ハードウェアを活用した高速な高能率画像処理

名古屋工業大学

福嶋 慶繁

2023/12/11　　コミュニケーションシステム（CS）研究会

自己紹介

- 名前：福嶋慶繁
- 所属：名古屋工業大学
- 専門：
  - 画像信号処理のアルゴリズム，並列化，近似高速化
  - 画像処理プログラミング言語
- 専門委員：
  - 画像工学研究会（IE） 委員（～2023/6，幹事）
  - PCSJIMPS（IE2種）幹事
  - 信号処理研究会（SIP）委員（～2023/6）
  - 電子情報通信学会英文誌D編集委員（～2023/6）他
  - 通信方式研究会（CS） 委員（～2020/6）

本日の話

- 画像処理の高速化に関する
  - ハードウェア
  - プログラミング

に関する話

- 要約：
  - ハードウェアの変化に対応するためにドメイン固有言語（DSL）を使いましょう

質問

- 普段，研究室・会社でプログラミングに使われている言語は？
  - Python
  - Matlab
  - C/C++
  - アセンブラ，CUDA

どれくらい違うのか？

- 実装を頑張るとどれくらい違うのか？
  - –画像の適応的なアップサンプルを例に
    - Directional Cubic Convolution Interpolation
      - エッジ方向に合わせてcubic補間する方法
      - https://en.wikipedia.org/wiki/Directional_Cubic_Convolution_Interpolation
    - 少しだけ複雑な画像処理

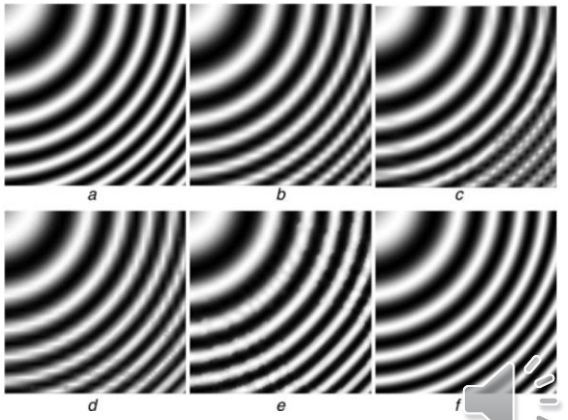


Fig. 7 *4 times zoom of the synthetic image 'Rings'*

a Original
b CC
c DFDF
d EASE
e ICBI
f Proposed

行数と実行結果

- Pythonネイティブ
  - **10**秒，**150**行
- Python（numpy, OpenCV）
  - **10**ms，**100**行
    - C++からライブラリ呼び出ししても大差なし
- C++（並列化ベクトル化フルチューン）
  - **1**ms，**1000**行
- DSL（Halide）
  - **2**ms，**50**行

チューニングパフォーマンス

バイラテラルフィルタ

- OpenCV(IPP)　：26.2ms（ライブラリ）
- Halide（DSL）：　5.5ms
- フルチューン　：　1.5ms

- 使用計算機
  - Cascade lake 10980XE １８コア
- OpenCVの関数は多コアで性能出るとは限らない

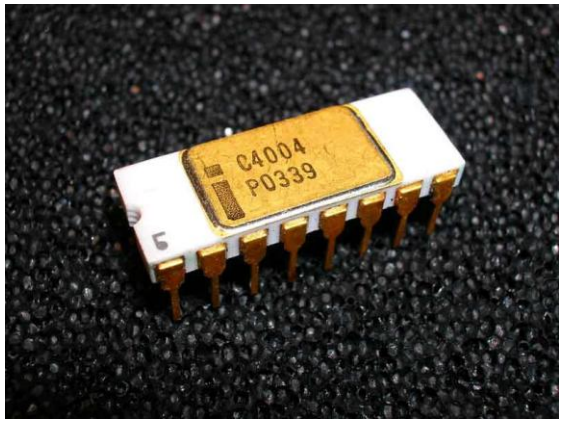
今日の内容

- プログラミングによる差
  - 書き方により**1万**倍の速度差
  - それなりの関数でもチューニングすると数十倍速くなることも
- フルチューンするためのコストとは？
- DSLはどうして短くて速いのか？

アーキテクチャ

計算機の高性能化

- ムーアの法則に従って集積回路のトランジスタ数は年々倍増
  - –多コア，ベクトル演算器，大容量キャッシュに



1971年：Intel 4004
トランジスタ2300個

35万倍

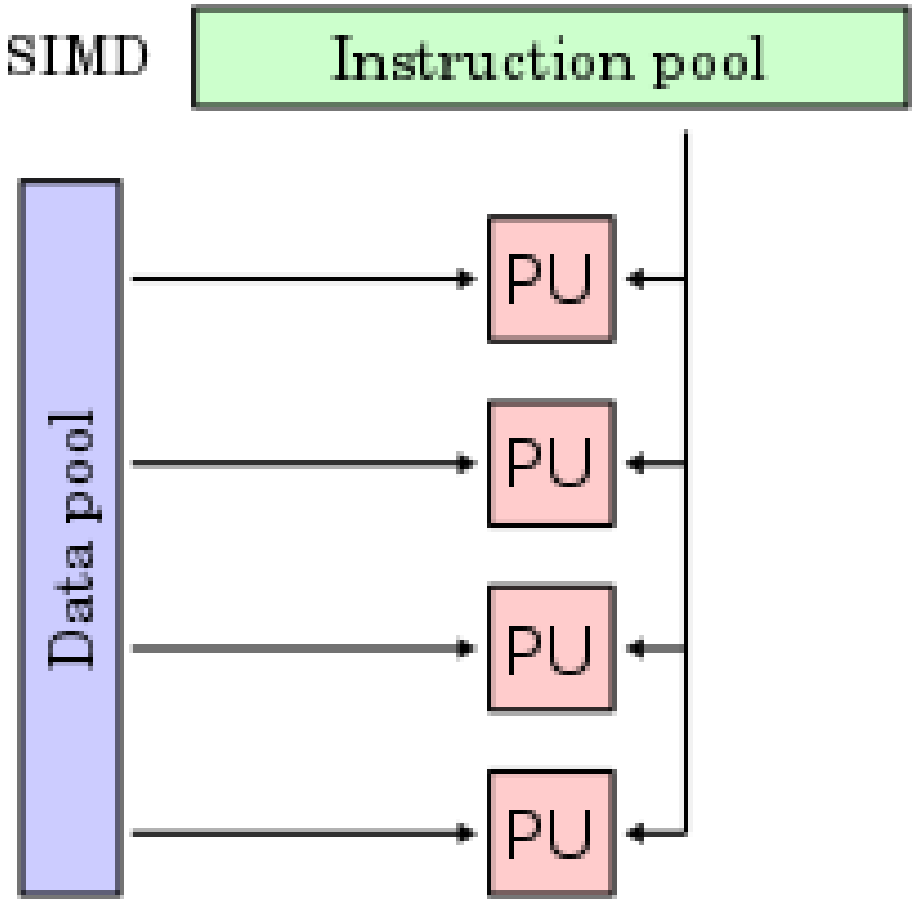
2011年：Core i7 3960X vs Intel 4004

40th Anniversary of the Microprocessor
https://newsroom.intel.com/press-kits/40th-anniversary-of-the-microprocessor/

計算機の並列性（CPU）

- サーバ向け最上位 Intel Xeon 8490Hは，1.9（2.9-3.5）GHz，60コア，**512ビットベクトル演算器**，FMAユニット２つ搭載，８ソケット，**AMXユニット**（1024-16bit）
  - 単純：1.9 GFLOPS→ 3.5 GFLOPS
  - 理想：89.1 TFLOPS(AVX)， 712.7 TFLOPS(AMX)
    - 2.9*60*16*2*2*8=89,088
    - 2.9*60*16*32*8=712,704
- **FLOPS(*Fl*oating-point *O*perations *P*er *S*econd)**
  - 秒間浮動小数点演算を何回計算できるかの指標

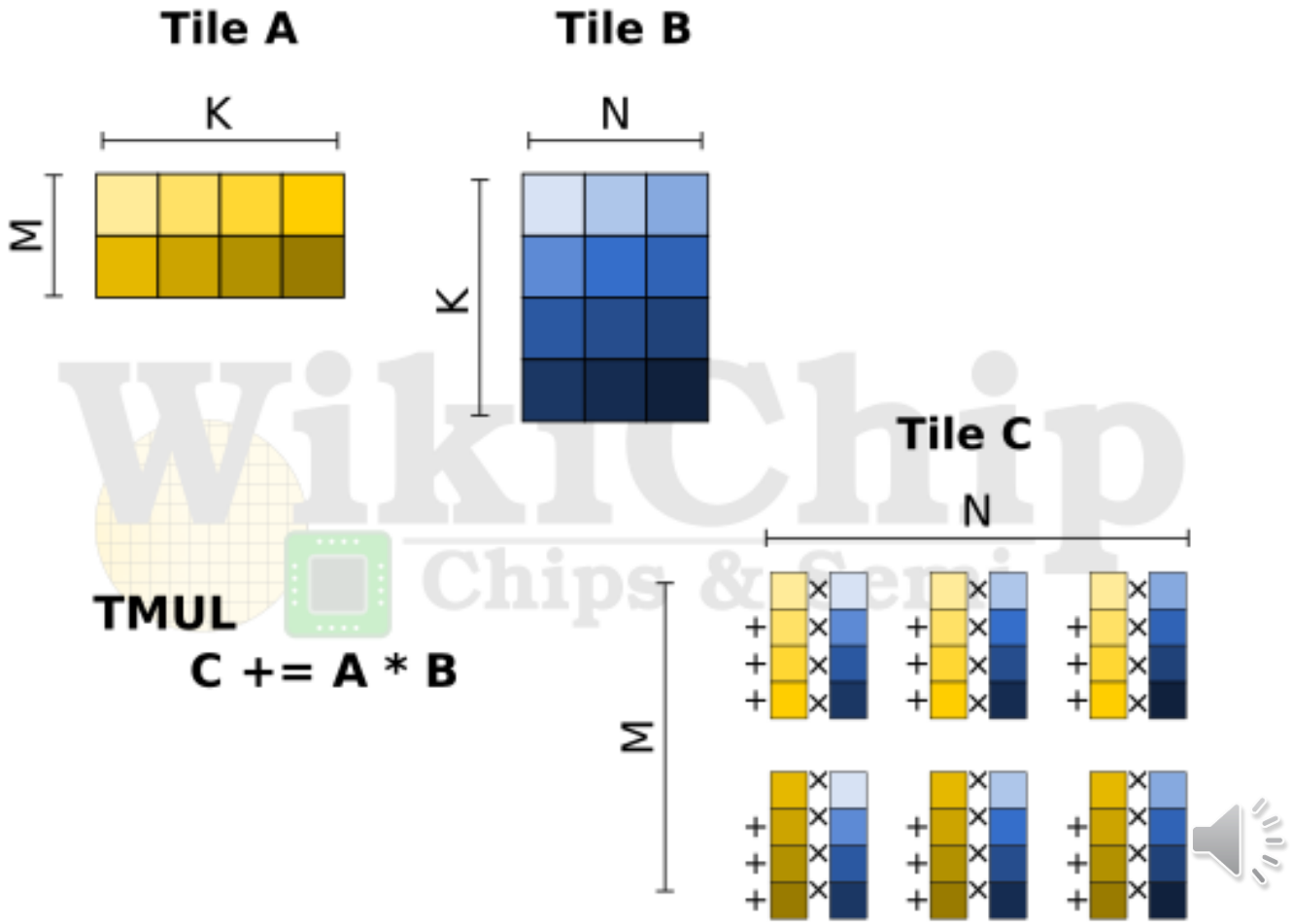
演算器（SIMD）



計算機の並列性（GPU）

- GPU NVIDIA H100は，1.0-1.6GHz，14,592コア，456Tensorコア，FMAユニット
  - 単純：1.0 GFLOPS→ 1.6 GFLOPS
  - 理想：46.7 TFLOPS (CC)， 747.1 TFLOPS (TC)
    - 1.6*14592*2=46694.4
    - 1.6*456*1024=747,110.4
- **FLOPS(*Fl*oating-point *O*perations *P*er *S*econd)**
  - 秒間浮動小数点演算を何回計算できるかの指標

演算器（テンソルコア）



機械学習専用演算器TPU

- Tensor Processing Unit (Google)
- 8bit or 16bit行列演算特化の専用回路
  - bf18 197 TFLOPS , int8 393 TFLOPS
- ポッド： 256チップ（上が256個）単位
- 100POPS
- 類似
  - Apple Neural Engine，Intel NPU他

https://dl.acm.org/doi/pdf/10.1145/3360307

FLOPS（参考）

名称	FLOPS
ENIAC 300	300 FLOPS
地球シミュレータ 1	35.86 TFLOPS
地球シミュレータ 2	122.4 TFLOPS
京	10.51 PFLOPS
神威・太湖之光	93.02 PFLOPS
富岳	442.01PFLOPS
GPU: GeForce RTX4090	82.6 TFLOPS
FPGA: Stratix 10	10 TFLOPS
CPU: Intel Core i9 7980XE	3.0 TFLOPS
Xeon Platinum 8170 x8 CPU	28 TFLOPS

