PHINEAS: An Embedded Heterogeneous Parallel Platform

Nikhil Khatri

Student - PES University
Intern - LinkedIn
nikhilkhatri97@gmail.com

Nithin Bodanapu

Student - PES University
Intern - Couchbase
nithinbodanapu97@gmail.com

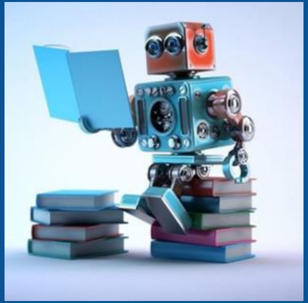
Dr. TSB Sudarshan

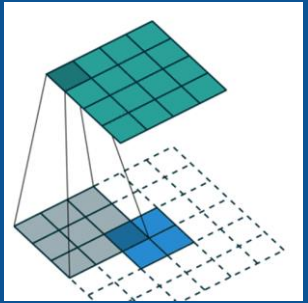
Dean of Research - PES University
sudarshan@pes.edu

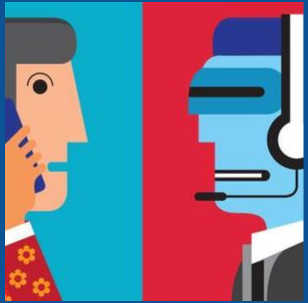
Motivation

- PES University currently does not have a cluster computer
- Students of HPC classes do not have a practical environment in which to apply their knowledge
- Machine learning, DIP and NLP are popular domains of research on campus.
  - Highly parallelizable
  - Frequently applied in embedded environments (Robots, embedded controllers)
  - No suitable hardware available for this task either

We need a parallel compute resource to meet the university's needs for ML, DIP and NLP







in **embedded environments**.

Cluster requirements

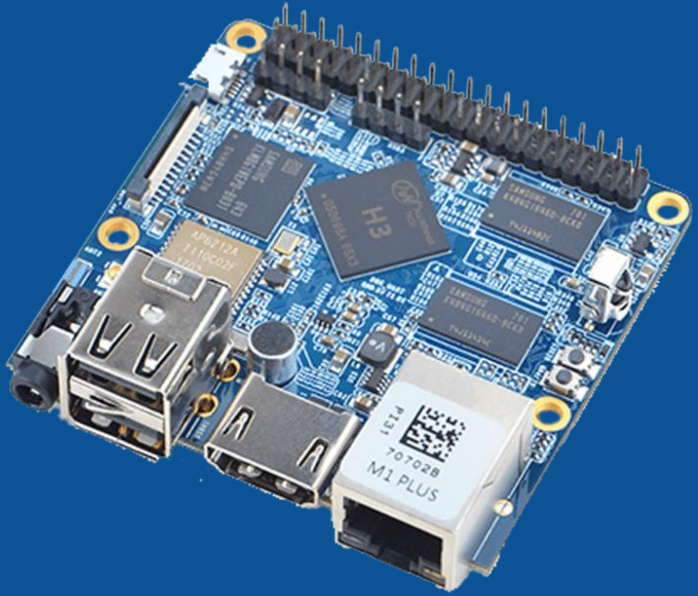
- Power efficient
- Suitably parallel
- Physically small
- Individually performant compute nodes (For single threaded workloads)
- Low latency interconnect

Infrastructure requirements

- ARM based SoC
- 1 GB+ RAM
- Gigabit networking
- Efficient power supply
- Sufficient storage

NanoPi M1 Plus

- Allwinner H3
    - ARM Cortex A7 CPU (x4)
    - Mali 400 MP2 GPU @ 600MHz
- 8 GB eMMC storage + microSD slot
- Gigabit networking
- I/O
    - Video
    - Audio
    - GPIO



