Capitulo 4: Levantamiento y pruebas de la Infraestructura

Después de haber revisado algunos sistemas embebidos (Adapteva Parallella, Intel Galileo Gen 1, Nvidia Jetson TK1 y Nvidia Jetson TX1), y luego de haber conocido sus características más sobresalientes y sus limitaciones, se escogió por realizar toda la infraestructura en el sistema embebido que tenía las mejores capacidades para el cómputo en alto rendimiento: La Jetson TX1.

En colaboración con la Universidad de Grenoble (Université Grenoble Alpes) y el laboratorio LIG en Francia, se pudo levantar la infraestructura para realizar las respectivas pruebas del sistema. En este capítulo se explicará con detalle las características de dicha infraestructura y la inclusión del manejador de paquetes en ella, indicando también las ventajas y desventajas del uso de esta herramienta.

Infraestructura de NVIDIA Jetson TX1

Una vez escogido como sistema embebido la NVIDIA Jetson TX1 para realizar las pruebas de rendimiento computacional y energético y Nix como manejador de paquetes funcional para encapsular los programas, se procedió a levantar la infraestructura de dicho sistema.

La infraestructura consta de 8 NVIDIA Jetson TX1, donde para fines prácticos cada uno lo llamaremos un Nodo. Cada uno de estos nodos estaba conectado directamente a un *switch* a través de un cable UTP tradicional. Un PC, en este caso de tipo portátil se conectó al *switch* para ser el Nodo Maestro. El portátil tenía como sistema operativo Debian. Un esquema gráfico resumido del sistema en general se muestra en la Ilustración 1.

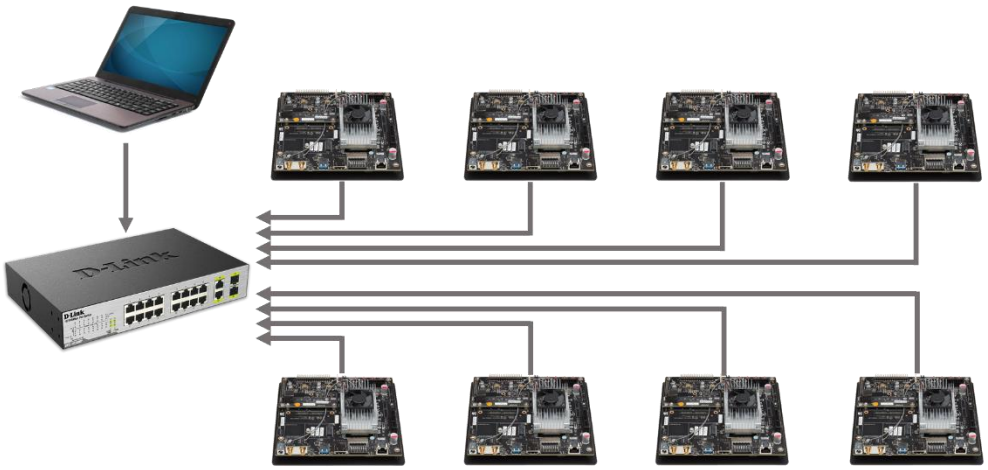

Ilustración 1 Esquema de la Infraestructura de las NVIDIA Jetson TX1

El Nodo Maestro (el portátil) se conecta a los demás Nodos por medio del *switch*. Para poder lograr lo anterior se requirió primero probar cada nodo que estuviera funcionando correctamente y por aparte. Para ello se instaló el sistema operativo con las mismas condiciones en cada nodo ejecutando los comandos correspondientes, luego se prueba su conexión a la red y al Nodo Maestro y luego se prueba instalando las otras librerias como OpenMPI y CUDA.