しかし，この性能を引き出すことはかなり難しい

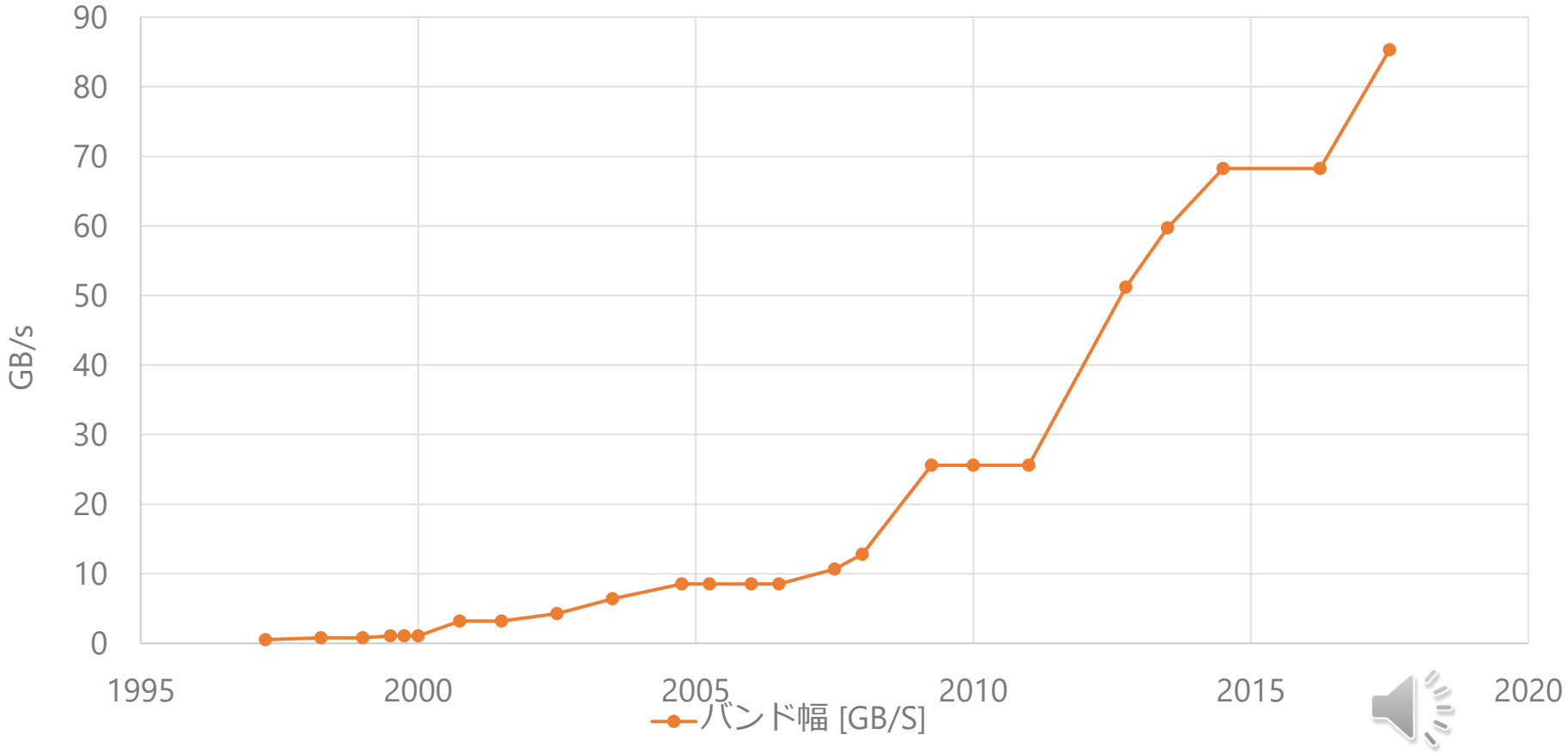
メモリ帯域

周波数ｘデータ転送回数ｘバス幅ｘチャネル数

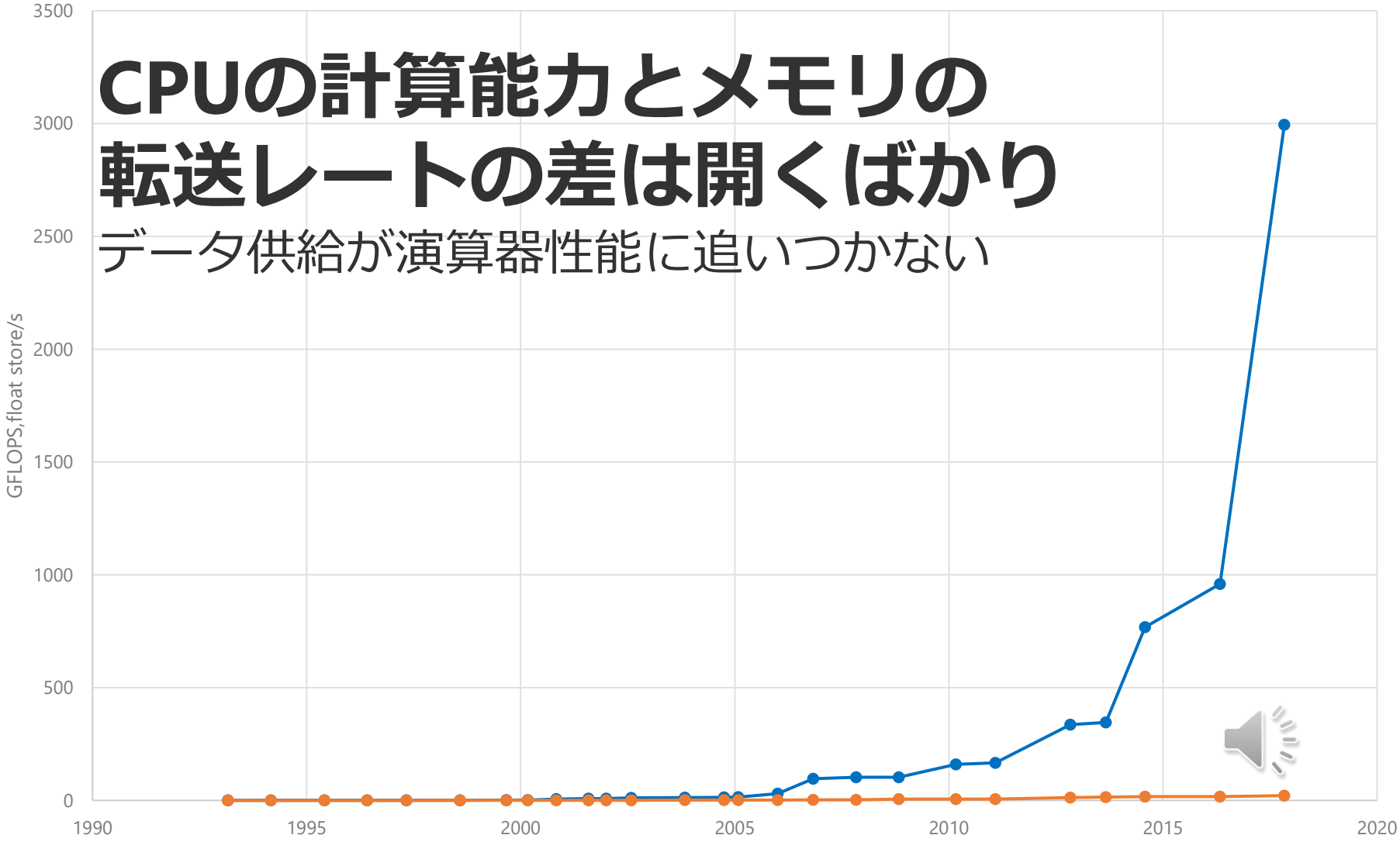
- DDR4 2666クアッドチャネルの場合
  - 周波数：1333 MHz
  - データ転送回数： 2 (DDRはdual data rate)
    - （周波数ｘ転送回数が2666という数字）
  - バス幅：8バイト=64bit

1333 x 2 x 8 x 4=85.3 GB/s

メモリ帯域

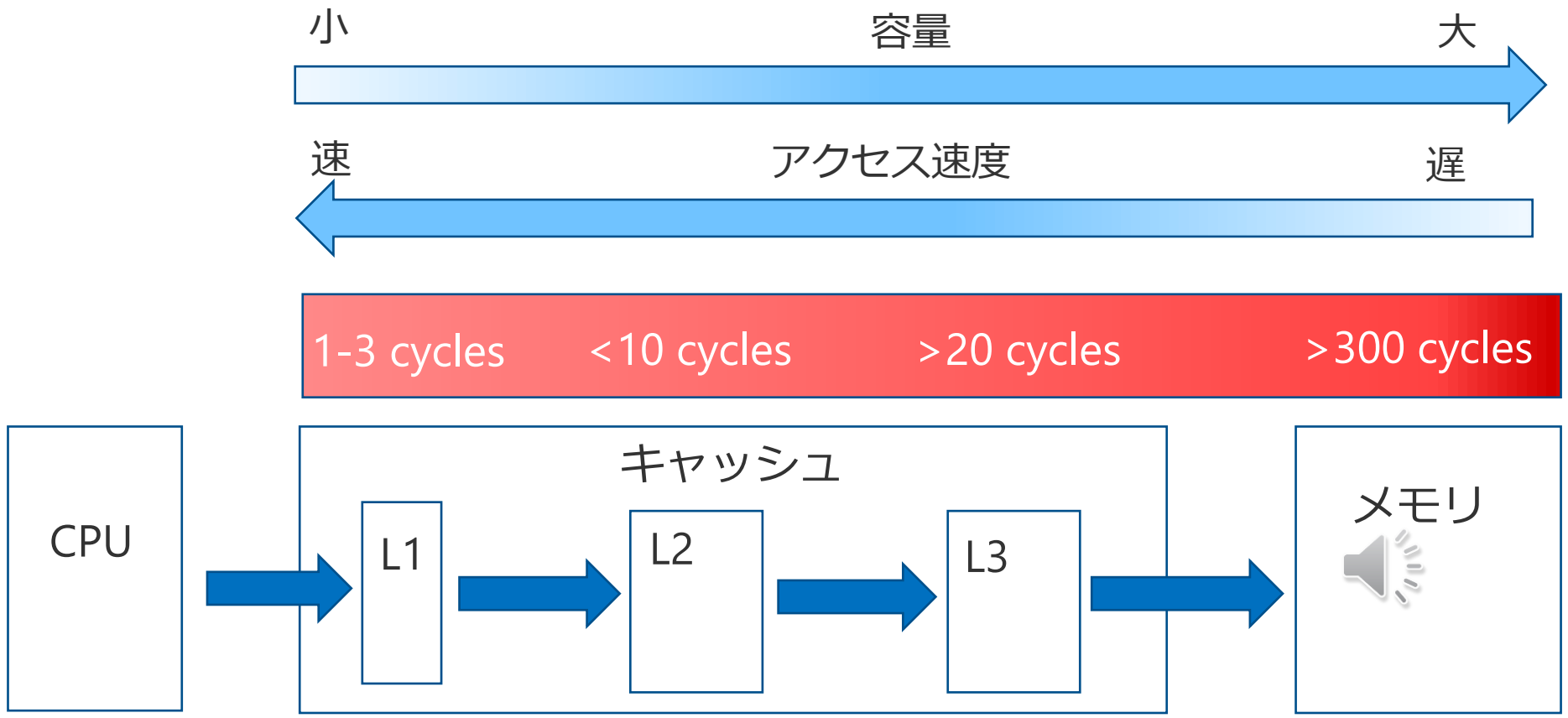

90
80
70
60
50
40
30
20
10
0
GB/s
1995
2000
2005
2010
2015
2020
バンド幅 [GB/S]