Networking

2 x 8 port gigabit switches

Power supply

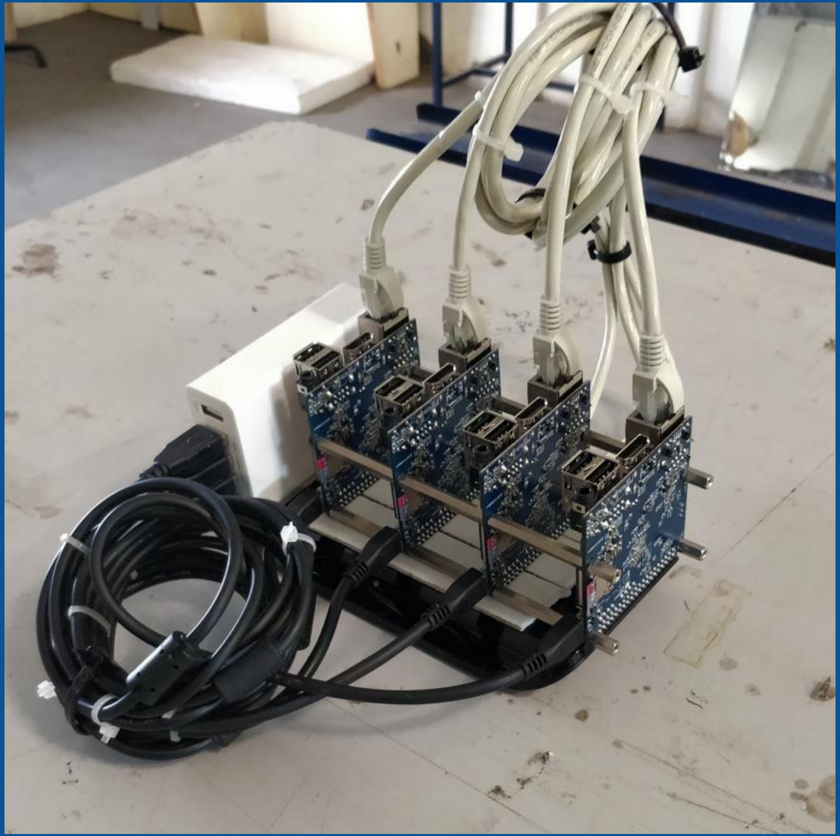
2 x 40 Watt USB power supplies

Cluster architecture

- 2 stacks of 4 nodes each
- Each stack has
  - 4 x NanoPi M1 Plus
  - Gigabit network switch
  - 5 port 40 Watt USB power supply
- Each stack is entirely independent
- Stacks can be added or removed freely (horizontal scaling)

Cost per stack: ₹ 16194 ~= $233

Dimensions: 25cm x 30cm x 30cm
(Mostly cabling)

