NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

GRADUATE PROGRAM
"COMPUTER SYSTEMS TECHNOLOGY"

MASTER'S THESIS

Managing OpenCL Services using Docker Containers

Pavlos Vinieratos

Supervisors: **Alex Delis,** Professor

ATHENS

MARCH 2016

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
"ΤΕΧΝΟΛΟΓΙΑ ΣΥΣΤΗΜΑΤΩΝ ΥΠΟΛΟΓΙΣΤΩΝ"

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Διαχείριση Υπηρεσιών OpenCL χρησιμοποιώντας Docker Containers

Παύλος Βινιεράτος

Επιβλέπων: **Αλέξης Δελής,** Καθηγητής

ΑΘΗΝΑ

ΜΑΡΤΙΟΣ 2016

MASTER'S THESIS

Managing OpenCL Services using Docker Containers

Pavlos G. Vinieratos
A.M.: M1123

SUPERVISOR: **Alex Delis,** Professor

MARCH 2016

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Διαχείριση Υπηρεσιών OpenCL χρησιμοποιώντας Docker Containers

Παύλος Γ. Βινιεράτος
Α.Μ.: M1123

ΕΠΙΒΛΕΠΩΝ: **Αλέξης Δελής,** Καθηγητής

Μάρτιος 2016

ABSTRACT

Despite the proliferation of general-purpose computing facilities in the form of server farms running on virtualized infrastructures, engineers and designers often find themselves at disadvantage when it comes to efficiently executing their CPU-intensive or renderings tasks. The wide-spread use of GPUs does offer help especially under the OpenCL framework that allows for the synthesis of programs able to run on heterogeneous platforms consisting of CPUs, GPUs, FPGAs and other specialized hardware. The use of Docker can certainly ease the transition of running jobs from a local machine to available clusters. Therefore, enabling OpenCL in Docker containers that can run on GPU-equipped server farms would undoubtedly offer a significant advantage for a wide range of clients. One key problem with setting up OpenCL in such an environment is that individual GPU/hardware makers provide their own implementation. In this paper, we propose an approach that can safely support GPU-accelerated OpenCL computing inside the Docker containers used by clients. The overall objective is to enable clients transparently use OpenCL in their Docker containers. We offer a management tool that a server operator can use to effectively schedule and log the GPU-enabled containers. Using our proposal, the operator can setup the GPU and OpenCL implementations once and is subsequently capable of facilitating service to jobs requiring GPU-computing. Clients will be able to run their containers without imposing for specific scheduling requirements and without worrying about installing OpenCL implementation drivers.

SUBJECT AREA: Cloud Computing

KEYWORDS: Docker, OpenCL, heterogeneous computing

ΠΕΡΙΛΗΨΗ

Σήμερα, η χρήση server farms για εκτέλεση virtualized εφαρμογών είναι σε άνθηση. Ο καθένας μπορεί να νοικιάσει ένα instance σε ένα server farm, με περισσότερους πόρους από τον προσωπικό τους υπολογιστή, για παράδειγμα, μεγαλύτερο εύρος ζώνης για να υποστηρίξει τη φιλοξενία ενός δικτυακού τόπου. Μηχανικοί, επιστήμονες και οι σχεδιαστές μπορούν να εκμεταλλευτούν τέτοια instances για πιο αποδοτική εκτέλεση CPU-intensive ή GPU-intensive εφαρμογών. Η ευρεία χρήση καρτών γραφικών βοηθά αρκετά, αξιοποιώντας το framework OpenCL, που επιτρέπει το compilation και την εκτέλεση προγραμμάτων σε ετερογενές υλισμικό, αποτελούμενο από CPU, GPU, FPGA και άλλο specialized hardware. Η χρήση του Docker μπορεί να διευκολύνει τη μετάβαση της εκτέλεσης εφαρμογών από προσωπικό υπολογιστή σε server farm. Επομένως, η δυνατότητα εκτέλεσης OpenCL μέσα σε Docker containers που τρέχουν σε GPU-equipped server farms θα προσφέρει αναμφίβολα ένα σημαντικό πλεονέκτημα για ένα ευρύ φάσμα πελατών. Ένα βασικό εμπόδιο στη διευκόλυνση αυτή, είναι οτι η κάθε εταιρία παραγωγής καρτών γραφικών προσφέρει το δικό της OpenCL implementation. Στην εργασία αυτή, προτείνουμε μια προσέγγιση που μπορεί να υποστηρίξει GPU-accelerated OpenCL computing μέσα σε Docker containers που χρησιμοποιούνται από τους πελάτες ενός server farm. Ο βασικός στόχος είναι να επιτρέψουμε στους πελάτες να χρησιμοποιούν OpenCL σε Docker containers. Προσφέρουμε ένα εργαλείο διαχείρισης που ένας διαχειριστής server farm μπορεί να χρησιμοποιήσει, για αποτελεσματικό προγραμματισμό και καταγραφή των GPU-enabled containers. Χρησιμοποιώντας την πρότασή μας, ο διαχειριστής μπορεί να ρυθμίσει την κάρτα γραφικών και το OpenCL implementation μία φορά, και στη συνέχεια να διευκολυνθεί η χρήση containers για εφαρμογές που απαιτούν GPU για την αποτελεσματική εκτέλεσή τους. Οι πελάτες θα είναι σε θέση να τρέξουν τα containers τους, χωρίς ειδικές απαιτήσεις στον προγραμματισμό τους και χωρίς την εγκατάσταση drivers και OpenCL implementations.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Cloud Computing

