TOSHIBA
Leading Innovation >>>

# CELL REGZA™における マルチコアソフト開発の実際

2010/01/28

境　隆二　高山 征大　加藤 宣弘　島田 智文
(株)東芝 デジタルメディアネットワーク社 コアテクノロジーセンター

Copyright 2010, Toshiba Corporation.

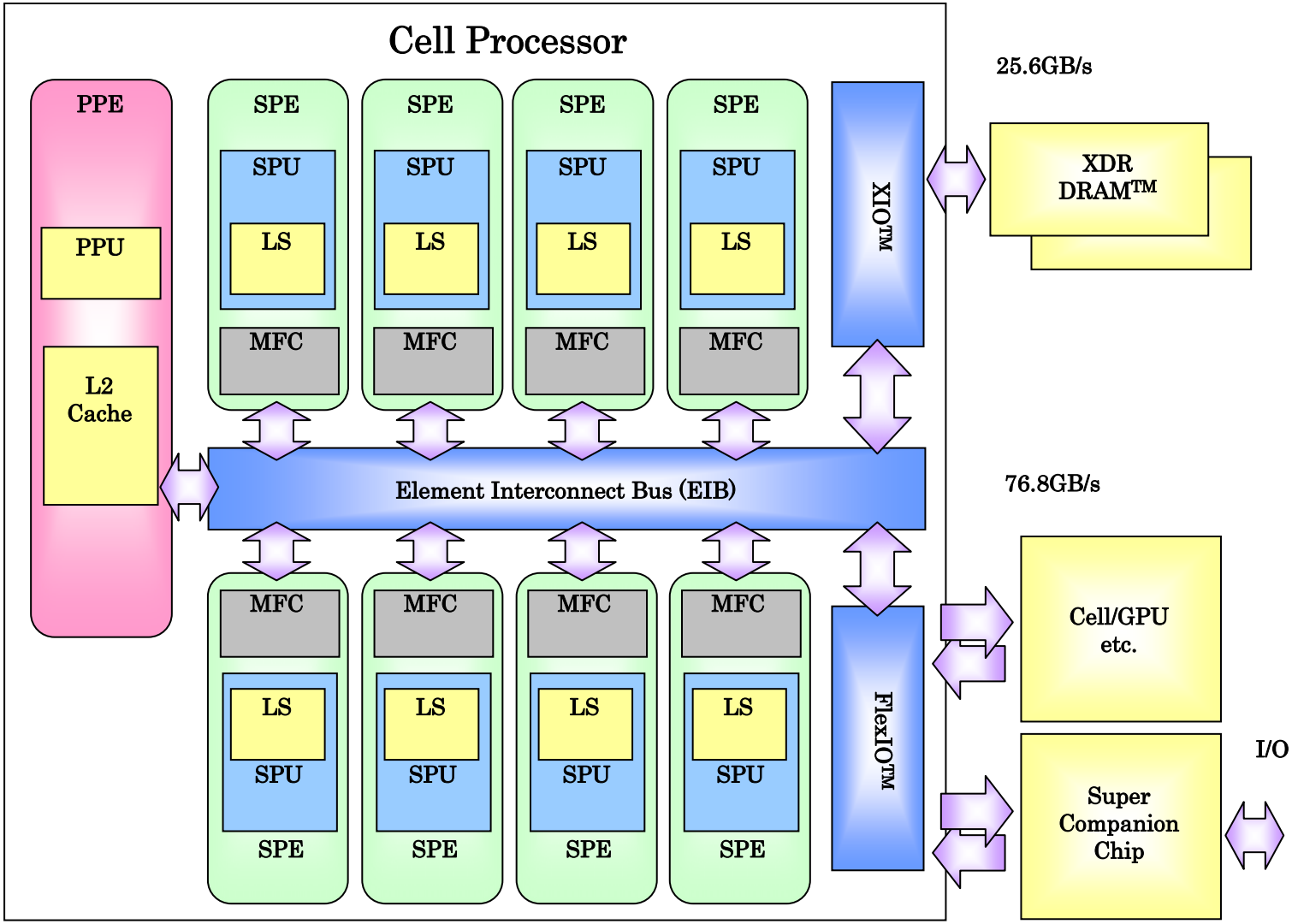
# 目次

- 背景
  - Cell BEアーキテクチャ
  - CELL REGZAのソフトウェア構成
- SPEの性能を引き出すプログラミング
  - 高性能プログラミングとコンパイラの最適化
  - C言語の問題点と対策
- 並列プログラミングモデル Molatomium
  - CELL REGZAにおけるマルチコア並列化
  - Molatomiumの並列動作モデル
  - 開発ツールのデモ
- まとめ

# 並列プログラミングは難しくない

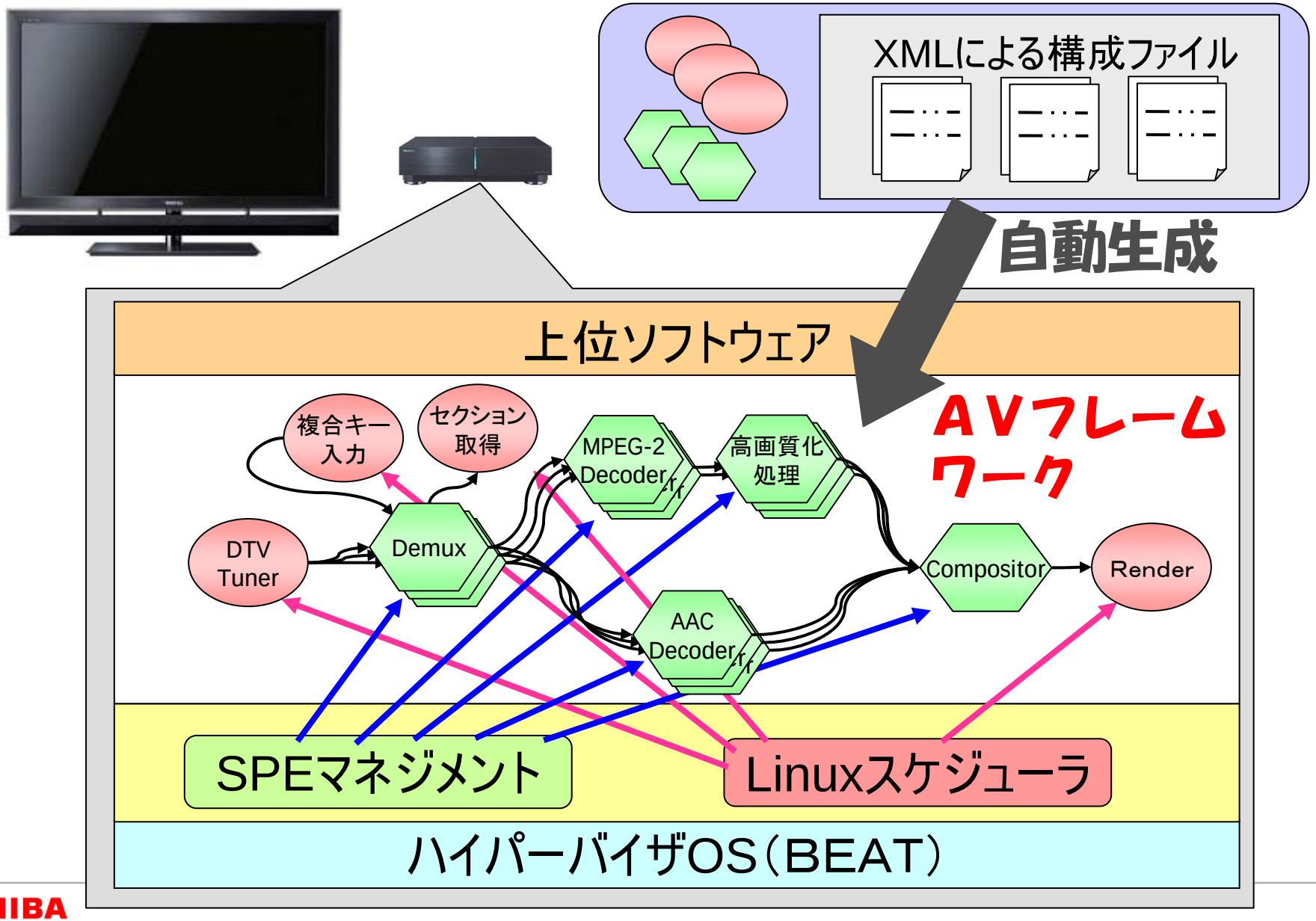
- プロセッサアーキテクチャの進歩
  - RISC　大規模レジスタファイル、局所性（キャッシュ）、コンパイラ最適化
  - パイプライン　命令遅延増大　スループット指向
  - スーパースケーラ、SIMD演算　→　マルチコア
- 命令レベルの並列計算は散々やってきた
  - C言語の問題
    - C言語におけるポインタ解析の限界
    - レジスタスピルのコード品質
  - コードの書き方を工夫して問題回避しコンパイラ最適化を活用
    - ローカル変数の活用とインライン関数
    - コンパイラの最適化技術は進んでいるが限界もある
- 最適化コードは、よく見るとタスクレベル並列性を表現
  - スレッド割り当てや同期を書くのは面倒だしバグが入りやすい
  - 並列性を直接表現できれば　→　Molatomiumの実行モデルと並列言語

