How to write parallel python code

1. Toy problem
2. Profiling
3. Vectorisation
4. Parallel execution
5. Wrapping up

1. Toy problem

1. **Toy problem**
    A. Matrix multiplication
    B. Code example
2. Profiling
3. Vectorisation
4. Parallel execution
5. Wrapping up

1.1. Matrix multiplication

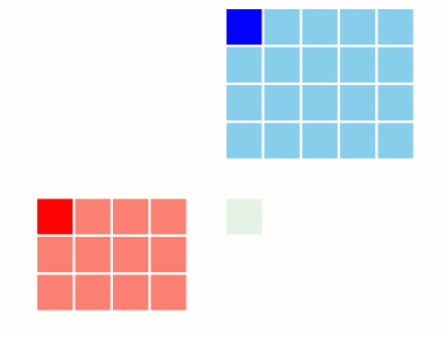
For three matrices A, B and C such that $A_{(m \times l)} \cdot B_{(l \times n)} = C_{(m \times n)}$

The elements of matrix C noted as $C_{(m \times n)} = \begin{bmatrix} c_{00} & \cdots & c_{0j} & \cdots & c_{0n} \\ \vdots & \ddots & \vdots & & \vdots \\ c_{i0} & \cdots & c_{ij} & \cdots & c_{in} \\ \vdots & & \vdots & \ddots & \vdots \\ c_{m0} & \cdots & c_{mj} & \cdots & c_{mn} \end{bmatrix}$

Are computed as follows: $c_{ij} = \Sigma_k^l (a_{ik} \cdot b_{kj})$

1.1. Matrix multiplication

$c_{ij} = \Sigma_k^l (a_{ik} \cdot b_{kj})$



For each element c_{ij} of C

1. Select:
    - The i^{th} **row** of A
    - The j^{th} **column** of B
2. Multiply $a_{i.}$ and $b_{.j}$ **element-wise**
3. **Reduce** the result with a sum operation

1.2. Code example

1. **Toy problem**
    A. Matrix multiplication
    B. **Code example**
2. Profiling
3. Vectorisation
4. Parallel execution
5. Wrapping up

1.2. Code example

1.2. Code example

```
import numpy as np

def matMulElem(aRow, bCol, l):
    cij = 0
    for k in range(l):
        ab = aRow[k] * bCol[k]  # multiply element-wise
        cij += ab  # reduce with a sum
    return cij

def matMul(A, B):
    (m, l) = A.shape
    (ll, n) = B.shape
    #assert l == ll  # check the shapes during dev.
    C = np.empty((m, n), dtype=A.dtype)
    for i in range(m):
        for j in range(n):
            C[i, j] = matMulElem(A[i, :], B[:, j], l)
    return C
```

2. Profiling

1. Toy problem
2. **Profiling**
    A. Test data (benchmark)
    B. Driver program
3. Vectorisation
4. Parallel execution
5. Wrapping up

2.1. Test data (benchmark)

2.1. Test data (benchmark)

```
def produceMatrices(m, n, l):
    A = np.random.rand(m, l).astype(np.float64)
    B = np.random.rand(l, n).astype(np.float64)
    return A, B, A@B

matricesLowL = [  # !! Use the same data across different runs
    produceMatrices(200, 400, 10)
    for _ in range(10)  # !! Have several (different) samples
]

matricesHighL = [  # !! Use the same data across different runs
    produceMatrices(20, 40, 10000)
    for _ in range(10)  # !! Have several (different) samples
]
```

2.2. Driver program

```
nRuns = 10

def runMainLowL():
    for _ in range(nRuns):  # !! Make several runs
        for A, B, expected in matricesLowL:
            C = matMul(A, B)
            # test the result (working code > fast code)
            assert np.allclose(C, expected)

def runMainHighL():
    for _ in range(nRuns):  # !! Make several runs
        for A, B, expected in matricesHighL:
            C = matMul(A, B)
            # test the result (working code > fast code)
            assert np.allclose(C, expected)

def main():
    runMainLowL()
    runMainHighL()
```

2.2. Driver program

```
import line_profiler as lp

# Dummy run to trigger any JIT and avoid the overhead during profiling
main()

pr = lp.LineProfiler()
pr.add_function(matMulElem)
pr.add_function(matMul)
pr.add_function(runMainLowL)
pr.add_function(runMainHighL)
prMain = pr(main)
prMain()

# Generates a file containing statistics to be examined later :
pr.dump_stats("kernProfOut/matMul.lprof")
```

```
        Timer unit: 1e-09 s

Total time: 75.4722 s
File: /tmp/ipykernel_18064/1713277412.py
Function: runMainLowL at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           def runMainLowL():
     4        11       3193.0    290.3      0.0      for _ in range(nRuns):  # !! Make several runs
     5       110     109096.0    991.8      0.0          for A, B, expected in matricesLowL:
     6       100        8e+10    8e+08     99.9              C = matMul(A, B)
     7                                                    # test the result (working code > fast code)
     8       100   47084245.0 470842.5      0.1              assert np.allclose(C, expected)

Total time: 629.483 s
File: /tmp/ipykernel_18064/1713277412.py
Function: runMainHighL at line 10

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    10                                           def runMainHighL():
    11        11       3633.0    330.3      0.0      for _ in range(nRuns):  # !! Make several runs
    12       110      92445.0    840.4      0.0          for A, B, expected in matricesHighL:
    13       100        6e+11    6e+09    100.0              C = matMul(A, B)
    14                                                    # test the result (working code > fast code)
    15       100   18443109.0 184431.1      0.0              assert np.allclose(C, expected)

Total time: 704.955 s
File: /tmp/ipykernel_18064/1713277412.py
Function: main at line 17

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    17                                           def main():
    18         1        8e+10    8e+10     10.7      runMainLowL()
    19         1        6e+11    6e+11     89.3      runMainHighL()

Total time: 384.568 s
File: /tmp/ipykernel_18064/2565869914.py
```

