組み込みプロセッサのメモリアーキテクチャに依存しない画像処理プログラムの記述と実行方式

金井達徳[†] 瀬川淳一[†] 武田奈穂美[†]

組み込みシステムにおける画像処理プログラムでは，ターゲットシステムのメモリアーキテクチャを考慮したプログラムを書くことが処理速度向上やバス使用率低減，消費電力削減等につながる．しかしプログラマには，プログラムを実行するプロセッサやメモリのアーキテクチャに関する深い知識や，それを活かすプログラミング技法が求められる．そのため，技術者不足の解消だけでなく，プログラムの生産性や品質向上の観点からも，プログラマの知識や経験に左右されにくい開発手法が必要である．そこで我々は，ターゲットシステムの持つローカルメモリや DMA，キャッシュ等のメモリアーキテクチャを意識せずにループレスで画像処理を記述し，それをターゲットシステムに合わせて実行するプログラミングシステムを開発した．画像処理において繰り返し実行する処理単位に必要なデータの範囲を陽に指定するプログラミングスタイルを導入することで，処理に必要なワーキングセットの計算を簡単化する．その情報を利用したヒューリスティクスを用いてターゲットのメモリアーキテクチャに適した実行順序をスケジューリングするのが特長である．本方式はプログラマの負担軽減だけでなくプログラムの再利用性も向上でき，組み込み画像処理プログラムの開発に対する現実的な解になる．

Memory Architecture Independent Description and Execution of Embedded Image Processing Programs

Tatsunori Kanai,[†] Jun'ichi Segawa[†] and Naomi Takeda[†]

Writing optimized image processing programs for a memory architecture of target system is a key to improve performance, reduce bus utilization and save power consumption. However, deep understanding of target system architecture and sophisticated programming techniques are required for programmers. Program development method not depending on knowledge and experience of programmers is needed for productivity and quality of programs. So, we developed a new programming system where we write loop-less image processing programs without considering particular memory architecture such as local memory, DMA and cache memory, and the programs are executed efficiently by adapting to target memory architecture of embedded processors or DSPs. Adopting a new programming style which is to describe range of image data for repetitive processing unit explicitly, we can calculate working set easily and schedule execution sequence suitable for a target system. This system will be a practical solution to reduce programmer's tasks and promote reusability of programs in embedded image processing.

1. はじめに

システム LSI 技術の進歩によって，携帯電話，デジタルカメラ，メディアプレーヤ，監視装置，車載機器，認証装置等，高い処理性能を持つプロセッサを搭載したさまざまな情報機器が開発されるようになっている．これらの機器の多くで静止画や動画を扱う機能が必要とされ，各種の画像処理プログラムを開発する機会が増加している．しかし現在の組み込みプロセッサ上の画像処理プログラム開発は以下に説明するような課題をかかえている．

第 1 の課題はプログラミングに要求される知識やスキルの増大である．組み込みプロセッサ上でプログラムを効率良く動作させるためには，実行するターゲットシステムのプロセッサやメモリのアーキテクチャと，それらを駆使するためのプログラミング手法を深く理解している必要がある．しかし画像処理を理解したうえでこのようなプログラミングができる技術者は少ない．

第 2 の課題はターゲットシステムのアーキテクチャの多様性である．異なるターゲットシステムで同じよ

† 株式会社東芝研究開発センター
Corporate Research & Development Center, Toshiba Corporation

うな画像処理アルゴリズムの実装が必要になることが多い．しかし特定のターゲットシステムのアーキテクチャを考慮して書かれたプログラムは，単にコンパイルしなおすだけで他のターゲット上で効率良く動作させることは難しく，プログラムの書き換えが発生して再利用性に乏しい．

本研究はこのような背景に基づいて行ったものである．局所処理あるいは近傍処理と呼ばれる基本的な画像処理[15]において，ターゲットシステムのメモリアーキテクチャに依存する処理の記述をプログラマから隠蔽することで，プログラミングに要求されるアーキテクチャの知識やスキルを軽減することを目指した．

本論文で述べる手法は，画像を表現する配列データの扱いに専用化した言語によって特定のターゲットシステムに依存しない画像処理プログラムを記述し，そのプログラムを各ターゲットシステムのメモリアーキテクチャにあわせて実行するのが特長である．画像の局所処理における各処理単位で必要になるデータの集合すなわちワーキングセットの情報を簡単に求められるように言語は設計されている．その情報を利用した簡単なヒューリスティクスによって各メモリアーキテクチャにとって最適な実行順序の近似解を求める，レンジスケジューリングと呼ぶ方式で処理を進める．そのためプログラマにはプログラミングスタイル上の制約を課すことになるが，それと引き換えにターゲットアーキテクチャに依存する面倒な処理の自動化を実現する現実的な手法になっている．

本論文では，2 章で組み込みプロセッサの画像処理プログラム開発における課題と本手法による解決法の概要を説明する．その後 3 章でプログラムの記述方法，4 章でワーキングセットの求め方，5 章でレンジスケジューリングによる実行制御方式に関して詳しく述べる．6 章で関連研究について論ずる．

2. 組み込み画像処理プログラム開発の課題

2.1 アーキテクチャに関する知識の必要性

組み込み機器には SoC（System on Chip）に集積された組み込みプロセッサや DSP（Digital Signal Processor）等が搭載される．これらのプロセッサの持つローカルメモリやキャッシュメモリの容量は画像データのサイズに比べて小さいため，画像処理プログラム作成時にはメモリアーキテクチャに特に配慮する必要がある．

多くのプロセッサは主記憶に比べて高速にアクセスできるローカルメモリを持っており，DMA（Direct Memory Access）コントローラをうまく制御することで，ローカルメモリと主記憶の間のデータ転送と，ローカルメモリ上のデータを使ったプロセッサの演算処理をオーバラップさせて高い処理性能を引き出すことができる．これをステージングと呼ぶ．また，メモリが独立して動作可能な複数バンクで構成されている場合には，複数のプロセッサや DMA コントローラの間でメモリへのアクセス競合が発生して性能が低下しないように，同時にアクセスする可能性があるデータはメモリの異なるバンクに割り付けることも必要になる．プロセッサがキャッシュを持つ場合には，キャッシュの容量を考慮してループの処理順序を変更するループブロッキング[2]等の手法でキャッシュのヒット率を向上させることもできる．

ローカルメモリやキャッシュをうまく使って主記憶のアクセスを減らすことは，バスを通ってチップ外のメモリをアクセスするという時間と電力を消費する処理を減らすことになる．その結果，処理速度の向上とともに，組み込みシステムで重要視されることの多い消費電力の削減にもつながる[5]．さらに，バスの使用率を下げることで，バスを共有する他のデバイスとの競合による速度低下も低減させることができる．

画像を扱うシステムは大きな処理能力を必要とするためマルチコアのアーキテクチャを採用することも多い．マルチコアではプログラムをうまく分割して，並列処理する際のデータ共有や同期のオーバヘッドが大きくならないように設計する必要がある．特に組み込み用のプロセッサではコア間でキャッシュの一貫性をとらないものも多い．そのため，プログラマが適切なタイミングでキャッシュのフラッシュやインバリデートの命令を使って，各コアのキャッシュと主記憶の間の一貫性をとるようにしなくてはならない．

SIMD 命令や信号処理用の特殊命令の利用[4]は，組み込みプロセッサや DSP 上での画像処理プログラムのホットスポットの高速化にとって効果的な最適化手法である．しかしこれらの最適化は先に述べたメモリ使用方法の最適化を行ったうえで適用しないと，その効果はメモリアクセスのレイテンシに隠れて無駄になる．

具体的なプロセッサの例として，**図 1** に本論文中でも実験に用いているデュアルコア DSP である ADSP-BF561[3] のメモリアーキテクチャの概略を示す．各コアは 32 KB の命令用ローカルメモリと 64 KB のデータ用ローカルメモリと 4 KB のスクラッチパッドメモリを持つ．命令用のローカルメモリのうち 16 KB と，データ用のローカルメモリのうち 16 KB あるいは 32 KB は，それぞれ命令キャッシュおよびデータ

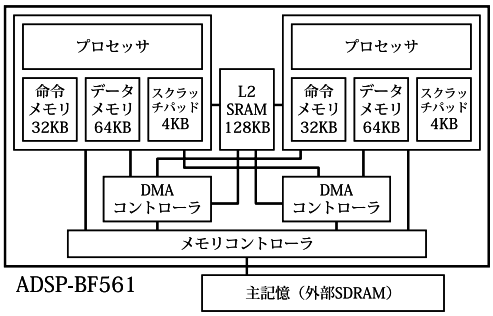

図 1 ADSP-BF561 の概略データパス
Fig. 1 Data path of ADSP-BF561 processor.

