PHINEAS: An Embedded Heterogeneous Parallel Platform

Nikhil Khatri(✉), Nithin Bodanapu(✉), and T. S. B. Sudarshan(✉)

Department of Computer Science and Engineering,
PES University, Bangalore 560085, India
nikhilkhatri97@gmail.com, nithinbodanapu97@gmail.com,
sudarshan.tsb@gmail.com

Abstract. With machine learning being applied to increasingly varied domains, the computational needs of researchers have increased proportionately. Hobbyists, researchers and universities are turning to building their own cluster computers to meet their high performance compute needs. These clusters are typically highly efficient, low cost ARM based platforms consisting of between 4 and 8 nodes. In this paper, we present PHINEAS: Parallel Heterogeneous INdigenous Embedded ARM System, a parallel compute platform which allows for distributed computation using MPI and OpenMP and which further leverages the on-board GPU to perform general purpose compute tasks. We describe the hardware components of the cluster, the software stack installed on each node and a host of common benchmark algorithms and their results. The results show that the cluster meets the stringent latency requirements of embedded systems. We further describe how the on-board GPU's OpenGL ES 2.0 programming model can be used to implement tasks such as image convolution and neural network inference which are common in intelligent embedded systems. Parallelisation of compute tasks across multiple GPUs is discussed as a method to combine the advantages of distributed and heterogeneous computing.

Keywords: Cluster computer · Embedded system · Heterogenous computer

1 Introduction

The class of platform that PHINEAS belongs to is frequently referred to as a Beowulf cluster [16]. These are described as "scalable performance clusters based on commodity hardware, on a private system network, with open source software (Linux) infrastructure" [12]. Common configurations typically include multiple nodes with the same hardware. This may be either the same processor, or, as in our case, the identical computer. Modern Beowulf clusters frequently make use of a class of computers titled SBCs (Single Board Computers). These include a processor, GPU, RAM, storage and I/O such as USB, ethernet and wireless communication all on one board. Most common clusters consist of anywhere

D. Abramson and B. R. de Supinski (Eds.): SCFA 2019, LNCS 11416, pp. 57–70, 2019.
https://doi.org/10.1007/978-3-030-18645-6_4

between 2 and 16 boards [1,15]. Extreme examples which consist of hundreds of nodes also exist. The RaspberryPi has emerged as the most commonly used SBC for Beowulf clusters. Its low cost, easy availability, power efficiency and excellent support make it a popular choice. Most literature on existing embedded parallel platforms provide data about performance scaling across nodes for common compute tasks such as matrix multiplication, image convolution and algorithms such as mergesort. However, there seems to have been little work done towards utilizing the GPUs on such boards for general compute tasks. To this end, we built **PHINEAS: Parallel Heterogeneous INdigenous Embedded ARM System**. This was one of the primary goals of our research: to use the on-board GPU to perform compute tasks.

2 Hardware and Construction

Any cluster like PHINEAS consists of three main components. Namely, the compute board, the power supply and the networking hardware. The power supply is usually dictated by the power draw associated with each board and the total number of boards in the cluster. The network switch must match the bandwidth afforded by the SBC and must provide enough ports to accommodate all boards and extra lines for extra-net connections.

2.1 Single Board Computer

The SBC chosen for the cluster affects performance more than any other part. When considering boards we used the following factors to guide our choice:

CPU. The CPU plays a lead role in all the computation done on the cluster. When comparing board CPUs, clock speed and number of cores play a critical role. Most of the boards considered had a clock speed greater than or equal to 1 GHz. Boards in this class typically have between 2 and 4 cores, with the rare exception having 8 cores. The number of cores defines the amount of parallelism we would be able to extract on each board using OpenMP.

GPU. The GPU is more important for our cluster than most other clusters since GPGPU compute is an important goal for us. However, this was not a critical point when making a choice since most boards have the same MALI 400 MP2 GPU clocked at around 500 MHz. A notable exception is the Raspberry Pi 3B+ which has a Broadcom GPU.

RAM. All boards we considered had 1 GB of RAM. This is critical for most applications which rely on data level parallelism since large data sets must be kept in memory and accessed frequently. Other than the Raspberry Pi, all boards have DDR3 RAM. The Raspberry Pi has the slower LPDDR2 generation of RAM.

