2018-01-01

Analysis Of High Performance Scientific Programming Workflows

Withana Kankanamalage Umayanganie Klaassen
University of Texas at El Paso, uma.umayanganie@gmail.com

ANALYSIS OF HIGH PERFORMANCE
SCIENTIFIC PROGRAMMING WORKFLOWS

W.K. Umayanganie Klaassen

Doctoral Program in Computational Science

APPROVED:

Shirley V. Moore, Ph.D., Chair

Ann Gates, Ph.D.

Raymond Rumpf, Ph.D.

Rajendra Zope, Ph.D.

Charles H. Ambler, Ph.D.
Dean of the Graduate School

ANALYSIS OF HIGH PERFORMANCE SCIENTIFIC PROGRAMMING WORKFLOWS

by

W.K. Umayanganie Klaassen

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

Doctoral Program in Computational Science

THE UNIVERSITY OF TEXAS AT EL PASO

May 2018

Table of Contents

Acknowledgements

This work was partially funded by the Department of Energy Office of Advanced Scientific Computing Research under grant no. DE-SC0006733 and by the Air Force Office of Scientific Research under Award No. FA9550-12-1-0476. We would like to acknowledge use of the Stampede supercomputer at Texas Advanced Computing Center, which is funded by the National Science Foundation. We acknowledge use of the Titan supercomputer at Oak Ridge Leadership Class Facility, which is funded by the Department of Energy. We acknowledge use of the Experimental Computing Laboratory (ExCL) maintained by the Future Technologies Group at Oak Ridge National Laboratory. We would also like to thank the members of the Fall 2015 CPS 5401 and Spring 2016 CPS 5310 courses and the lab instructor for those courses, Dr. Natasha Sharma, for helping with the case studies.

Abstract

Substantial time is spent on building, optimizing and maintaining large-scale software that is run on supercomputers. However, little has been done to utilize overall resources efficiently when it comes to including expensive human resources. The community is beginning to acknowledge that optimizing the hardware performance such as speed and memory bottlenecks contributes less to the overall productivity than does the development lifecycle of high-performance scientific applications. Researchers are beginning to look at overall scientific workflows for high performance computing. Scientific programming productivity is measured by time and effort required to develop, configure, and maintain a simulation experiment and its constituent parts, together with the time to get to the solution when the programs are executed. There is no systematic framework by means of which scientific programming productivity of the available tools can be evaluated. We propose an evaluation approach that compares programming workflows to identify productivity bottlenecks and suboptimal paths as well as productivity gains. Based on a set of predefined criteria we can evaluate both short-term and long-term productivity criteria. We use these results to suggest improvements to the programming environment or tools. This thesis includes three studies of scientific programming workflows: 1) We apply our evaluation approach to two case studies involving the use of numerical libraries. 2) We evaluate GPU programming models using software engineering complexity metrics. 3) We evaluate use of a high level directive based programming model and a source-to-source compiler with respect to productivity of programming FPGAs using a computational chemistry code. We compare the programmability and performance of the FPGA port with the GPU port of the same code.

Chapter 1: Introduction

A growing number of high performance computing systems include accelerators such as GPUs, and scientific applications must be programmed to take advantage of such hardware in order to make full use of the system. Future supercomputers must serve diverse workloads with efficient power usage within a reasonable budget[4]. Scientific applications need to adapt to new generations of supercomputers and make use of new architectures in order to achieve better performance and scalability. These inevitable changes in the programming environment have to be incorporated by taking programming workflow productivity into account.

Heterogeneous computing with device accelerators such as Field-Programmable Gate Arrays (FPGAs) is a potential area of interest for next generation supercomputers for solving exascale problems. The amount of additional effort invested into integrating device programming into existing workflows, for accelerating compute intensive kernels, is significant. When a new optimization is introduced into an existing software system, a productive option is a solution that improves performance with minimal additional cost of development and maintenance of the software. Although similar problems have been addressed by the software engineering community, as scientific simulations reach larger scale, data and task decomposition become increasingly complex and the computing environment itself becomes a variable, making it harder to adapt software engineering concepts into the domain constraints in scientific computing. To achieve the most efficiency in this process, software engineering principals and paradigms such as Object Oriented Programming (OOP) concepts may be applied selectively.

Thomas et.al. defines 'scientific computing productivity' as a quality measure of the process of achieving scientific results on high performance computing systems[22]. We view the productivity problem as a mathematical optimization problem with constraints and objective functions. The exact constraints and objective functions depend on the

relative costs of resources and on the user's goals and computing environment. We propose an evaluation framework for scientific programming productivity by selecting variables that represent time and effort required to develop, configure, and maintain a simulation experiment and its constituent parts, together with the time to get to the solution when the programs are executed, to help predict scientific programming productivity of possible programming workflows. We evaluate both short-term and long-term productivity. In chapter 2, we discuss related work on software engineering and metrics for high performance computing (HPC).

In chapter 3, we evaluate the problem solving process in a scientific programming workflow involving use of numerical libraries by two user categories, expert and novice, who attempt to solve a simulation problem utilizing numerical linear algebra. After observing the performance of both an expert user and the target users in carrying out the workflow, we analyze the results to determine bottlenecks and suboptimal paths in the target user workflows. By comparing development effort and resulting code performance for tool-assisted vs. expert workflows, we evaluate the impact of the tool on productivity.

Heterogeneous computing is a potential solution to serving diverse application workloads. Device programming for scientific applications tends to be a difficult and time-consuming task and until recently, has been done by using low-level programming models such as CUDA and OpenCL. Programmability and portability are critical for adapting new programming models for device programming. OpenACC is a directive based-programming model that has gained popularity due to its higher level of abstraction. In chapter 4, we compare code complexity and performance of CUDA, OpenCL, and OpenACC implementations for three benchmark codes. We find that OpenACC implementations have significantly fewer lines of code than the corresponding CUDA and OpenCL implementations but that the measured cyclomatic complexity is not always lower. When the OpenACC code is optimized, the performance does not show a drastic difference from CUDA and OpenCL implementations, although CUDA and OpenCL generally have slightly better performance.

In chapter 5, we evaluate the process of porting compute intensive parts of a computa-

tional chemistry application to FPGA devices and compare this process with that for GPU devices. FPGA programming has been primarily limited to Hardware Description Language (HDL) and FPGA programs were designed at Register Transfer level (RTL). Until recent introduction of OpenCL programming for FPGA by vendors, available HDL tools did not provide sufficient abstraction or portability for scientific programmers, limiting the use of FPGAs to hardware experts. The workflow we evaluate involves use of the OpenARC research compiler to translate OpenACC directives into OpenCL optimized for FPGAs. We use this workflow to port the GAMESS-SIMGMS [8] computational chemistry kernel containing the Hartree-Fock (HF) procedure to four different architectures, Intel Xeon® E5520 CPUs, Altera Stratix® V FPGAs, Intel Arria® 10 FPGAs and Nvidia® P100 GPUs. To our knowledge, this is the first use of a higher level FPGA programming model with a computational chemistry code. We evaluate the productivity of the programming workflows for the different architectures and compare the performance of the resulting executable codes.

Our research questions are the following:

- How can we evaluate scientific programming workflows to identify productivity bottlenecks and suboptimal paths, as well as to identify opportunities for productivity gains?
- How can we evaluate both short-term and long-term scientific programming productivity?
- What software metrics are suitable for evaluating programming models for GPU architectures?
- Are there possible optimizations to be gained from the recent advancements in FPGA technology for scientific programming workflows, and is the FPGA porting process productive compared to the porting process for a GPU architecture?

This thesis makes the following contributions:

- We propose an evaluation approach that compares programming workflows to clearly identify productivity bottlenecks and suboptimal paths as well as productivity gains. Based on a set of predefined criteria, we evaluate both short-term and long-term productivity criteria.

- We test existing software engineering complexity metrics for evaluating the productivity of GPU programming models using software engineering complexity metrics.

- We evaluate the productivity of using a high-level directive-based programming model to port a computational chemistry kernel to FPGAs.

- We compare the programmability and performance of the FPGA port with that of the GPU port for the same computational chemistry kernel. We achieve substantial speedups on both architectures.

Chapter 2: Related Research

2.1 Software Engineering Approach to Scientific Programming Productivity

According to[7], scientific computing has a growing problem with end-to-end productivity. Observed development problems are the following:

- Increasingly long and expensive development
- Higher risk of failure
- Growing maintenance costs
- Increasing difficulty of porting codes to next generation machines

The authors maintain that scientific computing productivity currently depends on multidisciplinary experts optimizing parallel code by hand. The main finding was that there exists an expertise gap between computing experts and domain scientists, as exemplified by the following aspects:

- There are vanishingly few individuals with the needed skills in scientific domain, programming languages, and hardware.
- Training takes years.
- Once skills are acquired, they are often not portable. According to[7], the main human resource intensive tasks are
- Porting and modifying existing parallel code
- Developing correct scientific programs
- Serial optimization and tuning

- Code parallelization and scaling

Our current research focuses on analyzing these tasks with respect to productivity.

The authors maintain that the key to improving scientific programming productivity is to reduce dependence on multidisciplinary experts and increase abstraction and automation by

- Providing computational abstractions reflecting the science and math of the problem domain
- Providing hardware-independent abstractions for tuning and parallelization
- Automating mapping of abstractions to hardware

One way to achieve abstraction and automation is through use of scientific libraries. Another way is to use higher level more abstract programming models or tools. Our work focuses on evaluating the productivity of scientific computing workflows that involve the use of parallel scientific libraries or abstract programming models.

2.2 Data Collection and Analysis Methods

Lorin et.al. present the idea of combining self-reported and automated data to improve programming effort measurement in [10]. Automated data captures accurate 'typed' data processing time while observation or self-reported data captures time spent on researcher insight. Incorrect measurement can introduce unexpected bias and lead to incorrect conclusions. They have observed from a set of pilot studies that the self-reported data directly from the users were inconsistent compared to the automated data collected using tools. This has motivated them to have a passive observer reporting the programmer time in order to obtain an accurate measure. They also changed their web-based interface to a paper-based activity log in which individual users entered start and stop times, hoping to eliminate the inconsistencies introduced, such as irritating the user when they had to

enter every compilation step even when they had insignificant syntax errors. Their second study concludes that more precise logs improve accuracy. They come to the conclusion that automatically collected data can be augmented by self-reported data, but the data provided by users can vary between users. Indirect methods that are easier to obtain, can be used if there are criteria that correlate well with effort such as lines of code.

Personal Software Process (PSP) [11] is a framework that guides software engineers for achieving better productivity for the software process, such as writing requirements, running tests, defining processes, and repairing defects. The authors provide a course and a textbook concentrating on individual user education showing them how to plan and track their work. The PSP aims to show engineers how to manage quality from the beginning of the job, how to analyze the results of each job, and how to use the results to improve the process for the next project and track performance against these predefined goals. The framework provides a planning script that guides the user's work. They record their time and defect data which they self-summarize from the logs, measure the program size, and enter these data in the plan summary form which is delivered with the finished product. Their process goal is to improve quality of work by producing zero-defect code within a planned schedule and costs, to try to get rid of traditional test-and-fix strategy which is time-consuming and costly.

