社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

[招待講演] 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

福嶋　慶繁†

† 名古屋工業大学
〒 466-8555 愛知県名古屋市昭和区御器所町
E-mail: †fukushima@nitech.ac.jp

あらまし　本稿では，画像処理における高能率計算を達成するために，並列化・ベクトル化プログラミング手法とそのデザインパターンや画像処理専用プログラミング言語について述べる．ムーアの法則に従ってトランジスタの集積度が上がり続けているが，一方で計算機アーキテクチャは複雑化しており，計算機の性能を生かすためにはアーキテクチャに合わせたプログラミングが必須である．また，データ I/O 性能の成長は演算能力の向上に比べてなだらかであり，この非対称性を考慮したプログラミングも重要である．実験では，これらを考慮することで画像の単純な線形変換や畳み込み処理，高度なアップサンプル処理が効率化されることを示す．
キーワード　画像処理プログラム，デザインパターン，高能率計算，並列化，ベクトル化

[Invited] High-Performance Computing Programming for Image Processing Based on Computer Architecture

Norishige FUKUSHIMA†

† Nagoya Institute of Technology
〒 466-8555 Gokiso-cho, Showa-ku, Nagoya, Aichi
E-mail: †fukushima@nitech.ac.jp

Abstract　In this report, we review a parallelized and vectorized programming for high performance image processing and its design pattern. Moore's indecates the number of transisters in a chip are exponentially increasing, but computer archtechture is also formming complex. For high performance computing, programming utilizing the knowledge of the archecture is essential. Beside, incresing of memory transfar speed is moderate than the computaion performance. The fact is also imortant for image processing programming. Experimentatl results show that simple image transformation, convolution, and complex upsampling are accelerated with effective programming.
Key words　Image Processing Programming, Design Pattern, High Performance Computing, Parallerization, Vectorization

1. はじめに

ムーアの法則 [1] に従って集積回路のトランジスタ数は年々倍増し，その集積度は上がり続けている．しかしながら，クロック数の成長は 2004 年ごろの Pentium 4 からほぼ頭打ちになっている．これは，放熱や電力消費の制約からクロック数を増加させることが困難になってきたことに起因する．その結果，増加したトランジスタの用途が，マルチコア化，ベクトルユニットの多様化やベクトル長の増加，キャッシュの大規模化，ハイパースレッディングなどの同時マルチスレッディング（SMT），ターボブースト，ビデオの符号化のためのエンコーダなどと多岐にわたっており，そのアーキテクチャは複雑化している．

マルチコア化が始まった Intel Core 2 アーキテクチャの登場（2006 年）付近から言われ始めた「フリーランチの終焉」は，ソフトウェアの性能向上をシングルコアの単純な性能向上に依存していたプログラマに対して，「これからはアーキテクチャに合わせたプログラミングが必要となる」というメッセージである．現在は，当時よりもさらに複雑性が増加しており，その集積回路の能力を引き出すためには，計算機のアーキテクチャを知り，そのアーキテクチャに合わせたプログラムを書く必要がある．2017 年の夏に発表されたコンシューマ向け最上位モデルである Core i9 7980XE は 18 コア，512 ビット（AVX-512）のベクト

ル演算ユニットに FMA 演算ユニットを搭載しており，一秒間あたりの浮動小数演算能力（FLOPS）は理論上，シングルコアでの単純なプログラミングに比べて 576 倍（18 コア ×16 ベクトル長 ×2:積和演算の同時実行）の差があり，さらには SMT，増加した L2 キャッシュ，拡張されたベクトル命令セットを効率的に使うことでそれ以上の性能差が生まれる．

本稿では，アーキテクチャに合わせた高能率画像処理プログラミングについて述べる．インテル社が自社の CPU の性能を実証するために画像処理ライブラリである OpenCV（1999 年，前身となる Image Processing Library (IPL) は 1997 年）を開発したように，画像処理と計算機アーキテクチャの関係は深い．近年，AMD Ryzen の登場や Intel からの新たな CPU の発表など CPU の並列化，ベクトル化が大きく進んでいる．本稿では，特に CPU（主に Intel などの x86）に着目して紹介する．

