A Framework for Faster Porting of Scientific Applications Between Heterogeneous Clouds

Waseem Ahmed[1], Mohsin Khan[2](✉), Adeel Ahmed Khan[2], Rashid Mehmood[3], Abdullah Algarni[1], Aiiad Albeshri[1], and Iyad Katib[1]

[1] Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, Kingdom of Saudi Arabia
waseem.pace@gmail.com,
{amsalgarni,aaalbeshri,iakatib}@kau.edu.sa

[2] Department of Computer Science and Engineering, HKBK College of Engineering, Visvesvaraya Technological University, Bangalore, India
mohsin1510@gmail.com, khan.aak004@gmail.com

[3] High Performance Computing Center, King Abdulaziz University, Jeddah, Kingdom of Saudi Arabia
rmehmood@kau.edu.sa

Abstract. The emergence of pay-as-you-use compute clouds has enabled scientists to experiment with the latest processor architectures and accelerators. However, the lack of standardization in cloud computing, more specifically in the interoperability context, makes the task of portability of applications between clouds challenging. Two main tasks that users of multi-vendor clouds will need to perform are porting cost analysis and faster source-to-source translation. Cost analysis is essential to help evaluate the feasibility and cost of portability. And any automation of the source-to-source translation step will help developers perform the translation faster while taking advantage of platform-specific features. This paper presents a framework that assists a developer in performing these two tasks. The first task is achieved using the *Maintainability Analyzer* module which generates unique funnel shaped patterns that give an insight about the maintainability of an application and its potential for porting. Different scientific applications from various domains, that were developed using different programming paradigms, were evaluated using this module. For the second task, a set of modules use a knowledge repository to perform source-to-source translations while ensuring the maintainability of the generated code. The framework has been tested with different architecture and library combinations with promising results.

Keywords: Maintainability · Code transformation
Source-to-source translation · Portability · Heterogeneous architecture

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2018
R. Mehmood et al. (Eds.): SCITA 2017, LNICST 224, pp. 27–43, 2018.
https://doi.org/10.1007/978-3-319-94180-6_5

1 Introduction

Proliferation of the pay-as-you-use offering by compute cloud vendors has provided a new opportunity and a financially viable alternative for researchers in the high performance computing (HPC) community looking for bigger and faster computational resources. Future offerings by cloud vendors are expected to include widely heterogeneous architectures with different processor and accelerator combinations, possibly from multiple processor and accelerator vendors. Although few challenges have prevented large scale deployment of scientific applications on the cloud, the HPC-on-the-cloud community is expected to grow.

One such challenge is to enable faster portability of applications from one cloud vendor to another. While live portability of scientific applications between different cloud vendors will enable researchers to experiment with different architecture combinations, a lot of challenges will have to be overcome for this to become a practical reality. Additionally, despite the many groups working on developing comprehensive standards, the progress in the cloud interoperability context, has been slow [1]. This lack of standardization in cloud computing has made the task of switching cloud providers more challenging for the end users.

An obvious solution is to design for portability. However, portability, although a desirable characteristic in generic software applications, neither is desirable nor an aim in the design of scientific applications. In scientific applications, kernels and libraries are highly optimized for specific platforms with device-specific optimizations performed to enable them to execute most efficiently on that particular architecture. Portability, thus, is neither an important need nor objective in scientific computing. This places scientific applications in a different niche in the cloud computing world.

Portability of scientific applications in some form from one platform to another, on the other hand, is more common. The process of porting and optimizing programs and libraries is repeated whenever system architecture, tools or requirements change [2]. Considering the large lifetime of scientific applications, and the change of architectures and system tools over this period, porting and optimizations become a natural and required part of scientific application maintenance.

From the portability perspective, this poses two main challenges. Firstly, the application will have to be correctly translated to the new architecture while maximizing the utilization of the underlying computing resources. Secondly, this may need to be done more frequently to take advantage of the newer and faster computing resources that a cloud vendor may introduce; regular scaling and upgradation of their systems is important to cloud vendors to attract more customers and for market survival.

Manual translation of serial code into its correct parallel equivalent is a lengthy and complex process that involves programmer creativity, domain expertise, large solution space exploration and intricate knowledge of both the computer architecture and the programming paradigm. The last few decades have seen a lot of work in the form of experience reports, case studies and real examples on the subject of manual parallelization. Although there has been a lot of research over the past few decades on automating this task, manual parallelization continues to outperform automatic parallelization tools, in the general case. Indeed, manual parallelization continues to dominate in the GPGPU community [3]. This trend can be expected to continue as the performance gap in terms of efficiency and computing resource utilization between automatic and manual parallelization continues to widen as newer and hybrid architectures are introduced into the market [3]. Also, the varying architectural spectrum of scientific computing being witnessed in HPC will necessitate a radical change in compiler design. Traditionally, compilers have focused on homogeneous architectures; code was either compiled for the host processor(s) or a cross compiler was used to compile for a target(s) processor. As heterogeneous architectures on the HPC horizon will employ a mix of multi-vendor processors, GPUs, FPGA and other accelerators, compiling code for these heterogeneous architectures will definitely become a prerequisite for compiler development.

