Creating a Raspberry Pi-Based Beowulf Cluster

Joshua Kiepert

June 20th, 2013

BOISE STATE UNIVERSITY

Introduction

Raspberry Pis have really taken the embedded Linux community by storm. For those unfamiliar, however, a Raspberry Pi (RPi) is a small (credit card sized), inexpensive single-board computer that is capable of running Linux and other lightweight operating systems which run on ARM processors. Figure 1 shows a few details on the RPi's capabilities.

[Figure 1]

The RPiCluster project was started a couple months ago in response to a need during my PhD dissertation research. My research is currently focused on developing a novel data sharing system for wireless sensor networks to facilitate in-network collaborative processing of sensor data. In the process of developing this system it became clear that perhaps the most expedient way to test many of the ideas was to create a distributed simulation rather than developing directly on the final target embedded hardware. Thus, I began developing a distributed simulation in which each simulation node would behave like a wireless sensor node (along with inherent communications limitations), and as such, interact with all other simulation nodes within a LAN. This approach provided true asynchronous behavior and actual network communication between nodes which enabled better emulation of real wireless sensor network behavior.

For those who may not have heard of a Beowulf cluster before, a Beowulf cluster is simply a collection of similar, (typically) commodity computer hardware based systems, networked together and running some kind of parallel processing software that allows each node in the cluster to share data and computation. Typically, the parallel programming software is MPI (Message Passing Interface), which utilizes TCP/IP along with some libraries to allow programmers to create parallel programs that can split a task into parts suitable to run on multiple machines simultaneously. MPI provides an API that enables both asynchronous and synchronous process interaction. My simulation did not require the computational power of a Beowulf cluster; it simply required a many distinct computers on a common network. A cluster environment provided a very convenient development platform for my distributed simulation thanks to its common file system and uniform hardware.

So, why I would want to build a Beowulf cluster using Raspberry Pis? The Raspberry Pi has a relatively slow CPU by modern standards. It has limited RAM, slow USB-based 10/100 Ethernet, and its operating system runs directly on a SD card. None of these “features” are ideal for a cluster computer! Well, there are several reasons. First, when your dissertation work requires the use of a cluster it is nice to ensure that there is one available all the time. Second, RPis provide a unique feature in that they have external low-level hardware interfaces for embedded systems use, such as I^2C, SPI, UART, and GPIO. This is very useful to electrical engineers (like myself) requiring testing of embedded hardware on a large scale. Third, having user-only access to a cluster (which is the case for most student-accessible systems) is fine if the cluster has all the necessary tools installed. If not however, you must then work with the cluster administrator to get things working. Thus, by building my own cluster I could directly outfit it with anything I might need. Finally, RPis are cheap! The RPi platform has to be one of the cheapest ways to create a cluster of 32 nodes. The cost for an RPi with an 8GB SD card is ~$45. For comparison, each node in one of the clusters available to students here at BSU, was about $1,250. So, for not much more than the price of one PC-based node, I could create a 32 node Raspberry Pi cluster!

One thing to keep in mind when building a cluster with RPis is to set expectations of performance appropriately. That is, we cannot expect the compute performance to be very good, even when compared to a single modern multi-core desktop computer. This was not a problem for my application because I just needed many independent processors running on a common network to run my distributed application. I didn't need a platform for high performance computing (HPC). Additionally, since the RPi uses an ARM processor, it has a different architecture than PCs, i.e. ARM vs x86. Thus, any MPI program created originally on x86 must be recompiled when deployed to the RPiCluster. Fortunately, this issue is not present for Java, Python, or Perl programs. Finally, because of the limited processing capability, the RPiCluster will not support multiple users simultaneously using the system very well. As such, it would be necessary to create some kind of time-sharing system for access if it ever needed to be used in such a capacity.

In summary, the RPiCluster is great if your development is focused only on distributed computing/network programming rather than parallel processing. That is, if the programs being developed for the cluster are distributed in nature, but not terribly CPU intensive. Compute-intensive applications will need to look elsewhere, as there simply is not enough “horse power” available to make the RPi a terribly useful choice for cluster computing. Of course, even compute-intensive applications could be tested on a small scale with the RPiCluster.

Building the System

There are really only five major components needed for a working cluster: computer hardware, Linux OS, an MPI library, an Ethernet switch, and possibly a router. Figure 2 shows the overall network architecture. The RPiCluster design consists of 32 RPi nodes, a 48-port 10/100 switch, Arch Linux ARM, and MPICH3. The total cost of the system was $1,967.21.

[Figure 2]

There are several operating systems available as pre-configured Linux images for the RPi: Raspbian “wheezy” (based on Debian) and Arch Linux for ARM (aka “alarm”) (raspberrypi.org). I chose to use Arch Linux for the RPiCluster. Arch Linux was chosen primarily for its minimalist approach which allowed me to install only what I needed. It also resulted in a system that boots in about 10 seconds. There is definitely a learning curve with Arch Linux, but fortunately it has some of the best organized information out there for learning Linux. An excellent place to start is the Arch Linux Beginner’s Guide.

As for an MPI implementation, I chose MPICH primarily due to my previous familiarity with it (OpenMPI is also available). On Arch Linux MPICH must be compiled from source, or installed from the Arch User Repository (AUR). I went with the compile-from-source method. You can get the source from mpich.org. OpenMPI is available directly in the Arch Linux *extra* repository.

One thing I found helpful when creating an Arch Linux image for the RPiCluster was QEMU. QEMU is a CPU architecture emulator and virtualization tool. It is capable of emulating an ARM architecture similar to the RPi. This allows us to boot an RPi image directly on our x86 system. There is one caveat: you need an RPI kernel for QEMU that knows how to deal with the QEMU virtualized hardware, which is not normally part of the RPi kernel image. This involves downloading the Linux kernel source, applying a patch to support the RPi, adding the various kernel features and drivers needed, and compiling.