キャッシュとして動作させることもできる．128 KB の L2 メモリは両方のコアからアクセスできる．2 つの DMA コントローラを備え，プロセッサの動作とは独立にローカルメモリと主記憶の間でデータを転送できる．このようなプロセッサの性能を最大限に引き出すためには，メモリの構成や DMA コントローラの使い方等を熟知してプログラムを作成する必要がある．

2.2 メモリアーキテクチャ依存の処理の自動化

組み込みプロセッサ上の画像処理プログラムでは，プログラマの持つアーキテクチャに関する詳細な知識やスキルの習得度合いが性能に現れやすい．汎用計算機の経験しかないプログラマが最初に作った DSP 上のオプティカルフローのプログラムは 1 フレームの処理に数十秒要していたのが，アーキテクチャを理解してさまざまなプログラミングテクニックを学んで書き換えるとほぼリアルタイムで処理できるようになった例を我々も経験している．汎用計算機では仮想記憶やコヒーレントキャッシュが有効に働くため，プログラミングに際してアーキテクチャに関する詳細な知識の必要性は低くなっている．組み込みシステムの開発件数の増加や，画像処理と組み込みシステムの両方の知識を備えた技術者の不足といった現実の課題を考えると，組み込みプロセッサのプログラミングにおいても，必要な知識の量や質を汎用計算機の場合と同様に軽減してゆくことが望まれる．

そこで我々は，メモリアーキテクチャを深く理解していないと効率良いプログラムを書きにくい，局所処理あるいは近傍処理と呼ばれる画像処理アルゴリズムを主な対象にして，組み込みシステムに対する深い知識を持たない画像処理の技術者でもターゲットシステムのアーキテクチャを活かしたプログラムを開発できる方式を検討した．局所処理とは，平滑化や輪郭抽出等の画像フィルタあるいは画面上の動き検出等，画像上の全点あるいは一定間隔でとった全サンプル点に対してその近傍の画素データを使った比較的単純な計算を繰り返し行う処理である．

本論文で述べるプログラム開発手法では，画像処理の技術者が特定のターゲットシステムを意識することなくアルゴリズムを記述できるように，性能に影響を及ぼしやすいループを陽に記述せずに配列データの処理を書くことのできるプログラミング言語を設計した．この言語で記述したプログラムは，ターゲットシステムのアーキテクチャを活かして効率良く実行できるように，それぞれのターゲットごとに用意するランタイムライブラリを呼び出しながら処理を進める C++のプログラムに変換する．こうすることでメモリアーキテクチャに依存する処理を自動化してプログラマからは隠蔽する．また，記述したアルゴリズムが特定のアーキテクチャに依存することを防ぎ，再利用しやすくすることにもつながる．

このような開発方式をできるだけ単純な仕組みで実現して現実のシステム開発に適用できるようにするために，対象を画像の局所処理に絞るのに加えて，プログラムのスタイルにも制約を課すことで処理系の実現が容易になるように記述言語を設計した．すなわち，プログラマには局所処理を行う近傍の範囲（これをワーキングセットと呼ぶ）を明示したプログラムを記述してもらうことで，その明示された近傍情報を用いた簡単なヒューリスティクスによってターゲットシステムにとって最適な処理手順の近似解を求める実行形態（これをレンジスケジューリングと呼ぶ）が本方式の特長である．

3. 画像処理プログラムの記述方法

実行するターゲットシステムのアーキテクチャに依存しないように画像処理プログラムを記述するために，本論文で述べるプログラミングシステムでは専用の記述言語を設計した．このプログラミング言語は配列データの処理をループレスで記述する関数型の言語になっている．

3.1 プログラムの基本構造

システムの入力となるプログラムは，**図 2** に示すエッジ検出プログラムの例のように，**module**，**input**，**output**，**param**，**function**，**body** の 6 つのセクションで構成される．**module** にはプログラム名を，**input** と **output** にはそれぞれ入力変数と出力変数を，**param** にはモジュール内で使う定数あるいは定数式を宣言する．**function** にはモジュール内部で使う関数を定義し，**body** にはこのモジュールのメインプログラムに相当するトップレベルの式を記述する．

データの型にはスカラー型と 2 次元の配列型がある．

module $EdgeDetection$
input $InputImage^{uint8}_{imagesize}$
output $OutputImage^{int16}_{imagesize}$
param $imagesize = \langle 480, 720 \rangle$
function $Smooth(N) = \bigodot_{\langle -1,-1 \rangle} N + \bigodot_{\langle -1,0 \rangle} N + \bigodot_{\langle -1,1 \rangle} N + \bigodot_{\langle 0,-1 \rangle} N + N + \bigodot_{\langle 0,1 \rangle} N + \bigodot_{\langle 1,-1 \rangle} N + \bigodot_{\langle 1,0 \rangle} N + \bigodot_{\langle 1,1 \rangle} N$

$Laplacian(M) = \bigodot_{\langle -1,0 \rangle} M + \bigodot_{\langle 1,0 \rangle} M + \bigodot_{\langle 0,-1 \rangle} M + \bigodot_{\langle 0,1 \rangle} M - 4 \times M$

body $OutputImage = Laplacian(Smooth(InputImage))$

図 2 エッジ検出プログラム
Fig. 2 Edge detection program.

スカラー型は C++言語の基本型と同様の 8/16/32/64 ビットの signed/unsigned の整数型と float および double 型，そして本記述言語特有の型である Coord 型のいずれかである．Coord 型は配列のインデックスやサイズを表す行と列を示す整数値のペアであり，〈 と 〉で行と列の値を囲んだ Coord 型のリテラルで表記する．**input** および **output** における入出力変数の定義には，スカラー型の変数名には上付き添え字で型を指定する．配列型の変数名には上付き添え字で要素の型を指定し，下付き添え字で行と列のサイズを Coord 型の定数で指定する．入出力変数以外は型の宣言は必要なく，プログラムの処理過程で必要な中間値の型は入力変数の型に基づいて C++のデフォルトの型変換ルール[20] に従って決定される．

function および **body** において関数本体およびトップレベルの式を記述するのに用いる基本演算（四則演算，シフト演算，論理演算，関係演算）の種類は C++と同等であるが，オペランドが配列の場合は配列の要素間の演算になる．オペランドに配列型とスカラー型が混在している場合にはスカラーは拡張されて，他の配列と同じ大きさで各要素が同じスカラー値である配列として扱われる．Coord 型のデータも同様で，Coord 型間の演算は行と列それぞれの間の演算になり，また Coord 型とスカラー型の間の演算は，同じスカラー値が Coord 型の行と列それぞれと演算する．

式の記述に用いる基本演算以外の演算として，組み込みのアグリゲーション関数と配列操作用の演算子を備えている．アグリゲーション関数は配列全体を入力として値を求める演算であり，全要素の総和を求める $\sum$，全要素中の最大値を求める $\triangle$，その最大値の配列中のインデックスを返す $\vec{\triangle}$，同様に最小値を求める $\triangledown$，最小値のインデックスを返す $\vec{\triangledown}$ を用意している．配列操作用の演算子に関しては次節で説明する．

3.2 ループレスな配列処理のための言語機能

簡単な配列間の演算のようなプログラムは 3.1 節で述べた機能を組み合わせて記述できる．しかし従来の手続き型言語においては多重ループで記述していたような複雑な処理を記述するには限界がある．そこで，より複雑な処理をループレスで簡潔に記述するために本言語では配列の構造を操作する $\bigodot$ と $\bigoplus$ の 2 つの演算子と，配列の要素ごとに関数を適用する @ 付き関数呼び出しの機能を導入する．

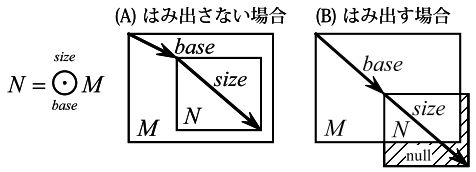

図 3 配列の切り出し演算
Fig. 3 Sub-arrray extraction operator.

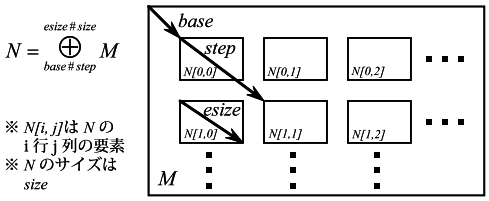

図 4 配列の繰返し切り出し演算
Fig. 4 Repeated sub-array extraction operator.