Como se mencionó en capítulos anteriores, CUDA posee un problema de instalación. Para poder instalar CUDA de forma casi transparente se requiere una herramienta de NVIDIA llamada Jetpack en la versión que soporte la NVIDIA Jetson que se vaya a usar (TX1 en nuestro caso). Lastimosamente la herramienta Jetpack solo se puede instalar si se tiene el sistema operativo Ubuntu 14.06. Afortunadamente existe en los foros una forma alterna de realizar esta instalación saltándose la Jetpack, sin embargo NVIDIA recomienda hacerlo por ella. -busqye la referencia-

Luego de la instalación y prueba de los ejemplos de CUDA, se dispone a realizar la configuración de la red de los Nodos. Para ello se sigue la guía de OpenMPI [1] para que cada nodo pueda conectarse y “verse” a los demás. Además de lo anterior, se configuró los nodos para que pudieran conectarse entre ellos sin clave usando otra guía para tal fin. -revisar si es otro capitulo de la guía de openmpi-.

Una vez que los Nodos están conectados, con las librerias instaladas y que se puedan comunicar entre sí, se procedió a hacer las pruebas de rendimiento, primero sin ninguna herramienta extra y luego con Nix, el Manejador de Paquetes Funcional. Esto se realizó de esta manera para poder comparar de primera mano los escenarios de construcción, compilación y ejecución de los Benchmarks de prueba de este sistema.

Benchmarks: La prueba de Rendimiento Computacional

Para poder probar el rendimiento del sistema es necesario usar alguna herramienta o un programa que permita revisar esta capacidad del sistema, además de ello, al ser un programa también se encapsularia por lo que al final las pruebas de uso igual funcionarían.

La mejor forma de probar el rendimiento de un sistema es utilizando alguna herramienta o programa que pueda medir y exigir al máximo el sistema. Para el caso del proyecto se usan los Benchmarks, una serie de programas (una *suite*) que tiene como objetivo probar el sistema de distintas maneras. Aunque existen un incontable numero de benchmarks y una amplia discusión de si estos son o no válidos en el día de hoy, se utilizará los benchmarks en los que prueban y rankean los supercomputadores en el mundo: Stream Benchmark y High Performance Linpack Benchmark.

STREAM Benchmark

-Memory Bandwidth and Machine Balance in Current High Performance Computers

https://www.researchgate.net/publication/51992086_Memory_bandwidth_and_machine_balance_in_high_performance_computers

- Stream http://www.cs.virginia.edu/stream/ref.html

El *Stream Benchmark* es un benchmark que se encarga de medir el ancho de banda (en Mb/s) de un sistema a través de operaciones simples, pero con un gran tamaño. Las operaciones que se usan son las siguientes:

$$\begin{array}{ll} COPY & A[i] = B[i] \\ SCALE & A[i] = k * B[i] \quad ; \quad k \in \mathbb{R} \\ ADD & A[i] = B[i] + C[i] \\ TRIAD & A[i] = B[i] + k * C[i] \quad ; \quad k \in \mathbb{R} \end{array}$$

Este benchmark se usa para medir la transferencia de los datos, específicamente del procesador con las distintas memorias, principalmente con el disco duro. La idea es tener una gran cantidad de operaciones, se aconseja que sea 4 veces mayor que el tamaño total de las memorias o 1 millón de operaciones, lo que sea más grande.

Existen dos formas de ejecución: Lineal y Paralelo. La forma Lineal es más sencilla de todas, hay que tener cuidado con la memoria caché y los datos no deben ser *cacheables*. Esta forma es para cuando se tiene 1 solo Nodo. La forma paralela es menos sencilla de ejecutar debido al uso de múltiples librerías (para la comunicación entre nodos). Esta forma es la ideal para probar n Nodos.

High Performance Linpack (HPL)

-The LINPACK Benchmark: past, present and future

http://www.netlib.org/utk/people/JackDongarra/PAPERS/146_2003_the-linpack-benchmark-past-present-and-future.pdf

El benchmark HPL (*High Performance Linpack*) se encarga de realizar y medir el rendimiento de un sistema basado en operaciones que involucran la resolución de una matriz. Inicialmente fue diseñado para revisar errores en los llamados de las operaciones de Linpack y tiempos de ejecución, luego empezaron a hacer suites completas para que pudiera funcionar con cualquier computador. Actualmente es el indicador de rendimiento que usan para probar la eficiencia de un supercomputador, aunque poco a poco se están complementando con otros Benchmarks, incluidos los de Energia.

