Apply SIMD to RMI

2024. 01. 31

Presentation by Yeojin Oh, ZhuYongjie, Boseng Kim

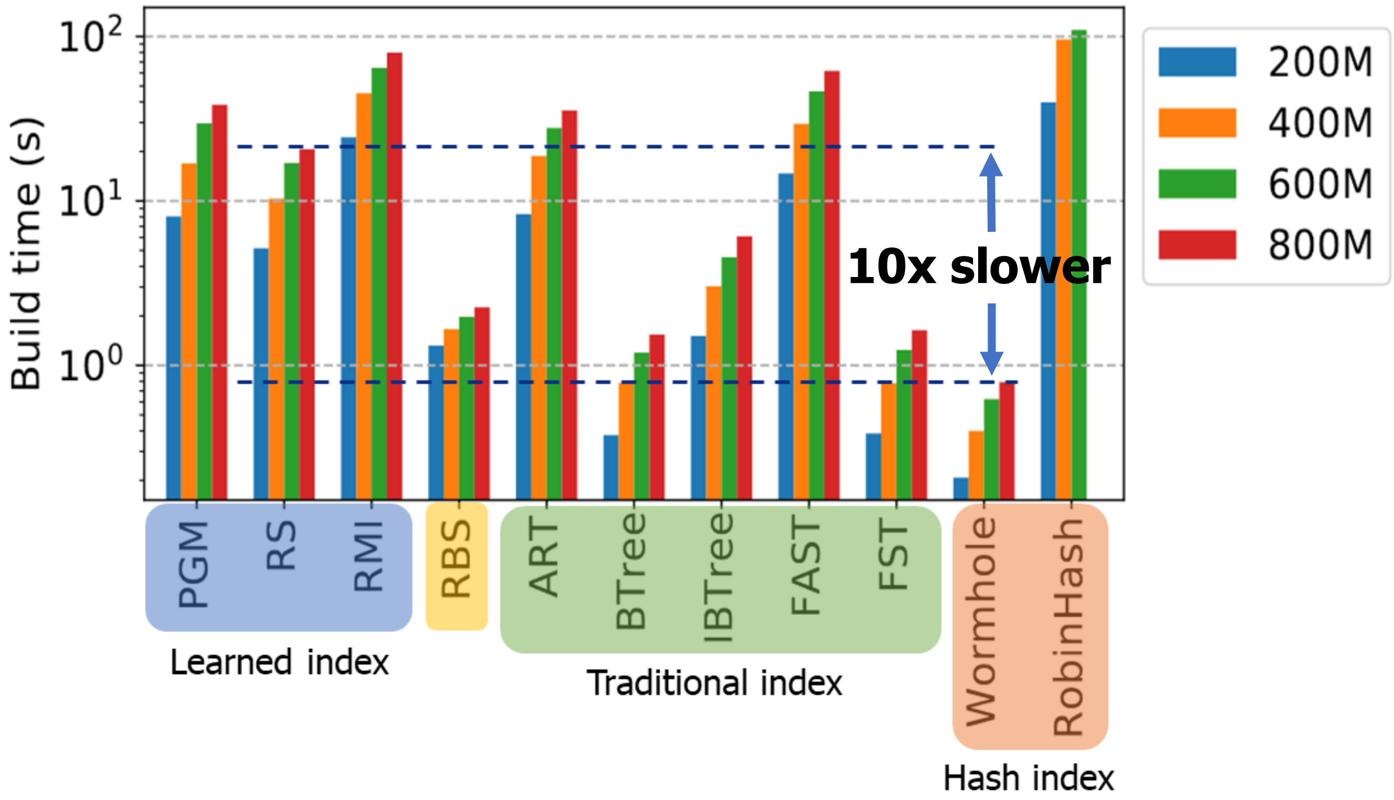
yeojinoh@dankook.ac.kr, aeashio1111@dankook.ac.kr, bskim1102@dankook.ac.kr

