Exocompilationによる Kernel-GHASHアルゴリズム アクセラレーション

斯瑞程

The Exo Language

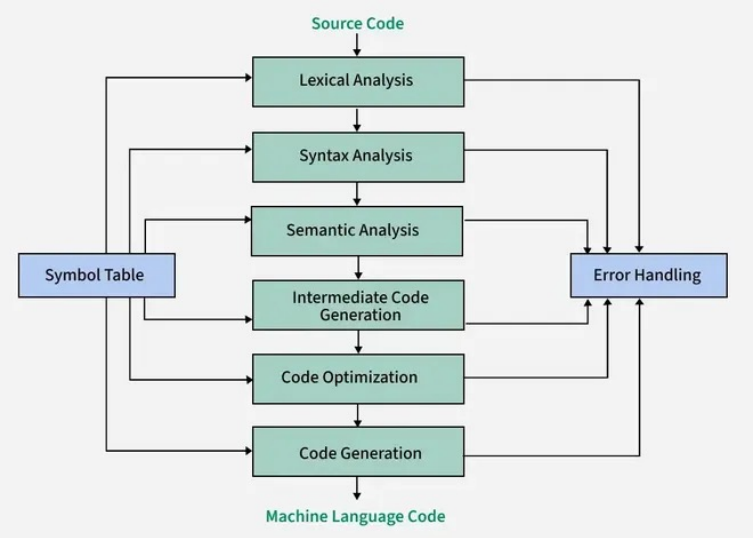
Exo言語とは？

- モチベーション：Hardware accelerationプログラミングを簡単化する
- 「Exocompilation」による明示的な手動最適化
- 正確性はExoコンパイラ側が保障する

Exocompilation for productive programming of hardware accelerators, Ikarashi et al., PLDI 2022
Exo 2: Growing a scheduling language, Ikarashi et al., ASPLOS 2025

Exocompilation

- 従来の手法



最適化はすべてコンパイラに任せる

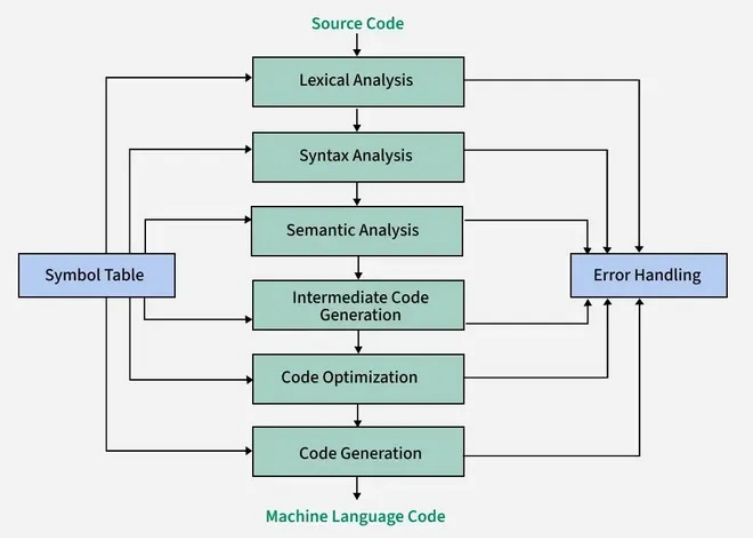
- Exo

```
# Reshape C_reg so we can map it into vector registers
avx = divide_dim(avx, "C_reg:_", 1, 8)
avx = repeat(divide_loop)(avx, "for i1 in _: _", 8, ["i2", "i3"], perfect=True)
avx = simplify(avx)

# Map C_reg operations to vector instructions
avx = set_memory(avx, "C_reg:_", AVX2)
avx = replace_all(avx, mm256_loadu_ps)
avx = replace_all(avx, mm256_storeu_ps)
avx = simplify(avx)
```

明示的に最適化可能

Exocompilation

- 従来の手法



最適化はすべてコンパイラに任せる

- Exo

```
# Reshape C_reg so we can map it into vector registers
avx = divide_dim(avx, "C_reg:_", 1, 8)
avx = repeat(divide_loop)(avx, "for i1 in _: _", 8, ["i2", "i3"], perfect=True)
avx = simplify(avx)

# Map C_reg operations to vector instructions
avx = set_memory(avx, "C_reg:_", AVX2)
avx = replace_all(avx, mm256_loadu_ps)
avx = replace_all(avx, mm256_storeu_ps)
avx = simplify(avx)
```

