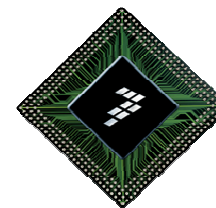


PowerPC® G4プロセッサの 性能を最大限に引き出すための アプリケーション最適化技法

ver 1.4



Kazuhiro Nagano
k.nagano@freescale.com

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. © Freescale Semiconductor, Inc. 2006.



ハイエンドプロセッサと最適化の重要性

最新のハイエンドプロセッサは動作周波数だけでなく処理の並列度も向上している

機能実現を優先した通常のプログラムは、並列処理されにくい

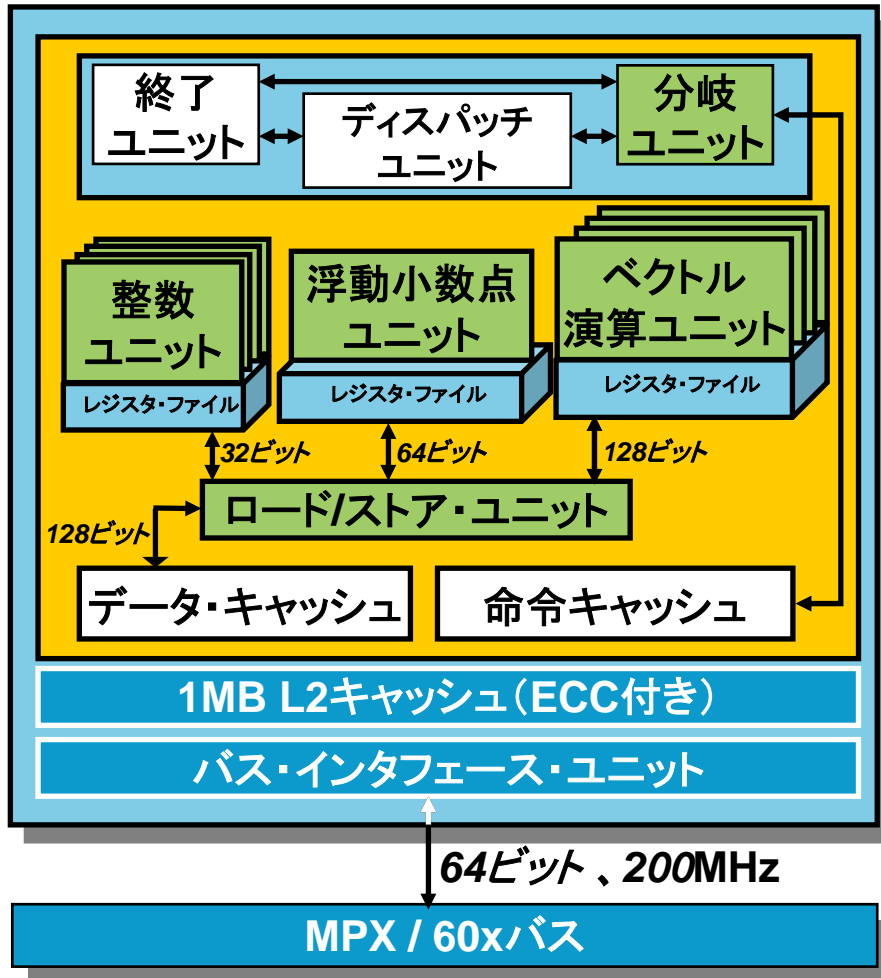
ハイエンドプロセッサの潜在性能を引き出しきれない

効率的なプログラミング
手法を習得

PowerPC G4であればソフトウェアでFPGA相当の演算性能を実現

最終製品の競争力が向上

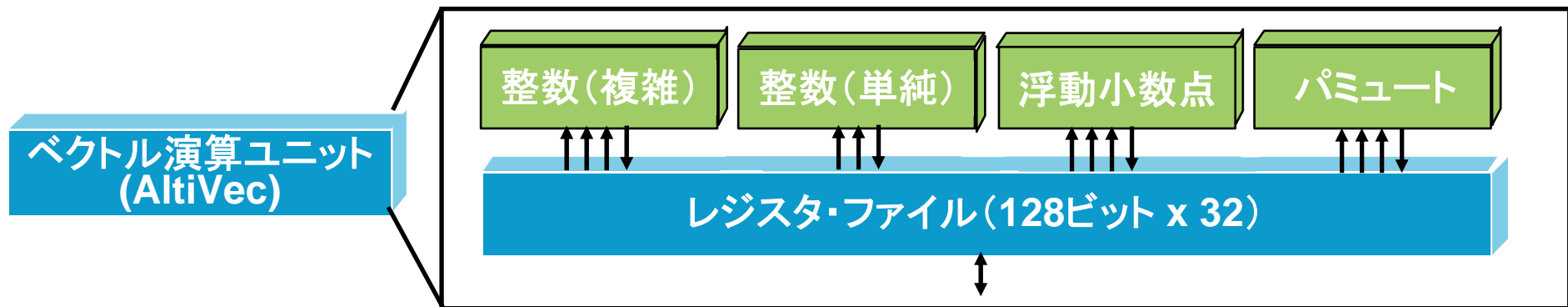
- **PowerPC G4の概要**
- **最適化プログラミングの概要**
- **最適化プログラミング・テクニックの応用例**



MPC7448ブロック図

- PowerPC G4は、組み込み市場におけるハイエンドのRISCプロセッサ
- 11個の実行ユニットを搭載
- 128ビット処理を行うベクトル演算ユニット (AltiVec) を搭載
- 同時に3命令+1分岐の命令発行が可能
- 演算ユニットごとに32個のレジスタで構成される豊富なレジスタ・ファイル

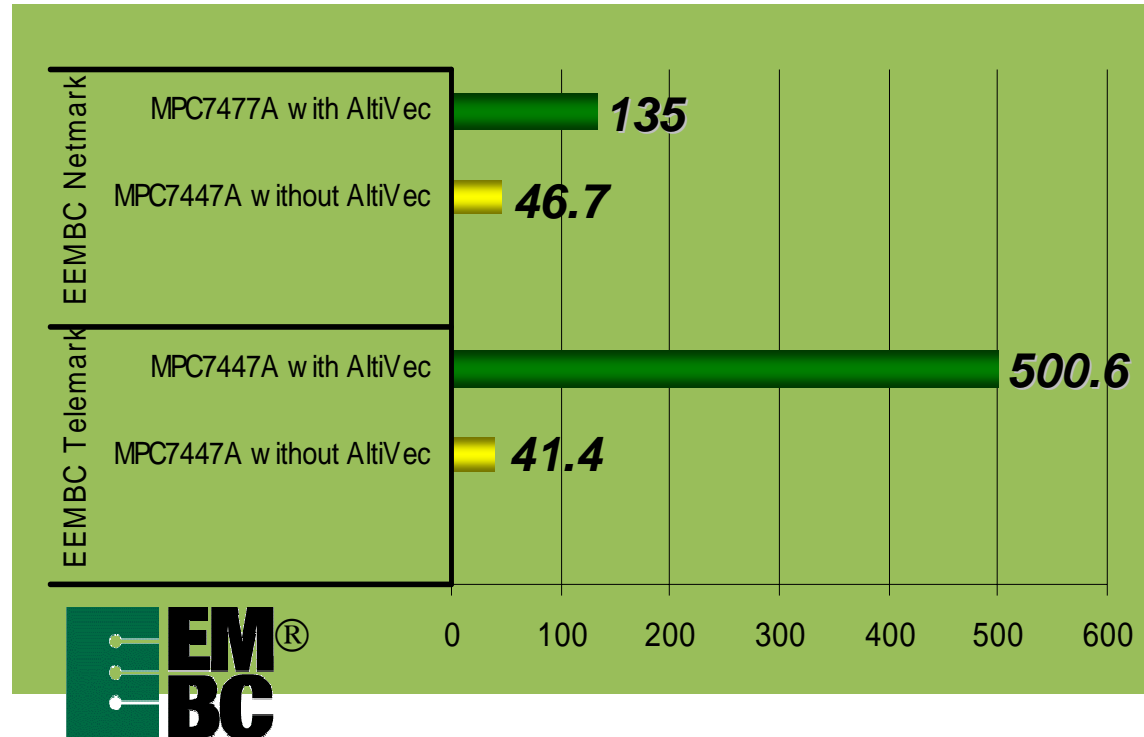
- AltiVecとは、PowerPCに実装された128ビット処理のテクノロジー全般
- AltiVec命令は、PowerPCアーキテクチャの拡張命令セットであり、128ビット処理を可能にする
- 162個のAltiVec命令は、組み込み関数を用いてC言語で使用可能
- 32個のAltiVec専用レジスタ・ファイル
- AltiVec命令に関する資料：[ALTIVECPIM](http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC7447A)(AltiVec Programming Interface Manual)
http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC7447A



ベクトル演算ユニット(AltiVec)のブロック図

AltiVecの優位性とライブラリ

- 2003年には、アプリケーションに特化した25個の関数が追加された
例: MPEG2/4, CRC (8, 12, 16, 24), MD5, Viterbi Decoderなど
- AltiVecテクノロジーは、ネットワーク、テレコム、イメージング、マルチメディアといったあらゆる市場で活用されている
- お客様のリクエストに基づき追加を継続中



memcpyとmemsetをライブラリの置き換えで高速化

LibCと名づけられたライブラリは、memcpyやmemset等の標準的な関数を最適化したもの。

LibCは、以下のサイトから入手可能。

<http://www.freescale.com/webapp/sps/site/overview.jsp?nodeId=018rH3bTdGmKqW5Nf2d9nb>

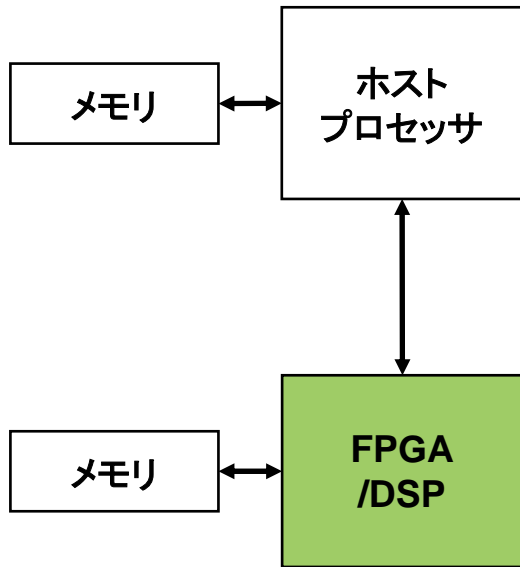
-使用方法

リンク時(Make時)に、"libmotoVec.a"が標準ライブラリより前に指定されるようにしておけば、プログラム内容を一切変更することなくAltiVecを活用可能

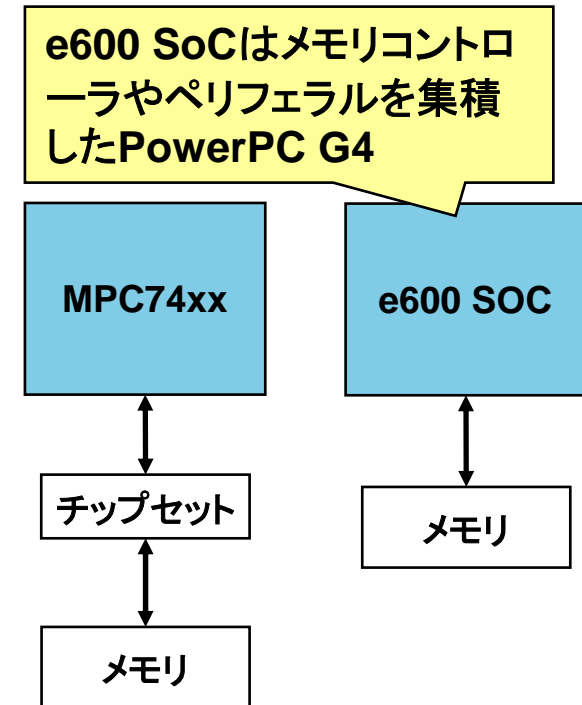
```
ld dhry21a.o dhry21b.o -l c:/sw/libmoto/libmotoVec.a -l c:/gcc/lib/gcc-lib/powerpc-eabisim/2.95.2/libgcc.a -o vec_dhry21.elf
```

PowerPC G4で実装面積を小さくする

- 外付けデバイスに任せていた信号処理を内部で完結できる
- デバイス間データ転送のボトルネックを解消
- 部品点数を従来より少なくし実装面積を抑えることが可能



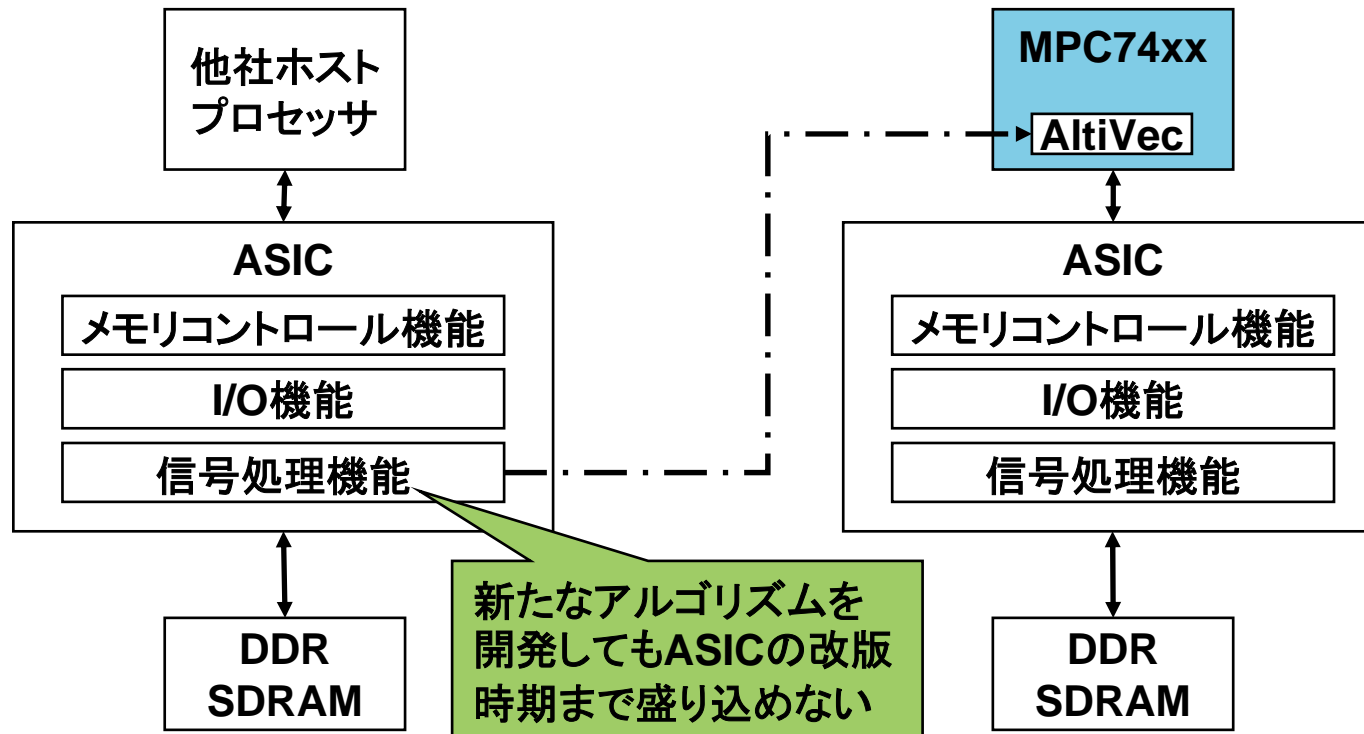
FPGAまたはDSPを使ったシステム



PowerPC G4を使ったシステム

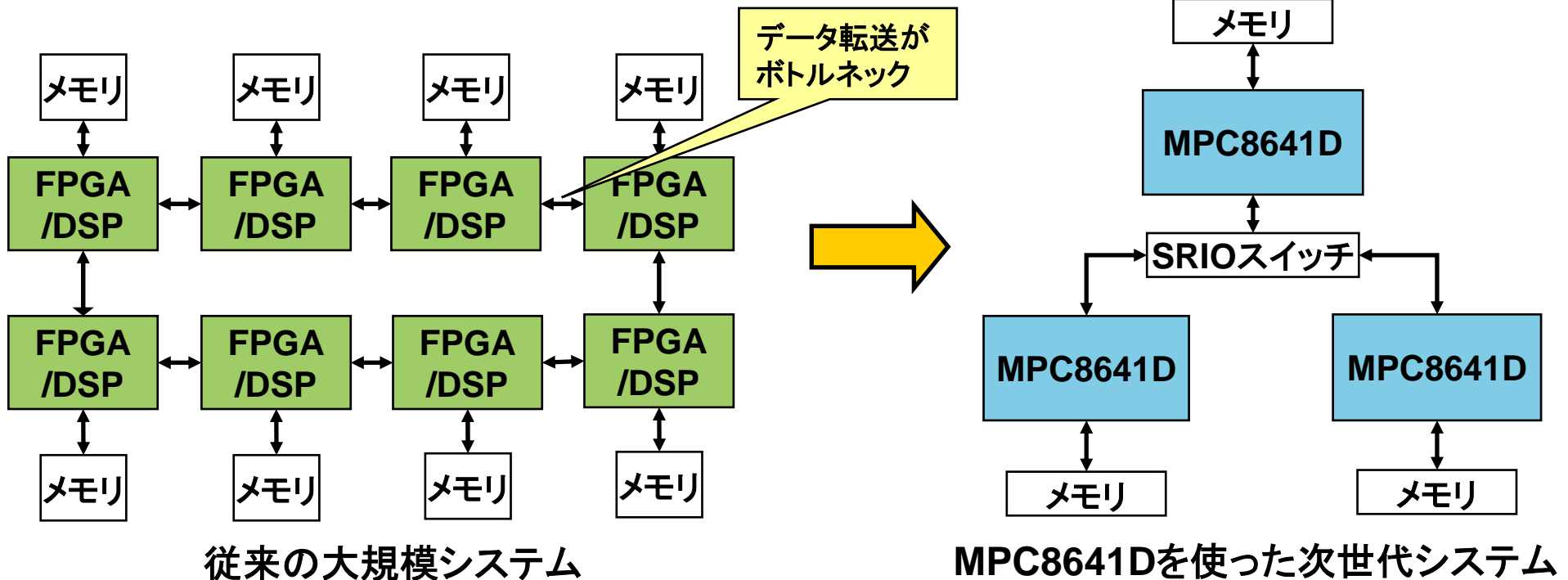
PowerPC G4で時間を手に入れる

- 負荷の重い処理の実現は、
 - 変更の可能性が無い標準的な処理 → ASIC
 - 変更の可能性のある処理 → AltiVecを活用しソフトウェアで実現
- ⇒新機能をASICの改版まで待たなくても製品に盛り込むことが可能

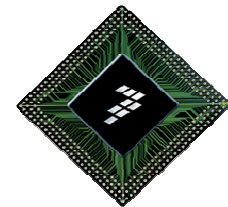


e600 SOCでシステムの拡張性を手に入れる

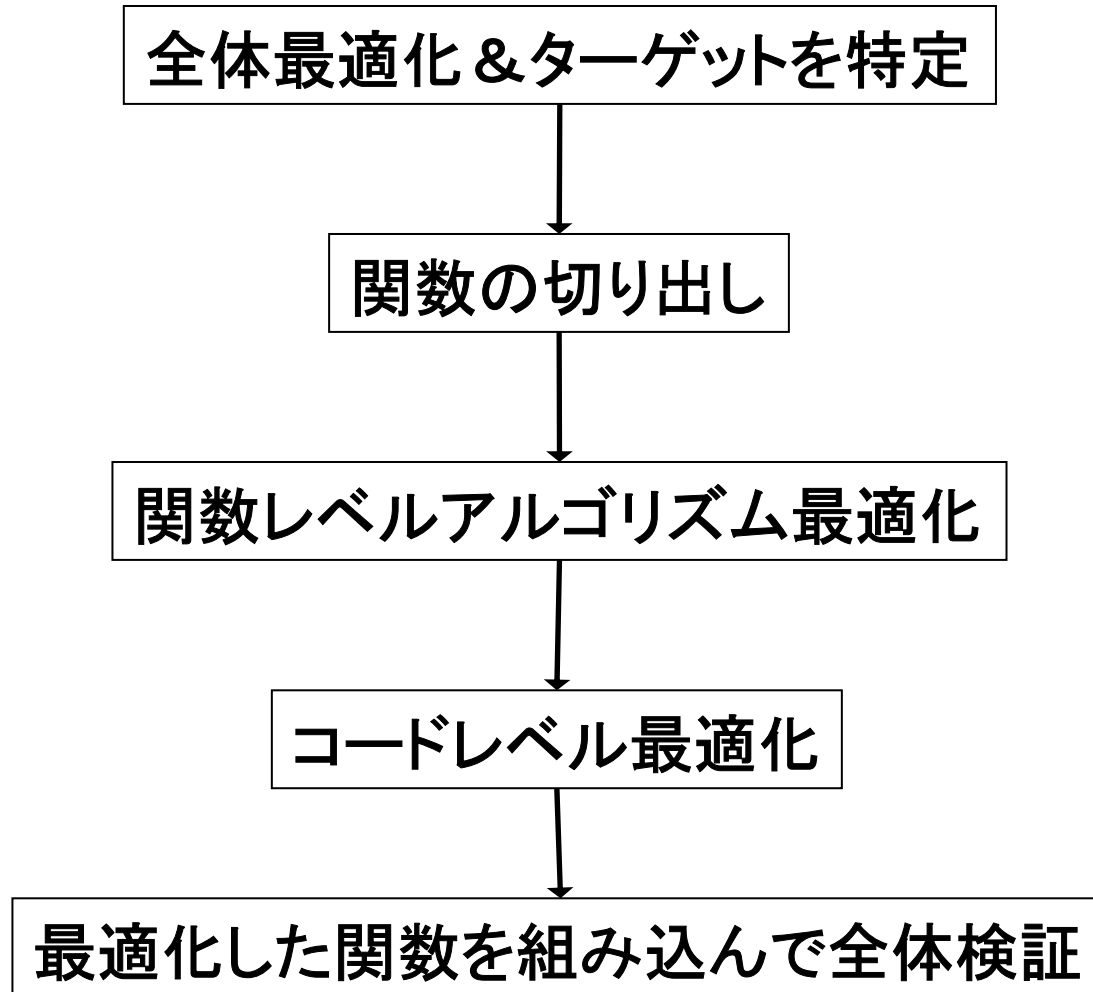
- 最新のe600(G4コア)を2個内臓したSoCタイプのMPC8641Dは、667MHzのDDR2やプロセッサ間通信を高速に行うSerial RapidIO (SRIO) をオンチップに搭載
- e600 SoCソリューションは、高い信号処理性能を持つコアと高速I/Oを備えるため、ミッド・レンジから超ハイエンドの性能要求まで満たす構成が取れる



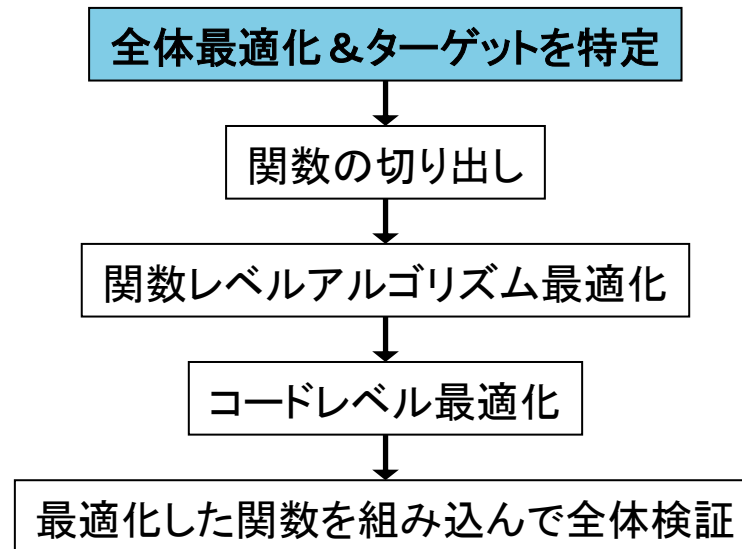
最適化プログラミングの基本



プログラムの最適化プロセス

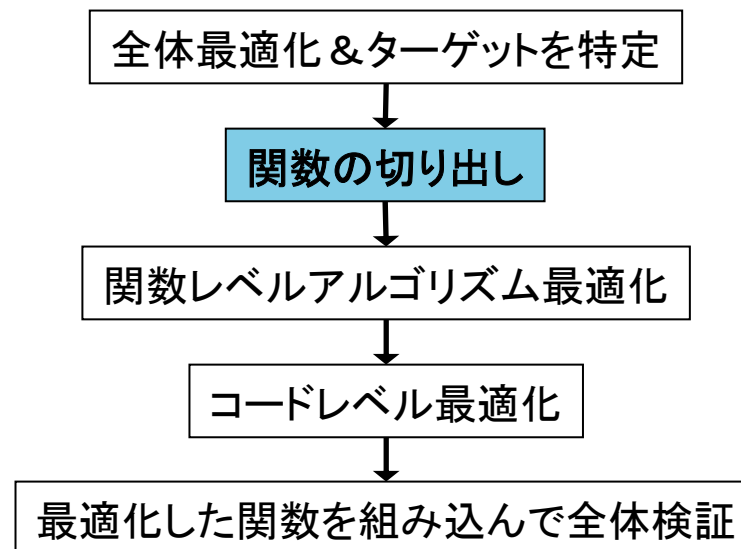


- プログラム全般的にオーバヘッドとなっている部分を改善する
- 特定プログラムに対する最適化は、全体の中で最も処理時間のかかっている処理を見極め、限定して行う

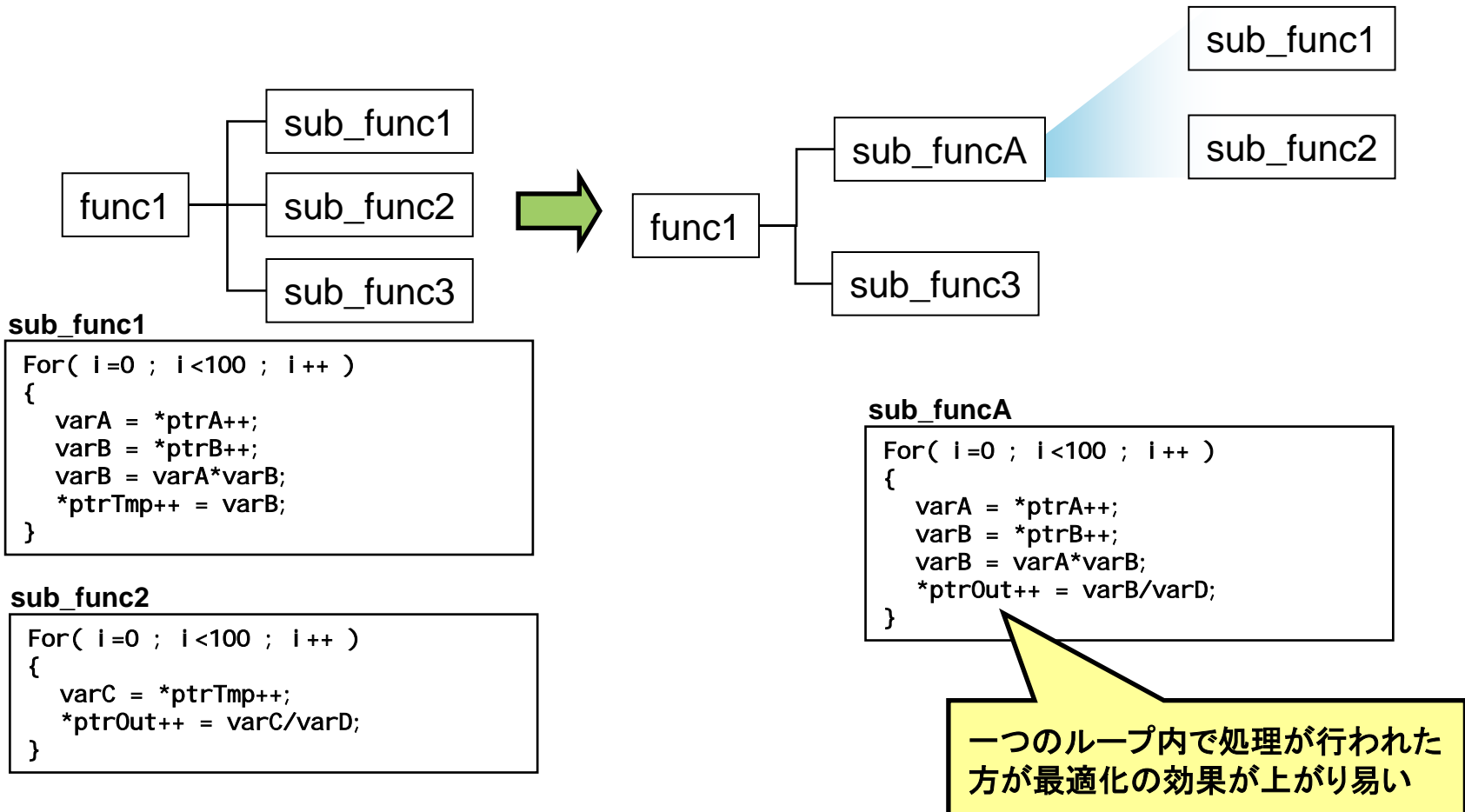


- **ハードウェア設定で改善可能な要素**
 - キャッシュがONになっていることを確認
 - 動的分岐予測はONになっていることを確認
 - ページ・アドレス変換ではなく、ブロックアドレス変換を使用していることを確認
 - ブロックアドレス変換設定時に、可能な限りライトバックモードを選択
 - マルチCPUで共有していないメモリ空間については、MMUの設定でM=0(コヒーレンシ非強制)に設定しておく
- **ソフトウェアで改善可能な要素**
 - 効率の良いコンパイラを使用する
 - 性能を重視するのであれば、C++でなくCを使用する
 - 関数のインライン展開を行って関数呼び出しのオーバーヘッドを抑える
 - Volatile修飾されている変数は、コンパイラによる最適化を妨げるので、メモリマップされた変数に対するアクセスは別関数に分けることで対応する

- 大量データをまとめて処理した方が最適化の効果が高いため、ループ処理は、ターゲット関数側で行う
- ターゲット関数は、記述されているファイルを別にして置き換え易くする

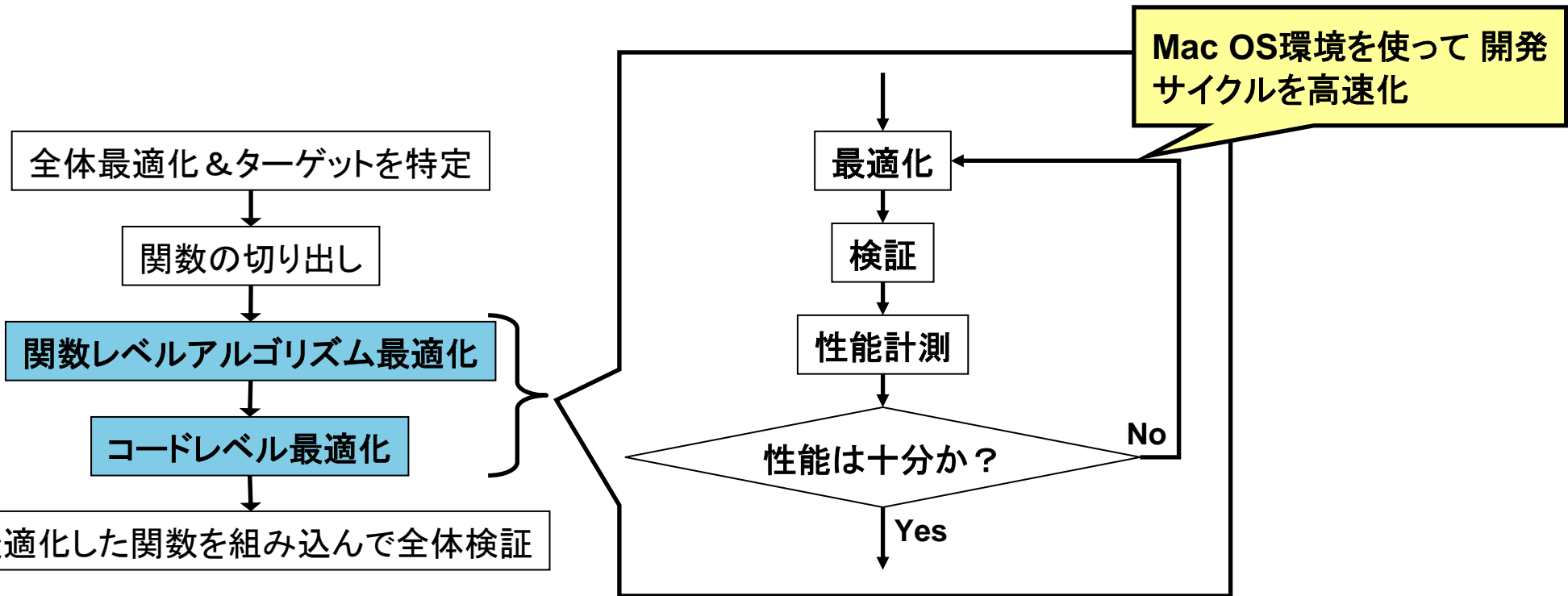


- 分かれている関数をまとめることで最適化の余地を広げる
- 最適化プログラミング後の最終形をイメージして、最適化な範囲で区切る



Mac OSを使って最適化プロセスを高速化

- 最適化コーディングは、PowerPC G4が搭載されているマックを活用することで効率良く行うことが可能



Mac OS環境を使った開発手順

- ①ファイル単位で切り出したターゲット関数をマックに移す
- ②ターゲット関数の単体検証環境を用意する
- ③単体検証と計測を同時に行いながらターゲット関数の最適化を十分に行う
- ④最適化が施された関数を、ターゲット・システムの関数と置き換えて全体検証

```
main()
{
    (入力データ生成)

    (時間計測処理-開始)
    target_func();
    (時間計測処理-終了)

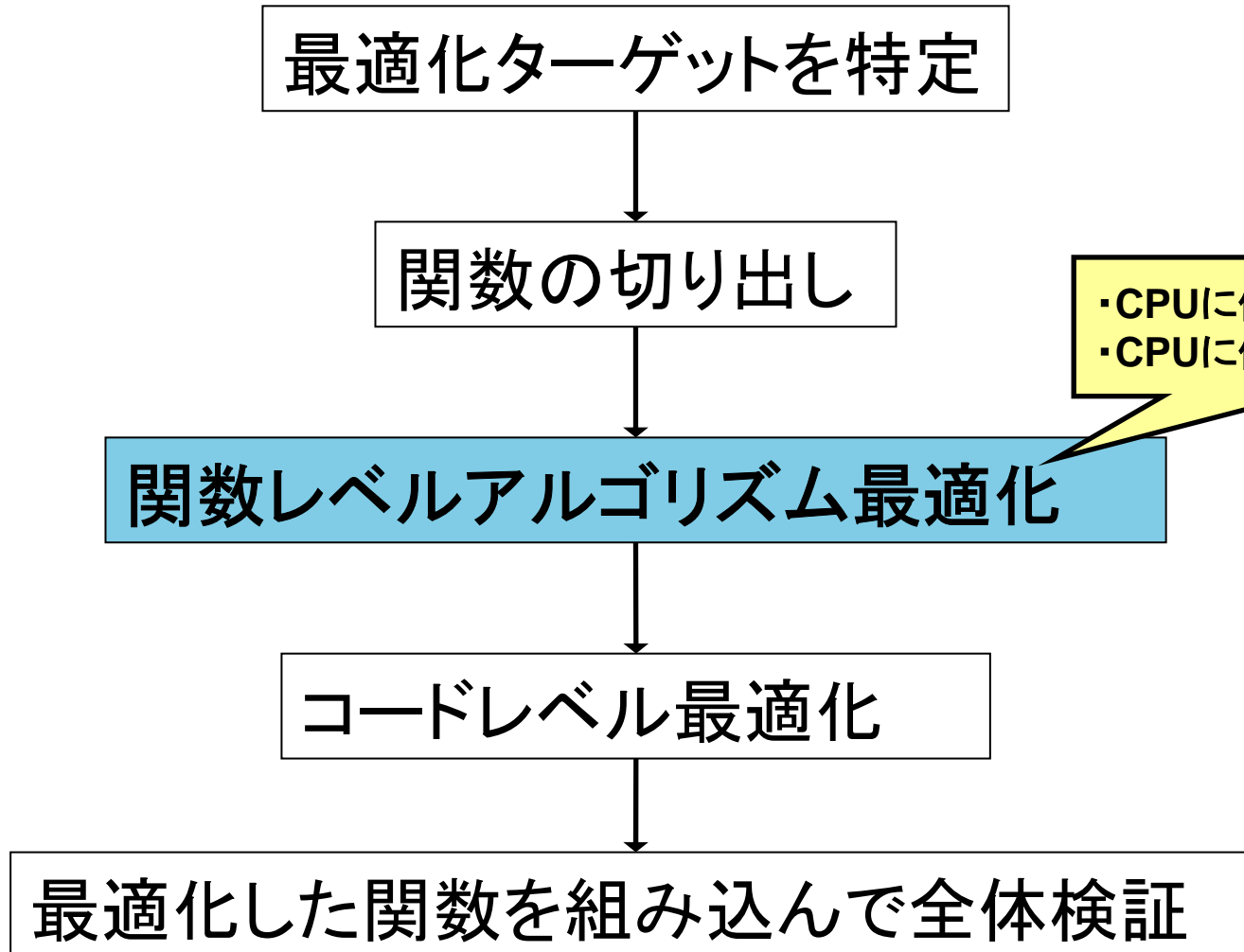
    (時間計測処理-開始)
    target_func_opt();
    (時間計測処理-終了)

    (検証処理)
}
```

段階的に変更を加えて性能を向上させる

- **コスト・メリット**
 - 開発ツール - 無償のXCODEが使用可能
 - ハードウェア - PowerPC G4が搭載されているマック
- **開発効率のメリット**
 - 検証プログラムの作成が容易 - 大規模なワーク領域を自由に使用可能
 - 高速にデバッグ可能 - 開発環境がインストールされているマシンとプログラムが動作するマシンが同一だから動作が速い
 - ユーザは最適化作業に集中できる

マシンを2台並べてステップ実行を行えば間違い探しも容易



- CPUに依存しない高速化
- 数学的に等価な別の処理方法に変えることで性能向上を図る

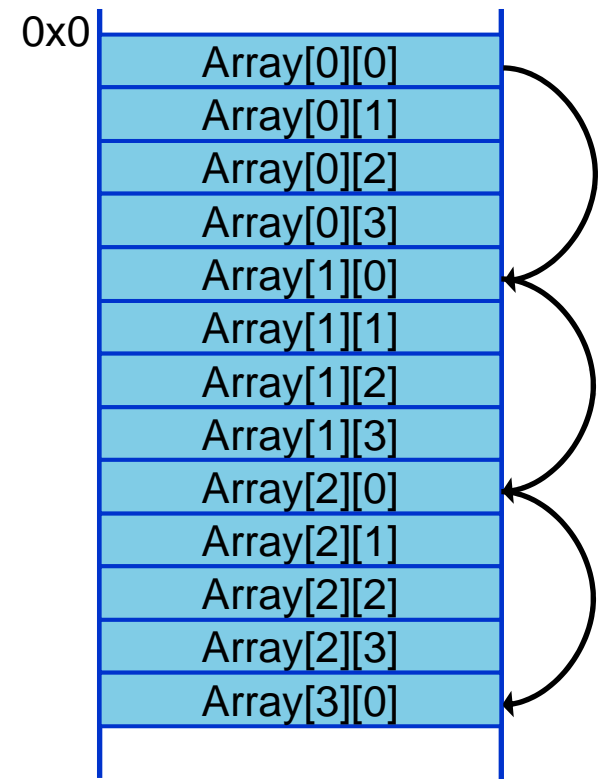
$$\begin{aligned}y(0) &= x(0) \\y(1) &= x(0) + x(1) \\y(2) &= x(0) + x(1) + x(2) \\y(3) &= x(0) + x(1) + x(2) + x(3) \\y(4) &= x(0) + x(1) + x(2) + x(3) + x(4) \\&\quad \text{(中略)} \\y(n) &= \sum X(n) \quad // (n-1)\text{回の加算処理が必要}\end{aligned}$$

$$\begin{aligned}y(0) &= x(0) \\y(1) &= y(0) + x(1) \\y(2) &= y(1) + x(2) \\y(3) &= y(2) + x(3) \\&\quad \text{(中略)} \\y(n) &= y(n-1) + x(n) \quad // 1\text{回の加算処理で実現}\end{aligned}$$

- CPUに依存した最適化
- 2次元配列を用いた次の例では、データのアクセスが連続せず効率が悪い
- 次頁で示す最適化を行いデータのアクセスを連続させることで、AltiVecの転送命令や外部メモリへのバースト転送と親和性が高くなる

```
Main()
{
    float Array[100][4];
}

int func(float Array[100][4])
{
    int i;
    float v1=0;
    for (i = 0; i<100; i++)
    {
        v1 += Array[i][0];
    }
    return (v1);
}
```



- 効率の良い連続アクセスは、変更した配列を用いることで実現可能

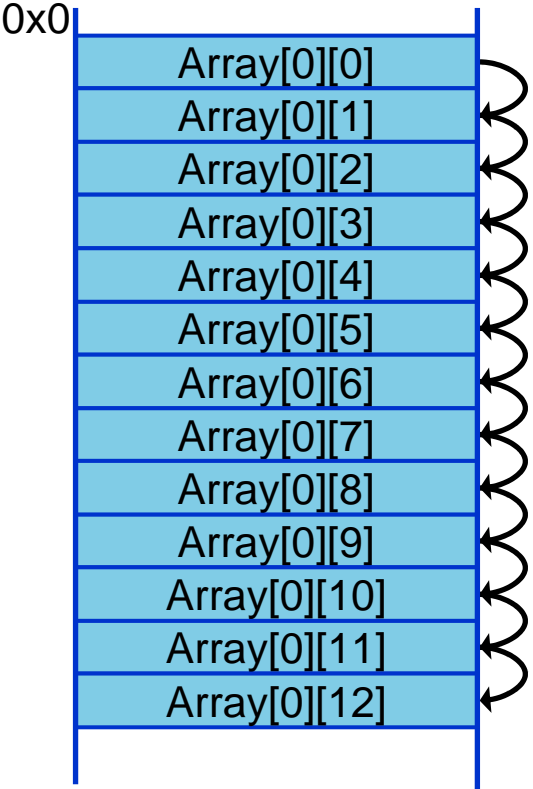
```
Main()
{
    float Array[100][4];
}
```



```
Main()
{
    float Array[4][100];
}
```

```
int func(float Array [][])
{
    int i;
    float v1=0;
    for (i = 0; i<=100; i++)
    {
        v1 += Array [0][i];
    }
    return (v1);
}
```