# Cell BE™ † アーキテクチャ

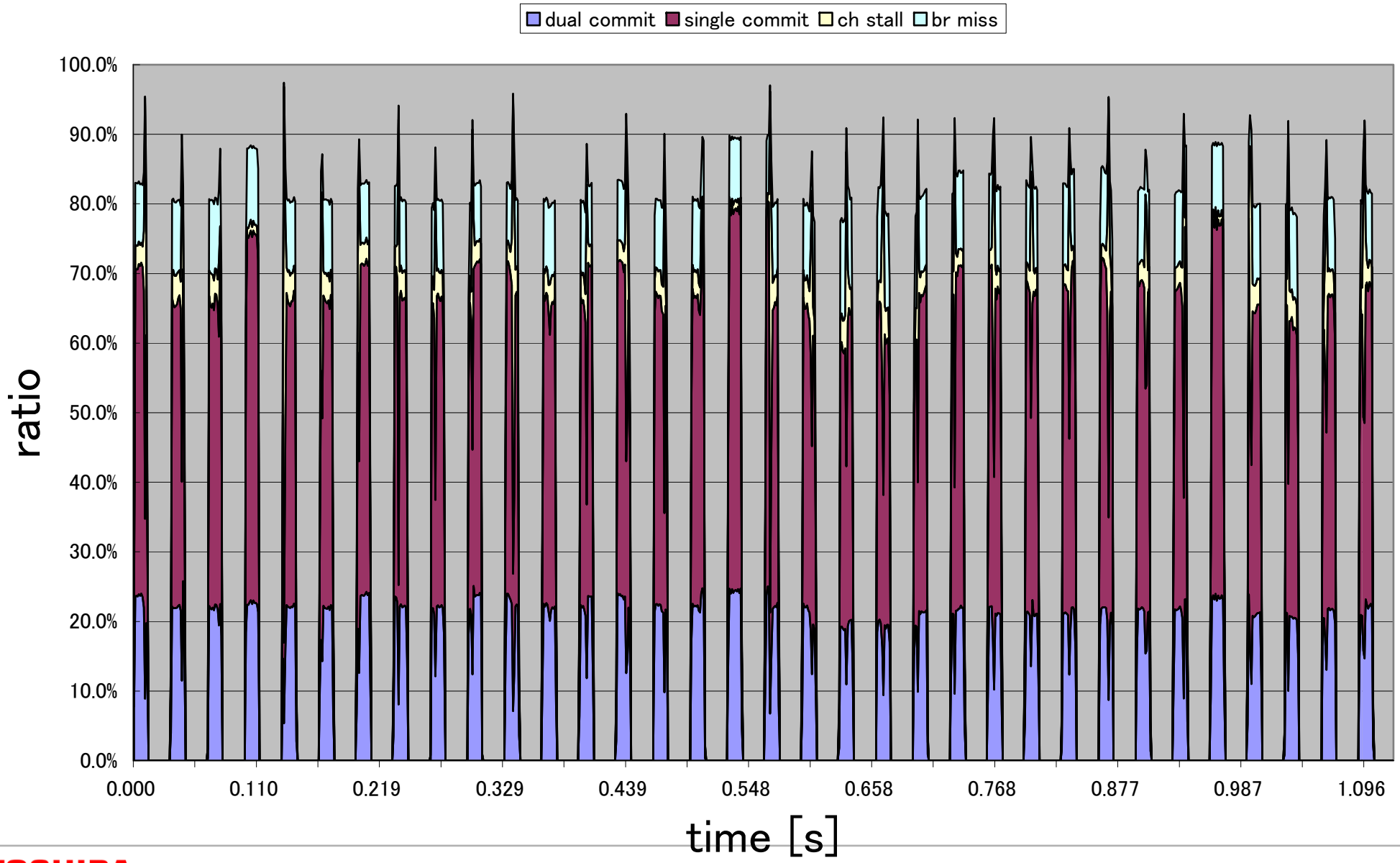



† Cell Broadband Engine™ は、株式会社ソニー・コンピュータエンタテイメントの商標

# CELL REGZAのソフトウェア構成



# MPEG2デコーダの周期実行の様子



# プログラムのSPEへのポーティング

- PCのリファレンスコードと比べて2～4倍遅くなる
  - バイトオペレーションが苦手（SPEは128ビットレジスタSIMDが基本）
  - 分岐のオーバーヘッドが大きい
- SIMD化
  - 4～16倍高速化
- 命令レベル並列化
  - パイプラインの隙間を埋める　～4倍高速化
- DMAと計算のオーバーラップ
  - ～2倍程度（計算量によって高速化率は変化）
- トータルで数十倍程度（アプリケーションによる）
  - IAでも同様の最適化を行えば、ほぼ同じこと

この2つの対策することでIAと同等の性能が得られる

# 命令レベル並列性

## ◆ 高性能ソフトウェア

- アルゴリズムの最適化　＞＞　計算密度
  - 冗長計算を減らすと依存関係が増える
  - SIMD命令によりトータルな命令数は減る

    →　より広い範囲で依存関係のない演算を集めて同時に実行

## ◆ C言語とコンパイラの最適化

- コンパイラの最適化ができること
  - データ依存のない演算の実行順序は入れ替え可能
  - ローカル変数をたくさん使ってメモリアクセス削減
- 階層構造を作って簡潔に記述

## ◆ 同じアルゴリズムでも2～5倍の性能差と1/2～1/5の記述量

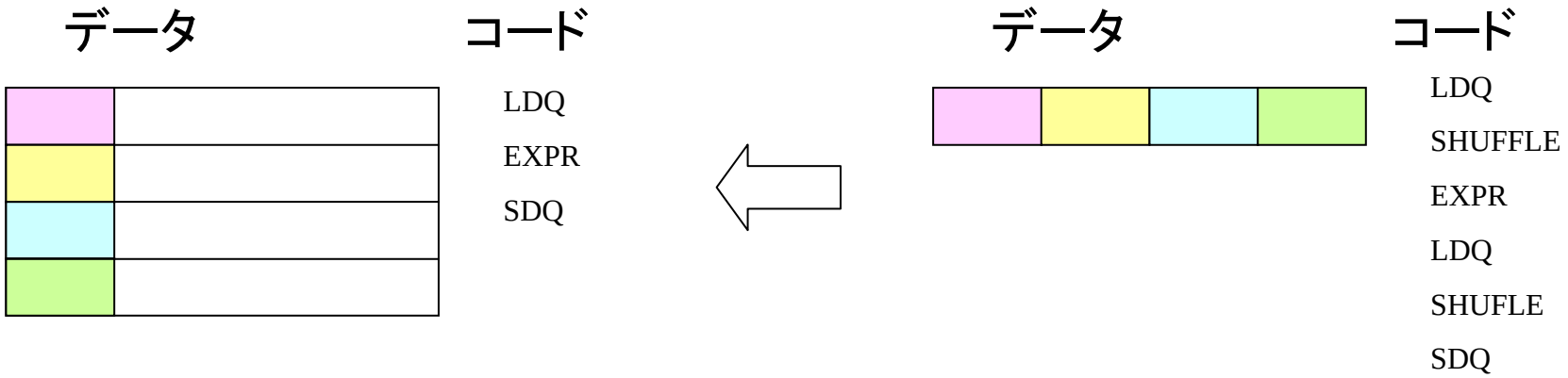
- コンパイラの最適化が有効に働くようにコードを書く　→　高い性能
- 同じコードパターンは関数にする　→　コードサイズ削減

# データアライメントの選択

- 構造体メンバのデータを16バイト境界に配置
- プログラムの入出力以外の中間データのすべてを16バイト境界に配置

  データスペースは増えるが

  アクセスコードを減らすことが可能
- 参照頻度の高い配列データも16バイト境界に配置



# 命令レベル並列処理のためのコーディング

```
// サンプルコード
for (i=0; i < N; i++) {
    C[i] = Func(A[i], B[i], ...);
}
```

Funcは、SIMD演算spu_maddでもよいし、
**数百命令**で構成されるinline関数でもよい

spu-gcc –O3 -funroll-loops …

```
// ループアンロール
for (i=0; i < N; i+=4) {
    C[i+0] = Func(A[i+0], B[i+0], ...);
    C[i+1] = Func(A[i+1], B[i+1], ...);
    C[i+2] = Func(A[i+2], B[i+2], ...);
    C[i+3] = Func(A[i+3], B[i+3], ...);
}
```

# 命令レベル並列処理のためのコーディング

```
// サンプルコード
for (i=0; i < N; i++) {
    C[i] = Func(A[i], B[i], ...);
}
```

spu-gcc –O3 -funroll-loops …

```
// ループアンロール
for (i=0; i < N; i+=4) {
    C[i+0] = Func(A[i+0], B[i+0], ...);
    C[i+1] = Func(A[i+1], B[i+1], ...);
    C[i+2] = Func(A[i+2], B[i+2], ...);
    C[i+3] = Func(A[i+3], B[i+3], ...);
}
```

RAWの依存関係がある
かも知れない

# コンパイラの最適化オプションを活用する

```
Filter (array_t *C, array_t *A, array_t *B) {
    // ループアンロール
    for (i=0; i < N; i+=4) {
        C[i+0] = Func(A[i+0], B[i+0], ...);
        C[i+1] = Func(A[i+1], B[i+1], ...);
        C[i+2] = Func(A[i+2], B[i+2], ...);
        C[i+3] = Func(A[i+3], B[i+3], ...);
    }
}
```

spu-gcc –O3 -fargument-noalias …

実際にはDMAによって配列データを転送するので
うまくいかないケースが多い

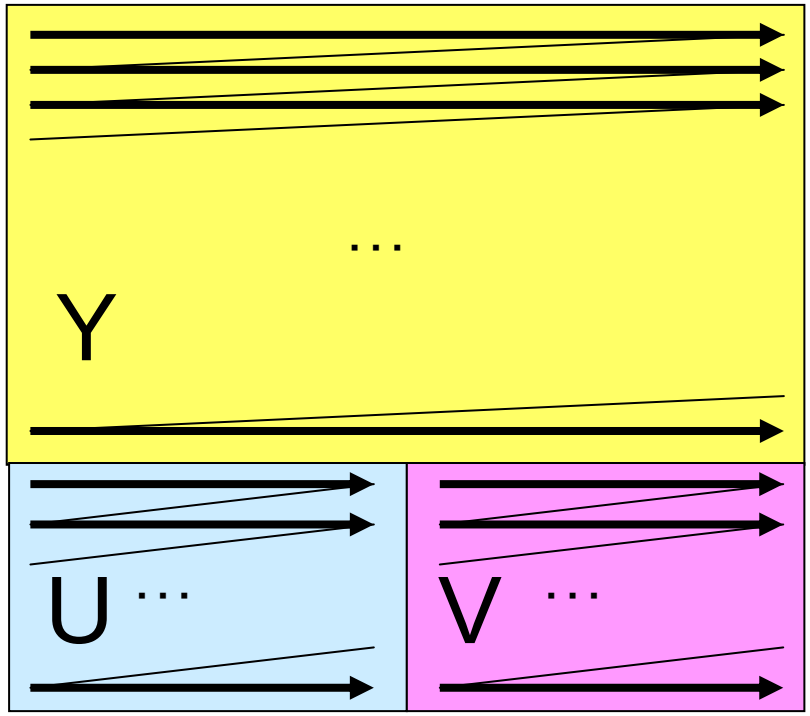
# プログラムコードをコンパクトに保つ

```
// ローカル変数の利用
for (i=0; i < N; i+=4) {
    array_t t[4];
    t[0] = Func(A[i+0], B[i+0], ...);
    t[1] = Func(A[i+1], B[i+1], ...);
    t[2] = Func(A[i+2], B[i+2], ...);
    t[3] = Func(A[i+3], B[i+3], ...);
    C[i+0] = t[0];
    C[i+1] = t[1];
    C[i+2] = t[2];
    C[i+3] = t[3];
}
```

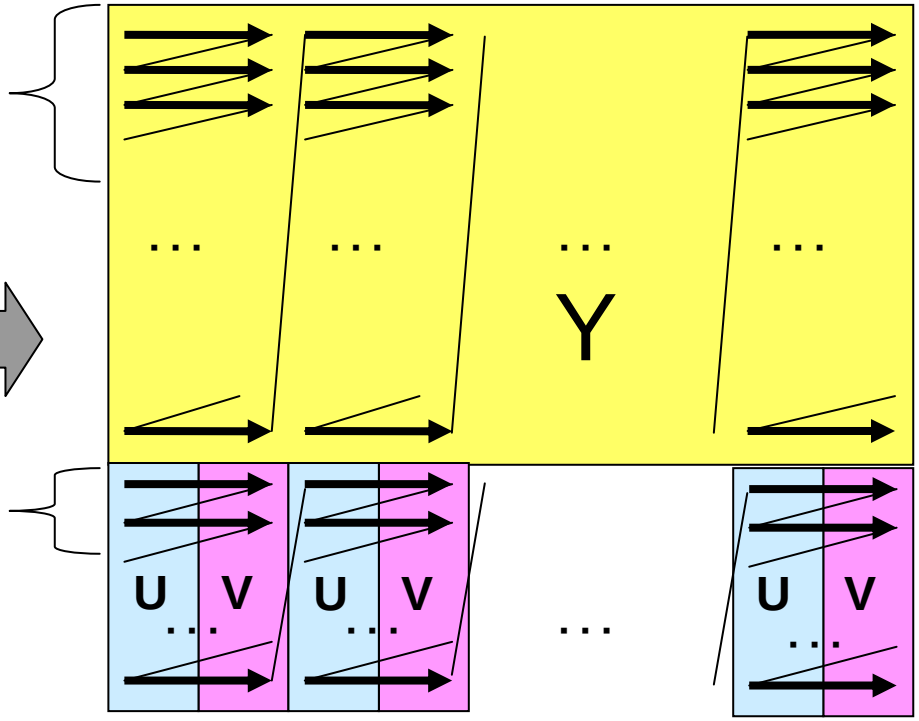
計算効率の向上により、DMAによるメインメモリアクセスのバンド幅が相対的に増大する