2. 計算機アーキテクチャの遍歴

2.1 クロック数とコア数

1971 年の Intel 4004 プロセッサ以来，そのクロック周波数は増え続け，2004 年の Pentium 4 では 3.8 GHz に到達した．しかしながら，その増加は頭打ちとなっており，2017 年現在の CPU もその限界周波数は 4 GHz 付近である．一方でマルチコア化は，2000 年の Pentium 4 による同時マルチスレッディング（ハイパースレッディング）による疑似的なマルチコア化から始まった．これは，現在使っていない演算回路を，他スレッドが使えるようにすることで疑似的にマルチコア化する方法で，わずかにダイを使うだけで，1 ～ 3 割の性能向上が得られる．実質的なマルチコア化は，2005 年の Pendium D からであり，この CPU は中に Pentium 4 を 2 つ積んだものであった．2006 年の Core 2 シリーズからは 4 コア搭載のモデルが登場し，近年までコンシューマ向けのモデルは 4 コア搭載の CPU が多かった．2009 年の Core i シリーズからは，マルチコア環境にハイパースレッディングが追加され，疑似的にコア数 ×2 のコアが使えるようになっている．また，最高スペックのモデルではそのコア数は増え続け，現在では 18 コア搭載している．

2.2 ベクトル演算器

SIMD 演算とは，複数のデータに対して同一の命令を一度に発行することで，同時処理するデータ量に応じた並列演算を行うことである [2]．Intel の SIMD 命令は，1997 年に MMX 演算（64 ビット整数命令）が登場して以来，SSE（128 ビット浮動小数点命令，1999 年），AVX（256 ビット浮動小数点命令，2011 年），AVX-512（512 ビット浮動小数点命令，2013 年）と発展し，ベクトル演算による計算性能の向上は順当に上がっている．Intel 以外には，AMD の 3D Now !（SSE 相当），ARM の NEON が SIMD 命令（128 ビット浮動小数点命令）がある．

浮動小数点における SIMD 命令は，Pentium III (1999 年) から SSE として搭載されている．初期の SSE は，2 クロックで SIMD データ幅の半分しか処理することができず，SIMD 命令と FPU 命令をインタリーブすることで，2 クロックで 3 つのデータを処理していた．Pentium 4 (2000 年) からは，2 クロックですべてのデータを処理できるようになり，Core 2 シリーズ（2006 年）で 1 クロックで 4 つのデータを処理できるようになり，同じ SSE 命令でも年代を追うごとにその性能が上がっている．そして，Core i シリーズの第 2 世代（2011 年，Sandy Bridge）で AVX 命令でデータ幅 8 になり，乗算と加算を同時に発行できる FMA 命令が Core i シリーズの第 4 世代（Haswell，2014 年）から使えるようになり，1 クロックで処理できる計算量が増加している．2016 年には Xeon Phi という外付け計算ユニットで 16 個の単精度浮動小数点が同時に処理できる AVX-512 が使えるようになり，2017 年にはコンシューマ向け CPU でも使えるようになった．マルチコア化に加えてベクトル長の増加，FMA の追加により，秒間浮動小数点演算回数を表す FLOPS は増加し続けている．

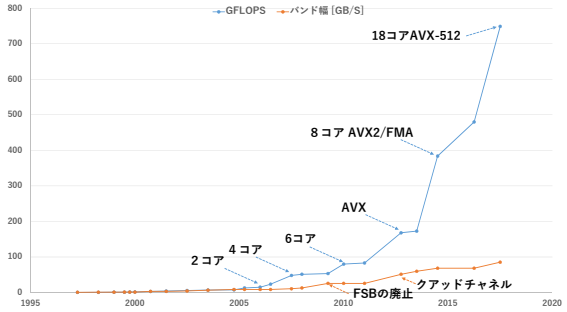

図 1　Intel CPU の年代別理論 FLOPS とメモリバンド．

演算能力の向上以外にも様々な命令が追加されている．四則演算 add, sub, mul, div や最大・最小値 max/min，高速な数値計算アルゴリズムがある逆数の高速計算 rcp やルートの逆数の高速計算 rsqrt といった基本的な命令から，比較 cmp，内積 dp，切り上げ，切り捨て，四捨五入の ceil, floor, round，各種ビット演算やビットカウント popcount まで年代を追うごとに様々な命令が追加されている．近年の特筆するべき追加命令は，ベクトルアドレッシングである．SIMD 命令でデータをメモリから取得する場合，データは連続している必要がある．そのため，RGB のカラー処理など演算する画素が飛び飛びである場合は，データを複数とって並び替える必要があった．離散的でも規則的なアクセスならば，複数の命令を組み合わせてシーケンシャルにデータを並べることが可能であるが，不規則な場合は，ベクトル演算でデータをロードすることができず，ほぼスカラ演算で代入する必要があった．ベクトルアドレッシングとは，飛び飛びのデータをベクトル演算で読み書きする命令である．読み込みである gather は AVX2（2014 年）から使え，書き込みの scatter は AVX-512（2017 年）から使える．