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Docker, OpenCL, ετερογενές υλισμικό

CONTENTS

1. INTRODUCTION 9

2. RELATED WORK 10

3. MOTIVATION 11

4. POSSIBLE SOLUTIONS AND COMPARISON 13

4.1 Master Container 13

4.2 Virtual GPUs 15

4.3 Managed Containers with Attached GPU 15

5. SYSTEM DESIGN AND IMPLEMENTATION 18

6. EVALUATION 20

7. FUTURE WORK 21

8. CONCLUSION 22

ABBREVIATIONS – ACRONYMS 23

REFERENCES 24

LIST OF IMAGES

Figure 1: One master container per GPU 14

Figure 2: One virtual GPU per container 15

Figure 3: Managed containers 16

1. INTRODUCTION

Let's start by explaining some basics needed for this paper. Docker is a relatively new project that allows packaging of an application with all its dependencies into a *container*. That container wraps everything that is needed by the application, so you can deploy that container on any machine and it will run the same. This way the developer doesn't have to care about the machine's environment, e.g. the installed version of a library being different from the one used in development or a different machine. Everything needed is specified in a *Dockerfile*, which is the recipe for building that container. Once the container is built, one can start running it, stop it, pause it etc. It works as a virtual machine, but its smaller and faster, since it's using the machine's kernel and resources, instead of having its own kernel and virtual resources. Docker makes it easy for developer teams to have a standardized development environment, and for anyone deploying applications or services on any server without having to worry about its underlying libraries and kernel.

OpenCL is a framework for writing programs that take advantage of specialized hardware, and can use different kinds of hardware. That kind of programming is called *heterogeneous programming*. A GPU, for example, is one type of specialized hardware, as it can do certain calculations much more efficiently than a CPU. By using small cores in big numbers, that are tuned for specific calculations, a GPU can be more efficient in some cases that a CPU. In contrast, a CPU is much more general. It is able to do a great number of jobs with acceptable performance, but is not specialized for all of them. Another type of specialized hardware is a custom FPGA (Field-Programmable Gate Array) for deep learning [1] [3], or bitcoin mining [4].

In OpenCL there is the concept of *kernels*. Kernels are programs that are specialized for running on the hardware that supports OpenCL. While the main application is running, the kernels are compiled and optimized for the hardware they are going to run on. This allows for better performance. Even more so, because they can also run in parallel. OpenCL is very useful for developing and using applications for scientific research, data mining, graphics rendering, medical diagnosis etc., because of the many calculations that these applications entail. Using GPUs can speed up these applications by orders of magnitude, therefore more work can be done.

2. RELATED WORK

While researching for current solutions, we found one paper [15] that uses virtual GPU's attached to virtual machines, and an API. They made a Linux driver for the virtual GPU, and all the virtual machines could run at the same time, doing their requests through their API. It's a good way to have high utilization of the GPU, but its very hard to keep it updated. Also, it has to be adjusted and optimized for each GPU model. In this paper, we will describe a more general solution.

Currently there is some research going on about sharing GPUs [7] by NVIDIA, but it's about GPUs on desktop computers, and not servers. On the other hand, there is more active research and development of APIs mapping OpenCL to custom hardware [1] [3] that could be combined with our proposal in this paper, to provide a widely available heterogeneous computing experience to server farms, and the possibility of more specialized server farms, that have specific hardware [4] attached to them.

