# 移流方程式の数値シミュレーションのSIMD演算による高速化

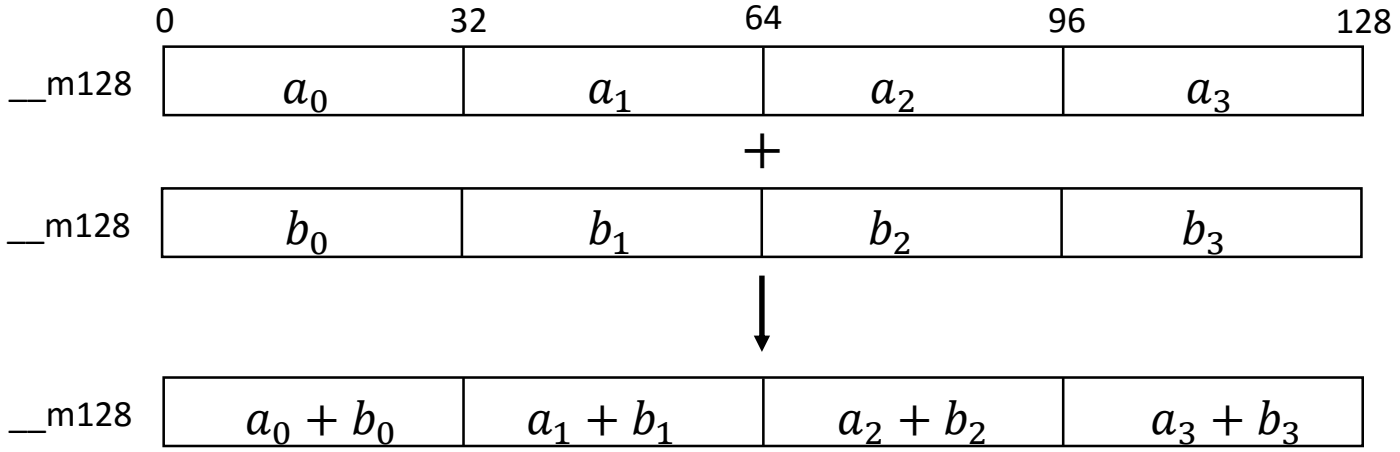
物理学類4年次　河田　隼季

# SIMD演算

SIMD : Single Instruction Multiple Data
ひとつの命令で複数のデータに対して処理を行う命令形式。
データ列に適用して、計算の並列化を行える。Intel CPUの組み込み関数。

・SSE
128bit SIMD用レジスタを用いて SIMD 演算を行う。
一本のレジスタに単精度(32bit)なら4つ、倍精度（64bit）なら2つを格納し、
1度に処理できる。

ex) 加算




※ __m128 : 128bit レジスタ

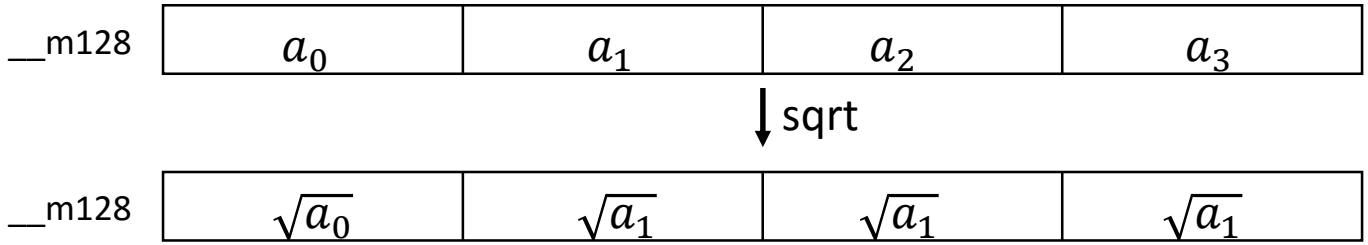
・AVX
SIMD用レジスタを 256bit へ拡張。
単精度なら8つ、倍精度なら4つのデータをレジスタに格納して扱える。