Networking. Networking is typically a significant overhead in distributed computing. To minimize this overhead it is essential that a high bandwidth ethernet port is offered by the board. Most boards offer ethernet ports with gigabit speeds. A notable exception was the Raspberry Pi which has a maximum speed of about 300 Mbps [10].

Power. Since one of the key features of a embedded parallel platform is efficiency, low power draw for each component is a must. Most boards that we considered did not deviate significantly from the 5 Volt/2 Ampere mark.

Storage. An often neglected feature of these SBCs is the storage options offered by them. This is because seldom do these boards offer more than a single SDcard slot. An alternate option is eMMC memory. This may either be soldered onboard, or be connected as an external module. eMMC memory has much higher read throughput than an SD card and also has better write resiliency [11]. Storage sizes typically range between 8 GB and 32 GB, with 8 being the minimum required for most operating systems.

Chosen Board. Keeping these factors in mind, we selected the NanoPi M1 Plus for our cluster [7]. This was chosen for its gigabit networking, 1 GB of DDR3 RAM and most importantly, 8 GB of eMMC storage. The board has a Allwinner H3 SOC, which has a quad-core Cortex-A7 CPU and a Mali-400 MP2 GPU. The availability of the board in India contributed strongly to our choice. Other boards we considered were the Raspberry Pi 3B+ [10], the Pine A64+[9] and the NanoPi Fire 3 [6]. It is important to note that our decision was based on the specifications of the various boards and the data provided by others. We did not benchmark these boards ourselves (Table 1).

2.2 Power Supply

For powering the entire cluster it is necessary that the chosen power supply is able to provide sufficient voltage to each node while also being able to provide the necessary total wattage of the cluster. A stable power source is hence necessary for consistent operation. We chose to use two 5-port USB hubs, each of which provides up to 40 W of power which is sufficient for 4 boards.

2.3 Network Switch

For communication among the nodes of the cluster it is essential to have a switch that can make use of the gigabit ethernet ports on the compute boards so as to avoid any network latency during computation. This trend is common in clusters where the boards support full gigabit networking [1]. Another technology available for ethernet is PoE - Power over Ethernet. This allows us to fuse the power delivery and networking into a single connection backbone. The drawback

of this is, that it limits gigabit connection to 100 Mbps, since the remaining lines are used for power delivery. Further, of the boards we considered, only the raspberry pi 3b+ supports PoE. Even this requires a special PoE HAT (Hardware Attached on Top) and a PoE capable switch which are typically more expensive.

2.4 PHINEAS Specification

The PHINEAS cluster consists of two stacks, each consisting of 4 NanoPi M1 Plus boards, with a 8-port gigabit switch and a USB power supply rated at 40W. Each stack has approximate dimensions 35 cm × 25 cm × 25 cm, making it suitable for embedded systems. The cluster has no moving parts and is thus structurally stable. If the eMMC storage is used as the boot partition, the microSD card can also be removed, resulting in a system without any loose components.

Table 1. Board specification comparison

Feature	Raspberry Pi 3 B+	Pine A64+	NanoPi Fire3	NanoPi M1 Plus
Processor	Broadcom CortexA53 (1.4 GHz) x 4	Cortex A53 (1.2 GHz) x 4	S5P6818 (1.4 GHz) x 8	Cortex A7 (1.2 GHz) x 4
GPU	Broadcom Videocore 4	Mali 400 MP2	Mali 400 MP4	Mali 400 MP2
RAM	1 GB LPDDR2	1 GB DDR3	1 GB DDR3	1 GB DDR3
Ethernet	Gigabit (300 Mbps)	Gigabit	Gigabit	Gigabit
Power draw	5 V/2.5 A	5 V/2 A	5 V/2 A	5 V/2 A
Storage	microSD	microSD	microSD	eMMC + microSD

3 Software Stack

In our cluster, we reserved one board for research involving the GPU and the remaining were used for the tests described in the next section. This is because the manufacturer recommends a different OS distribution when writing code for the GPU. Note: There are no hardware differences between the boards. For the 7 benchmark boards, we installed Linux 4.14 based on the mainline kernel. This was provided by FriendlyARM, the board manufacturer. For parallel programming within the board, we installed OpenMP. For distributing work across boards, we installed MPICH3 which is an implementation of the Message Passing Interface. This provides an easy to use API when distributing workloads across multiple machines connected over a network.