FLOPS vs float 転送レート

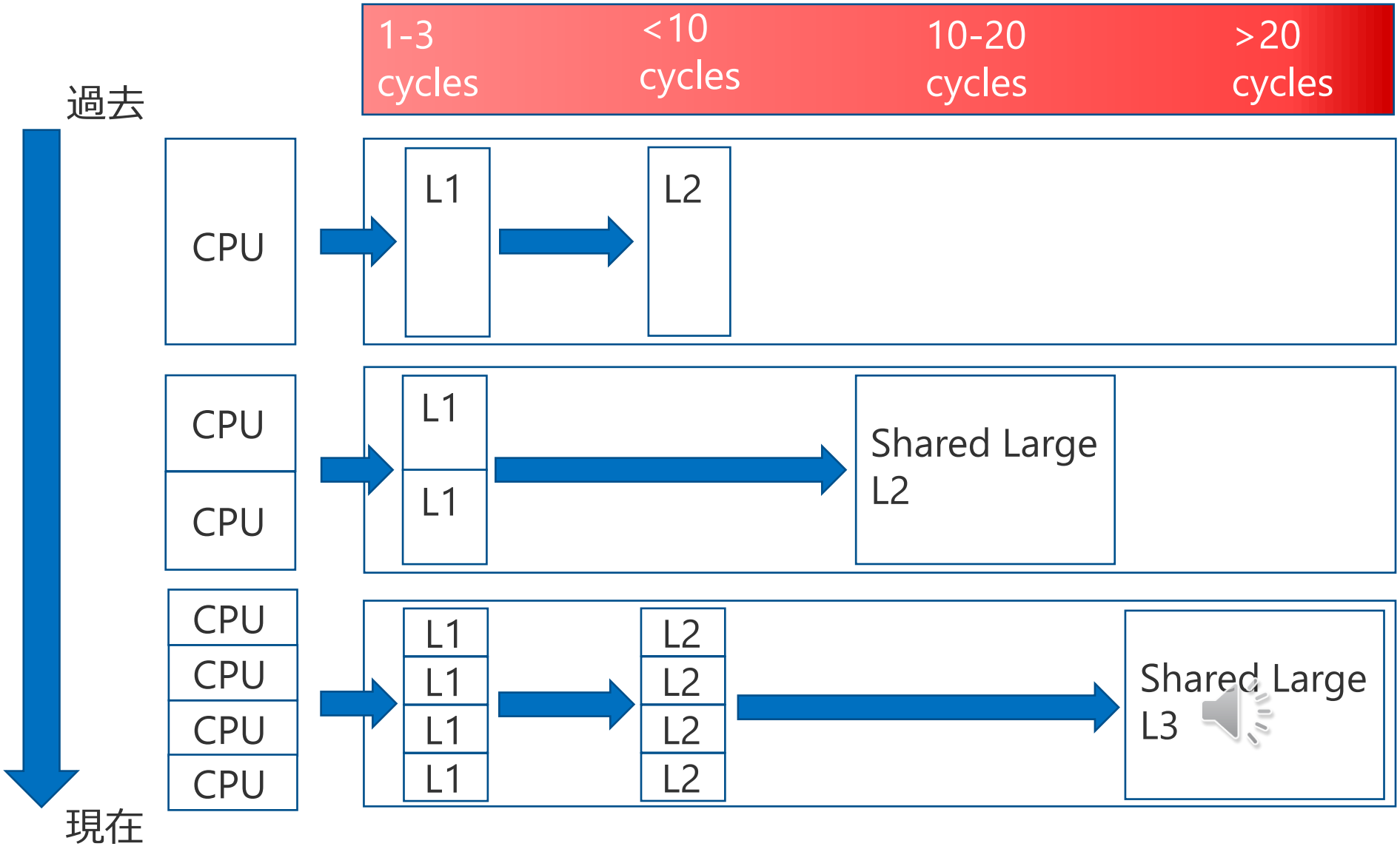


キャッシュ階層とレイテンシ

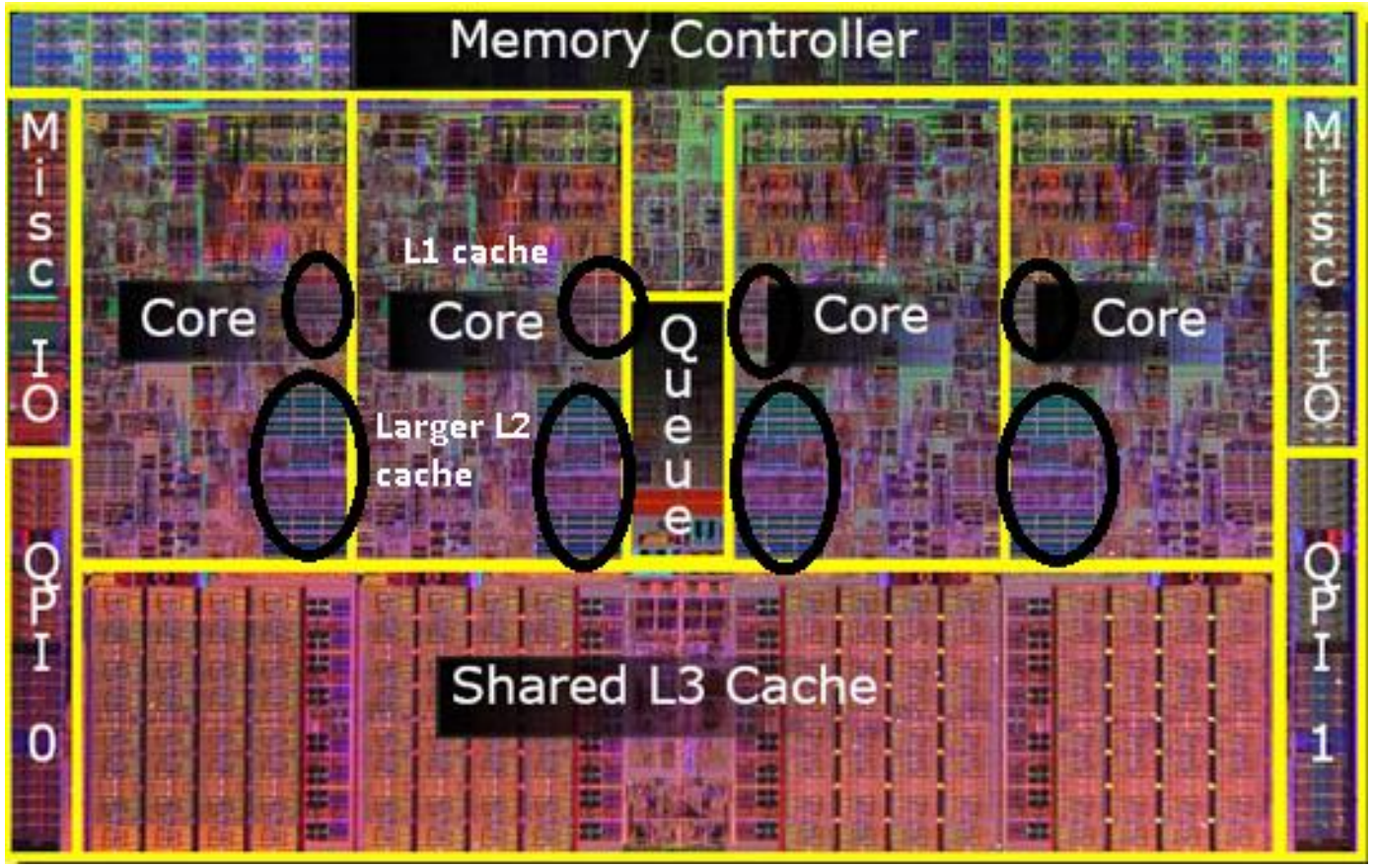
- キャッシュが階層構造になるとメモリからのレイテンシが隠蔽可能



キャッシュ階層とレイテンシ



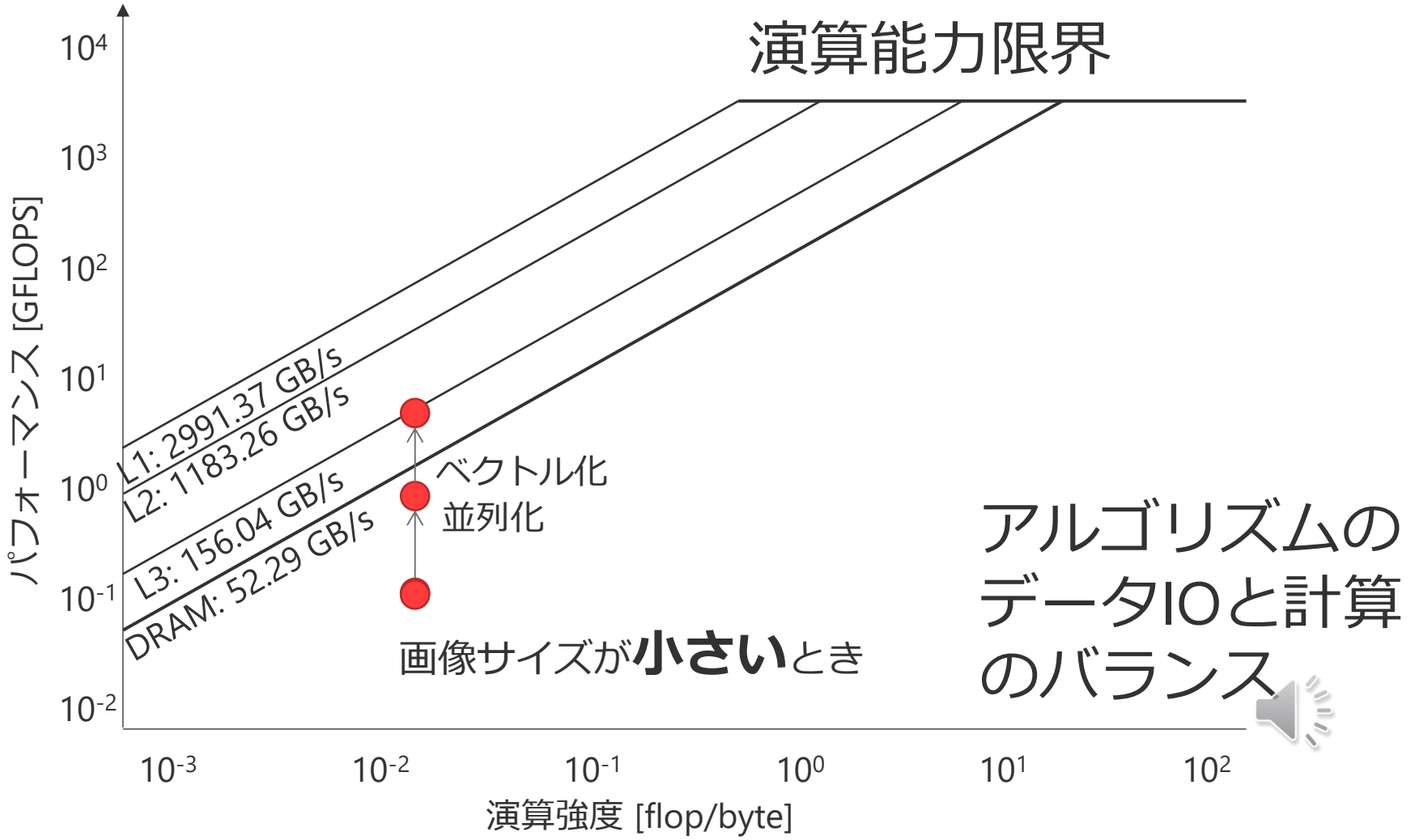
CPUの構造



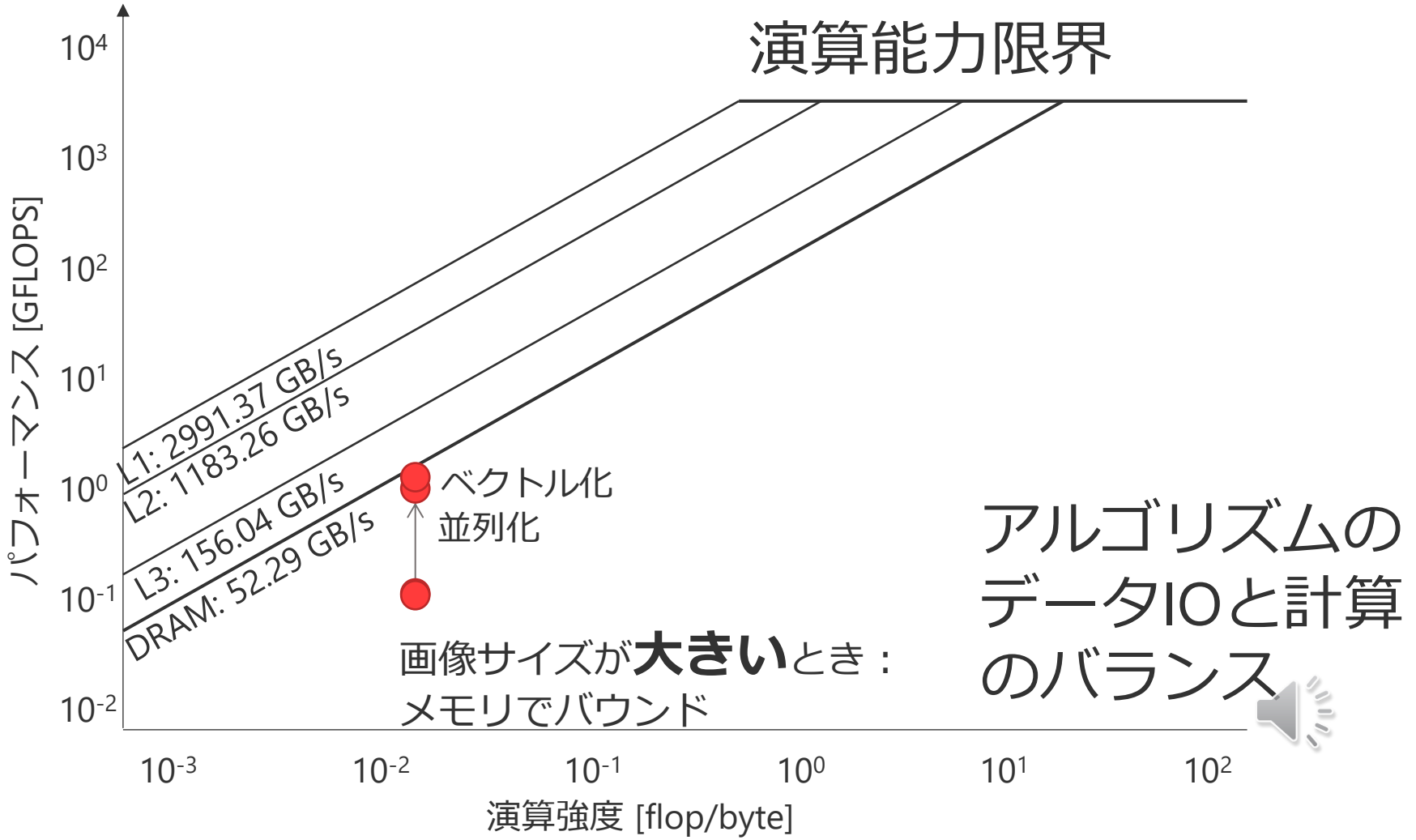
CPUの内部はかなりがキャッシュ

4 Core

ルーフラインによる解析



ルーフラインによる解析

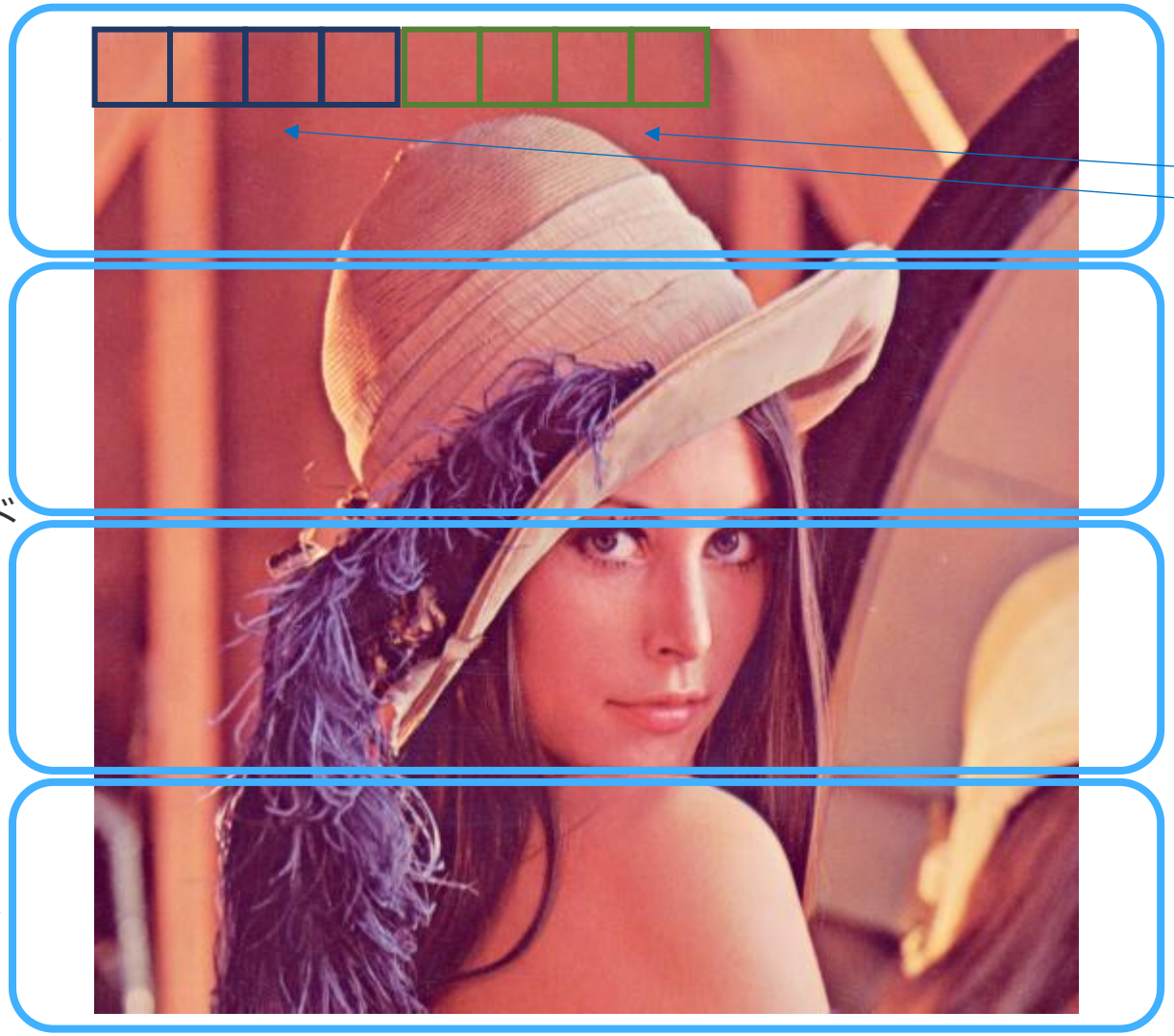


高速化の原則

- 並列化をうまく使わなければ速くならない
- メモリをうまく使わなければ速くならない
- 演算器をうまく使わなければ速くならない
- アクセラレータを使わなければ速くならない

画像処理プログラミング

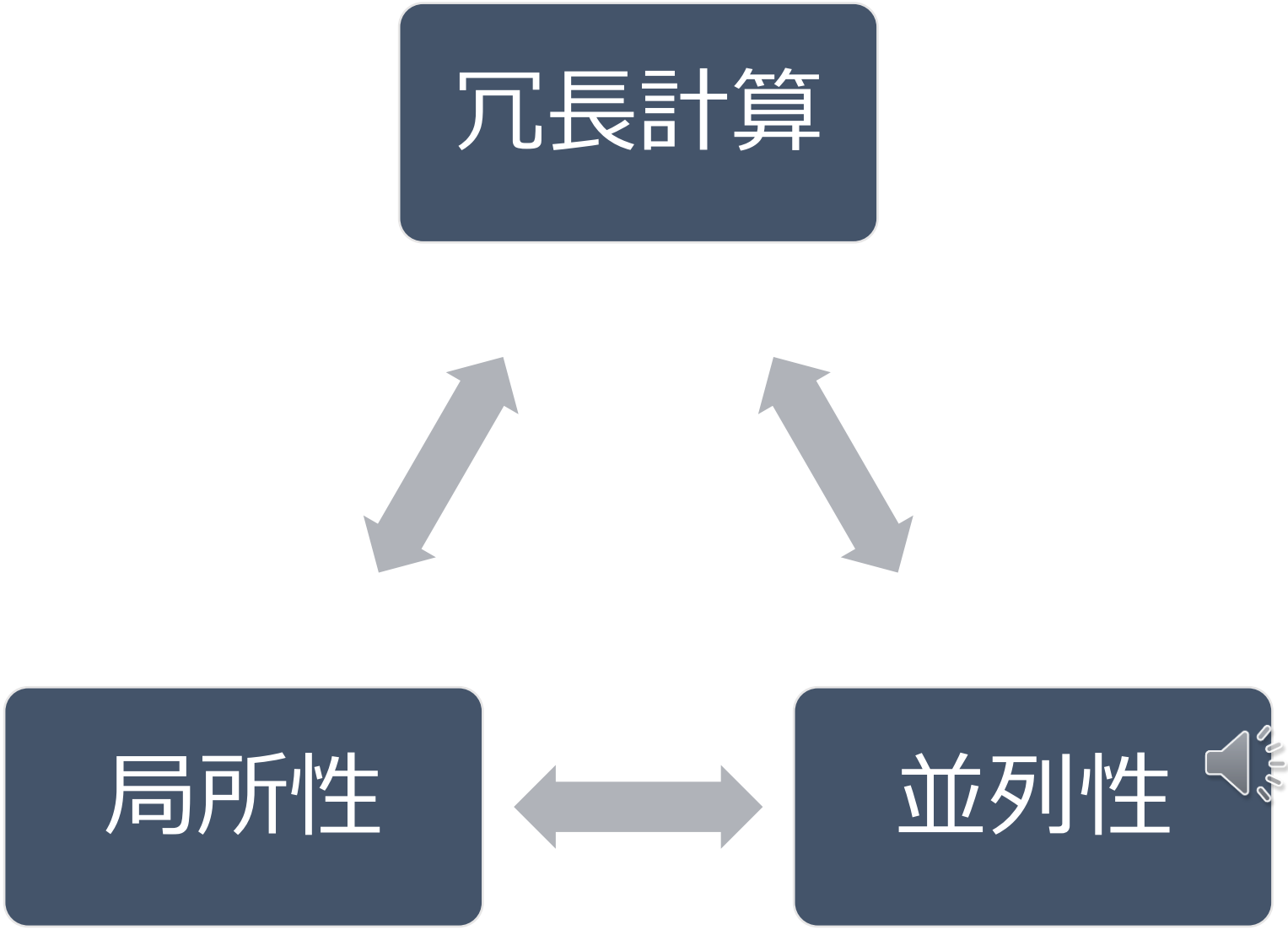
画像処理の並列化



画像処理における高性能計算

- 画像中の画素の計算を並列に，かつキャッシュ効率のよい計算する方法を考えること
  - 演算能力の向上
  - データ転送能力の向上
- 重要な点は下記３つのトレードオフ
  - 並列性
    - 演算能力を上げる
  - 局所性
    - データ転送能力を上げる
  - 冗長計算
    - 両者を満たすために必要な余分な処理

