を狙える変換であった．CPU の SIMD レジスタの大きさが増大することで，理論性能が向上しているトレンドの中で，SIMD 命令を活用するループ変換はさらに重要度が増してくるはずである．実験では，MulAdd,N 体シミュレーションについては Polly に勝ることができ，行列積については LLVM の最適化以上の性能向上を確かめることが出来た．

8.2　今後の課題

実装が完全ではなく，リダクション変数の取り扱いには特に課題が残っている．例えば，実験のアプリケーションのようなメモリ領域に対するリダクションではなく，単一のスカラーの変数に対して足しあわせていくようなリダクションがあるとき，正しく変換することができない．
また，変換の際に適切な SIMD 幅，アンロール幅，SIMD 化の次元数を決定するようなコスト見積りが必要である．多次元 SIMD 化はその性質上，次元数が大きくなるとコードサイズが $O(VF^{N-1})$ で増えていったり，メモリの参照するインデックスによっては SIMD レジスタの要素の取り出し，挿入のコードの増加分などもあり，性能向上を狙えるようなアプリケーションは多くない．闇雲に多次元 SIMD 化を行うわけでなく，適切にコストを見積り，性能向上が見込まれる変換を選択する必要がある．多次元 SIMD 化が有効なコードパターンを知るためにも，もっと広範なアプリケーションで実験する必要もある．さらに，AVX 以外の命令セットでの実験も必要である．
また，多次元 SIMD 化の正当性検査の実装も必要である．さらに，適用できるコードの範囲が狭いことも問題である．この実装だと，始めに Polly による変換を行うので，Polly の適用できる範囲に制限され，さらに，多次元 SIMD 化の正当性チェックによりさらに制限される．この問題に対して，Polly を経由せずに多次元 SIMD 化する方法も考えられる．

謝辞

修士生活について謝辞を述べるにあたって，何よりもまず，田浦 健次朗先生に感謝を申し上げます．数々の丁寧なご指導ありがとうございました．私は決して真面目で優秀な学生というわけではございませんでしたので，叱責を受けることもありましたが，気にかけてくださったこと感謝しております．秘書の高野 弘美さんは先生に叱られて落ち込んでる時に優しい言葉をかけてくださったり，仕事の仕方のアドバイスなど，大変感謝しております．
M1 の頃，OB の中谷 翔さんは机が隣ということもあり，気の合う話し相手になってくれて，大変楽しく過ごさせていただきました．林 伸也さんも良き先輩として様々なアドバイスを頂きました．当時，卒論生だった小澤 真里奈さんは研究分野が近いということもあり，話す機会がよくありました．彼女のおっとりとした性格に癒されたものです．
この他にもお世話になった数々の方々に，感謝申し上げます．奈良の家族にも感謝しております．家族のおかげで，大学院まで通うことが出来ました．いつか親孝行させてください．

発表文献

国内発表（査読なし）

- 藤田晃史，早水光，中島潤（東京大），塩谷亮太（名古屋大）．スーパースカラ・プロセッサ「雷上動」の設計と実装．**第 *200* 回計算機アーキテクチャ研究発表会**，東京，2014/1.

受賞

- 第 1 回高性能プロセッサ設計コンテスト プロフェッショナル部門 優勝. 2014/1.
- 情報処理学会 CS 領域奨励賞　2014/1.

参考文献

[1] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[2] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08*, page 101, 2008.

[3] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation - PLDI '04*, page 82, 2004.

[4] Martin Griebl. *Automatic parallelization of loop programs for distributed memory architectures*. 2004.

[5] Tobias Grosser, H Zheng, and R Aloor. Polly-Polyhedral optimization in LLVM. *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, pages 1–6, 2011.

[6] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. When polyhedral transformations meet SIMD code generation. *ACM SIGPLAN Notices*, 48(6):127, June 2013.

[7] Christian Lengauer. Loop parallelization in the polytope model. *CONCUR'93*, 1993.

[8] D. Nuzman and R. Henderson. Multi-platform Auto-vectorization. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 281–294. IEEE, March 2006.

[9] Dorit Nuzman and A Zaks. Outer-loop vectorization: revisited for short simd architectures. *... international conference on Parallel architectures ...*, pages 2–11, 2008.

[10] LN Pouchet. *Iterative optimization in the polyhedral model*. PhD thesis, UNIVERSITÉ DE PARIS-SUD 11, 2010.

[11] Konrad Trifunovic and Dorit Nuzman. Polyhedral-model guided loop-nest auto-vectorization. *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[12] MJ Wolfe, C Shanklin, and L Ortega. High performance compilers for parallel computing. 1995.

[13] 小澤真里奈 and 田浦健次朗. N 体シミュレーションの効率的な SIMD 化, 東京大学卒業論文. 2014.