2.3 メモリバンド幅

演算性能の伸びに比べ，主メモリからのデータバンド幅の伸びは少なく，その非対称性は広がり続けている．図 1 にコンシューマ向けの Intel CPU の理論 FLOPS とメモリバンド幅を年代別に示す．1990 年までは，メモリの I/O が演算よりも速かったが，現在では大きな開きがある．Core i シリーズではキャッシュですら L1 が 4，L2 が 12，L3 が 26-31 サイクルかかり，メモリにいたっては数百サイクルかかる．そのため，デー

タがキャッシュにない場合，演算のためにデータが届くのを待ち続けるため，計算速度はメモリバンド幅に律速する．現在の計算機で計算する場合，キャッシュにデータがヒットし続けるようにしなければ，いかにアルゴリズムが優れていて演算回数が少なかったとしても，速く動作させることができない．

2.4 ルーフラインモデル

FLOPS は演算のみに着目した指標であり，メモリ帯域の性能を反映していない．例えばデータ同士の加算は，浮動小数点演算は 1FLOP に付き 2 つのデータ読み込みと 1 度の書き込みが必要であり，単精度浮動小数点は 4 バイトであるため，全体で 12 バイトの読み書きが生じる．一方で演算は 1 度しかないため，1 演算する間に 12 バイト読み込めるマシンでなければ，演算はすべてメモリからのデータ到着待ちとなる．実際は，演算とロード・ストアは並列に行えるため，データ I/O のレイテンシは一部隠せるが，データが遠い位置にあれば隠蔽不可能なほど待ち時間が長くなる．

演算強度とは，ロードを待たずにどれだけ演算可能かを表す指標であり，FLOPS をデータのバンド幅で割った F/B 値で示される．また逆数である B/F 値は一度の浮動小数点演算に必要なメモリアクセス量を示す指標として使われる．上記の加算命令の場合，B/F 値は 12 以上でなければ計算機の演算性能をフルに使うことができない．しかし，スーパーコンピュータの京でも 128FLOPS/64GB/s=0.5 B/F であり，コンシューマ最上位の Core i9 7980XE は 748.8GFLOPS/85.31GB/s=0.114 B/F である．つまり，現在のコンピュータは演算強度が高い処理しかその演算能力を生かすことができない．演算強度は，レジスタにあるデータを使いまわすことで増加するため，局所的なデータに対して重たい処理が必要になる．

演算性能とメモリバンド幅を考えたパフォーマンス分析モデルとしてルーフラインモデル [3] がある．図 2 に Core i9 7980XE のルーフラインを示す．横軸が演算強度，縦軸がパフォーマンスであり，様々なプログラムがプロットされている．また，演算能力とメモリやキャッシュアクセス速度の上限の直線が描かれており，その上限までプログラムが効率化可能である．メモリ，キャッシュの上限は，横軸が F/B であることから $y = x$ の直線でバウンドされる．この形が屋根のようになっていることからルーフラインと呼ばれる．高演算強度プログラムは，並列化，ベクトル化を行うことでパフォーマンスが垂直に上昇し，演算能力上限まで向上する．一方，低演算強度プログラムは，その上昇が I/O 速度でバウンドされ，理論値よりも低いパフォーマンスしか出ない．この性能を上げるには，データ構造の変更，入力データサイズの間引きにより，I/O を削減したり，メモリからではなく，キャッシュからのロードに置き換える工夫が必要となる．他にも，アルゴリズムを変更し，パフォーマンスを下げてでも演算強度を高め，データがキャッシュに乗るようにすることで，並列化，ベクトル化の効果がより得らるプログラムが作れる．このように，ルーフラインにより，プログラムをどう高速化すればよいのかの指針が立つ．ルーフラインは Intel Parallel Studio を用いればプロットできる．今後も図 1 のようなメモリと CPU 速度の関係が続くようならば，演算強度を高めるアルゴリズムの設計を行うべきである．