To use the GPU, we installed the Linux-3.4 OS image which was provided by Allwinner, the manufacturer of the SOC.

4 Performance Benchmarks

As described in the previous section OpenMP and MPI were used to distribute the workload across the cluster. Various common algorithms were run on the PHINEAS cluster using 1, 2 .. 7 nodes incrementally. The time taken for each execution was recorded and plotted against number of nodes used. This provides us with an understanding of the speedup achieved for various workloads. The programs suitably exert the CPU, memory and networking of the cluster and its computers.

4.1 Monte Carlo Pi Estimation

Monte Carlo Pi estimation is a method to estimate the value of pi based on randomly generated values [5]. It calculates the ratio of number of points lying inside a circle against the number of points lying inside a square. We run multiple iterations to randomly generate points in two dimensional space and to parallelize it we distribute a chunk of iterations to each node. This test was done solely to show that for a large enough problem size we can achieve close to ideal speed up. In our case it was 6.89 for a 7 node cluster. The speedup graph obtained on PHINEAS is provided in Fig. 1. The X-axes shows the number of nodes used to distribute the workload of 1000000, 10000000 and 100000000 iterations to estimate Pi.

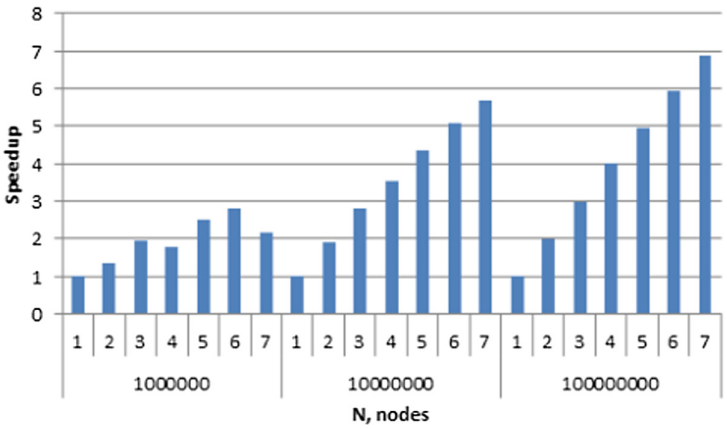

Fig. 1. Speedup observed for distributed Monte Carlo Pi estimation

4.2 Distributed Merge Sort

Merge sort is a simple sorting algorithm that has the added advantage of easily being parallelisable. The reason for using this as a benchmark is that it is a common algorithm that places a heavy load on the network as it requires the scatter

and gather of large array chunks across all the nodes of a cluster. The speedup graph obtained on PHINEAS is provided in Fig. 2. The x-axis shows the number of nodes used to distribute the workload of N (10000, 100000 and 1000000) elements in the array which is to be sorted. An existing C implementation using MPI was used [3].

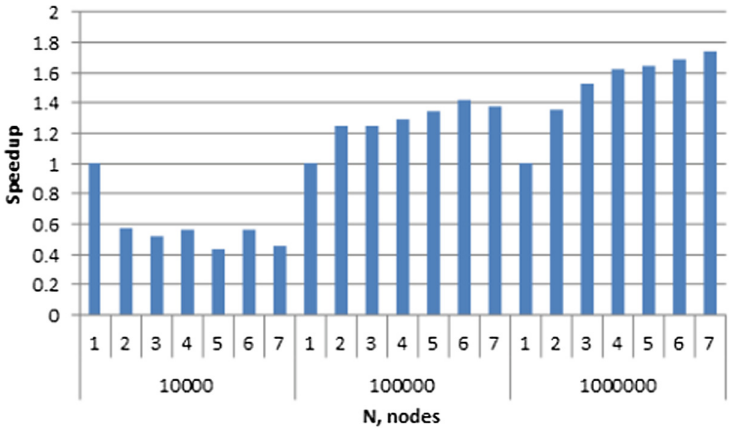

Fig. 2. Speedup observed for distributed Merge Sort

4.3 Image Convolution

One of the main applications for this cluster was robotics which involves frequent use of computer vision. One of the main tasks in computer vision is image convolution. Images can be convolved in an immensely parallel manner, where each pixel can theoretically be processed in parallel. The benchmark program works by dividing the image into vertical columns, applying the convolution filter on each of these and merging the segments of the image to get the final output. For the convolution the OpenCV library for Python was used [2]. The speedup graph obtained on PHINEAS is provided in Fig. 3. The X-axis shows the number of nodes across which the image to be convolved is distributed in a column by column fashion.