There are several tutorials available which provide details for running QEMU with an RPI system image. A couple references I found useful are Raspberry Pi under QEMU and Raspberry Pi with

Archlinux under QEMU. Aside from the need for a custom kernel, there are a couple system configuration changes needed within the RPi image to allow it to boot flawlessly. The changes primarily have to do with the fact that RPi images assume the root file system is on /dev/mmcblk0p2 and the boot partition is on /dev/mmcblk0p1. QEMU makes no such assumptions so you have to map /dev/sda devices to mmcblk0 devices on boot. With the RPi system image adjusted and the custom kernel built, starting QEMU is something like the following:

```
$ qemu-system-arm -kernel ./zImage -cpu arm1176 -m 256 -M versatilepb -no-reboot
-serial stdio -append "root=/dev/sda2 panic=0 rw" -hda archlinux-hf-2013-02-11.img
```

Once you have a kernel image (zImage) that is suitable for QEMU you can point it at the new kernel and the RPi system image. Figure 3 shows the boot screen of Arch Linux ARM under QEMU.

[Figure 3]

With a working Arch Linux image under QEMU, I applied system updates and installed the necessary packages for the cluster environment. Once configuration of the image was completed with QEMU, I was able to simply write the resulting image to each RPi SD card after modifying host name and static IP address, unique to each (I wrote a bash script to do this).

Below are a few of the steps involved for configuring an Arch Linux image for use in a cluster:

1. Change root password from default (root):
   ```
   # passwd
   ```
2. Full system update:
   ```
   # pacman -Syyu
   ```
3. Install typically needed packages, (bold packages are need for using NFS, OpenMPI, and software development:
   ```
   # pacman -Syy nfs-utils base-devel openmpi sudo adduser gdb
   ```
4. Set the timezone for your location (<city>):
   ```
   # timedatectl set-timezone America/<city>
   ```
5. Add a new user (<user>) and be sure to add groups wheel, audio, video, and uucp when prompted:
   ```
   # adduser <user>
   ```
6. Allow users that are part of the wheel group to use superuser permissions:
   ```
   # chmod 600 /etc/sudoers
   ```
7. Uncomment the line: %wheel = ... and save:
   ```
   # nano /etc/sudoers
   ```
8. Logout of root and login as the user who will be using MPI and generate ssh keys for password-less login:
   ```
   $ ssh-keygen -t rsa
   $ ssh-keygen -t dsa
   $ ssh-keygen -t edsa
   ```

Strictly speaking, there is nothing special needed for a working cluster other than MPI and SSH, however a shared file system makes life much easier (e.g. NFS). Cluster setup is basically just configuring a few files so that all nodes can find each other by name on the network and have password-less SSH access to one another. Key configuration files for cluster setup under Arch Linux:

NFS:	/etc/exports
NFS:	/etc/idmapd.conf
NFS:	/etc/fstab
SSH:	/home/<mpiuser>/.ssh/authorized_keys
Network:	/etc/hosts
Network:	/etc/hostname
Network:	/etc/network.d/ethernet-static
Network:	/etc/conf.d/netcfg

For the fine details on NFS, network, and SSH setup, see the Arch Wiki: NFS, Network, SSH.

Rack and Power Design

One aspect of the cluster design that required quite a lot of thought was the rack mounting system and power distribution method. In order to keep the cluster size to a minimum while maintaining ease of access, the RPis were stacked in groups of eight using PCB-to-PCB standoffs with enough room in between them for a reasonable amount of air flow and component clearance. This configuration suited our needs for power distribution very well since it allowed for a power line to be passed vertically along each stack. Using this orientation, four RPi stacks were assembled and mounted between two pieces of ¼" Plexiglas. This created a solid structure in which the cluster could be housed and maintain physical stability under the combined weight of 32 Ethernet cables. Figure 4 (a) shows some initial experimentation using this method of mounting. The Plexiglas layout for mounting each RPi stack was designed using EagleCAD, along with the power/LED PCB. Figure 4 (b) shows the layout.

[Figure 4]

There are two methods of powering an RPi. Each RPi has a microUSB port that enables it to be powered from a conventional powered USB port. Using this method would require a microUSB cable and a powered USB port for every RPi in the cluster, which would add a lot more cabling to deal with. It also means that there would be cabling on both sides of the cluster. Alternatively, there is an I/O header on the side of the RPi which contains a 5V pin that can be used to power the board externally. The latter option was chosen as it would also allow for additional customization and control of each node. A custom PCB was created to fit the I/O header to provide power and an RGB LED (since electrical engineers must, of course, have LEDs to show that their project is working). Figure 5 (a) and (b) show the circuit and PCB layouts, respectively.

[Figure 5]

As seen in Figure 5 (a), aside from a RGB LED and some connectors, there is also a poly fuse (PF1). This was included to maintain short-circuit protection in the event of a board failure. The RPi already has a poly fuse between the USB power connector and the 5V rail. That fuse is bypassed when using the 5V pin to power the board. JP1 provides a 5V interconnect vertically between the RPis in each stack.

With the power being directly distributed via the Power/LED board, it was necessary to find a good source of 5V with sufficient amperage to drive the whole cluster. Each RPi draws about 400mA at 5V (2W), thus we needed a minimum of 13A of 5V (65W) (and more for overclocking). You could, of course, buy a dedicated high output 5V power supply, but a great option is to use a standard PC power supply. PC power supplies are already designed for high loads on their 5V rails, and they are relatively cheap. The 430W Thermaltake (430W combined output for 5V, 12V, etc) we selected is rated to provide 30A at 5V (150W) and cost $36. We opted to purchase two supplies to keep the overall load very low on each and allow for overclocking or future expansion.

Final Tweaks

As mentioned earlier, the RPi processor supports overclocking. Another noteworthy feature is that the 512MB of RAM is shared by both GPU and CPU, and the amount of RAM reserved for the GPU is configurable. Both of these features are configurable through modification of parameters in /boot/config.txt. After some testing, the whole cluster was set to run in “Turbo Mode” which overclocks the ARM core to 1GHz and sets the other clocks (SDRAM etc..) to 500MHz. The result was a proportional increase in performance (~30%) and power draw. Since the cluster does not require a desktop environment, the GPU RAM reserve was reduced to 48MB. 48MB seems to be the smallest amount of RAM allowed for the GPU on the current version of Arch Linux ARM for the RPi (it wouldn’t boot with anything less).

