Leopold–Franzens–University
Austria, Innsbruck
&
Japan Advanced
Institute of Science and Technology
Japan, Ishikawa

Institute of Computer Science
Research Group:
DPS@LFU
(Distributed and Parallel Systems)
Research Group:
Inoguchi Lab@JAIST
(Security and Network Area)

Distributed GPGPU on Cloud GPU Clusters- Quick Tutorial

Master Thesis

Supervisor: Dr. Juan J. Durillo
Prof. Dr. Yasushi Inoguchi
Dr. Sashko Ristov

Martin Schuchardt

martin.schuchardt@student.uibk.ac.at

Innsbruck
2 July 2018

This tutorial will help you getting our sample code running on a developer workstation. Once you are familiar with our sample, it will be easier to troubleshoot common problems that might come with working on a real cluster, like firewalls, routing, name resolution, remote permissions, and many more. The packages and commands, we are using in this document, refer to Debian 9/Ubuntu 16.04. If you use another version or Linux distribution, finding proper counterparts should be easy. For our sample code, all required prerequisites and setup/tests are done using a setup-script and are described in this tutorial step by step.
We expect that you have basic knowledge of MPI, ssh, networking, and OpenCL.

setup/setupDistributedGPGPU.sh

We provide a setup script that basically does

- **download and configure Virtual Box and template base images for Debian 9/Ubuntu 16.04**
  We install Virtual Box 5.1 from the repository. At the time of writing, version 5.2 was the most current, but we found a bug in v5.2 that made another script (*vboxCreateInstances.sh*) hung during the creation of the cluster nodes. If you want to use a more current version, check if this script can work with the *"vboxmanage guestcontrol copyto"* command.
  Next, it downloads two base images for the future cluster worker instances: one based on Debian 9, the other on Ubuntu 16.04. These images contain all required prerequisites to be used as a worker: Virtual Box drivers (guest additions), CUDA, Intel OpenCL driver for CPUs, g++ compiler and tools, and OpenMPI.
  The worker instances will use two network interfaces: the first is connected to the Virtual Box "host only" interface, which allows communication between the host and all workers only, no connection outbound or inbound is possible. The host is connected by an additional virtual network interface and uses per default the subnet 192.168.56.0/24.
  The second interface is bridged to the hosts external interface, and can be either used for example to install additional packages from an Intranet/Internet repository, or detached for security reasons.
- **create a bunch of cluster worker instances using Virtual Box for testing and development**
  These worker instances are clearly not designed to be performant. The base image downloaded will be multi-attached read-only to all instances, and each instance has a private snapshot for its writes.
  Workers are lightweight (5.5GB for the base image plus approximately 150MB per worker instance), use one CPU core, 512MB RAM and even an older developer workstation can easily launch a bunch of workers. We found this environment convenient for development and all tests besides performance tests. Using the Intel OpenCL driver for CPUs, you can simulate OpenCL devices.
  The script asks how many instances should be created, what the IP for the first instance should be, which Linux base image ([ubuntu|debian]) to be used and needs the root password for the base image (we expect the instance to be used without permanent external network access, so use the insecure but easy to remember password "mpi").
  Additional clusters can be created at any time by calling a sub-script, e.g. "*./vboxCreateInstances.sh 5 192.168.56.20 debian*" for a 5 instances Debian cluster using the IP range 192.168.56.[20,24]. The corresponding *mpi.hostfile* for later use can be found in *setup/output/*.
- **install prerequisites on the host workstation like compilers, OpenMPI, OpenCL headers, ...**
  We install all software from the default repository of the Linux distribution. In general, the versions used are not important. Except for MPI, which requires that all instances that execute a program, use the same MPI version (and also a comparable OS and configuration). If you use for example Debian 9 or Ubuntu 16.04 as your host OS, you can extend your cluster worker instances with your host machine as being the master instance for even more convenient development. Or you can build a cluster base image based on the same OS/version you use on your host. Use the scripts (*setup/files-**) we used to configure the base image as templates.
- **test MPI connection with/within the Virtual Box cluster**
  MPI requires passwordless SSH and proper routing between the workers and master instance.
  For SSH, we generate an SSH RSA keypair (if not already exists) and distribute all public keys between the host and all worker instances (*~/.ssh/authorized_keys*). Additionally, we set a static IP on the host-only network IF on all cluster worker instances, and have defined these IPs/hostnames on the host and all worker instances (*~/.ssh/config*). If everything works, you should be able to connect to all host/workers with SSH without a password (e.g. *"ssh cluster20"* to connect to the worker instance with

IP 192.168.56.20).
MPI uses for communication per default the ports 1024-65,536 (plus port 22 for SSH) inbound and outbound. Firewalls should not be a problem when executing our sample on your local machine, but keep that in mind once you start with AWS clusters.
Last, we execute a simple MPI command on all worker instances using *"mpirun –hostfile ˜/mpi.hostfile hostname"*. That hostfile will be generated when creating the cluster. Our script does all that configuration. If successful, you will this *mpi.hostfile* in the home directory of the first cluster instance (copied from *setup/output* and then configured).

