スライドは以下のタイトルで検索してもらえれば，資料見れると思います．

計算機アーキテクチャを考慮した高能率画像処理プログラミング

名古屋工業大学

福嶋 慶繁

経歴

- 平成12年(2000) 3月
  - 岐阜県立岐阜高等学校 卒業
- 平成16年(2004) 3月
  - 名古屋大学 工学部 電気電子情報工学科 （電気電子コース）卒業（村瀬研究室）
- 平成18年(2006) 3月
  - 名古屋大学大学院 工学研究科 電子情報システム専攻 博士課程前期課程 修了（谷本研究室）現藤井研
- 平成21年(2009) 3月
  - 名古屋大学大学院 工学研究科 電子情報システム専攻 博士課程後期課程 修了（谷本研究室）
- 平成21年(2009) 4月
  - 名古屋工業大学大学院 工学研究科 創成シミュレーション工学専攻/工学部 情報工学科 助教
- 平成27年(2015) 4月
  - 名古屋工業大学大学院 工学研究科 創成シミュレーション工学専攻/工学部 情報工学科 准教授
- 平成28年(2016) 4月～
  - 名古屋工業大学大学院 工学研究科 情報工学専攻/工学部 情報工学科・創造工学教育課程 准教授

高能率画像処理プログラミングに重要なこと

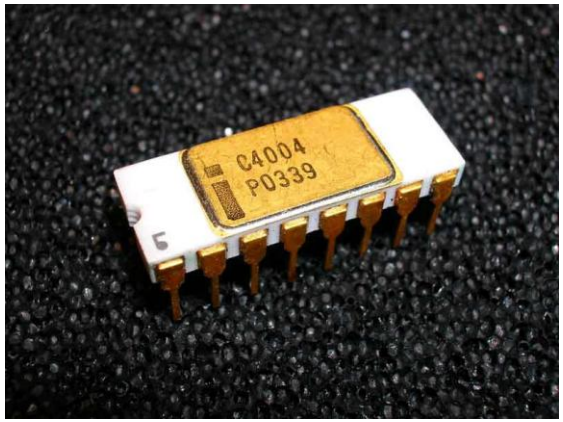
- アルゴリズム
  - 演算量
  - 演算順序と冗長処理
- ハードウェア
  - 並列・ベクトル処理
  - ローカリティ（キャッシュ利用率）

目次

- 計算機アーキテクチャの遍歴
- 高性能プログラミング
  - アムダールの法則
  - ルーフラインモデル
- 並列画像処理プログラミング
- ドメイン固有言語
- 実例

背景（1/4）

- 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



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/

背景（2/4）

- クロック数の向上は2004年ごろからほぼ頭打ち（Pentium 4からほぼ同じ）
- 増加したトランジスタは様々な用途に
  - マルチコア
  - 同時マルチスレッド（ハイパースレッディング）
  - ベクトル演算器（SSE，AVX，AVX512）
  - キャッシュの巨大化
  - ターボブースト

背景（3/4）

- 現在のコンシューマ向け最上位 Intel Core i9 7980XEは2.6 GHz，18コア，512ビットベクトル演算器，FMAユニット２つ搭載
  - 単純：2.6 GFLOPS
  - 理想：3.0 TFLOPS （倍精度：1.5TFLOPS）
    - 2.6*18*16*2*2
    - ターボブーストや増加したL2キャッシュ，SMTを活用するとさらに性能向上
- **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

背景（3/4）

- 現在のコンシューマ向け最上位 Intel Core i9 9980XEは3.0 GHz，18コア，512ビットベクトル演算器，FMAユニット２つ搭載
  - 単純：3.0 GFLOPS
  - 理想：3.5 TFLOPS （倍精度：1.8TFLOPS）
    - 3.0*18*16*2*2
    - ターボブーストや増加したL2キャッシュ，SMTを活用するとさらに性能向上
- **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

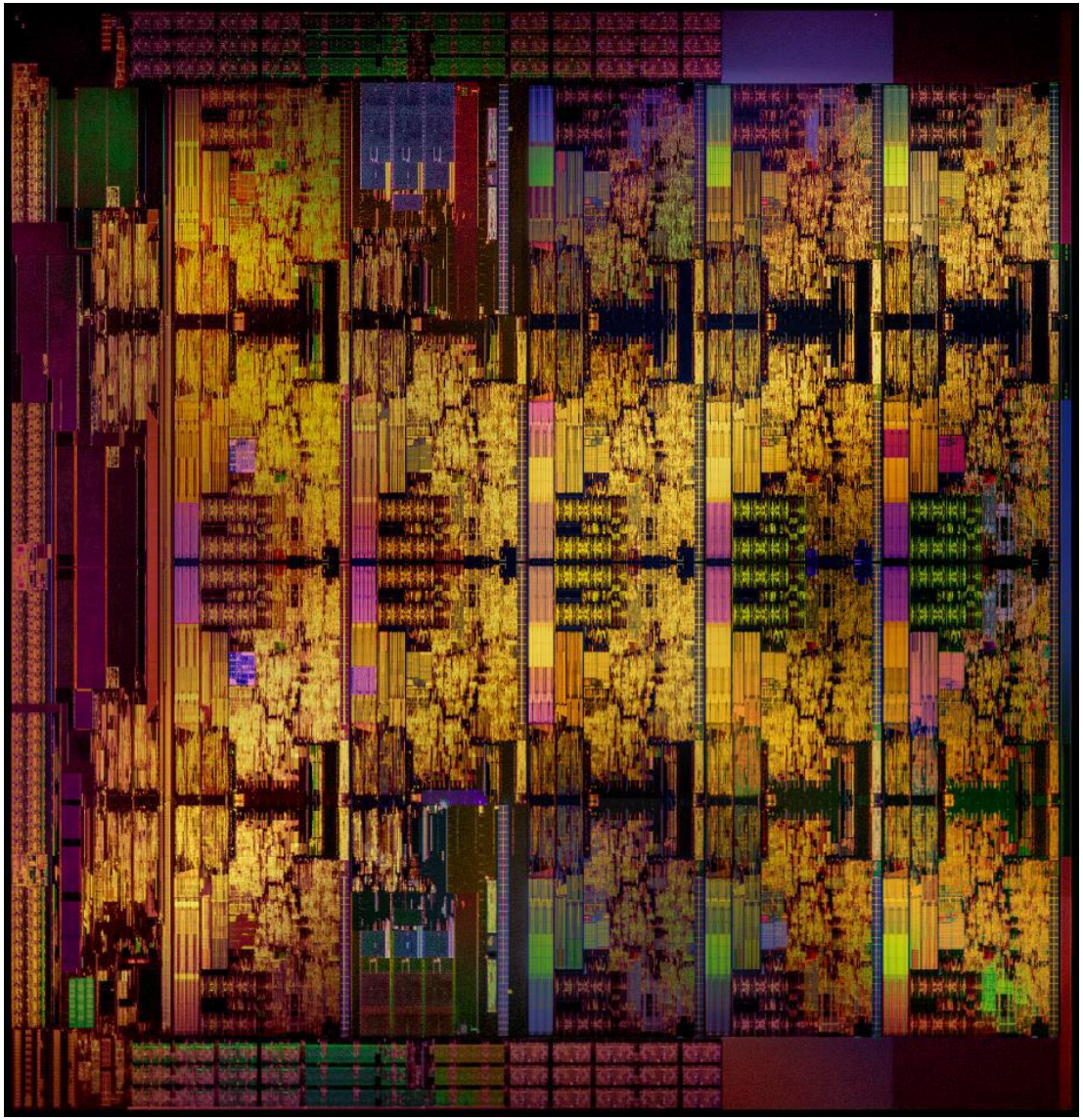
計算機の並列性（CPU）

- 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

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

Core i9 7980XE



16 Core

背景（4/4）

- 増えたトランジスタを有効に活用しなければ，演算効率が高めらない

「フリーランチの終焉」*

- 計算機アーキテクチャをより深く知り，それを活用したプログラミングが必須

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

発表概要

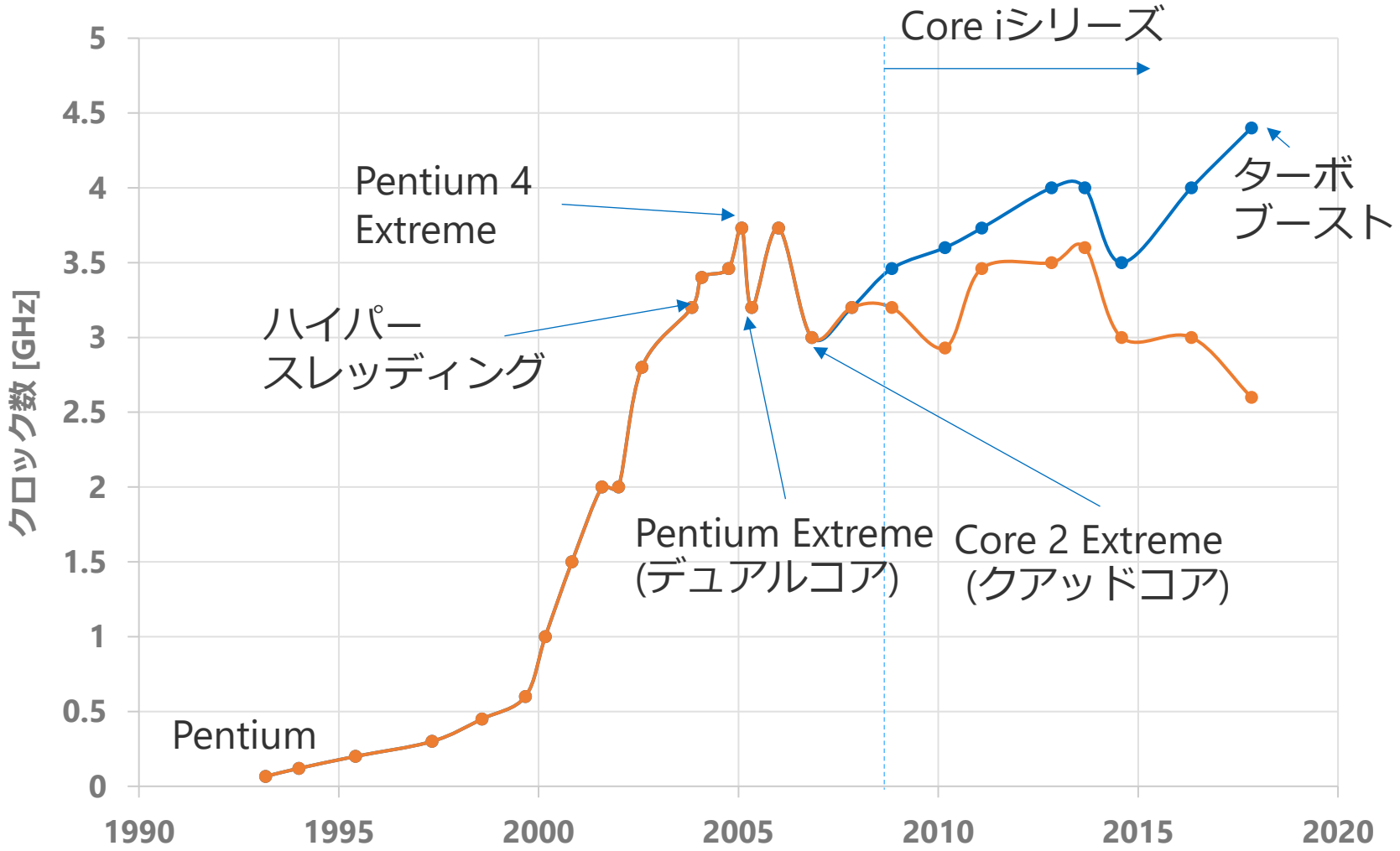
- 画像処理の高能率プログラミングについて
- 特にCPU最適化に特化して紹介

コンシューマ向け計算機の遍歴

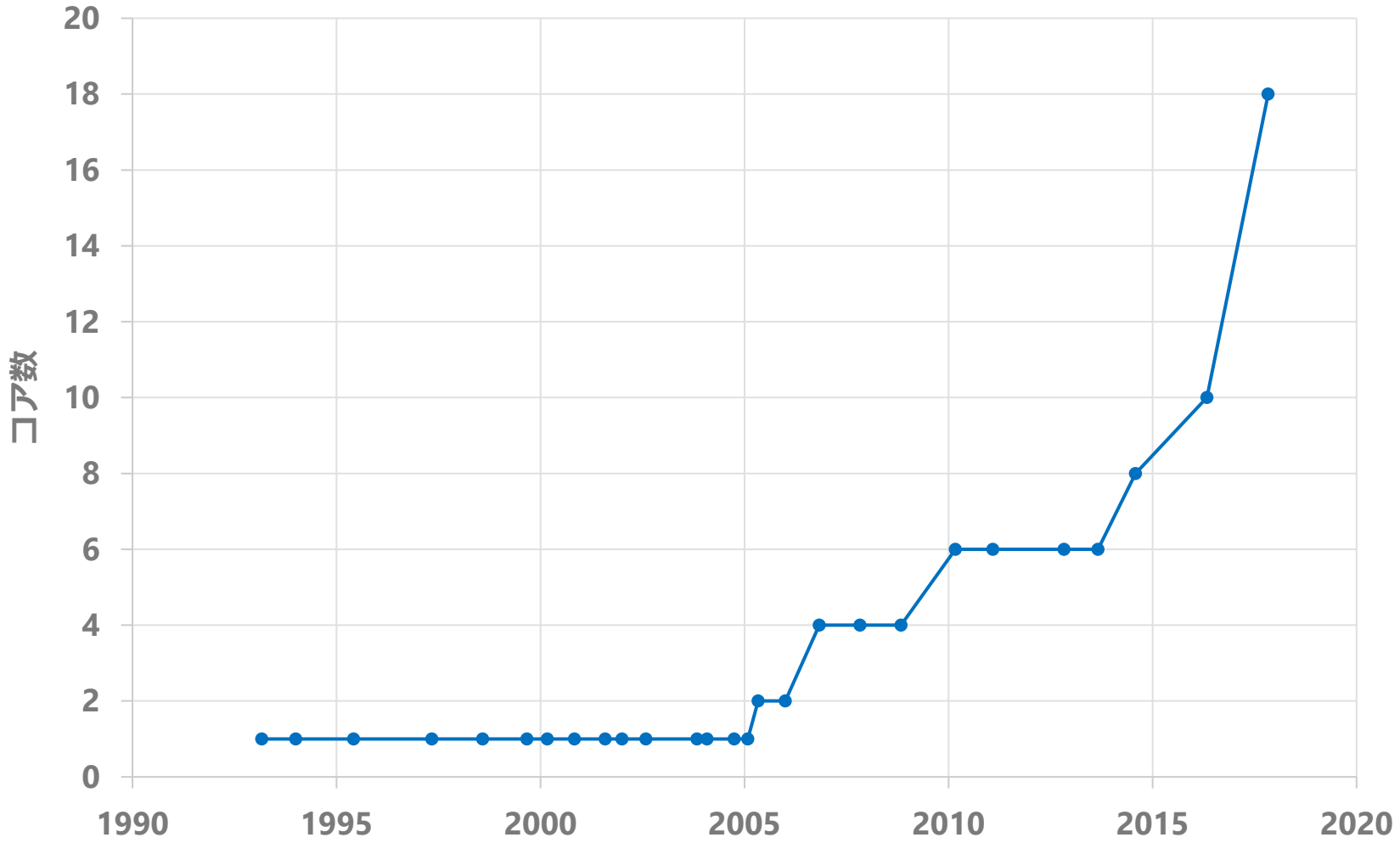
Intel CPUの歴史

年代	名称
1971	4004
…	…
1993	Pentium
1997	Pentium II
1999	Pentium III
2000	Pentium 4
2005	Pentium D
2006	Core 2
2008.11	Core i7（第１世代，Nehalem）
2011.1	Core i7（第２世代，Sandy Bridge）
2012.4	Core i7（第３世代，Ivy Bridge）
2013.6	Core i7（第４世代，Haswell）
2014.9	Core i7（第５世代，Broadwell）
2015.8	Core i7（第６世代，Skylake）
2016.8	Core i7（第７世代，Kabylake）
2017.10	Core i7（第８世代，Coffelake）

クロック数


Core iシリーズ
Pentium 4
Extreme
ターボ
ブースト
ハイパー
スレッディング
Pentium Extreme
(デュアルコア)
Core 2 Extreme
(クアッドコア)
Pentium
クロック数 [GHz]
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
1990
1995
2000
2005
2010
2015
2020

コア数



同時マルチスレッド

- Simultaneous Multithreading (SMT)
  - インテル：ハイパースレッディング
- CPUの様々な命令パイプラインが同時に使われることが少ないことを利用して，空いた演算器を仮想的にもう一つの物理演算器として動作させる処理
- わずかに数%の領域を追加するだけで10～30%の性能向上
  - キャッシュを共有するため，キャッシュのスラッシングが起きてパフォーマンスが低下することもある

マルチCPU，複数コア，SMT

- マルチコア
  - 現代のデフォルト
- 複数CPU
  - Xeonなどのサーバー用． ２～８個のCPUが複数ささる
- SMT
  - 仮想的にCPUの数が２倍に見える
- １つのCPUに２つのCPUが入っている場合
  - Pentium DやAMD Threadripper
  - CPUが巨大化

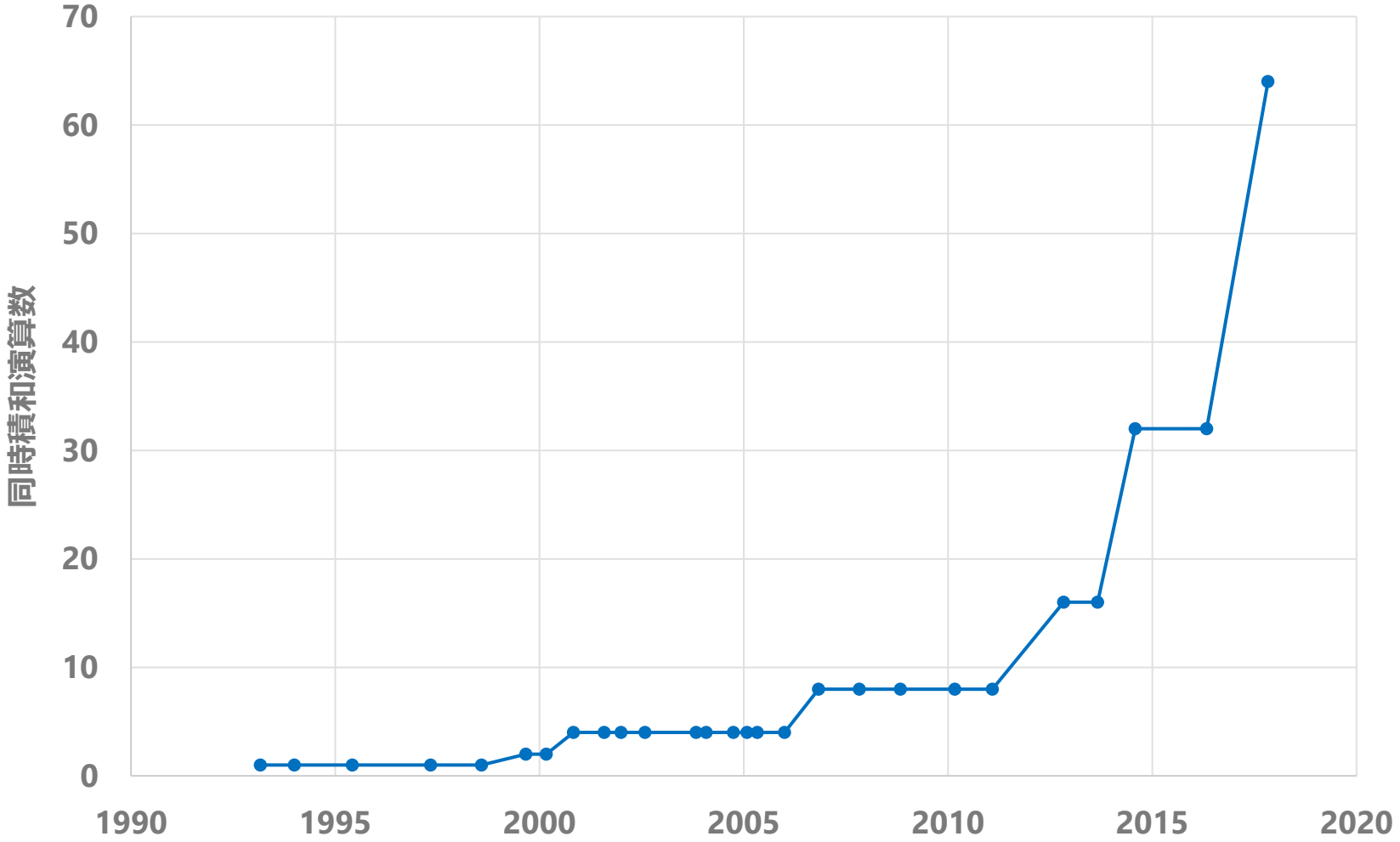
ベクトル演算

- 通常，１個もしくは１つづつで２個のデータに命令を発行して計算
- ベクトル演算はベクトル長だけのデータ同士（例えば４個）に同時に命令を発行
- １命令で複数のデータが処理できるため並列演算可能

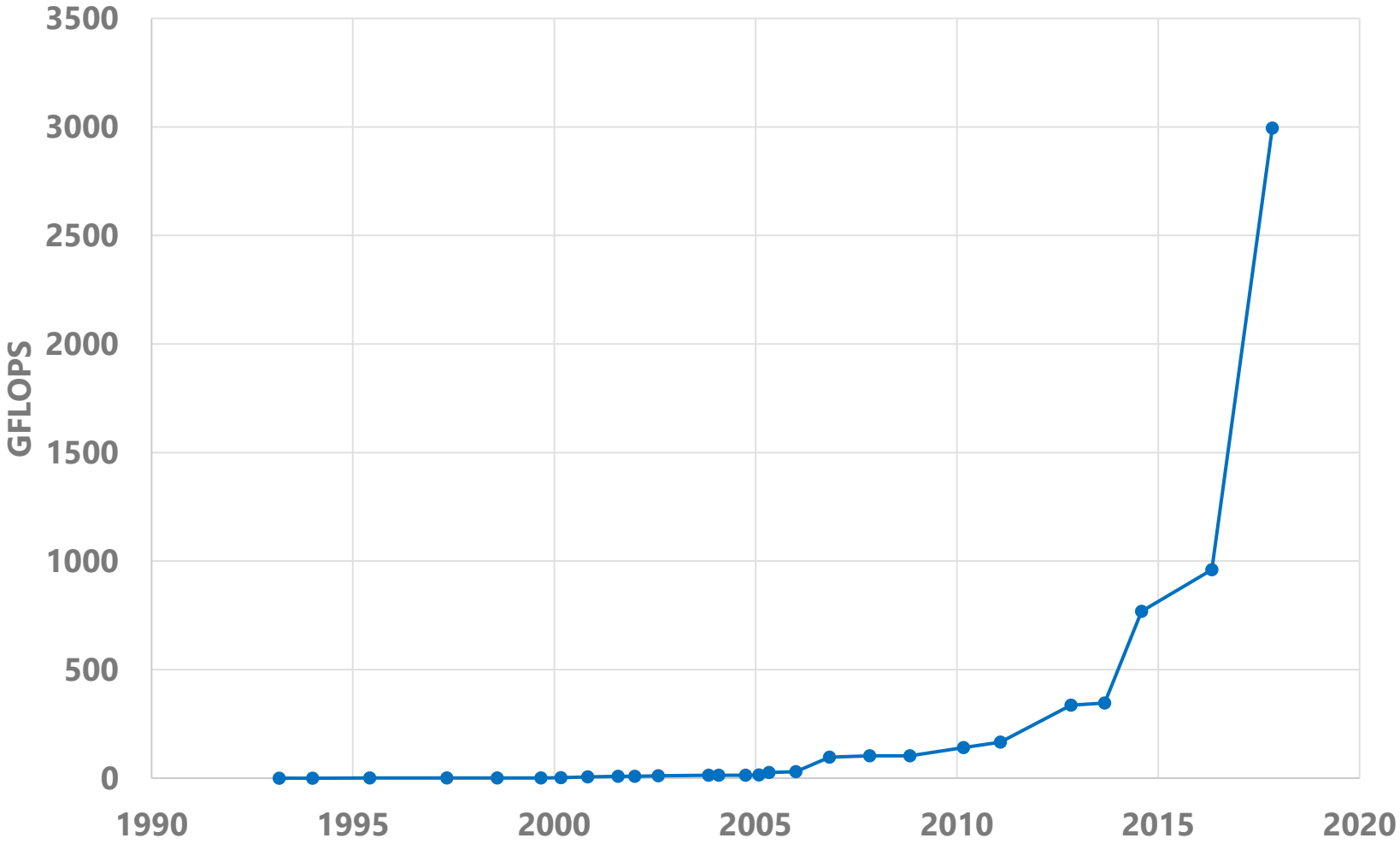
ベクトル演算の種類

- データのロード，ストア
- データの並び替え
- 四則演算，max/min，sqrt，比較，四捨五入，切り上げ切り捨て，ビット演算など
- FMA（fused multiply-add) (2014)
  - 掛け算と足し算ax+bを1命令で実行
- ベクトルアドレッシング
  - gather (2014)
  - scatter (2017)

同時積和演算数（ベクトル演算）



FLOPS


GFLOPS
3500
3000
2500
2000
1500
1000
500
0
1990
1995
2000
2005
2010
2015
2020

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
PFN（プライベートスパコン）	4.7 PFLOPS
暁光	19.14 PFLOPS
神威・太湖之光	93.02 PFLOPS
GPU: GeForce GTX 1080	8.872 TFLOPS
FPGA: Stratix 10	10 TFLOPS
CPU: Intel Core i9 7980XE	3.0 TFLOPS
Xeon Platinum 8170 x8 CPU	28 TFLOPS

Intel CPUの種別

- 省電力
  - Atom
- 廉価
  - Core i3, Pentium, Celeron
  - ターボブースト無,
  - ハイパースレッディングあったりなかったり
  - コア数が少ない
- ミドル
  - Core i5
  - ターボブースト有,
  - ハイパースレッディング無し
- 上位
  - Core i7
  - ターボブースト有,
  - ハイパースレッディング有り
  - コア数多い
- 最上位
  - Core i7/9 X
  - 名前にエキストリームがついている
  - Core i7の最上位スペック
  - メモリバンド幅が2倍

Intel CPUの種別（サーバー）

- Xeon
  - Xeon E3
    - ソケット数１
    - デュアルチャネル
  - Xeon E5
    - ソケット数１～４
    - クアッドチャネル
  - Xeon E7
    - ソケット数４～８
    - クアッドチャネル
- Xeon Scalable Processor
  - Platinum
    - ソケット数２～８
    - ~28Core
  - Gold
    - ソケット数２～４
    - ~22 Core
  - Silver
    - ~12 core
  - Bronze
    - ~8 core
- メモリが６チャネル

※XeonはECCメモリ対応

DRAM

- ダイナミック・ランダム・アクセス・メモリ
- CPU
  - SDRAM
  - DDR SDRAM， DDR2， DDR3， DDR4
  - Core i9 7980XE
    - DDR4 2666 x クアッドチャネル： 85.3 GB/s
  - Xeon Platinum
    - DDR4 2666 x オクタチャネル：128 GB/s
- GPU
  - GDDR5：336 GB/ s （Taitan X）
  - GDDR5X：320GB/s （GTX 1080）
  - HBM2：720 GB/s （Tesla P100）

DDRとGDDR

- GDDR
  - 速い
  - メモリ容量小
    - GPUに直結しないといけない
- DDR
  - 遅い
  - メモリ容量大（GPUの100倍も可能）

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

つまり

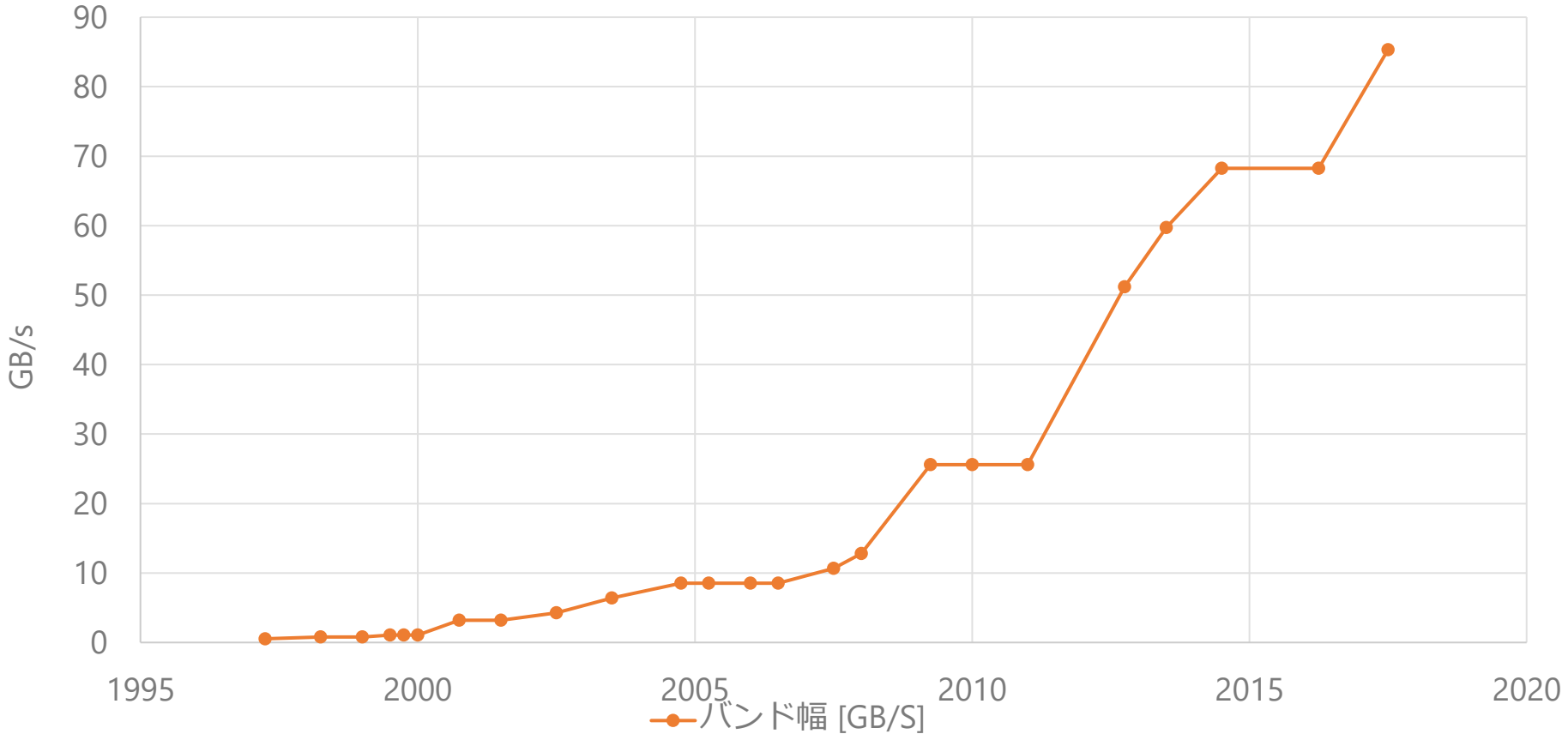
1333 x 2 x 8 x 4=85.3 GB/s

フロントサイドバス（FSB）

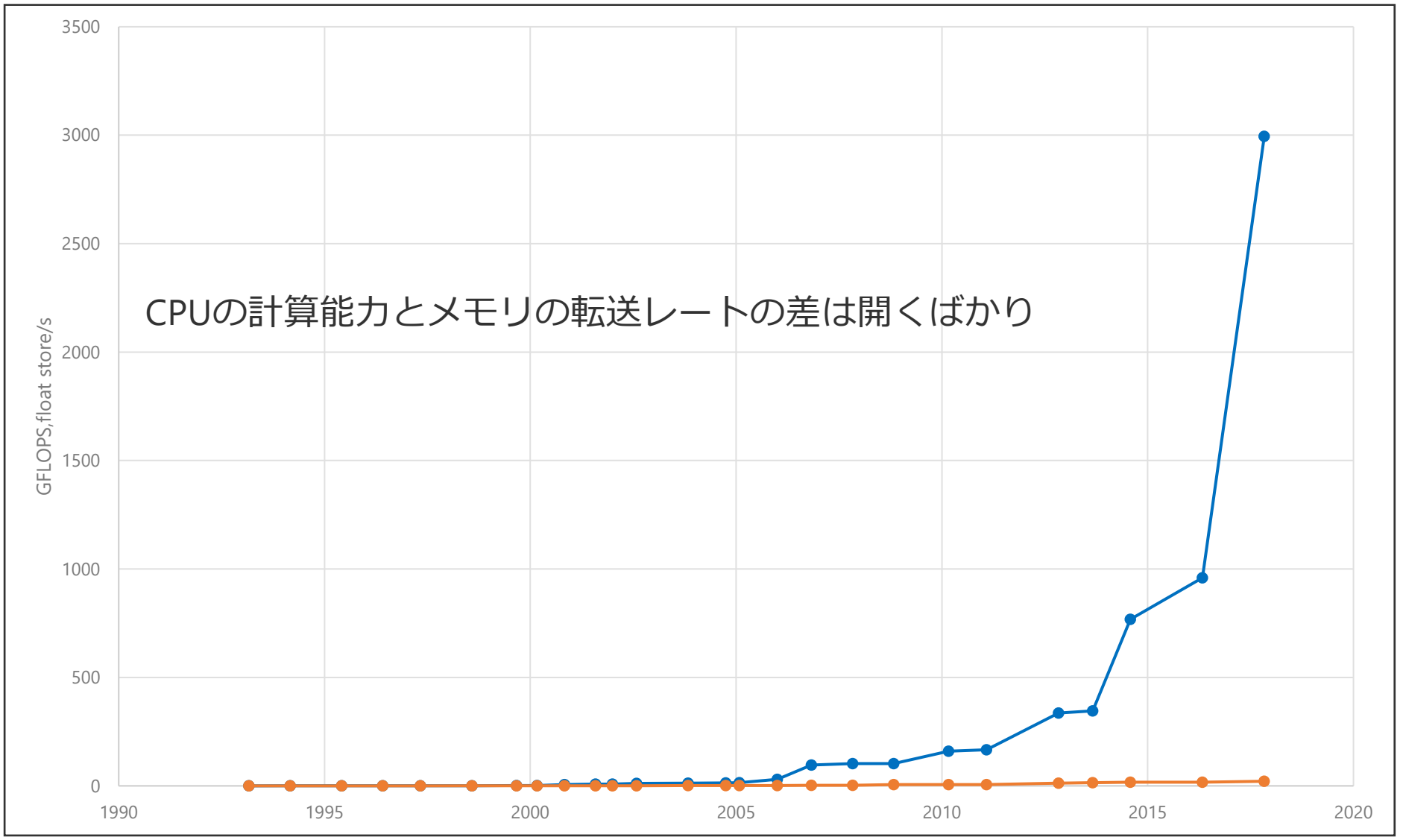
- CPUバスの一種
- おおむねメモリとCPUが通信するための帯域を表す
- 昔のPCはメモリ帯域よりもFSBで律速していることのほうが多かった