3.1. Array programming

1. Toy problem
2. Profiling
3. **Vectorisation**
    A. **Array programming**
        a. Indexing
        b. Broadcasting
    B. SIMD
4. Parallel execution
5. Wrapping up

3.1. Array programming

New paradigm :

- The "atoms" you manipulate are arrays, not their individual elements anymore
- Apply operations on entire arrays at once
- (Don't freak out, everyone needs some time to adapt)

Effects:

- Reduces loops overhead
- Relies on highly optimised subroutines
    - Coded in C (hence compiled), conveniently packaged in python modules
- **Python code remains high-level** (details are abstracted away)

3.1. Array programming

```
def betterMatMulElem(aRow, bCol):
    ab = aRow * bCol  # element-wise operation
    cij = np.sum(ab)  # reduction operation
    return cij
```

3.1. Array programming

```
def betterMatMulElem(aRow, bCol):
    ab = aRow * bCol  # element-wise operation
    cij = np.sum(ab)  # reduction operation
    return cij
```

- No more loops
- No need to use `l` anymore
    - Both arrays are vectors of size `l`
    - **Warning:** don't mess the shape of your arrays
- Uses (one of) numpy reduction operations
    - sum, prod, max, min, mean, std, ...
    - In *n* dimensions you can choose the axis on which to perform the reduction

3.1. Array programming

```
def betterMatMul(A, B):
    (m, l) = A.shape
    (ll, n) = B.shape
    #assert l == ll  # check the shapes during dev.
    C = np.empty((m, n), dtype=A.dtype)
    for i in range(m):
        for j in range(n):
            C[i, j] = betterMatMulElem(A[i, :], B[:, j])
    return C
```

```
       Timer unit: 1e-09 s

Total time: 37.405 s
File: /tmp/ipykernel_18064/3706775867.py
Function: betterMatMul at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def betterMatMul(A, B):
     2       200     181468.0    907.3      0.0      (m, l) = A.shape
     3       200      77181.0    385.9      0.0      (ll, n) = B.shape
     4                                               #assert l == ll  # check the shapes during dev.
     5       200     324735.0   1623.7      0.0      C = np.empty((m, n), dtype=A.dtype)
     6     22200    2187919.0     98.6      0.0      for i in range(m):
     7   8102000  800333946.0     98.8      2.1          for j in range(n):
     8   8080000        4e+10   4529.9     97.9              C[i, j] = betterMatMulElem(A[i, :], B[:, j])
     9       200      24170.0    120.8      0.0      return C

Total time: 26.9513 s
File: /tmp/ipykernel_18064/3811478808.py
Function: betterMatMulElem at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def betterMatMulElem(aRow, bCol):
     2   8080000 4568595627.0    565.4     17.0      ab = aRow * bCol  # element-wise operation
     3   8080000        2e+10   2675.6     80.2      cij = np.sum(ab)  # reduction operation
     4   8080000  763789143.0     94.5      2.8      return cij

Total time: 37.8206 s
File: /tmp/ipykernel_18064/614569483.py
Function: runBetterMainLowL at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def runBetterMainLowL():
     2        11       3374.0    306.7      0.0      for _ in range(nRuns):  # !! Make several runs
     3       110      87129.0    792.1      0.0          for A, B, expected in matricesLowL:
     4       100        4e+10    4e+08     99.9              C = betterMatMul(A, B)
     5                                                   # test the result (working code > fast code)
```

3.1. Array programming

That seems simple enough.

Can we go one step further ? Can we **remove** all loops ?

YES. Array programming is meant for that and provides you with different techniques to achieve this goal. We will see two of them:

- **Indexing** an array with other arrays of indices
- **Broadcasting** arrays of different shapes

3.1.1. Indexing

```
def noLoopMatMul(A, B):
    (m, l) = A.shape
    (ll, n) = B.shape
    #assert l == ll  # check the shapes during dev.
    i = np.arange(m, dtype=np.int32)  # [0, 1, 2, ... m-1]
    i = np.repeat(i, n)  # [0, 0, 0, ... 1, 1, 1, ...]
    j = np.arange(n, dtype=np.int32)  # [0, 1, 2, ... n-1]
    j = np.tile(j, m)  # [0, 1, 2, ... 0, 1, 2, ...]
    C = np.empty((m, n), dtype=A.dtype)
    aRows = A[i, :]
    #assert aRows.shape == (m*n, l)  # the shaaaaapes !
    bCols = B[:, j]
    #assert bCols.shape == (l, m*n)  # check theeeem !!
    # i, j = (0, 0), (0, 1), (0, 2), ... (1, 0), (1, 1), (1, 2), ...
    C[i, j] = np.sum(aRows * bCols.T, axis=1)
    return C
```

```
       Timer unit: 1e-09 s

Total time: 0.745057 s
File: /tmp/ipykernel_18064/2684593976.py
Function: runNoLoopMainLowL at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def runNoLoopMainLowL():
     2        11       6888.0    626.2      0.0      for _ in range(nRuns):  # !! Make several runs
     3       110      86930.0    790.3      0.0          for A, B, expected in matricesLowL:
     4       100  707457903.0    7e+06     95.0              C = noLoopMatMul(A, B)
     5                                                     # test the result (working code > fast code)
     6       100   37504948.0 375049.5      5.0              assert np.allclose(C, expected)

Total time: 4.35136 s
File: /tmp/ipykernel_18064/2684593976.py
Function: runNoLoopMainHighL at line 8

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     8                                           def runNoLoopMainHighL():
     9        11       7732.0    702.9      0.0      for _ in range(nRuns):  # !! Make several runs
    10       110      82214.0    747.4      0.0          for A, B, expected in matricesHighL:
    11       100 4342503504.0    4e+07     99.8              C = noLoopMatMul(A, B)
    12                                                     # test the result (working code > fast code)
    13       100    8762860.0  87628.6      0.2              assert np.allclose(C, expected)

Total time: 5.09699 s
File: /tmp/ipykernel_18064/2684593976.py
Function: noLoopMain at line 15

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    15                                           def noLoopMain():
    16         1  745291782.0    7e+08     14.6      runNoLoopMainLowL()
    17         1 4351699153.0    4e+09     85.4      runNoLoopMainHighL()

Total time: 4.97684 s
File: /tmp/ipykernel_18064/2732079577.py
```