3. MOTIVATION

In order to use OpenCL, some setting up is needed. CUDA (NVIDIA) [10], AMDAPP (AMD) [8] and Intel's own OpenCL [11], are the three most famous and used implementations of the common OpenCL framework API. Depending on the GPU on a machine, one has to install and setup the specific vendor's implementation and drivers. After that, the API is the same and can be used transparently. When a researcher, for example, wants to run his data mining program on his machine while developing, he sets the machine up. When the same researcher wants to run the program on a stronger machine than his own, let's say the university's server, he has to set OpenCL up again. Of course this might need the presence or assistance of the server maintainer and administrator. That would be required every time the OpenCL API or implementation is updated. This whole process is not productive, and it can take up valuable time from the researcher. That becomes even harder to do if he rents a GPU-equipped server, because then he has the same OpenCL installation problem, plus the problem of different libraries and his application's dependencies, that Docker tries to solve.

Using Docker, a researcher can create a Dockerfile describing the dependencies of his project, which usually are mathematical or rendering libraries, and then running some simulations. A graphic designer can make a Dockerfile with some visual rendering libraries and applications, and have it run to generate an HD video out of a raw video plus the edits. A hospital can create Dockerfiles that use medical and simulation libraries to run tests, helping diagnose a patient. Having the Dockerfile, one can build a Docker container and run an application on a local machine, and just as easily, upload the Dockerfile to a rented instance of a server farm and run the same application in an identical way there too. Several companies nowadays, like Amazon [2], Tutum [6], Orchard [5] allow people to submit their Dockerfiles, and build containers to run on their server farms, while charging for usage time. The last few years, utilization of such server farms has increased, and anyone can rent an instance. If someone needs a powerful machine for a specific project, it's much cheaper to rent an instance in a powerful server farm than to build or buy an equal machine resource-wise. A feature that these instances have, thanks to Docker, is that they run in total isolation between each other. This protects the host machine, and each instance from the others. If one fails, it doesn't take the whole system down. If some connections are required between instances of a client, then these are specified by the client, and during the instance initialization, the needed local connections are made. Some server farms have GPUs attached to them, therefore enabling people to use the GPUs for calculations, renderings or any other job that can highly benefit from using them. One way to do that is by running OpenCL inside Docker containers.

People have a hard time using OpenCL inside a Docker container, because of the setup that is required up front. Even if someone moves a Dockerfile to a server farm instance, they still have to setup OpenCL by finding info of the available GPU, searching if it supports OpenCL and which version, download that OpenCL implementation from the GPU's vendor website, install it, and make sure its set up correctly by running a few testing and benchmarking applications. After that, they can finally upload the Dockerfile and run as they do on a local machine. Of course, the *installing* step, might not even be possible with some server farms, because the instance might not have root privileges, so the client might need to call the server farm support and have them install it for them. This whole process can range from being very hard and time-consuming, to nearly impossible.

We want to simplify this process. We want to keep the Dockerfile self-contained, and let that same Dockerfile be the only requirement for someone to run their application on a rented, GPU-equipped server farm instance, without having to worry about the setup part of the process. Our proposal will allow anyone to upload their Dockerfile, and simply tick a checkbox on the server farm's website if they would like to utilize the GPU via OpenCL. We also provide the server farm owner with instructions on how to setup OpenCL on the server farm, and lastly provide a management application, that builds, schedules, runs and pauses containers of clients, and bills them for usage time. That application can be easily adjusted and reworked to fit pre-existing management application on existing server farms.

4. POSSIBLE SOLUTIONS AND COMPARISON

Currently, when a server farm has GPUs available, it attaches them to an instance, which is a virtual machine that sometimes has real hardware, or parts of it, attached. Virtual hardware can also be attached to the instances. For example, a server farm that has 4 CPUs with 8 cores each, can do many instance-hardware combinations. It could host 4 instances with 1 real CPU each, or 8 instances with 1 virtual CPU that has 4 real cores each. I could also even host 8 instances with 1 virtual CPU with 16 virtual cores each. In that last case, the load of the virtual cores is split and shared across the real cores of the host machine. The same can happen with GPUs, but GPU cores are not split. There are some early projects by NVIDIA [7] that try to split the GPU, but it's currently focused on desktop machines that need 2-3 virtual GPUs, and not servers that would need hundreds or thousands.

It becomes apparent that the problem of sharing GPUs, on personal computers or server farms, has no clear solution yet. The same problem exists for Docker containers running on a server farm. For that reason, we need to do some scheduling of the containers, so that only one container can get access to a GPU unit at a time.

There are three ways that could solve this problem. One way, is to have a master Docker container with access to the GPUs, that is managed by the server administrator. Another way, is to have one virtual GPU per client container. Lastly, a third way is to attach a GPU to the containers that need it. In all three possible solutions, only one container per GPU can run. So the number of available GPUs on a server farm, limits the number of GPU-enabled Docker containers that can be running at the same time. For example, on a server farm with 6 available GPUs, that limit would be 6. The rest of the containers that need a GPU will be paused and have to wait to be scheduled. Containers that don't need a GPU can run in any way the server farm owner wants.