- Core iシリーズから廃止
- メモリのバンド幅がそのままメモリ帯域

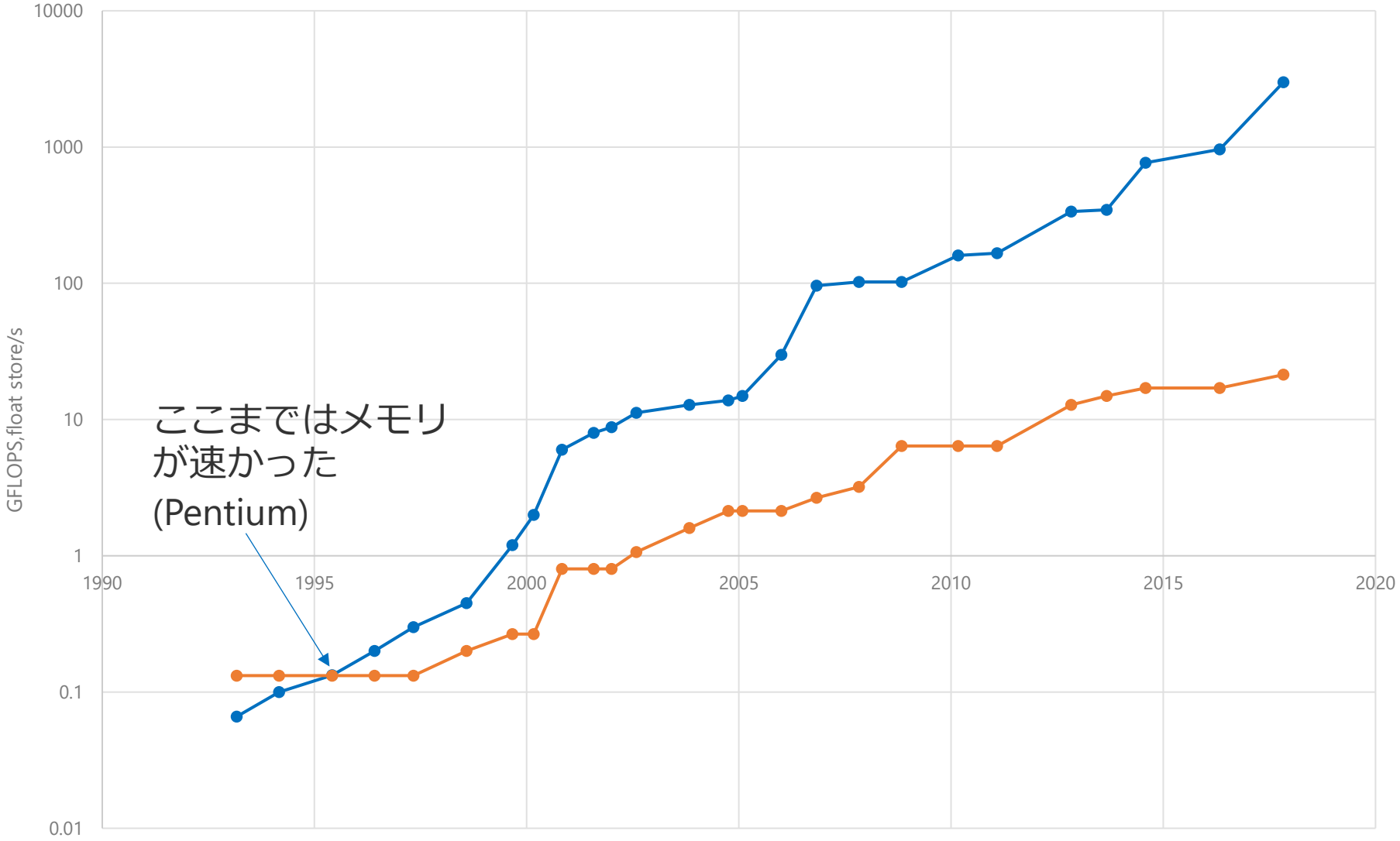
メモリ帯域


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

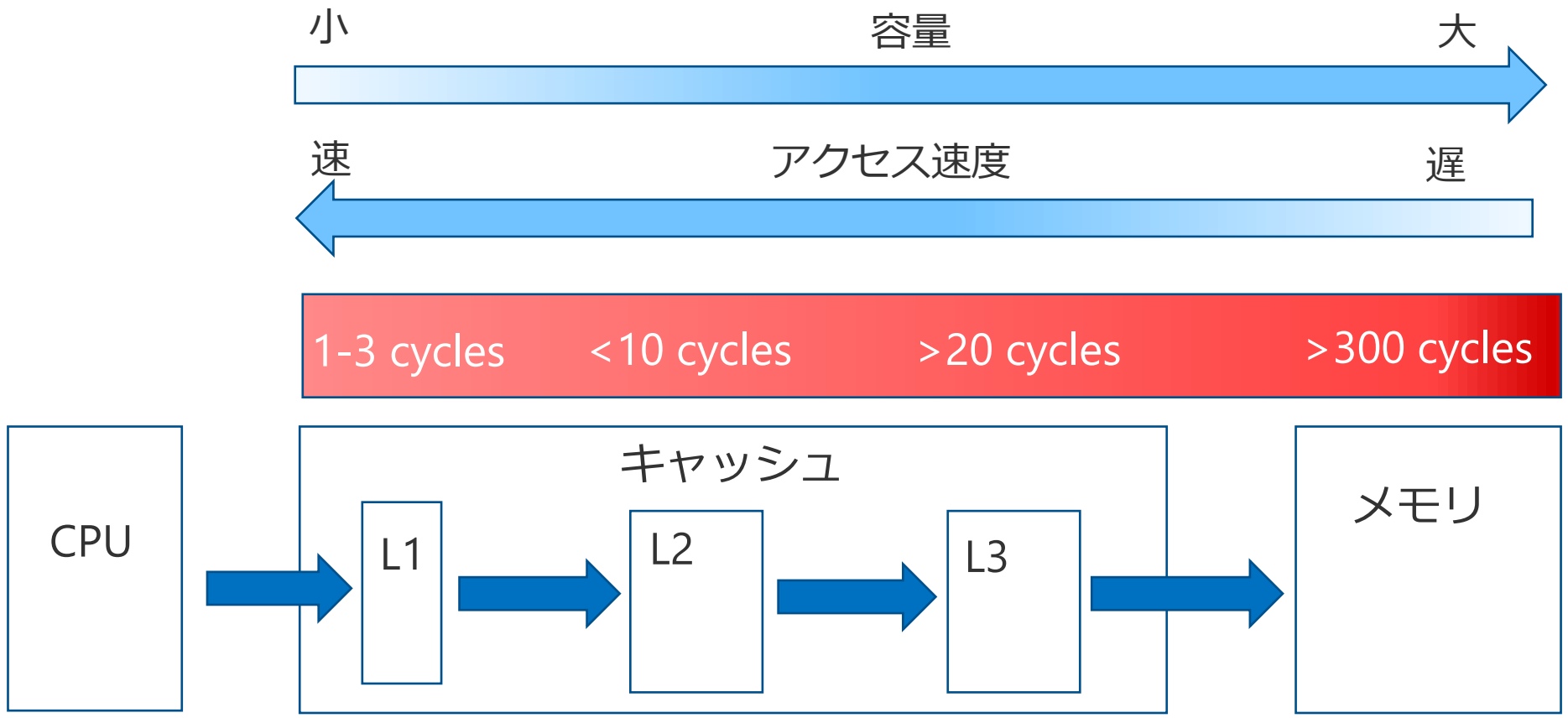


FLOPS vs float 転送レート（対数）33

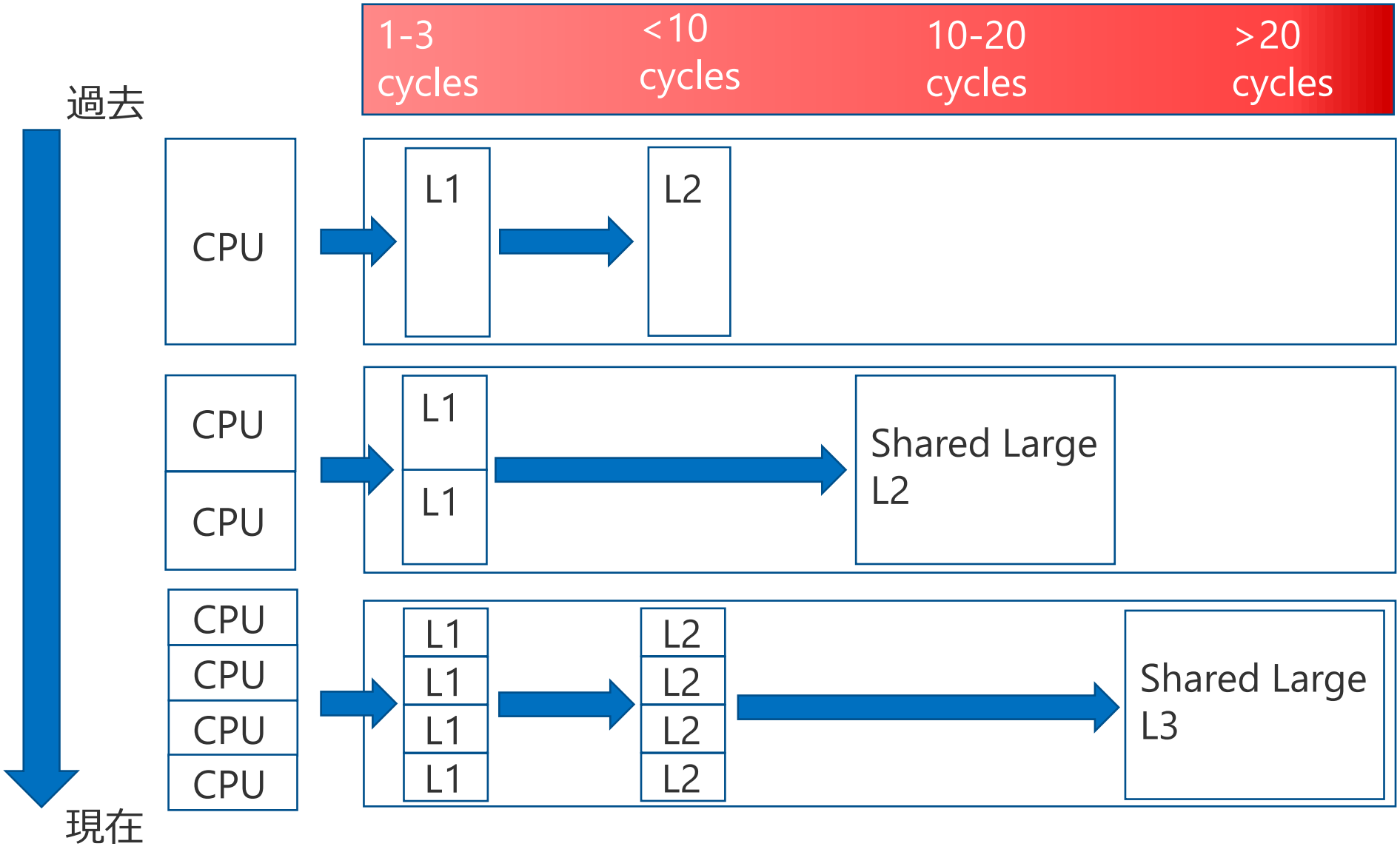


キャッシュ階層とレイテンシ

- 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

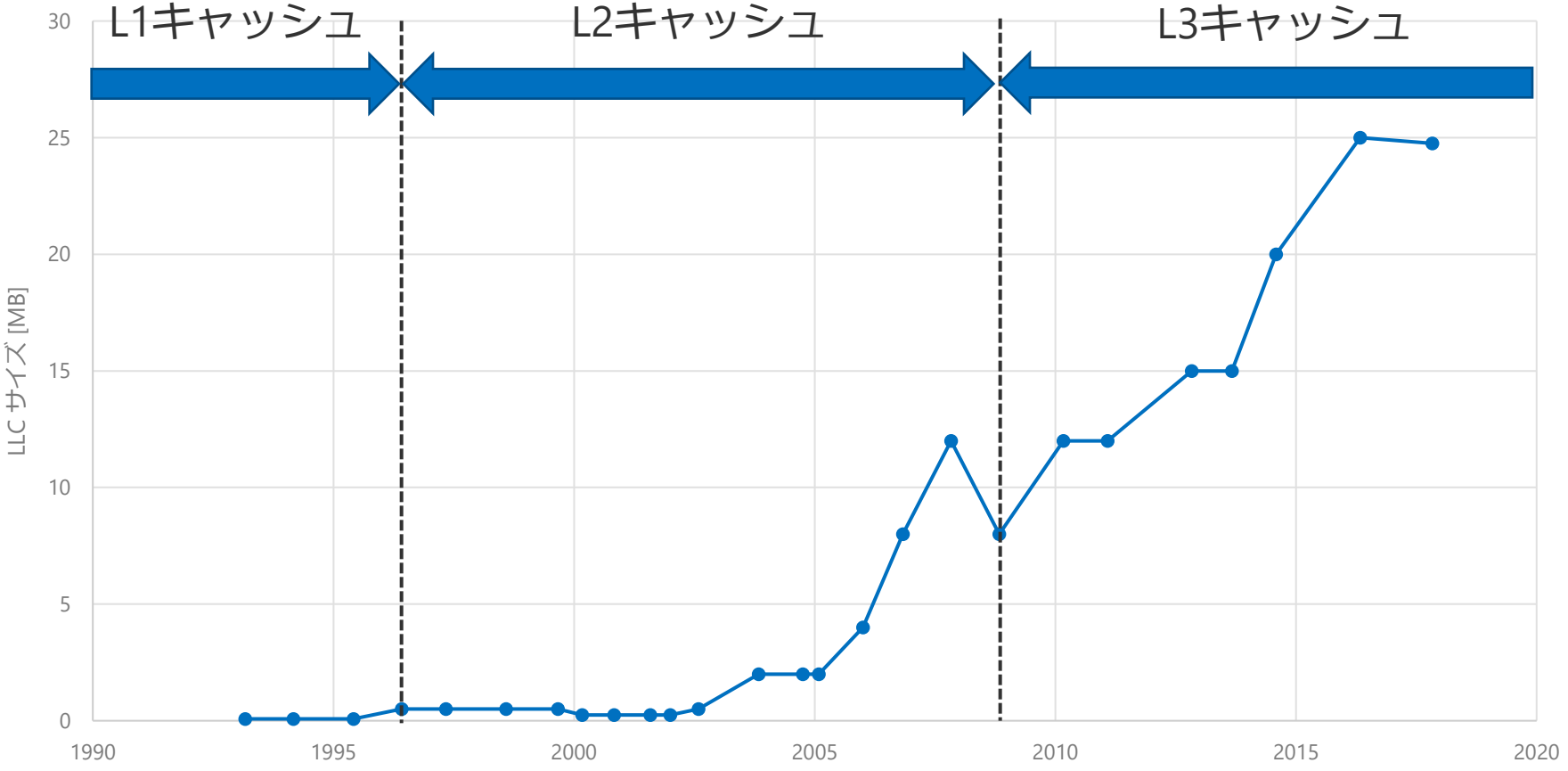


キャッシュ階層とレイテンシ



ラストレベルキャッシュ

- キャッシュの巨大化とともにレイテンシが増えて，L2キャッシュを挟むようになった



画像処理プログラミングで重要なこと

- 並列化
  - 並列にプログラムを実行する
- ベクトル化
  - ベクトル演算器を活用する
- メモリアクセスの抑制
  - L1～L3キャッシュを使う

並列化プログラミング

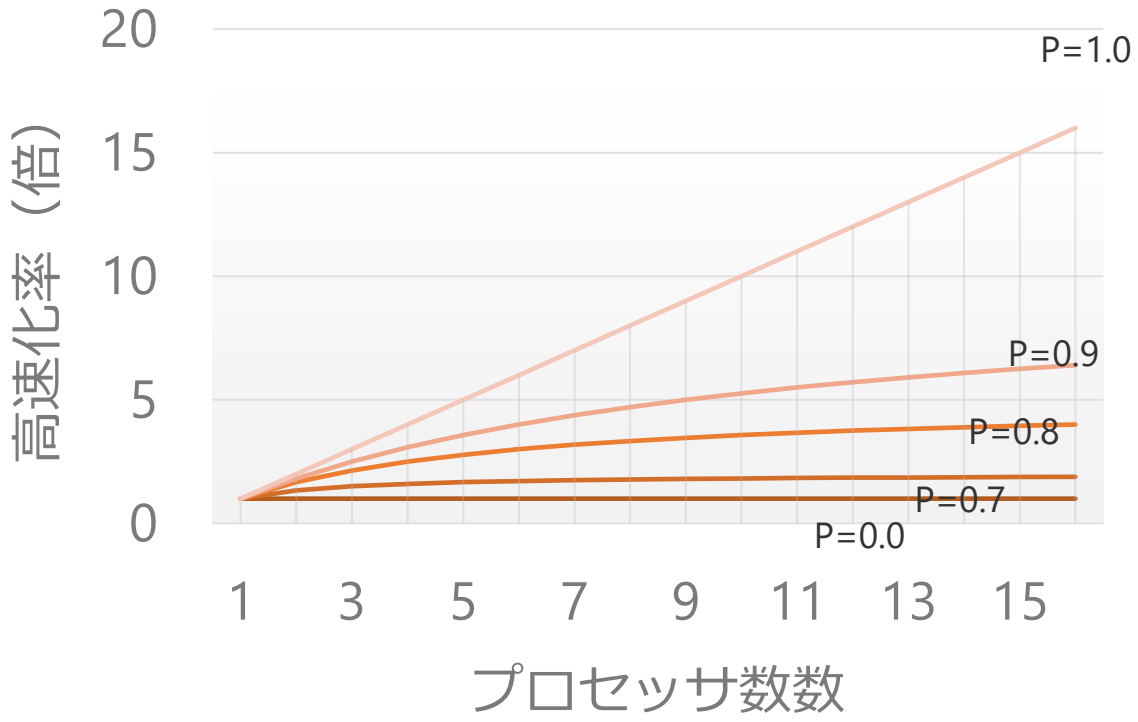
並列化処理の基本

- アムダールの法則
- フリンの分類
- 並列化・ベクトル化プログラミング

アムダールの法則

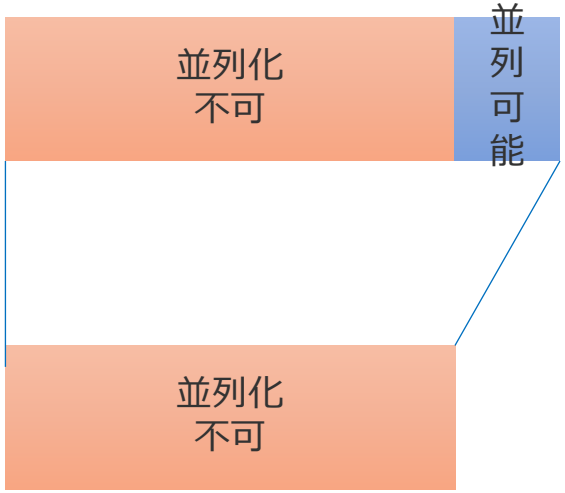
$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

S：高速化率
P：並列化率
N：プロセッサ数

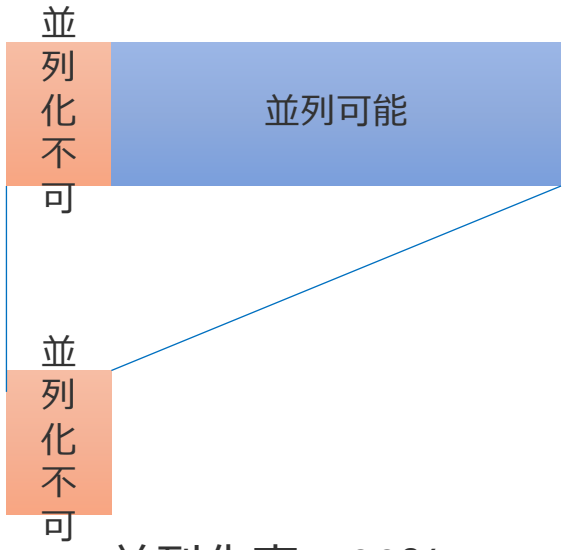


並列化の限界

並列化率：並列化可能な部分



並列化率　１０％



並列化率　90%

たとえ無限個のコアで並列化しても
並列化不可の部分は高速化不可能

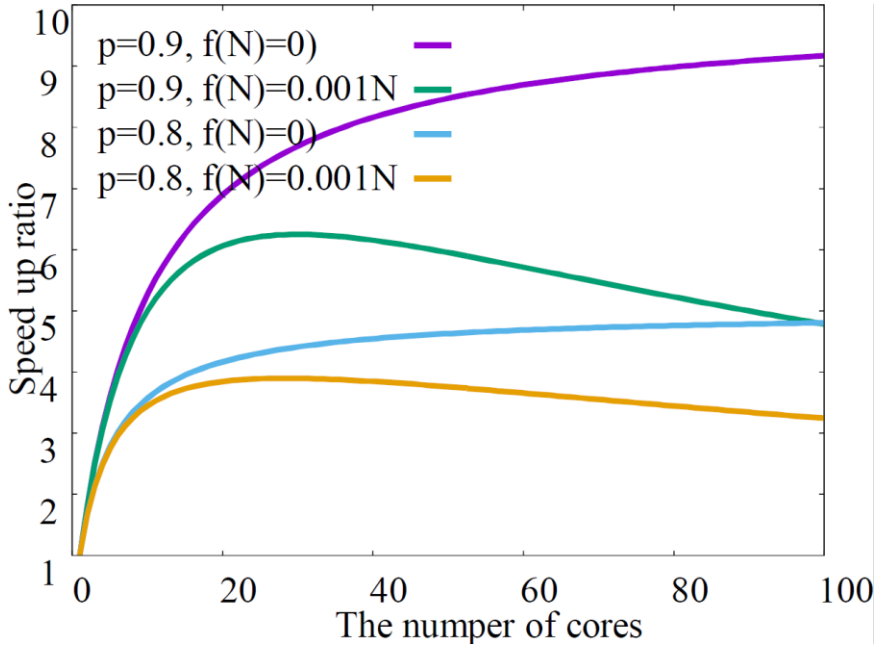
粒度と過分割

- 粒度
  - 分割された処理の大きさ
- 並列オーバーヘッド
  - 分割するほど分割・統合するための**オーバーヘッド**が増加
    1. 共有メモリのロック
    2. データの分配（データのコピーは時間がかかる）
    3. スレッドの生成，起動，同期
  - 粒度が大きいほどオーバーヘッドは隠蔽可能
  - 分割するほど負荷分散（大数の法則）

アムダールの法則（オーバーヘッド）

$$S = \frac{1}{(1-P) + \frac{P}{N} + f(N)}$$

S：高速化率
P：並列化率
N：プロセッサ数
$f(N)$：オーバーヘッド

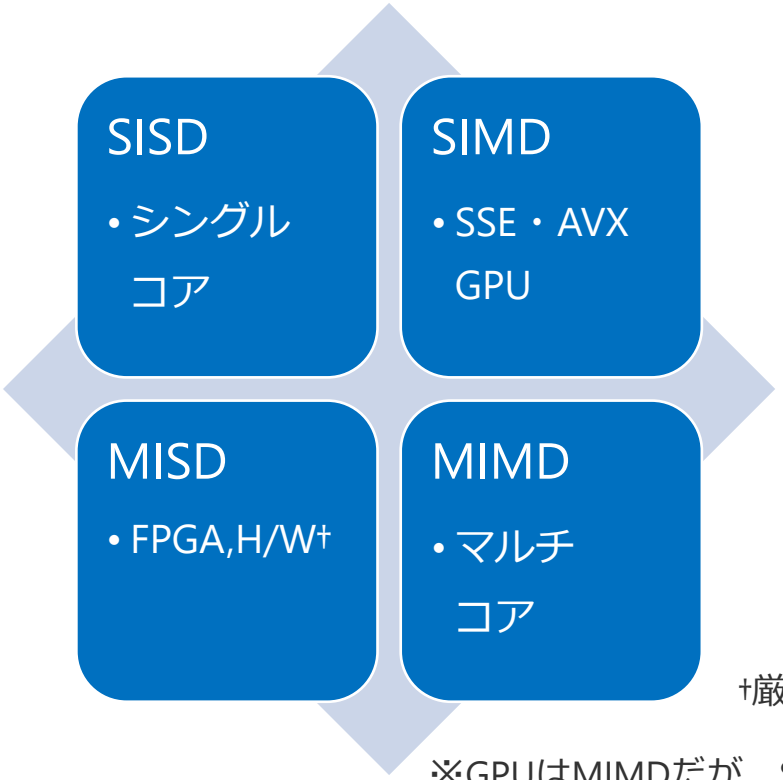


$f(N)$=0.001Nのコア数が増えると線形にオーバーヘッドが増えると仮定

並列化に最適ポイントが生じる

フリンの分類

命令の並行度とデータの並行度に基づく4つの分類



Single Instruction, Single Data stream (SISD)

Single Instruction, Multiple Data streams (SIMD)

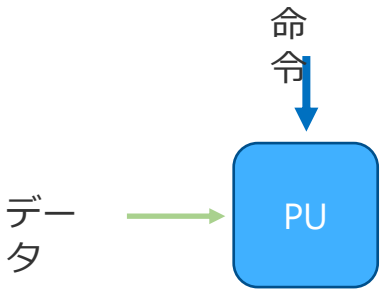
Multiple Instruction, Single Data stream (MISD)

Multiple Instruction, Multiple Data streams (MIMD)

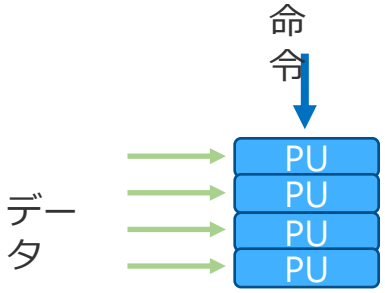
†厳密には多段に適応するため，MISDではないという専門家の意見も

※GPUはMIMDだが，SIMD風に書くときに最大のパフォーマンスを発揮する演算機
NVIDIAは，SIMT (Single Instruction, Multiple Thread) と呼称

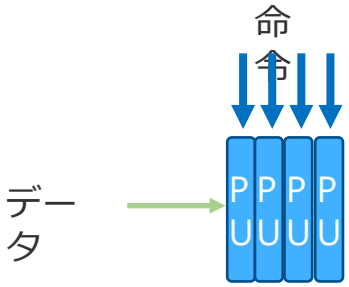
フリンの分類（図解）



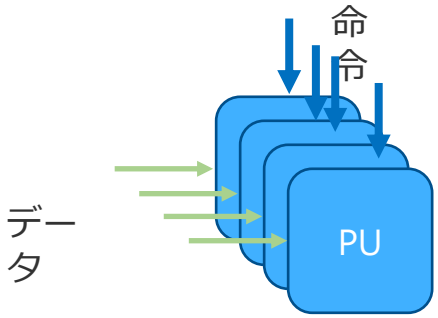
SISD：逐次プログラム



SIMD：一度の命令で複数のデータ処理



MISD：対象外



MIMD：マルチスレッドプログラミング

並列化とベクトル化プログラム

- 並列化
  - OpenMP
  - Pthreadよりも非常に簡単
- ベクトル化
  - intrinsics

OpenMPによる並列化：加算


```
void add(uchar* a, uchar* b, uchar* dest, int num)
{

        for(int i=0;i<num;i++)
        {
                dest[i] = a[i] + b[i];
        }
}
```

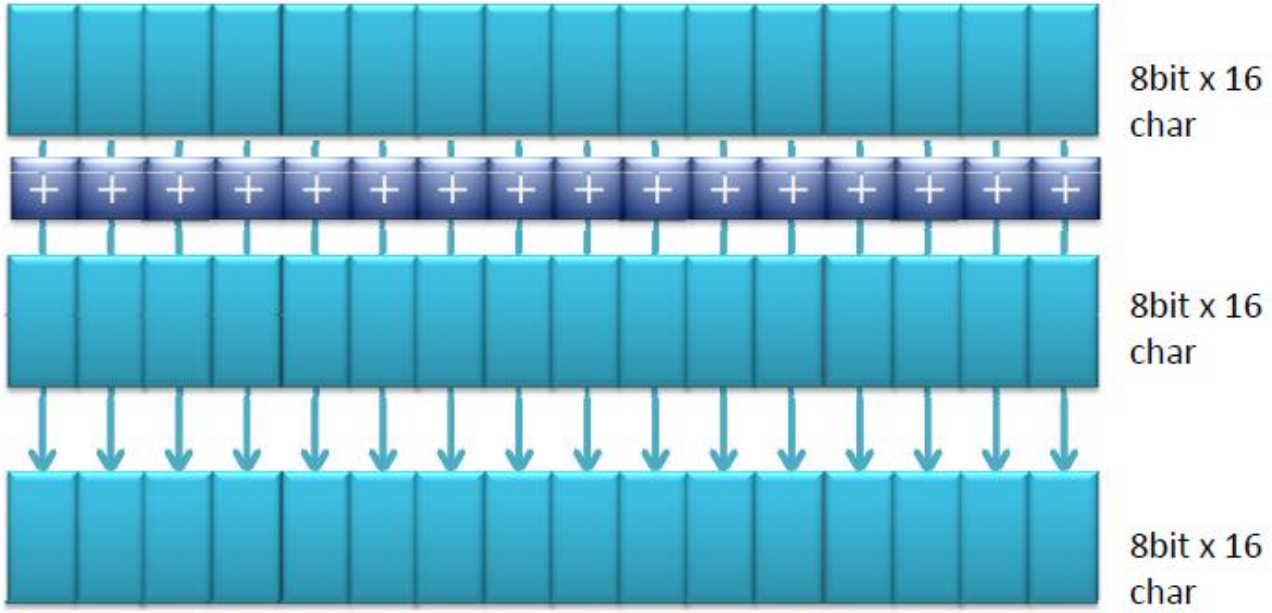
```
void add_omp (uchar* a, uchar* b, uchar* dest, int num)
{
        #pragma omp parallel for
        for(int i=0;i<num;i++)
        {
                dest[i] = a[i] + b[i];
        }
}
```

```
#pragma omp parallel for
```

この一行を追加するだけでforループが並列化される

SIMD演算

“1回の命令”で16個のデータを同時に足し算する例



intrinsicによる加算


```
void add_sse_uchar(uchar* a, uchar* b, uchar* dest, int num)
{
        for(int i=0;i<num;i+=16)
        {
        //メモリ上の配列A，Bを各をレジスタへロード
        __m128i ma = _mm_load_si128((const __m128i*)(a+i));
        __m128i mb = _mm_load_si128((const __m128i*)(b+i));

        //A,Bが保持されたレジスタの内容を加算してmaのレジスタにコピー
                ma = _mm_add_epi8(ma,mb);

        //計算結果のレジスタ内容をメモリ（dest）にストア
                _mm_store_si128((__m128i*)(dest+i), ma);
        }
}
```

16個づつ処理
→ループアンロール

GPUのように，メモリのロード・ストアが必要
（ただし，非常に高速）

たった５行？

必要な関数を呼び出すだけ！
レジスタの管理も不要

ベクトル並列化プログラム


```
void boxfilter(float* src, float* dest, int w, int h, int r)
{
	for(int j=r;j<h-r;j++)
	{
		float normalize = 1.0f/(float)((2*r+1)*(2*r+1));
		for(int i=r;i<w-r;i++)
		{
			float msum = 0.f;
			for(int l=-r;l<=r;l++)
			{
				for(int k=-r;k<=r;k++)
				{
					msum += src [w*(j+l)+i+l];
				}
			}
			dest[w*j+i]=msum*normalize;
		}
	}
}
```

ベクトル並列化プログラム


```
void boxfilter_sse_omp(float* src, float* dest, int w, int h, int r)
{
	#pragma omp parallel for
	for(int j=r;j<h-r;j++)
	{
		float normalize = 1.0f/(float)((2*r+1)*(2*r+1));
		__m128 mnormalize = _mm_set1_ps(normalize);
		for(int i=r;i<w-r;i+=4)
		{
			__m128 msum = _mm_setzero_ps();
			for(int l=-r;l<=r;l++)
			{
				for(int k=-r;k<=r;k++)
				{
				__m128 ms=_mm_loadu_ps(src+w*(j+l)+i+l);
					msum = _mm_add_ps(msum,ms);
				}
			}
			msum = _mm_mul_ps(msum,mnormalize);
			_mm_storeu_ps(dest+w*j+i,msum);
		}
	}
}
```

プログラムの書き方参考ページ

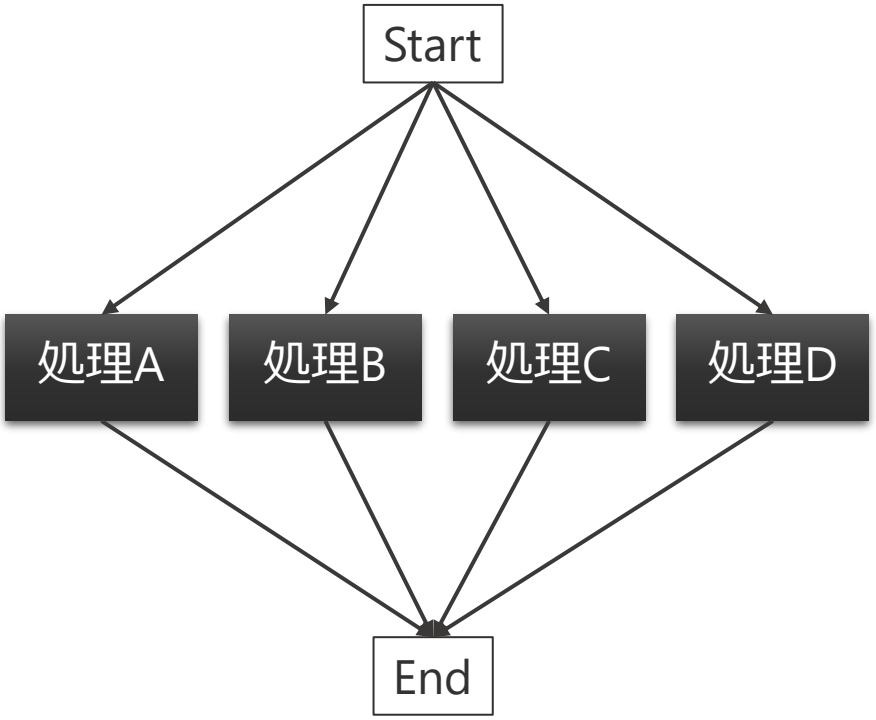
- 具体的な並列プログラムの書き方は以下を参照
  - 福嶋慶繁「マルチコアを用いた画像処理」SSII 2014
  - 福嶋慶繁「組み込み関数(intrinsic)を用いたSIMD化による高速画像処理入門」
    - 検索すると，Slideshareに資料があります．