3.1.1. Indexing

No loops through indexing seems to have some potential:

- Good performances when the matrices are small
- But **terrible** when the matrices get bigger. Why is that ?

Indexing arrays is ok unless it means **replicating** data beyond reason

`aRows.shape == (m*n, l)` and `bCols.shape == (l, m*n)` means allocating a lot of additional memory to store redundant data. It also means that `np.sum` has to work on large arrays, slowing the process even further.

How to deal with this situation ?

3.1.2. Broadcasting

What about broadcasting ?

Concept:

- Binary (element-wise) operations such as the multiplication will fail on two arrays of different shapes. Unless these shapes can be **broadcasted** together.
- Imagine matrices $A_{(m,l)}$, $B_{(l,n)}$ and a vector $V_{(l)}$
- `A * V` is accepted, numpy will multiply each (whole) column j of A with v_j
- In other words, numpy wil **broadcast** v along the rows of A
- However `B * V` and `V * B` are not accepted, but `V.reshape(l, 1) * B` will multiply each row i with v_i
- In other words, numpy wil **broadcast** v along the columns of B

3.1.2. Broadcasting

What about broadcasting ?

Broadcasting arrays with the same (number of) dimensions/axis is possible if:

- All dimensions are equal (hence no need for broadcast)
- For any pair of respective dimensions that differ, one is equal to 1 (then the array is broadcasted along that axis)

Broadcast of arrays of different (number of) dimensions/axis is also possible, but can be misleading/counterintuitive

Better safe than sorry: always reshape your arrays with 1's such that the number of axis is the same.

3.1.2. Broadcasting

What about broadcasting in practice ?

3.1.2. Broadcasting

What about broadcasting in practice ?

```
def betterNoLoopMatMul(A, B):
    (m, l) = A.shape
    (ll, n) = B.shape
    #assert l == ll  # check the shapes during dev.
    C = A.reshape(m, l, 1) * B.reshape(1, l, n)
    #assert C.shape == (m, l, n)  # you definitely want to check that shape
    C = np.sum(C, axis=1)  # apply the reduction along the second axis
    #assert C.shape == (m, n)  # The shape. Check again
    return C
```

```
       Timer unit: 1e-09 s

Total time: 2.3136 s
File: /tmp/ipykernel_18064/2613406731.py
Function: betterNoLoopMatMul at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def betterNoLoopMatMul(A, B):
     2       200     135647.0    678.2      0.0      (m, l) = A.shape
     3       200      50707.0    253.5      0.0      (ll, n) = B.shape
     4                                               #assert l == ll  # check the shapes during dev.
     5       200 1725674320.0    9e+06     74.6      C = A.reshape(m, l, 1) * B.reshape(1, l, n)
     6                                               #assert C.shape == (m, l, n)  # you definitely want to check that
shape
     7       200  587685486.0    3e+06     25.4      C = np.sum(C, axis=1)  # apply the reduction along the second axis
     8                                               #assert C.shape == (m, n)  # The shape. Check again
     9       200      54875.0    274.4      0.0      return C

Total time: 0.145728 s
File: /tmp/ipykernel_18064/4014536370.py
Function: runBetterNoLoopMainLowL at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def runBetterNoLoopMainLowL():
     2        11       4848.0    440.7      0.0      for _ in range(nRuns):  # !! Make several runs
     3       110      44404.0    403.7      0.0          for A, B, expected in matricesLowL:
     4       100  117819183.0    1e+06     80.8              C = betterNoLoopMatMul(A, B)
     5                                                       # test the result (working code > fast code)
     6       100   27860023.0 278600.2     19.1              assert np.allclose(C, expected)

Total time: 2.20486 s
File: /tmp/ipykernel_18064/4014536370.py
Function: runBetterNoLoopMainHighL at line 8

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     8                                           def runBetterNoLoopMainHighL():
     9        11       6917.0    628.8      0.0      for _ in range(nRuns):  # !! Make several runs
```

3.1. Array programming

Can we **compile** on top of array programming ?