4.1 Master Container

For this possible solution we would need one master container per GPU, with that GPU attached to it. When a client container needs a GPU, then it would have to send a message to a master container. After that, the master container would ask the client container for the OpenCL kernel files, any executables, the inputs and the expected output format (a file, a directory, no output, etc.). This method, illustrated in figure 1, gives the server administrator power over which container runs and when, so there is a billable unit (e.g. use of GPU for 1 hour).

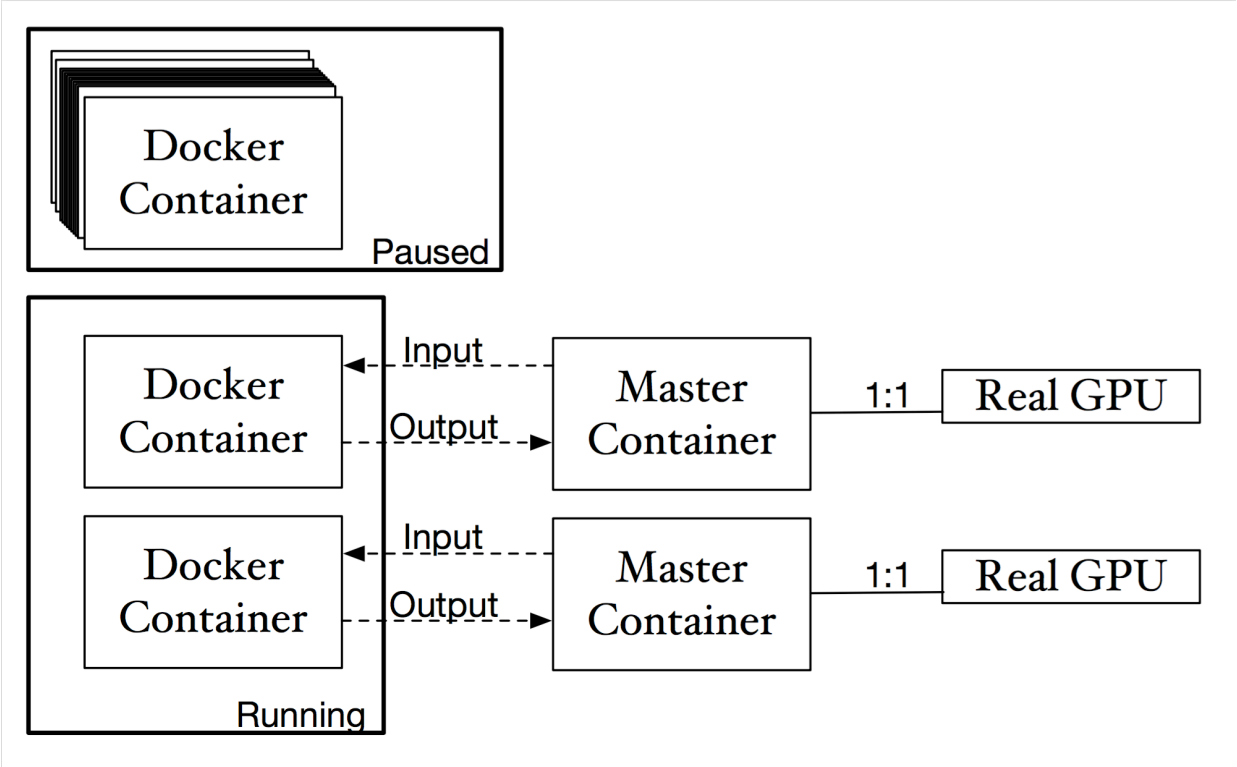

Figure 1: One master container per GPU

One problem with this possible solution is that the clients will still have to use some kind of API to communicate with the master container, which most probably would change from server farm to server farm, depending on the implementation. Different dev teams will create different APIs and implementations, so we cannot be certain that one server farm will provide the same usage as another. This would be a new problem that people who want to run containers on server farms would have. It's better than the original hassle, but not good enough. Also, another problem is that the server administrator would have to design this API, implement it and enforce it. Lastly, the master container would need to run executables that might need libraries it doesn't have installed, so the container would have to have an even bigger API that allows for dependencies.

4.2 Virtual GPUs

The possible solution of the virtual GPUs is that each client container would have a *vGPU* attached that will act as the messenger between the host machine and the client containers. For this solution, the server farm owner would need to design and implement virtual GPU system device driver, which is not an easy task, because it's not a one-size-fit-all solution. When an attached virtual GPU is asked to be used inside the container, the host machine would be notified that the specific client container tried to use the attached vGPU, and the host machine would immediately pause the client container, detach the vGPU and attach a real GPU, and then resume the container.

