

# ACCELERATING EVOLUTION:

PARALLELISATION OF EVOLUTIONARY ALGORITHMS ON GPU



ASHIRBAD SARANGI

*SC23M002*

# CONTENT

---

OBJECTIVE

GPU PRE-REQUISITES

EVOLUTIONARY ALGORITHMS

PLAN OF ACTION

TIMELINE

# OBJECTIVE

To try and implement evolutionary algorithms and compare the run time in CPU and GPU

# GPU PRE-REQUISITES



# GPU PRE-REQUISITES

---

- GPU – **G**raphics Processing Unit
- Fast amount of dense matrix multiplication due to parallelism

CUDA **B**asic Linear Algebra **S**ubprograms



**cuBLAS**

# WHAT IS CUBLAS GOOD FOR ?

- Anything that uses heavy linear algebra computations (on dense matrices) can likely benefit from GPU acceleration
  - Graphics
  - Machine learning (this will be covered next week)
  - Computer vision
  - Physical simulations
  - Finance
  - etc.....
- cuBLAS excels in situations where you want to maximize your performance by batching multiple kernels using streams.
  - Like making many small matrix-matrix multiplications on dense matrices
- cuBLAS selected column-first indexing

# NUMPY VS CUBLAS

Numpy	math	cuBLAS (<T> is one of S, D, C, Z, H)
<code>numpy.matmul(<math>\alpha</math>, <math>\chi</math>)</code>	$(\lambda \mathbf{A})_{ij} = \lambda (\mathbf{A})_{ij}$	<code>cublas&lt;T&gt;gemm(<math>\alpha</math>, <math>\chi</math>)</code>
<code>numpy.dot(<math>\chi</math>, <math>\gamma</math>)</code> (Multiply arguments element-wise)	$(A \circ B)_{i,j} = (A)_{i,j} (B)_{i,j}$	<code>cublas&lt;T&gt;gemm(<math>\chi</math>, <math>\gamma</math>)</code>
<code>numpy.matmul(<math>\mathbf{A}</math>, <math>\chi</math>)</code>	$\mathbf{A}\chi = \mathbf{C}$	<code>cublas&lt;T&gt;gemm(<math>\chi</math>, <math>\mathbf{A}</math>)</code>
<code>numpy.matmul(<math>\mathbf{A}</math>, <math>\mathbf{B}</math>)</code>	$\mathbf{C} \leftarrow \alpha \mathbf{A}\mathbf{B} + \beta \mathbf{C}$	<code>cublas&lt;T&gt;gemm(<math>\mathbf{A}</math>, <math>\mathbf{B}</math>)</code>

# ARRAY INDEXING

The arrays are linearized into one dimension, so we will use an indexing macro.

```
#define IDX2C(i,j,ld) (((j)*(ld))+(i))
```

Where “i” is the row, “j” is the column, and “ld” is the leading dimension.

In column major storage “ld” is the number of rows.



# FEATURES

- All of the functions defined in cuBLAS have four versions which correspond to the four types of numbers in CUDA C
  - S, s : single precision (32 bit) real float
  - D, d : double precision (64 bit) real float
  - C, c : single precision (32 bit) complex float (implemented as a float2)
  - Z, z : double precision (64 bit) complex float
  - H, h : half precision (16 bit) real float
- Functions used for Matrix / Vector Multiplication :
  - cublasSgemm → cublas S gemm
  - cublasHgemm
  - cublasDgemv → cublas D gemv

# EVOLUTIONARY ALGORITHMS



# EVOLUTIONARY ALGORITHMS

- Evolutionary Algorithms (EAs) are a family of optimization algorithms inspired by the process of natural selection. They are used to find approximate solutions to optimization and search problems.
- Key Concepts :
  - Natural Selection:
    - Mimics the process of natural selection where individuals with favorable traits are more likely to survive and reproduce.
  - Population:
    - Solutions are represented as individuals in a population. Multiple solutions coexist and evolve over generations.
  - Crossover and Mutation:
    - Individuals undergo genetic operations like crossover (recombination) and mutation to create new offspring.
  - Fitness Function:
    - Measures the quality of an individual. Individuals with higher fitness values are more likely to contribute to the next generation.

# GENETIC ALGORITHM

Initialise population size, number of generations,  
cross over rate and mutation rate

Repeat till number of generations :

- check fitness

- Select Fit parents

- Generate Offsprings from cross over rate

- Mutate offsprings

- Replace Population

Get Best solution

# ANT COLONY OPTIMISATION

Initialize pheromone levels

Repeat for a fixed number of iterations or until a convergence criterion is met:

- Place ants at the starting point

- For each ant:

  - Construct a solution by probabilistically selecting components

- Update pheromone levels based on the constructed solutions

- Evaporate pheromones

# PARTICLE SWARM OPTIMIZATION

Initialise number of particles, number of parameters, max iterations, cognitive parameters, social parameters, inertia weight

```
particles = initialize_particles(num_particles, num_dimensions)
```

For each particle initialize best position and fitness both personally and Collectively as Globally

Until max iterations:

- Update velocity

- Update position

- Check fitness

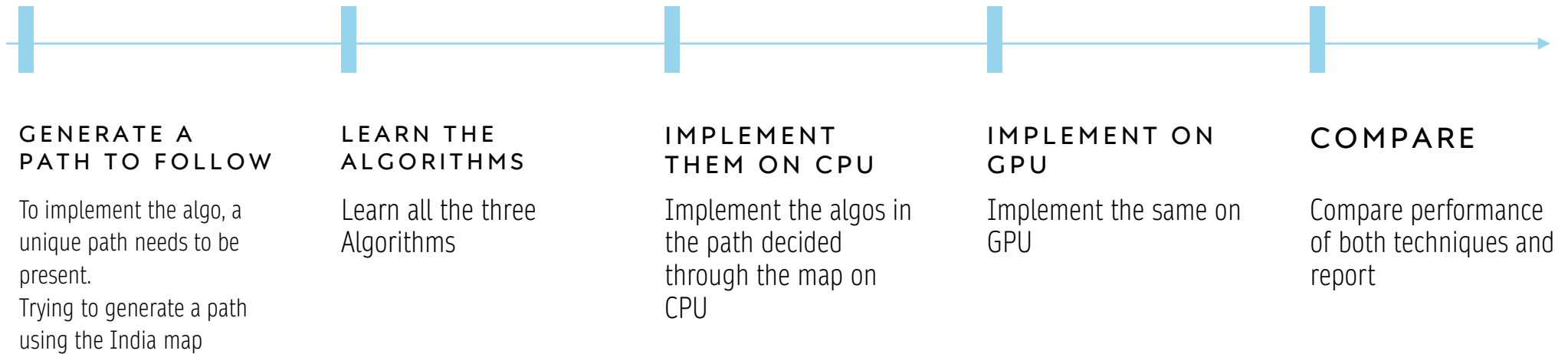
- Update personal best if current fitness is better than best fitness

- Update Global best

```
best_solution = global_best_position
```

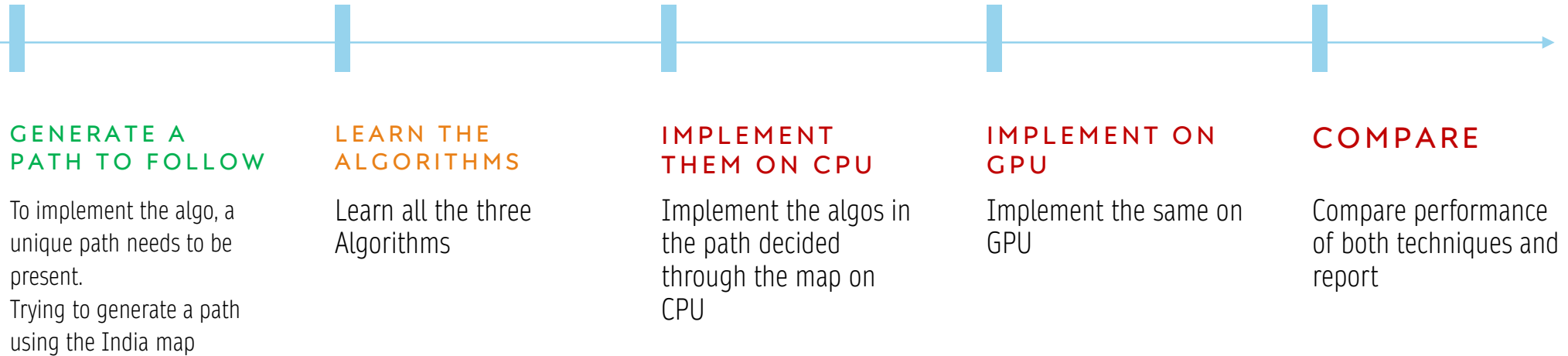
# PLAN OF ACTION





# PLAN OF ACTION

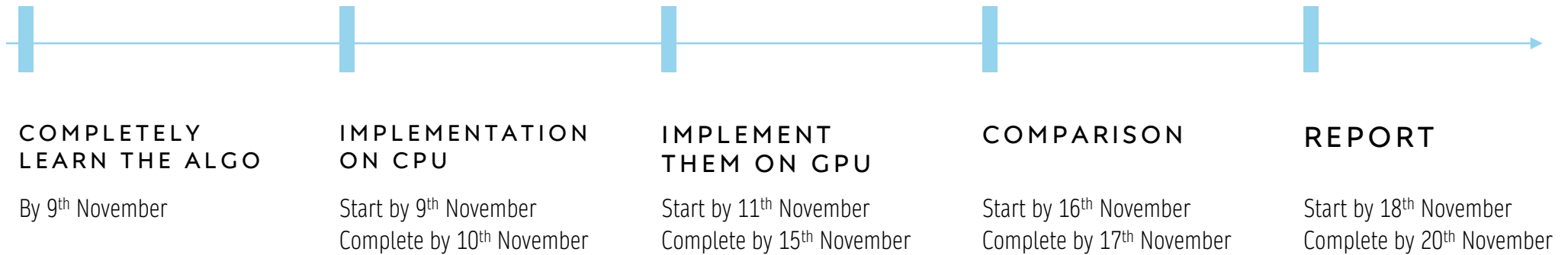




# PLAN OF ACTION

# TIMELINE





# TIMELINE

THANK YOU