3.1. Array programming

Can we **compile** on top of array programming ?

```
import numba as nb

@nb.jit(nopython=True)
def JITMatMulElem(aRow, bCol):
    ab = aRow * bCol
    cij = np.sum(ab)
    return cij
```

```
def JITMatMul(A, B):
    (m, l) = A.shape
    (ll, n) = B.shape
    #assert l == ll  # check the shapes during dev.
    C = np.empty((m, n), dtype=A.dtype)
    for i in range(m):
        for j in range(n):
            C[i, j] = JITMatMulElem(A[i, :], B[:, j])
    return C

@nb.jit(nopython=True)
def ALLJITMatMul(A, B):
    (m, l) = A.shape
    (ll, n) = B.shape
    C = np.empty((m, n), dtype=A.dtype)
    #assert l == ll  # check the shapes during dev.
    for i in range(m):
        for j in range(n):
            C[i, j] = JITMatMulElem(A[i, :], B[:, j])
    return C
```

```
       Timer unit: 1e-09 s

Total time: 8.00805 s
File: /tmp/ipykernel_18064/3028311925.py
Function: JITMatMul at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def JITMatMul(A, B):
     2       200     150399.0    752.0      0.0      (m, l) = A.shape
     3       200      67730.0    338.6      0.0      (ll, n) = B.shape
     4                                               #assert l == ll  # check the shapes during dev.
     5       200     239517.0   1197.6      0.0      C = np.empty((m, n), dtype=A.dtype)
     6     22200    2430270.0    109.5      0.0      for i in range(m):
     7   8102000  805080131.0     99.4     10.1          for j in range(n):
     8   8080000 7200047884.0    891.1     89.9              C[i, j] = JITMatMulElem(A[i, :], B[:, j])
     9       200      35901.0    179.5      0.0      return C

Total time: 9.55951 s
File: /tmp/ipykernel_18064/3172951264.py
Function: runJITMainLowL at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def runJITMainLowL():
     2        11       4940.0    449.1      0.0      for _ in range(nRuns):
     3       110      85043.0    773.1      0.0          for A, B, expected in matricesLowL:
     4       100 8897328073.0    9e+07     93.1              JITC = JITMatMul(A, B)
     5       100  601023133.0    6e+06      6.3              ALLJITC = ALLJITMatMul(A, B)
     6       100   37261322.0 372613.2      0.4              assert np.allclose(JITC, expected)
     7       100   23806104.0 238061.0      0.2              assert np.allclose(ALLJITC, expected)

Total time: 2.34402 s
File: /tmp/ipykernel_18064/3172951264.py
Function: runJITMainHighL at line 9

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     9                                           def runJITMainHighL():
    10        11       5135.0    466.8      0.0      for _ in range(nRuns):
```

3.2. SIMD

1. Toy problem
2. Profiling
3. **Vectorisation**
    A. Array programming
    B. **SIMD**
        a. Data type
        b. Memory layout
4. Parallel execution
5. Wrapping up

3.2. SIMD

- Same Instruction Multiple Data
- Human: Operations are applied on the **elements** of an array
- Machine: Operations are applied on a **register** of 8/16/32/64 bits (and beyond)
- SIMD: Fit as many elements as possible in a register and apply the operation ***once***
- Not possible without *unrolling* a loop
    - Possible to do it at low level (C/C++, assembly, ...)
    - In python, handled by libraries such as numpy, tensorflow, pytorch, ...

The gain in speed depends on the size of your data and the size of the registers. It is useful on a CPU, but on a GPU the registers are **A LOT** larger

3.2.1. Data type

```
def runSIMDMainLowL():
    for _ in range(nRuns):
        for A, B, expected in matricesLowL:
            A64, B64 = A.astype(np.float64), B.astype(np.float64)
            C1 = ALLJITMatMul(A64, B64)
            C2 = betterNoLoopMatMul(A64, B64)
            assert np.allclose(C1, expected)
            assert np.allclose(C2, expected)

            A32, B32 = A.astype(np.float32), B.astype(np.float32)
            C1 = ALLJITMatMul(A32, B32)
            C2 = betterNoLoopMatMul(A32, B32)
            assert np.allclose(C1, expected)
            assert np.allclose(C2, expected)
```

3.2.1. Data type

```
def runSIMDMainHighL():
    for _ in range(nRuns):
        for A, B, expected in matricesHighL:
            A64, B64 = A.astype(np.float64), B.astype(np.float64)
            C1 = ALLJITMatMul(A64, B64)
            C2 = betterNoLoopMatMul(A64, B64)
            assert np.allclose(C1, expected)
            assert np.allclose(C2, expected)

            A32, B32 = A.astype(np.float32), B.astype(np.float32)
            C1 = ALLJITMatMul(A32, B32)
            C2 = betterNoLoopMatMul(A32, B32)
            assert np.allclose(C1, expected)
            assert np.allclose(C2, expected)
```

```
Timer unit: 1e-09 s

Total time: 5.47566 s
File: /tmp/ipykernel_18064/1839227687.py
Function: runSIMDMainHighL at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def runSIMDMainHighL():
     2        11       7609.0    691.7      0.0      for _ in range(nRuns):
     3       110      85982.0    781.7      0.0          for A, B, expected in matricesHighL:
     4       100   30528503.0 305285.0      0.6              A64, B64 = A.astype(np.float64), B.astype(np.float64)
     5       100 1122040310.0    1e+07     20.5              C1 = ALLJITMatMul(A64, B64)
     6       100 2258037367.0    2e+07     41.2              C2 = betterNoLoopMatMul(A64, B64)
     7       100    9194658.0  91946.6      0.2              assert np.allclose(C1, expected)
     8       100    2886239.0  28862.4      0.1              assert np.allclose(C2, expected)
     9
    10       100   34882853.0 348828.5      0.6              A32, B32 = A.astype(np.float32), B.astype(np.float32)
    11       100 1016318758.0    1e+07     18.6              C1 = ALLJITMatMul(A32, B32)
    12       100  990708276.0    1e+07     18.1              C2 = betterNoLoopMatMul(A32, B32)
    13       100    8029643.0  80296.4      0.1              assert np.allclose(C1, expected)
    14       100    2937327.0  29373.3      0.1              assert np.allclose(C2, expected)

Total time: 1.49925 s
File: /tmp/ipykernel_18064/801465010.py
Function: runSIMDMainLowL at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def runSIMDMainLowL():
     2        11       7314.0    664.9      0.0      for _ in range(nRuns):
     3       110      68351.0    621.4      0.0          for A, B, expected in matricesLowL:
     4       100     835979.0   8359.8      0.1              A64, B64 = A.astype(np.float64), B.astype(np.float64)
     5       100  607792072.0    6e+06     40.5              C1 = ALLJITMatMul(A64, B64)
     6       100  124229157.0    1e+06      8.3              C2 = betterNoLoopMatMul(A64, B64)
     7       100   29275721.0 292757.2      2.0              assert np.allclose(C1, expected)
     8       100   22606869.0 226068.7      1.5              assert np.allclose(C2, expected)
     9
    10       100     817752.0   8177.5      0.1              A32, B32 = A.astype(np.float32), B.astype(np.float32)
    11       100  604812963.0    6e+06     40.3              C1 = ALLJITMatMul(A32, B32)
```