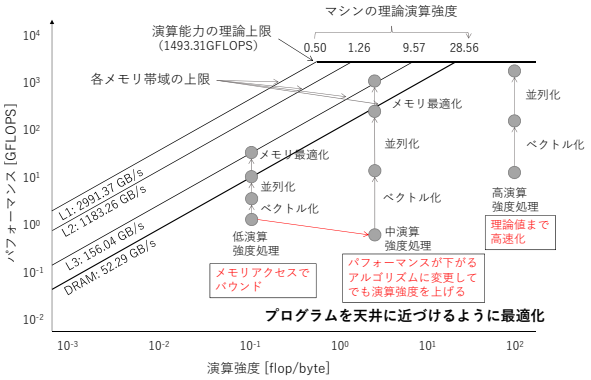

図 2 Intel CPU の年代別理論 FLOPS とメモリバンド．

3. 画像処理プログラミング

3.1 並列化プログラミング

アムダールの法則 [4] は，コア数 N が増加することでどれだけプログラムの性能が向上するかを示す法則である．

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

ここで N はコア数，P は並列実行可能な処理の割合である．もし，すべて並列化可能であれば，コア数倍計算能力が向上するが，並列化不可能な部分があればそこで効率が頭打ちになる．しかし実際には，並列化のためにタスクを起動して同期するオーバーヘッドが生じる．他にも，アルゴリズムを並列化するための事前処理が必要となり，その部分もオーバーヘッドとなるため，そのオーバーヘッドを含めて考慮されることが多い．オーバーヘッドをコア数の関数とすれば以下となる．

$$S(N) = \frac{1}{(1-P) + \frac{P}{N} + f(N)} \tag{2}$$

コア数を増やすほどオーバーヘッドが増える場合は，過分割するとパフォーマンスが落ちるため，最適な分割数が存在する．特に後述するスレッド並列化ではスレッドを起こしたり，同期させたりするためのオーバーヘッドが大きくなる．図 3 に $P = 0.8, 0.9$，オーバーヘッド $f(N) = 0, 0.01N$ があった場合の並列化効率を示す．オーバーヘッドが無い場合は，コアが増えるほど高速化するが，並列度に応じてその速度向上比は飽和する．オーバーヘッドが有る場合は，並列化には最適値が存在するようになる．

3.2 並列画像処理プログラミングのデザインパターン

画像処理では，適切な単位で処理を分割し，それらを並列に実行する．その単位は主に 3 つの場合に分けられる．一つ目は，画像 1 枚を処理する単位での並列化であり，画像 1 枚に対するすべての処理をパイプライン化する場合や R，G，B の各画素ごとに同じ処理を並列に実行する場合があり，この処理はスレッドレベル並列化で行われる．二つ目は，画像をブロック分割し，ブロック単位で並列化する場合である．画像を一定の幅の行や列単位で区切って帯状に分割したり，画像を $N \times N$ のブロックに分割したりして，その領域ごとに並列に実行する．この処理もスレッドレベル並列化で行われる．三つめは，1 画

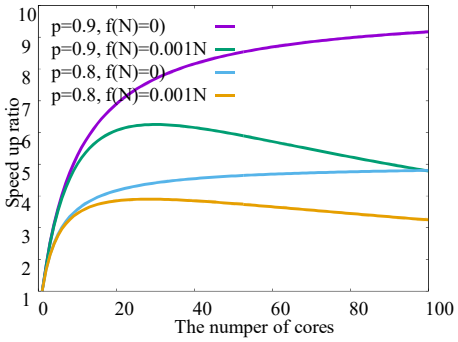

図 3　アムダールの法則.

素づつ並列に計算する場合である．この場合は，スレッド並列化ではオーバーヘッドが大きいため，ベクトル化で行われる．

画像処理には計算に順序依存関係があり，それを考慮すると，画像に対する操作の並列化パターンが 4 つに分類される [5].

（1） マップ：画素単位の操作．画素間の四則演算や三角関数の適用，閾値処理など.

（2） ステンシル：エリア単位の操作．畳み込み処理など.

（3） リダクション：画像から単一ないしは次元を減じた状態に出力する操作．ヒストグラムや画素全体の総和など.

（4） スキャン：前の計算状態に依存して画素単位に出力する操作．IIR フィルタや動的計画法など.

