Enhancing the Usability and Utilization of Accelerated Architectures via Docker

Nicholas Haydel
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, USA
nhaydel@nd.edu

Gregory Madey
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, USA
gmadey@nd.edu

Sandra Gesing
Center for Research Computing, Dept. of Computer Science and Engineering, University of Notre Dame
Notre Dame, USA
sandra.gesing@nd.edu

Abdul Dakkak, Simon Garcia de Gonzalo
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana-Champaign, USA
dakkak@illinois.edu, Grcdgnz2@illinois.edu

Ian Taylor
University of Notre Dame, USA
And Cardiff University, UK
itaylor1@nd.edu

Wen-mei W. Hwu
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana-Champaign, USA
w-hwu@illinois.edu

***Abstract*— Accelerated architectures such as GPUs (Graphics Processing Units) and MICs (Many Integrated Cores) have been proven to increase the performance of many algorithms compared to their CPU counterparts and are widely available in local, campus-wide and national infrastructures, however, their utilization is not following the same pace as their deployment. Reasons for the underutilization lay partly on the software side with proprietary and complex interfaces for development and usage. A common API providing an extra layer to abstract the differences and specific characteristics of those architectures would deliver a far more portable interface for application developers. This cloud challenge proposal presents such an API that addresses these issues using a container-based approach. The resulting environment provides Docker-based containers for deploying accelerator libraries, such as CUDA Toolkit, OpenCL and OpenACC, onto a wide variety of different platforms and operating systems. By leveraging the container approach, we can overlay accelerator libraries onto the host without needing to be concerned about the intricacies of underlying operating system of the host. Docker therefore provides the advantage of being easily applicable on diverse architectures, virtualizing the necessary environment and including libraries as well as applications in a standardized way. The novelty of our approach is the extra layer for utilization and device discovery in this layer improving the usability and uniform development of accelerated methods with direct access to resources.**

***Index Terms*—Virtualization, cloud computing, Docker, virtual machine, accelerated architectures**

I. Introduction

Cutting-edge parallel accelerators enable major performance gains on a plethora of algorithms, e.g., the accelerated versions of LTMDOpenMM [1], GPU-BLAST [2], SAMPO [3]. However, the recent advancement and power of such accelerators has grown disproportionality to their usage, even though they are available as local resources for research teams, as campus-wide resources or in distributed computing infrastructures, such as XSEDE. In spite of significant deployment of accelerators in the latter, the usage lags dramatically. The underutilization is caused by multiple factors such as insufficient cache bandwidth for diverse applications in the case of Xeon Phis, proprietary interfaces, and device discovery in general. While the first factor is a problem that should be tackled by hardware providers, the second and third problems can be tackled via software approaches. Recent toolkits, such as the dedicated CUDA Toolkit for NVIDIA GPUs [4], as well as OpenCL [5] and OpenACC [6], are generally applicable for GPUs, APUs, and many-core coprocessors providing extremely advanced APIs for utilizing such architectures. However, developers must become deeply familiar with diverse architectures and their specific characteristics. Additionally, there are duplicated APIs for common utilities e.g. to query for available devices and device memory or the utilization of resources. Furthermore, the developers often need to redevelop their code for each type and generation of hardware. A common API therefore, would provide a far more portable architecture for application developers. This cloud challenge proposal addresses this issue

978-0-7695-5697-0/15 $31.00 © 2015 IEEE
DOI 10.1109/UCC.2015.57

IEEE computer society

by providing a fully featured runtime environment, for transparent targeted deployment of libraries, drivers and applications for OpenCL, CUDA Toolkit and OpenACC. To achieve this we use Docker-based [7] accelerator containers to provide a virtualization of these libraries for deployment onto various infrastructures.

Virtualization is the foundation of cloud computing to enable virtual machine (VM) user environments on top of the physical hardware. VMs allow developers to optimize their applications for specific environments in order to decrease the development time and increase efficiency. However, these images can be large and require a hypervisor to facilitate communication between the virtual machine and the host operating system, thus hampering portability. Containers, on the other hand, are based on shared operating systems and thus more lightweight and efficient than hypervisors. Instead of virtualizing hardware, containers 1) rest on top of a single Linux instance with the ability to provide incremental snapshots, 2) do not require the duplication that VMs need, and 3) provide a small, neat capsule containing the target application and its dependencies. Using a tuned container system can often provide four-to-six times as many server instances per capacity compared to VMs. Containers use the Linux LXC user space tools, where applications run with their own file system, storage, CPU, and memory. While the hypervisor abstracts an entire device, containers just abstract the operating system kernel.