# YUVフレームのメインメモリ上のフォーマット

通常のラスタスキャン

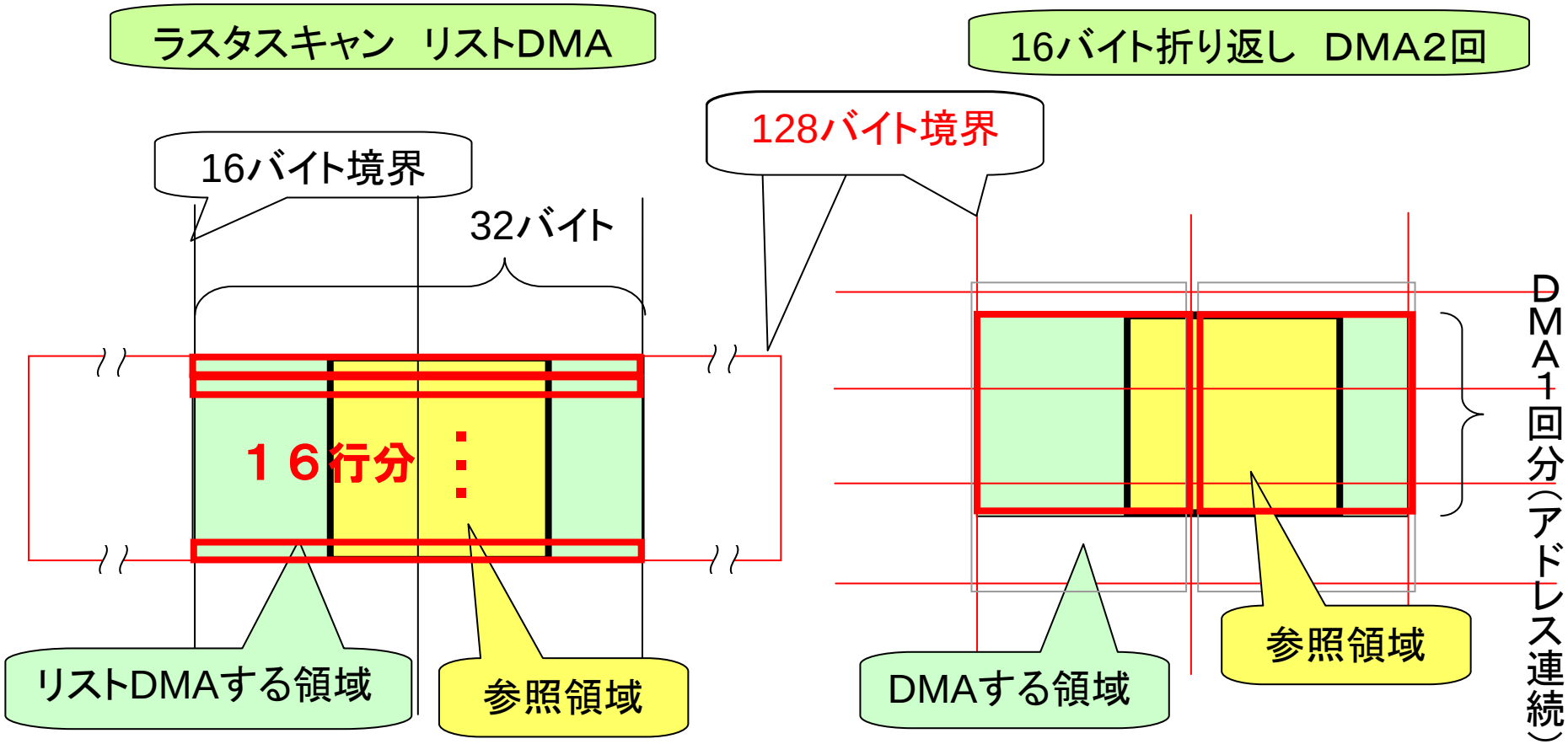



16バイト折り返しラスタスキャン

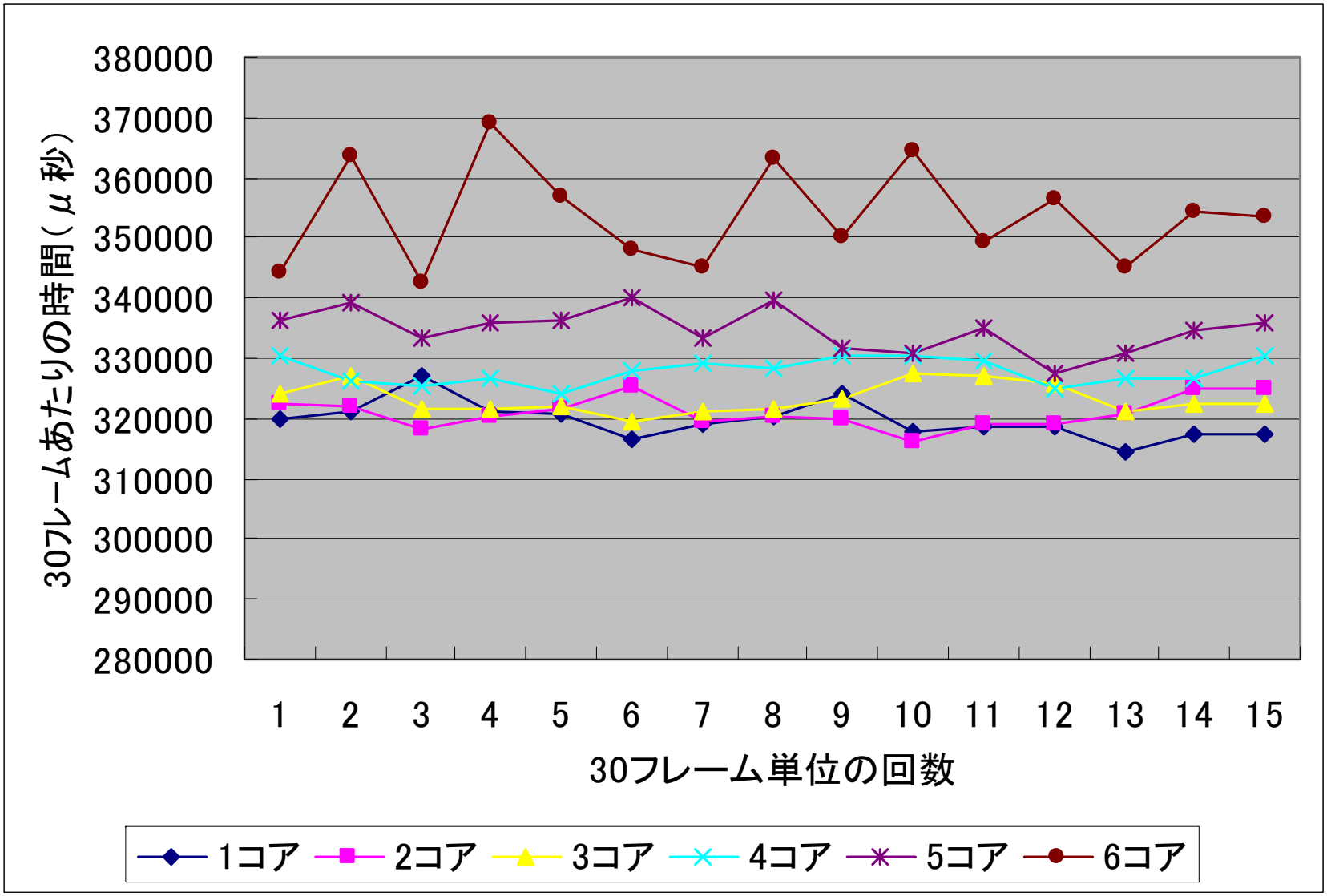



MBパックドより効率がよい

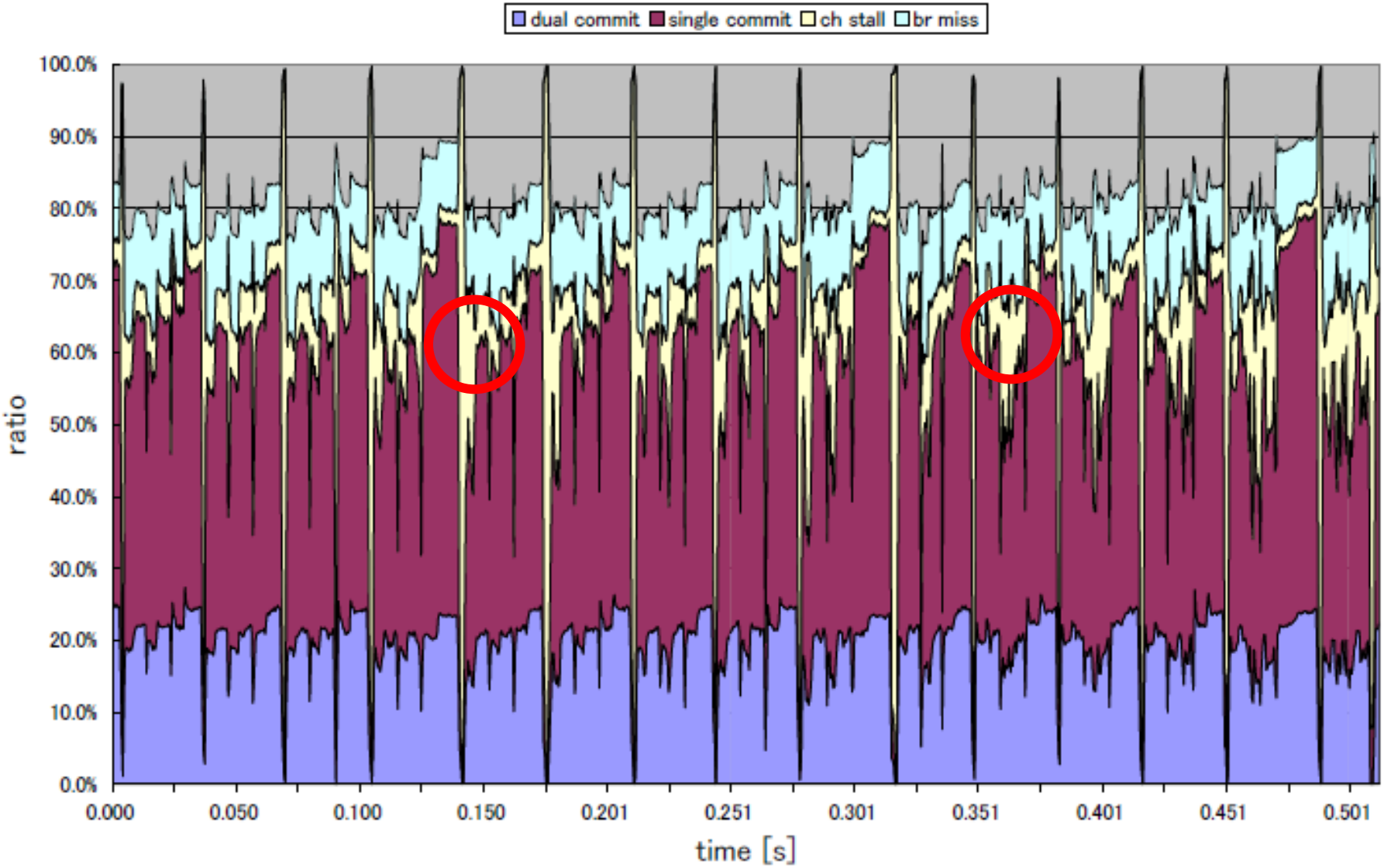
# 動き補償（Motion Compensationのデータ転送）



# 複数のMPEG2デコーダの同時実行時間



# 6SPE同時実行時の命令比率



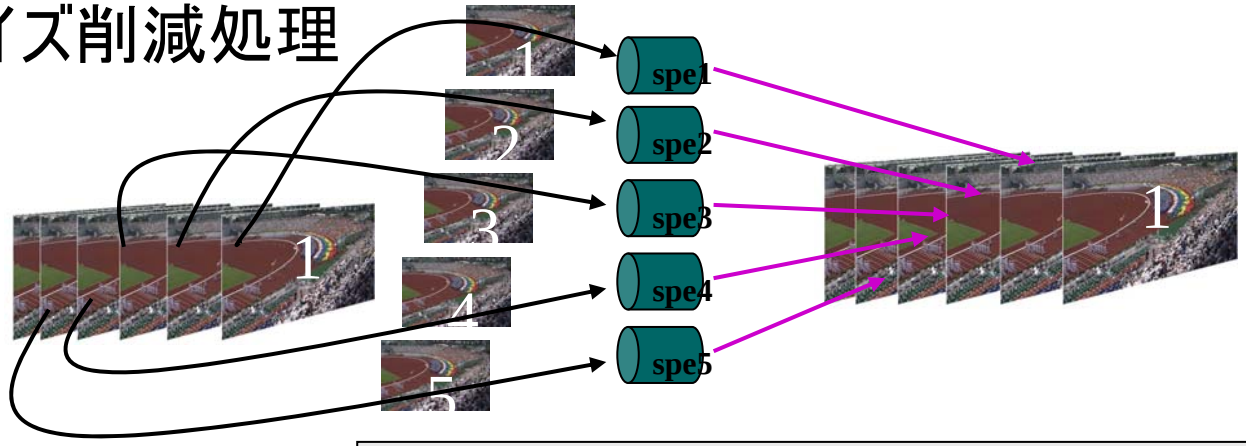
# シングルコア上でのSPEアプリのまとめ

- **命令レベル並列化**
  - コンパイラの最適化が有効に働くコーディングテクニック
  - C言語のポインタ解析の限界
- **DMAとの並列化**
  - 計算処理のブロック化　→　関数にまとめる
  - メインループでDMA処理とオーバーラップ　→　基本形
- **メモリアクセスバンド幅の削減**
  - 同時実行する場合に備えて
    - メインメモリ⇔ローカルストレージの転送量を削減
    - DMAレイテンシ増大に耐える設計　（DMAQライブラリを実装）

**SPEプログラミングでのタスク分割は、移植時点でほぼ完了**

# CELL REGZAにおけるマルチコアソフトの例

ノイズ削減処理




メモリ30Mバイト使用

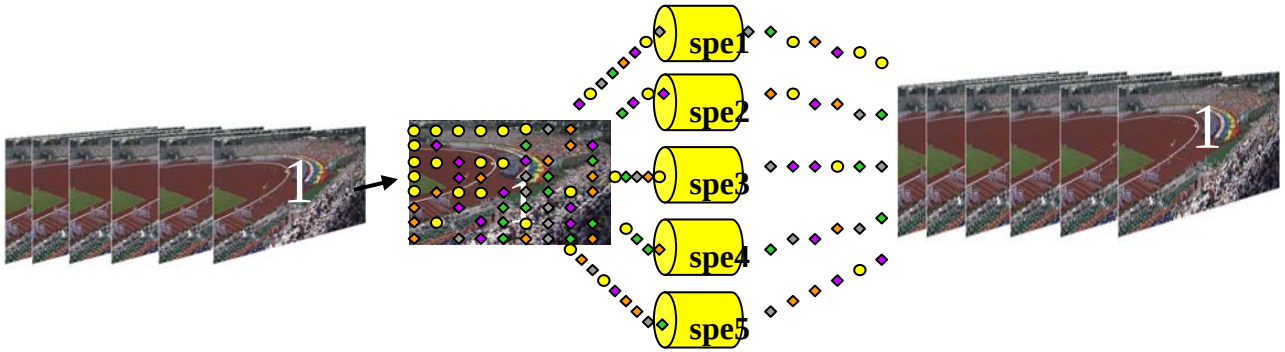
レイテンシ　66ミリ秒

1週間の作業
（テスト含む）

```