El programa se encarga de resolver un sistema de ecuaciones de una matriz densa (sin muchos valores nulos) de 64-bit utilizando operaciones lineales y en paralelo, según el sistema pueda realizarlo. La matriz es la siguiente:

$$Ax = b$$

Siendo A una matriz nxn, x un vector de n elementos y b otro vector de n elementos. El algoritmo computa el LU siendo $LU = A$ y hace los cálculos correspondientes para llegar al resultado. El algoritmo es de orden $O(n^3)$, más exactamente:

$$O(n^3) \Rightarrow \frac{2}{3}n^3 + 2n^2 + O(n)$$

operaciones de punto flotante de sumas y multiplicaciones.

Para este Benchmark se requiere algunas librerias especializadas. En nuestro proyecto se usaron las librerias *MPICH v2* (MPI), *gfortran* (Fortran), *CBLAS* (BLAS Library) y CUDA. Una vez instalado estas librerias se debe modificar un archivo llamado *HPL.dat*, que es aquel donde se especifica las rutas de estas librerias y los valores de entrada que van a tener nuestras pruebas.

Algunos valores a tener en cuenta son N, NB, P y Q. N representa el tamaño del problema, depende de la memoria total del sistema y mas o menos se rige por la siguiente formula:

$$N = \left(\sqrt{\frac{Tamaño\ Memoria\ (Bytes)}{8}}\right) * (\%\ Memoria)$$

Donde el dividendo “8” se refiere a que se está manejando valores de doble precisión (8 Bytes) y el “% de memoria” debe estar entre 80%-90% (0.8 – 0.9).

NB se refiere al tamaño del bloque del Grid, generalmente es una lista de múltiplos de 8 que inicia en 96 y termina en 256. Una optimización de N depende del valor de NB, de tal manera de que estén alineados de la siguiente manera:

$$NB = [96,104,112,120, \dots ,224, \dots ,256]$$

$$N_{Optimizado} \cong \left(\left\lfloor \frac{\lfloor N \rfloor}{NB} \right\rfloor\right)(N)$$

Finalmente, los valores P y Q indican el tamaño del Grid y dependen de los procesadores totales. Estas variables multiplicadas deben indicar el total de los procesadores del sistema, incluyendo todos los nodos, además P debe ser ligeramente menor o igual a Q.

$$P * Q = No.\ Procesadores\ sistema$$

$$No.\ Procesadores\ sistema = (No.\ Nodos)\left(\frac{No.\ Procesadores}{Nodos}\right)$$

$$P \approx Q\ ;P < Q\ (Muy\ poco)$$

Los resultados muestran, el desempeño máximo que alcanzó la máquina realizando algunas operaciones del problema (Rmax medido en GFlops/s), que puede

compararse con el rendimiento teórico del sistema (Rpeak) para dilucidar cuanto se está perdiendo de rendimiento en el problema.

Los valores exactos de entrada y de salida que se obtuvieron en este proyecto, se especificarán más detalladamente en el capítulo de los resultados.

Jacobi Benchmark

El Jacobi Benchmark es una implementación en paralelo del método de Jacobi para resolver sistemas de ecuaciones lineales. Una vez más utiliza la ecuación de matrices:

$$Ax = b$$

Y con ello se reescribe para poder encontrar la variable x según sus valores anteriores, volviéndolo un algoritmo de relajamiento, de la siguiente manera:

$$x_{k+1} = D^{-1}b + D^{-1}(L + U)x_k$$

Ya de esta forma se tiene una forma discreta para resolver el sistema de ecuaciones. Al igual que los otros benchmarks, el algoritmo contiene operaciones de cuenta del rendimiento de los sistemas, y, posee un valor máximo de iteraciones o valor mínimo de error.