DANKOOK UNIVERSITY

Dankook University
System Software Laboratory

Contents

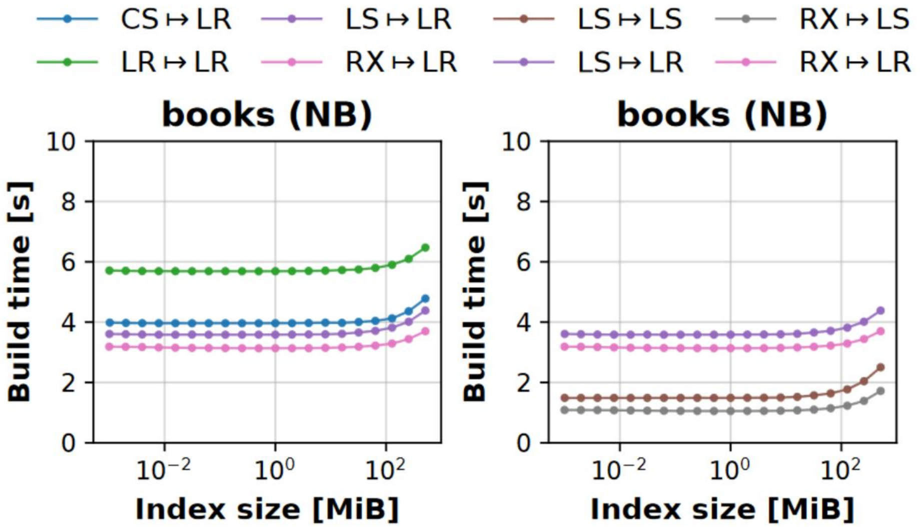
Problem



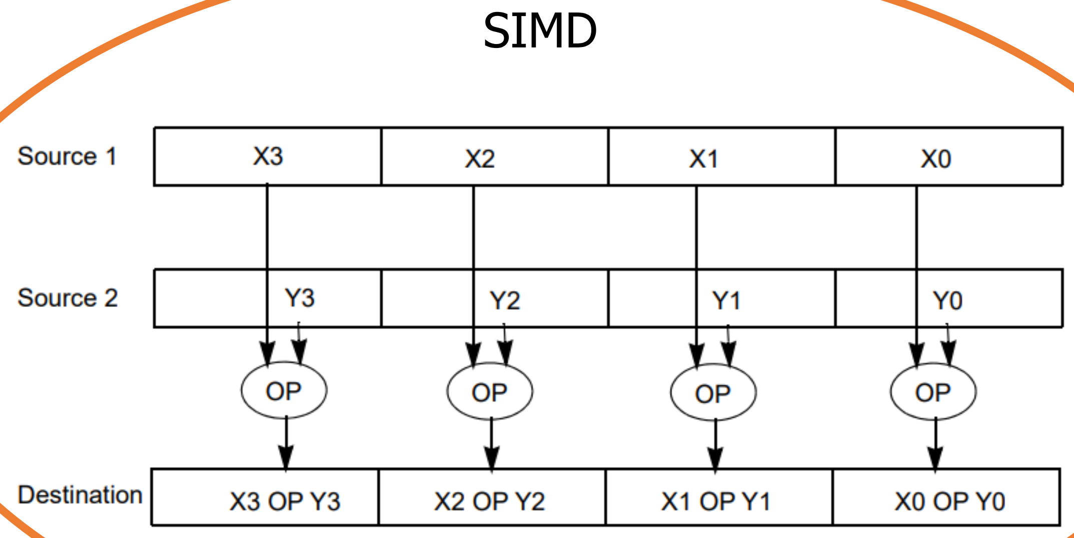
The learned index is 10x slower than the traditional index

Solution

Combination of model



(a) Layer 1 type (b) Layer 2 type



We will evaluate this!

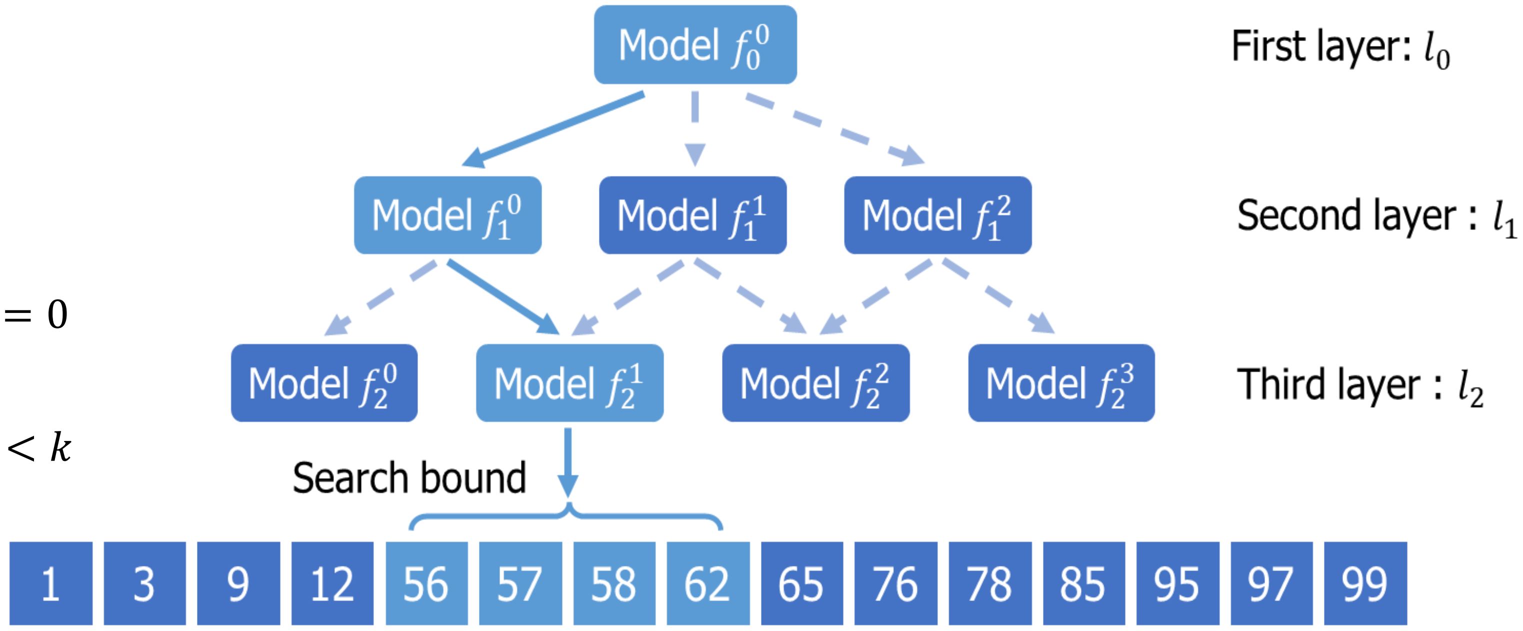
Structure of RMI

$$i = F_X(x_i) \times |D| = P(X \leq x_i) \times |D|$$

- D: dataset, $|D|$: size of dataset
- X: Random variable
- F_X: CDF of X
- x_i : Each key in the dataset D
- i: Index indicating the position of key x_i in the sorted array

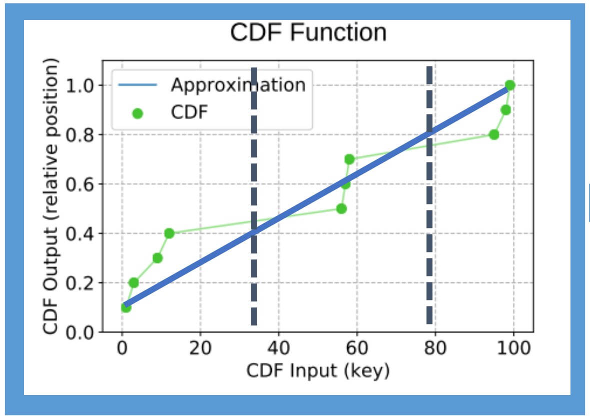
Structure of RMI

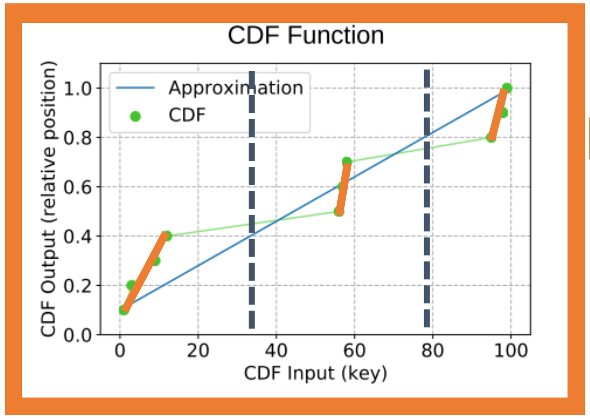
- Prediction
  - $[\![p]\!]_a^b \coloneqq \max(a, \min(p, b))$
    - p = predicted position
    - [a,b] : search bound
  - $f_{layaer\ level}^{model\ number}$
  - $$f_i(x) = \begin{cases} f_0^0(x) & i = 0 \\ f_i^{\left\lfloor [\![|l_i| \times \frac{f_{i-1}(x)}{n}]\!]_0^{|l_i|-1} \right\rfloor}(x) & 0 < i < k \end{cases}$$
  - $R(x) = f_{k-1}(x)$
- Error correction
  - $[R(x) - err, R(x) + err]$