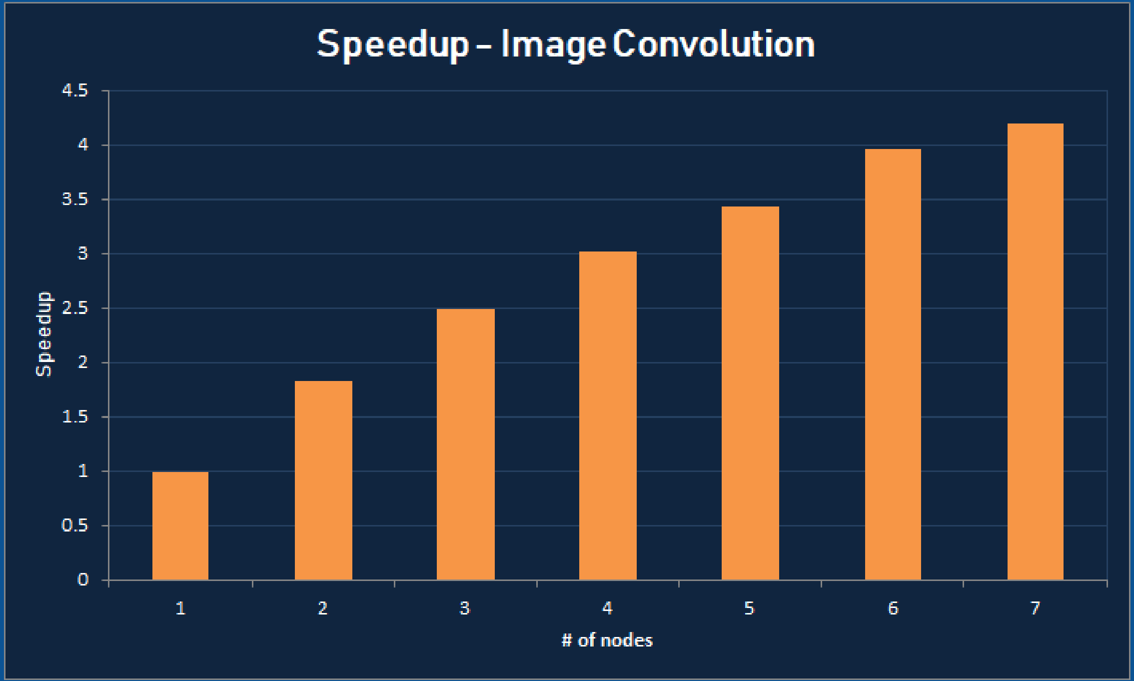


Performance Benchmarks

1. Image convolution
2. Matrix multiplication (hybrid)
3. DNN training

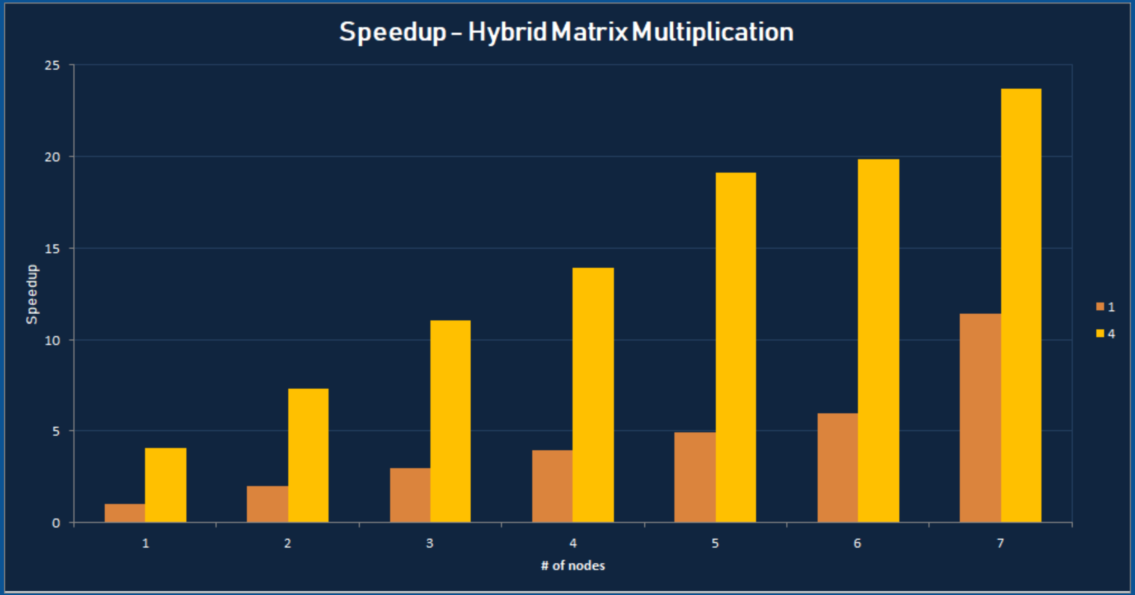
Image convolution

- Distributed program using MPI to count number of stars
- Master node reads in .TIF image (12788 x 40000)
- Partitions image vertically, sends to all nodes
- Each node runs cv2.adaptiveThreshold on its component
- Each node returns its local star count



Matrix multiplication (hybrid)

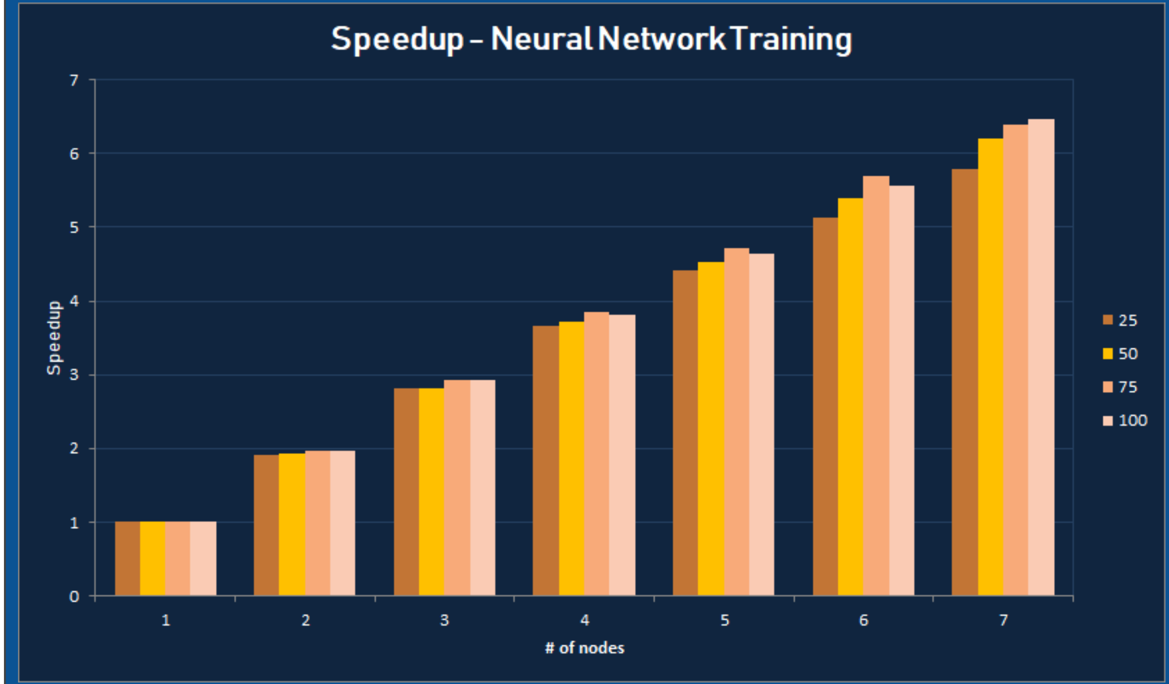
- C = A x B
  - each has dimensions NxN
- A is partitioned by columns, B is sent as a whole
- Each node forms one column of C
- Within each node, multiplication is parallelised using OpenMP
  - Static scheduling
  - 1 or 4 threads per node