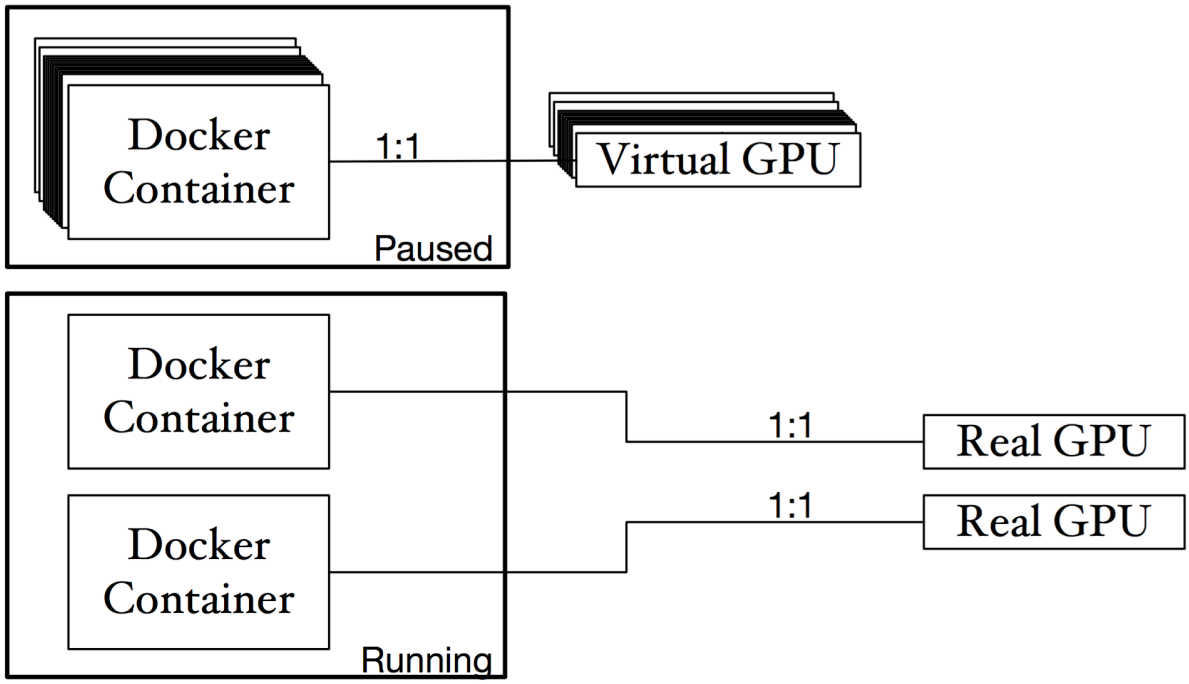

Figure 2: One virtual GPU per container

The problem with this approach, shown in figure 2 is the constant swapping that would need to happen. Some operating systems do not support hot-swapping of hardware, so if one of them runs inside that client container, it will crash, and the client would lose all the progress made so far by that container. Maybe we could avoid the hot-swapping if we could forward the requests made to the vGPU to our real GPU, but then the billing logic will be inside the vGPU device driver. That is not ideal, since a system device driver should not contain the company's billing logic.

4.3 Managed Containers with Attached GPU

This is the solution we chose to implement in this paper. When a client container needs the GPU, the client has to specify it beforehand. That's a small and easy change. It can be done through the server farm website by ticking a checkbox, or even inside the Dockerfile of the container, by adding a certain commented line. When the server knows that a specific container needs the GPU, it can automatically start it with a real GPU attached. What this means is that more that one containers will have the same real GPU attached. The benefit of this solution, as seen in figure 3, is the existence of a management application deployed on the server farm by the administrator. With it, we can make sure that only one container will run at any time, that will have full access to a

real GPU, and the rest of the containers will be either paused, attached to another GPU, or running without the need for a GPU.

