高機能アセンブラによる x86/x64 CPU 向け高速化テクニック

光　成　滋　生†

昨今のコンパイラやインタプリタの技術の進歩には目を見張るものがある.「アセンブリ言語で書くより，コンパイラなどの高機能言語に任せた方がよい」と言われて久しい．しかし，それでもなお多くのパソコンに搭載されている Intel 系 CPU の命令を真に生かしたコードを生成するのは難しいのが実情である．ここでは，C++を x86/x64 用高機能アセンブラとして利用するライブラリを紹介する．そして CPU に応じたコード生成，および文字列処理などのいくつかの基本的な処理における高速化テクニックの例をあげる．

An optimization technique for x86/x64 CPU by rich assembler

MITSUNARI Shigeo †

We propose a just-in-time assembler for x86/x64 using C++ and use it for code-generation, fast string processing, and some elementary functions.

1. はじめに

x86/x64 環境において C++をメインに開発し，一部の処理の高速化のためにアセンブリ言語を使う場合，いくつかの方法がある．

- コンパイラが持つインラインアセンブラの機能を使う．ただしこれはコンパイラの種類に強く依存し，バージョンによっても記法が異なることがある．また 64bit Visual Studio においてはインラインアセンブラの機能が廃止された．
- NASM，YASM，gas などのアセンブラを用いる．これは古くから使われる一般的な手法であるが，マクロや制御構造を用いて複雑な記述をしたい場合，アセンブラがもつ擬似命令を駆使する必要があり可読性にかけるきらいがある．
- LLVM，GNU lightning などのコンパイラ基盤を用いる．仮想機械をターゲットにした中間言語を生成するため x86/x64 以外の環境でも高速化が望める．ただし x86/x64 向けの最適化をしたい場合，抽象レイヤが挟まるため意図したコード生成の制御が難しい場合がある．

著者は x86/x64 をターゲットにした Xbyak☆ というアセンブラを開発している．これは C++のヘッダファイルのみからなり，標準的な C++コンパイラを用いて C++ソースファイルからインクルードするだけで使える．C++の文法を用いて制御構造を記述できるため NASM や YASM のような擬似命令を使う必要がない．そのため既存の手法に比べて可読性が高い．

また実行時にコード生成を行う JIT の機能も持っている．これにより実行時の CPU の特性に応じた最適化を行いやすい．以下ではこのアセンブラの紹介と，それを用いた最適化の例について述べる．

2. Xbyak の特長

Xbyak は次の特長を持つ．

- x86/x64 用 Windows, Linux, Mac OS X 上の Visual Studio/gcc/clang などに対応
- C++ヘッダファイルのみからなる
- MASM 形式に似せたアドレッシングを提供
- 実行時コード生成

Xbyak は一部の処理系依存のシステムコールを除いて標準的な C++で記述されている．C++ソースファイルからインクルードすると使え，外部ライブラリを用いたビルドの設定やライブラリのリンクが不要である．

内部 DSL による文法は，MASM などの Intel 記法に慣れた人になるべく違和感を抱かせないような設計を試みた．C++の文法上，引数に括弧をつける必要はあるが，たとえばコード 1 のような記述ができる．

ptr [...] や dword [...] などの表記は C++の演算子オーバーロードなどを用いて実現した．

† サイボウズ・ラボ
Cybozu Labs, Inc.

☆ http://homepage1.nifty.com/herumi/soft/xbyak.html

コード 1. コード例

```
// アドレッシング
add(ptr [ecx + edx * 4], eax);
movzx(rax, byte [rsp + rdi + 12]);

// 構造体のメンバへのアクセス
template<size_t N>
struct BoxT {
  int w[N];
  int h[N];
};
typedef BoxT<20> Box;
mov(eax, dword [esp+offsetof(Box, h)]);
```

構造体のメンバへのオフセットを取得する offsetof は stddef.h で定義されている標準マクロである．構造体のオフセットを複数言語で共用する場合，その値を他言語に伝えるのはやや面倒である．わずらわしさを避けるために，int や uint32_t などしか使わない，メンバの順序は動かさないなどの制約を入れることが多い．マクロやテンプレートを用いて配列のテーブルサイズを可変にすることも難しい．それに対して Xbyak では，C++コンパイラ自身が値をもっているためシームレスに扱える．