Performance

As discussed earlier, the RPiCluster was not built with the intent of creating a HPC platform. However, it is still interesting to see where it ended up, right? There are, of course, many methods for measuring the performance of a cluster. Initially, I did some individual CPU tests comparing various platforms with the RPi. Table 1 shows the specifications of the platforms used for comparison. Each platform was compared on the basis of single core/single thread integer/floating point performance using the GNU arbitrary precision calculator: *bc*.

[Table 1]

Figure 6 shows the performance of each platform, where the performance is measured in terms of how little time was required to complete the calculation (shorter time is better).

[Figure 6]

Of course, after I published some details of the RPiCluster, everyone wanted to know what kind of MPI parallel performance the RPiCluster has. So, I took some time to get HPL (often used for the TOP500 list) working on the RPiCluster to determine overall parallel computing performance. For comparison, I also setup HPL on a single Onyx node (multi-threaded to leverage all the cores). Table 2 shows the resources available and the HPL results after being tuned for peak performance on both platforms.

[Table 2]

As is seen in Table 2, the RPiCluster HPL performance is about one quarter that of the BSU cluster node (Onyx). Although, the RPiCluster can solve larger problems due to its greater available RAM.

Conclusion

Overall, the RPiCluster has successfully fulfilled its purpose. Since completing the build I have moved my dissertation work exclusively to the RPiCluster. I have found performance perfectly acceptable for my simulation needs, and have had the luxury of customizing the cluster software to fit my requirements exactly.

I would like to thank all the guys I work with in the Hartman Systems Integration Laboratory for all their help implementing the cluster design, especially Michael Pook, Vikram Patel, and Corey Warner. Additionally, this project would not have been possible without support from Dr. Sin Ming Loo and the Boise State University Department of Electrical and Computer Engineering. Additional details may be found in my original white paper, found here. I also created a video that shows the cluster running an MPI parallel program I developed to blink LEDs: YouTube.

Figures/Tables

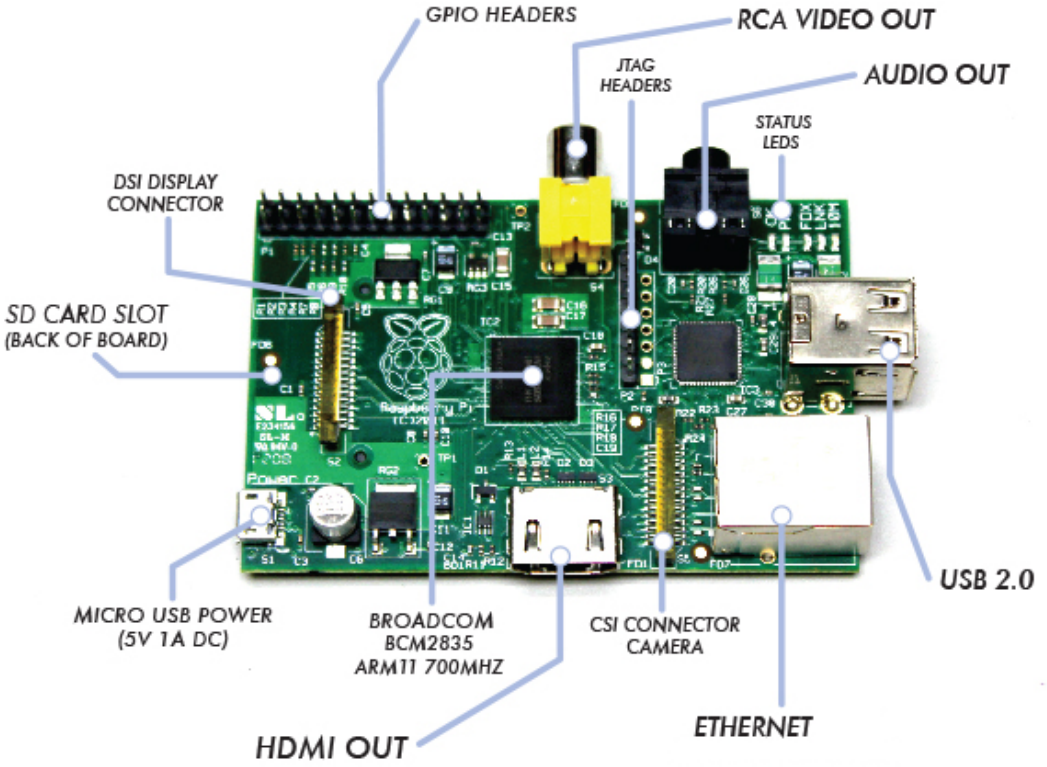


Figure 1: Raspberry Pi Model B (512MB RAM)

