

Cell Broadband Engine を対象とした、コンパイ
ラの自動SIMD化の実装

東京工業大学
大学院 情報理工学研究科
数理・計算科学専攻

岩橋 弥生
(08M37083)

平成22年 修士論文

指導教官 佐々 政孝 教授

平成22年2月18日

概要

SIMD (Single Instruction Multiple Data stream) 命令では、1 命令で同一演算、同一型の複数の命令を並列に実行できる。代表的な SIMD 命令としては、Intel の x86 アーキテクチャ用の SSE3、IBM の PowerPC アーキテクチャ向けの AltiVec、そして SONY、IBM、TOSHIBA の共同開発によって作られた、Cell Broadband Engine に搭載されている SPU プロセッサ向けの、SPU SIMD 命令などがあげられる。同一命令を並列実行する SIMD 命令は、大量の同一演算の命令を処理する必要があるマルチメディア処理に強い。よって、ゲームや物理現象のシミュレーションなどで高度な画像処理技術が求められている中、SIMD 命令を有効利用することが重要である。プログラムを SIMD 化する方法として現在用いられているのが、プログラムによるアセンブリ言語や SIMD 命令と 1 対 1 に対応する関数 (intrinsic ルーチン) を使って SIMD 並列化を行うことである。しかし、プログラムによる SIMD 命令の生成は特別な知識が必要になったり、プログラムがアーキテクチャ依存になってしまうという問題がある。よって、SIMD 命令を使用していないソースコードに対してコンパイラによって SIMD 命令を自動生成する技術を確立することが重要である。一方、フリーコンパイラで最も多く使われている GCC の自動 SIMD 化は、他の商用のコンパイラに比べ、自動 SIMD 化の性能が劣っている。GCC の自動 SIMD 化の精度を上げることは、SIMD 命令を使用したい多くのプログラマを助けることになると考えられる。そこで本研究では、Cell Broadband Engine の SPU プロセッサを対象に、SPU プロセッサ向けの GCC である、spu-gcc の自動 SIMD 化の補助機能を、コンパイラインフラストラクチャ COINS 上で実装した。テストプログラムとして、メディア処理のベンチマークである MediaBenchII を使用して評価を行った。その結果、MediaBenchII の中で、SIMD 化可能であるにも関わらず、spu-gcc で SIMD 化できなかったループの内、約 7.5% のループ文が新たに SIMD 化出来るようになり、そのループ文を含むメソッドで最大 60% 以下の実行時間となった。スカラ変換は、SIMD ベンチマークで性能を測ったところ、SIMD 化により最大で 10% の実行時間となった。

目次

第 1 章	はじめに	6
1.1	背景	6
1.2	本研究の概要	6
1.3	構成	7
第 2 章	準備	8
2.1	基本ブロック	8
2.2	制御フローグラフ	8
2.3	支配関係	9
2.4	支配木	12
2.5	制御依存	12
2.6	抽象構文木	13
2.7	シフト演算	15
2.7.1	論理シフト演算	15
2.7.2	算術シフト演算	15
第 3 章	Single Instruction / Multiple Data (SIMD)	18
3.1	Single Instruction / Multiple Data	18
3.1.1	SIMD プログラミング例	18
3.2	コンパイラによる自動 SIMD 化	19
3.2.1	データのアラインメント	20
3.2.2	scatter / gather 命令	23
3.2.3	データ型の混在	24
3.2.4	SIMD 命令セットの偏り	24
第 4 章	Cell Broadband Engine の説明	25
4.1	Cell Broadband Engine	25
4.2	PowerPC Processor Element	25
4.3	Synergistic Processor Element	25
4.4	プログラミング手法	26
第 5 章	GCC の自動 SIMD 化とその問題点	27
5.1	概要	27
5.2	GCC の自動 SIMD 化の構造	28
5.2.1	解析部	28

5.2.2	再構築部	29
第 6 章	並列化コンパイラ向け共通インフラストラクチャCOINS	30
6.1	概要	30
6.2	構成	30
6.3	COINS の特徴	31
6.4	高水準中間言語 HIR について	32
6.4.1	概要	32
6.4.2	具体表現	32
第 7 章	提案する SIMD 並列化向けループ変換	34
7.1	if 変換	34
7.2	ループ分割	38
7.3	スカラ拡張	39
第 8 章	実験と考察	41
8.1	実装	41
8.2	実験と考察	41
8.2.1	準備実験	41
8.2.2	実験	42
8.3	考察	43
8.3.1	if 変換	43
8.3.2	ループ分割	45
8.3.3	スカラ拡張	46
8.4	今後の課題	47
8.4.1	ループ分割の適用範囲	47
8.4.2	SIMD 化できない他の要因の解決	47
8.4.3	ループのベクタ化	47
第 9 章	関連研究	48
9.1	鈴木らの研究	48
9.2	Pande らの研究	48
9.2.1	Glimpses	48
9.2.2	AutoPort	49
第 10 章	まとめ	51

目次

2.1	基本ブロック	9
2.2	制御フローグラフ (CFG)	10
2.3	支配木	12
2.4	制御依存	13
2.5	式 2.1 に対する解析木	14
2.6	式 2.1 に対する抽象構文木	15
2.7	論理シフト演算：左シフト	16
2.8	論理シフト演算：右シフト	16
2.9	算術シフト演算：左シフト	17
2.10	算術シフト演算：右シフト	17
3.1	SIMD 化前	18
3.2	SIMD 化後	19
3.3	間違ったロードの仕方	21
3.4	正しいロードの仕方 1	22
3.5	正しいロードの仕方:2	23
4.1	Cell.B.E. の構造	26
5.1	spu-gcc の自動 SIMD 化機能デバッグ	28
6.1	COINS 構成図	31
7.1	ソースコードレベルでの if 変換器	37
7.2	ループ分割	39
8.1	spu-gcc の自動 SIMD 化 デバッグ情報	42
8.2	時間測定結果：if 変換	44
8.3	時間測定結果：スカラ拡張	46
9.1	Glimpses の構造	49
9.2	Glimpses のデモ	49
9.3	AutoPort の位置付け	50

表 目 次

2.1 簡単な文法の例	14
8.1 実験環境	43

第1章 はじめに

1.1 背景

SIMD (Single Instruction Multiple Data stream) 命令では、1 命令で同一演算、同一型の複数の命令を並列に実行できる。図の例だと足し算のスカラ命令 4 つを、複数のスカラデータを保存できる SIMD レジスタを使用して SIMD 命令 1 命令で実行している。代表的な SIMD 命令としては、Intel の x86 アーキテクチャ用の SSE3、IBM の PowerPC アーキテクチャ向けの AltiVec、そして SONY、IBM、TOSHIBA の共同開発によって作られた、Cell Broadband Engine に搭載されている SPU プロセッサ向けの、SPU SIMD 命令などがあげられる。

同一命令を並列実行する SIMD 命令は、大量の同一演算の命令を処理する必要があるマルチメディア処理に強い。よって、ゲームや物理現象のシミュレーションなどで高度な処理が求められている中、有効利用することが重要である。

プログラムを SIMD 化する方法として現在とられているのが、プログラマによるアセンブリ言語や SIMD 命令と 1 対 1 に対応する関数 (intrinsic ルーチン) を使って SIMD 並列化を行うことである。しかし、プログラマによる SIMD 命令の生成は特別な知識が必要になったり、プログラムがアーキテクチャ依存になってしまったりという問題がある。よって、通常のソースコードに対してコンパイラによって SIMD 命令を自動生成する技術を確立することが重要である。一方、フリーコンパイラである GCC の自動 SIMD 化は、他の商用のコンパイラに比べ、自動 SIMD 化の性能が劣っている。フリーのコンパイラで最も多く使われている GCC の自動 SIMD 化の精度を上げることは、SIMD 命令を使用したい多くのプログラマを助けることになる。

1.2 本研究の概要

本研究では、まず、Cell Broadband Engine の SPU プロセッサのコンパイラである spu-gcc の自動 SIMD 化機能の性能を調べ、SIMD 化できなかった原因を分析した。それに対し、コンパイラのフロントエンドで実装できる「if 変換」「ループ分割」そして「スカラ拡張」の手法を適用し、その効果を検証した。

1.3 構成

本論文の構成を以下に示す.

- 2章: 準備 (本論文で用いる用語の説明)
- 3章: Cell Broadband Engine の説明
- 4章: Single Instruction / Multiple Data (SIMD) とコンパイラの自動 SIMD 化の現状
- 5章: GCC の自動 SIMD 化とその問題点
- 6章: 並列化コンパイラ向け共通インフラストラクチャCOINS の説明
- 7章: 提案する SIMD 並列化向けループ変換
- 8章: 実験と考察
- 9章: 関連研究
- 10章: まとめ
- 参考文献

第2章 準備

本論文の説明にあたって予備知識が必要となる。本章ではフローグラフ等の予備知識の説明を行う。

2.1 基本ブロック

基本ブロック (Basic Block) あるいはブロックとはプログラムの一部分であり、そのブロックの実行を開始する文が、そのブロックの先頭の文だけであり、末尾が無条件飛び越し、あるいは条件つき飛び越しであるものである。基本ブロック途中への飛び込みや、基本ブロック途中からの飛び出しはない。基本ブロックと、そのブロック間の関係を示した制御フローグラフ (後述) は、プログラムを解析する基本的なグラフとして現在広く用いられている [1, 12, 3, 15]。図 2.1 の (a) のような C プログラムを基本ブロックに分割したものが (b) である。図中の矩形は基本ブロックを表しており、矩形の左上の L1 等は基本ブロックにつけた便宜的なラベルである。

2.2 制御フローグラフ

制御フローグラフ (Control Flow Graph : CFG) とは、プログラムの入り口から出口までの基本ブロックを、制御の流れに沿って図にしたグラフである [1, 12, 3]。CFG の節点 (ノード) は基本ブロックであり、ノード間は有向辺 (エッジ) で結ばれる。コンパイラでの最適化の多くは CFG の情報を用いている。

図 2.2 に、図 2.1 に対応する CFG を示す。図中の矩形は基本ブロックであり矢印はエッジを表している。また、矩形の中にある L1 等は基本ブロックに対して付けた便宜的なラベルである。各ノードのラベルは、図のそれと対応している。なお、定式化によっては、これ以外に入り口ノードと出口ノードを別途付加することもあるが、ここでは立ち入らない。

基本ブロック X と Y が存在し、X から Y への有向辺がある場合、X は Y の先行ノード (predecessor) といい、Y は X の後続ノード (successor) と言う。以下では X の predecessor を $pred(X)$ とし、X の successor を $succ(X)$ とする。たとえば図 2.2 の各ノードの $pred()$ と $succ()$ は以下ようになる。

- L1 : $pred(L1) = \{\}, succ(L1) = \{L2, L6\}$
- L2 : $pred(L2) = \{L1\}, succ(L2) = \{L3, L4\}$

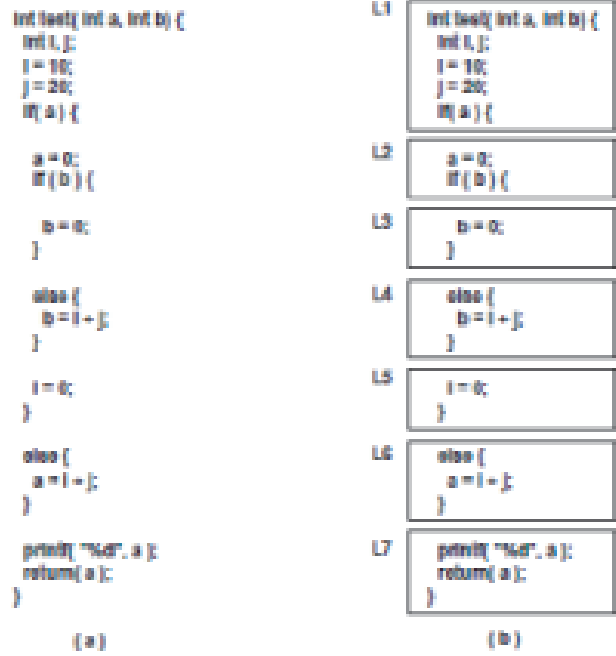


図 2.1: 基本ブロック

- $L3 : pred(L3) = \{L2\}, succ(L3) = \{L5\}$
- $L4 : pred(L4) = \{L2\}, succ(L4) = \{L5\}$
- $L5 : pred(L5) = \{L3, L4\}, succ(L5) = \{L7\}$
- $L6 : pred(L6) = \{L1\}, succ(L6) = \{L7\}$
- $L7 : pred(L7) = \{L5, L6\}, succ(L7) = \{\}$

2.3 支配関係

CFG 上のふたつのノード X, Y について, CFG の入り口から Y に達するどの路も必ず X を通る場合に, X は Y を支配 (dominate) するという [1, 12, 3]. また, X は Y の支配ノード (dominator) であるという. 以下では X の支配ノードの集合を $dom(X)$ と書く. 支配関係は反射的 (reflexive) である. すなわち, ノード X は自分自身を支配する. また, 支配関係は推移的 (transitive) である. すなわち, ノード X がノード Y を支配し, ノード Y がノード Z を支配するのであれば, ノード X はノード Z を支配する [12]. 例えば図 2.2 の CFG では以下のような支配関係がある.