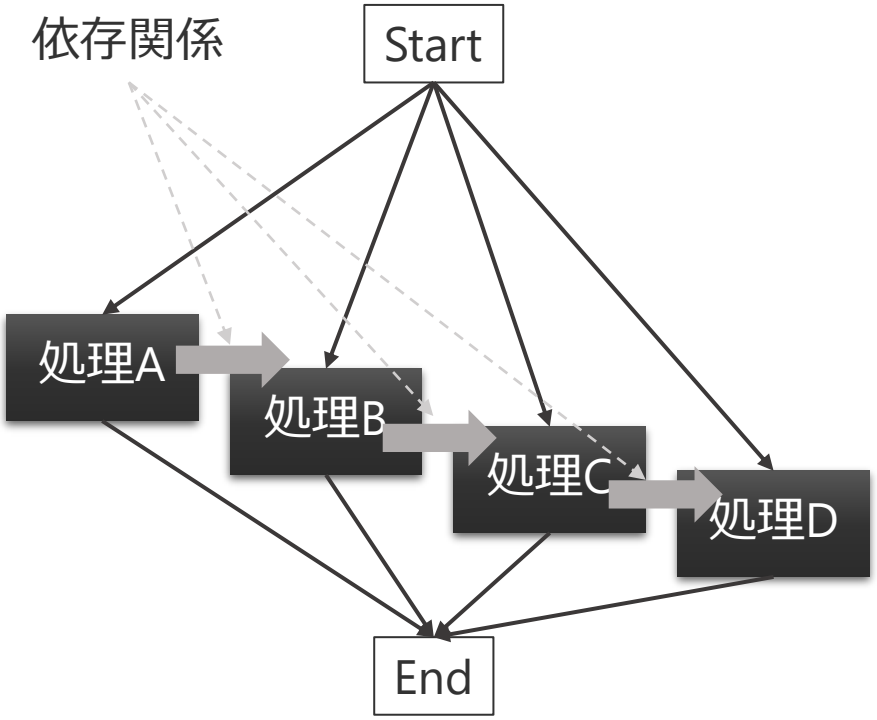
並列画像処理プログラミングデザインパターン

Partitioning

- データを分割する
- 計算を分割する
  - パイプライン計算

処理の依存関係





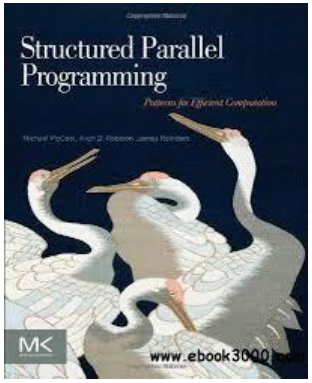
処理に依存関係なし：
並列化で性能向上する

処理に依存関係あり：
並列化で性能向上しない

構造化並列プログラミング

並列化のデザインパターン

効率のよい並列プログラムの形とパターンを示した最も詳しい教科書



原著
Structured Parallel Programming: Patterns for Efficient Computation
Michael McCool (著), James Reinders (著), Arch Robison (著)

翻訳
構造化並列プログラミング―効率良い計算を行うためのパターン
マイケル・マックール (著), 菅原 清文 (翻訳), エクセルソフト (翻訳)

並列化プログラミングの並列パターン

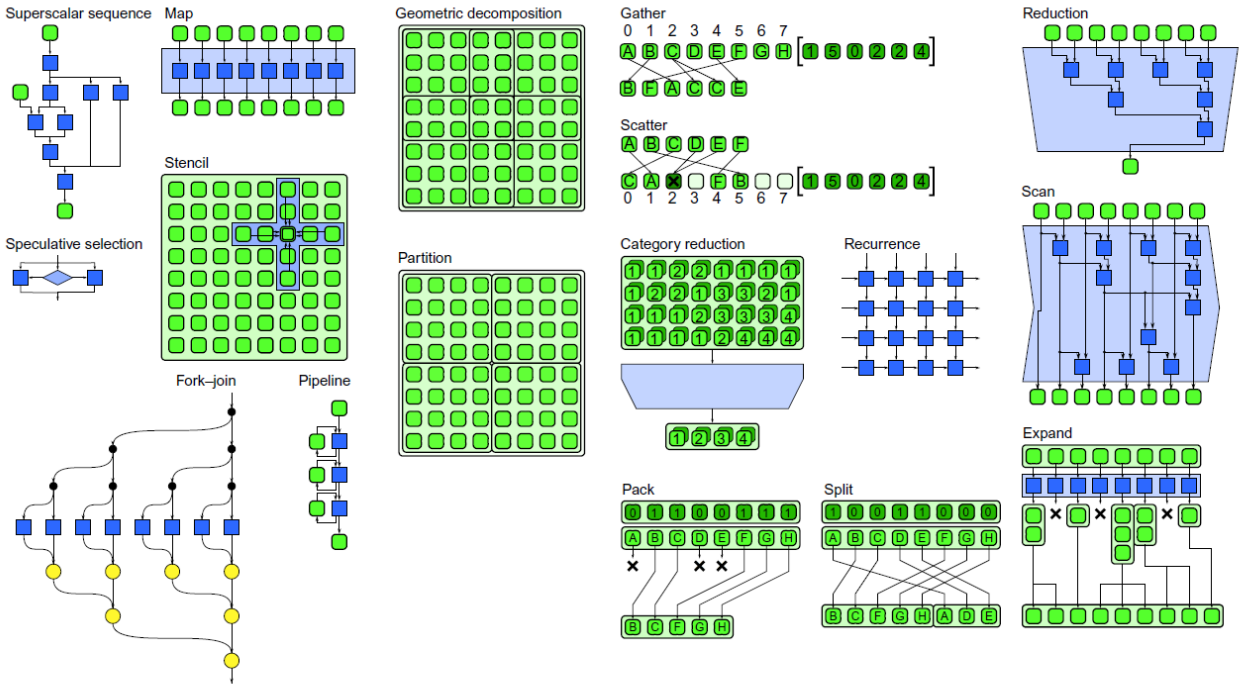


FIGURE 1.11

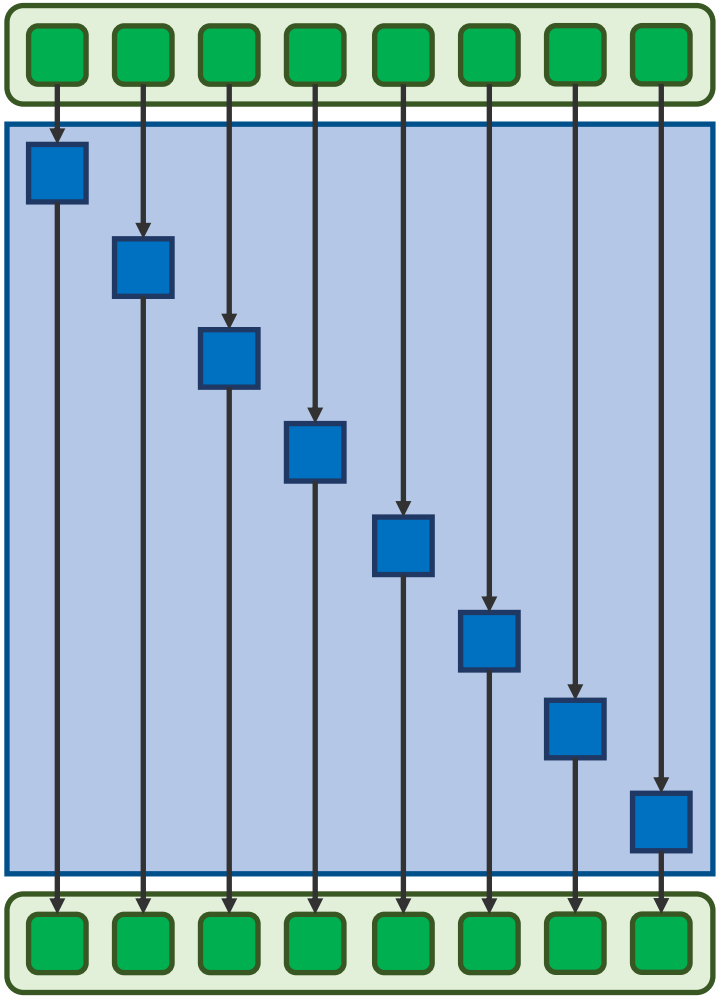
Overview of parallel patterns.

Structured Parallel Processing p.21

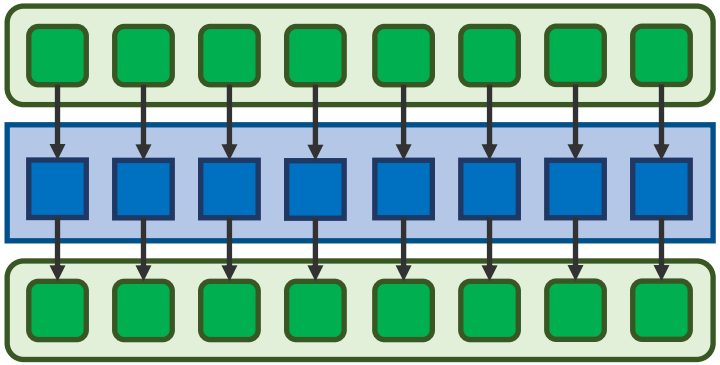
画像処理に頻出のパターン

- マップ
- ステンシル
- リダクション
- スキャン
- パイプライン
- フォークジョイン

マップ (Map)



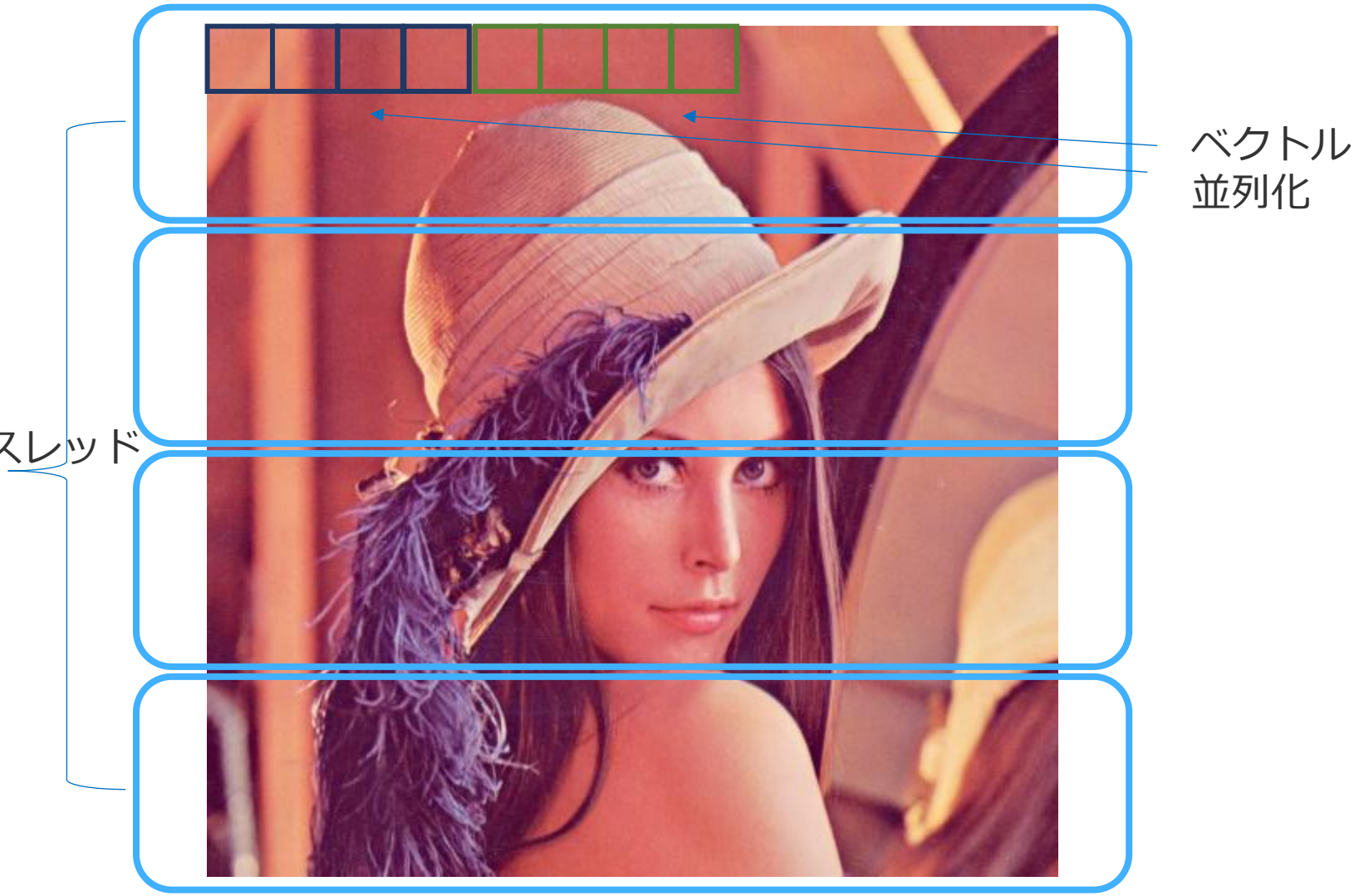
画像のベクトル・並列化の基本系



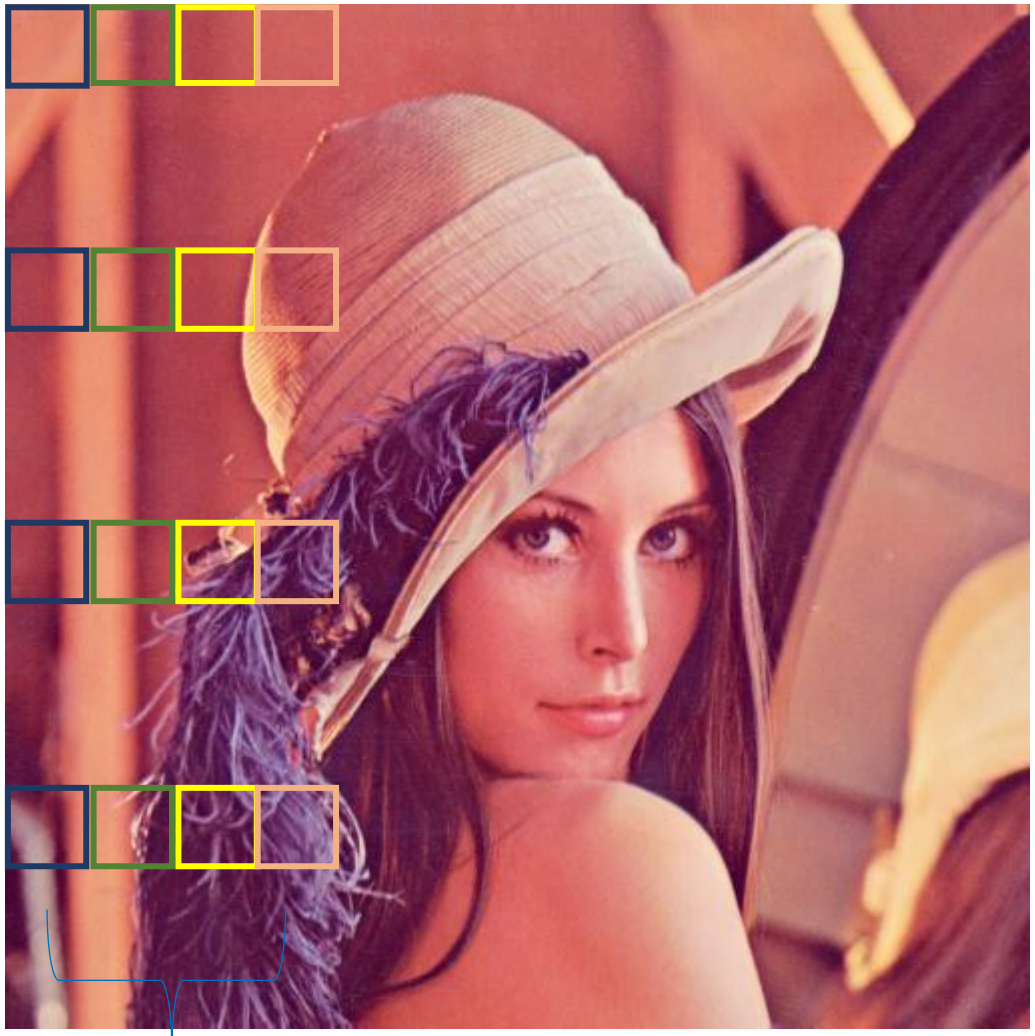
マップの画像処理

- 画像同士の四則演算
- コントラスト強調，ガンマ変換などの関数適用
- 閾値処理
- 色変換

マップの並列化とベクトル化



マップの並列化とベクトル化



ベクトル並列化

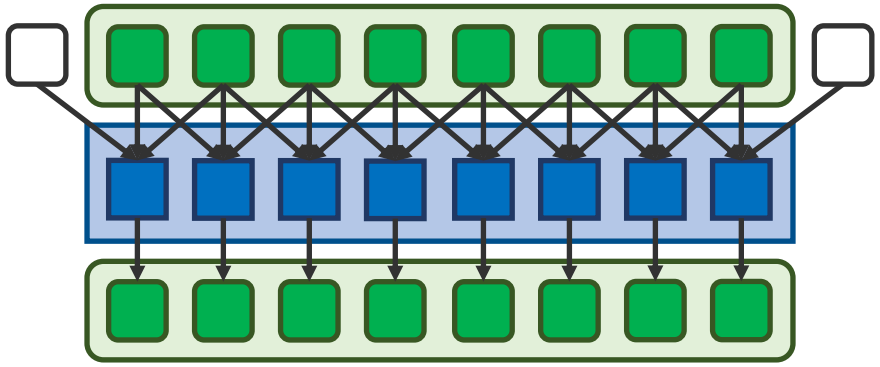
マルチスレッド並列化

だめな例

- 過分割
- キャッシュ効率最低

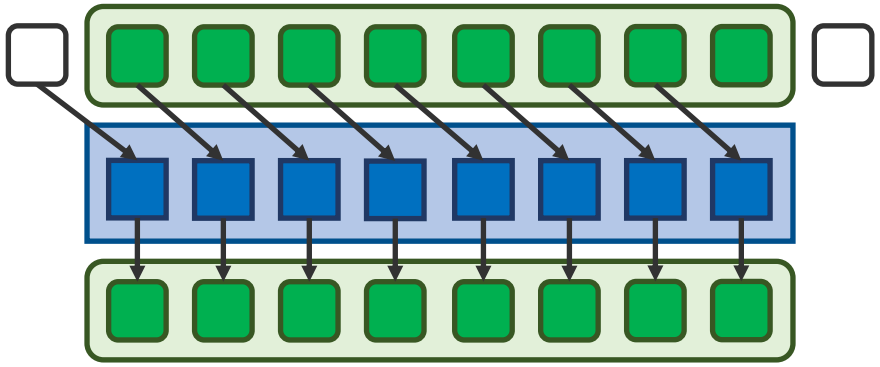
ステンシル (Stencil)

- マップ処理に入力に近傍画素が追加された多入力処理
- シフトしたマップの繰り返し



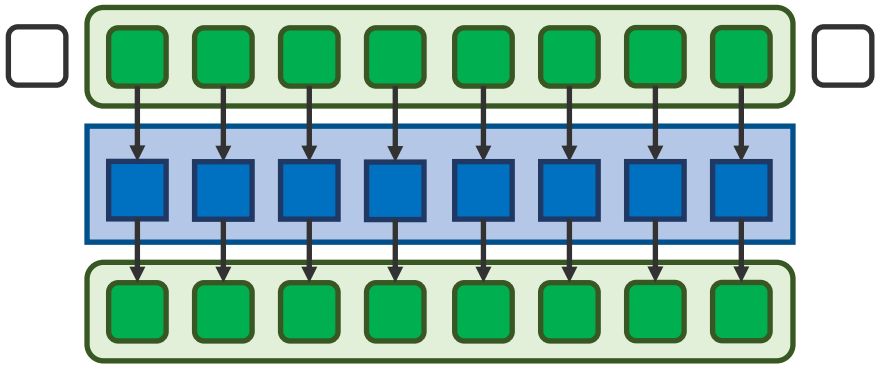
ステンシル (Stencil)

- マップ処理に入力に近傍画素が追加された多入力処理
- シフトしたマップの繰り返し



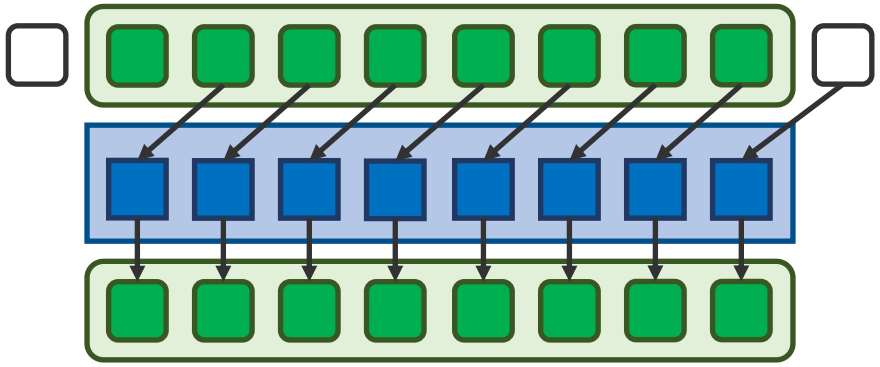
ステンシル (Stencil)

- マップ処理に入力に近傍画素が追加された多入力処理
- シフトしたマップの繰り返し



ステンシル (Stencil)

- マップ処理に入力に近傍画素が追加された多入力処理
- シフトしたマップの繰り返し

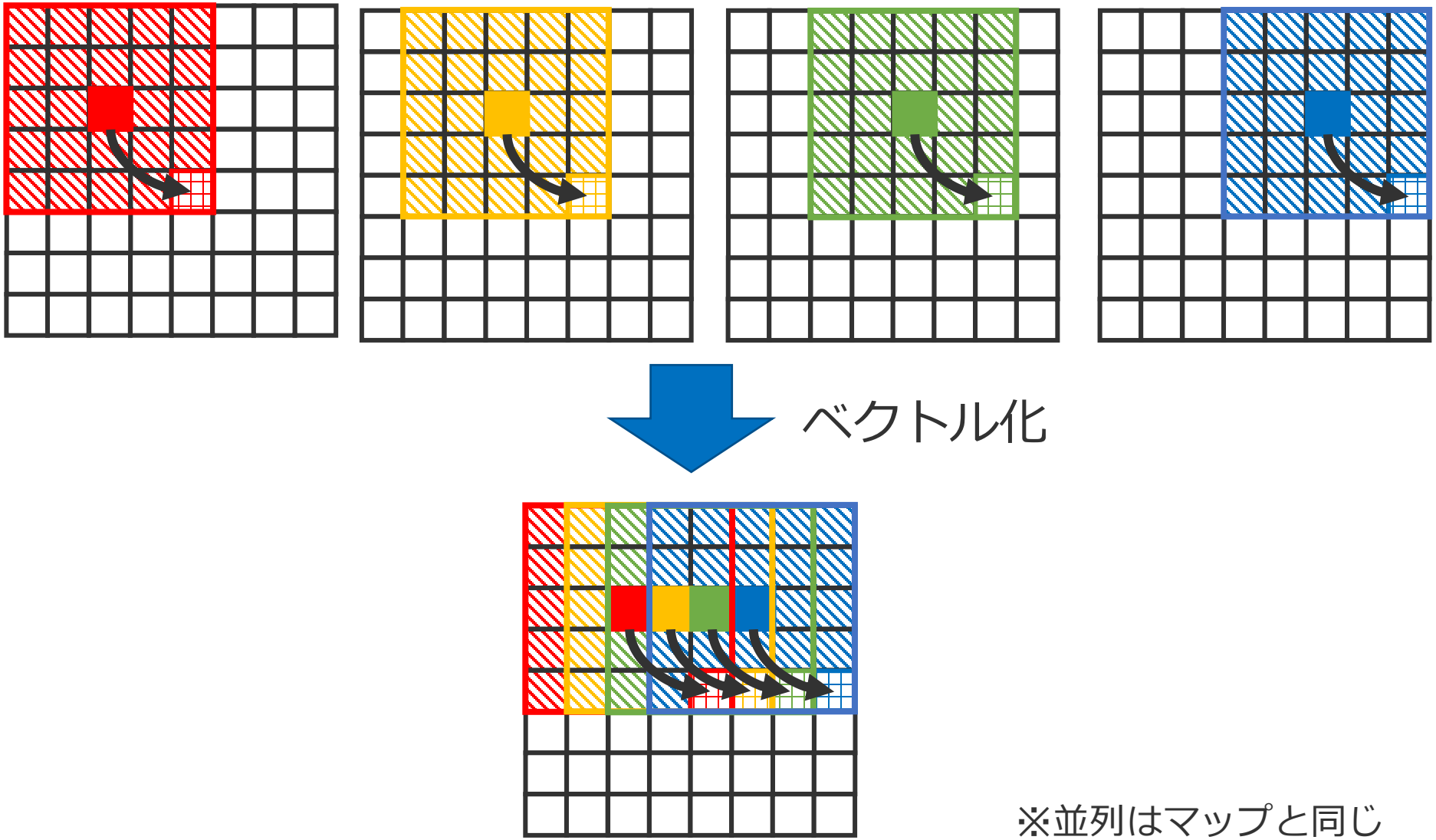


ステンシルの画像処理

- FIR畳み込みフィルタ
  - 移動平均フィルタ
  - ガウシアンフィルタ
  - バイラテラルフィルタ
  - ノンローカルミーン
  - ソーベルフィルタ
  - ラプラシアンフィルタ
  - 膨張・収縮などのモルフォロジ

    メディアンはソートが入るためここ入るか微妙

ステンシルのベクトル化

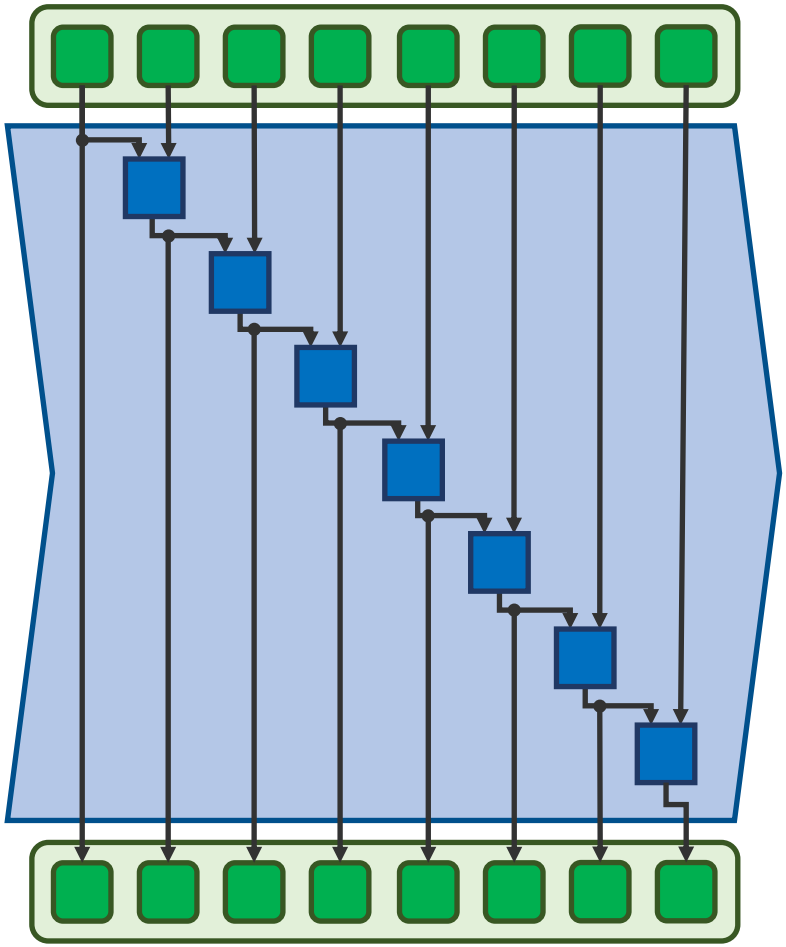


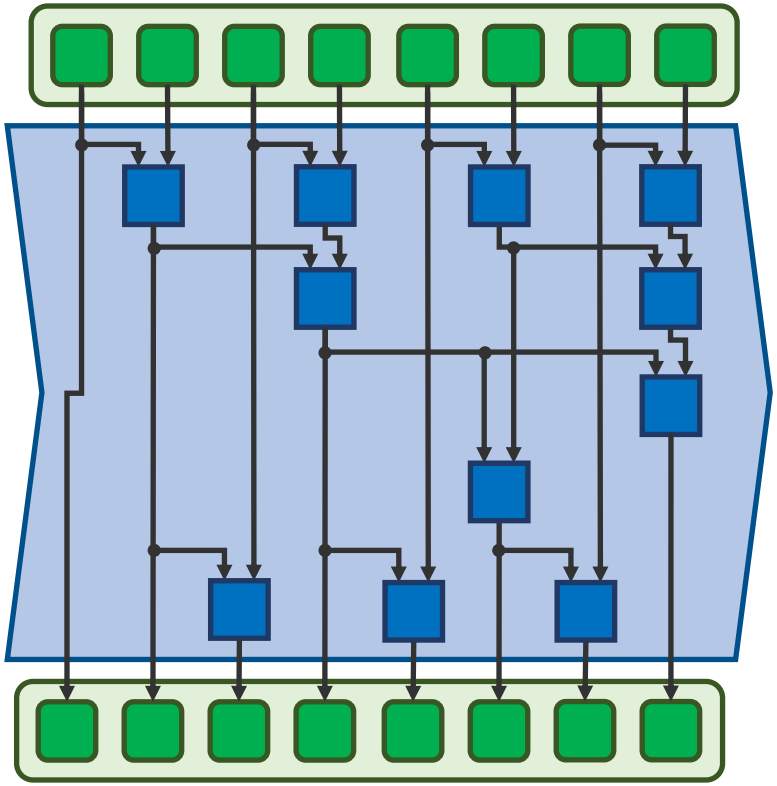
※並列はマップと同じ

スキャン (Scan)

一つ前の計算結果を使って次の要素を計算

依存関係がある





スキャンの画像処理

- リカーシブフィルタ，再帰処理
  - 移動平均のインテグラルイメージ
  - IIRフィルタ全般
    - ガウシアン近似
    - ラプラシアン近似
  - コサインインテグラルによるガウシアン近似
  - アクティブ探索（ヒストグラムの再帰）
  - FFTなどのバタフライ演算（過去，未来を使用）
    - 行列積で書けばこのパタンではないがコスト大

スキャンのマップ処理化

- 1次元信号はどうにもならないが，画像処理は2次元
- もう１次元の信号を見ればスキャン処理もマップ処理で表現可能

スキャンのマップ処理化

この方向の処理に依存関係あり

スキャンのマップ処理化

この方向の処理に依存関係あり

横方向に並列に切っても問題ない．サブ画像へのスキャン処理をマップ処理している

ベクトル化はこのままでは効率的にできない

スキャンのマップ処理化

転置して処理

横方向依存関係がなくなるため
縦方向にベクトル演算可能

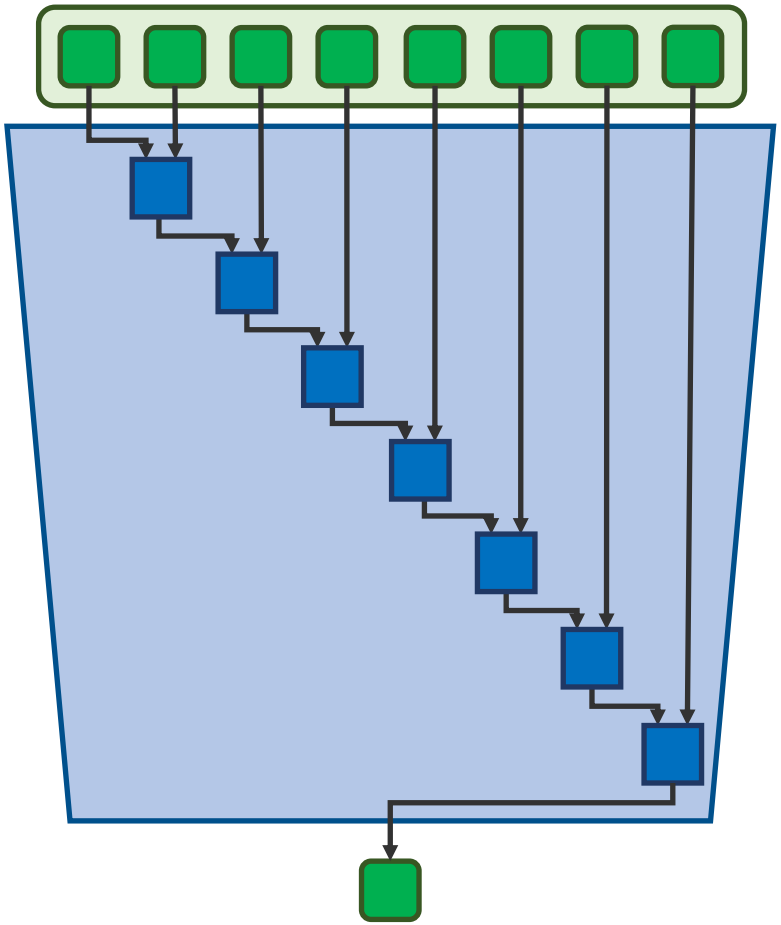
水平のベクトル間でマップ演算

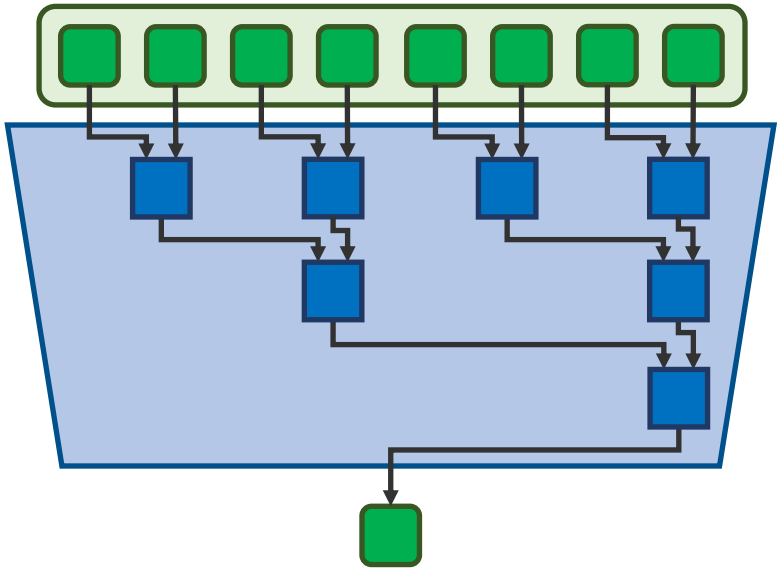
すべてがマップ演算

この方向の処理に依存関係あり

リダクション (Reduction)