Until recently, porting had to be performed by researchers only when newer computer resources were made available to them. This porting process, in most cases, lasted a few months. The ratio (R_t) of *period-of-ownership* of the new platform to *time-to-translate* was sufficiently large to justify the cost of porting. But with scientific applications starting to move to the cloud, the portability process will have to be placed in a different perspective.

To address these challenges, research on developing sophisticated automatic parallelization environments and frameworks to help improve developer productivity will be needed as they will have an important role to play in inter-cloud application porting. Also, the need to focus on maintainability and modularity during scientific application development will become more essential to facilitate easier portability between different architectural platforms.

This paper aims to address a few of the aforementioned challenges. More specifically, three main contributions of this work are as follows

1. A mechanism to qualitatively evaluate the feasibility of porting scientific applications between heterogeneous platforms
2. A source-to-source transformation process based on a knowledge repository to facilitate portability between widely disparate architectures
3. Approach to improve maintainability of scientific application by kernel extraction into blocks and the use of wrappers around them. While the concept of using wrappers to address interoperability problems is not new, this is the first attempt to use it to address cloud interoperability issues in HPC

The rest of the paper is organized as follows. The next section gives a background of the porting process from the perspective of scientific application development. Section 3 formally describes the relationship of the cost with other design parameters. Section 4 describes the working of the framework followed by its evaluation in Sect. 5. Section 6 places the work presented in this paper against similar work present in literature. This is followed by conclusion and future work.

2 Background

2.1 Maintenance of Scientific Applications

Unlike conventional software, requirements of scientific applications do not exhibit drastic change over time. As these applications are developed to study specific and well defined scientific phenomena which the scientists can clearly articulate and formally express, the requirements of these applications are relatively stable even over a period of time. Main changes made to software during its evolution fall in the following categories

1. When a newer environment is available (faster processors, interconnect, accelerators, programming paradigms, etc.) or if the existing platform is scaled
2. When the new environment offers faster execution and larger memory, scientists are provided with an opportunity to study larger problems, work at a finer granularity on the same problem or obtain results with a lesser degree of error, things which were not possible on the earlier platform. This scaling of the problem to take advantage of the new environment features may necessitate changes in software
3. When a more efficient algorithm or library for the application is available.

Reference to *change* or *maintenance* in the rest of the paper refers specifically to the changes listed above unless specified otherwise.

2.2 Porting Process

Consider an application that needs to be migrated from one heterogeneous platform (P_s) on a cloud to another (P_t). Consider an application, shown in Fig. 1 (a), that has been initially developed in CUDA on a GPU and uses the cublasDgemm function call from the CUBLAS library. If this code were to be migrated to a platform that uses an Intel MIC (Xeon Phi) and the Intel MKL, many changes to code will be needed even for a small and a simple application like dense matrix-matrix multiplication. Direct porting of code from P_s to P_t in such cases, although possible, is very challenging and prone to errors as the two representations are very different and do not have a one-to-one mapping as shown in Fig. 1. A better option would be to introduce an additional step in the translation process, i.e. translate the application to its equivalent representation on an intermediate reference platform (P_r) and then translate it from P_r to P_t. The primary advantage in using an intermediate platform for translation is that it reduces the number of mapping combinations required from m^2 to $2m$ (derived from [4]), where m is the number of platform options available.

In general, a translation from P_s to its intermediate representation on P_r would have a lot of platform specific code clipped or removed. For example, if the source is in CUDA, CUDA specific initializations and memory allocation routines will be eliminated in the first translation as can be seen in Fig. 1(a) and (b). Likewise, the pre-processor directives in OpenMP and OpenACC will be

```
#include<cuda_runtime.h>
#include<cublas_v2.h>
#include<helper_cuda.h>
...
cublasHandle_t hndl;
...
findCudaDevice(..);
status = cublasCreate(&hndl);
...
cudaMalloc(..);
cublasSetVector(n2,..);
...
cublasDgemm(hndl,rowmjr,...);
...
cublasGetVector(..);
...
cudaFree(..);
```

(a) Source platform (CUDA on a GPU)

```
#include <mkl.h>
...
...
cblas_dgemm(rowmjr, ...);
...
```

(b) Reference platform

```
#include <mkl.h>
#include <omp.h>
...
_declspec(target(mic))
void gemm(char rowmjr,...)   {
   cblas_dgemm(rowmjr,...);
}
...
...//Align = 64
 #pragma offload target(mic)
   in(A[0:M*K]:align(Align))
   in(B[0:K*N]:align(Align))
   inout(C[0:M*N]:align(Align))  {
    gemm(transa,...);
 }
```

(c) Target platform (OpenMP on an Intel MIC Xeon Phi)

Fig. 1. Application porting between heterogeneous architectures

removed. Platform specific library calls need to be mapped to their equivalent on the reference platform.