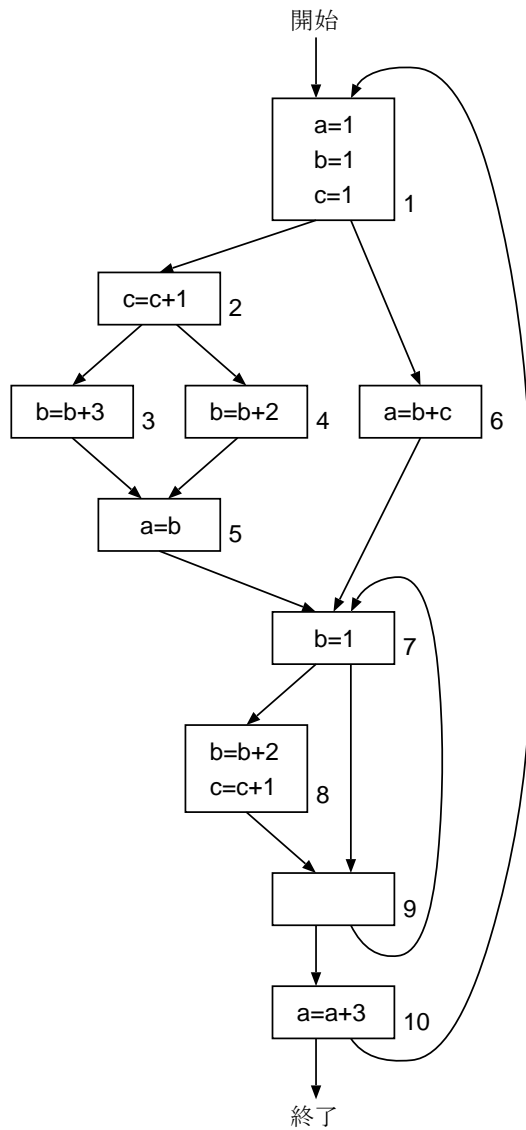


図 2.2: 制御フローグラフ (CFG)

$$dom(L1) = \{L1\}$$

$$dom(L2) = \{L1, L2\}$$

$$dom(L3) = \{L1, L2, L3\}$$

$$dom(L4) = \{L1, L2, L4\}$$

$$dom(L5) = \{L1, L2, L5\}$$

$$dom(L6) = \{L1, L6\}$$

$$dom(L7) = \{L1, L7\}$$

ノード X が ノード Y を支配し, $X \neq Y$ である場合, X は Y を厳密に支配 (strictly

dominate) するという [1, 12, 3]. 例えば図 2.2 の L1 は L3 を厳密に支配している. 以下, X を厳密に支配するノードの集合を $sdom(X)$ と書く.

ノード X が ノード Y を厳密に支配し, X から Y への路に, X 以外に Y を厳密に支配するノードがないとき, X は Y を直接支配 (immediately dominate) するという [1, 12, 3]. 以下では X が Y を直接支配するときに $X = idom(Y)$ と書く. 例えば 図 2.2 の CFG では以下のような直接支配の関係がある.

$$\begin{aligned} idom(L1) &= \{\} \\ idom(L2) &= \{L1\} \\ idom(L3) &= \{L2\} \\ idom(L4) &= \{L2\} \\ idom(L5) &= \{L2\} \\ idom(L6) &= \{L1\} \\ idom(L7) &= \{L1\} \end{aligned}$$

2つのノード X, Y について, Y からプログラムの出口に達するどの路も必ず X を通るとき X は Y を後支配 (postdominate) するという [1, 12, 3].

$$\begin{aligned} pdom(L1) &= \{L1, L7\} \\ pdom(L2) &= \{L2, L5, L7\} \\ pdom(L3) &= \{L3, L5, L7\} \\ pdom(L4) &= \{L4, L5, L7\} \\ pdom(L5) &= \{L5, L7\} \\ pdom(L6) &= \{L6, L7\} \\ pdom(L7) &= \{L7\} \end{aligned}$$

また, X が Y を後支配し, $X \neq Y$ であるとき, X は Y を厳密に後支配 (strictly postdominate) するといいい, X が Y を厳密に後支配し, Y から X への路にそれ以外に Y を厳密に後支配するノードがないとき, X は Y を直接後支配 (immediately postdominate) するという. X は Y を直接後支配するときに $X = ipdom(Y)$ と書くと, 図 2.2 の CFG では, 以下のようなになる.

$$\begin{aligned}
ipdom(L1) &= \{L7\} \\
ipdom(L2) &= \{L5\} \\
ipdom(L3) &= \{L5\} \\
ipdom(L4) &= \{L5\} \\
ipdom(L5) &= \{L7\} \\
ipdom(L6) &= \{L7\} \\
ipdom(L7) &= \{\}
\end{aligned}$$

2.4 支配木

支配木 (dominator tree) とは, CFG 上のあるノード X と, X を直接支配するノード Y を有向辺でむすんだグラフである. 得られたグラフは木構造となる. なぜなら, CFG のあるノード Z を直接支配するノードはひとつだからである. 支配木の根は CFG の入口ノードである. 以下, 支配木のノード X の子ノード Y を $Y \in domChild(X)$ と書く. 図 2.2 の CFG に対応した支配木を図 2.3 に示す.

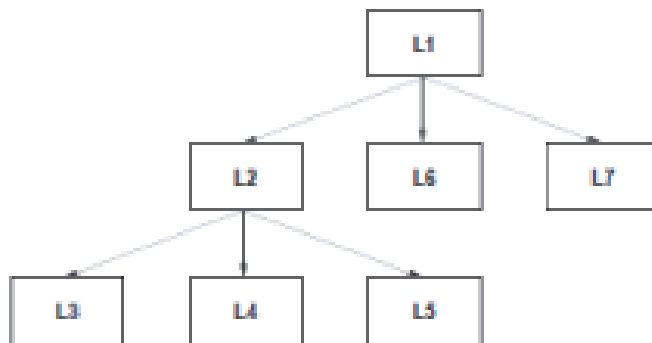


図 2.3: 支配木

2.5 制御依存

制御依存とは, ノード X からのある辺を辿ると, その後必ず Y を通るが, 別の辺をとれば Y は通らないことを表す. 次の 2 つが成り立つとき Y は X に制御依存すると定義される.

- X から Y への空でない路があり, Y はその路の X より後のすべてのノードを後支配する
- Y は X を厳密に後支配しない

この関係を図で表すと，図 2.4 のようになる．

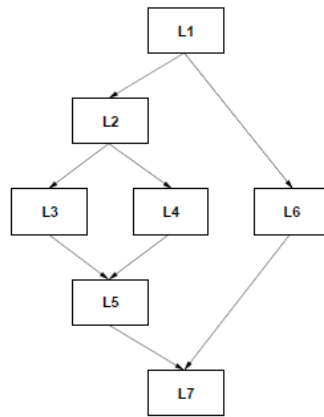


図 2.4: 制御依存

Y が制御依存するノード X から Y 方向へのエッジの集合を返す関数を $CD(Y)$ と書く．例えば図 2.2 の CFG の各ノードに対する制御依存は以下ようになる．

$$\begin{aligned} CD(L1) &= \{\} \\ CD(L2) &= \{\{L1 \rightarrow L2\}\} \\ CD(L3) &= \{\{L2 \rightarrow L3\}\} \\ CD(L4) &= \{\{L2 \rightarrow L4\}\} \\ CD(L5) &= \{\{L1 \rightarrow L2\}\} \\ CD(L6) &= \{\{L1 \rightarrow L6\}\} \\ CD(L7) &= \{\} \end{aligned}$$

2.6 抽象構文木

文が与えられたとき，文法によってその文がどのように導かれるか，つまり生成規則をどのように適用すればその文が得られるかを，木の形で示すと都合がよい．このような木を解析木 (parse tree) または導出木 (derivation tree) という [15]．

例として，式 2.1 に対する，表 2.1 の文法から導かれる解析木を図に示す．

$$i_1 := (i_2 + i_3) * i_4 / \text{num} \quad (2.1)$$

解析木によって，演算とそのオペランド (被演算子) がはっきり示される．例えば，図では， i_2 と i_3 を加えたものに i_4 をかけ，それを num で割ったものを i_1 へ代入することが読み取れる．

$$\begin{aligned}
S &\rightarrow i := E \\
E &\rightarrow E * E \\
E &\rightarrow E / E \\
E &\rightarrow E + E \\
E &\rightarrow E - E \\
E &\rightarrow (E) \\
E &\rightarrow i \\
E &\rightarrow \text{num}
\end{aligned}$$

表 2.1: 簡単な文法の例

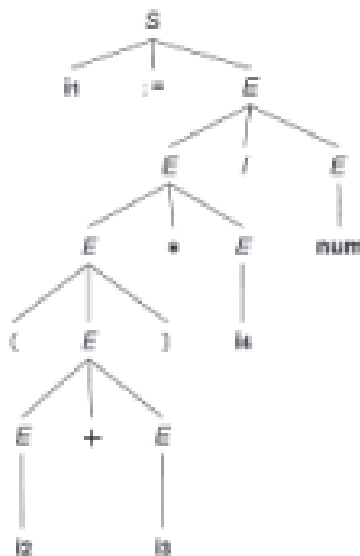


図 2.5: 式 2.1 に対する解析木

解析木は構文を正確に表しているが、演算の意味を示すには内部の節がやや冗長である。解析木のかわりに、解析木のうち演算に本質的なものだけを取り出した木を用いることがある。これは、節として演算子、節の子供として節の演算子のオペランドがくるようにした木であり、抽象構文木 (abstract syntax tree) あるいは単に構文木 (syntax tree) と呼ばれる。抽象構文木では、葉をつなげても、一般に文にはならない。式 2.1 に対する抽象構文木を図 2.6 に示す。抽象構文木には非終端記号 (表 2.1 の S と E) は現れない。演算に本質的でない構造、例えば子供が一人だけの枝や、 $()$ をまとめるためだけの構造は取り除かれる。その結果、一般に抽象構文木のほうが解析木より簡潔である。