実際には，他にも画素の入れ替えを行うギャザー，スキャッター，パック，スプリットなどのパターンもあるが割愛する．

また，任意の画像処理のプログラミングは，画像や行列への操作を 1 ステージとした多段階ステージのパイプライン処理で記述できる．単純な処理ならばパターン一つで画像処理が完成し，複数の処理ならば，それらのパイプラインを実行することになる．また，ステージに依存関係がなければ，ステージ単位で並列処理を行われる．例えば，ソーベルフィルタによるエッジ検出では，3×3 のカーネルを畳み込むステンシル処理を行い，出力画像を画素ごとの閾値処理であるマップ処理で出力する．各画素ごとにブロック単位で分散計算する場合は，移動平均フィルタとなるステンシル処理を行った後に，入力画像から移動平均の出力の差分を取って二乗するマップ処理を行い，最後にまたその処理した値を移動平均フィルタによるステンシル処理を行う．移動平均フィルタは，定数時間で実行するインテグラルイメージにより定数時間ででき，この場合は，スキャン処理で実装する．またセパラブル実装も可能であるため，縦のフィルタ，横のフィルタに分離し，そのステージを 2 段階に分けることで計算オーダーを下げることができる.

3.3　スレッドレベル並列化

スレッド並列化は，Pthreads などによりスレッド関数を自前で記述することで可能となる．しかしながら，この記述はコード量が増え，複雑性，可読性やメンテナンス性が著しく低下する．そのため，コンパイラ拡張である OpenMP や Intel Cilk Plus，コンパイラに依存しないライブラリ実装である Intel TBB や Microsoft Parallel Patterns Library の Concurency などを用いて並列化する．スレッド化時は，画像処理の多重ループの外側を各スレッドに分割して処理を行い，最も内側のループは時節のベクトル並列化することが多い.

3.4　ベクトル化

ベクトル演算のためには，連続するベクトル長分の配列を用意し，その配列同士を専用命令で演算することでベクトル演算が可能である．その配列は，シーケンシャルな画像配列に対して load 命令を出すことで，必要なベクトル長分だけ用意することができる．各データの先頭アドレスを 16, 32, 64 の倍数になるようにすることで I/O が高速化するが今回は説明を省略する．実際に配列にデータが格納されれば専用の命令関数をその配列に対して適用するだけでだけでベクトル化可能である．これらの SIMD 命令は，Intrinsic function を書くことで実現され，Visual Studio や GCC，ICC がサポートしている．また，OpenMP では OpenMP4.0 から SIMD のためのディレクティブが使うことができるため，簡単な SIMD 命令ならば OpenMP の活用が可能である．ただし，Visual Studio では現在 OpenMP2.0 までしか対応していない.

ベクトル化の難所は複雑なデータ構造へのアクセスにある．カラー画像の場合や飛び飛びのデータが必要な場合，転置が必要な場合など，必要な画素をベクトルとして用意するためにデータの並び替えが必要となる．最も簡単に任意の配列を用意するには，set 命令を用いて一つづつベクトルに値を代入することだが，これはスカラ演算であり処理が重たい．初学者がベクトル化を行った結果，計算速度が逆に遅くなってしまう場合は，セットを多用した結果であることが多い．理想的には，最終的に必要となるベクトルの構成要素を部分的に持つ複数のベクトルをロードし，それらのベクトル値を入れ替える shuffle, permute 命令と二つのベクトルの値を混合する blend 命令を多段に適用することで，所望のベクトルを用意することが高速化につながる．近年では，ベクトルアドレッシング命令である gather/scatter が追加され，不規則なデータの読み書きが比較的高速に動作する記述が容易にできるようになった．ただし，現状では，gather や scatter 命令よりも shuffle, permute, blend の組み合わせのほうが早いため，データの入れ替えにパターンがある場合は gather/scatter 命令を使わないほうが良い.

このベクトルアドレッシングは，重たい数値計算である超越関数（三角関数や指数関数）や複数の多項式が必要となる代数関数（例えば，$\sqrt{x}, \sqrt{(x+1)^3}$ などが混ざった関数）などをテーブル引きするときに有効である．テーブル参照は，要素に応じて不規則に適用されるため，ベクトルアドレッシング命令が無い場合は set 命令でしか対応できなかった．他にも，離れた位置の飛び飛びのデータが必要となる光線空間の処理なども高速化可能である.

3.5　ループフュージョンとタイリング

メモリアクセスの局所性を高めることは，並列化とベクトル化と同様に高速化にとって重要である．データがキャッシュに収まらない場合，演算以上にメモリアクセスの時間が長く，その場合は，メモリアクセスを減らすかキャッシュに収まるようにデータの処理単位を変えることが重要である.

