CODEC屋のx86/x64最適化

茂木 和洋 @ まるも製作所

自己紹介

- まるも製作所の中の人をしてます
- 就職活動の一環として大学4年の夏にMPEG-2デコーダを作っていたら某企業に拾ってもらえました
- 就職先の上司の縁で、通信系の研究所に飛ばされて、H.264/AVCのエンコーダを作ったりしてました
- 現在はファブレスLSIメーカに転職してオリジナルのCODECを作ってたりします

発表内容

- 動画CODECのプログラム的特徴
- SIMDとは
- x86/x64のSIMD
- SIMDの使い方
- SIMDに向く処理/向かない処理
- 動画CODECでのSIMD活用例
- SIMDコードTips

動画CODECのプログラム的特徴

- 4x4/8x8/16x16のブロック単位処理が主流
- 画素毎に独立に同じ処理を行うことが多い
- 個々の処理はそれほど重くないが、処理対象が多い
- 8bit or 16bit の整数演算がほぼ全て

動画CODECのプログラム的特徴

- 4x4/8x8/16x16のブロック単位処理が主流
- 画素毎に独立に同じ処理を行うことが多い
- 個々の処理はそれほど重くないが、処理対象が多い
- 8bit or 16bit の整数演算がほぼ全て

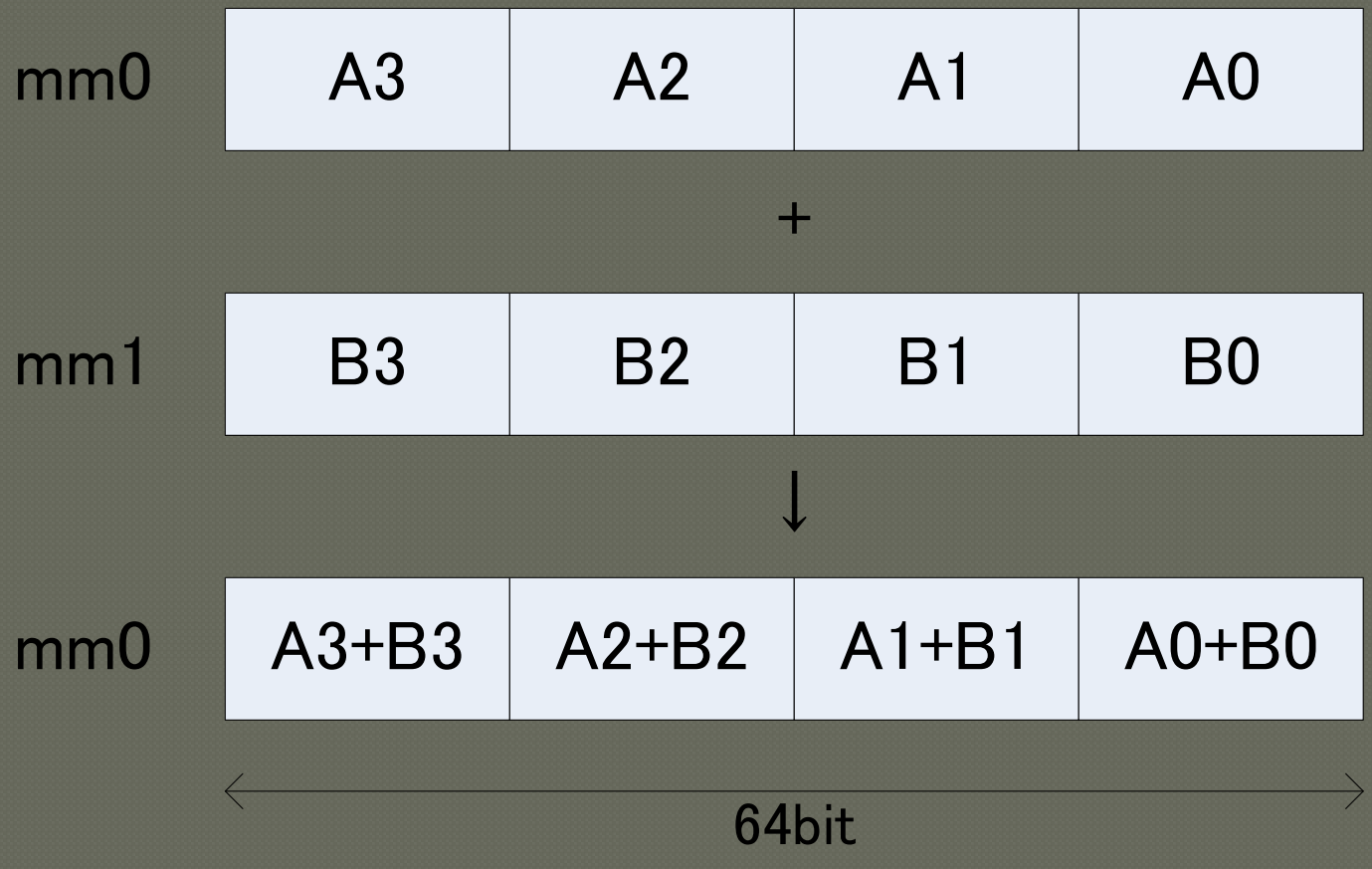
- 動画CODEC屋にとっては
  「最適化=SIMD化」

SIMDとは [1/2]

- 単一命令複数データ (Single Instruction Multiple Data)
- 大きなレジスタを8bit×8個とか16bit×4個などに分割して独立に同じ処理をする

SIMDとは [2/2]

SIMDの例： paddw mm0, mm1;



x86/x64 の SIMD [1/4]

- CPUの世代交代毎に新命令が追加されてきている
- MMX / SSE / SSE2 / SSE3 / SSSE3 / SSE4.1 / SSE4.2 / AVX / AVX2
- 動画CODECに重要な整数命令では
  - MMX で 64bit レジスタが
  - SSE2 で 128bit レジスタが
  - AVX2 で 256bit レジスタが

  それぞれ使えるようになる
- 段々並列度が上がり、便利な命令が追加されてきている

x86/x64 の SIMD [2/4]

	MMX	SSE	SSE2	SSE3	SSSE3	SSE4.1	SSE4.2	AVX	AVX2
Pentium4 [Willamette]	○	○	○						
Pentium4 [Presscotte]	○	○	○	○					
Core [Merom]	○	○	○	○	○				
Core 2 [Penryn]	○	○	○	○	○	○			
Core i7 [Nehalem]	○	○	○	○	○	○	○		
Core i3/i5/i7 [SandyBridge]	○	○	○	○	○	○	○	○	
?	○	○	○	○	○	○	○	○	○

x86/x64 の SIMD [3/4]

	MMX	SSE	SSE2	SSE3	SSSE3	SSE4.1	SSE4.2	AVX	AVX2
