



Project for EECE 5640

Yaming Zhang

HPC performance comparison and analysis between C, Python, Golang and Julia



Main tasks

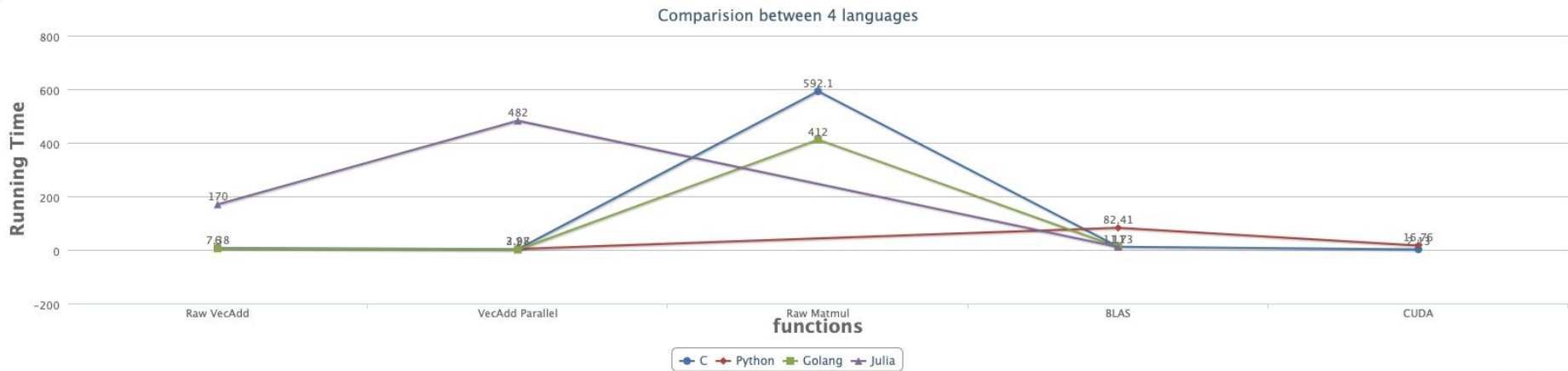
1. Implementing vector addition and matrix multiplication on all 4 languages
2. Compare the performance with multithreading
3. Using popular matrix multiplication package BLAS and GEMM function
4. Try to compare the GPU performance of multiplication(C && Python only)
5. All the benchmarks are done by 1,000,000 items vector, 10 threads and $512 * 512$ matrix



Main Results

Functions	C	Python	Golang	Julia
Raw VecAdd	7.38	5460.58	6	170
VecAdd Parallel	2.98	3.27	1	482
Raw Matmul	592.1	972034.18	412	22297
BLAS	11.73	82.41	11	11
CUDA	2.13	16.76		

Main Results





C VECTOR ADD

```
[zhang.yam@d1004 C]$ ./main
C VECTOR: 1.000000
C VECTOR: 7.378515 ms
C VECTOR PTHREADS: 1.000000
C VECTOR PTHREADS: 3.808991 ms
C VECTOR OMP: 1.000000
C VECTOR OMP: 2.978213 ms
```

Brute Force	7.378515 ms
Pthread	3.808991 ms
OpenMP	2.978213 ms




C MATRIX MULTIPLICATION

```
C MATMUL: 4.44488e+07
C MATMUL: 592.098306 ms
C MATMUL OMP: 4.44488e+07
C MATMUL OMP: 451.520895 ms
C MATMUL BLAS: 4.44488e+07
C MATMUL BLAS: 11.729042 ms
[zhang.yam@d1004 C]$ ./maincuda
C MATMUL CUDA: 4.44488e+07
C MATMUL CUDA: 2.134546 ms
```

Brute Force	592.098306 ms
OpenMP + block	451.520895 ms
BLAS.GEMM	11.729042 ms
CUDA	2.134546 ms

PYTHON VECTOR ADD



```
PYTHON VECTOR: 1.0
PYTHON VECTOR: 5460.586515 ms
PYTHON VECTOR ADD THREADING: 1.0
PYTHON VECTOR ADD THREADING: 5697.480051 ms
PYTHON VECTOR PYTORCH: 1.0
PYTHON VECTOR PYTORCH: 3.268715 ms
```

Brute Force	5460.586515 ms
MultiThreading	5697.480051 ms
Torch	3.268715 ms



Why is Python multithreading slower?

