ステンシル計算言語 Halide

村主崇行
special thanks: 似鳥啓吾

計算科学研究機構

July 9, 2014

Section 1

はじめに

ステンシル計算とは？

配列変数を更新していくタイプのアルゴリズムで、配列の各要素を、それぞれ近傍の要素の値のみに基づき、同じルールで更新するものをいう。

ステンシル計算とは？

配列変数を更新していくタイプのアルゴリズムで、配列の各要素を、それぞれ近傍の要素の値のみに基づき、同じルールで更新するものをいう。
要するに

ステンシル計算とは？

配列変数を更新していくタイプのアルゴリズムで、配列の各要素を、それぞれ近傍の要素の値のみに基づき、同じルールで更新するものをいう。
要するに
配列ベースの流体計算、MHD、等々

Halide とは？

Halide はステンシル計算プログラムの生成とチューニングのためのライブラリ。画像処理のために開発された。
[Ragan-Kelley et al., 2012, Ragan-Kelley et al., 2013]

	Paraiso	Halide
基本言語	Haskell	C++
コード生成対象	x86, CUDA	x86/SSE, ARM, Native Client, OpenCL, CUDA, ...
扱える次元	n 次元	n 次元
最適化の種類	ループ融合、同期	並列度、局所性、計算節約 の間のトレードオフ
自動チューニング	あり	なし

Section 2

Halide の文法

Halide の文法

プログラミング言語 Halide は、 C++のライブラリとして実装されている。
したがって Halide でプログラムするということは C++のプログラムを書くことになる。
基本的な型として、n 次元配列を表す `Func` と、配列添字変数を表す `Var` があり、これらを組み合わせてステンシル計算を記述する。

Halideの文法

プログラミング言語 Halide は、 C++のライブラリとして実装されている。
したがって Halide でプログラムするということは C++のプログラムを書くことになる。
基本的な型として、n 次元配列を表す `Func` と、配列添字変数を表す `Var` があり、これらを組み合わせてステンシル計算を記述する。

```
Func input = initial_condition.realize(NX,NY);
Var i,j;
Func cell2 = (input(i-1,j)+input(i,j)+input(i+1,j))/3;
Func output = (input(i,j-1)+input(i,j)+input(i,j+1))/3;
```

Halideの文法

プログラミング言語 Halide は、 C++のライブラリとして実装されている。
したがって Halide でプログラムするということは C++のプログラムを書くことになる。
基本的な型として、n 次元配列を表す `Func` と、配列添字変数を表す `Var` があり、これらを組み合わせてステンシル計算を記述する。

```
Func input = initial_condition.realize(NX,NY);
Var i,j;
Func cell2 = (input(i-1,j)+input(i,j)+input(i+1,j))/3;
Func output = (input(i,j-1)+input(i,j)+input(i,j+1))/3;
```

以上のような構文を実現するため、C++の演算子オーバーロードを多用している。例えば、複数引数関数のように見えるものは括弧演算子 `operator()` をオーバーロードすることで実現している。

Halide のプログラムの例

Halide ではステンシル計算をとても直感的な形で記述することができる。

Halideのプログラムの例

Halideではステンシル計算をとても直感的な形で記述することができる。
たとえば、以下のような拡散方程式の離散化解法は

$$C_{\mathrm{in}} = \text{initial condition} \tag{1}$$

$$C_2[i,j] = \frac{1}{3}\left(C_{\mathrm{in}}[i-1,j] + C_{\mathrm{in}}[i,j] + C_{\mathrm{in}}[i+1,j]\right) \tag{2}$$

$$C_{\mathrm{out}}[i,j] = \frac{1}{3}\left(C_2[i,j-1] + C_2[i,j] + C_2[i,j+1]\right) \tag{3}$$

Halideのプログラムの例

Halideではステンシル計算をとても直感的な形で記述することができる。
たとえば、以下のような拡散方程式の離散化解法は

$$C_{\mathrm{in}} = \text{initial condition} \tag{1}$$

$$C_2[i,j] = \frac{1}{3}\left(C_{\mathrm{in}}[i-1,j] + C_{\mathrm{in}}[i,j] + C_{\mathrm{in}}[i+1,j]\right) \tag{2}$$

$$C_{\mathrm{out}}[i,j] = \frac{1}{3}\left(C_2[i,j-1] + C_2[i,j] + C_2[i,j+1]\right) \tag{3}$$