Athlon	○	△							
Athlon XP	○	○							
Athlon 64 [ClawHammer]	○	○	○						
Athlon 64 [Venice]	○	○	○	○					
Bobcot	○	○	○	○	○				
Bulldozer	○	○	○	○	○	○	○	○	

x86/x64 の SIMD [4/4]

- 原則、並列度の高い命令や新しい便利な命令を使った方が高速になるものの・・・
- 古いCPUで動かなくなってしまうので、古い命令しか使わない実装も用意して動的に切り替える必要がある
- 正直な話、かなりしんどい
- SSE2を前提にしても良いのではと思うが、無印 Athlon (ThunderBird) ユーザから動かないというレポートがまだ来る

SIMDの使い方 [1/7]

- C/C++コンパイラは普通、使ってくれない
- 一般的な手法として次の３種がある
  - intrinsic 命令を使う
  - インラインアセンブラで書く
  - アセンブリ言語で関数を書いてリンクする
- 特殊手法としてこんな手段も
  - xbyak で書く
  - NT^2 (boost.simd) で書く

SIMDの使い方 [2/7]

```
#include <emmintrin.h>
void __stdcall add_residual_16x16_sse2(unsigned char *block, const short *residual)
{ // intrinsic 命令のコードサンプル
    __m128i w0,w1,w2,w3;
    __m128i zero = _mm_setzero_si128();
    for (int i=0;i<16;i++) {
            w0 = _mm_loadl_epi64((__m128i const *)(block+0));
            w1 = _mm_loadl_epi64((__m128i const *)(block+8));
            w2 = _mm_loadu_si128((__m128i const *)(residual+0));
            w3 = _mm_loadu_si128((__m128i const *)(residual+8));
            residual += 16;
            w0 = _mm_unpacklo_epi8(w0, zero);
            w1 = _mm_unpacklo_epi8(w1, zero);
            w0 = _mm_add_epi16(w0, w2);
            w1 = _mm_add_epi16(w1, w3);
            w0 = _mm_packus_epi16(w0, w1);
            _mm_storeu_si128((__m128i *)block, w0);
            block += 16;
    }
}
```