In this work we are motivated to provide a virtual container-based accelerator environment to give developers the freedom in which to develop the tools necessary for research for deploying their accelerator-based applications. Docker offers a lightweight, portable solution while providing necessary virtual runtime environments. Docker containers neatly encapsulate the application and the necessary dependencies, without sacrificing performance. In this proposal, we provide the architecture for the system and demonstrate that using Docker containers to encapsulate the accelerator toolkits is a viable proposition. We will show that by using this approach compared to a native installation, there is minimal performance loss when running applications that utilize the available accelerators such as the GPU or MICs.

The remainder of this paper is organized in the following way. The next section provides related work in the area of accelerated toolkits and approaches. Section 3 describes the architecture for the approach. Section 4 provides performance measurements we have recorded using this system. Section 5 discusses the portability of the Docker containers and Section 6 provides the scalability characteristics of our approach. Section 7 discusses the limitations of the approach and in Section 8, we conclude the work and briefly discuss future work we intend to do.

II. Related Work

Related studies to our work target two diverse areas. The first are studies investigating whether Docker compared to native solutions is feasible from a performance point of view. Many of them are concerned with whether or not Docker can close the performance gap between running applications in the native environment and running them inside a virtual machine. The second area involves approaches to ease the access to diverse architectures via unified libraries and APIs to address the challenge of utilizing novel architectures in the fast changing hardware landscape. While building a common API is future work, we have evaluated suitable frameworks.

A. Performance Studies

The first area is addressed by diverse publications. Tommaso *et al.* [8] examine the impact of Docker containers on the performance of bioinformatic tools when compared to the native system. The paper specifies three use cases concluding that the effect of Docker containerization of applications on their execution performance is negligible. Felter *et al.* [9] provide a comparison of native, container, and virtual machine environments using benchmarks relevant to cloud computing. The paper focuses on the performance impact on server workloads, concluding that, in many cases, container performance is equal to native performance while virtual machines tend to fall short. Morebito *et al.* [10] provide an evaluation of virtual machines, Docker, and native performance using a variety of benchmarking tools. The paper concludes that applications running in containers suffer from minimal performance loss without the overhead associated with virtual machines.

The majority of the evidence suggesting the viability of the Docker platform as a lightweight virtualization solution focuses on benchmarking Docker with applications run entirely on the CPU. In this proposal we provide evidence suggesting containerization as a viable solution for increasing the accessibility of accelerated approaches. While there are containers designed to facilitate the use of GPU's and MICs these containers are scarce.

B. Unified Libraries and APIs

The second area includes solutions with diverse foci for unifying access to diverse architectures. Various studies have been undertaken on performance portability techniques to adapt to changing hardware. We intend to use Tangram [11] as a C++ language extension that is applicable for diverse accelerated architectures and CPUs. Tangram is based on the idea that complex and highly optimized algorithm code for different architectures can be automatically generated from a set of simple code fragments called codelets.

In the Tangram language, a codelet is a code fragment that implements a particular computation. A spectrum is a unique computation associated with a collection of functionally equivalent codelets. Codelets in the same spectrum have the same name and function signature, but have different implementations. They can be implemented using different algorithms or the same algorithm but different optimization techniques.

There are different types of codelets. Codelets can be atomic or compound. Atomic codelets are self-contained while compound codelets invoke other spectrums. Compound codelets can be recursive if they invoke their own spectrum. Codelets can be scalar or vector. The difference is that scalar

codelets are oblivious to other elements of the same vector, whereas vector codelets can communicate between them under the assumption of Single-Instruction-Multiple-Data execution timing. In other words, vector codelets may perform actions unique to vector execution and are not meant to be scalarized. Both scalar and vector atomic codelets define computations on an element of an aggregate data structure (similar to OpenCL/CUDA) and both are vectorizable. If a spectrum is invoked from a function that is not a codelet, then that function is called a host function.

The Tangram language is designed as an extension of C++ with the qualifiers, primitives, and built-in containers (as in functional languages). Tangram's qualifiers assist with codelet management (codelet, tag, env), vectorization (vector, shared), and communicating optimization hints (mutable, tunable). Tangram's primitives assist with expressing common work partitioning and data parallel operations. Tangram's containers – coupled with the primitives – make characterization of memory access patterns and data flow easier, thereby making transformations such as coarsening, tiling, vectorization, privatization, blocking, and thread-level parallelization robust and predictable.