Halideではこのように記述できる。

```
Var i,j;
Func input = initial_condition.realize(NX,NY);
Func cell2 = (input(i-1,j)+input(i,j)+input(i+1,j))/3;
Func output = (input(i,j-1)+input(i,j)+input(i,j+1))/3;
```

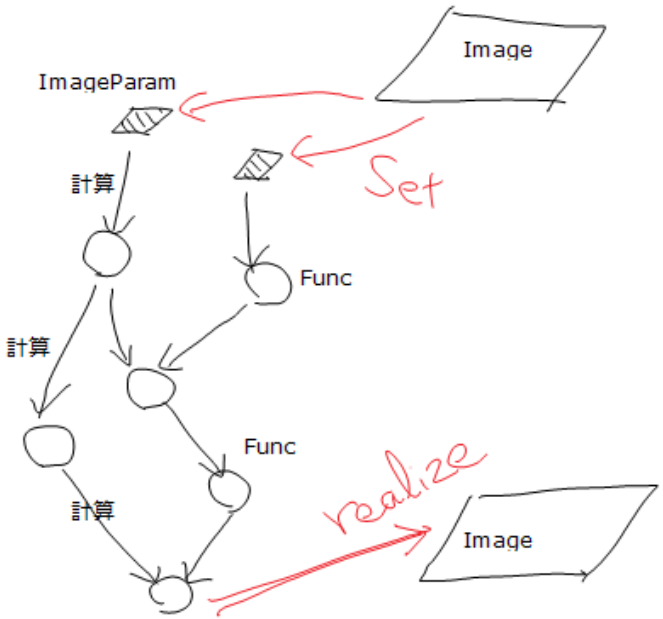
Halideのコード生成と実行

Halideで配列変数としてもっとも頻繁に登場するFunc型は、「その配列を計算するための手続き」を保持しているにすぎない。これに対しImage型は実際にメモリ上に確保された多次元配列である。
realize()関数を呼び出してFunc型をImage型に変換したとき、コード生成・最適化・実際の計算が行われる。(プログラムに変更がなければ、いったん生成されたコードは使いまわされる)

```
inPar.set(input);
output=cell3.realize(NX,NY);
```

Halide におけるステンシル計算の実装

- Image:実際にメモリ上に確保された配列
- Func:配列を操作するプログラムを構成するデータフローグラフの要素
- ImageParam:Image へのポインタ、データフローグラフの入力点



Halide におけるステンシル計算の実装例

```
Var x,y;                  // 配列添字を表す変数
Image input, output; // 計算結果を格納する配列変数
ImageParam inPar;     // Halide プログラムの入力
Func cell2 = a * inPar(x,y) + b * inPar(x+1,y); // Halide プログラム
Func cell3 = c * cell2(x,y) + d * cell2(x,y+1); // Halide プログラム

// 計算戦略を指定
cell3.split(y, yo, yi, 16).parallel(yo).parallel(yo).vectorize(x,4);
cell2.store_at(cell3,yo).compute_at(cell3,yi).vectorize(x,4);

for (int t=0; t<=MAX_T; ++t) {
  inPar.set(input);                  // プログラムに入力を設定
  output=cell3.realize(NX,NY);  // プログラムを実行
  std::swap(input, output);       // 出力を次のタイムステップの入力へ交換
}
```

Halideのコード生成

生成されたコードはその場で主プログラムにリンクされて実行される (Just in Time compile, JIT) のほか、C++言語ソースやアセンブリ、LLVM バイトコードをファイルに生成させ、のちほど利用する (Ahead of Time compile, AOT) こともできる。

```
compile_to_assembly
compile_to_bitcode
compile_to_c
compile_to_file
compile_to_function_pointers
compile_to_header
compile_to_lowered_stmt
compile_to_native
compile_to_object
compile_to_src
```

コデザイン用コード生成言語としての Halide/LLVM

例えば、Halide から .bc ファイルを生成した上で、Bulldozer 向けアセンブリを生成させることで、fma(融合加乗算) 命令を含むアセンブリを生成できる。

```
$ clang -O1 -march=bdver1 -ffp-contract=fast blur.bc -S
$ less blur.s
...
vfmaddss %xmm3, (%r11,%rdi,4), %xmm1, %xmm3
vfmaddss %xmm3, (%r11,%r10,4), %xmm1, %xmm3
vfmaddss %xmm4, (%r11,%rax,4), %xmm1, %xmm4
vfmaddss %xmm4, (%r11,%r14,4), %xmm1, %xmm4
vmulss %xmm0, %xmm4, %xmm4
...
```