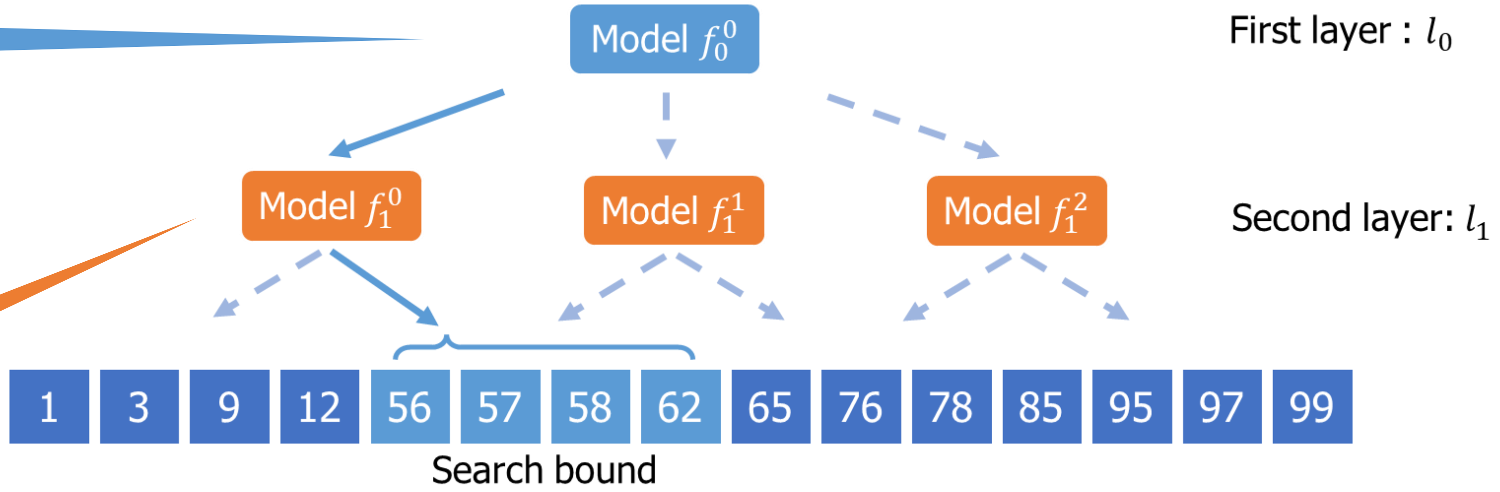


Structure of RMI

- Training Algorithm

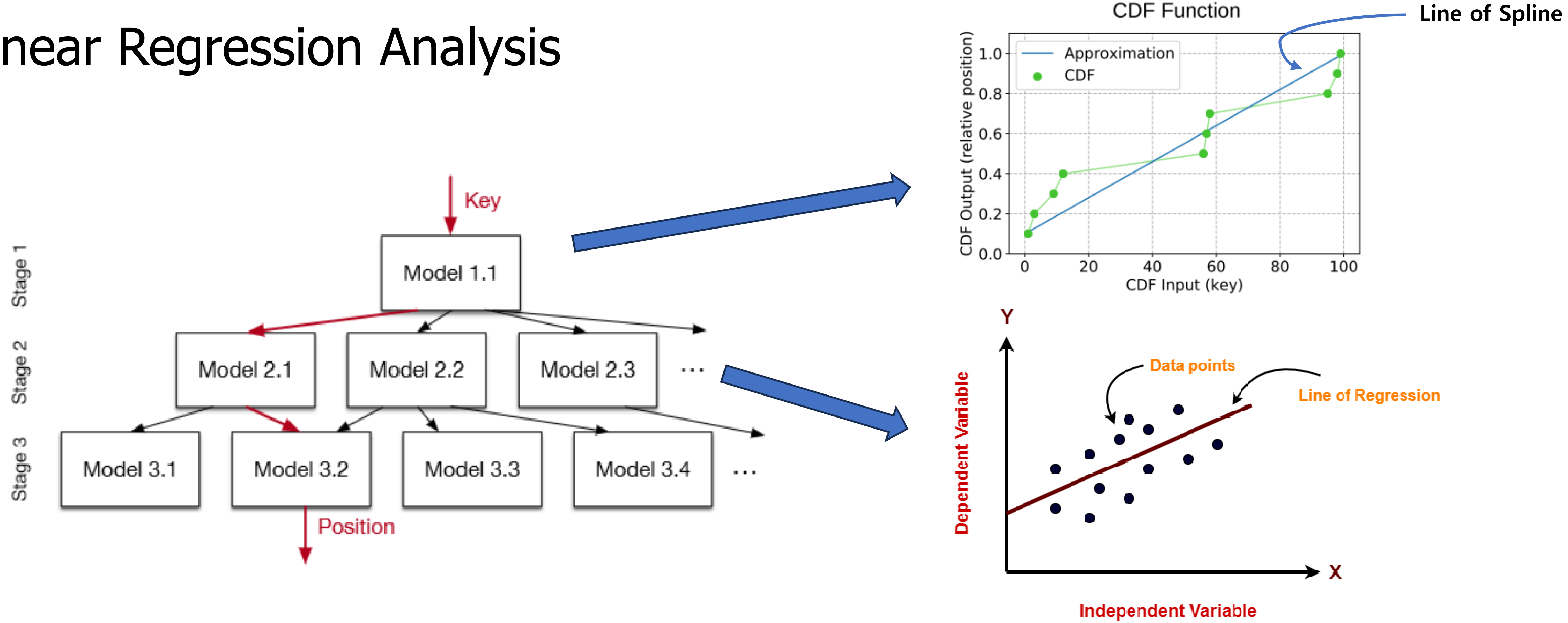






Linear Regression

- Linear Regression Analysis

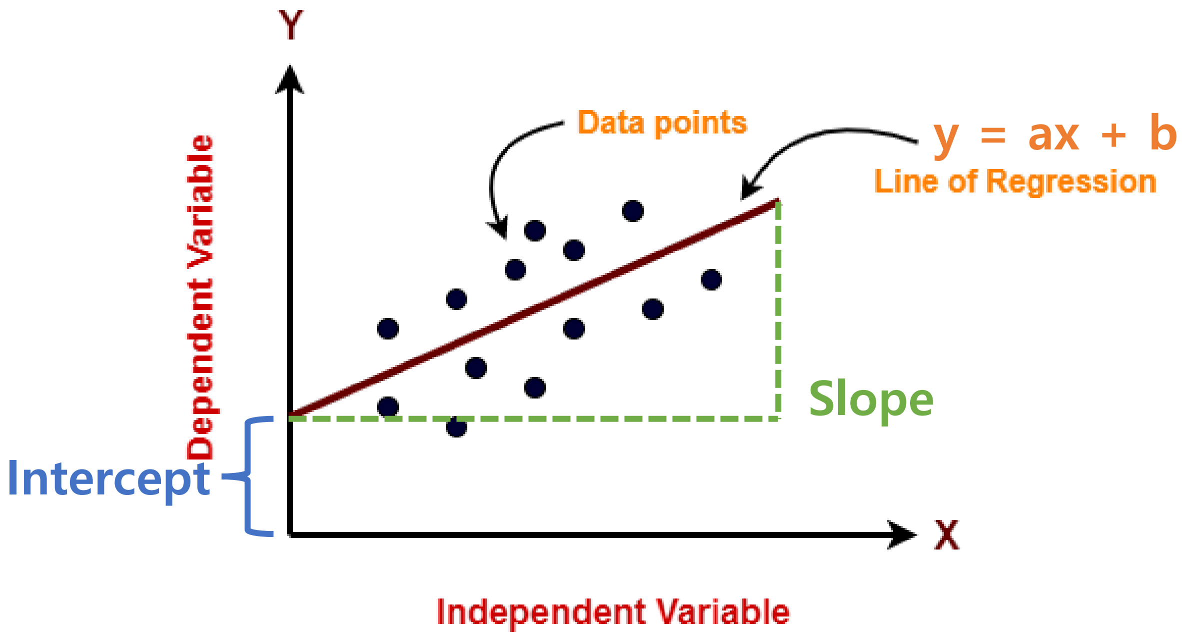


(1) The linear spline is responsible for presenting the overall data roughly (first and last key)

(2) Linear regression presents each segment of data in a relatively all key in its range (detailed manner)

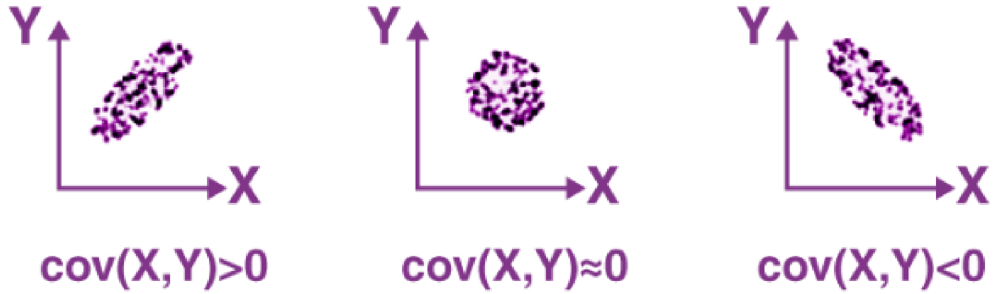