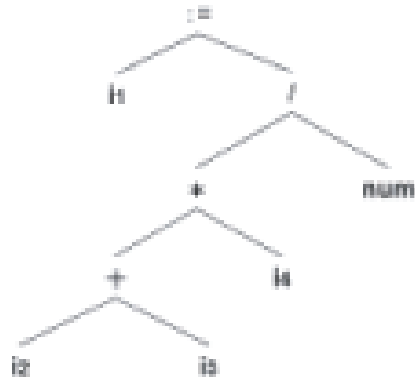


図 2.6: 式 2.1 に対する抽象構文木

2.7 シフト演算

シフト演算とは、2進数で表されたビットパターンを、右または左にずらす演算である。シフト演算には、「論理シフト演算」と「算術シフト演算」の二種類がある。

2.7.1 論理シフト演算

左シフトでは、溢れたビットは捨て、空いたビットには0を入れる。

右シフトでは、溢れたビットは捨てて、空いたビットには0を入れる。

図 2.7、2.8 を参照。

2.7.2 算術シフト演算

左シフトでは、符号ビットはそのままにし、溢れたビットは捨て、空いたビットには0を入れる。

右シフトでは、符号ビットはそのままにし、溢れたビットは捨て、空いたビットには符号ビットと同じビットを入れる。

図 2.9、2.10 を参照。

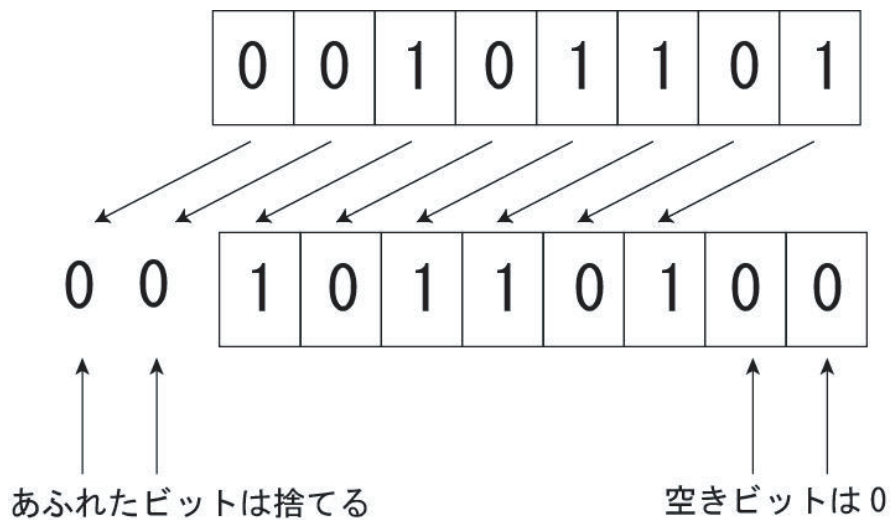


図 2.7: 論理シフト演算 : 左シフト

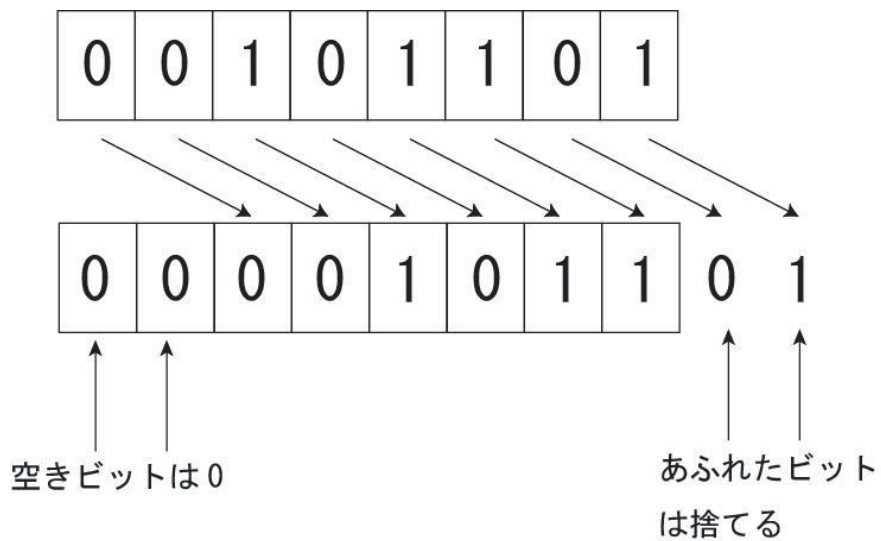


図 2.8: 論理シフト演算 : 右シフト

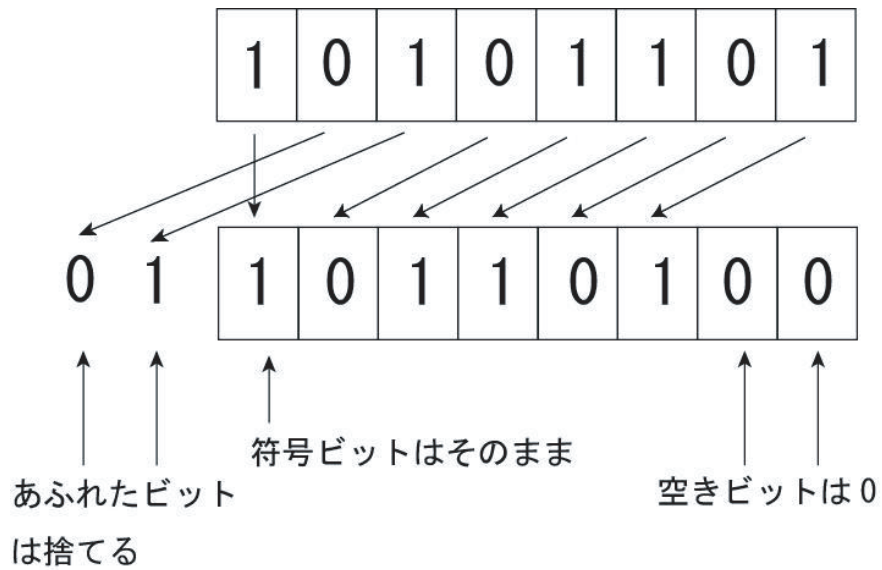


図 2.9: 算術シフト演算：左シフト

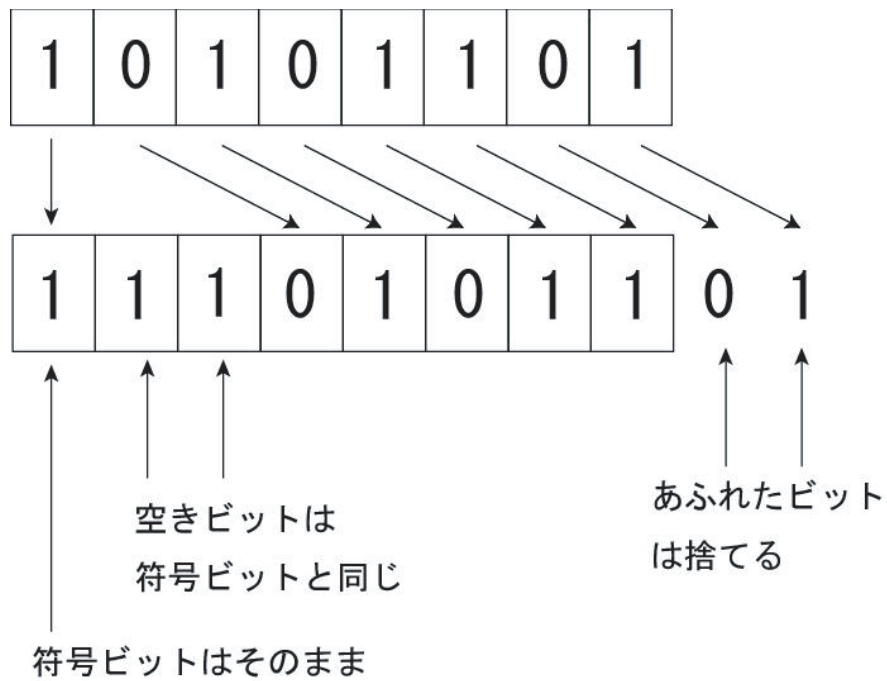


図 2.10: 算術シフト演算：右シフト

第3章 Single Instruction / Multiple Data (SIMD)

3.1 Single Instruction / Multiple Data

Single Instruction / Multiple Data (以下 SIMD) とは、複数のデータを1回の命令で同時に処理をする計算手法のことである。SIMD 命令は、アーキテクチャによって記述方式が異なる。

SIMD 命令は、大量のデータに同じ演算を行う処理に適しており、そのような処理の多く見られる画像処理のライブラリに頻繁に使用されている。

代表的な SIMD 命令としては、以下のものがある。

x86 (Intel プロセッサ)	MMX、SSE、3DNow!
PowerPC (IBM プロセッサ)	VMX、AltiVec
SPE (in Cell.B.E)	SPU intrinsic

3.1.1 SIMD プログラミング例

SIMD プログラミングの簡単な例として、Cell.B.E. の SPU イントリンジックを用いたものを紹介する。以下のループ文を SIMD 演算を用いて計算することを考える。

```
int i, N;
int a[N], b[N], c[N];

for(i = 0; i < N; i++){
    a[i] = b[i] * c[i];
}
```

図 3.1: SIMD 化前

このプログラムを、SPU の SIMD 演算を使用して書くと以下のようになる。

```

int i, N;
int a[N] __attribute__((aligned(16))); // アラインメントを16バイトに揃える
int b[N] __attribute__((aligned(16)));
int c[N] __attribute__((aligned(16)));

vector signed int *vec_a, *vec_b, *vec_c; // signed int 型のベクトル変数

vec_a = a;
vec_b = b;
vec_c = c;

for(i = 0; i < N/4; i++){
    vec_a[i] = spu_mult(vec_b[i], vec_c[i]);
}

for(i; i < N; i++){ // Nが4で割りきれなかった時のための処理
    a[i] = b[i] * c[i];
}

```

図 3.2: SIMD 化後

`vec0 = spu_mult(vec1, vec2)` というのが、SPU イントリンジックであり、`vec1` と `vec2` のそれぞれの対応する要素を掛け算し、`vec0` の対応する要素に入れる演算である。

3.2 コンパイラによる自動 SIMD 化

コンパイラによりプログラムで SIMD 命令を使用できる部分を探し出し、SIMD で計算するようにする最適化を、コンパイラの自動 SIMD 化という。

プログラムを SIMD 命令を使用して実行するには、プログラマーがコード内で対応する SIMD のアセンブリの命令や組み込み関数の命令を用いることで実現することもできる。しかし、それには、プログラマーが SIMD 命令に対する十分な理解が必要であり、特にアセンブリ言語を用いて記述した際には、プログラムの可読性が低くなってしまふ、という問題点がある。

そこでコンパイラによる自動 SIMD 化の最適化を用いることによって、プログラマーの負担を減らし、かつ、ソースコードの可読性を維持したまま SIMD 命令の効果により、計算時間を短縮することができる。しかしながら、コンパイラによる自動 SIMD 化の効果は限定的であり、その性能は発展途上である。以下の節では、コンパイラの自動 SIMD 化の直面する課題を説明する。

3.2.1 データのアラインメント

SPU ハードウェアは、レジスタからデータのロード・ストアをする際、16 バイトの境界にしかアクセスできないという制限を持っている。例として以下のプログラムを SIMD 命令を用いて計算することを考える。

```
int a[n] __attribute__((aligned(16)));
int b[n] __attribute__((aligned(16)));
int c[n] __attribute__((aligned(16)));

for( i = 0; i < n; i++){
    a[i+2] = b[i+1] + c[i+3];
}
```

このプログラムを、データのアラインメントを気にせずに計算してしまうと、図3.3のように、 $b[1]$ を含むベクトルには、 $b[0]$, $b[1]$, $b[2]$, $b[3]$ が格納されており、 $c[3]$ を含むベクトルには、 $c[0]$, $c[1]$, $c[2]$, $c[3]$ が格納されているので、この2つのベクトルを足し算すると出力されるベクトルは、 $b[0]+c[0]$, $b[1]+c[1]$, $b[2]+c[2]$, $b[3]+c[3]$ となり、目的の答えと違うものが出来てしまう。これは、SPU アーキテクチャが、 $b[1]$ の値をロードしようとした場合、16 バイトの境界である $b[0]$ の値から $b[3]$ の値を同時にロードするからである。

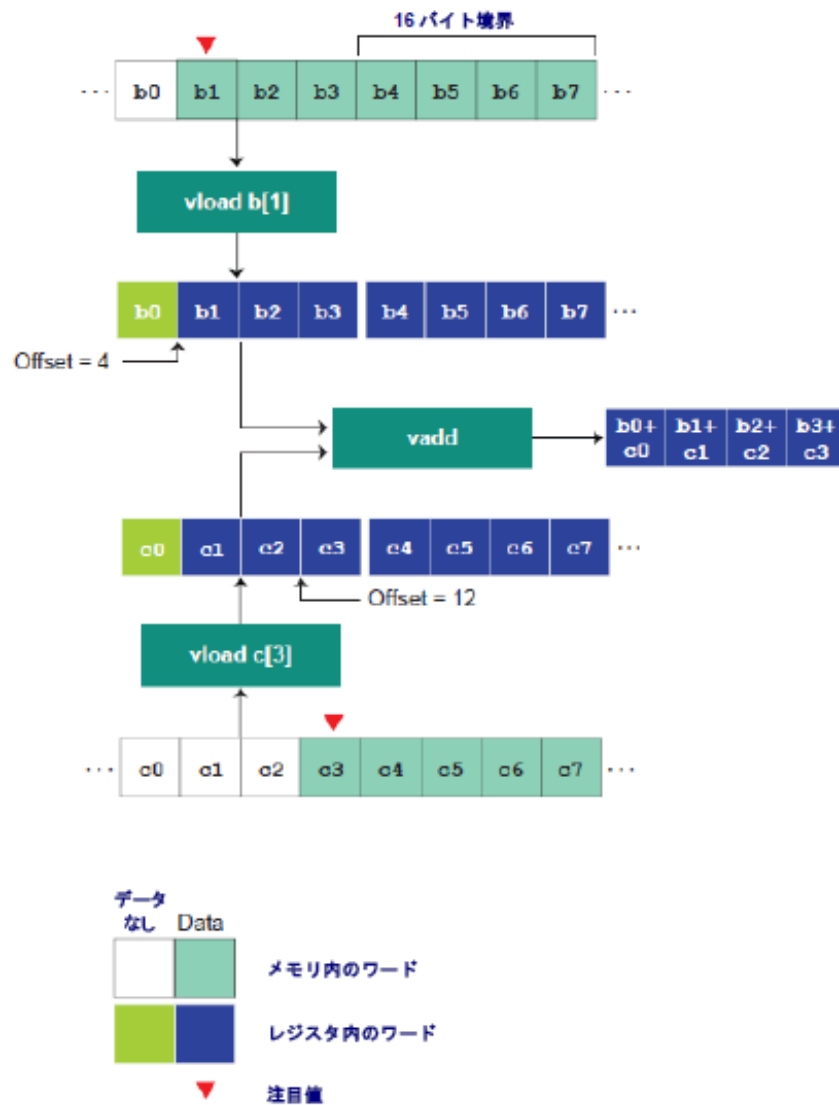


図 3.3: 間違ったロードの仕方

これを避けるためには、ベクトルのシャッフル命令を使う。シャッフル命令とは、二つのベクトルの要素を入れ替えて、新しいベクトルに出力する命令である。??のように、`b[0]`, `b[1]`, `b[2]`, `b[3]` が格納されているベクトルと、`b[4]`, `b[5]`, `b[6]`, `b[7]` が格納されているベクトルをシャッフルして各ベクトルの要素を入れ替え、`b[1]`, `b[2]`, `b[3]`, `b[4]` が格納されたベクトルを作る。以降のベクトルも同様にシャッフル命令を利用して作っていく。配列 `c` についても同様の操作を行う。