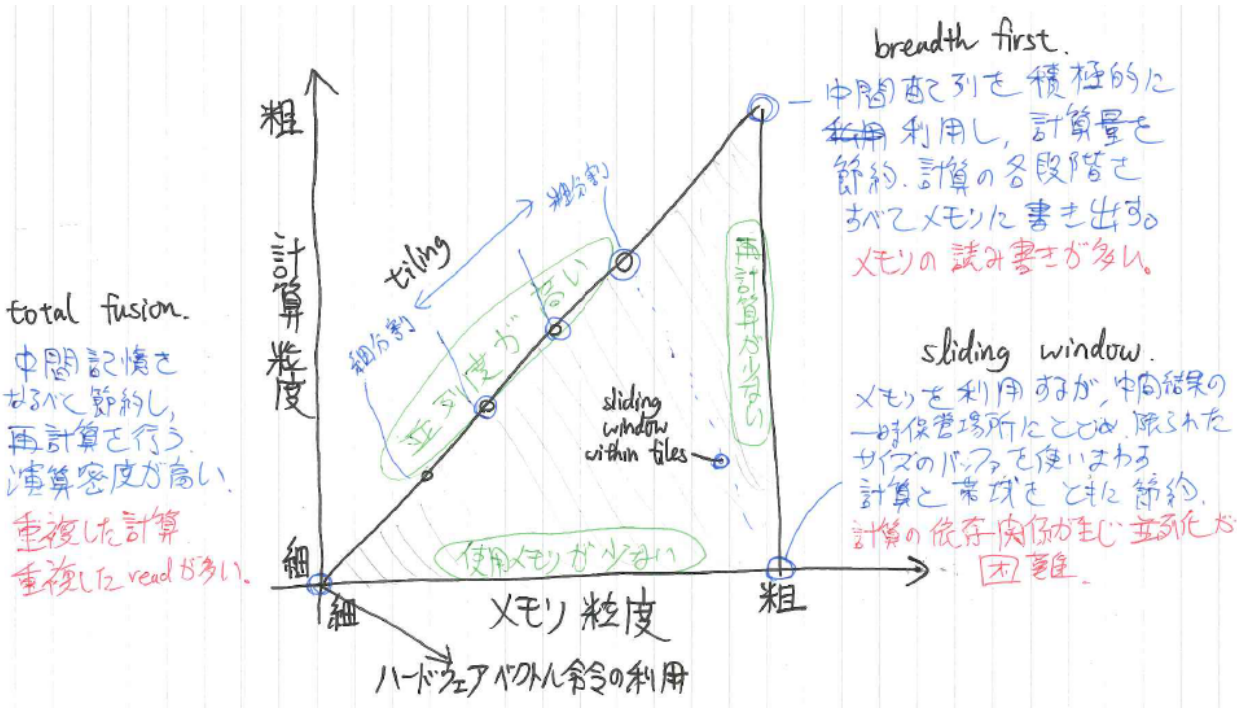
これは Grape-X のような fma をサポートするハードウェア向けのコード生成にも流用できると思われる。

Section 3

Halide の最適化

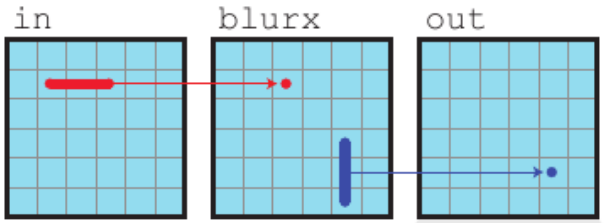
Halide の最適化空間

Halide は、ステンシル計算の最適化問題を、低並列度、演算重複、メモリ負荷、の間のトレードオフがある中で計算とメモリの粒度を操作することで、計算の速度を最大化する問題と捉えている。
計算とメモリの粒度とは何か？以下、拡散方程式を例に見ていく。