Los valores de entrada son esencialmente 2, “-t x y” y “-d num.”. Los valores de “-t x y” son obligatorios y expresan el número de procesos en X y Y (filas y columnas) respectivamente. La variable “-d <num>” es opcional, y representa el tamaño del dominio local, si no se ingresa su valor es predeterminado, y en este caso es 4096.

En el caso del proyecto se realizó este Benchmark en los Nodos, iniciando solo en 1 Nodo y luego conectándolo con 2, 4 y 8 Nodos. Al final de tomar los valores se pasó a realizar este mismo Benchmark con el Manejador de Paquetes Nix. Sin embargo, este tema se contará más adelante con un más amplio detalle en el capítulo de Resultados.

Integración con Nix: El Manejador de Paquetes Funcional

Después de realizar todas las pruebas para observar el comportamiento “tradicional” (por así llamarlo) del sistema que se levantó, es necesario ahora probarlo usando el Manejador de Paquetes Funcional Nix. La razón del uso de esta herramienta es su flexibilidad, portabilidad y por ende su escalabilidad con una fácil instalación y configuración del entorno.

Aunque era muy tentador encapsular todos los Benchmarks mencionados anteriormente, y hacerles las pruebas de desempeño, cabe considerar la ventana de tiempo del uso de los Tegra TX1. Por lo consiguiente se decidió encapsular 1 solo Benchmark, y el Jacobi Benchmark fue el elegido para su uso por considerarse un Benchmark estandarizado y sencillo de configurar. No obstante cabe resaltar que

aunque se decidió un solo Benchmark, el análisis no se pierde, puesto que esta prueba se puede extrapolar a los otros dominios y dar conclusiones respecto a ello.

Si el software ya está empaquetado y puesto en un canal (ya sea el global o uno privado) simplemente se ejecuta el comando de instalación y ya tiene el software en su sistema:

Nix-build -A OpenMPI

Pero si por el contrario (y en nuestro caso) no se encuentra las librerias, tocará anexarlas y encapsularlas primero. La idea general es *derivar* el software. Para lograr este objetivo se usa el Lenguaje Nix dentro de algo que se le conoce como receta. La receta es un conjunto de descripciones que se le indica a Nix cómo debe derivar el software, con qué librerias y en qué versión. Nix luego se encargará de armarlo, empaquetarlo y almacenarlo en el Store Local.

Integración Jacobi con NIX

Como se mencionó anteriormente, el objetivo es hacer el encapsulamiento del *Jacobi Benchmark* con todas las dependencias que requiera, analizando el proceso según su dificultad de empaquetar todo y luego revisar su desempeño computacional y energético.

Lo primero que se realiza es crear el proyecto dentro de *nixpkgs* clonando por wget y dentro de la carpeta se agrega el archivo *default.nix* que va a ser nuestra receta. Las fases principales de Nix al ejecutar la derivación son las siguientes:

1. Unpack Phase: Realiza la descompresión de archivos y entra a su carpeta.
2. Configure Phase: Realiza la operación "*./configure*".
3. Build Phase: Realiza la operación *"Make"*.
4. Install Phase: Realiza la operación "*Make Install*".

Sin embargo existen otras fases, y además de ello se puede especificar qué fases utilizar y si desea hacer específico en esa fase (modificar lo que hace la fase). El uso de fases se encuentra dentro del manual de Nix.

Los problemas o inconvenientes encontrados fueron en uso de librerias. Nix no "ve" las librerias nativas de CUDA para hacer el JacobiBench. Para ello se hace un proceso llamado Dynamic Linker para que pueda conectarse Nix con esas librerias.

También hubo problemas con el uso de GCC, ya que se requeria una versión más antigua que la que el sistema tenia, así que se decidió instalar GCC < 5 en Nix y así Nix derivaría con ese GCC.