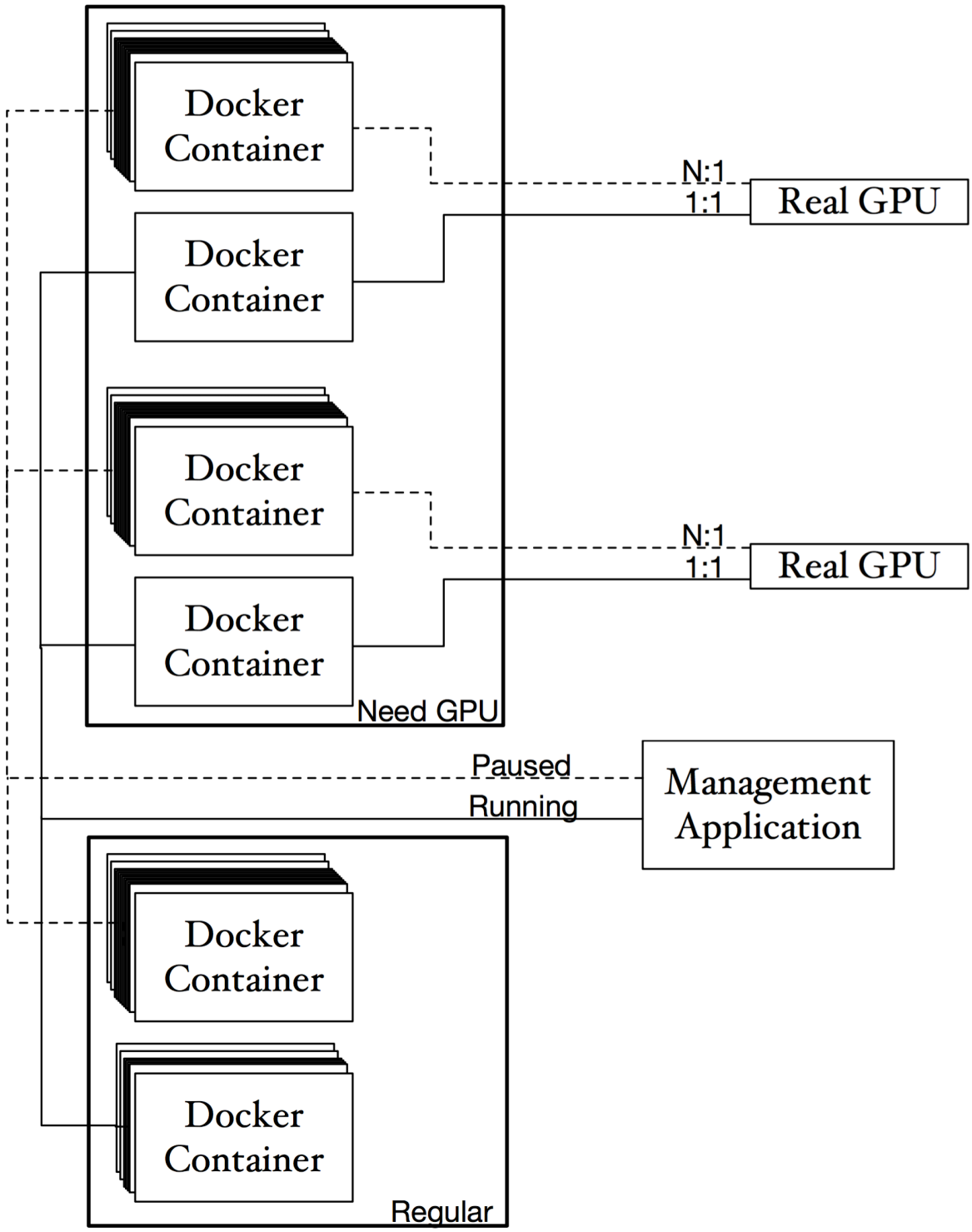

Figure 3: Managed containers

This way each server administrator can easily control the billing unit, and the containers that need GPU will always have complete access to all the cores of the GPU. After the billable unit of time has passed, the running container can be paused, and another container can be chosen to resume, based on a scheduler that can be decided by the server administrator. Notice that with this approach, if a container is marked as *needs-GPU*, then the client is going to be billed regardless of the actual usage of the GPU.

This means that it's great value for containers that render graphics or do other GPU-intensive jobs. If a client needs to run some processes that require a GPU and also some other processes that don't need a GPU (e.g. a web server), the best value for the project would be to have multiple containers. Some containers will be used for the GPU-related jobs, and others will be used for the jobs that don't explicitly need a GPU. All of the containers will be able to communicate with each other, by taking advantage of the way Docker connects containers, using local ports. This compartmentalization and separation is exactly what Docker is built for.