明示的に最適化可能

Exocompilation

- 任意のプラットフォームに向けてユーザレベルで命令を定義可能

```
@instr("{dst_data} = _mm256_cvtps_pd(_mm256_extractf128_ps({src_data}, 0));")
def avx2_convert_f32_lower_to_f64(dst: [f64][4] @ AVX2, src: [f32][8] @ AVX2):
    assert stride(dst, 0) == 1
    assert stride(src, 0) == 1

    for i in seq(0, 4):
        dst[i] = src[i]
```

Exocompilation

- 任意のプラットフォームに向けてユーザレベルで命令を定義可能

```
@instr("{dst_data} = _mm256_cvtps_pd(_mm256_extractf128_ps({src_data}, 0));")
def avx2_convert_f32_lower_to_f64(dst: [f64][4] @ AVX2, src: [f32][8] @ AVX2):
    assert stride(dst, 0) == 1
    assert stride(src, 0) == 1

    for i in seq(0, 4):
        dst[i] = src[i]
```

Exocompilation

- 正確性をSMT（SAT Solver）によって保障される
- しかしセマンティックをコンパイラに理解させる必要がある

Loop reordering. One of the most basic non-trivial loop transformations is loop-reordering. When can we rewrite `for` x do `for` y do s into `for` y do `for` x do s? This transformation is valid when the loop bounds commute with the body, and when any loop iterations that are moved past each other commute. To formulate these conditions, let a_x be the effect of the x-loop's bound-expressions and a_y similarly for the y-loop. Let x', y' be copies of these iteration variables s.t. $s' = [x \mapsto x'][y \mapsto y']s$. Let $a = \mathit{Eff}\ [\![s]\!]$ and $a' = \mathit{Eff}\ [\![s']\!]$. Then the reordering condition may be precisely stated as

$$\begin{array}{l}(\forall x, y.\ M\mathrm{Bd}(x, y) \implies \mathrm{Commutes}((a_x\,;a_y), a)) \\ \wedge \left(\begin{array}{r}\forall x, y, x', y'.\ M(\mathrm{Bd}(x, y, x', y') \wedge x < x' \wedge y < y) \\ \implies \mathrm{Commutes}(a, a')\end{array}\right)\end{array}$$

Loop fusion & fission. Another basic loop transformation is to fuse two loops together, or in reverse to fission one loop in two. When can we rewrite `for` x do $s_1\,;s_2$ into (`for` x do s_1)$\,;$`for` x do s_2? This is possible when the loop bound commutes with the first statement, and when the statements that get reordered commute with each other. Letting a_x be the effect of the loop bounds, $a_1 = \mathit{Eff}\ [\![s_1]\!]$, $s_2' = [x \mapsto x']s_2$ and $a_2' = \mathit{Eff}\ [\![s_2']\!]$ we can state fission conditions precisely as