Linear Regression

- Linear Regression Analysis

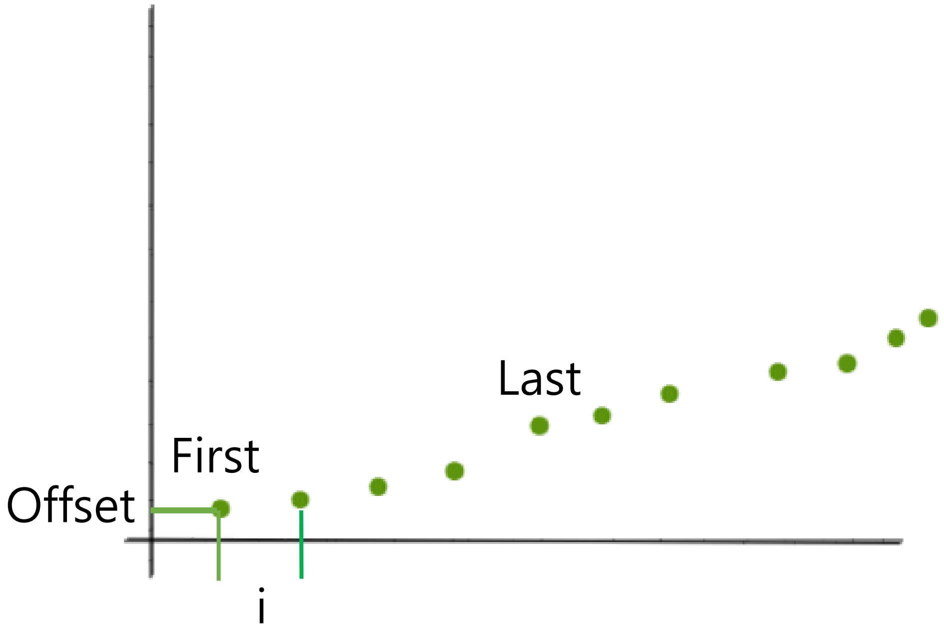


Linear regression requires **Slope** and **Intercept** to fit a line when expressing data

Linear Regression



- Linear Regression Analysis
  - Slope = $\underbrace{\text{Covariance(x,y)}}_{\text{The correlation between X and Y}}$ / $\underbrace{\text{Variance(x)}}_{\text{The distribution range of X}}$ (x = key, y = offset + i)



Collect x through interval and calculate y based on x.