画像処理計算のトレードオフ



時間的局所性

- 画像同士の積和はほぼメモリコピーと同速
  - ax+b
    - 行列プログラミングで書くと
    - Mat a,x,b;
    - Add(Mul(a,x),b)
    - 画素のスキャン2回.
    - これくらいの計算だとメモリコピー2回と計算時間はほぼ一緒
  - for
    - ai*xi+bi
    - スキャン一回で書けば，メモリコピー一回分くらいの計算時間
- 計算に必要な要素がそろった場合は，一回にループですべて計算すると大きく高速化
  - AddやMulの関数を最適化するだけでは，実現不可能
    - AddMul関数を作るなどしなければだめ

例えばSSIMの計算

- ぼかして足しての繰り返し
- 時間の局所性を上げるようにコードを書きなおすと．．．？

$$l(\boldsymbol{p},\boldsymbol{A},\boldsymbol{B}) = \frac{2\mu_{A\boldsymbol{p}}\mu_{B\boldsymbol{p}} + C_1}{{\mu_{A\boldsymbol{p}}}^2 + {\mu_{B\boldsymbol{p}}}^2 + C_1},$$

$$c(\boldsymbol{p},\boldsymbol{A},\boldsymbol{B}) = \frac{2\sigma_{A\boldsymbol{p}}\sigma_{B\boldsymbol{p}} + C_2}{{\sigma_{A\boldsymbol{p}}}^2 + {\sigma_{B\boldsymbol{p}}}^2 + C_2},$$

$$s(\boldsymbol{p},\boldsymbol{A},\boldsymbol{B}) = \frac{\sigma_{AB\boldsymbol{p}} + C_3}{\sigma_{A\boldsymbol{p}}\sigma_{B\boldsymbol{p}} + C_3}.$$

Widely used form

$$\mathrm{SSIM}\,(\boldsymbol{p},\boldsymbol{A},\boldsymbol{B}) = \frac{(2\mu_{A\boldsymbol{p}}\mu_{B\boldsymbol{p}} + C_1)(2\sigma_{AB\boldsymbol{p}} + C_2)}{(\mu_{A\boldsymbol{p}}^2 + \mu_{B\boldsymbol{p}}^2 + C_1)(\sigma_{A\boldsymbol{p}}^2 + \sigma_{B\boldsymbol{p}}^2 + C_2)}.$$

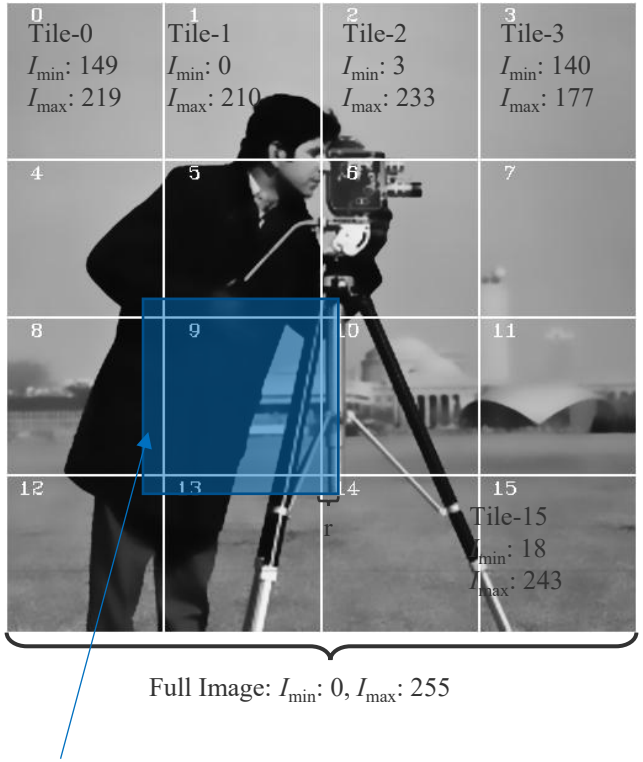
Method	Time [ms]
Naïve	21.01
Proposed	3.30

← 6.36 times faster

データの空間的局所性

- データはキャッシュに収まっている間に再利用する
  - Mat a,b,y;
  - y = Mul(a,x);
  - blur(y)
  - yのデータはメモリに加えて，いったんキャッシュに収まるが，そのキャッシュのサイズあふれるとメモリからとってこなければいけない
- タイリング
  - だいたい64x64～128x128画素の処理をするとキャッシュ効率が良い

タイリング



境界部分をオーバーラップして再計算

マクロブロックとして小さなブロック画像を処理することで空間的局所性の向上

問題点
「各ブロックで並列に計算する場合に冗長計算が必要」

例えばフィルタリング：
畳み込み半径分ブロックの外側の情報も存在しなければいけない

逐次計算なら問題ない
→順次計算していけばよい

ブロック境界が目立ってもよい処理なら簡単だが，ブロック境界を考えると冗長処理が必要

Intel CPU向けに最適化


```
void fast_blur(const Image &in, Image &blurred)
{
    m128i one_third = _mm_set1_epi16(21846);
#pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32)
    {
        m128i a, b, c, sum, avg;
        m128i tmp[(256 / 8) * (32 + 2)];
        for (int xTile = 0; xTile < in.width(); xTile += 256)
        {
            m128i *tmpPtr = tmp;
            for (int y = -1; y < 32 + 1; y++)
            {
                const uint16_t *inPtr = &(in(xTile, yTile + y));
                for (int x = 0; x < 256; x += 8)
                {
                    a = _mm_loadu_si128((m128i *)(inPtr - 1));
                    b = _mm_loadu_si128((m128i *)(inPtr + 1));
                    c = _mm_load_si128((m128i *)(inPtr));
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(tmpPtr++, avg);
                    inPtr += 8;
                }
            }
            tmpPtr = tmp;
            for (int y = 0; y < 32; y++)
            {
                m128i *outPtr = (m128i *)(&(blurred(xTile, yTile + y)));
                for (int x = 0; x < 256; x += 8)
                {
                    a = _mm_load_si128(tmpPtr + (2 * 256) / 8);
                    b = _mm_load_si128(tmpPtr + 256 / 8);
                    c = _mm_load_si128(tmpPtr++);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(outPtr++, avg);
                }
            }
        }
    }
}
```

さらに区切り方を変えれば性能が変わる

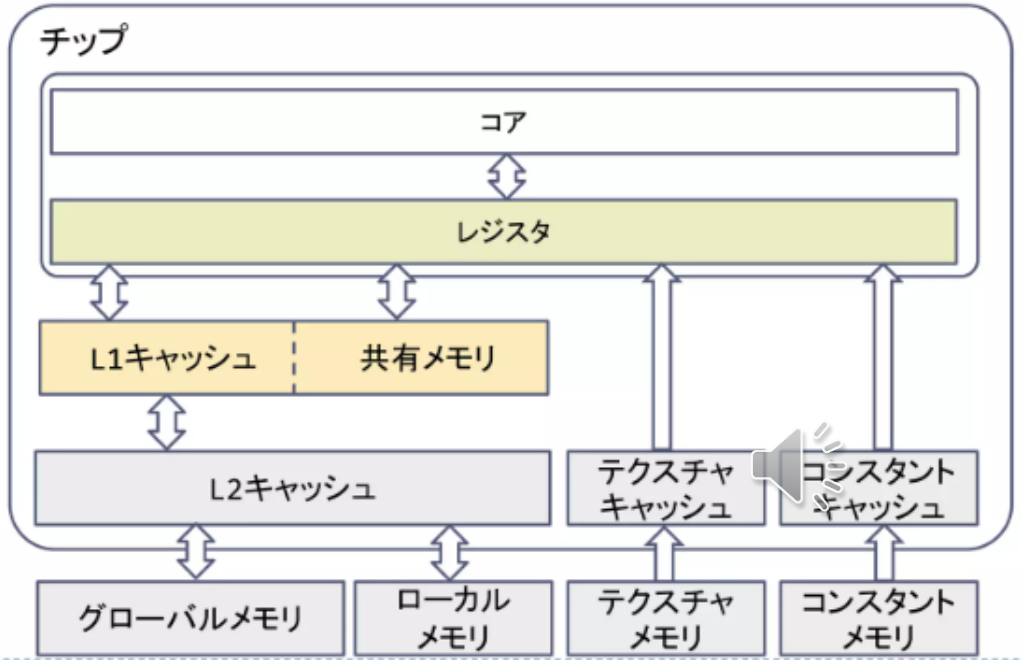
タイリングの違い

- ５ｘ５畳み込み
- タイリングの量を制御をすることで，パイプラインを最適化
- 一番上が基本
- コード量1関数3万行越えがｘ10個

```
F2D RR    a32s  E(1,1) PSNR 56.75 0.1179 x52.84
F2D RR    a32s  E(2,1) PSNR 56.75 0.0902 x69.06
F2D RR    a32s  E(4,1) PSNR 56.75 0.0787 x79.16
F2D RR    a32s  E(1,2) PSNR 56.75 0.0929 x67.06
F2D RR    a32s  E(2,2) PSNR 56.75 0.0782 x79.66
F2D RR    a32s  E(4,2) PSNR 56.75 0.0806 x77.29
F2D RR    a32s  E(1,3) PSNR 56.75 0.0863 x72.18
F2D RR    a32s  E(1,4) PSNR 56.75 0.0835 x74.60
F2D RR    a32s  E(1,5) PSNR 56.75 0.0852 x73.12
F2D RR    a32s  E(1,6) PSNR 56.75 0.0867 x71.85
F2D RR    a32s  E(1,7) PSNR 56.75 0.0887 x70.23
F2D RR    a32s  E(1,8) PSNR 56.75 0.0908 x68.61
F2D RR    a32s  B(1,2) PSNR 56.75 0.0931 x66.91
F2D RR    a32s  B(2,2) PSNR 56.75 0.0782 x79.66
F2D RR    a32s  B(4,2) PSNR 56.75 0.0803 x77.58
F2D RR    a32s  B(1,3) PSNR 56.75 0.0864 x72.10
F2D RR    a32s  B(1,4) PSNR 56.75 0.0835 x74.60
F2D RR    a32s  B(1,5) PSNR 56.75 0.0853 x73.03
F2D RR    a32s  B(1,6) PSNR 56.75 0.0866 x71.93
F2D RR    a32s  B(1,7) PSNR 56.75 0.0888 x70.15
F2D RR    a32s  B(1,8) PSNR 56.75 0.0906 x68.76
============ Conventional =============
Kelefouas normal      PSNR 57.80 0.1468 x42.44
Kelefouas reg.block   PSNR 57.80 0.1403 x44.24
Kelefouas etc         PSNR 57.80 0.1534 x40.61
```

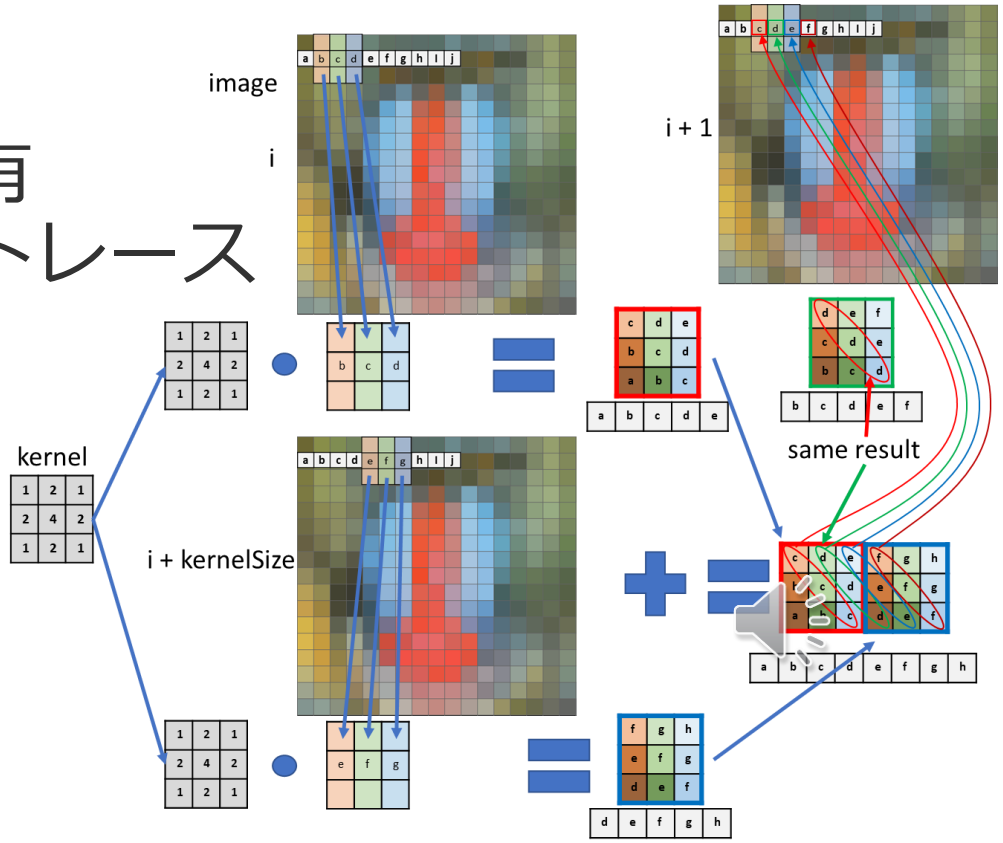
GPUの場合

- タイリングのように，使いまわすデータは共有メモリに明示的に冗長に保存して書く
- 並列化は書いたらある程度勝手に
  - ただし，より効率的な並列化記述方法あり



専用ハードの使用

- 行列積回路を使って画像処理を書き直す
  - 例：普通の畳み込みは行列積には簡単に変更不能
  - アルゴリズムを行列で記述しなおす必要有
    この場合，行列積のトレースの部分和



性能を出すためには？

- 表現したいアルゴリズム以外にハードウェアの特性を生かした部分の記述をしないといけない
- ハードウェアに関する深い知識がある人しかそのような高度は書けない？

「フリーランチの終焉」*

多コアCPUが出てきた時の言葉から20年

2000年のHT，2005年のPentium D

*Sutter, Herb. "The free lunch is over: A fundamental turn toward concurrency in software." *Dr. Dobb's Journal* 30.3 (**2005**): 202-210.

ドメイン固有言語

ドメイン固有言語

- Domain-Specific Language（DSL）
  - 何かに特化した専用プログラミング言語
  - チューリング完全ではないため，何でもは作れない
- 身近な例
  - SQL，CSS，Make

- 画像処理専用DSL
  - Halide (http://halide-lang.org/)
  - Darkroom (http://darkroom-lang.org/)
  - Forma (https://github.com/NVIDIA/Forma)

  - など，近年多数登場.

プログラミング：Halide

- アルゴリズムとスケジューリングの記述が分離
  - アルゴリズム
    - **どのような処理か(What)**
    - 画像処理の本質を記述
    - ハードウェアに依らず単一
  - スケジューリング（ハードウェア記述）
    - **どのように処理するか(How)**
    - 高速化についての記述
    - 並列化・ベクトル化・ループ変形・データ構造変形…
    - ただし，アルゴリズムの計算結果を変化させない
    - ハードウェアに依って適切に変更

Halide

- マルチプラットフォームに対応
  - CPU
    - X86, ARM, MIPS, Hexagon, Power-PC...
  - GPU
    - CUDA, OpenCL, OpenGL, Direct X 12...
  - OS
    - Linux, Windows, MacOS, Android, iOS...