このようにして新しくできた配列 `b`, `c` の値を格納したベクトルを足し合わせて、目的の演算結果を得る。最後に演算結果を配列 `a` に格納する際には、??の用に、配列の始めと終わりの値だけ注意して格納すればよい。

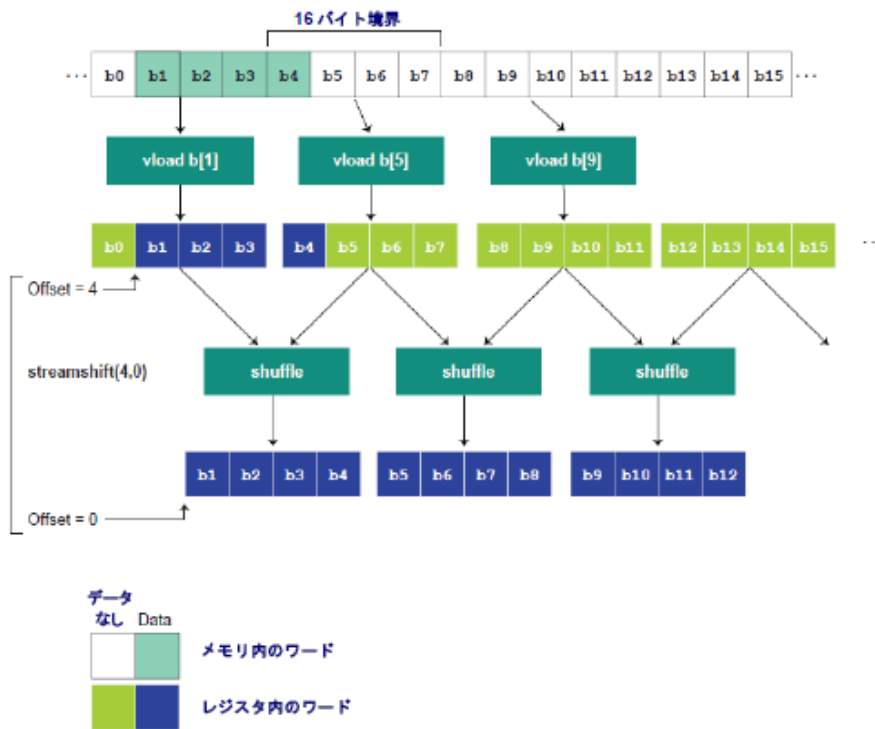


図 3.4: 正しいロードの仕方 1

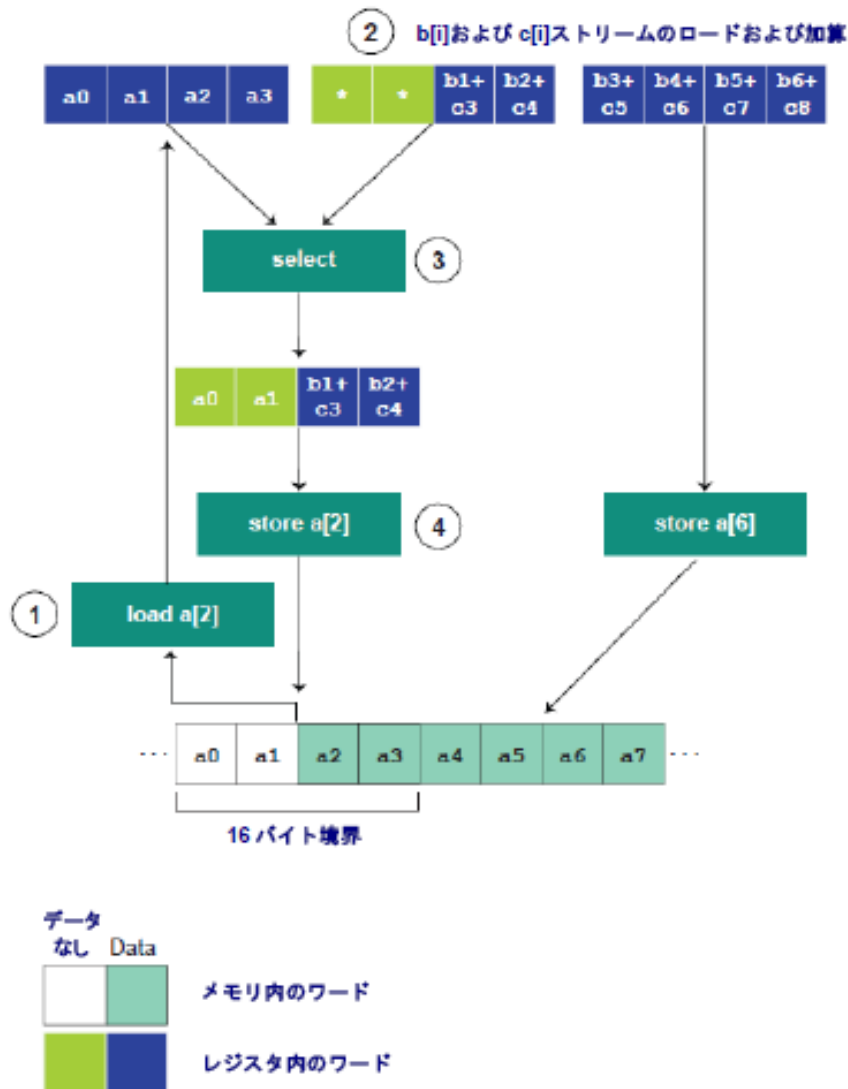


図 3.5: 正しいロードの仕方:2

3.2.2 scatter / gather 命令

scatter/gather 命令とは、メモリ上で散在するデータに対して、連続してアクセスする命令である。Cell Broadband Engine は、scatter/gather 命令をサポートしていないので、散在するデータに対して SIMD 命令を適用するのはオーバーヘッドがかかるので避けた方がよい。例として、次のプログラムは SIMD 化に不適切。

```
for(i = 0; i < n; i++){
    a[i] = b[2 * i];
}
```

3.2.3 データ型の混在

画像処理のプログラムには、さまざまなデータ型が混在し、あるデータ型から他のデータ型への変換が頻繁に行われる。しかし、SIMD 命令セットのベクトル型は 16 バイトの一つのベクトルに、char 型なら 16 個、short 型は 8 個、int 型、float 型は 4 個、そして double 型は 2 個という様に、データ型によって格納できる要素数が異なっているため、例えば char 型のデータから short 型のデータに変換した場合に一つのベクトルに格納される要素数が変化してしまう。プログラマーはそのような点に注意をしながら、最も効率よく SIMD 命令を使用することが必要となる。

3.2.4 SIMD 命令セットの偏り

SPU 命令セットは、命令によっては、int 型はサポートされているが、double 型はサポートされていないということがある。int 型は一番多くサポートされており、long long 型はあまりサポートされていない。double 型は計算速度が少し遅くなる。

第4章 Cell Broadband Engineの 説明

4.1 Cell Broadband Engine

Cell Broadband Engine(以降: Cell.B.E.)とは、SONY、IBMそしてTOSHIBAが共同で開発したプロセッサであり、SONYのPlayStation3に搭載されている。

Cell.B.E.は、高い並列計算能力を持っており、単純な計算が多い、画像処理に長けている。その理由となっている特徴として、以下のものがある。

特徴1 . 1つの制御プロセッサ (PPU) と、8つの演算用プロセッサ (SPU) から構成されている

特徴2 . 演算用のSPUプロセッサには、SIMD命令用のレジスタが128本積まれている

4.2 PowerPC Processor Element

PowerPC Processor Element (PPE) は、汎用の64ビットRISCプロセッサであり、システムの全体的な制御を行う。PPEおよび、Synergistic Processor Elements (SPE) 上で動作する全てのアプリケーションに対し、制御をする。

PPUは、PowerPCアーキテクチャ命令セットと、ベクトルSIMDマルチメディア拡張命令を実行する。

4.3 Synergistic Processor Element

8個のSynergistic Processor Element (SPE) はそれぞれ、128ビットのRISCプロセッサであり、大量のデータを扱い、集中的な計算を行うSIMDおよびスカラ演算のアプリケーションに特化している。SIMD命令の計算には長けている一方、スカラの計算速度は遅いという特徴を持つ。

Synergistic Processor Unitは、それぞれが、直接参照できる256KBのローカル・ストレージを持ち、メインメモリにはダイレクト・メモリ・アクセスと呼ばれる方法でアクセスをする。

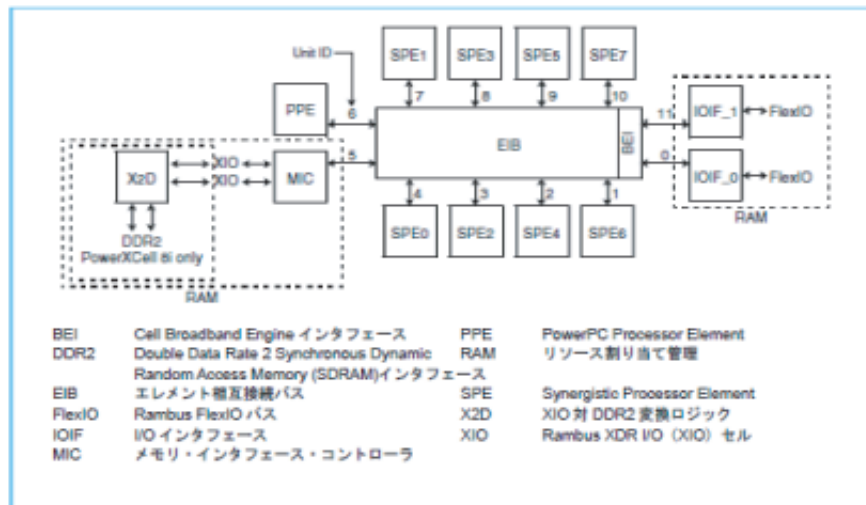


図 4.1: Cell.B.E. の構造

4.4 プログラミング手法

Cell Broadband Engine Architecture プロセッサを対象としたプログラミングを行う時、以下の二つの課題がある。

1. 複数のプロセッサ (SPE) で、並列計算可能な個所の特定
2. 個々のプロセッサ上で動くプログラムに対して、SIMD 命令が使用可能な箇所の特定

1 は関数単位でのプログラムの並列化である。問題となってくるのが、それぞれの SPE がデータの受け渡しをするためにメインメモリにアクセスする際に、オーバーヘッドが生じてしまうことであり、それをカバーできるような並列化を行わなくてはならない。

2 は演算単位の並列化である。問題となってくるのが、個々のデータの依存関係であり、SIMD 命令の使用で演算の順序を変えることによって、依存関係が変わらないようにしなければならない。

また、ソースファイルは PPE 用のものと、SPE 用のものを別々に用意する必要があり、コンパイルも PPE と SPE はそれぞれ別に行う。PPE、SPE 用のコンパイラとしては、フリーのもので GCC の派生である、ppu-gcc と spu-gcc がある。

第5章 GCCの自動SIMD化とその問題点

5.1 概要

GCCの自動SIMD化の最適化機能は、GCC version4.xから搭載されている。使用例は以下の通りである。例えば、x86のSSE SIMD命令を出力したい場合は、`gcc-4.0 -O2 -ftree-vectorize -msse -ftree-vectorizer-verbose=7 LoopParallel.c`のようにオプションを設定し、実行する。`-ftree-vectorize`でループのSIMD化を行い、これは通常最適化のレベル2 (`-O2`)と同時に使用することによって、SIMD化の効果を上げることができる。`-msse`はx86用のSIMD命令であるSSEを出力するためのオプションである。PowerPC用のAltiVecを出力したい場合は、代わりに`-maltivec`のオプションを付ける。`-ftree-vectorizer-verbose=N` (`N`は0から7までの値)では、SIMD化をする際の最適化の情報を見ることができる。参考として、図5.1に`-ftree-vectorizer-verbose=6`によって出力された解析情報を示す。SPE用のコンパイラである、`spu-gcc`にも自動SIMD化の機能は搭載されており、使い方は以下のようなになる。

```
spu-gcc -O2 -ftree-vectorize -ftree-vectorizer-verbose=7 LoopParallel.c
```

`spu-gcc`の場合は、ターゲットマシーンが決まっているので、`-msse`、`-maltivec`などのオプションを付ける必要はない。

```

jcapimin.c:125: note: not vectorized: too many BBs in loop.
jcapimin.c:130: note: not vectorized: too many BBs in loop.
jcapimin.c:130: note: vectorized 0 loops in function.

jcapimin.c:205: note: not vectorized: unhandled data-ref
jcapimin.c:205: note: vectorized 0 loops in function.

jcapimin.c:160: note: not vectorized: too many BBs in loop.
jcapimin.c:160: note: not vectorized: Bad inner loop.
jcapimin.c:162: note: not vectorized: too many BBs in loop.
jcapimin.c:162: note: vectorized 0 loops in function.

jcapimin.c:66: note: Unknown alignment for access: *cinfo_3
jcapimin.c:66: note: Alignment of access forced using peeling.
jcapimin.c:66: note: LOOP VECTORIZED.
jcapimin.c:69: note: Unknown alignment for access: *cinfo_3
jcapimin.c:69: note: Unknown alignment for access: *cinfo_3
jcapimin.c:69: note: Alignment of access forced using versioning.
jcapimin.c:69: note: Alignment of access forced using versioning.
jcapimin.c:69: note: LOOP VECTORIZED.
jcapimin.c:69: note: vectorized 2 loops in function.