This limitation is actually enforced by GIL. The **Python Global Interpreter Lock** (GIL) prevents threads within the same process to be executed at the same time.

GIL, is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once — [Python Wiki](#)

The GIL is necessary because Python's interpreter is **not thread-safe**. This global lock is enforced every time we attempt to access Python objects within threads. At any given time, only one thread can acquire the lock for a specific object. Therefore, CPU-bound code will have no performance gain with Python multi-threading.



Multiprocessing?

`multiprocessing` is a package that supports spawning **processes** using an API similar to the `threading` module.

When you use multiprocessing to open a second process, an *entirely new instance* of Python, with its own global state, is created. That global state is not shared, so changes made by child processes to global variables will be invisible to the parent process.



PYTHON MATRIX MULTIPLICATION

```
PYTHON MATMUL: 44448768.0  
PYTHON MATMUL: 972034.182517 ms  
PYTHON MATMUL TORCH: 44448768.0  
PYTHON MATMUL TORCH: 82.407801 ms  
Using CUDA Right Now...  
PYTHON MATMUL CUDA: 44448768.0  
PYTHON MATMUL CUDA: 16.757489 ms
```

Brute Force	972034.182517 ms
Torch	82.407801 ms
CUDA	16.757489 ms



Why does PyTorch or Numpy matmul so fast?

Docs > torch



BLAS and LAPACK Operations

`addbmm`

Performs a batch matrix-matrix product of matrices stored in `batch1` and `batch2`, with a reduced add step (all matrix multiplications get accumulated along the first dimension).

`addmm`

Performs a matrix multiplication of the matrices `mat1` and `mat2`.

GoLang: Goroutine vs Thread



Goroutine	Thread
Goroutines are managed by the go runtime.	Operating system threads are managed by kernel.
Goroutine are not hardware dependent.	Threads are hardware dependent.
Due to the presence of channel one goroutine can communicate with other goroutine with low latency.	Due to lack of easy communication medium inter-threads communicate takes place with high latency.
Goroutine does not have ID because go does not have Thread Local Storage.	Threads have their own unique ID because they have Thread Local Storage.
Goroutines are cheaper than threads.	The cost of threads are higher than goroutine.



GOLANG VECTOR ADD

```
GOLANG VECTOR: 1.000000  
GOLANG VECTOR: 6 ms  
GOLANG VECTOR GOROUTINE: 1.000000  
GOLANG VECTOR GOROUTINE: 1 ms
```

Brute Force	6 ms
Goroutine	1 ms

GOLANG MATMUL



```
GOLANG MATMUL: 44448768.000000  
GOLANG MATMUL: 412 ms  
GOLANG MATMUL GOROUTNE ROW: 44448768.000000  
GOLANG MATMUL GOROUTNE ROW: 412 ms  
GOLANG MATMUL GOROUTNE POINT: 44448768.000000  
GOLANG MATMUL GOROUTNE POINT: 575 ms
```

Brute Force	412 ms
Goroutine Row	412 ms
Goroutine Point	575 ms


GOLANG MATMUL



```
GOLANG MATMUL: 44448768.000000  
GOLANG MATMUL: 288 ms  
GOLANG MATMUL GOROUTNE ROW: 44448768.000000  
GOLANG MATMUL GOROUTNE ROW: 65 ms  
GOLANG MATMUL GOROUTNE POINT: 44448768.000000  
GOLANG MATMUL GOROUTNE POINT: 219 ms  
GOLANG MATMUL BLAS: 44448768.000000  
GOLANG MATMUL BLAS: 11 ms
```

