RISC-V Parallel Programming for Signal and Image Processing

This book and presented **labs** are devoted to exploring how modern RISC-V processors can provide an effective solution for parallel programming and high-performance processing using Python and optimized scientific libraries.

RISC-V represents a significant shift in the field of processor design. Unlike traditional proprietary instruction set architectures, RISC-V is open, modular, and freely available. This openness has encouraged rapid innovation in universities, research laboratories, and industry alike. Designers are free to implement, extend, and optimize the architecture according to the needs of specific applications. The clean and well-structured design of RISC-V also makes it particularly suitable for both high-performance computing platforms and modern embedded systems. As a result, it has become one of the most promising foundations for the next generation of computing technologies.

A central theme of this book is **performance through parallelism**. Modern RISC-V processors combine **multicore** architectures with powerful **vector-processing** capabilities. Multicore systems improve performance by executing multiple tasks simultaneously, while the RISC-V vector extension enables a single instruction to operate on multiple data elements at once. This form of data-level parallelism is especially well suited to applications in signal and image processing, where the same operations are repeatedly applied to large data sets. By combining multicore execution with vector operations, RISC-V offers a scalable and energy-efficient computing model capable of addressing demanding real-world problems.

Another important aspect of this book is the role of **Python** as a high-level development environment. Python has become one of the most widely used languages in scientific and engineering applications thanks to its clear syntax, readability, and extensive ecosystem of libraries.

Optimized libraries such as **NumPy**, **SciPy**, and OpenCV allow developers to implement complex algorithms quickly and with relatively little code. Although Python itself is a high-level language, many of its scientific libraries rely on highly optimized low-level implementations. This makes it possible to combine the productivity of Python with the performance of optimized numerical routines, including those that take advantage of vector instructions and other hardware-acceleration features available on modern RISC-V processors.

This book aims to bridge the gap between architectural concepts and **practical algorithm development**. It explains how computationally intensive operations can be implemented efficiently while remaining accessible to readers who may not have extensive experience in low-level programming. Through a series of practical examples, the book demonstrates how modern RISC-V processors and high-level programming tools can work together to create efficient, portable, and scalable solutions.

Ultimately, this book is intended for students, researchers, and engineers who are interested in high-performance computing, signal processing, image processing, and modern processor architectures.

Whether the reader is approaching RISC-V for the first time or seeking to deepen their understanding of parallel computing techniques, the goal of this work is to provide a clear and practical path toward efficient algorithm design on one of the most promising computing platforms of our time

Table of Contents

Lab 1 : SIMD - Basic vector programming and processing

Lab 1 : SIMD

Basic vector programming and processing

In this first programming and processing lab, we will write and run several **very simple examples** that are well suited for vector (SIMD) processing.
Each studied example is composed of **two parts**:

- A **serial implementation** program with the function under test, written in simple Python, and
- The same function **reimplemented** with **`NumPy`** library using the **RISC-V vecto**r extensions.

In this lab we focus on simple arithmetic functions such as **vector addition**, **matrix multiplication**, and the **scalar (`dot`) product**. These exercises introduce the basic principles and functionality of vector programming and processing. We will use different data types, including bytes (**`int8_t`**), integers (**`int32_t`**), and floating-point types.
After **assembling and linking the programs**, we obtain **executable code** that can be tested directly on our development board.
This enables us to **compare the execution times of scalar and vector implementations** and to evaluate the resulting **speedup**, which depends on both the data type and the vector/matrix size.
For each example, we also explain the role and meaning of the vector instructions used in the program. In this context, it is assumed that the reader is already familiar with the basic constructs of the C programming language.

1.1 Adding two vectors

The simplest starting example is the **addition of two vectors**. Since the **X60** processors on the **K1 SoC** integrate a vector processing unit with a register block consisting of **32 registers, each 256 bits** wide, we can easily determine the level of parallelism.