In the second step of the translation, i.e. from P_r to P_t, code that performs platform-specific initializations and memory allocations needs to be inserted at appropriate places. Also, generic library calls will have to be mapped to their equivalent calls on the target platform. Device-specific optimizations may need to be inserted where necessary. For example, 512-bit registers for MIC requires a 64-byte alignment and should thus, be specified in the code (Fig. 1(c)). This knowledge is absent in P_s's code.

Also, in cases where P_s has specialized extensions to standard library functions and equivalent functions are not available on either P_r or P_t, intelligent decisions based on the knowledge of the platform need to be taken. Moreover, if either P_s or P_t have multiple accelerators with static load distribution, the translation process will have to be more intelligently done as will be elaborated later in Sect. 3.3. Thus, the presence of an external knowledge repository in such a framework is very essential when platform-specific decisions and platform-specific optimizations need to be taken and made, respectively.

3 Cost of Porting

Before porting an application to another platform, it is important to assess whether it is economically more feasible to migrate the application or develop it from scratch. This section describes factors that influence porting cost.

3.1 Maintainability Related Costs

The design and maintainability of an application has a direct effect on its porting cost. An application that is poorly designed will have high maintenance; porting this application to another platform will thus involve a larger effort. Qualitatively or quantitatively determining the maintainability of an application is thus, an important step.

To determine the maintainability of an application, a *Maintainability Visualizer* has been developed as part of the framework. As described later in Sect. 4.2, the tool, besides providing a graphical summary of maintenance performed on a given application over its lifetime, helps a scientist or a developer to qualitatively evaluate the design of the architecture of the system and its potential for faster portability. The output of the module is a funnel-shaped pattern described in later sections and shown in Fig. 3.

From these funnel-shaped patterns, the maintainability cost (C_{main}) can be formally described as follows

$$C_{main} \propto \sum_{i=1}^{n_f} \delta_i - w_c n_f \tag{1}$$

where δ_i gives the magnitude of the total changes performed on source file i, w_c gives the width of the channel and n_f is the number of source files in the application. The value of w_c is user-defined and is an indicator of tolerance to change desired. A higher value of w_c indicates a higher tolerance level. In the plots shown in Fig. 3, w_c has been chosen as 0.1 (10%)

Additionally, a well designed architecture should be modular with low coupling between the components. Low coupling prevents change ripples [5] spreading to other parts of the code. This also makes the code more maintainable. Thus,

$$C_{main} \propto \sum_{i,j=1}^{n_c} g(c_i \rightarrow c_j) \forall i \neq j \tag{2}$$

where $g(c_i \rightarrow c_j)$ gives the degree of coupling (uni-directional) of component c_i with c_j.

3.2 Platform Related Costs

Besides costs related to maintainability, the source-and-target platform combination greatly influences the cost of porting. The choice of a platform to use as reference and as an intermediate representation is thus, an important decision. It should be chosen such that it is be easy to produce and easy to translate from and to different platforms. In this research, a system with a single node, single socket, single core, single thread, with no accelerator and using Intel MKL where needed is used as a reference platform (P_r).

With reference to P_r, if the cost required to develop a new application from scratch for a platform, P_i is denoted as c_i, then, an upper-bound for the porting cost from P_i to another platform, P_j or vice versa can be represented by $(c_i + c_j)$. Cost (c_i) is a function of the number and type of bottlenecks (kernels) present in the application, the number and type of library calls used in the application, and the maturity of the system tools and libraries available for P_i. For example, to convert a single *dgemm* call from a system with an Intel MIC using Intel MKL library to a system using an Nvidia K40 using *cublas*, CUDA code needs to be added for device-to-host and host-to-device data transfers, library initialization, device-specific memory allocation/deallocation and error checking while being placed correctly in the code and in the proper order. Also, the *dgemm* call needs to be replaced with its equivalent cublas call plus the additional parameters. For the reverse case, all corresponding CUDA code needs to be deleted. Thus,

$$C_{plat} \propto \sum_{i=1}^{m} (c_{lib})_i + \sum_{j=1}^{n} (c_b)_j \tag{3}$$

where, c_{lib} and c_b are platform-specific and relate to the cost of translation of a particular library call and bottleneck respectively. It can be seen from Eq. 3, that more bottlenecks or library calls an application uses, the higher will be is its associated porting cost.

From Eqs. 1, 2 and 3 we get

$$C_{mig} \propto \mathcal{F}(C_{main}, C_{plat}) \tag{4}$$

This implies that a system with low maintainability will have a higher cost of maintenance and hence a higher porting cost. Similarly, a platform that has a relatively more complex representation will be more costly to migrate.