# SIMD命令

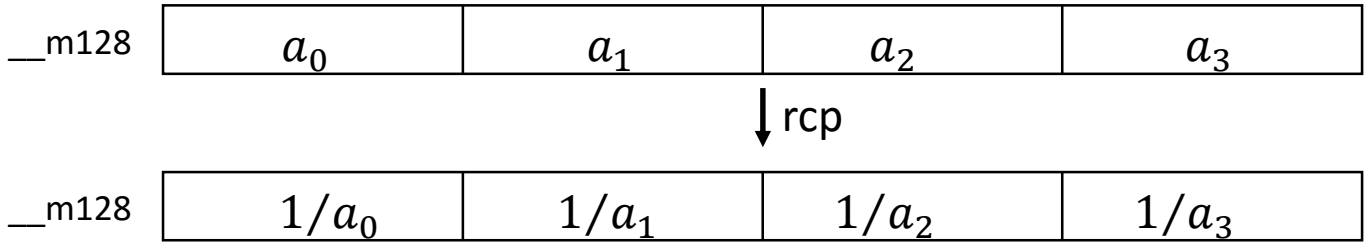
## ○演算系命令

・四則演算（加算・減算・乗算・除算）

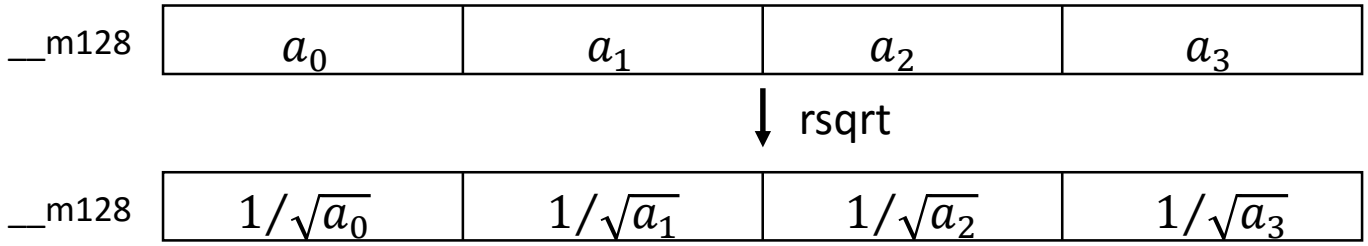
・平方根




・逆数




・逆数平方根

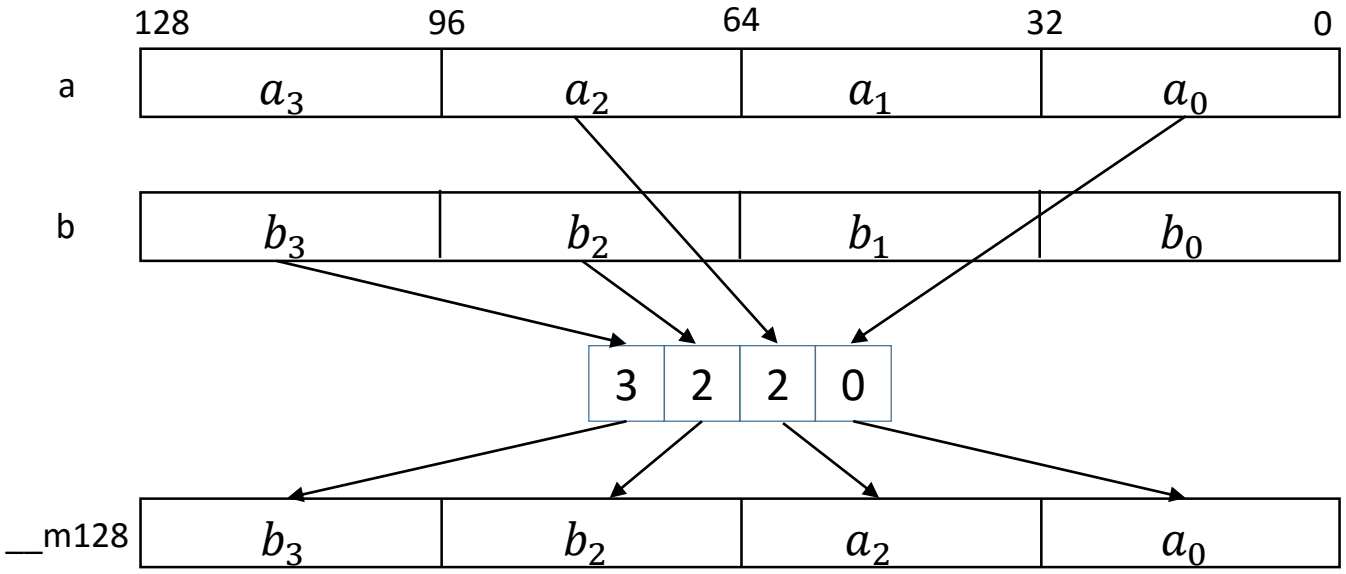


〇論理演算系命令
論理積（＆）、論理和（｜）、排他的論理和（^）……など

〇比較系命令
＝、＞、≧、最大値、最小値……など

〇並び替え
Shuffle 命令
_mm_shuffle_ps(__m128 a, __m128 b, int msk)
msk で定義する定数で、結果のベクトルに移動させる要素を指定する。

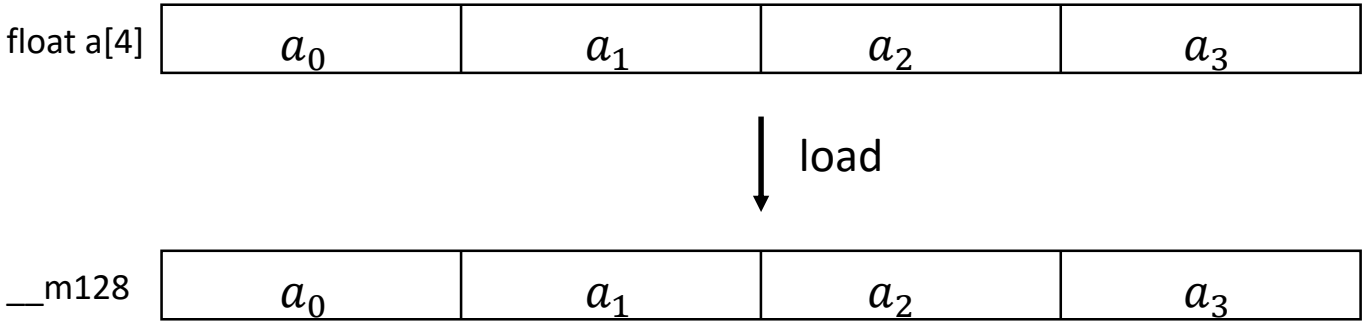
ex) _mm_shuffle_ps(a, b, _MM_SHUFFLE(3, 2, 2, 0))