実行時コード生成の例として，まず整数 n が与えられたときに「n を足す関数」を生成してみる（コード 2）．Xbyak::CodeGenerator を継承し，その中でメンバ関数として x86/x64 ニーモニックを呼び出す．

コード 2. 足し算関数を生成する

```
#include <xbyak/xbyak.h>
struct Code : Xbyak::CodeGenerator {
  Code(int n) {
    // 32bit OS
    mov(eax, ptr [esp + 4]);
    add(eax, n);
    ret();
  }
};
```

生成されたコードを実行するには，Code クラスのインスタンスを生成し，関数ポインタを取り出して呼び出せばよい．

コード 3. 足し算関数を呼び出す

```
int main(int argc, char *argv[]) {
  int n = argc == 1 ? 0 : atoi(argv[1]);
  Code code(n);
  int (*add)(int)
   = (int (*)(int))code.getCode();
  printf("%d\n", add(3));
}
```

コード 3 のプログラムの引数に 7 を与えてデバッガ上で実行すると，コード 4 のような関数が実行時に生成されていることを確認できる．ここでは add の引数が即値になっていることに注意する．プログラムに与える引数を変更するとそれに応じてこの即値も変わる．

コード 4. 生成された add 関数

```
mov  eax, dword ptr [esp+4]
add  eax, 7
ret
```

3. 実行時コード生成

ここでは実行時コード生成を適用しやすい例を述べる．『ビューティフルコード』[1] の 8 章では画像処理のためのその場コード生成について述べられている．引数を多く持つが，処理中はそれらの値が定数であるような関数が，画像処理ではよく現れる．例として与えられた二つの入力ビットマップデータから，ピクセル単位で論理演算をして新しいビットマップデータを出力する BitBlt 関数がある．

コード 5 はその一部を抜き出し簡略化したものである．ループの一番内側に switch 文があるため関数実行時に極めて多くの分岐命令が呼ばれる．予め op に応じた全てのパターンの関数を用意しておくと高速化ができる．ただし op の種類は多いが特定の op のみがよく使われる場合，使われないバイナリコードが増える．

1985 年の Windows では，要求されたときに op に対するコードを生成するミニコンパイラを持っていたとのことである．このような実行時コード生成処理は従来の静的なアセンブリ言語では記述できない．

コード 6 はその中核部分である．switch 文による分岐はコード生成時に一度行われるだけであることに注意する．また C/C++の通常の文法を用いてアセンブリ言語の制御構造を記述できるため可読性が高いことも分かる．

コード 5. BitBltC

```
for (int i = 0; i < y; i++) {
  for (int j = 0; j < x; j++) {
    switch (op) {
    case 0: *dst = 0; break;
    case 1: *dst &= *src; break;
    case 2: *dst ^= *src; break;
    case 3: *dst |= *src; break;
    ...
    }
    dst++; src++;
```

コード 6. Xbyak による BitBlt

```
L(".lp");
  switch (op) {
  case 0:
    mov(ptr [dst], eax); break;
  case 1:
    mov(eax, ptr [src]);
    mov(ptr [dst], eax); break;
  case 2:
    mov(eax, ptr [src]);
    xor(ptr [dst], eax); break;
  case 3:
    mov(eax, ptr [src]);
    or(ptr [dst], eax); break;
  }
  add(dst, 4); add(src, 4);
  sub(n, 1); jnz(".lp");
```

実行時コード生成の特長を利用した例として，たとえば正規表現の JIT エンジン Regen[3]，JavaScript 実行エンジン iv[4]，プレイステーション 2 エミュレータ PCSX2[5] の画像処理部分などで Xbyak が使われている．

4. 文字列検索の高速化

Intel の Penryn と呼ばれる Core2 以降の CPU には C の標準関数である strlen や strstr などを高速に処理するための命令群（pcmpestri, pcmpistri など: 以下文字列処理命令と呼ぶ）が追加されている．これらの命令は極めて複雑なパラメータを持ち使い方は難しいがその効果は目を見張るものがある．ここではそれらの一部を紹介する．