The next section further explains how platform heterogeneity can have an increasing influence on C_{plat}.

3.3 Mapping to Heterogeneous platforms

A large portion of the scientific computing community has been porting code to accelerators to speed up their applications. In most of the early research using accelerators like GPUs, all parallelizable portions of serial code were offloaded and executed on the GPU. Significant speedups have been reported in these cases with code being craftily moved to utilize the GPU architecture. Many source-to-source translators that converted OpenMP code to CUDA [6] were introduced to ease the GPGPU application programmer's burden. However, the host CPU remained unutilized in most of these cases; a hybrid approach where the load was balanced between the CPU and GPU was not considered.

Recently, many researchers have attempted to manually overlap CPU and GPU computations by adopting a hybrid approach [7–9]. By attempting to utilize all the available computational units, the speedup obtained in these cases was much higher than with just utilizing the computational units on the GPU. But as expected, the effort (C_{plat}) involved in obtaining the load-balanced parallelized version was also much higher.

Consider a computation being performed on a computer with n number of processors (or general purpose cores) and m number of coprocessors. Also, consider that the code can be separated into a serial (c_{ser}) and parallel (c_{par}) portion. The total time to compute is given by the following

$$t_{tot} = t_{ser} + max(t_{proc_1}, ..., t_{proc_n}, t_{acc_1}, ...t_{acc_m}) + t_{com} \tag{5}$$

where t_{ser} is the time taken to execute c_{ser}, t_{proc_i} and t_{acc_j} represent time taken to execute code on the processor ($proc_i$) and accelerator (acc_j) respectively, and t_{com} is the total time for data transfers between computational units.

The following also holds true for the parallel portion of the code.

$$c_{par} = c_{proc_1} + \ldots + c_{proc_n} + c_{acc_1} + \ldots + c_{acc_m} \tag{6}$$

where c_{proc_i} and c_{acc_1} represent the portion of the parallel code executed on the host processor and accelerator respectively.

In heterogeneous computing, the objective is to minimize t_{tot} and to increase the portion of c_{par} in the provided code. For the former, while it is obvious that t_{ser} and t_{com} need to be minimized, the relationship between t_{proc} and t_{acc} and its effect on t_{tot}, and the behavior of the ratio $c_{acc}/(c_{proc} + c_{acc})$ may vary between different platforms. All these factors have an influence on C_{plat} in Eq. 3. Determining the exact relationship between these values for various platform combinations is beyond the scope of this paper.

4 Description of Framework

This section gives an overview of the framework developed for help in application portability and describes portions of the framework presented in this paper.

4.1 Overview

Figure 2 shows the block diagram of the framework. It comprises of four modules and one data repository. The *Maintainability Visualizer* analyzes the history of changes made to application code over its lifetime and summarizes this information into a graphical pattern. The *Knowledge Repo* serves as a repository that contains descriptions about (1) specific accelerators and libraries and their mappings to and from the reference platform (P_r) and (2) patterns of commonly occurring kernels (or dwarfs) in scientific applications. The *Profiler* identifies and ranks the *blocks* or segments of code that appear as bottlenecks when executed on P_r. The *Block Extractor* extracts the blocks identified by the *Profiler*

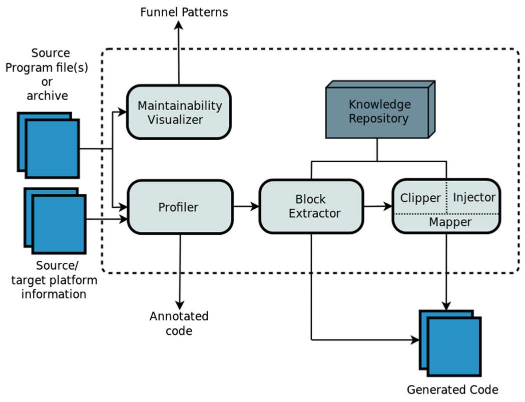

Fig. 2. Porting framework

into functions (similar to *Function extraction* in refactoring [10]) that also serves as a generic wrapper for the block. The main objective of the *Block Extractor* is to reduce the value of δ_i and $g(c_i \rightarrow c_j)$ in Eqs. 1 and 2. The *Code Transformer* consists of three complementary modules, *Clipper, Injector* and *Mapper*. The first two modules transform the application code from the source platform to its equivalent representation in P_r and from its representation in P_r to its equivalent on the target platform, respectively. The *Mapper* is responsible for code modifications on kernel patterns and library calls present in the *Knowledge Repo*.

The framework takes three inputs from the user. First is the application source which may be given as a single file, a source directory or a zipped archive. The second input provides descriptions of the existing platform - details regarding the processors, accelerators and libraries used. The third input provides similar descriptions about the target platform. The focus of this paper is on the *Maintainability Visualizer, Block Extractor* and a few functionalities of the *Code Transformer*.