```

図 5.1: spu-gcc の自動 SIMD 化機能デバッグ

5.2 GCC の自動 SIMD 化の構造

5.2.1 解析部

与えられたループが並列化可能であるか否かの解析をする。
解析の流れはおおよそ以下ようになる。

- 1 . ループの出口が一つである
- 2 . ループ内の演算が配列参照、もしくはポインタ参照である
- 3 . ループ内にスカラ変数によるループ繰越依存が無い
- 4 . 書き込みの配列のアラインメントが揃っている
- 5 . ループ依存のデータ依存がない
- 6 . ループ毎のデータの参照領域が、メモリ上で連続である
- 7 . 必要であれば、ループピーリングをする
- 8 . 演算に対し、データ型が適切である

以上の段階を、1 から順に条件に該当するか否かを調べていき、条件を満たさな

かった時点で解析部から出る。1 から 8 の全ての条件を満たした場合、次のループの再構築部に進む。

5.2.2 再構築部

ループ解析部の条件を全てクリアしたループを、並列化するために書き換えをする。

ループ再構築の流れはおおよそ以下のようになる。

1 . 実行時に、アラインメントのずれの値が分かるループに対して、その値に対し、並列化可能であった場合と、並列化不可能であった場合とで、異なるコードを実行するように設定する

例として、次のループ文を SIMD 化することを考える。

```
for(i=0; i<=11; i++){
    c[i+n] = a[i] + b[i];
    d[i+n] = a[i] + b[i];
}
```

このループ文は、n の値が配列 a, b, c, d をベクトルデータに格納した場合の、1 ベクトルデータ内の要素数よりも大きければ SIMD 化が可能であり、小さければ不可能である。もし、n の値がコンパイル時に分からない場合は、GCC は、SIMD 化可能だった場合のコードと、SIMD 化不可能であった場合のコードの 2 種類のループ文を用意し、それらに対し n の値を条件分岐としてジャンプさせる。

2 . 配列のアラインメントが揃っていないループに対し、ループピーリングにより、アラインメントを揃える。例として、次のループ文を SIMD 化することを考える。

```
for(i=0; i<n; i++){
    a[i+2] = b[i+12]-10;
    b[i+2] = a[i+7]*3;
}
```

配列 a, b は先頭のアドレスが 16 バイトにアラインメントされているとする。この時、初めの 2 回の実行文をループの外に出すことによって、残りのループ文をアラインメントを気にせずに実行することができる。

第6章 並列化コンパイラ向け共通インフラストラクチャCOINS

本章では、本手法の適用を行う際に利用した並列化コンパイラ向け共通インフラストラクチャ (COmpiler INfraStructure, COINS) について説明を行う。

6.1 概要

COINS は、コンパイラ研究の基盤となる共通のコンパイラインフラストラクチャの作成をテーマに 2000 年度より研究が進められてきたプロジェクトである [4]。現在他機関でも、多くの最適化コンパイラの研究、開発が行われているが [6, ?, ?], 本研究室の佐々政孝教授が COINS の研究プロジェクトに携わっているため、本研究室では COINS をインフラとして利用した研究が多く行われており、データの蓄積が行いやすいため、本手法の実装を行うインフラに COINS を選択した。

6.2 構成

一般にコンパイラは、フロントエンド (front end) とバックエンド (back end) から構成される。フロントエンドは、原始プログラム (source program) を中間コード (intermediate code) と呼ばれる内部形式に変換する。バックエンドは、中間コードを計算機の機械コード (machine code) に変換する。フロントエンドはさらに、字句解析器 (lexical analyzer)、構文解析器 (syntax analyzer)、意味解析器 (semantic analyzer) に分けられる。バックエンドは、最適化器 (optimizer) とコード生成器 (code generator) に分けられる。これら各部分は、コンパイラのフェーズと呼ばれる。

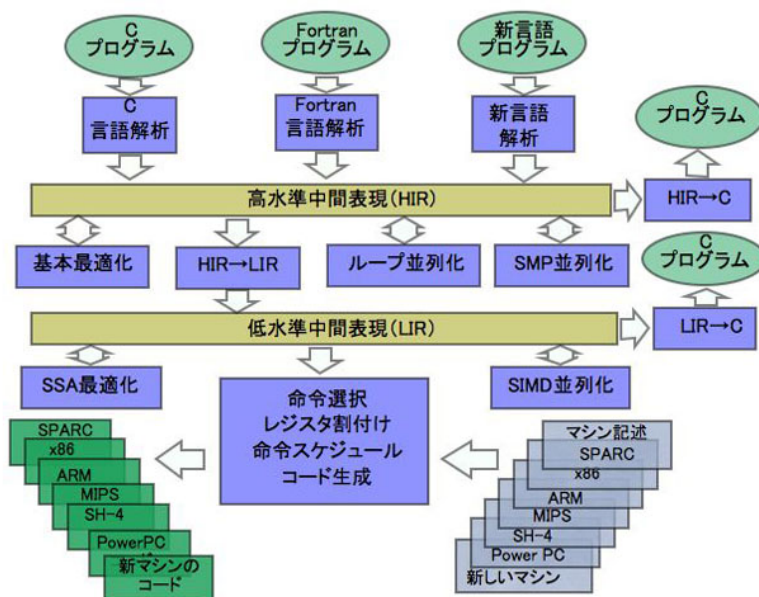


図 6.1: COINS 構成図

COINS では，複数の入力言語，複数の対象機種に対応する 2 つの中間コードがある (図 6.1) . 入力言語の論理構造に近いレベルの中間コードを，高水準中間表現 (high-level intermediate representation, HIR) と呼び，機械語に近いレベルの中間コードを，低水準中間表現 (low-level intermediate representation, LIR)

6.3 COINS の特徴

COINS の特徴として 2 つの中間表現 (HIR, LIR) がある. ここで，HIR はソースコードレベルの表現と対応し，LIR は命令レベルでの表現と対応する. 本研究の実装で使用する HIR は，コンパイラによる並列化を行うことができる中間表現で，下記のような特徴がある .

- if 文や for ループなどの制御構造が容易に認識できる
LIR だとこれらの制御構造は分岐命令を組み合わせることによって実現される
- 配列構造が容易に認識できる
LIR だと配列構造がそのアドレス計算を含むいくつかの命令列に分解されてしまう

本研究では，以上のような特徴から並列化のための変形を容易に行うことができるソースコードレベルの中間表現のある COINS を利用した.

6.4 高水準中間言語 HIR について

本節では、本研究で使用する中間表現である HIR について説明する [5].

6.4.1 概要

HIR は一般的な手続き型言語において共通的な概念を抽出し、特定の言語に依存しないように表現された抽象構文木である。HIR 自身は Java 言語で実装される。

6.4.2 具体表現

HIR をテキストで表すとき、節は

(オペレータ 型 第1子 第2子 … 第n子)

葉は

< 種別 記号 型 >

と表す。HIR ではコンパイル単位全体を1つの木として表現する。その中には一般に、複数の副プログラム定義が含まれる。例えば次のプログラムがあったとする。

プログラム 6.1 (入力プログラム)

```
int fact(int p)  if(p <= 1)  return 1;  else  return p * fact(p - 1);
```

このプログラムから生成される HIR は図 6.4.2 のようになる。

```

(prog
  <null 0 void>
  <nullNode>
  (subpDef void
    <subp <SUBP < int > false false int> fact>
  <null void>
    (labeldSt void
      (list <labelDef _lab1>)
      <block void
        (if void
          (cmpLe bool <var int p> <const int 1>)
          (labeldSt int
            (list <labelDef _lab3>)
            (return int <const int 1>))
          (labeledSt int
            (list <labelDef _lab4>)
            (return int
              (mult int
                <var int p>
                (call int
                  (addr <PTR <SUBP < int > false false int>>
                    <subp <SUBP < int > false false int> fact>))
                  (list
                    (sub int <var int p> <const int 1>))))))
            (labeledSt void
              (list <labelDef _lab5>))
            <null void>))))))
  )

```

第7章 提案するSIMD並列化向け ループ変換

7.1 if変換

ループ内にif文が存在すると、ループ内の基本ブロックの数が2つ以上となって
しまうため、GCCで自動SIMD化ができなくなる。

例として、以下のコードをspu-gccで-O2 -ftree-vectorize -ftree-vectorizer-verbose=6
のオプションを付けてコンパイルをすることを考える。

```
int a[i];
int i;
for(i=0; i<N; i++){
    if(a[i] > i){
        a[i] = i;
    }else{
        a[i] = i * i;
    }
}
```

この時、自動SIMD化のデバッグ情報として、次のメッセージが返ってくる。

```
note: not vectorized: too many BBs in loop
```

これは、ループ内に基本ブロックが2つ以上含まれているときに返されるメッセ
ジである。これに対し、ループ内にif文が含まれていた場合、if文の条件式をif文の
then部、else部に組み込むことによって基本ブロック数を減らす手法を提案する。

基本ブロックの除去

例として、次の形のif文を変換することを考える。

```
if(cond){
    t1 = S_t1;
    t2 = S_t2;
    ...
    tm = S_tm;
}else{
```

```

    e1 = S_e1;
    e2 = S_e2;
    ...
    en = S_en;
}

```

この時、cond とは、if の条件が true の場合は 1、false の場合は 0 となる変数であると定義する。この時、上の if 文は、cond を用いて以下の用に変形できる。

```

/** then-part */
t1 = cond * S_t1 + (1-cond) * t1;
t2 = cond * S_t2 + (1-cond) * t2;
...
tm = cond * S_tm + (1-cond) * tm;

/** else-part */
e1 = (1-cond) * S_e1 + cond * e1;
e2 = (1-cond) * S_e2 + cond * e2;
...
en = (1-cond) * S_en + cond * en;

```

この変形の意味を説明すると、if 文の条件が true の場合、すなわち、cond の値が 1 の場合は、then 部の変形である以下の部分、

```

t1 = cond * S_t1 + (1-cond) * t1;
t2 = cond * S_t2 + (1-cond) * t2;
...
tm = cond * S_tm + (1-cond) * tm;

```

は、次のようになる。

```

t1 = S_t1;
t2 = S_t2;
...
tm = S_tm;

```

それに対し、else 部の変形である以下の部分、

```

e1 = (1-cond) * S_e1 + cond * e1;
e2 = (1-cond) * S_e2 + cond * e2;
...
en = (1-cond) * S_en + cond * en;

```

は、次のようになる。

```
e1 = e1;
e2 = e2;
...
en = en;
```

この計算は、実際は何もしていないのと同じ意味を持ち、結果として、cond の値が 1 の場合は if 文の then 部に対応するコードのみ計算され、else 部は計算されない。

if 文の条件が false の場合、すなわち、cond の値が 0 の場合も同様に、then 部に対応するコードは計算されず、else 部に対応するコードのみ計算されることになる。

cond の作り方

cond を作るには、第 2 章で説明した算術シフト演算を使用する。例として、以下の if 文を考える。

```
int a[i];
int i;
if(a[i] >= i){
    ...
}else{
    ...
}
```

この時、if 文の条件式は $a[i] \geq i$ となる。この式は変形すると $a[i] - i \geq 0$ となる。ここで、 $a[i] - i$ の値を算術右シフトで 32 ビット右シフトさせた値を考えると、 $a[i] - i$ の値は int 型であるので、32 ビット右シフトをすると、全てのビットに符号ビットと同じビットがたつことになる。まとめると、

```
(a[i] - i) >> 32 = 0   if : a[i] >= i
(a[i] - i) >> 32 = -1  if : a[i] < i
```