スキャンの最終出力だけ
分割統治法





リダクションの画像処理

- 画像全体の
  - 平均
  - 分散
  - 最大，最小
  - 中央値
  - 最頻値
  - モーメント
  - ヒストグラム

など，画像の統計量一般

リダクションのマップ処理

- 分割統治法で考えればマップ処理と小さなデータのリダクション処理に分解可能
- 画像処理でリダクション処理が必要な場合画素位置に依存関係がない場合が多い

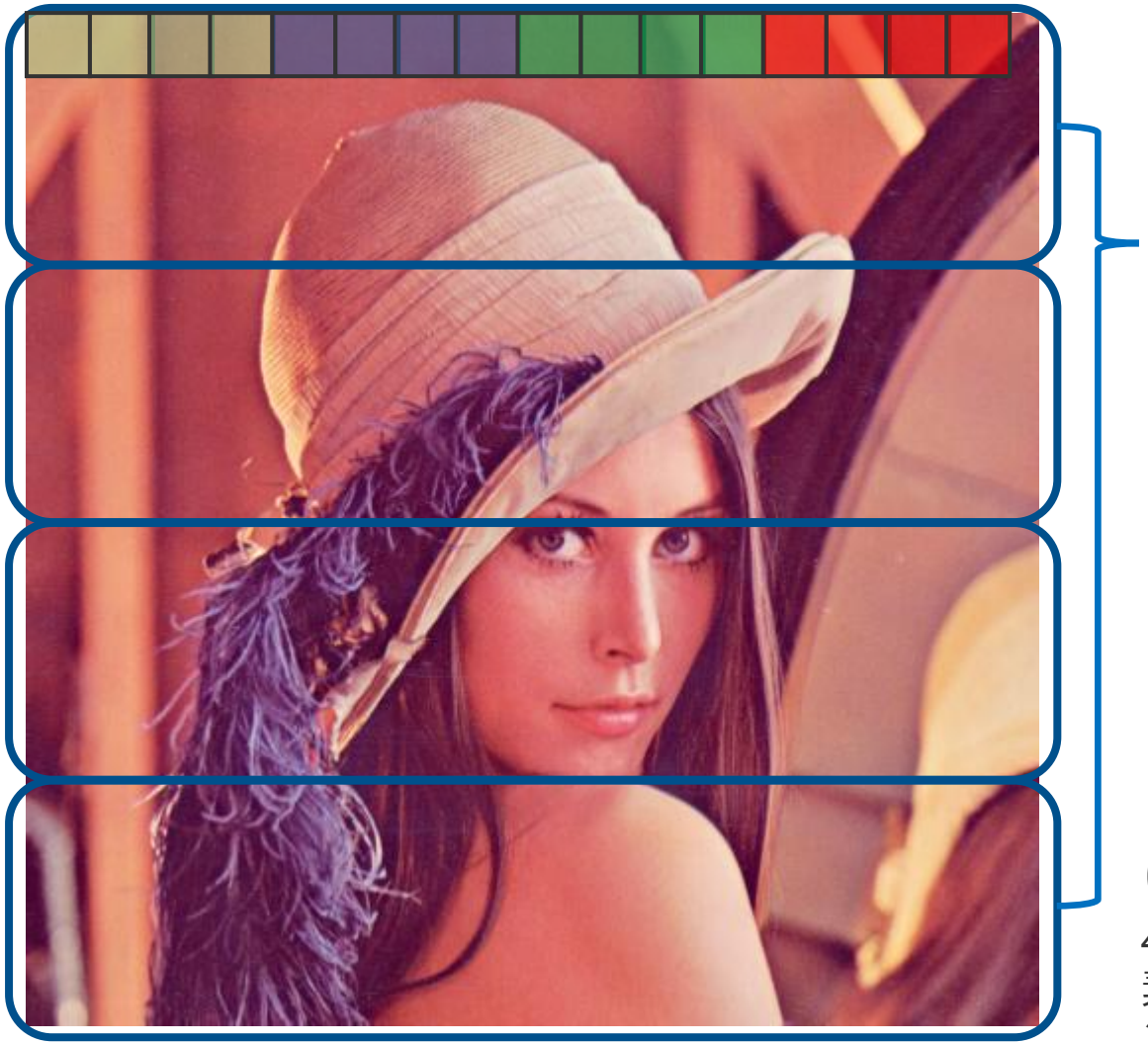
リダクション処理（総和）

分割統治法

サブ画像の各総和を独立にに計算（マップ処理）して，少数の結果をリダクションで総和すれば全画像の総和

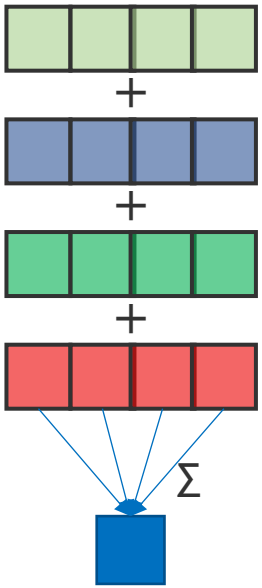
ベクトル化はこのままでは効率的にできない

リダクション処理（総和）



分割統治法

データの位置で分割して，
マップで加算し最後に総和

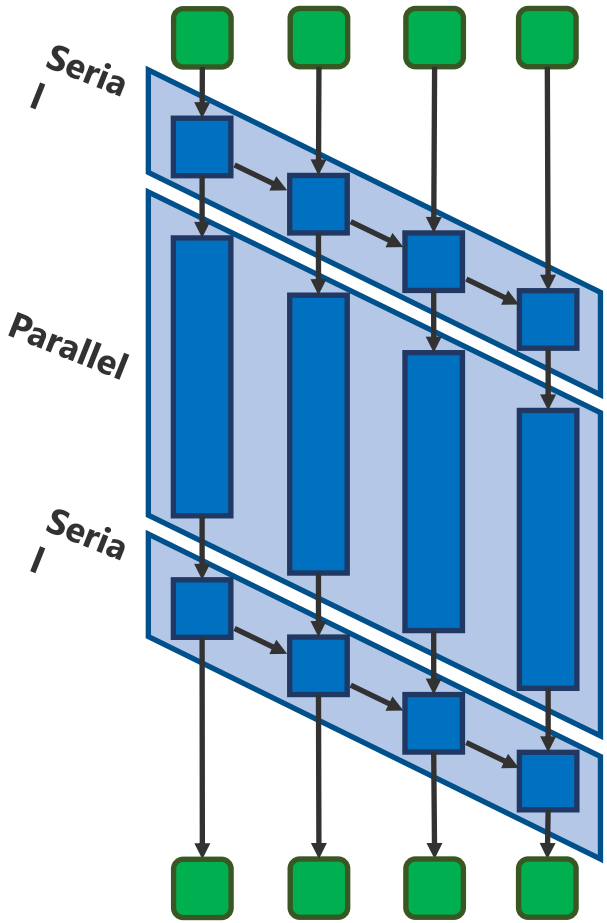


ほぼすべてがマップ演算

4並列4ベクトル長なら512x512=の画素の処理のうち4画素のリダクションを4回するだけ.

99.994%がマップ処理

パイプライン (pileline)



処理を多段に処理
粒度が大きい

パイプラインの画像処理

- 動画像処理
  - 複数の処理を分解してパイプライン化
  - 例えばカラー画像の閾値処理
    - カラーをグレイに変換
    - ソーベルフィルタ（横）
    - ソーベルフィルタ（縦）
    - エッジの閾値処理
  - の４つの処理を１～４フレーム目でずらしながら実行する

フォークジョン (Fork-join)

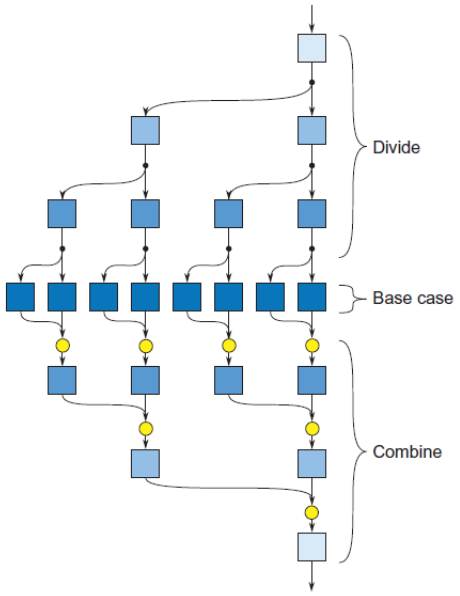


FIGURE 8.2

Nested fork–join control flow in a divide-and-conquer algorithm. For good speedup, it is important that most of the work occur deep in the nesting (more darkly shaded boxes), where parallelism is high.

- 基本形.
- 並列化はすべて Fork-join.
- タスクやフローを並列する基本形.

フォークジョンの画像処理

- 画像処理は単一のパターンでは表現できないことが多いため各処理を各デザインパターンに分解して統合する処理
- 例：ステレオマッチング
  - コスト計算：マップ
  - コスト集約：ステンシル or スキャン
  - デプス推定：リダクション

各処理を適切な粒度で分解して処理

プログラミング言語

- CPU
  - Intrinsics
  - OpenMP, Intel TBB
  - OpenCL
- GPU
  - Cuda
  - GLSL
  - OpenCL
- FPGA
  - OpenCL

ループヒュージョンとタイリング

演算強度

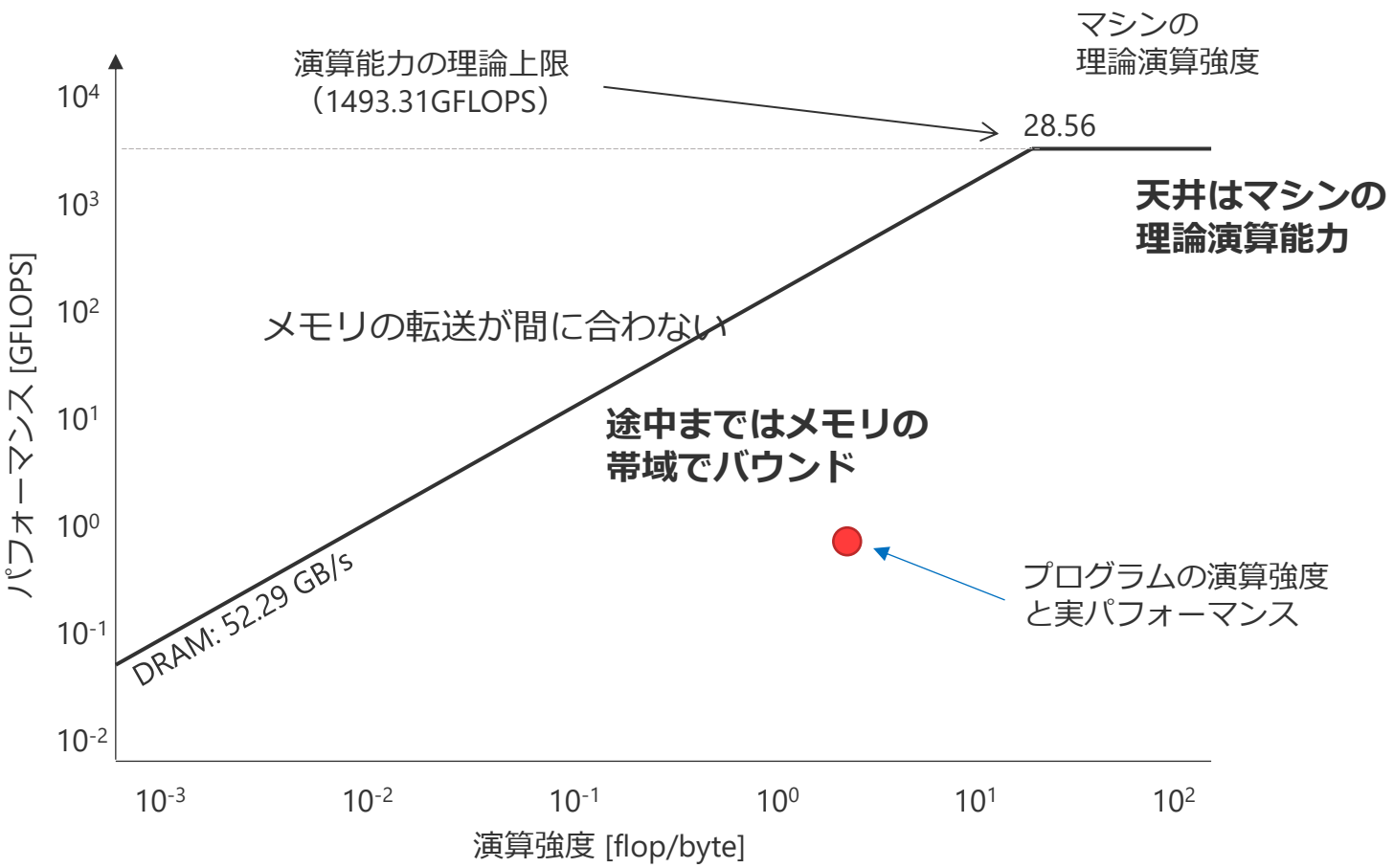
- ロードを待たずにどれだけ演算可能かを表す指標 (F/B)
  - FLOPSをデータのバンド幅で割った指標（F/B）
  - FLOPSは演算のみに着目した指標でメモリ帯域は考慮していない
- その逆数は浮動小数点演算をするために必要な帯域 (B/F)
- プログラムの演算強度や計算機の理論B/Fからどれだけ演算性能を引き出しているか解析するために使用

演算強度

- 例：浮動小数点演算の加算
  - ２つデータ（４バイト）読み込み
  - １つの演算
  - 結果１つの書き込み
  - つまり，１２バイトのIOで１つの計算
  - B/F＝１２のマシンでなければメモリ読み込み待ち
- 現在の計算機：
  - スーパーコンピュータ京
    - 64GB/s / 256 GFLOPS=0.25 B/F (float)
  - Core i7 7980XE
    - 85.31GB/s / 2995.2GFLOPS/=0.0228 B/F (float)

ルーフラインモデル

flopsとB/Fでプログラムを解析するモデル
由来：理論FLOPSで屋根の様に天井がクリップされるから

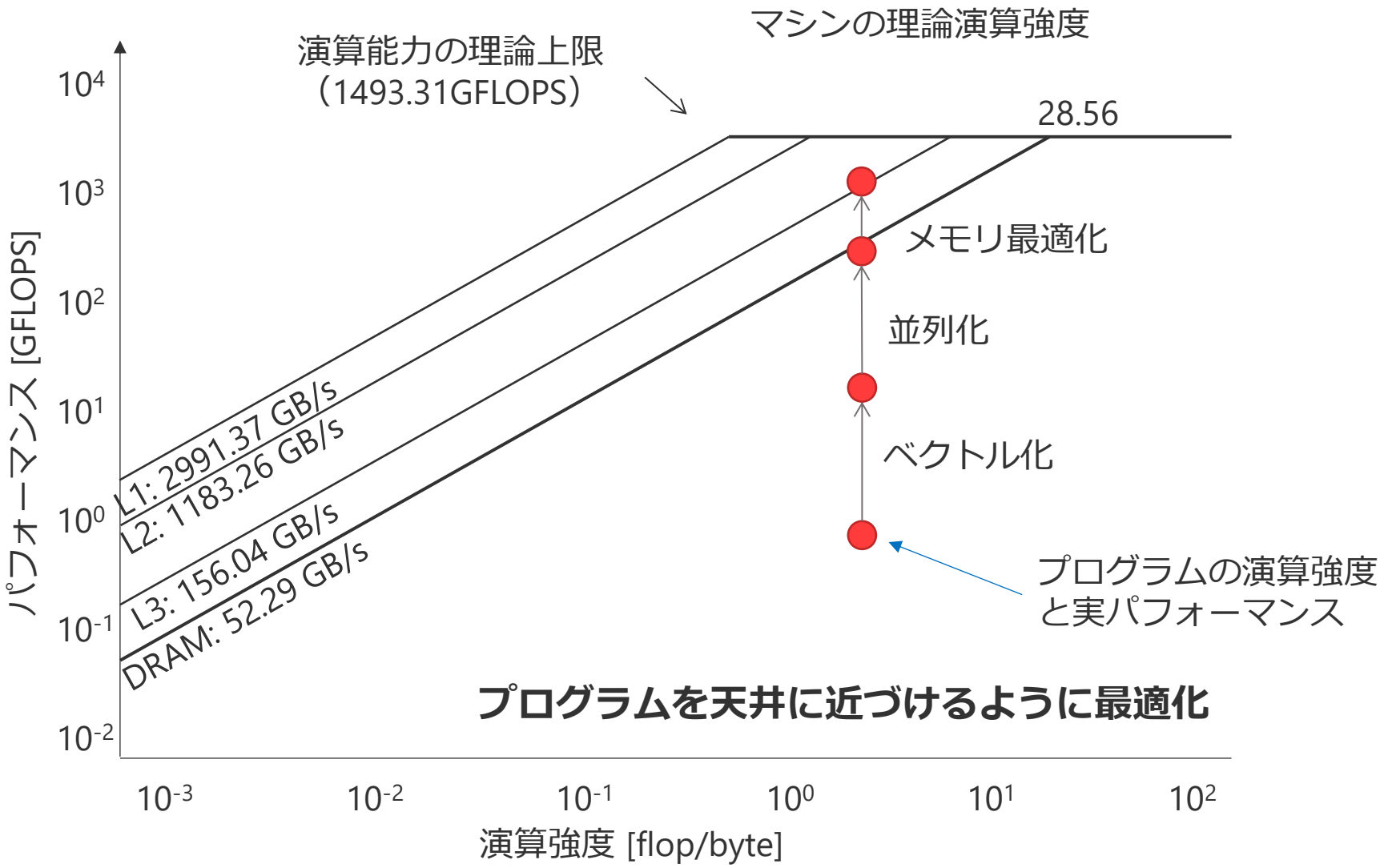


記憶素子のアクセス速度

- L1　　4サイクル
- L2　　12サイクル
- L3　　36サイクル
- メモリ　＞100サイクル

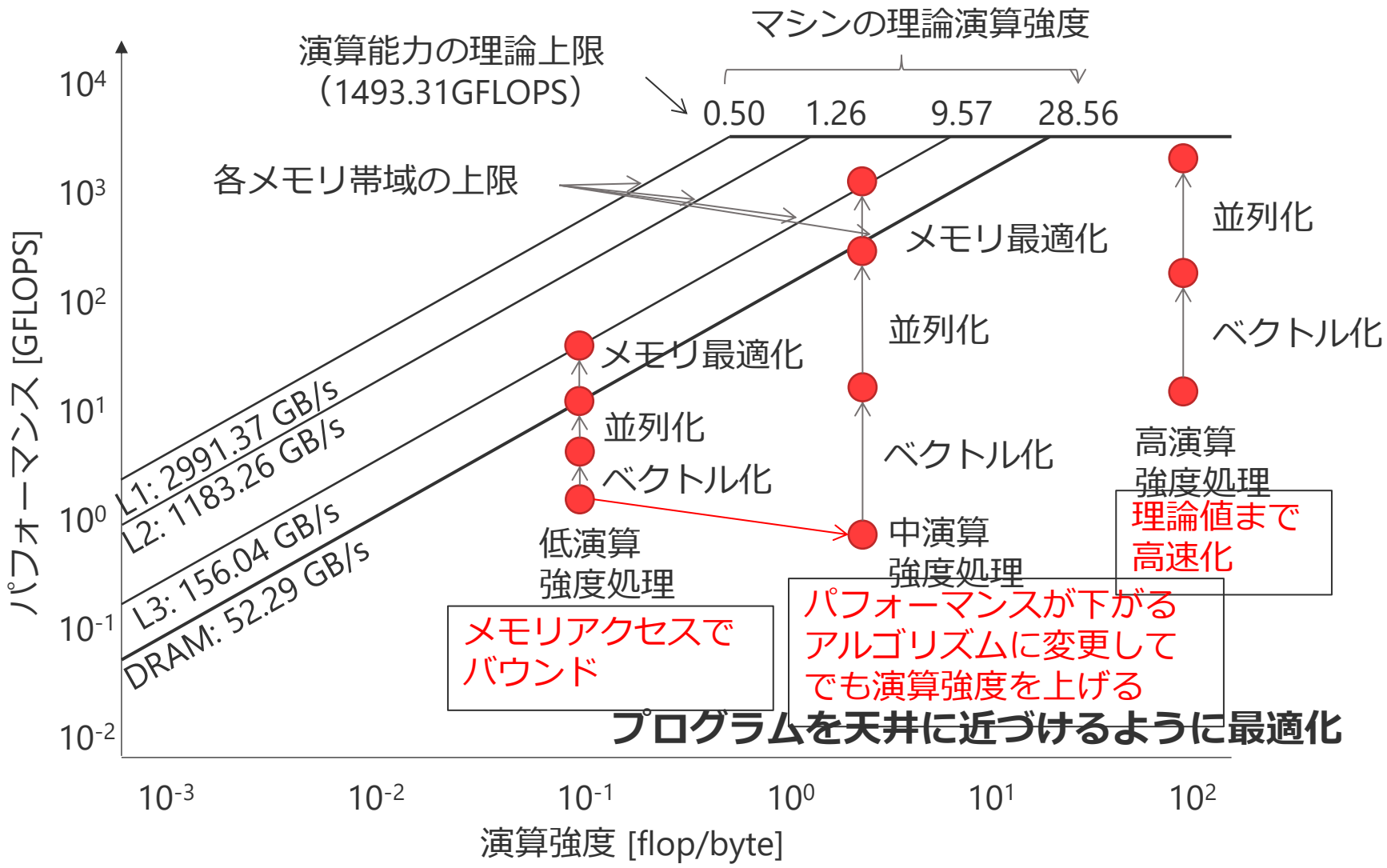
※ CPUはCore i7のHaswellの例

ルーフラインモデル



プログラムを天井に近づけるように最適化

ルーフラインモデル


マシンの理論演算強度
演算能力の理論上限
（1493.31GFLOPS）
0.50
1.26
9.57
28.56
各メモリ帯域の上限
メモリ最適化
並列化
ベクトル化
並列化
ベクトル化
メモリ最適化
並列化
ベクトル化
高演算
強度処理
中演算
強度処理
低演算
強度処理
理論値まで
高速化
パフォーマンスが下がる
アルゴリズムに変更して
でも演算強度を上げる
メモリアクセスで
バウンド
プログラムを天井に近づけるように最適化
L1: 2991.37 GB/s
L2: 1183.26 GB/s
L3: 156.04 GB/s
DRAM: 52.29 GB/s
パフォーマンス [GFLOPS]
演算強度 [flop/byte]
10^4
10^3
10^2
10^1
10^0
10^-1
10^-2
10^-3
10^-2
10^-1
10^0
10^1
10^2

ルーフラインモデル

- 演算強度が高いプログラム
  - 並列化・ベクトル化するだけ
- 演算強度が低いプログラム
  - ループヒュージョンによりメモリアクセス回数を減らす
  - タイリングによりキャッシュを使いまわす
  - 冗長に計算してでも，ループヒュージョン，タイリングを実行して，演算強度が高くなるようにプログラムを変更

ループヒュージョン

- 複数のループをまとめて１つのループで処理すること
  - ループカウントの回数が減る以上にメモリアクセスの総数が減ることで大幅に高速化
  - ただしプログラムが煩雑になる

- 例：
  - カラー画像の閾値処理
    - カラーをグレイに変換
    - ソーベルフィルタ（横）
    - ソーベルフィルタ（縦）
    - エッジの閾値処理

    画像を4回ループする

→ 画素単位に色変換，３ｘ３フィルタ，閾値処理をすればループを１回で済む

ループ順序入れ替え

- ループの順番を入れ替える
  - 操作の順を変える
    - 縦→横から横→縦にしてメモリをシーケンシャルにアクセス

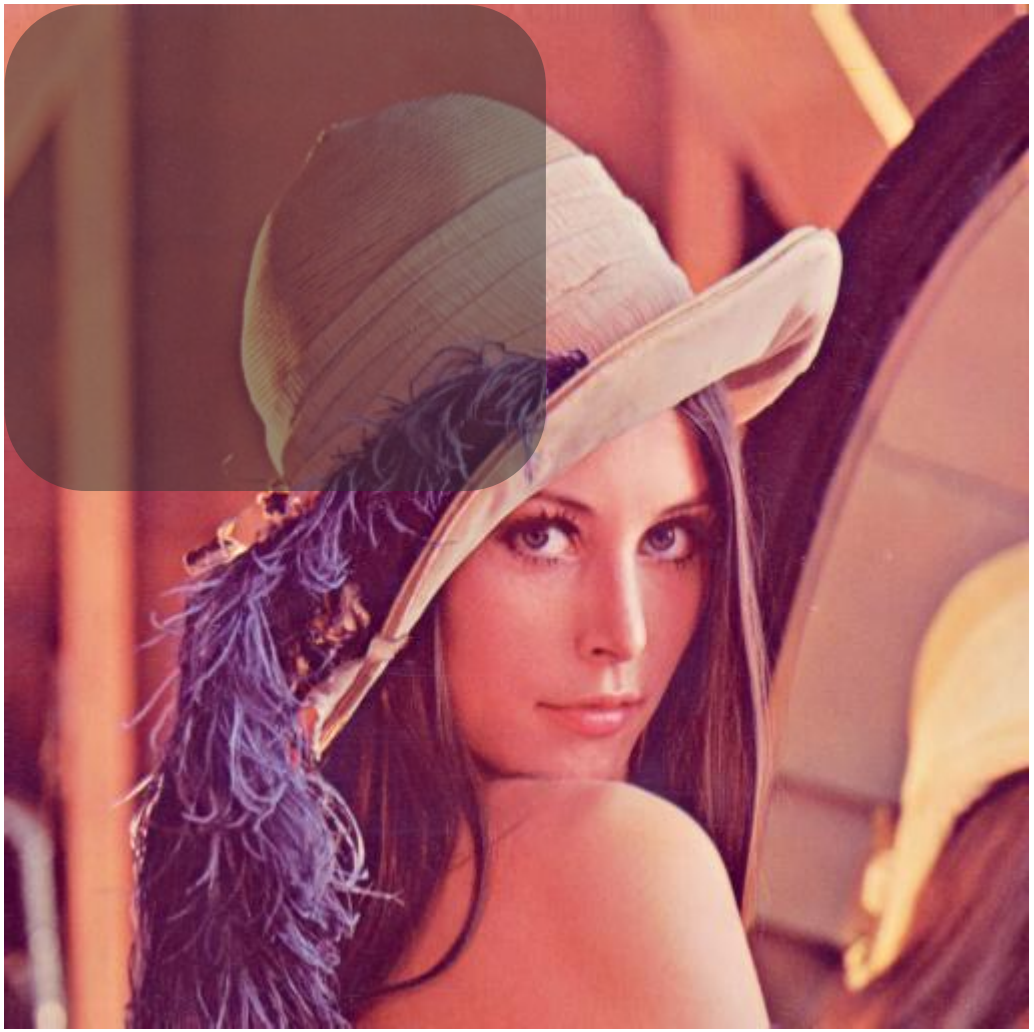
ループ順序入れ替え

- ループの順番を入れ替える
  - 操作の順を変える
    - 縦→横から横→縦にしてメモリをシーケンシャルにアクセス

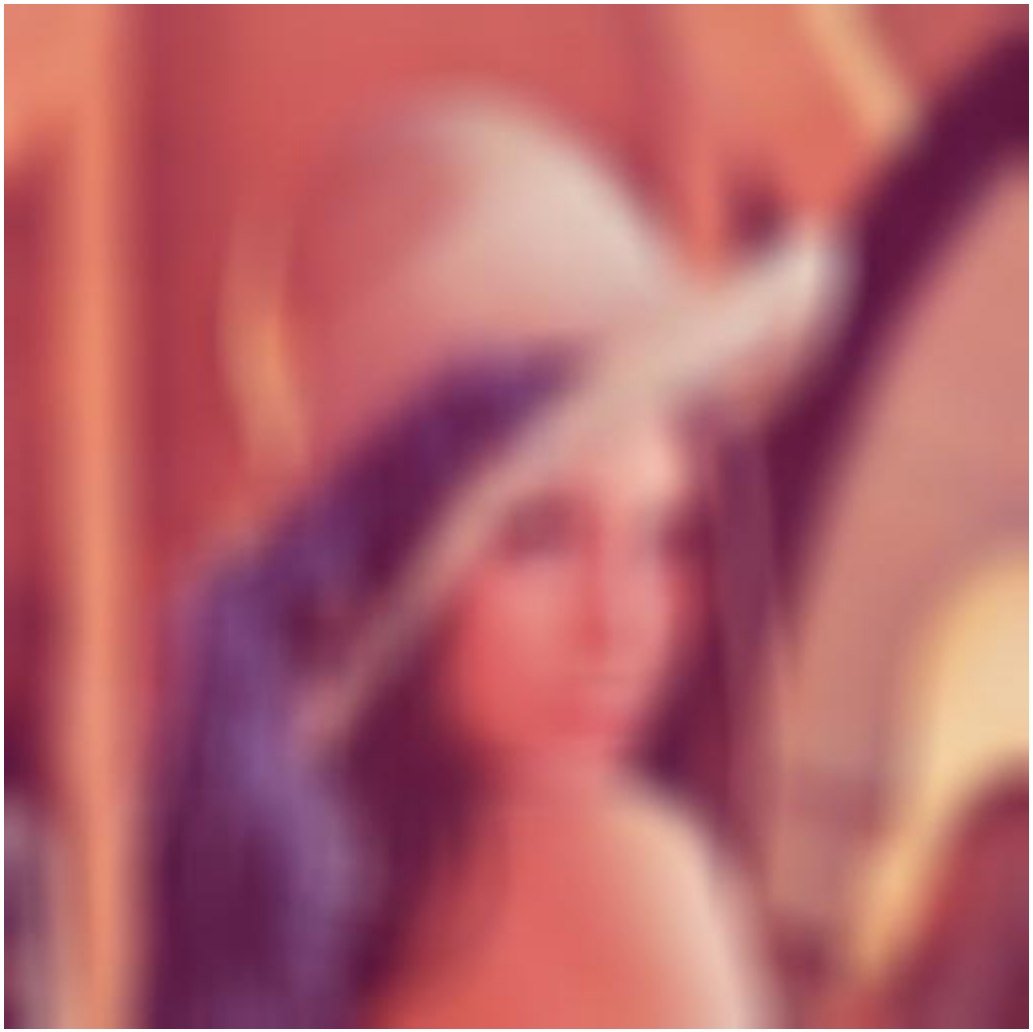
タイリング

- 画像をブロック分割して，ブロック単位で画像処理を行うこと
  - –画像をブロック分割してブロック単位に処理

移動平均フィルタ

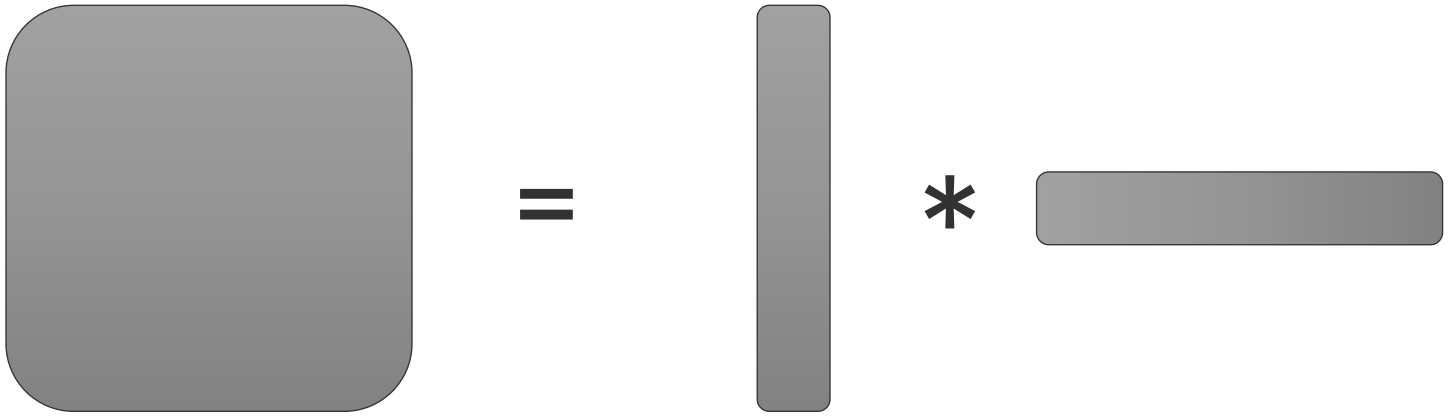


移動平均フィルタ

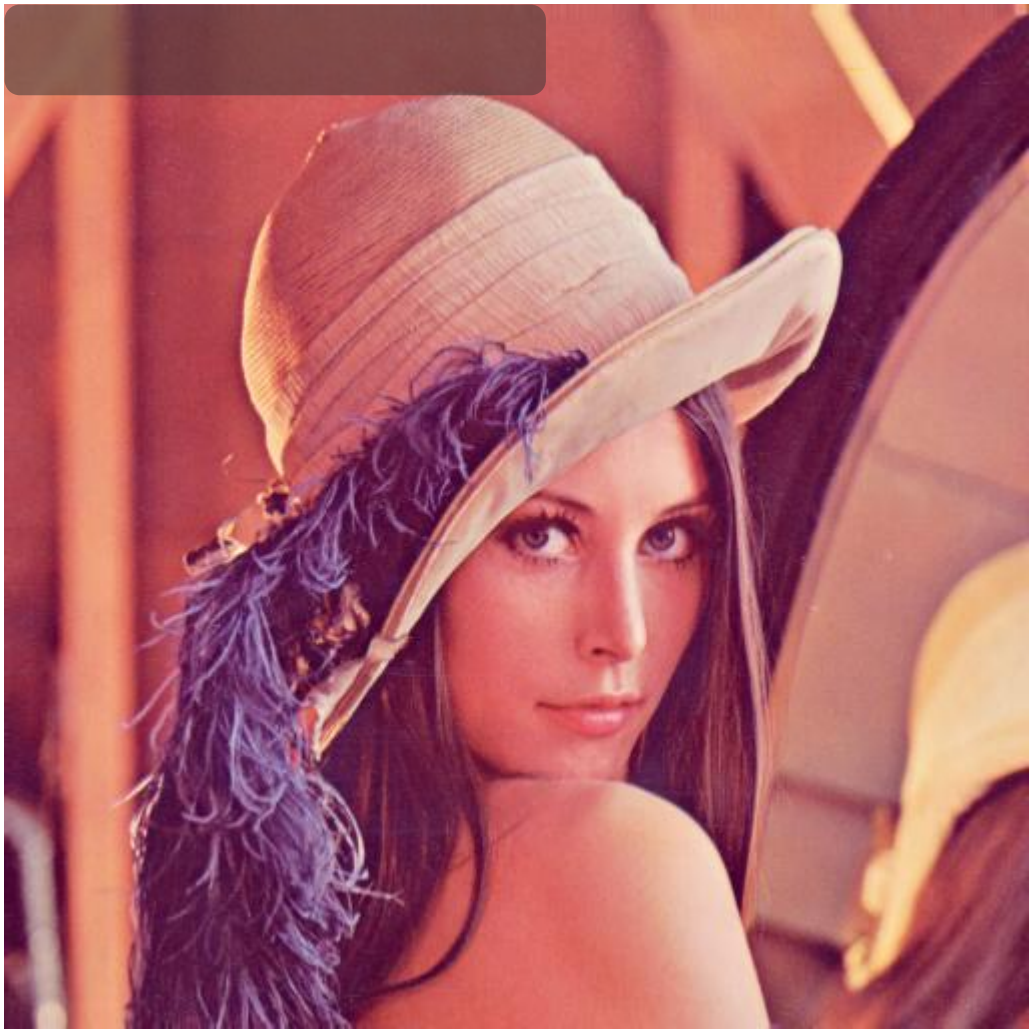


カーネルの可分性

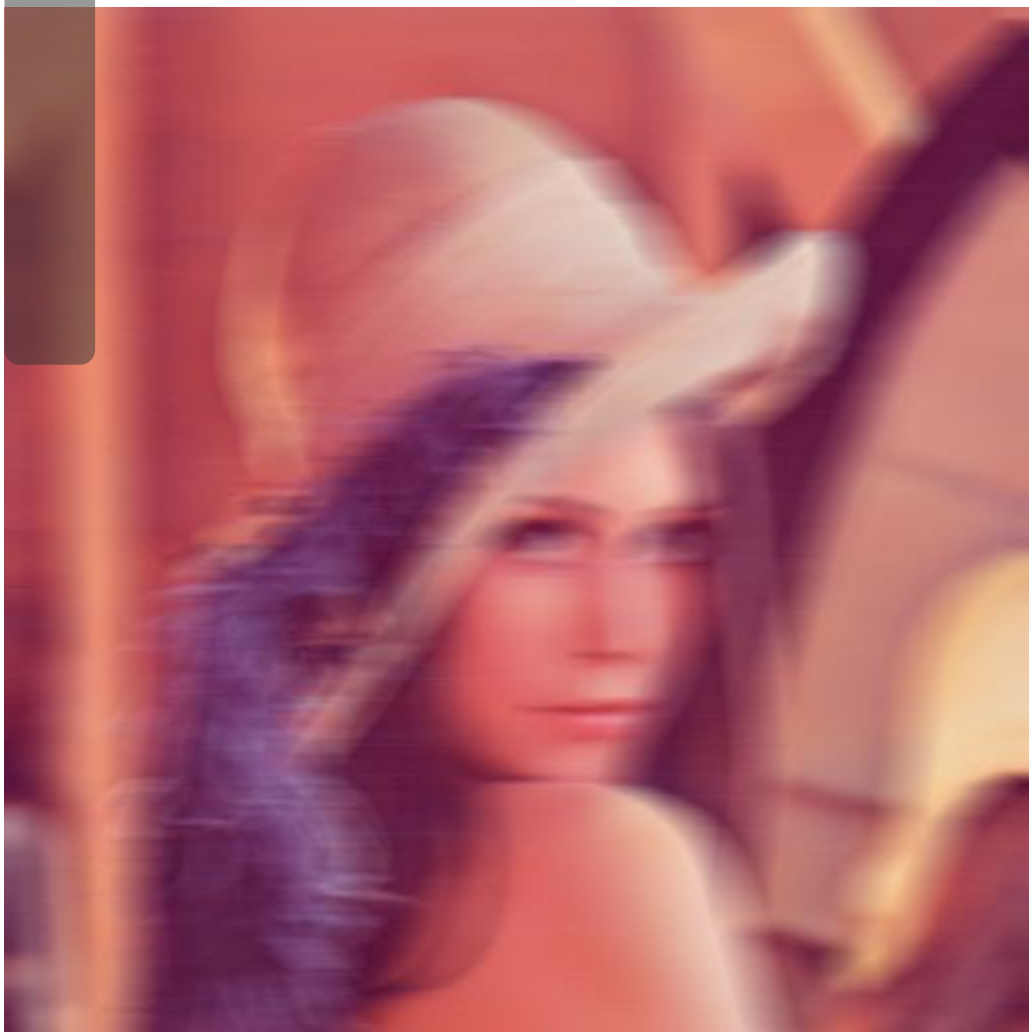
- カーネルが縦・横のフィルタに分解可能
  - 移動平均フィルタ
  - ガウシアンフィルタ
  - ラプラシアンフィルタ
- 計算オーダーが$O(r^2)$から$O(r)$に