Halideの最適化空間:粗メモリ、粗計算

$$
\begin{aligned}
C_{\text{in}} &= \text{initial condition} \\
C_{\text{blurx}}[i,j] &= \frac{1}{3}\left(C_{\text{in}}[i-1,j] + C_{\text{in}}[i,j] + C_{\text{in}}[i+1,j]\right) \\
C_{\text{out}}[i,j] &= \frac{1}{3}\left(C_2[i,j-1] + C_2[i,j] + C_2[i,j+1]\right)
\end{aligned}
$$

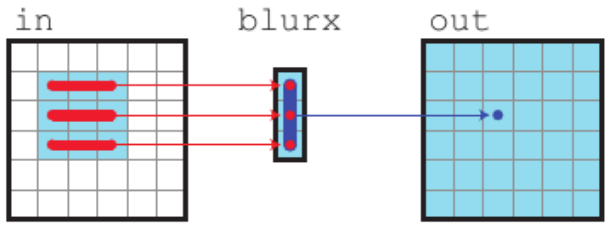


breadth first: each function is entirely evaluated before the next one.

$C_{\text{blurx}}[i,j]$ を全て計算し、メモリに置いてから、それを利用して $C_{\text{out}}[i,j]$ を計算する。

Halideの最適化空間:細メモリ、細計算

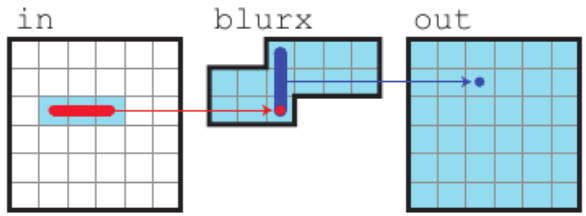
$$
\begin{aligned}
C_{\text{in}} &= \text{initial condition} \\
C_{\text{blurx}}[i,j] &= \frac{1}{3}\left(C_{\text{in}}[i-1,j] + C_{\text{in}}[i,j] + C_{\text{in}}[i+1,j]\right) \\
C_{\text{out}}[i,j] &= \frac{1}{3}\left(C_2[i,j-1] + C_2[i,j] + C_2[i,j+1]\right)
\end{aligned}
$$



total fusion: values are computed on the fly each time that they are needed.

中間記憶を最低限しか使わず、$C_{\text{out}}[i,j]$ の各要素を計算するたびに、必要な $C_{\text{blurx}}[i,j]$ の要素を全て再計算する。

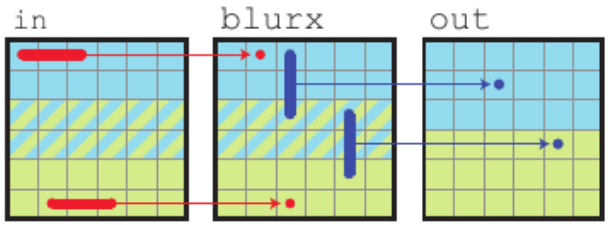
Halide の最適化空間:粗メモリ、細計算



sliding window: values are computed when needed then stored until not useful anymore.

$C_{\mathrm{out}}[i,j]$ の各要素を計算するたびに、新たな $C_{\mathrm{blurx}}[i,j]$ の要素を計算するが、それを保持しておき再利用。これ以上利用されなくなった時に破棄。中間結果をぴったり保持できるだけのバッファ(この例だと 3 行分)を使いまわす。

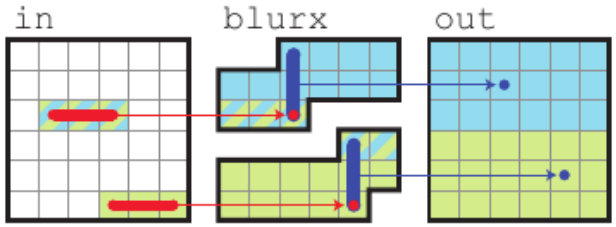
粗メモリ粗計算と細メモリ細計算の中間



tiles: overlapping regions are processed in parallel, functions are evaluated one after another.