4.2 Maintainability Visualizer

The primary function of this module is to provide an insight to the user about the maintainability of the application and, subsequently, aid him in taking a decision about platform porting. A higher degree of maintainability implies an easier portability. If the code base shows poor maintainability, rewriting the code for the target platform may be more cost effective.

The module uses a locally cloned Git repository for analysis. The history of each file, which includes commits, insertions, deletions, dates of commits, etc., is analyzed. Files are arranged in descending order of the number of changes made to them over their lifetime and associated with their corresponding magnitude of change. This generates an interesting funnel-shaped pattern which provides insights into the quality of software architecture and maintainability of the application. As will be seen later, these *funnel* patterns provide a good insight into the design of applications and their potential for portability.

A funnel with a relatively large mouth and a narrow rapidly tapering stem depicts an application that has been well designed, with a robust architecture where modifications over time have been restricted to a few files. For scientific applications, a funnel of this type may indicate that kernels (architecture specific code) were confined to a few files and only these files had to be changed when the application was migrated to a newer platform.

In contrast, a funnel with its mouth tapering slowly into a relatively thick stem indicates a poorly designed architecture. The cost of porting these systems to newer platforms would be more expensive. A funnel of this type would suggest rewriting of the application from scratch or a major restructuring or refactoring [10] of the application to improve its maintainability.

4.3 Knowledge Repo

The *Knowledge Repo* serves as one of the main constituents contributing to the intelligence of the framework. It is primarily used in decision making during code transformation.

Knowledge is stored in a two-level, dictionary-based scalable tree partitioned into sections. The following are a few types of knowledge that can be built in to the repository

– Mapping of standard scientific library functions from various platforms to P_r and vice versa. This includes function names, details about number, type and relative order of parameters and other specifics.
– Platform-specific device initialization calls, calls for memory allocation/deallocation and data transfers
– Platform-specific optimizations
– Patterns of commonly occurring kernels in scientific computation
– Information about platform-specific preprocessor directives

The efficiency of the framework is based on the maturity and comprehensiveness of this repository. Populating this repository with correct and pertinent knowledge about various platforms is, thus, an important pre-requisite for the efficacy of the framework.

4.4 Block Extractor

The main objective of this code is block extraction. It takes a block-annotated file from the *Profiler* as input and extracts this block into a separate function. This rearrangement of code, which has a positive influence on $g(c_i \rightarrow c_j)$ and also implicitly impacts $\sum_{i=1}^{n_f} \delta_i$, is referred to as *Function Extraction* in refactoring [10]. The block is embedded into a function with a generic signature which serves as a wrapper to facilitate easier code replacement in future by minimizing code ripples during code changes.

While the concept of using wrappers to address interoperability problems is not new [11–13], this is the first attempt of using them to address cloud interoperability issues in scientific computing.

4.5 Code Transformation

Although the basic translation of code to a newer architecture to obtain a correct functional executable is not difficult and can be performed in a reasonably short time, optimization of the ported code to take advantage of the features of the newer architecture requires a much longer time. Additionally, if the code has evolved over different architecture generations, identifying the (sometimes retained) previous platform-specific optimizations and correctly replacing them with the newer platform-specific optimizations or removing them altogether is also an intense task that requires skill. This module attempts to automate these transformation steps where possible.

Code transformation is done in three overlapping stages by the *Clipper*, *Injector* and *Mapper* sub-modules which constitute the *Code Transformer*.

The *Clipper* is responsible for deannotating, clipping or cleaning out platform-specific code from P_s's representation. For example, CUDA-specific calls (*cudamalloc)* or OpenMP-specific pre-processor directives (*#pragma offload*) are identified based on inputs from the *Knowledge Repo* and truncated from the source representation. For a few statements, relevant information from the statement is extracted into a temporary dictionary ($Dict_{temp}$) prior to clipping. If P_s's representation indicates the use of static load balancing between its heterogeneous computing units (described earlier in Sect. 3.3), pertinent function calls have to be intelligently identified and *fused* into a single call. Likewise with data.

The *Injector* module basically reverses the steps of the *Clipper* but the translation is from P_r's representation to the equivalent representation of P_t. Input from the *Profiler*, $Dict_{temp}$ and the *Knowledge Repo* is used to appropriately place OpenACC-specific preprocessor directives or CUDA-based calls (*cudasetvector*, *cudafree*), for example, where needed. Additionally, two intelligent decisions need to be taken if P_t has a heterogeneous architecture. Firstly, for every bottleneck, a decision about where to execute the kernel is to be taken; for example, some kernels may execute faster on the host itself, for some data-parallel type of kernels, it would be more profitable to execute it on the accelerator and for some, the fastest implementation might involve a combination of both. Secondly, if the choice of the first decision is a mixed implementation, the kernel (bottleneck) may need to be split into its equivalent call on each of the heterogeneous devices, if required. But only a skeleton for this set of functions can be generated as a proper load distribution between the devices will require adaptive refinement techniques on these devices to determine the best setting as described in Sect. 3.3.