4.1 文字列処理命令

文字列は C においては '\0' 終端による char の配列（以下 C 文字列と呼ぶ）が一般的である．しかし，途中に '\0' を含んでよい char の配列とその長さの組（以下 L 文字列）を扱いたいときも多い．C++の std::string は L 文字列を扱える．SSE4.1 の文字列処理命令は C 文字列と L 文字列の両方を扱える．そして処理の結果を ecx で得るか，xmm0 で得るかかによって異なり 4 通りのパターンがある（表 1）．

表 1 文字列処理命令の種類

出力 ＼ 入力	C 文字列	L 文字列
ecx	pcmpistri	pcmpestri
xmm0	pcmpistrm	pcmpestrm

まとめると文字列処理命令は pcmpXstrY の形をしていて

```
pcmpXstrY xmm, xmm/mem, imm8
```

という使い方をする．SSE 系の命令でありながらメモリを指す mem は 16byte アライメントされていなくてもよい．imm8 は 8bit の定数を表し，文字列の単位が 8bit/16bit，符号つき/符号無し，完全マッチ/文字の範囲指定/部分文字列など様々な設定が可能である．詳細は文献 (2) を参照されたい．

4.2 strstr の実装

ここでは strstr の実装を紹介する．strstr は入力テキスト C 文字列（text）から検索 C 文字列（key）がマッチするところを返す関数である．見つからなければ NULL を返す．

コード 7. strstr メイン部

```
1.  movdqu(xmm0, ptr [key]);
2. L(".lp");
3.  pcmpistri(xmm0, ptr [text], 12);
4.  lea(text, ptr [text + 16]);
5.  ja(".lp");
6.  jnc(".notFound");
7. //残りの文字列のマッチングを行う
```

コード 7 は strstr のメインループである．text は入力テキストのポインタ，key は検索文字のポインタが格納されたレジスタである．xmm0 レジスタに key の先頭 16byte を読み込む（1 行目）．text を 16byte ずつとりだし，その中に key が含まれるかを判定する（3 行目）．12 という定数は符号無し C 文字列として検索するためのフラグである．text ポインタを 16byte

増やし（4 行目），何もなければ繰り返す（5 行目）．そうではなく，text が ‘\0’ を含めば終了となる（6 行目）．key が 16byte をまたがって存在する場合は key の先頭がマッチしたという情報がえられる（7 行目以降）．文字列検索命令は検索結果をフラグレジスタの組み合わせて表すため，分岐の方法を正しく行わないと間違えたり遅くなったりすることがある．

pcmpistri の代わりに pcmpestri を使い，edx/eax に text と key の文字列の長さを保持するように修正すると, C 文字列を検索する strstr ではなく，L 文字列を検索する GNU 拡張の memmem 関数の実装ができる．

なお，文字列処理命令は途中に ‘\0’ を含んでいても必ず 16byte 読み込む．そのため 16byte がページ境界をまたがり，かつ次のページ境界が読み込み権限を与えられていない場合，アクセス違反が発生することがある．適当な境界処理を追加するか，そのような文字列を扱わないような工夫が必要である．

4.3 strstr のベンチマーク

前節の strstr のベンチマークを行った．環境は ubuntu 12.04 server(64bit) + Xeon X5650 + gcc 4.6.3 である．比較対象は gcc 4.6.3 の strstr（SSE4.1 使用），boost 1.51 の algorithm::boyer_moore（BM 法），quick search アルゴリズム（改良版 BM 法），strstrC（コード 8）である．

コード 8. strstrC

```
const char *strstr_C(const char *str,
  const char *key) {
  size_t len = strlen(key);
  while (*str) {
    const char *p = strchr(str, key[0]);
    if (p == 0) return 0;
    if (memcmp(p + 1, key + 1, len - 1)
      == 0) return p;
    str = p + 1;
  }
  return 0;
}
```

約 130MiB の UTF-8 エンコードされた日本語テキストから指定された key がいくつあるかを探し，次の文字列までの検索に byte あたりかかった CPU クロック数で比較する．この値が小さいほど速い．実行結果を表 2 に示す．ソースコードは `https://github.com/herumi/mie/tree/master/test/string` から取得できる． std::find の find メソッドは template で実装されたナイーブもので，今回実装した asm 版と比べると 10 倍近く遅いことが分かる．