//高画質化処理のMolコード一部
for (i in [0..x_sz-1]) {
    ret[i] = run_noise_reduction_x(i, …);
}
```

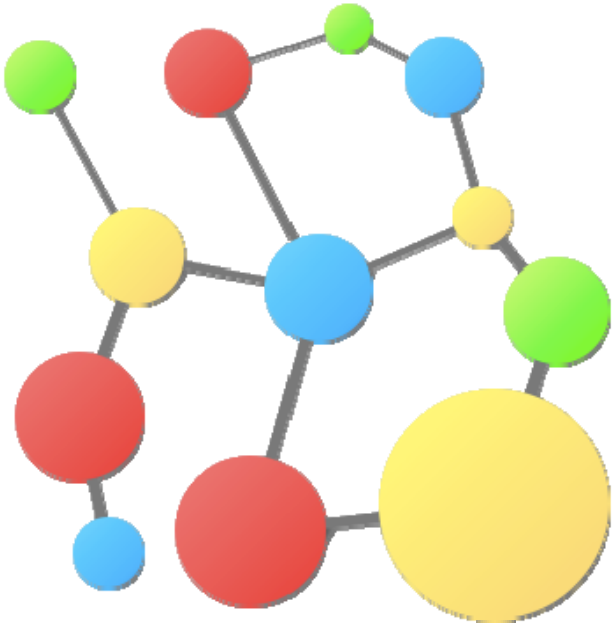
メモリ10Mバイトに削減

レイテンシ　33ミリ秒



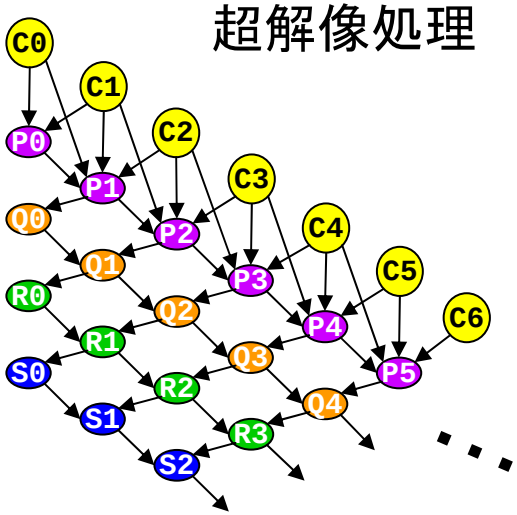
# Molatomiumのコンセプトと実行モデル

- 動機
- コンセプト
- プログラミングモデル
- 並列言語
- 仮想マシン
- 開発ツール

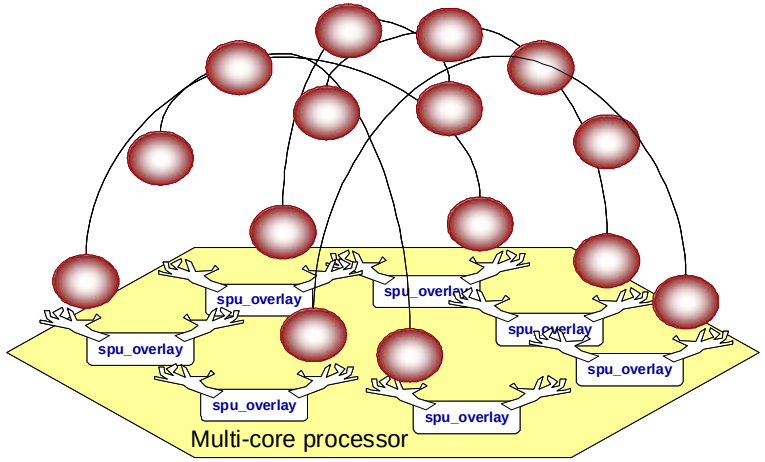


# 動機

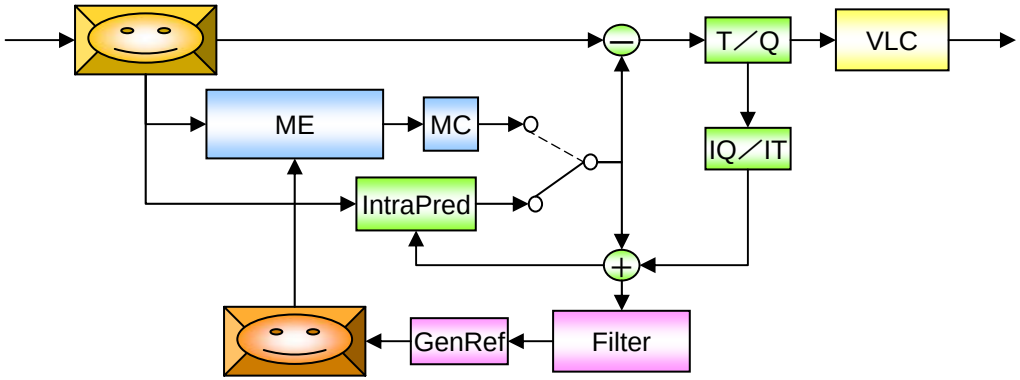
超解像処理




- タスクの依存関係が複雑
- 実行環境でコアに割り当て
-DAG構築のライブラリで
  -できるが煩雑なコード
  -並列アルゴリズムが見えない




ビデオエンコーダ



# 2つの考え方

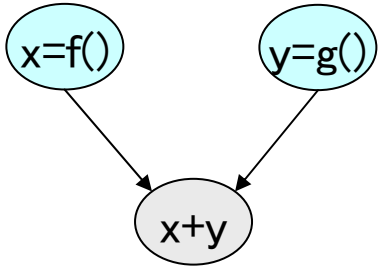
逐次 ⇒ 並列

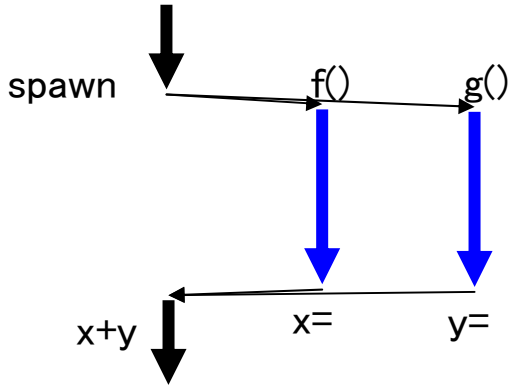
並列 ⊃ 逐次

# ライブラリによるソルーション(逐次 ⇒ 並列)

## Spawn Thread and Synchronization(TBB, Cilk)

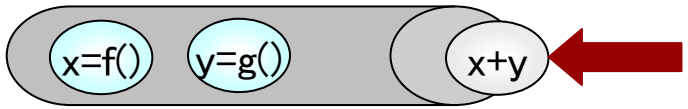
- spawn f()
- spawn g()
- wait f() and g()
- x+y