Min Zhang and Lorin Hochstein [27] introduced a method to fit a workflow model into captured data called a software engineering workflow analysis (SEWA). They analyze automatic data captured by the programming environment automatically by a tracing method and build a model that is dependent on eight factors which are coding, chunking, commenting, comparing, converting, computing, connecting and constraining. They summarize low level tasks into events under these categories. Using the above categories, they implement a model for programmer activity. They have developed an open source tool called ActivityGraph to visualize time series data and compute the programmer effort distributed among activities. This tool is introduced for general software development and they apply the same tool to the high performance computing (HPC) domain using observed data from

two previous case studies conducted by the University of Maryland, which used Hackstat and UMDInst to log the steps of the small HPC problems. They use the compile log files, editor log files and shell log files to grab the information. They categorize the data retrieved via these log files into events of the previous categories and standardize and write it into spreadsheets. Using a graphical tool for visualizing programmer's changes in a chunk, they develop heuristics based on examination of the diffs. Most of the preprocessing has been done manually, by observation. They categorize debugging and testing into one category claiming it is difficult to distinguish the different purposes of execution (similar to the problems we encountered when we drew the workflow diagrams for program execution). They iterate through the SEWA process and refine the model with steps such as identifying and labeling chunks with new activities according to their heuristics. Finally, they compare the SEWA model with the observed times to prove that their model is consistent. They state that activities such as 'thinking' can be only indicated in the observation data since there is no way to identify anything that does not involve typing on a keyboard using the tool.

A pragmatic methodology for the design and evaluation of scientific workflows in research-oriented web applications is presented in [6]. The authors carry out an in-depth usability study of their CoGe web application that provides a set of tools for exploring genomic datasets. Their method demonstrates how to identify bottlenecks in multi-step tasks and how to analyze bottlenecks. The visualization system introduced in their paper is used to analyze complex tasks associated with scientific workflows. Their analysis leads to suggestions for improvements in the current implementation of their CoGe web application. They have carried out a follow-up study and confirmed that their suggestions improved the user navigation in the web application. We have adopted a graphical representation of the workflow steps and compare novice users' paths with the expert user path. However, in contrast to[6], we assign metrics to the paths rather than only determining whether the novice user path deviates from the expert user path. Whereas their framework focuses mostly on execution performance, our evaluation framework focuses mostly on human factors and on code portability and long-term maintainability.

Studies of scientific programmer productivity are reported in [20],[28], [9]. These studies all have one metric to measure programmer productivity, namely the amount of time spent carrying out tasks. This metric focuses on short term productivity. In contrast, our evaluation framework can support additional evaluation criteria, such as portability and maintainability of the code, which affect productivity in the long term.

2.3 Software Metrics

As we stated in chapter 1, domain scientists face several challenges utilizing accelerator programming models when porting scientific codes to next generation machines.

The software engineering community has developed metrics that are widely used to assess design and maintenance costs and risks for large software development projects. A number of metrics for measuring software complexity have been proposed. One obvious metric is lines of code (LOC), also called source lines of code (SLOC), since this metric is easy to collect and has some relationship to costs of developing and maintaining the code [22]. However, SLOC is thought to account for only 30-35% of the costs of developing and maintaining a software project [26]. Cyclomatic complexity, first proposed in [7], is a graph theoretic complexity measure that depends only on the decision structure of the program and not at all on program size. The rationale for this metric is that it provides a quantitative basis for testability and maintainability. The cyclomatic complexity is the number of basis paths (maximum number of linearly independent paths) through a program, which is the number of decision statements (also called predicates) plus one. The authors in [10] showed that the correlation between the cyclomatic complexity and the number of errors in a program was about 0.90. Cyclomatic complexity of 10 is considered to be an acceptable limit. In particular, a large jump in error count at complexity 11 is shown in [11]. In [27], the authors extend the mathematical basis of cyclomatic complexity into the architectural design of a software system. Architectural design consists of the hierarchical structure of functions and their control interrelationships. The authors in [27]

define module design complexity to be the cyclomatic complexity of the reduced control graph of the module, where reduction is applied to eliminate any complexity that does not influence the interrelationship between design modules. The design complexity is then defined as the sum of the module complexities of the main module and its descendants. Properties of design complexity are derived that can be used to compute design complexity of integrated systems that have a nonempty intersection of modules. The authors also define integration complexity, which is relevant to integration testing, to be a basis set of subtrees of the program graph. The design and integration complexity metrics can be used to produce a testing strategy for the program. The metrics calculation and formulation of the testing strategy can be done by working from the program structure chart and pseudocode before the actual program has been written.

In [3], the authors compare SLOC and runtimes on an AMD Accelerated Processing Unit (APU) for OpenCL, C++AMP, and OpenACC implementations of four benchmarks – the LULESH, CoMD and XSBENCH proxy applications and the miniFE finite element benchmark. C++ AMP is a combination of a library and extensions to the C++11 standard to support GPU programming. Like OpenACC, C++ AMP relies on compilers to generate low-level code. The APU was an AMD A10-7850K APU operating at 3.7 GHZ, with DDR3 memory bandwidth of 33GB/s and 12 compute units (4 CPU and 8 GPU). OpenACC required minimal changes to the original serial codes, C++ AMP required an average of 15% more code, and OpenCL required an average of four times more code. The performance findings were that OpenCL was significantly better (2-5X) for the computationally intensive LULESH and CoMD codes, and that C++ AMP was either on par with or outperformed OpenACC.

2.4 FPGA Implementation of a Computationally Critical Exponential Calculation

Wielgosz et.al. [25] presents a Field Programmable Gate Array (FPGA) implementation of a fully pipelined and optimized implementation using HDL for the computationally critical exponential calculation of Gaussian Type Orbital (GTO), which is a part of the quantum chemistry computational system. The Kohn-Sham algorithm, which is a Self-Consistent Field (SCF) type procedure, allows the adoption of gradually adjustable data precision, and its iterative execution gives a speed boost to FPGAs in the final application. The authors achieve 2.5X to 20X speed-ups for the finite sum of the exponential products calculation in comparison to a general-purpose Central Processing Unit (CPU). The authorsÄô intention was also to make hardware implementation of calculating the orbital function universal and easily attachable to different quantum-chemistry software packages. The authors conclude that even with data stored in the processor's memory with the full optimizations, the FPGA is much faster than a processor. The authors state that there is a huge potential in hardware implementation of quantum chemistry computational routines. They present the hardware implementation part of the DFT algorithm in [24], on a Reconfigurable Application-Specific Computing (RASC) blade containing two computational FPGAs (Xilinx Virtx-4 LX200s) and two TIO ASICs. Their ultimate goal is to design a hardware unit to generate the exchange-correlation potential matrix, one of the most computationally intensive routines within the Kohn-Sham (KS) formulation of DFT. They design an orbital hardware module, including two FPGAs available on the RASC platform to communicate across both FPGAs through an external hub, which adds some overhead. The orbital calculation module achieves 3X acceleration, and the S matrix generation module works up to 16X faster than an Itanium 2 processer. The authors note that the capacity of Xilinx Virtex-4 LX200 internal memory is insufficient to store all the data and BRAM and FIFO memories share a single 128 bit-width bus. A custom mechanism for request tracking is introduced to handle data dispatching among internal memories. The single

128-bit RASC interface is considered to be the bottleneck of the RASC system and it can be overcome by employing a platform with more data transfer links.

Chapter 3: Analysis of Scientific Programming Workflows that use Numerical Libraries

3.1 Motivation

Other researchers have hypothesized that the higher level of abstraction offered by numerical libraries can help improve the productivity of scientific programming workflows. However, successful use of numerical libraries requires that users can select the correct numerical routines and use them properly. The evaluations of numerical libraries in the literature are usually done either by the library writers themselves or by expert users. Our goal was to evaluate the programming productivity of use of such libraries by non-expert users, since such users are representative of most domain scientists. We define workflow as the sequence of steps a user carries out to accomplish a programming task. Distinguishing characteristics of each scientific workflow step is required to analyze the scientific programming activities. We analyze the observations taken from a classroom study in this chapter. We clearly articulate the steps of a given workflow and the criteria by which productivity will be evaluated.

We discuss the observations that we make, and measure the performance of both an expert user and the target users in carrying out the workflow. We analyze the results to determine bottlenecks and suboptimal paths in the target user workflows. We present a graphical representation of workflows that enables comparison and propose a generally applicable methodology for evaluating scientific programming productivity according to specified criteria. We also show how our methodology leads to suggestions for improving programming productivity.

3.2 Evaluation Framework

In order to demonstrate a simple scientific programming workflow in the HPC domain, we clearly articulate the steps for the scientific problem solving approach. Next we define the criteria by which productivity will be evaluated. Then we observe and measure programming behavior of both expert and novice users. We analyze the results to determine bottlenecks and detect wrong or sub-optimal paths in the novice user workflows. Finally we use the results to suggest improvements to the programming environment. We set up experiments that involve novice users and an expert user who is used for defining a lower bound for the measurement criteria. We observe and measure programmer behavior for each problem. The users are given the same set of problems under the same conditions. We provide the same set of steps to follow and mark the time spent on every step for each problem. Users are asked to take note of every correct and incorrect step they follow in order to get to their final solution. These notations are used to articulate the workflow diagrams for each user. For a given case study, productivity is measured by a set of criteria related to the particular scenario. The evaluation criteria are predefined and all of a user's steps are analyzed to figure out how each step contributes to each criterion. Steps followed by a user may negatively or positively affect some of the measurement criteria. After carefully examining all the steps users have taken, a weighted or a numerical value is assigned for each criterion. We define the productivity vector by adding together all the values that come under each criterion. It is possible for one step to affect more than one criterion of the productivity vector. After completing the productivity vector, we compare the vectors for the novice users with those of the expert user to identify efficiency bottlenecks and wrong steps. Steps that diverge from the expert user's path may not necessarily be wrong steps if they do not result in a lower score for a given criterion, rather they could be alternative valid paths to the same goal. Thus, it is important to analyze not only the paths through the workflows, but also the metric values associated with each path to detect wrong or sub-optimal paths in the user workflows. Our productivity measurement vector is specific

to each problem or case study, but the framework can be generalized by defining different criteria for the vector for other workflow evaluation problems.

3.3 Domain Selection for the Classroom Study

We chose the domain of numerical linear algebra to develop and test our evaluation framework for scientific programming productivity. Scientific problem solving usually requires a background in numerical analysis, high performance computing (HPC) and software engineering. It also typically involves reading documentation (when available) or researching publications outside of the developer's area of expertise. Although there have been numerous advancements in numerical analysis and HPC libraries, selecting the correct library routines for the specific need of the user is a significantly hard task for the domain scientist. The probability of the user identifying the most portable and efficient library routine for a given problem is decreasing with the larger number of new versions and libraries added. Linear algebra libraries are among the most used HPC libraries by the domain scientist and these are often the most time-consuming part of scientific applications. Reducing the time for the calculations as well as using the correct version for best portability of the code is one of the most important skills of a HPC user. Since it is too complex and less efficient to use their own algorithms when solving linear algebra problems, domain scientists must know how to use the available libraries out of the vast number available that have been developed and optimized by the experts for decades. Scientists and engineers rely on linear algebra algorithms for solving problems in high-performance computing applications. Most domain scientists lack the in-depth knowledge needed for discovering and applying the most suited libraries that give the most efficient solution to a given problem. One incorrect step can have a snowballing effect on the next steps that could lead to inefficiency.