4.4 Hybrid Matrix Multiplication

We explored a hybrid approach of using OpenMP and MPI. This allowed us to parallelize workload across a cluster and also across all the cores of a processor, making full use of our Quad-Core CPUs. This is known to be an ideal method to distribute workload and optimize resources utilization. For a problem of $A \times B$ we send the matrix B to all the nodes and send a subset of rows of A to each node of the cluster. Within each node we compute the multiplication of rows to each column of B. Here there is scope for parallelization so that a row in A can

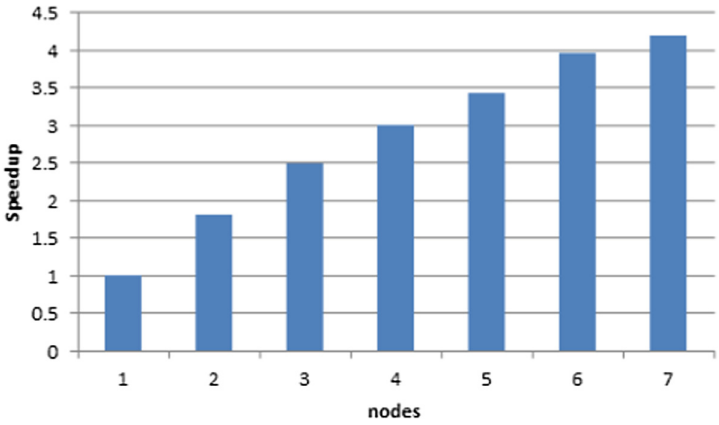

Fig. 3. Speedup observed for distributed image convolution

be multiplied by different columns of B simultaneously using multiple threads. This allows us to efficiently utilize all the resources available to us. In Figs. 4 and 5 we show the speedup obtained when using a single thread and multiple threads [4]. The X-axis shows the distribution of workload across the 7 nodes of PHINEAS with 1 and 4 threads used per node. The tabular form of the same data is given in Table 2.

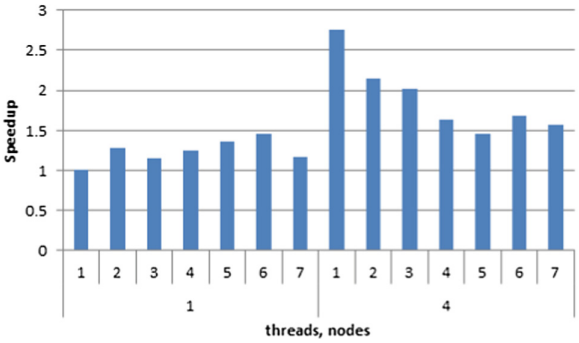

Fig. 4. Speedup observed for hybrid matrix multiplication with size of the matrices as 100×100

4.5 Neural Network Training

Deep Learning is one of the most researched areas in current times and is highly compute intensive and parallelisable. Taking this into consideration we parallelized an existing dense neural network built on python using the numpy library to train on the MNIST dataset. This network was parallelized using pyMPIch,

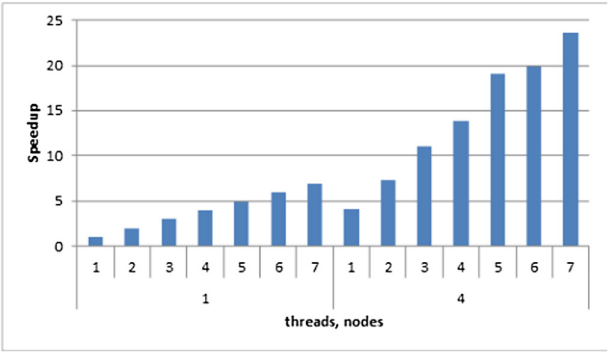

Fig. 5. Speedup observed for hybrid matrix multiplication with size of the matrices as 1000×1000