The Tangram language also provides support for expressing atomic and compound codelets, map and iterator utilities for expressing data parallelism, vector annotations, and tuning knobs. The Tangram compiler is implemented with the Clang and LLVM infrastructure. The compiler applies coarsening of mapped functions, automatic data placement, hierarchical composition of compound and atomic codelets, algorithm selection, and pruning of the tuning space to generate multiple kernels for the same computation pattern customized for the target architecture. The runtime performs Simultaneous Productive Micro-Profiling (SPMP) for more accurate tuning-space pruning and input-driven adaptation.

Thread coarsening has been done for GPUs [12] as well as for compiling from GPU-like programming models to CPUs [13-20]. Data placement adaptation has been accomplished by PORPLE [21] as well as prior works that focused on rule-based methods [22]. Hierarchical composition of kernels for adapting to various architecture hierarchies via nested parallelism has been done by NESL [23]. Autotuning has been explored thoroughly in the literature [24, 25]. Tangram unifies many of these techniques in a single workflow and also adopts a run-time approach based on hardware counters [26]. Performance portability from a single source has been an area of great interest. High-level languages such as Surge [27] have been proposed for targeting CPUs and GPUs, providing containers and primitive operations to the user. Surge does not support hierarchical composition of codelets like Tangram does. Libraries of algorithms and data structures for heterogeneous computing, such as Thrust [28] have also been used to target CPUs and GPUs from a single API. However, libraries are limited to the library developer's ability to anticipate architectures and tune to them, and do not have Tangram's ability to automatically synthesize new kernels given new device specs. Petabricks [29] allows the user to define codelets function and supports composition (decomposition more precisely) and autotuning. However, Petabricks focuses more on system-level optimization like task scheduling, managing data transfer among devices, or selection from methods or library kernels. Tangram focuses on kernel synthesis and architectural optimization for performance portability. Without high-performance library kernels, performance gain from system level optimization is limited. Delite [30] supports performance portability from a single source by providing a metaprogramming framework for creating domain-specific languages, while Tangram is a general purpose language.

III. Cloud Environment Architecture

The ability to create virtual versions of operating systems, servers, storage devices or networks has fostered the development of modern computing with a heavy influence on cloud utilization. Over the past few years we have seen tremendous growth of virtualization in the area of cloud computing. Virtualization has proved particularly useful in the area of software development and distribution. Developers can optimize their software for specific environments and provide these environments in the form of virtual images. Platforms such as VMware or OpenStack provide the infrastructure on which these images can run. In addition, VMware and similar software provide a sandbox in which the applications can run without altering the configuration of the host system. This provides an extra layer of security giving developers the ability to scale the software freely. However, these images can be costly in terms of memory because they require an underlying operating system and kernel in addition to the necessary dependencies for the application. Recently, there has been an increasing trend in using Docker to virtualize the images of runtime environments. Docker provides a platform for its users to create environments with only the minimal dependencies for the application. By utilizing the host kernel, Docker images provide a scalable alternative to virtual machines, which can accommodate increasing demands for virtualizations without sacrificing performance.

Virtualization is particularly useful in the field of scientific research. Software developed for simulations or mathematical modeling are often based on certain hardware and software requirements and being able to virtualize these environments makes the applications more accessible to researchers. One such application is SAMPO (Scalable Agent-based Mosquito POint model) that uses OpenCL, a multithreading API designed to utilize the GPU to model the prevalence of malaria-vectors in a population [3, 31]. The program requires a number of environment variables and libraries to be installed in order to work, which could cause problems for researchers. However, the initial setup can be done in a Docker container, which can then be shipped as an image with the application. Researchers will then be able to run the application inside the container without needing to perform any further setup.

There are no specific hardware requirements for Docker that are different from a virtual machine. It is the difference in the necessary software systems between Docker and virtual machines that make it a much more lightweight solution to virtualization. The major difference is that Docker containers

use shared operating systems as opposed to the hypervisor system of virtual machines. This means that containers only virtualize the kernel of the host operating system instead of the entire device allowing for a more efficient use of resources. This efficiency allows users to potentially run four to six times the number of application instances as with virtual machines [32]. Figure 1 shows the structure of a host machine running multiple applications both inside and on the host machine.