$\bigodot$ は**図 3** に示すように既存の配列から部分配列を切り出す演算子である．プログラム中では

$$\bigodot_{base}^{size} M$$

の形で記述し，切り出し開始位置をパラメータ *base* に，サイズを *size* に，それぞれ Coord 型の値で指定する．サイズが省略された場合はもとの配列 M と同じサイズになる．切り出す領域がもとの配列からはみ出た場合はその部分に null という特別な値が埋まる．なお基本演算のオペランドに null 値があると演算結果は null になる．

$\bigodot$ 演算子は既存の配列から部分配列を 1 つ切り出す操作であるのに対し，$\bigoplus$ 演算子は**図 4** に示すように既存の配列から複数の部分配列を繰り返して切り出し，それらの部分配列を要素とする配列を作成する操作である．プログラム中では

$$\bigoplus_{base\sharp step}^{esize\sharp size} M$$

の形で記述し，パラメータ *base* に最初の要素の切り出し開始位置を，*step* に切り出し位置の行方向および列方向のずらし幅を，*esize* に切り出す個々の配列のサイズを，*size* に切り出す配列の行方向および列方向の個数を，それぞれ Coord 型の値で指定する．すなわちこの演算子の返す値はサイズが *size* の配列で，その i 行 j 列の要素は配列 M から

$$\bigodot_{base+step\times\langle i,j\rangle}^{esize} M$$

で切り出したサイズ esize の部分配列になる．なお，$\bigoplus$ 演算の結果に対してさらに $\bigoplus$ を適用することは許さない．

式中で関数を呼び出す場合，関数呼び出しの引数に配列を指定すると配列全体を関数の仮引数に対応づけて関数が適用される．それに対して関数名の前に @ を付けると基本演算と同様に配列の要素ごとに関数が適用される．これを @ 付き関数呼び出しと呼ぶ．配列型の引数が複数ある場合には，同じインデックスの要素の組に対してそれぞれ関数が呼び出される．配列型の引数と混在したスカラー型の引数は同じサイズの配列に拡張されるのも基本演算の場合と同様である．配列の要素ごとに呼び出された関数の結果は 1 つの配列にまとめられる．@ 付き関数呼び出しは多くの関数型の言語で用いられる map に，あるいは手続き型言語におけるネストしたループに相当する．

3.3 画像処理プログラムの記述例

図 2 のエッジ検出[1),15)] プログラムでは，画像データの平滑化を行う *Smooth*() と，4-近傍を用いた 2 次微分値を求める *Laplacian*() の 2 つの関数を **function** に宣言し，**body** でそれらを合成してエッジ検出処理を記述している．*Laplacian*() は，画像データ全体を保持する配列 M から上下左右に 1 つずれた 4 つの配列を作り，それら 4 つの配列の和からもとの配列を 4 倍して引くことでラプラシアンフィルタの計算を記述している．*Smooth*() も同様に各画素データに対してその周囲 8 画素との平均値を求めるのだが，平均値を計算するための割り算のオーバヘッドを削減するために 9 で割らずにそのまま結果を返している．次の段の *Laplacian*() の処理は画素の相対値を使ってエッジを検出するため，各画素値が 9 倍になっていても問題はないためである．

基本演算のオペランドに null 値があると演算結果は null になるので，たとえば *Smooth*() の例では，画像の端の画素に対してはその上下左右いずれかの画素値が null になるので結果は null になる．さらにその値を参照する *Laplacian*() に関しては画面の四方の端 2 画素分が null になる．このように null 値の性質を使うと，画像処理のプログラムで頻出する画面の端の例外処理の記述を簡略化できる．

なお，*Smooth*() 関数は $\bigoplus$ 演算子と @ 付き関数呼び出しを使うと

$$Smooth(N) = @\sum \bigoplus_{\langle -1,-1\rangle\sharp\langle 1,1\rangle}^{\langle 3,3\rangle\sharp\rho_N} N$$

とコンパクトに書くことができる．この例では $\bigoplus$ を使って画像の各画素の周辺の 9 画素からなる 3 行 3 列の配列からなる配列を作り，その各配列に対して $\sum$ を @ 付き関数呼び出しで適用している．ここで $\bigoplus$ のパラメータ指定に用いている ρ_N は配列 N のサイズを Coord 型の値で返す組み込み関数である．総和を求める $\sum$ は配列内に null 値があれば結果は null になる．

図 5 はブロックマッチングによる動き検出を用いたオプティカルフロー計算のプログラム[1)] を $\bigoplus$ 演算子と @ 付き関数呼び出しを用いて記述した例である．処理の流れは**図 6** に示すように，*Previous* と *Current* の 2 枚の画像データに対して，*Current* 上に一定の間隔でとったサンプル点が *Previous* 上ではどこに相当するかを検出することで画面上の動きを検出する．*body* に記述したトップレベル式では，*Previous* 上の各サンプル点の周囲 *Psize* すなわち 60 行 60 列の領域と，*Current* 上の各サンプル点の周囲 *Csize* すなわち 8 行 8 列の領域に対応する，2 つの配列の配列を $\bigoplus$ 演算子で作り，その対応する組に対して *Search*() を @ 付き関数呼び出しで適用する．*Search*() はこの 2 つの配列 *dest* と *src* に対して，*dest* 中のすべての 8 行 8 列の領域を $\bigoplus$ 演算子で作り，その各要素と *src* の類似度を SAD（Sum of Absolute Differences）計算によって求め，一番類似度の高い要素のインデックス値と *dest* の中心との差を計算することで各点の動きベクトルを求めている．ここで @*src* はスカラーの拡張と同じことを配列に対して行う操作であり，

$$\bigoplus_{\langle 0,0\rangle\sharp\langle 0,0\rangle}^{\rho_{src}\sharp\rho_{dest}-\rho_{src}+1} src$$

と書いたのと同じ意味を持つ．*Search*() が条件式になっているのは，コントラストが低くて有効なマッチングがとれない点を，配列 *src* 中の最大値と最小値の差が *threshold* より小さいことで検出して無駄な計算を省くためである．

$$
\begin{array}{ll}
\textbf{module} & OpticalFlow \\
\textbf{input} & Previous^{uint8}_{imagesize},\quad Current^{uint8}_{imagesize} \\
\textbf{output} & MotionVector^{Coord}_{blks} \\
\textbf{param} & imagesize = \langle 480, 720\rangle,\quad threshold = 30 \\
 & Csize = \langle 8, 8\rangle,\quad Psize = \langle 60, 60\rangle \\
 & Cbase = \langle 0, 0\rangle,\quad Pbase = Cbase - (Psize - Csize)/2 \\
 & step = \langle 8, 8\rangle,\quad blks = (imagesize - Csize - Cbase)/step + 1 \\
\textbf{function} & Search(dest, src) = \begin{cases} \langle 0,0\rangle & when \quad \triangle src - \triangledown src < threshold \\ \vec{\triangledown}@SAD\left(\bigoplus_{\langle 0,0\rangle \sharp \langle 1,1\rangle}^{\rho_{src}\sharp\rho_{dest}-\rho_{src}+1} dest, @src\right) - (\rho_{dest} - \rho_{src})/2 & otherwise \end{cases} \\
 & SAD(x, y) = \sum \left| x - y \right| \\
\textbf{body} & MotionVector = @Search\left(\bigoplus_{Pbase\sharp step}^{Psize\sharp blks} Previous, \bigoplus_{Cbase\sharp step}^{Csize\sharp blks} Current\right)
\end{array}
$$

図 5 ブロックマッチングによる動き検出を用いたオプティカルフロー計算プログラム
Fig. 5 Optical flow calculation program using block matching for motion detection.

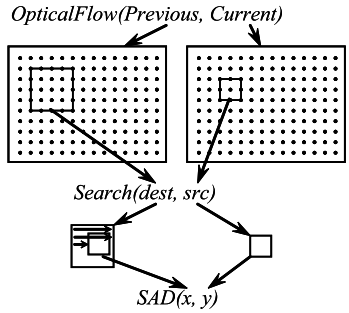

図 6 オプティカルフローの計算手順
Fig. 6 Calculation of optical flow.