例えば，閾値処理をする場合，ソーベルフィルタの次に閾値

処理の関数をかけるといった2段階の画像の操作をマージし，画素単位でソーベルと閾値処理をすると演算回数は同一にもかかわらず計算速度が向上する．このソーベルフィルタは縦横のカーネルに分解可能なセパラブルフィルタであり，ボックスフィルタやガウシアンフィルタもその一種である．セパラブルフィルタは横にフィルタし，そのフィルタ結果を縦にフィルタすることでカーネル分解し半径 r のフィルタの計算オーダーを $O(r^2)$ から $O(r)$ に下げる．しかしながら，セパラブルフィルタは2ステージのフィルタとなり，フィルタ済みの計算結果を保持する必要とステージごとの同期が必要である．これは，前処理の結果に広い依存関係があるため，2段階目の処理では全コアで共用化されているラストレベルキャッシュからの読み込みが必須となる．一方原始的な実装は，1ステージで計算が終わるためキャッシュ効率が高く，アルゴリズムによる計算時間の増加が，メモリアクセス時間の増加を下回り速くなることがある．このトレードオフを取るために，画像をブロックに区切るタイリング処理を行い，多段ステージの各処理をブロック単位で行うことで，すべて L2 キャッシュにのせたままの処理を試みる．ただし，ブロックに区切ることでブロック間に処理の依存関係が生じる．例えばブロックの上部を縦方向に畳み込みする場合，その一つ上のブロックの計算結果が必要になる．この依存関係を同期処理等で解決するにはメモリの排他制御が必要となるが，この処理は非常に重たい．そのため，この依存関係を，必要な箇所だけ冗長に計算することで切り離す．このように冗長計算を行うことで，データの局所化が達成でき，実質の速度が向上する場合がある．

3.6 ドメイン固有言語

並列化，ベクトル化，タイリングやループヒュージョンは，対象となる計算機アーキテクチャに合わせてプログラムする必要がある．その場合，CPU は SIMD と並列化プログラミングを，GPU の場合は CUDA を，FPGA の場合は HDL で書くことになり，プログラムの生産性は著しく落ちる．プログラミング言語 OpenCL を用いれば，CPU，GPU，FPGA 共通のコードを書くことも可能であるが，プログラムの書き方に応じてアーキテクチャにとって得手不得手が出る．CPU だけを取ってみても，世代ごとにアーキテクチャが異なり，最適なプログラミングが異なるため結局最適なプログラムを書き直す必要がある．加えて，どのように並列化単位を設定し，どの順番で計算するのが最適なのかをチューニングするために，あらゆるパターンのプログラムを書くことは非現実的である．

この問題を解決するために，画像処理に特化したドメイン固有のプログラミング言語 (Domain Spesific Language: DSL) が提案されている [6], [7]．Halide は活発に更新されている画像処理 DSL であり，C++に組み込む副作用のない関数型言語として作られている．プログラムは，画像に適用する関数の定義とその実行スケジュール，ベクトル化，並列化，タイリングのパラメータを記述するだけであり，あとはアーキテクチャに合わせた最適なコードを出力してくれるためプログラムの生産性が大幅に高まる．更には，この Halide は，画像処理ライブラリ OpenCV におけるディープラーニング処理の高速化等に使われたり，FPGA の拡張がされたり(注1)と，これからより注目が集まることが期待される．

4. 実　　例

4.1 マップ：コピー，コントラスト強調，指数関数

まず，マップ処理であるメモリコピー，$ax+b$ の線形変換（コントラスト強調），指数関数適用を行った．このような処理は，出力が入力の多項式で出力される関数である．

$$f(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{n-1} x^{n-1} \tag{3}$$

このマップ処理はメモリから取得したデータが再利用されない．そのため，画像サイズ小さければキャッシュにのるが大きければメモリからの直接の取得となる．また各処理の演算強度は，メモリコピーは演算がなく，線形変換は積和 1 回，指数関数は 10 次近似を使ったため，乗算 18 回，加算 10 回である．