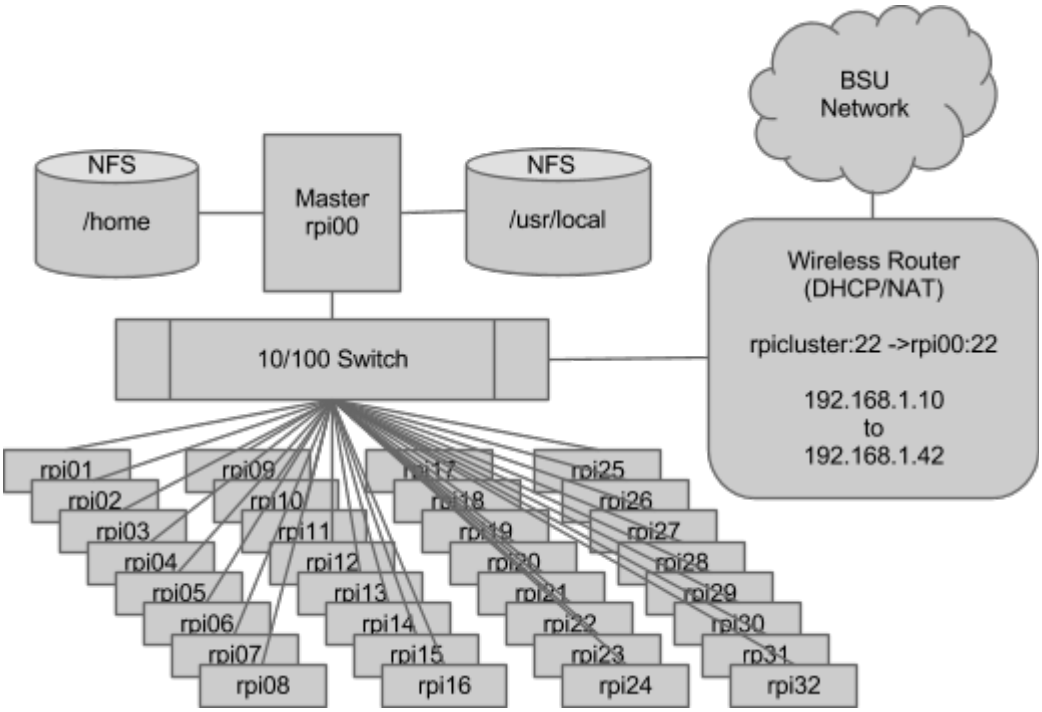


Figure 2: Cluster Design: RPiCluster Network Architecture

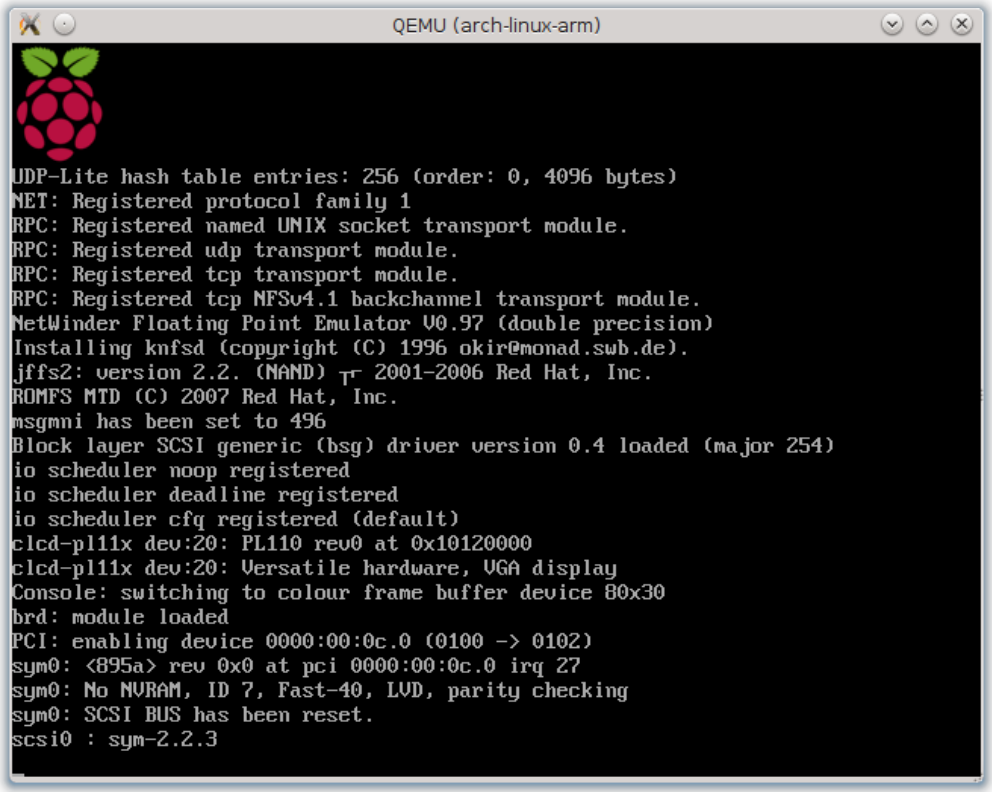
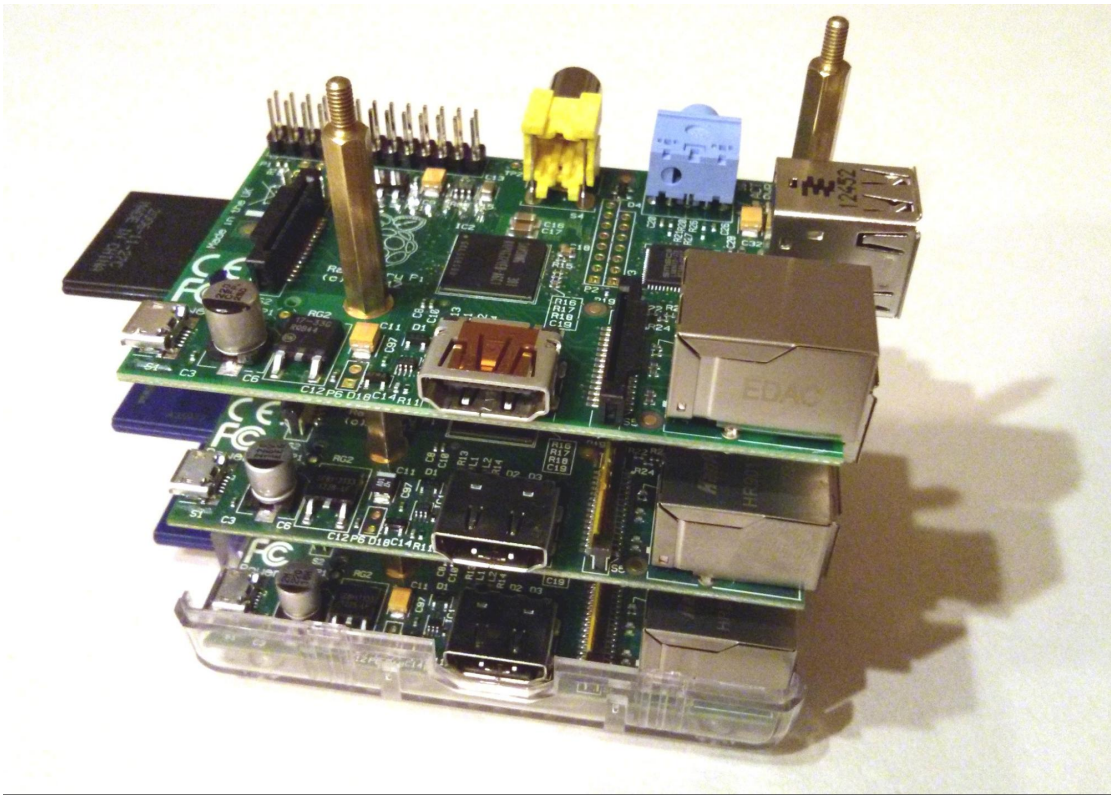
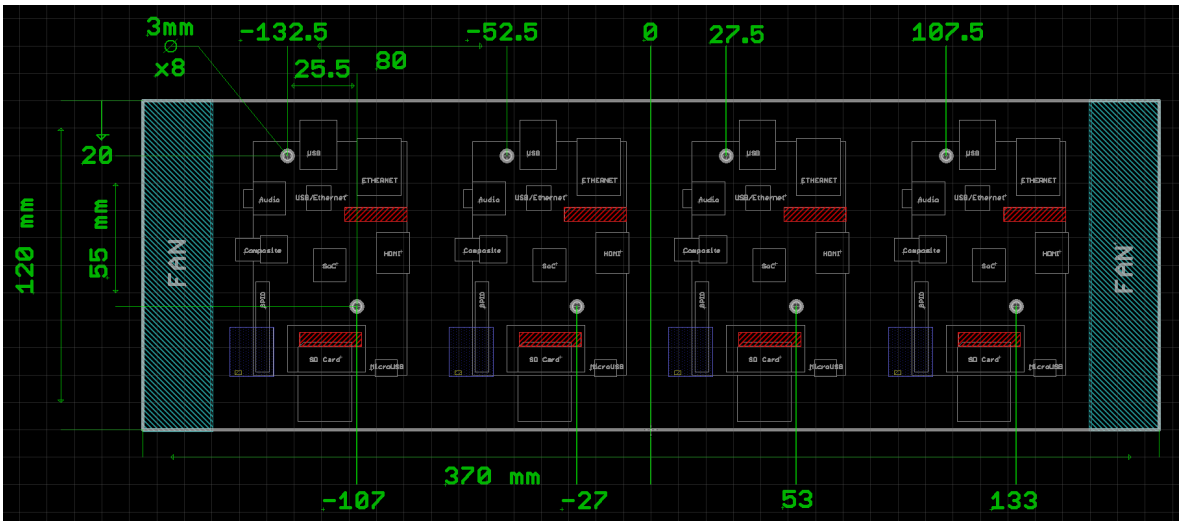