Finalmente el problema más grande que se tuvo fueron las librerias de CUDA que no se encontraban al compilar. Errores como "*libcuda.so.1 not found*" y "*libnvrm_gpu.so not found*" eran muy comunes, por lo que era necesario agregar las condiciones de compilación de "-lcuda" y copiando cada librería que no encontraba

en la carpeta “/lib” de CUDA. En general, fueron las siguientes librerias que se tuvo que hacer este proceso:

- *Libcuda.so -> …/lib/stubs -> -L*
- *Libcuda.so.1 -> /usr/lib/aaron64-linux-gnu/tegra/*
- *Libnvrm_gpu.so -> /usr/lib/aaron64-linux-gnu/tegra/*
- *Libnvrm.so -> /usr/lib/aaron64-linux-gnu/tegra/*
- *Libnvidia-fatbinaryloader.so.28.1.0 -> /usr/lib/aaron64-linux-gnu/tegra/ -> -li*
- *Libnvos.so -> /usr/lib/aaron64-linux-gnu/tegra/*

Ya cuando no generaba más errores de compilación, por fin funcionó el encapsulamiento, por lo que se podía ejecutar de la siguiente manera:

```
/mpirun /result/.../bin/jacobi-mpi -t 1 1
```

```
/nix/store/<hash>/mpirun  ./result/.../bin/jacobi-mpi -t 1 1
```

En resumen el software quedó de la siguiente manera:

- ~/nixpkgs/pkgs/misc
  - /OpenMPI3 (11Kb)
  - /Jacobi-bench (1.1Gb)
- GCC5

Mientras hace la derivación con pruebas, el sistema de archivos pesa 130Gb aproximadamente, pero luego el Nix/Store queda reducido a 21Gb.

El Nix-Store finalmente queda pesando 4.1Gb, y copiándolo a otro Nodo (Tegra TX1) que no tenía esta derivación funciona sin problemas. Sin embargo, esto nos pone a pensar acerca del tamaño en la derivación de los archivos, lo que al final se hizo fue poner un disco que se conecte por NFS y que en este disco se derive y se tenga el Nix-Store para todos los Nodos.

Con el NFS activado y Nix instalado, las ejecuciones de este software funcionan sin problemas, la comunicación de los nodos es correcta con el encapsulamiento y al mismo tiempo éste es portable (e incluso único en el sistema). Lo único que hay que tener en cuenta es tener cuidado con la eliminación de archivos o el montaje de esta unidad, ya que un mal montaje puede eliminar toda la información.

En el siguiente capítulo se indicará los resultados que se obtuvieron en cada prueba, desde los Benchmarks en estado tradicional así como usando el Manejador de Paquetes Nix, y a partir de ellos se darán las conclusiones del uso de este tipo de herramienta.

-mejorar esas letras en la ejecución

-Anexo nada? Se deja asi o se quiere poner un proceso?

Trabajos citados

[1] «MPI Tutorial,» [En línea]. Available: http://mpitutorial.com/. [Último acceso: Marzo 2019].

[2] «Docker Get Started,» [En línea]. Available: https://docs.docker.com/get-started/. [Último acceso: Febrero 2019].

[3] «Docker Web Page,» [En línea]. Available: https://www.docker.com/what-container#/virtual_machines.

[4] «What is Heterogeneous Computing?,» 2017. [En línea]. Available: http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-computing/.

[5] «What is edge computing and how it's changing the network,» [En línea]. Available: https://www.networkworld.com/article/3224893/internet-of-things/what-is-edge-computing-and-how-it-s-changing-the-network.html.

[6] F. Bonomi, R. Milito, J. Zhu y S. Addepalli, «Fog Computing and Its Role in the Internet of Things,» ACM, 2012.

[7] S. Yi , Z. Hao, Z. Qin y Q. Li, «Fog Computing: Platform and Applications,» IEEE, 2015.

[8] Y. Ai, M. Peng y K. Zhang, «Edge Computing Technologies for Internet of Things: A Primer,» Elsevier, 2017.

[9] S. Yi, C. Li y Q. Li, «A Survey of Fog Computing: Concepts, Applications and Issues,» 2015.

[10] P. Lucas, J. Ballay y M. McManus, Trillions: Thriving in the Emerging Information Ecology, 2012.

[11] «Package Manager,» [En línea]. Available: https://en.wikipedia.org/wiki/Package_manager. [Último acceso: 2017].