図 4 に各処理のメモリコピーや C++実装からの速度向上比と画像サイズの関係を示す．画像サイズは縦横同じの正方サイズである．なお，各実装は AVX でベクトル化，OpenMP で並列化した．画像サイズが小さい時は線形変換も exp 演算も C++実装の数倍高速化している．ただし，非常に小さい場合は，並列化のオーバーヘッドが大きくその比率が低下してる．一定サイズを超えるとその速度向上比は一定となり，線形変換にいたっては C++実装と変わらなくなる．これは，メモリからのロードが計算の支配的要素を占めるからである．exp 関数は重たい計算をしている間にロード可能なためそのレイテンシが隠せ，C++演算に比べて 2 倍の向上比を維持している．またメモリコピーとの比較を見ると，線形変換はほぼメモリコピーと速度が同である．exp 関数は，途中まではメモリコピーよりも重たいが，一定のサイズからメモリコピーと同じ速度になる．これは，画像サイズが大きくなるにつれて，データがキャッシュからあふれ，主メモリから読み込むため，メモリ I/O で律速していくからである．なお，速度向上比が大きく低下する画像サイズは L 3 キャッシュサイズ（今回は 8MB）に相当する．線形変換は演算コストが非常に低く L1 キャッシュからのロードと比べてもほとんどわからないが，指数関数適用はそれなりの演算強度があるため，L1，L2 キャッシュからのロードに比べると計算時間がかかるためベクトル化，並列化の効果が小さなサイズの時は見える．つまり，map 演算は，キャッシュに乗っている場合は並列化やベクトル化の効果があり，そうでないときにはほぼメモリコピーと等速となる．

4.2 ステンシル：バイラテラルフィルタ

バイラテラルフィルタ [8] は，画像の輝度差に応じて重みが決まる畳み込みであり，ステンシル処理の一種である．画像 $\mathrm{I} = \{r, g, b\}$ のバイラテラルフィルタ結果 J は以下となる．

$$J_p = \frac{1}{N} \sum_{q \in \omega_p} \exp\left(\frac{\|p-q\|^2}{-2\sigma_s}\right) \exp\left(\frac{\|I_p - I_q\|^2}{-2\sigma_r}\right) I_q \tag{4}$$

ここで p, q は画素位置，ω はフィルタカーネルの範囲，$\sigma_{s,r}$ は

(注1)：https://www.halide2fpga.com/

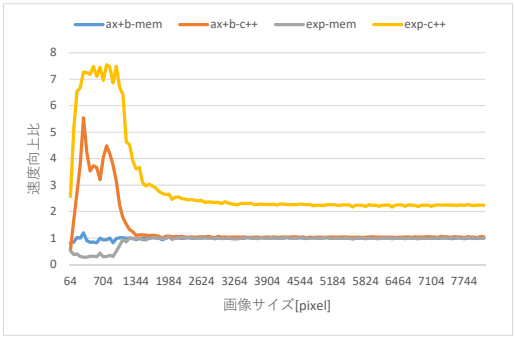

図 4　マップ演算のメモリコピーおよび C++実装からの速度向上比.

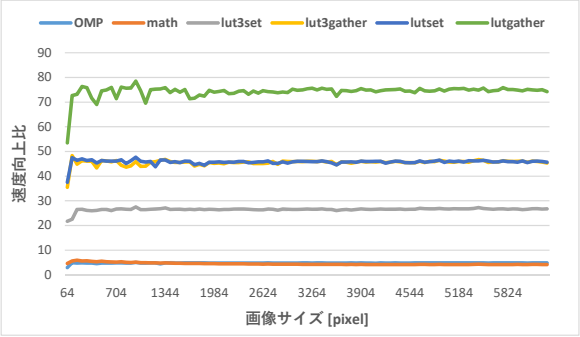

図 5　バイラテラルフィルタの計算時間.

平滑化パラメータであり，N は正規化項である．特にカラー画像の場合，その重みは以下で定義される．

$$\exp\left(\frac{\|I_p - I_q\|^2}{-2\sigma_r}\right) = \exp\left(\frac{(r_p - r_q)^2}{-2\sigma_r}\right)\exp\left(\frac{(g_p - g_q)^2}{-2\sigma_r}\right)\exp\left(\frac{(b_p - b_q)^2}{-2\sigma_r}\right) \quad (5)$$

各色分解され関数を指数法則でまとめれば以下となる．

$$\exp\left(\frac{\|I_p - I_q\|^2}{-2\sigma_r}\right) = \exp\left(\frac{(\sqrt{(r_p - r_q)^2 + (g_p - g_q)^2 + (b_p - b_q)^2})^2}{-2\sigma_r}\right) \quad (6)$$