3.2.2. Memory layout

- Same Instruction Multiple Data
- First step is to fetch data from memory
- Faster if the data is contiguous in memory
    - In memory, everything is in one dimension
    - Matrix is in two, two adjacent elements may not be contiguous
- **Row-major :** (*aka* C-style arrays)
    - Elements on a same **line** are contiguous
    - **Lines** are written consecutively in memory (as if concatenated)
- **Column-major :** (*aka* Fortran-style arrays)
    - Elements on a same **column** are contiguous
    - **Columns** are written consecutively in memory (as if concatenated)

Contiguous case : copying a block of memory directly in the register (1 instruction)

Non-contiguous (sparse) case : fetch individual elements from memory to fill the register ($n \geq 1$ instructions)

```
def runOrderedSIMDMainLowL():
    for _ in range(nRuns):
        for A, B, expected in matricesLowL:
            A32F, B32C = np.asarray(A, dtype=np.float32, order='F'), np.asarray(B,
            C1 = ALLJITMatMul(A32F, B32C)
            C2 = betterNoLoopMatMul(A32F, B32C)
            assert np.allclose(C1, expected)
            assert np.allclose(C2, expected)

            A32C, B32F = np.asarray(A, dtype=np.float32, order='C'), np.asarray(B,
            C1 = ALLJITMatMul(A32C, B32F)
            C2 = betterNoLoopMatMul(A32C, B32F)
            assert np.allclose(C1, expected)
            assert np.allclose(C2, expected)

def runOrderedSIMDMainHighL():
    for _ in range(nRuns):
        for A, B, expected in matricesHighL:
            A32F, B32C = np.asarray(A, dtype=np.float32, order='F'), np.asarray(B,
            C1 = ALLJITMatMul(A32F, B32C)
            C2 = betterNoLoopMatMul(A32F, B32C)
            assert np.allclose(C1, expected)
            assert np.allclose(C2, expected)

            A32C, B32F = np.asarray(A, dtype=np.float32, order='C'), np.asarray(B,
            C1 = ALLJITMatMul(A32C, B32F)
            C2 = betterNoLoopMatMul(A32C, B32F)
            assert np.allclose(C1, expected)
            assert np.allclose(C2, expected)
```

```
        Timer unit: 1e-09 s

Total time: 1.46178 s
File: /tmp/ipykernel_18064/2743443893.py
Function: runOrderedSIMDMainLowL at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def runOrderedSIMDMainLowL():
     2        11       7088.0    644.4      0.0      for _ in range(nRuns):
     3       110      53767.0    488.8      0.0          for A, B, expected in matricesLowL:
     4       100     789590.0   7895.9      0.1              A32F, B32C = np.asarray(A, dtype=np.float32, order='F'),
np.asarray(B, dtype=np.float32, order='C')
     5       100  562505297.0    6e+06     38.5              C1 = ALLJITMatMul(A32F, B32C)
     6       100   52104290.0 521042.9      3.6              C2 = betterNoLoopMatMul(A32F, B32C)
     7       100   27016564.0 270165.6      1.8              assert np.allclose(C1, expected)
     8       100   20998652.0 209986.5      1.4              assert np.allclose(C2, expected)
     9
    10       100     790365.0   7903.6      0.1              A32C, B32F = np.asarray(A, dtype=np.float32, order='C'),
np.asarray(B, dtype=np.float32, order='F')
    11       100  551852455.0    6e+06     37.8              C1 = ALLJITMatMul(A32C, B32F)
    12       100  203058369.0    2e+06     13.9              C2 = betterNoLoopMatMul(A32C, B32F)
    13       100   21837809.0 218378.1      1.5              assert np.allclose(C1, expected)
    14       100   20764980.0 207649.8      1.4              assert np.allclose(C2, expected)

Total time: 4.43287 s
File: /tmp/ipykernel_18064/2743443893.py
Function: runOrderedSIMDMainHighL at line 16

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    16                                           def runOrderedSIMDMainHighL():
    17        11       9144.0    831.3      0.0      for _ in range(nRuns):
    18       110      80840.0    734.9      0.0          for A, B, expected in matricesHighL:
    19       100   37112954.0 371129.5      0.8              A32F, B32C = np.asarray(A, dtype=np.float32, order='F'),
np.asarray(B, dtype=np.float32, order='C')
    20       100 2507458973.0    3e+07     56.6              C1 = ALLJITMatMul(A32F, B32C)
    21       100  785866848.0    8e+06     17.7              C2 = betterNoLoopMatMul(A32F, B32C)
    22       100    7173019.0  71730.2      0.2              assert np.allclose(C1, expected)
    23       100    2679635.0  26796.3      0.1              assert np.allclose(C2, expected)
```

3. Vectorisation

What did we learn ?