- **ハードウェアによったチューニングが容易**
  - アルゴリズムを変えることなく，様々なハードウェアに依ってスケジュールだけを変更可能
  - 深層学習の最適化バックエンドはHalideから構成（されていた）

C++による3x3ボックスフィルタ 44

```
void blur(const Image &in, Image &blurred)
{
    Image tmp(in.width(), in.height());
    for (int y = 0; y < in.height(); y++)
        for (int x = 0; x < in.width(); x++)
            tmp(x, y) = (in(x - 1, y) + in(x, y) + in(x + 1, y)) / 3;
    for (int y = 0; y < in.height(); y++)
        for (int x = 0; x < in.width(); x++)
            blurred(x, y) = (tmp(x, y - 1) + tmp(x, y) + tmp(x, y + 1)) / 3;
}
```

Intel CPU向けに最適化


```
void fast_blur(const Image &in, Image &blurred)
{
    m128i one_third = _mm_set1_epi16(21846);
#pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32)
    {
        m128i a, b, c, sum, avg;
        m128i tmp[(256 / 8) * (32 + 2)];
        for (int xTile = 0; xTile < in.width(); xTile += 256)
        {
            m128i *tmpPtr = tmp;
            for (int y = -1; y < 32 + 1; y++)
            {
                const uint16_t *inPtr = &(in(xTile, yTile + y));
                for (int x = 0; x < 256; x += 8)
                {
                    a = _mm_loadu_si128((m128i *)(inPtr - 1));
                    b = _mm_loadu_si128((m128i *)(inPtr + 1));
                    c = _mm_load_si128((m128i *)(inPtr));
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(tmpPtr++, avg);
                    inPtr += 8;
                }
            }
            tmpPtr = tmp;
            for (int y = 0; y < 32; y++)
            {
                m128i *outPtr = (m128i *)(&(blurred(xTile, yTile + y)));
                for (int x = 0; x < 256; x += 8)
                {
                    a = _mm_load_si128(tmpPtr + (2 * 256) / 8);
                    b = _mm_load_si128(tmpPtr + 256 / 8);
                    c = _mm_load_si128(tmpPtr++);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(outPtr++, avg);
                }
            }
        }
    }
}
```

Halideによる3x3ボックスフィルタ


```
Func halide_blur(Func in)
{
    Func tmp, blurred;
    Var x, y, xi, yi;
    // The algorithm
    tmp(x, y) = (in(x - 1, y) + in(x, y) + in(x + 1, y)) / 3;
    blurred(x, y) = (tmp(x, y - 1) + tmp(x, y) + tmp(x, y + 1)) / 3;
    // The schedule
    blurred.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8)
        .parallel(y);
    tmp.compute_at(blurred, x).vectorize(x, 8);
    return blurred;
}
```

アルゴリズム部分

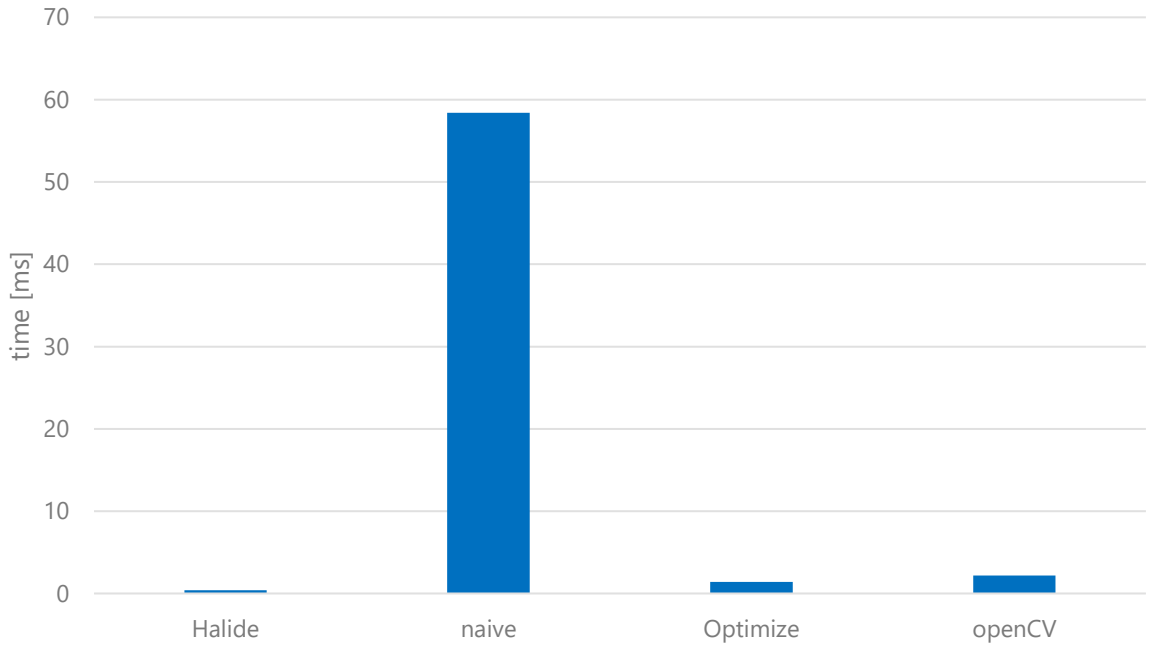
スケジューリング部分

Halideの性能

- 計算機
  - CPU : Intel Core i7-8550U
  - RAM : 8GB
  - OS : Windows 10 Pro
- 入力画像
  - 512x512 グレー画像
- ガウシアンフィルタのパラメータ
  - σ = 5
  - カーネル半径 = 15

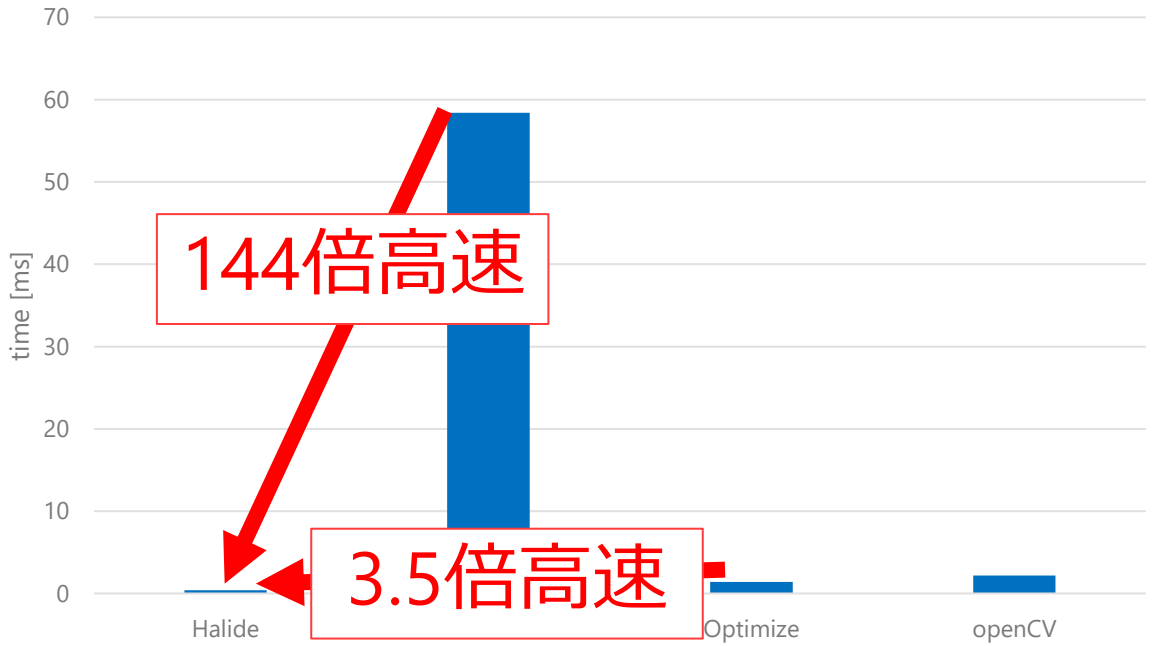
実験結果

- Halide vs ナイーブな実装 vs 手作業で最適化 vs OepnCV
  - Halide … 0.406005 ms
  - ナイーブ(naive) … 58.4152 ms
  - 手作業最適化(optimized) … 1.40956 ms
  - OpenCV … 2.16409 ms



実験結果

- Halide vs ナイーブな実装 vs 手作業で最適化 vs OepnCV
  - Halide … 0.406005 ms
  - ナイーブ(naive) … 58.4152 ms
  - 手作業最適化(optimized) … 1.40956 ms
  - OpenCV … 2.16409 ms

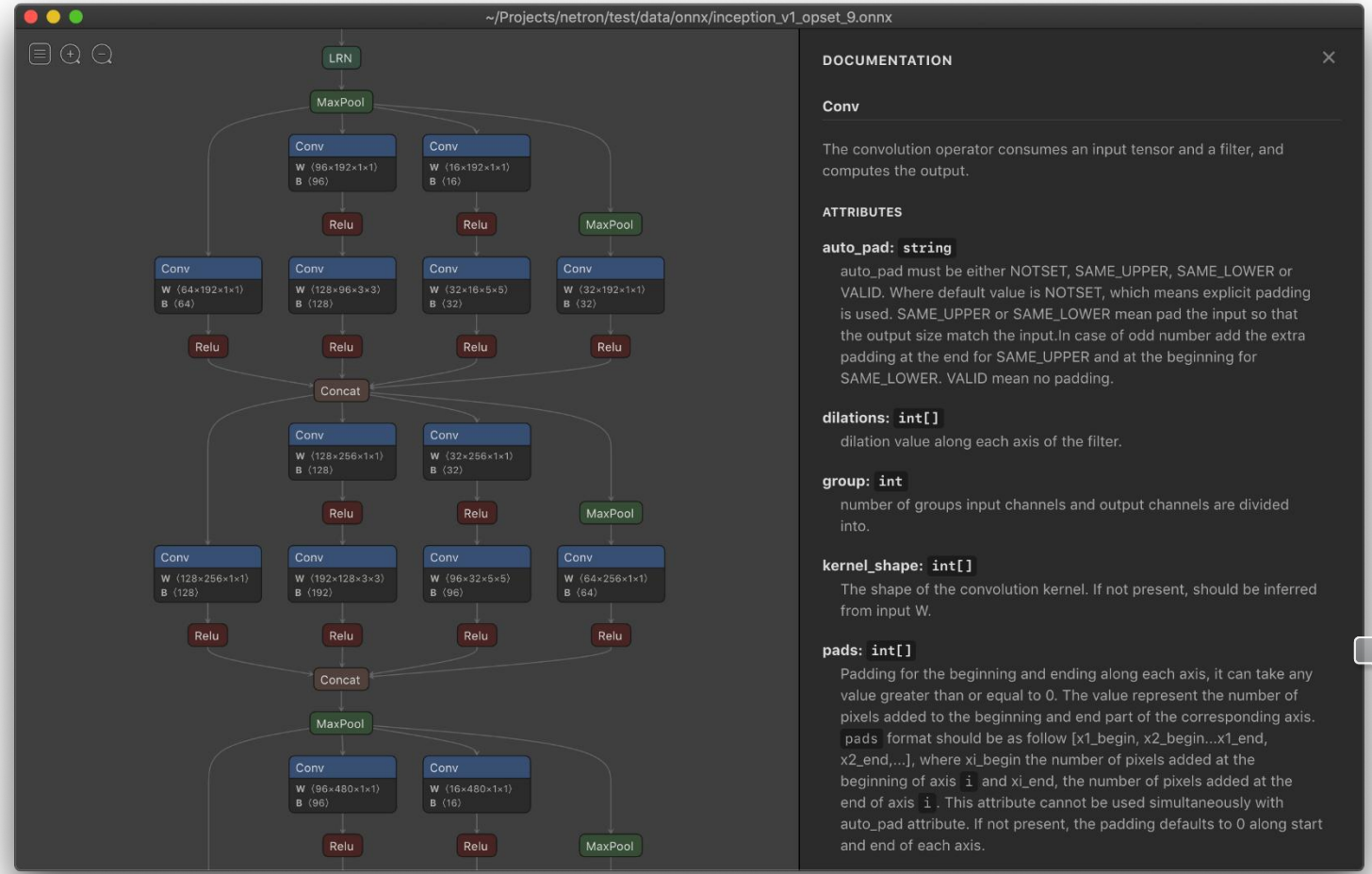


実は身近なDSL

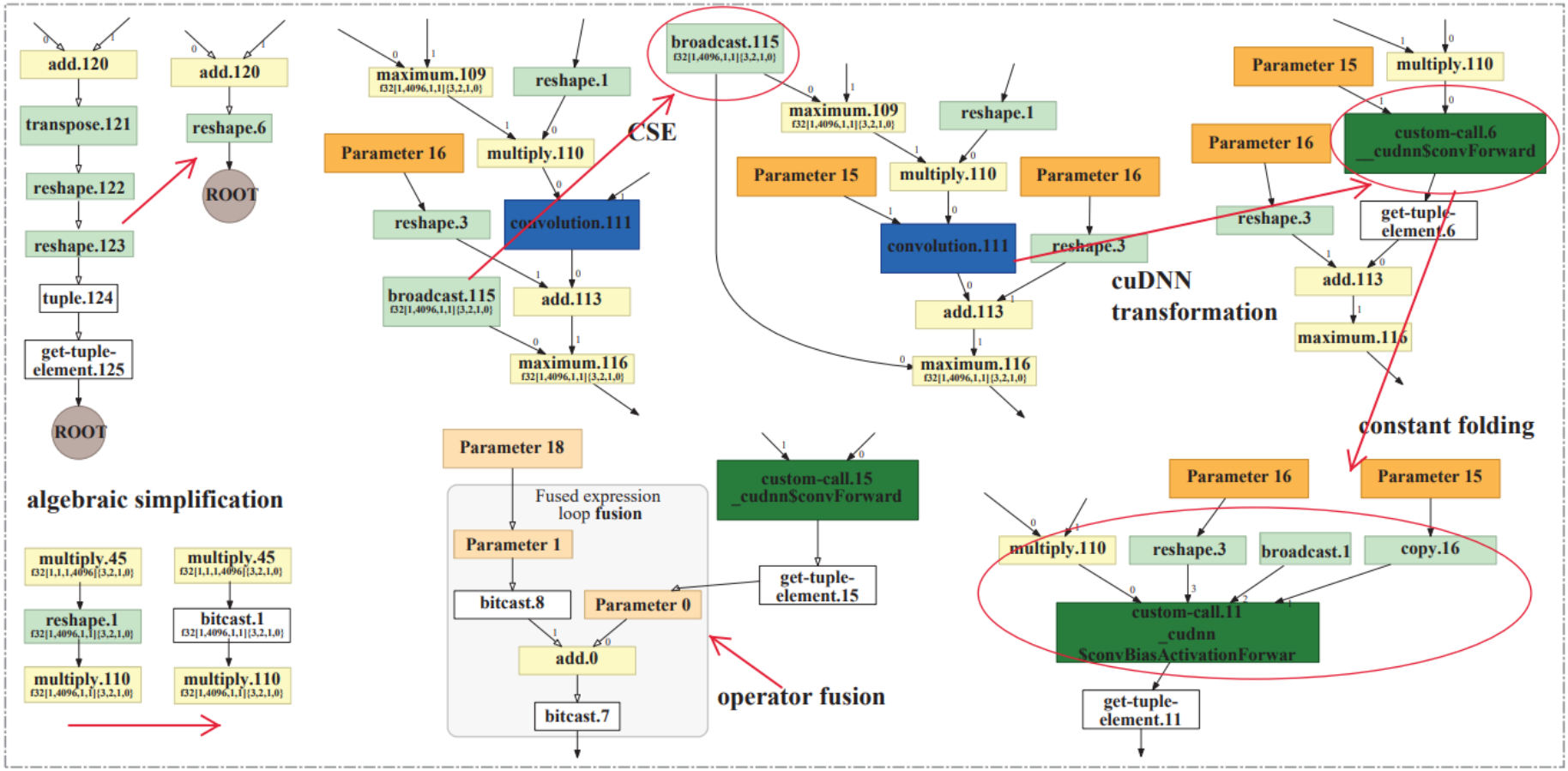
- PyTorch，TensorFlowなどを使ってPythonで書かれた機械学習コードは**実はDSL**
- ネットワーク構造を記述する文法をPythonに埋め込んだネットワーク構造記述言語に等しい→出力はネットワーク構造
- ネットワークを書いた通りに設計するだけで，書いた順番通りに動かない.