Figure 3: Arch Linux ARM under QEMU

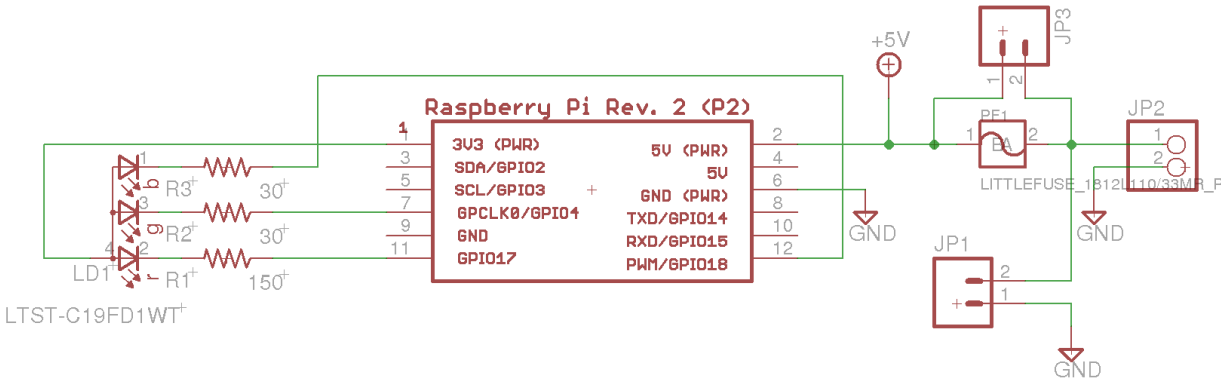


(a) Rack Mounting RPis Using PCB Stand-offs

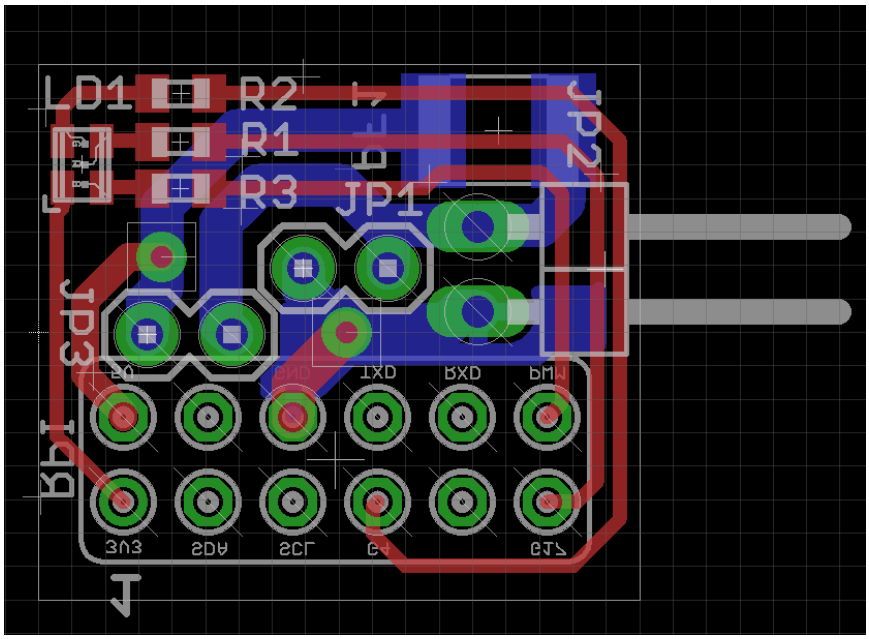


(b) Plexiglass Layout (Top/Bottom of RPiCluster Rack)