配列を１次元または２次元のタイルに分割し、タイル１個分を記憶できるだけのバッファを用意し、その中ではすべての中間結果を保持する。
タイルバッファを使いまわす。
タイルバッファのサイズがキャッシュに収まるときには高性能が期待できる。

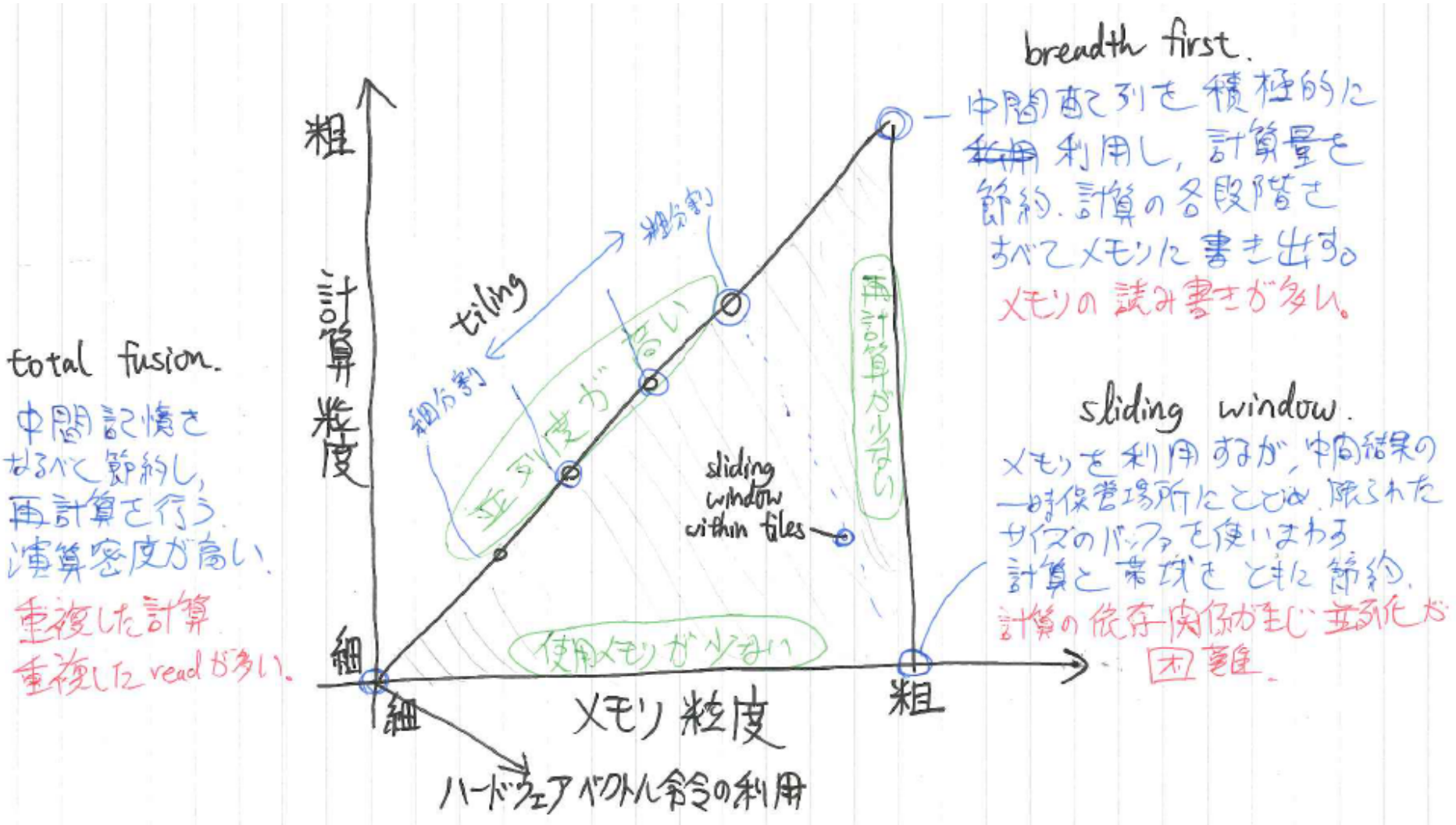
すべての中間



sliding windows within tiles:
tiles are evaluated in parallel
using sliding windows.