NETRONによる可視化

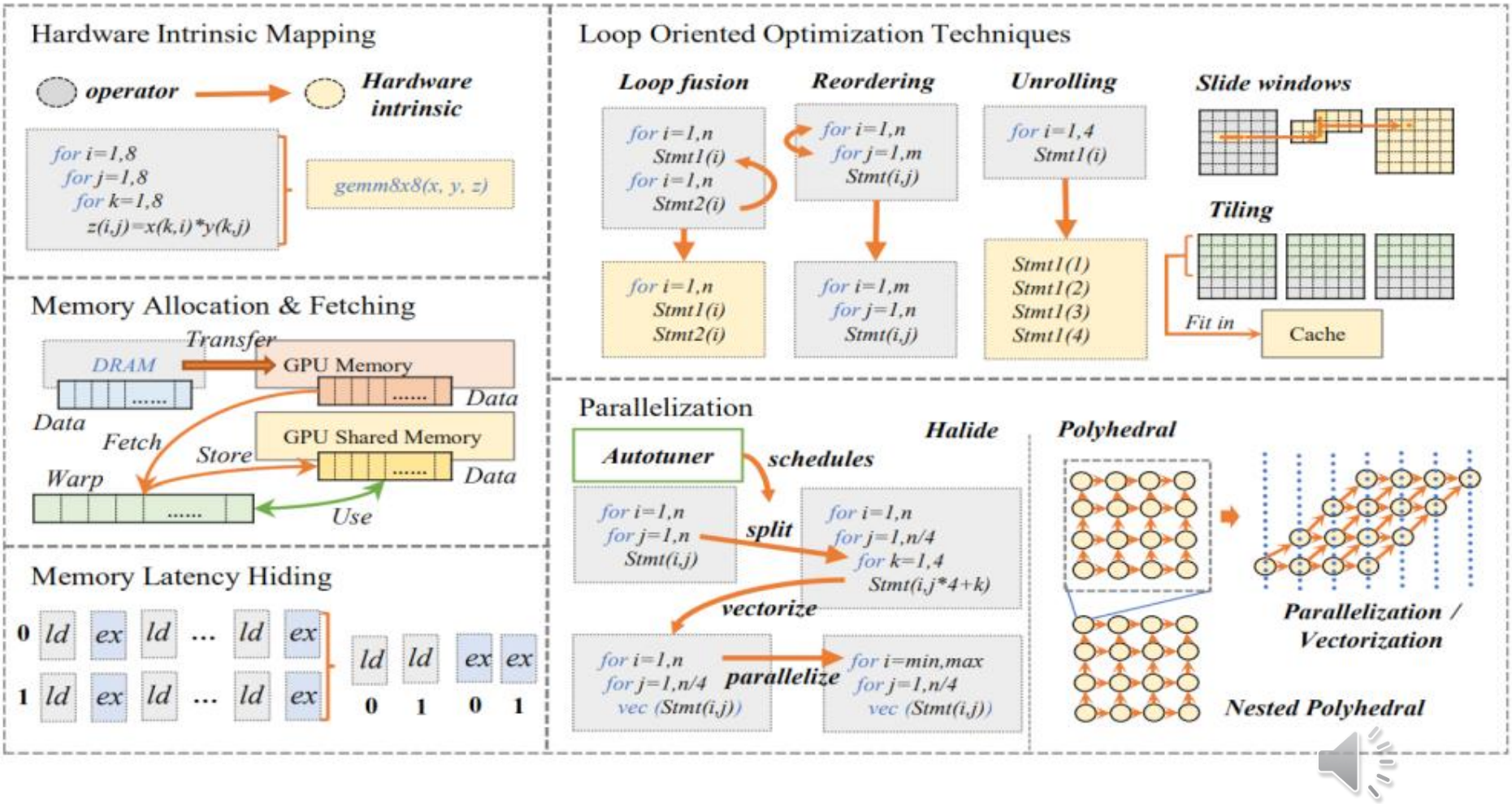
- Netron is a viewer for neural network, deep learning and machine learning models.



グラフ最適化の例


add.120
transpose.121
reshape.122
reshape.123
tuple.124
get-tuple-element.125
ROOT
add.120
reshape.6
ROOT
algebraic simplification
multiply.45
reshape.1
multiply.110
multiply.45
bitcast.1
multiply.110
maximum.109
reshape.1
Parameter 16
multiply.110
reshape.3
convolution.111
broadcast.115
add.113
maximum.116
CSE
broadcast.115
maximum.109
reshape.1
Parameter 15
multiply.110
Parameter 16
convolution.111
reshape.3
add.113
maximum.116
cuDNN transformation
Parameter 15
multiply.110
Parameter 16
custom-call.6 __cudnnSconvForward
reshape.3
get-tuple-element.6
add.113
maximum.116
constant folding
Parameter 18
Fused expression loop fusion
Parameter 1
bitcast.8
Parameter 0
add.0
bitcast.7
custom-call.15 _cudnnSconvForward
get-tuple-element.15
operator fusion
Parameter 16
Parameter 15
multiply.110
reshape.3
broadcast.1
copy.16
custom-call.11 _cudnn SconvBiasActivationForwar
get-tuple-element.11

低レベル最適化の例


Hardware Intrinsic Mapping
operator
Hardware intrinsic
for i=1,8
for j=1,8
for k=1,8
z(i,j)=x(k,i)*y(k,j)
gemm8x8(x, y, z)
Memory Allocation & Fetching
Transfer
DRAM
GPU Memory
Data
Data
Fetch
Store
GPU Shared Memory
Data
Warp
Use
Memory Latency Hiding
0 ld ex ld ... ld ex
1 ld ex ld ... ld ex
ld ld ex ex
0 1 0 1
Loop Oriented Optimization Techniques
Loop fusion
for i=1,n
Stmt1(i)
for i=1,n
Stmt2(i)
for i=1,n
Stmt1(i)
Stmt2(i)
Reordering
for i=1,n
for j=1,m
Stmt(i,j)
for i=1,m
for j=1,n
Stmt(i,j)
Unrolling
for i=1,4
Stmt1(i)
Stmt1(1)
Stmt1(2)
Stmt1(3)
Stmt1(4)
Slide windows
Tiling
Fit in
Cache
Parallelization
Autotuner
schedules
Halide
for i=1,n
for j=1,n
Stmt(i,j)
split
for i=1,n
for j=1,n/4
for k=1,4
Stmt(i,j*4+k)
vectorize
for i=1,n
for j=1,n/4
vec (Stmt(i,j))
parallelize
for i=min,max
for j=1,n/4
vec (Stmt(i,j))
Polyhedral
Parallelization / Vectorization
Nested Polyhedral

Deep Learning Compiler

- Apache TVM（ワシントン大）
  - https://tvm.apache.org
- TensorFlow XLA（Accelerated Linear Algebra）（Google）
  - https://www.tensorflow.org/xla
- Glow（Meta）
  - https://github.com/pytorch/glow
- Tensor Comprehensions（Meta）
  - https://github.com/facebookresearch/TensorComprehensions
- Tiramisu（MIT）
  - https://github.com/Tiramisu-Compiler/tiramisu
- Intel nGraph（Intel）
  - https://www.intel.co.jp/content/www/jp/ja/artificial-intelligence/ngraph.html
- ONNC (Open Neural Network Compiler)
  - https://github.com/ONNC/onnc (更新ほぼない)

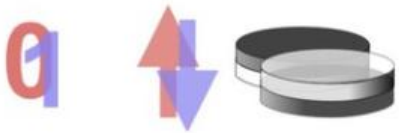
量子コンピュータの動作原理

古典ビット

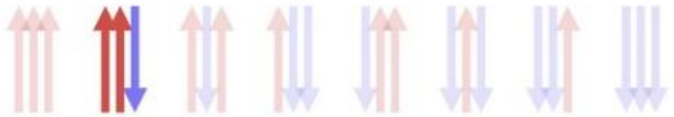
0か1のいずれかの状態

量子ビット

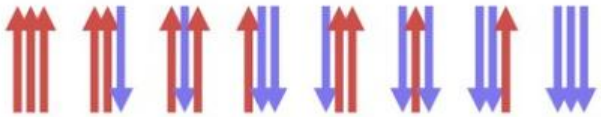
0と1を重ね合わせて同時に表す



3ビットで8つの状態のいずれかを表す

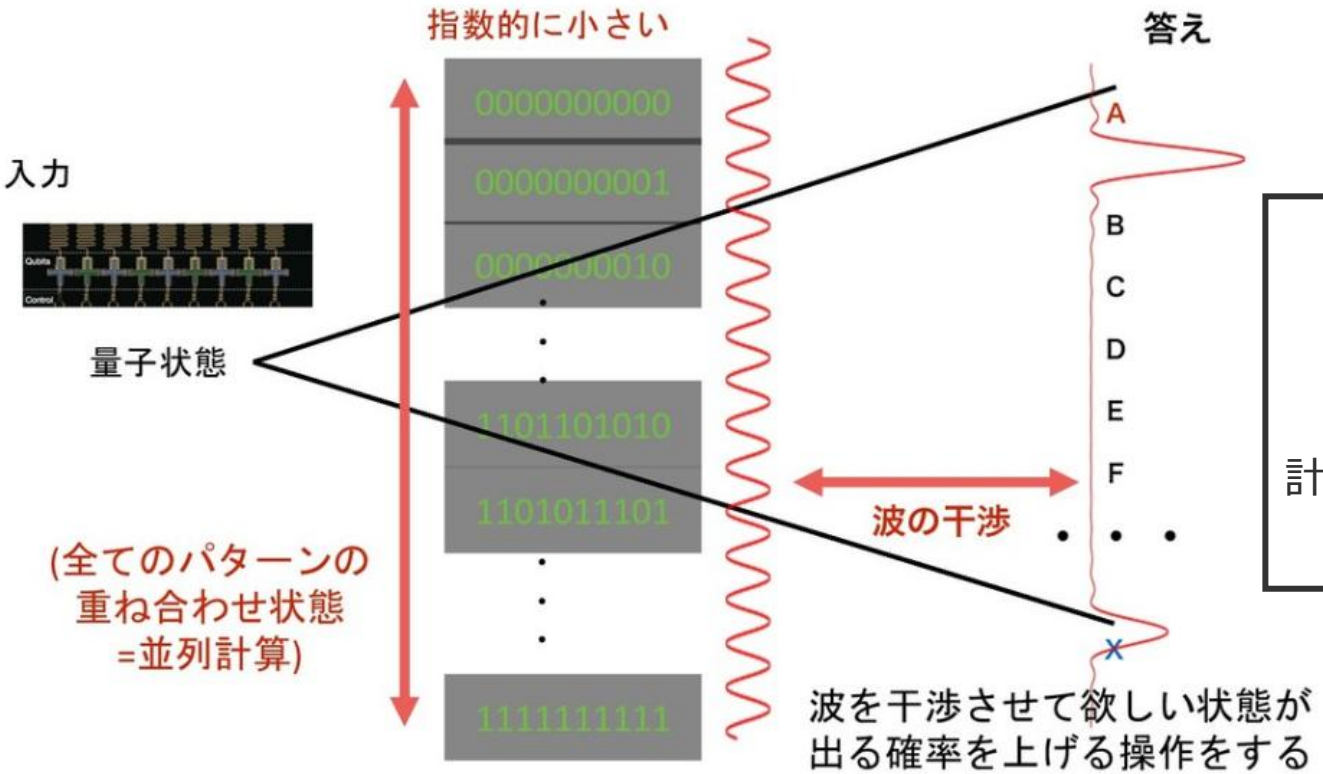


3量子ビットで8つの状態を同時に表せる



量子の重ね合わせ状態を利用 → 0と１の2値情報を確率的に表現

量子コンピュータの動作原理



重ね合わせ状態を利用
確率的な状態として保持

⇩

計算量を減らせる場合が存在

状態の操作→テンソルの操作→全て行列演算

cuQuantum

cuQuantum

・古典コンピュータで量子回路シミュレーションを行うシミュレータ

・cuStatevec・cuTensorNetの２つのライブラリを持つ

cuStatevec

・量子アルゴリズムを表現するための状態ベクトルを扱うライブラリ

・状態ベクトルの記述・量子ゲート操作・マルチGPU計算などを行う

cuTensorNet

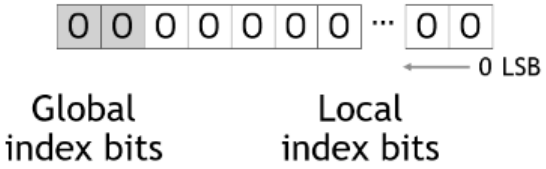
・量子回路をテンソルネットワークで表現・計算するライブラリ

・テンソルの定義・縮約を行い量子状態に適用する

cuStatevec

cuStatevec

- 状態ベクトルを配列で表現
  - リトルエンディアン方式で格納
- 量子ビットを分割してメモリに格納
  - GPUで管理（Global）とホストメモリで管理（Local）



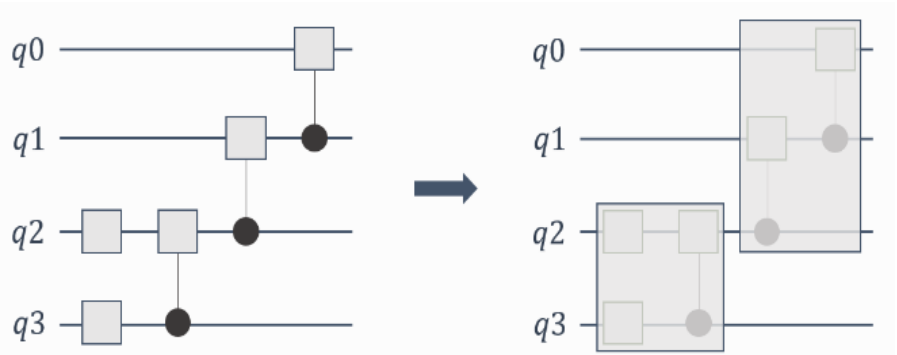
状態ベクトルの表現[1]

量子ゲート操作

量子ゲートの計算コストが大きい
- 複数の量子ゲートをまとめる

⇩

計算・メモリコストの削減



量子ゲート操作[2]

cuStatevec

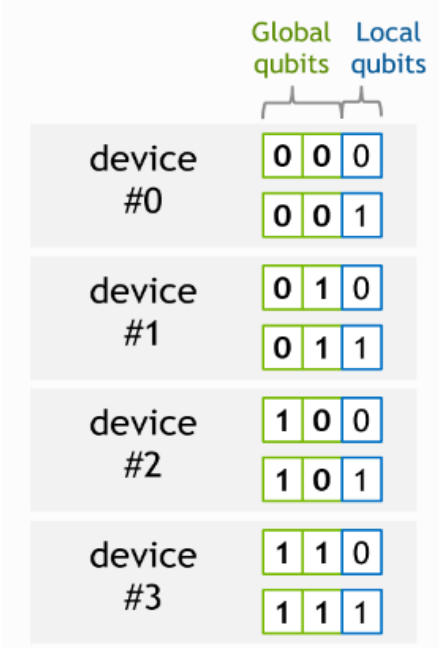
1量子ビット増えると指数関数的にメモリ使用量が増加
- 30qubit以上の場合シングルGPUではメモリ不足