Figure 4: RPiCluster Rack Components



(a) RPiCluster Power/LED Board Schematic



(b) RPiCluster Power/LED Board PCB Layout

Figure 5: RPiCluster Power/LED Board

Table 1: Single Core/Single Thread Integer/Floating-Point Platforms

Platform	Cost	OS	CPU	Cores	MHz
BSU Cluster Node	$1,250	Fedora 16	Intel Xeon E3-1225 (x86_64)	4	3,100
Thinkpad T410	$1,450	Linux Mint 13	Intel Core i7 M620 (x86_64)	4	2,670
Chromebook	$250	Arch Linux ARM	Samsung Exynos 5250 ARM Cortex-A15 (ARMv7)	2	1,700
Beagle Bone Black	$45	Arch Linux ARM	Sitara ARM Cortex-A8 (ARMv7)	1	1,000
Raspberry Pi	$35	Arch LInux ARM	BCM2708 ARM1176JZF-S (ARMv6)	1	700

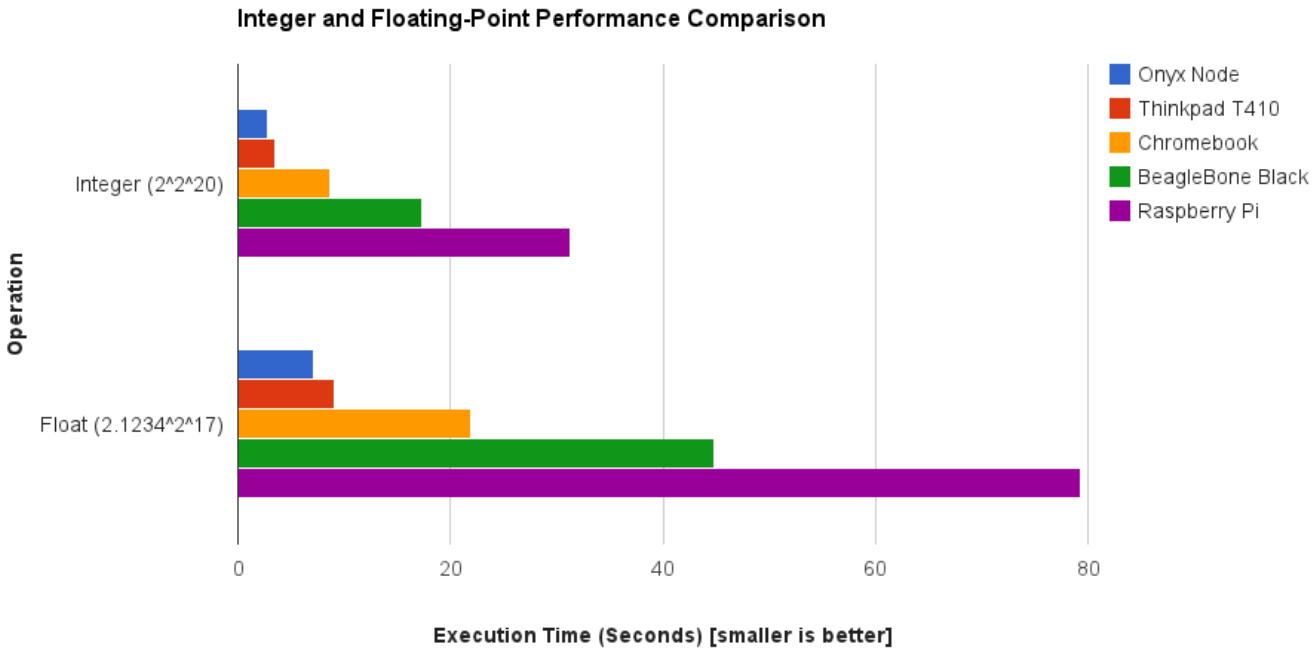
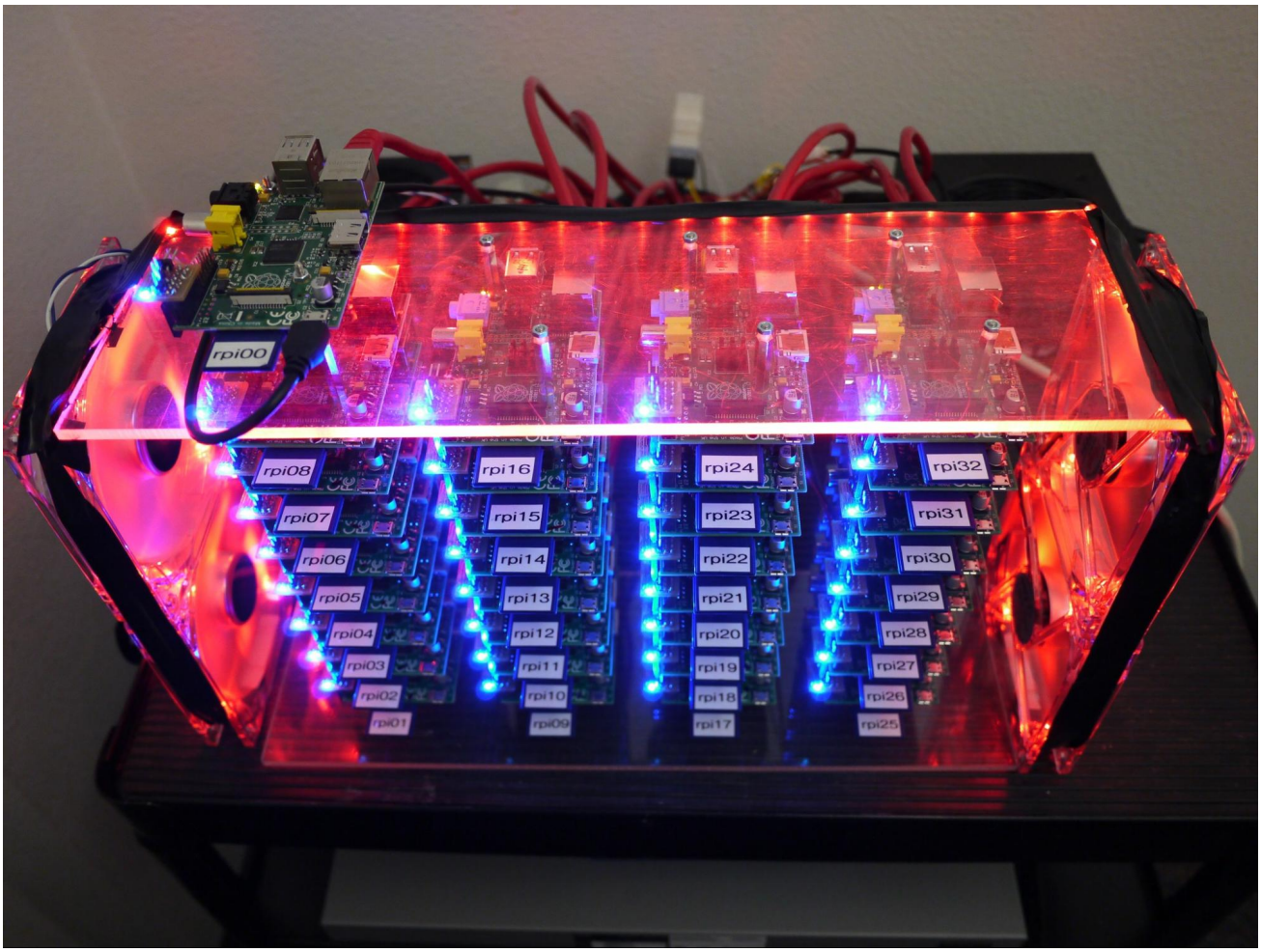