配列を１次元または２次元のタイルに分割し、タイル１個分を記憶できるだけのバッファを用意し、その中では最低限のの中間結果を保持するバッファを使いまわす。タイルごとの計算は並列に行う。

Halide の最適化空間



Halideがサポートするプログラム変換の種類

ステンシル計算の実装空間を探求するため、Halide はさまざまなプログラム変換をサポートする。

- 分割:1..(M*N) までのループを 1..M と 1..N の二重ループに分割する
- 融合:1..M と 1..N の 2 つのループを 1 つにまとめてしまう
- スレッド並列化:各添字を並列に計算
- アンロール:固定長のループを全部展開してループを消す
- ベクトル化:固定長のループをベクトル命令に置き換える
- タイル化:2 次元のブロックを用意しそれを更新
- 入れ子ループの入れ替え
- 特殊化:条件変数が真の場合と偽の場合でループを分けループの中身を簡単化

プログラム変換:ループ分割

```
EXPORT Func& Halide::Func::split ( Var  old,
                                   Var  outer,
                                   Var  inner,
                                   Expr factor
                                 )
```

Split a dimension into inner and outer subdimensions with the given names, where the inner dimension iterates from 0 to factor-1.

The inner and outer subdimensions can then be dealt with using the other scheduling calls. It's ok to reuse the old variable name as either the inner or outer variable.

Examples:

tutorial/lesson_05_scheduling_1.cpp, tutorial/lesson_08_scheduling_2.cpp, and tutorial/lesson_09_update_definitions.cpp.

プログラム変換:ループ融合

```
EXPORT Func& Halide::Func::fuse ( Var inner,
                                  Var outer,
                                  Var fused
                                )
```

Join two dimensions into a single fused dimenion.

The fused dimension covers the product of the extents of the inner and outer dimensions given.

Examples:

tutorial/lesson_05_scheduling_1.cpp.

プログラム変換:並列化

EXPORT Func& Halide::Func::parallel (Var var)

Mark a dimension to be traversed in parallel.

Examples:

tutorial/lesson_04_debugging_2.cpp, **tutorial/lesson_05_scheduling_1.cpp**, **tutorial/lesson_08_scheduling_2.cpp**, and **tutorial/lesson_09_update_definitions.cpp**.

プログラム変換:アンロール

EXPORT Func& Halide::Func::unroll (Var var)

Mark a dimension to be completely unrolled.

The dimension should have constant extent - e.g. because it is the inner dimension following a split by a constant factor. For most uses of unroll you want the two-argument form.

Examples:

tutorial/lesson_05_scheduling_1.cpp.

EXPORT Func& Halide::Func::unroll (Var var,
int factor
)

Split a dimension by the given factor, then unroll the inner dimension.

This is how you unroll a loop of unknown size by some constant factor. After this call, var refers to the outer dimension of the split.

プログラム変換:ベクトル化

EXPORT Func& Halide::Func::vectorize (Var var)

Mark a dimension to be computed all-at-once as a single vector.

The dimension should have constant extent - e.g. because it is the inner dimension following a split by a constant factor. For most uses of vectorize you want the two argument form. The variable to be vectorized should be the innermost one.

Examples:

tutorial/lesson_05_scheduling_1.cpp, tutorial/lesson_08_scheduling_2.cpp, and tutorial/lesson_09_update_definitions.cpp.

EXPORT Func& Halide::Func::vectorize (Var var,
int factor
)

Split a dimension by the given factor, then vectorize the inner dimension.

This is how you vectorize a loop of unknown size. The variable to be vectorized should be the innermost one. After this call, var refers to the outer dimension of the split.

プログラム変換:タイル分割

```
EXPORT Func& Halide::Func::tile ( Var  x,
                                  Var  y,
                                  Var  xo,
                                  Var  yo,
                                  Var  xi,
                                  Var  yi,
                                  Expr xfactor,
                                  Expr yfactor
                                )
```

Split two dimensions at once by the given factors, and then reorder the resulting dimensions to be xi, yi, xo, yo from innermost outwards.

This gives a tiled traversal.

Examples:
tutorial/lesson_05_scheduling_1.cpp, and tutorial/lesson_08_scheduling_2.cpp.

Section 4

ベンチマーク実験

ベンチマーク対象と環境

以下のようなマシンで、拡散程式ソルバをスケジュールや問題サイズを変えて実行し、性能の変化を調べた。似鳥さん、マシンを用意してくれてありがとう！

マシン名	armagnac0
CPU	AMD Opteron$^{(\mathrm{TM})}$ Processor 6376
freq	1400MHz
cores	64
Memory	256GB