5. SYSTEM DESIGN AND IMPLEMENTATION

Starting this project, we had a computer with Linux installed, equipped with an AMD GPU. That computer will play the role of a server farm. We first looked up the GPU model, to find the OpenCL version that is compatible with it. Then downloaded and installed AMDAPP (AMD's OpenCL implementation), with the GPU's drivers and OpenCL headers, for the specific GPU model. We then verified that OpenCL programs acknowledge the existence of the GPU as a calculation unit, and made sure we could use it for actual calculations. After that, we created a simple Docker container, attached the GPU to it, and ran the same OpenCL test programs. To attach a GPU to a Docker container, we have to add the *--device* flag, pointing to the */dev* file that points to the GPU of the machine. After making sure that OpenCL works inside the container, we wrote the management program that preprocesses the client Dockerfiles and checks if they need the GPU. If they do, it keeps them in a list of GPU enabled containers. A container is then chosen out of that list, and is allowed to run for a certain amount of time, and then is paused. Then another container is chosen. This process goes on and on.

This procedure can be done by a GPU-equipped server farm owner to provide clients with easy access to the GPUs. The owner will have to find the GPU model, and check which OpenCL version is compatible with it. The next step would be to download and install the drivers and OpenCL implementation for that specific GPU model. In case of many GPUs that are the same model, no change to the procedure is needed. If the GPUs are different models or from different vendors, they would have to be attached to another server of the server farm. Unfortunately, that's a limitation of the operating systems. Afterwards, the owner should make sure OpenCL works by running OpenCL tests and benchmarks. The next step is to run the same tests and benchmarks, but this time inside a Docker container. After all that is done, the server farm is ready to be used for GPU-intensive tasks inside Docker containers.

Another requirement for the owner is that they have to add the interface for selecting if a Docker container needs a GPU to the client account website. This setting can be as simple as a checkbox, and should be set per container. As we have it, this setting can also be set by adding a commented line to the Dockerfile of a container, such as *#MGMT_NEEDS_GPU 1* or with an environment variable *ENV MGMT_NEEDS_GPU 1*. This line can be parsed by the server when the client uploads the Dockerfile.

For a client, there is only one change that has to be done, to be able to run their container on a server farm, and that is the *needs-GPU* setting we described above. After that, they can use their Dockerfile to build their container on their local machine, and then they can upload the Dockerfile to the server farm, and use it the exact same way they would on their local machine. By adding the line on the Dockerfile, this process of setting up becomes easier than ticking the checkbox on the server farm's website. If it's a commented line, it will be disregarded by other machines or servers that don't have the proposed setup. If it's an environment variable, it will not affect the execution on other machines or servers since they will not use that variable.

Last but not least comes the management application. Currently, the server farms that provide people the ability to run virtual machines or Docker containers as a service, already have a way to schedule and calculate the billing for each of them. And that depends on their business logic. We provide a management application that does the scheduling of the Docker containers based on the *round robin* scheduler, and *bills* a container one credit per unit of time, which is set to 10 seconds, for demonstrative

purposes. A server farm owner can simply extract the part of our application that determines if a container needs a GPU or not, and add it to their existing management application. They will keep using their own schedulers and their own way of billing.

Our management application does the following. Initially, it scans a predefined directory on the system that the clients Dockerfiles are in. For each Dockerfile, it checks if it contains the special *needs GPU* line. After that, it builds a Docker container out of each Dockerfile. It stores the names of all the containers in two lists, by separating the ones that need a GPU from the ones that don't. Then the scheduling begins. Every 10 seconds, the application asks the scheduler, which is *round robin*, to tell us which container is the next to run. So the first time the scheduler is asked, it will return the first container. The second time, it will return the second container, etc. When all the containers have run for 10 seconds, the scheduler will return the first container again, like in the beginning. When the application knows which container is the next to run, it has to check if it's already running. In that case, it lets it continue for the next 10 seconds, while also adding another billable unit to it. Otherwise, it will pause the currently running container, and prepare the next in line. The preparation is to unpause it, if it has previously run, or to initialize it if it hasn't. While this process is taking place, a similar process for the non-GPU containers also takes place. The difference between the two processes, is that we can have one GPU-enabled container running for each GPU on the server, but we can have many CPU-bound containers mapped to each CPU. In the end, when the application is signaled to end, it stops all running containers, and does some cleanup.

6. EVALUATION

As stated before, the changes that need to happen, from the users' perspective, are minimal. They just have to add a commented line or an environment variable to their Dockerfile. That does not affect the execution of the Docker container in any way if it's running on any other machine, but it most certainly helps when it's running on a GPU-equipped server farm with the structure we proposed in this paper.

On the server farm's side, the setup of OpenCL implementation needs to be done internally. This way the installation can be optimized, since the owner has total access to the hardware, and can be used transparently by the clients, without them needing to know any details about the implementation and installation. The owner, via the website, would provide clients with the specs of the GPUs available to them, so they can roughly compare the efficiency gains from running their containers on that server farm, compared to their local machine. Of course, different GPU models can be available, with different pricing.