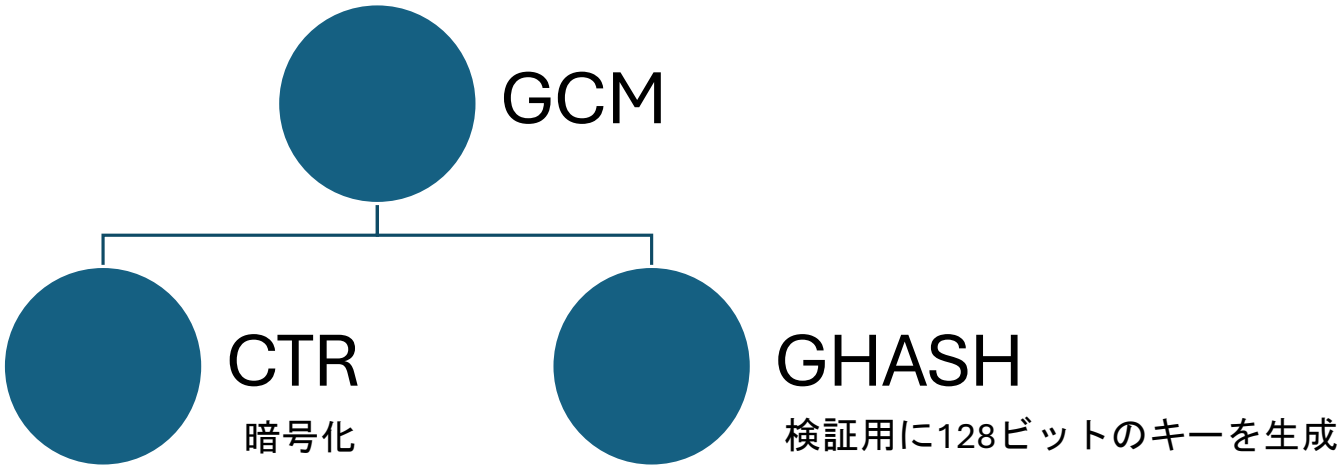
$$\begin{array}{l}(\forall x.\ M\mathrm{Bd}(x) \implies \mathrm{Commutes}(a_x, a_1)) \wedge \\ (\forall x, x'.\ M(\mathrm{Bd}(x, x') \wedge x' < x) \implies \mathrm{Commutes}(a_1, a_2'))\end{array}$$

Loop removal. In order for the rewrite `for` x do $s \rightsquigarrow s$ to be safe, the variable x must not be free in s, s must be idempotent, and the loop must run for at least one iteration. If $a = \mathit{Eff}\ [\![s]\!]$, then these conditions are precisely

$$(\exists x.\ D\mathrm{Bd}(x)) \wedge \mathrm{Shadows}(a, a)$$

Exocompilation for productive programming of hardware accelerators, Ikarashi et al., PLDI 2022

Kernel-GHASH



- ライブラリ関数としてTLS/HTTPS・IPsecなどのモジュールに利用される
- ソフトウェア実装：lib/crypto/gf128hash.c

Kernel-GHASH

- パラメータ
- acc：Accumulation
- key：検証用キー
  - 暗号用キーの対偶として理解して良い
- data：検証対象

$$acc_i = (acc_{i-1} \oplus data_i) \cdot key$$

```
static void __maybe_unused ghash_blocks_generic(struct polyval_elem *acc,
                                                const struct polyval_elem *key,
                                                const u8 *data, size_t nblocks)
{
    do {
        acc->lo ^=
            cpu_to_le64(get_unaligned_be64((__be64 *)(data + 8)));
        acc->hi ^= cpu_to_le64(get_unaligned_be64((__be64 *)data));
        polyval_mul_generic(acc, key);
        data += GHASH_BLOCK_SIZE;
    } while (--nblocks);
}
```

Kernel-GHASH

$acc_i = (acc_{i-1} \oplus data_i) \cdot key.$

- 実質GF128という有限体（Galois Field）上の演算

$acc_i = (acc_{i-1} + data_i) \cdot key. \quad (GF128)$

- 有限体上の乗算：

1. まず多項式乗算を行う
2. そして簡約不可能な多項式に対して余数をとることによって結果を有限体に抑える

   mod x^128 + x^127 + x^126 + x^121 + 1

Kernel-GHASH

mod x^128 + x^127 + x^126 + x^121 + 1

- 特殊な専用多項式
- 簡約操作はわずかなshiftとxor操作でできる

```
/*
 * Cancel out the low 128 bits of the product by adding multiples of
 * G(x) = x^128 + x^127 + x^126 + x^121 + 1.  Do this in two steps, each
 * of which cancels out 64 bits.  Note that we break G(x) into three
 * parts: 1, x^64 * (x^63 + x^62 + x^57), and x^128 * 1.
 */

/*
 * First, add G(x) times c0 as follows:
 *
 * (c0, c1, c2) = (0,
 *                 c1 + (c0 * (x^63 + x^62 + x^57) mod x^64),
 *         c2 + c0 + floor((c0 * (x^63 + x^62 + x^57)) / x^64))
 */
c1 ^= (c0 << 63) ^ (c0 << 62) ^ (c0 << 57);
c2 ^= c0 ^ (c0 >> 1) ^ (c0 >> 2) ^ (c0 >> 7);

/*
 * Second, add G(x) times the new c1:
 *
 * (c1, c2, c3) = (0,
 *                 c2 + (c1 * (x^63 + x^62 + x^57) mod x^64),
 *         c3 + c1 + floor((c1 * (x^63 + x^62 + x^57)) / x^64))
 */
c2 ^= (c1 << 63) ^ (c1 << 62) ^ (c1 << 57);
c3 ^= c1 ^ (c1 >> 1) ^ (c1 >> 2) ^ (c1 >> 7);

/* Return (c2, c3).  This implicitly multiplies by x^-128. */
a->lo = cpu_to_le64(c2);
a->hi = cpu_to_le64(c3);
```