表 2　文字列検索処理の実行時間 [cycle]

検索文字	find[1]	strstr[2]	strstrC	qs[3]	bm[4]	asm[5]
a	3.37	1.64	0.73	4.7	6.76	0.6
ab	3.04	1.13	0.64	3.12	6.27	0.3
1234	3.23	1.12	0.93	1.99	3.23	0.29
これは	7.39	1.15	7.81	3.4	1.74	0.8
00...0[6]	3.27	1.18	1.05	0.7	0.93	0.3
AB...Z[7]	2.8	0.46	0.43	0.54	0.56	0.27

[1] std::string の find メソッド
[2] gcc 4.6.3 strstr（SSE4.1 使用）
[3] Quick Search アルゴリズム（BM 法の改良版）
[4] boost 1.51 の algorithm::boyer_moore
[5] 今回実装したもの
[6] 0 が 11 個並んだ文字列
[7] A から Z までの長さ 27 の文字列

面白いことに strstr とコード 8 の strstrC を比較すると，key が ASCII 文字列のとき strstrC の方が速い．gcc の strstr は内部で SSE4.1 を使ってはいるが，あまり速い実装ではない．gcc において strstrC が速いのは gcc の strchr が SSE4.1 を適切に用いて高速に動作するためと思われる．

VC や Intel C コンパイラの strstr は gcc よりも高速な実装がなされている．ただ std::find は Visual Studio 2012 においても速くはない．

なお，テキストが UTF-8 エンコードされた日本語であるため 0xe3 を多く含む．key が ‘これは’ のときに strstrC が遅いのは先頭 byte の 0xe3 にマッチする箇所が増え，memcmp の呼び出し回数が増えるためと推測される．

Quick Search アルゴリズムは BM 法よりよいことが分かる．ただしこれらのアルゴリズムが最も有利と思われる ‘AB...Z’ という文字列に対してすら，asm 版の方が約 2 倍速かった．これは qs や bm が一文字読んで，テーブル引きしてオフセットを追加しジャンプするというアルゴリズムなので，CPU のパイプラインに乗りにくいためと思われる．

asm 版は文字列の長さや種類を問わず 1byte あたりの検索に 1cycle 未満の時間しかかかっていないことに注意する．また検索命令のフラグ imm8 を変更することで，一文字にマッチする関数 strchr だけでなく，[a-z][0-9] といった範囲指定のマッチングにも対応でき，実行速度も一文字マッチングと同じである（findChar_range[6] 参照）．

4.4 strcasestr の実装

ここでは ASCII の大文字小文字を無視してマッチングを行う strcasestr の実装を行う．文字列検索命令は直接は使えない．まず予め key 文字列を小文字に変換しておく．そして text の文字を随時小文字に変換しながら処理すればよい．コード 7 の 3 行目において pcmpistri を呼ぶ前に text から 16byte データをとりだして小文字に変換する処理を追加する．

与えられた文字 c が大文字なら小文字にする処理は `if ('A' <= c && c <= 'Z') { c += 'a'-'A'; }` と記述できる．1byte ずつ変換していては効率が悪いためまとめて処理を行う．そのためこの処理を if を使わないコードに置き換える．まず二つの byte 値を比較して大きければ 0xff を，そうでなければ 0 を返す関数を用意する．

```
int gt(x, y) { return x > y ? 0xff : 0; }
```

大文字から小文字への変換はコード 9 となる．

コード 9. 大文字から小文字への変換

```
c += gt(c,'A'-1) & gt('Z'+1,c) & ('a'-'Z');
```

16byte データの各 byte ごとに gt(x, y) を行う命令は pcmpgtb である．この命令を用いて 16byte ずつ大文字を小文字に変換する．strcasestr は strstr の 70%程度の速度で動作した．