The general workflow of solving a problem using a numerical library consists of the following steps:

1. Prepare input files

2. Select appropriate method based on the problem characteristics

3. Find appropriate library routines

4. Construct the program

5. Compile, execute and debug the program

6. Validate the results

A key goal of numerical libraries is to achieve performance portability. Performance portability is defined in [5] as the amount of user code that can be compiled for diverse architectures and obtain the same, or nearly the same, performance as an architecture specialized version of that code. Performance portability is achieved by libraries through abstraction. The routines and their functionality remain the same across platforms, and the platform-specific optimizations and parallelization are hidden inside the implementation.

We defined and evaluated programming workflows for two numerical linear algebra problem types. The first case study is solution of a dense linear system using the linear algebra library, LAPACK, and working from an existing code to port it to the Stampede supercomputer. The second case study is the implementation of solving a sparse linear system from a finite element analysis using the PETSc library. We chose to carry out our experiments and evaluation using the Stampede1 supercomputer at the Texas Advanced Computing Center (TACC). It was comprised of 6400 nodes, 102400 processor cores, 205 TB total memory, 14 PB total and 1.6 PB local storage. The cluster contained 160 racks of primary compute nodes, each with dual Xeon E5-2680 8-core processors, Xeon Phi coprocessor, and 32 GB RAM. It also contained 16 large-memory nodes with 32 cores and 1 TB RAM each, and 128 compute nodes with Nvidia Kepler K20 GPUs, and other nodes for I/O (to a Lustre filesystem), login, and cluster management. Stampede1 provided a peak performance of nearly 10 petaflops (PF), or nearly 10 quadrillion math operations per second.

3.4 User Categorization

We assume that all our users have the basic understanding required for solving domain specific scientific problems. The expert HPC users understand the architecture of high performance computers and how the architecture of high performance computers affects the speed of programs run on the machine. They also have knowledge of how the memory access affects the speed of HPC programs, Amdahl's law for parallel and serial computing, the importance of communication overhead in high performance computing, some of the general types of parallel computers, how different types of problems are best suited for different types of parallel computers, and aspects of message passing on MIMD machines. Other than the above stated ideas, an expert user who deals with large-scale matrix calculations on HPC machines has to have experience with compiled language programming. Usually most scientific code requires being familiar with languages like Python, Fortran or C in addition to a certain understanding of linear algebra and other mathematical libraries. We categorize the users who are lacking parts of the above knowledge as novice users Novice users represent future domain scientists.

3.5 Case Study 1: Porting a Dense Linear Algebra Code to Stampede

3.5.1 Legacy Code Example

We chose an existing C++ code that initializes a small linear system of three equations and three unknowns and calls two LAPACK routines to factor the matrix and then solve the system. The author of the code provided instructions on how to compile and link it using the GNU C++ compiler and the Netlib version of LAPACK. The code is written in an older style and does not use portable LAPACK data types nor does it use the recommended C++ interface. We chose this example because it is a small example of the more general problem of porting a legacy code to a new computer system. We ob-

tained the legacy code that uses LAPACK routines to solve a dense linear system available online from the link `https://dynamithead.wordpress.com/2012/06/30/introduction-to-how-to-call-lapack-from-a-cc-program-example-solving-a-system-of-linear-equations/`. The author has provided custom header files for LAPACK and given hints about routines he uses for solving the dense linear equation.

The example code uses LAPACK to solve the linear system. LAPACK is one of the most widely used standard software libraries in scientific computing. Domain scientists from many disciplines rely heavily on linear algebra algorithms. The LAPACK Fortran 90 codebase provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, singular value problems and matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) [14]. LAPACK can handle simple and complex dense and banded matrices, but not general sparse matrices. LAPACK effectively makes use of cache-based architectures. For C and C++ developers, a portable extended version of LAPACK called LAPACKE is provided [15]. LAPACKE uses native C data representation and allows the user to specify whether matrices are stored in row major or column major order. Row major order is usual for C/C++ codes, and column major order is used in Fortran codes. Thus, for maximum portability and efficiency, a C/C++ developer should use the LAPACKE extension if it is available.

The LAPACK interface has become a de facto standard for numerical dense linear algebra. Documentation and a reference implementation are provided in the Netlib software repository at `http://www.netlib.org/lapack/`. Vendors have adopted the interface and implemented versions of the LAPACK routines tuned for their platforms and compilers. For example, on Intel platforms such as Stampede, LAPACK is part of the Intel Math Kernel Library (MKL). The Stampede User Guide has instructions for how to link Fortran and C codes with MKL and refers the user to Intel documentation for MKL. The problem for many users is that the vendor libraries include LAPACK routines, but the library is not called LAPACK, and the link line for the Netlib version of LAPACK will not work for the vendor version. To achieve the most efficient code, developers should link their

code with the vendor library. If they link with the refercnce Netlib version of the library using –llapack, their code will still work but it will be inefficient, sometimes by an order of magnitude or more.

For our test case, the instructions provided by the code author are to compile and link the code using

```
g++ -llpack
```

To use the MKL library on Stampede, the user should compile and link using

```
icpc --mkl=sequential --I$TACC_MKL_DIR/include
```

where `icpc` is the name for the Intel C++ compiler and `TACC_MKL_DIR` is an environment variable that is defined by default because the Intel compiler suite and MKL library are loaded by default when the user logs into Stampede.

3.5.2 Experiment Setup

We conducted the experiment with ten users and evaluated the programming workflows against that of the expert user. Users were provided with general guidelines to follow and instructed to enter their start and end times for each step, including both correct and incorrect (or unfruitful) steps. These records were used later to create the workflow graphs for evaluation.

The expert user was the instructor of the CPS 5401 Introduction to Computational Science class at the University of Texas at El Paso in fall semester of 2015. The novice users were students in the class. The students had received instruction in the functionality and use of numerical libraries, including LAPACK, but had not been instructed in this particular workflow on Stampede. The students were all also co-enrolled in or had previously taken MATH 5329 Numerical Analysis in which course they had learned about numerical dense linear algebra methods. We observed the novice users while they worked on the problem. They were instructed to ask for help if they got stuck for a long time. When they asked

for help, we checked that they had recorded their progress so far and gave a hint to enable them to move forward.

The ideal workflow for porting the code to Stampede consists of the following tasks:

1. Transfer the program file to Stampede
2. Lookup the correct commands in the Stampede User Guide for compiling the file
3. Insert missing include directives
4. Change the LAPACKE include directive to include mkl_lapacke.h
5. Change the data types for routine arguments to be portable LAPACK types
6. Call `LAPACKE_dgetrf` correctly to factor the matrix
7. Call `LAPACKE_dgetrs` correctly to solve the triangular system
8. Compile the program and fix any errors
9. Execute the program and validate the result

   To evaluate the productivity of the workflow, we assigned metrics with respect to the following criteria:

   (a) Time to complete the workflow

   (b) Correctness of the program

   (c) Portability of the program

   (d) Maintainability of the program

After analyzing the results of the above experiment, we created an LAPACK Porting Guide to try to improve the available user documentation.

3.5.3 Results

The expert workflow for case study 1 is illustrated in Figure 3.1, and the corresponding steps with timings are given in Table 3.1. Workflows for three of the novice users are illustrated in figures 3.2 through 3.4. The corresponding steps and user reported timings are given in tables 3.2 through 3.4 respectively. Arrows indicate progression from one activity to another. Backtrack arrows indicate that the user performed tasks that took time that did not make actual progression since they go back to do the same task again.

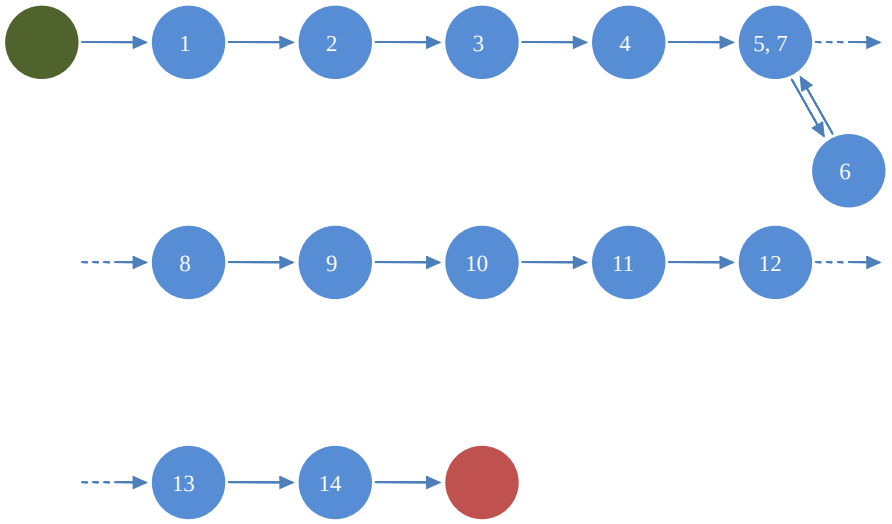

Figure 3.1: Workflow of expert user for case study 1

Table 3.1: Workflow of the Expert User for Case study 1

Step	Task	Time (minutes : seconds)
	Start	
1	Click on the link	1:00
2	Copy and paste on the editor	1:00
3	Save file	1:00
4	Lookup compile command in STAMPEDE User Guide	1:00
5	Compile and check for errors	0:30
6	Insert directive `'include <iostream>'`	0:00
7	Recompile and look for errors	2:00
8	Replace lapacke.h with mkl_lapacke.h	2:00
9	Lookup `LAPACKE_dgetrf` file reference	2:00
10	Change data types in the call sequence	5:00
11	Lookup `LAPACKE_dgetrfs` file reference	1:00
12	change call sequences	4:00
13	recompile	0:30
14	Run	0:30
	End	
	Total Time	21:00

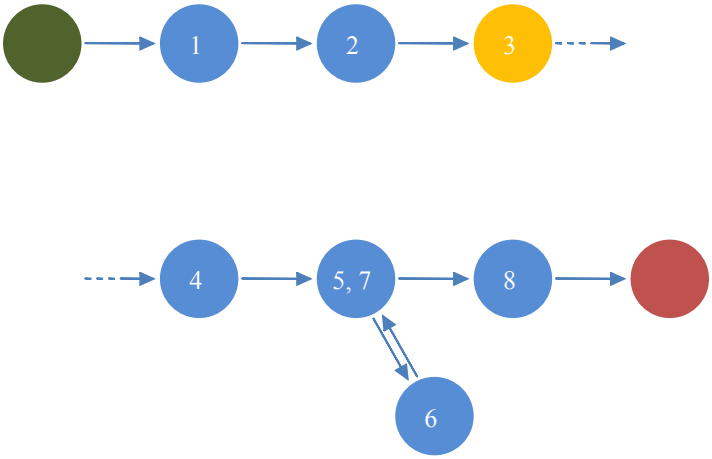

Figure 3.2: Workflow of novice user 4 for case study 1

Table 3.2: Workflow of the Novice User 4 for Case study 1

Step	Task	Time (minutes : seconds)
	Start	
1	Click on the link	1:00
2	Copy and paste on the editor	7:00
3	Download code author provided header files	4:00
4	Lookup compiler command in STAMPEDE User Guide	3:00
5	Compile with g++ and check for errors	0:30
6	Insert directive `include <iostream>`	8:00
7	Recompile and look for the error	1:00
8	Run	1:00
	End	
	Total Time	25:00