〇 load / store

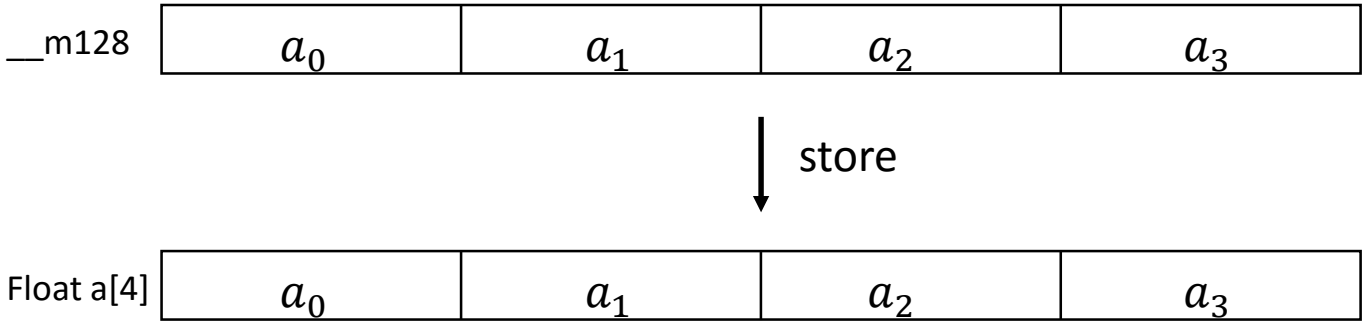
load : メモリからレジスタへの読み込み

_mm_load_ps(float *a)　*a は読み込みたいデータ列の先頭ポインタ




store : レジスタからメモリへの書き込み

_mm_store_ps(float *a, __m128 _a)　*aは書き込みたいメモリのアドレス



# SIMD利用の例

## ○ベクトルの内積の計算の高速化（SSE）

```
#include <immintrin.h> // SIMDを利用するためのヘッダファイル

Float dotproduct(float vec1[], float vec2[], int n)
{
        __m128 _sum = {0};
        __m128 _v1, _v2;
        float a[4] = {0};

        for(int i = 0; i < n; I += 4){
                _v1 = _mm_load_ps(&vec1[i]); //_v1にvec1のデータ列を読み込む
                _v2 = _mm_load_ps(&vec2[i]); //_v2にvec2のデータ列を読み込む

                _v1 = _mm_mul_ps(_v1, _v2); //_v1と_v2を乗算し、結果を_v1に格納
                _sum = _mm_add_ps(_sum, _v1);    //_v1を_sumに加算
        }
        _mm_store_ps(a, _sum); // float a[4]に_sumを書き込む
        return a[0] + a[1] + a[2] + a[3];
}
```

## ○多体問題専用計算機　Phantom-GRAPE

重力相互作用　$\mathbf{a}_i = \sum_j \frac{m_j \mathbf{r}_{ij}}{\left(r_{ij}^2 + \varepsilon^2\right)^{3/2}}$

の計算の高速化

# SIMD でどのくらい速くなる？

- 「SSEでは一度に4つの演算を行えるのだから、4倍速になるのか？」
  というとそうでもない。コンパイラが並列度を抽出し自動で最適化する。

- ただし、コンパイラは十分賢いわけではないので、常にプログラムを十分に（速度）最適化できるわけではない。SIMDの使いようによっては大きな高速化を実現できる可能性がある。

- メモリにアクセスするのではコスト（時間）がかかるので、
  高速化を行うためにはできる限りレジスト間で演算が済むようにする。
  SIMDは同じような演算を繰り返すループ文で有効。

# 移流方程式

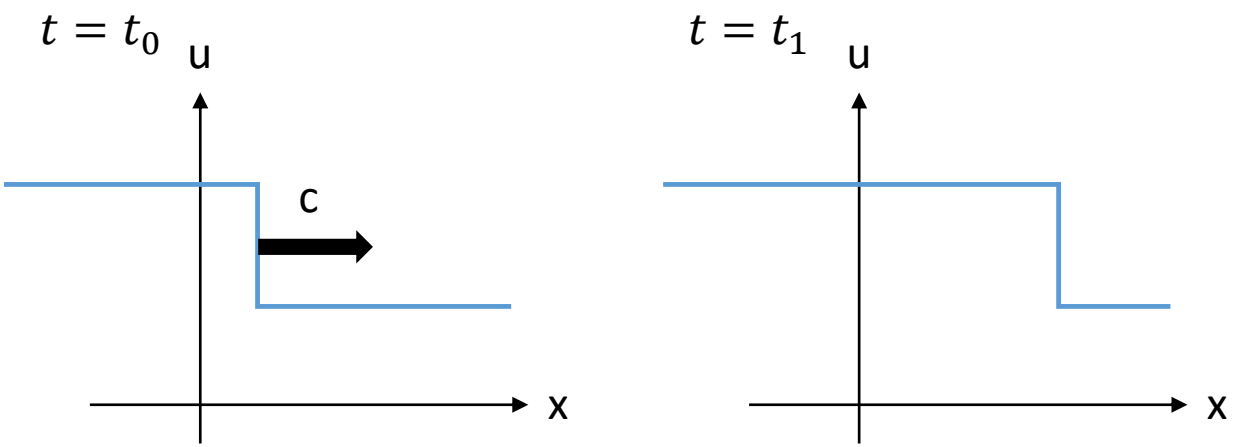
物理量$u(t,\ \boldsymbol{x})$が速度 $\boldsymbol{c}$ で移流するとき

$$\frac{\partial u(t,\boldsymbol{x})}{\partial t} + \boldsymbol{c}\cdot\boldsymbol{\nabla} u(t,\boldsymbol{x}) = 0$$