SIMDの使い方 [3/7]

```
void __stdcall add_residual_16x16_sse2(unsigned char *block, const short *residual)
{// インラインアセンブラのコードサンプル
    __asm {
            mov esi, residual;
            mov edi, block;
            mov ecx, 16;
            pxor xmm7, xmm7;
    LOOP_HEAD:
            movq xmm0, qword ptr [edi+0];
            movq xmm1, qword ptr [edi+8];
            movdqu xmm2, oword ptr [esi+ 0];
            movdqu xmm3, oword ptr [esi+16];
            add esi, 32;
            punpcklbw xmm0, xmm7;
            punpcklbw xmm1, xmm7;
            paddw xmm0, xmm2;
            paddw xmm1, xmm3;
            packuswb xmm0, xmm1;
            movdqu oword ptr [edi+0], xmm0;
            add edi, 16;
            sub ecx, 1;
            jnz LOOP_HEAD;
    };
}
```

SIMDの使い方 [4/7]

```
; アセンブリ言語でのコードサンプル
; nasm (__stdcall) 形式

section .text

global _add_residual_16x16_sse2@8
_add_residual_16x16_sse2@8:
    push edi;
    push esi;
    push ecx;
    mov edi, [esp+12+ 4];
    mov esi, [esp+12+ 8];
    mov ecx, 16
    pxor xmm7, xmm7;
LOOP_HEAD:
    movq xmm0, [edi+0];
    movq xmm1, [edi+8];
    movdqu xmm2, [esi+ 0];
    movdqu xmm3, [esi+16];
    add esi, 32;
    punpcklbw xmm0, xmm7;
    punpcklbw xmm1, xmm7;
    paddw xmm0, xmm2;
    paddw xmm1, xmm3;
    packuswb xmm0, xmm1;
    movdqu [edi+0], xmm0;
    add edi, 16;
    sub ecx, 1;
    jnz LOOP_HEAD;
    pop ecx;
    pop esi;
    pop edi;
    ret 8;
```

SIMDの使い方 [5/7]

- intrinsicで書く
  - 利点：楽 / gccとVCで同じコードが使える / 32bitと64bitで同じコードが使える
  - 欠点：コンパイラのレジスタ管理の品質が・・・
- インラインアセンブラで書く
  - 利点：スタックの管理やレジスタ退避が不要
  - 欠点：gccとVCで文法が違う / 64bit のVCでは使えない
- アセンブリ言語で関数を書いてリンク
  - 利点：コンパイラを選ばない（アセンブラは選ぶ）
  - 欠点：スタック管理・呼び出し規約等の意識が必要 / 必ず関数呼び出しになる（インライン展開されない）

SIMDの使い方 [6/7]

- xbyakについては・・・
  - もっと詳しい人がいるのでそちらに聞いてね

SIMDの使い方 [7/7]

- NT^2 とは
  - x86/x64 だけでなく、PowerPC の AltiVec やARM の NEON も統一的に扱えるようにしようという提案
  - 拡張後のものがboost.simdとして提案されている
  - http://www.slideshare.net/faithandbrave/boostsimd
  - 詳細は上記の日本語訳プレゼンを参照

SIMDが効く処理 [1/5]

- ひと固まりのデータに対して、各要素に同じ処理を行う場合
  - YUV <-> RGB 変換
  - FIR フィルタ
- 専用命令が用意されている処理
  - 動き検索のコスト評価 (psadbw)
- 処理の中でクリッピングがある場合