We ran the management application and it shows that all containers can take advantage of the available GPUs, one container per GPU at a time, for full access. The setup for the management application can be easily adjusted and adapted to work with an already existing management application for server farms. In our runs, the GPU was utilized at 100%, since one GPU container was running after the other. In our runs, a Docker container can be paused in 47ms and unpaused in 37ms. These numbers do not change no matter how many containers are active on the server. So in roughly 85ms for every time unit, the previously running GPU-enabled container, and the next one is unpaused and allowed to run.

Compared to the other research we found [13] [15], our proposal is more general, since the server farm team is doing the low level work, that needs specialization and total access to the hardware, and it's very approachable for people like medical experts, researchers, designers etc. to take advantage of the powerful hardware on server farms.

7. FUTURE WORK

In the future we would like to experiment with different specialized hardware. The setup would be the same, but the implementation details will be different. Custom FPGAs could be attached, and after installing an OpenCL implementation that covers them, they can also be used by Docker containers. This way, server farms can be more specialized in specific areas of development and application execution, not necessarily OpenCL, such as *Internet of Things* [12] applications. If a server farm has Arduino [9] or Raspberry Pi [14] boards attached, Docker containers can use them for development and testing of IoT applications, before shipping to customers. We think this is an interesting direction this project can take.

8. CONCLUSION

This paper described how to setup the software and hardware of a server farm to provide a better experience for clients who use GPU-enabled Docker containers on this server farm. We suggest that instead of each client setting up his own containers to work with the underlying hardware of the server farm, the server administrator is the one who should set that up. This way the clients can easily run their containers on their local machine or other server farms without readjusting their software and setup. The server administrator will be able to do a better job setting the software up, since he has total control over the software and hardware of the farm, and can use professional help for optimization. Also, we provide a management application that can be adjusted and retrofitted to the existing management application of any server farm, in order to make the necessary connections between the hardware and the containers.

ABBREVIATIONS – ACRONYMS

OpenCL	Open Computing Language
CPU	Central Processing Unit
GPU	Graphics Processing Unit
vGPU	Virtual Graphics Processing Unit
FPGA	Field-Programmable Gate Array
API	Application Program Interface
IoT	Internet of Things

REFERENCES

[1] Docker, https://www.docker.com [Προσπελάστηκε 3/3/16]
[2] OpenCL, https://www.khronos.org/opencl [Προσπελάστηκε 3/3/16]
[3] Altera for OpenCL, https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.highResolutionDisplay.html [Προσπελάστηκε 3/3/16]
[4] CUDA, https://developer.nvidia.com/accelerated-computing-resources [Προσπελάστηκε 3/3/16]
[5] AMDAPP, http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk [Προσπελάστηκε 3/3/16]
[6] Intel OpenCL, https://software.intel.com/en-us/intel-opencl [Προσπελάστηκε 3/3/16]
[7] Folding at Home, https://folding.stanford.edu/home [Προσπελάστηκε 3/3/16]
[8] Altera for OpenCL, https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.highResolutionDisplay.html [Προσπελάστηκε 3/3/16]
[9] Nervana, http://www.nervanasys.com [Προσπελάστηκε 3/3/16]
[10] Open Source FPGA Bitcoin Miner, https://www.github.com/progranism/Open-Source-FPGA-Bitcoin-Miner [Προσπελάστηκε 3/3/16]
[11] Amazon EC2 Container Service, http://aws.amazon.com/documentation/ecs [Προσπελάστηκε 3/3/16]
[12] Tutum - Docker Hosting, https://www.tutum.co [Προσπελάστηκε 3/3/16]
[13] Orchard - Instant Docker hosts in the cloud, https://www.orchardup.com [Προσπελάστηκε 3/3/16]
[14] Virtual GPU Technology, http://www.nvidia.com/object/grid-technology.html [Προσπελάστηκε 3/3/16]
[15] Internet of Things Global Standards Initiative, http://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx [Προσπελάστηκε 3/3/16]
[16] Arduino, https://www.arduino.cc [Προσπελάστηκε 3/3/16]
[17] Raspberry Pi, https://www.raspberrypi.org [Προσπελάστηκε 3/3/16]
[18] NVIDIA Docker, https://www.github.com/NVIDIA/nvidia-docker [Προσπελάστηκε 3/3/16]
[19] Tsan-Rong Tien and Yi-Ping You, “Enabling OpenCL support for GPGPU in Kernel-based Virtual Machine", *Software - Practice and Experience (44:483-510),* 2014
[20] J. P. Walters et al., “GPU Passthrough Performance: A Comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL Applications", *Cloud Computing (CLOUD) (636-643),* 2014.