アルゴリズム最適化



アルゴリズム最適化

- CPUに依存した最適化
- アプリケーションは、許容できる最小の精度で演算することで高速化の余地が広がる

演算精度、高

並列度、高

倍精度浮動小数点
double

- 浮動小数点ユニットで実現可能
- SIMD処理は不可

単精度浮動小数点
float

- 浮動小数点ユニットで実現可能
- AltiVecで実現可能
- 最大4並列処理が可能

16ビット整数型
short
unsigned short

- 整数ユニットで実現可能
- AltiVecで実現可能
- 最大8並列処理が可能

8ビット整数型
char
unsigned char

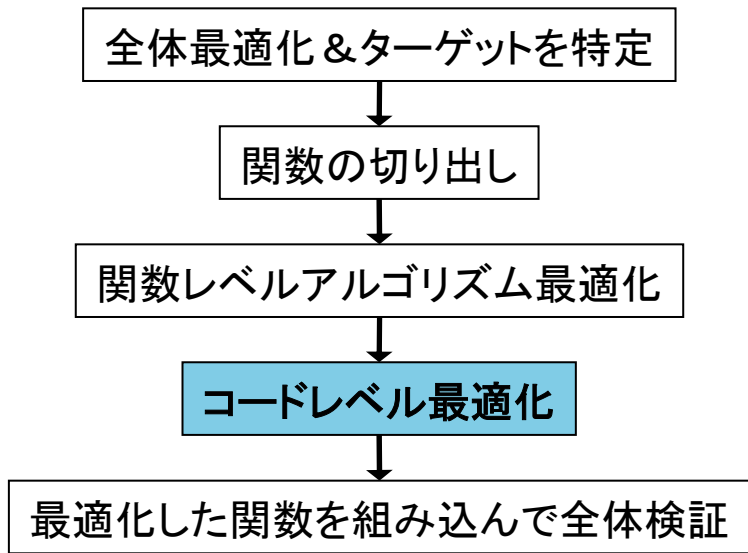
- 整数ユニットで実現可能
- AltiVecで実現可能
- 最大16並列処理が可能

- `mem_cpy`関数は冗長な転送処理であることが多いため、なるべく `mem_cpy`関数を使わずに済む処理フローで実現する
- ループ処理は、なるべく大きなデータ単位に対して行う
- ただし、キャッシュからあふれてしまうほどの大きいデータ単位は、メモリアクセスがオーバヘッドとして現れてくるため、処理単位をキャッシュに収まる適正なデータサイズに区切る

アルゴリズム最適化

コードレベルの最適化は、次の3点を追求する最適化手法

- 最小の命令発行数で処理 → **コードレベル最適化①**
- 効率が良いデータ・フロー → **コードレベル最適化②**
- 命令発行の最大スループット → **コードレベル最適化③**



- AltiVecとして実装されている128ビットのSIMD(Single Instruction Multiple Data stream)命令を用いることで少ない命令数で等価な処理の実現が可能
- AltiVecの主要な命令
 - ベクタ転送命令 `vec_ld()`, `vec_st()`
 - 算術命令 `vec_add()`, `vec_sub()`, `vec_madd()`
 - パミュート命令 `vec_perm()`

コードレベル最適化①

AltiVec命令をC言語フォーマットで使うために組み込み関数を使用する

AltiVecのベクタ転送命令

- 128ビット長の変数は、ベクタ型で変数を定義することで使用可能
- 転送の高速化は、AltiVec命令を用い128ビット転送を行うことで可能
- この性能比較は、それぞれL1キャッシュアクセスが前提

コードレベル最適化①

```
unsigned long *tmppIn, *tmppOut;
tmppIn = (unsigned long*)InputBuf;
tmppOut = (unsigned long *)OutputBuf;
for(i=0;i<2000;i++)
{
    *tmppOut++ = *tmppIn++;
}
```

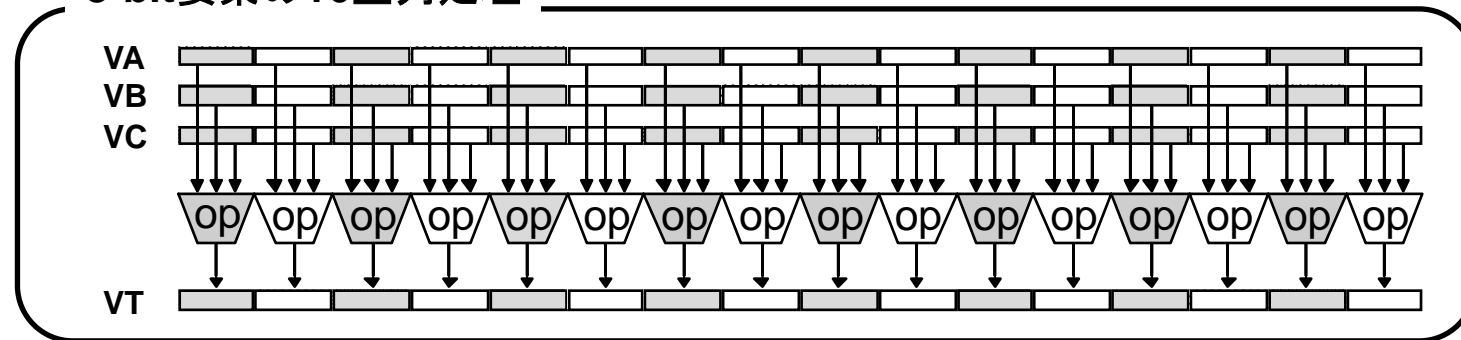
時間:1508


AltiVecの
転送命令
で最適化

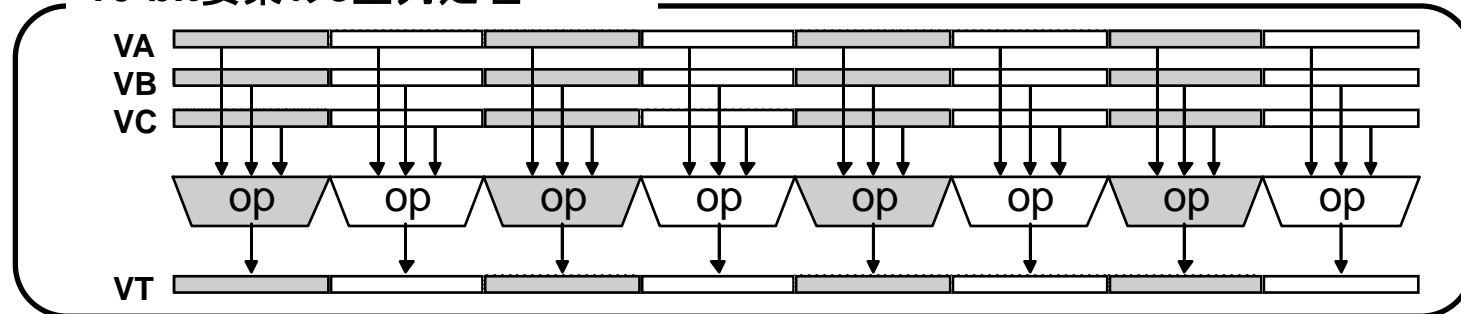
```
vector unsigned char t0;
ptrIn = (unsigned char*)InputBuf;
ptrOut = (unsigned char*)OutputBuf;
for(i=0;i<500;i++)
{
    t0 = vec_ld(0, ptrIn);
    ptrIn += 16;
    vec_st( t0, 0, ptrOut);
    ptrOut += 16;
}
```

時間:380

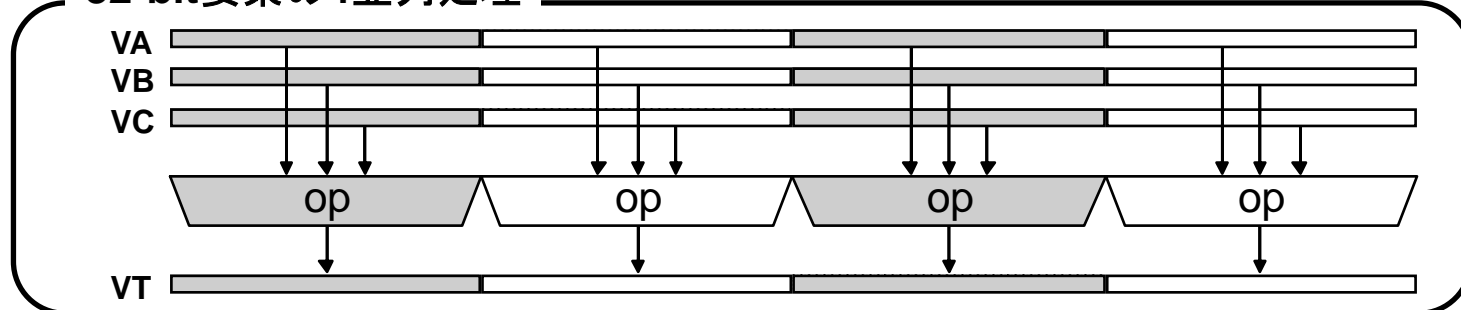
8-bit要素の16並列処理



16-bit要素の8並列処理



32-bit要素の4並列処理

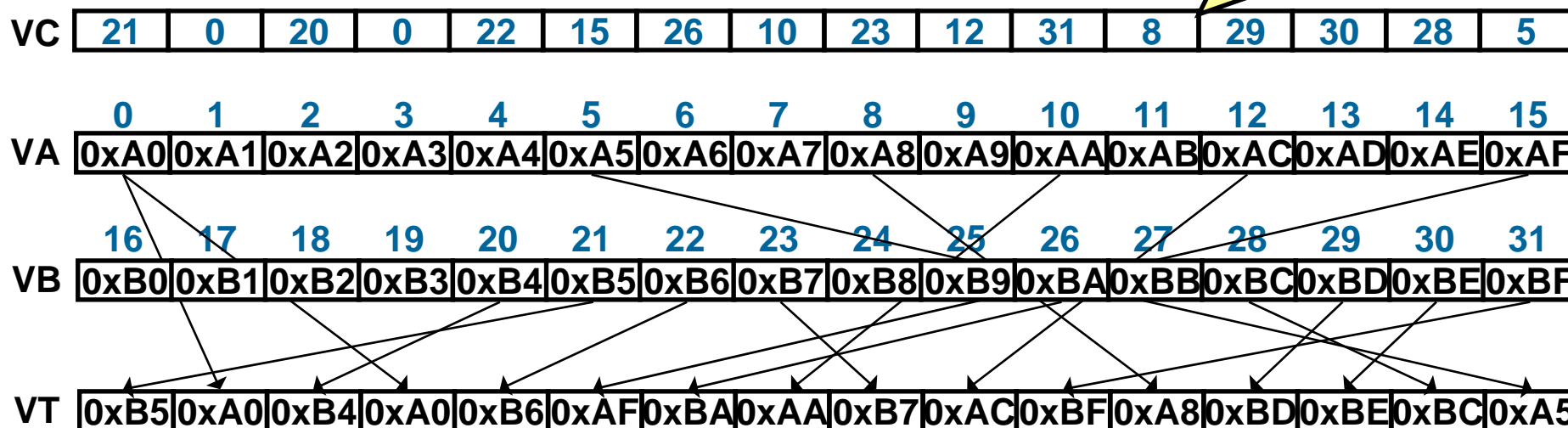


- パミュート命令は、パミュート・ユニットを利用して2つの128ビットデータから任意のデータを選択してバイト単位の並べ替えを行う

コードレベル最適化①

VTに格納するデータを指定するインデックス

VT = vec_perm(VA,VB,VC)



パミュート・ユニットの活用例 ~ RGBからYCbCrへの変換

コードレベル最適化①

8bit RGB データ x 16

R ₁	G ₁	B ₁	R ₂	G ₂	B ₂	R ₃	G ₃	B ₃	R ₄	G ₄	B ₄	R ₅	G ₅	B ₅	R ₆
G ₆	B ₆	R ₇	G ₇	B ₇	R ₈	G ₈	B ₈	R ₉	G ₉	B ₉	R ₁₀	G ₁₀	B ₁₀	R ₁₁	G ₁₁
B ₁₁	R ₁₂	G ₁₂	B ₁₂	R ₁₃	G ₁₃	B ₁₃	R ₁₄	G ₁₄	B ₁₄	R ₁₅	G ₁₅	B ₁₅	R ₁₆	G ₁₆	B ₁₆

$$Y_n = \frac{8432}{32768} (R_n) + \frac{16425}{32768} (G_n) + \frac{3176}{32768} (B_n) + 16$$

$$Cb_n = \frac{-4818}{32768} (R_n) + \frac{-9527}{32768} (G_n) + \frac{14345}{32768} (B_n) + 128$$

$$Cr_n = \frac{14345}{32768} (R_n) + \frac{-12045}{32768} (G_n) + \frac{2300}{32768} (B_n) + 128$$

Y ₁	Y ₂	Y ₃	Y ₄	Y ₅	Y ₆	Y ₇	Y ₈	Y ₉	Y ₁₀	Y ₁₁	Y ₁₂	Y ₁₃	Y ₁₄	Y ₁₅	Y ₁₆
Cb ₁	Cb ₂	Cb ₃	Cb ₄	Cb ₅	Cb ₆	Cb ₇	Cb ₈	Cb ₉	Cb ₁₀	Cb ₁₁	Cb ₁₂	Cb ₁₃	Cb ₁₄	Cb ₁₅	Cb ₁₆
Cr ₁	Cr ₂	Cr ₃	Cr ₄	Cr ₅	Cr ₆	Cr ₇	Cr ₈	Cr ₉	Cr ₁₀	Cr ₁₁	Cr ₁₂	Cr ₁₃	Cr ₁₄	Cr ₁₅	Cr ₁₆

8bit YCbCr data x 16

色変換処理の過程でパミュート命令を活用

v0	R1	G1	B1	R2	G2	B2	R3	G3	B3	R4	G4	B4	R5	G5	B5	R6
v1	G6	B6	R7	G7	B7	R8	G8	B8	R9	G9	B9	R10	G10	B10	R11	G11
v2	B11	R12	G12	B12	R13	G13	B13	R14	G14	B14	R15	G15	B15	R16	G16	B16

パミュート命令(vec_perm)のためのインデックス情報

c0	0	3	6	9	12	15	18	21	1	4	7	10	13	16	19	22
c1	2	5	8	11	14	17	20	23	x	x	x	x	x	x	x	x
c2	8	11	14	17	20	23	26	29	9	12	15	18	21	24	27	30
c3	10	13	16	19	22	25	28	31	x	x	x	x	x	x	x	x

```

d0 = vec_perm(v0,v1,c0);
d1 = vec_perm(v0,v1,c1);
d2 = vec_perm(v1,v2,c2);
d3 = vec_perm(v1,v2,c3);
    
```

色種別ごとにまとめてレジスタへ格納

d0	R1	R2	R3	R4	R5	R6	R7	R8	G1	G2	G3	G4	G5	G6	G7	G8
d1	B1	B2	B3	B4	B5	B6	B7	B8								
d2	R9	R10	R11	R12	R13	R14	R15	R16	G9	G10	G11	G12	G13	G14	G15	G16
d3	B9	B10	B11	B12	B13	B14	B15	B16								

vec_unpackh()

R1	R2	R3	R4	R5	R6	R7	R8
----	----	----	----	----	----	----	----

サンプルコード:

<http://www.freescale.com/webapp/sps/site/overview.jsp?nodeId=02VS0I81285Nf2F9DHMbVXVDcM>

Altivecを使った最適化例(続き) ~ RGBからYCbCrへの変換

コードレベル最適化①

1つのベクタレジスタに格納された8個の要素それぞれに同一の係数を掛けてゆく

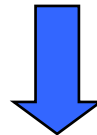
	R ₁	R ₂	R ₃	R ₄	R ₅	R ₆	R ₇	R ₈
	X	X	X	X	X	X	X	X
k0	8434	8434	8434	8434	8434	8434	8434	8434
k1	16425	16425	16425	16425	16425	16425	16425	16425
k2	3176	3176	3176	3176	3176	3176	3176	3176
	-4818	-4818	-4818	-4818	-4818	-4818	-4818	-4818

この係数データを用意する時に効率の良いコーディング方法を以下に示します

最適化前

```
k0 = (vector signed short)(8434, 8434, 8434, 8434, 8434, 8434, 8434, 8434);  
k1 = (vector signed short)(16425, 16425, 16425, 16425, 16425, 16425, 16425, 16425);  
k2 = (vector signed short)(3176, 3176, 3176, 3176, 3176, 3176, 3176, 3176);
```

それぞれの代入でメモリ・アクセスが発生



最適化後

```
vector signed short Special Constants =  
(vector signed short)(8432, 16425, 3176, -4818, -9527, 14345, 0, 0);  
k0 = vec_splat(Special Constants, 0); /* 8432を複製 */  
k1 = vec_splat(Special Constants, 1); /* 16425を複製 */  
k2 = vec_splat(Special Constants, 2); /* 3176を複製 */
```

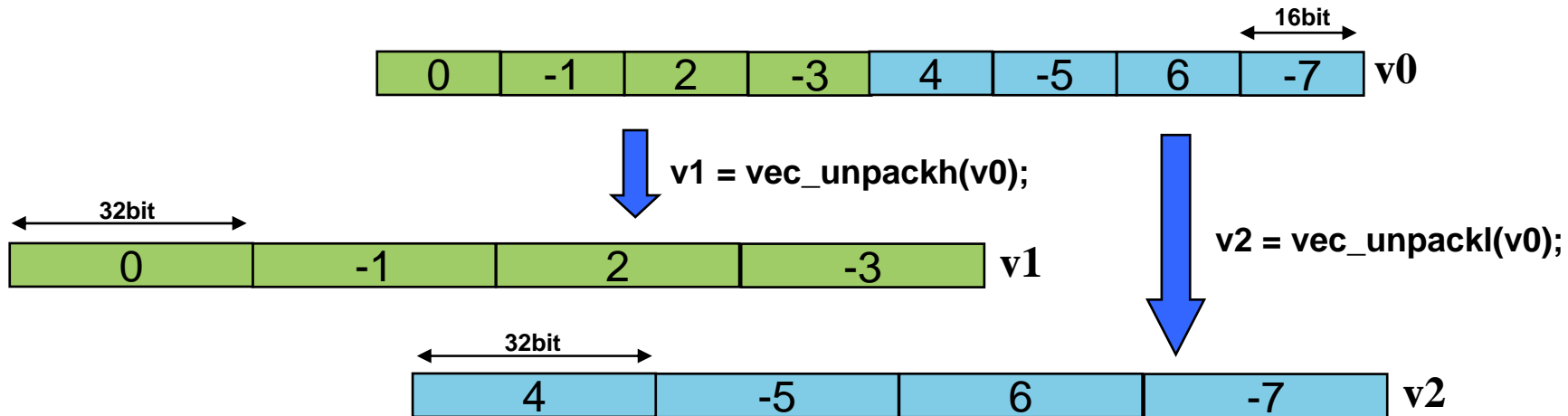
コードレベル最適化②

ベクタ・パミュート・ユニットで実現されるため、メモリ・アクセスを伴わない

データサイズの変更(符号付の場合)

コードレベル最適化①

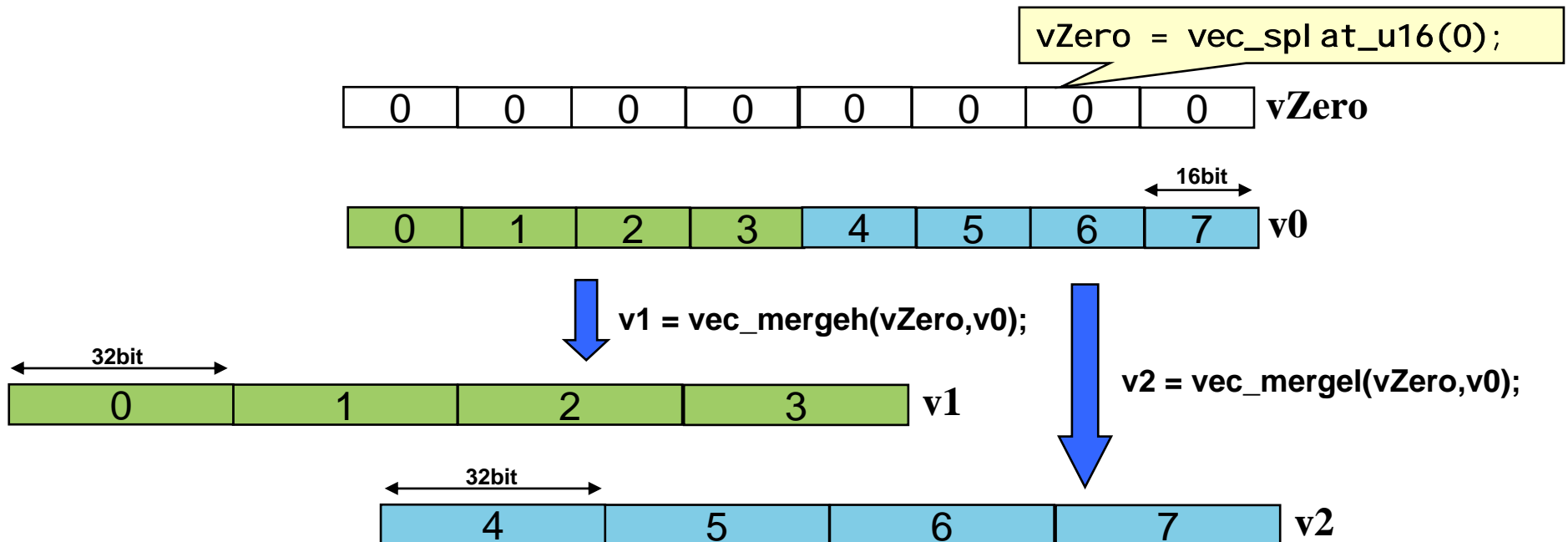
- 符号付き16ビットデータを、符号付き32ビットに拡張する場合は、以下の命令を用いる
- 符号付き8ビットデータを、符号付き16ビットデータに拡張する時にも使用可能



データサイズの変更(符号無しの場合)

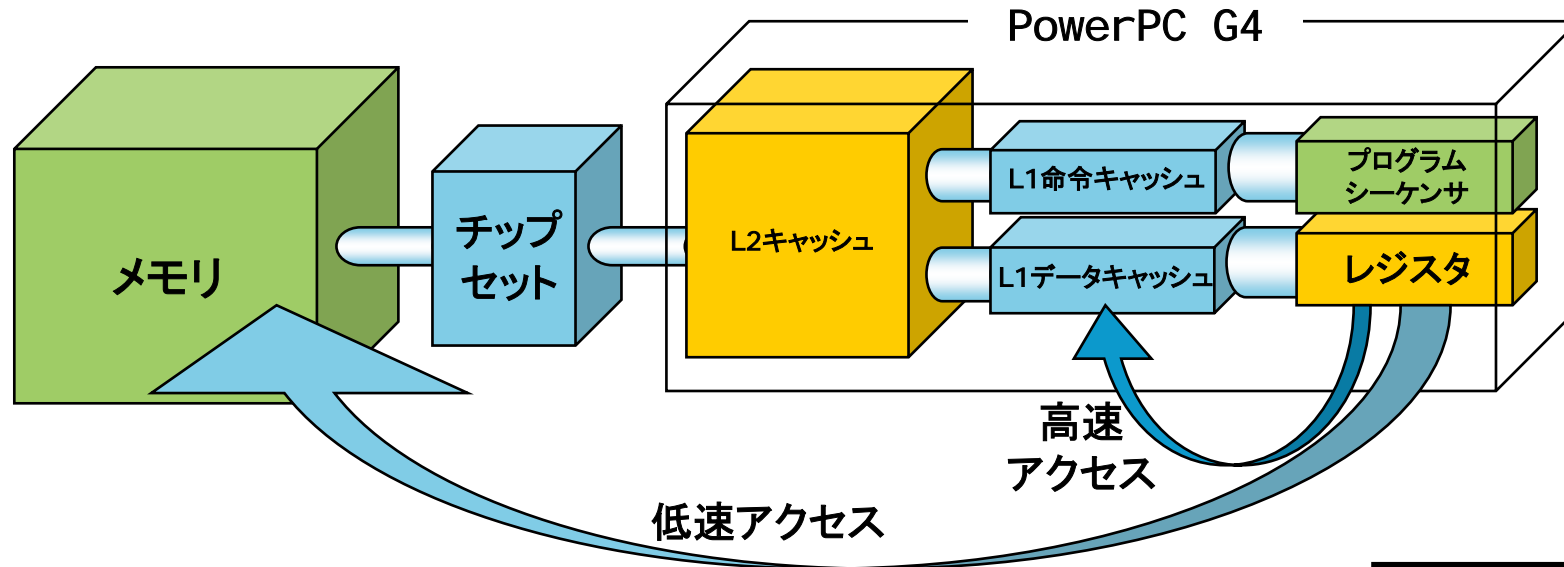
コードレベル最適化①

- 符号無し16ビットデータを、符号付き/無し32ビットに拡張する場合は、以下の命令を用いる
- 符号無し8ビットデータを、符号付き/無し16ビットデータに拡張する時にも使用可能



効率が良いデータ・フローの実現方法

- 転送にかかる処理時間は、同じロード命令が発行された場合でも、対象データが展開されている場所によって変化する
- 外部メモリからキャッシュへのデータ転送は、コア内部の処理とは独立に実行可能
- 効率の良いデータ・フローを実現するためには、必要なデータ量が、必要な時に、必要な期間だけキャッシュに留まるよう制御する



コードレベル最適化②

効率が良いデータ・フローの必要性

- **AltiVec**命令で高速したつもりのプログラムでも外部メモリアクセスを行っている場合はコアが空回りする
- 最適化によりコアの処理が高速になる程、外部メモリへのアクセスが遅いためストールが問題になる。

```
vector unsigned char t0;
ptrIn = (unsigned char*)InputBuf;
ptrOut = (unsigned char*)OutputBuf;
for(i=0;i<500;i++)
{
    t0 = vec_ld(0, ptrIn);
    ptrIn += 16;
    vec_st(t0, 0, ptrOut);
    ptrOut += 16;
}
```

L1キャッシュからロード

時間: 380

コアサイクルに比例した値(目安)

```
vector unsigned char t0;
ptrIn = (unsigned char*)InputBuf;
ptrOut = (unsigned char*)OutputBuf;
for(i=0;i<500;i++)
{
    t0 = vec_ld(0, ptrIn);
    ptrIn += 16;
    vec_st(t0, 0, ptrOut);
    ptrOut += 16;
}
```

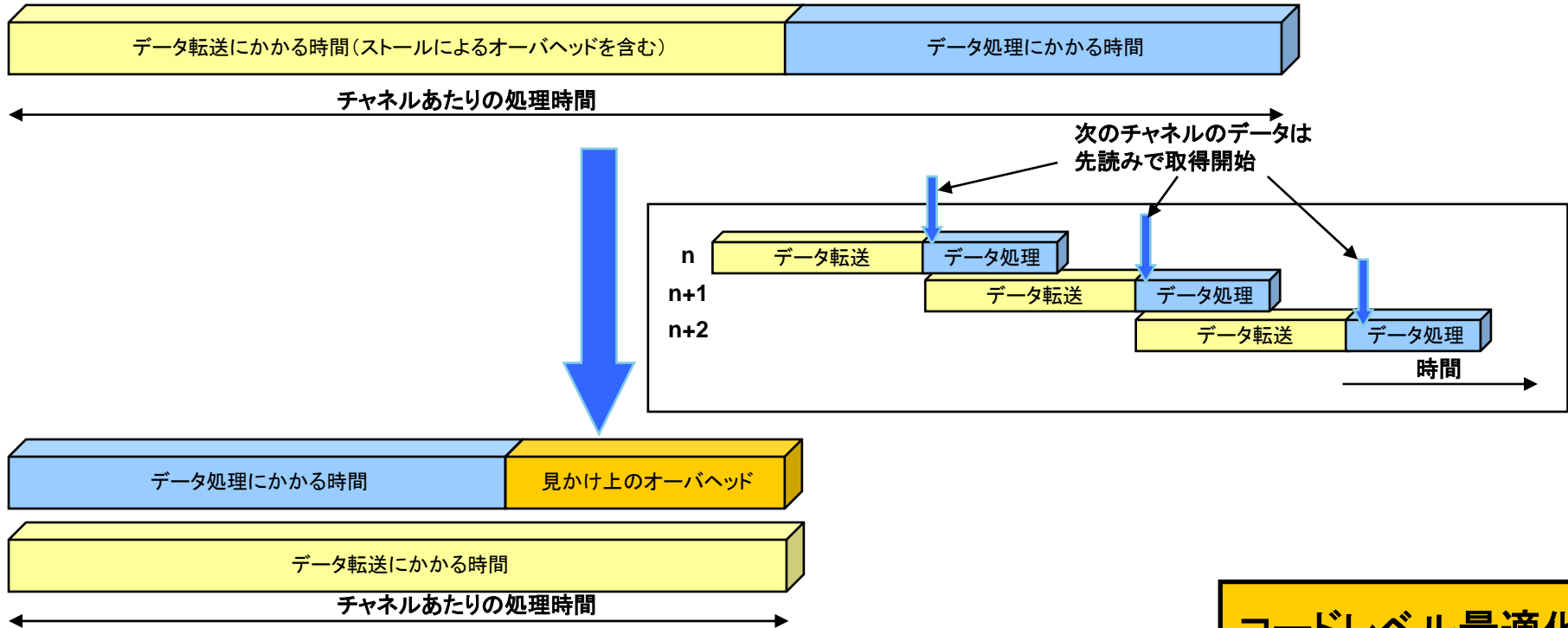
外部メモリからロード

時間: 8056

コードレベル最適化②

効率の良いデータアクセスを実現手法

- L1キャッシュに対するデータ・アクセスは、スループット1サイクルで実行可能
- L2キャッシュ、外部メモリアクセスは、スループットが複数サイクルになりストール要因となる
- コアがデータ処理をしている間にあらかじめデータの出し入れを行なわせることで、ストールに伴うオーバーヘッドを見かけ上、極小化することが可能



コードレベル最適化②

キャッシュ制御命令のための32バイト処理

- ディスパッチ・ユニットが、ロード命令を早めに発行するように制御するが性能向上には限界がある
- 効率の良いデータの先読みを実現するためにはキャッシュ制御命令を活用
- キャッシュラインは32バイト単位で構成されているため、キャッシュ制御命令の有効範囲を考慮し、ループ中は32バイト単位でデータアクセスさせる

```
vector unsigned char t0;
ptrIn = (unsigned char*)InputBuf;
ptrOut = (unsigned char*)OutputBuf;
for(i=0;i<500;i++)
{
    t0 = vec_ld(0, ptrIn);
    ptrIn += 16;
    vec_st(t0, 0, ptrOut);
    ptrOut += 16;
}
```

時間: 8056



ループ展開で
32バイトずつ
処理を行う

```
vector unsigned char t0,t1;
ptrIn = (unsigned char*)InputBuf;
ptrOut = (unsigned char*)OutputBuf;
for(i=0;i<250;i++)
{
    t0 = vec_ld(0, ptrIn);
    t1 = vec_ld(16, ptrIn);
    ptrIn += 32;
    vec_st(t0, 0, ptrOut);
    vec_st(t1, 16, ptrOut);
    ptrOut += 32;
}
```

時間: 8056

コードレベル最適化②

dcbt命令を使ったデータ先読み

- **dcbt(データ・キャッシュ・ブロック・タッチ)**命令を用いることで指定されたアドレスが含まれる32バイトが事前にL1データ・キャッシュに転送される

```
void __dcbt(void *, int);  
vector unsigned char t0,t1;  
ptrIn = (unsigned char*)InputBuf;  
ptrOut = (unsigned char*)OutputBuf;  
for(i=0;i<250;i++)  
{  
    __dcbt(ptrIn, 32);  
    t0 = vec_ld(0, ptrIn);  
    t1 = vec_ld(16, ptrIn);  
    ptrIn += 32;  
    vec_st( t0, 0, ptrOut);  
    vec_st( t1, 16, ptrOut);  
    ptrOut += 32;  
}
```

Codewarriorに用意されている組み込み関数を用いてdcbt命令を使用

(ptrIn+32)番地から32バイトをL1キャッシュに転送開始

時間: 2504

コードレベル最適化②

dst命令を使ったブロック単位のデータ先読み

- PowerPC G4の拡張命令セットで用意されているdst(データ・ストリーム・タッチ)命令を用いることで任意のブロックサイズで先読みが可能
- dst命令をC言語で使う場合、vec_dst()のフォーマットで使用

```
vector unsigned char t0,t1;
ptrIn = (unsigned char*)InputBuf;
ptrOut = (unsigned char*)OutputBuf;
for(i=0;i<250;i++)
{
    __dcbt(ptrIn, 32);
    t0 = vec_ld(0, ptrIn);
    t1 = vec_ld(16, ptrIn);
    ptrIn += 32;
    vec_st( t0, 0, ptrOut);
    vec_st( t1, 16, ptrOut);
    ptrOut += 32;
}
```


dst命令で
ブロック先読み

```
vec_dst( ptrIn, 0x00100200, 0);
for(i=0;i<250;i++)
{
    t0 = vec_ld(0, ptrIn);
    t1 = vec_ld(16, ptrIn);
    ptrIn += 32;
    vec_st( t0, 0, ptrOut);
    vec_st( t1, 16, ptrOut);
    ptrOut += 32;
}
```

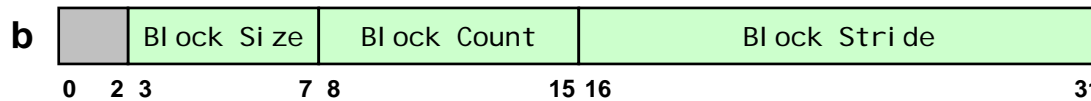
コードレベル最適化②

vec_dst()の命令フォーマット

コードレベル最適化②

- 広い範囲のデータ(最大128KB)を先読み指定することが可能
- 下図のように飛び飛びにメモリからキャッシュへデータを先読みさせることも可能
- 'Block Size'で指定する単位は、vectorであることに注意(1vectorは16バイト)

データ・ストリーム・タッチ命令 vec_dst(a, b, c)



Block Size = 1 – 32 (vector)
Block Count (n) = 1 – 256
Block Stride = 1 – 32768 (byte)
これらのパラメータは、最大値を入力する場合に限り、0を指定する。例えば'Block Size'に0を入力すると32(vector)が設定される。

データ・ストリーム・タッチ命令により先読みされるメモリ

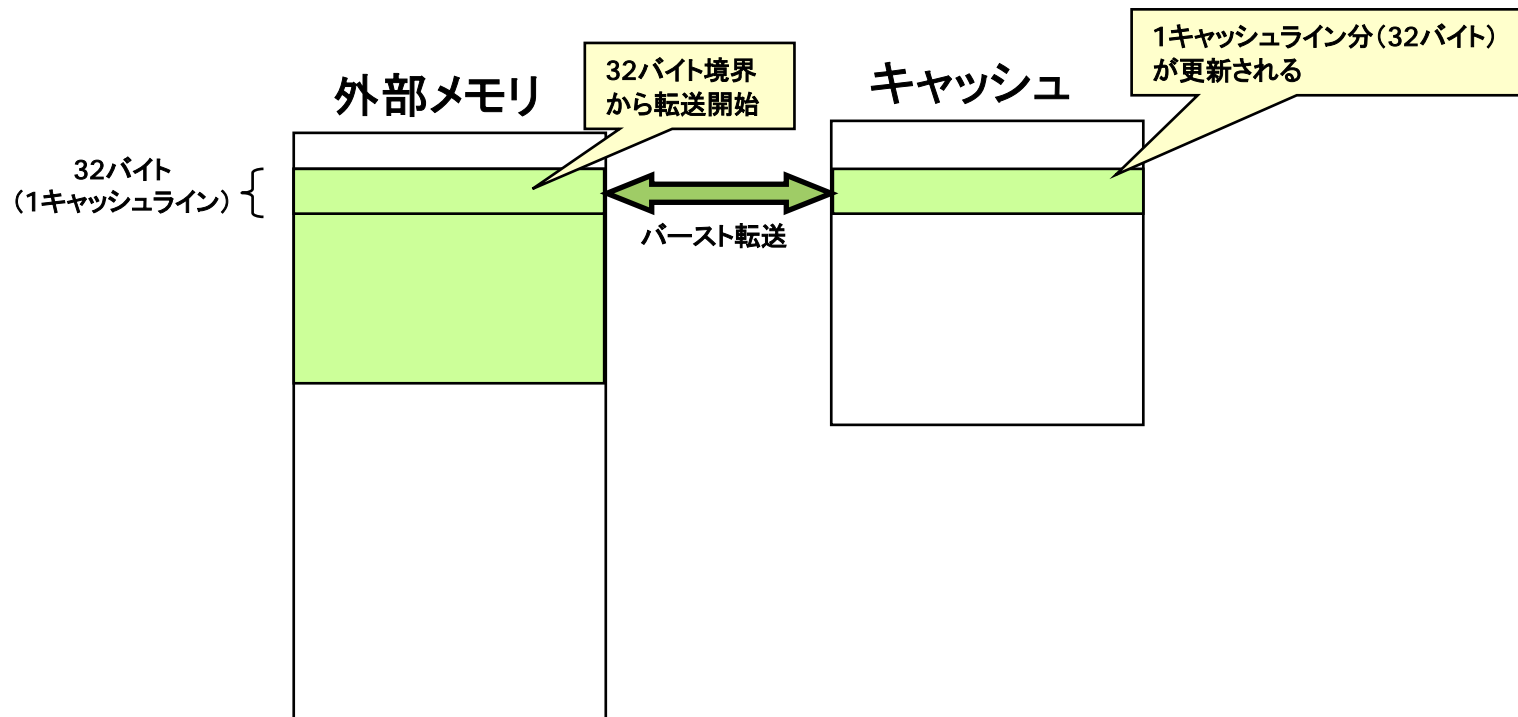
Block Size (vector)



コアと外部メモリ間転送の仕組み

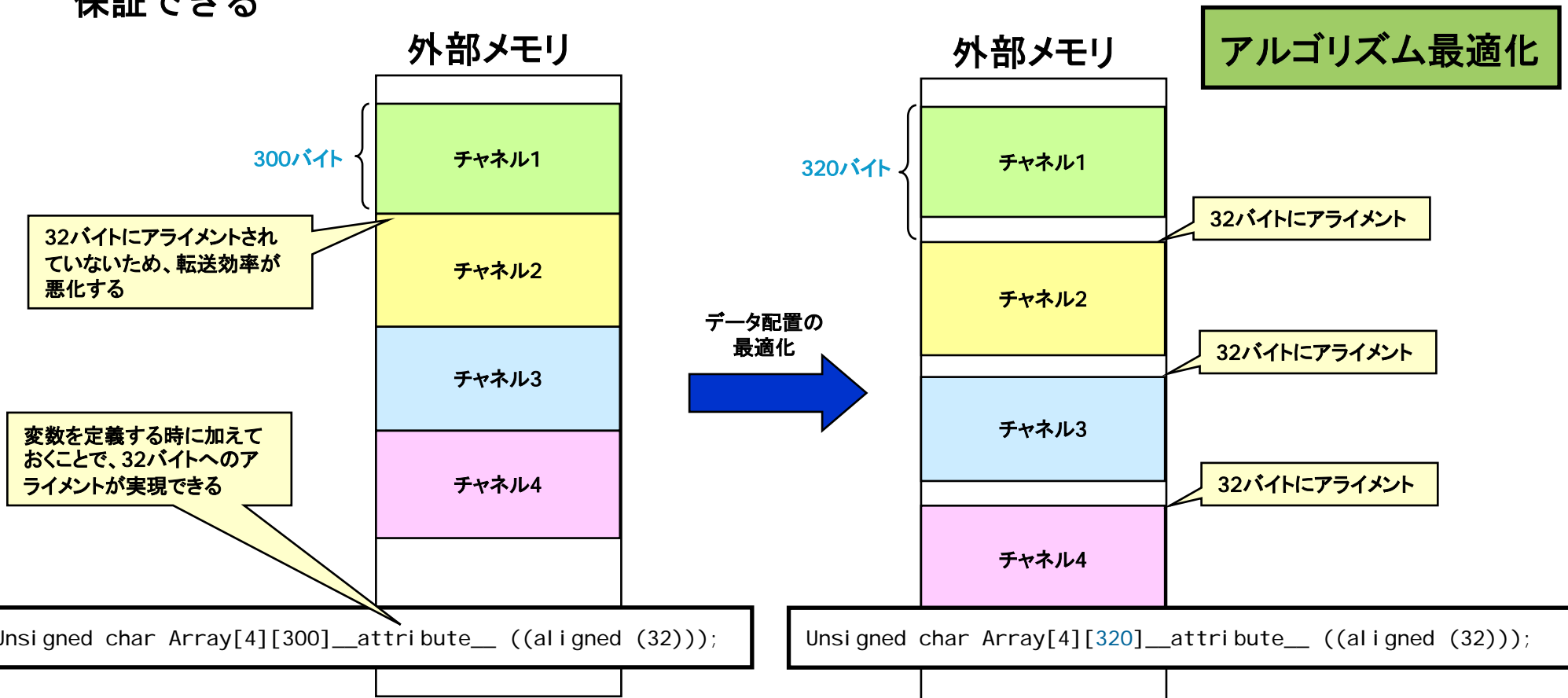
- 外部メモリとコア間のデータ転送は、必ず32バイト単位のバースト転送で行われる
- バースト転送は、32バイト境界から行われるため、配列の先頭アドレスが32バイトにアライメントされていると以下の効果が見込める
 - 端数データのために必要になる余計なバースト転送を削減
 - AltiVecのロード/ストア命令が活用できる
 - キャッシュ制御命令の適用が容易になる