## Queue and Dispatch(ALF, OpenCL)

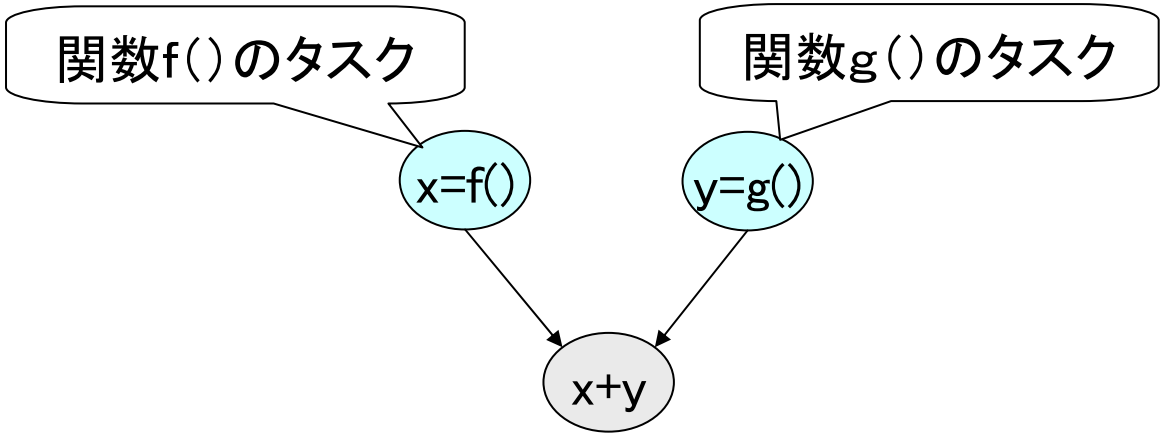
- enqueue f()
- enqueue g()
- enqueue x+y
- depends_on(f, g) or with event_wait_list argument to (x+y)
- finalize



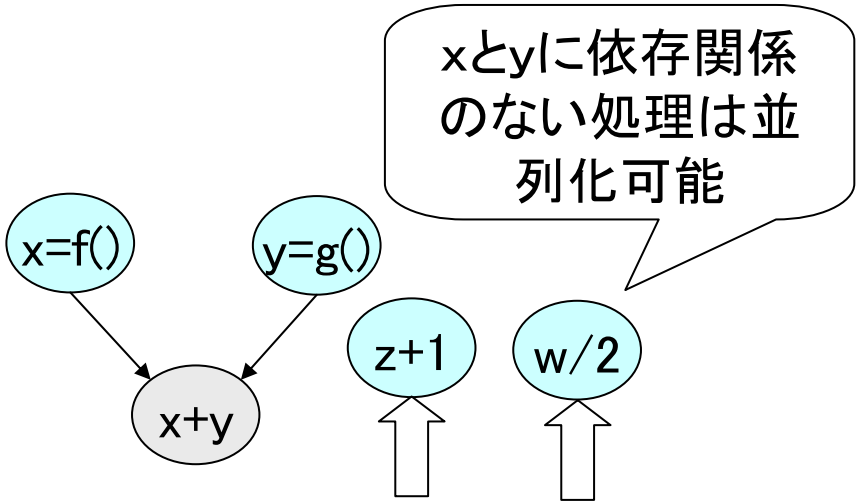

- OpenCL requires event or barrier functions for synchronization

# Molatomiumのコンセプト（並列 ⊃ 逐次）

- 計算
  - DAGで表現できる
  - 直列化する必要なし
  - どのように構造を表現すべきか



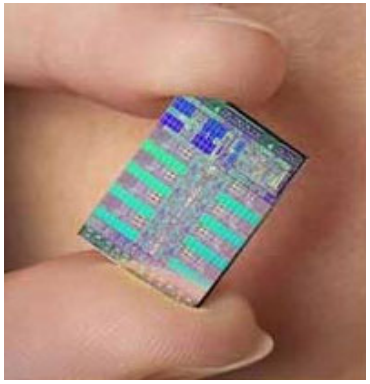

- 構文構造のパワーを使う
  - タスクを命令とみなす
  - 実行順序はRead After Writeの依存関係によって制御される




- 暗黙的なデータ依存を使う
- 並列処理のための明示的なアノテーションは必要ない

# 命令レベル並列性の自然な拡張

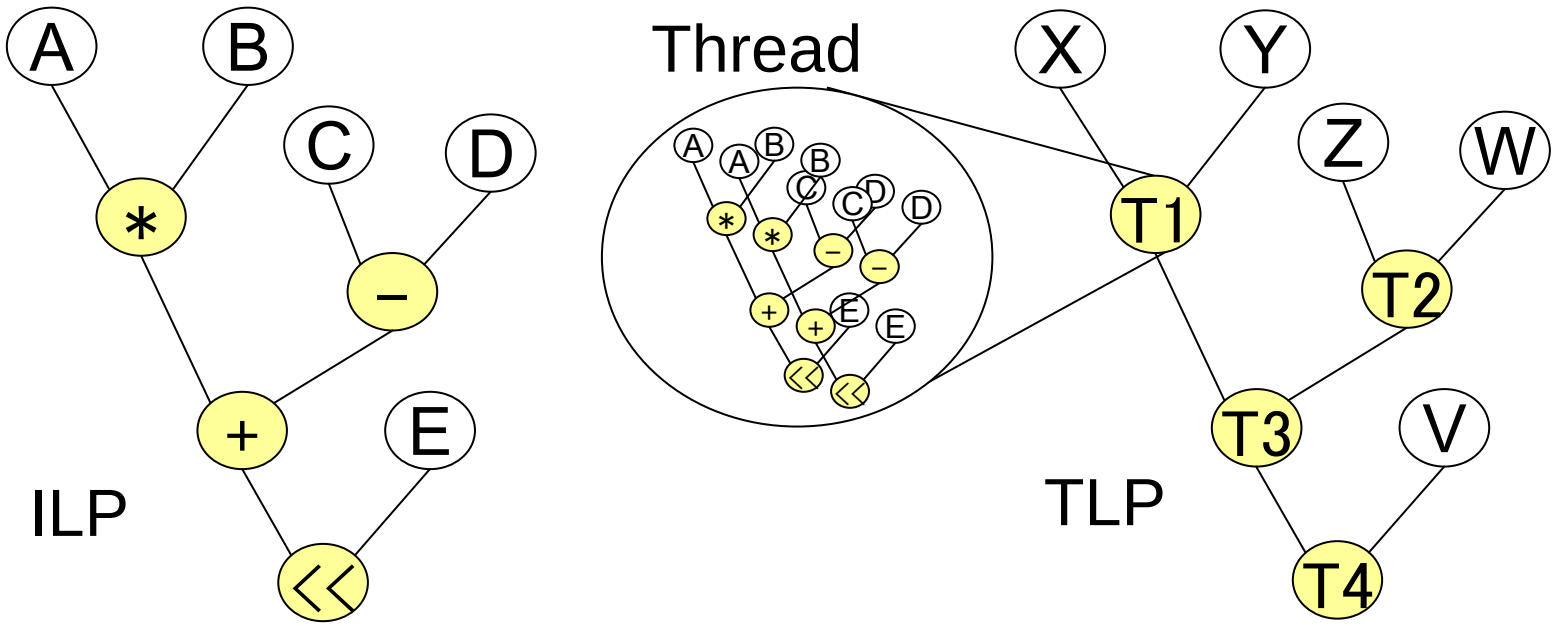
- パフォーマンスプログラマはC／C++言語の命令レベル並列性について慣れ親しんでいる
  - 特にライブラリ呼び出しとして表現するSIMD命令によって訓練されている.
- お互いに依存関係のないインライン関数は、命令レベルで完全に並列実行可能
- Cellプロセッサでのプログラミングの経験から
  - 200命令を超える関数呼び出し
    - コンパイラの最適化で命令が並列化され2倍の性能がえられる



Cell

# ILP（命令レベル並列）モデルをスレッドレベル並列に拡張

- 実行して完了するスレッド（＝タスク）を、1つの仮想命令としてあらわされる仮想マシンを定義
- Molatomiumの実行環境上で仮想命令としてのスレッドは、スーパースケーラプロセッサの命令のように、並列に実行されるprocessor.