The responsibility of the *Mapper* module is to search for every replaceable function or library call between the P_r and P_t representations based on data from the *Knowledge Repo*. If found, pertinent code in P_r needs to be replaced

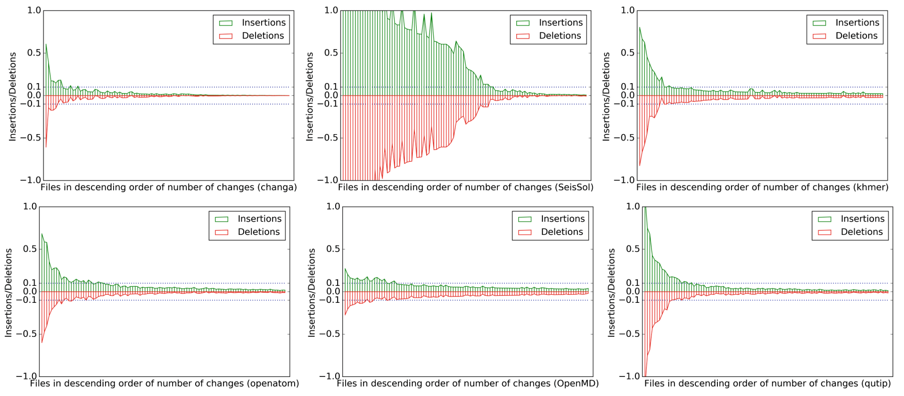

Fig. 3. *Funnels* for a few scientific applications generated by the *Maintainability Visualizer*

with its corresponding representation in P_t using information from $Dict_{temp}$, if needed. If an irreplaceable function or library call (a few iterative solvers in Intel's MKL do not have equivalent definitions in the CUDA standard libraries) is found, appropriate messages need to be provided to the user.

5 Evaluation of Framework

Development on the framework was largely done in Python 2.7 on the Linux platform.

The *Maintainability Visualizer* module was used to analyze a few popular scientific applications maintained on GitHub. The Git repository was cloned locally and provided as input to the module. Details about the applications are given in Table 1. The *Funnel* patterns generated by the module for these applications are shown in Fig. 3.

In case of ChanGA and OpenAtom, the *Funnel* has a relatively broad mouth and a very narrow tapering stem. As implied by the funnel, only a few files in these applications subsumed most changes over the years. Porting applications with such *funnels* to newer platforms should be easier. An important observation made was that these applications have been developed in Charm++, which is a machine independent parallel programming system which allows programs written using this system to run unchanged on different architectures [14]. The generated *funnel* pattern for these two applications very well substantiates this claim and our reasoning about the portability.

QuTiP, developed in Python, has a typical funnel shape with a large mouth slowly tapering into a narrow stem. OpenMD, on the other hand, has an interesting funnel with a very narrow mouth tapering into a stem. Considering its lifetime of 12 years, the funnel implies that the application has been designed with good foresight and an architecture robust to change. Porting this application to a newer platform will also be easy.

Table 1. Application details

Name	Domain	Language	Years of dev.	Last commit
ChaNGa [15]	N-body Gravity solver	Charm++	13	Nov 2015
khmer [16]	Nucleotide Sequence k-mer counting	Python	5	Nov 2015
OpenAtom [17]	Atomic and molecular system simulation based on quantum chemical principles	Charm++	10	Dec 2015
OpenMD [18]	Molecular Dynamics	C++	12	Jan 2016
QuTiP [19]	Simulation of dynamics of closed and open quantum systems	Python	5	Jan 2016
SeisSol [20]	Numerical Simulation of Seismic wave phenomenon and earth quake dynamics	C++	1	Jan 2016

SiesSol is a relatively new application in the set with development started a little more than a year back. However, the extremely large mouth of the funnel and a thick neck made it stand out among the funnels generated. On further study of the source files of the application, it was observed that many of the files were auto-generated. These appear to be code for commonly occurring kernels used in Scientific applications. Porting these kind of application to newer platforms will have a high cost as changes are not confined to a limited set of files.

The framework's applicability was studied with synthetic benchmark applications using a small set of frequently occurring kernels in scientific computing. Porting of these applications between different architecture combinations (no accelerator, single accelerator, multiple accelerators) and different implementations (textbook code, OpenBLAS, Intel's MKL BLAS and Nvidia's CUBLAS) were contrasted.