となり、これより cond は次のように書ける。

```
cond = (a[i] - i) >> 32 + 1
```

同様にして、if 文の条件が $a[i] > i$ 、 $a[i] \leq i$ 、 $a[i] < i$ の場合はそれぞれ、 $a[i] - i - 1$ 、 $i - a[i]$ 、 $i - a[i] - 1$ を右に 32 ビットシフトさせればよい。 $a[i] > i$ の値が char の時は 8 ビット、short の時は 16 ビット、int/float の時は 32 ビット右にシフトさせる。

以上の手法により、if 文を変形させる。変形対象とする if 文は、入出力が無く、ループの出口が一つであり、関数呼び出しが無いループ文 (for 文または while 文) に含まれ、if 文の条件式として CMP_GT, CMP_GE, CMP_LT, CMP_LE, CMP_EQ, CMP_NE のいずれかを使用しているものとする。ループが入れ子になっている場合は、内側のループから順に変換を施す。

7.1 に if 変換全体の設計図を示す。

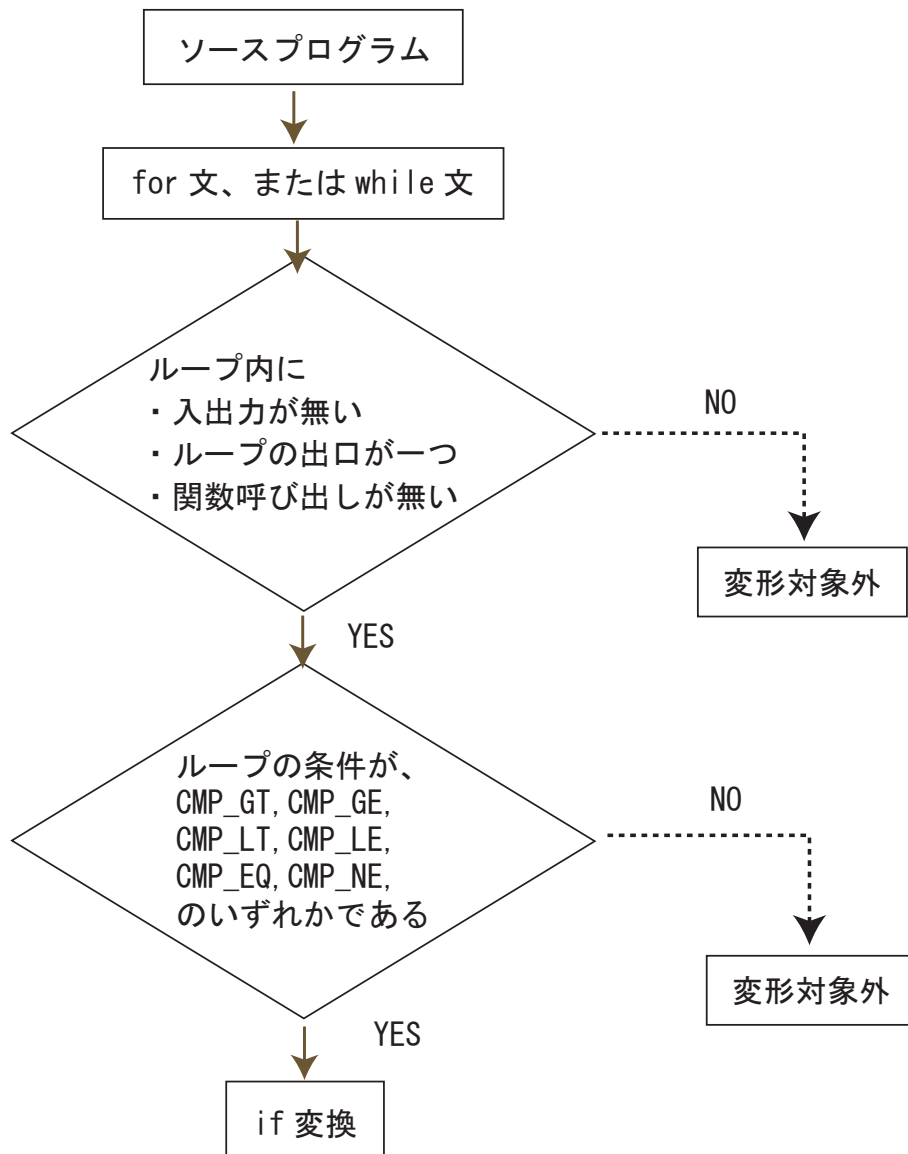


図 7.1: ソースコードレベルでの if 変換器

最後に例として、

```

int a[i];
int i;
for(i=0; i<N; i++){
    if(a[i] > i){
        a[i] = i;
    }else{
        a[i] = i * i;
    }
}

```

のコードを if 変換すると次のようになる。

```

int a[i];
int i;
for(i=0; i<N; i++){
    a[i] = ((a[i]-i)>>32+1) * i + (1-((a[i]-i)>>32+1)) * a[i];
    a[i] = (1-((a[i]-i)>>32+1)) * (i*i) + ((a[i]-i)>>32+1) * a[i];
}

```

7.2 ループ分割

ループ内に、自動 SIMD 化可能な文と自動 SIMD 化不可能な文が混在していた場合、そのループ文は SIMD 化不可能と判定される。例として、次のコードを考える。

```

int a[N], b[N];
int i;

for(i=0; i<N; i++){
    a[i] = i*i;
    b[i] = a[i] + i;
    printf("a[%d] = %d, b[%d] = %d\n", i, a[i], b[i]);
}

```

このループ文は、ループ内の上 2 つのコードは SIMD 化可能であるが、

```

printf("a[%d] = %d, b[%d] = %d\n", i, a[i], b[i]);

```

は関数呼び出しをしているので SIMD 化不可能である。よって、ループ全体でも SIMD 化不可能と判断される。このようなループに対し、SIMD 化可能なコードと、SIMD 化不可能なコードを別々のループ文に分けることで、SIMD 化可能なコードが同じループ内にある SIMD 化不可能なループによって SIMD 化されない状態になることを防ぐ。

例のコードは、以下のようにループ分割することで、ループ内の上 2 つのコードを SIMD 化することができる。

```

for(i=0; i<N; i++){
    a[i] = i*i;
    b[i] = a[i] + i;
}

for(i=0; i<N; i++){
    printf("a[%d] = %d, b[%d] = %d\n", i, a[i], b[i]);
}

```

尚、ここでループ分割の対象とするループ文は、入出力が無く、ループ内に出口が一つであり、かつループ繰越依存が無いものとする。SIMD 化不可能であると判

断されるコードは、関数呼び出しのあるコードとし、それ以外はSIMD化可能であるとした。ループが入れ子になっている場合は、内側のループから順に変換を施す。

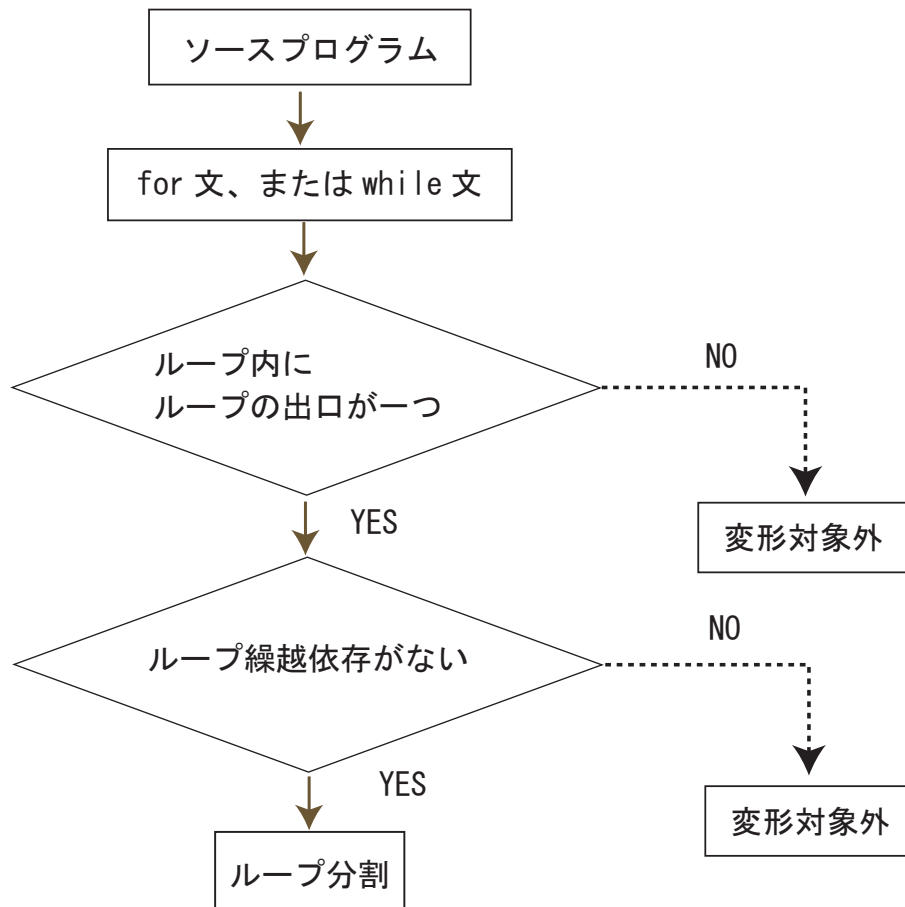


図 7.2: ループ分割

7.3 スカラ拡張

ループ内のスカラ変数を配列データに変換することによって、ループ内のデータの依存関係を消去する変換を「スカラ拡張」という。[14] 例として次のコードを考える。

```
for(i=0; i<N; i++){
    T = a[i];
    a[i] = b[i];
    b[i] = T;
}
```

このコードは、配列 a、b の値を取り換えるコードである。この時、ループの一番初めの文と、一番終わりの文は、変数 T によりループ繰越逆依存関係にある。よって、

このループ文は並列化不可となる。しかし、以下のコードの様に、変数 T を配列データにすることによって、並列化可能とすることができる。

```
T[0] = T;    //前処理
for(i=0; i<N; i++){
    T[i] = a[i];
    a[i] = b[i];
    b[i] = T[i];
}
T = T[N];    //後処理
```

スカラ拡張により変数の依存関係を除去できるのは、変数の依存関係が「変数の値を使用している」ことからではなく、「変数のメモリ位置を利用している」ことから起きている場合である。上の例で言えば、変数 T は、配列 a、b の値を交換するための一時格納場所として、毎回のループ文で使用されており、毎回のループ内での値を使用している訳ではない。

少し複雑な例として、次のコードを考える。

```
for(i=0; i<N; i++){
    T = T + a[i] + a[i+1];
    a[i] = T;
}
```

このコードをスカラ拡張して書くと、次のようになる。

```
T[0] = T;
for(i=0; i<N; i++){
    T[i+1] = T[i] + a[i] + a[i+1];
    a[i] = T[i+1];
}
T = T[N];
```

この時点で、1 番目の文と 2 番目の文との間の依存関係は無くなったが、1 番目の文によるループ繰越依存があるため、このループ文は並列化不可である。しかしこの文は、次のように変更すると SIMD 化が可能となる。

```
T[0] = T;
for(i=0; i<N; i++){
    T[i+VF] = T[i] + a[i] + a[i+1];
    a[i] = T[i+VF];
}
T = T[N]+T[N+1]+...+T[N+VF];
```

ここで VF とは SIMD 化を行う際の並列度であり、char の場合は 16、short は 8、int・float は 4、double は 2 となる。

今回実装したスカラ拡張は、式が $s += \dots$ 、 $s -= \dots$ 、 $s *= \dots$ の形をしているものに限った。

第8章 実験と考察

8.1 実装

実装は、COINS 上の高水準中間表現 HIR 上に行った。COINS の HIR は、ソースコードの意味に近い木構造のデータ構造をしており、GCC も内部にそのような木構造を持つため、GCC のフロントエンドに実装する際もほぼ同じプログラミングをすればよいと考えられる。COINS 上で実装したのはソースコードレベルの変形だけであり、コード変換後の COINS の HIR を C 言語に逆変換したものを出力ファイルとした。これを Cell Broadband Engine の SPU プロセッサのコンパイラである、spu-gcc に自動 SIMD 化をするオプションを立てて入力し、バイナリファイルを出力した。

8.2 実験と考察

テストプログラムとして、画像処理のベンチマークである、MediaBenchII [11] と SIMD ベンチマーク [19] を使用した。MediaBenchII は静止画、動画の符号化・復号化をするソフトウェアであり、大学や研究機関を中心に画像処理のベンチマークとして利用されている。静止画では、JPEG や JPEG2000 規格での符号化・復号化、動画では、h264/MPEG-4 AVC、MPEG-2、MPEG-4、H263 規格での符号化・復号化のライブラリが用意されている。SIMD ベンチマークは、COINS の SIMD 化部分の実装を担当した、島根大学の鈴木貢教授が設計したものである。