1次元

$$\frac{\partial u(t,x)}{\partial t} + c\frac{\partial u(t,x)}{\partial x} = 0$$

解析解　$u(t_1,x) = u(t_0,\ x - c\Delta t)$　　$(\Delta t = t_1 - t_0)$



# 差分法による数値解法

数値解法の安定性や精度は、

空間微分と時間微分の差分をどのようにとるかに依る。

今回は簡単のために1次精度風上差分（$c > 0$）について考える。

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} + c\frac{u_j^n - u_{j-1}^n}{\Delta x} = 0$$

$$\Leftrightarrow \quad u_j^{n+1} = u_j^n - \nu\left(u_j^n - u_{j-1}^n\right) \qquad \left(\nu = \frac{c\Delta t}{\Delta x}:\ \text{クーラン数}\right)$$

この計算の高速化を考える

2次元への適用は時間分割法から1ステップ時間発展させるとき

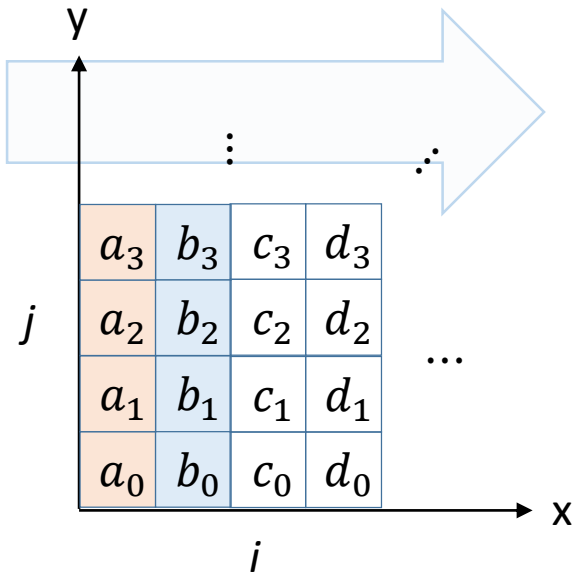
$$u^{n+1} = L_x^{1/2} L_y\, L_x^{1/2} u^n$$

のように x方向に$\Delta t/2$、y方向に$\Delta t$、x方向に$\Delta t/2$ 計算すればよい。

⇒ X方向とy方向のそれぞれでSIMDの適用を考えていく。

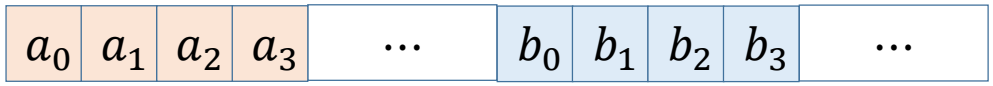
# X方向の差分計算

u[ i ][ j ]