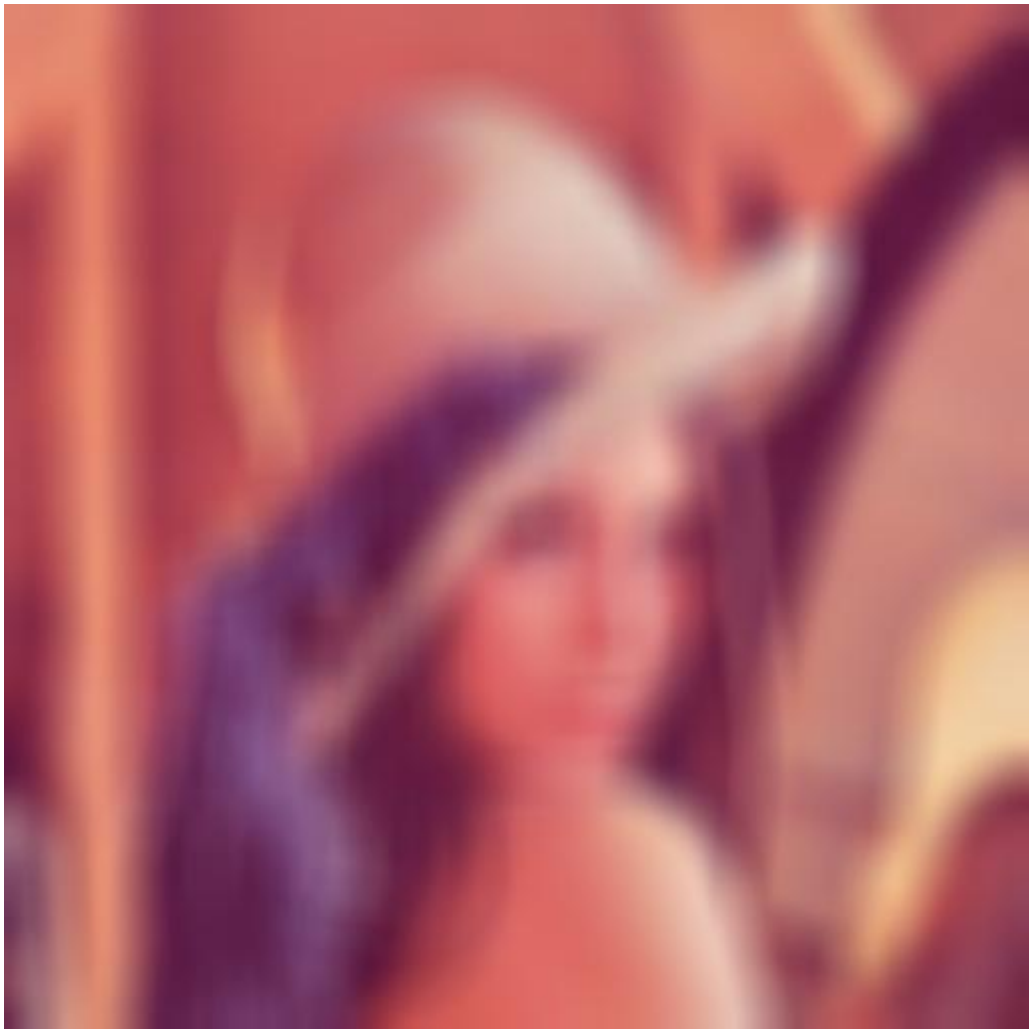
セパラブルフィルタ



セパラブルフィルタ



セパラブルフィルタ

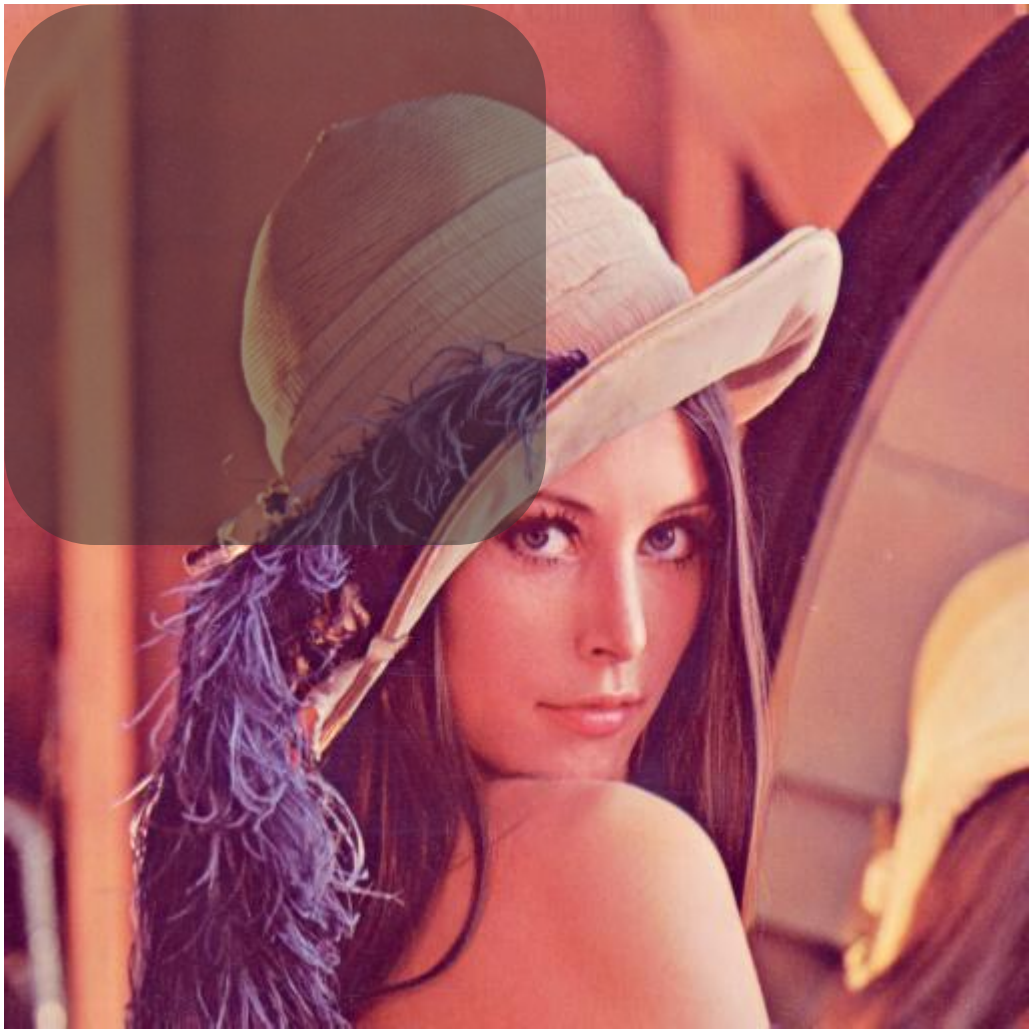


セパラブルフィルタの問題

- アルゴリズムにより計算オーダーが下がったが，一体出力画像を保持して，再度画像全体を画像を操作する2ステージ必要
- 画像データの読み込み量が増えてキャッシュヒット率が低下
- タイリングにより解決

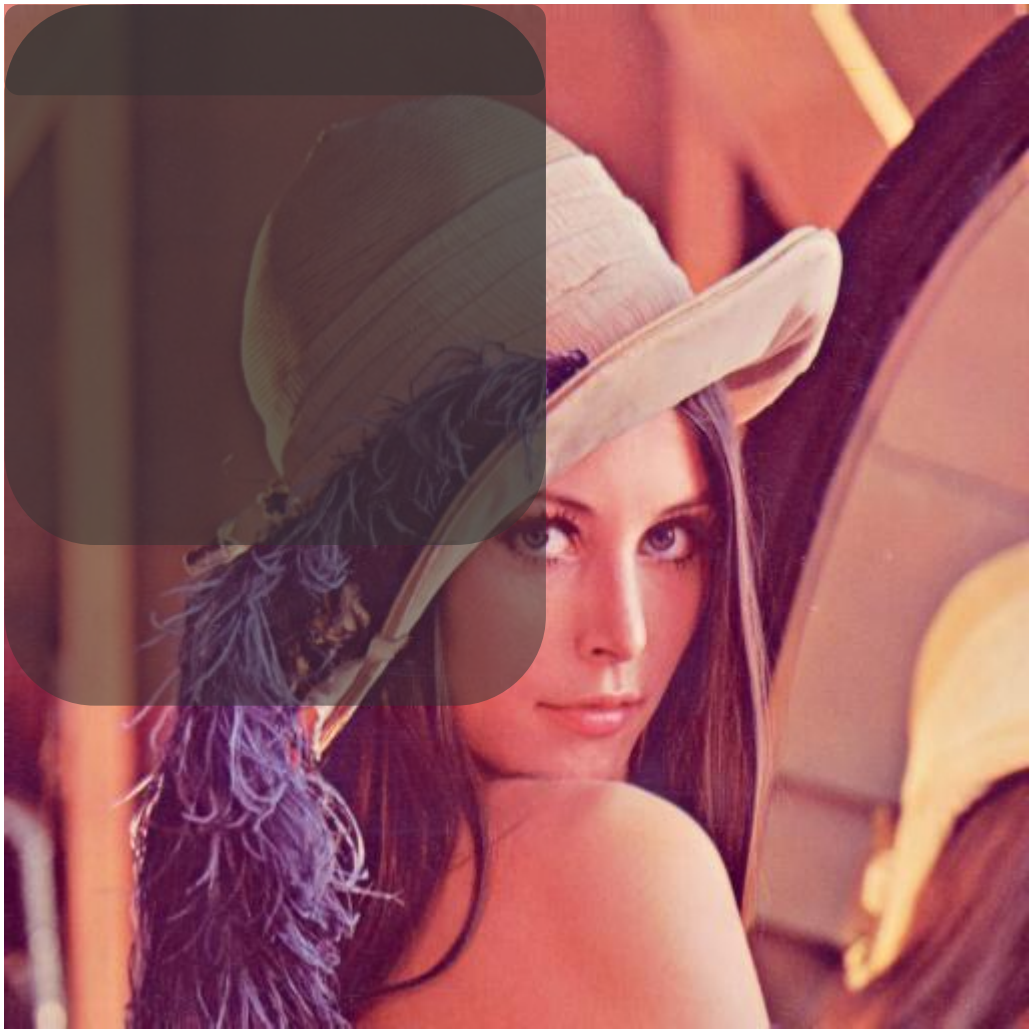
セパラブルフィルタブロック化 104

最初にここだけぼかす



セパラブルフィルタブロック化 105

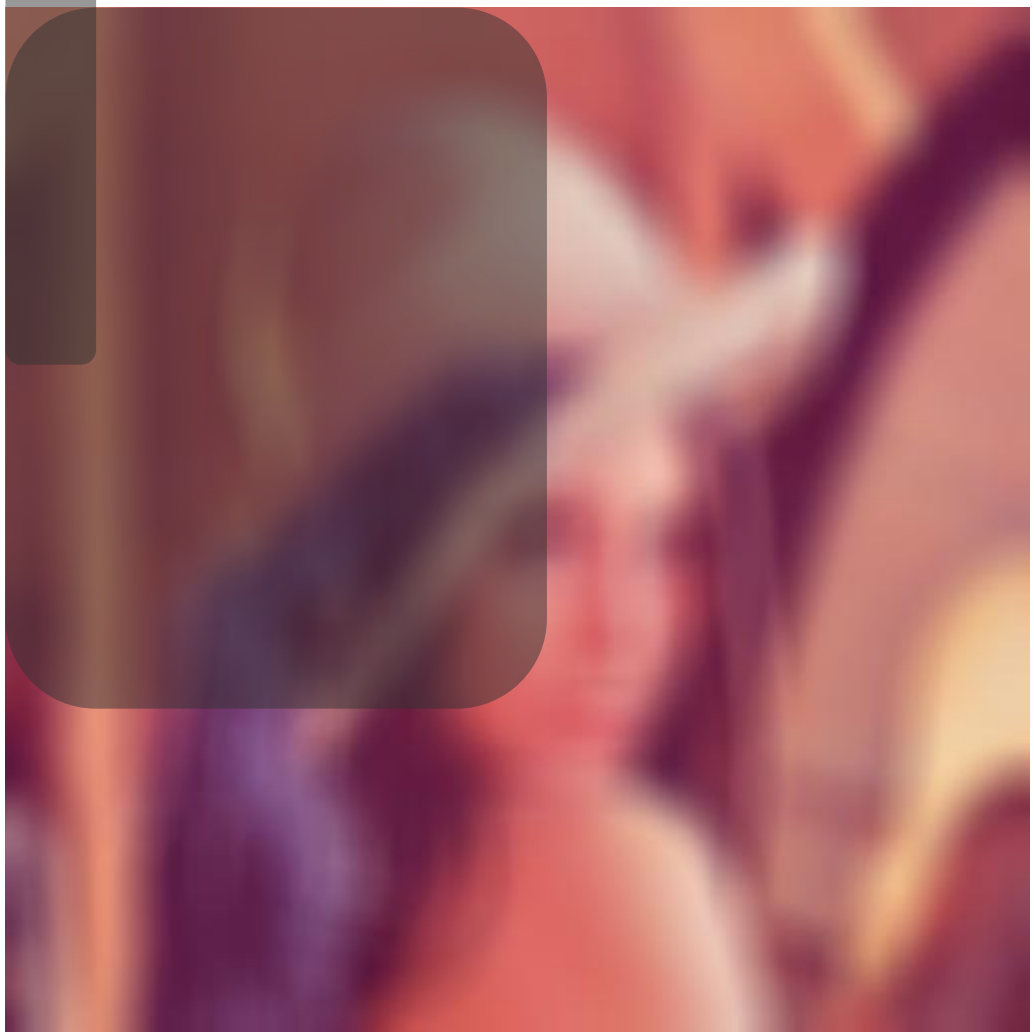
まず水平フィルタ
ただし縦に広めに
ぼかす



セパラブルフィルタブロック化

次に垂直フィルタ縦に広めにぼかした分，縦のカーネルがぼかす範囲をはみ出たとしてもOK

残りのブロックも同様に処理



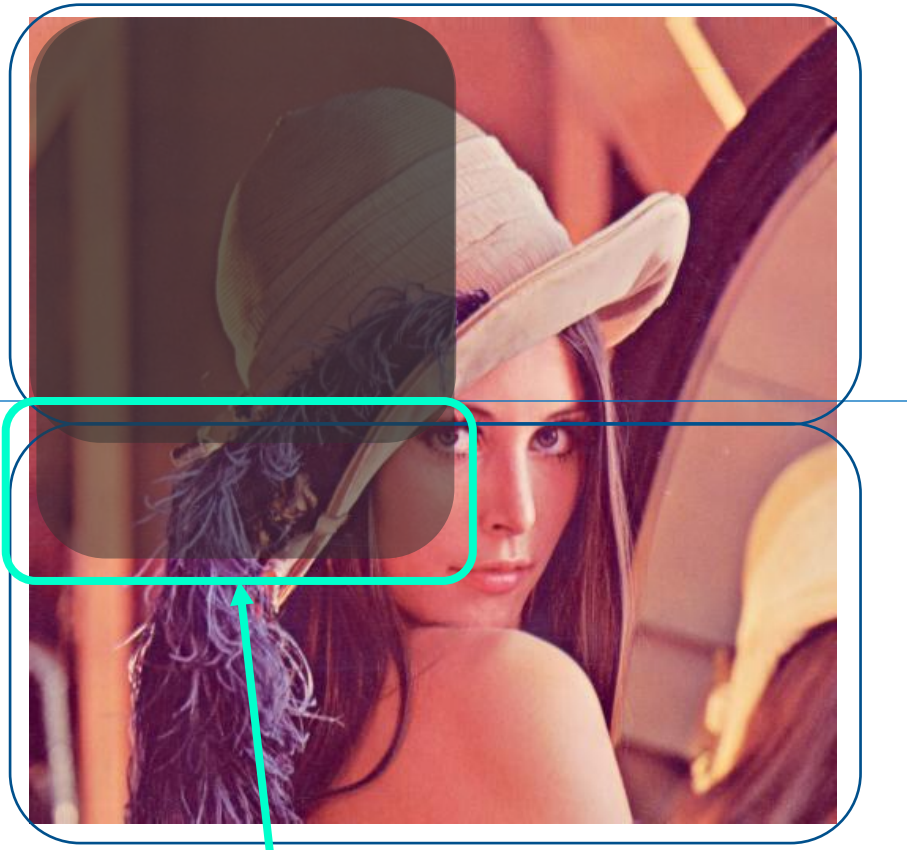
ブロック化の利点

- 2ステージ必要なのは同じ
- ただし，次のステージですく使うかつ記憶しておかないといけないメモリ量が少ない
  - メモリの時間的局所性が向上
- つまり，L2キャッシュなどの近いメモリに情報が載るため高速化
- タイリングにより解決

データアクセスの局所性

- 空間的局所性
  - あるデータがアクセスされた場合，その周囲のデータもアクセスされる可能性が高い
    - 画像処理では，メモリアクセスが連続になるように工夫していれば通常満たしている
- 時間的局所性
  - あるデータがアクセスされた場合，近いうちに再度そのデータがアクセスされる可能性が高い
    - 画像自体が大きい場合，ブロック分割して処理を集中させると高まる

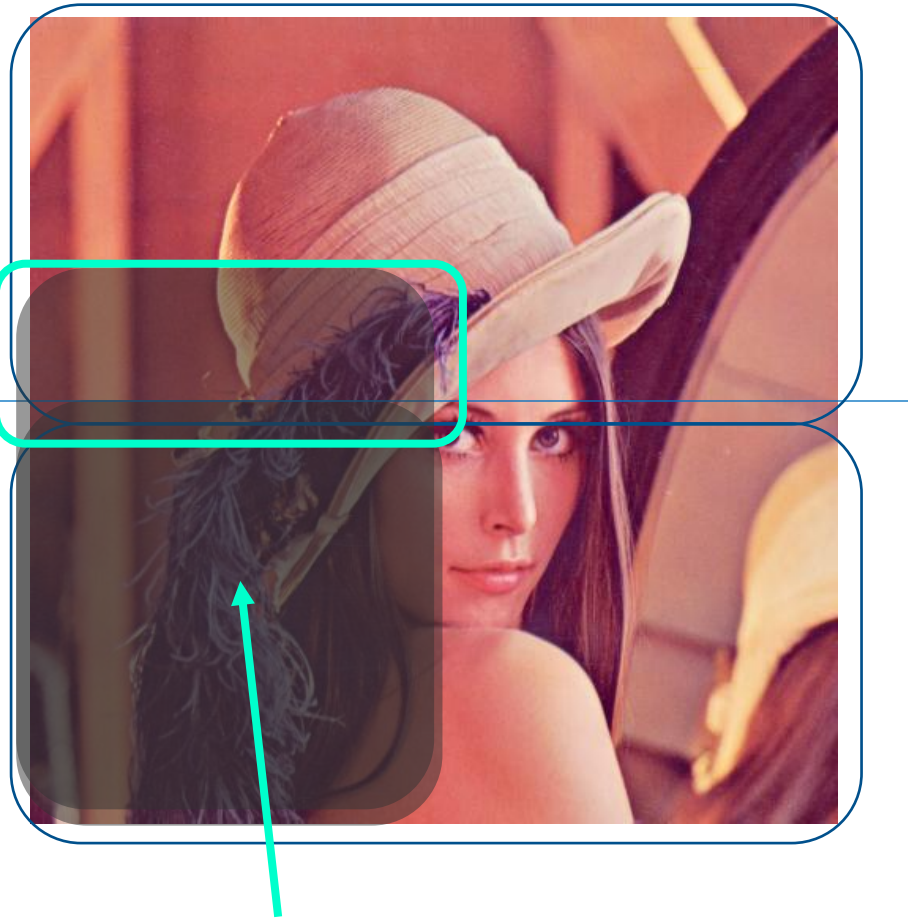
タイリングと並列化



- 並列化
  - 例えば上と下に画像分割
- ２ステージのため同期待ちが発生
  - 同期処理は非常に重たい

冗長な部分の水平演算結果は他のコアが担当予定.
この処理を待っていたら，同期処理が重たい

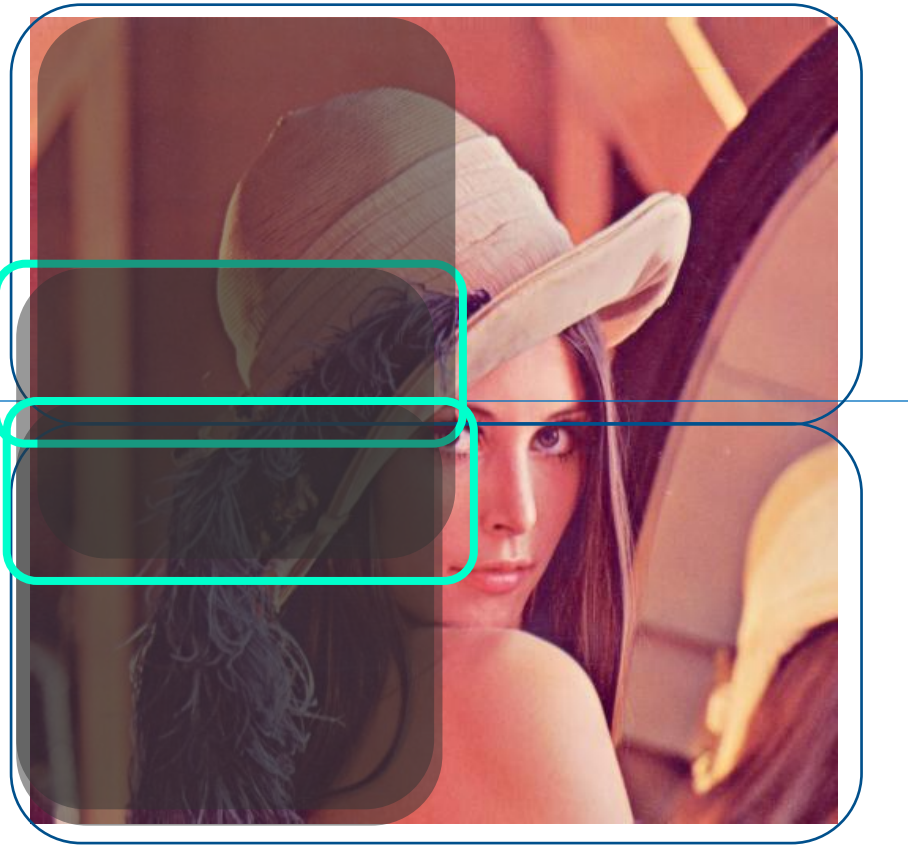
タイリングと並列化



一方で下側も上側の計算領域が必要
この処理を待っていたら，同期処理が非常に重たい

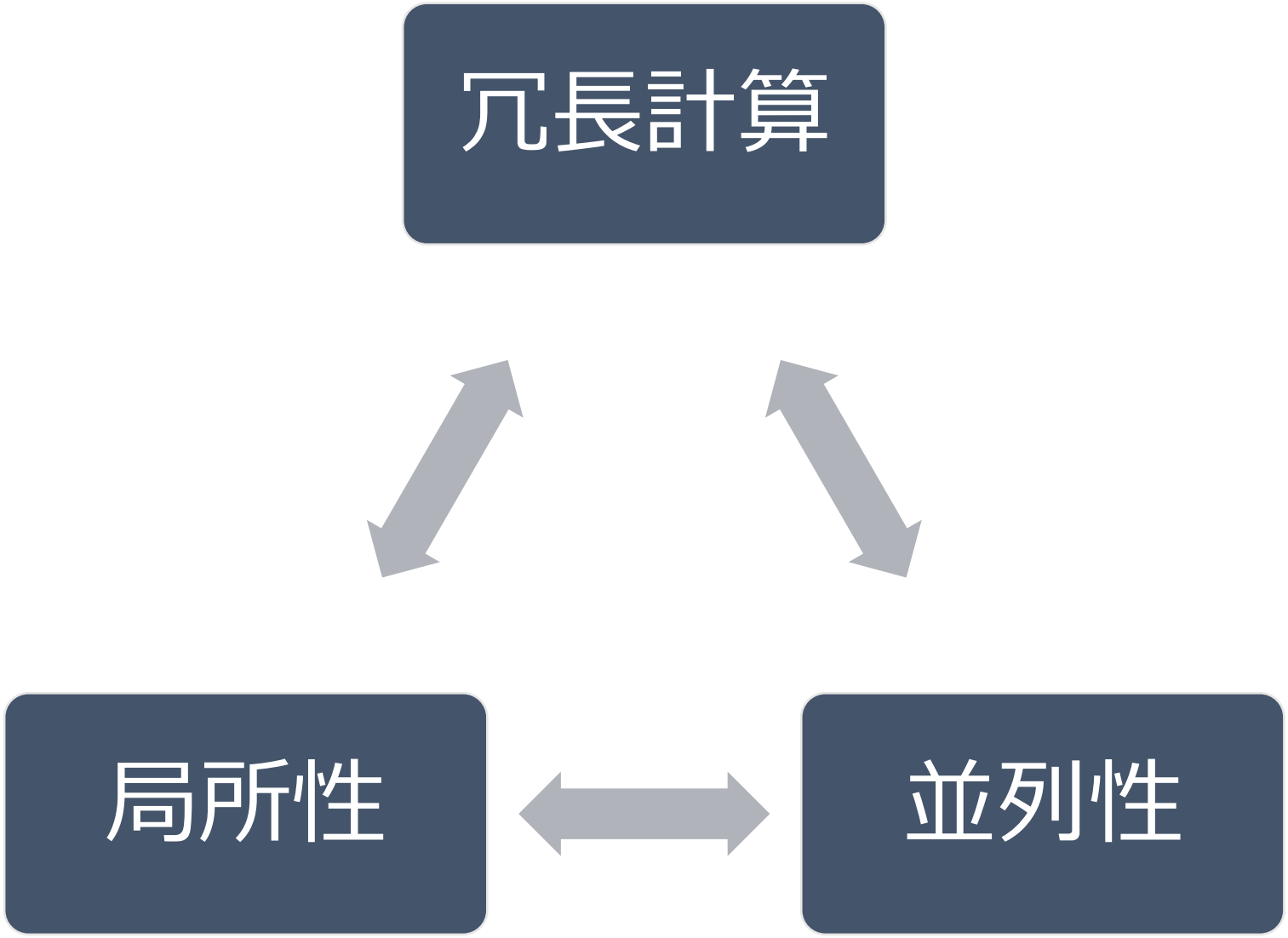
- 並列化
  - 例えば上と下に画像分割
- ２ステージのため同期待ちが発生
  - 同期処理は非常に重たい

タイリングと並列化



- 並列化
  - 例えば上と下に画像分割
- 冗長な領域を各コアで独自計算
  - 同じ場所を同じ計算が発生するため演算量が増加
- 同期を待たなくても並列化可能

画像処理計算のトレードオフ



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

- 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.

Halide (1/3)

- 特徴
  - C++に組み込む形で使う関数型プログラミング言語
  - SSE，AVX，ARM, Cuda, OpenCLに出力可能
  - 画像処理プログラムを並列化・ベクトル化済みのコードとして出力
  - OpenCVのディープラーニング用のモジュールに搭載

Halide

- 何を計算するのか（アルゴリズム：Func），それをいつどこでどのように実行するのか（Schedule）定義する関数型プログラミング言語
- 計算するアルゴリズムをFuncで定義したら，それをどの順番で計算し，どれをベクトル化し，どう並列化して，どのようにタイルを区切るかをScheduleで定義可能
- C++などと違って，すべての演算を細かくすべて書く必要がない
- Forループを書かなくてよいCudaプログラムに近い

Halideによる移動平均フィルタ

(a) Clean C++ : 9.94 ms per megapixel

```
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;
}
```

(c) Halide : 0.90 ms per megapixel

```
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.chunk(x).vectorize(x, 8);

 return blurred;
}
```

tiling

multithreading

(b) Fast C++ (for x86) : 0.90 ms per megapixel

```
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);
}}}}}
```

vectorization

fusion

Halideのチュートリアル

- フィックスターズ
  - Halide による画像処理プログラミング入門
    - https://www.halide2fpga.com/halide_programming_tutorial/

Halide2FPGA

- そろそろノイマン型コンピュータは限界？
  - メモリ帯域とCPUの差が限界.
  - CPU中の回路もキャッシュばかりで演算器載っていない
- カメラパイプラインなどのよく使われてきた画像処理チップ以外にも，ディープラーニング専用チップなど必要な計算に特化したものも多く登場
- FPGAで任意のプログラムを，必要な演算とデータの転送を最適にかつ最大に並列化して実行可能な有効な回路を！

Halide2FPGA

- Halide2FPGA
  - HalideがFPGAをバックエンドに持てるように拡張
  - GENESISコンパイラによって，Halideコードを Vivado HLS向けC/C++コードに変換

Helium

- レガシーなコード（アセンブリバイナリ）をHalideに自動変換
- 最新で最適なコードに再変換
- http://projects.csail.mit.edu/helium/

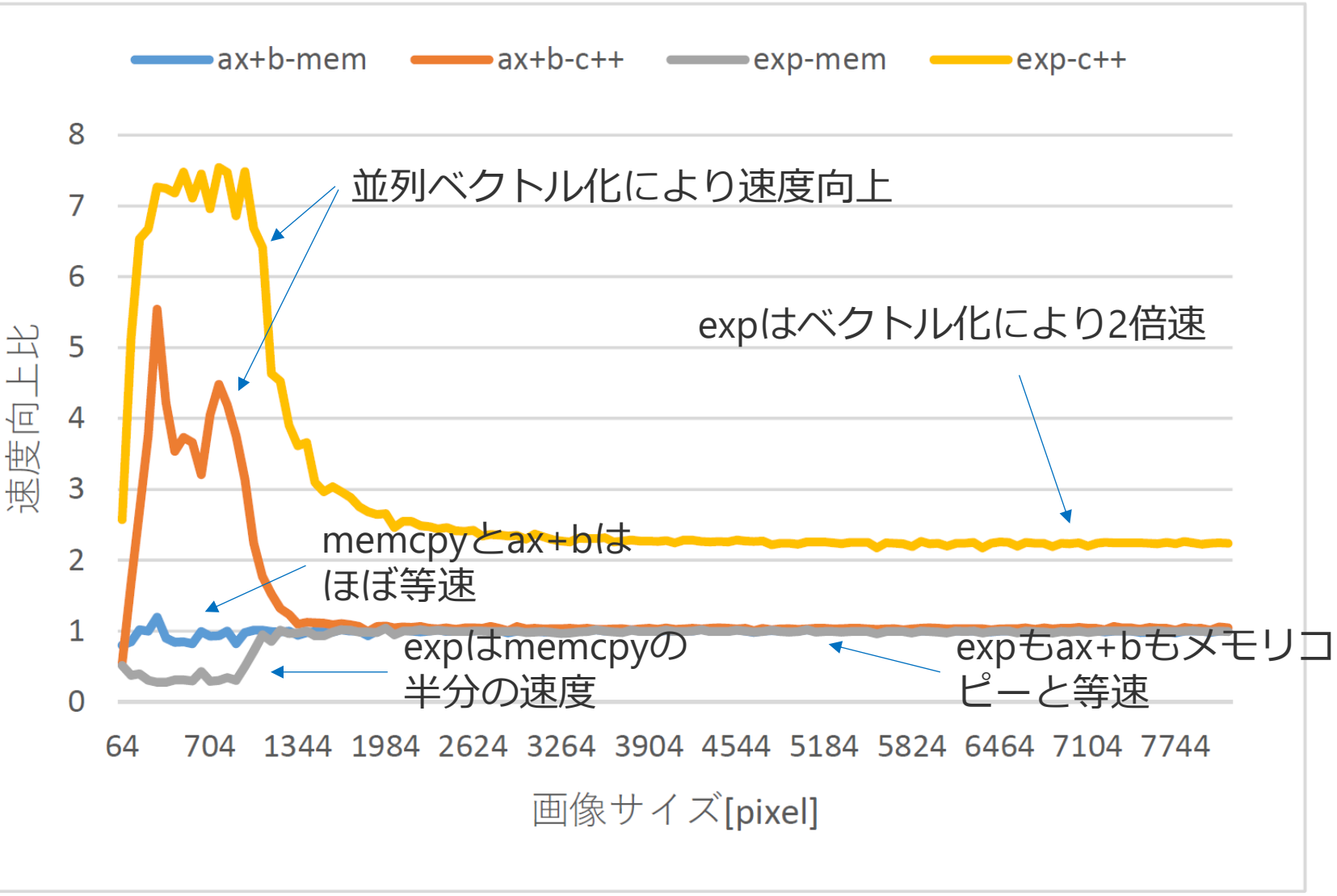
- Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, Saman Amarasinghe, “Helium: Lifting High-Performance Stencil Kernels from Stripped x86 Binaries to Halide DSL Code,” in Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2015.