また、準備実験として、IBM の商用 Cell.B.E. 向けコンパイラである XL C/C++ Multicore Acceleration for Linux を使用し、spu-gcc との性能比較をした。

8.2.1 準備実験

spu-gcc の自動 SIMD 化の性能を測るために、準備実験として、IBM の商用 Cell.B.E. 向けコンパイラである XL C/C++ Multicore Acceleration for Linux (以下 XLC MA) の自動 SIMD 化機能との比較を行った。テストプログラムとして MediaBenchII の h263dec、h263enc、mpeg2dec、djpeg、jpg2000dec を用いた。

これらのプログラムに対し、XLC MA では `-O2 -qhot=simd -qreport` の自動 SIMD 化のオプションを、spu-gcc では `-O2 -ftree-vectorize -ftree-vectorizer-verbose=6` のオプションを立ててそれぞれコンパイルした。この実験から、XLC MA の方が、spu-gcc よりも自動 SIMD 化できたループ文の数が多いことが分かった。そして、XLC MA

で自動 SIMD 化できたのにも関わらず、spu-gcc で自動 SIMD 化できなかったループ文とそのデバッグ情報を集計し、どのような理由で spu-gcc では SIMD 化が出来ないと判断されたか分析をした。その結果を以下に示す。

SPU-GCCの自動SIMD化デバッグ情報	ループ数	%
number of iterations cannot be computed	7	14.29
too many BBs in loop.	9	18.37
stmt not supported	2	4.08
unsupported unaligned store.	3	6.12
can't determine dependence between	13	26.53
unsupported use in stmt.	1	2.04
complicated access pattern.	2	4.08
unhandled data-ref	7	14.29
Bad inner loop.	1	2.04
can't create epilog loop 1.	1	2.04
Unknown alignment for access	1	2.04
value used after loop.	2	4.08
TOTAL	49	100

図 8.1: spu-gcc の自動 SIMD 化 デバッグ情報

XLC MA で SIMD 化できて、spu-gcc で SIMD 化不可能であったループ文は全部で 53 個であり、spu-gcc のデバッグ情報には図 8.1 の 12 種類があった。このうち、本研究のコード変換により、SIMD 化可能となりうるものが「too many BBs in loop」と「complicated access pattern」の二つに含まれていた。デバッグ情報「too many BBs in loop」とは、ループ内に 2 つ以上の基本ブロックが含まれている場合に出力されるもので、これらの 12 個のループ文には全てに if 文もしくは、三項間演算子「？」による条件文が含まれていた。

8.2.2 実験

準備実験で、SIMD 化可能なループにも関わらず、spu-gcc では SIMD 化できなかったループに対し、本手法で提案するソースコードの変形をほどこした後で spu-gcc で SIMD 化できるかどうか実験を行った。

if 変換は、準備実験で「too many BBs in loop」のデバッグ情報を出力し、SIMD 化出来なかったループ 12 個に対して、ループ分割は「complicated access pattern」の内の、ループ分割が適用できるとみなしたループ 1 つに対して変換を施した。

その結果、if 変換により新たに 3 つのループ文が、ループ分割により新たに 1 つのループ文が spu-gcc によって自動 SIMD 化可能となった。

実験環境は、表 8.1 に示す。

表 8.1: 実験環境

機種	PlayStation3 CECHKxx
プロセッサ種別	Cell Broadband Engine: Synergistic Processor Element
プロセッサ数	CPU: 1(PPE)+8(SPE), GPU: 1
メモリ容量	主メモリ:256MB, VRAM:256MB
オペレーティング環境	Yellow Dog Linux v6.1

8.3 考察

8.3.1 if変換

if変換(7.1節参照)によってSIMD化可能となったループ文は、MediaBenchIIベンチマークの中の以下の3つであった。

jpg2000dec/jasper-1.701.0/src/libjasper/base/jas_seq.c 268 行目

```
for (j = matrix->numcols_, data = rowstart; j > 0; --j, ++data) {
    *data = (*data >= 0) ? ((*data) >> n) : (-((-(*data)) >> n));
}
```

jpg2000dec/jasper-1.701.0/src/libjasper/base/jas_seq.c 289 行目

```
for (j = matrix->numcols_, data = rowstart; j > 0; --j, ++data) {
    v = *data;
    if (v < minval) {
        *data = minval;
    } else if (v > maxval) {
        *data = maxval;
    }
}
```

h263enc/tmn-1.7/putbits.c 125 行目

```
while (m--) {
    bit[length-m] = (val & (1<<m)) ? '1' : '0';
}
```

これらのループに if 変換を施したところ、ループ内の基本ブロックが減り、SIMD 化できるようになった。これらのループに対し、SIMD 化前のものと SIMD 化後のもので時間測定を行った。

測定は、各ループを計算しているメソッドをベンチマークから抜き出し、引数を与えて実行したものを測定した。引数の設定は、実際にベンチマークを動かした時に、該当するメソッドに渡している引数の値を参考にした。時間測定は、以下の 3 つの場合で行った。

Case 1 . 本手法での変換無しのソースコードを、`spu-gcc -O2` でコンパイルした場合

Case 2 . 本手法での変換無しのソースコードを、`spu-gcc -O2 -ftree-vectorize` でコンパイルした場合

Case 3 . 本手法での変換を施した後に、`spu-gcc -O2 -ftree-vectorize` でコンパイルした場合

該当するメソッドをベンチマークから抜き出して実行した理由としては、ベンチマーク全体を Cell.B.E. 上で動作させる際には、どの部分を PPU または SPU で実行させるかによって全体のパフォーマンスに差が生じてしまうので、そのような議論を避けるためである。

時間測定の結果を図 8.2 に示す。

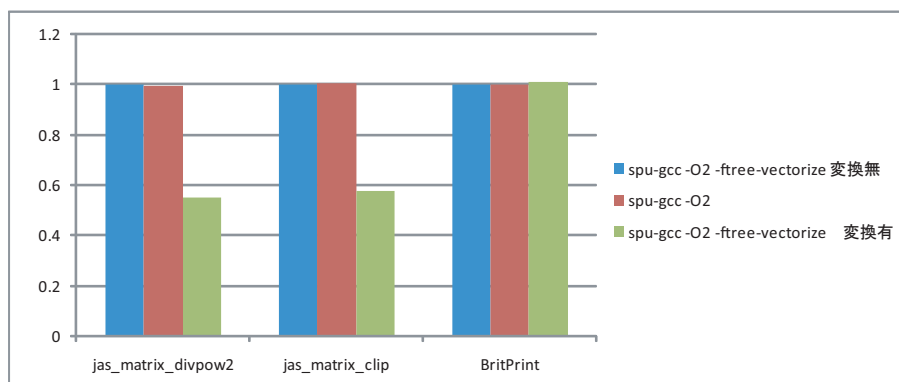


図 8.2: 時間測定結果 : if 変換

グラフは、上記の Case 2 でコンパイルした実行ファイルの実行時間を 1 とした時の、各メソッドの実行時間である。jas_matrix_divpow2 と jas_matrix_clip の 2 つのメソッドでは、それぞれ 0.55 倍、0.57 倍の実行時間となった。

BitPrint に関しては、効果が見られなかった。ソースコードを解析したところ、これは BitPrint の for 文の回る回数が少なすぎたためと考えられる。

SIMD 化されなかったループの考察

if 変換を施しても spu-gcc で SIMD 化できなかったものについては、以下のような原因があった。

- 1 . COINS で HIR から C へ逆変換した際に、コードが少し変形され、for 文の中身に新たな文が追加されてしまった。
- 2 . if 変換で「too many BBs in loop」のフェーズで SIMD 化不可と判断されることは無くなったが、まだ他の要因で SIMD 化が出来ない場合

8.3.2 ループ分割

ループ分割 (7.2 節参照) によって SIMD 化可能となったループ文は MediaBenchII の中の以下であった。

h263enc/tmn-1.7/mot_est.c 133 行目

```
for (k = 0; k < 5; k++) {  
    Min_FRAME[k] = INT_MAX;  
    MV_FRAME[k].x = 0;  
    MV_FRAME[k].y = 0;  
    MV_FRAME[k].x_half = 0;  
    MV_FRAME[k].y_half = 0;  
}
```

この for 文は、ループ分割によって次のように変換された。

```
for (k = 0; k < 5; k++) {  
    Min_FRAME[k] = INT_MAX;  
}  
  
for (k = 0; k < 5; k++) {  
    MV_FRAME[k].x = 0;  
    MV_FRAME[k].y = 0;  
    MV_FRAME[k].x_half = 0;  
    MV_FRAME[k].y_half = 0;  
}
```

この時、一番目のループ文が SIMD 化可能となる。この関数についての時間測定を行ったところ、spu-gcc でコンパイルすると、正確な動作をしなくなる、という問題が発生し、時間測定は行えなかった。ソースコードを見る限りは、for 文の回る回数が少なすぎるため、SIMD 化の効果は期待できないと考えられる。また、このメソッドは、複雑な制御文が多く存在したため、そもそもが SPU 向けではなく、PPU 向けのメソッドであった。

8.3.3 スカラ拡張

スカラ拡張 (7.3 節参照) の効果は、SIMD ベンチマークの sum で行った。スカラ拡張を行ったループ文は以下である。

```
for (; cnt > 0; cnt--) {  
    sum += *b;  
    b++;  
}
```

ループ内の sum、*b のデータ型が char、short、float、double の場合にそれぞれ実験を行った。int 型のスカラ拡張は spu-gcc で既にサポートされていたため、変換は行わなかった。

時間測定は、上のループ文を 10000 回まわし、以下の 3 つの場合で行った。

Case 1 . 本手法での変換無しソースコードを、spu-gcc -O2 でコンパイルした場合

Case 2 . 本手法での変換有りのソースコードを、spu-gcc -O2 でコンパイルした場合

Case 3 . 本手法での変換を施した後に、spu-gcc -O2 -ftree-vectorize でコンパイルした場合

図 8.3 に実験結果を示す。

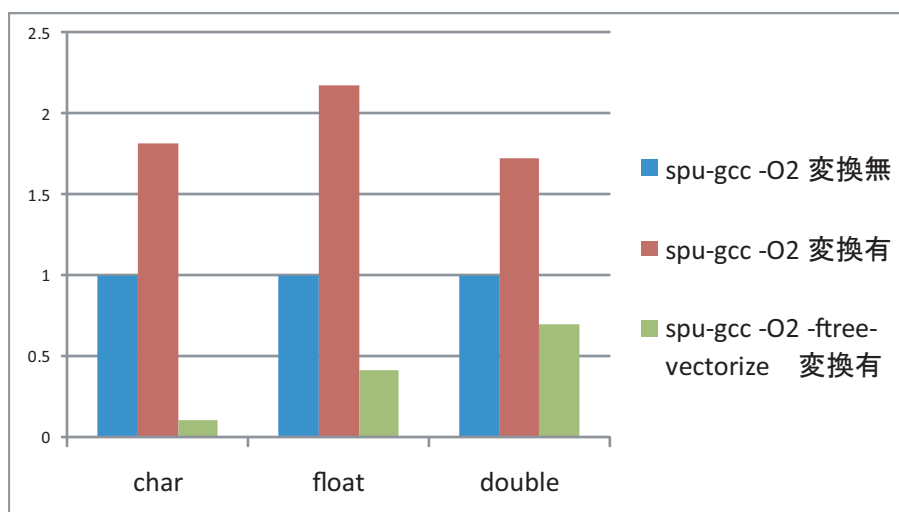


図 8.3: 時間測定結果：スカラ拡張

SIMD 化の効果は、double、float、char の順に大きかった。これは SIMD 化の並列度が高ければ高いほど効果が得られることが原因であると考えられる。Case2 の場合の実行結果は、Case1 の場合の実行結果の 2 倍弱の時間がかかってしまった。スカラ変換を施しても SIMD 化されなかった場合は、かなりのオーバーヘッドがかかってしまうことが分かる。

8.4 今後の課題

8.4.1 ループ分割の適用範囲

本研究では、ループ分割の対象としているループ文はループ繰越依存がないものに限っている。しかし、ループ繰越依存があるループに関しても、ループ繰越依存がある文と無い文に分け、その情報を考慮した上でループ分割をすることが可能である。ループ繰越依存のあるループもループ分割の対象とすることによって、spu-gccの自動SIMD化の効果がより期待できるようになると考えられる。

8.4.2 SIMD化できない他の要因の解決

本研究の3つの手法により、SIMD化が可能となると見込まれたループ文は上で上げた以外にも存在したが、他の理由でSIMD化はできなかった。以下にSIMD化できなかった主な理由を挙げる。