Kernel-GHASH

- GF(2)上の多項式乗算

- 多項式乗算例
- $(x+1)(x+1) = x^2 + x + x + 1 = x^2 + 1$

- キャリーレス

```c
/* Do a 32 x 32 => 64 bit carryless multiplication. */
static u64 clmul32(u32 a, u32 b)
{
    /*
     * With 32-bit multiplicands and one term every 4 bits, there are up to
     * 32 / 4 = 8 one bits per column when each multiplication is written
     * out as a series of additions in the schoolbook manner.  The value 8
     * fits in 4 bits, so the carries don't overflow into the next term.
     */
    u32 a0 = a & 0x11111111;
    u32 a1 = a & 0x22222222;
    u32 a2 = a & 0x44444444;
    u32 a3 = a & 0x88888888;

    u32 b0 = b & 0x11111111;
    u32 b1 = b & 0x22222222;
    u32 b2 = b & 0x44444444;
    u32 b3 = b & 0x88888888;

    u64 c0 = (a0 * (u64)b0) ^ (a1 * (u64)b3) ^
         (a2 * (u64)b2) ^ (a3 * (u64)b1);
    u64 c1 = (a0 * (u64)b1) ^ (a1 * (u64)b0) ^
         (a2 * (u64)b3) ^ (a3 * (u64)b2);
    u64 c2 = (a0 * (u64)b2) ^ (a1 * (u64)b1) ^
         (a2 * (u64)b0) ^ (a3 * (u64)b3);
    u64 c3 = (a0 * (u64)b3) ^ (a1 * (u64)b2) ^
         (a2 * (u64)b1) ^ (a3 * (u64)b0);

    /* Add all the intermediate products together. */
    return (c0 & 0x1111111111111111) ^
           (c1 & 0x2222222222222222) ^
           (c2 & 0x4444444444444444) ^
           (c3 & 0x8888888888888888);
}

/* Do a 64 x 64 => 128 bit carryless multiplication. */
static void clmul64(u64 a, u64 b, u64 *out_lo, u64 *out_hi)
{
    u32 a_lo = (u32)a;
    u32 a_hi = a >> 32;
    u32 b_lo = (u32)b;
    u32 b_hi = b >> 32;

    /* Karatsuba multiplication */
    u64 lo = clmul32(a_lo, b_lo);
    u64 hi = clmul32(a_hi, b_hi);
    u64 mi = clmul32(a_lo ^ a_hi, b_lo ^ b_hi) ^ lo ^ hi;

    *out_lo = lo ^ (mi << 32);
    *out_hi = hi ^ (mi >> 32);
}
```

Kernel-GHASH

- しかし、普通のALUではキャリー付き乗算器しかない
- →ソフトウェア実装では、キャリーレスの結果を得るために、乗数を分解して計算を行っている
- コメント：The value 8 fits in 4 bits, so the carries don't overflow into the next term.

$$(a_0x^0 + a_4x^4 + a_8x^8 + a_{12}x^{12} + a_{16}x^{16} + a_{20}x^{20} + a_{24}x^{24} + a_{28}x^{28})^2$$