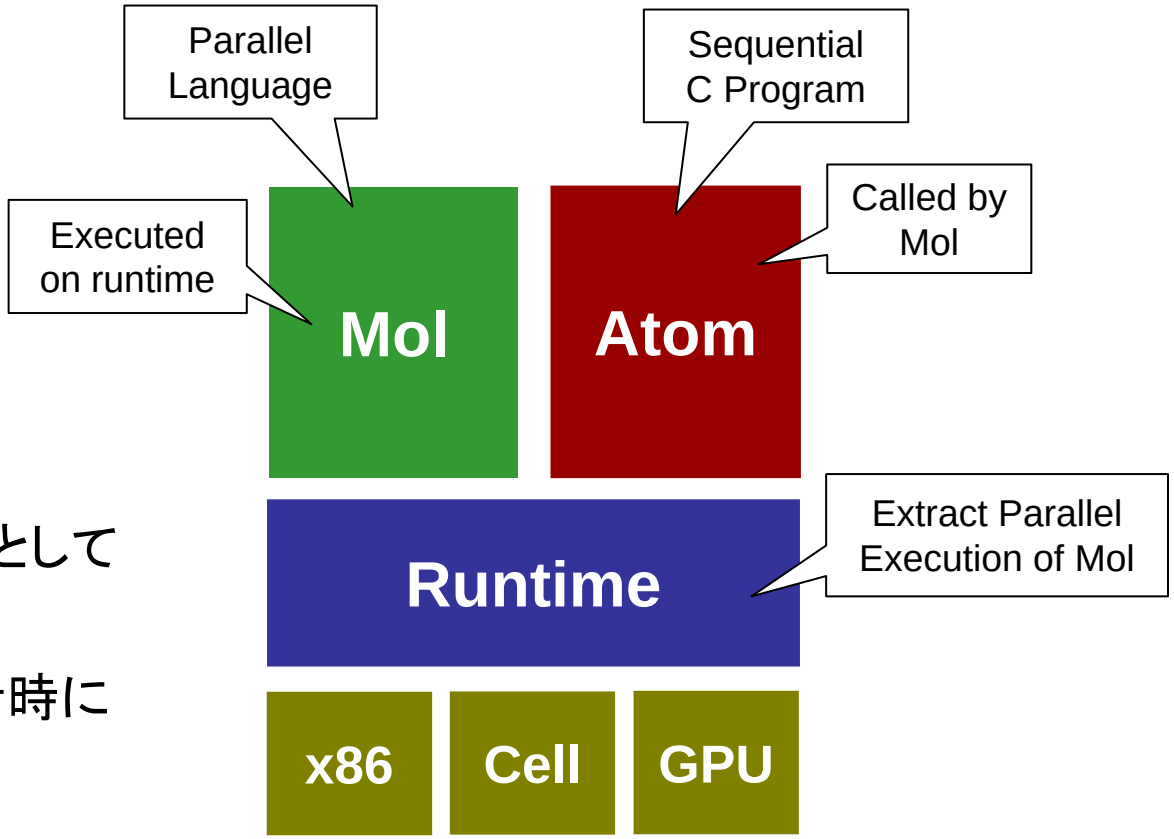
# プログラミングモデル

## 並列処理の実装

- Molatomium = Mol & Atom
  - Atom
  - Mol
  - ランタイム

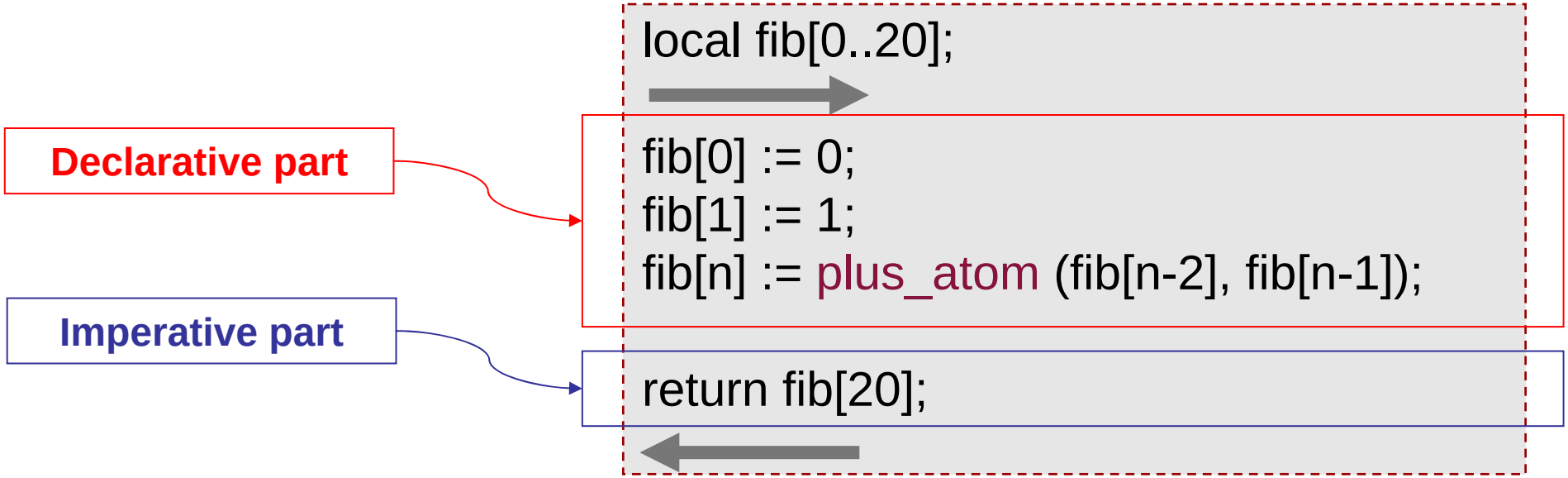
## ランタイム（実行環境）

- 仮想マシン
  - Atomは一つのバイトコードとして仮想マシンに埋め込まれる
  - 並列実行はプログラム実行時にダイナミックに展開される



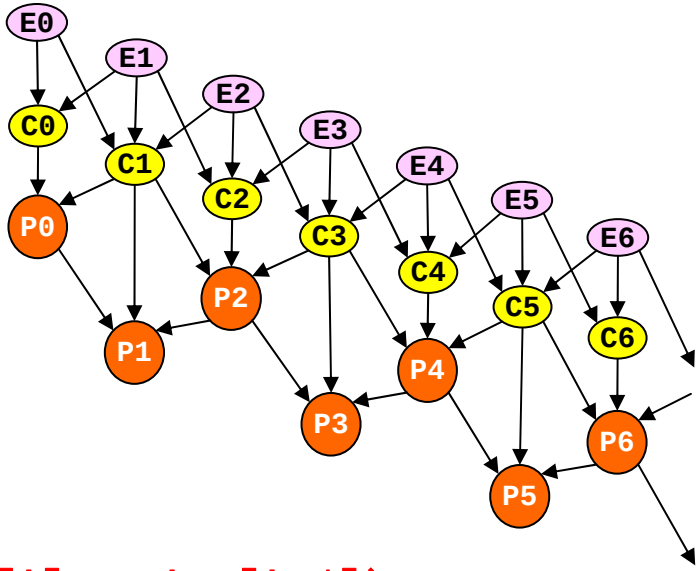
# Mol：ハイブリッド言語（Declarative & Imperative）

- Declarative part
  - いくつかの遅延評価式を宣言
  - それぞれの宣言はスレッド上で実行される
- Imperative part
  - C言語とほぼ同じ
  - 仮想マシンでの実行を通して依存グラフ（DAG）を構築



# 超解像アルゴリズムの例

```
frame(input) {
    local line[0..59];
    local pocs[0..59] outside(0);
    local ccr[0..59] outside(0);
    local edge[0..59] outside(0);
    loop = get_loop();
    edge[i] := run_edge(input[i]);
    ccr[i]  := run_ccr(edge[i-1], edge[i], edge[i+1]);
    pocs[i] := (i%2==0)
        ? run_pocs_even(ccr[i-1], ccr[i], ccr[i+1], loop)
        : run_pocs_odd(pocs[i-1], ccr[i], pocs[i+1], loop);
    for (i in [0..59]) {
        line[i] = pocs[i];
    }
    ...
```

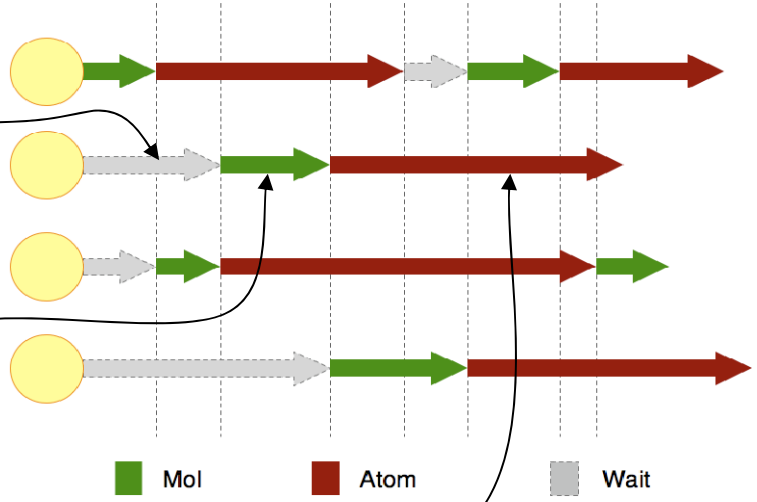


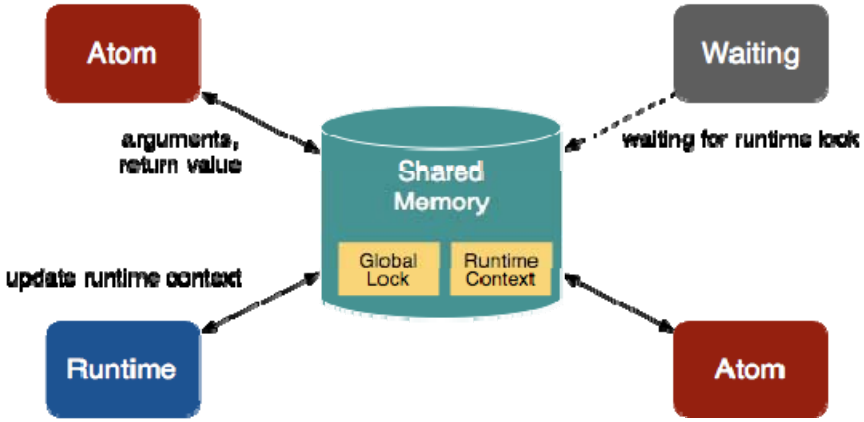
# 仮想マシン

- Mol言語はレジスタベース仮想マシンのバイトコードにコンパイルされる
- それぞれのレジスタは状態をもつ
  - Invalid
  - Calculating
  - Valid
- Load 命令による参照
  - 配列の各要素や変数も状態を持つ
    - Invalidが参照されると対応する宣言的部分の遅延式が呼び出される
      - このとき状態をCalculatingにして依存関係グラフにエントリを作成
    - Calculatingが参照されるとロード命令を依存関係グラフに接続
      - 計算が終わって値がきまると状態をValidにする
    - Validが参照されると、普通のロード命令と同じ動作を行う

# ランタイムの実行モデル

- Runtimeスイッチ
  - コアがロックを取るまで待つ
  - 共有メモリからコンテキストをロード
- Mol（バイトコード）の解釈実行
  - 実行可能なAtomが見つかるまでバイトコードを実行
  - バイトコードのオペランドの一つがReadyでなければ、そのバイトコードを依存グラフにつなぐ
- Atomの実行
  - 共有メモリへランタイムコンテキストを保存
  - ロックを開放しAtomを実行



# 配列エントリと変数のメモ化

- 単一代入
  - 副作用がないことが条件
- 二重に計算するのを防ぐ
  - Memoized by default
- データ並列は、メモ化される配列によって表現
  - それぞれの配列エントリへの参照がAtom実行を起動する（要求駆動）