Table 2. Time for matrix multiplication under varying sizes and constraints like Matrix size, number of threads and number of nodes

Nodes	Threads	N	Time	Nodes	Threads	N	Time
1	1	100	0.109478	1	1	1000	159.870361
2	1	100	0.085684	2	1	1000	79.573329
3	1	100	0.095737	3	1	1000	53.578156
4	1	100	0.087392	4	1	1000	40.160633
5	1	100	0.080584	5	1	1000	32.261839
6	1	100	0.075198	6	1	1000	26.876678
7	1	100	0.094543	7	1	1000	23.273858
1	4	100	0.039621	1	4	1000	39.305208
2	4	100	0.051147	2	4	1000	21.83931
3	4	100	0.054234	3	4	1000	14.459377
4	4	100	0.067129	4	4	1000	11.512601
5	4	100	0.07509	5	4	1000	8.369552
6	4	100	0.064968	6	4	1000	8.057398
7	4	100	0.069967	7	4	1000	6.747061

a wrapper for the MPI library implemented in C. Hardware used for efficient training of neural networks involves the use of expensive GPUs especially for large complex networks. Such hardware is typically expensive and not easily available. Our aim is to provide a cost effective solution to this problem. The benchmark that we ran was of a dense neural network tested with 25, 50, 75 and 100 hidden units and we have seen that the speedup increases as the number of hidden units increases [8]. The idea behind showcasing this as a benchmark is to show how we can make use of a simple embedded system to reduce training time in a neural network. The speedup graph is given in Fig. 6. The X-axis consists of

number of nodes the workload is distributed across different number of hidden units in the neural network.

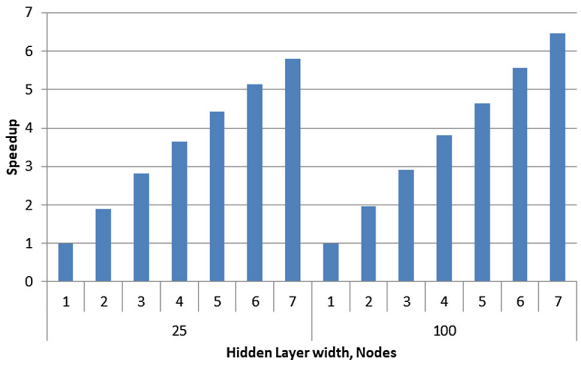

Fig. 6. Speedup observed for distributed dense neural network training with 25 hidden units

5 Graphics Processing Unit

A key goal of our project was to use the on-board GPU to perform general purpose computation, since this has not been achieved by any cluster of this class previously. The GPU on the NanoPi M1 Plus is the Mali-400 MP2. This, being a slightly older and lower performance GPU does not support modern programming environments such as OpenCL. This greatly hampers the ability to carry out non-graphical computation on the board. The API supported by the GPU is OpenGL ES 2.0 which is described in greater detail in the following subsection.

5.1 OpenGL ES 2.0

OpenGL ES is a cross-platform API for rendering 2D and 3D graphics on embedded and mobile systems. The 2.0 variant was the first API to support programmable shaders for a mobile or embedded environment [13]. When writing a program using OpenGL ES 2.0, output is controlled primarily through the two shaders: the vertex shader and the fragment shader.

Vertex Shader. The input to the vertex shader is a set of vertex attribute objects. One vertex is provided as input to each instance of the vertex shader. Each vertex consists of 4 attributes, representing location in x, y and z coordinates, and a fourth coordinate used for projection and transformations. The vertex shader is responsible for transforming or re-positioning the input vertex and providing a single transformed output vertex. On the Mali-400 MP2 there is

one physical vertex shader. For our purposes, we do not perform any computation in the vertex shader. We simply provide the vertices of two triangles, which between them cover the entire screen.