3.4 記述言語の特徴と C++との連携

与えられた配列から必要に応じて部分配列を作って計算に用いることで，ループを使わずに画像処理を記述するのが本論文の言語のプログラミングスタイルである．部分配列は仮想的に実装されるものであって実際に配列の部分コピーを作ることはないので，新たなメモリ領域やデータコピーのオーバヘッドを気にしなくてもよい．その処理に必要な座標変換は処理系内部で行うので，プログラマが複雑なインデックス計算の式を記述しなければならない場面は少なくなる．

配列の全要素に独立に作用する演算を組み合わせて画像全体にわたる処理を記述するため，画像の局所処理のような並列性のある画像処理アルゴリズム[15]は，本言語で記述して次章以降で述べる方式で効率良く実行することができる．たとえば，各画素に対して近傍画素から独立に値を計算できる各種の画像補正やフィルタの処理，エッジ検出等の特徴抽出処理，画像認識に用いられる細線化等の前処理，動画像の複数フレームのデータを使った物体抽出や動き検出，複数の視点からの画像を用いたステレオマッチングによる立体認識のような処理，さらにそれらの処理の組合せ等は，本言語で記述可能な問題である．

一方，逐次的な依存性のある処理，すなわち配列の要素間で処理の順序付けが必要な処理は本言語ではうまく記述できない．たとえば画像の境界線抽出や連結成分のラベル付けのように，途中までの処理結果に基づいて次に処理する画素を決めるような処理はその典型である．また，画像圧縮におけるジグザグスキャンのような決まった順序で処理することが必要な場合も同様である．なお，問題によっては関数を再帰的に呼び出すことで記述できる場合もあるが，どのような問題にも一般的に適用できる手法ではない．

このように本言語には得意な問題と不得意な問題があるが，本言語で記述したプログラムは C++のコードに変換して実行する方式を用いており，本言語が得意な処理はこの言語で記述し，それ以外の処理は C++で記述してリンクして実行する．たとえば画像認識の前処理や特徴抽出といった並列性を持つ処理は本言語で記述し，残りの認識処理は C++で記述する．

なお図 2 や図 5 のプログラム例にも示すように，本言語では数式に近い記法でプログラムを記述する．このような記法は，アグリゲーション関数や配列の切り出し演算子に特別な記号を用いることでプログラムを直感的に把握しやすくする効果を期待して採用した．プログラムは数式エディタを使って作成し，MathML[18]に変換して C++へのトランスレータに入力する．

4. 実行制御のためのワーキングセット情報

ループレスで記述したプログラムを効率良く実行できるように，ターゲットのメモリアーキテクチャを考慮した実行順序のスケジューリングを行う．その際に，トップレベルの配列の各要素の処理に必要なデータの範囲を表すワーキングセットと呼ぶ情報を求めて利用

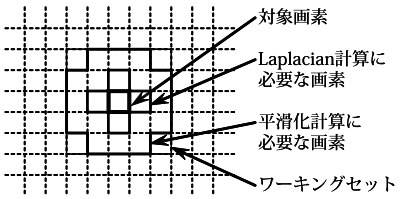

図 7 エッジ検出プログラムのワーキングセット
Fig. 7 Working set of EdgeDetection program.

する．

4.1 ワーキングセットの例

プログラムの **body** に書かれたトップレベルの式は，結果として配列を返すか，あるいは配列に対して $\sum$ や $\bigtriangledown$ 等のアグリゲーション関数を適用した値を返す．この配列のことを本論文では結果配列と呼ぶ．結果配列のサイズはこのプログラムが解かなければならない問題のサイズを反映している．

結果配列のインデックス $\langle i,j\rangle$ の要素の値を求めるのに必要なデータは，各入出力変数の配列上の特定の部分配列である．その部分配列の集合を結果配列の $\langle i,j\rangle$ の要素のワーキングセットと呼ぶ．ワーキングセットには結果を記録するための出力変数上の対応する領域も含まれる．

たとえば図 2 のエッジ検出プログラムの結果配列の $\langle i,j\rangle$ の画素値のワーキングセットは，結果を格納するための出力変数 *OutputImage* の $\langle i,j\rangle$ の 1 画素分のデータに加えて，**図 7** に示すように入力変数 *InputImage* の $\langle i,j\rangle$ の上下左右あわせた 5 画素に対して，それぞれの周辺 9 画素を選択した領域になる．ただし以降の処理を簡単にするためにワーキングセットは最小の矩形領域で近似して扱うので，*InputImage* のワーキングセットは $\langle i-2,j-2\rangle$ をベースとするサイズ $\langle 5,5\rangle$ の矩形領域になる．

図 5 のオプティカルフローのプログラムの場合には，$\langle i,j\rangle$ の位置の動きベクトルを求めるのに必要なワーキングセットは，**body** の式の *Search*() 関数の引数になっている 2 つの配列の配列

$$\bigoplus_{Pbase\sharp step}^{Psize\sharp blks} Previous, \qquad \bigoplus_{Cbase\sharp step}^{Csize\sharp blks} Current$$

と結果を返す配列 *MotionVector* のそれぞれの $\langle i,j\rangle$ の要素になる．ここでワーキングセットの決定に際しては，@ 付きで呼び出している *Search*() 関数より先の処理は無視してかまわない．これは図 6 に示すように，*Search*() の処理ではその引数に与えられた部分配列の範囲内のデータしか参照しないからである．本記述言語では関数内で参照するデータは必ず引数として渡されるため，この性質は @ 付き関数呼び出しに対して一般的に成り立つ．

4.2 ワーキングセット情報の求め方

ワーキングセットはプログラムの **body** に書かれたトップレベルの式を展開して，そのオペランドに適用された $\bigodot$ 演算子や $\bigoplus$ 演算子のパラメータを見ることで簡単に計算することができる．その手順は，

(1) 式の標準形への変換
(2) オペランドごとのワーキングセット情報の計算
(3) 入出力変数ごとのワーキングセット情報の計算

の 3 つのステップからなる．

第 1 ステップではまず，トップレベルの式中に現れる @ の付かない関数呼び出しを **function** に書かれた定義に従って展開する．@ 付き関数呼び出しはそれ以上展開する必要はない．

次に必要に応じて独立に処理する複数の式に分割する．たとえば画面全体の画素値の最大値と最小値を求め，その値を使って各画素値を 0 から 255 の間に広く分布するように補正してコントラストを高める次のような処理を考える．

$$(M-\bigtriangledown M)\times 255/(\bigtriangleup M-\bigtriangledown M)$$

このとき，アグリゲーション関数 $\bigtriangleup$ と $\bigtriangledown$ で最大値と最小値を求める処理と補正処理は 1 つのループで同時に実行することはできないので，まず最大値と最小値を求め，その後で補正処理をするように分割する．一般的には，式中でアグリゲーション関数を配列に適用し，その結果のスカラー値がさらに他の配列に合わせて拡張されている部分が分割する点である．

以上の操作によってトップレベルの式は，オペランドが基本演算と @ 付き関数呼び出しで結合されている標準形に変換できる．このとき各オペランドは，式の結果配列と同じサイズの配列かスカラーのいずれかになっている．そのため，結果配列の要素 $\langle i,j\rangle$ のワーキングセットは各オペランドのインデックス $\langle i,j\rangle$ の要素の集合に相当する．

第 2 ステップではすべての配列型のオペランド O_k のワーキングセット情報を計算する．まず配列型の各オペランド O_k は，M を入出力変数として，

$$M, \qquad \bigodot_{base}^{size} M, \qquad \bigoplus_{base\sharp step}^{esize\sharp size} M$$

のいずれかの形に正規化できる．これは，$\bigoplus$ は多重に使えないことに加え，$\bigodot$ を多重に適用する場合は 1 つの $\bigodot$ に併合し，$\bigoplus$ の前後に $\bigodot$ を組み合わせる場合は 1 つの $\bigoplus$ に併合してワーキングセット情報を計算できるためである．

オペランドのワーキングセット情報は，正規化した各オペランド O_k に対して**表 1** のように定義した

表 1 オペランドのワーキングセット情報
Table 1 Working set information of operands.