コードレベル最適化②



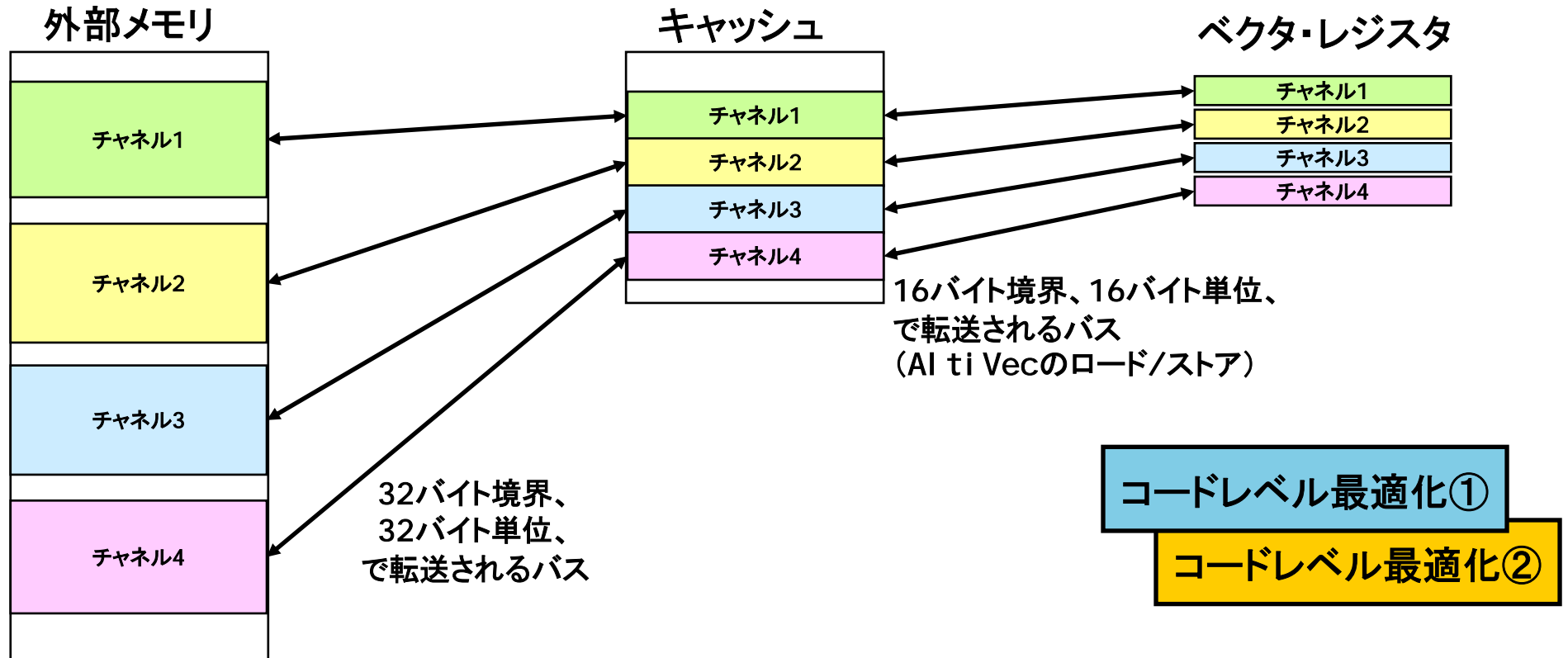
アライメントを意識した配列の定義方法

- 一つの配列の中に複数チャンネル分のデータが混在している場合、サイズ次第でチャンネル1以外のアライメントが外れる
- チャンネルごとのサイズを32バイトの倍数にしておくことで、32バイトアライメントが保証できる



アライメントと転送単位の理解

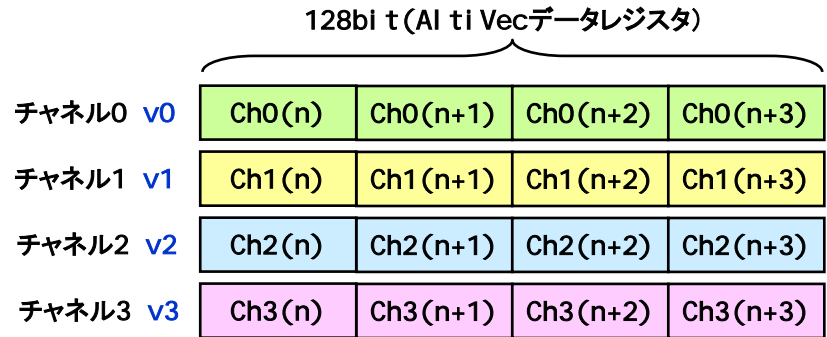
- AltiVecのロード/ストア命令は、アクセス先が16バイトアライメントされている必要がある
- データの先読み、先入れ替え等のキャッシュ制御を使うためには、32バイトアライメントされている必要がある
- 配列は32バイトでアライメントしておけば両方に対応可能



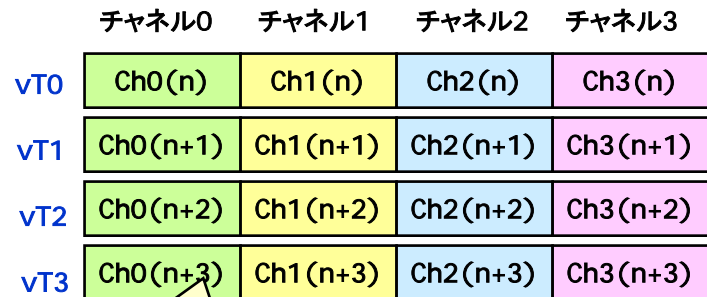
マルチチャネル処理に適したデータの並び

- 以下の例が示すように並べ替えるとAltiVecの演算が効率的に適用できる
- 以下の例は、32ビットデータを並べ替えを行うためのサンプルコード（アップル社提供）
- この考え方を応用することで16ビットデータ、8ビットデータについても同様に並べ替えが可能

```
void transposeVc ( vector float vT[4],  
vector float v0, vector float v1,  
vector float v2, vector float v3 )  
{  
vector float vI0, vI1, vI2, vI3;  
  
vI0 = vec_mergeh ( v0, v2 );  
vI1 = vec_mergeh ( v1, v3 );  
vI2 = vec_mergel ( v0, v2 );  
vI3 = vec_mergel ( v1, v3 );  
  
vT[0] = vec_mergeh ( vI0, vI1 );  
vT[1] = vec_mergel ( vI0, vI1 );  
vT[2] = vec_mergeh ( vI2, vI3 );  
vT[3] = vec_mergel ( vI2, vI3 );  
}
```



左のプログラムを実行



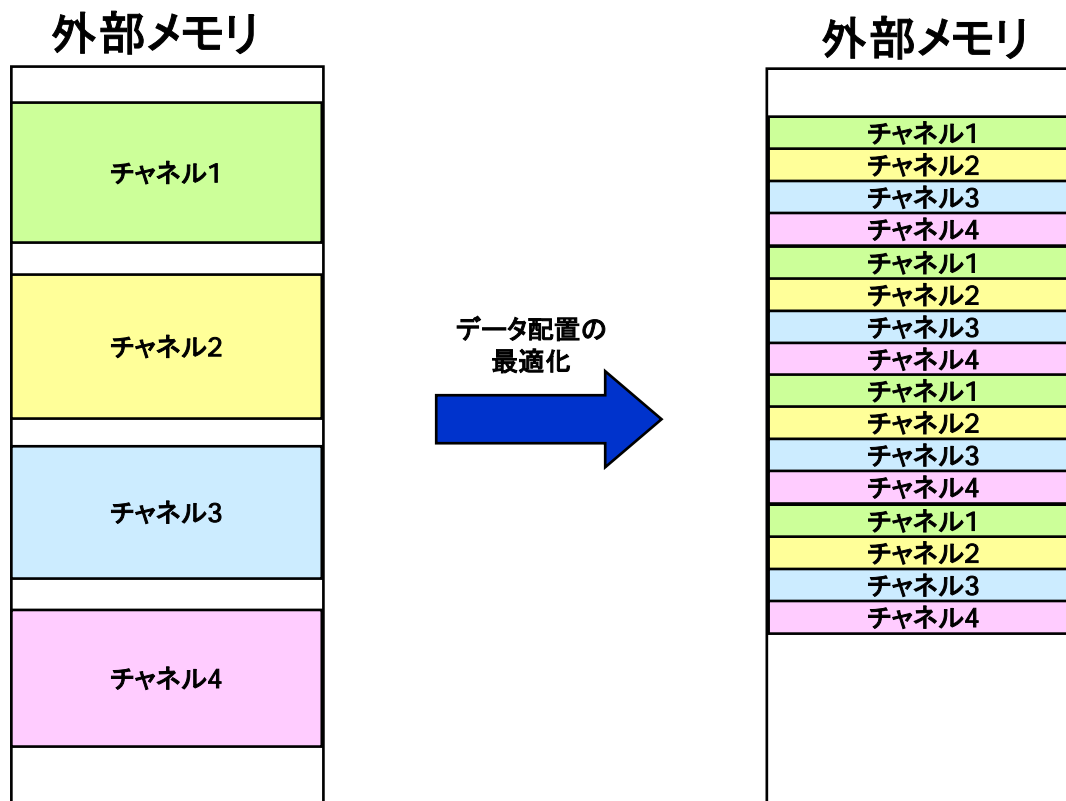
マルチチャネル処理に適したデータの並びを達成！

コードレベル最適化①

元データ自体の並び替え

- 前頁で解説した例は、レジスタに転送してから並べ替える処理自体がオーバーヘッド
- 可能であれば、元のデータ自体を所望の並びで配置しておく

アルゴリズム最適化



- Power PC G4のキャッシュのモードは、ライトバックとライトスルーのいずれかを選択可能(通常はライトバックで使用される)
- ライトスルーの特徴
 - ストア命令が発行された時に、キャッシュ及び外部メモリのデータを更新する
 - キャッシュ内データの入れ替え時にストールが発生しない
 - 冗長な外部メモリアクセスが頻繁に発生するため、外部メモリへの帯域が狭いとストア命令発行時にストールが発生する
- ライトバックの特徴
 - ストア命令が発行された時に、キャッシュのみデータを更新する
 - キャッシュがあふれない限り外部メモリアクセスが発生しない
 - キャッシュ内データの入れ替えが起きる時に**ストール**が発生する

dcbz命令やdcbf命令による最適化により解消可能(→次頁)

コードレベル最適化②

dcbz命令を使ったデータの先入れ替え

- データ・キャッシュ・ブロック・ゼロ(dcbz)命令を用いると、指定アドレスが含まれるキャッシュラインがオールゼロに変更される
- dcbz命令を発行しておくことで、ストア命令が実行される前にキャッシュの入れ替え作業を済ませておくことが可能

```
void __dcbz(void *, int);

vector unsigned char t0,t1;
ptrIn = (unsigned char*)InputBuf;
ptrOut = (unsigned char*)OutputBuf;
for(i=0;i<250;i++)
{
    __dcbz(ptrOut, 0);
    t0 = vec_ld(0, ptrIn);
    t1 = vec_ld(16, ptrIn);
    ptrIn += 32;
    vec_st( t0, 0, ptrOut);
    vec_st( t1, 16, ptrOut);
    ptrOut += 32;
}
```

Codewarriorに用意されている組み込み用関数を宣言

(ptrOut)番地が含まれる32バイトをゼロクリア

時間:4040

コードレベル最適化②

dcbf命令を使ったデータの強制退去

- dcbf(データ・キャッシュ・ブロック・フラッシュ)命令を用いることで指定されたキャッシュ・ブロックの最新データがメモリにあり、キャッシュにないことを保証する
- キャッシュに留まる必要が無いデータについては、dcbf命令を用いて明示的に退去させておくことで次の効果が得られ性能が向上する
 - キャッシュ内データの余計な入れ替え作業が減る
 - 優先度が高い他のデータがキャッシュに留まれる機会が増える

```
vec_dst( ptrIn, 0x00100200, 0);  
for(i=0;i<NUM;i++)  
{  
    __dcbz(ptrOut, 0);  
    t0 = vec_ld(0, ptrIn);  
    t1 = vec_ld(16, ptrIn);  
    __dcbf(ptrIn, 0);  
    ptrIn += 32;  
    vec_st( t0, 0, ptrOut);  
    vec_st( t1, 16, ptrOut);  
    ptrOut += 32;  
}
```

入力データの参照が終わったら即座にキャッシュを無効化

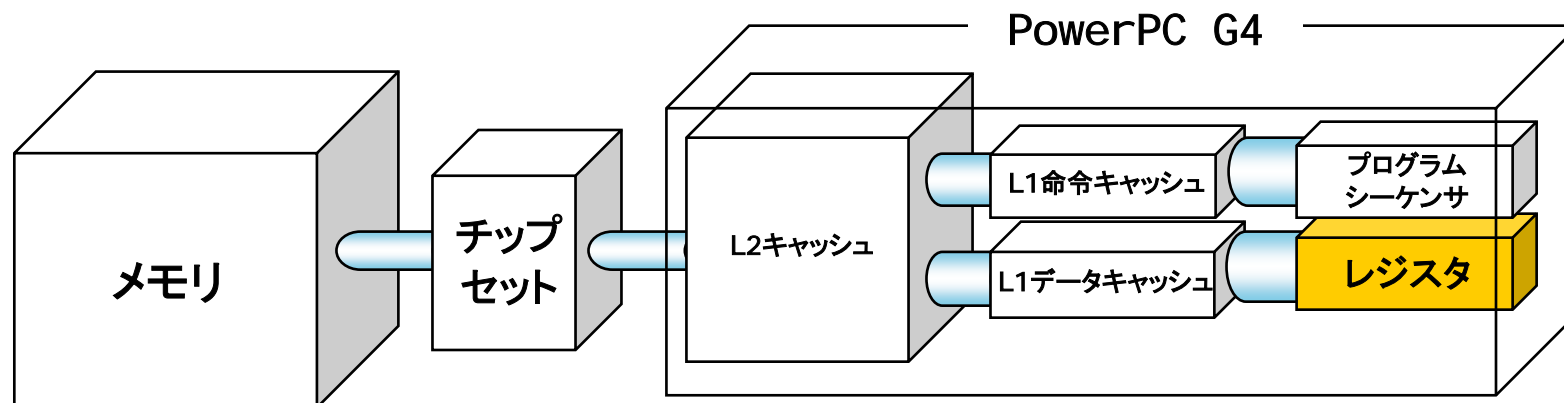
コードレベル最適化②

データの流をコントロール可能なロード/ストア命令

- キャッシュの入れ替え作業が起きる時、通常最も昔に参照されたキャッシュ・エントリ(32バイト)が追い出される(キャストアウト)ことになる
- この最も昔に参照されたと見なされているキャッシュ・エントリの状態をLRU(least recently used)ステートにあるという
- ロード/ストア命令でアクセスされキャッシュに取り込まれた状態のキャッシュ・エントリは、通常まずMRU(most recently used)の状態となる
- AltiVecのロード/ストア命令とデータ・ストリーム・タッチ命令には、キャッシュに取り込まれたデータを、強制的にLRU状態にする命令が用意されている
 - `vec_ldl()`
 - `vec_stl()`
 - `vec_dstt()`

コードレベル最適化②

- データは、キャッシュもしくはメモリからレジスタに転送する場合には、必ずロード命令の発行が伴う
- ベクトル演算ユニットのレジスタとして512バイト(32個x16バイト)用意されている
- 頻繁に参照される少量のデータは、このレジスタに常駐させておくことで、高速化を図る(例:vec_permで使用されるIndex情報)
- ワークレジスタとして使用するレジスタ数を少なくするために、同じ変数レジスタを使い回す →レジスタのPUSH/POPが減る



コードレベル最適化②

レジスタの有効活用

- 実装されているベクタ・レジスタは32個であるため、ループ中で使用する変数を32個以内するとスタック退避/復元が省略されて高速化が期待できる
- 以下の例で示すとおり、同じ変数を何度も使いまわすことで使用する変数を減らすことが可能
- ループ中で使用される変数が少なければ、vK0、vK1といった定数が格納されているベクタ変数がレジスタに常駐する確率が高くなる(多少使用するコンパイラに依存する)
- レジスタに常駐させたい変数については、定義時にregisterを付けると効果がある場合もある(使用するコンパイラに依存する)
- レジスタを使いまわし過ぎると、依存関係が増えて前の命令完了を待たなければならないケースが増えることがあるので注意する

```
for (i=0; i<100; i++)
{
    v0 = vec_ld(0, ptrIn);
    ptrIn += 16;
    v1=vec_sr(v0, vShift);
    v2=vec_and(v0, vMask);
    v3=vec_perm(vK0, vK1, v1);
    v4=vec_perm(vK0, vK1, v2);
    v5=vec_add(v3, v4);
    vSum=vec_sum4s(v5, vSum);
}
```

レジスタの数を
意識した最適化



```
register vector unsigned char
vK0, vK1, vShift, vMask;

for (i=0; i<100; i++)
{
    v0 = vec_ld(0, ptrIn);
    ptrIn += 16;
    v1=vec_sr(v0, vShift);
    v0=vec_and(v0, vMask);
    v1=vec_perm(vK0, vK1, v1);
    v0=vec_perm(vK0, vK1, v0);
    v0=vec_add(v0, v1);
    vSum=vec_sum4s(v0, vSum);
}
```

コードレベル最適化②

- 命令発行は、同時に3命令 + 1分岐を行うことが可能
- 命令発行のスループットを低下させる要因
 - 発行される命令間の依存関係
 - ブランチ命令等による命令発行パイプラインの乱れ
 - スループットが悪い割り算処理
 - 発行される命令が同一実行ユニットに集中
- 命令発行のスループットを向上させるために有効な意識
 - 処理間の依存関係を解消
 - If文等をブランチ命令が発行されない処理に置き換える
 - 割り算は別の処理に置き換える
 - 同時に発行される命令を各実行ユニットに分散させる

コードレベル最適化③

依存関係の解消を意識した最適化

- 浮動小数点の積和演算命令は、実行に5サイクルかかるがパイプライン処理が行われている限り見かけ上1サイクルで実行される
- 処理間に依存関係がある場合、パイプライン処理が成立せずストールが発生
- 分割して並列に処理を行うことで、依存関係が解消され性能向上が見込める

```
c = 1.0f;  
c = c + a[0]*a[0];  
c = c + a[1]*a[1]; //4cycle stall  
c = c + a[2]*a[2]; //4cycle stall  
c = c + a[3]*a[3]; //4cycle stall  
  (中略)  
c = c + a[99]*a[99]; //4cycle stall  
c = c + a[99]*a[99]; //4cycle stall
```

直前の処理による結果が
得られないと実行できない

➡
処理分割
で最適化

```
c0 = 1.0f; c1 = c2 = c3 = 0;  
c0 = c0 + a[0]*a[0];  
c1 = c1 + a[1]*a[1];  
c2 = c2 + a[2]*a[2];  
c3 = c3 + a[3]*a[3];  
c0 = c0 + a[4]*a[4];  
c1 = c1 + a[5]*a[5];  
  (中略)  
c0 = c0 + a[96]*a[96];  
c1 = c1 + a[97]*a[97];  
c2 = c2 + a[98]*a[98];  
c3 = c3 + a[99]*a[99];  
c = c0 + c1 + c2 + c3;
```

コードレベル最適化③

if文のベクトル処理化

- If文をコンパイルすると、ブランチ命令が生成されてしまいパイプラインが乱れる
- If文の代わりにvec_cmp命令vec_sel命令を使って総当りで処理すればパイプラインを乱さない
- パイプラインが乱れなければ高速化の余地がさらに広がる→後述

スカラー処理

```
signed long* pt_bufRTmp = pt_bufR;  
signed long* pt_bufWTmp = pt_bufW;  
  
for( j = 0; j < size/sizeof(long); j++ )  
{  
    if(*pt_bufRTmp>=0)  
    {  
        *pt_bufWTmp++ = *pt_bufRTmp++;  
    }  
    else  
    {  
        *pt_bufWTmp++ = -*pt_bufRTmp++;  
    }  
}
```

コードレベル最適化①

コードレベル最適化③

入力データの絶対値を出力

Altivec命令で最適化

ベクトル処理

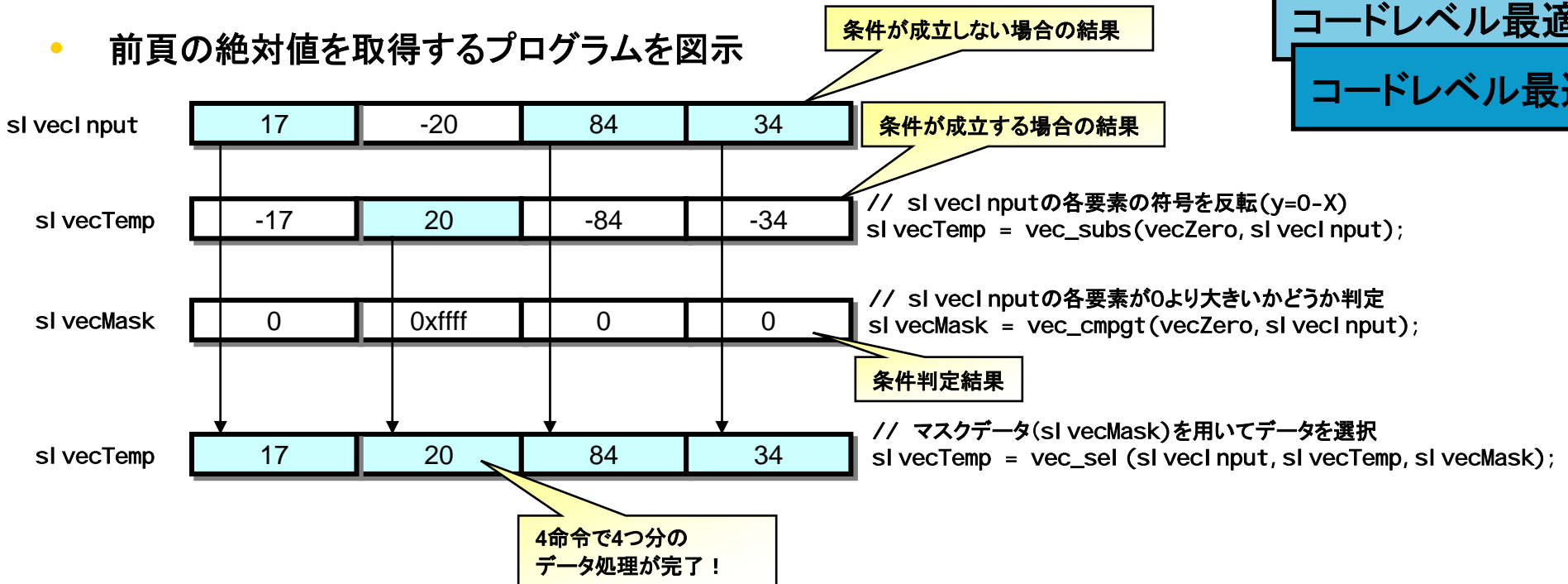
```
for( j = 0; j < size/(4*sizeof(long)); j++ )  
{  
    si vecInput = vec_ld(0, pt_bufRTmp);  
    si vecMask = vec_cmpgt(vecZero, si vecInput);  
    si vecTemp = vec_subs(vecZero, si vecInput);  
    si vecTemp = vec_sel( si vecInput, si vecTemp, si vecMask);  
    vec_st( si vecTemp, 0, pt_bufWTmp);  
    pt_bufWTmp +=4;  
}
```

//入力データを取得
//マスク用データを取得
//符号反転(y=0-x)
//結果を選択して結合
//結果のストア
//ポインタのアップデート

if文のベクトル処理化(続き)

- 前頁の絶対値を取得するプログラムを図示

コードレベル最適化①
コードレベル最適化③



入力データについて条件判定を行い、結果を保持

真の場合と偽の場合、両方とも総当りで計算

vec_sel 命令を使って期待した処理が行われたデータを選択

パイプラインが乱れない
SIMDで処理が可能
命令の発行も並列化され易い

switch-case文のベクトル処理化①

- switch文による条件分岐処理のベクトル処理化を考える
- 以下のプログラムでは、入力データを4で割った余りの値を判定して、入力データに施される処理がswitch-case文で場合分けされている

スカラー処理

```
void vec_case(unsigned long *src, unsigned long *dst, unsigned long count)
{
    unsigned long int i;
    unsigned long ulTemp0;
    unsigned long ulTemp1;
    for (i = 0; i < count; i++)
    {
        ulTemp0 = *src++;
        switch((ulTemp0)%4)
        {
            case 0:
                ulTemp1 = 0;
                break;
            case 1:
                ulTemp1 = ulTemp0 - 1;
                break;
            case 2:
                ulTemp1 = ulTemp0 + 2;
                break;
            default:
                ulTemp1 = (ulTemp0 + 1)*2;
        }
        *dst++ = ulTemp1;
    }
}
```

コードレベル最適化①

コードレベル最適化③

switch文は、多くの条件分岐処理で構成されているため、ループの中では、条件分岐に伴って発生するパイプラインの乱れがオーバーヘッドになる。

switch-case文のベクトル処理化②

- Altivec命令によるベクトル処理化を行うために、まず次のように処理を変えて考える
- 全ての“case”について総当りで処理を行いマスクしながら所望のデータを抽出する方法であればAltivec命令を用いた時に条件分岐を行わずに結果を得ることが可能

ベクトル処理化を意識したスカラー処理

```
void vec_case(unsigned long *src, unsigned long *dst, unsigned long count)
{
    unsigned long int i;
    unsigned long ulTemp0, ulTemp1;
    unsigned long mask0, mask1, mask2, mask3;
    for (i = 0; i < count; i++)
    {
        ulTemp0 = *src++;
        mask0 = (ulTemp0 == 0)? 0xffff : 0;
        mask1 = (ulTemp0 == 1)? 0xffff : 0;
        mask2 = (ulTemp0 == 2)? 0xffff : 0;
        mask3 = (ulTemp0 == 3)? 0xffff : 0;
        //case 0:
        ulTemp1 = 0 & mask0;
        //case 1:
        ulTemp1 = (ulTemp0 - 1) & mask1;
        //case 2:
        ulTemp1 = (ulTemp0 + 2) & mask2;
        default:
        ulTemp1 = ((ulTemp0 + 1)*2) & mask3;
        *dst++ = ulTemp1;
    }
}
```

この処理形式であれば、複数のデータについてまとめて処理が可能
->vec_cmpeq命令が活用できる

コードレベル最適化①

コードレベル最適化③

ベクトル処理化した場合、このマスク処理には、vec_sel命令が活用できる

switch-case文のベクトル処理化③

- データが、8ビットや16ビットデータであれば、さらに並列化を行うことが可能
- 総当りを行うため、'case'が増えるに伴って命令発行数が増加する点に注意

ベクトル処理

```
void vec_case(unsigned Long *src, unsigned Long *dst, unsigned Long count)
{
    vector unsigned int vui Val 0 = vec_splat_u32(0);
    vector unsigned int vui Val 1 = vec_splat_u32(1);
    vector unsigned int vui Val 2 = vec_splat_u32(2);
    vector unsigned int vui Val 3 = vec_splat_u32(3);
    vector unsigned int vui Temp0, vui Temp1, vui VecInput;
    vector bool int vecMask1, vecMask2, vecMask3;
    unsigned Long int i;
    for (i = 0; i < (count>>2); i++)
    {
        vui VecInput = vec_ld( 0 , src );
        vui Temp0 = vec_and( vui VecInput , vui Val 3 );
        vecMask1 = vec_cmpeq( vui Temp0 , vui Val 1 );
        vecMask2 = vec_cmpeq( vui Temp0 , vui Val 2 );
        vecMask3 = vec_cmpeq( vui Temp0 , vui Val 3 );
        vui Temp0 = vui Val 0; //case0の処理
        vui Temp1 = vec_sub( vui VecInput , vui Val 1 ); //case1の処理
        vui Temp0 = vec_sel( vui Temp0, vui Temp1, vecMask1 ); //必要な結果だけを抽出
        vui Temp1 = vec_add( vui VecInput , vui Val 2 ); //case2の処理
        vui Temp0 = vec_sel( vui Temp0, vui Temp1, vecMask2 ); //必要な結果だけを抽出
        vui Temp1 = vec_add( vui VecInput , vui Val 1 ); //defaultの処理
        vui Temp1 = vec_add( vui Temp1 , vui Temp1 ); //defaultの処理
        vui Temp0 = vec_sel( vui Temp0, vui Temp1, vecMask3 ); //必要な結果だけを抽出
        vec_st( vui Temp0, 0 , dst);
        src += 4;
        dst += 4;
    }
}
```

コードレベル最適化①

コードレベル最適化③

2.5倍の高速化を実現

条件分岐なしで処理を記述可能！

条件分岐先の命令数が多い場合

- 以下で示す例は、func1(),func2()をそれぞれAltiVec命令で高速化し、if文と同様に双方を実行する
- 条件判定結果が真か偽に偏っている場合、さらにアルゴリズム最適化の余地がある(→次頁)

スカラー処理

```
signed long* pt_bufRTmp = pt_bufR;  
signed long* pt_bufWTmp = pt_bufW;  
  
for( j = 0; j < size/sizeof(long); j++ )  
{  
    if(*pt_bufRTmp>=0)  
    {  
        *pt_bufWTmp++ = func1();  
    }  
    else  
    {  
        *pt_bufWTmp++ = func2();  
    }  
}
```

コードレベル最適化①

コードレベル最適化③

AltiVec命令で最適化

ベクトル処理

```
for( j = 0; j < size/(4*sizeof(long)); j++ )  
{  
    sivecInput = vec_ld(0, pt_bufRTmp);  
    sivecMask = vec_cmpgt(vecZero, sivecInput);  
    v1 = vec_func1();  
    v2 = vec_func2();  
    sivecTemp = vec_sel(v1, v2, sivecMask);  
    vec_st(sivecTemp, 0, pt_bufWTmp);  
    pt_bufWTmp +=4;  
}
```

条件が真の場合と偽の場合の処理が行われているが、それぞれが4倍の高速化を達成できていれば全体では2倍の高速化を達成可能

```
//入力データを取得  
//マスク用データを取得  
//ベクタ化された関数を呼び出し  
//ベクタ化された関数を呼び出し  
//結果を選択して結合  
//結果のストア  
//ポインタのアップデート
```

条件分岐先の命令数が多い場合(続き)

アルゴリズム最適化

- 以下の例では、条件判定結果が、統計的にどちらかに偏る場合には片方の処理が省略されるため、さらなる高速化が期待できる

```
for( j = 0; j < size/(4* sizeof(long)); j++ )
{
    sl vecInput = vec_ld(0, pt_bufRTmp);           //入力データを取得
    sl vecMask = vec_cmpgt(vecZero, sl vecInput); //マスク用データを取得
    if( vec_any_eq( vec_edge.vec , (vector bool int)zero ))
    {
        v1 = vec_func1();                         //ベクタ化された関数を呼び出し
    }
    if( vec_any_ne( vec_edge.vec , (vector bool int)zero ))
    {
        v2 = vec_func2();                         //ベクタ化された関数を呼び出し
    }
    sl vecTemp = vec_sel( v1, v2, sl vecMask);     //結果を選択して結合
    vec_st(sl vecTemp, 0, pt_bufWtmp);           //結果のストア
    pt_bufWtmp +=4;                             //ポインタのアップデート
}
}
```

一つでも条件が成立していたデータがあれば実行

一つでも条件が成立していないデータがあれば実行

ほとんどの場合で、条件判定の結果が、全て真、または全て偽であれば、全体としてはほぼ4倍の高速化を実現

- PowerPCのコアには、ループカウンタとなるレジスタが用意されていて、このレジスタを使われるとループの折返しが高速に処理される
- C言語で高速化を意識する場合、while文の代わりにfor文を使えば、ループごとに1サイクル高速に処理される

```
while(loopcount<0)
{
    func();
    loopcount-=8;
}
```



```
for (i=0; i<loopcount/8; i++)
{
    func();
}
```

G4プロセッサで型変換(キャスト)を実現する手段

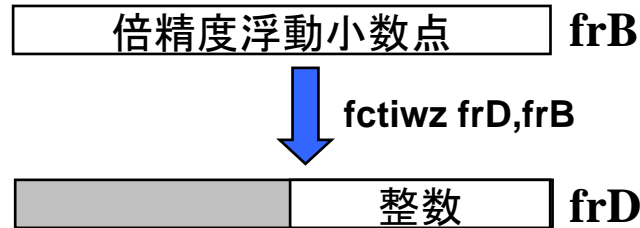
コードレベル最適化①

- 単精度浮動小数点形式->整数型
 - スカラーユニット ->ハードウェアでサポートされている命令で処理
(fctiwz)
 - AltiVec ->ハードウェアでサポートされている命令で処理
 - vec_cts(アセンブリ言語: vctxs)
 - vec_ctu(アセンブリ言語: vcfsx)
- 整数型->単精度浮動小数点形式
 - スカラーユニット ->ソフトウェアで処理
 - AltiVec ->ハードウェアでサポートされている命令で処理
 - vtec_ctf(アセンブリ言語: vcfux)

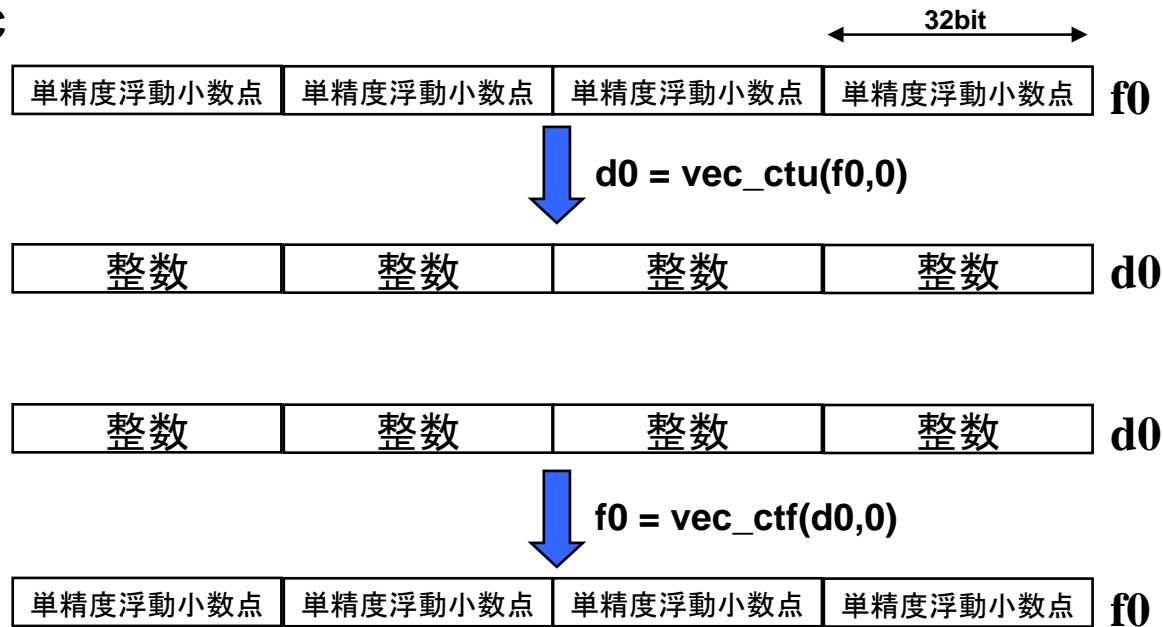
ハードウェア(命令)でサポートされている型変換

コードレベル最適化①

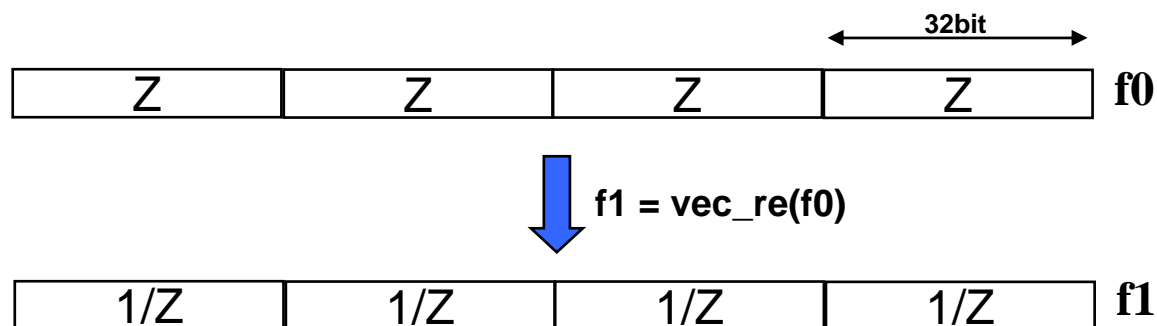
- スカラーユニット(浮動小数点ユニット)



- AltiVec



- 割り算は、演算系の命令の中で唯一スルーポットが1でない命令
- $Y = X / Z$ の割り算は $Y = X \times (1/Z)$ で置き換えが可能
- 単精度の浮動小数点形式であれば、AltiVecのvec_reで4つ分の逆数値をスルーポット1サイクルで算出可能

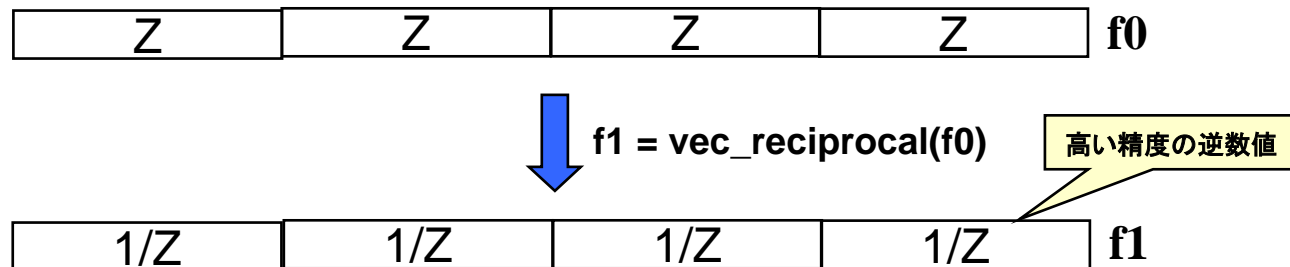


コードレベル最適化①

コードレベル最適化③

- `vec_re`は、一定の誤差が生じる
- `vec_re`の代わりに、以下に記述されている関数`vec_reciprocal()`を使うことで精度の高い逆数値が得られる

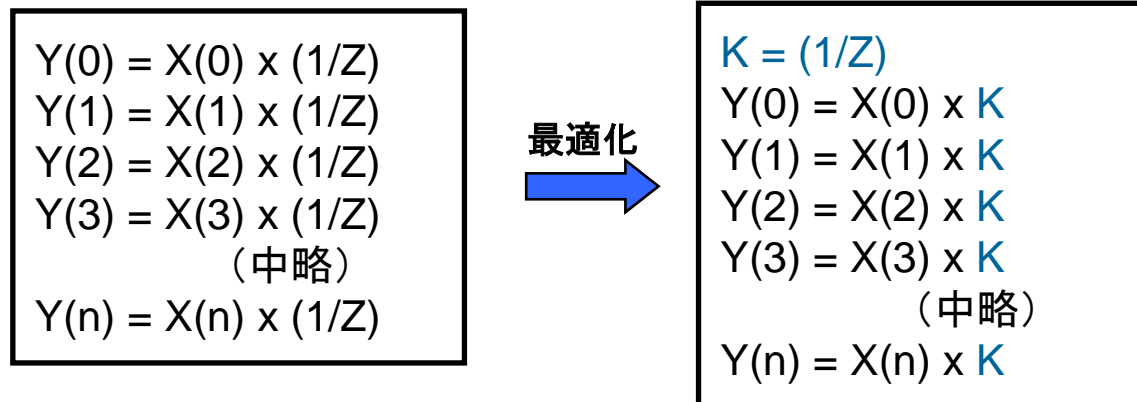
```
//Newton Raphson法により改良された逆数の算出
inline vector float vec_reciprocal( vector float v )
{
    vector float reciprocal = vec_re( v );
    return vec_madd( reciprocal, vec_nmsub( reciprocal, v, vec_float_one()), reciprocal );
}
//(vector float)(1.0, 1.0, 1.0, 1.0)値の生成
inline vector float vec_float_one( void )
{
    return vec_ctf( vec_splat_u32(1), 0);
}
```



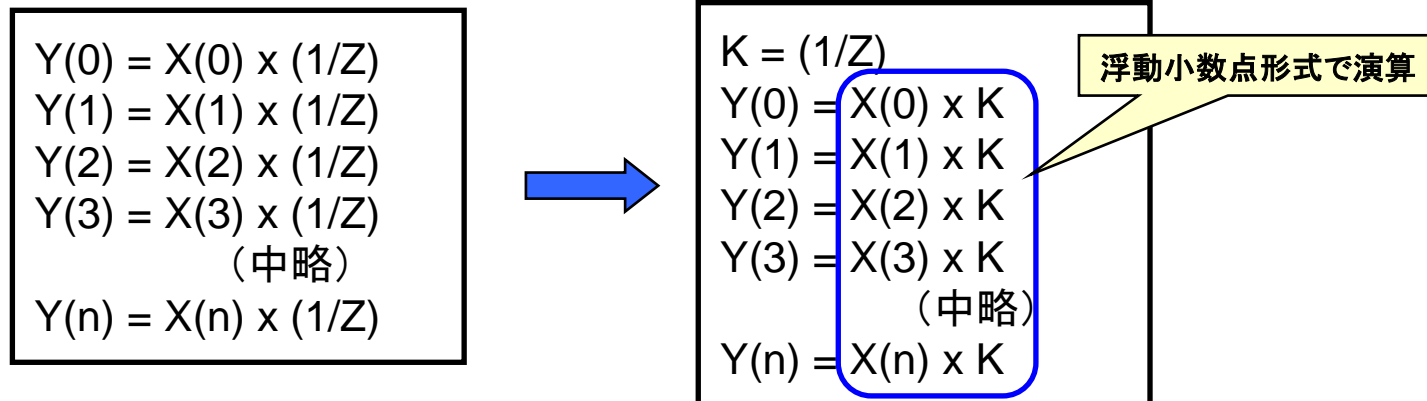
割り算の最適化-分母値が不変の場合

アルゴリズム最適化

- $Y = X / Z$ の割り算で、分母の Z 値が固定されている場合、あらかじめ逆数値を用意しておく $\rightarrow (1/Z) = K(\text{定数})$ としておく
- 毎回逆数を算出する手間が省け高速化が期待できる