4.5 CPU 判別によるコード選択

strstr のメイン部分であるコード 7 は実は SandyBridge 向けのものであり，Xeon X5650 では遅かった．逆に，コード 7 の 4 行目において lea の代わりに add を使うと Xeon X5650 では約 10%速くなった．このようなケースでは CPU に応じたコードを用意するのが望ましい．bool 値変数 isSandyBridge を CPU が SandyBridge のとき true となる変数とすると，コード 10 のように記述できる．

コード 10. 命令ディスパッチ

```
 if (isSandyBridge) {
  lea(a, ptr [a + 16]);
  ja(".lp");
 } else {
  jbe(".headCmp");
  add(a, 16);
  jmp(".lp");
L(".headCmp");
 }
```

isSandyBridge はコード生成の前に一度だけ CPU 判別を行い値を設定すればよい．

このように CPU の種類によって特定の部分だけコードを変えたい場合も，実行時コード生成を行う Xbyak では自然に記述できる．

5. ビットベクトルの中の 1 を数える

簡潔データ構造では巨大なビット列のある範囲にある 1 の数を数える関数 rank をよく使う．長さ $n(< 2^{32})$ のビット列を v, ビット列の i 番目 $(0 \leq i < n)$ を v[i](=0 or 1) とかくことにする．rank を `rank(k) := # { i | 0 <= i <= k, v[i] = 1 }` として定義する．rank を計算するために，32bit 整数配列 b1 と 8bit 整数配列 b2 の二つのテーブルを用意する．b1 は 256bit 毎の 1 の累積数で初期化し，b2 は 64bit 毎の累積，ただし 256bit 毎に 0 初期化する．後者は 256bit 未満の 1 の数となり 8bit 整数変数に格納できメモリサイズを減らせる．

コード 11. rank1 の実装

```
for (int i = 0; i < n / 256; i++) {
 b1[i] = rank(i * 256);
}
for (int i = 0; i < n / 64; i++) {
 b2[i] = rank(i*64) - rank((i&~3)*64);
}
uint32_t rank(uint32_t i) const {
 uint64_t mask = (2ULL<<(i & 63))-1;
 return a_[i / 256] + b_[i / 64]
  + popcnt(org_[i / 64] & mask);
}
```

popcnt は SSE4.2 から搭載されている 64bit 整数の中の 1 の数を数える命令である．この実装の場合，もとのビットベクトル 1bit あたりに必要なメモリは $(32 + 8 * 4)/256 = 1/4$ である．

元のデータ，b1，b2 をインタリーブに混ぜることでメモリアクセスの局所性が向上し高速化できる[7]．今回はその文献を参考にしつつ，データの間にパディングを入れない実装をした（rank1）．

次に 1bit あたりに必要なメモリ量の低減を考える．256bit 単位ではなく 512bit 単位で b1 を作成する．すると 1 の数が最大 511 となり 8bit 整数変数の b2 に入らない．そのため b2 には累積ではなく部分和をそのまま保持する必要がある．64bit 毎の部分和では 8bit 変数が 8 個必要になり必要なメモリ量が 256bit のと

きと変わらない．そのため 128bit 毎の部分和が必要になる．

コード 12. rank2 のテーブル初期化

```
for (int i = 0; i < n / 512; i++) {
  b1[i] = rank(i * 512);
}
b2[0] = 0;
for (int i = 1; i < n / 128; i++) {
  b2[i] = rank(i*128)-rank((i-1)*128);
}
```

この場合 rank の計算の中で $0 \leq n < 4$ 個の uint8_t の値の和（コード 13）が必要になる．以下ではこれを高速に求める方法を与える．

コード 13. n($0 \leq n < 4$) 個の和

```
int sum1(uint8_t data[4], int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += data[i];
  }
  return sum;
}
```

Xeon X5650 + gcc 4.6.3 の環境において XorShift による乱数生成の時間を込みで約 35cycle かかった．これをループ展開することで 26cycle にまで減少した（コード 14）．

コード 14. switch 文による n 個の和

```
int sum2(uint8_t data[4], int n) {
  int sum = 0;
  switch (n) {
    case 3: sum += data[2];
    case 2: sum += data[1];
    case 1: sum += data[0];
  }
  return sum;
}
```

n がランダムな場合，switch 文において分岐予測がヒットする確率は 25%である．分岐予測が外れたときのペナルティは経験的に約 20cycle と推測されるため，平均では約 15cycle のペナルティを受けていると予想される．そこで分岐予測が不要な実装を考える．

x86/x64 には MPEG などのコーデック向けに用意された psadbw という SSE 命令がある．これは与えられた二つの 8 個の byte データの差の絶対値の和を求める命令である．