Programming for both shaders is done using GLSL ES (OpenGL Shading Language), a C-like language which provides data types, mathematical operations and inbuilt variables to aid in programming the shaders [13]. Example code for the vertex shader is shown.

```
attribute vec4 vPosition;

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

vPosition provides the input vertex coordinates. These are simply copied over to *gl_Position*, which is the variable where the output vertex must be placed for each vertex shader instance.

Fragment Shader. After the vertex shader, the primitive shapes undergo rasterization to generate fragments, each of which has a specific depth and color value. It is the fragment shader's responsibility to color each fragment using its coordinates in the window and other optional variables such as textures and samplers.

On the Mali-400 MP2 there are 2 physical fragment shaders. This is where we perform the majority of our computation. A simple example of a fragment shader is shown.

```
void main(){
        gl_FragColor = vec4(1.0, 0, 0, 1.0);
}
```

This shader simply colors all fragments red. The *gl_FragColor* variable expects an output in RGBA form (Red, Green, Blue and Alpha) [13].

5.2 Image Convolution

One task which easily lends itself to GPU computation is image convolution. Convolution involves performing a matrix product of a given kernel (or convolution matrix) with each mxm submatrix of the image. Kernels may be 1D, 2D or 3D in cases where color is also used. This task is inherently parallel since the kernel can be applied to each submatrix of the image independently. To perform this task, the first challenge is how to provide an image to the fragment shader, where we intend to perform our computation. To do this, we make use of a feature of OpenGL called textures. We are able to treat an image as a texture and sample from it in the fragment shader. To bind the image as a texture, we load the image outside of the shaders, in our C code. To do this, we first generate

a texture object, bind the new texture and copy over the data from the image, which we read in from a bitmap (.BMP) file. We can then access this texture from the fragment shader through a $sampler2D$ variable. In our example, we use the Sobel filter for the x coordinate. This is a kernel which can detect vertical lines. For our implementation, we chose to hardcode the kernel values, but these could easily be encoded as another texture or by using the $glGetUniformLocation$ function of OpenGL. The code for the Sobel filter is shown. Applying the sobel filter to a 1920×1080 pixel image, we consistently saw frame-rates in excess of 35 FPS, which is sufficient to meet real-time requirements.

5.3 Neural Network Inferencing

With the recent boom in computational power, machine learning techniques have gained massive popularity and are applied to increasingly diverse domains. Neural networks in specific are applied to a variety of domains with great success. Digital image processing has seen great advancement through the use of neural networks. This has trickled through to embedded systems where machine learning powered image processing algorithms are used for obstacle avoidance and human interaction. However, there has been limited work towards implementing neural networks on GPUs for embedded systems. In addition, this has never been discussed in the context of ARM based embedded clusters. In this section we describe a simple implementation for a dense fully connected neural network. The goal of this is to show that it is possible to implement fairly complicated networks using the restrictive OpenGL ES 2.0 API. In our implementation, all inner products involved in a single layer are handled in parallel by the fragment shader. The input for each layer is prepared and passed in by the draw loop in the C program. The output of each layer is also parsed by the same.

In a neural network, a layer is made up of multiple neurons. Each neuron takes in a vector of the previous layer's output and performs an inner product of this with a weight vector. The result of this inner product is typically passed through an activation function such as a RelU or sigmoid function.

In a single layer of a neural network, each node's output is independent and can be calculated concurrently. Thus, we assign one fragment shader to each node of a layer. For this, each fragment shader needs a way to locate its input values and its weights. For our implementation, we provide both weights and inputs in 1 dimensional arrays to the fragment shader. This is done in the C code by getting a uniform location using $glGetUniformLocation$ which provides a named location which can be accessed by the fragment shader. Then, before calling the rendering pipeline for each layer, we populate the weights, previous layer's output and metadata concerning the size of the current and preceding layer in uniform locations. In the fragment shader, we loop over all outputs of previous layer, multiply these with the appropriate weights and accumulate these in a local variable. This accumulated variable is then output through the red channel of the output color vector.