- 整数型で割り算を行う場合、固定小数点演算で同様の手法が可能
- しかしながら、ダイナミックレンジが小さいため浮動小数点形式で演算するより精度が悪くなる
- 以下の演算を行う場合、型変換を行って浮動小数点で計算すると計算精度を犠牲にしないで高速化が図れる
- スループットを悪化させずに実行可能

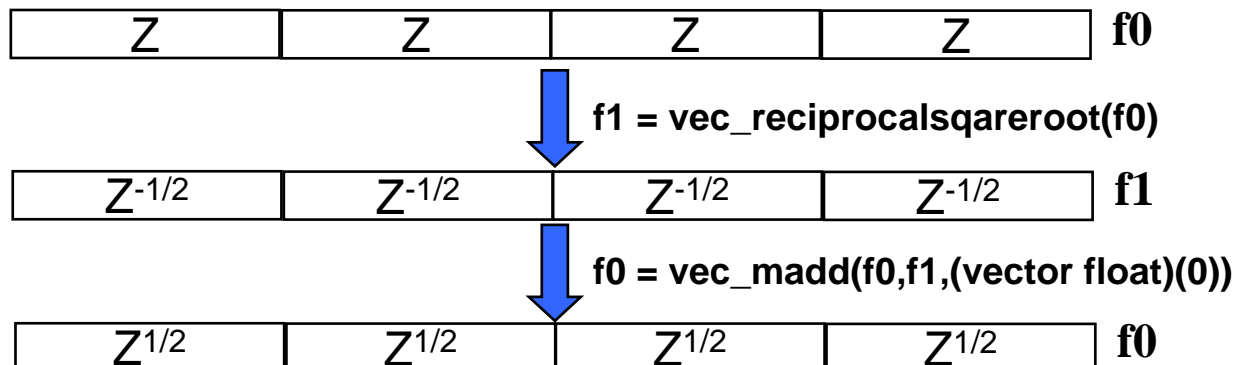


精度の高い平方根の算出手法

- 平方根を高速に算出するには、平方根の逆数を算出できるvec_rsqртеを使用する
- ただし、vec_rsqртеは一定の誤差が生じるため、ニュートン法を用いて精度の高い平方根の逆数を算出する

```
//Newton Raphson法により改良された平方根逆数の算出
inline vector float vec_reciprocal_squareroot( vector float v )
{
    vector float v_const = (vector float)( 0 , 0.5 , 3.0 , 0 );
    vector float v0 , v1 , v2 , rsqrte;
    //(vector float)(0, 0, 0, 0)値の生成
    v0 = vec_splat( v_const , 0 );
    //(vector float)(0.5, 0.5, 0.5, 0.5)値の生成
    v1 = vec_splat( v_const , 1 );
    //(vector float)(3.0, 3.0, 3.0, 3.0)値の生成
    v2 = vec_splat( v_const , 2 );
    rsqrte = vec_rsqрте( v );
    v1 = vec_madd( rsqrte , v1 , v0 );
    rsqrte = vec_madd( rsqrte , rsqrte , v0 );
    rsqrte = vec_nmsub( rsqrte , v , v2 );
    v1 = vec_madd( v1 , rsqrte , v0 );
    v0 = vec_madd( v1 , v , v0 );
    return ( v0 );
}
```

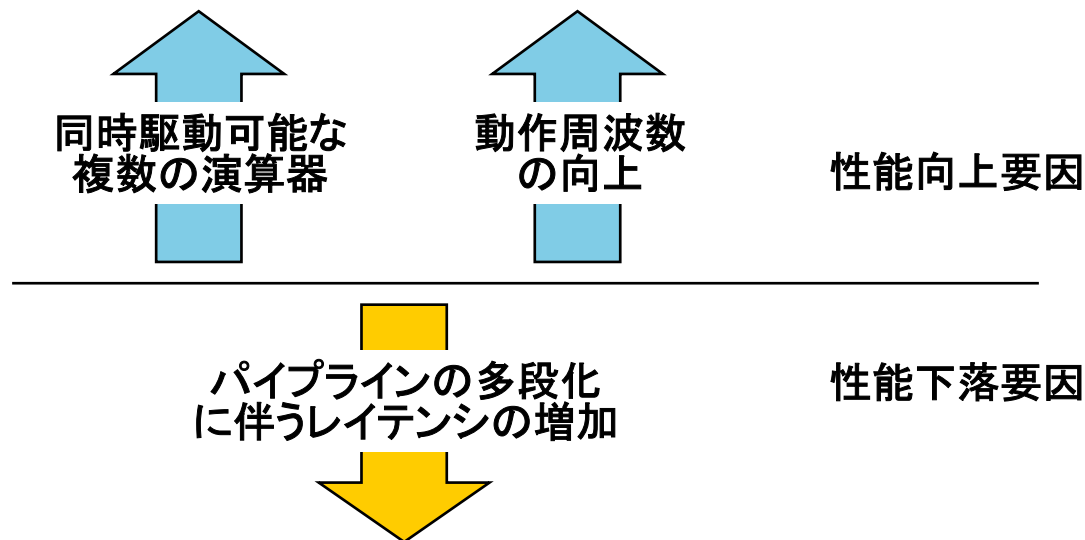
コードレベル最適化③



命令のレイテンシと命令発行の効率性

最近のプロセッサが性能向上を実現するために行っていること

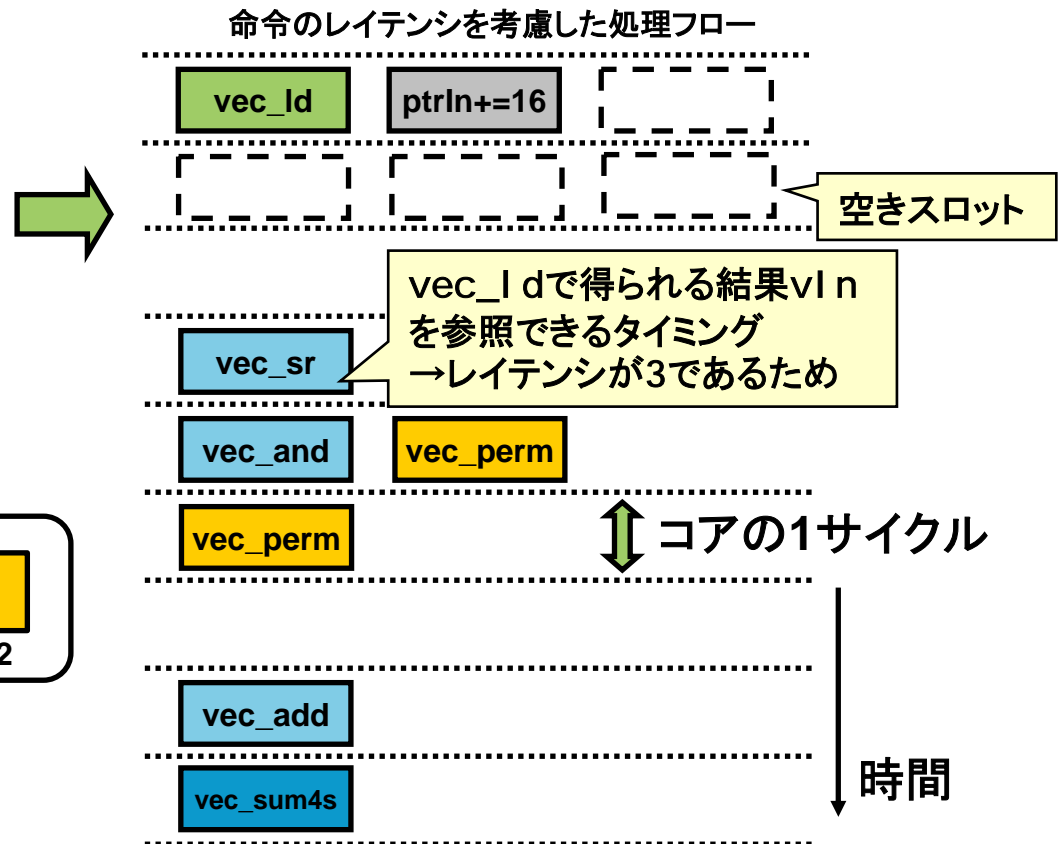
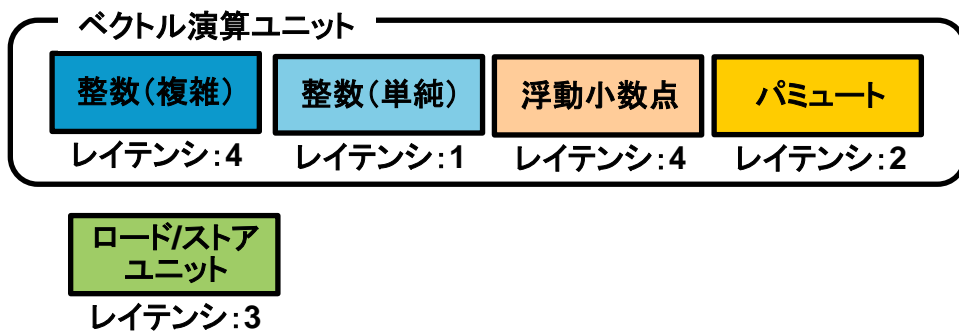
- 複数の演算器を同時に稼働させることを可能にする (PowerPC G4は3命令+1分岐命令を同時に実行可能)
- 高い動作周波数を実現するために、命令発行のパイプラインを多段化



レイテンシがもたらす悪影響

- 以下のCで記述したプログラムが、一対一でアセンブリコードに変換されると仮定
- それぞれの命令間で依存関係が無く、同時に駆動する演算ユニットが異なれば、毎回3命令ずつ発行される
- しかし、レイテンシ等により実際の命令発行のタイミングは以下のようなになる

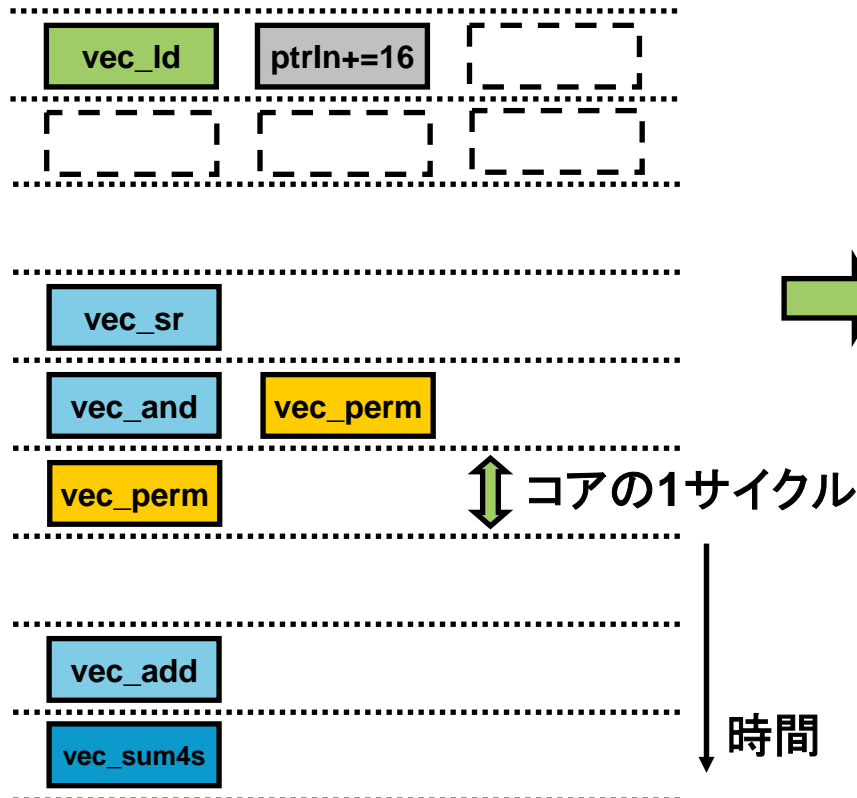
```
for (i=0; i<800/16; i++)  
{  
    vl n = vec_ld(0, ptrIn);  
    ptrIn += 16;  
    vl nHi gh=vec_sr(vl n, vShift);  
    vl nLow=vec_and(vl n, vMask);  
    vRes0=vec_perm(vTable0, vTable0, vl nHi gh);  
    vRes1=vec_perm(vTable0, vTable0, vl nLow);  
    vout=vec_add(vRes0, vRes1);  
    vSum=vec_sum4s(vout, vSum);  
}
```



命令発行のスループット

- PowerPC G4は、3命令/サイクルの命令発行ができるにも関わらず、通常のAltiVecコードの命令発行は0.8~1.5命令/サイクルに留まることが多い
- 用意されている豊富なレジスタを生かして最適化を図ることで、3命令/サイクルのスループットを実現することが可能になる

命令のレイテンシを考慮した処理フロー



8命令を9サイクルで処理
→0.89命令/サイクルのスループット

時間

命令発行のスループットを向上させるための手法を考察

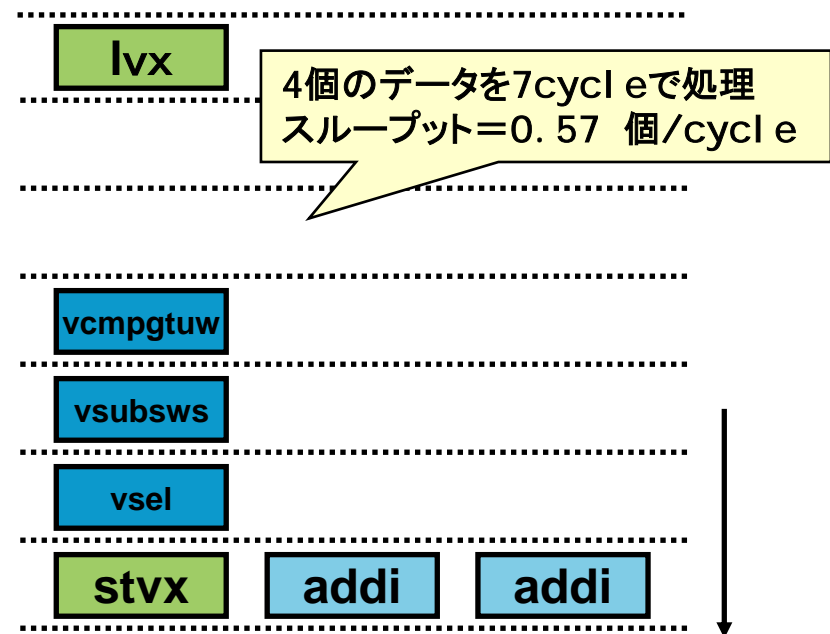
- If文の最適化で解説したサンプル・プログラムを用いて解説
- 命令は、参照するレジスタの更新完了を待って開始するため、ストールが起きる

絶対値取得プログラム

```
for( j = 0 ; j < 100 ; j++ )
{
    vIn = vec_ld(0,ptrIn);
    vMask = vec_cmpgt(vZero,vIn);
    vTmp = vec_subs(vZero,vIn);
    vTmp = vec_sel(vIn,vTmp,vMask);
    vec_st(vTmp, 0, ptrOut);
    ptrIn += 4;
    ptrOut += 4;
}
```

コンパイル
&
実行

命令のレイテンシを考慮した処理フロー



ループ展開を用いた高速化①

コードレベル最適化③

ループ展開を用いるメリット

- 命令の発行スループットが向上する
- プログラミングが容易

デメリット

- 同時に稼働させたい実行ユニットが重複しがちになる
- ループ中に使用される変数が増える
→変数が多過ぎるとスタック退避/復元処理が増えてかえって速度が低下

絶対値取得プログラム

```
for( j = 0 ; j < 100 ; j++ )
{
    vIn = vec_ld(0,ptrIn);
    vMask = vec_cmpgt(vZero,vIn);
    vTmp = vec_subs(vZero,vIn);
    vTmp = vec_sel(vIn,vTmp,vMask);
    vec_st(vTmp, 0, ptrOut);
    ptrIn += 4;
    ptrOut += 4;
}
```

ループ展開

使用する変数
が増加

```
for( j = 0 ; j < 50 ; j++ )
{
    vInA = vec_ld(0,ptrIn);
    vInB = vec_ld(16,ptrIn);
    vMaskA = vec_cmpgt(vZero,vInA);
    vMaskB = vec_cmpgt(vZero,vInB);
    vTmpA = vec_subs(vZero,vInA);
    vTmpB = vec_subs(vZero,vInB);
    vTmpA = vec_sel(vInA,vTmpA,vMaskA);
    vTmpB = vec_sel(vInB,vTmpB,vMaskB);
    vec_st(vTmpA, 0, ptrOut);
    vec_st(vTmpB, 0, ptrOut);
    ptrIn += 8;
    ptrOut += 8;
}
```

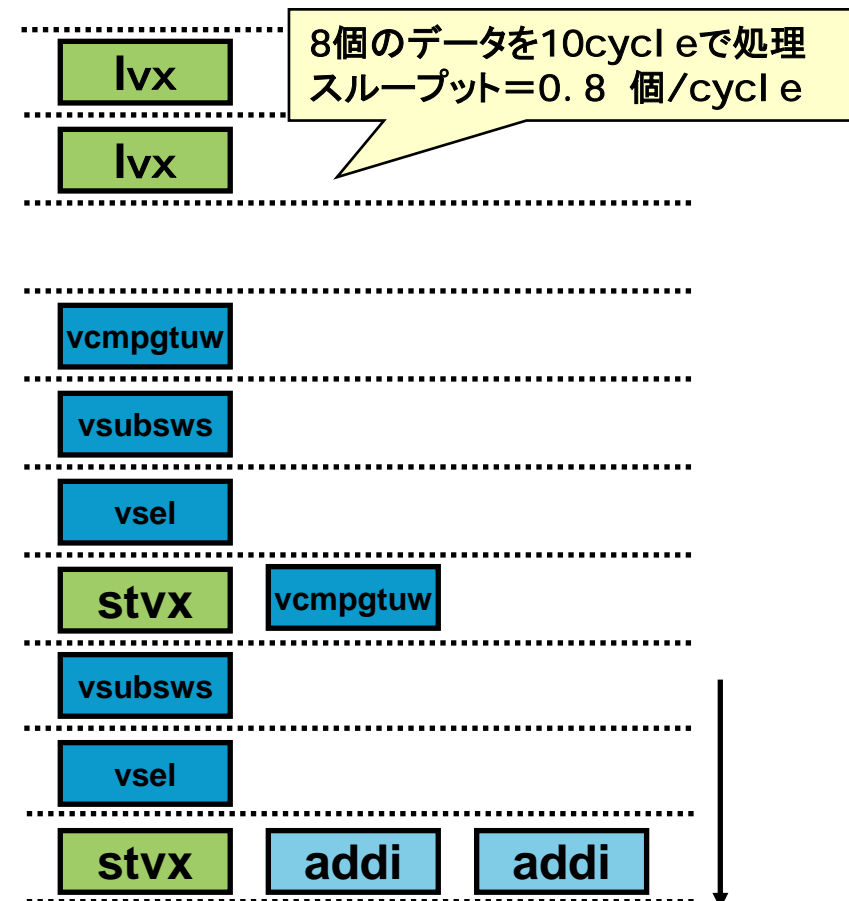
ループ展開を用いた高速化②

- ループ展開を用いることで空きスロットが減り性能が向上する
- ただし、稼動する実行ユニットが偏っていると効果が出にくい

絶対値取得プログラム

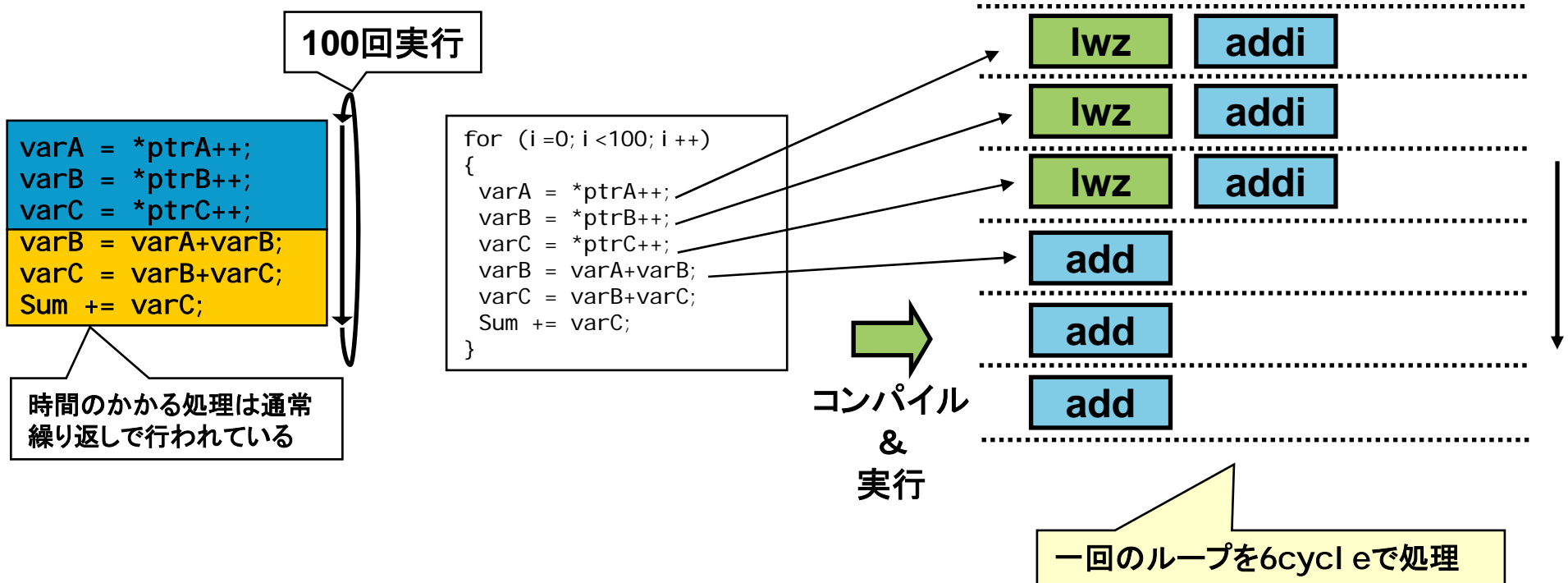
```
for( j = 0 ; j < 50 ; j++ )
{
    vInA = vec_ld(0,ptrIn);
    vInB = vec_ld(16,ptrIn);
    vMaskA = vec_cmpgt(vZero,vInA);
    vMaskB = vec_cmpgt(vZero,vInB);
    vTmpA = vec_subs(vZero,vInA);
    vTmpB = vec_subs(vZero,vInB);
    vTmpA = vec_sel(vInA,vTmpA,vMask);
    vTmpB = vec_sel(vInB,vTmpB,vMask);
    vec_st(vTmpA, 0, ptrOut);
    vec_st(vTmpB, 0, ptrOut);
    ptrIn += 8;
    ptrOut += 8;
}
```

コンパイル
&
実行

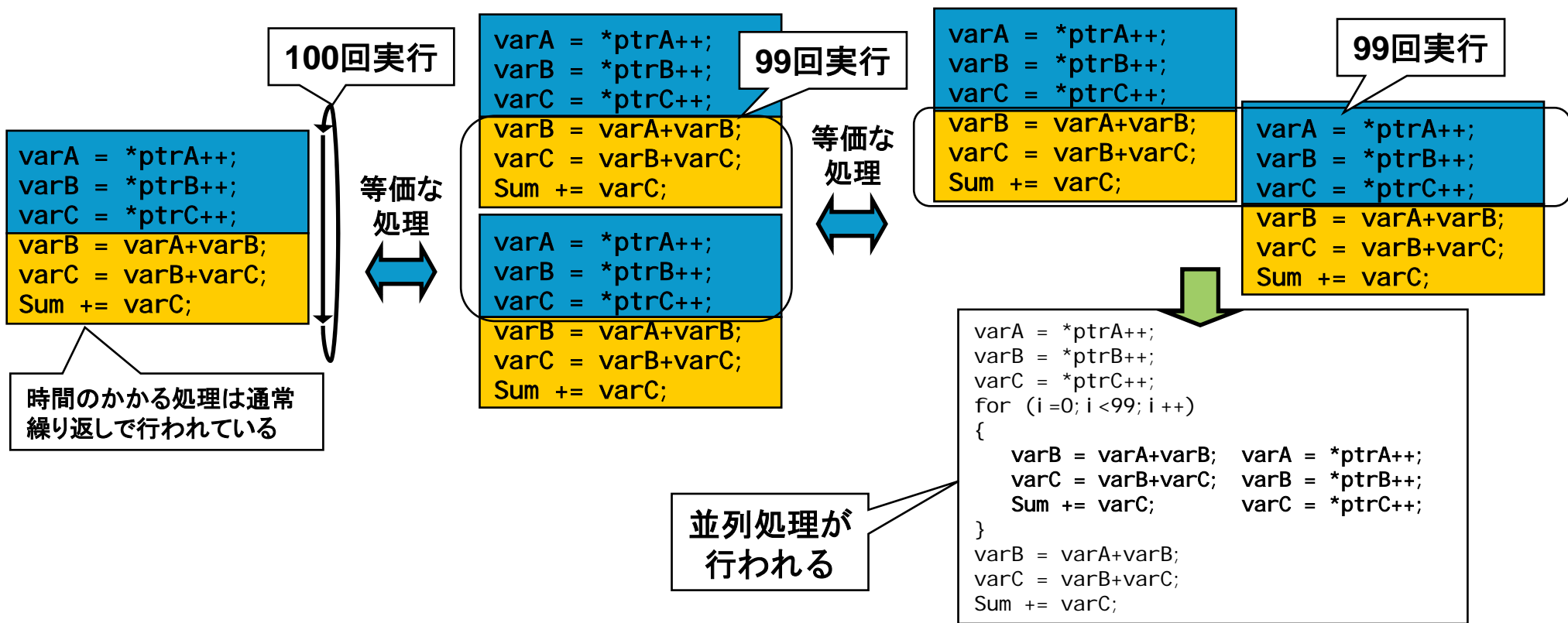


ソフトウェアパイプライン処理①

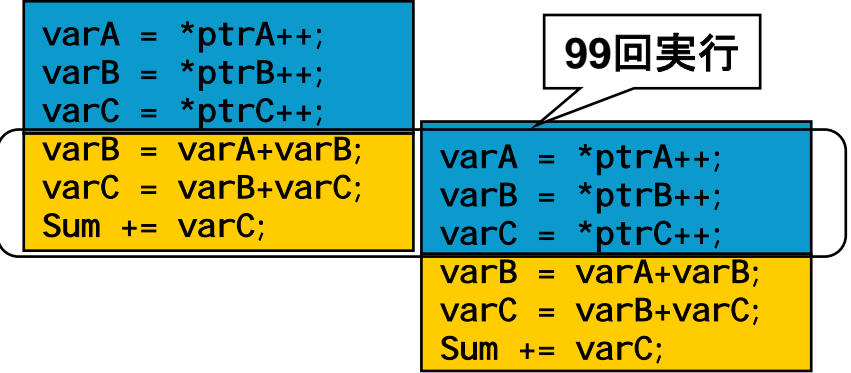
- ソフトウェアパイプライン処理を導入すれば潜在処理性能を引き出しやすくなる
- ここではロードが完了するまで直後の2サイクルは参照できないためadd命令が待たされる



- ソフトウェアパイプライン処理は、n番目の後半処理とn+1番目の前半処理を交互に実行することで依存関係を引き伸ばすテクニック



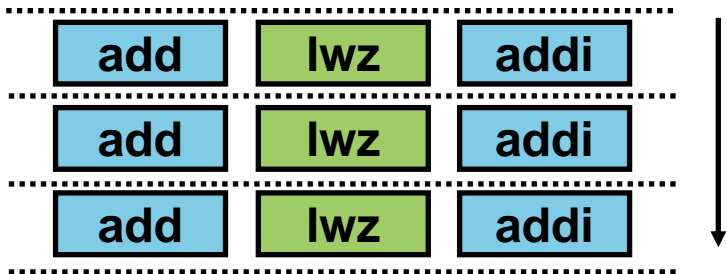
- 以下で示すとおりループ中で3命令同時発行可能という特長を生かすことができる



```
for (i=0; i<99; i++)  
{  
  varB=varA+varB; varA = *ptrA++;  
  varC=varB+varC; varB = *ptrB++;  
  Sum += varC; varC = *ptrC++;  
}
```

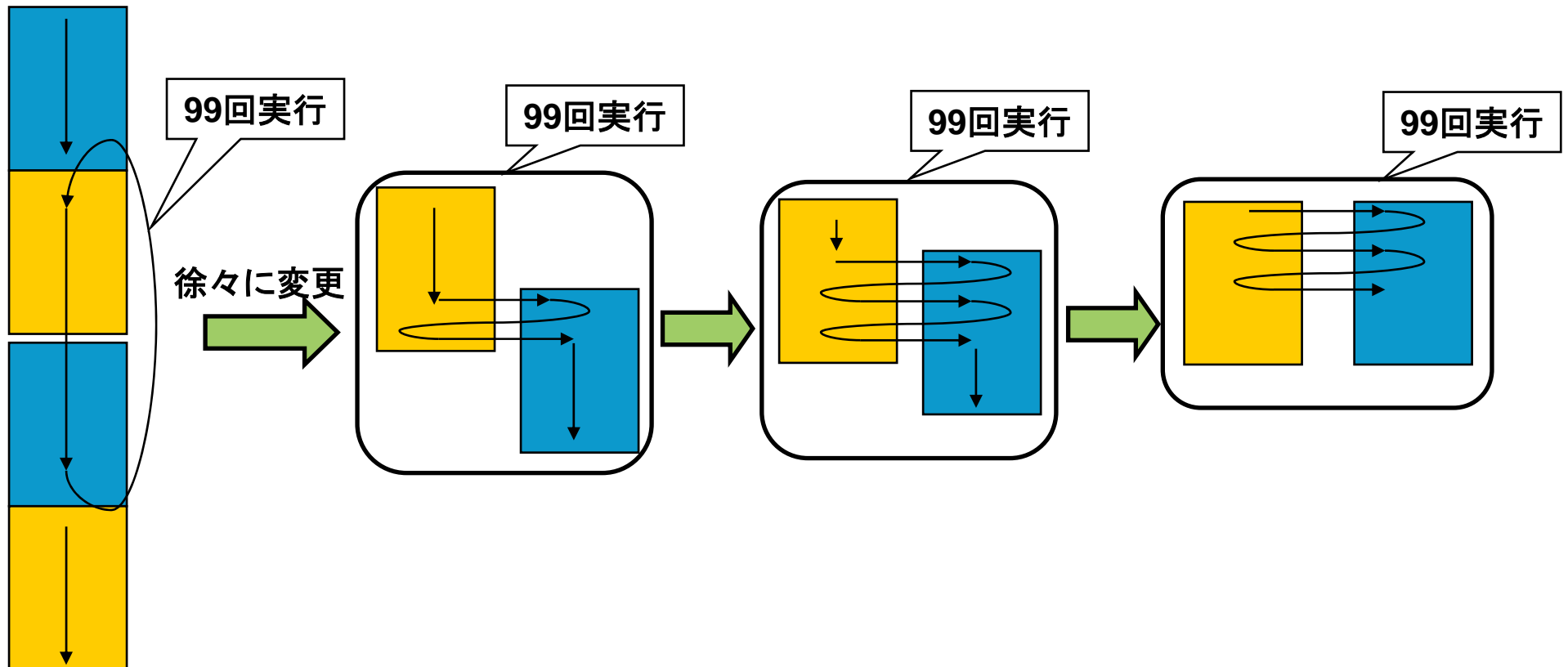
コンパイル
&
実行

一回のループを3cycleで処理

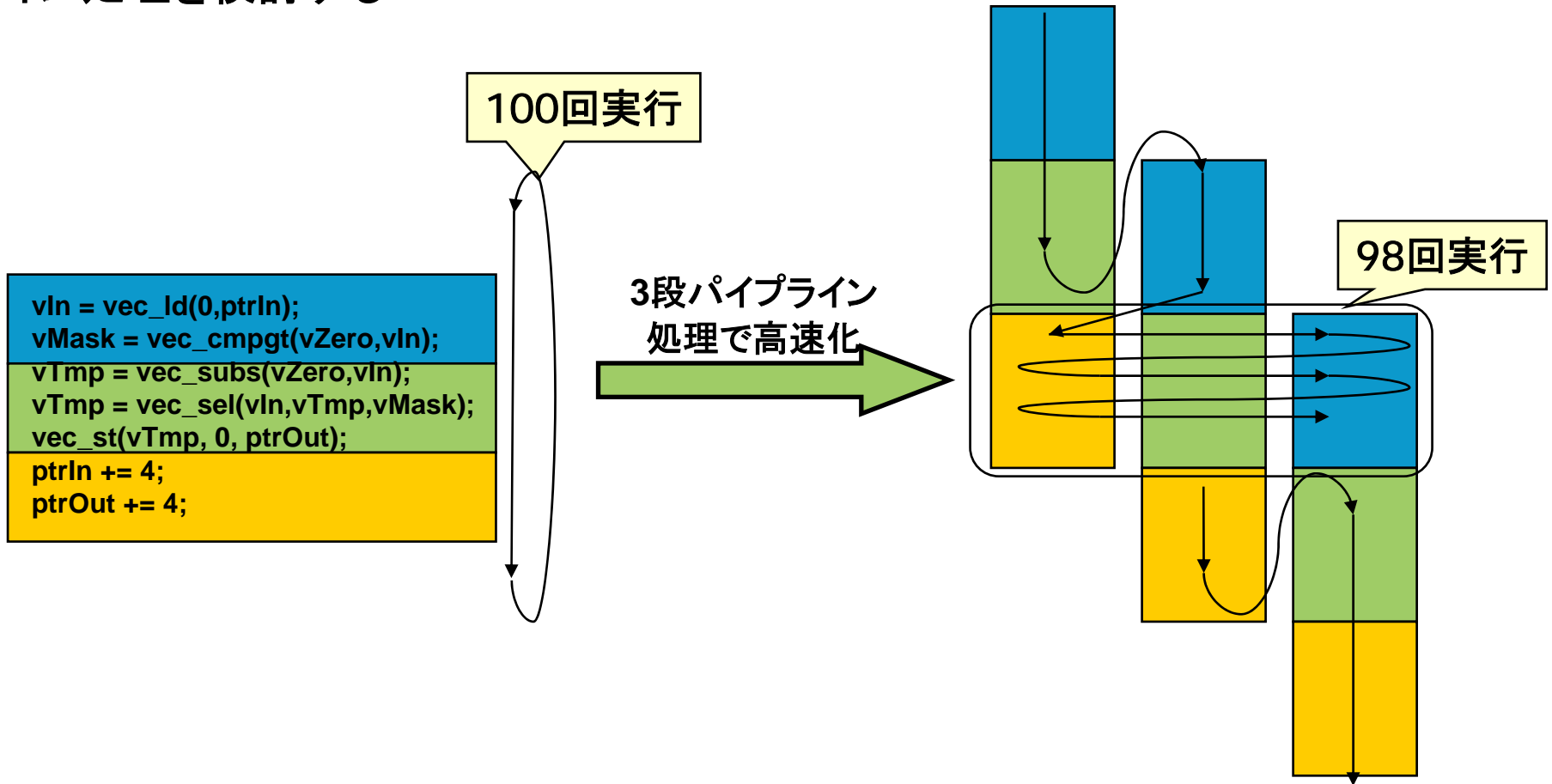


ソフトウェアパイプライン処理④

- ソフトウェアパイプライン処理は、ソフトウェアのバグが発生し易い
- 原因を迅速に見極めるために徐々に変更すると効率的
- 表計算ソフトを使ってプログラムを編集すると以下の作業が容易

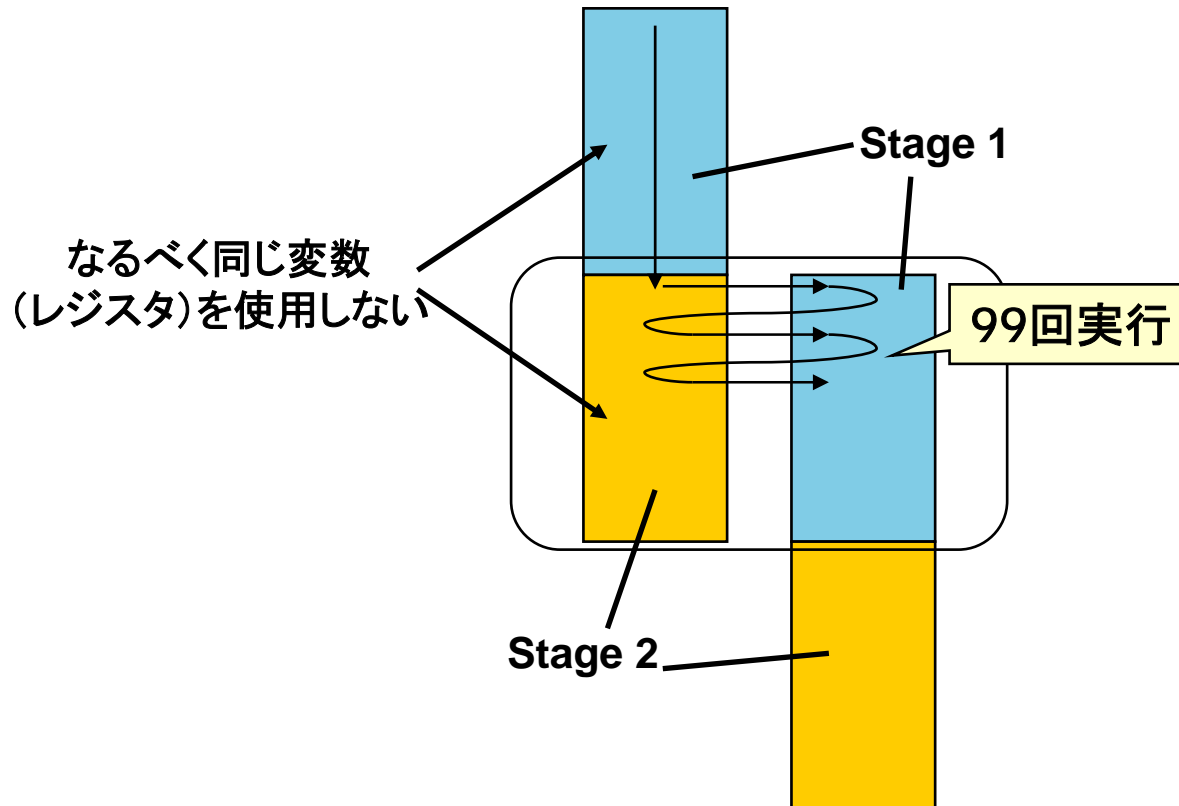


- 2段パイプライン処理で空きスロットを活用しきれなかった場合には、3段のパイプライン処理を検討する



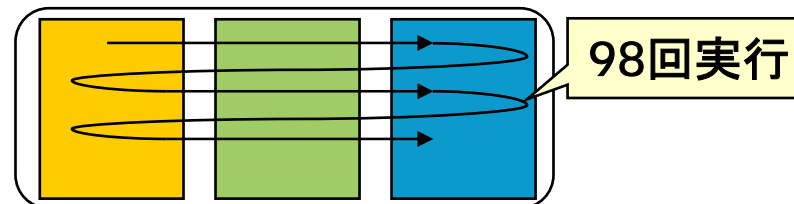
パイプライン処理を導入する際の注意点

- パイプライン処理を行う時は、 n 番目のStage 2と、 $(n+1)$ 番目のStage 1が交互に処理されるため、レジスタの値が誤って変更されないように注意する
- If文やSwitch文に相当する処理をブランチ命令を使って実現すると、この最適化手法が適さなくなるため、パイプラインを乱さない処理方法に変更する



- データパイプライン処理により命令発行スループットの向上が見込めるが、以下の場合にストールが起きるのでループ部には改善の余地がある
 - 同時に発行される命令が同じ実行ユニットになる場合
 - 同時に発行される命令全てがベクトル・ユニットに発行される場合
 - 直前の実行結果が必要な場合

コードレベル最適化③



同時に発行される命令が同じ実行ユニットに偏る場合

コードレベル最適化③

- 実行ユニットを分散させるように命令発行を行う
→全ての命令の情報は、ユーザーズマニュアル(MPC7450 RISC Microprocessor Family User's Manual)の6章(Instruction Timing)で解説
- 同一実行ユニットへの命令発行は、3命令に1回の割合以内で行われるように意識

Mnemonic	Primary	Extend	Form	Unit	Cycles
vand	04	1028	VX	VIU1	1
vperm	04	43	VA	VPU	2:1
vmaddfp	04	46	VA	VFPU	4:1

