From Scalar Compute to Vector Compute

Song Liu (song.liu@bristol.ac.uk)

GA 18, Fry Building,

Microsoft Teams (search "song liu").

Scalar Compute in R is SLOW

- R is very similar to C when it comes to scalar computing (the operand is a single datum).
  - Similar syntax, with minor differences.
- However, we have seen in the previous lab that **scalar compute is very slow in R**.
  - Particularly so when **loops** are involved.
- Is R really slow then?

Matrix Multiplication in R

- `%*%` is matrix multiplication operator in R.
  - `A %*% B` computes matrix multiplication between matrix `A` and `B`,
- Let us multiply two 1000 by 1000 matrices in R:

```
# generating two 1000X1000 random matrices
A <- matrix(rnorm(1000*1000), nrow = 1000, byrow = T)
B <- matrix(rnorm(1000*1000), nrow = 1000, byrow = T)

# built-in matrix multiplication
start = Sys.time()
C <- A%*%B
end = Sys.time()

print(end - start)
```

Matrix Multiplication in R

- It takes R 1 second to do the above multiplication.
- It takes C 5.7 second to do the same multiplication.
    - Using the code we wrote in lab 7.
- Why does R outperform C in matrix multiplication?

Single Instruction Single Data

- When C was born (1970s), CPU designs were very primitive.
- CPU can only process **a handful numbers at a time**.
    - One **single** CPU instruction can only work on a **single** data set.
    - For example, an `add` instruction can only add two numbers at a time.
    - This processing paradigm is called **Single Instruction Single Data (SISD)**.
- No doubt, SISD is inefficient in data science programs with large data sets.

Single Instruction Multiple Data

- However, modern consumer CPUs (starting from 1990s) support a feature called Single Instruction Multiple Data (SIMD).
  - One single CPU instruction can operate on a large number of data sets.
  - For example, a single SIMD `add` instruction can add multiple pairs of number at the same time.
  - Popular SIMD instruction sets include MMX, SSE, SSE2, AVX2, AVX512.
- SIMD makes CPUs suitable for vector compute.
- R (and many other languages that appeared later than C) supports vector compute natively via SIMD.

Vector Compute

- When R interpreter executes `A%*%B`, it automatically **translates your code into SIMD instructions** then execute it on CPU efficiently.
- By default, C compiler does **not** translate your code into more efficient SIMD instructions.
- Thus, R is faster in the matrix multiplication example.

Vector Compute

- In fact, R is a kind of `array programming` language, that are designed for processing large vector/matrix.
- Accelerated vector compute makes R suitable for statistics, machine learning and many other scientific and engineering applications.

To Sum Up

- Scalar Compute: C > R.
- Vector Compute: R > C.
    - C can support vector compute via other extension libraries, such as Intel's Math Kernel Library (MKL), or NVIDIA's Compute Unified Device Architecture (CUDA).

Elementwise Vector Operators

- You can find reference to most of these from ART 2.4
- Vector Addition adds each dimension of vectors.

```
a <- c(1,2,3,4)
b <- c(1,2,3,4)
c <- a+b
[1] 2 4 6 8
```

- Vector Subtraction `-`, Multiplication `*`, Division `/` works the same way.
    - Do not confuse `*` with `%*%`, `*` is dimension-wise multiplication.

Elementwise Vector Operators

- Logical operations are performed on each dimension of the vector.
  - ```
    a <- c(1,2,3,4)
    print(a > 2)
    [1] FALSE FALSE TRUE TRUE
    ```
- Many math functions operate on each element of the input vector (`sin,cos,exp,log`):
  - ```
    exp(c(1,2,3,4))
    [1]  2.718282  7.389056 20.085537 54.598150
    ```
- This behavior is called "vector in, vector out".