[12] «Nix Package Manager,» [En línea]. Available: https://nixos.org/nix/about.html.

[13] «NixOs,» [En línea]. Available: https://nixos.org/.

[14] M. Geveler, D. Ribbrock, D. Donner, H. Ruelmann, C. Hoppke, D. Schneider, D. Tomaschewski y S. Turek, «The ICARUS White Paper: A Scalable Energy-Efficient, Solar-Powered HPC Center Based on Low Power GPUs,» Springer, 2017.

[15] «Montblanc-Project,» [En línea]. Available: http://www.montblanc-project.eu/.

[16] J. Saffran, G. Garcia, M. Souza, P. Penna, M. Castro, L. Góes y H. Freitas, «A Low-Cost Energy-Efficient Raspberry Pi Cluster for Data Mining Algortihms,» Springer, 2017.

[17] F. P. Tso, D. White, S. Jouet, J. Singer y D. Pezaros, «The Glasgow Raspberry Pi Cloud: A Scale Model for Cloud Computing Infraestructures».

[18] B. Bzeznik, O. Henriot, V. Reis, O. Richard y L. Tavard, «Nix as HPC Package Management System,» ACM, 2017.

[19] M. Plauth y A. Polze, «Are Low-Power SoCs Feasible for Heterogeneous HPC Workloads?,» Springer, 2017.

[20] J. Guerreiro, A. Illic, N. Roma y P. Tomás, «Performance and Power-Aware Classification for Frequency Scaling of GPGPU Applications,» Springer, 2017.

[21] N. Saurabh, D. Kimovski, S. Ostermann y R. Prodan, «VM Image Repository and Distribution Models for Federated Clouds: State of the Art, Possible Directions and Open Issues,» Springer, 2017.

[22] J. Breitbart, S. Pickartz, J. Weidendorfer y A. Monti, «Viability of Virtual Machines in HPC,» Springer, 2017.

[23] «Amazon Web Services,» 2017. [En línea]. Available: https://aws.amazon.com/es/.

[24] «RISC vs CISC,» 2000. [En línea]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/index.html. [Último acceso: 2017].

[25] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski y S. Futral, «The Spack Package Manager: Bringing Order to HPC Software Chaos,» ACM, 2015.

[26] «Easy Build Web Page,» [En línea]. Available: https://easybuild.readthedocs.io/en/latest/. [Último acceso: Febrero 2019].

[27] N. Rajovic, A. Rico, N. Puzovic y C. Adeniyi-Jones, «Tibidabo: Making the case for an ARM-based HPC system,» Elsevier, 2013.

[28] B. Bashari Rad, H. John Bhatti y M. Ahmadi, «An Introduction to Docker and Analysis of its Performance,» IJCSNS International Journal of Computer Science and Network Security, 2017.

[29] M. Geimer, K. Hoste y R. McLay, «Modern Scientific Software Management Using EasyBuild and Lmod,» First International Workshop on HPC User Support Tools, New Orleans, 2014.

[30] «Spack: A flexible package manager that supports multiple versions, configurations, platforms, and compilers.,» [En línea]. Available: https://spack.io/. [Último acceso: Febrero 2019].

[31] E. Dolstra, The Purely Functional Software Deployment Model, Utrecht, 2006.

[32] «The Parallella Board,» [En línea]. Available: https://www.parallella.org/. [Último acceso: Marzo 2019].

[33] «Intel Galileo (Arduino web page),» [En línea]. Available: https://www.arduino.cc/en/ArduinoCertified/IntelGalileo. [Último acceso: Marzo 2019].

[34] «Intel Galileo Gen 2 Board (Specs),» [En línea]. Available: https://ark.intel.com/content/www/us/en/ark/products/83137/intel-galileo-gen-2-board.html. [Último acceso: Marzo 2019].

[35] «Jetson TK1,» [En línea]. Available: https://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html. [Último acceso: Marzo 2019].

[36] «Nvidia Jetson Systems,» [En línea]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/. [Último acceso: Marzo 2019].