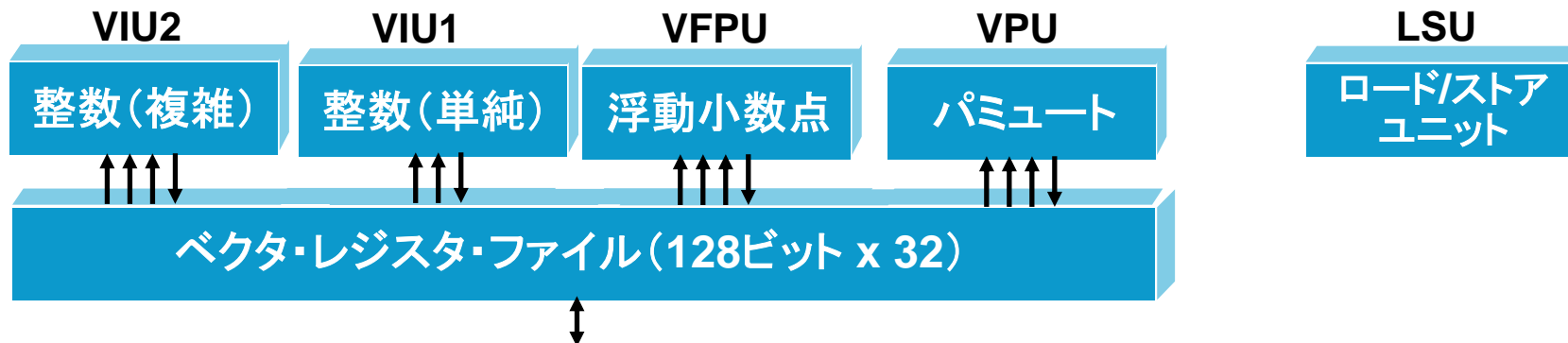
命令発行先の
実行ユニット

最適化の際に
着目すべき項目

同時に発行される命令全てがベクトル・ユニットに発行される場合

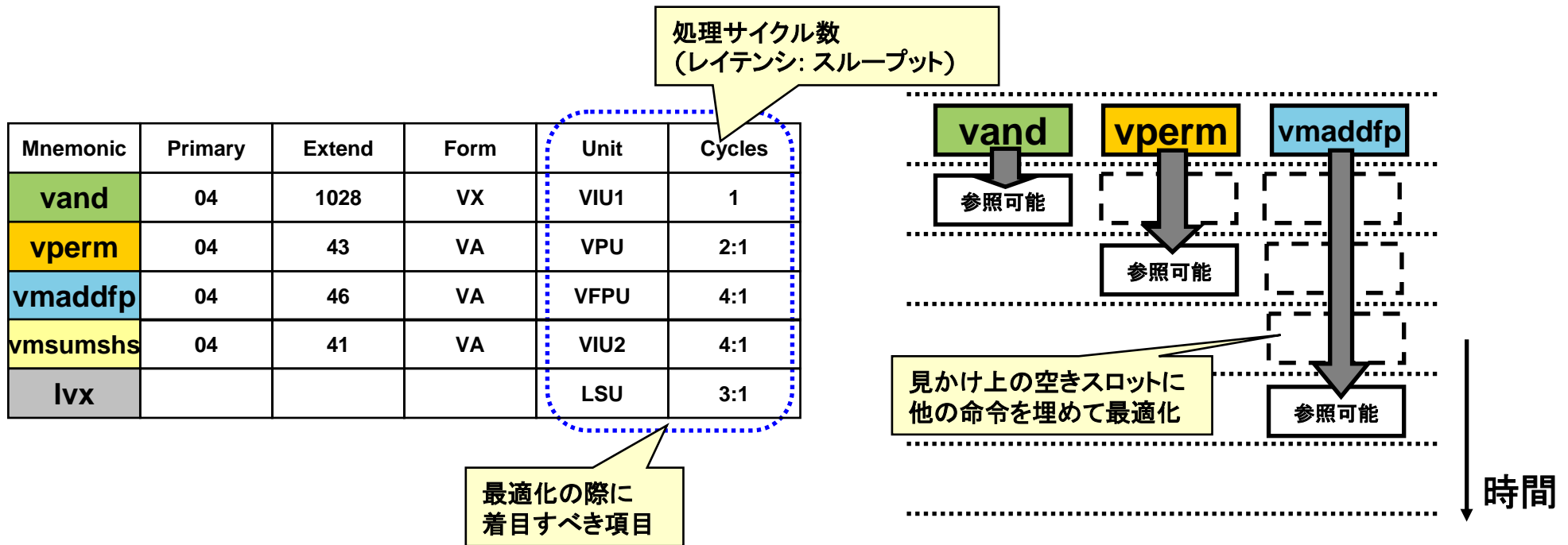
コードレベル最適化③

- 同時に発行できる命令は3つ(分岐命令以外)
- AltiVecとして実装されているベクトル・ユニットは4つ -VIU2,VIU1,VFPU,VPU
- これらベクトル・ユニットに対しては、2命令まで同時命令発行が認められている(制限)



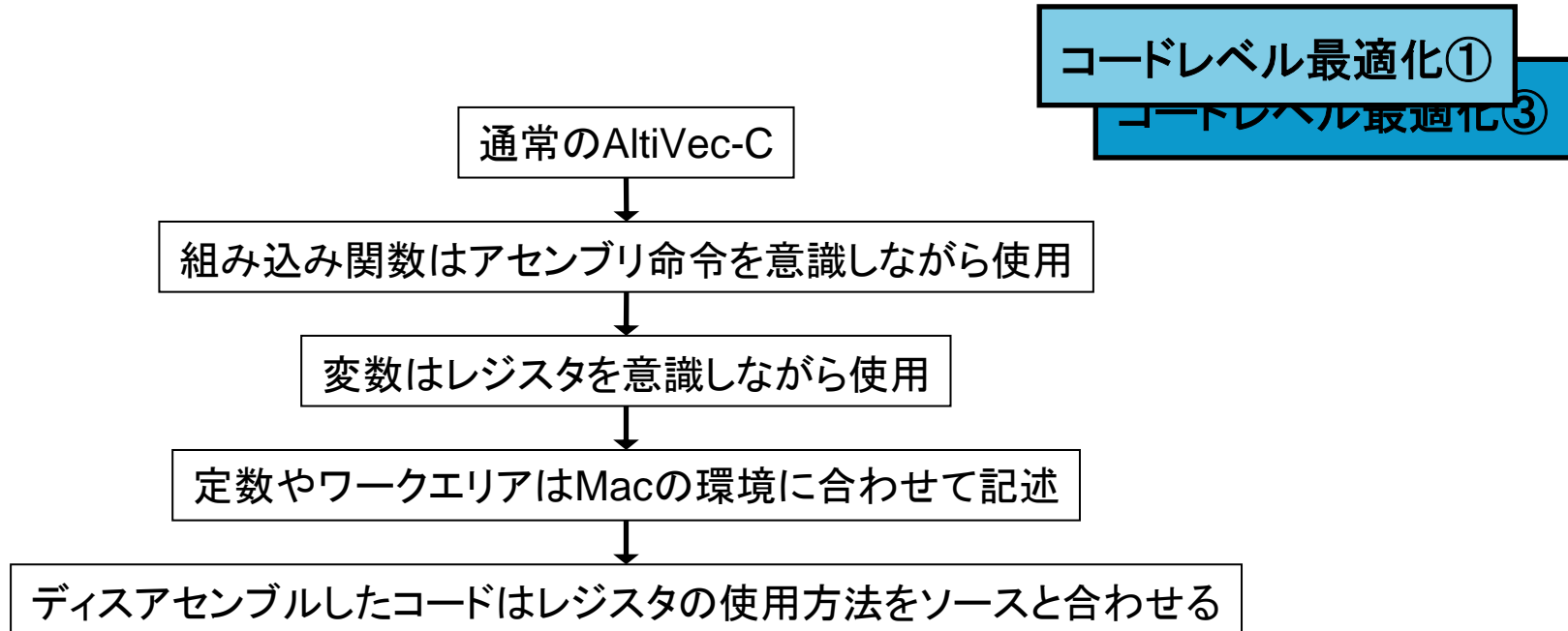
直前の実行結果が必要な場合

- スループットが1サイクルの命令は、依存関係が無い限り見かけ上1サイクルで実行される
- 実行レイテンシが1の命令(例: vand)の演算結果は、直後のサイクルで参照可能
- 実行レイテンシが4の命令(例: vmaddfp)の演算結果は、4サイクル後に参照可能
- ストールを避けるために、実行レイテンシを考慮して命令間の間隔を空ける



アセンブリ言語で関数を記述するメリット

- アセンブリ言語で記述することで、C言語で記述した時に生じる冗長なスタック退避/復元を省略可能になる
- アセンブリ・コーディングを行う場合は、以下のプロセスで行うと効率的



組み込み関数はアセンブリ命令を意識しながら使用

- **Altivec**命令を使用可能にする組み込み関数は、アセンブリ言語と一対一で対応していないものがある
- 一対一で対応している関数のみでインプリメントした場合、後の作業が効率的
- 比較命令と同等の機能は、引数の渡す順番だけ変えると他の関数で代用可能

`vec_abs()` →if文のベクトル処理化参照
`vec_cmp_le()` →`vec_cmpge()`で置き換え
`vec_cmp_lt()` →`vec_cmpgt()`で置き換え

変数はレジスタ数を意識しながら使用

- アセンブリ化を視野に入れたプログラムを作る場合、変数をレジスタ数の範囲で定義して使いまわす

```
unsigned char *pucl nput, *pucOutput;  
vector unsigned char vuc1 nput, vucOutput;  
vector bool char vbcMask;  
  
vuc1 nput = vec_l d(0, pucl nput);  
vbcMask = vec_cmpgt(vuc1 nput, vec_0);  
//真の場合の処理  
vucOutputA = vec_perm(vucTabl e0_15, vucTabl e16_31, vuc1 nput);  
//偽の場合の処理  
vuc1 nput = vec_sub(vuc1 nput, vec_32);  
vucOutput = vec_perm(vucTabl e32_47, vucTabl e48_63, vuc1 nput);  
//結果の選択  
vucOutput = vec_sel (vucOutputA, vucOutputB, vbcMask);  
vec_st(vucOutput, 0, pucOutput);
```

実装されているベクタレジスタ数
以内で変数を定義するv0-v31

```
unsigned char *pucl nput, *pucOutput;  
vector unsigned char v0, v1;  
vector bool char v2;  
  
v0 = vec_l d(0, pucl nput);  
v2 = vec_cmpgt(v0, v30);  
//真の場合の処理  
v1 = vec_perm(v15, v16, v0);  
//偽の場合の処理  
v0 = vec_sub(v0, v31);  
v1 = vec_perm(v17, v18, v0);  
//結果の選択  
v1 = vec_sel (v0, v1, v2);  
vec_st(v1, 0, pucOutput);
```

```
unsigned char *pucl nput, *pucOutput;  
vector unsigned char v0, v1;  
vector unsigned char v2;  
  
v0 = vec_l d(0, pucl nput);  
v2 = (vector unsigned char)vec_cmpgt(v0, v30);  
//真の場合の処理  
v1 = vec_perm(v15, v16, v0);  
//偽の場合の処理  
v0 = vec_sub(v0, v31);  
v1 = vec_perm(v17, v18, v0);  
//結果の選択  
v1 = vec_sel (v0, v1, (vector bool char)v2);  
vec_st(v1, 0, pucOutput);
```

長いソースの場合、変数v0は
使いまわすために定義する
ときではなく、使用するとき
に型を合わせる

定数やワークエリアはMacOSの環境に合わせて記述

- MacOS環境で開発するアプリケーションは、アセンブリ言語で記述する時に絶対アドレスを指定することができない
- アセンブリ言語で開発する際には、必要なデータ、ワーク領域は上位の関数内で全て配列に入れておき、配列の引数をサブ関数(Asm_func)に渡すことで実現する

```
Func()  
{  
    vector float Array[100]; // ワーク領域  
    vector float vConst = (vector float)( 1.0 , 2.0 , 3.0 , 4.0 );  
}
```

この関数のままアセンブリ化すると、これらのパラメータにアクセスできない

```
Func()  
{  
    vector float vec[101]; // vec[0]-vec[99]->ワーク領域  
    v[100] = (vector float)( 1.0 , 2.0 , 3.0 , 4.0 );  
    Asm_func((unsigned char*)vec);  
}
```

サブ関数を用意し、関数をコールする前にアセンブリ言語で記述できない処理を吸収する

ディスアセンブルしたコードのレジスタ参照方法をソースと合わせこむ

- コンパイラが生成するアセンブリ・コードは、ソースと参照するレジスタが異なるので、デバッグを容易にするために合わせておく
- コンパイラが生成した余計なスタック退避/復元処理を省く

基本形

```
gcc -S func.c
```

gccの場合、-S のコンパイラ・オプションを付けることでアセンブリ・コード(*.s)を生成可能

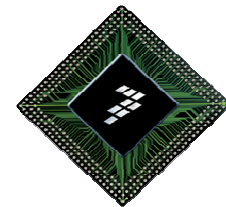
他のオプションとの併用

```
gcc -faltivec -S -O2 func.c
```

AltiVecコードをコンパイルするためのオプション

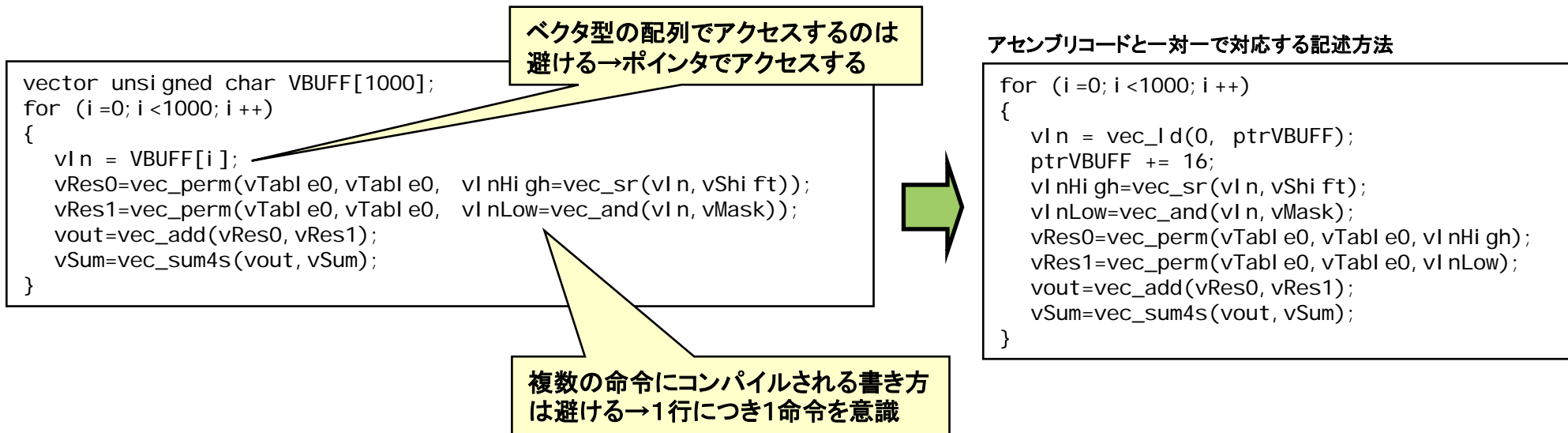
最適化を行う場合のオプション

限界性能の見極めと改善



アセンブリコードを意識したC言語の記述方法

- コンパイルされた時に、アセンブリコードと対応が取り易くなるプログラムを記述すると以下のメリットがある
 - ソフトウェアパイプライン処理を導入した時に効率の良いコンパイル結果が見込める
 - 性能を限界まで引き上げることを目的としたアセンブリ言語の導入が容易
 - 限界性能の見極めが容易



- 最小の命令発行数で処理が実現できた後、そのプログラムの限界性能を見積もる
- 通常、プログラムの処理時間は特定のループ処理によって決定付けられることが多い
- PowerPC G4は、クロック当たり3命令発行することが可能であることを考慮する

```
for (i=0; i<1000; i++)  
{  
    vl n = vec_ld(0, ptrIn);  
    ptrIn += 16;  
    vl nHi gh=vec_sr(vl n, vShift);  
    vl nLow=vec_and(vl n, vMask);  
    vRes0=vec_perm(vTabl e0, vTabl e0, vl nHi gh);  
    vRes1=vec_perm(vTabl e0, vTabl e0, vl nLow);  
    vout=vec_add(vRes0, vRes1);  
    vSum=vec_sum4s(vout, vSum);  
}
```

理想的にコンパイルされれば8命令で処理される

3命令/クロックのペースで命令発行されれば、理想的な実行サイクルは、約3サイクルとなる(8命令÷3)

プログラムを実行して実時間を計測

ループ以外の処理時間を無視すると理想的な全体の処理時間は
3000ns(3サイクル×1000(ループ回数)×1ns(@1GHz))

実際の処理時間と理想的な処理時間の間に大きな差がある場合は以下の遅延要因が考えられる

- メモリアクセスに伴うストール(例:外部メモリアクセス)
- 3命令同時に発行されていない
- コンパイラが冗長なコードを生成する(レジスタのPUSH/POP等)

遅延要因を減らして目標の理想的な処理時間に限りなく近づける

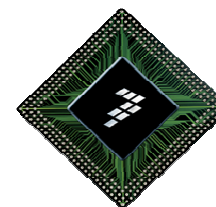
- 別要素によるボトルネックが、前頁で解説した理想的な処理時間を実現する妨げになる場合がある
- ボトルネックを正確に把握することで、以下のメリットがある
 - 改善すべき項目が分かる
 - 高速化のゴール設定が正しくできる
- ボトルネックとなっている項目から優先的に改善することで、効率良く開発を進めることが可能

外部メモリアクセス依存のボトルネック

- 外部メモリアクセスでボトルネックになっている時に、コア内部の処理をいくら高速化しても外部メモリアクセスの速度以上の向上は見込めない
- 改善策
 - キャッシュ⇔外部メモリ間のデータ転送量を必要最小限に留めることを目指す
 - テーブル参照で行っている処理を計算で求める
 - 中間値のサイズがL2キャッシュサイズを超えないように設計する
- 高速化のゴール設定
 - 改善を行ったプログラムについて、演算処理を省き、データ転送処理だけ残したプログラムを用意する
 - `dcbt`, `dcbz`, `dcbf` 命令等を用いて、高速化を図る
 - 得られた結果が、メモリアクセスがボトルネックになる場合の限界性能となる

- 基本的に演算ユニットは1種類ずつ用意されている(整数演算ユニット内の3つの“シンプル専用”のみが同一機能を保有)
- ループ内部で特定の演算ユニットへ命令発行が集中している場合にソフトウェアパイプライン処理を導入しても、「理想的な実行サイクル=命令数÷3」とはならない
- 改善策
 - 別の演算ユニットを用いる命令を用いて等価な処理を実現する
 - テーブル参照で求める代わりに計算で求める(テーブル参照処理でロードストアユニットに命令発行が集中している場合)
- 高速化のゴール設定
 - 最も使用される演算ユニットが全体の1/3以上を占めるのであれば、「理想的な実行サイクル=最も使用される演算ユニットへ発行される命令数」となる
 - コア内部で行う限界性能を正確に把握可能になる

開発ツールの活用方法



ベクタ型変数の各要素へのアクセス方法

- ベクタ型の変数には複数のデータがまとめて格納されている
- ベクタ型変数内の各要素にアクセスするには以下の方法が有効

```
typedef union
{
    vector unsigned int vec;
    unsigned long element[4];
}ULvector;

typedef union
{
    vector unsigned short vec;
    unsigned short element[8];
}USvector;

typedef union
{
    vector unsigned char vec;
    char element[16];
}Cvector;

ULvector ULVa;
unsigned long 3rdElement;

ULVa.vec = vec_id(0, pt_input);
3rdElement = ULVa.element[2];
```

Unionを用いて用途に合わせて型を定義する

Char型であれば1ベクタにつき16要素が含まれる

①ベクタ型の変数に16バイトまとめてデータを転送する

②転送されたデータについて3番目のデータだけを使用可能

はじめてのAltiVecコーディング

- AltiVecコーディングに慣れるまでは以下に示す手順でコーディングを行うことでデバッグ作業の負担が軽い開発が可能

オリジナルのCコード

それぞれのステップで検証を行うことでデバッグにかかる時間を短縮

AltiVecを使用しないで並列化を意識したCコード

部分的にAltiVec命令が使用されているCコード

全般的にAltiVec命令が使用されているCコード

```
do
{
    func( *ptrChn++ );
} while
```

```
do
{
    func( *ptrChn++ );
    func( *ptrChn++ );
    func( *ptrChn++ );
    func( *ptrChn++ );
} while
```

```
do
{
    ULVa. vec = vec_Id(0, ptrChn);
    func(ULVa. ele[0]);
    func(ULVa. ele[1]);
    func(ULVa. ele[2]);
    func(ULVa. ele[3]);
    ptrChn += 16;
} while
```

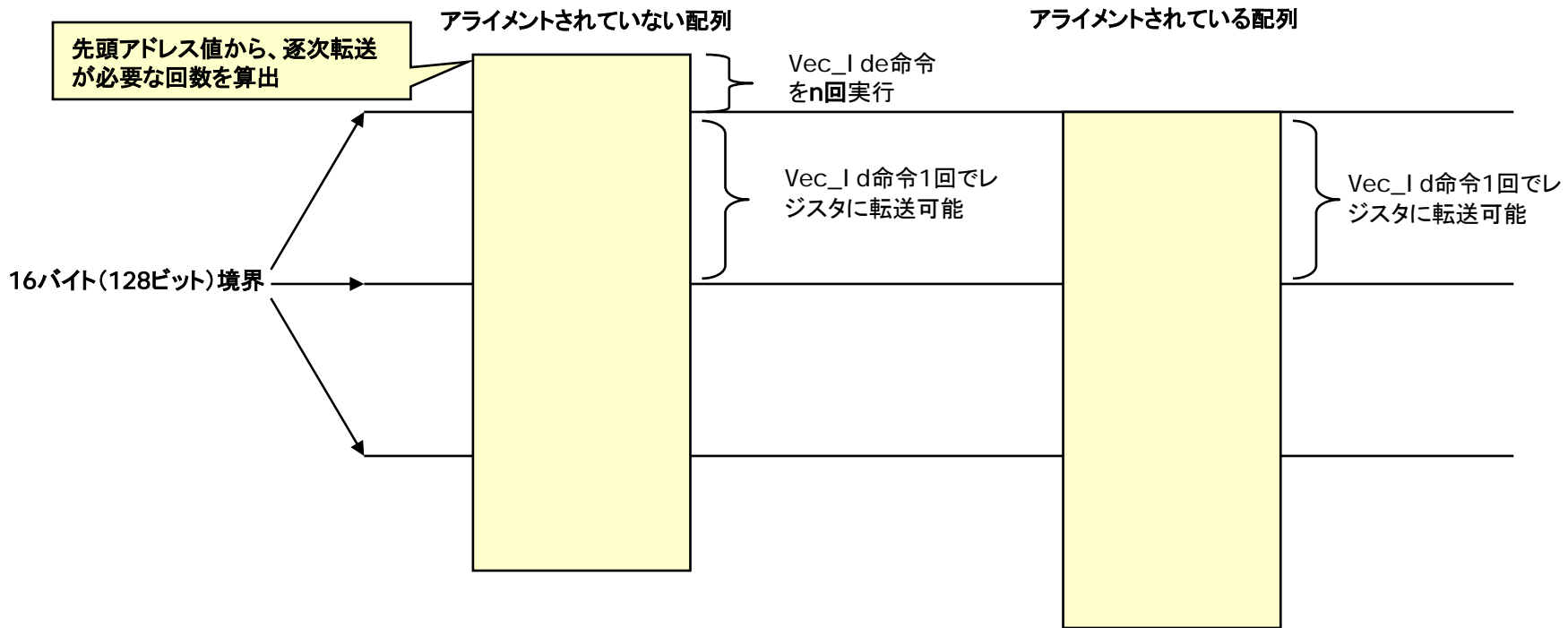
データ・アクセスだけAltiVec命令を活用

```
do
{
    ULVa. vec = vec_Id(0, ptrChn);
    vec_func(ULVa. vec);
    ptrChn += 16;
} while
```

ユニオン型のベクタ型で定義された処理はコンパイル効率が悪くなるので最終的には通常のベクタ変数に書き直した方がよい

ベクタ化された関数に置き換え

- AltiVecのロード/ストア命令を使うためには、処理対象のデータが配置されているアドレスが16バイト境界にアライメントされている必要がある
- 16バイト境界に配置されていない場合、端数部分のデータに対しては逐次処理等の工夫が必要になる



- 以下のフォーマットで定義した場合、配列V1の先頭アドレスが16バイトにアライメントされる

```
signed long int V1[4] __attribute__ ((aligned (16)));
```

- 以下のフォーマットで定義した場合、配列V2の先頭アドレスが32バイトにアライメントされ、キャッシュ制御命令の適用が容易になる

```
signed long int V2[4] __attribute__ ((aligned (32)));
```

- ベクタ型で変数を定義すると、実メモリ上では自動的に16バイト境界にアライメントされる
- ヒープ領域は16バイトアライメントされており、基本的にMalloc()で定義された領域も自動的に16バイト境界にアライメントされるが、アライメントされない環境もあるので、次ページで解説する方法を使うと汎用的にアライメントを保証できる

mallocと32バイトアライメント

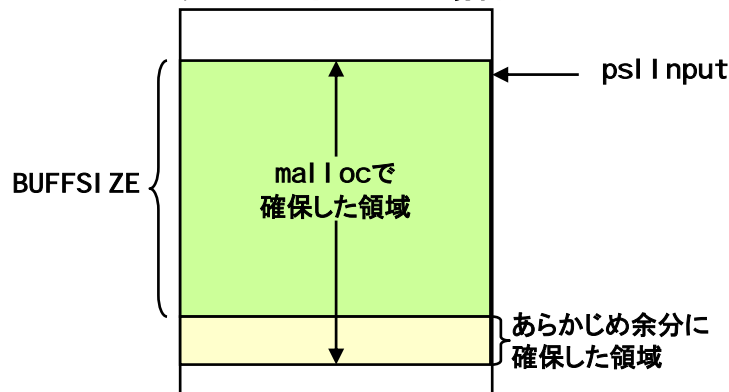
- dcbf、dcbz等のキャッシュ制御命令を活用するには、32バイトアライメントを保証すると適用し易くなる
- malloc()を用いてメモリ領域を確保した場合、32バイトでアライメントされるとは限らないためmalloc()を用いる場合に32バイトアライメントを実現する例を以下に示す

```
pslInputBase = (signed long*)malloc(31+sizeof(long)*BUFFSI ZE);  
pslInput = ((unsigned long)pslInputBase+31)/32)*32;
```

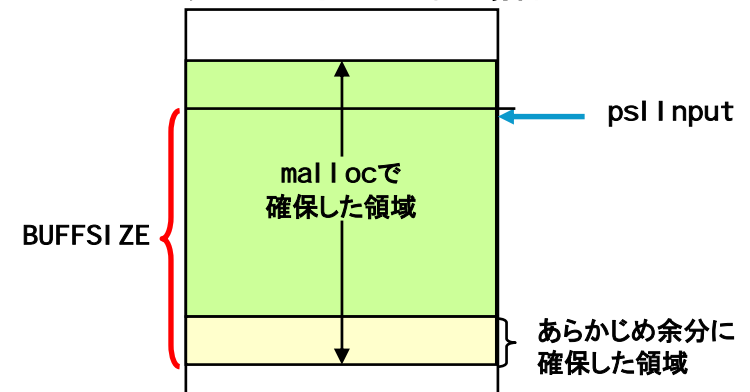
ここでは31バイト分だけ余分にメモリ領域を確保する

このポインタが指し示すアドレスは32バイトアライメントが保証された

先頭アドレスが32バイトでアライメントされている場合



先頭アドレスが32バイトでアライメントされていない場合



- gccコンパイラを使用してインライン・アセンブラで関数を記述する方法を以下に示す
- このフォーマットを用いてdcbz命令等を使ったハードウェア特有の機能を使用した最適化を行うことが可能

例1)

Codewarrior `__dcbz(dst, 0);`

gcc `__asm __volatile ("dcbz 0, %0" : : "r" (dst));`

例2)

Codewarrior `__dcbt(ptrIn, 32);`

gcc `__asm __volatile ("dcbt %0, %1" : : "r" (ptrIn), "r" (32));`

インライン・アセンブラを使ったサイクル・カウント測定

- gccコンパイラを使用する場合に、パフォーマンスを評価するには、コアに実装されているカウンタ機能をインライン・アセンブラを用いて利用することで簡単に測定することが可能
- タイム・ベース・レジスタは、バスに同期して4バス・サイクル毎に1ずつインクリメントされる。
- 以下のサンプルのように始点と終点の値で差を4倍すれば、バス・サイクルで実時間を測定することが可能

```
unsigned int starttime, endtime;

//終点のカウンタ値取得
__asm __volatile("mftb %0" : "=r"(starttime) : );

//任意の測定対象アプリケーションの実行
func();

//終点のカウンタ値取得
__asm __volatile("mftb %0" : "=r"(endtime) : );

//測定区間のサイクル数を表示
fprintf( stdout, "Bus cycle count = [%d] \r\n", 4*(endtime - starttime));
```

アセンブリ言語で記述する場合の関数フォーマット

Main.c

```
extern unsigned long tickGet( void );  
unsigned long start_time, end_time;  
  
start_time = tickGet();  
func();  
end_time = tickGet();
```

Codewarrior for Embedded PPC

用のフォーマット

tickGet.s

```
.text  
.align 2  
.global  
tickGet:  
mftb r3  
blr
```

gcc用のフォーマット

tickGet.s

```
.text  
.align 2  
.globl  
_tickGet:  
mftb r3  
blr
```

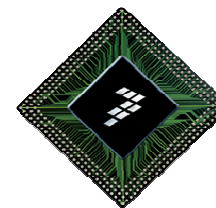
各ツールの正確なアセンブリ
フォーマットは、マニュアルで確
認する

ツールでディスアセンブラを行って
フォーマットを確認するのも有効

アセンブリ言語を利用することの弊害と対応

- 特定の関数にアセンブリ言語を利用した場合、その関数の中では最大の効果を上げることができる反面、性能面でデメリットが生じる場合がある。
- Altivec命令は、Altivec専用のレジスタを使用するため、通常のスカラー処理に比べて関数呼び出しの前後で生じるスタック退避/復元処理にかかる時間が多い。
- そのため、アセンブリ言語を使用するよりも、イントリンシック関数を使ってC言語フォーマットで記述し、上位の関数内でインライン展開させてAltivec専用レジスタのスタック退避/復元処理を省略させた方が性能アップを図れる場合がある。
- 例えば、関数内においてスタック退避/復元のオーバーヘッドが相対的に無視できない程多ければインライン展開する価値がある。
- 全てのベクタレジスタについて復元と退避を行うためには64回のロード/ストア命令が必要になることを念頭に入れておく。

最適化プログラミング・テクニックの応用例

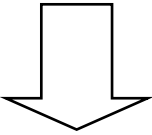


データのコヒーレンシを保つ手法の最適化を検討

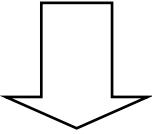
外部メモリを2つ以上のプロセッサで共有



G4プロセッサからは、データのコヒーレンシを保つために、共有エリアにはキャッシュ禁止モードでアクセス



キャッシュ禁止で設定されている外部メモリに対するアクセスは、バースト転送が行われない



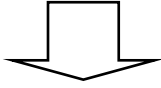
効率の悪いシングルビートアクセスになるため転送帯域を使いきれない

キャッシュ禁止モードを使わない方法で高速化

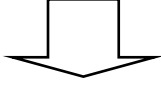
外部メモリを2つのプロセッサで共有



データプリフェッチとバースト転送を使うために、共有エリアもキャッシュ可能領域(ライトバックモード)で使用する



G4プロセッサからは、データのコヒーレンシを保つために、ソフトウェアで明示的にキャッシュ内データの消去と掃き出しを行う



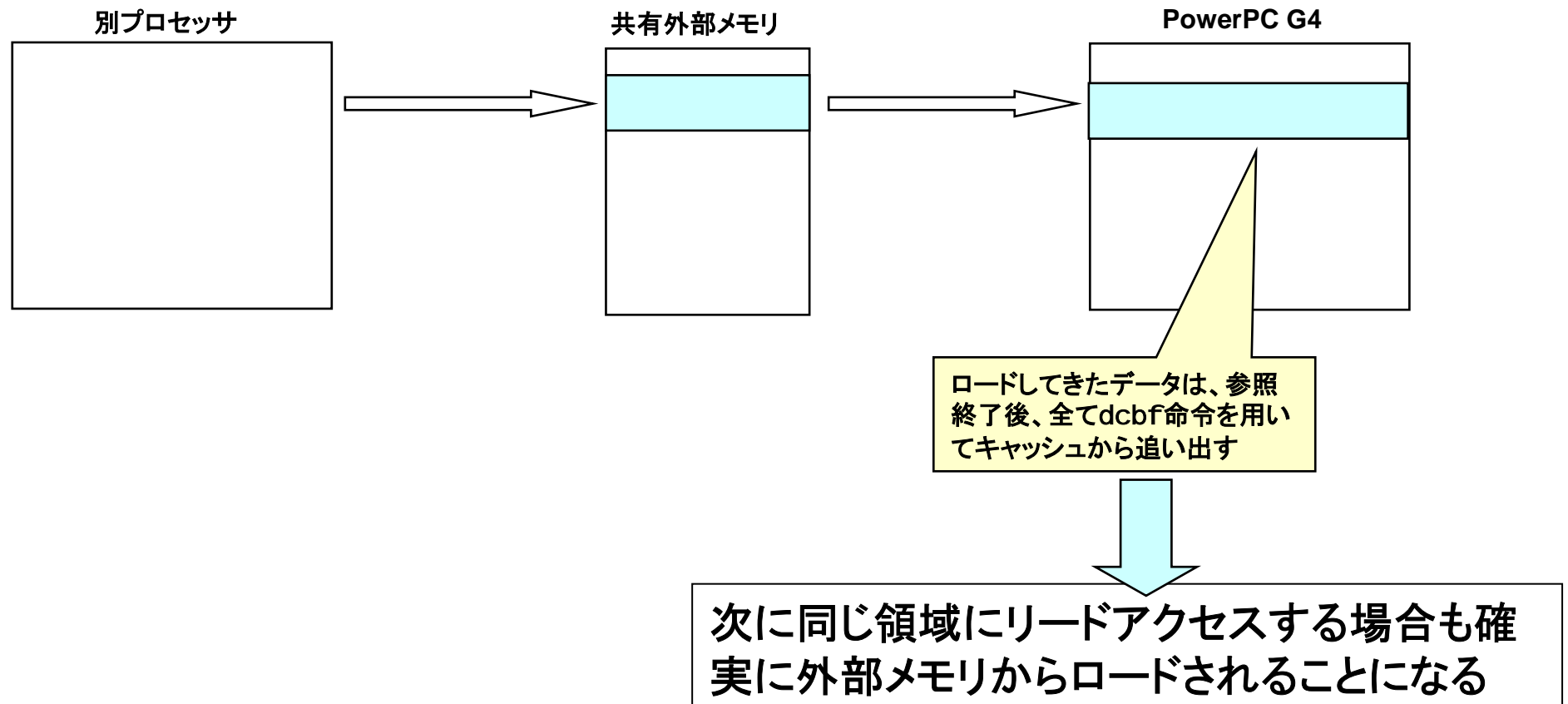
G4プロセッサと外部メモリ間は常にバースト転送で送られる



転送帯域を最大限に活用できて性能が向上する

キャッシュ可能でコヒーレンシを保つ工夫(受信の場合)

PowerPC G4側で受信する場合



dcbf命令を使ったプログラミング(受信の場合)

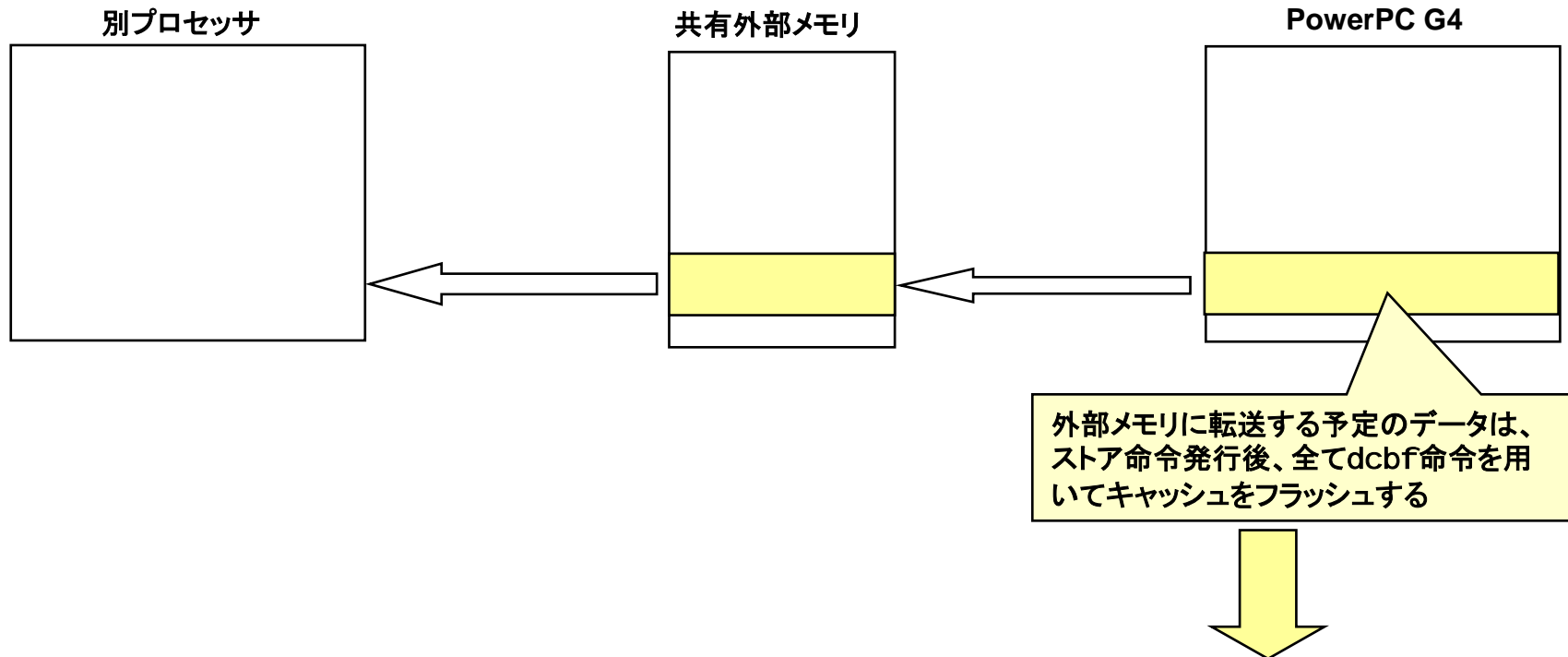
- dcbf(データ・キャッシュ・ブロック・フラッシュ)命令を用いることで指定されたキャッシュ・ブロックの最新データがメモリにあり、キャッシュにないことを保証する

```
vector unsigned char t0,t1;
ptrIn = (unsigned char*)InputBuf;
ptrOut = (unsigned char*)OutputBuf;
for(i=0;i<250;i++)
{
    t0 = vec_ld(0, ptrIn);
    t1 = vec_ld(16, ptrIn);
    __dcbf(ptrIn, 0);
    ptrIn += 32;
    vec_st( t0, 0, ptrOut);
    vec_st( t1, 16, ptrOut);
    ptrOut += 32;
}
```

ptrIn番地が含まれるキャッシュライン(32バイト)をフラッシュ

キャッシュ可能でコヒーレンシを保つ工夫(送信の場合)

PowerPC G4側で受信する場合



外部メモリに送り出したいデータは確実に送られる

dcbf命令を使ったプログラミング(送信の場合)

- 送信の場合も、受信の場合と同様にdcbf命令を用いることになる
- このdcbf命令により明示的にデータがキャッシュから外部メモリに送り出される

```
vector unsigned char t0,t1;
ptrIn = (unsigned char*)InputBuf;
ptrOut = (unsigned char*)OutputBuf;
for(i=0;i<250;i++)
{
    t0 = vec_ld(0, ptrIn);
    t1 = vec_ld(16, ptrIn);
    __dcbf(ptrIn, 0);
    ptrIn += 32;
    vec_st( t0, 0, ptrOut);
    vec_st( t1, 16, ptrOut);
    __dcbf(ptrOut, 0);
    ptrOut += 32;
}
```

ptrOut番地が含まれるキャッシュライン(32バイト)を無効化し、変更されているデータをキャッシュから外部メモリに転送

テーブル参照(32エントリ)の高速化

コードレベル最適化①

コードレベル最適化③

最適化前

```
unsigned char *pucl nput;  
unsigned char *pucOutput;  
unsigned char ucTabl e[32]={31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};  
  
for(i =0, i <16, i ++)  
{  
    *pucOutput = ucTabl e[*pucl nput];  
    *pucOutput++;  
    *pucl nput++;  
}
```

Al ti Vec命令で最適化

最適化後

```
unsigned char *pucl nput;  
unsigned char *pucOutput;  
vector unsigned char vuc1 nput;  
Vector unsigned char vucOutput;  
vector unsigned char vucTabl e0_15=(vector unsigned char) (31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16);  
vector unsigned char vucTabl e16_31=(vector unsigned char) (15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0);  
  
vuc1 nput = vec_l d(0, pucl nput);  
vucOutput = vec_perm(vucTabl e0_15, vucTabl e16_31, vuc1 nput);  
Vec_st(vucOutput, 0, pucOutput);
```

8ビットサイズで32エントリの
テーブルであれば2つのベク
タ・レジスタに納まる

インデックス
0

インデックス
15

インデックス
16

インデックス
31

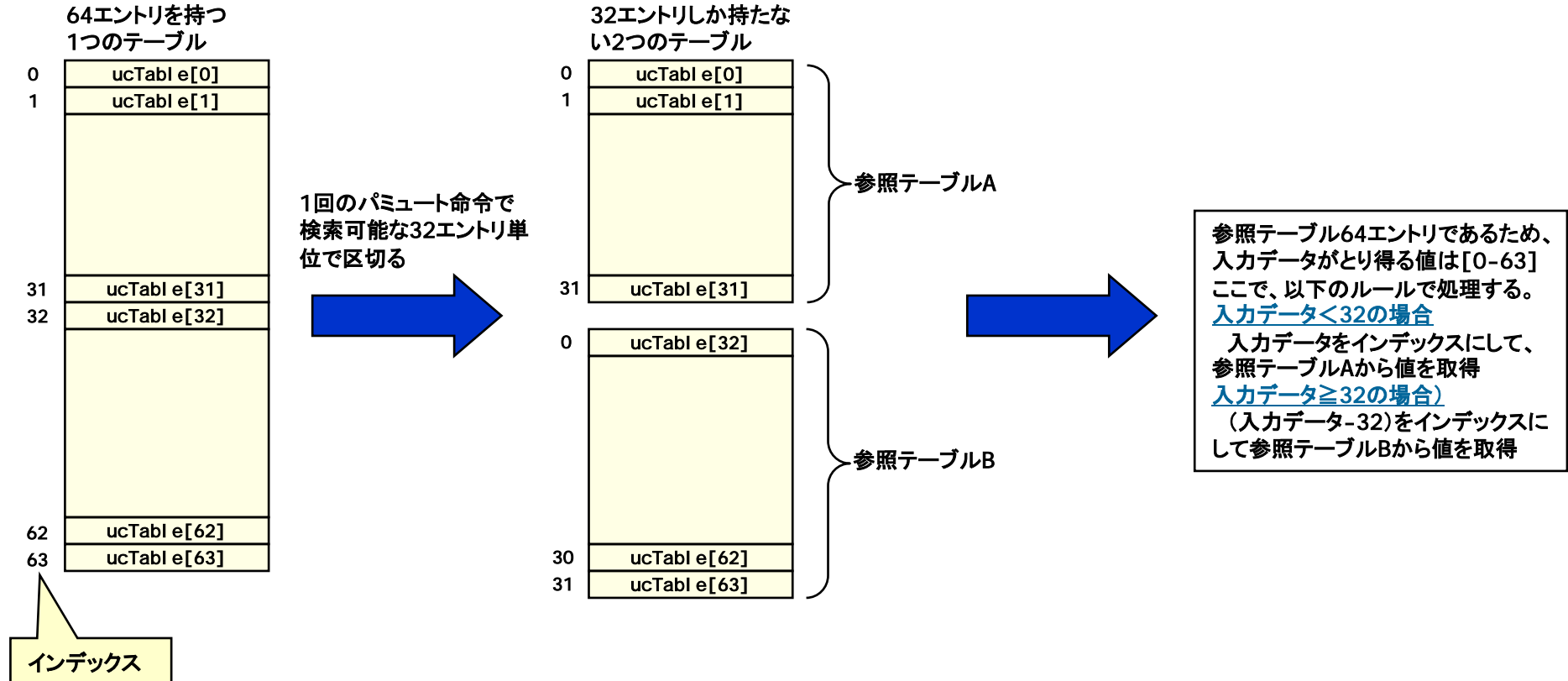
16個分の入力データを1
サイクルで実行!