`pasdbw(x, y) := sum[abs(x[i]-y[i])|i=0..7]`

この命令を使うと条件分岐をなくすことができ，コード 15 は乱数生成時間を含めて 10cycle で実行できる．

コード 15. psadbw による n 個の和

```
union {
  uint8_t data[4];
  uint32_t s;
} ci;
x = ci.s & ((1U << (n * 8)) - 1);
movd(xmm0, x); // xmm0 = x
pxor(xmm1, xmm1); // zero clear
psadbw(xmm0, xmm1); // sum
```

最後に 128bit の popcnt の実装について述べる．与えられた 128bit データ `[H:L]`（`H`, `L` はそれぞれ 64bit）に対して長さ n（$0 \leq n < 128$）のマスクをとって上位，下位の popcnt を加算する．mask を生成する場合，n が 64 以下か否かで場合分けが必要になる．

コード 16. 128bit のマスク

```
// input [L:H]
uint64_t mask = 2 << (n & 63) - 1;
uint64_t mL = (n & 64) ? -1 : mask;
uint64_t mH = (n & 64) ? mask : 0;
L &= mL;
H &= mH;
```

コード 16 に対して gcc は条件分岐を使ったコードを生成した．前述のように条件分岐命令は分岐予測が外れるとペナルティが発生する．そこでコード 17 のように条件分岐命令を使って回避した．

コード 17. cmovz を使った 128bit のマスク

```
or(mL, -1);
and(n, 64);
cmovz(mL, mask);
cmovz(mask, n);
```

cmovz はゼロフラグが 1 のときに mov を実行する命令である．ゼロフラグが 1 になるのは `and(n, 64)` の結果が 0，つまり $n < 64$ のときである．cmovz 命令はフラグを変更しないため，一度の判定だけで二つの cmovz 命令を実行できる．

これらのコード片を組み合わせて作った関数を rank2

とし，ベンチマークを行った．与えられたビットベクトルに対して，ランダムな i についての rank(i) を 100 万回呼び出したときの 1 回あたりの平均処理 cycle（ループや乱数生成時間を含む）を求めた．テーブルを構成する時間は含めていない．比較対象として sdsl[8] の 2012 年 9 月 5 日づけのバージョンを用いた．ベンチマーク環境は 4 章と同じである． rank2 はテーブル追加サイズが rank1 や sdsl の半分になった分，演算コストが増えたためやや遅い．しかし，ビットベクトルのサイズが大きいところでは sdsl より速い．これはメモリアクセスの減少が演算コストのペナルティをカバーしているためと思われる．なお，このベンチマークは環境や乱数のとり方などに強くする．また sdsl は設定オプションなど機能を細かく変更できるため，この結果はあくまで一つの目安であることに注意されたい．

表 3　rank 関数の実行時間 [cycle]

ベクトルサイズ	rank1	rank2	sdsl
0.06M	6.43	18.33	11.34
0.25M	7.26	18.49	11.96
1M	8.11	20.58	13.18
4M	13.48	25.50	17.83
16M	16.51	29.44	20.07
64M	32.76	49.89	37.93
256M	72.24	106.27	103.00
1024M	83.07	125.30	131.02
4096M	93.80	138.30	150.53

6. 指数関数 exp の近似計算

この章では e^x の近似値を高速に求める手法について述べる．float と double の両方について実装した[10]．ここでは double について紹介する．演算精度は誤差が最大 1e-16 程度を目標に設計した．この章はシンポジウムでは紹介していない．

コード 18 は double に対する指数関数 expd の疑似コードである．コード中の定数 A, B, C2 などについては後述する．

6.1 アルゴリズムの方針

指数関数の和が積になる性質とテイラー展開を元に考える．

- $\exp(x+y) = \exp(x)\exp(y)$
- $\exp(x) = 1 + x + x^2/2 + x^3/6 + x^4/24 + \cdots$