Indexing Element(s) in a Vector

- Use `[]` to index a single element
  - The index of the first element is 1 not 0!
  - ```
    a <- c(1,2,3,4)
    print(a[1])
    [1] 1
    print(a[5])
    [1] NA
    ```
  - Index the element that beyond the range of the vector will return `NA`.
    ```

Indexing Element(s) in a Vector

- You can index more than one elements at a time, simply by using another vector as indices:

```
a <- c(1,2,3,4)
b <- c(1,3,4)
print(a[b])
[1] 1 3 4
#or simply
print(a[c(1,3,4)])
[1] 1 3 4
```

- You can also use conditional expression to index a vector:

```
a <- c(1,2,3,4)
print(a[a>2])
[1] 3 4
```

`:` Symbol

- `:` symbol generates a vector of a range of values.

  ```
  a <- 1:3
  print(a)
  [1] 1 2 3
  ```

- Use `:` to access multiple contiguous elements in a vector.

  ```
  a <- c(5,6,7,8)
  print(a[1:3])
  [1] 5 6 7
  ```

`for` Loop

- Syntax `for (variable in vector)`
- Two for loops below are the same.
- ```
  for (element in a){
      print(element)
  }
  ```
- ```
  for (i in 1:length(a)){
      print(a[i])
  }
  ```

Matrix Construction

- You can create a matrix from a vector using `matrix` function:

```
a <- c(1,2,3,4)
A <- matrix(a, nrow = 2)
print(A)
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

- `nrow` parameter specifies the number of rows in the matrix.

```
a <- c(1,2,3,4)
A <- matrix(a, nrow = 1)
print(A)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
```

Matrix Construction

- `matrix` will always fill out the matrix by columns. If you want to fill out the matrix by rows:

```
A <- matrix(1:4, nrow = 2, byrow = T)
print(A)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

- You can get dimension of a matrix using `dim` function.

```
A <- matrix(c(1,2,3,4), nrow = 2, byrow = T)
print(dim(A))
[1] 2 2
print(dim(A)[1]) # the number of rows of A
[1] 2
print(dim(A)[2]) # the number of cols of A
[1] 2
```

Indexing Elements in a Matrix

- Elements in matrices can be indexed by `[,]`.

```
A <- matrix(1:4, nrow = 2)
print(A[1,1])
[1] 1
print(A[1:2,1])
[1] 1 2
```

- You can access a `i` -th row of a matrix by using `[i,]`

```
A <- matrix(1:4, nrow = 2)
> print(A)
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> print(A[1,])
[1] 1 3
```

- You can access a `i` -th column of a matrix by using `[,i]`

Matrix Operators

- `+, -, *, /` works on matrix in element-wise fashion too.

```
A <- matrix(1:4, nrow = 2, byrow = T)
B <- matrix(1:4, nrow = 2, byrow = T)
print(A+B)
     [,1] [,2]
[1,]    2    4
[2,]    6    8
```

- `%*%` works as matrix multiplication

```
print(A%*%B)
     [,1] [,2]
[1,]    7   10
[2,]   15   22
```

Matrix Transposition

- Matrix Transpose is done by `t` function.

```
a <- matrix(1:6, nrow = 2)
print(a)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
t(a)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Matrix Inversion

- Matrix Inversion: `solve` function.

```
A <- matrix(1:4, nrow = 2)
print(A%*%solve(A))
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Conclusion

- **R uses SIMD instructions to accelerate vector compute** while C does not by default.
- Scalar Compute: C is faster than R.
- Vector Compute: R is faster than C.

Homework

1. Familiarize with commands that are listed in the slides.

Homework

2. Write an R function `dist`, takes two vectors: `a` and `b` representing two points in a space. `dist` outputs the euclidean distance between two points.
    - Your function should work for input vectors in **any dimension**.
    - `dist(v,0)` will return the length of vector `v`.
    - **No for-loops**.
    - Make test cases, and check your function.
    - Hint: type, `?sum` and `?sqrt` in the command line and read the manual.

Homework (submit)

3. Write an R function `angle`, takes two vectors: `a` and `b`. It outputs the angle between `a` and `b` in degree.
    - The result should be returned **in degree**, not in radian.
    - Use `dist` function you just wrote.
    - Make test cases, and check your function.
    - Hint: Type `?acos` in command line and read the documentation.