テーブル参照(64エン트리)の高速化

コードレベル最適化①

コードレベル最適化③

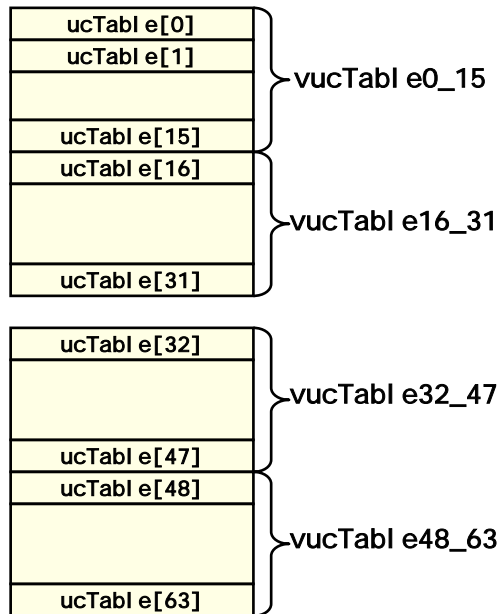
- 2つの32エン트리・テーブルに分割します。



テーブル参照(64エントリ)の高速化(続き)

- 2つのテーブルそれぞれについて計算し、vec_sel命令で結果を選択する

コードレベル最適化③



```
unsigned char *puclnput;
unsigned char *pucOutput;
vector unsigned char vucInput;
vector unsigned char vucOutput;
vector bool char vbcMask;
vector unsigned char vucTable e0_15=(vector unsigned char) (省略);
vector unsigned char vucTable e16_31=(vector unsigned char) (省略);
vector unsigned char vucTable e32_47=(vector unsigned char) (省略);
vector unsigned char vucTable e48_63=(vector unsigned char) (省略);
vector unsigned char v32 =
(vector unsigned char) (32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32);

vucInput = vec_ld(0, puclnput);
vbcMask = vec_cmpgt(vucInput, v32);
//真の場合の処理
vucOutputA = vec_perm(vucTable e0_15, vucTable e16_31, vucInput);
//偽の場合の処理
vucInput = vec_sub(vucInput-v32);
vucOutputB = vec_perm(vucTable e32_47, vucTable e48_63, vucInput);
//結果の選択
vucOutput = vec_sel(vucOutputA, vucOutputB, vbcMask);
vec_st(vucOutput, 0, pucOutput);
```

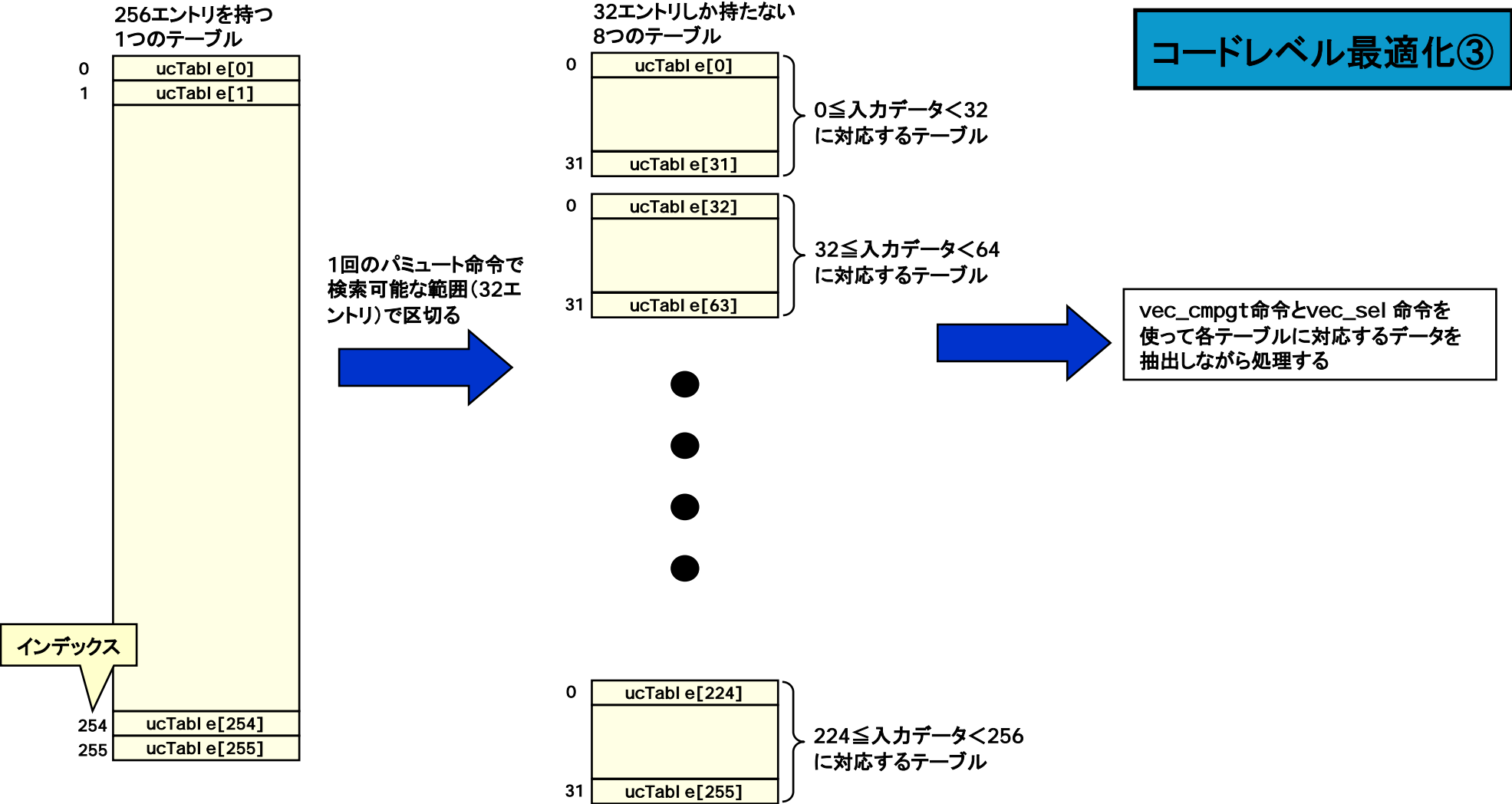
テーブルのデータは、vec_ld命令で配列からアクセスする方法も可能

次のパミュート命令における3つ目の引数は下位5ビットしか見ないので省略可能

64エントリのテーブル参照についても、ベクタレジスタに格納されている16個の入力データを同時に処理することが可能

256エントリのテーブル参照への応用

- 8つのテーブルそれぞれについて計算し、vec_sel命令で結果を選択する



256エントリのテーブル参照プログラム

最適化前

```
unsigned char *pucInput;  
unsigned char *pucOutput;  
extern unsigned char ucTable[256];  
for(i=0, i<16, i++)  
{  
    *pucOutput = ucTable[*pucInput];  
    *pucOutput++;  
    *pucInput++;  
}
```

コードレベル最適化③

最適化後

```
unsigned char *pucInput;  
unsigned char *pucOutput;  
extern unsigned char ucTable[256];  
#include "vec_lut.h"  
  
Vec_lut(pucInput, pucOutput);
```

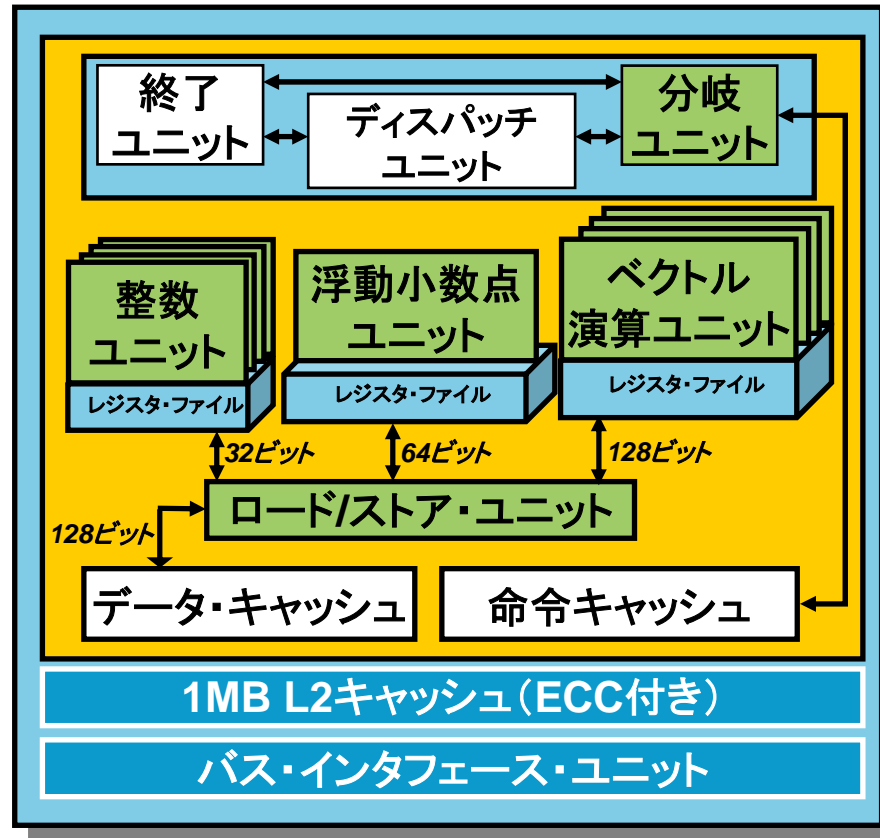
インライン展開
で高速処理

vec_lut.h

```
inline void vec_lut(unsigned char *pfrmIn, unsigned char *pfrmOut)  
{  
    vector unsigned char v0, v1, v2, v3;  
    vector bool char v4;  
    vector unsigned char v5 = (vector unsigned char)( 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32);  
    int i;  
    unsigned char *pucTable1;  
  
    v3 = vec_splat_u8(0); // v3 = 0 (constant)  
    pucTable1 = (unsigned char *)refTable;  
    v0 = vec_splat_u8(0);  
    v1 = vec_ld( 0, pfrmIn);  
    for(i = 0; i < 8; i++)  
    {  
        v4 = vec_cmpgt( v5, v1);  
        v2 = vec_perm( vec_ld( 0, pucTable1 ), vec_ld( 16, pucTable1 ), v1);  
        pucTable1 += 32;  
        v2 = vec_sel( v3, v2, v4 );  
        v0 = vec_xor( v0, v2 );  
        v1 = vec_sub( v1, v5 );  
    }  
    vec_st( v0, 0, pfrmOut );  
}
```

非効率なユニット間転送

- PowerPC G4は、演算ユニットごとに独立したレジスタセットを保有するが以下のメリット・デメリットを持つ
 - 豊富なレジスタが用意されているので、余計なスタック-レジスタ間転送(PUSH/POP)が発生しにくい
 - ユニートをまたぐレジスタ間転送は、スタック(L1データキャッシュ)経由となり効率が悪い



エン트리数の多いテーブル参照処理の最適化①

- テーブル参照処理にAltiVecを使うことで高速化が期待できるのは256エン트리(入力:8ビット)まで
- 2^{16} のエン트리があるテーブル参照ではAltiVecの効果は期待できない(総当りで処理するには多すぎる!)
- ベクタ変数の各要素を使ってテーブル参照(入力:16ビット)を行う場合、通常通り整数ユニットのレジスタにデータを移動してストア命令で実現することになる

```
typedef union
{
    vector unsigned short vec;
    unsigned short ele[8];
}USvector;

USvector USVI n, USV0ut;
extern unsigned short Table[];
USVI n. vec = vec_ld(0, ptrInput);

USV0ut. ele[0] = Table[USVI n. ele[0]];
USV0ut. ele[1] = Table[USVI n. ele[1]];
USV0ut. ele[2] = Table[USVI n. ele[2]];
USV0ut. ele[3] = Table[USVI n. ele[3]];
USV0ut. ele[4] = Table[USVI n. ele[4]];
USV0ut. ele[5] = Table[USVI n. ele[5]];
USV0ut. ele[6] = Table[USVI n. ele[6]];
USV0ut. ele[7] = Table[USVI n. ele[7]];

v0 = vec_add(USV0ut. vec, v1);
```

アセンブリ言語に一つ一つ
で対応する記述に変更
→性能は同等

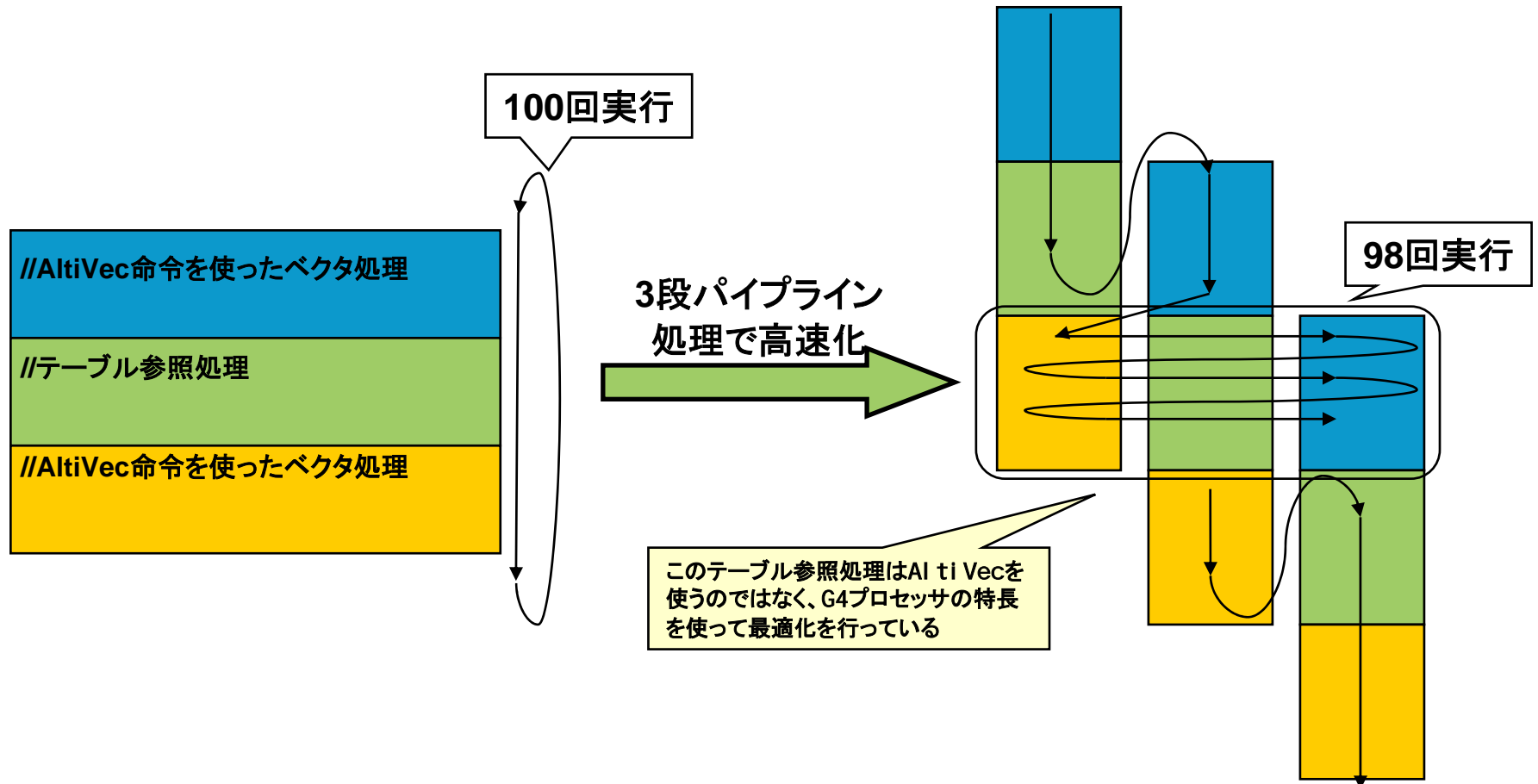
```
unsigned short ptrTEMP[8]__attribute__((aligned(16)));
unsigned short t0, t1, t2, t3, t4, t5, t6, t7;

vec_st(VSVI n. vec, 0, (unsigned short*)ptrTEMP);
t0 = ptrTEMP[0]; t0 = Table[t0];
t1 = ptrTEMP[1]; t1 = Table[t1];
t2 = ptrTEMP[2]; t2 = Table[t2];
t3 = ptrTEMP[3]; t3 = Table[t3];
t4 = ptrTEMP[4]; t4 = Table[t4];
t5 = ptrTEMP[5]; t5 = Table[t5];
t6 = ptrTEMP[6]; t6 = Table[t6];
t7 = ptrTEMP[7]; t7 = Table[t7];
ptrTEMP[0]=t0;
ptrTEMP[1]=t1;
ptrTEMP[2]=t2;
ptrTEMP[3]=t3;
ptrTEMP[4]=t4;
ptrTEMP[5]=t5;
ptrTEMP[6]=t6;
ptrTEMP[7]=t7;
USV0ut. vec = vec_ld(0, (unsigned short*)ptrTEMP);
```

26個のロードストア命令
で実現される

エントリ数の多いテーブル参照処理の最適化②

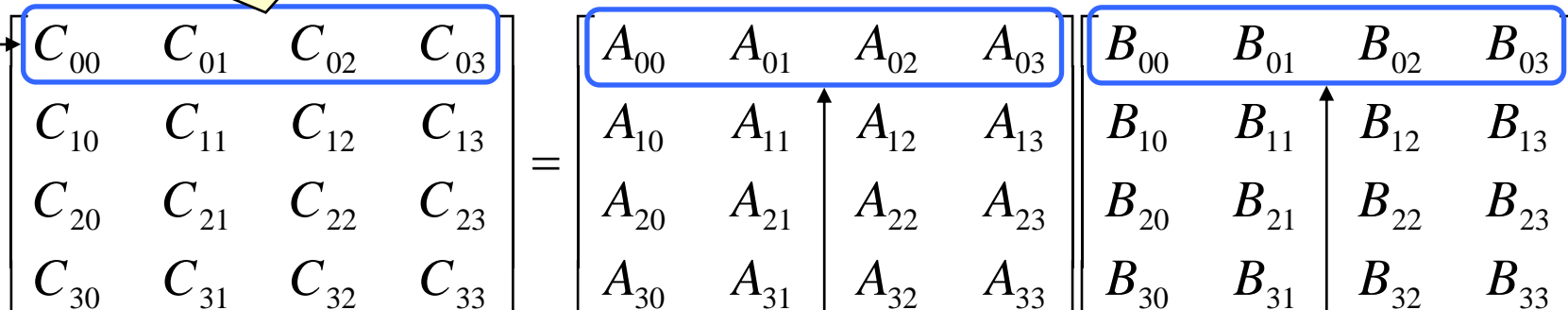
- 前頁のテーブル参照処理で使用するリソースは、ロードストアユニットと整数演算ユニットのレジスタセットとなる
- 以下の3段パイプライン処理を導入して、テーブル参照処理を一つのステージに割り付ければ、使用するリソースがステージ間で重複しないので効果的



浮動小数点形式で処理する場合の最適化①

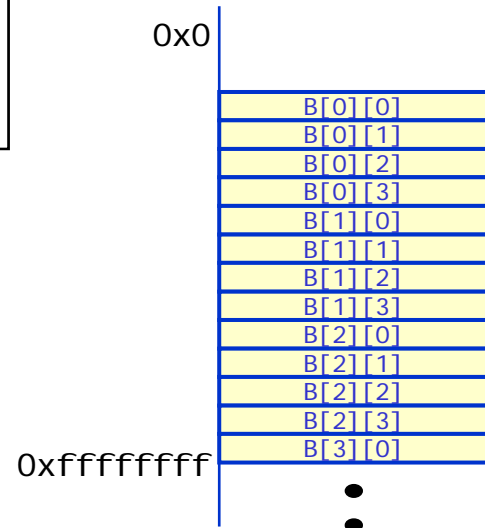
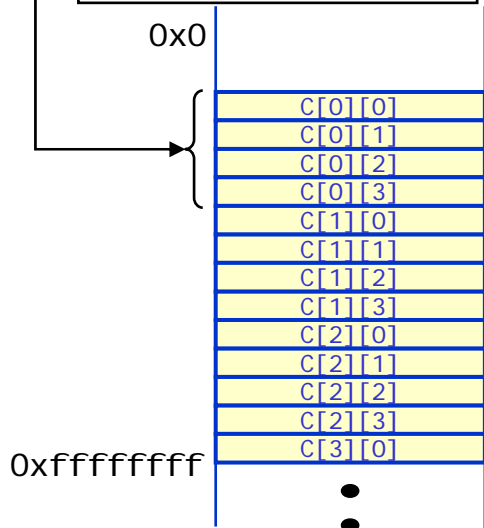
コードレベル最適化①

アドレスが連続している



```
Main()
{
  float A[4][4];
  float B[4][4];
  float C[4][4];
}
```

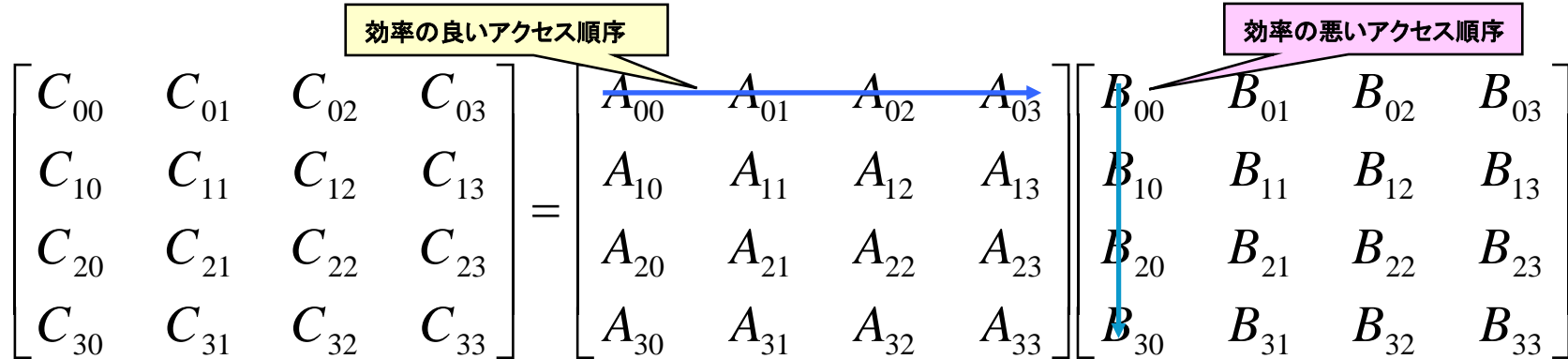
データの入出力については、
行列の行方向へ順番に行われ
るように意識すると高速化
が図れる



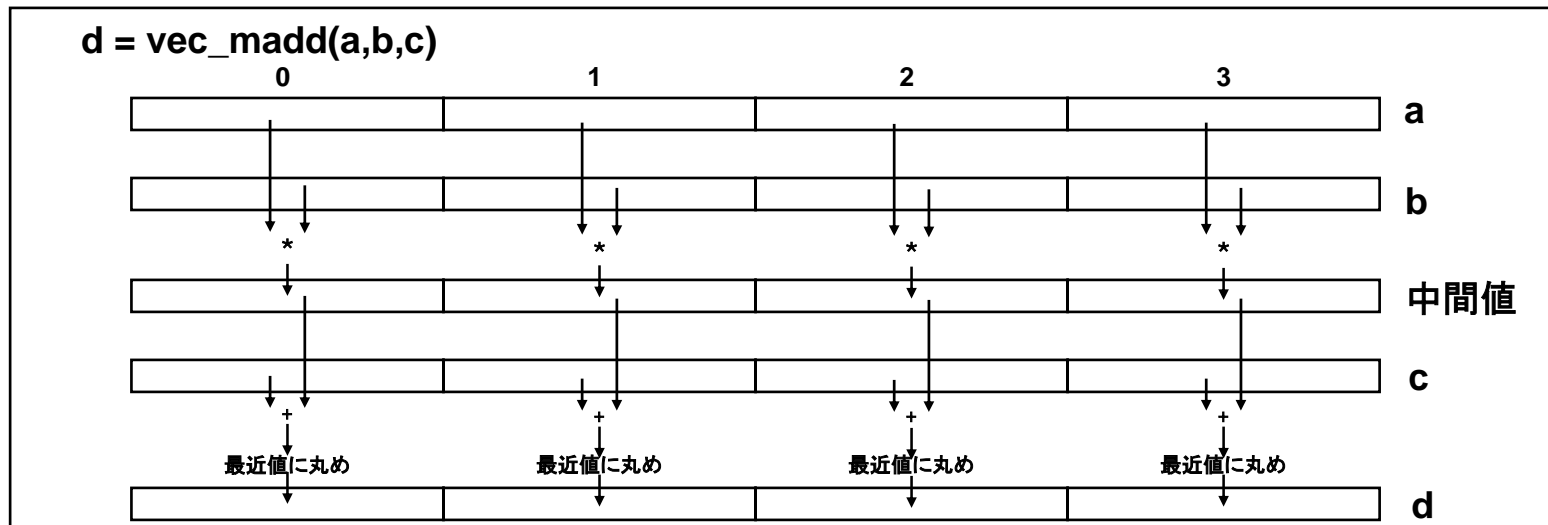
浮動小数点形式で処理する場合の最適化②

- ここでC00のデータを計算する場合、行列Bの要素へのアクセスにベクタ・ロード命令が使えないため効率が悪い

$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20} + A_{03}B_{30}$$

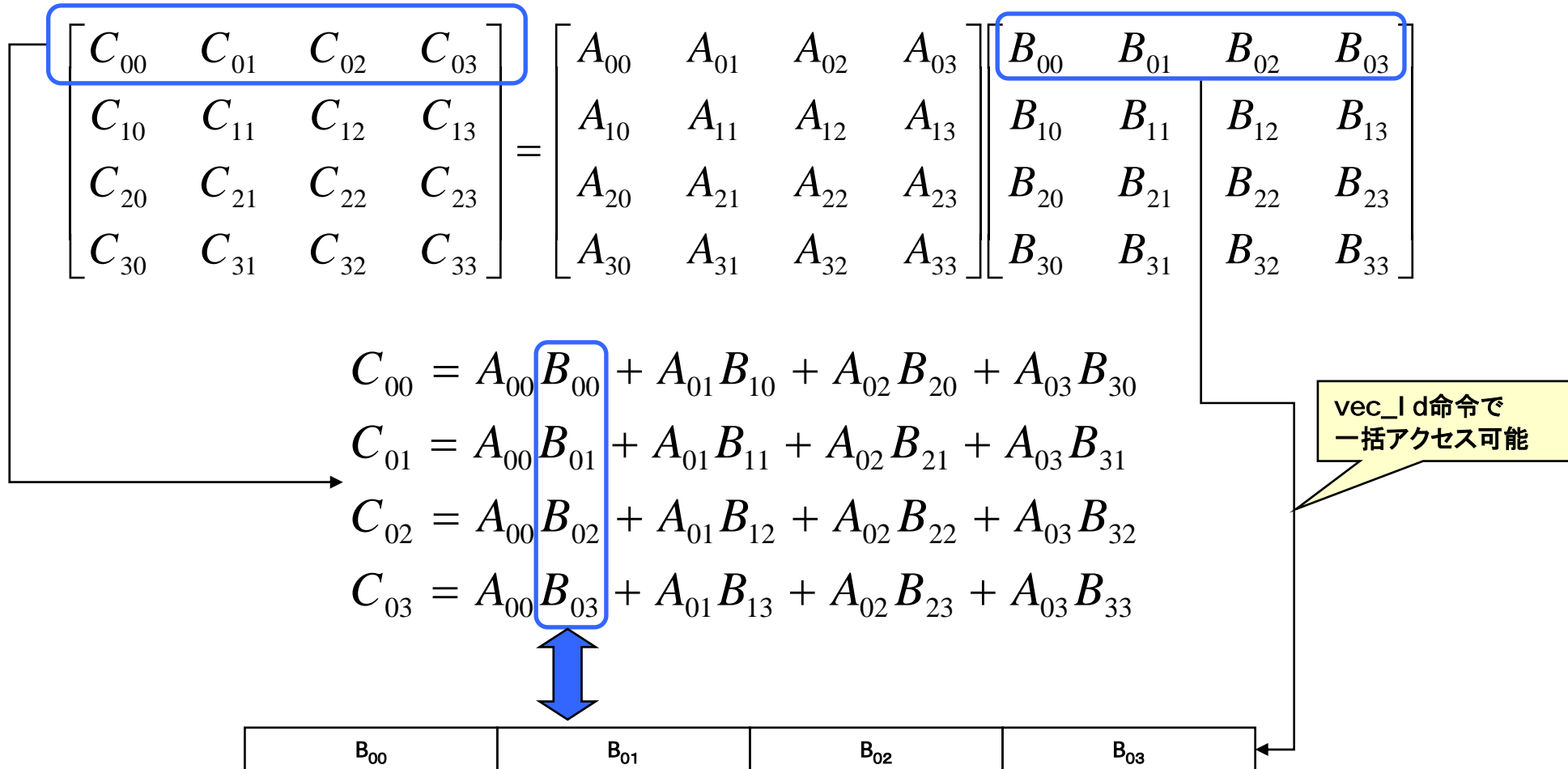


Altivecを用いてFloat型の積和演算を実現する場合、以下のvec_madd命令を使用



浮動小数点形式で処理する場合の最適化③

- そこで、AltiVec命令の特長を生かしてコーディングするために、出力となる行列Cの一行分に注目する

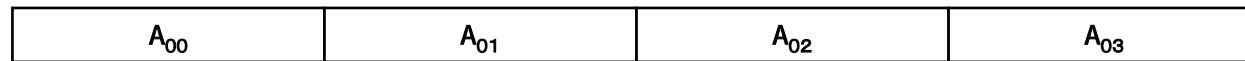


浮動小数点形式で処理する場合の最適化④

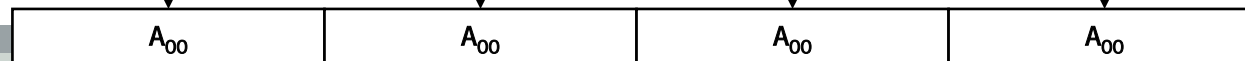
- 行列Aの行要素をメモリからレジスタへベクタ・ロード命令(`vec_ld`)を用いて転送
- `vec_splat`命令を用いて、この後の演算に適したデータ並びを実現

$$\begin{bmatrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} B_{00} & B_{01} & B_{02} & B_{03} \\ B_{10} & B_{11} & B_{12} & B_{13} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \end{bmatrix}$$

`vec_ld`命令で
一括アクセス可能



`vec_splat`命令
で展開



浮動小数点形式で処理する場合の最適化⑤

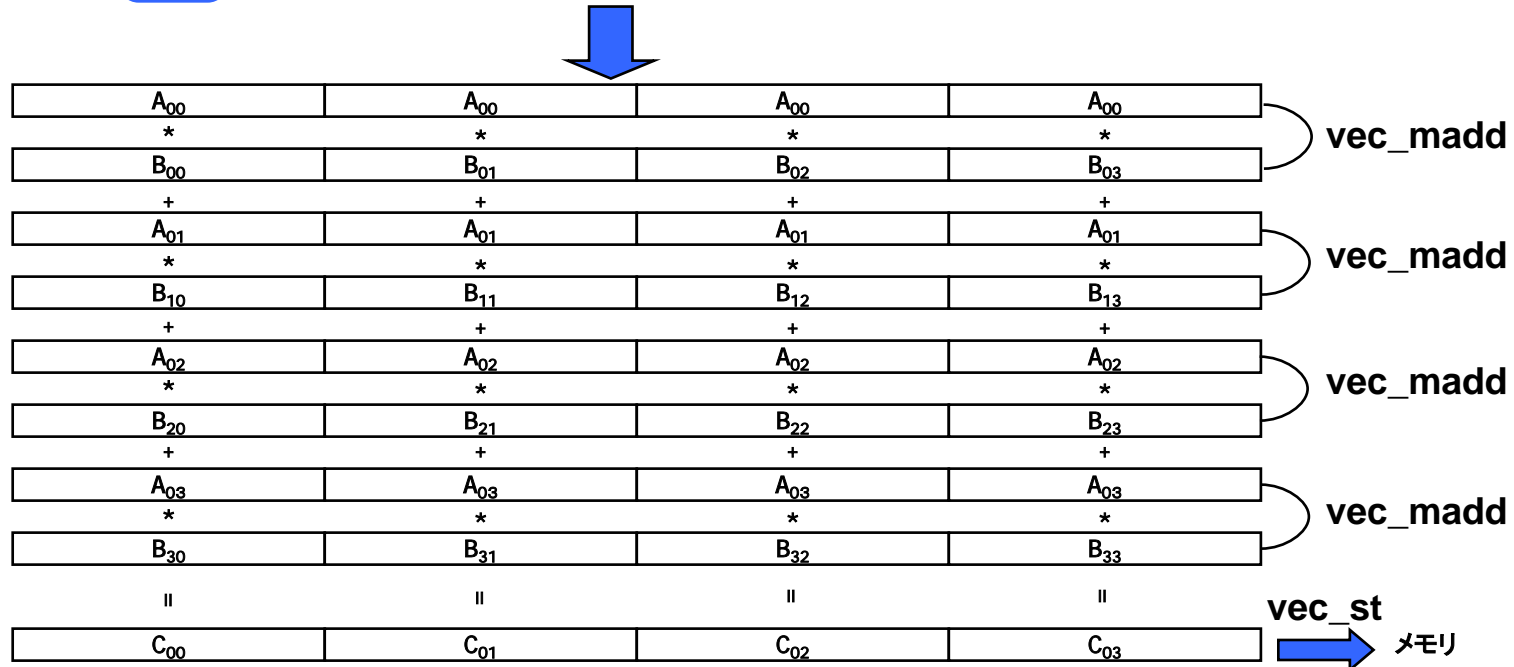
- 所望のデータ並びが完成した後は、以下に示すように4回分の積和演算命令で4個の結果が得られる

$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20} + A_{03}B_{30}$$

$$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21} + A_{03}B_{31}$$

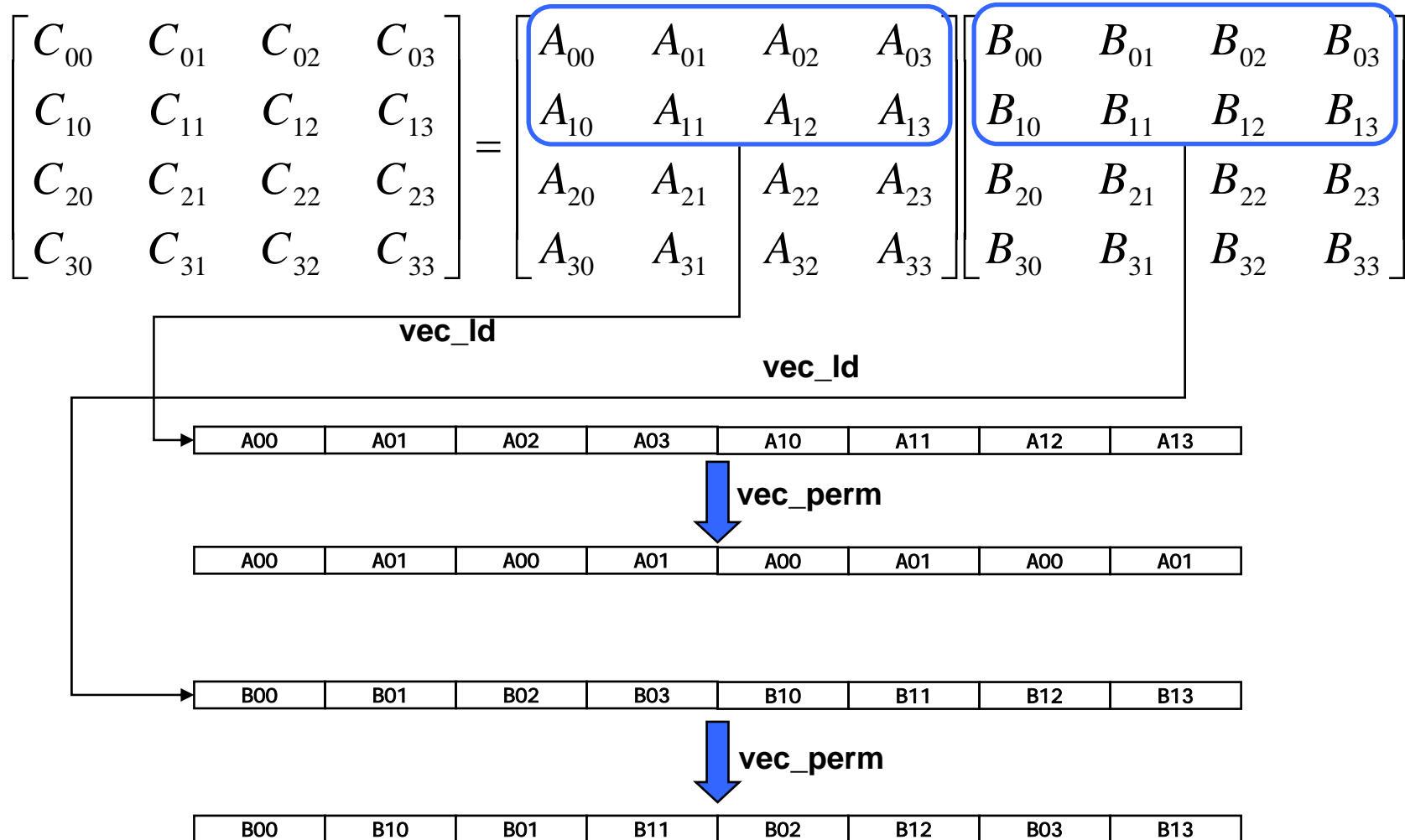
$$C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22} + A_{03}B_{32}$$

$$C_{03} = A_{00}B_{03} + A_{01}B_{13} + A_{02}B_{23} + A_{03}B_{33}$$



入力:16ビット、出力:32ビットで処理する場合の最適化①

- 16ビット整数型、もしくは固定小数点型で対応できる場合にはさらに高速化可能
- `vec_perm`命令を用いて、この後の演算に適したデータ並びを実現

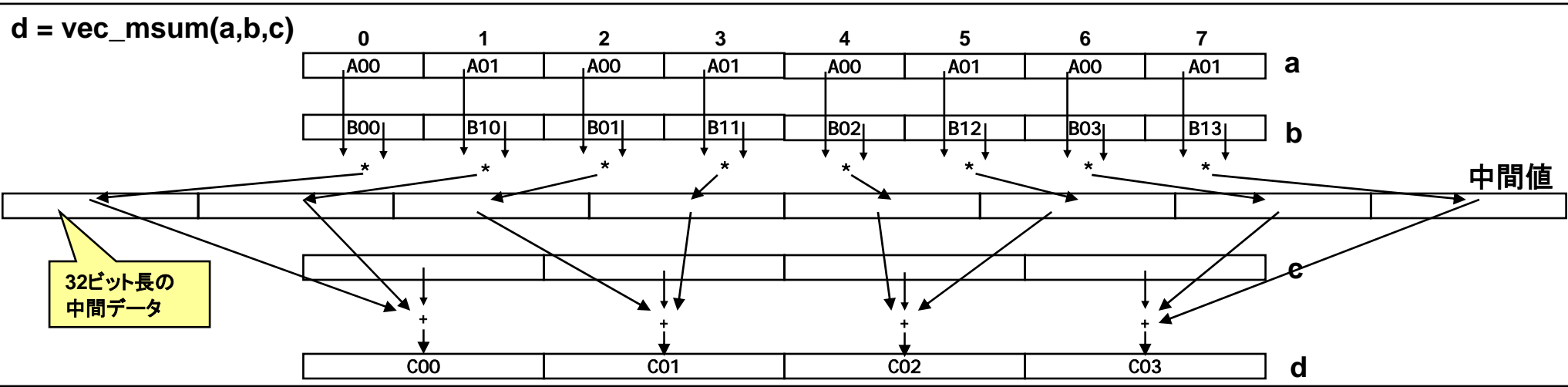


入力:16ビット、出力:32ビットで処理する場合の最適化②

- 所望のデータ並びが完成した後は、2回分の積和演算命令で4個の結果が得られる

$$\begin{bmatrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} B_{00} & B_{01} & B_{02} & B_{03} \\ B_{10} & B_{11} & B_{12} & B_{13} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \end{bmatrix}$$

$$\begin{aligned}
 C_{00} &= A_{00} B_{00} + A_{01} B_{10} + A_{02} B_{20} + A_{03} B_{30} \\
 C_{01} &= A_{00} B_{01} + A_{01} B_{11} + A_{02} B_{21} + A_{03} B_{31} \\
 C_{02} &= A_{00} B_{02} + A_{01} B_{12} + A_{02} B_{22} + A_{03} B_{32} \\
 C_{03} &= A_{00} B_{03} + A_{01} B_{13} + A_{02} B_{23} + A_{03} B_{33}
 \end{aligned}$$