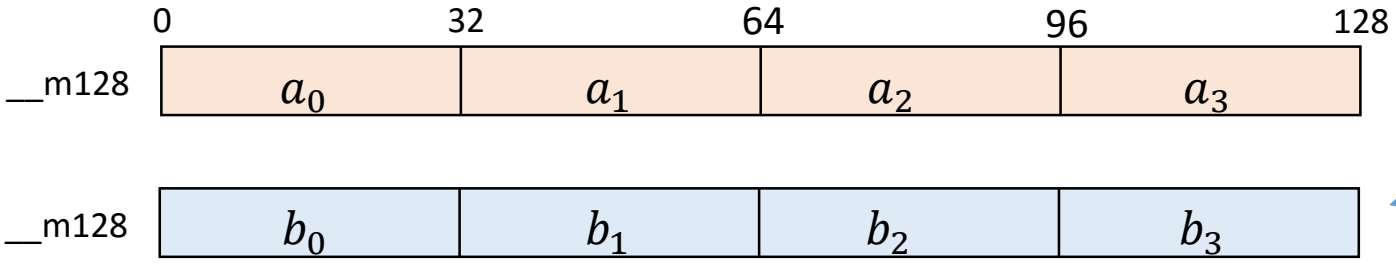

・loadを行うには、読み込みたいデータ列の先頭のポインタが必要。
・メモリ上でデータ列が連続しているのでそのままレジスタへ load できる。

u[ i ][ j ]（メモリ上）

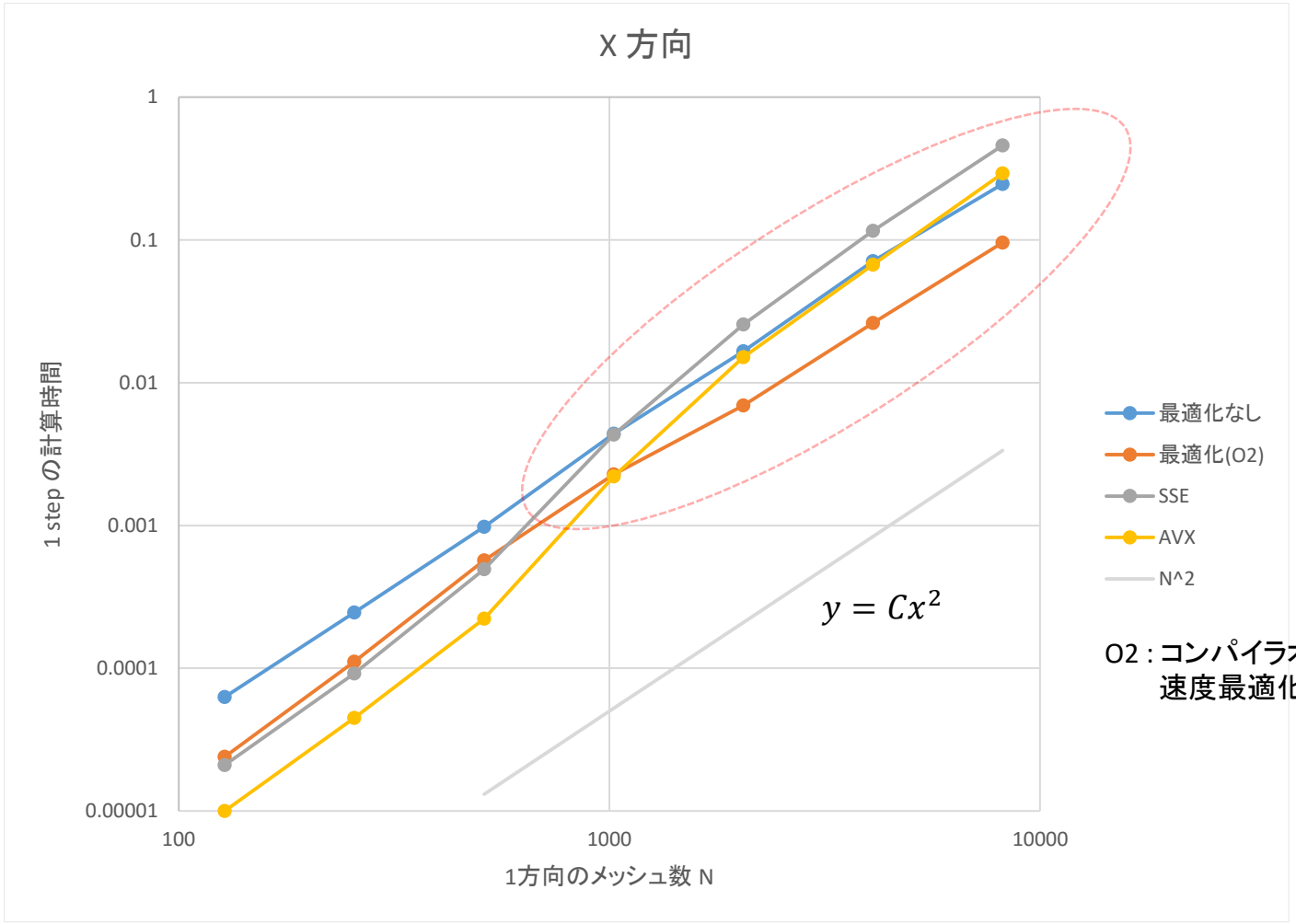



load

load




差分計算

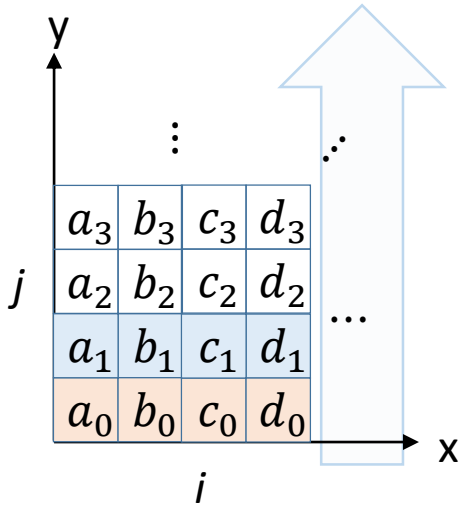



O2 : コンパイラオプション
速度最適化

N < 1000 では O2最適化と比べ、AVXで約2.5倍の高速化
N > 1000 では SIMD の方が遅くなる
メモリアクセスの問題なのか？ コンパイラに起因するものなのか?（原因検討中）

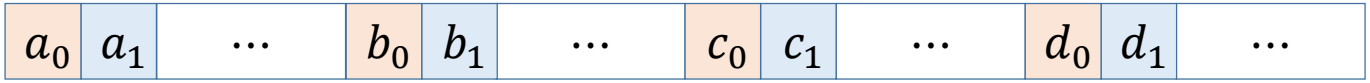
# y方向の差分計算

u[ i ][ j ]