- **4 elements** for **64-bit** data (**double** words: **long integers** or **double**-precision floats).
- **8 elements** for **32-bit** data (**words**: **integers** or single-precision **floats**),
- **32 elements** for **8-bit** data (**bytes**),

1.1.1 Serial execution (interpretation): `addser_float64.py` (default)

```
-------------------------------------------------------------------------------# #
#addser_float64.py
import numpy as np
import time
N = 10_000_000
a = np.random.rand(N)
b = np.random.rand(N)
t0 = time.time()
# Serial addition using Python loop
c = [a[i] + b[i] for i in range(N)]
t1 = time.time()
elapsed = t1 - t0
print("Time:", elapsed)
# ------------------------------------------------
# Performance estimation in GFLOPs
# ------------------------------------------------
# Each iteration performs 1 floating-point addition
```

```
flops_per_element = 1
total_flops = N * flops_per_element
gflops = total_flops / elapsed / 1e9
print("Estimated performance:", gflops, "GFLOPs")
```

```
$ python3 add_vect_perf.py
Time: 24.109723806381226
Estimated performance: 0.0004147704088320272 GFLOPs
```

Code explanation:

Importing Libraries

```
import numpy as np  # used to create numerical arrays
import time         # used to measure execution time
```

Define Problem Size

```
N = 10_000_000   #10 million elements
```

Create Two Arrays

```
a = np.random.rand(N); b = np.random.rand(N)
```

`np.random.rand(N)` creates:

- a **NumPy** array of size N
- values between 0 and 1
- type: `float64` (by default)

Start Timing (in seconds)

```
t0 = time.time()
```

Element-wise Addition

```
c = [a[i] + b[i] for i in range(N)
```

What happens:

- Python loops from `0` to `N-1`
- For each `i`: Fetch `a[i],` Fetch `b[i],` Add them, Store result in a Python list: `c`
  for each `i`: interpreter executes addition (serial execution!) – **no RVV**
  - Python interpreter executes each iteration

Print Execution Time

```
print("Time:", time.time() - t0)
```

How the **GFLOPs** estimate is computed

For each element:

- one floating-point operation : `a[i] + b[i]`

So the performance is estimated as :

$$\text{GFLOPs} = \frac{N}{\text{execution time} \times 10^9}$$

The following examples execute the same program on **different data types** :

```
# addser_int32.py
import numpy as np
import time
N = 10_000_000
# Create two int32 arrays
a = np.random.randint(0, 1000, size=N, dtype=np.int32)
b = np.random.randint(0, 1000, size=N, dtype=np.int32)
t0 = time.time()
# Serial element-by-element addition (no vectorization)
c = [a[i] + b[i] for i in range(N)]
t1 = time.time()
elapsed = t1 - t0
print("Time:", elapsed)
# ------------------------------------------------
# Performance estimation (integer operations)
# ------------------------------------------------
# Each iteration performs 1 integer addition
ops_per_element = 1
total_ops = N * ops_per_element
gops = total_ops / elapsed / 1e9
print("Estimated performance:", gops, "GOPS")
```

```
$ python3 add_vect_int32_perf.py
Time: 22.230340719223022
Estimated performance: 0.00044983566047428143 GOPS
```

```
# addser_int8.py
import numpy as np
import time
N = 10_000_000
# Create two int8 arrays
a = np.random.randint(-128, 127, size=N, dtype=np.int8)
b = np.random.randint(-128, 127, size=N, dtype=np.int8)
t0 = time.time()
# Serial element-by-element addition
c = [int(a[i]) + int(b[i]) for i in range(N)]
t1 = time.time()
elapsed = t1 - t0
print("Time:", elapsed)
# ------------------------------------------------
# Performance estimation (integer operations)
# ------------------------------------------------
# Each iteration performs 1 integer addition
ops_per_element = 1
total_ops = N * ops_per_element
gops = total_ops / elapsed / 1e9
print("Estimated performance:", gops, "GOPS")
```

```
$ python3 add_vect_int8_perf.py
Time: 29.504695415496826
Estimated performance: 0.0003389291046450754 GOPS
```

Remark:

Note that the execution operating on **`int8`** is slower than on **`int32`** ?