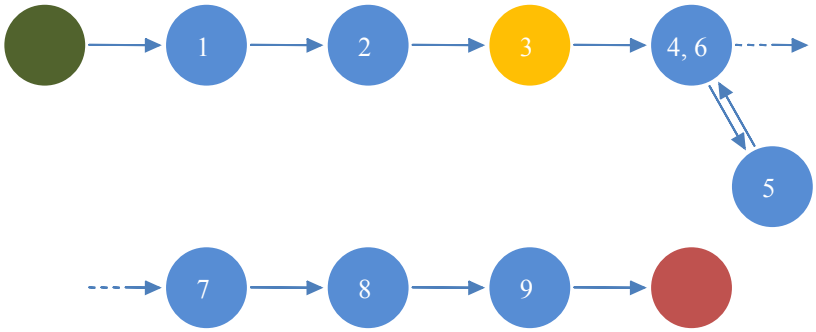

Figure 3.3: Workflow of novice user 3 for case study 1

Table 3.3: Workflow of the Novice User 3 for Case study 1

Step	Task	Time (minutes : seconds)
	Start	
1	Click on the link	0:30
2	Copy and paste on the editor	0:30
3	Upload author provided header files	0:30
4	compile with g++ and check for errors	1:00
5	Insert directive `include <iostream>`	0:30
6	Compile and check error	1:00
7	fix the error	4:00
8	Recompile	0:30
9	Run	0:30
	End	
	Total Time	9:00

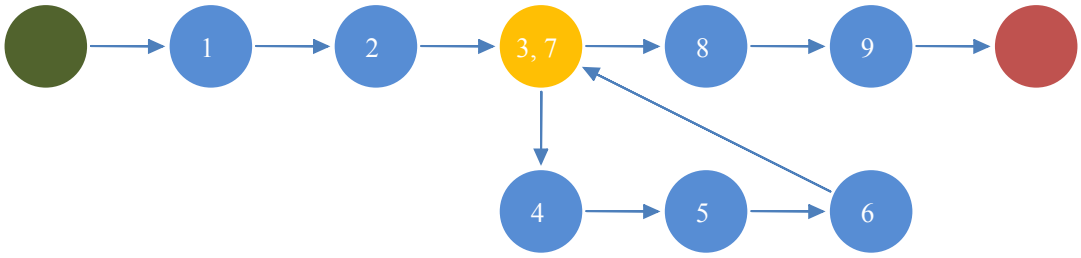

Figure 3.4: Workflow of novice user 1 for case study 1

Table 3.4: Workflow of the Novice User 1 for Case study 1

Step	Task	Time (minutes:seconds)
	Start	
1	Click on the link	0:30
2	Copy and paste on the editor	3:30
3	Compile with g++	0:30
4	Look for a different compiler	0:30
5	Insert directive `include <iostream>`	8:00
6	Download the LAPACKE header files	2:00
7	Recompile and look for the error	1:00
8	Use a different header file	1:00
9	Recompile	0:30
10	Run	0:30
	End	
	Total Time	18:00

3.5.4 Analysis

We compared the workflow diagrams of the novice users with that of the expert user and looked for the branches where the novice users deviate. These different paths were assigned evaluation metrics for each criterion in order to obtain a productivity measurement scheme.

Development Time

Development time can be negatively affected by the following deviations of the steps by spending more time than necessary. For this particular problem of porting legacy code for the use of LAPACK library on STAMPEDE machine efficiency is directly associated with time and it is negatively affected by spending more time by users spending time going on wrong paths.

Development time was affected by the following wrong steps:

1. Using g++ instead of the intel C++ compiler
2. Oblivious to the fact that LAPACK is already installed in Stampede

Correctness

If the user gets a wrong result or partially wrong result, accuracy gets a negative value with a weight. Following steps could lead users to get wrong results.

1. Positive: Correct answer
2. Negative: Being oblivious to row vs. column major ordering
3. Negative: Using Netlib reference version header files rather than MKL header files

These negative steps did not cause an incorrect answer with this particular code. Being oblivious to row vs. column major ordering did not affect the solution in this case since the matrix was symmetric, but for a nonsymmetric matrix, the result could have been incorrect. Using header and library files from different implementations of LAPACK did

not affect the correctness in this case but doing this is not a best practice and could lead to incorrect results. Since the negative steps did not affect correctness for this case study, we do not include them in the metric value.

Portability

Portability can be negatively affected by using a deprecated interface and using custom header files. Usage of old interfaces could be incompatible with extensions of the library. These are the steps that affected portability in this particular problem:

1. Negative: Not using portable LAPACK data types

   For example, the type lapack_int is used for integer arguments to LAPACK routines. This portable type may be implemented differently on different platforms but still retains the same semantics.

2. Negative: Using the deprecated C interface rather than the standard LAPACKE interface

Maintainability

Long-term maintainability of the code is affected by the following steps:

1. Negative: Using the custom header files instead of MKL header files

   The custom header files would need to be maintained along with the program and possibly updated.

Productivity Vector

Presented vector of measurement for evaluation takes development time of obtaining the end-to-end results, code portability, and correctness of the obtained results and maintainability of the code into consideration.

[*Development-time Correctness Portability Maintainability*]

1. Evaluating the workflow of User 4 against expert user

   Development-time – Total time was 25 minutes against the expert user time of 20 which gives us -5 in the vector

   Correctness – 0 since the result was correct

   Portability – Use of custom files and not using LAPACKE data types -2Wm

   Maintainability – User has used a custom header file. -1Wp

   $$[-5 \quad +1Wc \quad -2Wp \quad -1Wp]$$

2. Evaluating the workflow of User 3 against expert user

   Development-time – Total time was 9 minutes against the expert user time of 20 minutes which gives us +11 in the vector.

   Correctness – 0 since the result was correct

   Portability – Use of custom files -1Wp

   Maintainability – User has used a custom header file. -1Wm

   $$[+11 \quad +1Wc \quad -1Wp \quad -1Wm]$$

3. Evaluating the workflow of User 1 against expert user

   Development-time – Total time was 18 minutes against the expert user time of 20 which gives us +2 in the vector.

   Accuracy – 0 since the result was correct

   Portability – Use of custom files -2Wp

   Maintainability – User has used a custom header file. -1Wm

   $$[+2 \quad 0 \quad -2Wp \quad -1Wm]$$

Depending on the weights assigned for the correctness, portability, and maintainability metrics, the most productive workflow might be that of the expert user or of the novice user who completed the task in the least amount of time. In reality, the times self-reported by the novice users are not accurate, since they did not include time asking for help or looking for answers. Novice users used a 'quick and dirty' approach whereas expert user took time to rewrite the code to achieve long term maintainability and portability.

3.6 Case study 2: Solving a Sparse Linear System from a Finite Element Analysis

3.6.1 Problem Definition

Finite element analysis is a method used to solve systems of partial differential equations (PDEs) that model physical problems in application areas such as computational fluid dynamics and structural mechanics. When the PDEs are discretized, the result is a sparse linear system, often very large, that needs to be solved. Both direct and iterative methods have been developed to solve such systems. Large systems may need to be solved with iterative methods and/or in parallel because of memory or computational bottlenecks. PETSc stands for The Portable, Extensible Toolkit for Scientific Computation and the library is developed at Argonne National Laboratory together with many collaborators. The PETSc library has been developed for the purpose of assisting computational scientists in solving PDE problems [2]. PETSc includes a collection of both direct and iterative solvers for sparse linear systems with both serial and parallel implementations. PETSc employs the Message Passing Interface (MPI) for parallel solutions. PETSc supports C, C++, FORTRAN and Python applications and contains a large number of parallel linear and nonlinear equation solvers for large-scale problems. Some of the PETSc modules deal with index sets (IS), including permutations, for indexing into vectors, renumbering, etc; vectors (Vec); matrices (Mat) (generally sparse); managing interactions between mesh data structures and vectors and matrices (DM) over fifteen Krylov subspace methods (KSP); dozens of precondition-

ers, including multigrid, block solvers, and sparse direct solvers (PC); nonlinear solvers (SNES); and time steppers for solving time-dependent (nonlinear) PDEs, including support for differential algebraic equations (TS). To evaluate the sparse linear system portion of the finite element analysis workflow, we chose matrices from the DRIVCAV collection. These matrices are from modeling 2D fluid flow in a driven cavity. The physical problem represented by the driven cavity is a square in cross section, with velocity equal to zero on three walls, and equal to one at the fourth wall, in the direction parallel to the fourth wall. This results in a circulating flow, similar to that which would occur in a notch in an infinite flat plate, with the notch cut perpendicular to the free stream flow direction over the plate. To produce the matrices, the flow was modeled using the incompressible Navier Stokes equations. These were discretized using the Galerkin finite element method and linearized using Newton's method. The matrices are non-symmetric and indefinite. They are difficult to solve using iterative methods like preconditioned Krylov subspace methods, because it is difficult to find an effective preconditioner. The matrices can be successfully solved using direct methods like frontal or skyline solvers, but as the size of the matrix and the Reynolds number increases, the filling in the lower (L) and upper (U) triangular factors increases, and this ultimately limits the use of these solvers. The specific matrices we chose were E40R000 (40 x 40 elements, Reynolds number 0, symmetric indefinite) and E40R0500 (40 x 40 elements, Reynolds number 500, real unsymmetric). Each of these matrices is 17281 by 17281, with 553956 entries. The matrices and the accompanying right hand side vectors are available on the Matrix Market website [1].

3.6.2 Experimental Setup

We evaluated the workflow using a set of eight novice users, similar to the previous experiment. These novice workflows were also evaluated against the expert user's workflow. Similar to the previous experiment timings and different paths the users took were recorded.

The workflow for solving the linear system using PETSc consists of the following tasks:

1. Download the matrix and right hand side vector from Matrix Market.

2. Convert the matrix and right hand side vector to the PETSC binary format.

3. Determine the properties of the matrix.

4. Based on the matrix properties, choose an appropriate solution method.

5. Implement the solution using the PETSc software.

6. Execute the program.

7. Evaluate the results.

8. If results are unsatisfactory, go back to step 4.

To evaluate the productivity of the workflow in two different contexts, we divided our subjects into two groups. One group used the Stampede and PETSc documentation to try to implement the workflow manually. The second group used the Lighthouse tool [12,13] to try to implement the workflow.