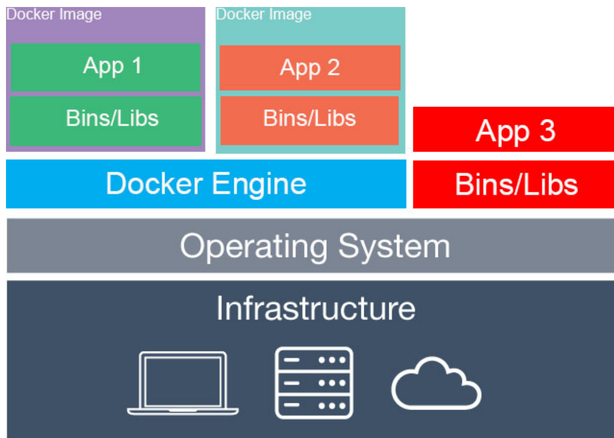

Figure 1: The software architecture of multiple containers on a host system. Containers are coordinated by the Docker engine while in parallel also other applications can run on the host machine (App 3).

In addition to removing the hypervisor, containers utilize a layered file system allowing them to share common files and libraries when necessary. This is the foundation on which the Docker engine's version system is built. Images can make use of the pre-existing libraries and binaries during the build process, decreasing build time and space consumption while increasing portability. Computing resources such as CPU and memory usage may also be dynamically allocated at the runtime of the container.

IV. PERFORMANCE

Virtual machines typically perform slower than the host system when given the same task due to unnecessary overhead that come with the installation of an entirely new system. While this may be only a minor inconvenience in some cases, in other areas, specifically research fields, where computing resources and availability are limited, execution time and size of the programs need to be minimized. Docker offers a minimal solution to this issue without sacrificing performance. This is verified through a plot of similar data for a computation heavy algorithm: SGEMM, a matrix multiplication algorithm. The specific SGEMM implementation used to generate the data utilized the GPU of the host system through the OpenCL API (see Figure 2).

For both machines the plots of the execution times overlap completely providing concrete evidence that running applications in a Docker environment does not result in performance loss. A comparison of the runtime of the SAMPO simulation on a host machine with an NVIDIA GeForce GTX 700 GPU and in a Docker container on the same machine suggests that there is little to no difference between running programs on a host or in a Docker environment.

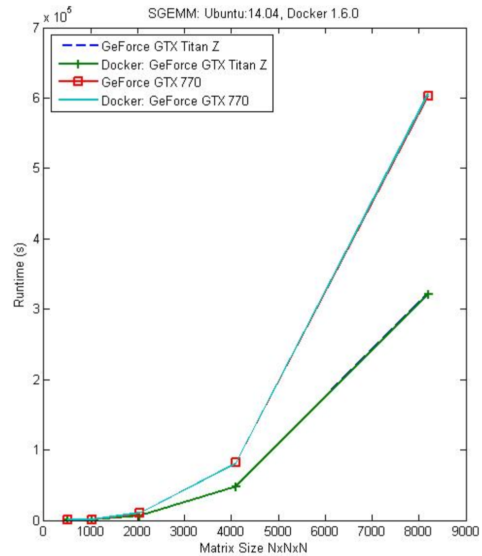

Figure 2. Execution times for the SGEMM algorithm on a host machine and in a Docker container. Data was gathered from to different machines one with an NVIDIA GeForce GTX 770 GPU and the other with an NVIDIA GeForce GTX Titan Z GPU.

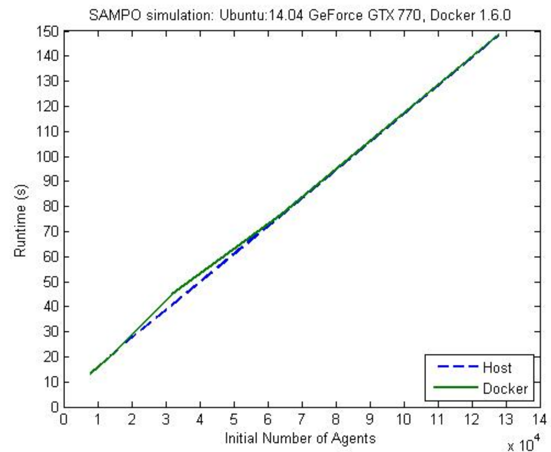

Figure 3. Run time of the SAMPO simulation both inside a Docker container and on the host machine.