O_k	$owbase(O_k)$	$owstep(O_k)$	$owsize(O_k)$
M	$\langle 0,0\rangle$	$\langle 1,1\rangle$	$\langle 1,1\rangle$
$\bigodot_{base}^{size} M$	$base$	$\langle 1,1\rangle$	$\langle 1,1\rangle$
$\bigoplus_{base\sharp step}^{esize\sharp size} M$	$base$	$step$	$esize$

$owbase(O_k)$，$owstep(O_k)$，$owsize(O_k)$ と，オペランド O_k が参照する入出力変数 M を返す $owvar(O_k)$ の 4 つ組で定義する．この情報は，結果配列のインデックス $\langle i,j\rangle$ の要素の計算に用いる O_k の要素は，入出力変数 $owvar(O_k)$ の $owbase(O_k)+owstep(O_k)\times\langle i,j\rangle$ から始まる $owsize(O_k)$ の大きさの範囲のデータ，すなわち本論文の記述言語で表記すると

$$\bigodot_{owbase(O_k)+owstep(O_k)\times\langle i,j\rangle}^{owsize(O_k)} owvar(O_k)$$

であることを示している．

第 3 ステップではオペランドのワーキングセット情報を変数ごとにまとめて，変数のワーキングセット情報を作成する．このとき本システムの記述対象が画像の局所処理であることから，同じ入出力変数をアクセスするオペランド O_k と O_l に対して，

(1) $owbase(O_k)$ と $owbase(O_l)$ は近接している，
(2) $owstep(O_k)$ と $owstep(O_l)$ は等しい，

という 2 つの前提をおく．もしこの前提が成り立たない変数がある場合には $owvar(O_k)$ と $owvar(O_l)$ は仮想的に違う変数として扱う．

プログラム中のすべての配列型の入出力変数 A_l のワーキングセット情報は，$wbase(A_l)$，$wstep(A_l)$，$wsize(A_l)$，$ebyte(A_l)$ の 4 つ組で定義する．$wbase(A_l)$ は入出力変数 A_l をアクセスするすべてのオペランド O_k のワーキングセット情報から

$$\min_k\big(owbase(O_k)\big)$$

と計算し，さらに $wsize(A_l)$ は

$$\max_k\big(owbase(O_k)+owsize(O_k)\big)-wbase(A_l)$$

と計算する．また，$wstep(A_l)$ は先に述べた前提から $owstep(O_k)$ に等しく，$ebyte(A_l)$ はこの配列 A_l の要素のメモリ上のサイズである．

こうして求めた変数のワーキングセット情報を用いると，結果配列の $\langle i,j\rangle$ の要素のワーキングセットは配列型のすべての入出力変数 A_l の

$$\bigodot_{wbase(A_l)+wstep(A_l)\times\langle i,j\rangle}^{wsize(A_l)} A_l$$

の集合になる．各入出力変数の $\langle i,j\rangle$ のワーキングセットは下隣の $\langle i+1,j\rangle$ や右隣の $\langle i,j+1\rangle$ のワーキングセットと重なりがある場合もある．このような重なりをうまく利用すればメモリアクセスを削減する実行制御が可能になる．

4.3 レンジのワーキングセット

前節で求めた結果配列の $\langle i,j\rangle$ の要素のワーキングセットと同様に，レンジのワーキングセットすなわち結果配列の $\langle i,j\rangle$ から始まるサイズ $\langle r,c\rangle$ の範囲の値の計算時にアクセスする各入出力変数の範囲を定義できる．各変数 A_l のアクセス範囲 $Area(A_l,i,j,r,c)$ を

$$\bigodot_{wbase(A_l)+wstep(A_l)\times\langle i,j\rangle}^{wsize(A_l)+wstep(A_l)\times\langle r-1,c-1\rangle} A_l$$

と定義すると，その集合がレンジのワーキングセットになる．

$Area(A_l,i,j,r,c)$ の要素数 $Elements(A_l,r,c)$ は

$$wsize(A_l)+wstep(A_l)\times\langle r-1,c-1\rangle$$

の行成分と列成分の積で計算でき，その範囲のデータを記憶するのに必要なメモリ量 $Memory(A_l,r,c)$ は

$$Elements(A_l,r,c)\times ebyte(A_l)$$

となる．次章で説明するように，このメモリ量 $Memory(A_l,r,c)$ を使ってレンジスケジューリングにおけるレンジサイズを決定する．

例として図 5 のオプティカルフローのプログラムの結果配列の $\langle i,j\rangle$ から始まるサイズ $\langle r,c\rangle$ のレンジのワーキングセットを求めてみる．まずレンジは結果配列上に定義されるので，それに対応する出力変数 $MotionVector$ の範囲は

$$\bigodot_{\langle i,j\rangle}^{\langle r,c\rangle} MotionVector$$

となる．さらにその範囲の処理に必要な入力変数 $Previous$ の範囲は

$$\bigodot_{\langle -26+8\times i,-26+8\times j\rangle}^{\langle 60+8\times(r-1),60+8\times(c-1)\rangle} Previous$$

となり，もう 1 つの入力変数 $Current$ の範囲は

$$\bigodot_{\langle 8\times i,8\times j\rangle}^{\langle 8+8\times(r-1),8+8\times(c-1)\rangle} Current$$

となる．この 3 つの部分配列の集合がレンジのワーキングセットになる．

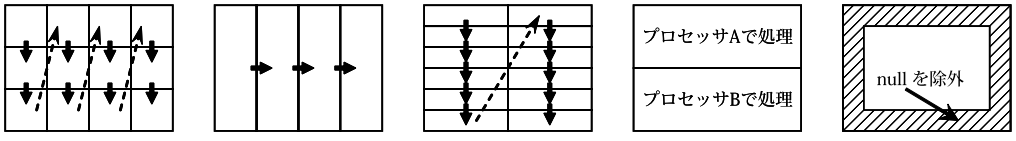

図 8 レンジスケジューリング
Fig. 8 Range scheduling.

この例のように，レンジすなわち結果配列上の範囲を決めると，そのレンジのワーキングセットを計算することで，レンジの処理に必要な各入出力変数の部分配列が決まる．このとき部分配列の大きさはプログラムによってさまざまで，必ずしも小さなサイズの配列にならず元の配列全体になる場合もある．たとえば元画像とマスク配列が入力として与えられ，元画像の各画素の近傍画素とマスク配列全体を使って計算した値を対応する結果配列の画素値にするようなフィルタの処理を考える．このとき，結果配列上に設定したレンジのワーキングセットは，元画像上の対応する部分配列と，マスク配列全体からなる．すなわち，レンジをどのような大きさに設定しても，マスク配列は必ず配列全体がワーキングセットに含まれることになる．

5. レンジスケジューリングによる実行制御

ターゲットシステムのメモリアーキテクチャに適したプログラムの実行プランは，ワーキングセット情報を利用したレンジスケジューリングと呼ぶ方式によって作成する．本章ではレンジサイズを決めるヒューリスティクスを中心に，デュアルコア DSP 上での実験結果を交えて説明する．

5.1 レンジスケジューリング

本システムの言語で記述したプログラムは C++のプログラムに変換して実行する．基本的な変換パターンは，*body* に書かれたトップレベルの式を展開し，その結果配列の計算をするために 2 重ループを生成する．さらにその中で @ 付き関数呼び出しがあればそれに対応する配列処理のための 2 重ループを入れ子にする．

しかし基本的な変換だけで得られるプログラムは，たとえば 3.3 節で説明した *Laplacian*() の処理を C 言語で

```
for(i=1; i<479; i++)
  for(j=1; j<719; j++)
    R[i][j]=M[i+1][j]+M[i-1][j]+M[i][j+1]
           +M[i][j-1]-4*M[i][j];
```

と書いたようなものに等しい．このプログラムはこのままでどんなプロセッサ上でも効率良く動くわけではない．プロセッサのキャッシュサイズが小さい場合にはループブロッキングを行うように書き換えればキャッシュミスを減らせる．またデータを DMA でローカルメモリに持ってきて処理するアーキテクチャの場合には，適切な単位でステージングを行ってデータの転送と処理がオーバラップするように書き換えて性能を引き出せる．

本論文のシステムでは，このような書き換えをレンジスケジューリングと呼ぶ考え方で統一的に扱う．プログラムのトップレベルの式の結果配列全体を最上位のレンジとし，そのレンジを以下で説明するさまざまな観点からさらに小さい矩形領域のレンジに区切るとともに，区切ったレンジ間の実行順序を決めていく．最も小さなレンジ内は上の例と同様に単純な 2 重ループで処理する．

レンジの区切り方とレンジ間の実行順序は，

(1) ローカルメモリへのデータのステージング
(2) キャッシュを考慮したループブロッキング
(3) 式の評価の中間結果領域の管理
(4) マルチコアプロセッサでの並列処理
(5) null 値を含む配列の処理の最適化

の観点から，プログラムのワーキングセットとターゲットシステムのメモリアーキテクチャを考慮して決める．**図 8** にそれぞれの場合のレンジの決め方の例を示す．レンジは階層的に区切られるため，図 8 のそれぞれの図は上位の階層で区切られたレンジをさらにどう区切るかを示している．すなわち図 8 の各図の最も外側の枠は上位から与えられたレンジを表し，その中にある矩形領域がそれぞれの観点で区切ったレンジを表す．(1) から (3) において各レンジ間をつなぐ矢印はレンジを処理する順序を示している．(1) から (3) に関しては次節以降で詳しく説明する．図 8 の (4) はマルチコアの場合の最も単純なスケジューリングであり，レンジをコア数で分割してそれぞれのコアが処理を行う方式である．この例ではレンジを 2 つに分けて 2 つのプロセッサで並列に処理する分割例を示している．より細かなレンジに分割してレンジのプールを作り，各コアはそこからレンジを取り出して処理するようにして負荷を均衡化する方式も可能である．図 8 の (5) の

null の最適化は，配列中の null 値の部分を回避してループを回すことで無駄な処理を省くものである．具体的には 3.3 節で述べたような画像の周辺部分の例外的な処理を省く効果がある．