1.1.2 Parallel (RVV) execution

```
-------------------------------------------------------------------------------------
# addvect_float64.py
import numpy as np
import time
size = 10_000_000
a = np.random.rand(size).astype(np.float64)
b = np.random.rand(size).astype(np.float64)
s_time = time.time()
# Vectorized addition
c = a + b
e_time = time.time()
elapsed = e_time - s_time
print(f"Time of {size} : {elapsed:.4f} sec")
# ------------------------------------------------
# Performance estimation in GFLOPs
# ------------------------------------------------
# Each element requires 1 floating-point addition
flops_per_element = 1
total_flops = size * flops_per_element
gflops = total_flops / elapsed / 1e9
print(f"Estimated performance: {gflops:.4f} GFLOPs")
-------------------------------------------------------------------------------------
$ python3 add_vect_mp_float64_perf.py
Time of 10000000 : 0.0776 sec
Estimated performance: 0.1288 GFLOPs
-------------------------------------------------------------------------------------
```

Code explanation:

Importing Libraries

```
import numpy as np  # used to create numerical arrays
import time # used to measure execution time
```

Define Problem Size

```
N = 10_000_000 #10 million elements x8 bytes=80 MB per array
```

Create Two Arrays

```
a = np.random.rand(size).astype(np.float64) ..
b = np.random.rand(size).astype(np.float64)
```

A **`NumPy`** array of size N, Values between 0 and 1, Type: **`float64`** (by default)

Start Timing (in seconds)

```
t0 = time.time()
```

Element-wise Addition

```
c = a + b  # for all 10 million elements;NOT a Python loop
```

`NumPy` performs numerical computations by calling highly optimized C routines, avoiding the use of Python-level loops and therefore **eliminating interpreter overhead**.

The **data are stored in contiguous memory arrays**, which allows efficient access patterns and enables a tight compiled loop to process the elements directly. Because the computation occurs in compiled code, the implementation can exploit SIMD and vector instructions provided by modern processors to accelerate operations. These optimizations can take advantage of the **RVV** (RISC-V Vector Extension) to perform parallel operations on multiple data elements simultaneously, significantly improving numerical performance.

Print Execution Time

```
e_time =time.time()
print(f"Time of {size} : {e_time-s_time:.4} sec")
```

```
# addvect_int32.py
import numpy as np
import time
size = 10_000_000
# Create two int32 arrays
a = np.random.randint(0, 1000, size=size, dtype=np.int32)
b = np.random.randint(0, 1000, size=size, dtype=np.int32)
s_time = time.time()
# Vectorized integer addition
c = a + b
e_time = time.time()
elapsed = e_time - s_time
print(f"Time of {size} : {elapsed:.4f} sec")
# ------------------------------------------------
# Performance estimation (integer operations)
# ------------------------------------------------
# Each element performs 1 integer addition
ops_per_element = 1
total_ops = size * ops_per_element
gops = total_ops / elapsed / 1e9

print(f"Estimated performance: {gops:.4f} GOPS")
```

```
$ python3 add_vect_mp_int32_perf.py
Time of 10000000 : 0.0506 sec
Estimated performance: 0.1977 GOPS
```

```
# addvect_int8.py
import numpy as np
import time
size = 10_000_000
# Create two int8 arrays (range -128 to 127)
a = np.random.randint(-128, 128, size=size, dtype=np.int8)
b = np.random.randint(-128, 128, size=size, dtype=np.int8)
s_time = time.time()

# Vectorized integer addition
c = a + b
e_time = time.time()
elapsed = e_time - s_time
print(f"Time of {size} : {elapsed:.4f} sec")
# ------------------------------------------------
# Performance estimation (integer operations)
# ------------------------------------------------
# Each element performs 1 integer addition
ops_per_element = 1
total_ops = size * ops_per_element
gops = total_ops / elapsed / 1e9
print(f"Estimated performance: {gops:.4f} GOPS")
```

```
$ python3 add_vect_mp_int8_perf.py
Time of 10000000 : 0.0118 sec
Estimated performance: 0.8449 GOPS
```

Remarks:

Note that the execution operating on **`int8`** is much faster (~ 4 times) than on **`int32`** ?

Comparing with serial execution : **31.60** sec is **2633 times faster (?)**

Discuss this result !

To do:

Analyze and execute the following examples on your board.