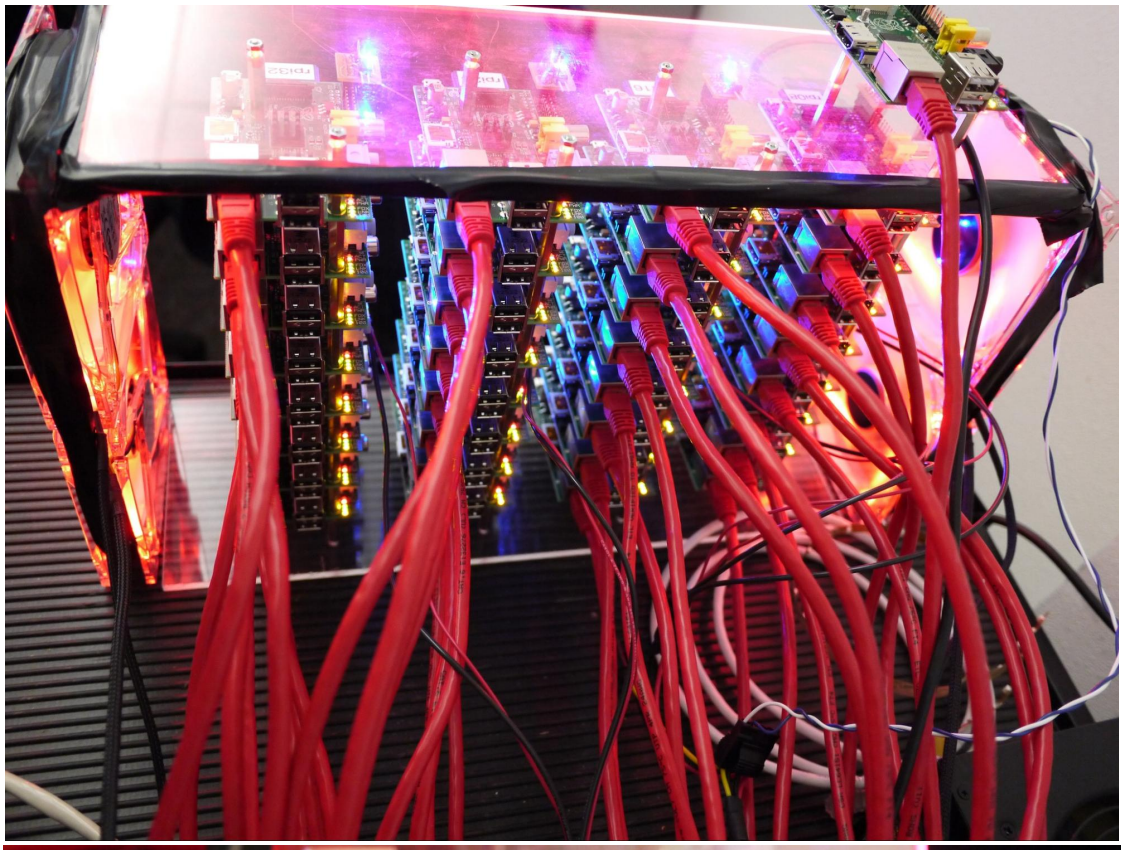
Figure 6: Integer and Floating-Point Performance (using bc)