- ノート PC だと一晩かかった LLVM のビルドが数分
- `llvm$ make -j` fork 爆弾事件

ベンチマーク対象と環境

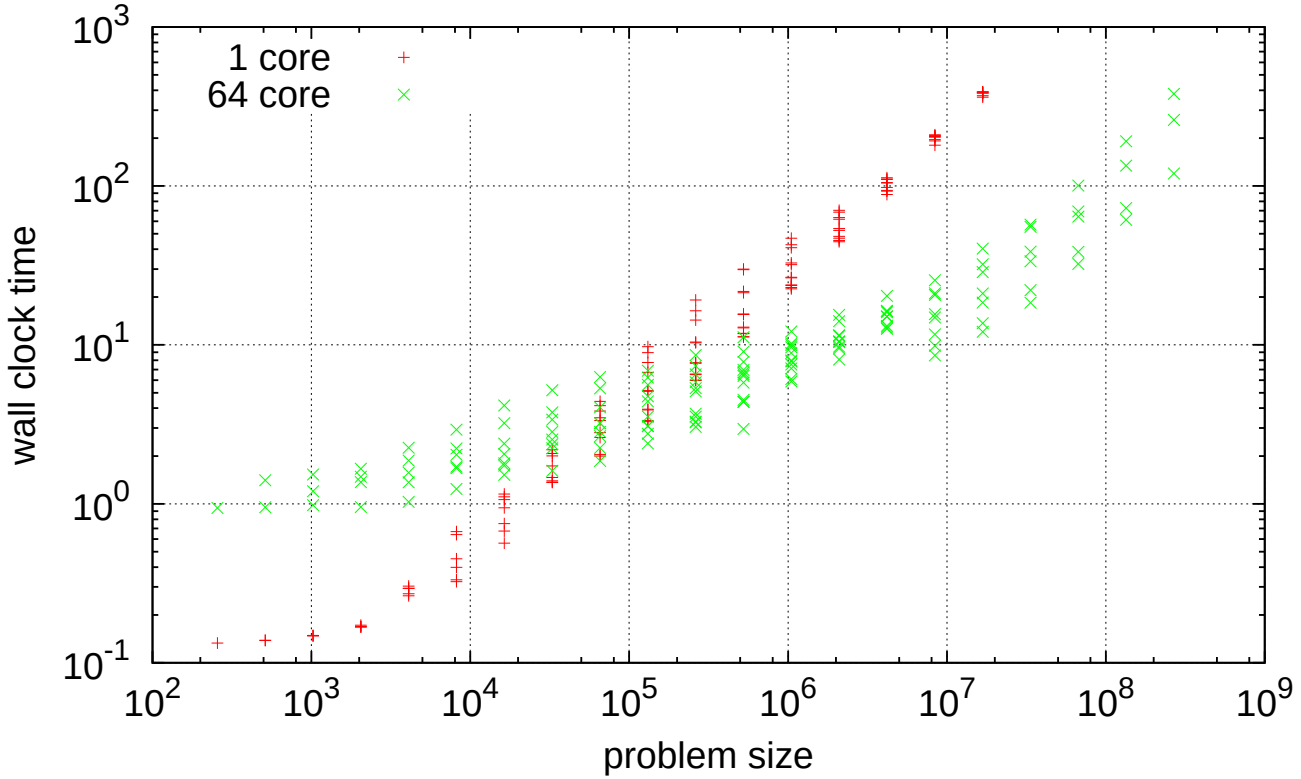
シミュレーション対象アルゴリズム:

$$
\begin{aligned}
C_{\mathrm{in}} &= \text{initial condition} \\
C_{\mathrm{blurx}}[i,j] &= \frac{1}{3}\left(C_{\mathrm{in}}[i-1,j] + C_{\mathrm{in}}[i,j] + C_{\mathrm{in}}[i+1,j]\right) \\
C_{\mathrm{out}}[i,j] &= \frac{1}{3}\left(C_2[i,j-1] + C_2[i,j] + C_2[i,j+1]\right)
\end{aligned}
$$