まず入力値 x をテーブル引きするための値 s とテイラー展開に使うための小さな値 t に分ける．

$$x = s + t, \quad \exp(x) = \exp(s)\exp(t).$$

t が $|t| < 2^{-12}$ ならば $t^4/24 < 2^{-52}$．double の精度は 53bit なのでテイラー展開は 3 次の項までとする．

round(x) を x を整数に丸める関数とする．

$$x = \mathrm{round}(x) + (x - \mathrm{round}(x)).$$

$s = \mathrm{round}(x)$，$t = x - s$ とすると $|t| \le 1/2$．$|t| < 2^{-12}$ とするために $x' = 2048x$ を考え $x' = s' + t'$ と分解すると，

$$x = \mathrm{round}(2048x)/2048 + t'/2048.$$

そうすると $n = \mathrm{round}(2048x)$ を整数として，$\exp(n/2048)$ の値をテーブルで保持すればよいということになる．ここでテーブルサイズを考える．$\exp(x) = 0$ となる x は $\log(\texttt{DBL\_MIN}) = -708$，$\exp(x) = \texttt{Inf}$ となる x は $\log(\texttt{DBL\_MAX}) = 709$．

したがって n の範囲はおおよそ -708×2048 から 709×2048 の間の整数となる．このときテーブルサイズは $(708 + 709) \times 2048 \times 8 = 22$MiB となる．これは巨大であるためこの方法は使えない．

テーブルサイズを減らすために別の方法を考える．2048 倍していたが，必ずしもこの値は整数である必

コード 18. expd の実装

```
union di {
  uint64_t i;
  double d;
};
double expd(double x) {
  const double A = 2048/log(2);
  const double B = 3ULL << 51;
  const double RA = 1 / A;
  const C2 = 0.16666666685227835;
  const C3 = 3.0000000027955394;
  di di;
  di.d = x * A + B;
  uint64_t iax = tbl[di.i & 2047];
  double t = (di.d - B) * RA - x;
  uint64_t u = ((di.i + 2095104) >> 11)
               << 52;
  double y = (C3 - t) * (t * t) * C2
              - t + 1;
  di.i = u | iax;
  return y * di.d;
}
```

要性はない．コンピュータは 2 の巾乗の計算を簡単にできるため，2048 に近い値を用いて基底の変換を試みる．すなわち

$$\exp(n/\alpha) = 2^{n/2^\beta}$$

となるように α，β を選ぶ．ここで $\alpha = 2048/(\log 2)$ とすると $\beta = 12$．この α がコード 18 の A である．n を 2048 で割った商と余りを q，r とする ($n = 2048q + r$)．すると

$$\exp(n/\alpha) = 2^{n/2048} = 2^q 2^{r/2048}.$$

このように変形すると $0 \le r < 2048$ に対する $2^{r/2048}$ の値のみをテーブルに保持すればよい．テーブルサイズは $2048 \times 8 = 16$KiB である．

次に round 処理について考察する．SSE4.1 で搭載された roundpd を使う方法もあるが double 型変数 x に 2^{52} を足すと仮数部が丸められて整数になるトリックを使う（実測したところこちらのほうがやや速かった）．ただし，x は負の場合もあるので $2^{52} + 2^{51}$ を足す（この値がコード 18 の B である）．

最後にテイラー展開の係数の C2, C3 について説明する．

$1 + x + x^2/2 + x^3/6 = 1 + x + (1/6)x^2(3 + x)$ において 1/6 の代わりに $C2 = 0.16666666685227835$ を，3 の代わりに $C3 = 3.0000000027955394$ を利用すると誤差の平均が 1〜2%減少した．この値は $\alpha = \log(2)/4096$ として

$$I := \int_0^\alpha \left(\exp(x) - (a + x + cx^2 + dx^3)\right)^2 dx$$

が最小になるように (a, c, d) を求めた．

6.2 ベンチマーク

float, double に対する std::exp と今回実装したもの fmath::exp, fmath::expd を比較した[10]．環境は今までと同様に Xeon X5650 + gcc 4.6.3 である．標準関数に比べて 4〜10 倍の高速化となっている．