Covariance = $\frac{1}{n}\Sigma_{i=1}^{n}(\chi_i - mean_x) \times (y_i - mean_y)$

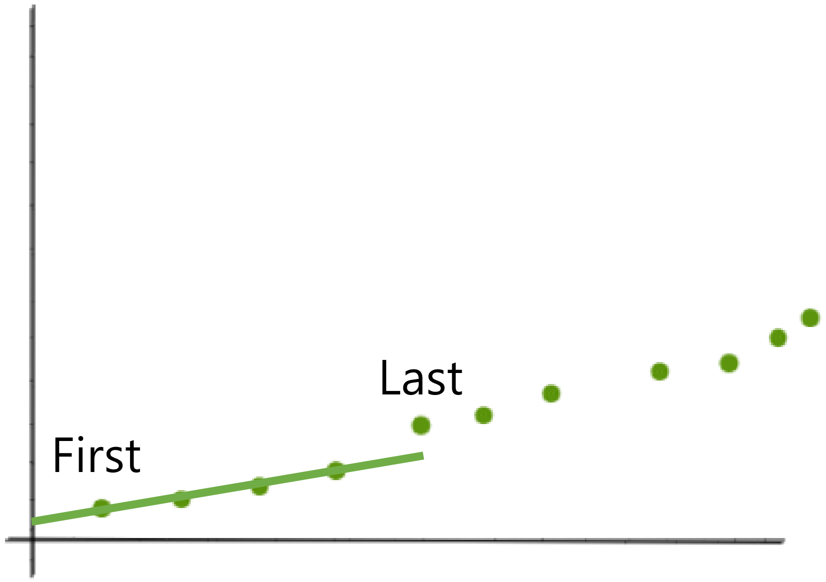
Variance :

$$dx = x_i - mean_x$$
$$mean_x = mean_x + \frac{dx}{i}$$
$$dx_2 = x_i - mean_x$$
$$m_2 = m_2 + dx \times dx_2$$
$$var = \frac{m_2}{n}$$

Slope = Convariance / Variance * compression_factor

Linear Regression

- Linear Regression Analysis
  - Slope = $\underbrace{\text{Covariance(x,y)}}_{\text{The correlation between X and Y}}$ / $\underbrace{\text{Variance(x)}}_{\text{The distribution range of X}}$ (x = key, y = offset + i)



Collect x through interval and calculate y based on x.

Covariance = $\frac{1}{n}\Sigma_{i=1}^{n}(\chi_i - mean_x) \times (y_i - mean_y)$

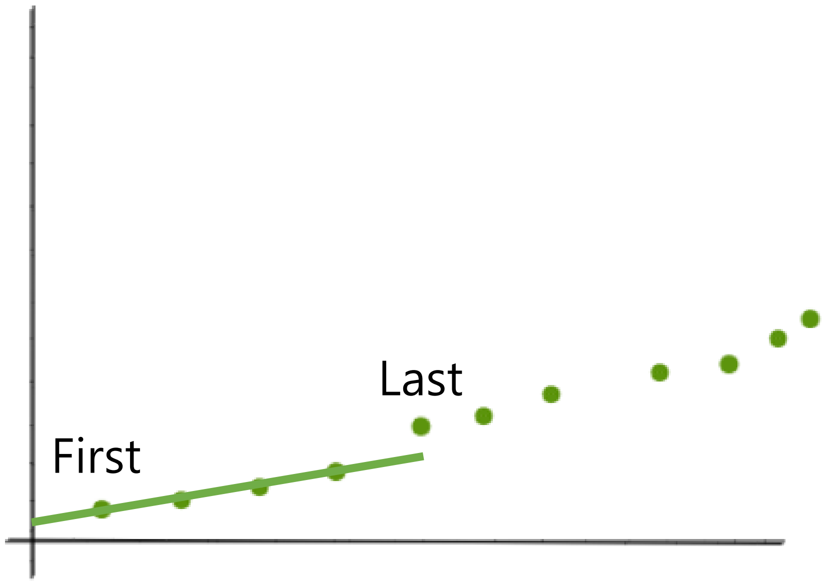
Variance :

$$dx = x_i - mean_x$$
$$mean_x = mean_x + \frac{dx}{i}$$
$$dx_2 = x_i - mean_x$$
$$m_2 = m_2 + dx \times dx_2$$
$$var = \frac{m_2}{n}$$

Slope = Convariance / Variance * compression_factor

Linear Regression

- Linear Regression Analysis
  - Intercept = y – (slope * x)



Collect x through interval and calculate y based on x.

Covariance = $\frac{1}{n}\Sigma_{i=1}^{n}(\chi_i - mean_x) \times (y_i - mean_y)$

Variance :

$$dx = x_i - mean_x$$
$$mean_x = mean_x + \frac{dx}{i}$$
$$dx_2 = x_i - mean_x$$
$$m_2 = m_2 + dx \times dx_2$$
$$var = \frac{m_2}{n}$$

Intercept = $mean_y * compression\_factor - slope * mean_x$

Linear Regression

- Linear Regression Analysis
  - There are some issues with this linear regression model