For each context, we evaluated productivity with respect to the following criteria:

1. Time to execute the program

2. Portability of the program

3. Maintainability of the program

4. Accuracy of the solution

5. Reusability of the results

Unlike in the first case study, where getting the code to run correctly with the vendor version of LAPACK would result in good performance, the performance of the PETSc workflow depends heavily on the choice of solver. With dense linear algebra methods, the number of steps is deterministic. With iterative methods for sparse systems, however, the number of steps to converge to a solution within the desired error tolerance depends

on how well suited the solver method is for the problem and on the effectiveness of the pre-conditioner. Also, the program may terminate without converging to a solution if the specified maximum number of steps is exceeded, but unless the user has output whether or not convergence occurred, she may incorrectly assume that the current approximation to the solution is correct. With PETSc, the same code can be used for different methods, with the choice of solver and pre-conditioner specified on the command line. Some level of expertise is required to know what solvers and pre-conditioners to specify and to figure out the PETSc command-line options to use to implement these choices.

Data were collected for both matrices for both groups. The subjects were instructed to solve the easier system first, followed by the harder system, and to use a relative error tolerance of 1e-10. In addition to collecting data from test subjects, we also collected data from the expert user carrying out the workflow in each of the two contexts.

3.6.3 Lighthouse

Lighthouse is a framework for creating, maintaining, and using a taxonomy of available software that can be used to build highly-optimized linear algebra computations [19], [17]. It aims to aid developers seeking to learn available tools for their programming tasks and to help them fit various parts together. Lighthouse assists scientists to explore the available libraries and apply the corresponding numerical software that suits their problem best. Lighthouse targets both developer and application's productivity enhancement by providing an environment that facilitates the user's selection of tools for dense and sparse linear algebra computations.

Lighthouse attempts to combine expert knowledge, machine learning-based classification of existing numerical software collections, and automated code generation and tuning to enable users to discover and apply the best available numerical software out of a several libraries covering a broad space of sequential and parallel solution methods for dense and sparse linear algebra. An organized classification of software as well as a variety of code generation and optimization capabilities and information about the code in the form of

automatically extracted documentation is provided for users. Lighthouse uses taxonomy-based search for identifying solution methods with code generation and optimization capabilities to accommodate a variety of different use cases that may arise in HPC software development. They claim to provide an interface that is more accessible and user friendly than the usual HPC tools available for advanced users and to be the first framework that offers a searchable ontology of linear algebra software with code generation and tuning capabilities. They also claim that it provides functionality- and performance-based search of high performance numerical software capabilities with current support for sequential and parallel dense and sparse linear algebra computations provided by the LAPACK, PETSc, and SLEPc libraries. We use Lighthouse in our second case study to evaluate how its functionalities can help users in solving sparse linear systems using the PETSc library and we give suggestions for improvement to the tool.

3.6.4 Results

As explained in section 3.5.3, we illustrate our observations and measurements of the user workflows using a workflow graph as in [20]. The expert user's workflow without using Lighthouse is illustrated in Figure 3.5. The corresponding steps and timings are given in Table 3.5.

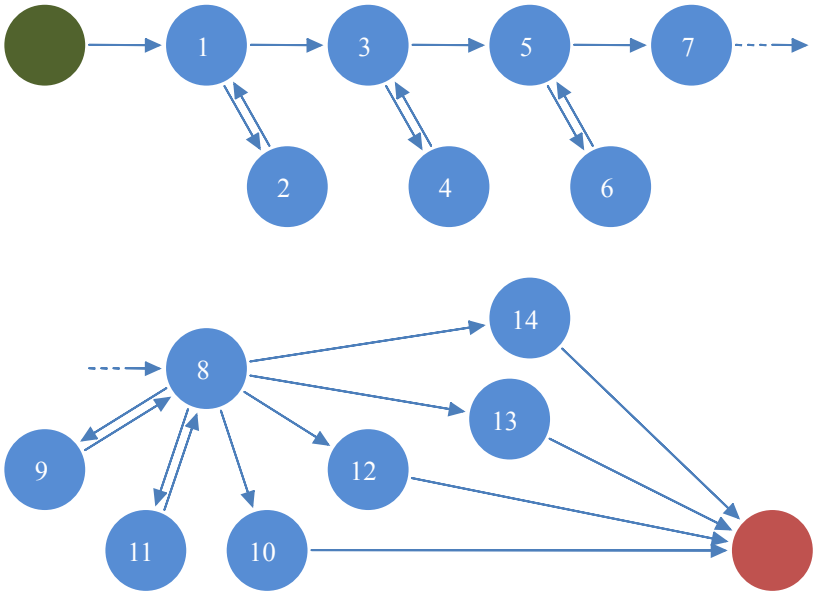

Figure 3.5: Workflow of expert user for case study 2, without Lighthouse

Table 3.5: Workflow of the expert user 1 for case study 2

Step	Task	Time (minutes : seconds)
	Start	
1	Downloaded matrix and right hand side vector from Matrix Market	2:00
2	Looked in the PETSc examples for code that could read Matrix Market format (unsuccessfully)	2:00
3	Looked for a program to convert the matrix and right hand side vector to the PETSC binary format and found mm2petsc code	3:00
4	Compiled mm2petsc with errors	1:00
5	Switched to PETSc 3.5 from the Stampede default version of 3.6	1:00
6	Converted the matrix from Matrix Market format to PETSc format and found that the mm2petsc program doesn't work for vectors	4:00
7	Modified mm2petsc to work for vectors and converted right hand side to PETSc binary format	12:00
8	Searched for PETSc ksp examples and found existing ex18.c, modified ex18.c to read matrix and vector from separate files	8:00
9	Attempted to solve using GMRES solved without success	2:00
10	Successfully solved in 3 iterations using GMRES with fieldsplit command pre-conditioner	2:00
11	Attempted to solve using MINRES solver without success	1:00
12	Successfully solved in xxx iterations using BICG solver	2:00
13	E40r0500: Attempted to solve using GMRES without success	2:00
14	E40r0500: Successfully solved in 427 iterations using BICG	2:00
	End	
	Total Time	44:00

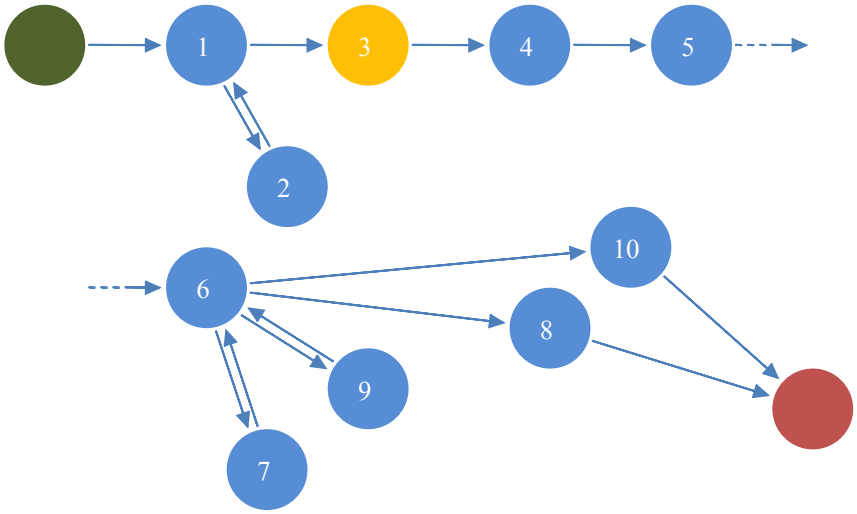

Figure 3.6: Workflow of expert user for case study 2, using Lighthouse

Note that the export workflow using Lighthouse does not include the tasks of converting the input files from Matrix Market format to PETSc binary format nor trying different solvers unsuccessfully.

Table 3.6: Workflow of the expert user 1 for case study 2, with Lighthouse

Step	Task	Time (minutes : seconds)
	Start	
1	Download the matrix and right hand side vector from Matrix Market.	2:00
2	Lookup the PETSc examples for code that could read matrix and vector format(unsuccessfully)	2:00
3	Tried Lighthouse using PETSC for sparse linear solutions by uploading the matrix E40r0000 but Lighthouse failed to analyze but it generated code template linear_solver.c	10:00
4	Modified linear_solver.c to read matrix and vector format (ascii)	1:00
5	Modified linear_solver.c to read separate files	4:00
6	Compiled using provided makefile	5:00
7	Attempted to solve using gmres solver without success	2:00
8	Solved successfully using fieldsplit preconditioner in 3 iterations	2:00
9	Solved successfully using bicg solver with 427 iterations	2:00
10	Solved E40r0500 using bicg solver in 546 iterations	2:00
	End	
	Total Time	32:00

The workflow for a novice user in the group that manually solved the problem is given below. A novice user's workflow without using Lighthouse is illustrated in Figure 3.7. The corresponding steps and timings are given in Table 3.7. The workflow diagram shows extensive suboptimal paths indicating that the students had to get a lot of help to get to the correct path for solving the problem.

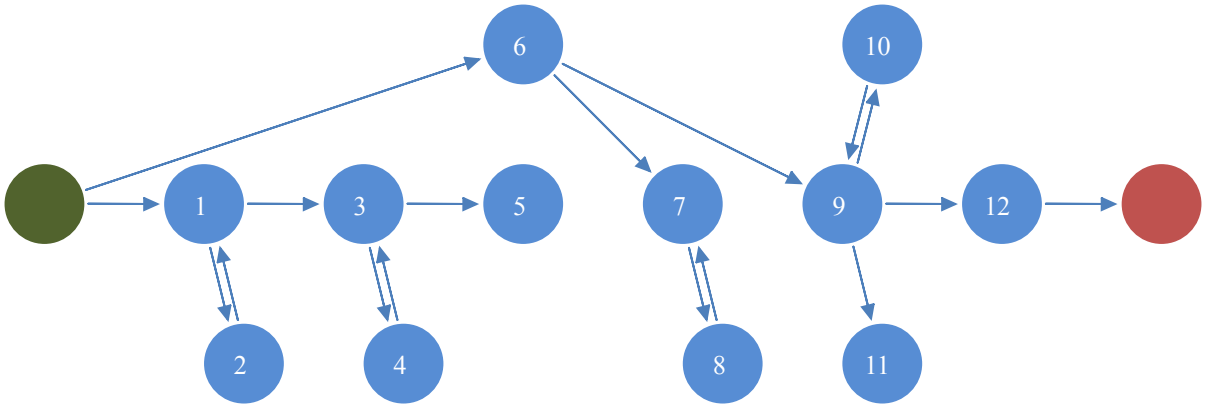

Figure 3.7: Workflow of a novice user for case study 2, without Lighthouse

Table 3.7: Workflow of a novice user for case study 2, without Lighthouse

Step	Task	Time (minutes : seconds)
	Start	
1	Copy and unzip file from stampede	4:00
2	Looked for PETSc functions (preconditioned iterative metnods)	4:00
3	Find the PETSc soclution code	3:00
4	Compiled ext18.c with errors	1:00
5	Search ways to run the ext18.c successfully	13:00
6	Got Instructer's help and the binary file was provided	1:00
7	Copy the file linear_solver.c to Stampede	1:00
8	Compiled the code without success	8:00
9	Got Instructer's help and the executable file was provided	1:00
10	For Matrix E40r0000 Attempted to solve using default command without converging	9:00
	End	
	Total Time	45:00

3.6.5 Analysis

The most efficient way found by the expert user to solve the symmetric indefinite system was using GMRES with the fieldsplit Shur preconditioner. The methods tried by the expert user are shown in Table 3.8.