1.1.3 Parallel execution with optimized C code

Here is a C program that:

- Allocates three vectors **`a, b, c`**
- Each contains **10,000,000 elements**
- Type: **double (64-bit float)**
- Measures the execution time of the addition loop only

```
//  add_vect_float64_perf.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 10000000

int main() {
    // Allocate memory
    double *a = (double*) malloc(SIZE * sizeof(double));
    double *b = (double*) malloc(SIZE * sizeof(double));
    double *c = (double*) malloc(SIZE * sizeof(double));
    if (a == NULL || b == NULL || c == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }
    // Initialize vectors
    for (long i = 0; i < SIZE; i++) {
        a[i] = (double) rand() / RAND_MAX;
        b[i] = (double) rand() / RAND_MAX;
    }
    // Start timing (only addition loop)
    clock_t start = clock();
    for (long i = 0; i < SIZE; i++) {
        c[i] = a[i] + b[i];
    }
    clock_t end = clock();
    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Time for %d double additions: %f seconds\n", SIZE, time_spent);
    // ------------------------------------------------
    // Performance estimation in GFLOPs
    // ------------------------------------------------
    // Each iteration performs 1 floating-point addition
    double flops = (double) SIZE;
    double gflops = flops / (time_spent * 1e9);
    printf("Estimated performance: %f GFLOPs\n", gflops);
    // Simple checksum (avoid optimization removal)
    double checksum = 0.0;
    for (long i = 0; i < SIZE; i++)
        checksum += c[i];
    printf("Checksum: %f\n", checksum);
    free(a);
    free(b);
    free(c);
    return 0;
}
```

Compilation for scalar (serial) execution – no compilation options !

```
gcc -march=rv64gcv -mabi=lp64d add_vect_float64_perf.c -o
add_vect_float64_perf
$ ./add_vect_float64_perf
Time for 10000000 double additions: 0.424914 seconds
Estimated performance: 0.023534 GFLOPs
Checksum: 10000789.032793
```

Compilation for vector (parallel) execution – with compilation options !

```
$ gcc -O2 -march=rv64gcv -mabi=lp64d add_vect_float64_perf.c -o
add_vect_float64_perf
$ ./add_vect_float64_perf
Time for 10000000 double additions: 0.163942 seconds
Estimated performance: 0.060997 GFLOPs
Checksum: 10000789.032793
```

Remark:

For the same computation, the **RVV-optimized `NumPy`** implementation (**0.1288 GFLOPs**) achieves approximately **twice the performance** of the compiler-optimized C version (**0.060997 GFLOPs**).

Notes:

`NumPy` achieves this performance by exploiting parallelism through RVV vector instructions, typically operating with **`LMUL`** set to **`m1`**.

1.2 Average value of vector

The following example calculates the average value for the given vector (set) values.

- **Serial version** → pure Python loop (no vector instructions)
- **Vectorized version** → **`NumPy` reduction** (uses SIMD/RVV)

1.2.1 Serial average

```
-------------------------------------------------------------------------------------------
# average_serial_float64.py
import numpy as np
import time
size = 1_000_000
# Create vector (float64)
a = np.random.rand(size)
start = time.time()
# Serial computation of sum
total = 0.0
for i in range(size):
    total += a[i]

average = total / size
end = time.time()
elapsed = end - start
print(f"Serial average: {average}")
print(f"Time: {elapsed:.6f} sec")
# ------------------------------------------------
# Performance estimation in GFLOPs
# ------------------------------------------------
# Each iteration performs 1 addition for sum
# Division for average counts as 1 FLOP
flops = size + 1
gflops = flops / elapsed / 1e9

print(f"Estimated performance: {gflops:.6f} GFLOPs")
-------------------------------------------------------------------------------------------
$ python3 ave_vector_float64_perf.py
Serial average: 0.4996371399315683
Time: 2.033808 sec
Estimated performance: 0.000492 GFLOPs
-------------------------------------------------------------------------------------------
```