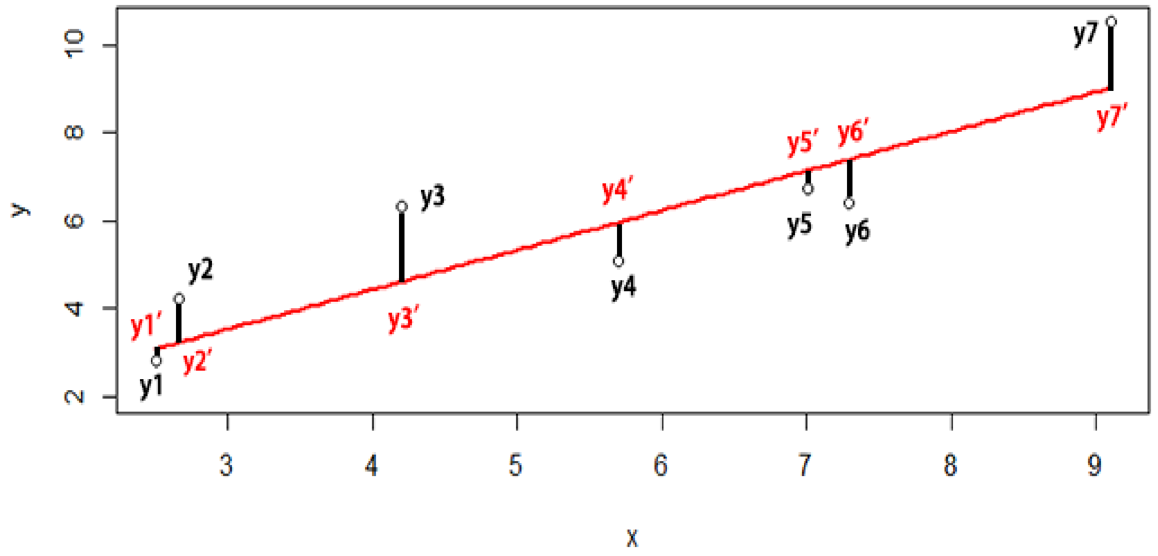
```
for (std::size_t i = 0; i != n; ++i) {
    auto x = *(first + i);
    std::size_t y = offset + i;

    double dx = x - mean_x;
    mean_x += dx /  (i + 1);
    mean_y += (y - mean_y) / (i + 1);
    c += dx * (y - mean_y);

    double dx2 = x - mean_x;
    m2 += dx * dx2;
}
```

Involving too many iterative calculations

Excessive non independent data

Linear Regression

- Linear Regression Analysis
  - Need additional linear regression model (linear least squares)



Linear Least Squares:

Advantage :

Linear regression model

No data dependency

Unified data type

SIMD Friendly

Essence :

Find a line that minimize the sum of all $|y' - y|^2$

Linear Regression

- Linear Regression Analysis
  - Need additional linear regression model (linear least squares)

Formulations for Linear Regression [edit]

The three main linear least squares formulations are:

- *Ordinary least squares* (OLS) is the most common estimator. OLS estimates are commonly used to analyze both experimental and observational data.

  The OLS method minimizes the sum of squared residuals, and leads to a closed-form expression for the estimated value of the unknown parameter vector β:

  $$\hat{\beta} = (\mathbf{X}^{\mathsf{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{y},$$

  where $\mathbf{y}$ is a vector whose ith element is the ith observation of the dependent variable, and $\mathbf{X}$ is a matrix whose ij element is the ith observation of the jth independent variable. The estimator is unbiased and consistent if the errors have finite variance and are uncorrelated with the regressors:[2]

  $$\mathrm{E}[\,\mathbf{x}_i\varepsilon_i\,] = 0,$$

  where $\mathbf{x}_i$ is the transpose of row i of the matrix $\mathbf{X}$. It is also efficient under the assumption that the errors have finite variance and are homoscedastic, meaning that $E[\varepsilon_i^2|x_i]$ does not depend on i. The condition that the errors are uncorrelated with the regressors will generally be satisfied in an experiment, but in the case of observational data, it is difficult to exclude the possibility of an omitted covariate z that is related to both the observed covariates and the response variable. The existence of such a covariate will generally lead to a correlation between the regressors and the response variable, and hence to an inconsistent estimator of β. The condition of homoscedasticity can fail with either experimental or observational data. If the goal is either inference or predictive modeling, the performance of OLS estimates can be poor if multicollinearity is present, unless the sample size is large.
- *Weighted least squares* (WLS) are used when heteroscedasticity is present in the error terms of the model.
- *Generalized least squares* (GLS) is an extension of the OLS method, that allows efficient estimation of β when either heteroscedasticity, or correlations, or both are present among the error terms of the model, as long as the form of heteroscedasticity and correlation is known independently of the data. To handle heteroscedasticity when the error terms are uncorrelated with each other, GLS minimizes a weighted analogue to the sum of squared residuals from OLS regression, where the weight for the i^{th} case is inversely proportional to $var(\varepsilon_i)$. This special case of GLS is called "weighted least squares". The GLS solution to an estimation problem is

  $$\hat{\beta} = (\mathbf{X}^{\mathsf{T}}\boldsymbol{\Omega}^{-1}\mathbf{X})^{-1}\mathbf{X}^{\mathsf{T}}\boldsymbol{\Omega}^{-1}\mathbf{y},$$

  where Ω is the covariance matrix of the errors. GLS can be viewed as applying a linear transformation to the data so that the assumptions of OLS are met for the transformed data. For GLS to be applied, the covariance structure of the errors must be known up to a multiplicative constant.

Ordinary least squares (OLS)

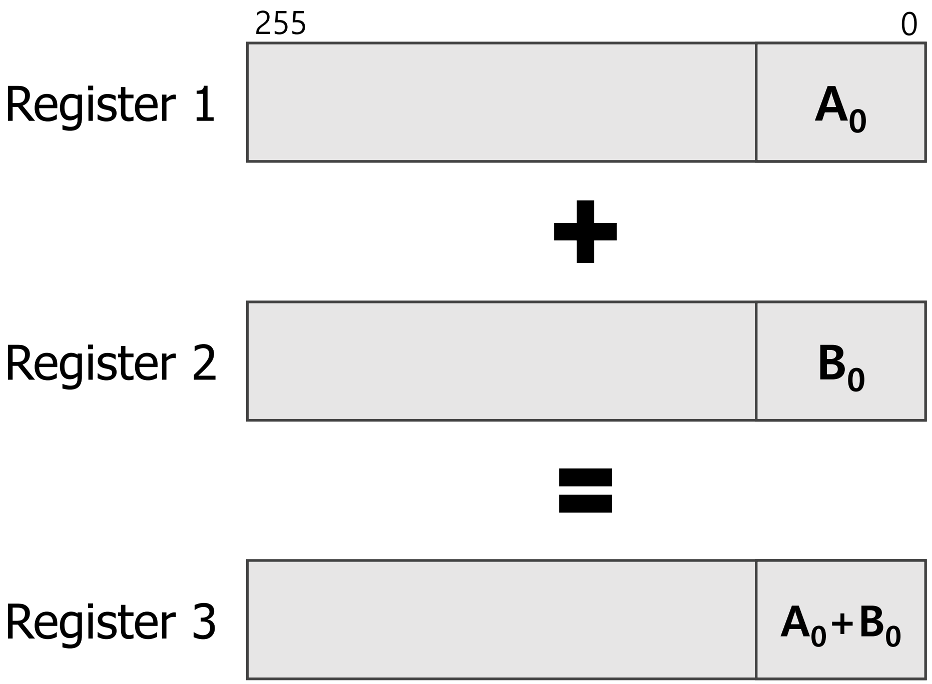
Weighted least squares (WLS)