- 係数が高々8
- 最後にmod 2すれば良い

```
u32 a0 = a & 0x11111111;
u32 a1 = a & 0x22222222;
u32 a2 = a & 0x44444444;
u32 a3 = a & 0x88888888;

u32 b0 = b & 0x11111111;
u32 b1 = b & 0x22222222;
u32 b2 = b & 0x44444444;
u32 b3 = b & 0x88888888;
```

```
u64 c0 = (a0 * (u64)b0) ^ (a1 * (u64)b3) ^
     (a2 * (u64)b2) ^ (a3 * (u64)b1);
u64 c1 = (a0 * (u64)b1) ^ (a1 * (u64)b0) ^
     (a2 * (u64)b3) ^ (a3 * (u64)b2);
u64 c2 = (a0 * (u64)b2) ^ (a1 * (u64)b1) ^
     (a2 * (u64)b0) ^ (a3 * (u64)b3);
u64 c3 = (a0 * (u64)b3) ^ (a1 * (u64)b2) ^
     (a2 * (u64)b1) ^ (a3 * (u64)b0);
```

Kernel-GHASH

- PCLMULQDQ命令
- SSE（128ビットレジスタ）で直接64 x 64 → 128のキャリーレス乗算を行う

ExoにおけるPCLMULQDQ実装

- 残念ながらExoにはPCLMULQDQ命令が取り組んでいない
- さらに、PCLMULQDQを実装するためのビット操作演算子などすら存在していない
- 一から作る必要がある

ExoにおけるPCLMULQDQ実装

- Exoを改造する
- フロントエンドとバックエンド（Cのプリンタ）
- 新しくサポートさせた機能：
  - u64
  - and、xor
  - shift