⇩

複数GPUを使用して量子ビットをシミュレート

マルチGPU計算

・複数GPUを使用することで30qubit以上を表現可能
・並列的な演算を複数のGPUに分散することで高速化
・バット処理を適用可能
- 各状態ベクトルのAPI呼び出しを
バッチ処理することで高速化



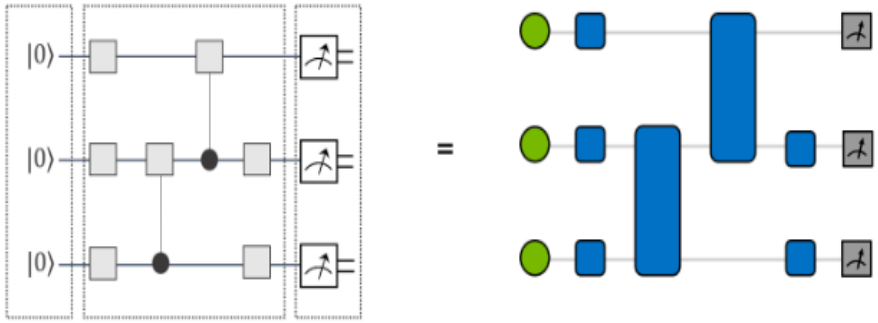
マルチGPU上での状態ベクトルの表現[3]

cuTensorNet

cuTensorNet

・量子回路はテンソルネットワークで表現可能
  - テンソルの縮約・スライス・分解を用いて量子ゲートを表現

・グラフ分割に基づくコスト最適テンソルネットワークの収縮経路探索
  - 縮約コストを削減・GPUでのテンソルネットワーク計算を高速化



量子回路とテンソルネットワークの関係[4]

例）

1量子ビットの初期状態：サイズ2のベクトル
1量子ビットの測定操作：サイズ２のベクトル
N量子ビットゲート：ランク２Nのテンソル

その他DSL

- Tiramisu Compiler
  - ポリヘドラルコンパイラ
  - Halideよりも表現力が高い
    - HalideはRNNなどが書けない
- Exo Lang
  - 決められたPython上の文法で書くと，チューニングを分離可能な言語
    出力は最適化C言語

```
# example.py
from __future__ import annotations
from exo import *

@proc
def example_sgemm(
    M: size,
    N: size,
    K: size,
    C: f32[M, N] @ DRAM,
    A: f32[M, K] @ DRAM,
    B: f32[K, N] @ DRAM,
):
    for i in seq(0, M):
        for j in seq(0, N):
            for k in seq(0, K):
                C[i, j] += A[i, k] * B[k, j]
```

まとめ

- プログラミング方法の違いが，圧倒的な性能差を生み出す
- ハードウェア進化のせいでその差が拡大
- この問題を解決するものがDSL
  - 自動・半自動最適化
    - 演算器の使用，メモリの最適化，並列化
  - ハードウェアデプロイ先の多様化
    - CPU（x86/64, ARM），GPU，FPGA，量子計算機？

Approximate Computing

Approximate Computing

- ある程度のエラーを許容した高性能計算を考える方法
- 画像処理におけるApproximate Computing
  - マルチスケール処理
  - タイリングの冗長性のカット
  - 関数近似
  - 低ラング近似
  - etc…
- アルゴリズム毎に近似高速化方法は存在するが，ここでは，画像処理アルゴリズムに依存しない汎用的な方法を中心に考える

高性能計算のために

- 並列性
- 局所性
- 冗長計算
- 近似計算

これらを考慮したコード生成可能に

- ただし，Halideは，最適化した結果出力が同一になるように最適化
- つまり近似を許容しない
- これらを包括した新たな言語設計が必要

画像処理

基本的

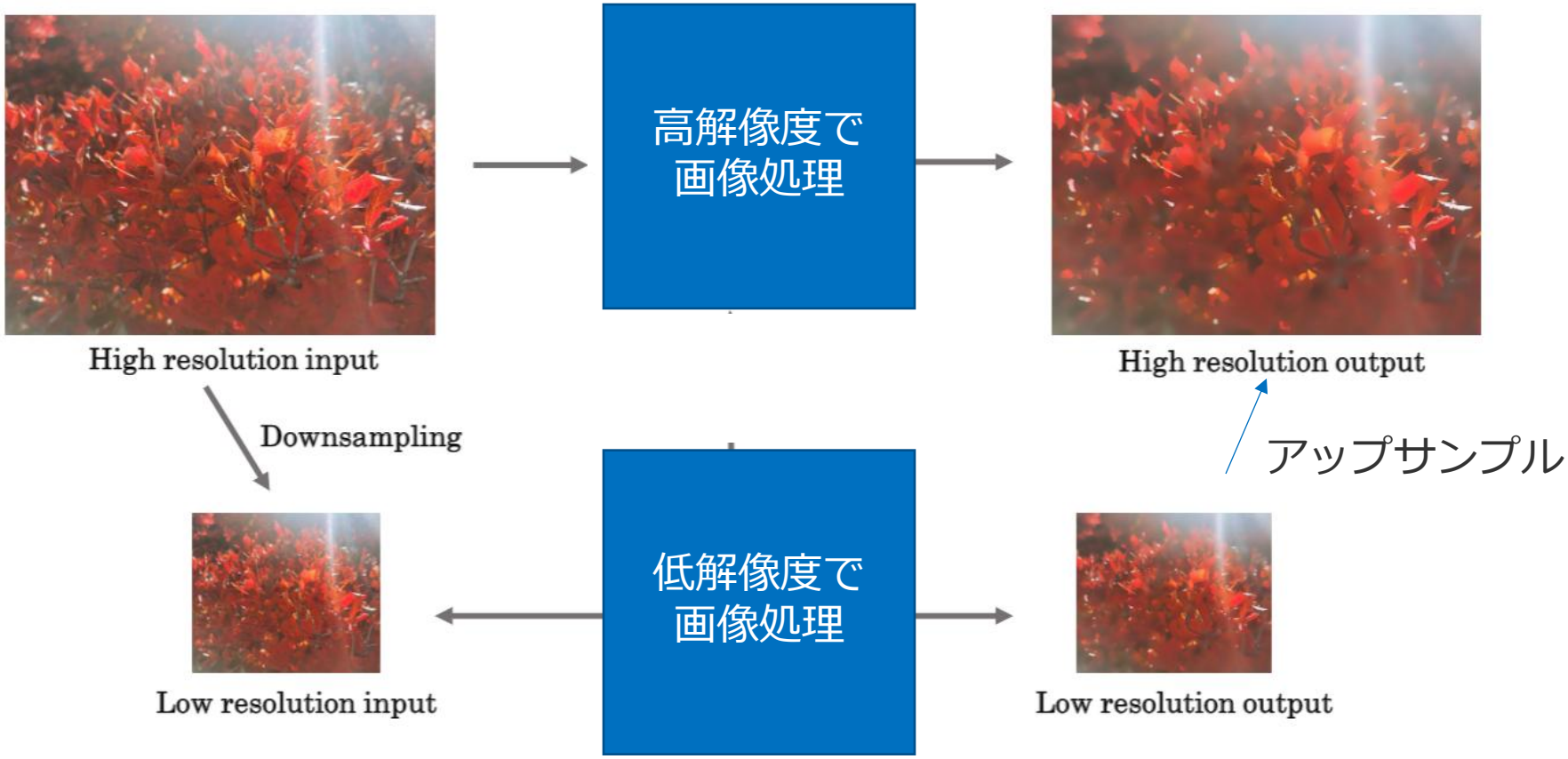
- 点の処理
  - 閾値処理
  - 輝度変換
- エリア処理
  - 畳み込み
  - モルフォロジー
- 統計情報
  - 総和
  - 平均
  - 分散

左の組み合わせで画像処理が設計

個々には高速化の方法は様々

汎用的な方法は？

マルチスケール処理



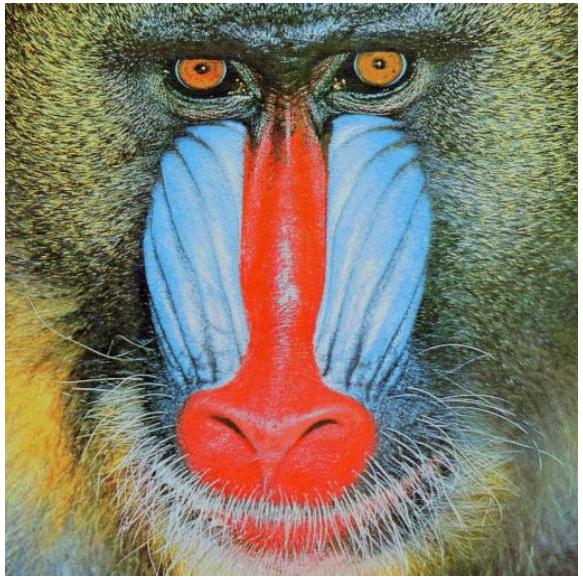
処理する情報を圧縮（リサイズ）することによる高速化

サイズが小さくなり高速化．キャッシュに収まり更に高速化

Joint Upsample

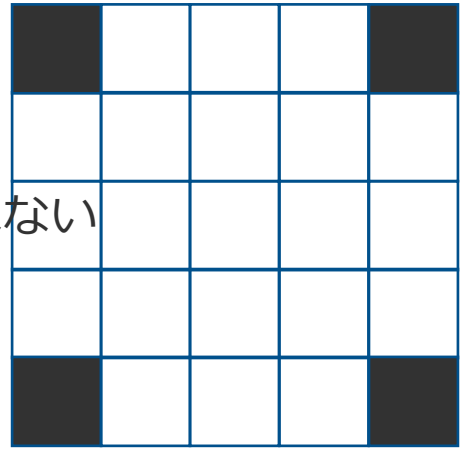
- 高解像度画像の情報を利用したアップサンプル
  - 通常のアップサンプルより高精度
  - 速度も，超解像等よりもかなり高速
- アップサンプル時に高解像度画像のエッジ情報を利用して，輪郭を維持してアップサンプル
  - Joint Bilateral Upsample
  - Guided Image Upsample
  - Bilateral Guided Upsample
  - Local LUT Upsample

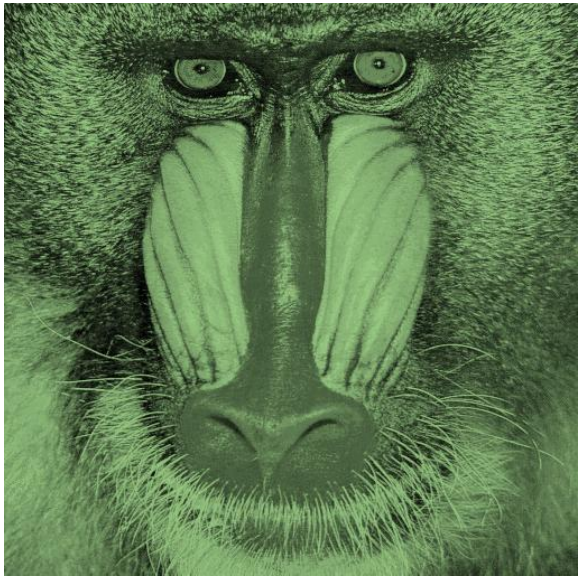
Joint Upsample



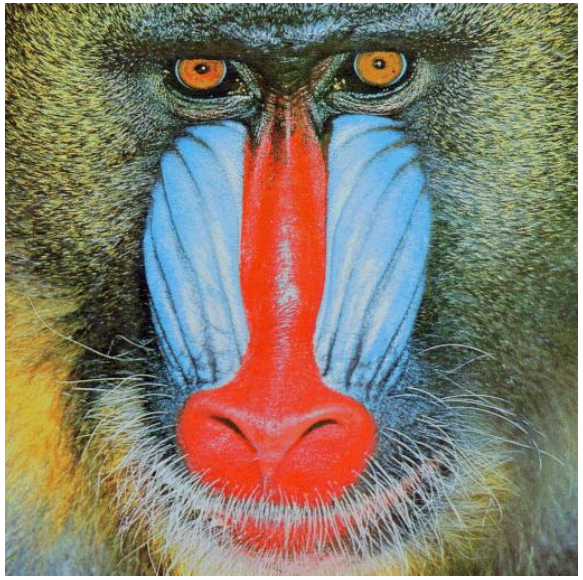
ダウンサンプル
画像処理

空欄の情報はない

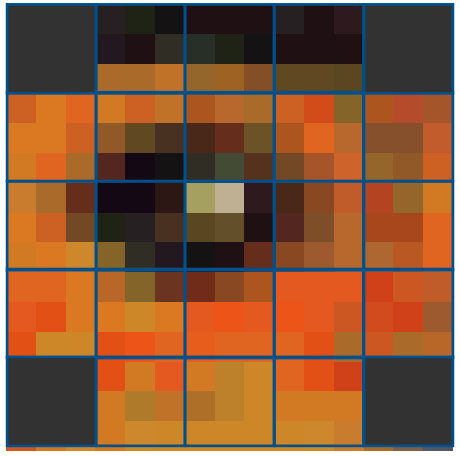


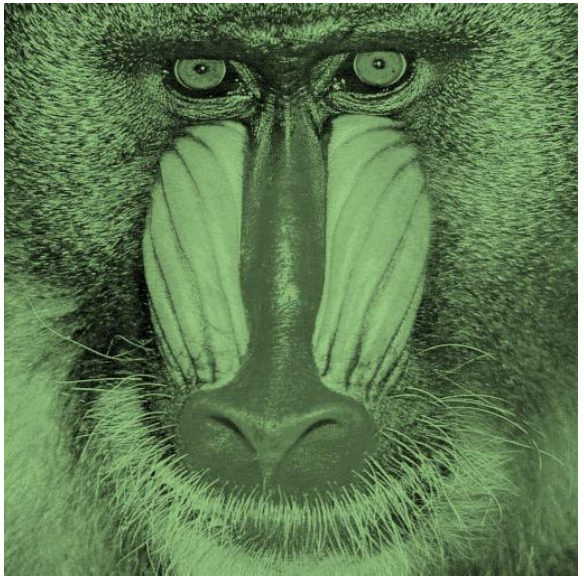


Joint Upsample



ダウンサンプル
画像処理



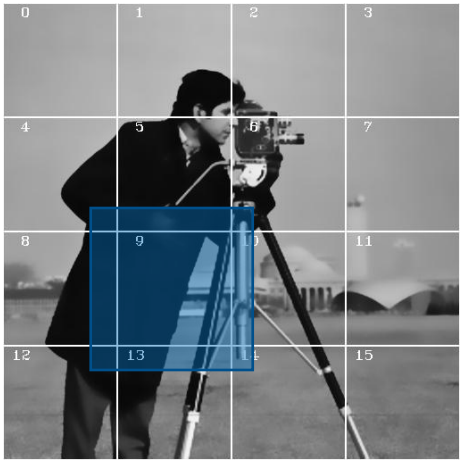


色変換・次元圧縮

- RGBをYとして１チャネルで処理
  - 処理量が1/3
- RGBをR,G,Bとして3チャネルを1チャネルづつとして処理
  - RGBに依存関係がある処理などの高速化
    - $r^2+b^2+g^2$などの計算がY^2で済む
- RGBをPCAで変換して2チャンネルで処理
  - チャネル数を減らす次元圧縮により高速化
    - $r^2+b^2+g^2$などの計算がX^2+Y^2で済む
  - 上記よりは近似量が少なく済む