1. コンパイル時にイテレーション回数が分からない
2. ポインタの参照領域解析が複雑であることによる、データ依存解析の限界
3. 複数データ間での演算

1は、spu-gccはループのイテレーション回数が分からなかったときは、その回数が1ベクトルに格納する要素数よりも小さかった場合の可能性を考慮し、安全のためSIMD化不可と判断する。これに対しては、イテレーション回数が小さかった場合と、十分に大きかった場合で場合分けをしたコードを出力するようにする方法が考えられる。

2に対しては、ポインタの別名解析を強化する必要があると考えられる。

3に対しては、ソースコードレベルでの変換ではなく、アセンブリレベルでの操作をする必要がある。

8.4.3 ループのベクタ化

現在、spu-gccでは、ループ内のデータの依存関係を解析し、ループの再構築を行う、ようなループのベクタ化機能は実装されていない。(このループベクタ化の機能は、FORTRANコンパイラには実装されている)ベクタ化の実装を行うことで、spu-gccの自動SIMD化の効果をより向上できるようになると考えられる。

第9章 関連研究

9.1 鈴木らの研究

本手法を実装した COINS 上でも SIMD 命令の生成を行っている [17]。鈴木らは、SIMD 命令セット向け最適化を、「ベクタ化を軸としたソースコードレベルで可能な最適化」と、「そのような変換を施されたプログラムに対して適切な SIMD 命令を生成する最適化」の 2 段階に分割し、後者を中心に実装を行っている。SIMD 命令と COINS の低水準中間表現で SIMD 命令と対応する動作テンプレートとの照合を行い、SIMD 命令を作成している。しかし、現在の実装 (COINS1.4.2.2) では、ソースコードレベルで変形を行った一部のプログラムしか変形ができていない。また、SIMD 特有の命令である shuffle 命令などに対応していない。今後の課題として、COINS によるループのベクタ化の実装が残されており、総合すると COINS の SIMD 並列化は実用レベルには至っていない状態である。

9.2 Pande らの研究

並列化されていないプログラムコードを Cell.B.E. にポータリングするには、プログラムの解析や、Cell.B.E. の PPU, SPU それぞれの特徴を十分に理解していることが必要となり、プログラマにとって容易なことではない。

米国ジョージア工科大学の Santosh Pande 教授ら [16] は、並列化されていないソースコードを Cell.B.E. にポータリングする際の補助となるソフトウェアをいくつか開発している。

9.2.1 Glimpses

Glimpses はプログラムの解析をするソフトウェアであり、「プログラムの挙動を理解すること」、「SPU プロセッサ上にポータリングできる部分の候補を示すこと」、また「SPU へポータリングした際のパフォーマンスを測ること」ができる。このツールにより、プログラマはプログラムのどの部分を SPU にポータリングするべきかの判断が容易にできるようになる。

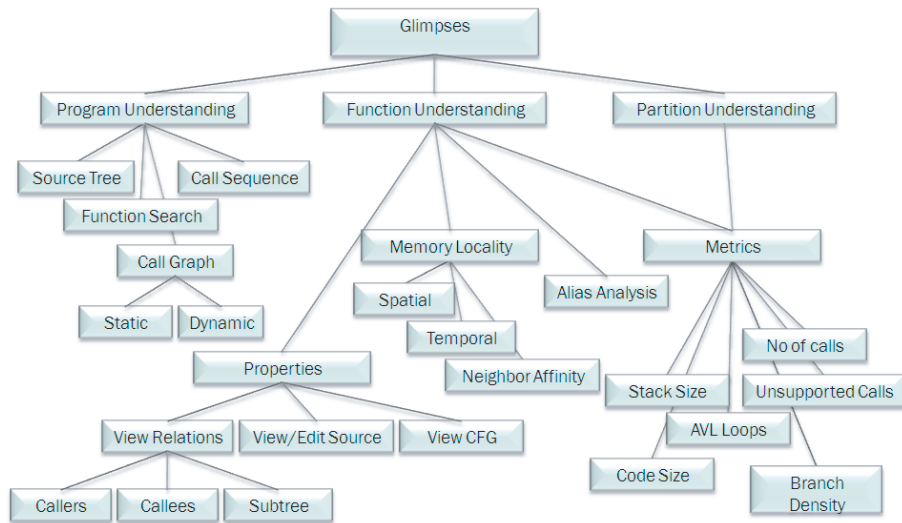


図 9.1: Glimpses の構造

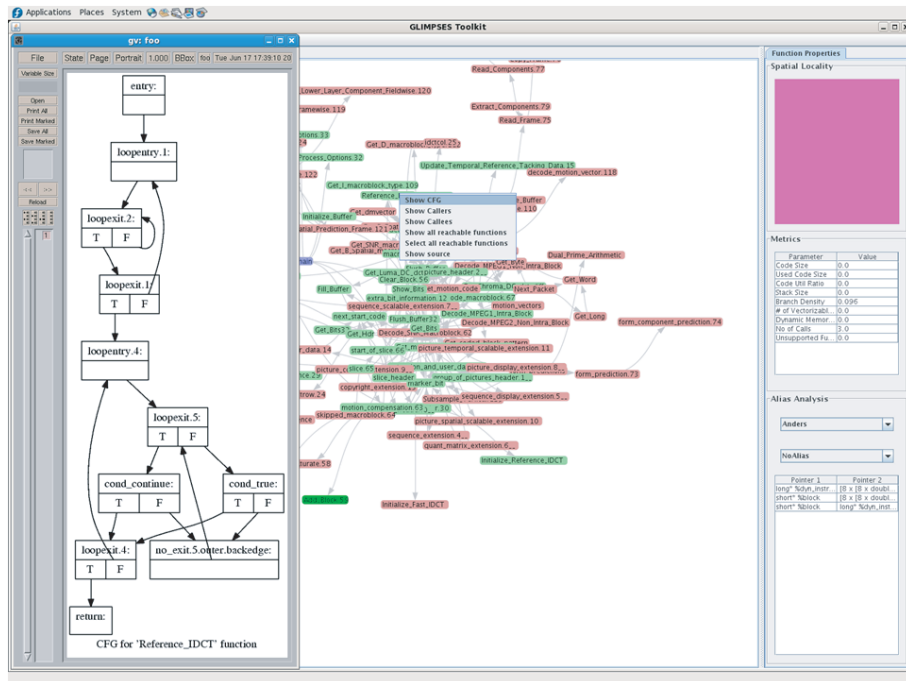


図 9.2: Glimpses のデモ

9.2.2 AutoPort

Glimpses で Cell.B.E. の PPU, SPU のそれぞれで動かすコードが決まったならば、次は元のソースコードを PPU 用、SPU 用にそれぞれ書き換えコンパイルする、という作業をしなければならない。この際、PPU、SPU 間のメモリ転送 (DMA 転送) の

処理、それに伴うデータのアラインメントなどを考慮しなければならない。AutoPort は、これらの処理をプログラマの代わりに実行し、元のプログラムから PPU 用コードと SPU 用コードを出力してくれる。

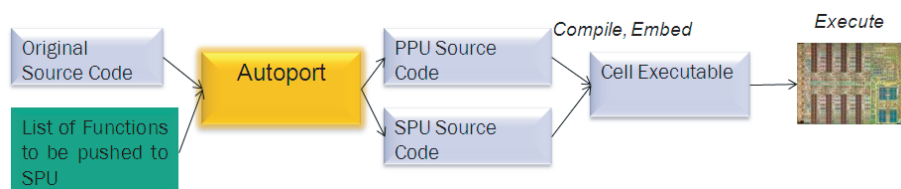


図 9.3: AutoPort の位置付け

第10章 まとめ

本研究では、Cell.B.E. の SPU プロセッサのコンパイラである、spu-gcc の自動 SIMD 化に向けたソースコードレベルでの変換を行った。変換は COINS 上で実装し、画像処理のベンチマークである、MediaBenchII と SIMD ベンチマークで評価を行った。その結果、MediaBenchII の中で、SIMD 化可能であるにも関わらず、spu-gcc で SIMD 化できなかったループの内、約 7.5% のループ文が新たに SIMD 化出来るようになり、そのループ文を含むメソッドで最大 60% 以下の実行時間となった。スカラー変換は、SIMD ベンチマークで性能を測ったところ、SIMD 化により最大で 10% の実行時間となった。しかし、ループの回る回数が少ない場合や、演算用の SPU プロセッサに適していないようなループ文に対しては、SIMD 化した際の効果が見られなかった。また、本手法を適用しても、コンパイル時にイテレーション回数が分からない、ポインタ型によりデータの依存関係が解読できない、複数のデータ型にまたがる演算がされている、などの理由により SIMD 化不可能と判断されるループ文が多く存在した。spu-gcc での自動 SIMD 化の不可判定は、いくつかの要因から不可とされることが多いため、今後は本研究以外の要因も取り除いていく必要がある。

謝辞

本研究を進めるにあたり多大なるご指導ご鞭撻を頂いた、東京工業大学 数理・計算科学専攻教授の佐々政孝先生、また COINS の使い方について教えて下さった電気通信大学の渡辺坦教授、拓殖大学の岩澤教授、そして島根大学の鈴木貢准教授に深く感謝の意を表します。

また、佐々研究室の皆様にはさまざまな面で助力を頂きました。あらためまして、ここに深くお礼申し上げます。

参考文献

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools 2nd ed.* Addison Wesley, 2006.
- [2] Alexandre E. Eichenberger , Kathryn O'Brien , Kevin O'Brien , Re E. Eichenberger , Peng Wu , Tong Chen , Peter H. Oden , Daniel A. Prener , Janice C. Shepherd , Byoungro So , Zehra Sura , Amy , Tao Zhang , Peng Zhao , Michael Gschwind, "Optimizing Compiler for a CELL Processor" Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques Pages: 161 - 172 Year of Publication: 2005 ISBN ISSN:1089-795X , 0-7695-2429-X
- [3] Andrew W. Appel. *Modern Compiler Implementation in Java second edition.* Cambridge University Press, 2002.
- [4] COINS Project. COINS home page. <http://www.coins-project.org/> .
- [5] COINS Project. 高水準中間表現の概要, 2004. <http://www.coins-project.org/050303/base/HirOutline.pdf>.
- [6] GNU-Project. GCC homepage. <http://gcc.gnu.org/>.
- [7] IBM: Software Development Kit for Multicore Acceleration Version 3.0 "Programming Tutorial"
- [8] IBM: XL C/C++ for Multicore Acceleration for Linux, V10.1 "Getting Started with XL C/C++ Version 10.1"
- [9] Jaishri M. Waghmare, "Seminar Report on Auto-vectorization in GCC" Submitted as partial fulfillment of the requirements for the degree of Master of Technology Department of Computer Science and Engineering Indian Institute of Technology, Bombay, April 14, 2008
- [10] Manuel Hohenauer, Felix Engel, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, "A SIMD optimization framework for retargetable compilers", ACM Transactions on Architecture and Code Optimization (TACO) archive Volume 6 , Issue 1 (March 2009) Article No.2 2009 ISSN:1544-3566
- [11] MediaBench Consortium, <http://euler.slu.edu/~fritts/mediabench/>

- [12] 中田育男. コンパイラの構成と最適化 第2版. 朝倉書店, 2009.
- [13] 中西 悠, 渡邊 啓正, 本多 弘樹, ”コードの性能可搬性を提供する SIMD 向け共通記述方式 (コンパイラ)”, 情報処理学会論文誌. コンピューティングシステム 48(SIG_13(ACS_19)) pp.95-105 2007年8月15日
- [14] Randy Allen & Ken Kennedy, ”Optimizing Compilers for Modern Architectures”, MORGAN KAUFMANN PUBLISHERS
- [15] 佐々政孝. プログラミング言語処理系. 岩波書店, 1989.
- [16] Santosh Pande, College of Computing, Georgia Institute of Technology, <http://www.cc.gatech.edu/~santosh/>
- [17] 鈴木 貢, 藤波 順久: ”COINSにおける SIMD 並列化 ”, コンピュータソフトウェア, Vol.25, No.1, pp.65-81, 2008年1月.
- [18] 鈴木貢, 小川大介, 室田. 樹, 渡邊坦: ”SIMD ベンチマークの設計と実装 ”, 情報処理学会論文誌: コンピューティングシステム, Vol.46, No.SIG 16 (ACS 12), pp.95-107, 2005年12月.
- [19] 鈴木 貢, 電気通信大学, SIMD ベンチマークの設計と実装, 情報処理学会論文誌. コンピューティングシステム, 46(SIG_16(ACS_12)) pp.95-107 2005年12月15日
- [20] SONY: ”Cell Broadband Engine アーキテクチャ用 C/C++ 言語拡張 Version 2.3” CBEA JSRE Series Cell Broadband Engine Architecture Joint Software Reference Environment Series, 2006.