- 足りない部分がある。。

ExoにおけるPCLMULQDQ実装

- 正確性保障

- シフト検査（u64に対して）
  - $0 \leq shift < 64$

- Bound check
  - x[i << 64]の時、配列外アクセスあるか

- SMTを定義した

ExoにおけるPCLMULQDQ実装

- さらに。。

- Exoは「スケジューリング」操作がコードを等価変換する
- 例えばshift操作をforループの式に移動される時、正確性をどう保障するか

- 変換後のコードの正確性を保障するために、以上の検査を再帰的に行う

ExoにおけるPCLMULQDQ実装

- 以上のことによって、Exoは PCLMULQDQを実現する能力を持つようになった
- PCLMULQDQシリーズを実装

- GHASH乗算をアセンブリのように実装
- ユーザレベルの検証

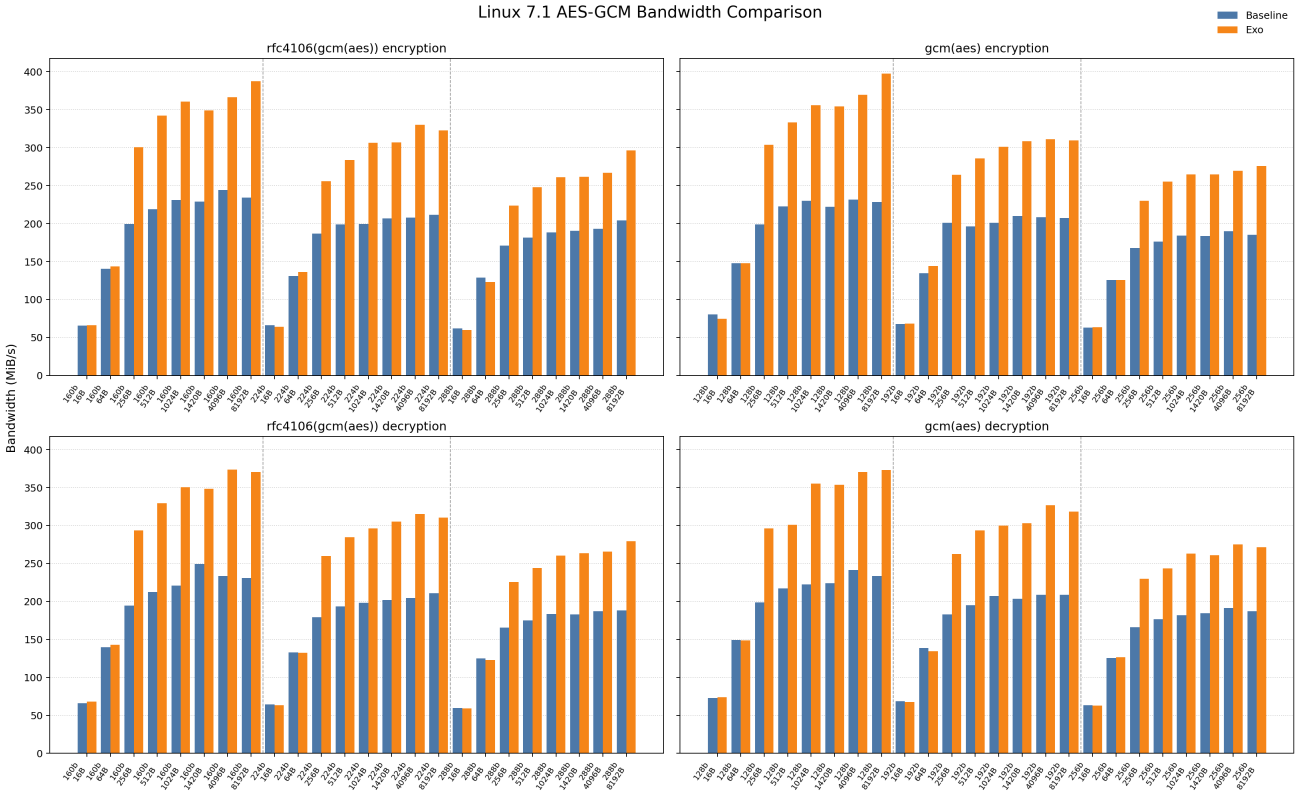
実装	スループット
gf128hash.c	0.73-0.75 GB/s
Exo-pclmulqdq	16.3-17.1GB/s

```
mm_loadu_si128(acc_reg, acc)
mm_loadl_epi64(gfpoly_reg, gfpoly)

for chunk in seq(0, nblocks / 8):
    mm_setzero_si128(lo)
    mm_setzero_si128(mi)
    mm_setzero_si128(hi)

    for j in seq(0, 8):
        mm_loadu_si128(a, powers[j, 0:2])
        mm_loadl_epi64(a_xored, powers_xored[j : j + 1])
        mm_loadu_si128(tmp, data[8 * chunk + j, 0:2])
        mm_bswap128_si128(b, tmp)

        if j == 0:
            mm_xor_si128_inplace(b, acc_reg)

        mm_clmulepi64_si128_00(tmp, a, b)
        mm_xor_si128_inplace(lo, tmp)

        mm_swap64_si128(b_xored, b)
        mm_xor_si128_inplace(b_xored, b)

        mm_clmulepi64_si128_11(tmp, a, b)
        mm_xor_si128_inplace(hi, tmp)

        mm_clmulepi64_si128_00(tmp, a_xored, b_xored)
        mm_xor_si128_inplace(mi, tmp)

    mm_xor_si128_inplace(mi, lo)
    mm_xor_si128_inplace(mi, hi)
    mm_swap64_si128(swapped, lo)
    mm_clmulepi64_si128_00(tmp, gfpoly_reg, lo)
    mm_xor_si128_inplace(mi, swapped)
    mm_xor_si128_inplace(mi, tmp)

    mm_swap64_si128(swapped, mi)
    mm_clmulepi64_si128_00(tmp, gfpoly_reg, mi)
    mm_xor_si128(acc_reg, hi, swapped)
    mm_xor_si128_inplace(acc_reg, tmp)

mm_storeu_si128(acc, acc_reg)
```