これらのレンジを区切る観点は，たとえばマルチコア用に分割したレンジをさらにステージングの単位で分割し，さらに null 領域を除外したレンジを中間結果のレンジで区切るというように，ターゲットにあわせて適宜組み合わせる．具体的な組合せ方に関しては次節で述べる．

5.2 ローカルメモリへのステージング処理

ステージング処理，すなわち外部の主記憶にあるデータを DMA でローカルメモリに持ってきて処理するタイプのプロセッサでは，1 回のステージングの範囲をレンジに対応させる．ステージングに使うローカルメモリのサイズを LS とするとき，レンジのサイズが $\langle r, c\rangle$ とすると，ダブルバッファで DMA と演算処理を並列に進めるためには，すべての配列型の入出力変数 A_k に対して

$$\sum_k Memory(A_k, r, c) \leq LS/2$$

となることが必須条件である．

この条件を満たすようにレンジサイズが決まれば，図 8 の (1) に示すように，その大きさで上位のレンジを複数のレンジに区切り，それらのレンジを上から下へ，そして左から右へと処理を進める．下方向を優先するのは，隣接するレンジのワーキングセット間で重複するデータをリングバッファで管理する場合に，行方向のデータが分離しないようにしてプログラムの SIMD 化やベクトル化をしやすくするためである．

レンジサイズ $\langle r, c\rangle$ を決める戦略には幅優先，要素数優先，高さ優先の 3 種類が考えられる．それぞれ幅 c，要素数 $r \times c$，高さ r が前記の条件を満たす範囲内でできるだけ大きくなるようにサイズを決める．左右に隣接するレンジのワーキングセットに重なりがある場合には，幅優先にすることで主記憶からのデータ転送量を削減できる．一方，処理の対象領域がリングバッファのリングのつなぎ目にかかる頻度は高さ優先の方が少ないため，実行時の座標変換のオーバヘッドが小さくなる．最適なレンジサイズはアプリケーションプログラムに依存するので 1 つの戦略に決めるのは難しいが，本システムではデフォルトの戦略として要素数優先を採用する．これは幅優先と高さ優先の中間的な性質を持つことと，多くのアプリケーションで DMA 回数削減の効果が期待できるためである．

デュアルコア DSP である BF561（コアクロック 600 MHz）上で図 2 のエッジ検出と図 5 のオプティカルフローのプログラムを幅優先，要素数優先，高さ優先の 3 種類の戦略で実行したときの評価データを**表 2** に示す．このとき，トップレベルのレンジは結果配列のサイズから決まり，エッジ検出は *OutputImage* のサイズである $\langle 480, 720\rangle$，オプティカルフローは *MotionVector* のサイズである $\langle 60, 90\rangle$ になる．そのトップレベルのレンジをまず 2 つのコアで並列処理するために図 8 の (4) のように上下 2 つのレンジに分け，そのレンジをステージングのレンジサイズで図 8 の (1) のように分割し，次に図 8 の (5) のように null 領域の除外の最適化を行っている．さらにエッジ検出に関しては図 8 の (3) のように中間結果領域を考慮したレンジ分割を行っており，これに関しては 5.4 節で詳しく説明する．

表 2 BF561 でのステージング戦略の評価
Table 2 Evaluation of staging strategy on BF561.

		幅優先	要素数優先	高さ優先
Edge-Detection	size	$\langle 11, 360\rangle$	$\langle 64, 64\rangle$	$\langle 240, 16\rangle$
	cycle	$1.25e7$	$1.19e7$	$1.35e7$
	dma	46	50	48
	word	$1.30e5$	$1.35e5$	$1.52e5$
Optical-Flow $Psize = \langle 60, 60\rangle$	size	$\langle 1, 10\rangle$	$\langle 5, 4\rangle$	$\langle 9, 1\rangle$
	cycle	$7.45e9$	$7.32e9$	$7.32e9$
	dma	302	146	362
	word	$1.69e5$	$1.81e5$	$4.23e5$
Optical-Flow $Psize = \langle 32, 32\rangle$	size	$\langle 1, 28\rangle$	$\langle 8, 8\rangle$	$\langle 25, 1\rangle$
	cycle	$1.73e9$	$1.65e9$	$1.64e9$
	dma	122	50	182
	word	$1.02e5$	$1.11e5$	$2.28e5$

$1.23e5$ は 1.23×10^5 を表す

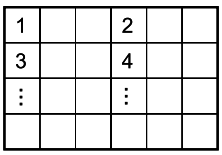

図 9 ダブルリングバッファリング時のレンジの処理順序
Fig. 9 Evaluation sequence for double ring buffering.

画像データは主記憶である SDRAM におき，ステージングには 32 KB のローカルメモリ上に作成した 2 つの 16 KB のリングバッファを用いる．ステージングのために分割したレンジの実行順序は図 8 の (1) の基本順序を変形して，**図 9** に示す順に処理を進める．図 9 は図 8 の (1) と同様に上位のレンジをステージングのサイズで分割したレンジの集合を表しており，そのレンジ群を左右に分け，それぞれから図中に示した番号の順に交互にレンジを選択して処理する．このようなレンジの進め方をする理由は，並列に動作可能な 4 KB のサブバンクから構成されている BF561 の

ローカルメモリを活かすためである．すなわち，隣接するレンジのワーキングセットに重なりがある場合はローカルメモリをリングバッファとして使って重なり部分のデータを再利用する．このとき，あるレンジを処理中に隣のレンジのデータを DMA 転送すると，プロセッサと DMA コントローラが同じサブバンクにあるデータにアクセスする状況が発生する．このような状況を避けるために，2 つのリングバッファを異なるサブバンク上に用意し，図 9 に示す順序で，バッファ A を使ってレンジ 1 を処理している間にバッファB にレンジ 2 のデータを転送し，バッファB を使ってレンジ 2 を処理している間にバッファA にレンジ 3 のデータを転送するようにすると，レンジ 1 と 3 のような隣接するレンジで重なるデータを再利用したとしても，プロセッサと DMA コントローラの間で同一サブバンクへのアクセスが発生しない．以上のサブバンクの最適化に加えて，プロセッサからも 2 つのオペランドを同時にアクセスできるように，各リングバッファ内をさらに異なるサブバンク上の 2 つの 8 KB の領域に分け，各入出力変数のデータはいずれかの領域内に収まるように制約をかけてレンジのサイズを計算する．

評価はそれぞれのプログラムに対して，選ばれたレンジサイズ（size），実行に要したクロックサイクル数（cycle），総 DMA 回数（dma），全転送ワード数（word）を測定した．オプティカルフローは *Psize* を $\langle 60, 60\rangle$ と $\langle 32, 32\rangle$ に変えてワーキングセットの大きさが異なる 2 通りの場合を測定した．このときプログラム中の閾値 *threshold* の値を 0 にして最大負荷になるように設定した．表 2 の結果から分かるように，サイクル数で見ると，演算量の小さいエッジ検出プログラムでは要素数優先の結果が良くなっている．演算量が多いオプティカルフローではリングバッファのオーバヘッドが少ない高さ優先で良い結果が出ている．ただし，幅優先と要素数優先のサイクル数の差に比べて高さ優先と要素数優先の差は小さいことや，高さ優先における転送ワード数の多さを考えると，デフォルトを要素数優先とするのは妥当な選択である．

5.3 キャッシュヒット率の向上

プロセッサがキャッシュメモリを持つ場合には図 8 の (2) のように，与えられたレンジをキャッシュメモリが有効に働くことが期待できる幅のレンジに分割するループブロッキング[2] を行う．n ウェイのキャッシュメモリの場合のレンジの幅は以下の手順で決める．

まずすべての入出力変数 A_k を n 個のグループ G_i に分ける．この分け方は各変数の $wsize(A_k)$ の範囲を記憶するのに必要なメモリ量の和がグループ間で均衡するような近似解でよい．キャッシュメモリの容量を CS とするとき，各グループ G_i に対して，それに属する各入出力変数 A_k のワーキングセット情報から

$$\sum_{A_k \in G_i} Memory(A_k, 1, w_i) \leq CS/n$$

を満たす最大の w_i を求め，その最小値を正規化した

$$\lceil width / \lceil width / \min_i(w_i) \rceil \rceil$$

をレンジの幅にする．正規化によって同じ分割数を持つ最小の幅にすることでキャッシュのヒット率向上が期待できる．*width* は上位のレンジの幅である．レンジの高さは上位のレンジと同じにする．