Table 3.8: Transcript of expert user session for solution of E40R0000

c557-603.stampede(9)$ ./ex18 -ksp_type gmres -ksp_rtol 1e-10 -ksp_converged_reason -f input/e40r0000.bin -rhs input/e40r0000_rhs1.bin Linear solve did not converge due to DIVERGED_ITS iterations 10000 Number of iterations = 10000 Residual norm 0.000127714
c557-603.stampede(10)$ ./ex18 -ksp_type gmres -ksp_rtol 1e-10 -pc_type fieldsplit -pc_fieldsplit_type schur -pc_fieldsplit_detect_saddle_point -ksp_converged_reason -f input/e40r0000.bin -rhs input/e40r0000_rhs1.bin Linear solve converged due to CONVERGED_RTOL iterations 3 Number of iterations = 3 Residual norm 8.32361e-09
c557-603.stampede(11)$ ./ex18 -ksp_type minres -ksp_rtol 1e-10 -pc_type fieldsplit -pc_fieldsplit_type schur -pc_fieldsplit_detect_saddle_point -ksp_converged_reason -f input/e40r0000.bin -rhs input/e40r0000_rhs1.bin Linear solve did not converge due to DIVERGED_INDEFINITE_MAT iterations 2 Number of iterations = 2 Residual norm 8.88668e-05
c557-603.stampede(12)$ ./ex18 -ksp_type bicg -ksp_rtol 1e-10 -ksp_converged_reason -f input/e40r0000.bin -rhs input/e40r0000_rhs1.bin Linear solve converged due to CONVERGED_RTOL iterations 427 Number of iterations = 427 Residual norm 3.83479e-10

The most efficient method found by the expert user for solving the non symmetric system was the bicg method.

Table 3.9: Transcript of expert user session for solution of E40R0500

c557-603.stampede(13)$ ./ex18 -ksp_type gmres -ksp_rtol 1e-10 -ksp_converged_reason -f input/e40r0500.bin -rhs input/e40r0500_rhs1.bin Linear solve did not converge due to DIVERGED_ITS iterations 10000 Number of iterations = 10000 Residual norm 0.00139111
c557-603.stampede(14)$ ./ex18 -ksp_type bicg -ksp_rtol 1e-10 -ksp_converged_reason -f input/e40r0500.bin -rhs input/e40r0500_rhs1.bin Linear solve converged due to CONVERGED_RTOL iterations 546 Number of iterations = 546 Residual norm 1.22714e-10

The expert solutions were obtained using PETSC version 3.5 and by modifying $PETSC_DIR/src/ksp/ksp/examples/tests/ex18.c to read the matrix and the right-hand side from different files. The same solutions were easily obtained by modifying the code template produced by Lighthouse to read the matrix and right-hand side from different files and to compute the residual norm. Once the basic code has been compiled and made to run, the different solvers and pre-conditioners can be explored using runtime command-line options.

The most time consuming step in the expert workflow was converting the input files to PETSc binary format.

We divided the novice users into two groups of five and one was instructed to use Stampede and PETSc documentation and the second group was instructed to also use Lighthouse. Since none of the novice users were able to complete the case study 2 workflow at the first attempt we handed the mm2petsc and mv2petsc codes and they were able to complete the task. Only the second group was able to generate the linear_solver.c code template using Lighthouse but was unable to modify it to read the matrix and vector from

separate files during the next hour.

Special PETSc routines are provided to load a matrix (resp. vector) from a file and it takes several steps, that the novice user would need to look up the documentation for these routines and understand it to figure out how to modify the code. The original linear_solver.c code and the modified version by the expert user are given in Appendix. Since we had to give extensive help for solving the problem, we evaluate productivity for the second problem more qualitatively than quantitatively. If the code from Lighthouse is run using the run instructions provided by Lighthouse, solution vector is written to the output directory. To the novice user, it may appear that the method converged and a valid solution was generated. The expert user knew to add the option ksp_converged_reason so that PETSc would report whether the method converged or not. The expert user also had previous knowledge of sparse iterative methods that enabled her to know which solves to try. Choosing an effective preconditioner is also an expert skill. A preconditioner transforms the sparse matrix so that it is better conditioned which will hopefully help the iterative method to converge faster or to converge at all. The expert user had previous knowledge that the fieldsplit preconditioner is often effective for symmetric indefinite systems resulting from incompressible flow CFD problems.

Chapter 4: Code Complexity versus Performance and Programmability for Heterogeneous Programming Models on GPUs

4.1 Motivation

Graphics Processing Units (GPUs) are widely used as parallel accelerators in high-performance computing. Device-specific programming models require a steep learning curve and the expertise to program such hardware is limited.Device programmability plays a significant role in choosing new architectures to optimize prevalent scientific applications. While one often sees the statement that GPU programs written in a low-level GPU programming model such as CUDA or OpenCL have higher code complexity than those written using a higher-level directives-based model, little work has been done to quantify this assessment and to relate it in a quantitative way to software development and maintenance costs. It is important for developers of scientific applications, especially when a large collaborative development team is involved, to have a realistic estimate of the tradeoffs between performance and productivity so that they can make informed choices about what programming models to use. The software engineering community has developed metrics that are widely used to assess design and maintenance costs and risks for large software development projects. In this chapter, we discuss the assessment of three GPU programming models – CUDA, OpenCL and OpenACC – on three benchmark codes using software engineering metrics.

4.2 Heterogeneous Programming Models

Heterogeneous programming languages such as CUDA and OpenCL expose many low-level hardware details and can be used to achieve high performance. The memory hierarchy can be managed explicitly and memory accesses can be carefully handled to ensure optimiza-

tions such as coalescing. The code also explicitly controls the mapping of parallelism at multiple levels such as work groups, threads, and warps. While CUDA is specifically for NVIDIA GPUs, OpenCL is intended to be portable across platforms.

Directive-based programming of GPUs has been proposed as an alternative to using low-level languages such as CUDA C and OpenCL. This technique, which uses "directive" or "pragma" statements to annotate source code written in traditional high-level languages such as FORTRAN, C, and C++, is intended to allow a single code base to work across multiple computational platforms.

The OpenACC programming model provides a directive-based approach for device programming. OpenACC is a standard for high-level pragmas that enables C/C++ and FORTRAN programmers to utilize massively parallel coprocessors with much of the convenience of OpenMP. The OpenMP 4 specification adds directives for running code on target devices (accelerators), identifying work to be done by threads, and organizing threads into teams (an abstraction for blocks) and teams into a league (an abstraction for grids). It is possible that OpenACC and OpenMP will merge in the future.

4.3 Evaluation Approach

To measure source lines of code and cyclomatic complexity for C and C++ codes, we used the static analysis tool src-stat available as part of the Oxbow application characterization toolkit under development at Oak Ridge National Laboratory [21]. Src-stat uses the freely available cloc software to count lines of code (`https://github.com/AlDanial/cloc`). The version of cloc currently in use in Oxbow is version 1.60. To calculate cyclomatic complexity on a per-function basis, Oxbow uses the freely available pmccabe software (`http://packages.ubuntu.com/trusty/devel/pmccabe`). The version of pmccabe currently in use in Oxbow is version 2.6. To measure cyclomatic complexity, design complexity, and essential complexity of C, C++, and Fortran codes, we used the commercial McCabe IQ tool – Developers Edition (`http://www.mccabe.com/iq.htm`), for which we obtained a

30-day evaluation license. Essential complexity is a measure of the degree to which a module contains unstructured constructs. This metric measures the degree of structuredness and the quality of the code and is used to predict the maintenance effort. The McCabe IQ tool also classifies modules of a program as low, medium, or high risk, depending on combined results of all the measured metrics.

4.4 Test Codes

For our test codes, we used C versions of the Game of Life and LULESH with CUDA, OpenCL, and OpenACC GPU implementations, and a Fortran90 version of CloverLeaf with OpenACC and CUDA Fortran GPU implementations.

4.4.1 Game of Life

For our first test code, we used CUDA, OpenCL, and OpenACC extensions to the C version of the Game of Life (GOL) code. The original C version and the CUDA, OpenCL, and OpenACC implementations are all provided as tutorial codes on the OLCF Github tutorial website (`https://github.com/olcf/game_of_life_tutorials`). The GOL is an example of cellular automaton that utilizes a two-dimensional stencil. For each game iteration, the integer value of each cell in the 2D game grid (alive or dead, represented by 1 or 0, respectively) is determined by summing its eight closest neighbors and then applying the game rules, with the initial game state randomly generated. Cell updates are not propagated through until the end of each iteration, leaving the board static during calculations. We used the default 1024x1024 grid size with 1,000,000 iterations.

4.4.2 Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics

The different GPU versions of the LULESH code that we tested were obtained from the LULESH website (`https://codesign.llnl.gov/lulesh.php`). LULESH, which stands for Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics, is a proxy applica-

tion for the ALE3D hydrodynamics code. LULESH is designed to provide simpler but still full-featured hydrodynamics problems. The calculations that model continuum material properties and material interactions in the presence of applied forces can consume up to one third the runtime of scientific applications. Similar to ALE3D, LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. A node on the mesh is a point where mesh lines intersect. LULESH is built on the concept of an unstructured hex mesh. Indirection arrays that define mesh relationships are used. The default test case for LULESH appears to be a regular Cartesian mesh, but this is for simplicity only – it is important to retain the unstructured data structures as they are representative of what a more complex geometry will require. We tested serial, OpenACC, and CUDA versions of LULESH version 2.0 [12]. Although we were able to do static analysis of the OpenCL version, we have not yet been able to compile and run it due to missing header files in the distribution. We used the default mesh size of 303.

4.4.3 CloverLeaf Mini-App

For our third test code, we used the CloverLeaf mini-app from the Mantevo suite. CloverLeaf solves the compressible Euler equations on a Cartesian grid, using an explicit, second-order accurate method. Each cell stores three values: energy, density, and pressure. A velocity vector is stored at each cell corner. CloverLeaf currently solves the equations in two dimensions, but a 3D implementation has been started. The computation in CloverLeaf has been broken down into "kernels", which are low level building blocks with minimal complexity. Each kernel loops over the entire grid and updates one (or some) mesh variables, based on a kernel-dependent computational stencil. Control logic within each kernel is kept to a minimum with the intent of allowing maximum optimization by the compiler. We tested CloverLeaf version 1.3, using the Fortran 90 OpenACC implementation and the mixed Fortran90/C + NVIDIA CUDA implementation. We have only analyzed the existing source code of the OpenCL version.

4.5 Testing Environment

We compiled and ran our test codes on the Titan supercomputer at Oak Ridge National Laboratory [`https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/`]. For Game of Life and LULESH, we used the default PGI programming environment (PGI version 16.5 compilers) together with the default CUDA toolkit (version 7.5) as of August 2016. For CloverLeaf, we had to switch to the Cray compiling environment (CCE version 8.5.0) to get the CUDA and OpenACC versions of the code to run without error. For all three codes, we were able to use the provided Makefiles, with some minor changes for LULESH and CloverLeaf, to compile the codes. We ran each version of each code on a single compute node of Titan with an attached Kepler K20 GPU, since we were interested in the relative GPU performance of the different codes.