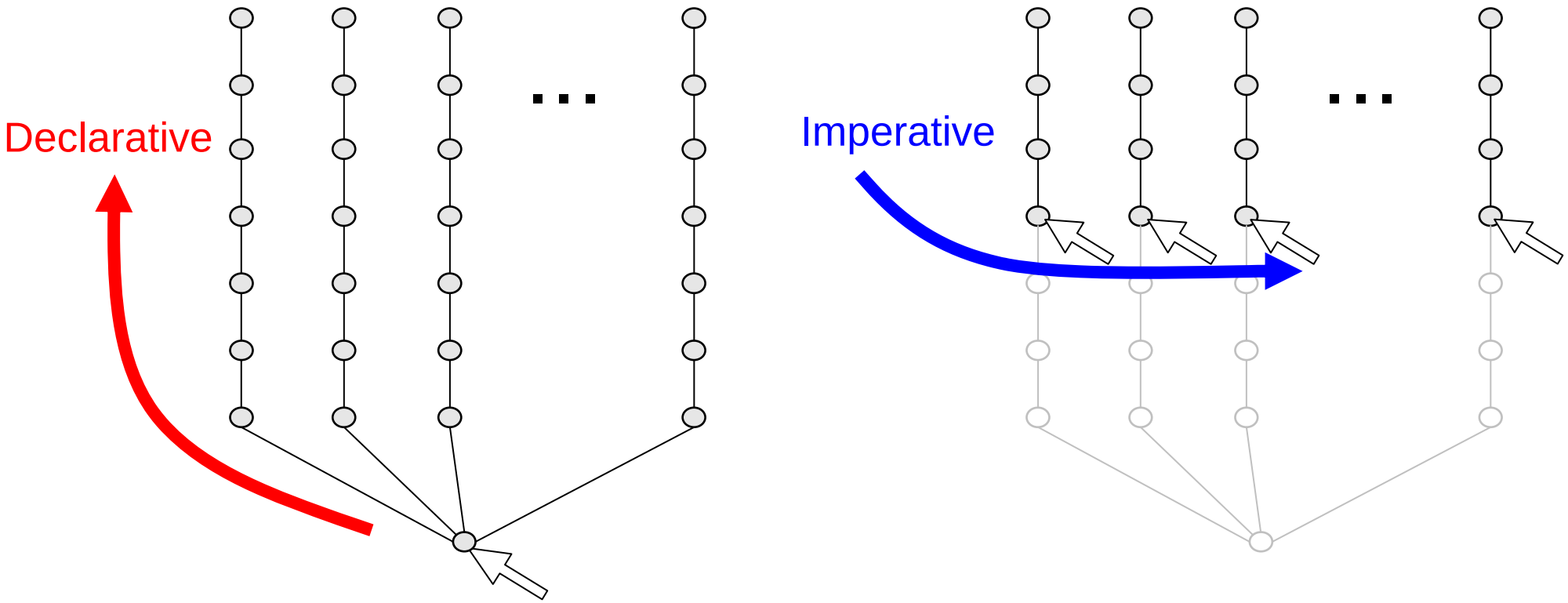
From wikipedia

Memoizaton is an optimization technique used primarily to speed up computer programs by having function calls avoid repeating the calculation of results for previously-processed inputs

# 遅延評価(Lazy evaluation)

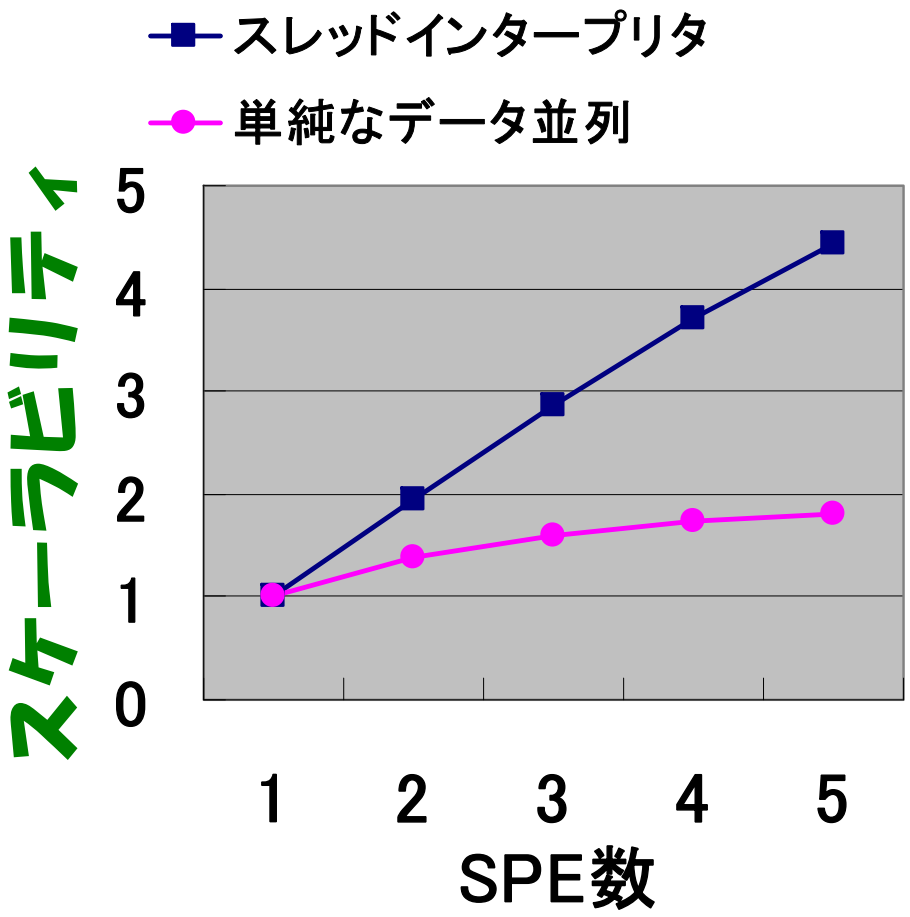
## ◆ 遅延評価式(宣言的部分)