BF561 でローカルメモリのうち 16 KB をデータキャッシュとして使う構成にしたときの，エッジ検出とオプティカルフローのプログラムの評価データを**図 10** と**図 11** に示す．このとき 5.2 節における評価と同様に，まずトップレベルのレンジを 2 つのコアで並列処理するために上下 2 つのレンジに分け，そのレンジをループブロッキングのレンジサイズで図 8 の (2) のように分割し，次に null 領域の除外の最適化を行って

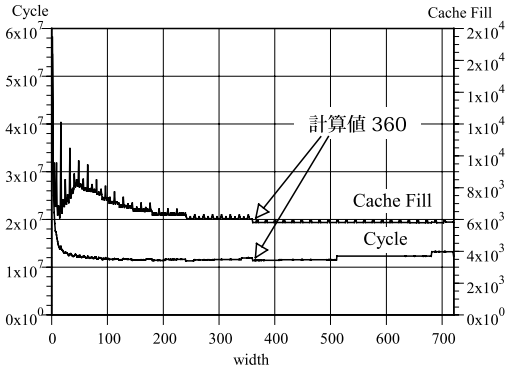

図 10 エッジ検出プログラムの評価
Fig. 10 Evaluation of EdgeDetection program.

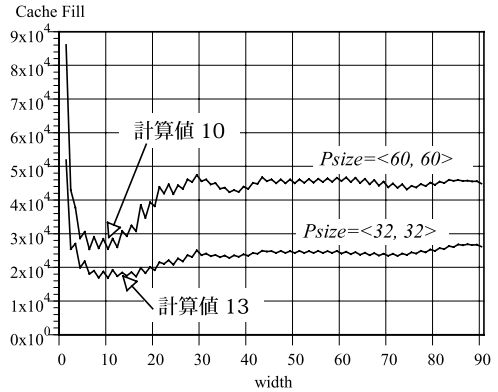

図 11 オプティカルフロープログラムの評価
Fig. 11 Evaluation of OpticalFlow program.

いる．さらにエッジ検出に関しては 5.4 節で説明する中間結果領域を考慮したレンジ分割を行っている．

図 10 と図 11 のグラフは，ブロッキングするレンジの幅を 1 から最大幅（結果配列全体の幅）まで変化させた場合の 1 つのコアのキャッシュフィル回数と実行に要したサイクル数（エッジ検出のみ）をプロットしたものに，上記の計算法で求めた最適なレンジ幅を記したものである．どちらの場合もキャッシュフィル回数が底を打つあたりのレンジ幅を計算できており，エッジ検出では実行時間に関しても良い値になっている．なおオプティカルフローは演算量が多いので，レンジの幅によらず実行時間はほぼ一定である．

BF561 のデータキャッシュは 2 ウェイセットアソシアティブ方式で，16 KB のキャッシュバンクは 4 つの 4 KB のサブバンクから構成される．各サブバンクは 64 のセットからなり，ブロックサイズは 32 バイトである．メモリは 4 KB 単位で 4 つのサブバンクに順に対応付けられているため，メモリアドレスが決まるとキャッシュされるサブバンクが一意に決まるのが特長である[14]．そこで，各入出力変数 A_k の $wsize(A_k)$ の範囲を記憶するのに必要なメモリ量の最大値を求め，その値が 1 KB 以下なら 1 ウェイ分のキャッシュサイズは 2 KB，2 KB 以下なら 4 KB，それ以上なら 8 KB として上記のレンジ幅の計算を行っている．

5.4 式の評価の中間結果領域の管理

エッジ検出プログラムにおいて画像データを平滑化してラプラスフィルタにかけるように，複数の関数を合成した処理の場合には，関数の適用を展開してしまうと関数の仮引数に対応する中間結果を保持する領域は不要になる．しかし平滑化後の各画素値は周辺の複数の画素のラプラスフィルタの処理に必要であることからも分かるように，一般には中間結果を保持せずに結果配列の要素ごとに毎回必要な計算をすることは計算量の観点からは最適ではない．一方，平滑化処理後の中間結果を一度主記憶に記録し，ラプラスフィルタの処理時に再び主記憶から読み出すように実行することは，メモリアクセスが増加するので好ましくない．そこで，レンジの処理における配列アクセスには局所性があるので，高速なローカルメモリに用意した一時バッファを中間結果の記録用に使いまわしながら少しずつ処理を進めれば効率良く処理ができる．つまり，図 8 の (3) に示すように，この少しずつ進める範囲をレンジとして上位のレンジを細分化して順次実行を進める．

このときのレンジの大きさは中間結果のワーキングセットに相当する情報を用いて，ステージングの場合と同様の方法で決定することができる．すなわち 4.2 節で入出力変数のワーキングセット情報を求めたのと同様の手順で，@ 付き呼び出しではない単純な関数呼び出しの箇所に仮の変数 T_k を置いてそのワーキングセット情報を求めると，これが中間結果のワーキングセット情報になる．中間結果を記憶するのに用いるローカルメモリ上の一時バッファの量を TS とするとき，レンジのサイズ $\langle r, c\rangle$ はすべての配列型の中間結果 T_k に対して

$$\sum_k Memory(T_k, r, c) \leq TS$$

となることが必須条件である．このレンジサイズ $\langle r, c\rangle$ は c ができるだけ大きくなるように決める．これは，隣接するレンジ間で中間結果に重なりがある場合にできるだけ多くの中間結果を再利用するためである．

5.2 および 5.3 節における評価に用いた BF561 上の処理系では，各プロセッサの 4 KB のスクラッチパッドメモリを一時バッファに使っている．この最適化を行わず，中間結果を保持しないで結果配列の各要素ごとに必要な計算をすべて行った場合，5.2 および 5.3 節で示した評価結果の実行に必要なサイクル数がどちらも 7 割程度増加する．使用するスクラッチパッドメモリ上の一時バッファの量を減らすと実行サイクル数が増えることからも，中間結果を記憶するための一時バッファのサイズはできるだけ大きくとれることが望ましい．現在の実装では一時バッファのサイズは固定にしているが，今後ローカルメモリやキャッシュの一部を一時バッファのために利用するようにすると，5.2 および 5.3 節に示したレンジサイズの計算時に一時バッファの最適なサイズをあわせて計算する必要がある．この最適解を求める方式は今後の課題である．

6. 関 連 研 究

6.1 配列処理や画像処理用のプログラミング言語

APL[12] およびその流れをくむ APL2[10] や J 言語[11] は配列をメインのデータ型に位置づけ，配列間の基本演算に加えて配列の構造を操作する多様な組み込み演算子を提供している．それらの演算子を組み合わせることにより，配列を用いるさまざまなプログラムをループレスで簡潔に表現できる．MATLAB[8),16)] も配列を扱う言語として現在広く使われており，さまざまな応用領域向けにさらに多様な組み込み関数を提供している．本論文の記述言語も配列間の演算を数式に準じた記法で組み合わせることで，APL や MATLAB の持つプログラム表現の簡潔さを取り入れることをねらっている．ただしそれらの言語とは異な

り記述対象を画像処理に絞っているので，配列の構造を操作する演算子を配列の（繰返し）切り出しに限定している．

配列間の演算を基本にする APL 等とは異なり，配列の処理の記述に向いた特別な制御構造を導入するアプローチをとっているのが SISAL[7]，SAC[13]，Sassy[9] 等の言語である．これらの言語では，ジェネレータで繰返し範囲を決め，本体で繰返し要素の処理を行い，その処理結果を集約して結果の配列を構成する，特徴的なループ構文を提供している．このような構文の導入により，配列処理のプログラムを書きやすくするとともに並列処理をしやすくしている．配列処理プログラムの簡潔な記述という点では APL 系や本論文の言語には及ばないが，配列演算を組み合わせるスタイルと違って配列の要素ごとに異なる処理が必要なプログラムも記述しやすい特長がある．特に Sassy ではループ伝播依存がある処理も書けるので記述能力は高い．また，SAC と Sassy は C 言語ベースなので従来の C 言語プログラムとも相性が良い．本論文の言語で記述したプログラムと同等のプログラムをこれらの言語の持つループ構文を使って書くことは可能であるので，本論文で述べたメモリアーキテクチャに適応した実行のための最適化手法はこれらの言語にも適用できる．ただしそのためには，次節で説明するようにプログラムのデータ依存性解析を行ってワーキングセット情報を求める必要があり，$\odot$ と $\bigoplus$ 演算子のパラメータから容易にワーキングセット情報を計算できる本論文の言語に比べると実装が複雑になる．なお，これらの言語は並列処理を考慮して関数型あるいは単一代入でプログラムを記述するスタイルをとっているのは，本論文の言語とも共通する．