SIMDが効く処理 [2/5]

```
// YUV -> RGB 変換処理
void yuv2bgra(
    unsigned char *bgra, const unsigned char *luma,
    const unsigned char *cb,const unsigned char *cr,
    int width, int height) {
    for (int y=0;y<height;y++) {
        for (int x=0;x<width;x++) {
            int lw = (luma[x] - l_offset) * l_scale;
            int cbw = cb[x] - c_offset;
            int crw = cr[x] - c_offset;
            int b = (lw + cbw*ub_scale + round) >> shift;
            int g = (lw + cbw*ug_scale + crw*vg_scale + round) >> shift;
            int r = (lw + crw*vr_scale + round) >> shift;
            bgra[x*4+0] = clip_u8(b);
            bgra[x*4+1] = clip_u8(g);
            bgra[x*4+2] = clip_u8(r);
            bgra[x*4+3] = 0xff; // dummy alpha
        }
        bgra += (width*4);
        luma += width;
        cb += width;
        cr += width;
    }
}
```

SIMDが効く処理 [2/5]

```
// YUV -> RGB 変換処理
void yuv2bgra(
    unsigned char *bgra, const unsigned char *luma,
    const unsigned char *cb,const unsigned char *cr,
    int width, int height) {
    for (int y=0;y<height;y++) {
        for (int x=0;x<width;x++) {
            int lw = (luma[x] - l_offset) * l_scale;
            int cbw = cb[x] - c_offset;
            int crw = cr[x] - c_offset;
            int b = (lw + cbw*ub_scale + round) >> shift;
            int g = (lw + cbw*ug_scale + crw*vg_scale + round) >> shift;
            int r = (lw + crw*vr_scale + round) >> shift;
            bgra[x*4+0] = clip_u8(b);
            bgra[x*4+1] = clip_u8(g);
            bgra[x*4+2] = clip_u8(r);
            bgra[x*4+3] = 0xff; // dummy alpha
        } // このループを 4 or 8 画素単位の SIMD 処理に置き換える
        bgra += (width*4);
        luma += width;
        cb += width;
        cr += width;
    }
}
```

SIMDが効く処理 [3/5]

```
// FIR フィルタ (3 tap)
void filter_3x1(
    short *dst,
    const short *src,
    int length
    const short *weight) {
    for (int i=0;i<length;i++) {
            int w = src[i-1] * weight[-1];
            w += src[i+0] * weight[0];
            w += src[i+1] * weight[+1];
            w = (w+round) >> shift;
            dst[i] = clip_s16(w);
    }
}
```

SIMDが効く処理 [3/5]

```
// FIR フィルタ (3 tap)
void filter_3x1(
    short *dst,
    const short *src,
    int length
    const short *weight) {
    for (int i=0;i<length;i++) {
            int w = src[i-1] * weight[-1];
            w += src[i+0] * weight[0];
            w += src[i+1] * weight[+1];
            w = (w+round) >> shift;
            dst[i] = clip_s16(w);
    } // このループを 4 or 8 要素単位の SIMD 処理に置き換える
}
```

SIMDが効く処理 [4/5]

```
// 専用命令がある場合 (動き検索のブロックコスト評価)
int sad_16x16(
    const unsigned char *block,
    const unsigned char *ref_frame,
    int ref_stride) {
    int sad = 0;
    for (int y=0;y<16;y++) {
        for (int x=0;x<16;x++) {
            sad += abs(block[x]-ref_frame[x]);
        }
        block += 16;
        ref_frame += ref_stride;
    }
    return sad;
}
```

SIMDが効く処理 [4/5]

```
// 専用命令がある場合 (動き検索のブロックコスト評価)
int sad_16x16(
    const unsigned char *block,
    const unsigned char *ref_frame,
    int ref_stride) {
    int sad = 0;
    for (int y=0;y<16;y++) {
        for (int x=0;x<16;x++) {
            sad += abs(block[x]-ref_frame[x]);
        } // このブロックが psadbw に置き換え可能
        block += 16;
        ref_frame += ref_stride;
    }
    return sad;
}
```

SIMDが効く処理 [5/5]

```
// クリッピングを伴う処理
unsigned char clip_u8(short val) {
    if (val < 0) { return 0; }
    if (val > 255) { return 255;}
    return (unsigned char)255;
}

short clip_s16(int val) {
    if (val < -32768) { return -32768; }
    if (val > 32767) { return 32767; }
    return (short)val;
}

short clip(short val, short min, short max) {
    if (val < min) { return min; }
    if (val > max) { return max; }
    return val;
}
```