実例

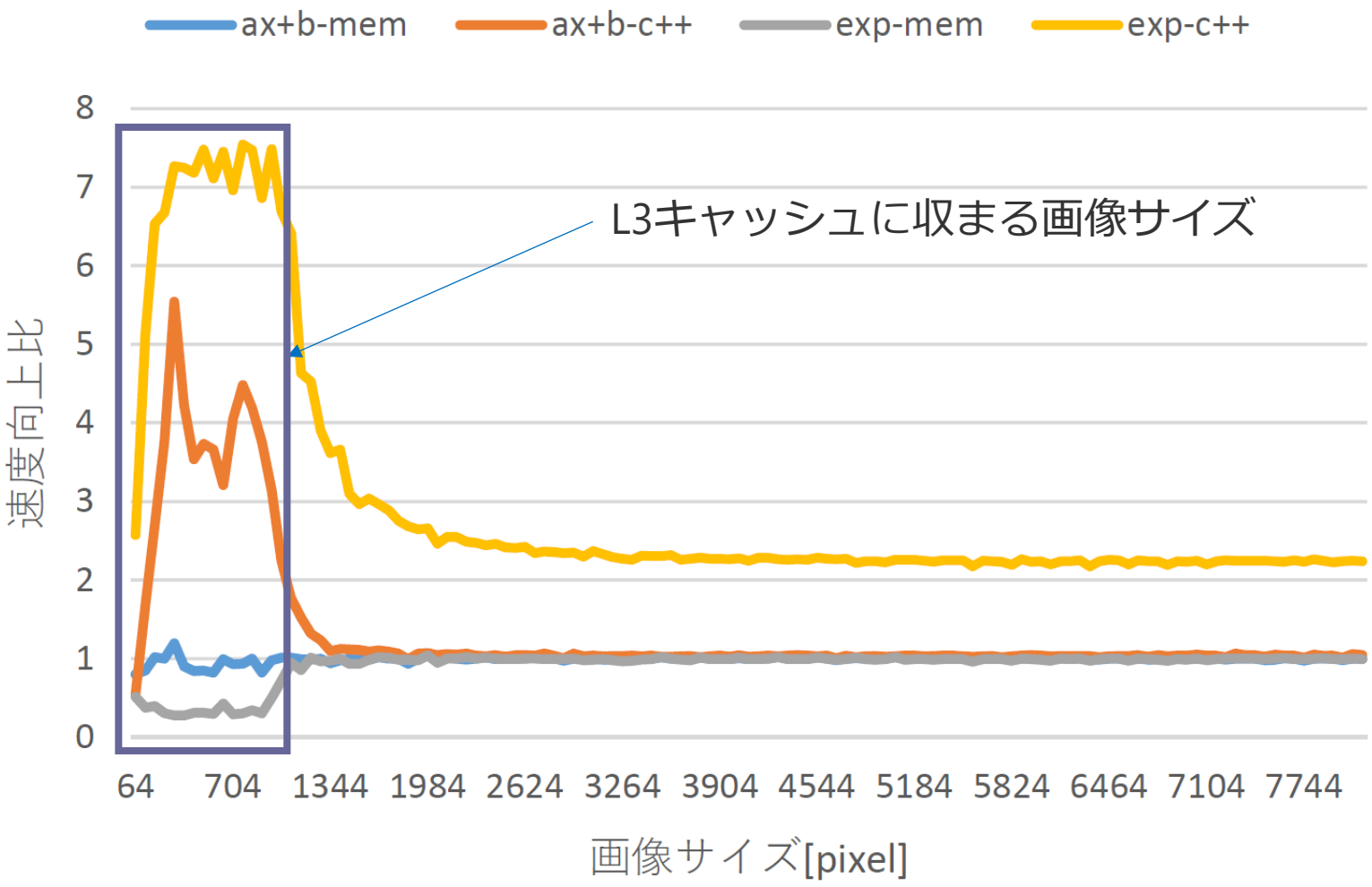
ポイントオペレータ

- 特徴
  - 画像を読み込んで画素単位に何か演算をするマップ処理
  - 演算強度が低い
- 比較対象
  - メモリコピー
    - 並列演算，ベクトル演算で自前実装
  - ax+b：
    - 乗算１回，加算１回（積和１回）
  - 指数計算
    - 10次テイラー展開で計算コストを上げる（乗算18回，加算10回）
- 比較方法
  - C++実装（並列化有り）とベクトル化・並列化の速度向上比
  - メモリコピーとの速度比較

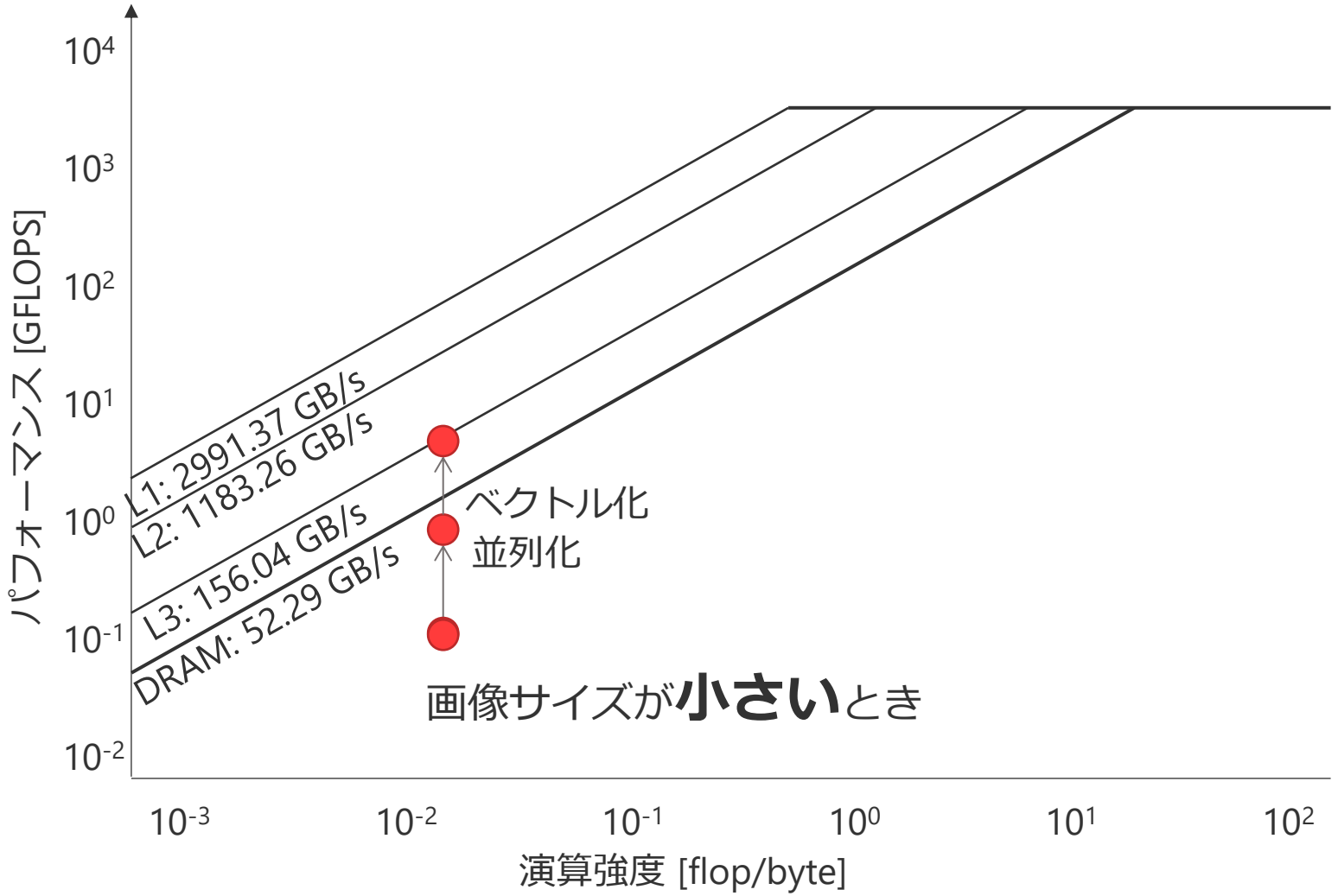
ポイントオペレータ結果



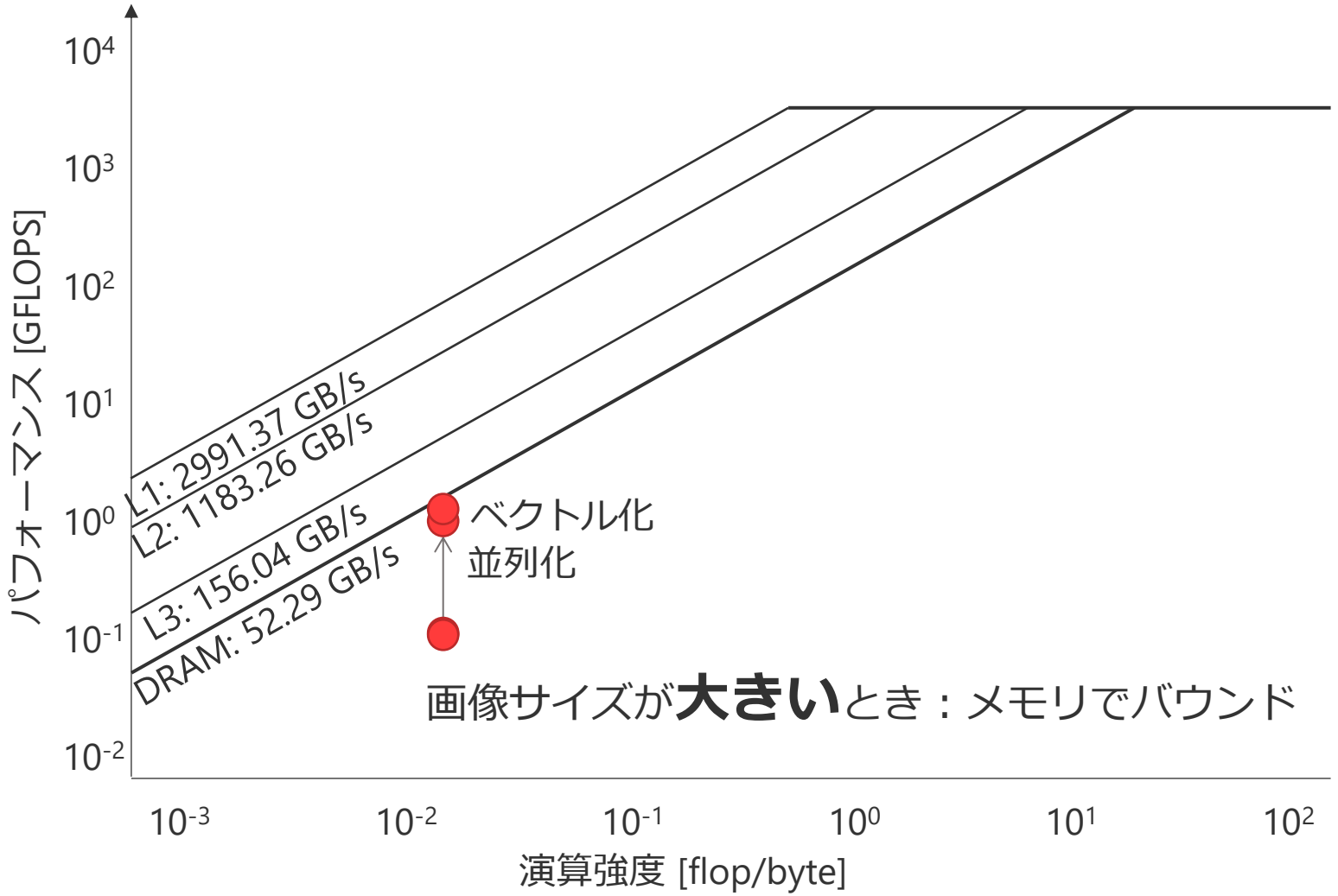
ポイントオペレータ結果



ルーフラインによる解析



ルーフラインによる解析



ポイントオペレータ結果

- ax+bは演算が入っているがメモリコピーと速度は同じ
- ax+bとexp(x)は10倍以上の演算量の差があるが実際は2倍しかない
- これらはほとんどメモリ律速しているため

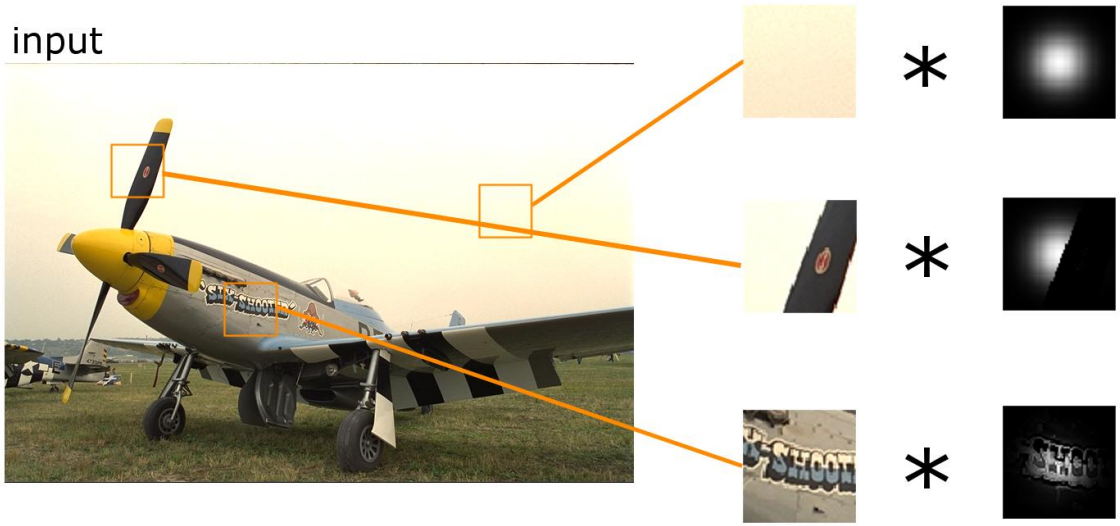
バイラテラルフィルタ

- 代表的なエッジ保存平滑化フィルタ
- 注目画素との距離と色の差で重みを決定

空間ガウス　　色ガウス

$$J_p = \frac{1}{N} \sum_{q \in \omega_p} \exp\left(\frac{\|p-q\|^2}{-2\sigma_s}\right) \exp\left(\frac{\|I_p - I_q\|^2}{-2\sigma_r}\right) I_q$$

convolution



FIRフィルタのベクトル化

J，I：出力画像，入力画像

width：画像の横幅，height：画像の縦幅，

rj：フィルタカーネルの縦幅,ri：フィルタカーネルの横幅

ss：空間ガウスシグマ，sr：レンジガウスシグマ

```
For(j=0; j<height; j++)
    For(i=0; i<width; i++)
        For(kj=-rj; kj<rj; kj++)
            For(ki=-ri; ki<ri; ki++)
                J[j][i]=exp((kj+ji)^2/-2ss) exp((I[j+kj][i+ki]- I[j][i])^2/-2sr) I[j+kj][i+ki];
```

FIRフィルタのベクトル化


```
For(j=0; j<height; j++)
    For(i=0; i<width; i+=4)
        For(kj=-rj; kj<rj; kj++)
            For(ki=-ri; ki<ri; ki++)
```

同時計算

$$J[j][i+0]=\exp((kj+ji)^2/-2ss)\ \exp((I[j+kj][i+ki+0]-I[j][i+0])^2/-2sr)\ I[j+kj][i+ki+0];$$

$$J[j][i+1]=\exp((kj+ji)^2/-2ss)\ \exp((I[j+kj][i+ki+1]-I[j][i+1])^2/-2sr)\ I[j+kj][i+ki+1];$$

$$J[j][i+2]=\exp((kj+ji)^2/-2ss)\ \exp((I[j+kj][i+ki+2]-I[j][i+2])^2/-2sr)\ I[j+kj][i+ki+2];$$

$$J[j][i+3]=\exp((kj+ji)^2/-2ss)\ \exp((I[j+kj][i+ki+3]-I[j][i+3])^2/-2sr)\ I[j+kj][i+ki+3];$$

画素操作の x 座標をループアンローリングしてベクトル長４でベクトル化

テーブル参照

- 事前に計算し，テーブル参照すればexpなどの超越関数は計算しなくてよい
- indexは正数でないと困るので，入力の絶対値を取る
  - 二乗も除算も乗算もexp計算もいらない代わりに絶対値が必要

$$EXP[|n|] = \exp(-\frac{n^2}{2\sigma})$$

テーブル参照とベクトル化


```
For(j=0; j<height; j++)
    For(i=0; i<width; i+=4)
        For(kj=-rj; kj<rj; kj++)
            For(ki=-ri; ki<ri; ki++)
```

同時計算

$$J[j][i+0]=EXP(|kj+ji|)\ \exp((I[j+kj][i+ki+0]-I[j][i+0])^2/-2sr)\ I[j+kj][i+ki+0];$$

$$J[j][i+1]=EXP(|kj+ji|)\ \exp((I[j+kj][i+ki+1]-I[j][i+1])^2/-2sr)\ I[j+kj][i+ki+1];$$

$$J[j][i+2]=EXP(|kj+ji|)\ \exp((I[j+kj][i+ki+2]-I[j][i+2])^2/-2sr)\ I[j+kj][i+ki+2];$$

$$J[j][i+3]=EXP(|kj+ji|)\ \exp((I[j+kj][i+ki+3]-I[j][i+3])^2/-2sr)\ I[j+kj][i+ki+3];$$

空間ガウスをテーブル化．スカラで位置の絶対値を計算しテーブル引き後，ベクトルに単一の値を代入

空間ガウスはカーネルの位置kj,kiだけに依存し，通常画像よりも大きくない
画素位置ごとのテーブルが計算自体いらない

テーブル参照とベクトル化


```
For(j=0; j<height; j++)
    For(i=0; i<width; i+=4)
        For(kj=-rj; kj<rj; kj++)
            For(ki=-ri; ki<ri; ki++)
                J[j][i+0]=GS[kj][ji]exp((I[j+kj][i+ki+0]- I[j][i+0])^2/-2sr) I[j+kj][i+ki+0];
                J[j][i+1]=GS[kj][ji]exp((I[j+kj][i+ki+1]- I[j][i+1])^2/-2sr) I[j+kj][i+ki+1];
                J[j][i+2]=GS[kj][ji]exp((I[j+kj][i+ki+2]- I[j][i+2])^2/-2sr) I[j+kj][i+ki+2];
                J[j][i+3]=GS[kj][ji]exp((I[j+kj][i+ki+3]- I[j][i+3])^2/-2sr) I[j+kj][i+ki+3];
```

テーブル引きした同一の値を入れるだけ

画素の差分とexp計算

テーブル参照とベクトル化


```
For(j=0; j<height; j++)
    For(i=0; i<width; i+=4)
        For(kj=-rj; kj<rj; kj++)
            For(ki=-ri; ki<ri; ki++)
                J[j][i+0]=GS[kj][ji]exp((D[j+kj][i+ki+0])^2/-2sr) I[j+kj][i+ki+0];
                J[j][i+1]=GS[kj][ji]exp((D[j+kj][i+ki+1])^2/-2sr) I[j+kj][i+ki+1];
                J[j][i+2]=GS[kj][ji]exp((D[j+kj][i+ki+2])^2/-2sr) I[j+kj][i+ki+2];
                J[j][i+3]=GS[kj][ji]exp((D[j+kj][i+ki+3])^2/-2sr) I[j+kj][i+ki+3];
```

テーブル引きした同一の値を入れるだけ

差分をDとして表現
差分自体はベクトル計算可能

テーブル参照とベクトル化


```
For(j=0; j<height; j++)
    For(i=0; i<width; i+=4)
        For(kj=-rj; kj<rj; kj++)
            For(ki=-ri; ki<ri; ki++)
                J[j][i+0]=GS[kj][ji]EXP(D[j+kj][i+ki+0]|) I[j+kj][i+ki+0];
                J[j][i+1]=GS[kj][ji]EXP(D[j+kj][i+ki+1]|) I[j+kj][i+ki+1];
                J[j][i+2]=GS[kj][ji]EXP(D[j+kj][i+ki+2]|) I[j+kj][i+ki+2];
                J[j][i+3]=GS[kj][ji]EXP(D[j+kj][i+ki+3]|) I[j+kj][i+ki+3];
```

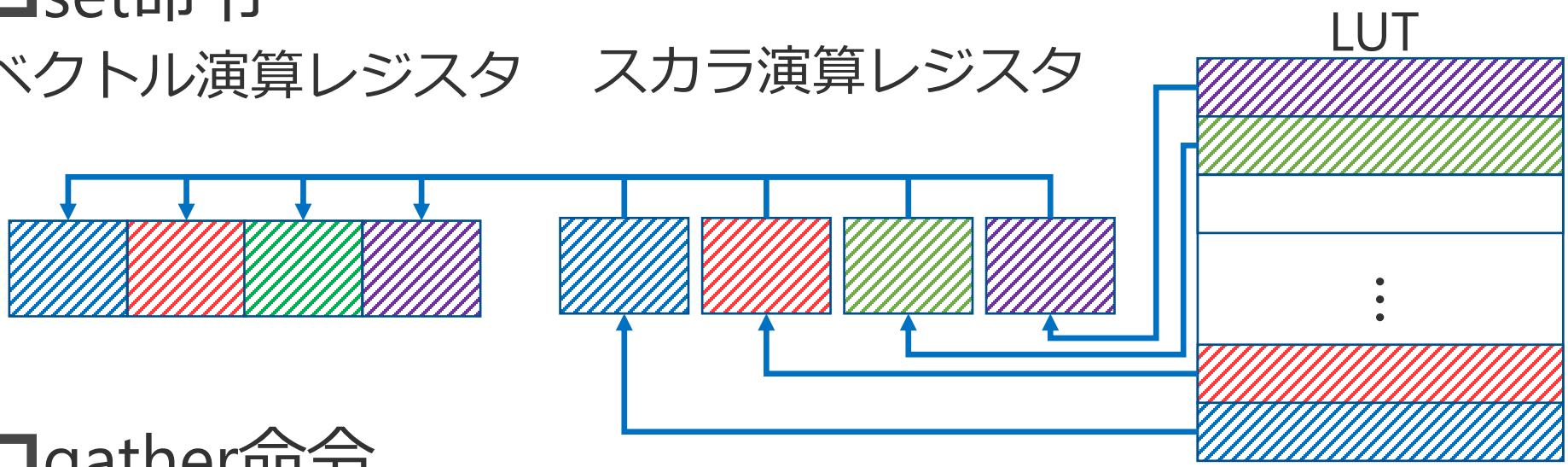
テーブル引きした同一の値を入れるだけ

差分の絶対値でテーブル参照

テーブル参照：setとgather

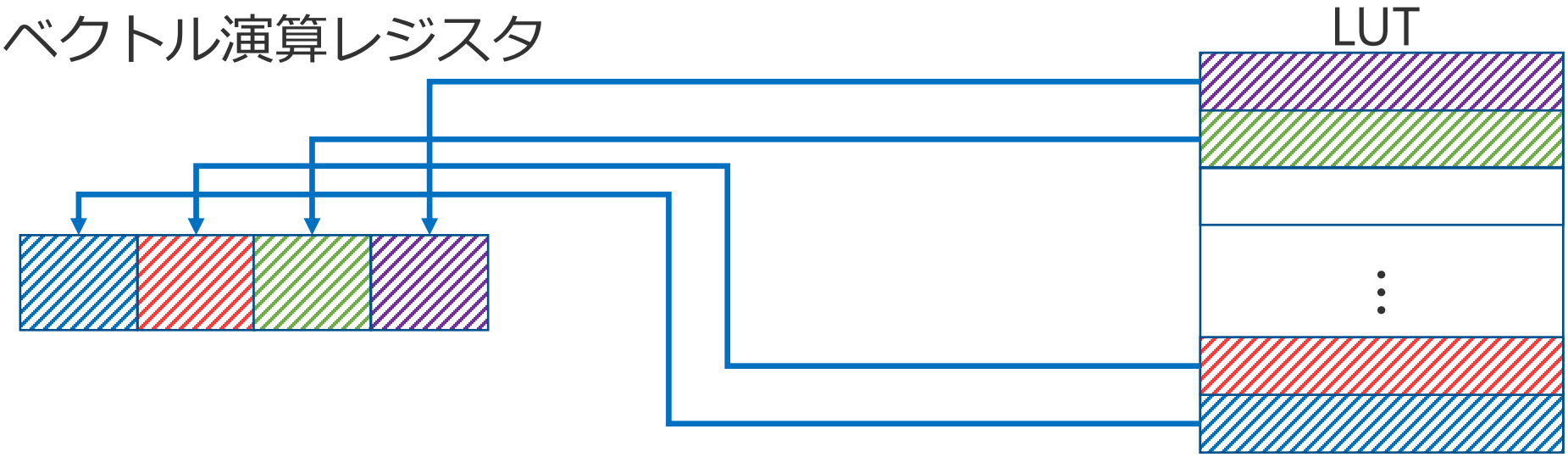
■set命令

ベクトル演算レジスタ　スカラ演算レジスタ



■gather命令

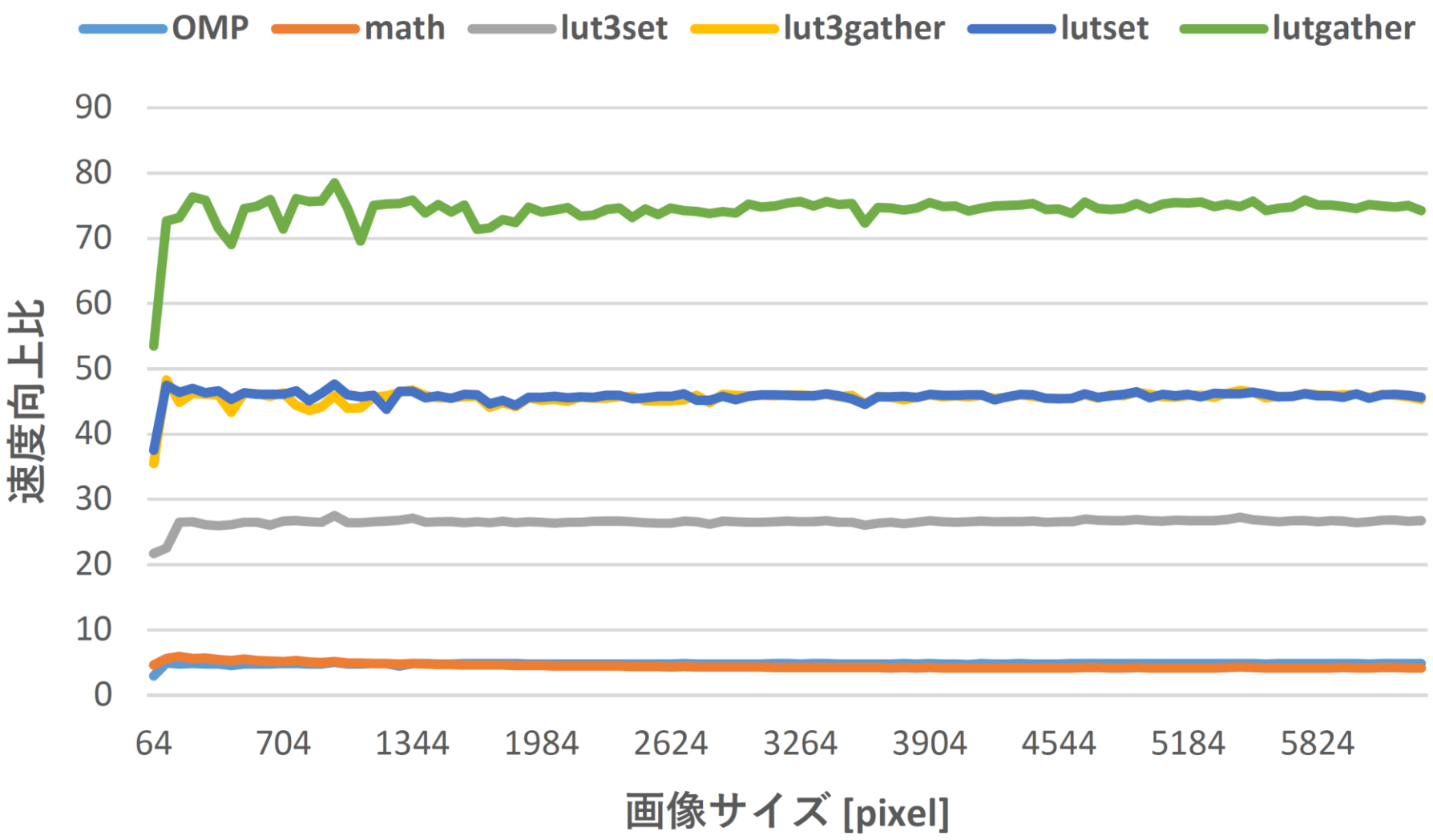
ベクトル演算レジスタ



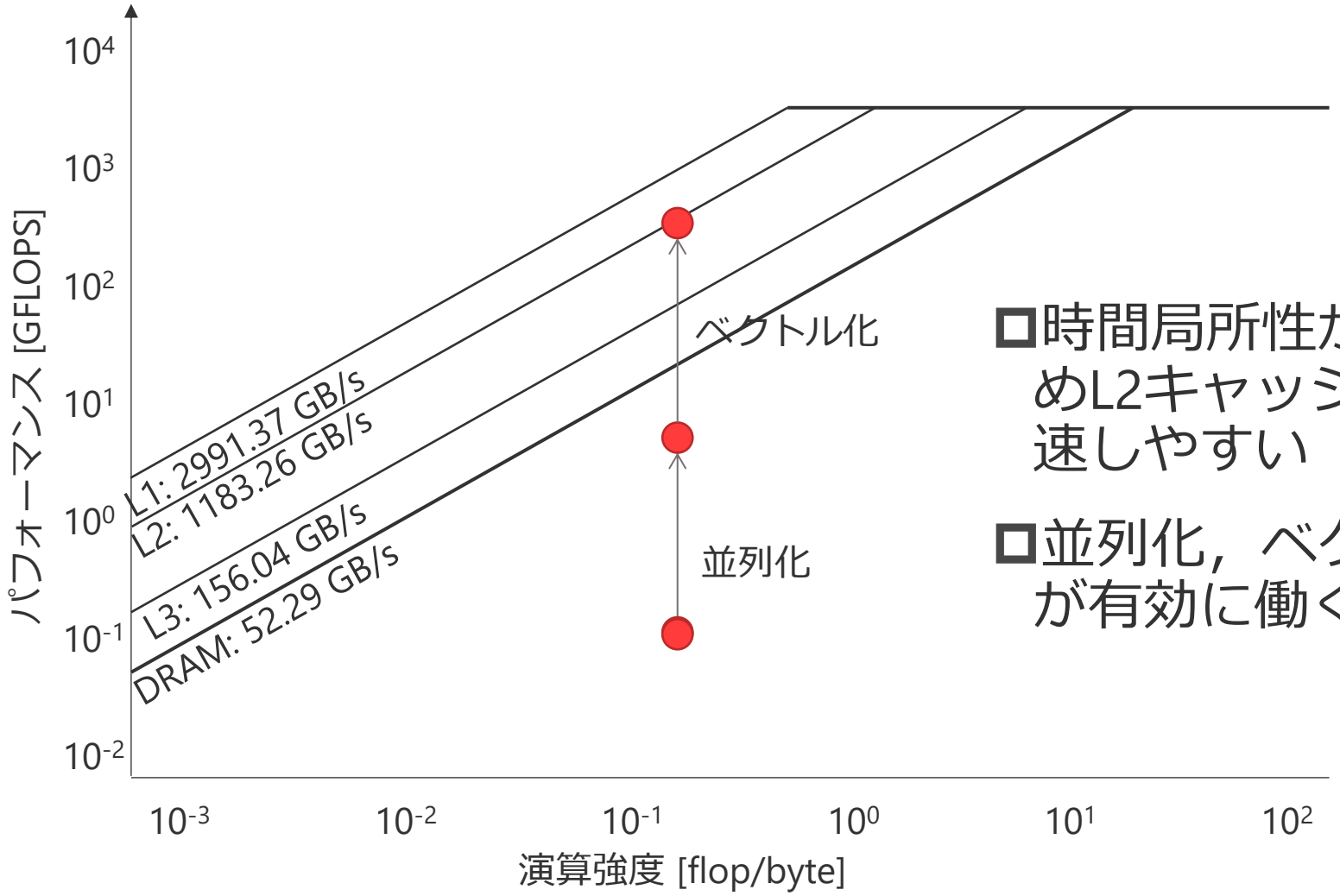
実験条件

- 並列化のみ
  - omp
- 並列化とベクトル化
  - math:指数計算をベクトル演算で実装
  - lut3set:RGBのLUTをそれぞれset x3
  - lut3gather:RGBのLUTをそれぞれgather x3
  - lutset:RGBのLUTを量子化してset x1
  - lutgather:RGBのLUTを量子化してgather x1