```
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 i;
        int neuron_number = int(gl_FragCoord[0]);

        for(i=0; i<prev_layer_width; i++){
                acc += float(weights[neuron_number
                                * prev_layer_width + i])
                        * float(inputs[i]);
        }
        gl_FragColor = vec4(acc/255.0, 0.0, 0.0, 0.0);
}
```

In this example 255 is used to normalise the output of each layer to a value less than 1. This value must be changed depending on the expected maximum output of each layer to prevent clipping. Setting this to a very large value however would lead to loss of precision.

While this paper does not discuss the performance of the system, some considerations regarding performance are fundamental to the final design. Compilation of the vertex and fragment shaders is a fairly expensive task and hence should be minimized. In our implementation, we are able to use a single vertex shader and a single fragment shader, both of which are compiled only once at the very beginning. Further, it is efficient to have a single size for the viewPort since then only a single viewPort of the required dimension is created. For this reason, we chose to create a viewPort of the dimension $largestlayerwidth \times 1$. This would ensure that for each layer, we would have at least as many fragment shader instances as the number of neurons in the layer. Some layers have fewer neurons than the maximum. For these, we avoid placing an if within the fragment shader as this reduces the efficiency on a SIMD processor such as a GPU [14]. Instead, we simply calculate these redundant values and ensure we do not use them when we process the output in the C program. Further, it is essential that we provide a single constant size to the weight and input arrays in the fragment shader. For this, we ensure that the input array is as large as the widest layer, and the weights array is as large as the maximum product of widths of two consecutive layers.

5.4 Usability

In its current form, with the limited interface provided by openGL ES 2.0, it is the view of the authors that any potential speedup gained by using the GPU will

be offset by the increased effort required to write a suitable fragment shader and to develop efficient code to communicate with the GPU. To make achieving this speedup less taxing, some form of lightweight framework on top of the existing OpenGL ES 2.0 would be imperative. Further, if a program were able to utilise all 8 GPUs simultaneously, one could expect a significant speedup. This would merit the extra effort required in writing such a program.

Acknowledgments. The authors would like to thank Dr. Kiran D C of Presidency University-Bangalore. His original work towards an embeddable cluster provided the basis for this work. The authors would also like to thank PES University for providing the funding necessary for building PHINEAS.

References

1. 96-core arm supercomputer using the nanopi-fire3. https://climbers.net/sbc/nanopi-fire3-arm-supercomputer/. Accessed 30 Sept 2018
2. Distributed image convolution. https://github.com/arundasan91/MPI---Message-Passing-Interface/blob/master/Image-Scatter-Gather-Tutorial.md
3. Distributed merge sort. https://github.com/racorretjer/Parallel-Merge-Sort-with-MPI/blob/master/merge-mpi.c
4. Hybrid matrix multiplication. http://assets.duet.to/dkl.cs.arizona.edu/teaching/csc522-fall16/examples/hybrid-openmp-mm.c
5. Monte carlo estimation. https://github.com/kiwenlau/MPIPI/blob/master/Montecarlo/mpipi.c
6. NanoPi Fire3. http://wiki.friendlyarm.com/wiki/index.php/NanoPi_Fire3. Accessed 30 Sept 2010
7. NanoPi M1 Plus. http://wiki.friendlyarm.com/wiki/index.php/NanoPi_M1_Plus
8. Neural network training. https://github.com/DT42/neural-network-model-manipulations/blob/master/mnist-nn-data-parallelism.py
9. PINE A64+/PINE A64. https://www.pine64.org/?page_id=1194. Accessed 30 Sept 2010
10. Raspberry Pi 3 model B+ product page. https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/. Accessed 30 Sept 2018
11. What is eMMC memory – software support - reliance nitro. https://www.datalight.com/solutions/technologies/emmc/what-is-emmc. Accessed 30 Sept 2018
12. What's a beowulf? http://www.beowulf.org/overview/faq.html
13. M̃unshi, A., G̃insburg, D., Shreiner, D.: OpenGL ES 2.0 Programming Guide. Pearson, London (2009)
14. Fung, W.W., Sham, I., Yuan, G., Aamodt, T.M.: Dynamic warp formation and scheduling for efficient GPU control flow. In: 40th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 407–420 (2007)
15. Kiepert, J.: Creating a Raspberry Pi-based beowulf cluster, pp. 1–17. Boise State University (2013)
16. Sterling, T.L.: Beowulf Cluster Computing with Linux. MIT Press, Cambridge (2002)