Generalized least squares (GLS)

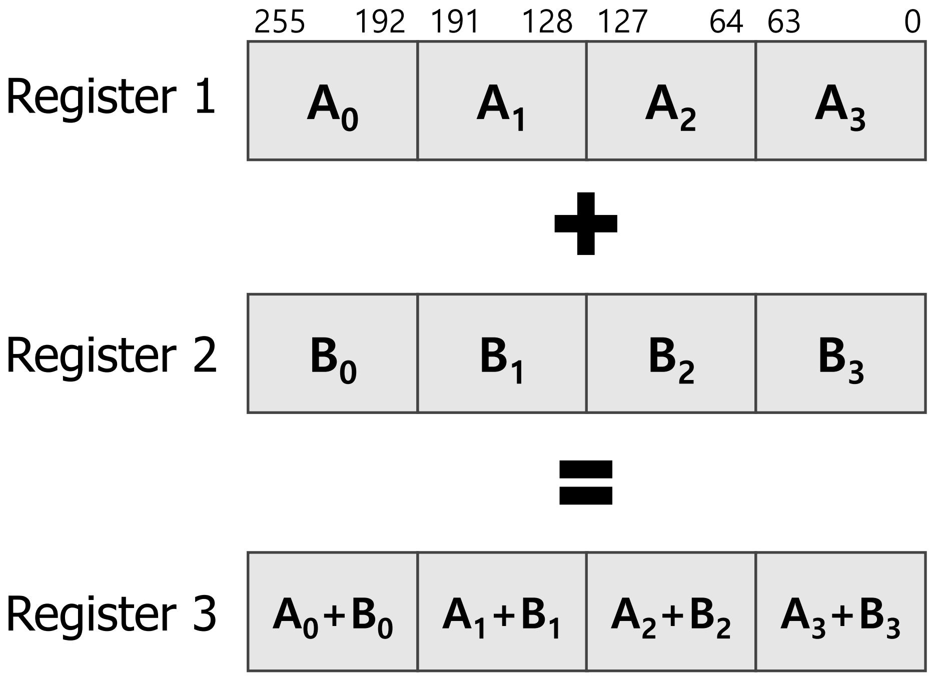
What is SIMD?

X_n : 64bit

- SISD
  - Single Instruction Single Data



- SIMD
  - Single Instruction Multiple Data



Intel Extensions for SIMD

- Instruction set
  - MMX(Pentium), SSE(Pentium III), AVX(Sandy Bridge), AVX2(Haswell), AVX-512(Sky-lake), …

- Packed Data type

Data type	Bits	Description
__m128	128	32bit float X 4
__m128d	128	64bit double X 2
__m128i	128	8,16,32,64bit Integers
__m256	256	32bit float X 8
__m256d	256	64bit double X 4
__m256i	256	8,16,32,64bit Integers
__m512	512	32bit float X 16
__m512d	512	64bit double X 8
__m512i	512	8,16,32,64bit Integers

- Register

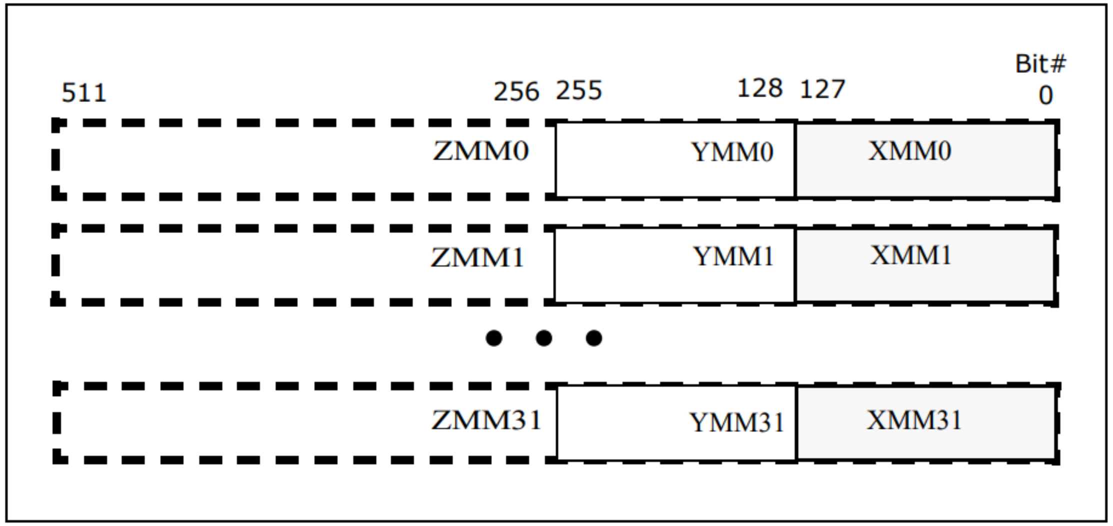


Figure 15-1. 512-Bit Wide Vectors and SIMD Register Set

SISD VS SIMD

- SISD

```
void add_sisd(int *a, int *b, int *dest){
    dest[0] = a[0] + b[0];
    dest[1] = a[1] + b[1];
    dest[2] = a[2] + b[2];
    dest[3] = a[3] + b[3];
    dest[4] = a[4] + b[4];
    dest[5] = a[5] + b[5];
    dest[6] = a[6] + b[6];
    dest[7] = a[7] + b[7];
}
```

- SIMD

```
void add_simd(int *inA, int *inB, int *dest){
    __m256i a = _mm256_load_si256((__m256i*)(inA));
    __m256i b = _mm256_load_si256((__m256i*)(inB));
    __m256i c = _mm256_add_epi32(a, b);
    _mm256_store_si256((__m256i*)(dest), c);

}
```