4.6 Testing Procedure

For the GOL C, CUDA C, and OpenCL codes and the LULESH C++, CUDA, and OpenCL codes, we measured SLOC and cyclomatic complexity using the Oxbow tools. We ran the McCabe IQ tool on the CloverLeaf Fortran90, Fortran90 + OpenACC, and FORTRAN portion of CloverLeaf_CUDA to measure cyclomatic complexity, design complexity, and essential complexity. We used the Oxbow tools to measure cyclomatic complexity of the CUDA portion of CloverLeaf_CUDA. We obtained the runtimes by using internal timers that called either `gettimeofday()` or `MPI_Wtime()`. The timings did not include initialization and setup parts of the codes.

4.7 Results: Code Complexity versus Performance

The runtimes for the different versions of the three codes are shown in Table 4.1. We have been unable so far to compile and run the OpenCL versions of LULESH and CloverLeaf. The CUDA version of GOL is an optimized version that uses shared memory. The total

lines of code for the different versions of the codes are shown in Table 4.2. The total cyclomatic complexities are shown in Table 4.3.

Table 4.1: Runtimes of code versions (in seconds)

Implementation	**GOL**	**LULESH**	**CloverLeaf**
Serial (MPI)	13.9	83.6	1458
OpenACC	0.26	17.8	16.9
CUDA	0.18	0.55	14.6
OpenCL	0.22		

Table 4.2: SLOC counts for code versions

Implementation	**GOL**	**LULESH**	**CloverLeaf**
Serial (MPI)	97	7240	9768
OpenACC	118	7918	8499
CUDA	157	8693	9005
OpenCL	293	6491	

Table 4.3: Total cyclomatic complexities for code versions

Implementation	**GOL**	**LULESH**	**CloverLeaf**
Serial (MPI)	22	1077	1302
OpenACC	22	997	1129
CUDA	29	1284	1115
OpenCL	44	723	

The source code for the serial version of GOL is actually a sequential code, but for LULESH and CloverLeaf we used the reference MPI versions and ran them with one process. This explains why the SLOC and total cyclomatic complexities are higher. The MPI

version of both codes also uses a different domain decomposition from the MPI+OpenACC and MPI+CUDA versions. The OpenCL version of LULESH is for a single node and thus does not include MPI code, making its total SLOC count and cyclomatic complexity appear lower than the CUDA and OpenACC versions – if the MPI code were included, these metrics would be as high or higher than that of the CUDA code. The Oxbow tools understand OpenACC, CUDA, and OpenCL codes as far as SLOC counts, but they appear to not distinguish between cyclomatic complexity between the OpenACC versions with and without the directives – that is, the directives are not considered when computing the cyclomatic complexity. This makes sense since the directives do not specify control flow although the code they get compiled to might. The distributions of function cyclomatic complexities for the LULESH versions are shown in Figures 4.1 through 4.4.

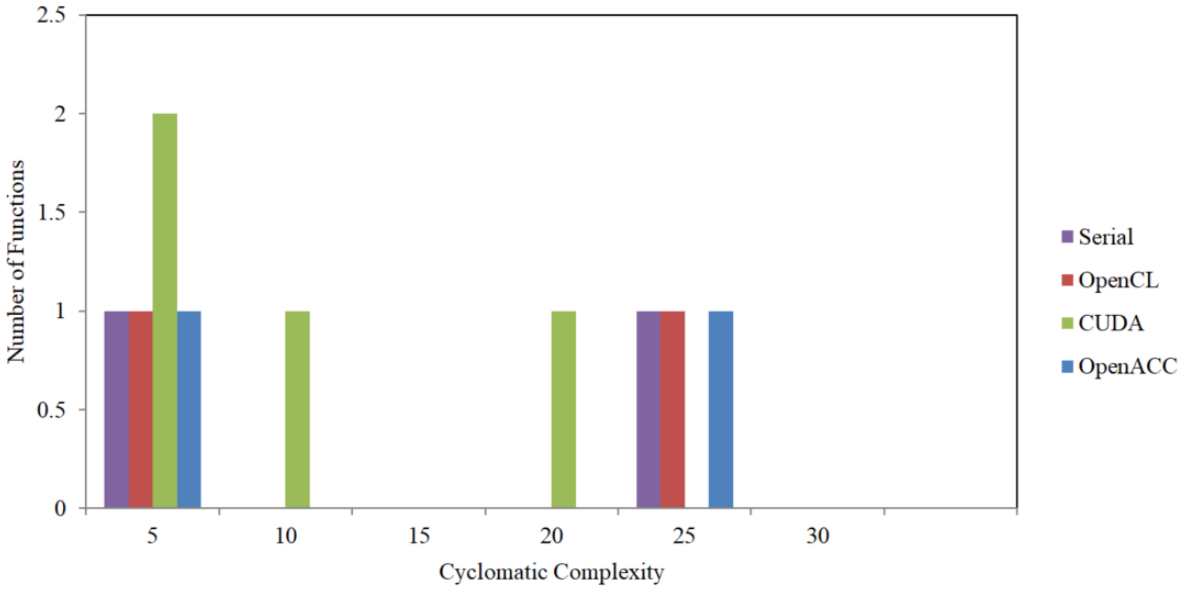

Figure 4.5: Cyclomatic complexity comparison of the versions of LULESH code

Figure 4.5 illustrates an overall picture of the cyclomatic complexity of the LULESH code. The CUDA versions tend to have a larger number of functions with low cyclomatic complexity since CUDA kernels have few branches.