より画像処理に専用化したプログラミング言語も研究されている．たとえば IAL[6] は本論文と同様に画像の局所処理を対象にした言語で，周辺画素を使った演算を定義するテンプレートと画像データとを組み合わせて処理を記述する．このような問題領域に特殊化した演算をベースにした言語は，汎用の配列演算を組み合わせる APL や本論文の言語と比べると，想定されたパターンの処理は簡潔に記述できるが記述しやすい問題の範囲は小さくなる．

6.2 言語処理系と最適化手法

プログラムの実行に必要なワーキングセット情報を用いてターゲットシステムのメモリアーキテクチャを考慮した実行手順の最適化を行うのが本論文のシステムの特長である．一方，C や Fortran のような手続き型言語の並列化コンパイラでも，データ依存性解析の 1 つとして，プログラムを解析して各実行文が参照あるいは更新する配列の部分を特定して最適化のために用いている[19]．この技術を使えば，配列の大きさがあらかじめ決まっていれば従来の手続き型言語のプログラムでもワーキングセット情報を求めることは可能である．しかしプログラムの解析には，解析システムを開発・保守するためのコストと，解析処理にかかるコストの両方がかかる．前節で述べた SAC や Sassy であれば配列をアクセスする処理は専用のループ構文に集約されるので解析しやすくなるが，インデックス変数の使われ方等を解析する必要がなくなるわけではない．本論文のシステムでは，ワーキングセットの計算に必要な情報を $\odot$ と $\bigoplus$ 演算子のパラメータに書かせることでトランスレータにおける詳細な依存性解析を不要にし，ワーキングセットを計算するコストを下げている．このような方式はプログラマに負担を押し付けていることにもなるが，配列演算を組み合わせた簡潔な記述を可能にすることでその負担を軽減できると考えている．ワーキングセット情報の計算やそれを用いたレンジスケジューリングは実行時に行ってもオーバヘッドにならないくらいに軽くできるのが本方式の特徴であり，論文中で評価に用いた処理系でも仮想配列を実現するクラスが実行時に処理するように実装している．ワーキングセットの計算を簡単にできることは処理系の開発や保守に要するコストの軽減にもつながる．

ターゲットのメモリアーキテクチャにあわせた最適化の手法は，従来から数値計算や並列処理等の分野で使われてきたさまざまな手法にワーキングセット情報を組み合わせている．5.3 節で説明したキャッシュを考慮したレンジサイズの計算は広く使われているループブロッキングのブロックサイズの決定にワーキングセット情報を利用したものである．5.2 節のローカルメモリを使ったステージングはプログラミング技法としては定石であるが，従来は実際にどの部分のデータをメモリに転送して処理するかはプログラマが問題ごとに設計をすることが多かった．ワーキングセット情報を使うことでステージングの範囲の決定と実行順序の制御を自動化できるのが本論文の手法の特長である．中間結果の領域の最適化手法には，配列間の演算の中間結果に必要な領域の削減を目指したものが多い．たとえば数値計算用のクラスライブラリである Blitz++[17] では C++のテンプレートを利用した expression templates と呼ぶ技法を用いて式の中間結果を保持する領域を削減している．それに対して 5.4 節で述べた最適化方式は，画像処理で中間結果を近傍の

複数画素の処理に使うような場合に，中間結果をうまく再利用することを目的としている．

7. おわりに

本論文では，組み込み機器に搭載する画像の局所処理を対象に，特定のターゲットシステムに依存しないように書いたプログラムを，ターゲットのメモリアーキテクチャに適応させて実行する方式を提案した．配列データの処理を関数型でループレスに記述できるように設計した言語は，画像処理の教科書に出てくる抽象度の高い説明に近い形式で簡潔にプログラムを記述できる．ワーキングセットを計算しやすいように言語を設計することで，それを用いた簡単なヒューリスティクスによってターゲットシステムにあわせた実行順序のレンジスケジューリングが可能なことを示した．DSP を用いて典型的な画像の局所処理プログラムで実験した結果でも良好な実行順序が得られている．

今後の課題は言語の記述能力と最適化技術の向上である．本論文のプログラミングシステムは，処理の対象範囲を必要最小限の大きさの部分配列として切り出していくプログラミングスタイルをプログラマに課している．画像の局所処理を超えて画像認識等のより複雑なアルゴリズムを書こうとすると，このような最適なワーキングセットを意識したプログラミングは難しくなる．プログラムの変換時により深く解析してワーキングセットを求められれば，プログラマの負担も減り，より多様なアルゴリズムを記述できるようになると期待できる．

参考文献

1) 安居院猛，長尾智晴：C 言語による画像処理入門，昭晃堂 (2000).
2) Allen, R. and Kennedy, K.: *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*, Morgan Kaufmann Publishers, San Francisco (2001).
3) Analog Devices Inc.: *ADSP-BF561 Blackfin Processor Hardware Reference* (2005).
4) Bik, A.J.C.: *The Software Vectorization Handbook*, Intel Press (2004).
5) Catthoor, F., Danckaert, K., Kulkarni, C., Brockmeyer, E., Kjeldsberg, P.G., Achteren, T.V. and Omnes, T.: *Data Access and Storage Management for Embedded Programmable Processors*, Kluwer Academic Publishers, Boston (2002).
6) Crookes, D., Morrow, P.J. and McParland, P.J.: IAL: A parallel image processing programming language, *Communications, Speech and Vision, IEE Proceedings I*, Vol.137, No.3, pp.176–182 (1990).
7) Gaudiot, J.-L., DeBoni, T., Feo, J., Böhm, W., Najjar, W. and Miller, P.: The Sisal Model of Functional Programming and its Implementation, *Proc. 2nd AIZU International Symposium on Parallel Algorithms/Architectures Synthesis*, pp.112–123 (1997).
8) Gonzalez, R.C., Woods, R.E. and Eddins, S.L.: *Digital Image Processing using MATLAB*, Pearson Education, New Jersey (2004).
9) Hammes, J., Draper, B.A. and Böhm, A.P.W.: Sassy: A Language and Optimizing Compiler for Image Processing on Reconfigurable Computing Systems, *Proc. International Conference on Vision Systems*, pp.83–97 (1999).
10) International Business Machines Corporation: *APL2 Programming: Language Reference*, 2nd Edition, *SH21–1061–01* (1994).
11) Jsoftware. http://www.jsoftware.com/
12) 橘川　孚，宇土正浩，宮崎正俊：初めて学ぶ APL，共立出版 (1994).
13) Scholz, S.-B.: Single Assignment C — Functional Programming Using Imperative Style, *Proc. 6th International Workshop on the Implementation of Functional Languages* (1994).
14) Singh, K.: *Using Cache Memory on Blackfin Processors* (*EE–271*), Analog Devices Inc. (2005).
15) 田村秀行：コンピュータ画像処理，オーム社 (2002).
16) 上坂吉則：MATLAB プログラミング入門，牧野書店 (2000).
17) Veldhuizen, T.L.: Arrays in Blitz++, *Proc. 2nd International Scientific Computing in Object-Oriented Parallel Environments*, pp.223–230, Springer-Verlag (1998).
18) W3C Math Home. http://www.w3.org/Math/
19) Wolfe, M.: *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, Redwood City (1995).
20) レイリシュナー：C++ランゲージクイックリファレンス，オライリー・ジャパン (2004).

(平成 19 年 1 月 22 日受付)
(平成 19 年 5 月 7 日採録)

金井 達徳（正会員）

昭和 59 年京都大学工学部情報工学科卒業．平成元年同大学大学院工学研究科博士後期課程研究指導認定退学．同年（株）東芝入社．以来，高信頼システム，マルチメディアサーバ，データベースシステム，システム LSI 設計支援技術等の研究開発に従事．電子情報通信学会会員．

瀬川 淳一

平成 11 年九州大学大学院システム情報科学研究科知能システム学専攻修士課程修了．同年（株）東芝入社．以来，XML 変換技術，メディア処理フレームワーク，組み込みソフトウェア開発支援技術等の研究開発に従事．

武田奈穂美

昭和 63 年東北大学工学部通信工学科卒業．平成 2 年同大学大学院工学研究科博士課程前期 2 年の課程電気及通信工学専攻修了．同年（株）東芝入社．以来，LSI 設計支援技術，映像配信，画像符号化等の研究開発に従事．