Multiplication speedup for 1000 x 1000 matrices

Neural Network training

- Computation from each layer distributed across nodes
- Both forward propagation, and backpropagation are parallelised
- We see very good speedup even with just two nodes
- A wider hidden layer results in marginally better speedup
- Uses MPI4Py to distribute work



Can our boards do more?

Yes, with the GPU

Heterogeneous computation

- Our boards have Mali 400 MP2 GPUs
- These can be utilised through the OpenGL ES2.0 interface
- These have not been previously used for GPGPU computation
- This could increase the overall computational capability of the platform

OpenGL ES2.0

- Lightweight interface, meant for low intensity graphic processing
- 2 shaders per program
  - Vertex shader
    - Applies location transforms to pixels in viewport
    - We use this to tell fragment shader about adjacent points
  - Fragment shader
    - Applies texture to triangles in the viewport
    - Use this to read input image, and modify display

Example usage

- Image convolution using a 3x3 kernel
- Vertex shader is a passthrough to fragment shader
- Fragment shader performs computation and outputs data through the gl_FragColor variable.

Shaders

```
attribute vec4 vPosition;

void main() {
    gl_Position = vPosition;
};
```

```
uniform sampler2D inputImageTexture;
varying mediump vec2 blurCoordinates[6];

void main() {
    mediump vec4 sum = vec4(0.0);
    sum += texture2D(inputImageTexture, (vec2(gl_FragCoord[0], gl_FragCoord[1]) + vec2(-1, +1)) / 512.0) * 1.0;
    sum += texture2D(inputImageTexture, (vec2(gl_FragCoord[0], gl_FragCoord[1]) + vec2(-1, 0)) / 512.0) * 2.0;
    sum += texture2D(inputImageTexture, (vec2(gl_FragCoord[0], gl_FragCoord[1]) + vec2(-1, -1)) / 512.0) * 1.0;

    sum += texture2D(inputImageTexture, (vec2(gl_FragCoord[0], gl_FragCoord[1]) + vec2(1, 1)) / 512.0) * -1.0;
    sum += texture2D(inputImageTexture, (vec2(gl_FragCoord[0], gl_FragCoord[1]) + vec2(1, 0)) / 512.0) * -2.0;
    sum += texture2D(inputImageTexture, (vec2(gl_FragCoord[0], gl_FragCoord[1]) + vec2(1, -1)) / 512.0) * -1.0;
    gl_FragColor = sum;
};
```

GPU Neural Network Inferencing

- Neural networks are known to be a workload conducive to GPU computation
- None of the common GPU ML libraries are compatible with the OpenGL ES2.0 interface.

DNN Shader

```
#define MAX_OUTPUT 4096

precision mediump float;

uniform float weights[100];
uniform float inputs[10];

uniform int this_layer_width;
uniform int prev_layer_width;

void main() {

    float acc = 0.0;

    int neuron_number = int(gl_FragCoord[0]);
    int base_weight = neuron_number * prev_layer_width;

    int i;
    for(i=0; i<prev_layer_width; i++){
        acc += float(weights[base_weight + i]) * float(inputs[i]);
    }

    gl_FragColor = vec4(acc/MAX_OUTPUT, 0, 0, 1);
}
```

Challenges & Possibilities

- OpenGL ES2.0 does not provide an easy way to get output
- Output is through one RGBA color per fragment
  - 8 bits per color
  - No existing way to encode a 32 bit float in this
- One GPU does not give best performance
  - Distributed GPU computation would make up for this
  - Would require Higher level interface than ES 2.0

Conclusion

- Highly parallel computation can be achieved on an inexpensive efficient cluster.
- The on-board GPU can be further leveraged to extract performance

Thank You