- **test the DistributedGPGPU-library within the Virtual Box cluster**
  Next, we compile the library and test a simple distributed matrix multiplication on the cluster. If the compilation fails, probably some includes or library paths are not configured correctly. Once compiled, we copy the binary and Makefile to the first cluster instance, which will be used as the master instance. The Makefile is required to distribute the binary to all cluster worker instances. MPI offers functionality for automatic distribution of the executed binary (*–preload-binary*), but we had problems with some MPI versions with that command (binary distributed, but execution flag not set), so we decided to use scp for the binary distribution, and offer this function in the Makefile.
  Last, we need a proper *mpi.hostfile*, which has already been mentioned in the MPI test.
  Basically, it contains the hostname of all worker instances (cluster20, cluster21, ...) where the number represents the last quadruple of the IP address used in the host-only network. The parameter (slots=1) states, that one MPI process should be started on that instance. You could increase this integer on real clusters if they have for example two OpenCL devices like two GPUs. Our test cluster offers only one device, the single CPU core.
  All workers need to have the same number of slots defined, we expect to have a homogeneous cluster.
  An exception is the master node: the MPI master process counts as a process, and we want to use the OpenCL device on the master as well. Therefore, we launch an additional worker MPI process on the same host, which will do computations with the OpenCL device, while the master process only deals with cluster management and data distribution. So, the instance that hosts our master node and the additional worker, requires having a number of slots increased by one, compared to all other workers.
  The *setupDistributedGPGPU.sh* does all this configuration and executes the sample matrix multiplication by calling mpi with this command: *mpirun –n 1 ./sampleMMul-simple N=727*. The parameter *N=727* defines the test matrix size to be 727x727. In the end, the computations are verified.

- **install and configure the AWS console**
  Last, the AWS CLI will be installed and configured. The configuration affects two parts: locally, the user credentials, and remote, on the AWS administration servers, some settings for the cluster environment.
  You will be asked for your AWS access key, secret access key and preferred region. The selected region has impacts on the costs of AWS instances and of course on the latency between your computer and the cluster. As already mentioned, we use homogeneous clusters only. For that reason, your on-site machine will probably only be used to connect to a cluster, upload the input data and code, compile remote and wait until the computations have been completed. So, latency might not be a major concern, choosing a cheap region could be more interesting.
  The remote configuration concerns general firewall settings (cloud configuration bound to the EC2 user, nothing local) for the AWS cluster, like external SSH/Smb/NetBT access, and creates a user with restricted permissions, that is only allowed to create g2 or t2.micro instances (g2 is the typical GPU instance, t2 a CPU only instance, which is much cheaper and can be used for tests).
  We need this user on the master node, and since for real-world computations this will be an EC2 instance, we don't like to store personal AWS user credentials on that off-site instance.
  Next, we setup a placement group (a special setting that GPU instances should be at the same location and connected with a fast interface).
  The g2 and t2 images for AWS instances are frequently updated. Also, their identifications differ between regions. Therefore, the script searches for the most recent image in the selected region and stores it in the local configuration file (*config.cfg*).
  No AWS cluster will be created now. This is done by calling *awsCreateInstances.sh keys/mpi-us-east-1.pem mpi-us-east-1 5* as an example, assuming that your generated key file (and identical key name) are *"mpi-us-east-1"* and you want to create 5 instances.

The script has been tested and written for Debian 9/Ubuntu 16.04/Amazon Linux. Even if you use a completely different operating system, the steps executed should be easy to understand and rewritten for any kind of Linux.
As a side effect, the script generates a file *setup/config.cfg* which is used to create clusters on demand.
Please do not think of the install script as a one-click installer. It is separated into 10 small blocks so that if one fails, you can fix whatever difference in your environment made the script fail, restart the script and continue after skipping all already completed steps. There are some basic checks that subsequent calls of the script skip already completed actions, but in general, this setup-script has been designed to be an installation guide.

Prerequisites: OpenCL

To see your OpenCL configuration as like as the library will use it, use the code in *code/showDevices*. Compile and execute this helper with *make run*. If you get warnings or errors, check the library and include paths of your environment with the paths that are specified in our Makefile. Also, verify your OpenCL drivers.
If you need to make changes to this code/Makefile, you will probably have to do the same for our other programs, too.
Once the helper has been executed, it will list and enumerate all devices on all OpenCL platforms. We use by default device 0. If the OpenCL driver works correct, our code will list among another device specific information something like the following:

```
 ...
*** platform_id[0], device_id[0], ACC_DEVICE=0 **************
  CL_DEVICE_NAME:        Intel(R) Core(TM) i5-3570 CPU @ 3.40GHz
 ...
```

Listing 1: Output of showDevices, showing a CPU OpenCL device

Here, a CPU device has been found on platform 0, device 0. We are using the total counter *ACC_DEVICE*, that enumerates all found devices. We see here the first found device, so the counter's value is 0. This device will be used by default by the preprocessor variable *ACC_DEVICE*. If you want to use the library on heterogeneous machines, you will need to write your own code and replace that variable accordingly.