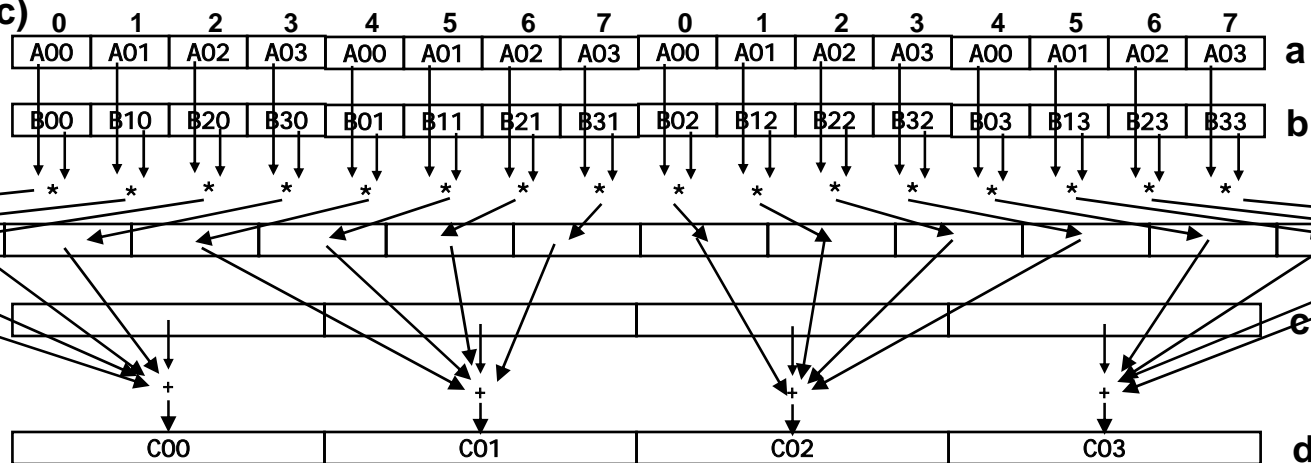
入力:8ビット、出力:32ビットで処理する場合の最適化

- 入力:8ビット精度が許容できれば、1回分の積和演算命令で4個の結果が得られる

$$\begin{bmatrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} B_{00} & B_{01} & B_{02} & B_{03} \\ B_{10} & B_{11} & B_{12} & B_{13} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \end{bmatrix}$$

$$\begin{aligned}
 C_{00} &= A_{00} B_{00} + A_{01} B_{10} + A_{02} B_{20} + A_{03} B_{30} \\
 C_{01} &= A_{00} B_{01} + A_{01} B_{11} + A_{02} B_{21} + A_{03} B_{31} \\
 C_{02} &= A_{00} B_{02} + A_{01} B_{12} + A_{02} B_{22} + A_{03} B_{32} \\
 C_{03} &= A_{00} B_{03} + A_{01} B_{13} + A_{02} B_{23} + A_{03} B_{33}
 \end{aligned}$$

d = vec_msum(a,b,c)



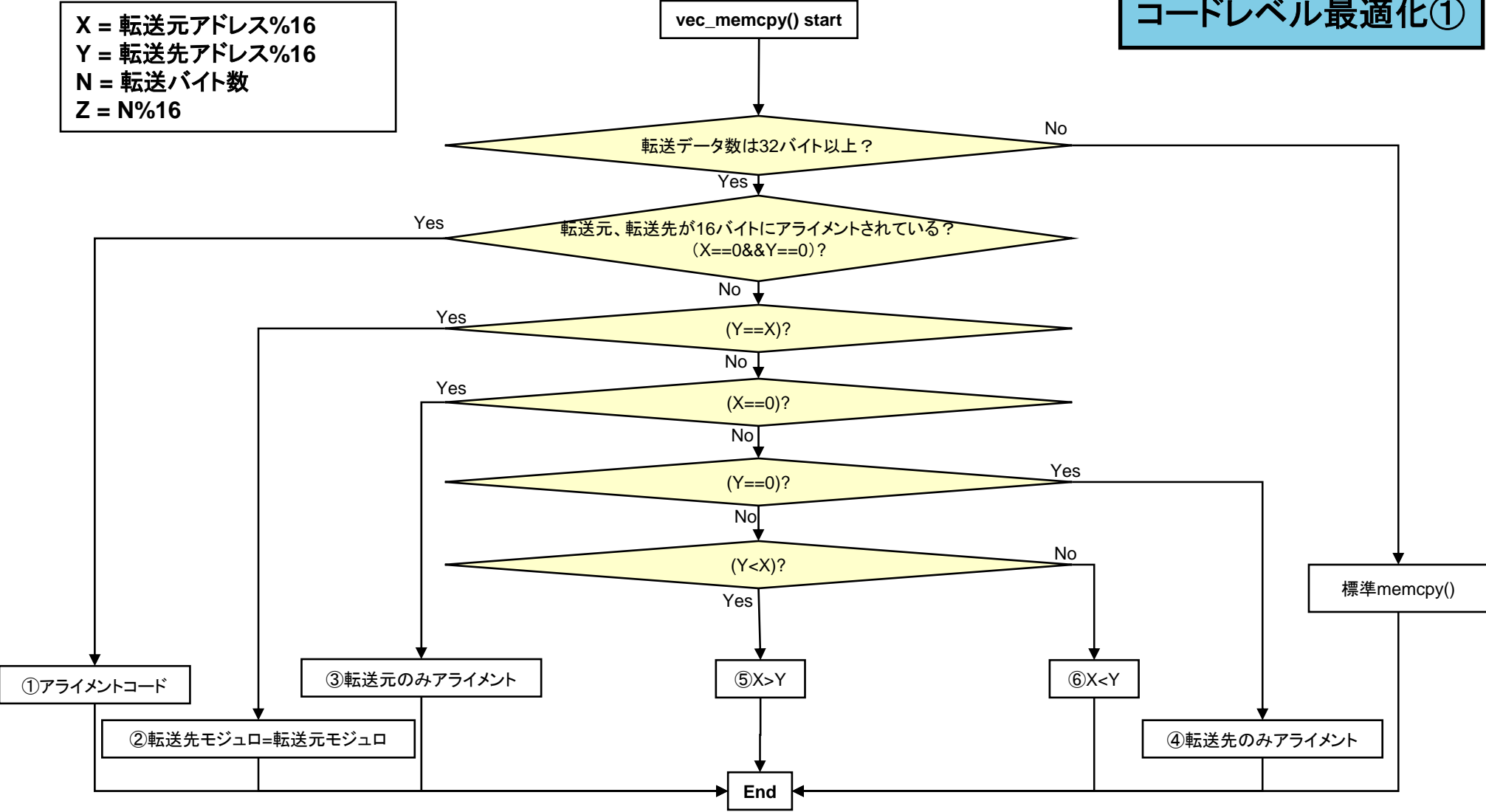
中間値

16ビット長の
中間データ

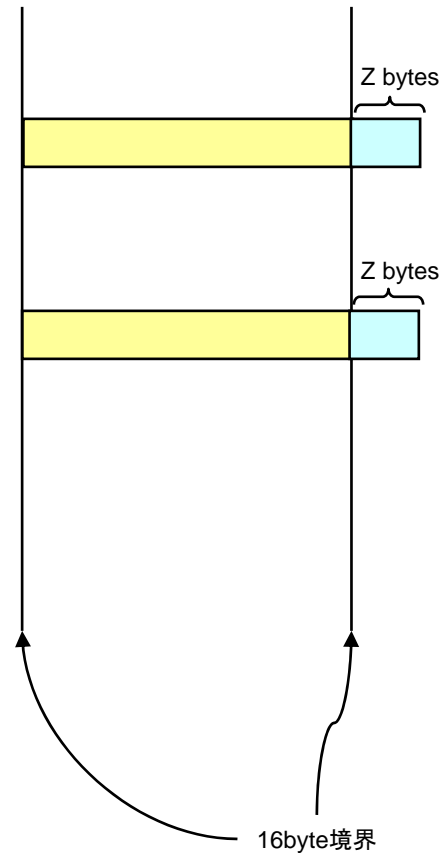
アライメントのずれを吸収するmemcpy関数の実装(場合分け)

コードレベル最適化①

X = 転送元アドレス%16
Y = 転送先アドレス%16
N = 転送バイト数
Z = N%16



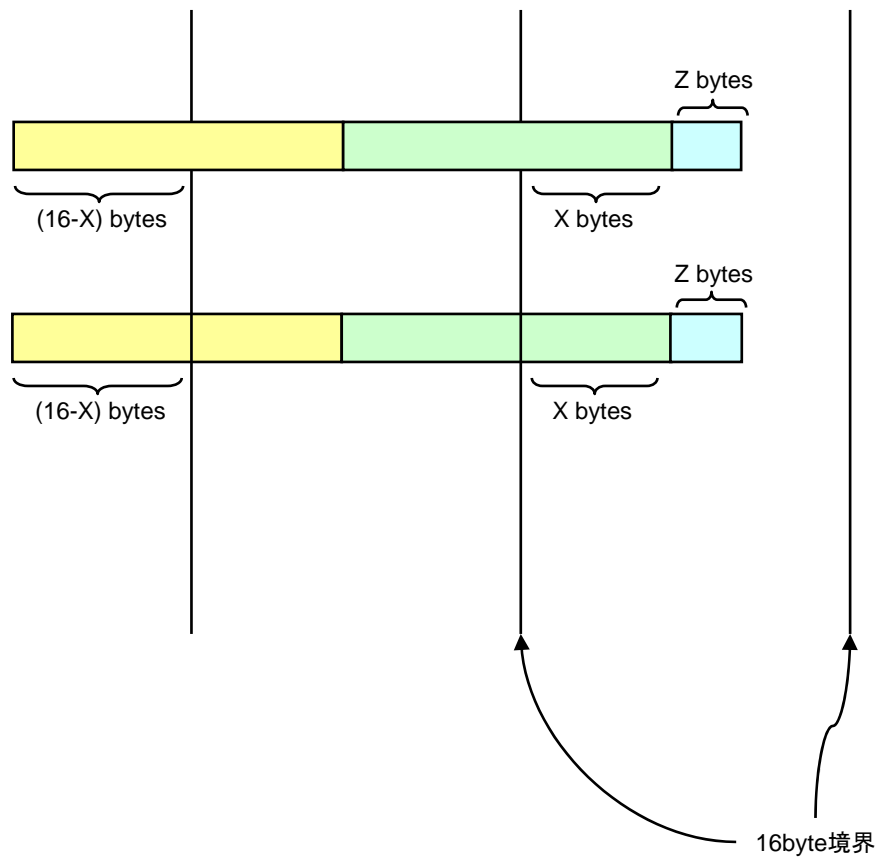
転送元と転送先のアドレスがアライメントされている場合 (X=Y=0)



```
vector unsigned char v0, v1, d0;  
vector unsigned char vPerm;  
vector unsigned char vTemp;  
signed long i;  
unsigned long X = (unsigned int)src & 0x0f;  
unsigned long Y = (unsigned int)dst & 0x0f;  
unsigned long Z = count & 0x0f;  
unsigned long SRC = (unsigned long)src;  
unsigned char ucTemp;
```

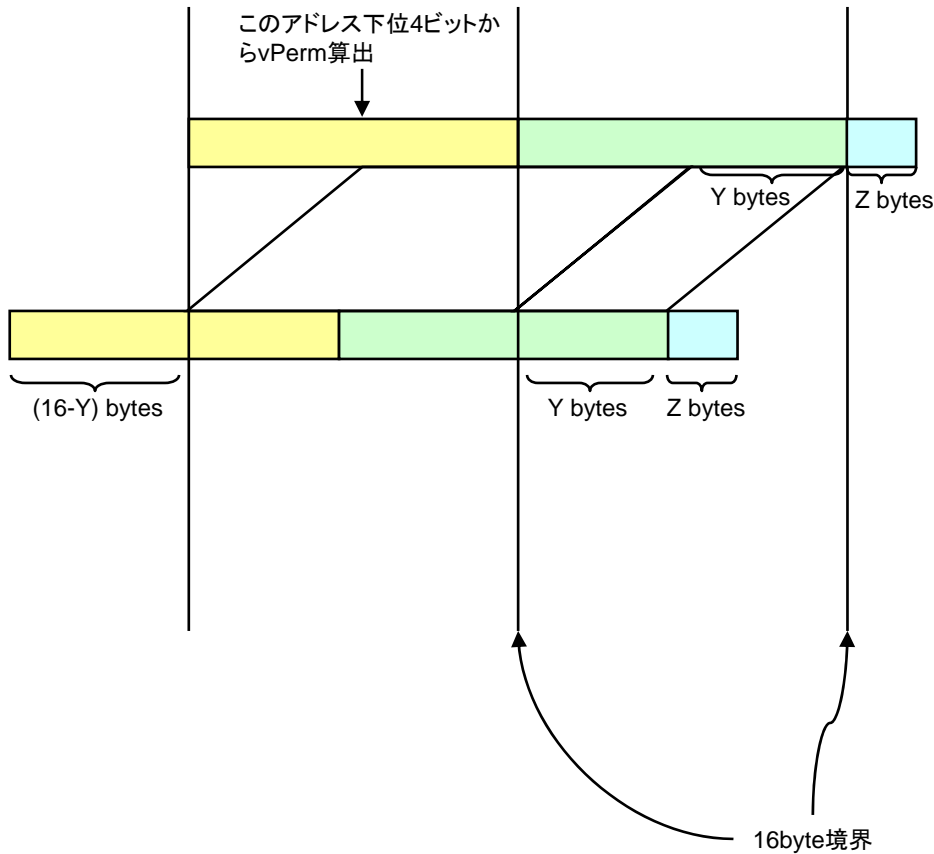
```
if( (X == 0) && ( Y == 0 ) )  
{  
    for(i=0; i<(count>>4); i++)  
    {  
        v0 = vec_ld(0, src);  
        src += 16;  
        vec_st(v0, 0, dst);  
        dst += 16;  
    }  
    //端数処理  
    for(i=0; i<Z; i++)  
    {  
        ucTemp = *src++;  
        *dst++ = ucTemp;  
    }  
    return;  
}
```

(転送元アドレス%16)と(転送先アドレス%16)が同値の場合(X=Y)



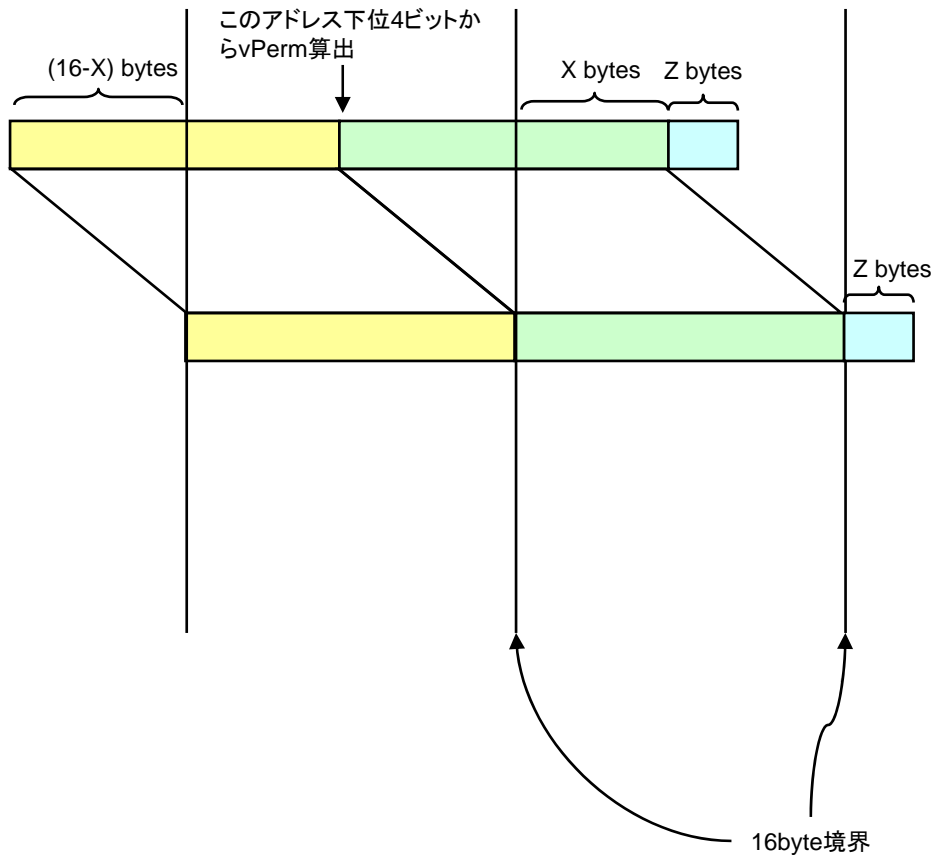
```
if( X == Y )
{
    for(i=0; i<(16-X); i++)
    {
        ucTemp = *src++;
        *dst++ = ucTemp;
    }
    for(i=0; i<(((count)>>4)-1); i++)
    {
        v0 = vec_ld(0, src);
        src += 16;
        vec_st(v0, 0, dst);
        dst += 16;
    }
    //端数処理
    for(i=0; i<(Z+X); i++)
    {
        ucTemp = *src++;
        *dst++ = ucTemp;
    }
    return;
}
```

転送元アドレスのみがアライメントされている場合 (X=0)



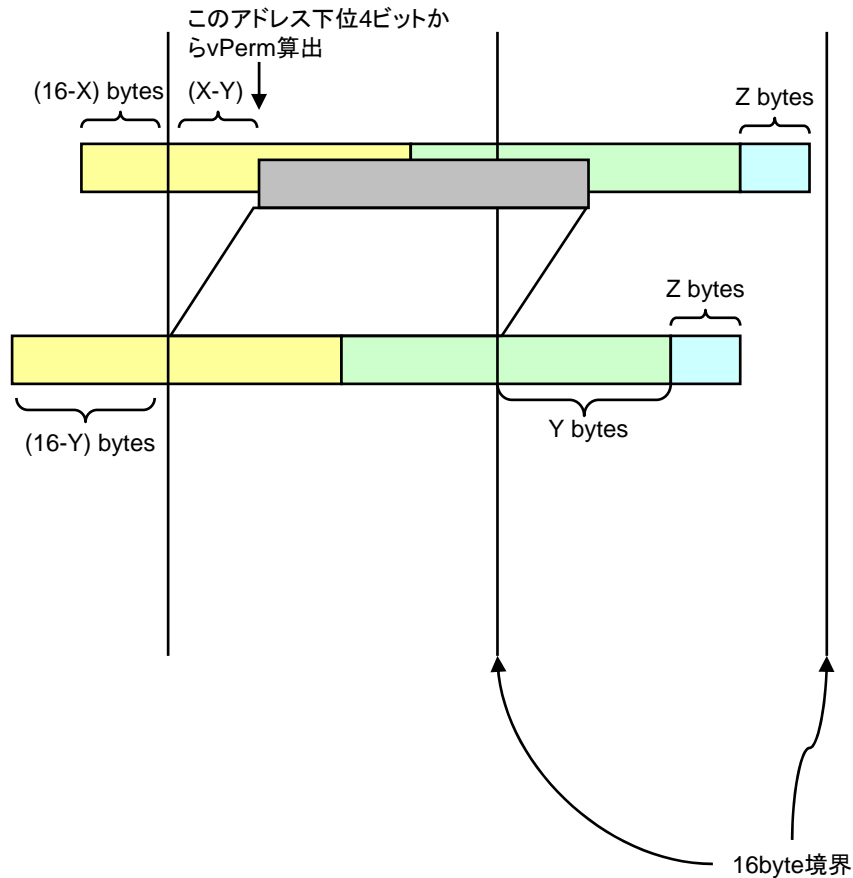
```
if( X == 0 )
{
    vPerm = vec_lvsr(0, dst);
    v0 = vec_ld(0, src);
    vTemp = vec_perm(v0, v0, vPerm);
    for(i=0; i<(16-Y); i++)
    {
        vec_ste(vTemp, 0, dst);
        dst += 1;
    }
    src += 16;
    for(i=0; i<((count>>4)-1); i++)
    {
        v1 = vec_ld(0, src);
        src += 16;
        d0=vec_perm(v0, v1, vPerm);
        vec_st(d0, 0, dst);
        v0=v1;
        dst += 16;
    }
    vTemp=vec_perm(v0, v1, vPerm);
    for(i=0; i<Y; i++)
    {
        vec_ste(vTemp, 0, dst);
        dst += 1;
    }
    for(i=0; i< Z; i++)
    {
        ucTemp = *src++;
        *dst++ = ucTemp;
    }
    return;
}
```

転送先アドレスのみがアライメントされている場合 (Y=0)



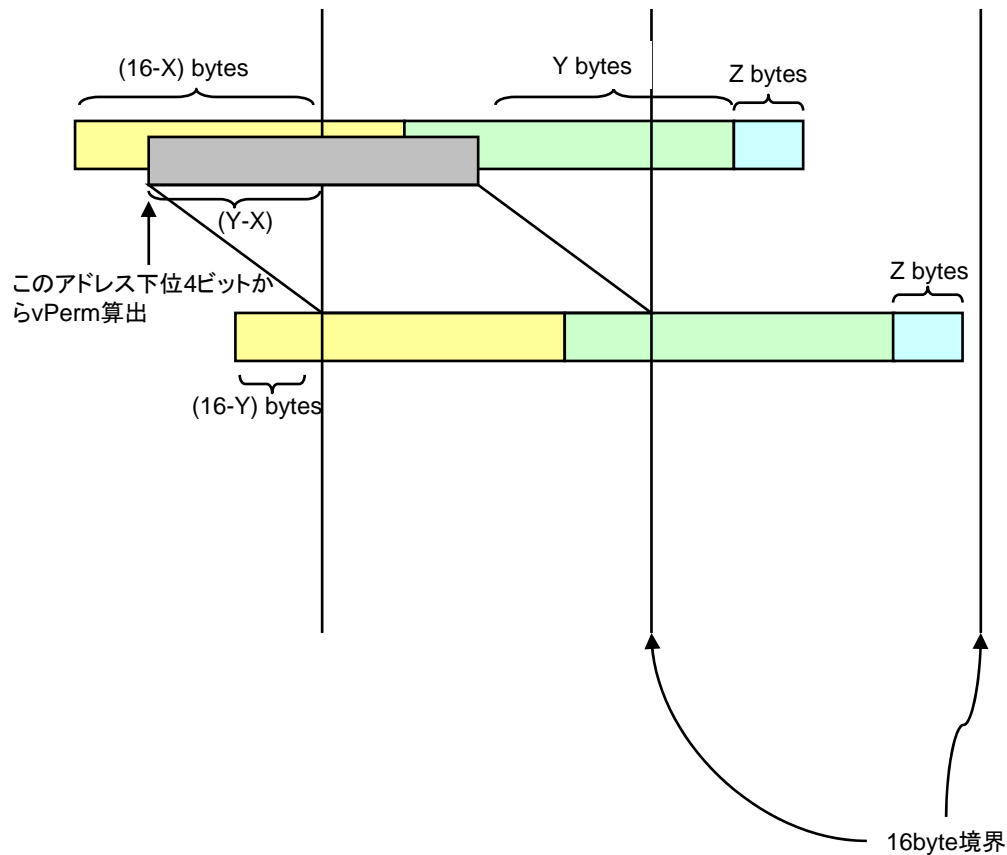
```
if( Y == 0 )
{
    vPerm = vec_lvs1( 0, src);
    v0 = vec_ld(0, src);
    src += 16;
    for(i=0; i<((count>>4)-1); i++)
    {
        v1 = vec_ld(0, src);
        src += 16;
        d0=vec_perm(v0, v1, vPerm);
        vec_st(d0, 0, dst);
        v0=v1;
        dst += 16;
    }
    v1 = vec_ld(0, src);
    d0=vec_perm(v0, v1, vPerm);
    vec_st(d0, 0, dst);
    dst += 16;
    //端数処理
    for(i=0; i < Z; i++)
    {
        ucTemp = *src++;
        *dst++ = ucTemp;
    }
    return;
}
```

(転送元アドレス%16)が(転送先アドレス%16)より大きい場合(X>Y)



```
if( X > Y )
{
    vPerm = vec_lvsr(0, (unsigned char *) (X-Y));
    v1 = vec_ld(0, src);
    src += 16;
    v0 = vec_ld(0, src);
    src += 16;
    vTemp=vec_perm(v1, v0, vPerm);
    for(i=0; i<(16-Y); i++)
    {
        vec_ste(vTemp, 0, dst);
        dst += 1;
    }
    for(i=0; i<((count>>4)-1); i++)
    {
        v1 = vec_ld(0, src);
        src += 16;
        d0=vec_perm(v0, v1, vPerm);
        vec_st(d0, 0, dst);
        v0=v1;
        dst += 16;
    }
    src -= 16;
    vTemp=vec_perm(v0, v1, vPerm);
    for(i=0; i<Y; i++)
    {
        vec_ste(vTemp, 0, dst);
        dst += 1;
    }
    //端数処理
    for(i=0; i<Z; i++)
    {
        ucTemp = *src++;
        *dst++ = ucTemp;
    }
    return;
}
```

(転送元アドレス%16)が(転送先アドレス%16)より小さい場合 ($X < Y$)



```
// if( X < Y )
vPerm = vec_lvsl( 0, (unsigned char *) (Y-X));
v0 = vec_ld(0, src);
src += 16;
vTemp=vec_perm(v0, v0, vPerm);
for(i=0; i<(16-Y); i++)
{
    vec_ste(vTemp, 0, dst);
    dst += 1;
}
for(i=0; i<((count>>4)-1); i++)
{
    v1 = vec_ld(0, src);
    src += 16;
    d0=vec_perm(v0, v1, vPerm);
    vec_st(d0, 0, dst);
    v0=v1;
    dst += 16;
}
v1 = vec_ld(0, src);
d0=vec_perm(v0, v1, vPerm);
for(i=0; i<Y; i++)
{
    vec_ste(d0, 0, dst);
    dst += 1;
}
//端数処理
for(i=0; i<Z; i++)
{
    ucTemp = *src++;
    *dst++ = ucTemp;
}
return;
```

アライメントのずれを吸収する処理を1パターンで対応

- アライメントのずれに関わらず1パターンで処理させた場合、場合分けの手法に比べて処理時間はかかるがコードサイズが小さくなる

15番地先を指定することで、アライメントされている時に限り同じ領域をアクセスさせることができるため、ループの最後に余計な16バイトにアクセスさせずに済む

```
vPerm0 = vec_lvsl( 0 , src ); //For load
vPerm1 = vec_lvsl( 0 , dst ); //For store
vPerm2 = vec_lvsl( 0 , dst ); //For store
for (i=0; i<(count)>>4; i++)
{
    //For load
    v0 = vec_ld( 0 , src );
    v1 = vec_ld( 15 , src );
    v2 = vec_perm( v0 , v1 , vPerm0 );
    src += 16;

    //For store
    v0 = vec_ld( 0 , dst );
    v1 = vec_ld( 15 , dst );
    v0 = vec_perm( v0 , v0 , vPerm1 );
    v1 = vec_perm( v1 , v1 , vPerm1 );
    v0 = vec_perm( v0 , v2 , vPerm2 );
    v1 = vec_perm( v2 , v1 , vPerm2 );
    vec_st( v1 , 15 , dst );
    vec_st( v0 , 0 , dst );
    dst += 16;
}
return;
```

アライメントされていないストアに対応させる場合、余計な部分を上書きしないようにするためロード版より複雑になる

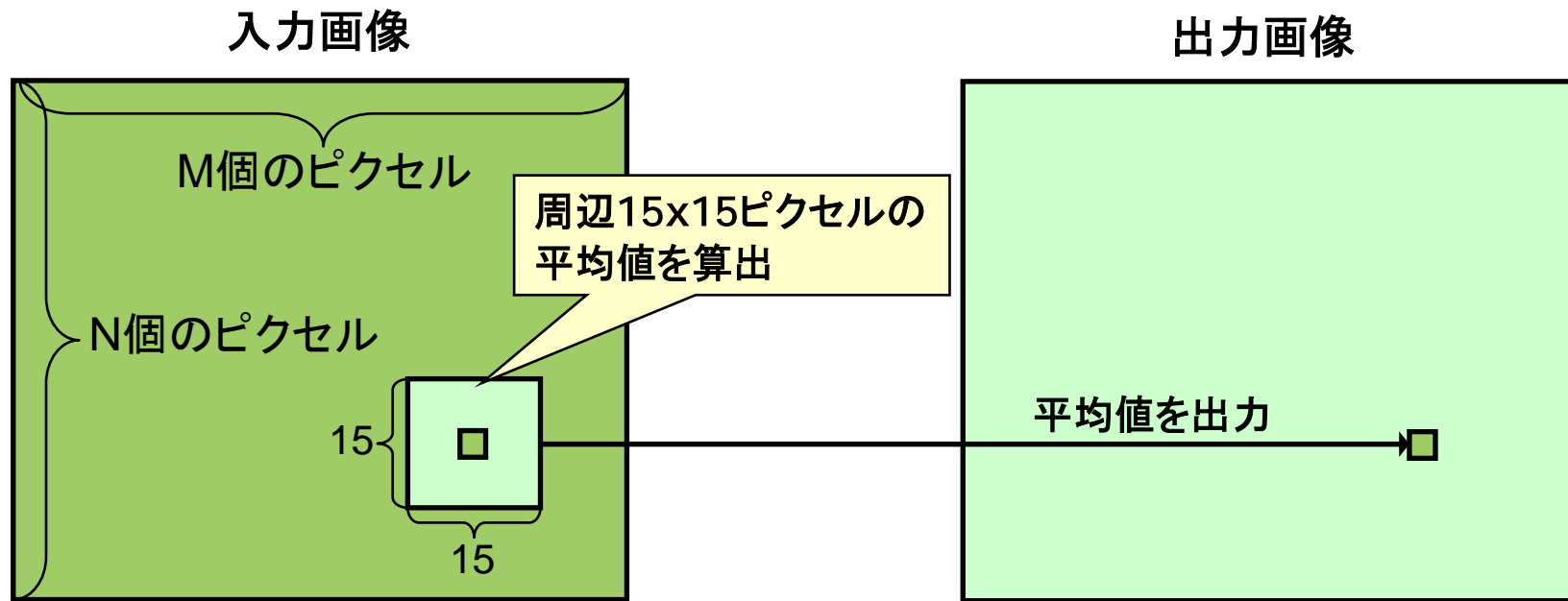
アライメントされている時には無効なデータが書き込まれることになるが続くvec_stで上書き可能

- ループ内で複数のデータをまとめて処理する場合、端数処理が必要な場合が出てくる
- AltiVecを用いて数十倍の高速化を実現できた時には、スカラでの端数処理によるオーバヘッドが目立つことがある
- 一方で端数処理用だけに別途AltiVecコードを用意するのは作業効率が悪い
- データストア先が中間バッファの場合、このバッファを冗長にしておき、端数処理を用意せずに済みます
- システム全体で見て、最終的にデータを出力する時のみ端数処理を意識する

端数処理をしなくて済むように冗長なバッファを用意する

- 任意の数 X (16ビット)について近傍の32の倍数を算出する時は次の手法を用いる
 - $Y \leq X$ 、かつ、 $Y=32$ の倍数、を満たす最大の Y
 $Y = X \& 0\text{ffc}0$
または、 $Y = X/32$
 - $Y \geq X$ かつ、 $Y=32$ の倍数、を満たす最小の Y
 $Y = (X+31) \& 0\text{ffc}0$
または、 $Y = ((X+31)/32) * 32$
- ループの繰り返し回数を定義する時やアライメントされているバッファをmallocなどにより確保する時に有効

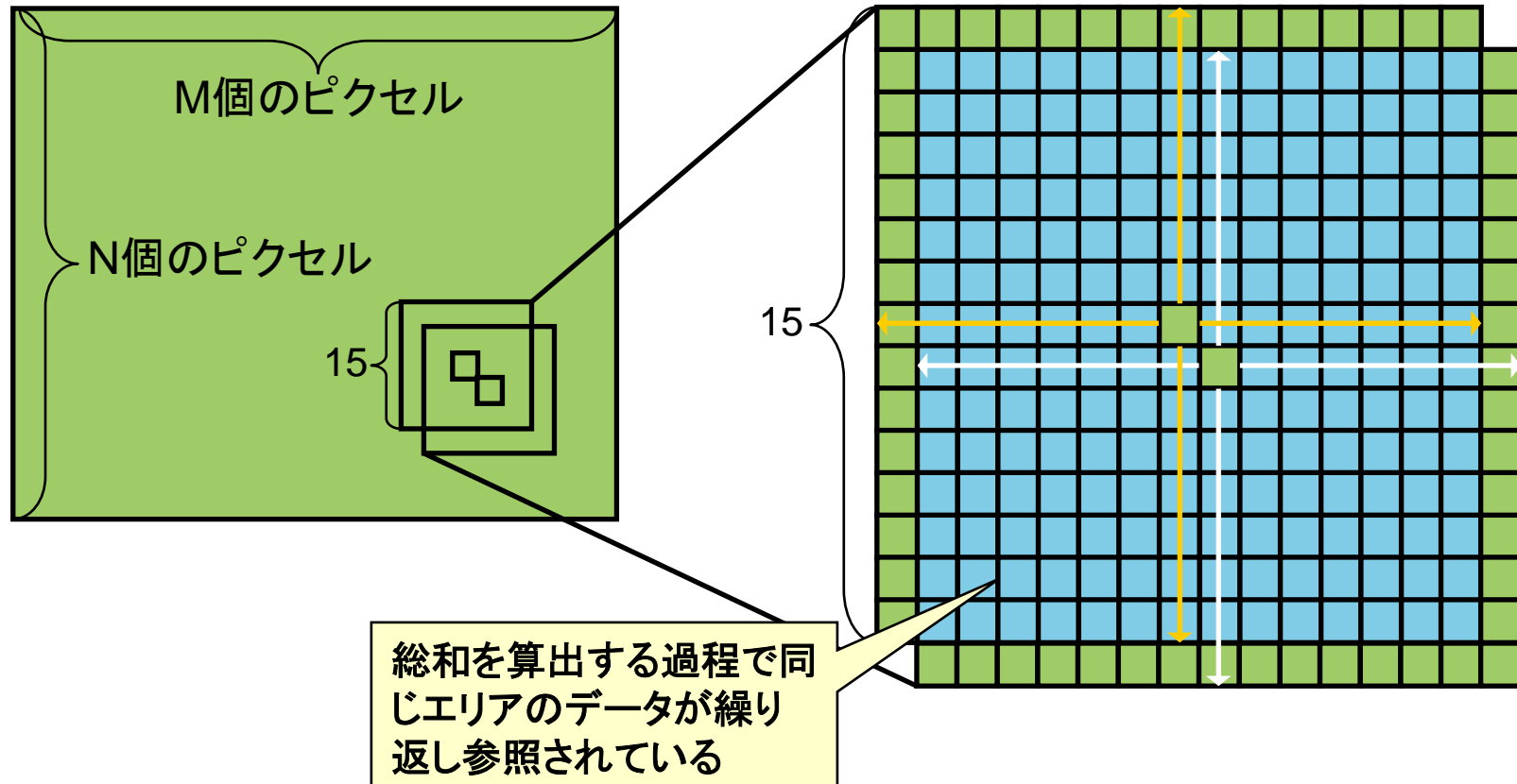
2次元画像の加重平均フィルタを用いて最適化を実践



ここで、各ピクセルは入出力共に16bit/pixelとする

アルゴリズムレベルの最適化手法から考察

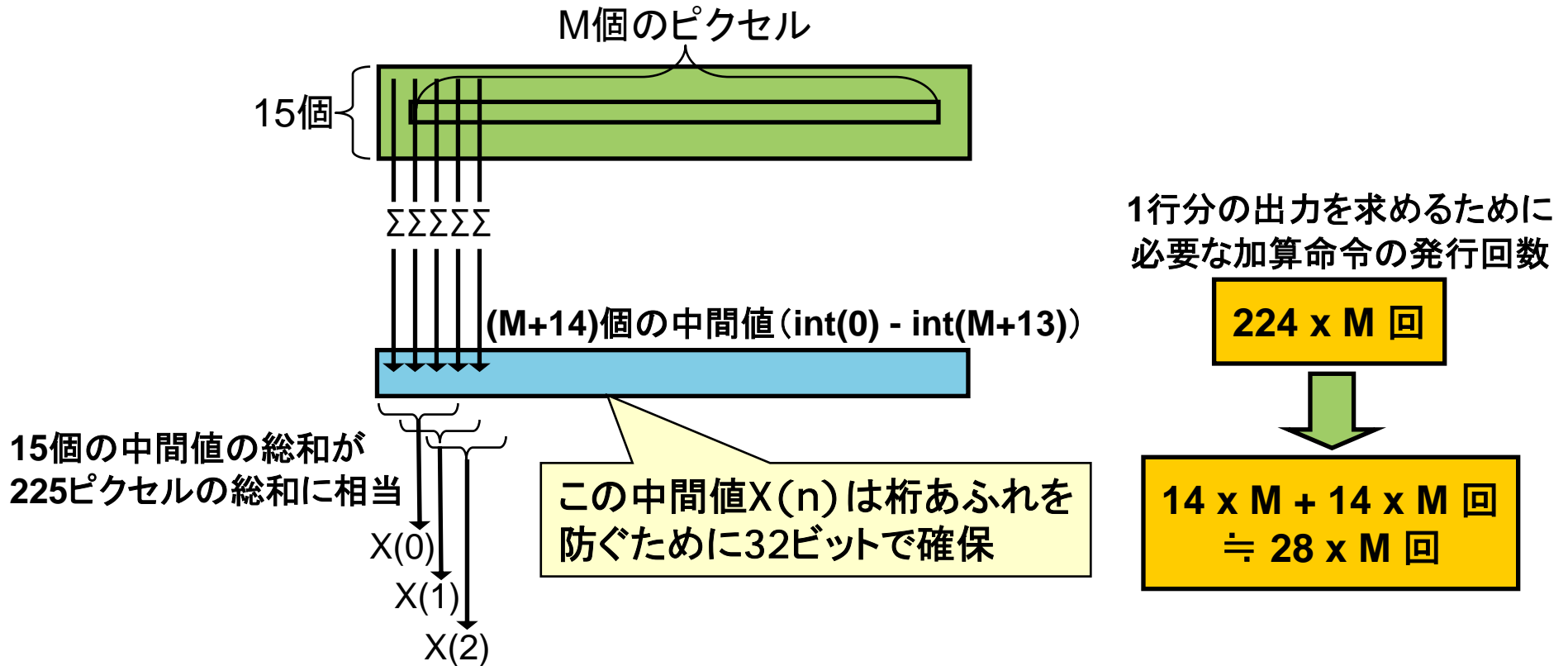
- Altivec命令を用いてコードレベル最適化を行う前にアルゴリズムレベルの最適化を行う
- 冗長な繰り返し処理を見極め、計算過程のデータを中間値として保存しておく



PowerPC G4に適した中間値の保存方法を選択

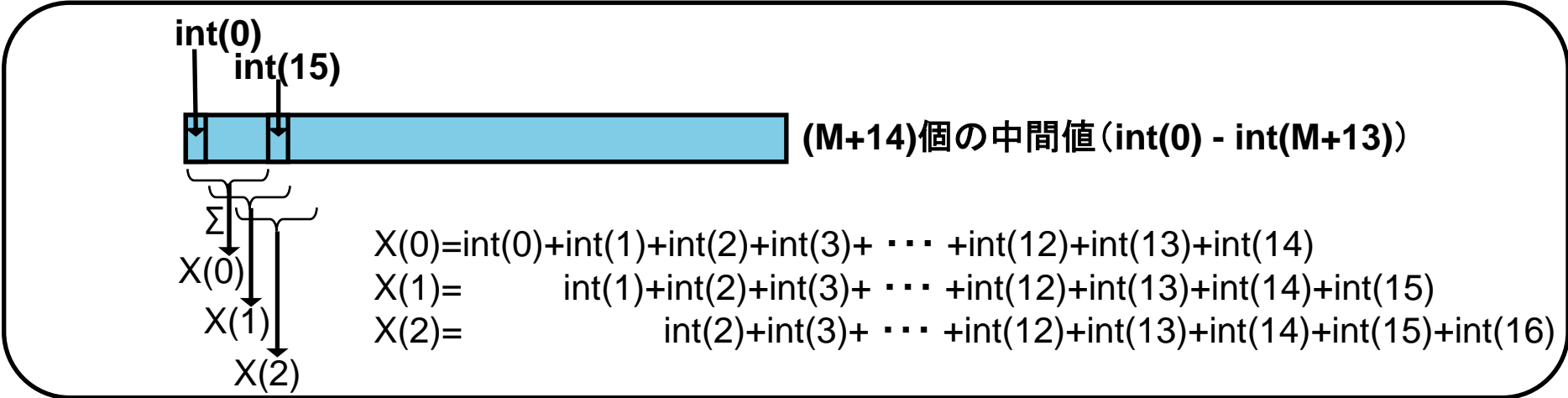
アルゴリズム最適化

- 列方向に加算した中間結果を保存



中間値から総和を求める処理をさらに改善

アルゴリズム最適化



14回の加算で行う処理
を2回の加減算で実現

$$X(0) = \text{int}(0) + \text{int}(1) + \text{int}(2) + \dots + \text{int}(14)$$
$$X(1) = X(0) + \text{int}(15) - \text{int}(0)$$
$$X(2) = X(1) + \text{int}(16) - \text{int}(1)$$