Both SAMPO and the SGEMM implementation utilize the available accelerators on the machine, which the Docker engine supports through the use of runtime flags. The flag used to specify the NVIDIA GeForce GTX 770 for the SAMPO container is shown below in Figure 4.

```
--device /dev/nvidiactl:/dev/nvidiactl
```

Figure 4. Syntax for specifying available accelerators for a Docker image to access.

V. Portability

Because of its neat encapsulation of runtime environments and the ability to cater to the diverse accelerated architectures, Docker provides a portable utility for accelerated approaches. However, even without the unnecessary overhead associated with providing a secondary operating system, Docker images can become relatively large. When setting up an environment for some applications the necessary dependencies can develop into an image that might be too large to feasibly transfer over a network. Figure 5 shows a plot of the pull times for Docker images of various sizes on the Notre Dame network and an off campus network.

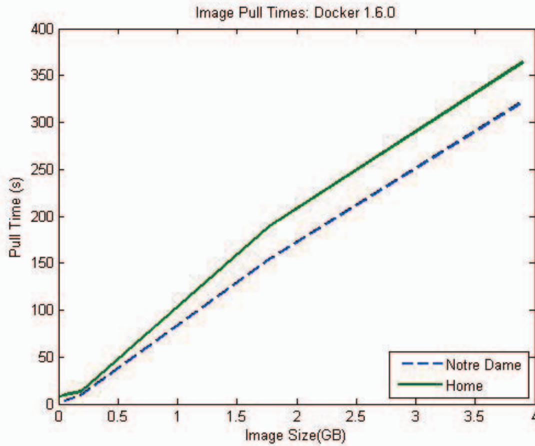

Figure 5. Elapsed time for pulling images of various sizes both on the Notre Dame secure network an on a network off campus.

The Docker engine provides an infrastructure to cater to these larger, more complex environments known as Dockerfiles. Dockerfiles contain a series of commands used to set up the environment from a base image. The Docker engine provides a framework to build the image from the series of command in a dockerfile. While the resulting environment may be hundreds of megabytes the Dockerfile from which it is built may only be a few kilobytes. The Docker platform not only allows developers to ship an image with their application, but also a Dockerfile, which the user can then build themselves resulting in a much faster software transfer.

To further reduce the pull time for applications, Docker also provide a composition tool which allows a user to define

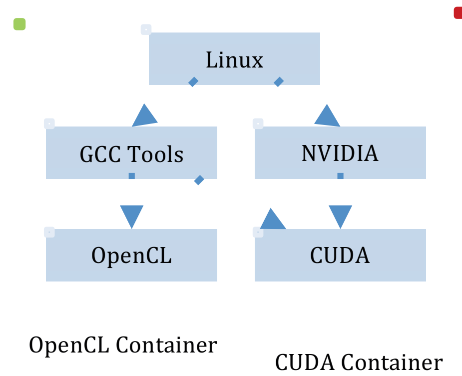

Figure 6. An example of container composition support in Docker.

a container in terms of a collection of more primitive containers. This decreases the pull time when components of an application are shared across containers. In the Figure 6, we define two containers, for OpenCL and CUDA execution. The red outline shows that a CUDA container is defined as a composition of a base Linux container, GCC tools, the NVIDIA driver, and CUDA tools. The OpenCL container, in green, is composed of a base Linux image along with the GCC and OpenCL tools. On nodes that use both CUDA and OpenCL containers, the node need not fetch the base Linux and GCC tools from the Docker repository. Like the DockerFile, the Docker composition is specified using a text file (called "`docker-compose.yml`") and are composed using the `docker-compose` tool.

VI. Scalability

The ease of creation and distribution of these custom environments gives developers the freedom to optimize their code without the need to focus on portability. This freedom that containers offer has led to widespread adoption throughout the developer and user community with more than 100,000 Dockerized applications and more than 33,000 GitHub projects with Docker in the name [11]. This growth is supplemented by other companies adopting the framework including Google.

The rapid adoption of Docker in the development community is largely a product of Docker's ability to streamline the development process. Virtual machines are shared in vendor proprietary repositories, creating vendor lock-in, which significantly hampers portability. The shared operating system approach of containers offer a more lightweight and efficient solution to the portability issue as well as performance. This feature, as well as drastically decreased overhead allows Docker images to be packaged in neat capsules containing only the application and its necessary dependencies. The Docker engine provides support for unpacking and building the images through the use of Dockerfiles and the ability to compress and load images. Efficient packaging coupled with lightweight containers

provide developers with the portability necessary to increase development and distribution speed.