Kernelにおける検証

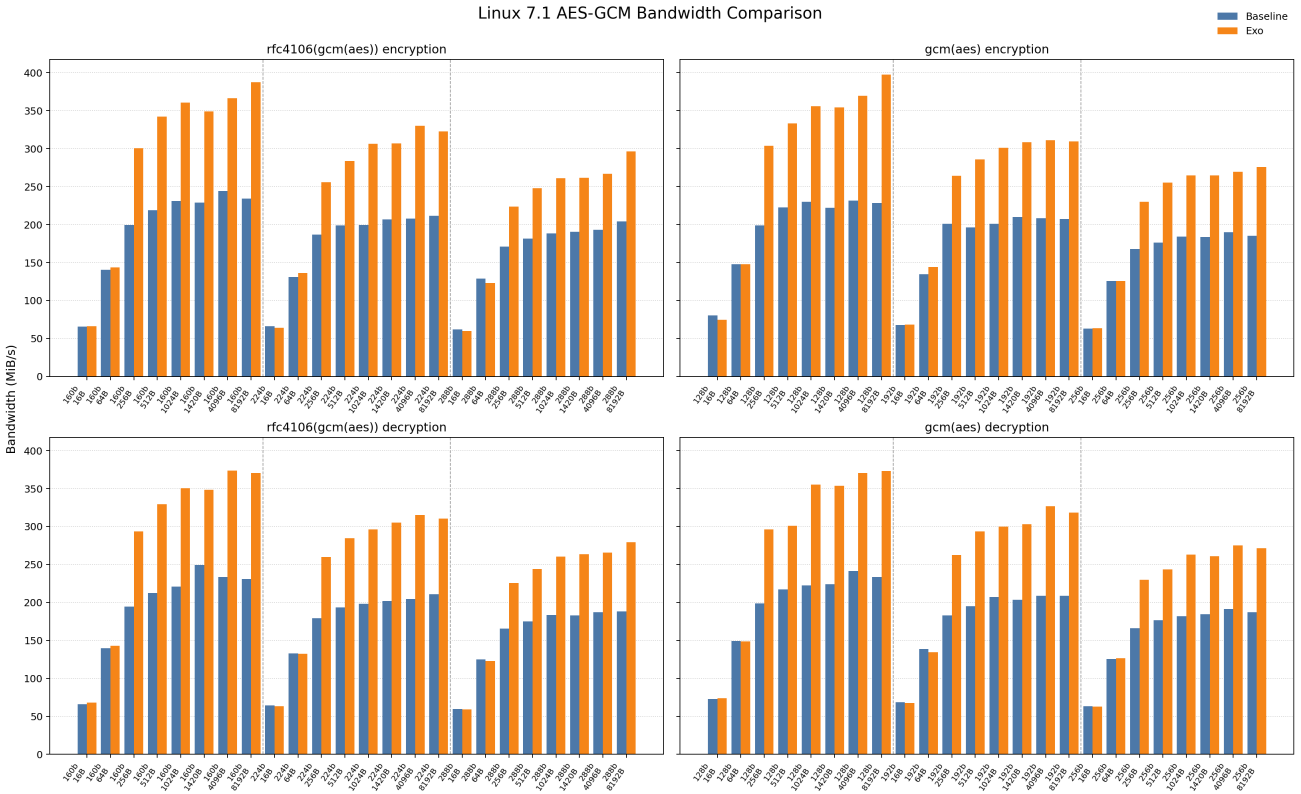
- 検証ツール：tcrypt
- 検証対象：スループット（速度）
- Compiling Configuration（Kconfig）を調整する必要がある
  - CTRとGHASH両方のハードウェアアクセラレーションをオッフ
  - ベンチマークモードをオン
  - Crypto AESをモジュールに変更（ベンチマークのためにユーザ空間に見えるようにする）