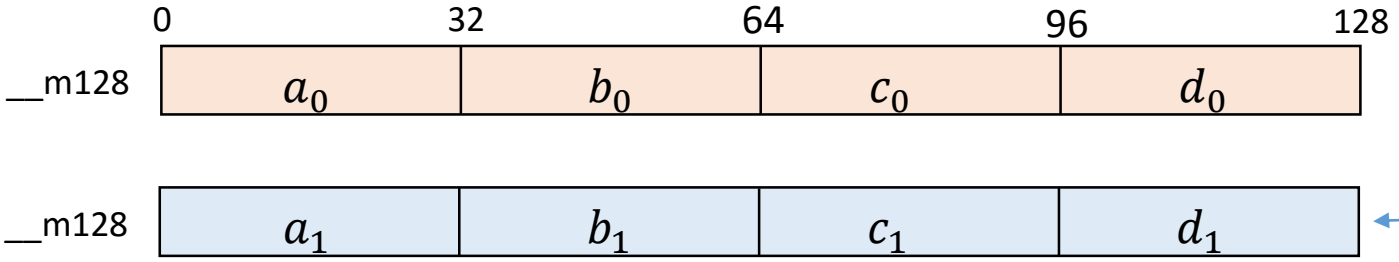

load するときにはデータ列の先頭ポインタを指定する。
メモリ上でデータが不連続なので、
そのままではレジスタに load できない。

u[ i ][ j ]（メモリ上）



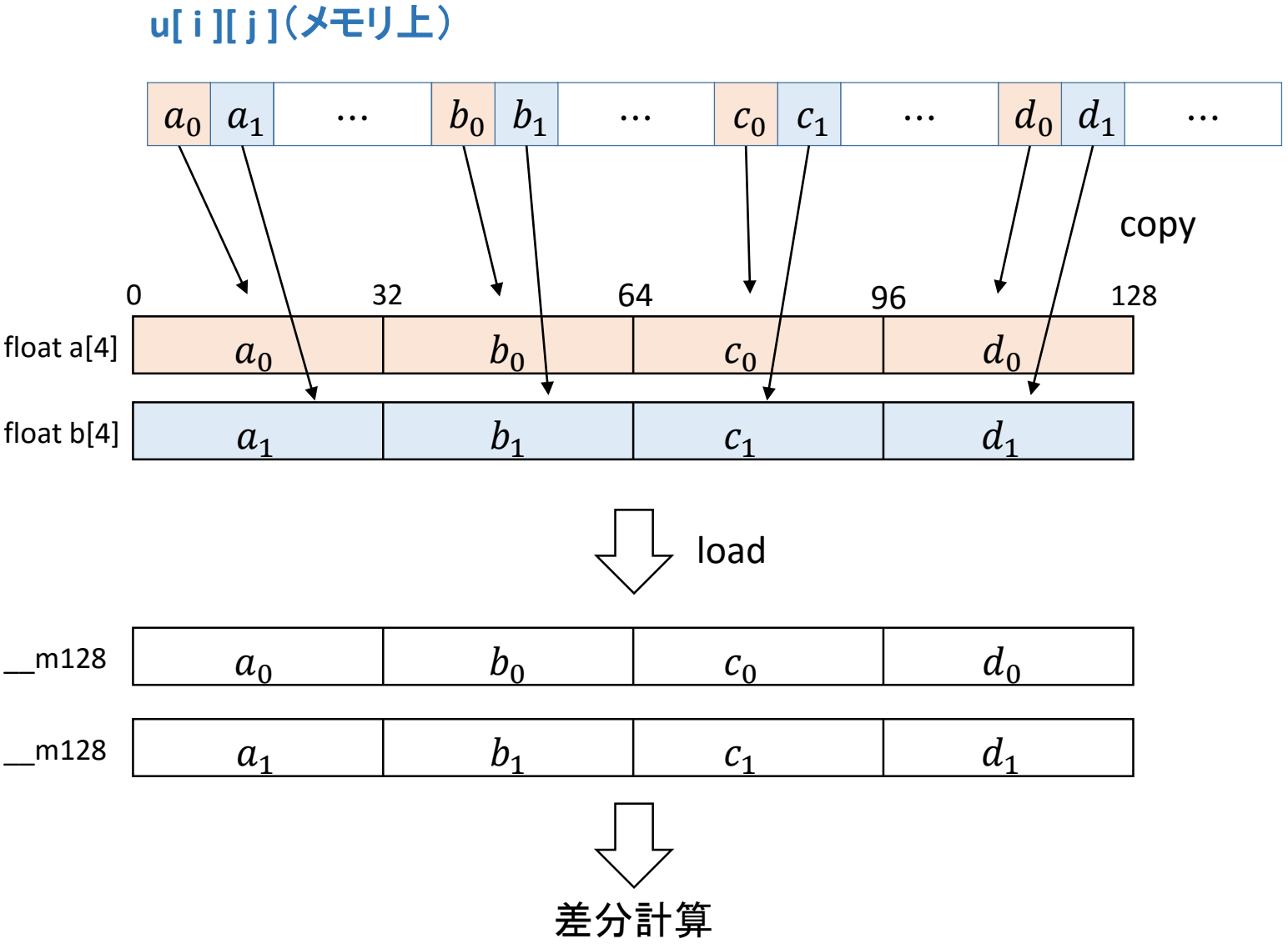

load

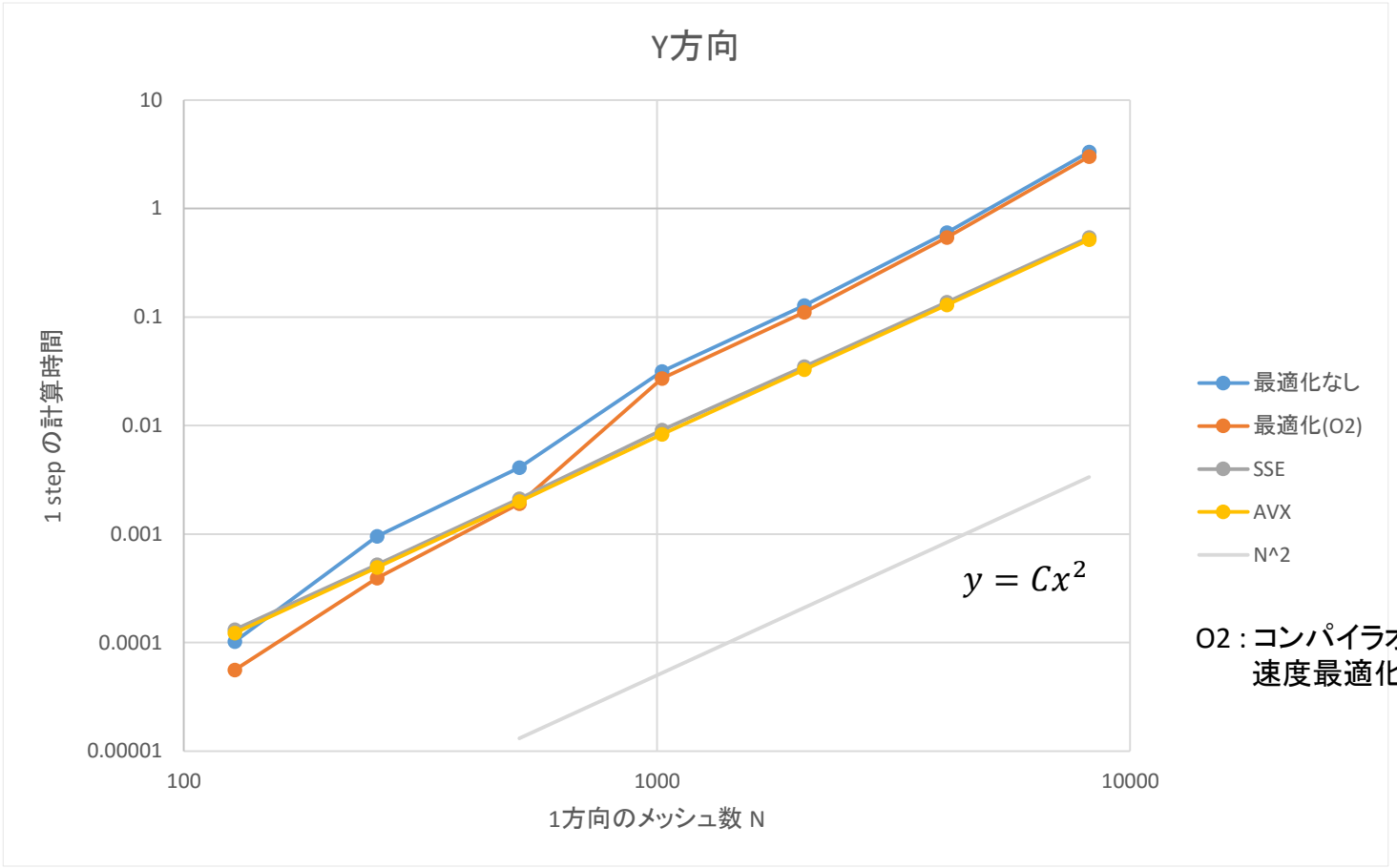
load



# 方法①　別の配列にコピー

要素数4の配列を用意し