- Array programming is great, but use it with caution
    - The best approach may not be easy/intuitive (thinking in n dimensions)
    - Compiled loops may be faster than no loop at all
    - Especially if using indexing/masking techniques

3. Vectorisation

What did we learn ?

- The data type is important
    - Do not use large datatypes if you don't need it
    - Be aware that on some GPUs, float64 is not available (only float32)
    - Operations on ints are faster, if you don't need floats, don't use them
- The order of multi-dimensional arrays may drastically impact performance
    - If you know in advance how you will address your arrays, use that knowledge
    - In doubt, compare the trade-offs of converting arrays *Vs.* apply operation in a sub-optimal way

WARNING : Such optimisation is specific to using arrays and is often very local. It is easy to loose track of time and waste precious development work optimising small details of your code.

Always profile at higher level (*e.g.* using `cProfile`) to focus your efforts on real bottlenecks *first*.

4.1. Multi-threading/processing

1. Toy problem
2. Profiling
3. Vectorisation
4. **Parallel execution**
    A. **Multi-threading/processing**
        a. Concepts
        b. The python situation
    B. Data & Memory
5. Wrapping up

4.1.1. Concepts

Multi-processing :

- Your program (a process) can **fork** (*i.e.* duplicate itself) to create child processes
- Each process handles privately its own memory and resources. **The OS ensures that privacy**
- **Inter-process communication** is done through **pipes**, files, message queues **(*e.g.* MPI)**, sockets (*e.g.* the network), **shared memory**

Multi-threading :

- Your program (a process) can **spawn** several threads of execution
- They are not different processes, they can access the same memory and resources
- Lighter and more powerful than multi-processing, but comes with added complexity
    - *Concurrent access* to resources
    - Issues related to *race conditions*, (*e.g.* deadlocks, starvation, ...)
    - Needs a good understanding of **locks** (mutex), semaphores, **barriers** ...

4.1.2. The python situation

Multi-threading exists in python, **BUT** :

- Python does not want race conditions, deadlocks nor any related issues
- The only way to ensure this (at python level) is by synchronising all operations
- This is possible through the use of the **GIL (Global Interpreter Lock)**
    - All (python) instructions have to
        1. Lock the GIL (automatic)
        2. Execute (this is your code)
        3. Release the GIL (automatic)
    - Concurrent threads are put on hold until they lock the GIL
- Consequently, only one thread at a time can run

Multi-processing also exists in python :

- A python *process* is not only your code, but also the full interpreter executing it
- Several processes imply several instances of the interpreter, hence several GIL
- Only way to achieve "true" parallelisation in python

4.1. Multi-threading/processing

```
import multiprocessing as mp
import psutil

# on a cluster the number of CPU-cores <= number of usable cores
threadNum = len(psutil.Process().cpu_affinity())
#threadNum = os.cpu_count()
```

4.1. Multi-threading/processing

```
import multiprocessing as mp
import psutil

# on a cluster the number of CPU-cores <= number of usable cores
threadNum = len(psutil.Process().cpu_affinity())
#threadNum = os.cpu_count()
```

```
@nb.jit(nopython=True)
def parallelJITMatMulElem(args):
    aRow, bCol = args
    ab = aRow * bCol
    cij = np.sum(ab)
    return cij
```

4.1. Multi-threading/processing

```
def parallelMatMul(A, B):
    (m, l) = A.shape
    (ll, n) = B.shape
    #assert l == ll  # check the shapes during dev.
    C = np.empty((m, n), dtype=A.dtype)
    chunksize = int(np.ceil(m*n/threadNum))
    # Avoid spawning processes directly, use a pool
    with mp.Pool(threadNum) as pool:
        it = ((A[i, :], B[:, j]) for i in range(m) for j in range(n))
        C.flat = list(pool.imap(parallelJITMatMulElem, it, chunksize=chunksize))
    pool.join()  # Wait for all processes to complete
    return C
```

4.1. Multi-threading/processing

4.1. Multi-threading/processing

```
@nb.jit(nopython=True)
def parallelJITMatMulRow(args):
    aRow, B = args
    ab = aRow[:, np.newaxis] * B
    ci = np.sum(ab, axis=0)
    return ci

def rowParallelMatMul(A, B):
    (m, l) = A.shape
    (ll, n) = B.shape
    #assert l == ll  # check the shapes during dev.
    C = np.empty((m, n), dtype=A.dtype)
    # on a cluster the number of CPU-cores <= number of usable cores
    chunksize = int(np.ceil(m/threadNum))
    with mp.Pool(threadNum) as pool:
        it = ((A[i, :], B) for i in range(m))
        C[:, :] = list(pool.imap(parallelJITMatMulRow, it, chunksize=chunksize))
    pool.join()
    return C
```