SIMDが効く処理 [5/5]

```
// クリッピングを伴う処理
unsigned char clip_u8(short val) {
    if (val < 0) { return 0; }
    if (val > 255) { return 255;}
    return (unsigned char)255;
} // 8 or 16 要素をまとめて packuswb で処理可能

short clip_s16(int val) {
    if (val < -32768) { return -32768; }
    if (val > 32767) { return 32767; }
    return (short)val;
} // 4 or 8 要素をまとめて packssdw で処理可能

short clip(short val, short min, short max) {
    if (val < min) { return min; }
    if (val > max) { return max; }
    return val;
} // 4 or 8 要素をまとめて pmaxsw/pminsw で処理可能

// これらが利用できると、分岐命令を潰せるので大幅に高速化する
// SSE4.1 (Core 2/Penryn 以降) で short 以外の pmax/pmin が追加されたので使い所が増加
```

SIMDが効かない処理 [1/5]

- 出力データからのフィードバックがある処理 (例：IIRフィルタ)
- 入力データに応じて処理内容が変化する処理（例：適応フィルタ）
- メモリネックな処理

SIMDが効かない処理 [2/5]

```
// 出力データからのフィードバック処理がある場合
void filter_iir(
    short *dst,
    const short *src,
    int length) {
    int pre = src[0];
    for (int i=0;i<length;i++) {
        dst[i] = clip_s16((src[i]+pre+1)>>1);
        pre = dst[i];
    } // フィードバックがあると SIMD 化不能
}
```

- シリアルな処理は並列(SIMD)化できない

SIMDが効かない処理 [3/5]

- 入力データに応じて処理が変わる場合（適応フィルタ）
  - 例：H.264 のデブロックフィルタ
- 分岐が1つならば両パターンを計算してビットマスク合成することで高速化できる場合も
- 実際にx264はデブロックフィルタをその手法で実装し、高速化している

SIMDが効かない処理 [4/5]

- メモリネックな処理はSIMD化してもあまり効果がない
- SIMDの使い方で出した add_residual_16x16_sse2() はメモリネックな処理の例

SIMDの使い方 [3/7]

```
void __stdcall add_residual_16x16_sse2(unsigned char *block, const short *residual)
{// インラインアセンブラのコードサンプル
    __asm {
            mov esi, residual;
            mov edi, block;
            mov ecx, 16;
            pxor xmm7, xmm7;
    LOOP_HEAD:
            movq xmm0, qword ptr [edi+0];
            movq xmm1, qword ptr [edi+8];
            movdqu xmm2, oword ptr [esi+ 0];
            movdqu xmm3, oword ptr [esi+16];
            add esi, 32;
            punpcklbw xmm0, xmm7;
            punpcklbw xmm1, xmm7;
            paddw xmm0, xmm2;
            paddw xmm1, xmm3;
            packuswb xmm0, xmm1;
            movdqu oword ptr [edi+0], xmm0;
            add edi, 16;
            sub ecx, 1;
            jnz LOOP_HEAD;
    };
}
```

SIMDが効かない処理 [4/5]

- メモリネックな処理はSIMD化してもあまり効果がない
- SIMDの使い方で出したadd_residual_16x16_sse2() はメモリネックな処理の例
- blockに書き戻すのではなく、最終出力先に直接出力することで不要なメモリIOを減らすのが有効

SIMDの使い方 [3/7]