The foundation for this growth is Docker's online repository for functional environments: Docker Hub. Docker Hub is a system of repositories similar to Github where users can upload images to public or private repositories. Docker users can then pull these image and then run them as containers or build off of them. As of early 2015 developers have downloaded over 320 million Docker images from the Docker Hub.

VII. Limitations

Though these containers provide less overhead than that of a virtual machine, which must include an entire operation system, containers can become large depending on the necessary libraries and binaries. Images should be minimized as much as possible in order to adequately reduce network transfer time. In addition, images can be saved to tar files then compressed or shared via Dockerfiles, which are much smaller.

In terms of portability Docker has some pitfalls, the most prominent being running the Docker engine on systems that do not use a Linux kernel. The Docker engine requires a Linux kernel for support. To get around this issue, the Docker engine is run on top of a small Linux virtual machine on systems that do not have the necessary kernel. Additionally, applications run inside Docker containers on other operating systems must support running on the host system.

VIII. Conclusions and Future Work

Docker containers offer a lightweight solution to virtualization and provide unprecedented access to accelerated approaches. Containers utilize the kernel present on the host machine providing significantly less overhead than virtual machines. Because of the lack of overhead, containerized applications perform as if they are run natively, with little performance loss. Containerized applications can easily be optimized for different accelerators as well, then shipped relatively quickly. These applications can then run inside their respective containers on the native accelerator virtually no loss of performance. The infrastructure for the growth of containerization is already in place with the series of online repositories and their web portal: Docker Hub.

While there has been significant work done benchmarking Docker containers and comparing these environments to native and virtual implementations, all of the tests focus solely on the CPU performance. This is partly because of a lack of Docker environments equipped for accelerated processing. This issue can be overcome by simply producing more of these environments and making them readily available on Docker Hub. However, with such inconsistent acceleration methods and hardware, it is difficult to create an environment supporting the diverse accelerated approaches. The development of a common API for compiling and running programs that utilize accelerated approaches could use Docker to deploy environments suited for each application. The common API would allow faster development of the necessary Docker environment and thus quicker application deployment. While we have evaluated solutions to build on and have selected Tangram as basis for such a common API, the work to accomplish it necessitates further developments. Additionally, we intend to translate Tangram code into CUDA, OpenCL, OpenACC and other identified vendor interfaces, this new work will enable very efficient portable accelerator code for developers. To accomplish this great deal of work, we have written an NSF proposal with further partners acknowledged below and if the proposal is successful, we will start with four use cases as show cases: LTMDOpenMM, GPU-BLAST, SAMPO and Axparafit. This way we can provide our solution not only to developers but also to end user communities, namely the biology and co-phylogeny communities.

ACKNOWLEDGMENT

The authors would like to thank the partners of the NSF proposal: Nikolaos Sahinidis and Nikolaos Ploskas (Carnegie Mellon University), who develop GPU-BLAST and accomplished corresponding performance tests; Sudhakar Pamidighantam (Indiana University), who brought in the SEAGrid science gateway as application area and Christopher Sweet and James Sweet (University of Notre Dame), who develop LTMDOpenMM.

References

[1] James C Sweet, Ronald J Nowling, Trevor Cickovski, Christopher R Sweet, Vijay S Pande, and Jesus A Izaguirre, "Long Timestep Molecular Dynamics on the Graphical Processing Unit," *Journal of chemical theory and computation*, pp. 9(8):3267-3281, 2013.

[2] Panagiotis D Vouzis and Nikolaos V Sahinidis, "GPU-BLAST: Using graphics processors to accelerate protein sequence alignment," *Bioinformatics*, vol. 27, no. 2, pp. 182-188, 2011.

[3] Klaus Kofler, Gregory Davis, and Sandra Gesing, "SAMPO: An Agent-based Mosquito Point Model in OpenCL," Agent-Directed Simulation Symposium (ADS 2014), Simulation Series Vol 46 #1, pp. 36-45, Curran Associates, Inc., ISBN 9781629939469, 2014.

[4] CUDA, (http://www.nvidia.com/object/cuda_home_new.html).

[5] OpenCL, (https://www.khronos.org/opencl/).

[6] OpenACC, (http://www.openacc-standard.org/).