コピー、load する。



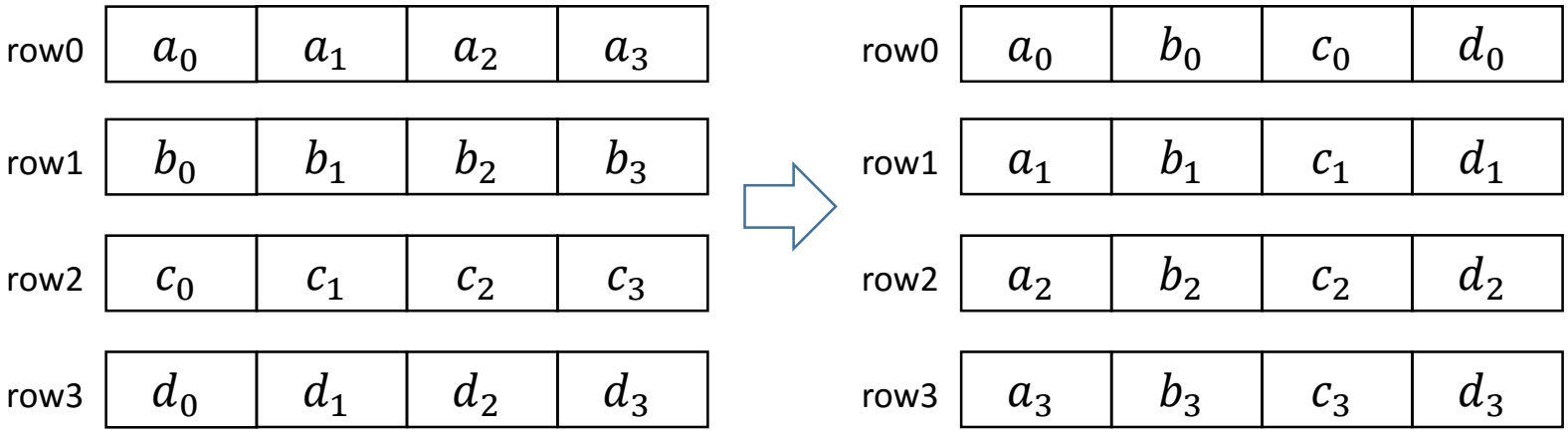



O2 : コンパイラオプション
速度最適化

SSEとAVXで大きな違いはないものの、Nが大きければ3~倍の高速化

# 方法②　行列の転置

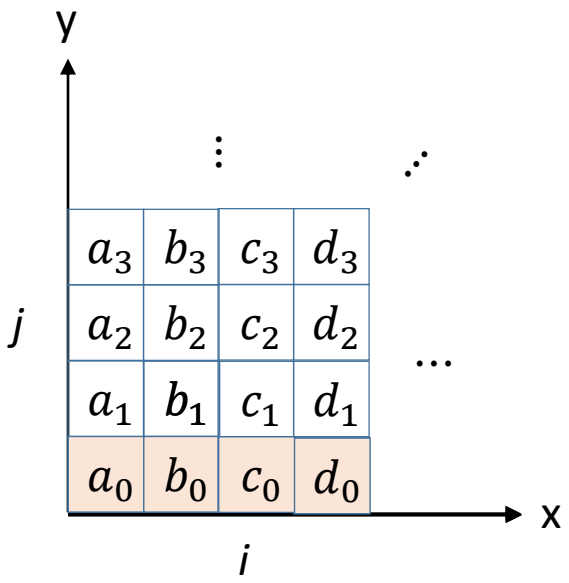
_MM_TRANSPOSE4_PS(__m128 row0, __m128 row1, __m128 row2, __m128 row3)
単精度浮動小数点値 4×4 行列を転置するマクロ関数