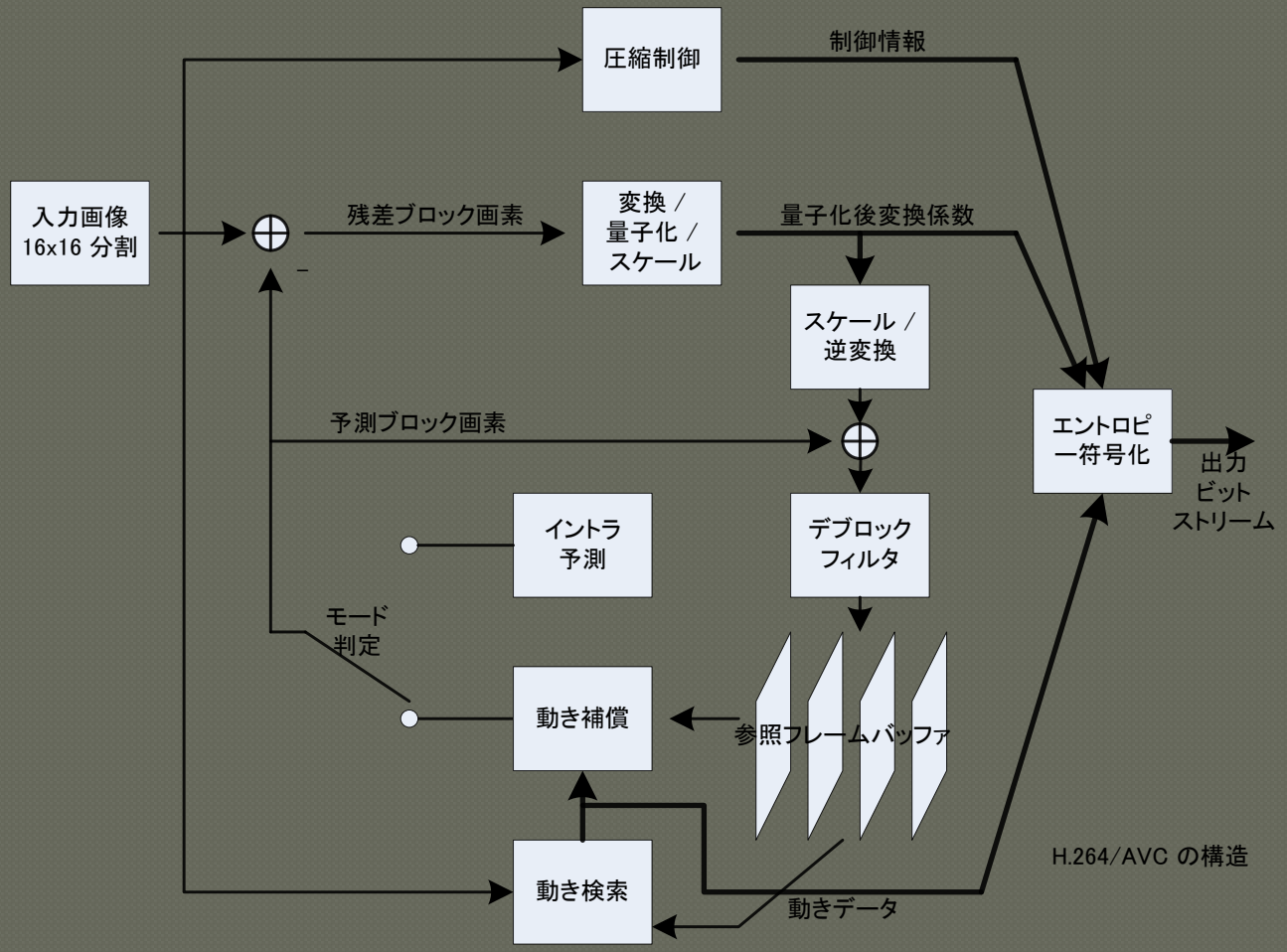
```
void __stdcall add_residual_16x16_sse2(
    unsigned char *frame,
    int frame_stride,
    unsigned char *block,
    const short *residual)
{// インラインアセンブラのコードサンプル
    __asm {
            mov esi, residual;
            mov edi, frame;
            mov eax, block;
            mov edx, frame_stride;
            mov ecx, 16;
            pxor xmm7, xmm7;
    LOOP_HEAD:
            movq xmm0, qword ptr [eax+0];
            movq xmm1, qword ptr [eax+8];
            add eax, 16;
            movdqu xmm2, oword ptr [esi+ 0];
            movdqu xmm3, oword ptr [esi+16];
            add esi, 32;
            punpcklbw xmm0, xmm7;
            punpcklbw xmm1, xmm7;
            paddw xmm0, xmm2;
            paddw xmm1, xmm3;
            packuswb xmm0, xmm1;
            movdqu oword ptr [edi+0], xmm0;
            add edi, edx;
            sub ecx, 1;
            jnz LOOP_HEAD;
    };
}
```

SIMDが効かない処理 [5/5]