バイラテラルフィルタの結果



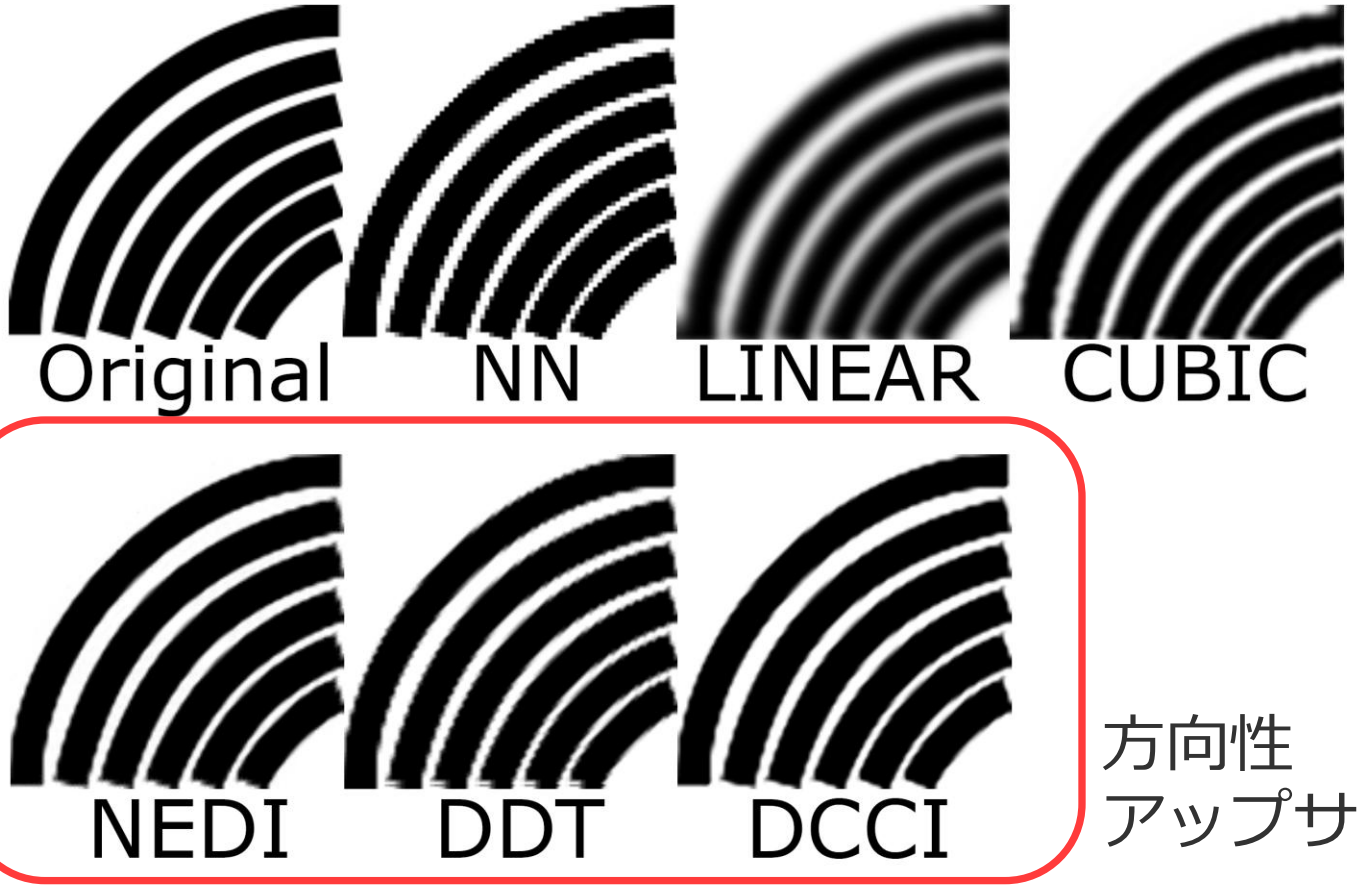
BFのルーフライン解析



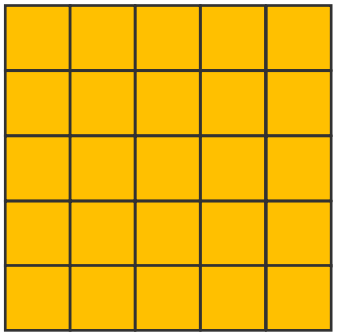
- 時間局所性が高いためL2キャッシュで律速しやすい
- 並列化，ベクトル化が有効に働く

方向性アップサンプル

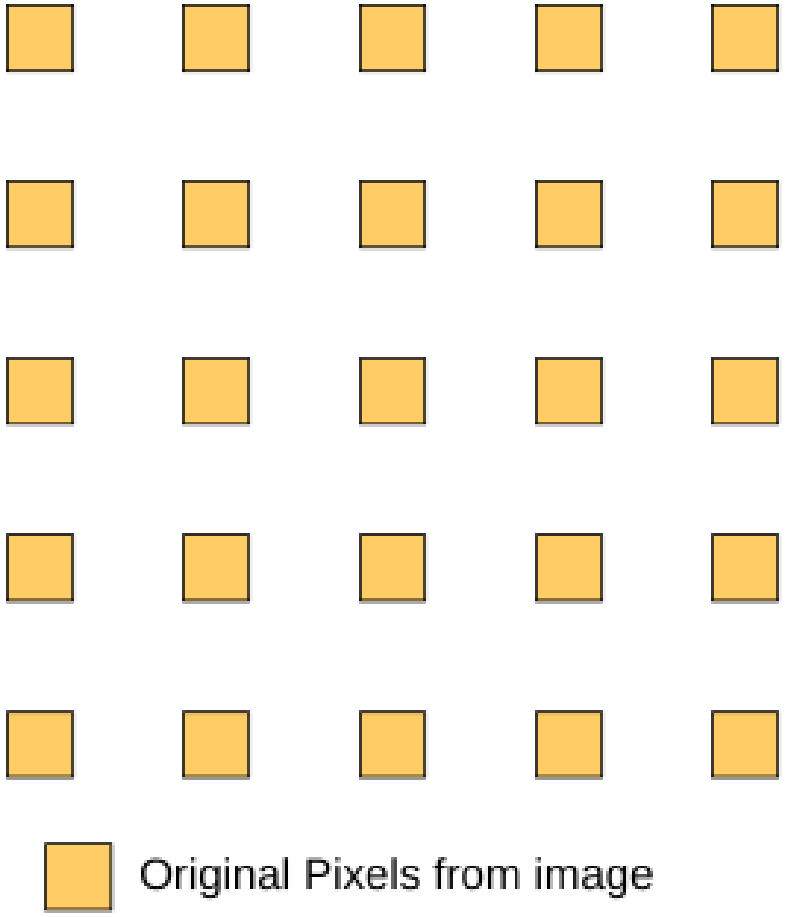
- 画像のアップサンプルをするときにエッジを考慮して段階的にアップサンプル



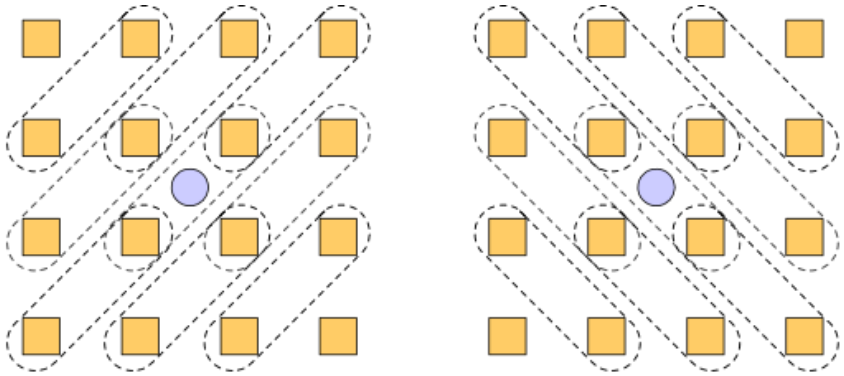
方向性アップサンプル



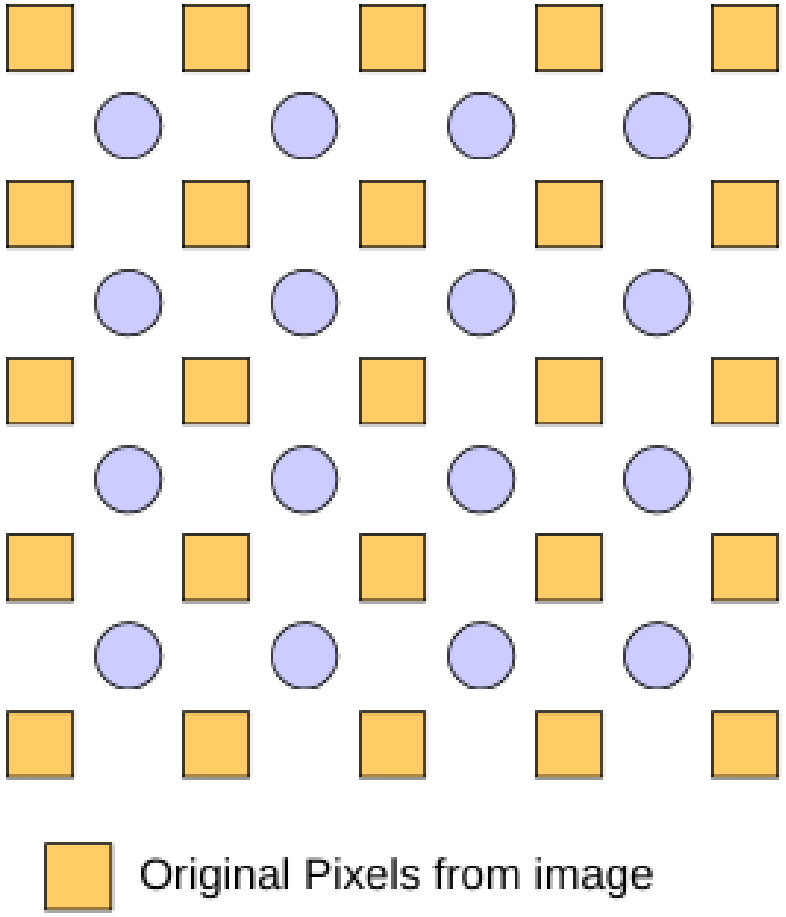
input image



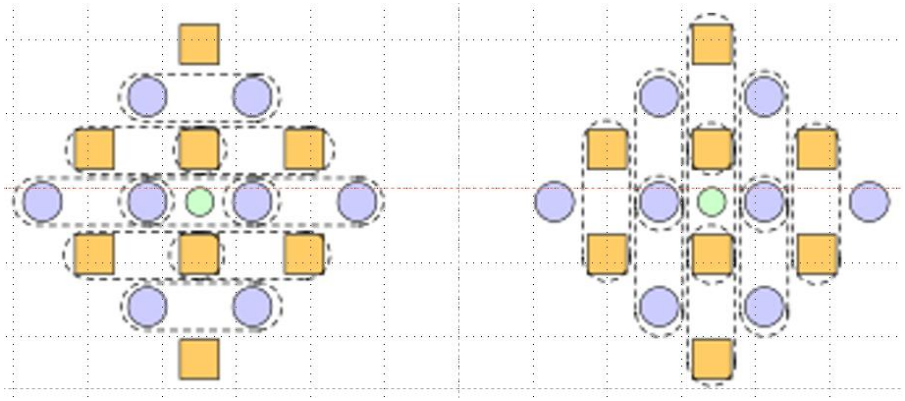
方向性アップサンプル



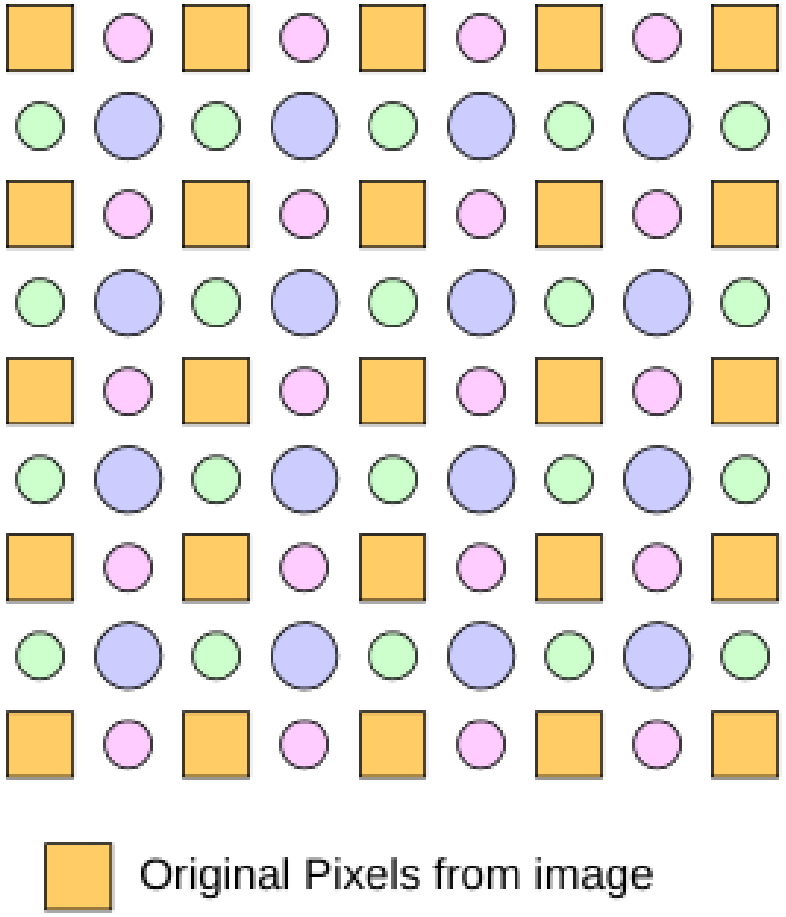
対角の画素のエッジの
強さを見て補間



方向性アップサンプル



補間画素も使った近傍画素のエッジの強さを見て補間



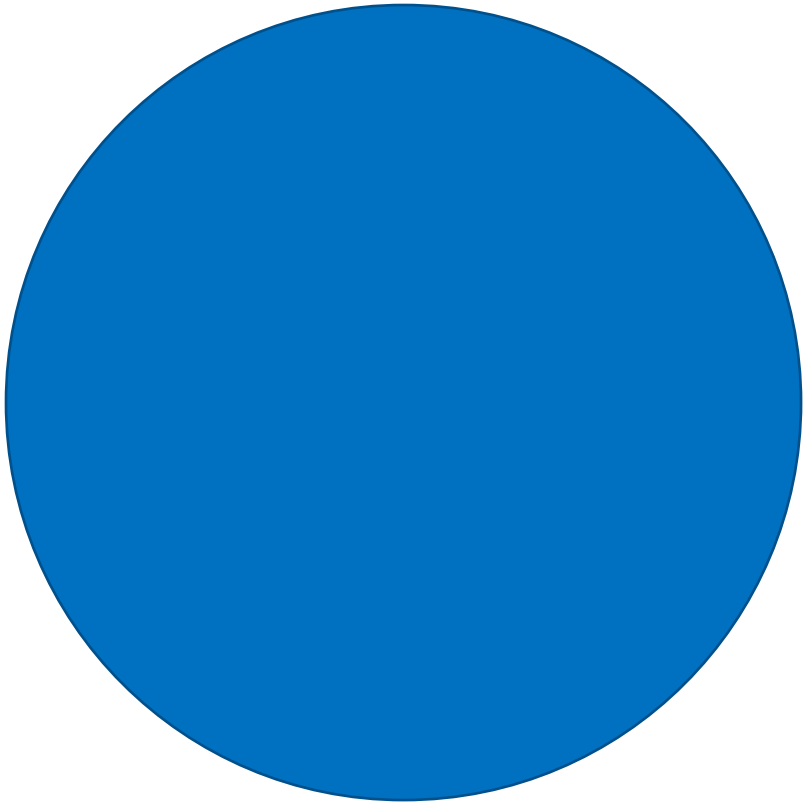
高速実装結果

- 2段階のアップサンプルをループヒュージョンして１つ統合
  - 768x512の画像アップサンプル
- C++実装
  - 59.12ms
- 並列化
  - 10.31ms
- ベクトル化
  - 0.52ms
- ループヒュージョン
  - **0.33ms**

※参考
キュービック補間（OpenCV実装）
0.4ms

セパラブルフィルタ

- 結果！



まとめ

まとめ

- CPUの歴史
  - 並列化だけでなくベクトル演算パフォーマンスも向上
  - メモリの速度の伸びは悪い
- 画像処理の並列化パターン
  - 並列画像処理プログラミングのデザインパターン
- 実例
  - コントラスト強調閾値処理など
  - バイラテラルフィルタ
  - 方向性アップサンプル

謝辞：本研究は科研費費JP17H01764の助成を受けた.

メモ

メモ

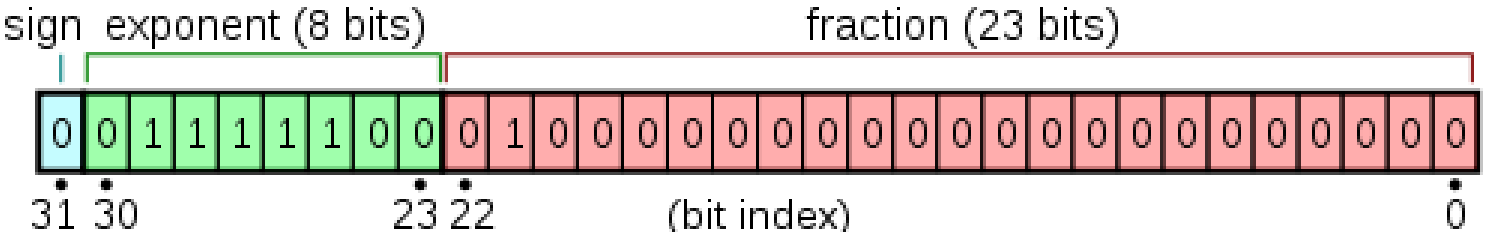
- PDEなどの演算回数以上にメモリアクセスが重たい最適化はメモリがボトルネックになって速度が低下

非正規化数について

- 浮動小数点演算は 0 付近の値を表現できない
- 非正規化数を使って精度が上がるが，演算速度は非常に遅くなる
  - 正規化数の演算は，ハードウェアで実装
  - 非正規化数の演算は，ソフトウェアで実装
- 精度がいらないなら，非正規化数を使わないようにすると速い

非正規化数とは

- IEEE754形式



- 正規化数

$$(-1)^{sign} \times 1.fraction \times 2^{exponet-127}$$

- 非正規化数

$$(-1)^{sign} \times 0.fraction \times 2^{0-127}$$

非正規化数とは（２／２）

- 正規化数は，ハードウェア実装
  - 正規化数は，ほぼ常に発生する
- 非正規化数は，ソフトウェア実装
  - 非正規化数は，あまり発生しない

非正規化数が発生すると，
パフォーマンスが著しく低下

非正規化数を発生させないことが重要

FTZとDAZ

- FTZ (Flush To Zero)
  - 浮動小数点演算において，演算結果が非正規化数の場合，0とする
- DAZ (Denormals Are Zero)
  - 浮動小数点への入力が，非正規化数の場合，0とする

- 命令のコントロールレジスタによって，設定可能
- 非正規化数が発生した場合の対処方法

FIRフィルタにおける非正規化数 155

- バイラテラルフィルタの場合

$$\boldsymbol{J_p} = \frac{1}{K_p}\sum_{\boldsymbol{q}\in\Omega}\omega_{\boldsymbol{p},\boldsymbol{q}}\boldsymbol{I_q}$$

$$\omega_{\boldsymbol{p},\boldsymbol{q}}^{bi} = \underbrace{\exp\left(\frac{-\|\boldsymbol{p}-\boldsymbol{q}\|_2^2}{2{\sigma_s}^2}\right)}_{\text{空間ガウシアン}}\underbrace{\exp\left(\frac{-\|\boldsymbol{I_p}-\boldsymbol{I_q}\|_2^2}{2{\sigma_r}^2}\right)}_{\text{値域ガウシアン}}$$

非正規化数が特に発生しやすい項

$$\exp\left(\frac{-\|\boldsymbol{I_p}-\boldsymbol{I_q}\|_2^2}{2{\sigma_r}^2}\right) = \exp\left(-\frac{\left(\boldsymbol{I_p^R}-\boldsymbol{I_q^R}\right)^2+\left(\boldsymbol{I_p^G}-\boldsymbol{I_q^G}\right)^2+\left(\boldsymbol{I_p^B}-\boldsymbol{I_q^B}\right)^2}{2{\sigma_r}^2}\right)$$

FIRフィルタにおける非正規化数 156

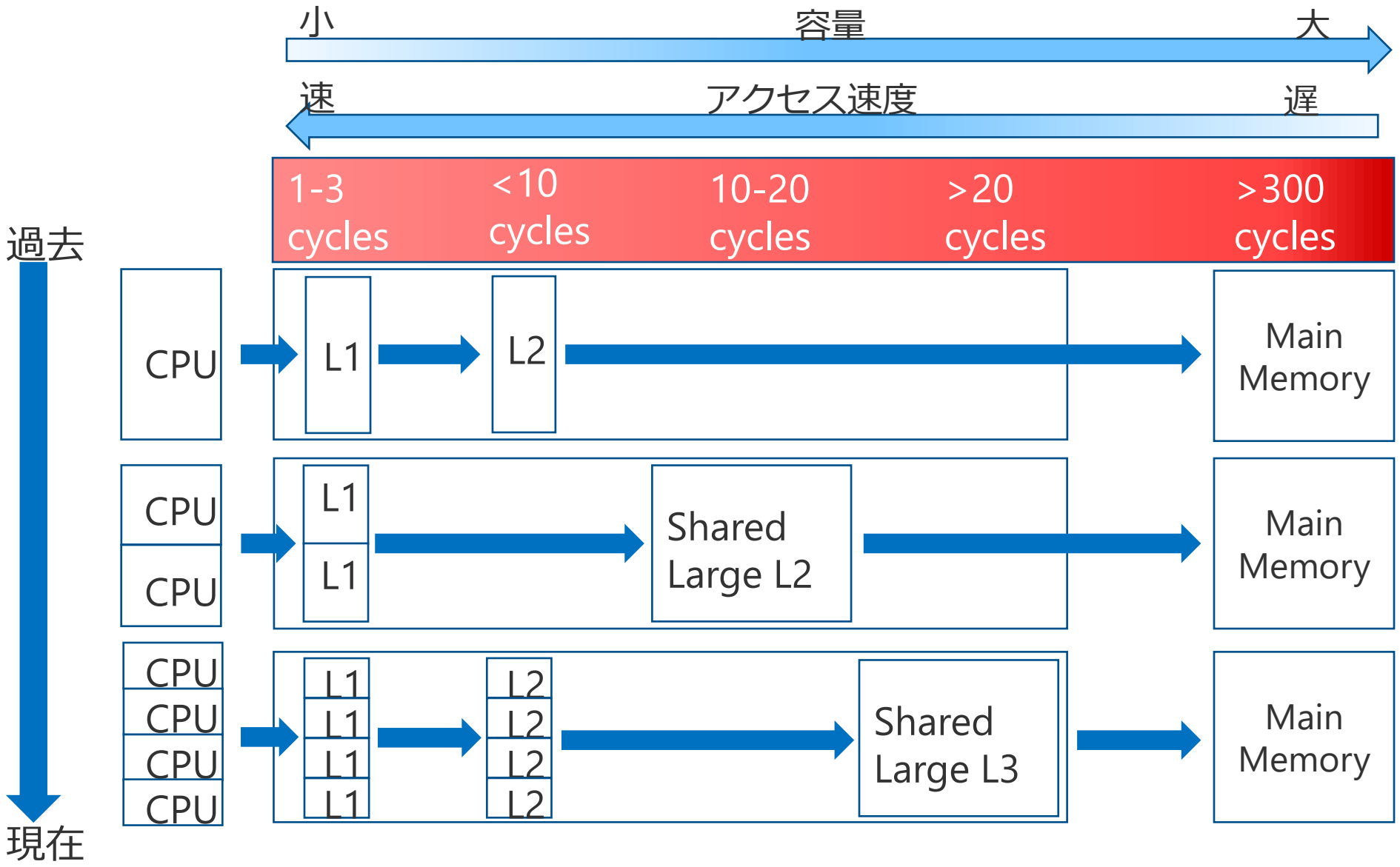
- バイラテラルフィルタの場合

$$\omega_{\boldsymbol{p},\boldsymbol{q}}^{bi} = \max\left(\exp\left(\frac{-\|\boldsymbol{p}-\boldsymbol{q}\|_2^2}{2{\sigma_s}^2}\right), \text{非正規化数でない数値}\right)$$
$$\max\left(\exp\left(\frac{-\|\boldsymbol{I_p}-\boldsymbol{I_q}\|_2^2}{2{\sigma_r}^2}\right), \text{非正規化数でない数値}\right)$$

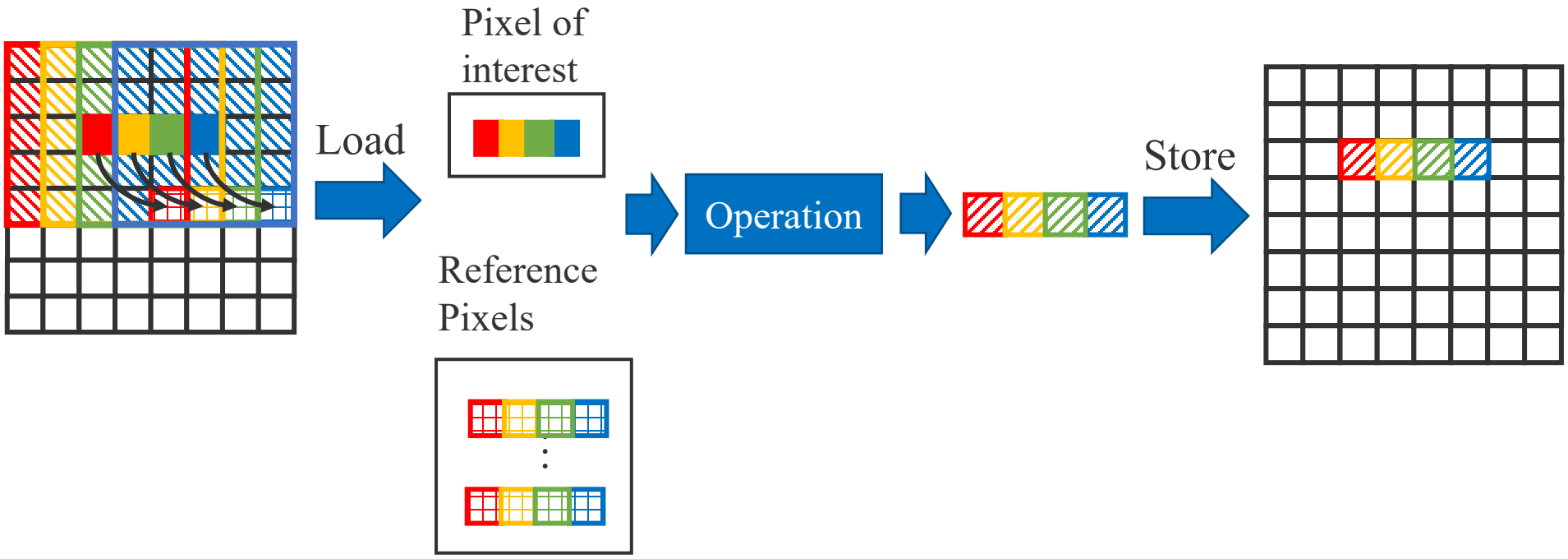
非正規化数の数値が発生しないように，クリップ

例えば，FLT_EPSILON= 1.192092896e-07F

キャッシュ階層とレイテンシ



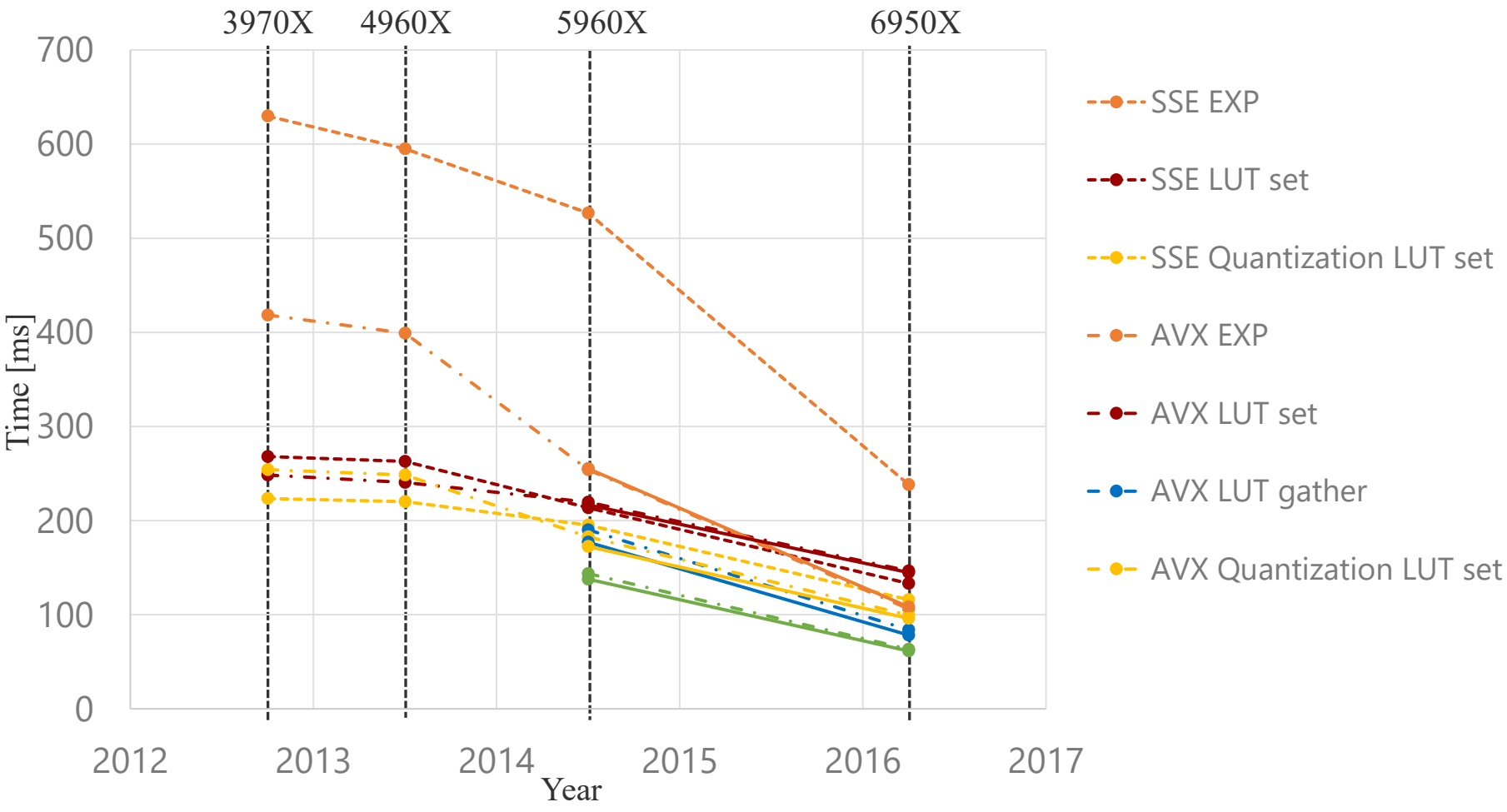
ステンシルのベクトル化



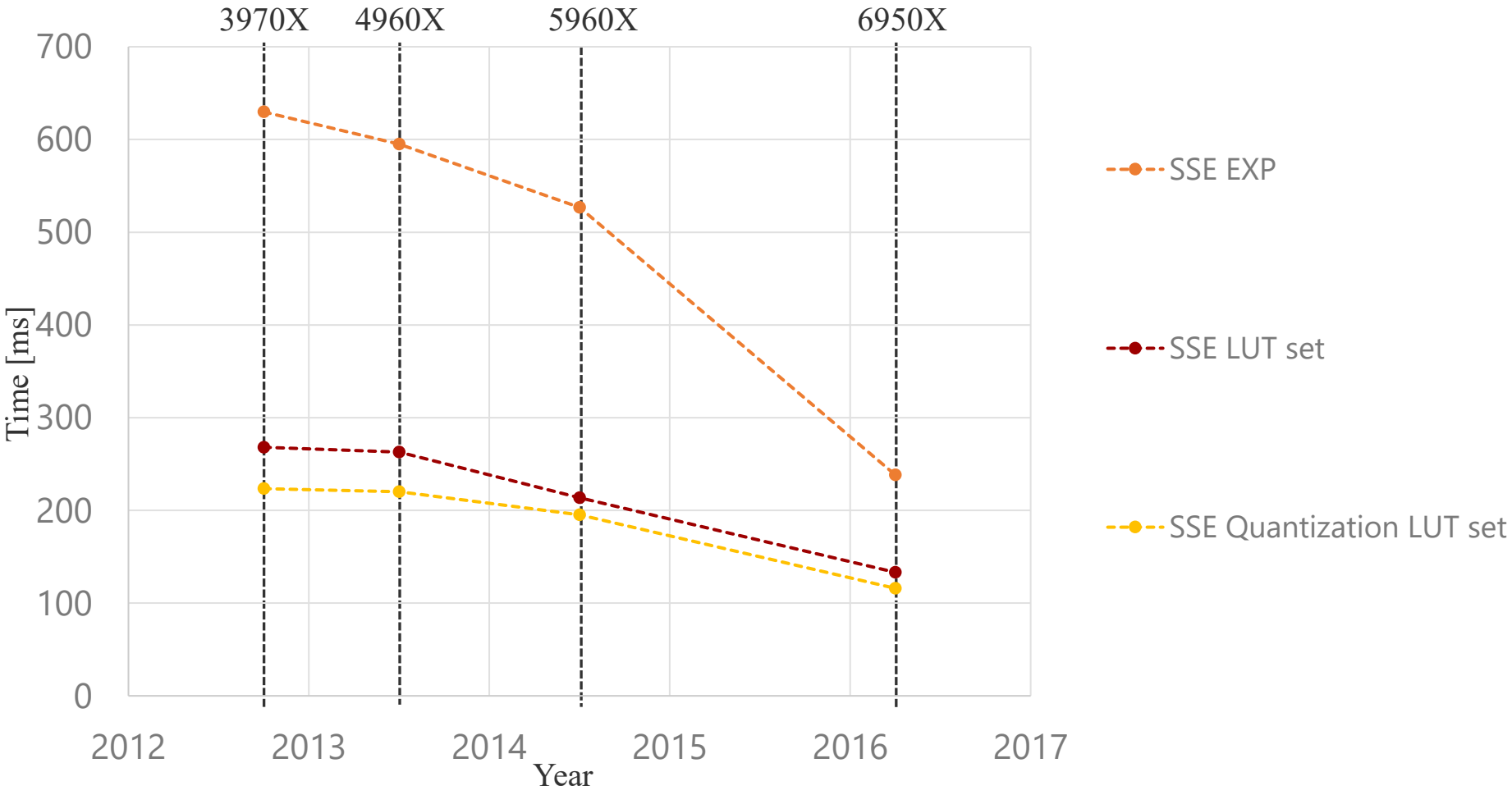
世代別BFとNLMの計算結果

- 2013年～2016年のPCを変えてBFとNLMの計算結果を比較

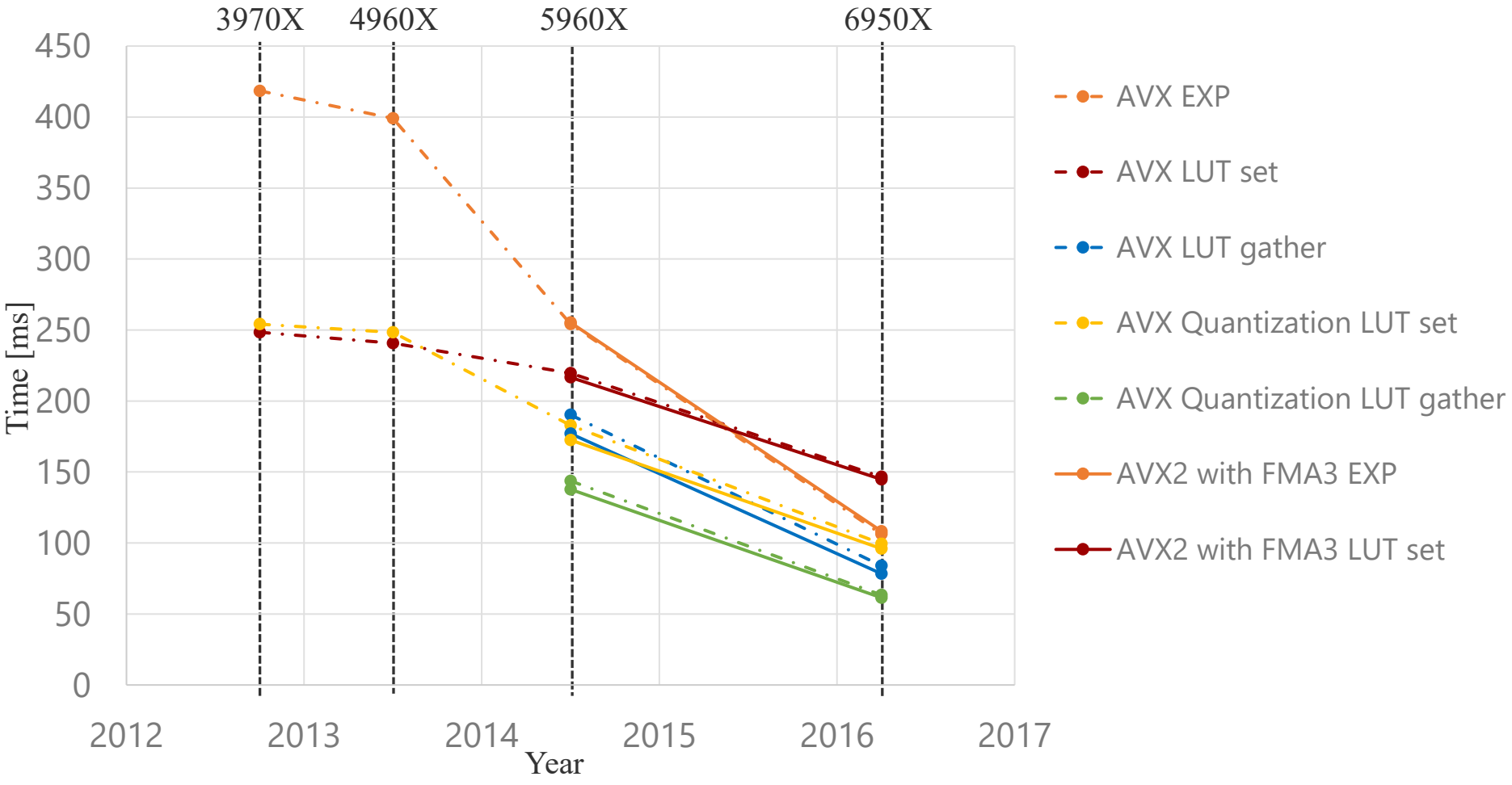
BF


3970X
4960X
5960X
6950X
700
600
500
400
300
200
100
0
Time [ms]
2012
2013
2014
2015
2016
2017
Year
SSE EXP
SSE LUT set
SSE Quantization LUT set
AVX EXP
AVX LUT set
AVX LUT gather
AVX Quantization LUT set