6 Related Work

The emergence of compute clouds while opening up new frontiers for the HPC community has brought with it many challenges. One main challenge is application portability. For general applications, close to live application porting is already offered by many cloud vendors. For example, for multi-cloud deployment, applications packaged as Docker images [21] can just be dropped on any of the cloud vendors supporting Docker containers regardless of the underlying architecture, operating system or libraries. Other approaches like [22] offer solutions for multiple-cloud deployment which require the adoption of their design process from the application's inception.

In case of scientific applications, this may not be currently possible or desired. For example, when porting a scientific application to a newer platform or when a cloud provider decides to upgrade to a newer architecture, the scientist will want the application to take full advantage of the underlying platform's special characteristics. If the newer platform is not offering any significant advantage over the existing platform, the portability will not be justified and thus, not needed. Also, if the cost of porting exceeds the cost of new development, porting should not be an option.

Source-to-source translators and auto-parallelizing environments play an important role in portability. There has been a lot of research in developing source-to-source compilers for scientific applications in the last few decades. A few of them have met with mixed success [23–26] and have no active development in the recent past. Among the recent ones, some [24,27,28] use standard C or annotated C (OpenMP/ OpenACC/ OpenCL) representation as input to generate an equivalent representation for a single or a very small set of parallel architectures. Others [2,14,29] have introduced a new implicitly parallel input representation paradigm, the use of which is largely confined to a small set of researchers. Also, most of the source-to-source transformers generate generic code that may not be optimized for a particular architecture; device-specific optimizations have not been not considered. For example, particular grid combinations may result in faster CUDA kernel execution on some GPUs and not in others.

[30] most closely resembles the approach presented in this paper. Like a few other approaches, it uses code in which parallel portions have already been identified using OpenMP/OpenACC syntax. Similar to [27], it identifies kernels based on pre-defined patterns. These and other library calls are then mapped to a target platform. But, like many other approaches, targets a single platform.

The main impediment in applying these and other source-to-source compilers to application porting is that they are all uni-directional; i.e. translation of code is from a reference platform to a target platform but not vice versa. For application portability it is important that the translation works both ways. Moreover, considering the increasing adoption of accelerators in the scientific community, represented by increasing heterogeneity of architectural choices available, this becomes more important.

Also, to the best of our knowledge, there is no existing work that qualitatively or quantitatively evaluates the feasibility of scientific application portability or gives an indicator to its cost.

7 Conclusion and Future Work

Porting of scientific applications typically involves a few man years of development and is a large and expensive exercise. In the past, this typically happened once or twice in the lifetime of an application when large grants were procured every few years for a faster and newer system. But the pay-as-you-use philosophy of the compute cloud is expected to dramatically alter this equation with larger and newer systems being made more frequently available by the cloud vendors. The emergence of compute clouds has provided new avenues for researchers; scientists will have an opportunity of experimenting with different accelerator and platform combinations. But for this to become a practical reality in scientific computing, a domain which is characterized by heavily optimized platform-specific kernels and libraries, two tasks need to be automated. First is portability analysis to assess the feasibility of portability and second is source-to-source translation.

This paper presented a framework that addresses these challenges. A preliminary evaluation of the proposed framework on a set of architecture and library combinations have shown encouraging results. Future work will focus on developing a comprehensive and mature *Knowledge Repo* in order to properly evaluate all the modules. A thorough qualitative analysis of the framework's efficacy and its influence on programmer productivity during application portability over a wider range of platform combinations would also be performed.

Acknowledgements. The work on this paper was supported by the HPC Center at the King Abdulaziz University, Jeddah, Saudi Arabia.

References

1. Ortiz Jr., S.: The problem with cloud-computing standardization. IEEE Comput. **7**, 13–16 (2011)
2. Ansel, J.: Autotuning programs with algorithmic choice. Ph.D. thesis, Massachusetts Institute of Technology (2014)
3. Garcia, S., Jeon, D., Louie, C., Taylor, M.B.: The Kremlin oracle for sequential code parallelization. IEEE Micro **32**, 42–53 (2012)
4. Lam, M., Sethi, R., Ullman, J., Aho, A.: Compilers: Principles, Techniques and Tools (2006)
5. Madhavji, N.H., Fernandez-Ramil, J.C., Perry, D.E.: Software Evolution and Feedback: Theory and Practice. John Wiley & Sons Ltd., New York (2006)
6. Lee, S., Min, S.-J., Eigenmann, R.: Openmp to GPGPU: a compiler framework for automatic translation and optimization. ACM Sigplan Not. **44**(4), 101–110 (2009)
7. Tomov, S., Dongarra, J., Baboulin, M.: Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Comput. **36**(5), 232–240 (2010)
8. Vömel, C., Tomov, S., Dongarra, J.: Divide and conquer on hybrid GPU-accelerated multicore systems. SIAM J. Sci. Comput. **34**(2), C70–C82 (2012)
9. Vetter, J.S., Glassbrook, R., Dongarra, J., Schwan, K., Loftis, B., McNally, S., Meredith, J., Rogers, J., Roth, P., Spafford, K., Yalamanchili, S.: Keeneland: bringing heterogeneous GPU computing to the computational science community. Comput. Sci. Eng. (2011)
10. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Programs. Addison-Wesley, Reading (1999)
11. Ahmed, W., Myers, D.: Concept-based partitioning for large multidomain multifunctional embedded systems. ACM Trans. Des. Autom. Electron. Syst. (TODAES) **15**(3), 22 (2010)
12. Braun, F., Lockwood, J., Waldvogel, M.: Protocol wrappers for layered network packet processing in reconfigurable hardware. IEEE Micro **22**, 66–74 (2002)