Brute Force	288 ms
Goroutine Row	65 ms
Goroutine Point	219 ms
BLAS	11 ms

Take a glance at BLAS GEMM code in Golang



```
// dgemmParallel computes a parallel matrix multiplication by partitioning
// a and b into sub-blocks, and updating c with the multiplication of the sub-block
// In all cases,
// A = [ A_11  A_12 ... A_1j
//       A_21  A_22 ... A_2j
//       ...
//       A_i1  A_i2 ... A_ij]
//
// and same for B. All of the submatrix sizes are blockSize×blockSize except
// at the edges.
//
// In all cases, there is one dimension for each matrix along which
// C must be updated sequentially.
// Cij = \sum_k Aik Bki, (A * B)
// Cij = \sum_k Aki Bkj, (A^T * B)
// Cij = \sum_k Aik Bjk, (A * B^T)
// Cij = \sum_k Aki Bjk, (A^T * B^T)
//
// This code computes one {i, j} block sequentially along the k dimension,
// and computes all of the {i, j} blocks concurrently. This
// partitioning allows Cij to be updated in-place without race-conditions.
// Instead of launching a goroutine for each possible concurrent computation,
// a number of worker goroutines are created and channels are used to pass
// available and completed cases.
```


Take a glance at BLAS GEMM code in Golang

```
// wg is used to wait for all
var wg sync.WaitGroup
wg.Add(parBlocks)
defer wg.Wait()

for i := 0; i < m; i += blockSize {
    for j := 0; j < n; j += blockSize {
        workerLimit <- struct{}{}
        go func(i, j int) {
            defer func() {
                wg.Done()
                <-workerLimit
            }()

            leni := blockSize
            if i+leni > m {
                leni = m - i
            }
            lenj := blockSize
            if j+lenj > n {
                lenj = n - j
            }

            cSub := sliceView64(c, ldc, i, j, leni, lenj)

            // Compute A_ik B_kj for all k
```

Take a glance at BLAS GEMM code in Golang



On a computer with a hierarchical memory the blocked form can be much more efficient than the point form if the blocks fit into the high speed memory, as much less data transfer is required.

So BLAS takes advantages of blocked algorithms and parallelism

JULIA VECTOR ADD

```
[zhang.yam@d1027 JULIA]$ JULIA_NUM_THREADS=2 julia main.jl
JULIA VECTOR: 1.0000003330100622
JULIA VECTOR: 170 milliseconds
JULIA VECTOR threads: 1.0000003330100622
JULIA VECTOR threads: 245 milliseconds
```

```
[zhang.yam@d1027 JULIA]$ JULIA_NUM_THREADS=8 julia main.jl
JULIA VECTOR: 1.0000003330100622
JULIA VECTOR: 172 milliseconds
JULIA VECTOR threads: 1.0000003330100622
JULIA VECTOR threads: 482 milliseconds
```

```
[zhang.yam@d1027 JULIA]$ JULIA_NUM_THREADS=16 julia main.jl
JULIA VECTOR: 1.0000003330100622
JULIA VECTOR: 159 milliseconds
JULIA VECTOR threads: 1.0000003330100622
JULIA VECTOR threads: 534 milliseconds
```

# of threads	time
2	245 ms
8	482 ms
16	534 ms

JULIA Matmul



```
JULIA MATMUL: 4.4448768e7  
JULIA MATMUL: 22297 milliseconds  
JULIA MATMUL LA: 4.4448768e7  
JULIA MATMUL LA: 633 milliseconds  
JULIA MATMUL BLAS: 4.4448768e7  
JULIA MATMUL BLAS: 11 milliseconds
```

Brute Force	22297 ms
LA pkg	633 ms
BLAS	11 ms

Other functions in JULIA



There are still many other functions in Julia that could help us explore more on High Performance Computing, like `MPI.jl`, `CUDA.jl` and so on.

Conclusion



C and Golang both have extreme performance on high performance computing and normal running time. And Golang even runs faster than C thanks to the lightweight goroutines of Golang. That is maybe the reason why Golang has been so popular these years.

Python has the worst performance among all of the 4 languages. Even with CUDA and GPU, the running time of matrix multiplication is slower than CUDA in C. But thanks to Python's package Pytorch, we could use GPUs easily on the cluster and do difficult tensor processing of machine learning and deep learning.

The running time of using BLAS library is similar among C, Golang and Julia. That's mainly because the implementation of BLAS is similar among all the languages with blocked algorithms and parallelism to increase cache locality.

Julia's performance is not so impressive on basic vector addition and matrix multiplication but Julia does have a variety of packages that support HPC, like BLAS, LAPACK, MPI.jl, CUDA.jl and so on. There's more than we could imagine to explore more into Julia.



Thank You!