SISD VS SIMD

- SISD

```
<+0>:     endbr64
<+4>:     mov    (%rsi),%eax
<+6>:     add    (%rdi),%eax
<+8>:     mov    %eax,(%rdx)
<+10>:    mov    0x4(%rsi),%eax
<+13>:    add    0x4(%rdi),%eax
<+16>:    mov    %eax,0x4(%rdx)
<+19>:    mov    0x8(%rsi),%eax
<+22>:    add    0x8(%rdi),%eax
<+25>:    mov    %eax,0x8(%rdx)
<+28>:    mov    0xc(%rsi),%eax
<+31>:    add    0xc(%rdi),%eax
<+34>:    mov    %eax,0xc(%rdx)
<+37>:    mov    0x10(%rsi),%eax
<+40>:    add    0x10(%rdi),%eax
<+43>:    mov    %eax,0x10(%rdx)
<+46>:    mov    0x14(%rsi),%eax
<+49>:    add    0x14(%rdi),%eax
<+52>:    mov    %eax,0x14(%rdx)
<+55>:    mov    0x18(%rsi),%eax
<+58>:    add    0x18(%rdi),%eax
<+61>:    mov    %eax,0x18(%rdx)
<+64>:    mov    0x1c(%rsi),%eax
<+67>:    add    0x1c(%rdi),%eax
<+70>:    mov    %eax,0x1c(%rdx)
<+73>:    retq
```

```
void add est){
    dest
    dest
    dest
    dest
    dest
    dest
    dest
    dest
}
```

- SIMD

```
<+0>:     endbr64
<+4>:     vmovdqa (%rsi),%ymm0
<+8>:     vpaddd (%rdi),%ymm0,%ymm0
<+12>:    vmovdqa %ymm0,(%rdx)
<+16>:    retq
```

```
void a
    ));
    ));
    _mm256_store_si256((__m256i*)(dest), c);
}
```

Why SIMD?

- Advantage
  - **Load** several data at once
    - Less time than retrieving each value individually
  - **Operate** serval data at once
    - The SIMD system works by loading up multiple data points at once
    - The operation being applied to the data

  ➔ Reduce the cost of Operations (Overhead per Element or Operations)

- Disadvantage
  - Require vectorization
  - Require human labor
  - Hardware-dependent

SIMD Experiments

- Compare Array addition performance

- add_sisd()

```
void add_sisd(int *a, int *b, int *dest){
    for (int i=0; i<SIZE; i++){
        dest[i] = a[i] + b[i];
    }
}
```

- add_sisd_unroll()

```
void add_sisd_unroll(int *a, int *b, int *dest){
    for (int i=0; i<SIZE; i+=8){
        dest[i] = a[i] + b[i];
        dest[i+1] = a[i+1] + b[i+1];
        dest[i+2] = a[i+2] + b[i+2];
        dest[i+3] = a[i+3] + b[i+3];
        dest[i+4] = a[i+4] + b[i+4];
        dest[i+5] = a[i+5] + b[i+5];
        dest[i+6] = a[i+6] + b[i+6];
        dest[i+7] = a[i+7] + b[i+7];
    }
}
```

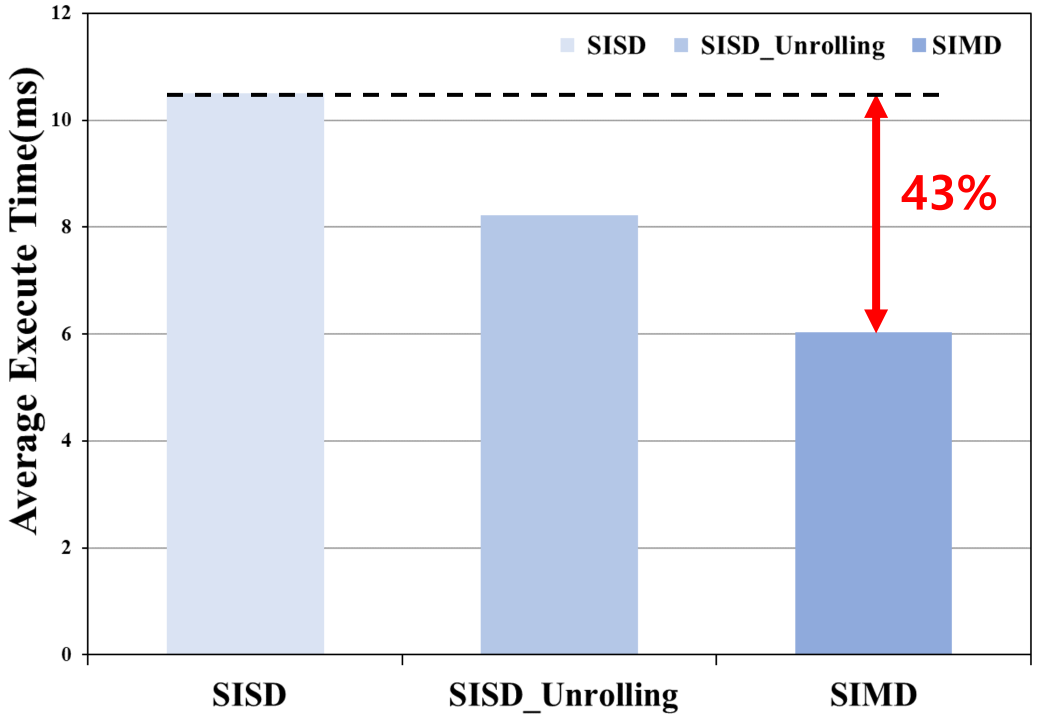
- add_simd()

  - Intel AVX2 Instruction Set

```
void add_simd(int *inA, int *inB, int *dest){
    for (int i = 0; i < SIZE; i+=8) {
        __m256i a = _mm256_load_si256((__m256i*)(inA + i));
        __m256i b = _mm256_load_si256((__m256i*)(inB + i));
        __m256i c = _mm256_add_epi32(a, b);
        _mm256_store_si256((__m256i*)(dest + i), c);
    }
}
```

SIMD Experiments

- Compare Array addition performance
  - Addition between arrays with 5 million elements



➔ 43% performance improvement on SIMD compared to SISD

Future work

- Choosing a SIMD-friendly linear regression technique
- Implement the linear regression technique
- Implement Code with SIMD

Thank you