13. Gharsali, F., Meftali, S., Rousseau, F., Jerraya, A.A.: Automatic generation of embedded memory wrapper for multiprocessor SoC. In: Proceedings of the Design Automation Conference (DAC 2002) (2002)
14. Acun, B., Gupta, A., Jain, N., Langer, A., Menon, H., Mikida, E., Ni, X., Robson, M., Sun, Y., Totoni, E., Wesolowski, L., Kale, L.: Parallel programming with migratable objects: Charm++ in practice. In: SC (2014)
15. Jetley, P., Gioachin, F., Mendes, C., Kale, L.V., Quinn, T.: Massively parallel cosmological simulations with ChaNGa. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–12. IEEE (2008)
16. Crusoe, M.R., Alameldin, H.F., Awad, S., Boucher, E., Caldwell, A., Cartwright, R., Charbonneau, A., Constantinides, B., Edvenson, G., Fay, S., et al.: The khmer software package: enabling efficient nucleotide sequence analysis. F1000Res. **4** (2015)
17. Bohm, E., Bhatele, A., Kale, L.V., Tuckerman, M.E., Kumar, S., Gunnels, J.A., Martyna, G.J.: Fine-grained parallelization of the Car-parrinello ab initio molecular dynamics method on the IBM blue gene/L supercomputer. IBM J. Res. Dev. **52**(1.2), 159–175 (2008). OpenAtom
18. Meineke, M.A., Vardeman, C.F., Lin, T., Fennell, C.J., Gezelter, J.D.: Oopse: an object-oriented parallel simulation engine for molecular dynamics. J. Comput. Chem. **26**(3), 252–271 (2005). OpenMD 1
19. Johansson, J., Nation, P., Nori, F.: Qutip: an open-source python framework for the dynamics of open quantum systems. Comput. Phys. Commun. **183**(8), 1760–1772 (2012). QuTiP
20. Breuer, A., Heinecke, A., Rettenberger, S., Bader, M., Gabriel, A.-A., Pelties, C.: Sustained petascale performance of seismic simulations with SeisSol on SuperMUC. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2014. LNCS, vol. 8488, pp. 1–18. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07518-1_1
21. Docker (2015). https://www.docker.com/. Accessed Jan 2015
22. Ardagna, D., Di Nitto, E., Casale, G., Petcu, D., Mohagheghi, P., Mosser, S., Matthews, P., Gericke, A., Ballagny, C., D'Andria, F., et al.: Modaclouds: a model-driven approach for the design and execution of applications on multiple clouds. In: Proceedings of the 4th International Workshop on Modeling in Software Engineering, pp. 50–56. IEEE Press (2012)
23. Liao, S.-W.: Suif Explorer: An Interactive and Interprocedural Parallelizer. Ph.D. thesis, Stanford (2000)
24. Dave, C., Bae, H., Min, S.-J., Lee, S., Eigenmann, R., Midkiff, S.: Cetus: a source-to-source compiler infrastructure for multicores. IEEE Comput. (2009)
25. Blume, B., Eigenmann, R., Faigin, K., Grout, J., Hoeflinger, J., Padua, D., Petersen, P., Pottenger, B., Rauchwerger, L., Tu, P., Weatherford, S.: Polaris: the next generation in parallelizing compilers (1994)
26. Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W.K., Liao, S.-W., Tseng, C.-W., Hall, M.W., Lam, M.S., Hennessy, J.L.: Suif: an infrastructure for research on parallelizing and optimizing compilers (1994)
27. Nugteren, C., Corporaal, H.: Bones: an automatic skeleton-based C-to-CUDA compiler for GPUs. ACM Trans. Architect. Code Optim. (TACO) **11**(4), 35 (2014)
28. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: Pluto: a practical and fully automatic polyhedral program optimization system. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 2008), Tucson, AZ, June 2008. Citeseer (2008)

29. Ansel, J.: Petabricks: a language and compiler for algorithmic choice. Master's thesis, MIT (2009)
30. Tan, W.J., Tang, W.T., Goh, R.S.M., Turner, S., Wong, W.-F.: A code generation framework for targeting optimized library calls for multiple platforms. IEEE Trans. Parallel Distrib. Syst. **26**(7) (2015)