-

$$X(n) = X(n-1) + \text{int}(n+14) - \text{int}(n-1)$$

1行分の出力を求めるために
必要な加減算命令の発行回数

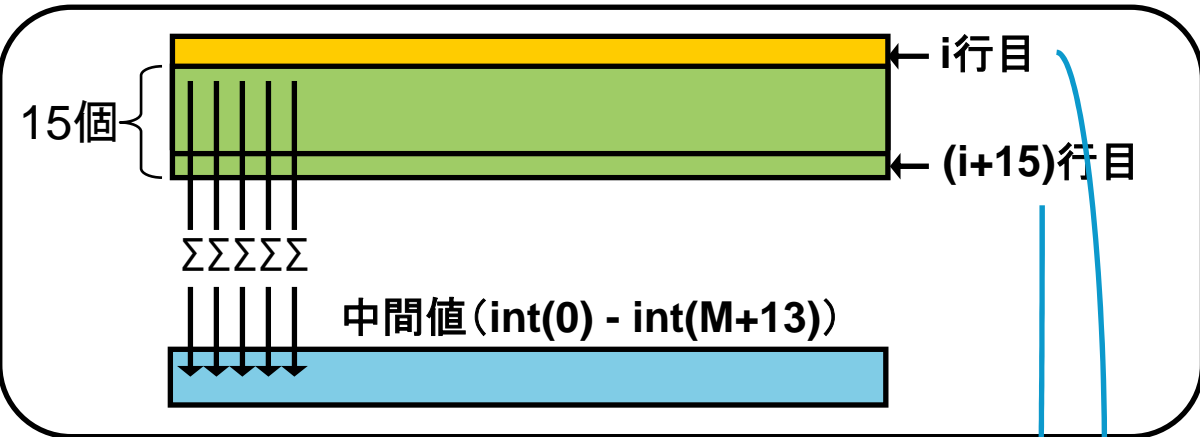
28 x M 回



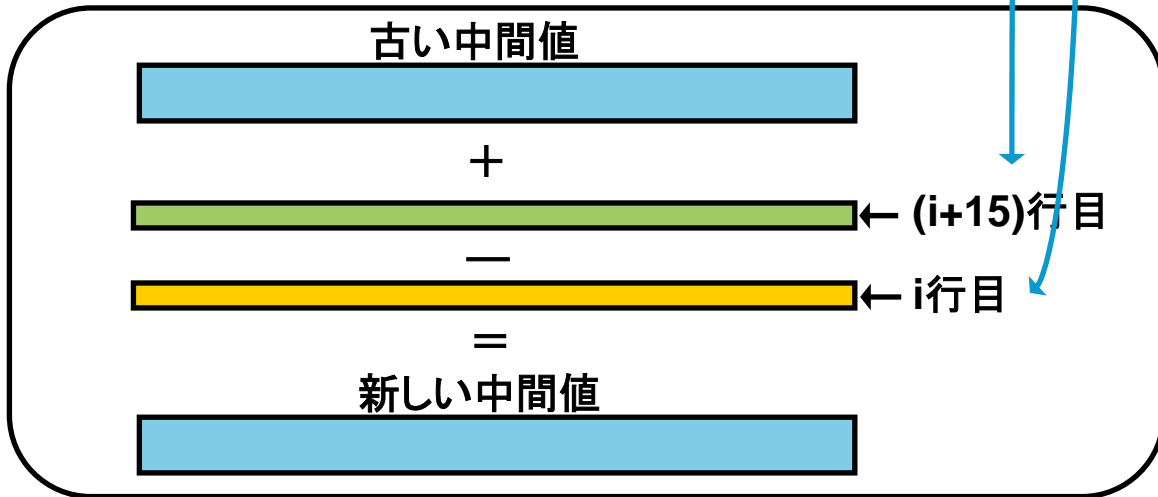
14 x M + 2 x (M-1) + 14 回
≒ 16 x M 回

中間値を算出する処理をさらに改善

アルゴリズム最適化



14回の加算で行っていた処理を2回の加減算で実現



1行分の出力を求めるために必要な加減算命令の発行回数

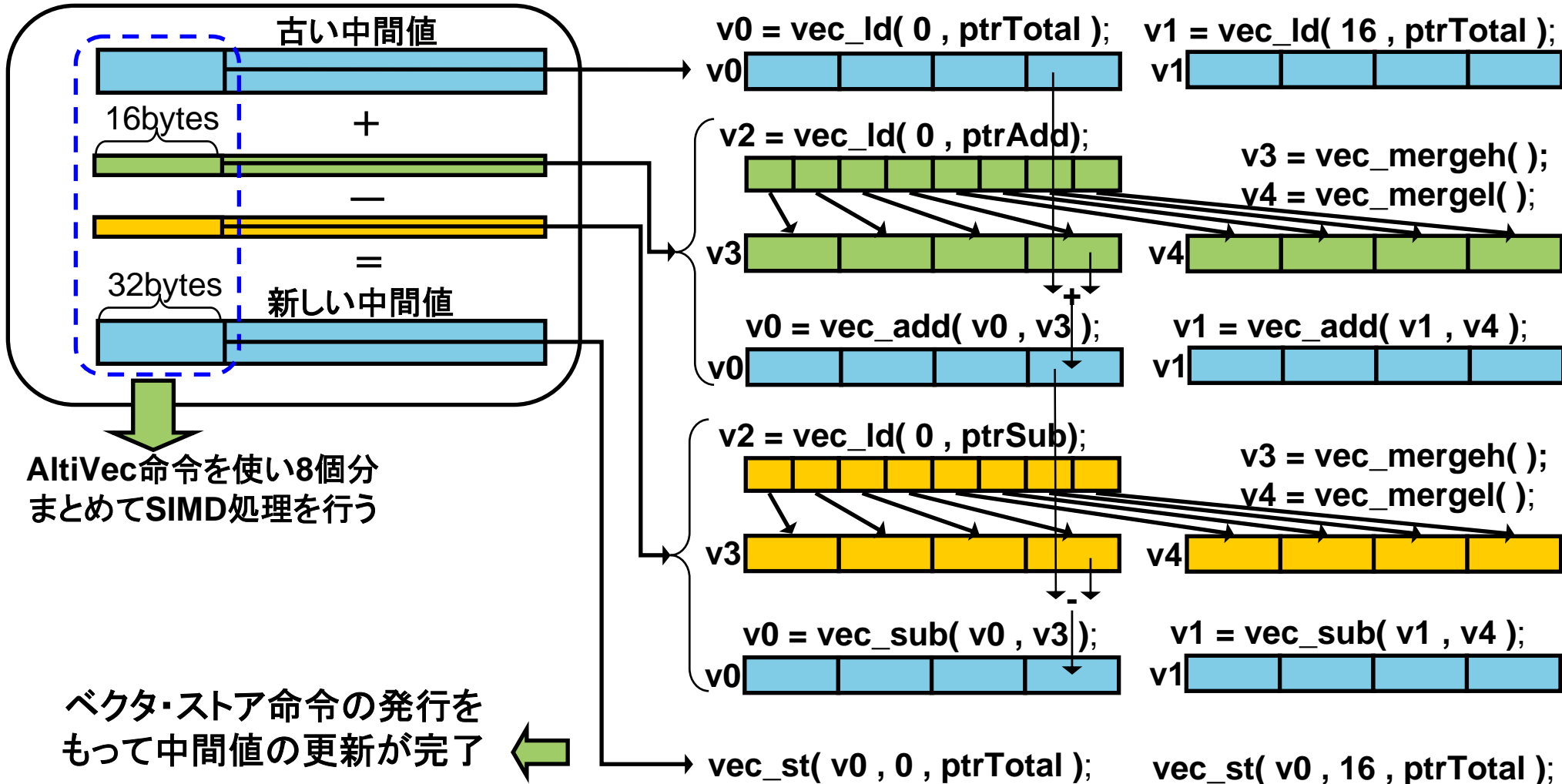
16 x M 回

$2 \times M + 2 \times (M-1) + 14$ 回
 $\doteq 4 \times M$ 回

Altivec命令を用いたコードレベル最適化

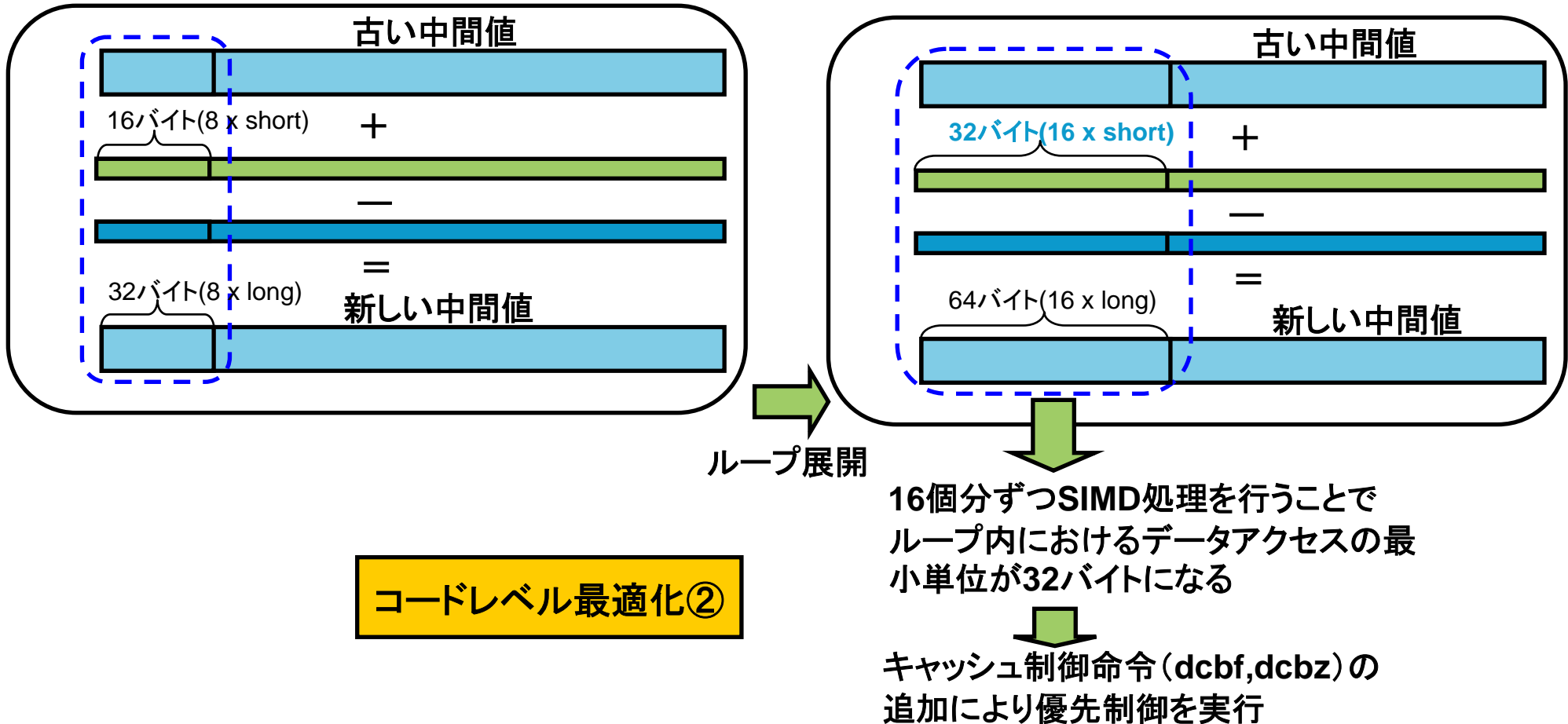
コードレベル最適化①

- 中間値を算出する処理をAltivec命令を用いて4倍に高速化



キャッシュ制御命令を用いたコードレベル最適化

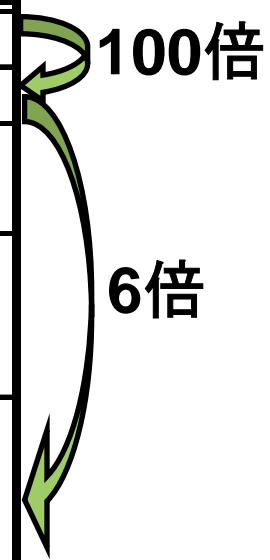
- キャッシュラインサイズ(32バイト)でアクセスするためにループ展開を行う



- 性能は正しい手順で最適化を加えてゆくことで段階的に向上する
- 最適化の初期段階で行うアルゴリズム最適化が極めて重要

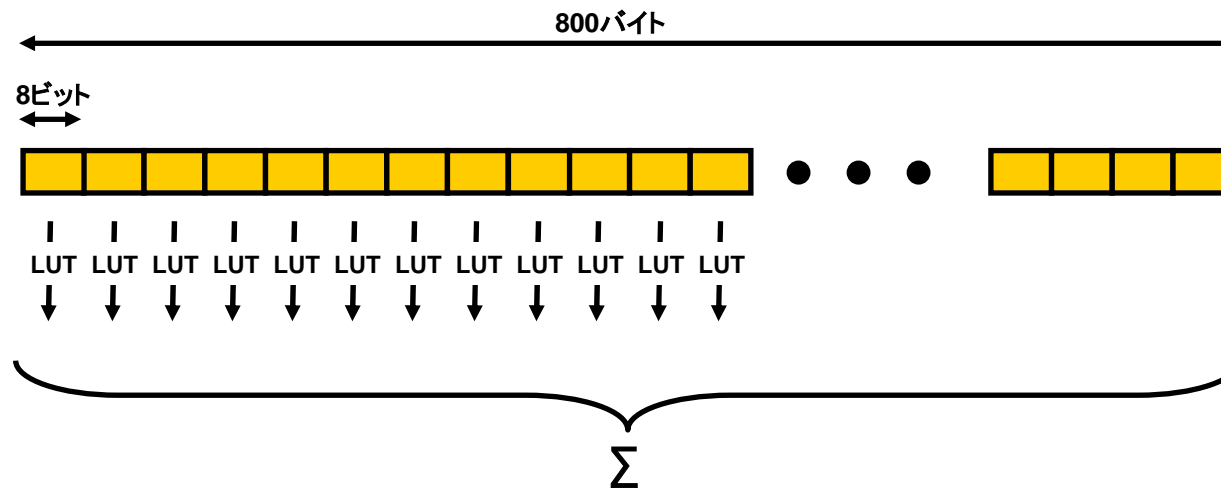
MPC7457 Core:1GHz, Bus:100MHz	
使用した最適化手法	Time(ns/pixel)
オリジナル	6444.0
アルゴリズム最適化	65.8
アルゴリズム最適化 AltiVecによる並列処理	21.6
アルゴリズム最適化 AltiVecによる並列処理 キャッシュの優先制御	18.1
アルゴリズム最適化 AltiVecによる並列処理 キャッシュの優先制御 データの先読み	9.9

コードレベル最適化



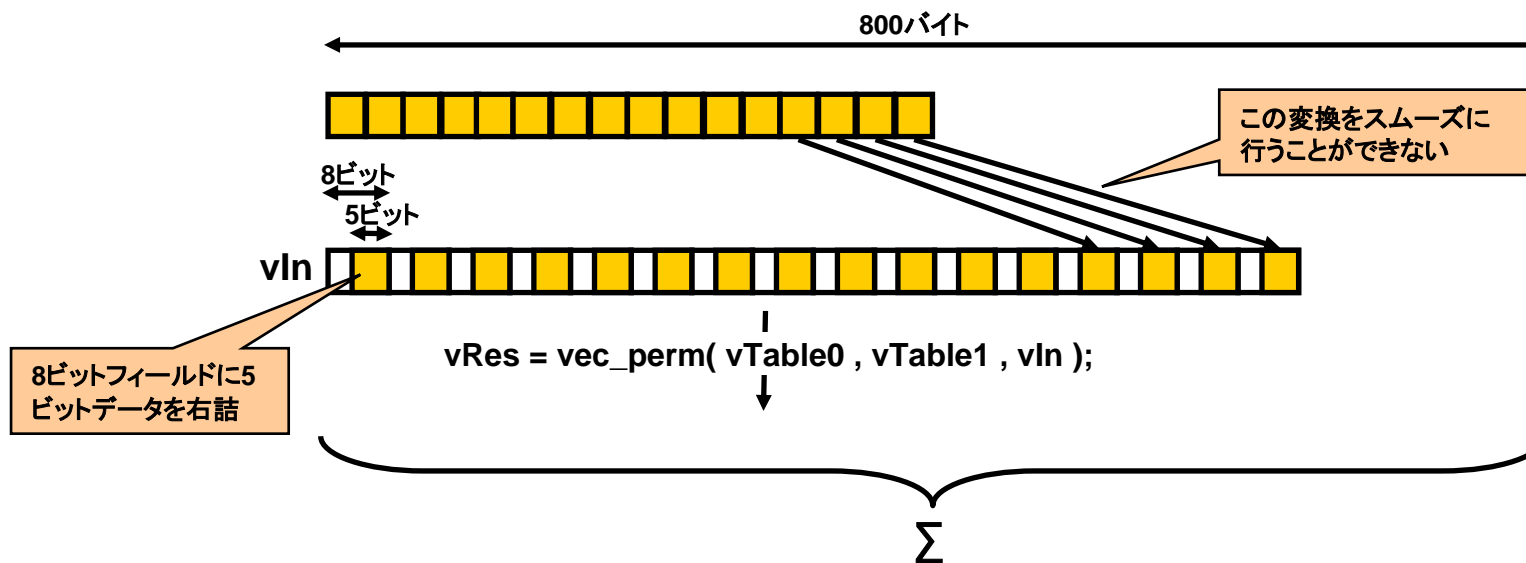
ビットカウント処理を例にした最適化の思考プロセス①

- 800バイト分の空間において、1がセットされている数を計測する処理を考える
- まず、テーブル参照(LUT: Look Up Table)を用いて8ビットずつ処理するアルゴリズムを考える
- ここで、16ビット毎にLUT処理してしまうと、テーブルが64Kバイト(2^{16})も必要になりメモリアクセスの効率性が落ちるため、256バイト(2^8)で済む8ビット毎のLUTで行う



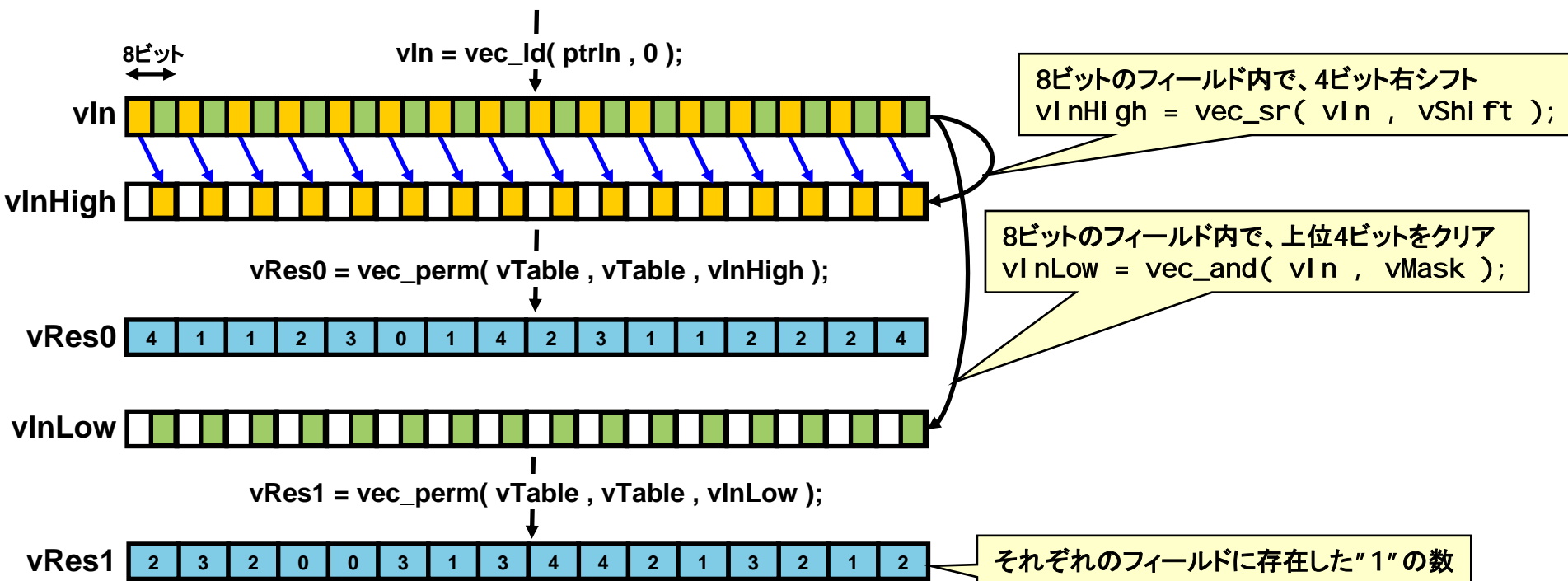
ビットカウント処理を例にした最適化の思考プロセス②

- ここでAltiVecのパミュート命令を使った最適化を考える
- パミュート命令は、本質的に16個まとめて32エントリのテーブル参照を行う処理であるため、活用の可能性を探る
- パミュート命令を用いて32エントリのテーブル参照を行う際には、入力データを5ビット以下の単位であることが望ましい(以下のイメージ)
- ただし以下の例では入力データを5ビットに区切る処理を効率的に行えないという障害が残る



ビットカウント処理を例にした最適化の思考プロセス③

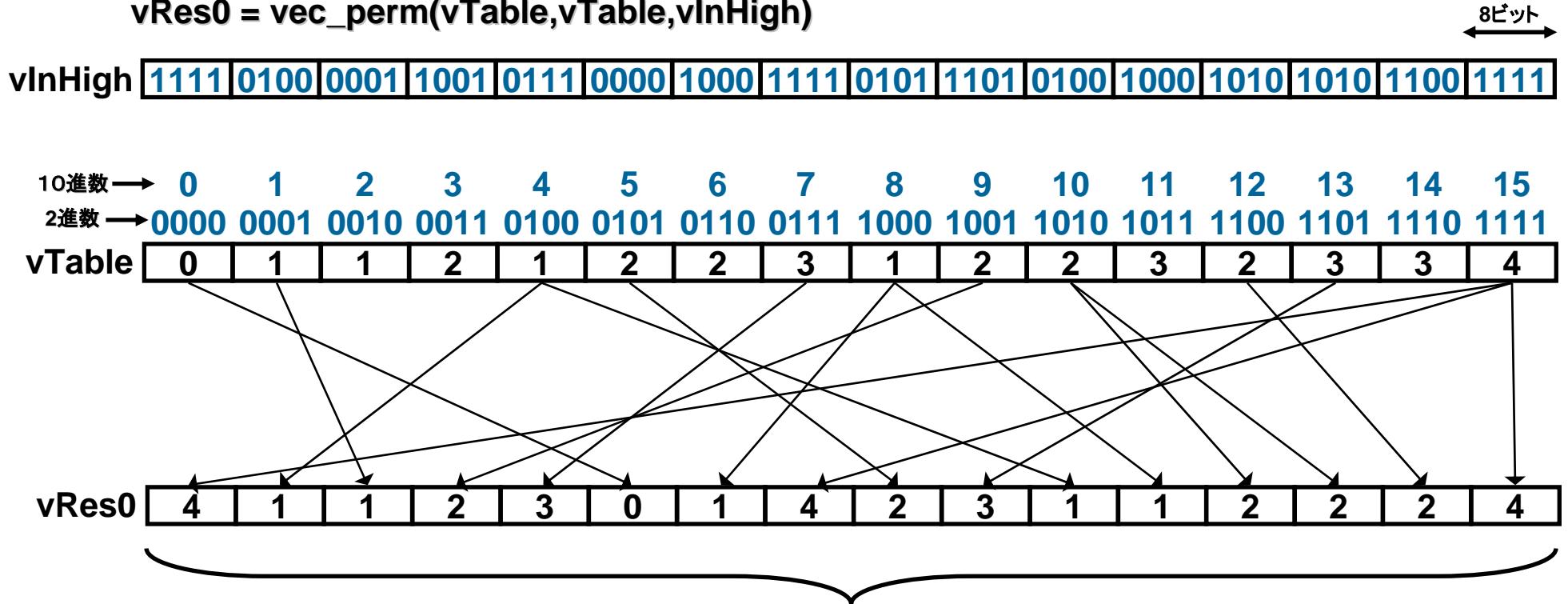
- 入力データを区切る処理を効率的に行うためには、4ビット単位に入力データを区切り、16個まとめて16エントリのテーブル参照を行う



ビットカウント処理を例にした最適化の思考プロセス④

- パミュート命令を活用して、16エントリ(入力4ビット)のテーブル参照を行う

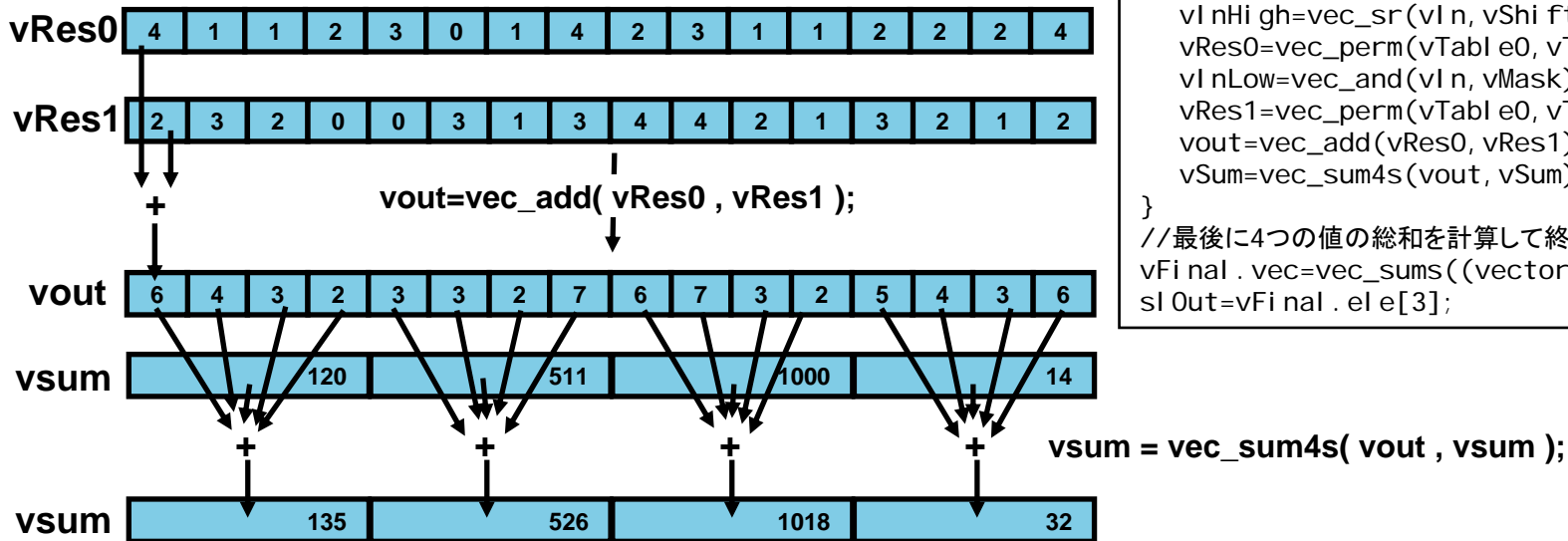
vRes0 = vec_perm(vTable, vTable, vInHigh)



全て加算するとvInHighに含まれる"1"の数

ビットカウント処理を例にした最適化の思考プロセス⑤

- 8ビット型に1がセットされていた総数を格納しておく、オーバフローしてしまうため、32ビット型で格納する



```

for (i=0; i<800/16; i++)
{
    vIn = vec_ld(0, ptrIn);
    ptrIn += 16;
    vInHigh=vec_sr(vIn, vShift);
    vRes0=vec_perm(vTable0, vTable0, vInHigh);
    vInLow=vec_and(vIn, vMask);
    vRes1=vec_perm(vTable0, vTable0, vInLow);
    vout=vec_add(vRes0, vRes1);
    vSum=vec_sum4s(vout, vSum);
}
//最後に4つの値の総和を計算して終了
vFinal.vec=vec_sums((vector_signed_int)vSum, vZero);
slOut=vFinal.ele[3];
    
```

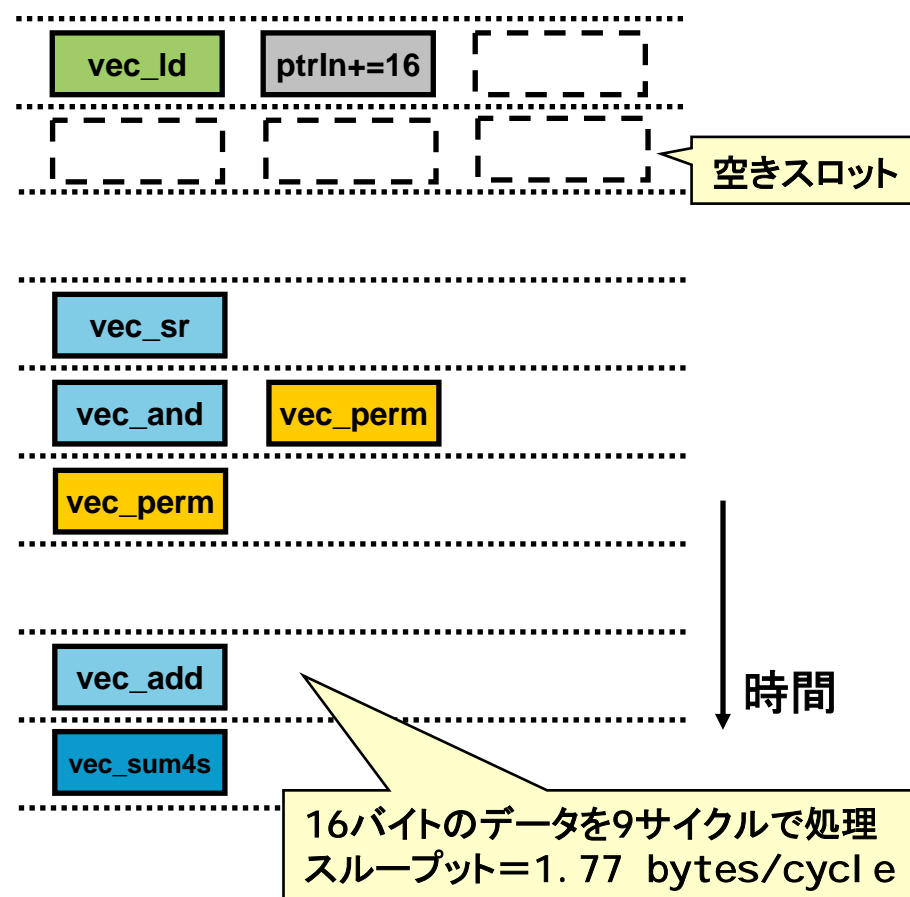
ビットカウント処理を例にした最適化の思考プロセス⑥

- ループ処理の中で行われている処理に注目すると非効率な空きスロットが目立つ
- この空きスロットを活用することでさらに性能を向上させることを検討する

```
for (i=0; i<800/16; i++)  
{  
    vln = vec_ld(0, ptrln);  
    ptrln += 16;  
    vlnHigh=vec_sr(vln, vShift);  
    vlnLow=vec_and(vln, vMask);  
    vRes0=vec_perm(vTblE0, vTblE0, vlnHigh);  
    vRes1=vec_perm(vTblE0, vTblE0, vlnLow);  
    vout=vec_add(vRes0, vRes1);  
    vSum=vec_sum4s(vout, vSum);  
}
```



命令のレイテンシを考慮した処理フロー



ベクトル演算ユニット

整数(複雑)

整数(単純)

浮動小数点

パミュート

レイテンシ:4

レイテンシ:1

レイテンシ:4

レイテンシ:2

整数ユニット

整数(複雑)

整数(単純)

レイテンシ:3 - 23

レイテンシ:1 - 2

ロード/ストア
ユニット

レイテンシ:3

浮動小数点ユニット

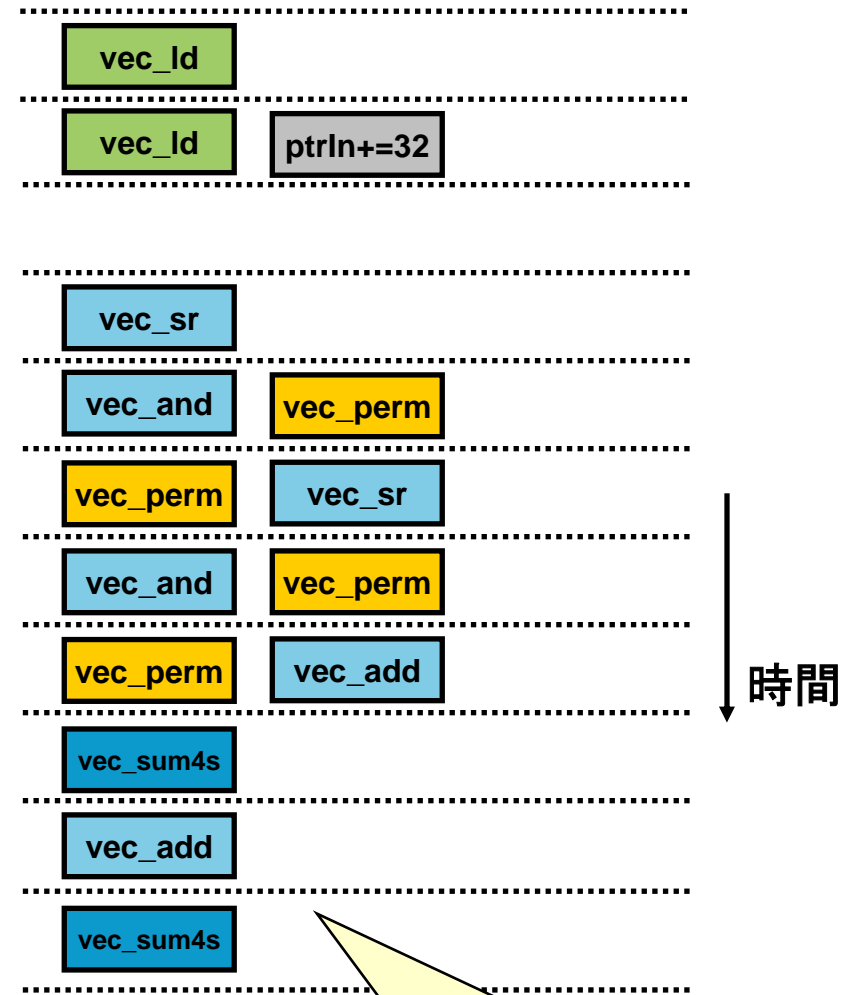
浮動小数点

レイテンシ:5 - 35

ビットカウント処理を例にした最適化の思考プロセス⑦

- ループ展開を用いて最適化

```
for (i=0; i<800/16/2; i++)  
{  
  vl nA = vec_ld(0, ptrIn);  
  vl nB = vec_ld(16, ptrIn);  
  ptrIn += 32;  
  vl nHi ghA=vec_sr(vl nA, vShi ft);  
  vl nLowA=vec_and(vl nA, vMask);  
  vRes0A=vec_perm(vTbl e0, vTbl e0, vl nHi ghA);  
  vRes1A=vec_perm(vTbl e0, vTbl e0, vl nLowA);  
  vl nHi ghB=vec_sr(vl nB, vShi ft);  
  vl nLowB=vec_and(vl nB, vMask);  
  vRes0B=vec_perm(vTbl e0, vTbl e0, vl nHi ghB);  
  vRes1B=vec_perm(vTbl e0, vTbl e0, vl nLowB);  
  voutA=vec_add(vRes0A, vRes1A);  
  vSumA=vec_sum4s(voutA, vSumA);  
  voutB=vec_add(vRes0B, vRes1B);  
  vSumB=vec_sum4s(voutB, vSumB);  
}
```



32バイトのデータを11サイクルで処理
スループット=2.91 bytes/cycle

ビットカウント処理を例にした最適化の思考プロセス⑧

さらにパイプライン処理で最適化

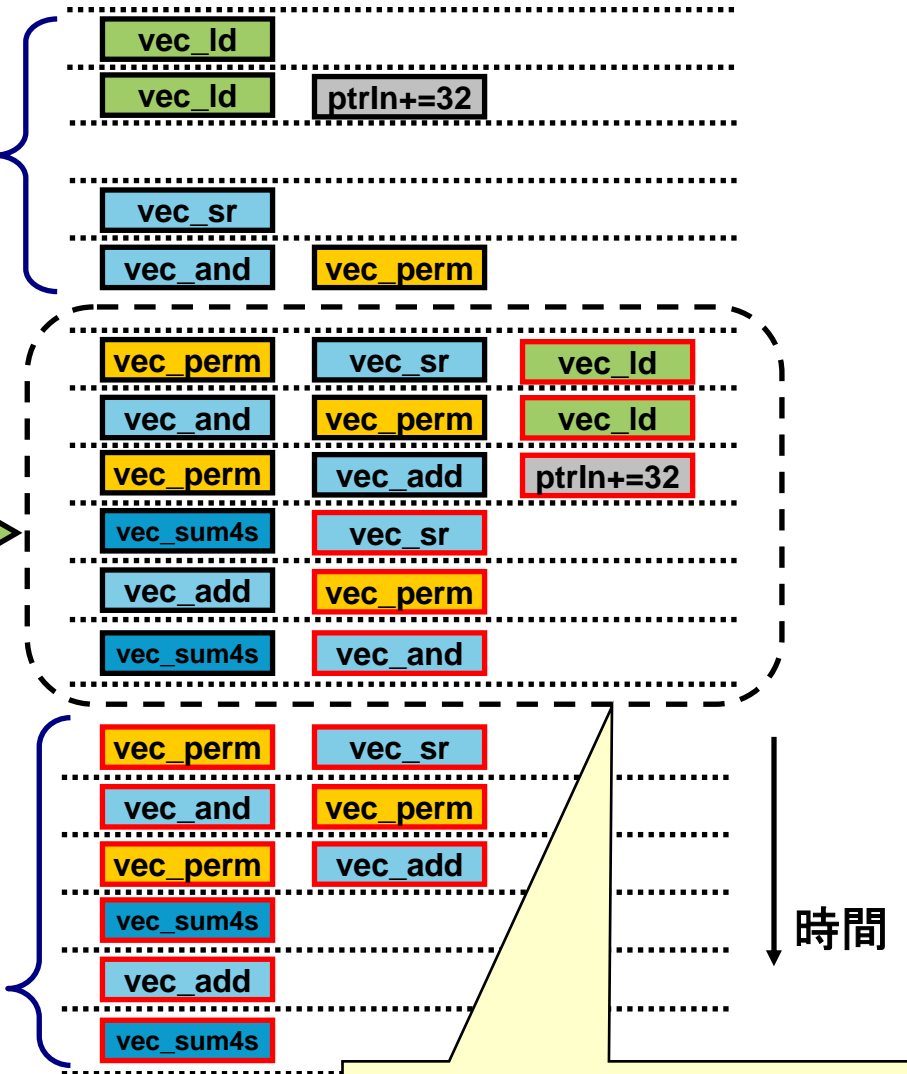
```

vInA = vec_ld(0, ptrIn);
vInB = vec_ld(16, ptrIn);
ptrIn += 32;
vInHi ghA=vec_sr(vInA, vShi ft);
vInLowA=vec_and(vInA, vMask);
vRes0A=vec_perm(vTabl e0, vTabl e0, vInHi ghA);
for (i=0; i<(800/16/2)-1; i++)
{
    vRes1A=vec_perm(vTabl e0, vTabl e0, vInLowA);
    vInHi ghB=vec_sr(vInB, vShi ft);
    vInA = vec_ld(0, ptrIn);
    vInLowB=vec_and(vInB, vMask);
    vRes0B=vec_perm(vTabl e0, vTabl e0, vInHi ghB);
    vInB = vec_ld(16, ptrIn);
    vRes1B=vec_perm(vTabl e0, vTabl e0, vInLowB);
    voutA=vec_add(vRes0A, vRes1A);
    ptrIn += 32;
    vSumA=vec_sum4s(voutA, vSumA);
    vInHi ghA=vec_sr(vInA, vShi ft);
    voutB=vec_add(vRes0B, vRes1B);
    vRes0A=vec_perm(vTabl e0, vTabl e0, vInHi ghA);
    vSumB=vec_sum4s(voutB, vSumB);
    vInLowA=vec_and(vInA, vMask);
}
vRes1A=vec_perm(vTabl e0, vTabl e0, vInLowA);
vInHi ghB=vec_sr(vInB, vShi ft);
vInLowB=vec_and(vInB, vMask);
vRes0B=vec_perm(vTabl e0, vTabl e0, vInHi ghB);
vRes1B=vec_perm(vTabl e0, vTabl e0, vInLowB);
voutA=vec_add(vRes0A, vRes1A);
vSumA=vec_sum4s(voutA, vSumA);
voutB=vec_add(vRes0B, vRes1B);
vSumB=vec_sum4s(voutB, vSumB);
    
```

端数処理

ループ部

端数処理



32バイトのデータを6サイクルで処理
スループット=5.33 bytes/cycle

ビットカウント処理を例にした最適化の思考プロセス⑨

- 実測値で5.3bytes/cyclesの性能が出なかった場合は以下の2点について調べる
 - コンパイラは予想通りのコード生成しているか？
 - 確認方法
 - ソースコードをツールのディスアセンブラ機能を使って、生成されるアセンブリコードを確認する
 - 対応策
 - C言語レベルで並べ替えて再コンパイルしてみる
 - registerを用いてベクタ変数を定義してみる
 - 別のコンパイラを試してみる
 - アセンブリコードで記述する
 - メモリアクセスはL1キャッシュから行われているか？
 - 確認方法
 - CPUに備わっているパフォーマンス・モニタの機能を使って、キャッシュミスを起こした回数をレポートさせる(ただし、この機能はMacOS上では使用不可)
 - データフローをキャッシュサイズと照らし合わせながら予測する
 - 対応策
 - データの先読み等を行い、メモリアクセスの効率を高める

- MPC7447Aのユーザーズ・マニュアル (MPC7450UM.pdf)

http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC7448&nodeId=0162468rH3bTdG8653

- AltiVecのインストラクションセット (ALTIVECPIM.pdf , ALTIVECPEM.pdf)

<http://www.freescale.com/webapp/sps/site/overview.jsp?nodeId=02VS0I81285Nf2>

- DINK32 (Sandpoint共通のコマンドラインデバッガ) のページ(英語)

http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=DINK32&parentCode=MPC7447A&nodeId=018rH3bTdG8653

- Freescale社が提供するAltiVecを使ったライブラリとサンプル・プログラム

<http://www.freescale.com/webapp/sps/site/taxonomy.jsp?nodeId=02VS0I81285Nf2F9DH>

- AltiVecユーザ同士が情報交換を行うサイト、MacOSで動作するサンプルコードも入手可能

<http://www.simdtech.org/altivec>

- Apple Computer Incが提供するAltiVecを使ったサンプル・プログラム

<http://developer.apple.com/hardware/ve/examples.html>