表 4 exp の実行時間 [cycle]

関数	cycle	誤差
std::exp(float)	167.9	-
fmath::exp	13.9	7.4e-8
std::exp(double)	70.14	-
fmath::expd	17.39	1.11e-16

7. ま と め

C++で記述できる x86/x64 JIT アセンブラを紹介した．よく「C は高級アセンブラ」という喩えが使われるが，Xbyak は C++「を」高級アセンブラとして使うためのものである．

もちろん世の中は LLVM などの抽象化された機構が主流であり，Xbyak はそういったものに機能面で太刀打ちできない．しかし，今回紹介した文字列命令は当分は x86/x64 CPU 以外には搭載されないだろう．そういった命令を扱いたい場合は抽象レイヤを挟むのはまどろっこしい．Xbyak を使うと読みやすく，CPU に応じた自由度の高い最適化を行える．著者は glibc の strstr の実装よりも Xbyak で記述されたものを美しいと感じる．

また，strstr や exp などの古くからある標準関数ですら必ずしも十分に最適化されているわけではないということがわかった．

最新 CPU の機能をどう使えばより高速な処理ができるか，パズルのような遊びを楽しみたい．

8. 質 疑 応 答

頂いた質問と発表時に応答したものに多少加筆した．

- Q.（東京大学情報基盤センター 田中さん）是非使ってみたいが，使い方に関して確認したい．関数単位でしかコード生成できなくて，その中のコードはすべてアセンブリ言語で書く必要があるのか?
- A. はい．正確には関数でないものも生成できる．
- Q.（同上）レジスタ割当は最近は人間がおこなうよりコンパイラに任せた方が良さそうだが，明示的に行わなくてはいけないことで不自由は感じないのか?
- A. 感じない．暗号処理では人間が割り当てる方がよいケースも多い．
- Q.（同上）簡潔データ構造を扱うコードとしては，岡之原さんのものよりも SDSL 等の方が速度のことを考えられているのではないか?
- A. 岡之原さんのものは紹介だけで，ベンチマークの対象ではない．発表後のこの論文では SDSL と比較を行った．
- Q.（ソニー・コンピュータエンタテインメント 藤波さん）Xbyak では，C++の関数単位でコード生成を行いますが，もっと複雑な処理をしたい場合に，例えば通常の呼び出し規約を無視して高速化を行うようなことをしたくなると思いますが，そのような使い方はサポートされていますか.
- A. はい．一つ目の質問と被るが Xbyak 自体は呼び出し規約を知らない．コード生成を正しく行うのはユーザの責任である．したがって，通常の呼

び出し規約を無視した独自の呼び出し規約を使った関数を作ることもできる．文献 (11) で実装した暗号処理では独自呼び出し規約を設定することで 10%ほどの高速化を実現した．

参 考 文 献

1) Andy Oram, Greg Wilson 編 Brian Kernighan, Jon Bentley, まつもとゆきひろ他著久野禎子 久野靖訳：ビューティフルコード，オライリージャパン (2008).
2) Intel : http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
3) 新屋良磨 光成滋生 佐々政孝:並列化と実行時コード生成を用いた正規表現マッチングの高速化，第 53 回プログラミングシンポジウム予稿集 (2012).
4) Constellation : https://github.com/Constellation/iv
5) http://pcsx2.net/.
6) https://github.com/herumi/mie/blob/master/include/mie/string.hpp
7) Takeshi Yamamuro : A x86-optimized rank & select dictionary for bit sequences http://www.slideshare.net/maropu0804/a-x86optimized-rankselect-dictionary-for-bit-sequences
8) https://github.com/simongog/sdsl
9) 光成滋生：高速な倍精度指数関数 exp の実装 http://www.slideshare.net/herumi/exp-9499790
10) https://github.com/herumi/fmath/
11) B. Jean-Luc, G. Jorge E, M. Shigeo, O. Eiji, R. Francisco, T. Tadanori : High-speed software implementation of the optimal ate pairing over Barreto-Naehrig curves, pp. 21–39, Pairing'10(2010). http://homepage1.nifty.com/herumi/crypt/ate-pairing.html