[7] Docker (https://www.docker.com/).Paolo Di Tommaso, Emilio Palumbo, Maria Chatzou, Pablo Prieto, Michael L Heuer, Cedric Notredame, “The impact of Docker containers on the performance of genomic pipelines,” *Peer J Prints*, 2015.

[8] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. "An updated performance comparison of virtual machines and linux containers." *technology* 28 (2014): 32.

[9] Roberto Morabito, Jimmy Kjallman, Miika Komu, “Hypervisors vs. lightweight virtualization: a performance comparison,” (http://www.researchgate.net/publication/273756984).

[10] Growth Statistics (http://venturebeat.com/2015/04/14/docker-raises-95m-led-by-insight-venture-partners/)

[11] Li-Wen Chang, Abdul Dakkak, Christopher I Rodrigues, Wen-mei Hwu, “Tangram: a High-level Language for Performance Portable Code Synthesis,” Eighth Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2015), Prague, January 2015.

[12] Alberto Magni, Christophe Dubach, and Michael O'Boyle, "A large-scale cross-architecture evaluation of thread-coarsening," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013.

[13] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng D Liu, and Wen-mei W Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," Center for Reliable and High-Performance Computing, 2012.

[14] Nadav Rotem, "Intel OpenCL SDK Vectorizer," in LLVM Developer Conference Presentation, 2011.

[15] Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R Gaster, and Bixia Zheng, "Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors," in Parallel architectures and compilation techniques, 2010, pp. 205-216.

[16] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg, "pocl: A performance-portable OpenCL implementation," International Journal of Parallel Programming, pp. 1-34, 2014.

[17] Ralf Karrenberg and Sebastian Hack, "Improving performance of OpenCL on CPUs," in Compiler Construction, 2012, pp. 1-20.

[18] Hee-Seok Kim, Izzat El Hajj, John Stratton, Steven Lumetta, and Hwu Wen-Mei, "Locality-centric thread scheduling for bulk-synchronous programming models on CPU architectures," in Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, 2015, pp. 257-268.

[19] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee, "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters," Proceedings of the 26th ACM international conference on Supercomputing, pp. 341-352, 2012.

[20] John A Stratton, Sam S Stone, and Hwu W Wen-Mei, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," Languages and Compilers for Parallel Computing, pp. 16-30, 2008.

[21] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen, "Porple: An extensible optimizer for portable data placement on GPU," in Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014, pp. 88-100.

[22] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," Parallel and Distributed Systems, IEEE Transactions on, vol. 22, pp. 105-118, 2011.

[23] Guy Blelloch, "Nesl: A nested data-parallel language," Pittsburgh, Technical Report 1992.

[24] Clint R Whaley, Antoine Petitet, and Jack Dongarra, "Automated empirical optimizations of software and the ATLAS project," Parallel Computing, vol. 27, pp. 3-35, 2001.

[25] Markus Püschel, José Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert Johnson, "Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Alogorithms," International Journal of High Performance Computing Applications, vol. 18, pp. 21-45, 2004.

[26] Shirley Browne, Jack Dongarra, Nathan Garner, Kevin London, and Philip Mucci, "A scalable cross-platform infrastructure for application performance tuning using hardware counters," in Supercomputing, ACM/IEEE 2000 Conference, 2000, pp. 42-42.

[27] Saurav Muralidharan, Michael Garland, Bryan Catanzaro, Albert Sidelnik, and Mary Hall, "A collection-oriented programming model for performance portability," in Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2015, pp. 263-264.

[28] Nathan Bell and Jared Hoberock, "Thrust: A productivity-oriented library for CUDA," Astrophysics Source Code Library, vol. 1, December 2012.

[29] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe, "PetaBricks: A Language and Compiler for Algorithmic Choice," SIGPLAN Not., vol. 44, no. 6, pp. 38-49, June 2009.

[30] Arvind Sujeeth, Kevin Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," ACM Transactions on Embedded Computing Systems (TECS), vol. 13, pp. 134-134, 2014.

[31] SM Niaz Arifin, Ying Zhou, Gregory J. Davis, James E. Gentile, Gregory R. Madey, and Frank H. Collins. "An agent-based model of the population dynamics of Anopheles gambiae." Malaria journal 13, no. 1 (2014): 424.

[32] James Bottomley, “Why You Need to Care about Container Virtualization”, Collaboration Summit 2014, http://collaborationsummit2014.sched.org/speaker/jejb_#.VgBgNnt2zyA, 2014.