- 出力データからのフィードバックがある処理 (例：IIRフィルタ)
- 入力データに応じて処理内容が変化する処理（例：適応フィルタ）
- メモリネックな処理

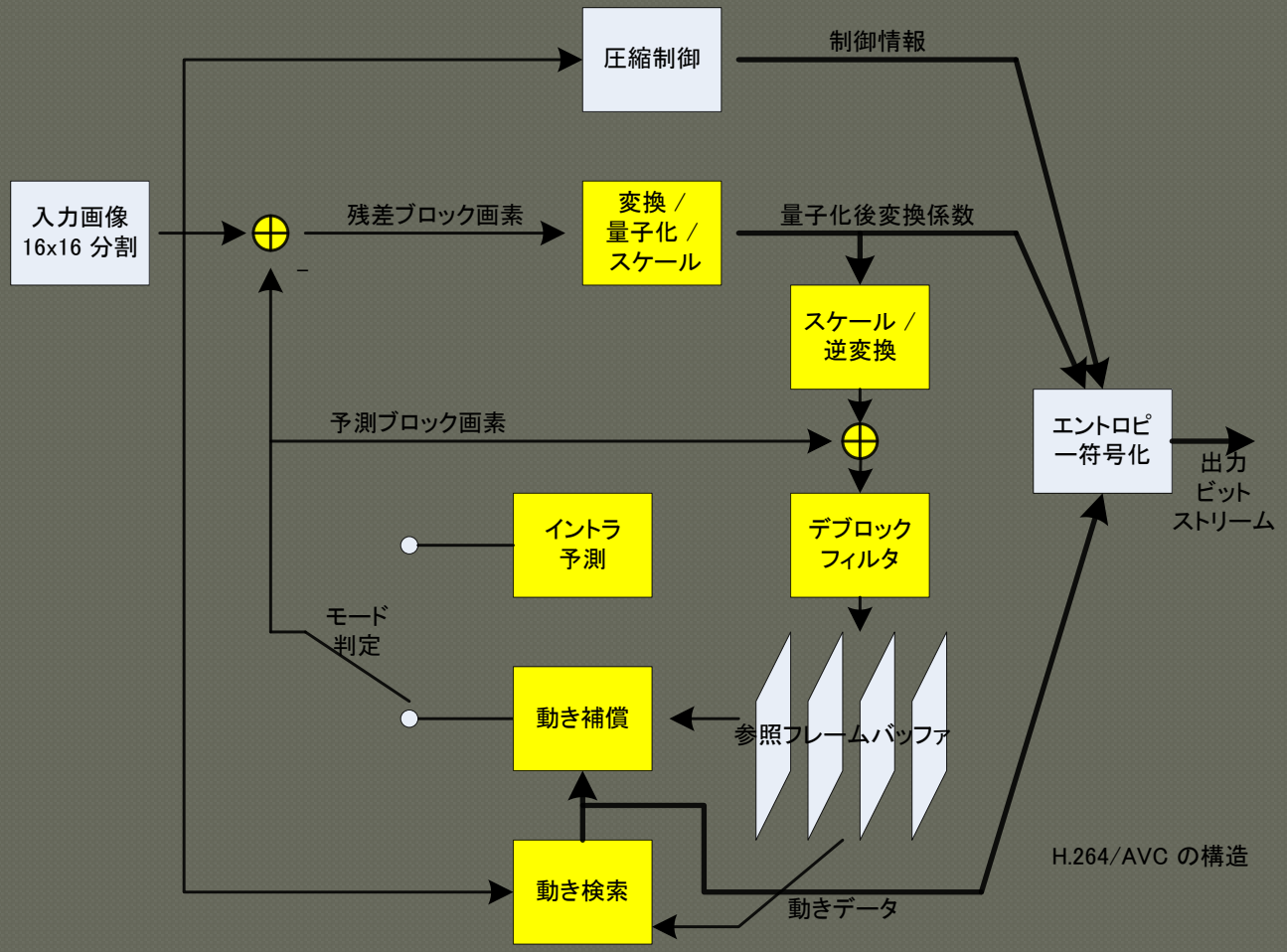
- オリジナルのCODECを作る時はこうした処理を避けよう

動画CODECでのSIMD活用例



H.264/AVC の構造

動画CODECでのSIMD活用例



H.264/AVC の構造

SIMDコードTips [1/6]

- 実装の動的切り替え
- 16 byte alignment の重要性
- 64bitビルドでのSIMD利用

SIMDコードTips [2/6]