タイル情報の削減

- 冗長処理領域の削減
- 境界は画像処理結果に影響する量が少ないため近似計算しても影響が少ない

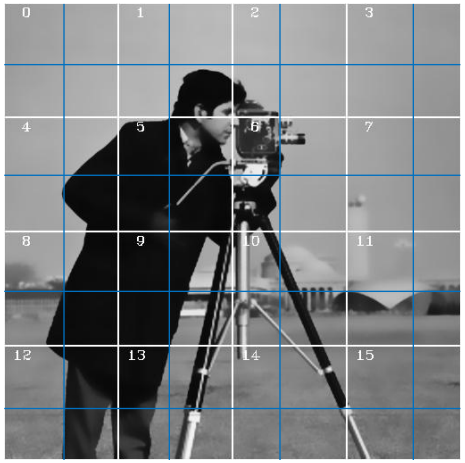


コア数

「多コアは境界領域の冗長範囲が拡大」

- GPU：数百
- CPU：数コア？
  - AMD Threadripper 64core128thread

16分割



256分割

その他

- 量子化・テーブル参照
- 周波数変換による近似
- SVD・EVDによる低ランク近似
- 関数近似
- 再帰フィルタによる計算

符号化と高速化

- 汎用的な高速化

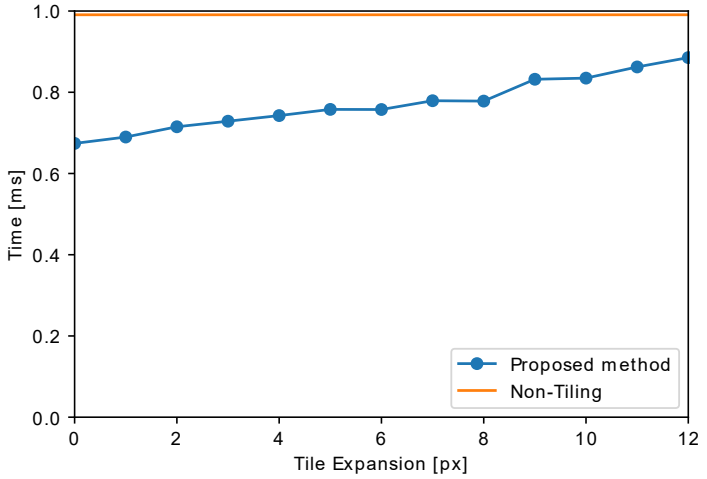
  「処理する領域，計算のまびき・圧縮」

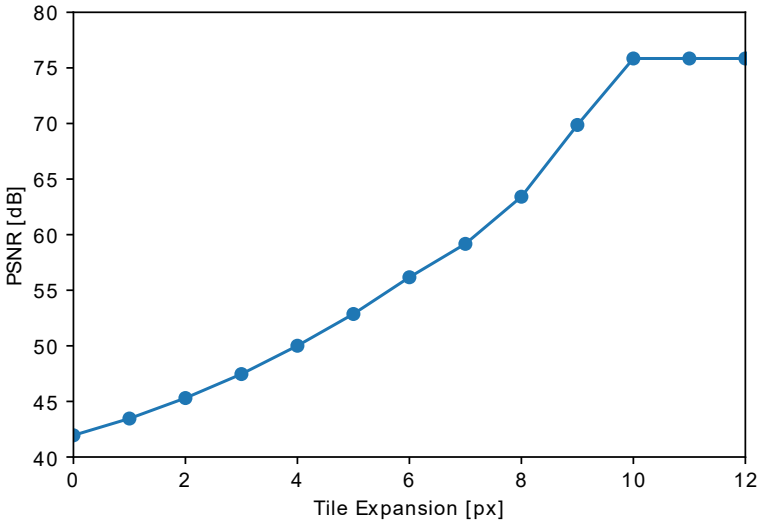
- RD曲線のように近似精度と情報量のトレードオフ
  - 圧縮：近似精度vs圧縮率
  - 高速化：近似精度vs速度
- キャッシュの場合
  - 圧縮：時間的・空間的な相関性を使った削減
  - 高速化：キャッシュの空間的，時間的局所性
- これらを達成するソフトウェア
  - 符号化：エンコーダ
  - 高速化：コンパイラ・プログラミング言語

事例

タイリング領域の削減

- タイルの領域を削減することによる近似高速化
- プログラミング言語Halideを拡張して実現
  - 汎用的なTiling
    - Y. Tsuji and N. Fukushima, "Halide and OpenMP for Generating High-Performance Recursive Filters," in Proc. International Workshop on Advanced Image Technology (IWAIT), Jan. 2020
  - 再帰フィルタ
    - H. Takagi, N. Fukushima, "An Efficient Description with Halide for IIR Gaussian Filter," in Proc. Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA), Dec. 2020





タイリング領域の削減


```
RecFilterDim x("x",image_width), y("y",image_height);
RecFilter F("Gaussian");
F.set_clamped_image_border();

// initialize the IIR pipeline
F(x,y) = image(x,y);

// add the filters: causal and anti-causal
F.add_filter(+x, gaussian_weights(sigma, order));
F.add_filter(-x, gaussian_weights(sigma, order));
F.add_filter(+y, gaussian_weights(sigma, order));
F.add_filter(-y, gaussian_weights(sigma, order));

// specify filtering algorithm
F.algorithm(VYV);
F.set_tol(tol);

// specify the length of redundancy calculation
F.set_redundancy(2*sigma);

// tile the filter
F.split(x, tile_width);
F.split(y, tile_height);

// schedule the filter
F.set_vectorization_width(vector_width)
F.cpu_auto_schedule();

// JIT compile and run
Buffer<float> out(F.realize());
```

アルゴリズム

```
F.compute_root()
  .split(x, xo, xi, split_width)
  .split(y, yo, yi, split_width)
  .reorder(xi, yi, xo, yo)
  .vectorize(xi, vector_width)
  .parallel(xo);
  .parallel(yo);

F_final_causal.compute_at(F, xo)
  .split(y, yo, yi, split_width);
F_final_causal.update(0)
  .split(y, yo, yi, split_width)
  .reorder(rk, yi, xo, yo)
  .vectorize(yi, vector_width)
  .parallel(xo);
  .parallel(yo);
F_final_causal.update(1)
  .split(y, yo, yi, split_width)
  .reorder(rxf, yi, xo, yo)
  .vectorize(yi, vector_width)
  .parallel(xo);
  .parallel(yo);

F_init_causal.compute_at(F_final_causal, xo)
  .split(y, yo, yi, split_width)
  .reorder(rk, yi, xo, yo)
  .vectorize(yi, vector_width)
  .parallel(xo);
  .parallel(yo);
F_init_causal.update(0)
  .split(y, yo, yi, split_width)
  .reorder(rk, yi, xo, yo)
  .vectorize(yi, vector_width)
  .parallel(xo);
  .parallel(yo);
F_init_causal.update(1)
  .split(y, yo, yi, split_width)
  .reorder(rk, yi, xo, yo)
  .vectorize(yi, vector_width)
  .parallel(xo);
  .parallel(yo);
```

スケジュール
抽象化した記述

プログラミング言語Halideでは書けない処理の実現

タイリング領域の削減


```
Buffer<float> iBuffC;
/** calc impulse response for causal scan**/
Buffer<float> iBuffA;
/** calc impulse response for anti causal scan**/
RDom rti(0, iBuffC.height());
Expr rx;

/** k is k=0,...,K **/

Func F_init_causal;
rx = tile * xo - rti;
F_init_causal(xi,xo) = input(xo*tile_width+xi);
F_init_causal(rk,xo) += select(rk>=k,
 iBuffC(0,k-1)*input(xo*tile_width+rk-k+1), 0);
F_init_causal(rk, xo) += select(rk == k,
 sum(iBuffC(k, rti)*input(select(rx<0,-rx,rx))),0);

Func F_final_causal;
F_final_causal(xi, xo) = undef;
F_final_causal(rk, xo) = F_init_causal(rx, xo);
F_final_causal(rxf,xo) =
 sum(ff_c(rk)*input(xo*tile_width+rxf-rk, xo))
 +sum(fb(rk)*F_final(rxf-rk-1, xo));

Func F_init_anti_causal;
rx = tile * xo + tile_width - 1 + rti;
F_init_anti_causal(xi, xo) = input(xo*tile_width+xi);
F_init_anti_causal(tile_width-1-rk, xo) += select(
  rk>=k,iBuffA(0,k-1)*input(xo*tile_width-rk+k-1),0);
F_init_anti_causal(tile_width-1-rk, xo) +=
 select(rk == k, sum(iBuffA(k, rti)*input(rx<length,
 rx, 2*length-rx-1,)), 0);

Func F_final_anti_causal;
F_final_anti_causal(xi, xo) = undef;
F_final_anti_causal(tile_width-1-rk, xo) =
 F_init_anti_causal(tile_width-1-rx, xo);
F_final(tile_wifth-1-rxf,xo) =
 sum(ff_a(rk)*
 input(xo*tile_width+tile_width-1-rxf+rk+1,xo))
 +sum(fb(rk)*F_final(tile_width-1-rxf+rk+1,xo));

Func F;         //ouput
F(x) = F_final_causal(x%tile_width, x/tile_width)
 +F_final_anti_causal(x%tile_width, x/tile_width);
```

Halideはアルゴリズムとスケジュールに分離して記述．ただし処理領域を間引くことができない

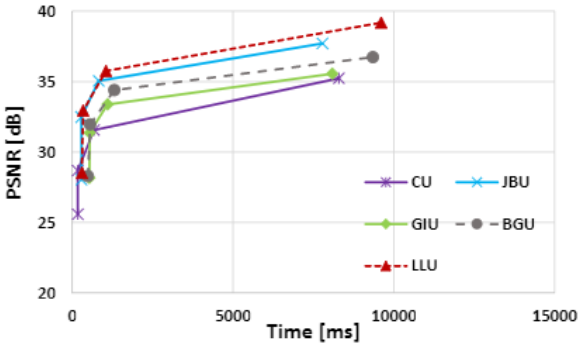
近似計算が入るためHalideのコードは，アルゴリズム・スケジュールが入り混じったコードが出力

Halideのアルゴリズム部にかけばそれなりにどんな処理も書ける

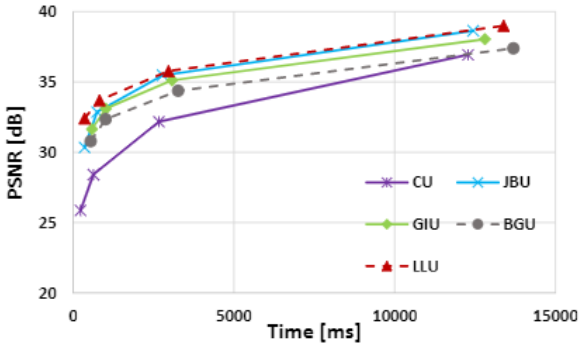
そのHalideコードの自動生成するプログラミング言語・コンパイラの生成

アップサンプルによる高速化

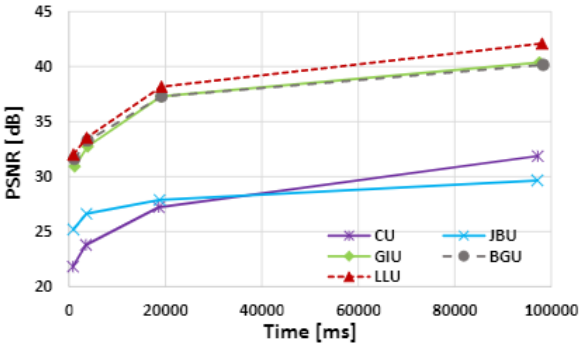
- 間引き率を変えながら，画像処理を低解像度で実行して，アップサンプル
  - H. Tajima, T. Tsubokawa, Y. Maeda, and N. Fukushima, "Fast Local LUT Upsampling," in Proc. International Conference on Computer Vision Theory and Applications (VISAPP), Feb. 2020.
- エッジを考慮したアップサンプルは，単純なCubicに比べてかなり良いRDカーブ



(a) Iterative bilateral filtering

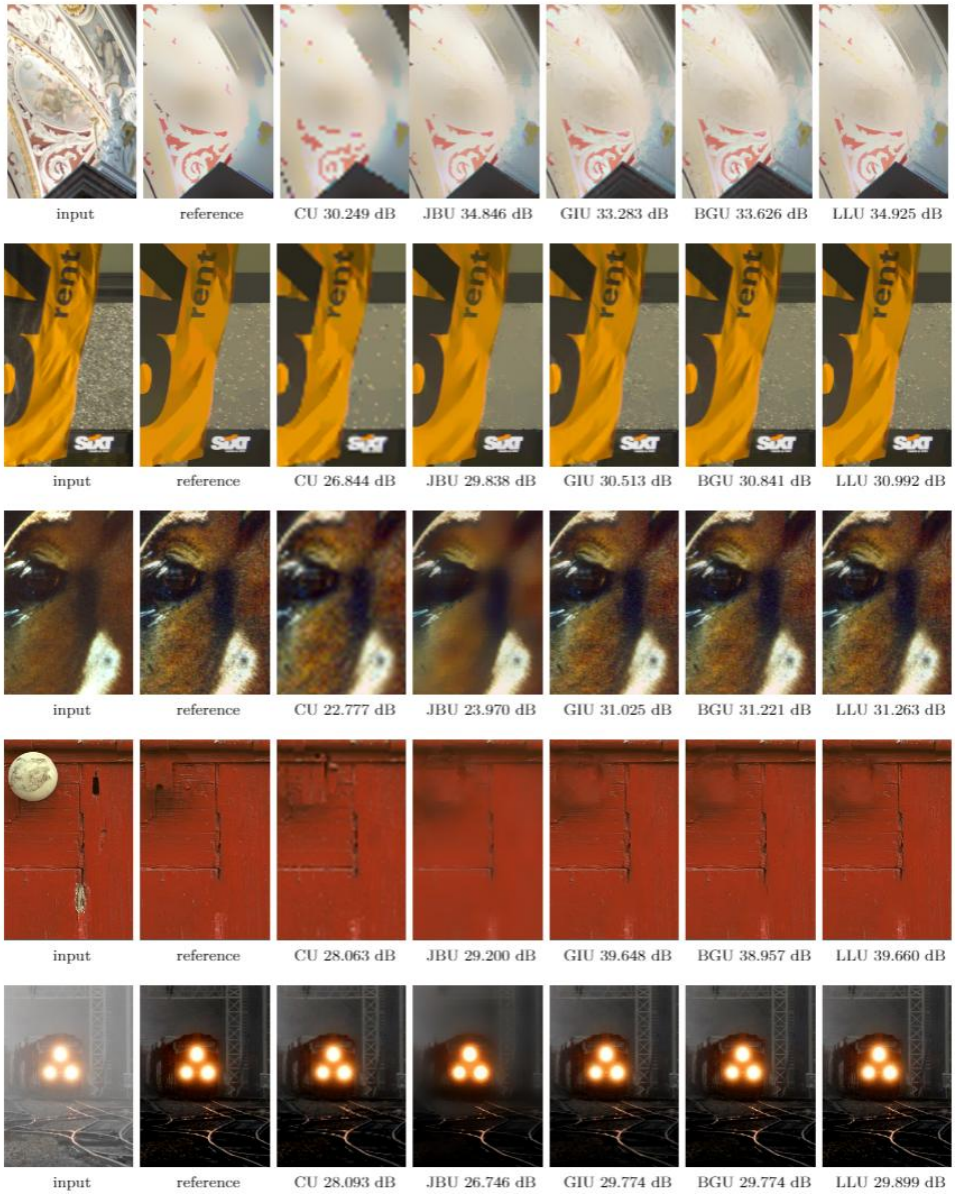


(b) ℓ_0 smoothing



(c) Local Laplacian filtering

アップサンプルによる高速化



- 様々な画像処理を高性能にアップサンプルするアップサンプルアルゴリズム
- プログラミング言語としての設計はまだ

研究の展開

- 高速化の汎用的な方法をプログラミング言語，コンパイラに落とすことで，「Approximated Computing」が可能な画像処理コンパイラの実現
  - アルゴリズムを設計したら，その近似・高速化をコンパイラに任せることが可能に
  - 細かな高速化をコンパイラに任せることで，本質の設計に集中
  - 性能のトレードオフを取る時間の大幅な削減
  - loss関数（PSNRやSSIM）を考慮した計算フローの最適化
  - 符号化のような汎用的なツール群をコンパイラ・プログラミング言語として落としていく
    - 最近の深層学習フレームワークは，深層学習特化型のApproximated Computingコンパイラ