AVX において、単精度浮動小数点値 8×8 行列の転置は
permute（置換操作）命令や shuffle 操作命令、unpack 命令を駆使して実現できる。
Reference :
http://stackoverflow.com/questions/25622745/transpose-an-8x8-float-using-avx-avx2

# 計算過程の概略

u[ i ][ j ]



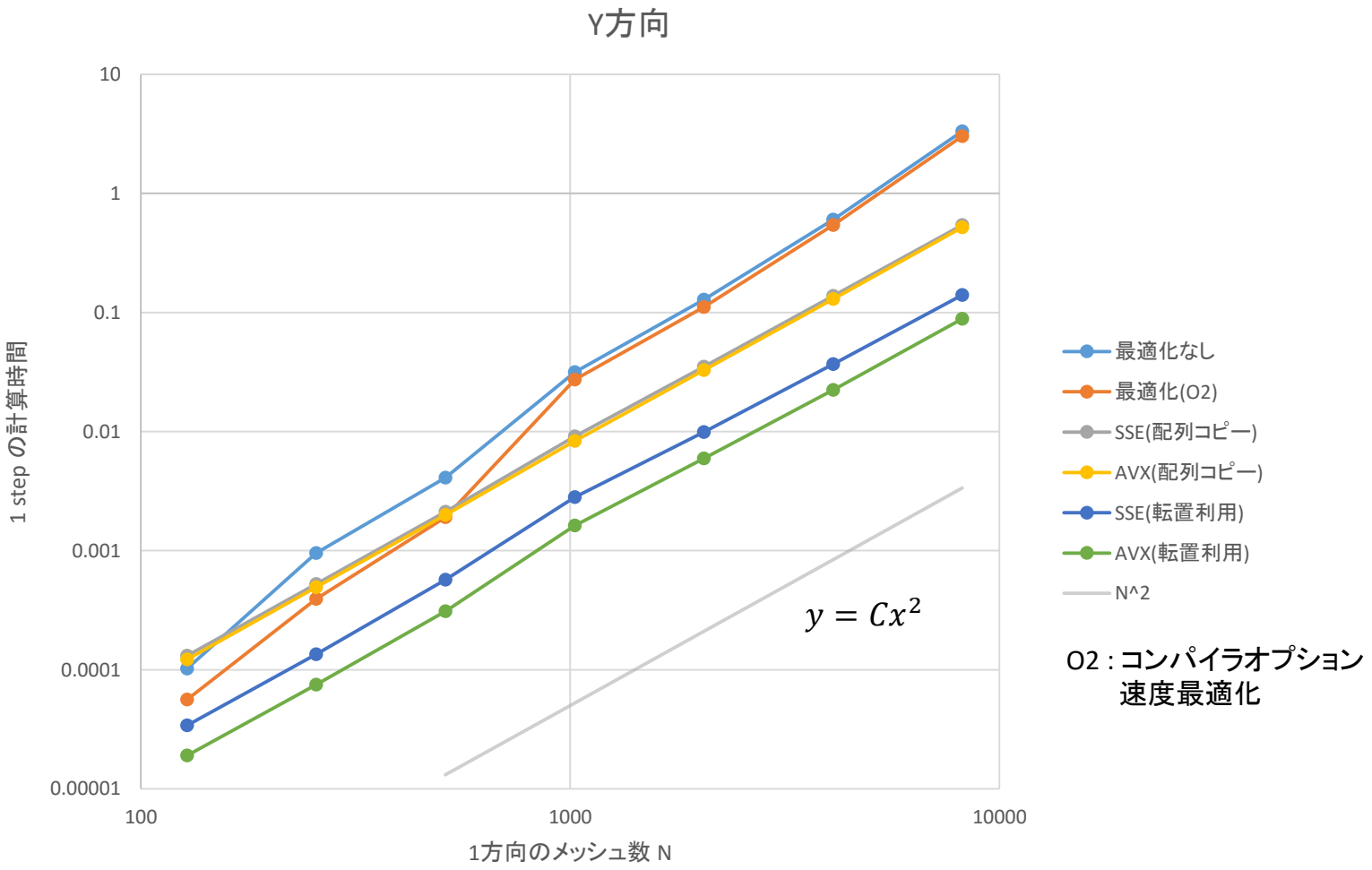

load

| | | | | |
|---|---|---|---|---|
| row0 | $a_0$ | $a_1$ | $a_2$ | $a_3$ |
| row1 | $b_0$ | $b_1$ | $b_2$ | $b_3$ |
| row2 | $c_0$ | $c_1$ | $c_2$ | $c_3$ |
| row3 | $d_0$ | $d_1$ | $d_2$ | $d_3$ |

transpose

| | | | | |
|---|---|---|---|---|
| row0 | $a_0$ | $b_0$ | $c_0$ | $d_0$ |
| row1 | $a_1$ | $b_1$ | $c_1$ | $d_1$ |
| row2 | $a_2$ | $b_2$ | $c_2$ | $d_2$ |
| row3 | $a_3$ | $b_3$ | $c_2$ | $d_3$ |

差分計算




先ほどの別の配列にコピーする方法よりも高速化を行うことができた。
N>1000 のとき、O2最適化を比べ、SSEで10~倍、AVXで17~倍の高速化。

# まとめ

X方向の差分計算では、N>1000でSIMD適用の方が遅くなってしまう。
→原因を検討していく。

y方向の差分計算では、行列の転置を利用することで大きく高速化を実現。
SSEで10~倍、AVXで17~倍。

# 今後の展望

1. ３次元への拡張
   ３次元配列のデータ構造を考えると、これらのSIMDの適用は
３次元の移流方程式の差分計算にもすぐに拡張できる。

2. 高精度差分法への適用
   高精度のMUSCL法へ適用していく