- 実装の動的切り替え
  - C++ 継承/仮想関数
    - 仮想関数の解決がクソ重い
    - 場合によってはSIMD化の最適化効果を食いつぶしてしまう
  - 関数ポインタ
    - アセンブリ言語で関数を書く場合のほぼ唯一の選択肢
    - 仮想関数ほどでないものの、関数呼び出しはコストが高い（Mのオーダーで呼び出される処理では気にした方が良い）
  - C++ テンプレート
    - SIMD処理を使うcore部分と、core間を繋ぐlogicに分割
    - coreを引数にとるテンプレートクラスとしてlogicを実装して、関数呼び出し等の頻度を下げる

SIMDコードTips [3/6]

```
class foobar_core_sse2 {
public:
    static inline void huga(); // 中でSIMD処理
    static inline void hoge(); // 同上
    ... <略>
};
// nosimd 等も同様に作る

template<typename _T> class foobar_logic : public foobar_interface {
public:
    void sequence_of_proc() {
            _T::huga();
            // 途中で C/C++ の方が書きやすい処理を入れたり
            _T::hoge();
            ... <略>
    }
};

foobar_interface *foobar_interface::create() {
    // 本来は cpuid で実装を切り替える
    new foobar_logic<foobar_core_sse2>();
}
```

SIMDコードTips [4/6]

- 16 byte alignment の重要性
  - movdquとmovdqaで速度が4倍違う(penrynでのデータ/alignmentの取れているアドレスに対して)
  - 16 byte alignment があれば、SSE2 命令でもメモリをソースオペランドに書ける (なければ、不正アクセス例外)
  - レジスタが空いて、ループアンロールしやすくなる

SIMDコードTips [5/6]

```
; // alignment 保証がない場合
        mov esi, residual;
        mov edi, block;
        mov ecx, 16;
        pxor xmm7, xmm7;
    LOOP_HEAD:
        movq xmm0, qword ptr [edi+0];
        movq xmm1, qword ptr [edi+8];
        movdqu xmm2, oword ptr [esi+ 0];
        movdqu xmm3, oword ptr [esi+16];
        add esi, 32;
        punpcklbw xmm0, xmm7;
        punpcklbw xmm1, xmm7;
        paddw xmm0, xmm2;
        paddw xmm1, xmm3;
        packuswb xmm0, xmm1;
        movdqu oword ptr [edi+0], xmm0;
        add edi, 16;
        sub ecx, 1;
        jnz LOOP_HEAD;

; // Core i5 2500 で 0x4000000 (64*1024*1024) 回
; // の呼び出しで
; // avg: 1390, max: 1482, min: 1326 [msec]
```

```
; // alignment 保証がある場合
        mov esi, residual;
        mov edi, block;
        mov ecx, 8;
        pxor xmm7, xmm7;
    LOOP_HEAD:
        movq xmm0, qword ptr [edi+0];
        movq xmm1, qword ptr [edi+8];
        movq xmm2, qword ptr [edi+16];
        movq xmm3, qword ptr [edi+24];
        punpcklbw xmm0, xmm7;
        punpcklbw xmm1, xmm7;
        punpcklbw xmm2, xmm7;
        punpcklbw xmm3, xmm7;
        paddw xmm0, [esi+0];
        paddw xmm1, [esi+16];
        paddw xmm2, [esi+32];
        paddw xmm3, [esi+48];
        packuswb xmm0, xmm1;
        packuswb xmm2, xmm3;
        movdqa oword ptr [edi+0], xmm0;
        movdqa oword ptr [edi+0], xmm2;
        add esi, 64;
        add edi, 32;
        sub ecx, 1;
        jnz LOOP_HEAD;
; // avg: 953, max: 983, min: 936 msec
```

SIMDコードTips [6/6]

- 64bitビルドでのSIMD利用
  - Microsoft Visual C++ では、64bit ビルドでインラインアセンブラが利用できない
  - VC では intrinsic を使うか、アセンブリで関数を書くか以外の選択肢がないが…
  - Intel C/C++ Compiler (現 Intel Composer XE) なら 64bit ビルドでもインラインアセンブラが使えるので、そちらに逃げる方法あり

おまけ

- SandyBridge の qucik sync video とは
  - CPUにMPEG-2 の HW デコーダと H.264/AVC の HW エンコーダが載ってる
  - ソフト側でやるのは HWデコーダ/HWエンコーダ呼び出すだけ (ここが Intel Media SDK 部分)
  - GPGPUとは別の意味で x86/x64 最適化から外れる