4.1. Multi-threading/processing

4.1. Multi-threading/processing

```
nb.config.NUMBA_NUM_THREADS = threadNum

@nb.jit(nopython=True, parallel=True)
def parallelALLJITMatMul(A, B):
    (m, l) = A.shape
    (ll, n) = B.shape
    C = np.empty((m, n), dtype=A.dtype)
    #assert l == ll  # check the shapes during dev.
    for i in nb.prange(m):
        C[i, :] = parallelJITMatMulRow((A[i, :], B))
    return C
```

```
        Timer unit: 1e-09 s

Total time: 116.189 s
File: /tmp/ipykernel_18064/2255151162.py
Function: runParallelMainLowL at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def runParallelMainLowL():
     2        11      14377.0   1307.0      0.0      for _ in range(nRuns):
     3       110     584777.0   5316.2      0.0          for A, B, expected in matricesLowL:
     4       100    4717042.0  47170.4      0.0              A32C, B32F = np.asarray(A, dtype=np.float32, order='C'),
np.asarray(B, dtype=np.float32, order='F')
     5       100        1e+11    1e+09     92.4              C1 = parallelMatMul(A32C, B32F)
     6       100 8365083170.0    8e+07      7.2              C2 = rowParallelMatMul(A32C, B32F)
     7       100  156282110.0    2e+06      0.1              C3 = parallelALLJITMatMul(A32C, B32F)
     8       100  180303087.0    2e+06      0.2              assert np.allclose(C1, expected)
     9       100   76994064.0 769940.6      0.1              assert np.allclose(C2, expected)
    10       100   70714718.0 707147.2      0.1              assert np.allclose(C3, expected)

Total time: 77.7176 s
File: /tmp/ipykernel_18064/2255151162.py
Function: runParallelMainHighL at line 12

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    12                                           def runParallelMainHighL():
    13        11      33581.0   3052.8      0.0      for _ in range(nRuns):
    14       110     326820.0   2971.1      0.0          for A, B, expected in matricesHighL:
    15       100  140001746.0    1e+06      0.2              A32C, B32F = np.asarray(A, dtype=np.float32, order='C'),
np.asarray(B, dtype=np.float32, order='F')
    16       100        6e+10    6e+08     83.5              C1 = parallelMatMul(A32C, B32F)
    17       100        1e+10    1e+08     15.6              C2 = rowParallelMatMul(A32C, B32F)
    18       100  508769777.0    5e+06      0.7              C3 = parallelALLJITMatMul(A32C, B32F)
    19       100   52209793.0 522097.9      0.1              assert np.allclose(C1, expected)
    20       100   14579042.0 145790.4      0.0              assert np.allclose(C2, expected)
    21       100   12513314.0 125133.1      0.0              assert np.allclose(C3, expected)

Total time: 193.914 s
File: /tmp/ipykernel_18064/2255151162.py
```

4.2. Data & Memory

1. Toy problem
2. Profiling
3. Vectorisation
4. **Parallel execution**
    A. Multi-threading/processing
    B. **Data & Memory**
        a. Shared memory Vs messages
        b. Concurrent access
5. Wrapping up

4.2.1. Shared memory Vs messages

Remember, processes have to communicate.

That communication is implicitly handled for you by the multiprocessing libray:

1. The function to run and its arguments are serialised
2. The child processes are created and receive a message consisting in the serialised code and data
3. Each child process de-serialises the received function to execute and applies it on the data
4. The returned value (even if None) is serialised and sent back to the parent process

4.2.1. Shared memory Vs messages

Remember, processes have to communicate.

If the data is too large, that approach can become slow and/or consume too much memory, since (part of) the original data is copied by each process.

You can choose to avoid this by storing your data in a part of the memory you allow to be shared with the child processes.

- Access to the data will be slower (since it is no longer local in each process)
- **But** the time for a child process to start will be shorter **and the same regardless of the size of your data**
- Profile and choose the best approach for your use-case, it really depends on your data and what you do with it

4.2.2. Concurrent access

Sharing memory is a great way to reduce memory consumption and speed up parallel code. But you need to be cautious.

What happens if two processes try to access the same data at the same time ? Well, that is exactly the kind of issue the GIL solves.

1. **Write** and **Write** : you either corrupted your data or erased a returned value with another
2. **Write** and **Read** : the writing process should be fine, but the data read by the reading process is most likely corrupted
3. **Read** and **Read** : this is ok, the data is not modified, so both processes read consistent data

By default, shared memory allocated through the multiprocessing library is protected by **locks**.

- In case of doubt, you can safely ignore these problems and simply allocate some shared memory.
- But then only one process at a time can access the shared memory.
- It could still be faster since you skip the serialisation + message passing step

4.2.2. Concurrent access

Sharing memory is a great way to reduce memory consumption and speed up parallel code. But you need to be cautious.

If you know what you are doing, you can disable this lock.

- Matrices A and B are never **written to**, only **read from** (sc. 3.)
- Matrix C is only **written to** (by the child processes) (sc. 1. ?)
- Different child processes never write on the same cell of the matrix C (not sc. 1. ?)

Is it safe to disable the lock on C ?