2.2.1 Vector average (`NumPy mean()` operator)

```
-------------------------------------------------------------------------------------------
# average_vector_float64.py
import numpy as np
import time
size = 1_000_000
# Create vector (float64)
a = np.random.rand(size)
start = time.time()
# Vectorized computation of average
average = np.mean(a)
end = time.time()
elapsed = end - start
print(f"Vectorized average: {average}")
print(f"Time: {elapsed:.6f} sec")
# ------------------------------------------------
# Performance estimation in GFLOPs
# ------------------------------------------------
# np.mean performs sum of 'size' elements (1 addition per element)
# and 1 division at the end
flops = size + 1
gflops = flops / elapsed / 1e9

print(f"Estimated performance: {gflops:.6f} GFLOPs")
import numpy as np
```

```
import time
size = 1_000_000
# Create vector (float64)
a = np.random.rand(size)
start = time.time()
# Vectorized computation of average
average = np.mean(a)
end = time.time()
elapsed = end - start
print(f"Vectorized average: {average}")
print(f"Time: {elapsed:.6f} sec")
# ------------------------------------------------
# Performance estimation in GFLOPs
# ------------------------------------------------
# np.mean performs sum of 'size' elements (1 addition per element)
# and 1 division at the end
flops = size + 1
gflops = flops / elapsed / 1e9
print(f"Estimated performance: {gflops:.6f} GFLOPs")
```

```
$ python3 ave_vector_np_float64_perf.py
Vectorized average: 0.5003316888346073
Time: 0.002881 sec
Estimated performance: 0.347125 GFLOPs
```

To do:

Why the performance of the optimized (vectorized) solution is :

0.347125 GFLOPs/0.000492 GFLOPs = 705 higher ?

Write and test the correspong **average_vectorized** programs for **int_32** and **int_8** types of data.

1.3 `dot` product

1.3.1 `NumPy` (accelerated) implementaion with `dot` operator

The following Python code computes the dot product (scalar product) of two input vectors.

A remainder :

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 b_1 + a_2 b_2 + a_3 b_3$$

```
--------------------------------------------------------------------------------
# dot_product_numpy_bw.py
import numpy as np
import time
# Vector size
N = 10_000_000   # 10 million elements
# Create double precision vectors
a = np.random.rand(N).astype(np.float64)
b = np.random.rand(N).astype(np.float64)
# Warm-up (important for fair timing)
np.dot(a[:1000], b[:1000])
# ------------------------------
# Measure execution time
# ------------------------------
start = time.perf_counter()
result = np.dot(a, b)       # NumPy BLAS / vectorized kernel (RVV if supported)
end = time.perf_counter()
elapsed = end - start
# ------------------------------
# Performance metrics
# ------------------------------
print(f"Dot product result: {result}")
print(f"Vector size: {N}")
print(f"Execution time: {elapsed:.6f} seconds")
# FLOPS estimation
flops = 2 * N                # 1 multiply + 1 add per element
gflops = flops / elapsed / 1e9
print(f"Estimated FLOPs: {flops:.3e}")
print(f"Performance: {gflops:.2f} GFLOPS")
# ------------------------------
# Memory bandwidth estimation
# ------------------------------
bytes_read = 2 * N * 8        # read a[i] and b[i] (float64)
bandwidth = bytes_read / elapsed / 1e9
print(f"Data moved: {bytes_read/1e6:.2f} MB")
print(f"Estimated memory bandwidth: {bandwidth:.2f} GB/s")
--------------------------------------------------------------------------------
Dot product result: 2500175.8414340126
Vector size: 10000000
Execution time: 0.039427 seconds
Estimated FLOPs: 2.000e+07
Performance: 0.51 GFLOPS
Data moved: 160.00 MB
Estimated memory bandwidth: 4.06 GB/s
--------------------------------------------------------------------------------
```

1.3.2 C (RVV optimized) implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
size_t N = 10000000;
double dot_f64(const double *a, const double *b, size_t n)
{
    double sum = 0.0;
    for (size_t i = 0; i < n; i++)
        sum += a[i] * b[i];  // 1 multiplication + 1 addition per iteration
    return sum;
}

int main(int argc, char **argv)
{
    clock_t start, end;
    double elapsed_sec;
    if (argc != 2) {
        printf("Usage: %s vector_size\n", argv[0]); return 1; }
    N = (size_t) atoll(argv[1]);
    // Allocate memory
    double *a = malloc(N * sizeof(double));
    double *b = malloc(N * sizeof(double));
    if (!a || !b) {  printf("Memory allocation failed\n"); return 1;}
    // Initialize vectors
    for (size_t i = 0; i < N; i++) {  a[i] = 1.0; b[i] = 2.0; }
    double result = 0.0;
    // Start timing
    start = clock();
    result = dot_f64(a, b, N);
    end = clock();
    elapsed_sec = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Scalar execution time in sec: %.6f\n", elapsed_sec);
    printf("Scalar dot product = %.6f\n", result);
    // ------------------------------------------------
    // Performance estimation in GFLOPs
    // ------------------------------------------------
    // Each iteration: 1 multiplication + 1 addition = 2 FLOPs
    double flops_per_element = 2.0;
    double total_flops = N * flops_per_element;
    double gflops = total_flops / elapsed_sec / 1e9;
    printf("Estimated performance: %.6f GFLOPs\n", gflops);
    free(a); free(b);
    return 0;
}
```

Compilation with optimization for use of RVV instructions:

```
$ gcc -O2 -march=rv64gcv  dot_prod_perf.c -o dot_prod_perf
$ ./dot_prod_perf 10000000
Scalar execution time in sec: 0.039761
Scalar dot product = 20000000.000000
Estimated performance: 0.503005 GFLOPs
```

Remark:

The execution time (and performance) for Python (**`NumPy`**) is **almost the same** as the execution time for the optimized C code !

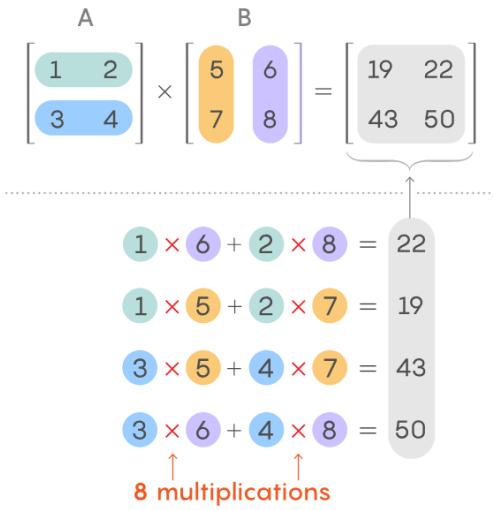
To do

Write and test the corresponding **`int32`** and **`int8`** versions !

1.4 Matrix multiplication

1.4.1 Matrix multiplication with `NumPy @` operator

A remainder :



```
# mmult_perf.py
import numpy as np
import time
import sys

def matrix_multiply(size):
    print("Matrix multiplication using NumPy")
    print("Matrix size:", size, "x", size)
    # Generate matrices (float64)
    A = np.random.rand(size, size).astype(np.float64)
    B = np.random.rand(size, size).astype(np.float64)
    # Warm-up (important for fair timing)
    C = A @ B
    # ----------------------------
    # Measure execution time
    # ----------------------------
    t0 = time.perf_counter()
    C = A @ B
    t1 = time.perf_counter()
    elapsed_time = t1 - t0
    print(f"Execution time: {elapsed_time:.6f} seconds")
    # ----------------------------
    # Prevent optimization removal
    # ----------------------------
    print("Checksum:", np.sum(C))
    # ----------------------------
    # FLOP estimation
    # ----------------------------
    flops = 2 * size**3
    gflops = flops / elapsed_time / 1e9
    print(f"Total FLOPs: {flops:.3e}")
    print(f"Estimated performance: {gflops:.2f} GFLOPS")
    # ----------------------------
    # Memory bandwidth estimation
    # ----------------------------
    bytes_moved = 3 * size**2 * 8   # A read + B read + C write
    bandwidth = bytes_moved / elapsed_time / 1e9
    print(f"Data moved: {bytes_moved/1e6:.2f} MB")
    print(f"Estimated memory bandwidth: {bandwidth:.2f} GB/s")
```

```
if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python matrix_numpy.py SIZE")
        print("Example: python matrix_numpy.py 256")
        sys.exit(1)