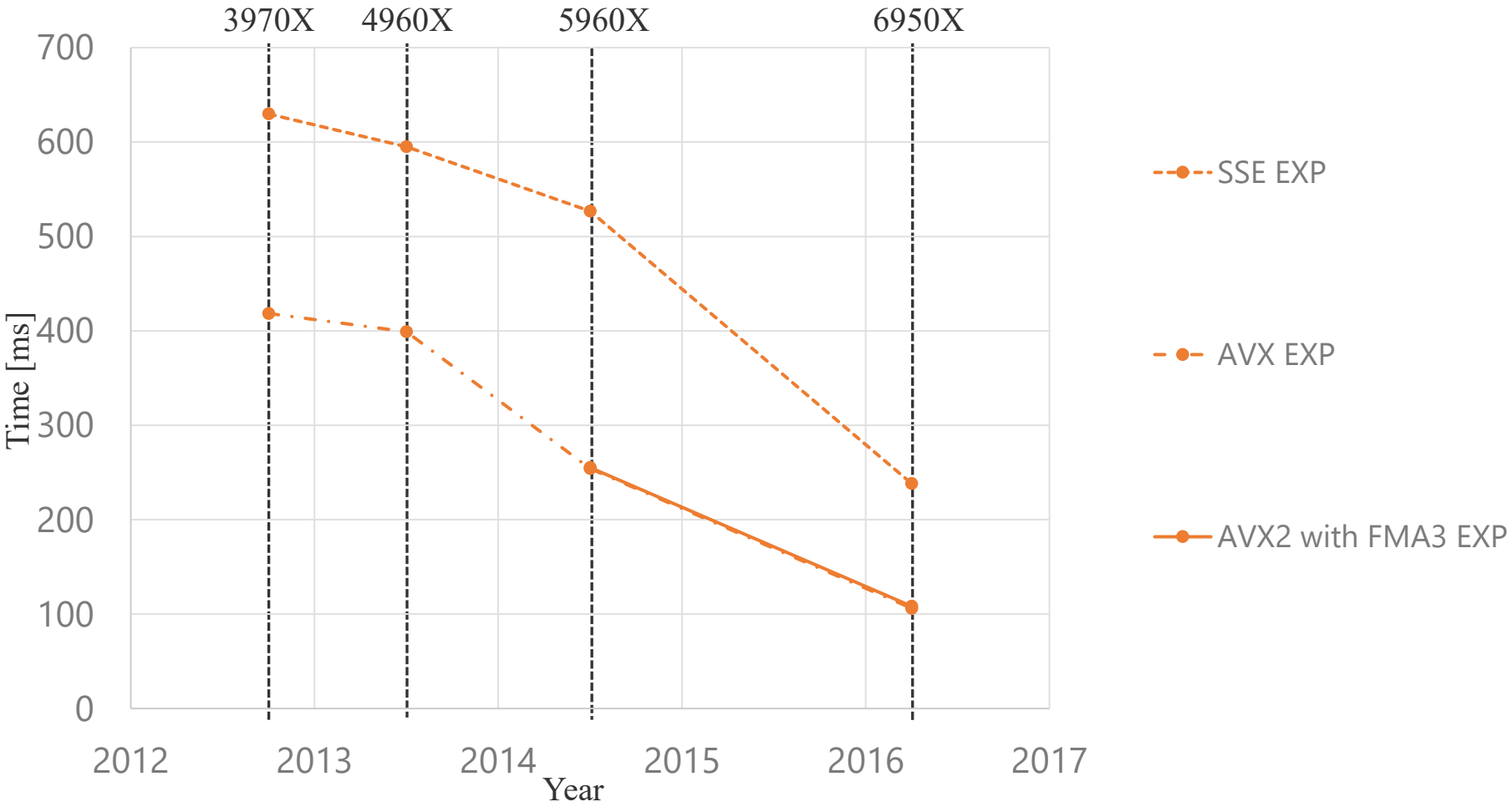
BF: SSE


3970X
4960X
5960X
6950X
700
600
500
400
300
200
100
0
Time [ms]
2012
2013
2014
2015
2016
2017
Year
SSE EXP
SSE LUT set
SSE Quantization LUT set

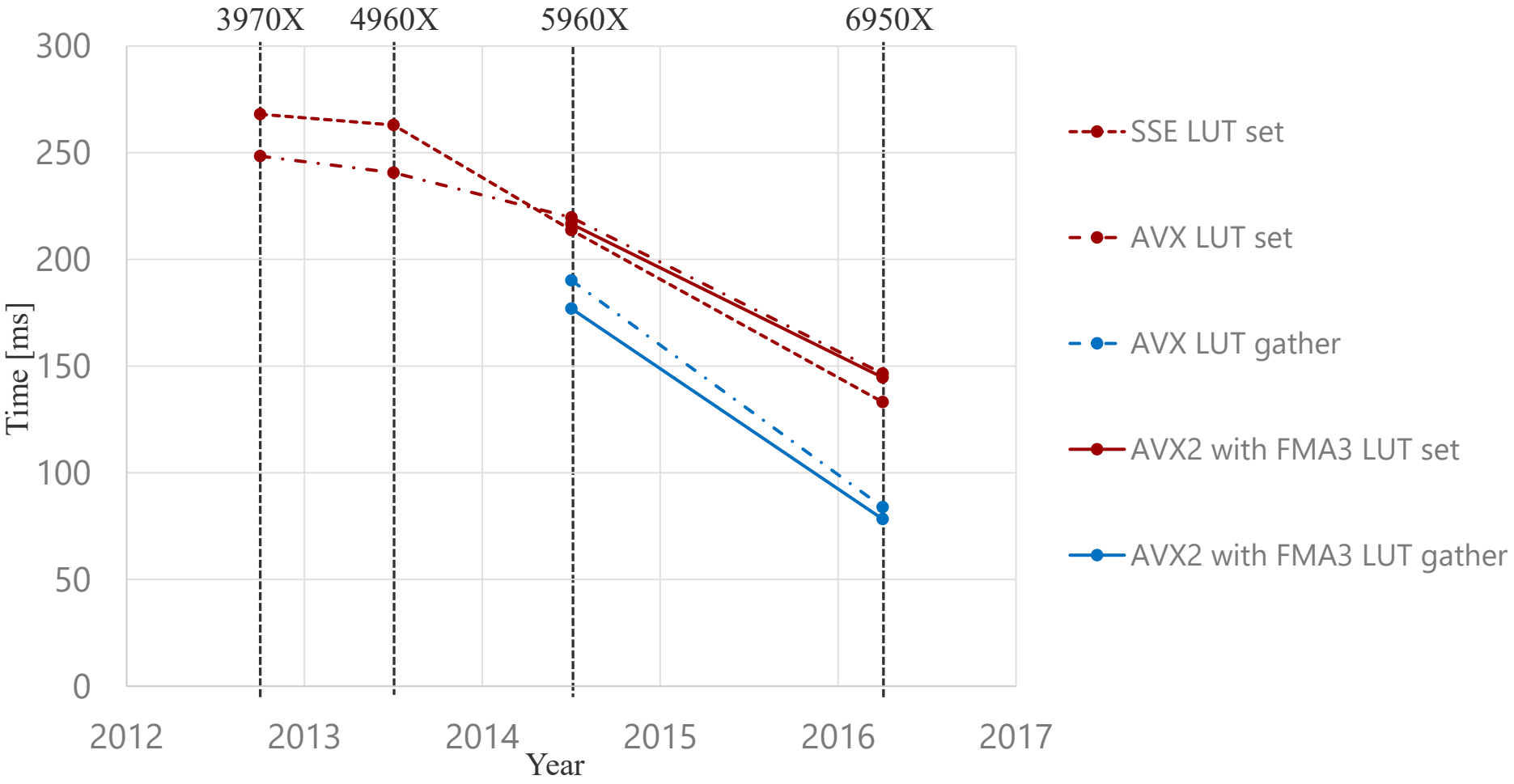
BF: AVX/AVX2


3970X
4960X
5960X
6950X
450
400
350
300
250
200
150
100
50
0
Time [ms]
2012
2013
2014
2015
2016
2017
Year
AVX EXP
AVX LUT set
AVX LUT gather
AVX Quantization LUT set
AVX Quantization LUT gather
AVX2 with FMA3 EXP
AVX2 with FMA3 LUT set

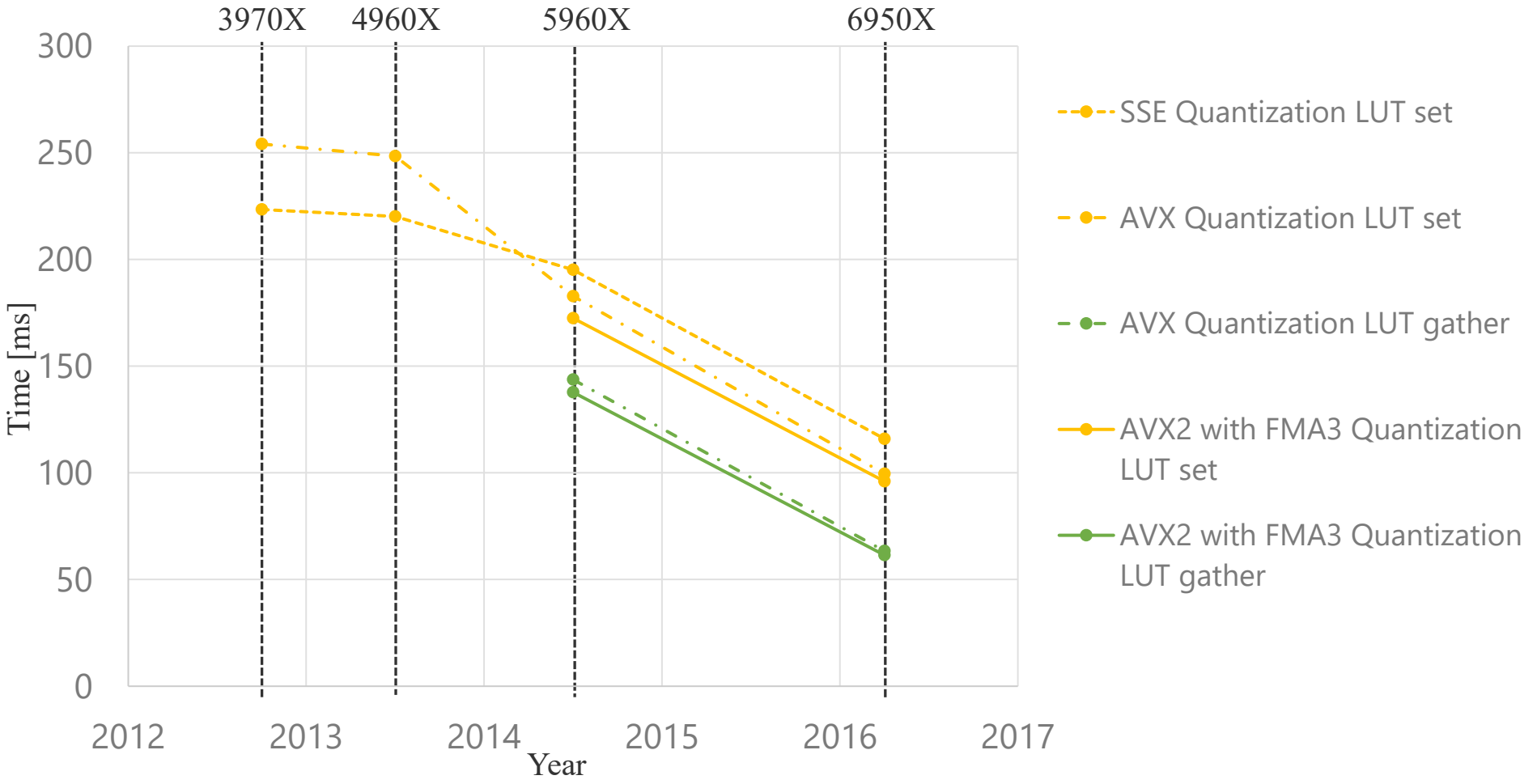
BF: EXP


3970X
4960X
5960X
6950X
700
600
500
400
300
200
100
0
Time [ms]
2012
2013
2014
2015
2016
2017
Year
SSE EXP
AVX EXP
AVX2 with FMA3 EXP

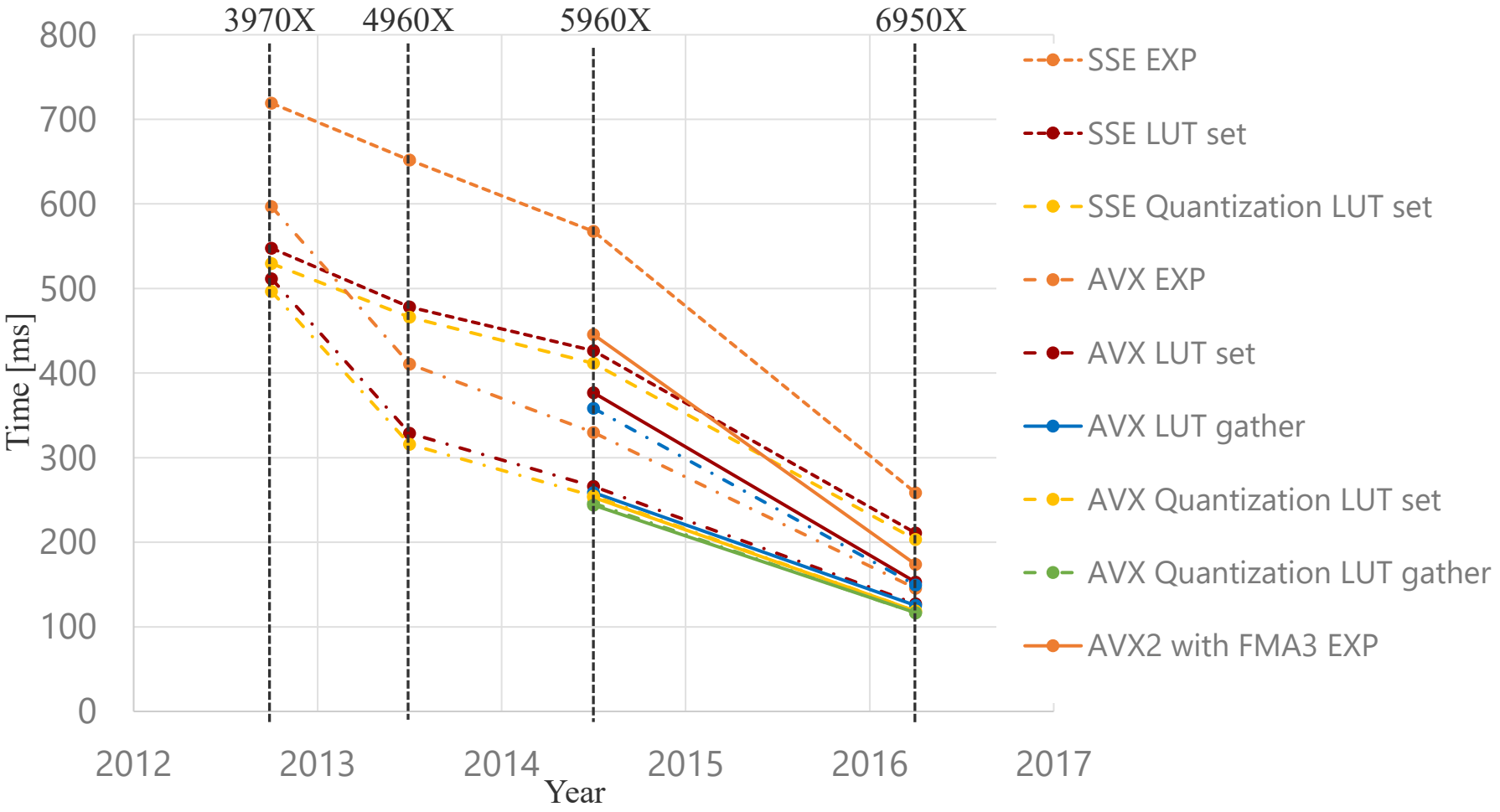
BF: LUT


3970X
4960X
5960X
6950X
300
250
200
150
100
50
0
Time [ms]
2012
2013
2014
2015
2016
2017
Year
SSE LUT set
AVX LUT set
AVX LUT gather
AVX2 with FMA3 LUT set
AVX2 with FMA3 LUT gather

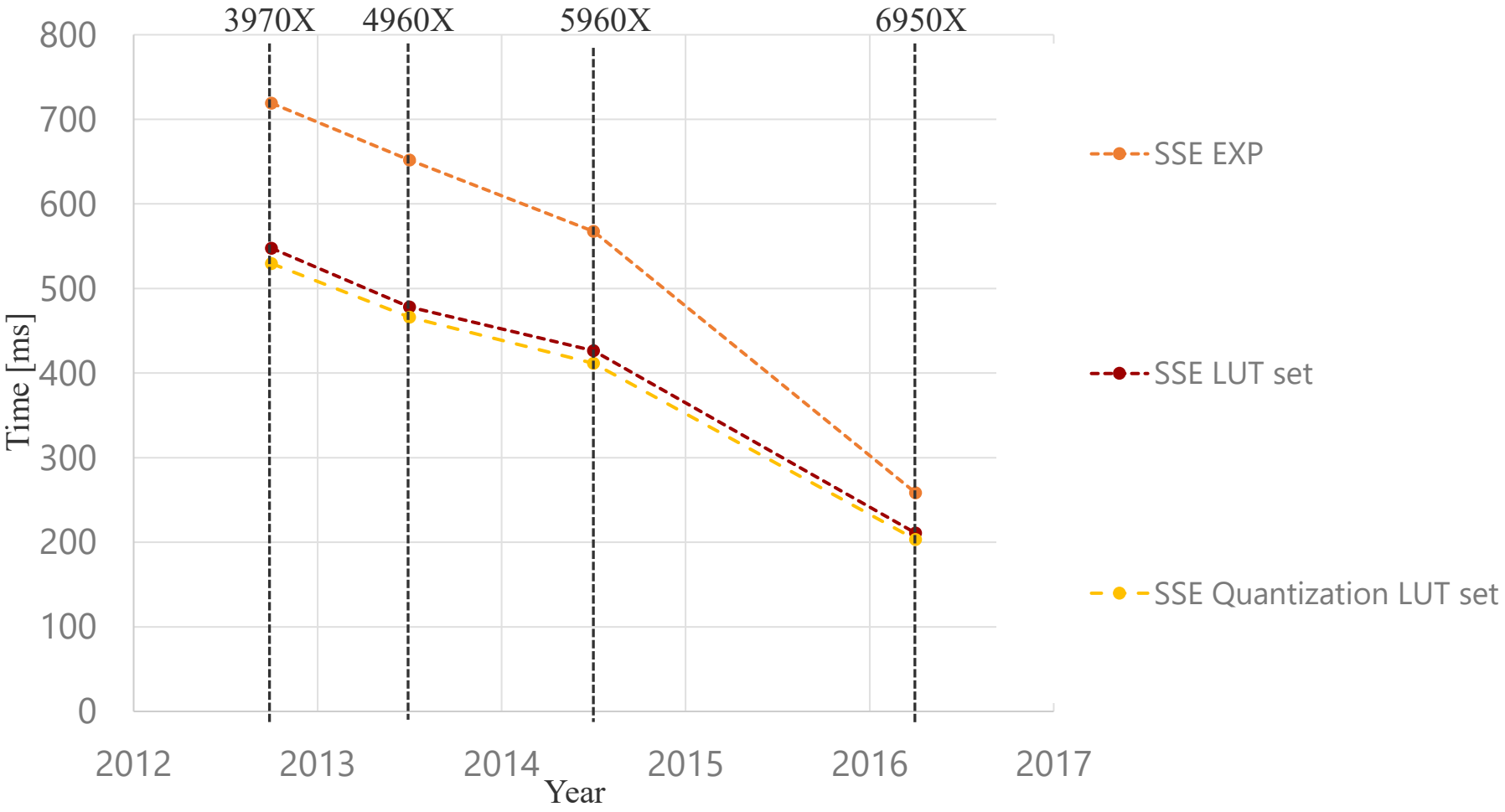
BF: Quantization LUT



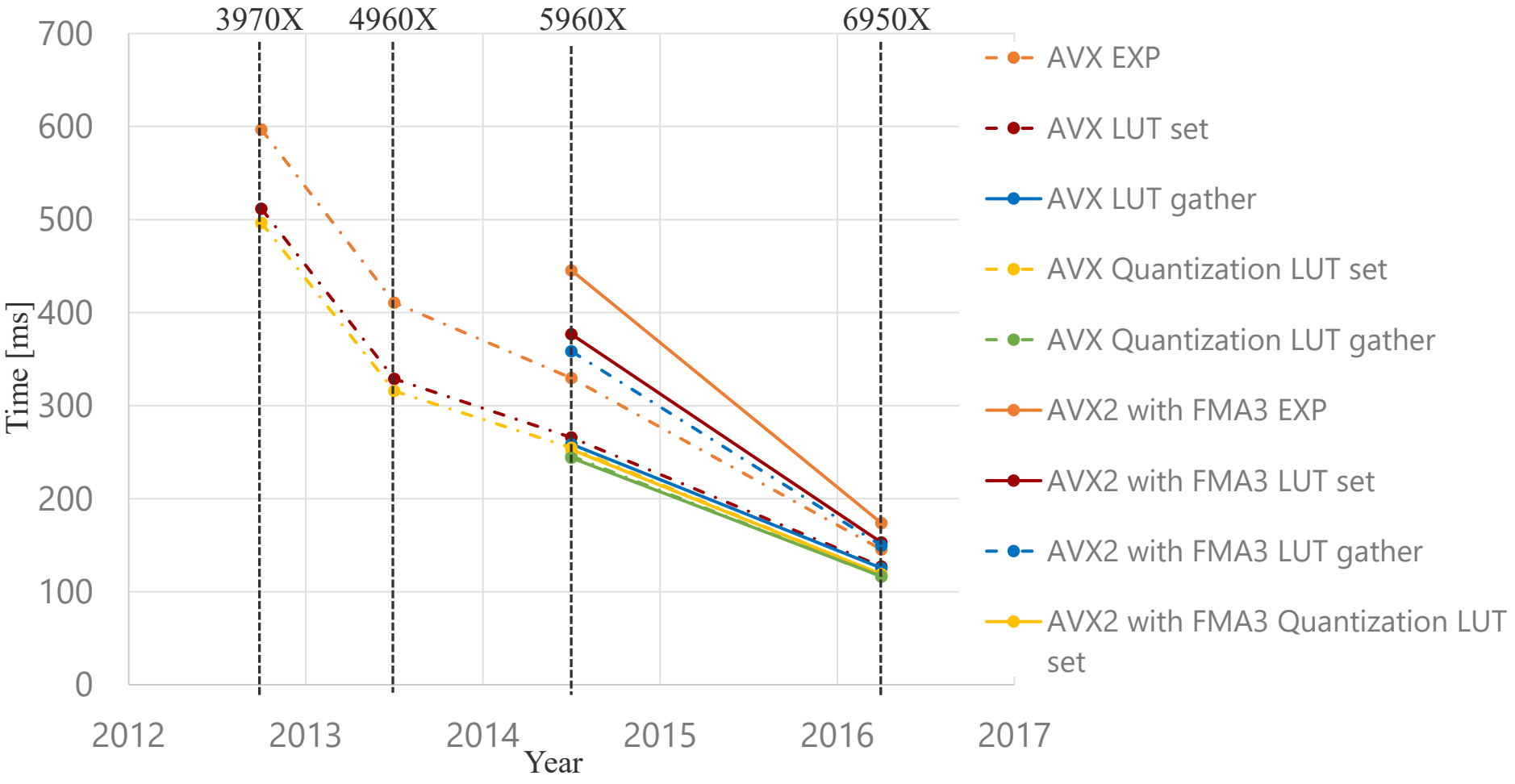
NLMF


3970X
4960X
5960X
6950X
800
700
600
500
400
300
200
100
0
Time [ms]
2012
2013
2014
2015
2016
2017
Year
SSE EXP
SSE LUT set
SSE Quantization LUT set
AVX EXP
AVX LUT set
AVX LUT gather
AVX Quantization LUT set
AVX Quantization LUT gather
AVX2 with FMA3 EXP

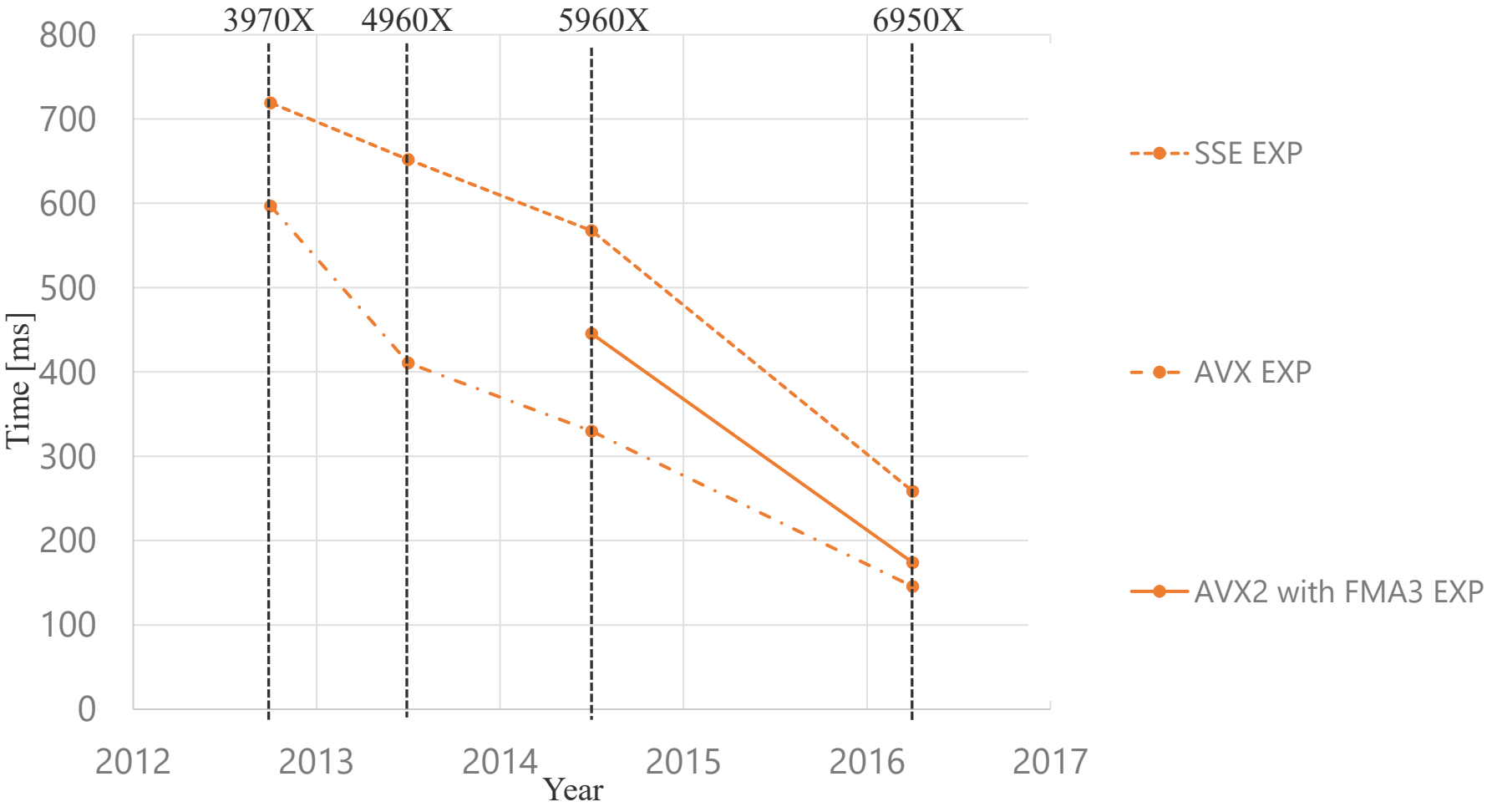
NLMF: SSE


3970X
4960X
5960X
6950X
800
700
600
500
400
300
200
100
0
Time [ms]
2012
2013
2014
2015
2016
2017
Year
SSE EXP
SSE LUT set
SSE Quantization LUT set

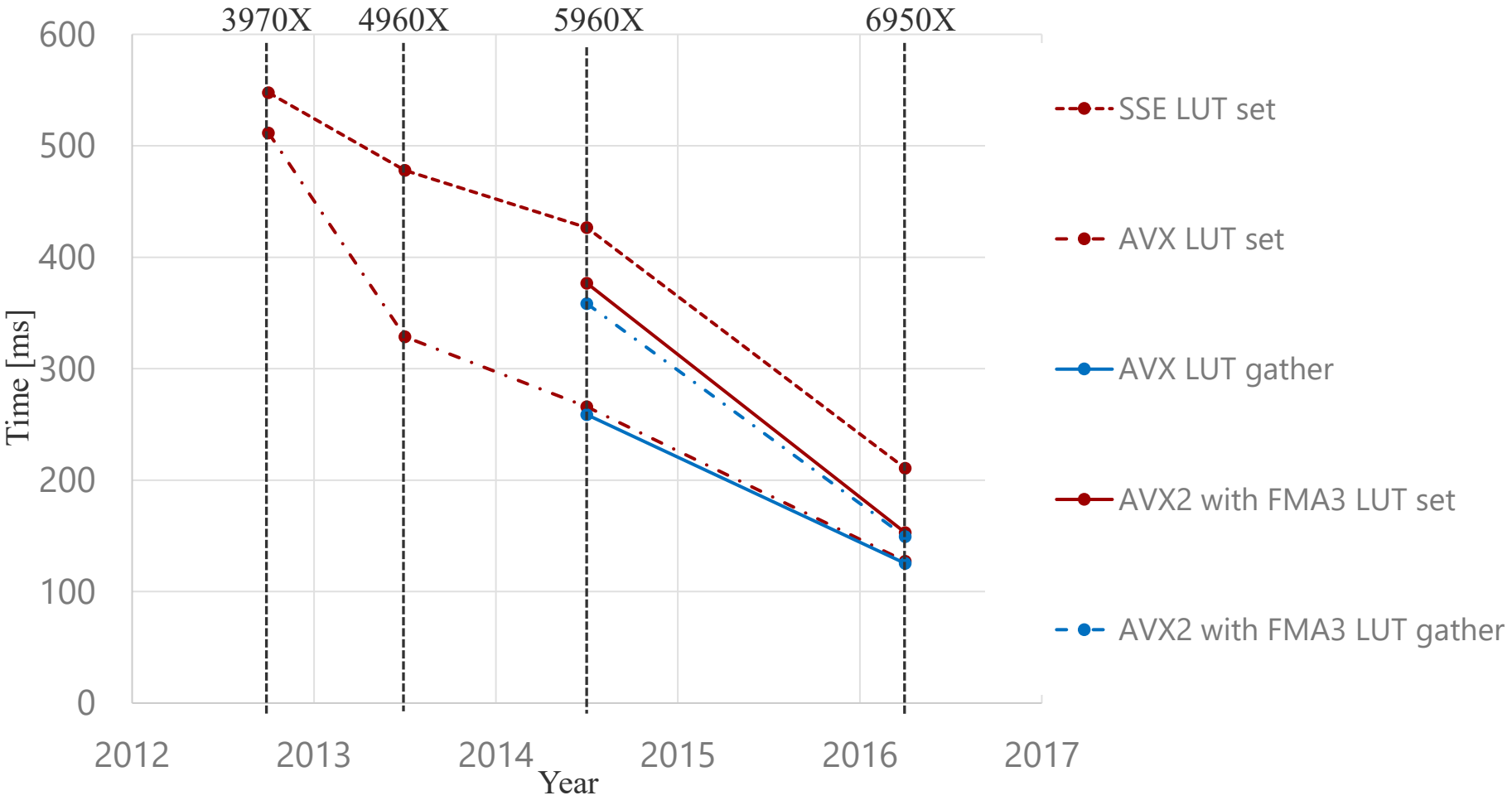
NLMF: AVX/AVX2



NLMF: EXP


3970X
4960X
5960X
6950X
800
700
600
500
400
300
200
100
0
Time [ms]
2012
2013
2014
2015
2016
2017
Year
SSE EXP
AVX EXP
AVX2 with FMA3 EXP

NLMF: LUT


3970X
4960X
5960X
6950X
600
500
400
300
200
100
0
Time [ms]
2012
2013
2014
2015
2016
2017
Year
SSE LUT set
AVX LUT set
AVX LUT gather
AVX2 with FMA3 LUT set
AVX2 with FMA3 LUT gather

NLMF: Quantization LUT