4.2. Data & Memory

Time to use the big guns

4.2. Data & Memory

Time to use the big guns

```
def produceMatrices(m, n, l):
    A = np.asarray(np.random.rand(m, l), dtype=np.float32, order='C')
    B = np.asarray(np.random.rand(l, n), dtype=np.float32, order='F')
    return A, B

matrices = [  # !! Use the same data across different runs
    produceMatrices(200, 400, 100000)
    for _ in range(3)  # !! Have several (different) samples
]
```

4.2. Data & Memory

```
def initPool(AArray, BArray, CArray):
    global A  # declare A, B and C global across the processes
    global B
    global C
    A = AArray  # These arrays need to be stored in shared memory
    B = BArray
    C = CArray

def parallelMatMulElemInPlace(args):
    i, j = args
    ab = A[i, :] * B[:, j]
    C[i, j] = np.sum(ab)
```

4.2. Data & Memory

```
f32Type = np.ctypeslib.as_ctypes_type(np.float32)


def parallelSharedMatMul(A, B):
    (m, l) = A.shape
    (ll, n) = B.shape
    #assert l == ll  # check the shapes during dev.
    CBase = mp.Array(f32Type, m*n, lock=False)
    C = np.frombuffer(CBase, A.dtype).reshape(m, n)
    chunksize = int(np.ceil(m*n/threadNum))
    with mp.Pool(threadNum, initializer=initPool, initargs=(A, B, C)) as pool:
        it = ((i, j) for i in range(m) for j in range(n))
        list(pool.imap(parallelMatMulElemInPlace, it, chunksize=chunksize))
    pool.join()
    return C
```

4.2. Data & Memory

4.2. Data & Memory

```
def parallelMatMulRowInPlace(args):
    i, = args
    ab = A[i, :][:, np.newaxis] * B
    C[i, :] = np.sum(ab, axis=0)

def rowParallelSharedMatMul(A, B):
    (m, l) = A.shape
    (ll, n) = B.shape
    #assert l == ll  # check the shapes during dev.
    CBase = mp.Array(f32Type, m*n, lock=False)
    C = np.frombuffer(CBase, A.dtype).reshape(m, n)
    chunksize = int(np.ceil(m/threadNum))
    with mp.Pool(threadNum, initializer=initPool, initargs=(A, B, C)) as pool:
        it = ((i,) for i in range(m))
        list(pool.imap(parallelMatMulRowInPlace, it, chunksize=chunksize))
    pool.join()
    return C
```

4.2. Data & Memory

```
def parallelSharedMain():
    for _ in range(3):
        for A, B in matrices:
            A32Base, B32Base = mp.Array(f32Type, A.flat, lock=False), mp.Array(f32
            AFlat, BFlat = np.frombuffer(A32Base, dtype=np.float32), np.frombuffer
            AShared, BShared = AFlat.reshape(A.shape), BFlat.reshape(B.shape)
            C1 = rowParallelMatMul(A, B)
            C2 = parallelSharedMatMul(AShared, BShared)
            C3 = rowParallelSharedMatMul(AShared, BShared)
            C4 = parallelALLJITMatMul(A, B)
            expected = A@B
            assert np.allclose(C1, expected, atol=1.)
            assert np.allclose(C2, expected, atol=1.)
            assert np.allclose(C3, expected, atol=1.)
            assert np.allclose(C4, expected, atol=1.)
```

```
Timer unit: 1e-09 s

Total time: 461.824 s
File: /tmp/ipykernel_18064/3905965916.py
Function: parallelSharedMain at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def parallelSharedMain():
     2         4      10734.0   2683.5      0.0      for _ in range(3):
     3        12     100776.0   8398.0      0.0          for A, B in matrices:
     4         9        5e+10    6e+09     11.8              A32Base, B32Base = mp.Array(f32Type, A.flat, lock=False),
mp.Array(f32Type, B.flat, lock=False)
     5         9     166147.0  18460.8      0.0              AFlat, BFlat = np.frombuffer(A32Base, dtype=np.float32),
np.frombuffer(B32Base, dtype=np.float32)
     6         9  123761260.0    1e+07      0.0              AShared, BShared = AFlat.reshape(A.shape),
BFlat.reshape(B.shape)
     7         9        1e+11    1e+10     22.9              C1 = rowParallelMatMul(A, B)
     8         9        2e+11    2e+10     35.2              C2 = parallelSharedMatMul(AShared, BShared)
     9         9        5e+10    6e+09     11.8              C3 = rowParallelSharedMatMul(AShared, BShared)
    10         9        8e+10    9e+09     17.6              C4 = parallelALLJITMatMul(A, B)
    11         9 2711130807.0    3e+08      0.6              expected = A@B
    12         9   16761546.0    2e+06      0.0              assert np.allclose(C1, expected, atol=1.)
    13         9    4374450.0 486050.0      0.0              assert np.allclose(C2, expected, atol=1.)
    14         9    4032228.0 448025.3      0.0              assert np.allclose(C3, expected, atol=1.)
    15         9    3929688.0 436632.0      0.0              assert np.allclose(C4, expected, atol=1.)
```

5. Wrapping up

1. Toy problem
2. Profiling
3. Vectorisation
4. Parallel execution
5. **Wrapping up**

5. Wrapping up

There are two ways to *"do more work per unit of time"*:

- Vectorise your data
    1. Good to speed up operations at low level
    2. Combine it with array programming
    3. Makes better use of computing resources
- Execute operations in parallel
    1. Augment the number of parallel workers
    2. Possibly heavy overhead
    3. If your data is large, use shared memory
    4. Better used at higher level
    5. More powerful if the worker code is heavy

5. Wrapping up

What is the best strategy ? **None**. Each are good at something different. As usual, test the different approaches and tradeoffs, profile and choose the best according to your specific use-case.

Do not hesitate to use a *sub-optimal* solution but very *easy/fast to develop*. You can come back later to optimise further.

It *is* counter-intuitive, but optimising a large project in successive passes yields better results than focusing efforts on localised bottlenecks (even truer if the development time allocated to optimisation is fixed).