    size = int(sys.argv[1])
    matrix_multiply(size)
```

```
$ python3 mmult_perf.py 256
Matrix multiplication using NumPy
Matrix size: 256 x 256
Execution time: 0.090529 seconds
Checksum: 4200945.435874556
Total FLOPs: 3.355e+07
Estimated performance: 0.37 GFLOPS
Data moved: 1.57 MB
Estimated memory bandwidth: 0.02 GB/s
```

1.4.2 C (RVV optimized) implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char **argv) {
    if(argc != 2) {
        printf("Usage: %s MATRIX_SIZE\n", argv[0]);
        printf("Example: %s 256\n", argv[0]);
        return 1;
    }
    int N = atoi(argv[1]);
    printf("Matrix multiplication in C\n");
    printf("Matrix size: %d x %d\n", N, N);
    // Allocate matrices
    double **A = malloc(N * sizeof(double*));
    double **B = malloc(N * sizeof(double*));
    double **C = malloc(N * sizeof(double*));
    for(int i=0; i<N; i++) {
        A[i] = malloc(N * sizeof(double));
        B[i] = malloc(N * sizeof(double));
        C[i] = malloc(N * sizeof(double));
    }
    // Initialize matrices with random values
    srand((unsigned int)time(NULL));
    for(int i=0; i<N; i++) {
        for(int j=0; j<N; j++) {
            A[i][j] = (double)rand() / RAND_MAX;
            B[i][j] = (double)rand() / RAND_MAX;
            C[i][j] = 0.0;
        }
    }
    // ---------------------------
    // Measure execution time
    // ---------------------------
    clock_t start = clock();
    // Standard matrix multiplication (C = A * B)
    for(int i=0; i<N; i++) {
        for(int j=0; j<N; j++) {
            double sum = 0.0;
            for(int k=0; k<N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
    clock_t end = clock();
    double elapsed_sec = (double)(end - start) / CLOCKS_PER_SEC;
```

```
    printf("Execution time: %.6f seconds\n", elapsed_sec);
    // ---------------------------
    // Prevent compiler optimization
    // ---------------------------
    double checksum = 0.0;
    for(int i=0; i<N; i++)
        for(int j=0; j<N; j++)
            checksum += C[i][j];
    printf("Checksum: %.6f\n", checksum);
    // ---------------------------
    // Estimate FLOPs and GFLOPS
    // ---------------------------
    double flops = 2.0 * N * N * N;  // 2*N^3 FLOPs
    double gflops = flops / elapsed_sec / 1e9;
    printf("Estimated performance: %.2f GFLOPS\n", gflops);
    // ---------------------------
    // Free memory
    // ---------------------------
    for(int i=0; i<N; i++) {
        free(A[i]);
        free(B[i]);
        free(C[i]);
    }
    free(A); free(B); free(C);
    return 0;
}
```

```
$ gcc -O2 -march=rv64gcv  mmultvector.c -o mmultvector
$ ./mmultvector 256
Matrix multiplication in C
Matrix size: 256 x 256
Execution time: 0.401244 seconds
Checksum: 4199559.493769
Estimated performance: 0.08 GFLOPS
```

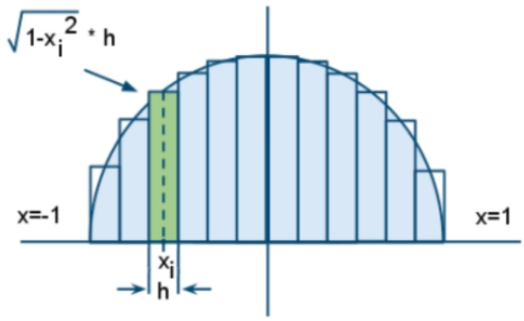
Remark:

The execution time for Python (**`NumPy`**) is **much shorter (4 times)** as the execution time for the optimized C code !

To do

Write and test the corresponding **`int32`** and **`int8`** versions !

1.5 Pi value calculations



In this program:

1. **`x*x`** → **`1`** multiply per element → N multiplies
2. **`1.0 + x*x`** → 1 addition per element → N adds
3. **`4.0 / (...)`** → **`1`** division per element → N divides
4. **`np.mean(y)`** → N adds + 1 division