超越関数である指数計算は重たいため，テーブル参照で高速化する．EXP2[|d|] = $\exp(\frac{d^2}{-2\sigma})$ となるテーブルがあれば，差の絶対値を入力すれば，各色成分ごとの重みを得ることができるため指数関数の計算は不必要である．また，指数法則でまとめた式は平方根の計算をし，整数に丸めれば近似ながらもテーブル参照が 1 度で済む．この場合，実際に必要なテーブルサイズは round($\sqrt{3 \times 255^2}$) = 442 である．3×255^2 サイズのテーブルを持てば近似は必要ないが，テーブルサイズが巨大となるためこのような処理を行った．図 5 にバイラテラルフィルタの各高速化実装の C++実装からの高速化比率を示す．サイズが小さなときは並列化のオーバーヘッドのため高速化率が若干低いが，それ以外の領域では画像サイズに依存しないことがわかる．これは，FIR フィルタの畳み込みは，フィルタする画素とその次の画素で使用する領域が大きくオーバーラップするためキャッシュが再利用され，メモリの転送速度に律速しないためである．また，set よりも gather が速く，LUT を 3 つから 1 つにまとめてることで更に速度向上していることがわかる．

4.3　ループヒュージョン：方向性アップサンプル

最後に，方向性アップサンプルの例を示す．これは，アップサンプルする際に，市松模様の順に画素を段階的に埋め，次段処理で補間済み画素も用いて補間することでことで精度を上げる処理である．今回の例では方向性キュービックアップサンプル [9] を用いた．この処理では，例えば画像を 2 倍にアップサンプル時には，2 段階で処理される．この多段処理をマージし，ループフュージョンすることで，処理の演算強度を上げることでその効果を上げる．計算時間は，C++コードは 59.12 ms であったのが，並列化して 10.31 ms，ベクトル化して 0.52 ms となった．更にループヒュージョンすることで 0.33 ms となり，この処理速度は OpenCV で提供されている Cubic 補間よりも処理速度が速い．

5.　おわりに

本稿では，計算アーキテクチャと画像処理プログラミングの関係をまとめた．計算のパフォーマンスはルーフラインモデルにより解析可能であり，画像処理プログラムは一般に演算強度が低い．そのため，高速化のためには，並列化，ベクトル化といった処理だけでなく，アルゴリズムを変更したり，ループフュージョン・タイリングをすることで冗長計算をしてでも演算強度を高めることで，より高速化できることを示した．また，近年のベクトルアドレッシング命令であるギャザーを使ってテーブル参照がより高速化可能であることを示した．本稿では CPU を中心に説明したが GPU であってもメモリアクセスと演算能力の関係の制約は変わらない．高能率プログラミングを行うためには，これらを考慮して行う必要があるだろう．

謝辞　本稿を執筆にあたり，実験を手伝っていただいた本学博士後期課程の前田慶博氏および福嶋研究室学生諸君に感謝する．本研究は科研費 JP17H01764 の助成を受けた．

文　献

[1] G.E. Moore, “Cramming more components onto integrated circuits,” Electronics Magazine, vol.19, 1965.
[2] M. Flynn, “Some computer organizations and their effectiveness,” IEEE Trans. on Computers, vol.C-21, no.9, pp.948–960, 1972.
[3] S. Williams, A. Waterman, and D. Patterson, “Roofline: an insightful visual performance model for multicore architectures,” Communications of the ACM, vol.52, no.4, pp.65–76, 2009.
[4] G.M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” Proc. Spring Joint Computer Conference, pp.483–485, AFIPS ’67, 1967.
[5] M.D. McCool, A.D. Robison, and J. Reinders, Structured parallel programming: patterns for efficient computation, Elsevier, 2012.
[6] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” ACM SIGPLAN Notices, vol.48, no.6, pp.519–530, 2013.
[7] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, “Darkroom: compiling high-level image processing code into hardware pipelines.,” ACM Trans. Graph., vol.33, no.4, pp.144–1, 2014.
[8] C. Tomasi and R. Manduchi, “Bilateral filtering for gray and color images,” Proc. International Conference on Computer Vision, pp.839–846, 1998.
[9] D. Zhou, X. Shen, and W. Dong, “Image zooming using directional cubic convolution interpolation,” IET image processing, vol.6, no.6, pp.627–634, 2012.