- 左辺へのアクセスから右辺の計算を展開する
- プログラム実行によるDAGの構築を表現
- 命令的部分との組み合わせで局所性を改善

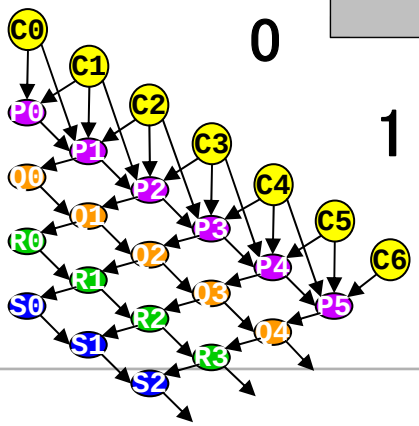


# 超解像度処理の評価

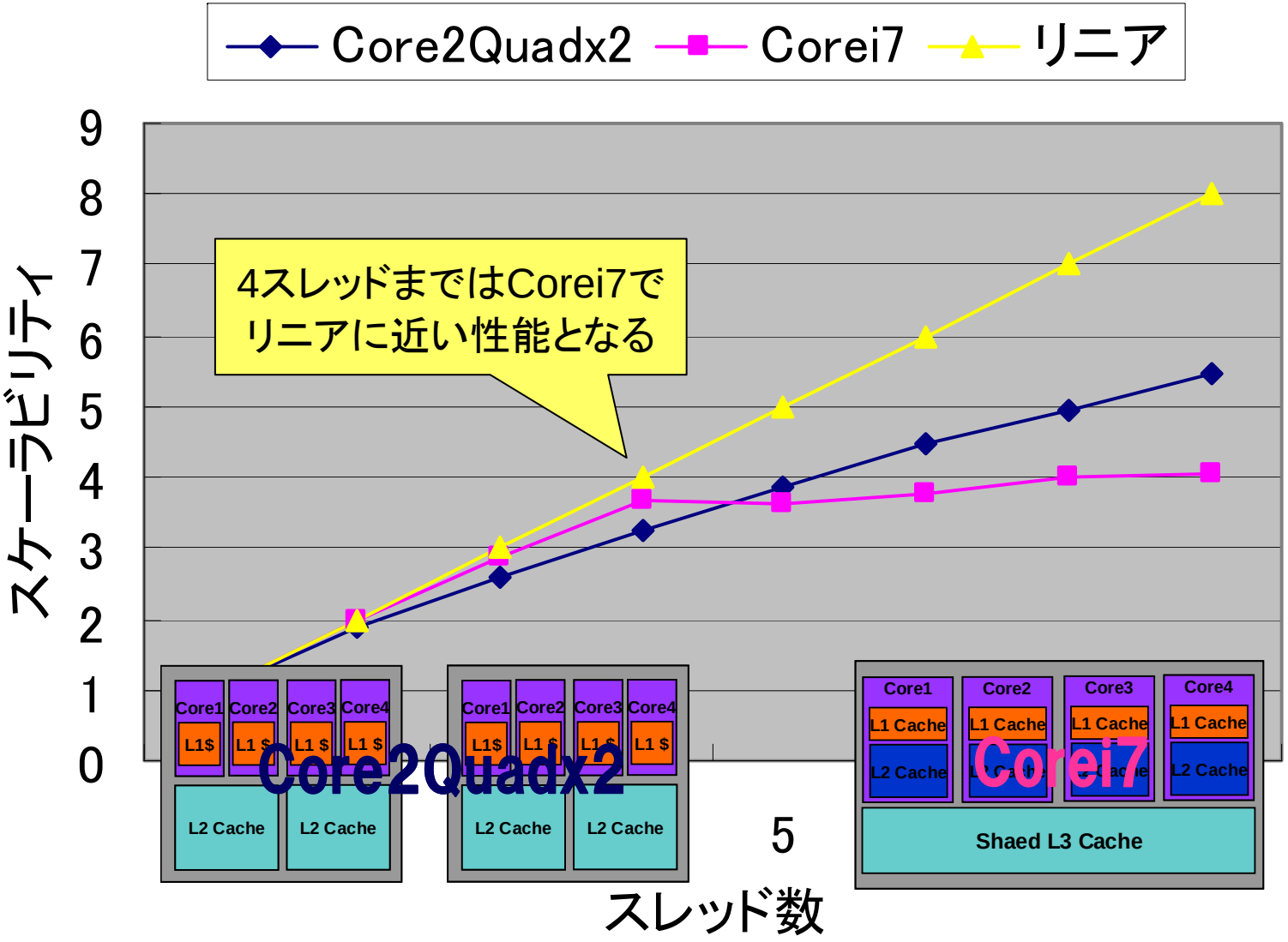
## 性能スケーラビリティ




## ランタイムオーバーヘッド

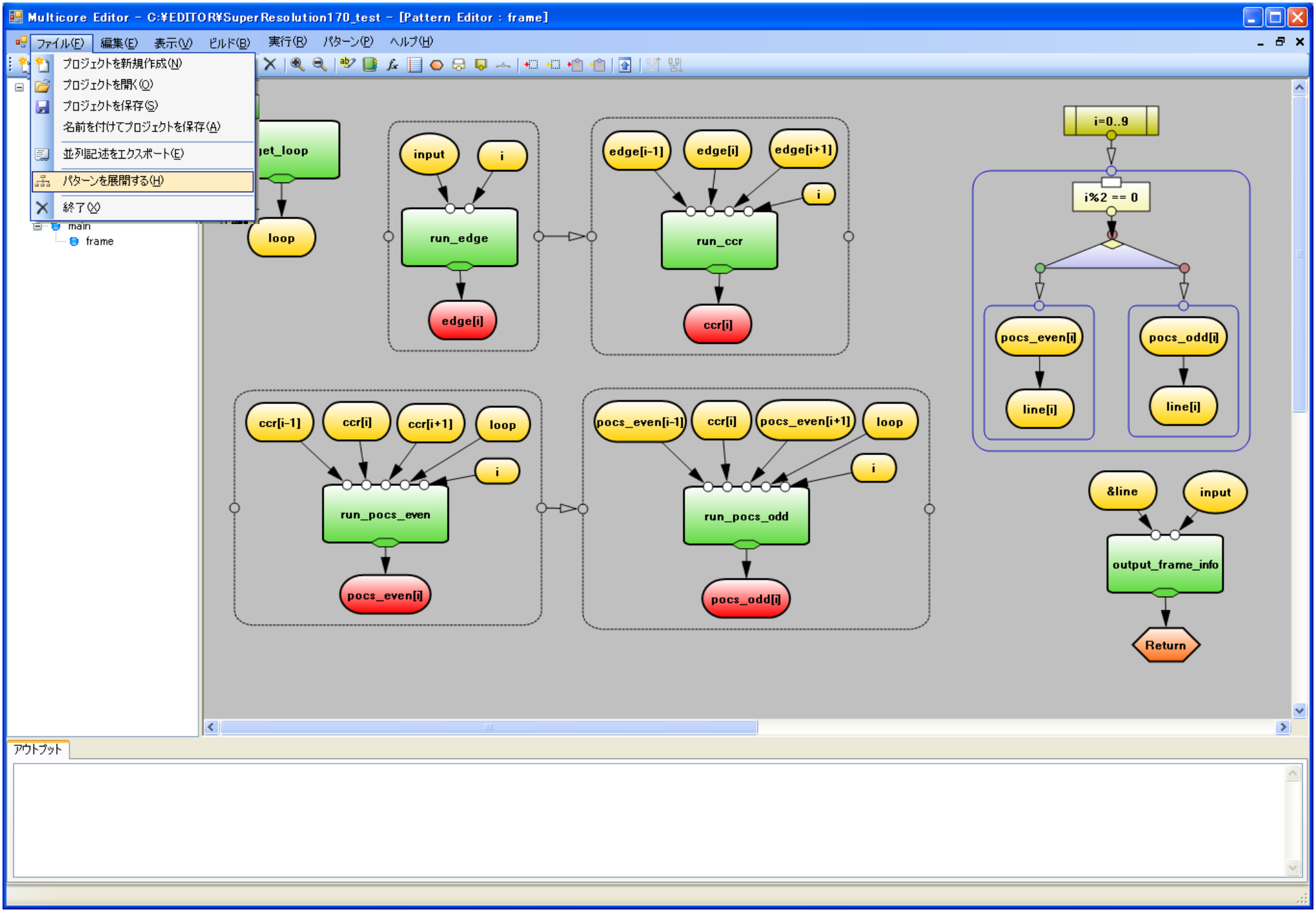


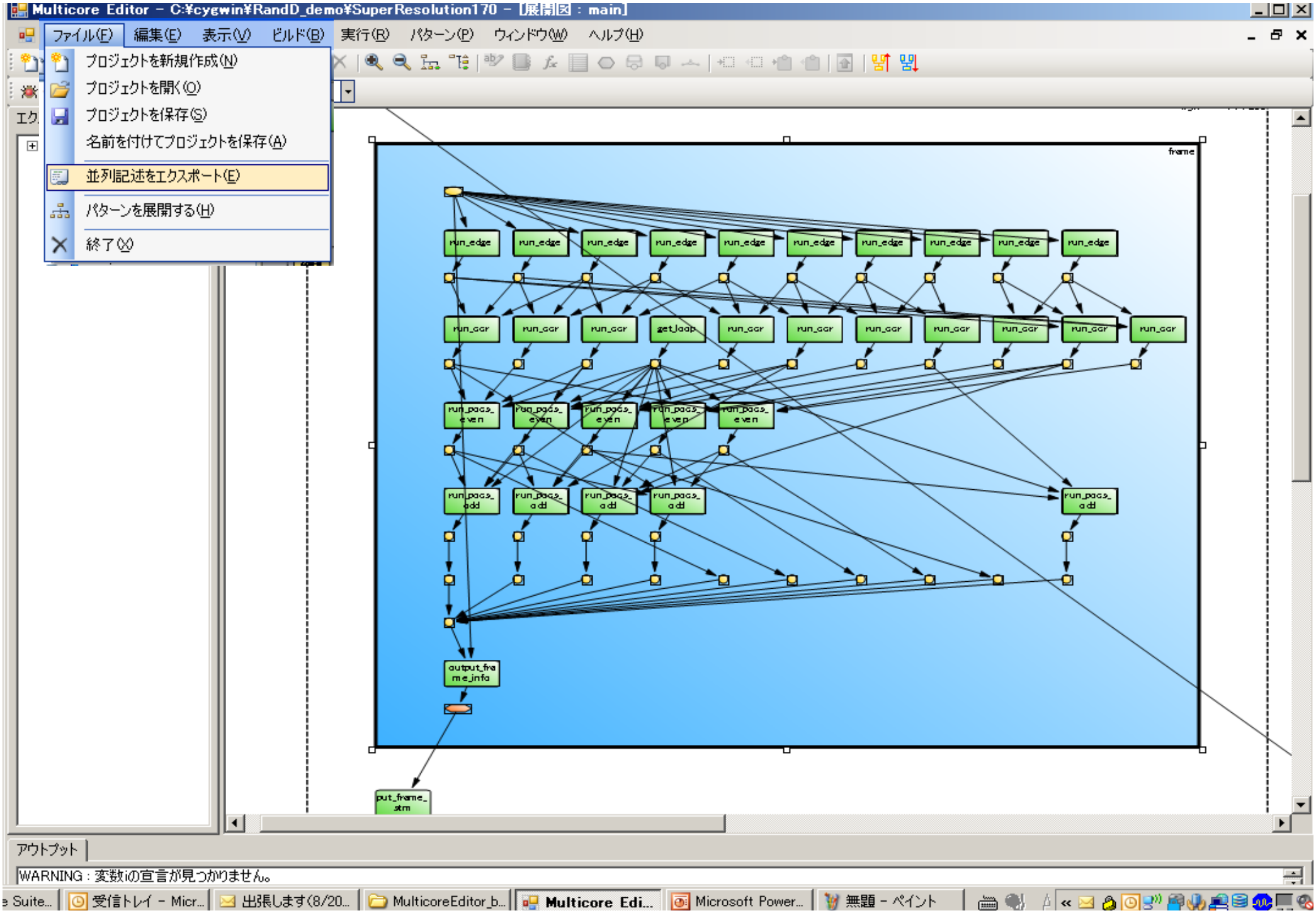
# IAマルチコア上での超解像処理のスケーラビリティ



# ビジュアルプログラミングツール

- Mol言語への導入として
  - Mol言語のコンセプトは？
  - 並列プログラム構成方法を学ぶための速習コース
  - 並列実行の原理を理解しやすい
- より直感的な並列プログラミングの方法
  - 並列実行のスケーラビリティを認識するために


Multicore Editor - C:¥EDITOR¥SuperResolution170_test - [Pattern Editor : frame]
ファイル(F)
編集(E)
表示(V)
ビルド(B)
実行(R)
パターン(P)
ヘルプ(H)
プロジェクトを新規作成(N)
プロジェクトを開く(O)
プロジェクトを保存(S)
名前を付けてプロジェクトを保存(A)
並列記述をエクスポート(E)
パターンを展開する(H)
終了(X)
main
frame
get_loop
loop
input
i
run_edge
edge[i]
edge[i-1]
edge[i]
edge[i+1]
i
run_ccr
ccr[i]
ccr[i-1]
ccr[i]
ccr[i+1]
loop
i
run_pocs_even
pocs_even[i]
pocs_even[i-1]
ccr[i]
pocs_even[i+1]
loop
i
run_pocs_odd
pocs_odd[i]
i=0..9
i%2 == 0
pocs_even[i]
line[i]
pocs_odd[i]
line[i]
&line
input
output_frame_info
Return
アウトプット


Multicore Editor - C:¥cygwin¥RandD_demo¥SuperResolution170 - [展開図 : main]
ファイル(F)
編集(E)
表示(V)
ビルド(B)
実行(R)
パターン(P)
ウィンドウ(W)
ヘルプ(H)
プロジェクトを新規作成(N)
プロジェクトを開く(O)
プロジェクトを保存(S)
名前を付けてプロジェクトを保存(A)
並列記述をエクスポート(E)
パターンを展開する(H)
終了(X)
frame
run_edge
run_car
get_loop
run_pacs_even
run_pacs_odd
output_frame_info
put_frame_stm
アウトプット
WARNING : 変数iの宣言が見つかりません。
Multicore Edi...
Microsoft Power...
無題 - ペイント

Multicore Editor - C:¥EDITOR¥SuperResolution170_test - [STMコードビュー : C:¥EDITOR¥SuperResolution170_test¥SuperResolution170.stm]

ファイル(F) 編集(E) 表示(V) ビルド(B) 実行(R) パターン(P) ヘルプ(H)

- SuperResolution170
  - get_loop
  - run_edge
  - run_ccr
  - run_pocs_even
  - run_pocs_odd
  - get_frame_stm
  - put_frame_stm
  - output_frame_info
  - main
    - frame

```
7   extern put_frame_stm();
8   extern output_frame_info();
9
10  main()
11  {
12      seq{
13          for(i in [0..])
14          {
15              local fr_in;
16
17              fr_in = get_frame_stm(i);
18              if(fr_in == -1)
19              {
20                  break;
21              }
22              put_frame_stm(frame(fr_in));
23          }
24      }
25
26      frame(input)
27      {
28          local line[0..9];
29          local pocs_even[0..9];
30          local pocs_odd[0..9];
31          local ccr[0..9];
32          local edge[0..9];
33          local loop;
34
35          pocs_even[i] := run_pocs_even(ccr[i-1],ccr[i],ccr[i+1],loop,i);
36          pocs_odd[i] := run_pocs_odd(pocs_even[i-1],ccr[i],pocs_even[i+1],loop,i);
37          edge[i] := run_edge(input,i);
38          ccr[i] := run_ccr(edge[i-1],edge[i],edge[i+1],i);
39
40          loop = get_loop();
41          for(i in [0..9])
42          {
43              if(i%2 == 0)
44              {
45                  line[i] = pocs_even[i];
46              }
47              else
```

アウトプット

```
WARNING : 変数iの宣言が見つかりません。
WARNING : 変数edge[10]の宣言が見つかりません。
WARNING : 変数iの宣言が見つかりません。
WARNING : 変数iの宣言が見つかりません。
WARNING : 変数iの宣言が見つかりません。
WARNING : 変数pocs_even[10]の宣言が見つかりません。
WARNING : 変数iの宣言が見つかりません。
WARNING : 変数&lineの宣言が見つかりません。
```

# まとめ

- CELL REGZAを実現
  - 多くのSPEプログラムを実装（主に画像処理）
  - SPEのSIMDだけでなくPPEのAltivecもやっている
  - 命令レベル並列　→　タスクレベル並列　（自然な拡張へ）
- 並列環境「Molatomium」
  - 実用的な並列実行モデルと実行環境
  - Mol言語による並列性の記述
  - Atom（C関数）を一つのバイトコードとして埋め込んだ仮想マシン
- 画像処理以外は？
  - やはり画像処理（デジタルメディアネットワーク社に所属）
  - よりきれいな画像を求めて処理負荷は増大しつづける
  - 短期間で十分な性能を実現できる環境が必要

TOSHIBA

Leading Innovation >>>