1.5.1 `NumPy` implemetation with RVV acceleration

```
-------------------------------------------------------------------------------------
import numpy as np
import time
# Number of samples
N = 50_000_000
print("Computing Pi using NumPy vector operations...")
print("Samples:", N)
t0 = time.perf_counter()
# Generate vector of x values
x = np.linspace(0.0, 1.0, N, dtype=np.float64)
# Vectorized computation
y = 4.0 / (1.0 + x*x)
# Integration (mean value method)
pi_est = np.mean(y)
t1 = time.perf_counter()
elapsed_time = t1 - t0
print("Estimated Pi =", pi_est)
print("Error =", abs(np.pi - pi_est))
print("Time =", elapsed_time, "seconds")
# ---------------------------
# Estimate FLOPs and GFLOPS
# ---------------------------
# FLOPs: x*x (1 mul) + 1 + (1 div) + mean (N adds + 1 div)
flops = 4 * N
gflops = flops / elapsed_time / 1e9
print(f"Estimated performance: {gflops:.2f} GFLOPS")
-------------------------------------------------------------------------------------
$ python3 pivect_perf.py
Computing Pi using NumPy vector operations...
Samples: 50000000
Estimated Pi = 3.1415926507579397
Error = 2.8318534184279542e-09
Time = 2.421664182999848 seconds
Estimated performance: 0.08 GFLOPS

-------------------------------------------------------------------------------------
```

1.5.2 C `(accelerated)` implemetation

Let us compare this program with **C implementation**

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

int main(int argc, char **argv) {
    if(argc != 2) {
        printf("Usage: %s NUM_SAMPLES\n", argv[0]);
        printf("Example: %s 50000000\n", argv[0]); return 1;
    }
    long N = atol(argv[1]);
    printf("Computing Pi using C (float64)\n");
    printf("Number of samples: %ld\n", N);
    double *x = malloc(N * sizeof(double));
    double *y = malloc(N * sizeof(double));
    if(x == NULL || y == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }
    // Generate x values
    for(long i = 0; i < N; i++) {
        x[i] = (double)i / (double)(N - 1);
    }
    clock_t start = clock();
    // Vectorized-like computation
    for(long i = 0; i < N; i++) {
        y[i] = 4.0 / (1.0 + x[i]*x[i]);
    }
    // Integration (mean)
    double sum = 0.0;
    for(long i = 0; i < N; i++) { sum += y[i]; }
    double pi_est = sum / (double)N;
    clock_t end = clock();
    double elapsed_sec = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Estimated Pi = %.15f\n", pi_est);
    printf("Error = %.15f\n", fabs(M_PI - pi_est));
    printf("Execution time = %.6f seconds\n", elapsed_sec);
    // ---------------------------
    // Estimate FLOPs and GFLOPS
    // ---------------------------
    // x*x (1 mul), 1.0 + x*x (1 add), 4.0 / (...) (1 div) per element
    // mean: N adds + 1 div → total ~4*N FLOPs
    double flops = 4.0 * N;
    double gflops = flops / elapsed_sec / 1e9;
    printf("Estimated performance: %.2f GFLOPS\n", gflops);
    free(x);free(y);
    return 0;
}
```

```
$ gcc -O2 -march=rv64gcv  pivect_perf.c -o pivect_perf
$ ./pivect_perf 50000000
Computing Pi using C (float64)
Number of samples: 50000000
Estimated Pi = 3.141592650758739
Error = 0.000000002831054
Execution time = 1.613802 seconds
Estimated performance: 0.12 GFLOPS
```

To do

Analyze and execute the following examples. Compare the performances.

Write and test the corresponding **`float32`**

1.6 Summary

This work focuses on the use of an **optimized `NumPy` library** to improve the performance of **vector and matrix processing** on systems based on the **RISC-V architecture with vector instructions**.
Modern RISC-V processors support vector extensions that allow multiple data elements to be processed at the same time, which significantly increases computational efficiency.

Since **`NumPy`** is designed for numerical computing using arrays, it is particularly suitable for taking advantage of this type of hardware acceleration when an optimized version of the library is used.

In this study, simple examples of vector and matrix processing were implemented using **`NumPy`**, including vector addition, scalar multiplication, matrix operations, and basic numerical transformations. Instead of processing elements one by one, **`NumPy`** performs operations on entire arrays using **optimized low-level routines** that can utilize the RISC-V vector extension.

This approach reduces execution time and improves overall performance while keeping the Python code clear and easy to understand.
The results demonstrate that combining Python with an optimized NumPy implementation provides an efficient solution for numerical computing on modern RISC-V platforms.
Even for simple vector and matrix operations, the use of vector instructions leads to noticeable performance improvements.
This confirms that high-level programming in Python can still achieve efficient execution when supported by optimized libraries and modern processor architectures such as RISC-V with vector processing support.