Kernelにおける検証



Kernelにおける検証

- ブロック数が8以上の時に有効になるように設定していることが完全に反映されている
- 40%~70%ぐらいの差が出た



第二部

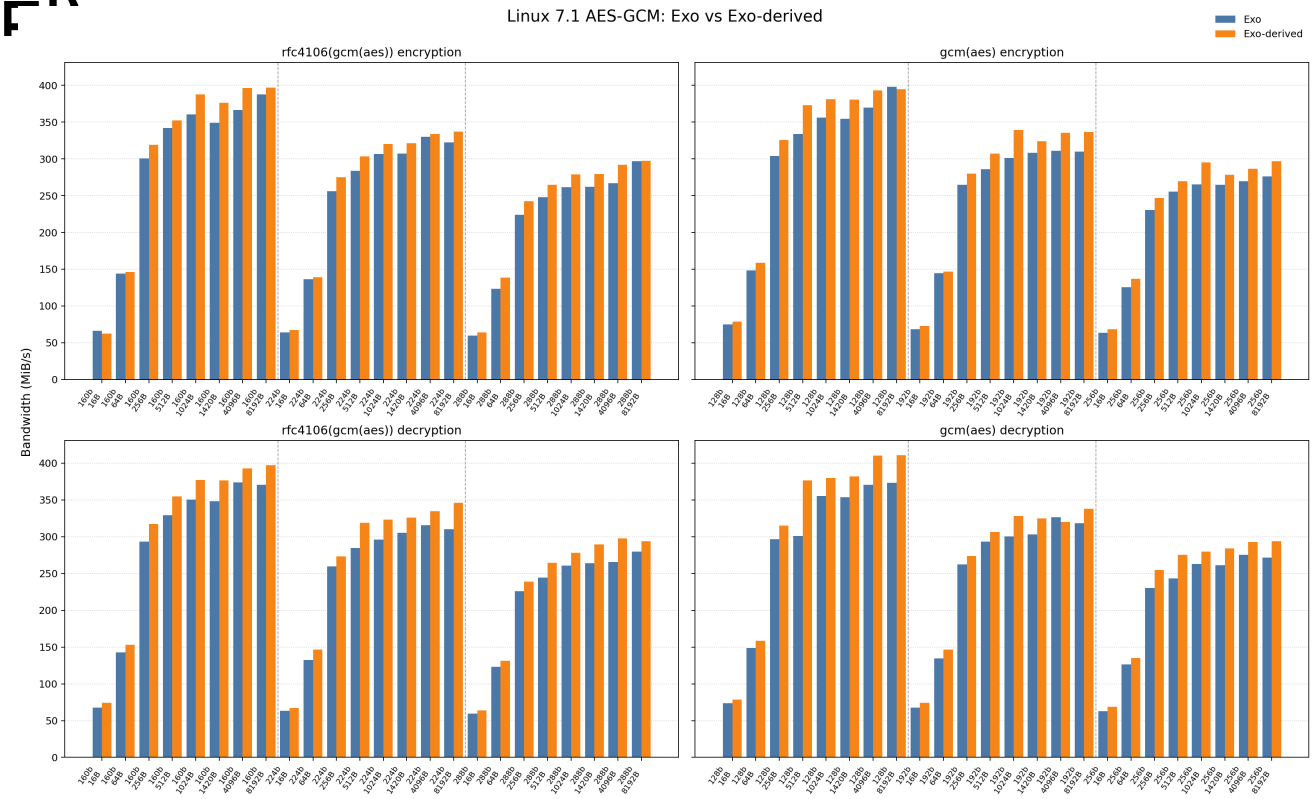
- Exoによる「スケジューリング」操作でソフトウェア実装をハードウェアアクセラレーションに変換する
- Ad-hoc
- 実現したもの：
- GF(2)のセマンティック実装
- gf128hashのループ展開と融合
- CLMUL32/64のインライン化
- パタンマッチングのコード書き換え

第二部

- 入力：gf128hash.cのようなアルゴリズム
- 自動スケジューリングされる
- 出力：PCLMULQDQハードウェア実装

- 手動より10%ぐらいの性能向上

第二部


Linux 7.1 AES-GCM: Exo vs Exo-derived
Exo
Exo-derived
rfc4106(gcm(aes)) encryption
gcm(aes) encryption
rfc4106(gcm(aes)) decryption
gcm(aes) decryption
Bandwidth (MiB/s)
0
50
100
150
200
250
300
350
400

Kernelのハードウェアアクセラレーション

- 現在のExo-derivedよりさらに20%~30%の性能向上を有している

- 原因：
  - ExoはC言語しか出力できない
  - Cコンパイラによってspill、スクジューリングされる
  - 手動fine-tuningには簡単に勝てない

感想

- Exoは「拡張可能」ではあるが、まだ不十分である
  - APIが混雑であり、どこがコンパイラどこがユーザなのかはかなり曖昧

- カーネルレベルのハードウェアアクセラレーションには向いてない
  - Cコンパイラに依存
  - 低レベルのスケジューリング能力を持っていない

五十嵐さんからの返信（抜粋）

- Exo's "user-extensibility" is limited by the set of APIs we chose to expose to users … where those aren't enough

- I've also experienced some frustration where Exo's abstraction was "too high.” … I think the fundamental issue is that we just need a lower-level abstraction for some cases … the lower the language, the more difficult the safety analysis gets, so that's a tradeoff

ありがとうございました