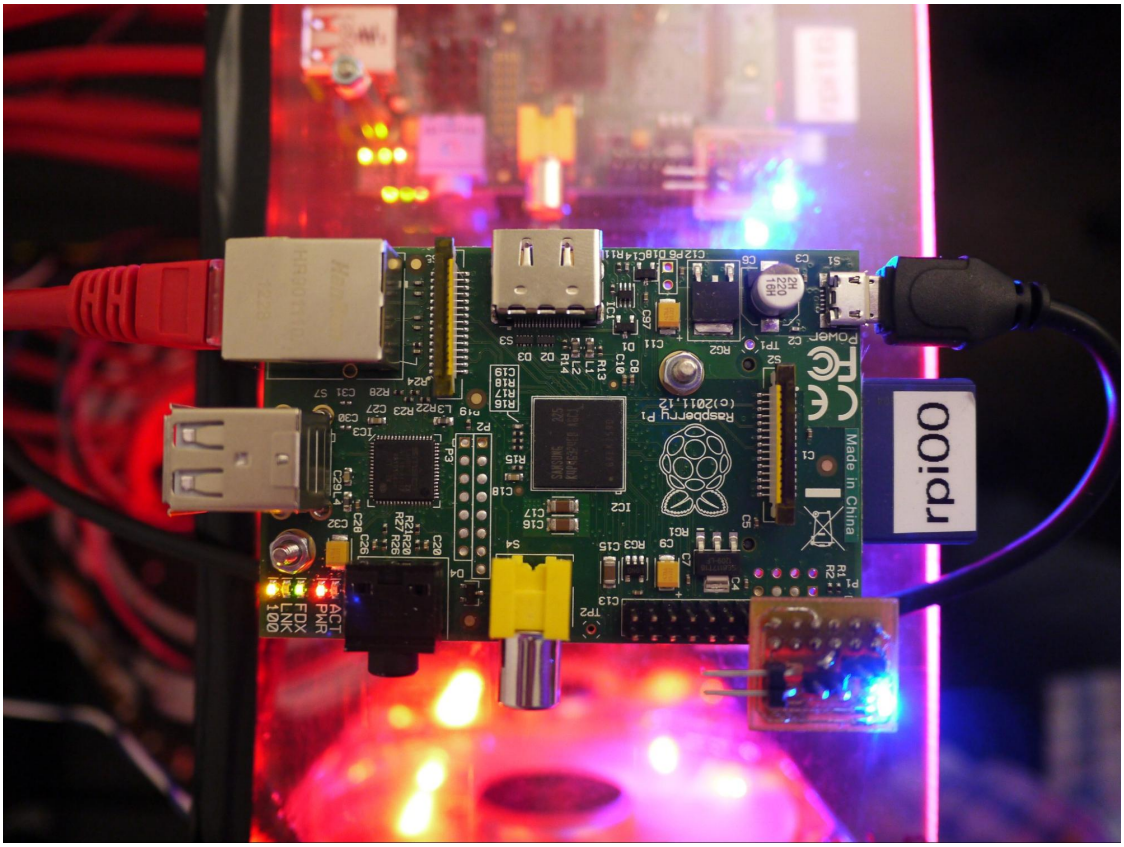
Table 2: Resources/HPL Performance

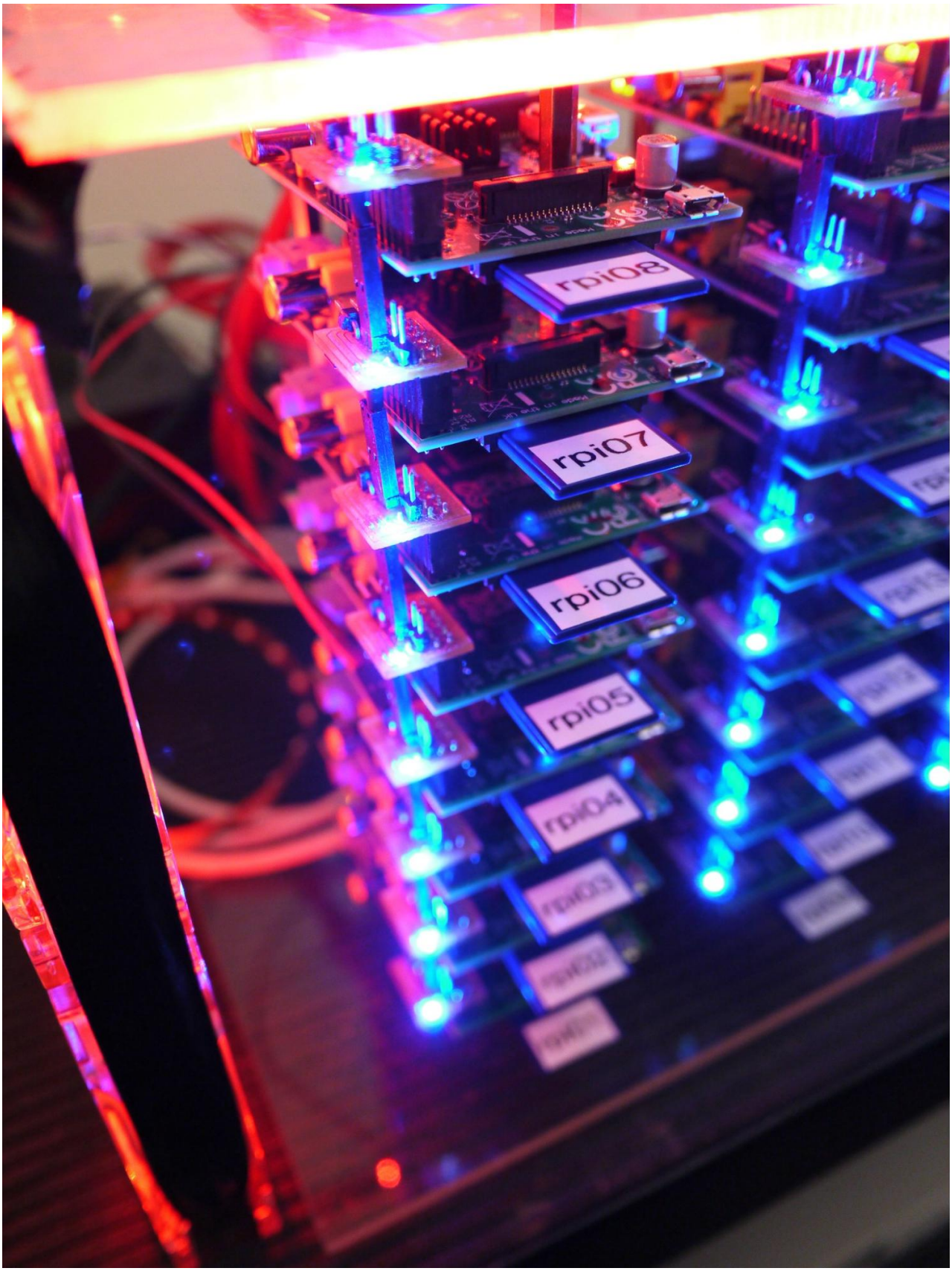
Feature	Onyx Node	RPiCluster
Processing Cores	4	32
Clock Speed (MHz)	3,100	1,000
Total RAM (GB)	8	16
Available RAM (GB)	5	14.6
Available Disk Space (GB)	85	176
HPL GFLOPS	43.41	10.13

Pictures


rpi00
rpi08
rpi07
rpi06
rpi05
rpi04
rpi03
rpi02
rpi01
rpi16
rpi15
rpi14
rpi13
rpi12
rpi11
rpi10
rpi09
rpi24
rpi23
rpi22
rpi21
rpi20
rpi19
rpi18
rpi17
rpi32
rpi31
rpi30
rpi29
rpi28
rpi27
rpi26
rpi25




rpi00
Made in China


rpi08
rpi07
rpi06
rpi05
rpi04