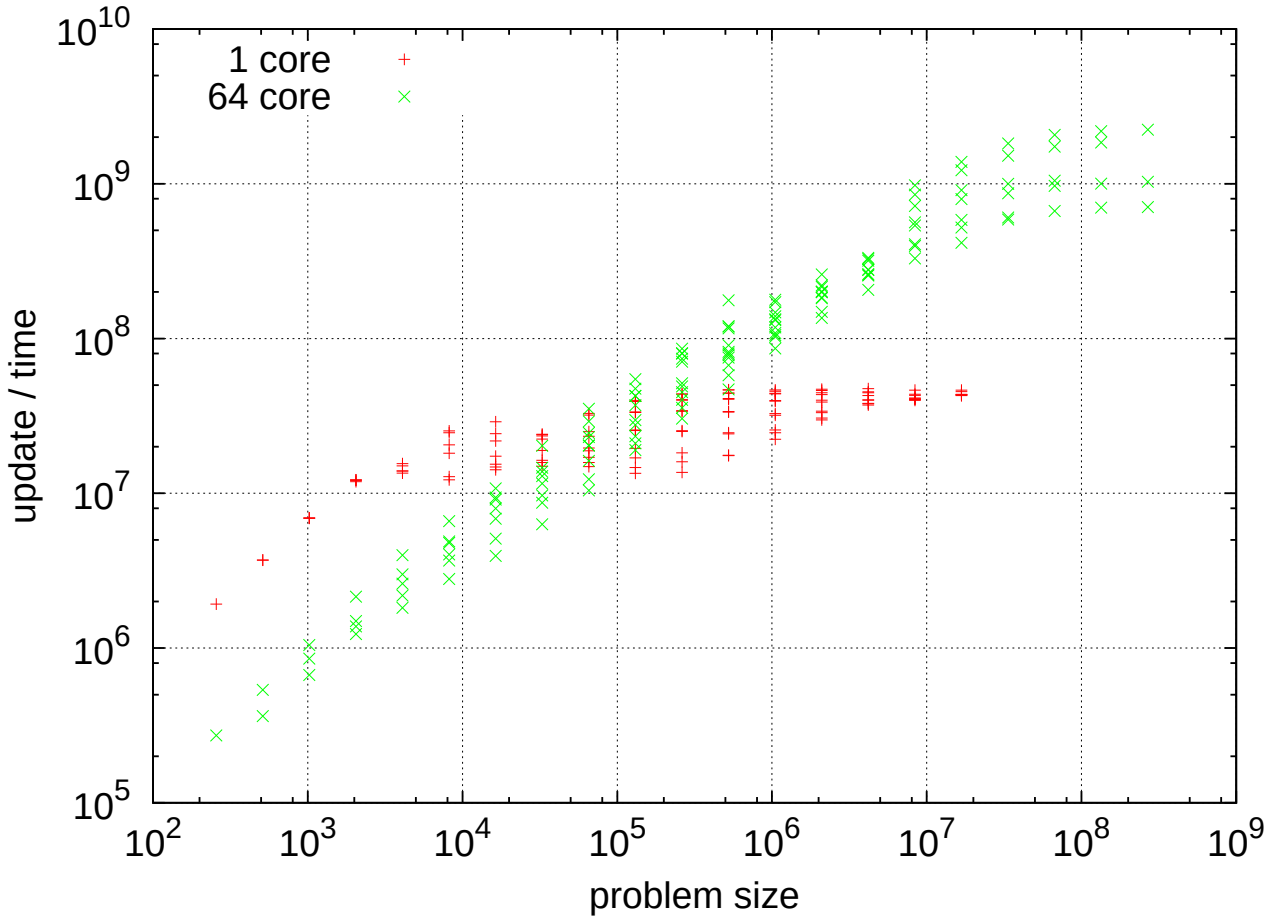
問題サイズ NX, NY をそれぞれ $[16..2^{15}]$ まで変化させて (コード生成時間を含む) 実行時間を測定する。
1 core:とくにコード変換なし
64 core:以下のコード変換を施す

```
cell3.split(y, yo, yi, 16).parallel(yo).vectorize(x,4);
cell2.store_at(cell3,yo).compute_at(cell3,yi).vectorize(x,4);
```

実行時間


1 core
64 core
wall clock time
problem size
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}
10^{8}
10^{9}

実行効率


1 core
64 core
update / time
problem size
10^{10}
10^{9}
10^{8}
10^{7}
10^{6}
10^{5}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}
10^{8}
10^{9}

参考文献 I

Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S. P., and Durand, F. (2012).
Decoupling algorithms from schedules for easy optimization of image processing pipelines.
ACM Trans. Graph., 31(4):32.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. (2013).
Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines.
ACM SIGPLAN Notices, 48(6):519–530.

Section 5

リサイクルボックス

Subsection 1

(4)

```
subroutine kernel__velderiv()
```