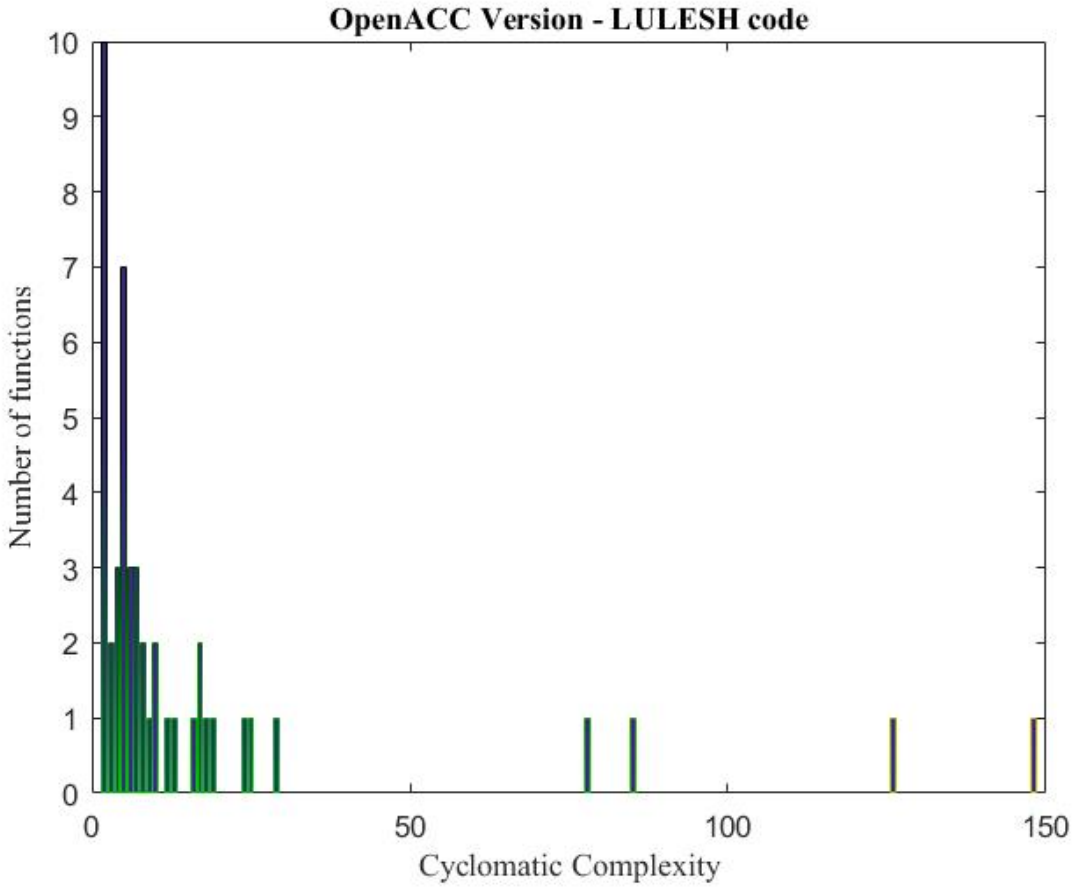

Figure 4.1: Cyclomatic complexity of the OpenACC version of LULESH

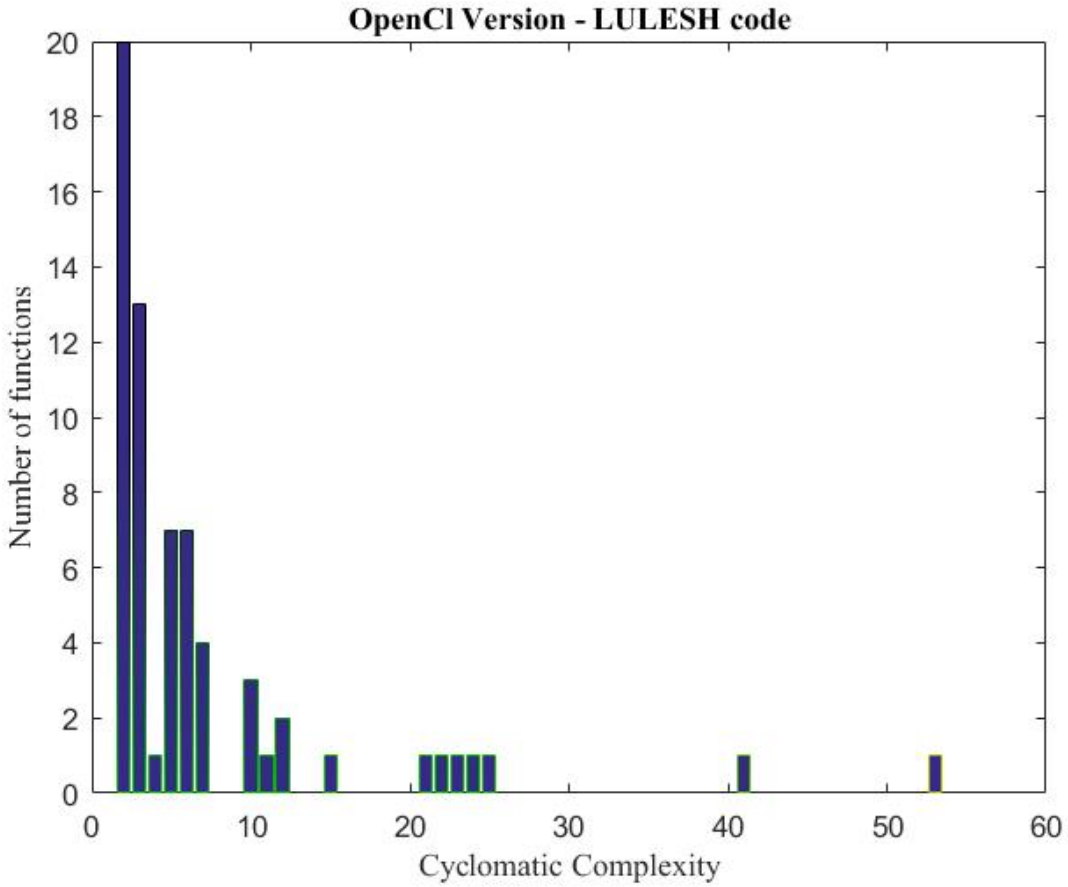

Figure 4.2: Cyclomatic complexity of the OpenCL version of LULESH

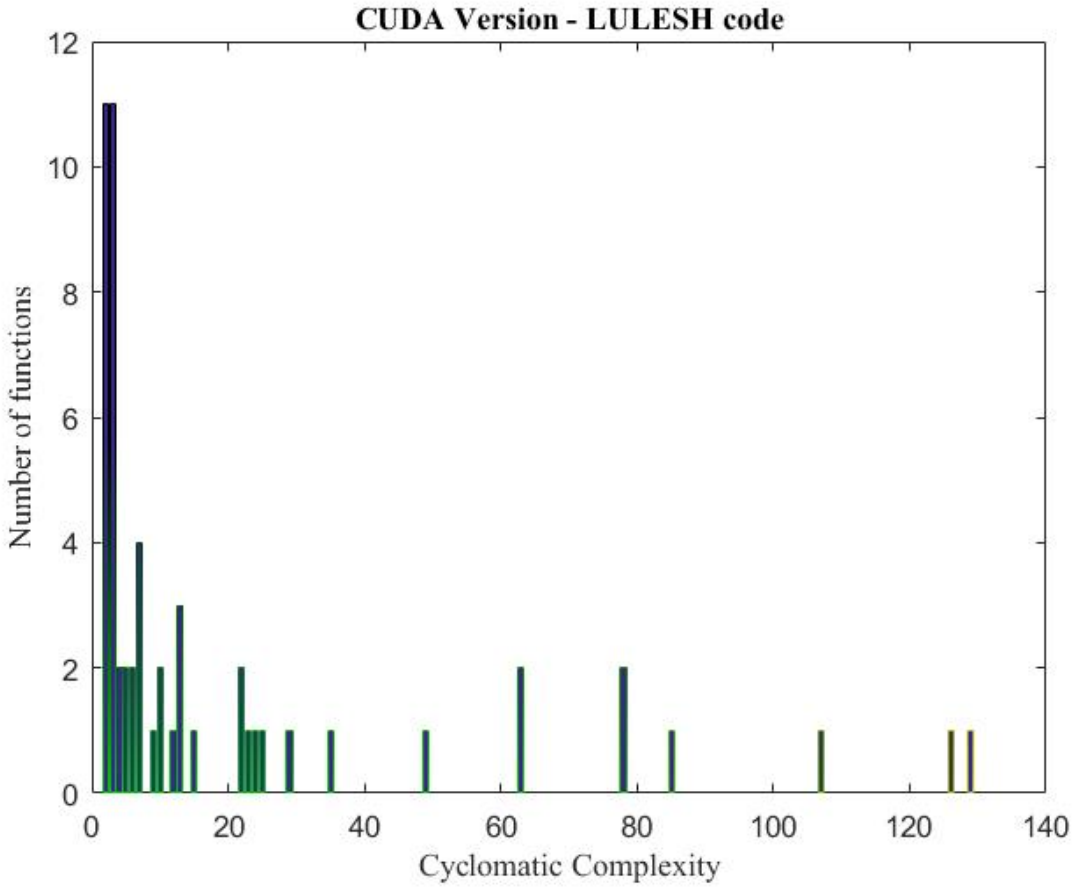

Figure 4.3: Cyclomatic complexity of the CUDA version of LULESH

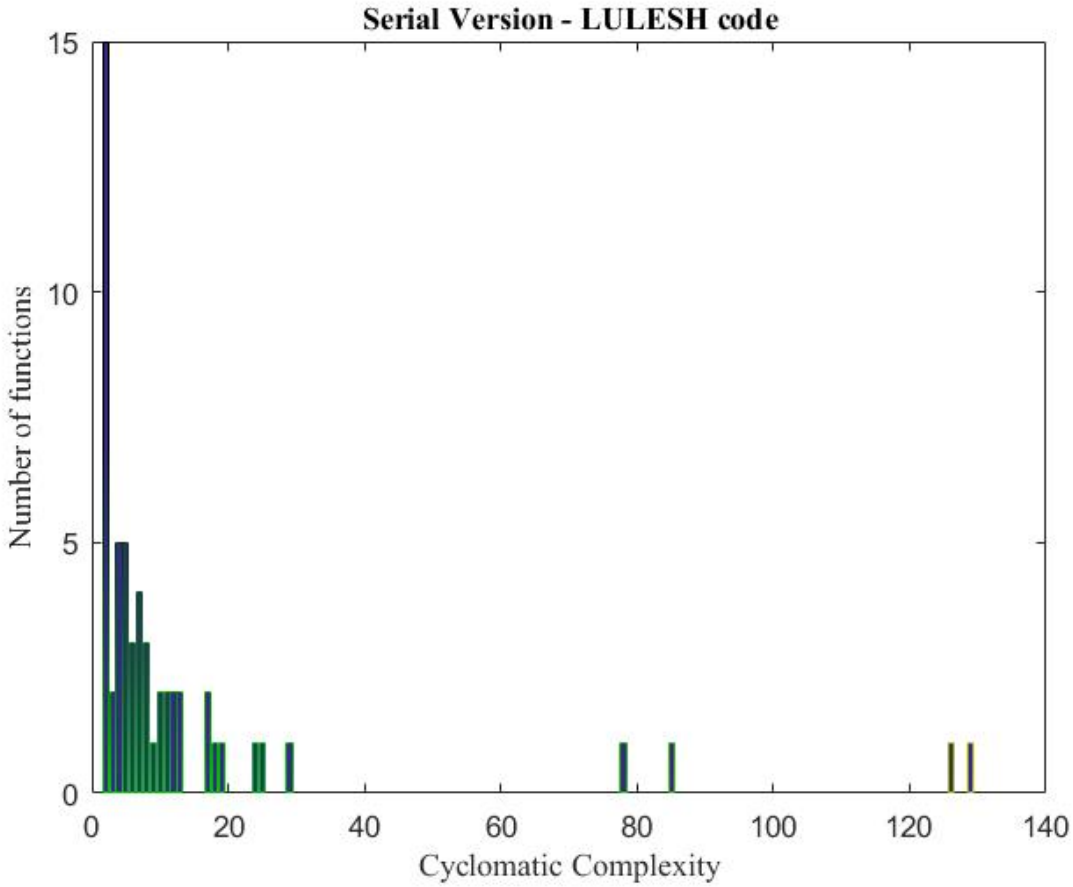

Figure 4.4: Cyclomatic complexity of the serial version of LULESH

The McCabe IQ tool considers modules to be low risk if cyclomatic complexity is 10 or less and essential complexity is 4 or less, medium risk if cyclomatic complexity is greater than 10 and essential complexity is 4 or less, and high risk if essential complexity is greater than 4. We do not show essential complexity in the figures, but all modules in all codes had essential complexity 4 or less. However, several modules in all versions of the LULESH and CloverLeaf codes have cyclomatic complexity greater than 10.

Chapter 5: Evaluating Productivity of a Scientific Workflows on a Heterogeneous Architecture with FPGAs and GPUs

5.1 Reconfigurable Computing on Next Generation Supercomputers

HPC systems containing conventional x86 architectures have encountered the problems of limited performance gain with CPUs alone, and the cooling and power supply cost are high. Reconfigurable devices such as FPGAs, incorporating different instruction-set architectures from general purpose CPU architectures could be available on upcoming supercomputers. Low power consumption and the flexibility of custom interfacing with devices could lower the communication latency on FPGAs.

A substantial time is spent by scientific application developers to utilize new accelerators, at the same time keeping the program output to remain correct, despite the changing technical specifications of the programming environment, overtime. Thus, heterogeneous system workflows utilizing FPGA accelerators may require more design effort compared to pure software application workflows and impact programmer productivity.

5.2 Problem Description

We port a GAMESS-SIMGMS computational chemistry kernel to two FPGA based systems - older Altera Stratix® V FPGA and the newer Arria® 10 FPGA, using the OpenARC compiler. So far, the compiler has been tested on smaller known benchmarks, but scientific applications haven't been implemented with the compiler on FPGAs. We port the same kernel to GPUs using OpenARC and The workflow of porting to Altera Stratix® V FPGA is evaluated against porting to a Nvidia Tesla® GPU.

The problem of predicting the performance and productivity of porting a scientific code

on a new device such as a FPGA is not straightforward. The discussion hereafter, on the porting process of an application kernel on FPGAs and GPUs, is presented as preliminary guidance from an application programmer's point of view in the context of the lessons learned from the previous chapters.

5.3 GAMESS-SIMINT Hartree-Fock Quantum Chemistry Method

Following is an excerpt from Keipert, K.(kkeipert@anl.gov), "Knowledge is power: quantum chemistry on novel computer architectures", describing the HF quantum chemistry method. GAMESS [8] computational chemistry application encompasses of several computational chemistry empirical, semi-empirical, and ab initio methods in quantum mechanics which are used to compute the structure and properties of molecular systems. The GAMESS-SIMINT Hatree-Fock quantum chemistry method computations computes molecular properties and is used as a starting point for higher accuracy, and for more computationally demanding methods. The computational bottleneck of the Hatree-Fock procedure lies in the construction of the Fock matrix.

Fock (HF) self-consistent field (SCF) method is the construction of the Fock matrix $F_{\mu\nu}$ by

$$F_{\mu\nu} = H_{\mu\nu}^{core} + \sum_{\lambda\sigma}^{AO} D_{\lambda\sigma}[2(\mu\nu|\lambda\sigma) - (\mu\lambda|\nu\sigma)]$$

where H^{core} is the core Hamiltonian matrix, the sum runs over all atomic orbitals (AO), D is the density matrix, and μ, ν, λ, and σ are indices which range over n basis functions. Each $(\mu\nu|\lambda\sigma)$ quantity is an electron repulsion integral (ERI) defined by:

$$(\mu\nu|\lambda\sigma) = \int d\boldsymbol{r}_1 d\boldsymbol{r}_2 \phi_\mu(\boldsymbol{r}_1)\phi_\nu(\boldsymbol{r}_1) r_{12}^{-1} \phi_\lambda(\boldsymbol{r}_2)\phi_\sigma(\boldsymbol{r}_2)$$

where ϕ are Gaussian basis functions centered at atomic coordinates and r_{12} is the distance $|r_2 - r_1|$. Each Gaussian basis function is a linear combination of k primitive Gaussians centered on a nucleus, defined by:

$$\phi_\alpha = \sum_k N_{k\alpha} x^a y^b z^c e^{-\zeta_k r^2}$$

where $N_{k\alpha}$ is a contraction coefficient, x, y, z are the Cartesian coordinates of the nucleus, a, b, c are positive integers controlled by the angular momentum of the basis function ($L = a+b+c$), ζ is an exponent which controls the width of the orbital, and $r2 = x2+y2+z2$. Basis functions are grouped into sets called shells with the same angular momentum and atomic center. For computational efficiency, ERIs are computed at the granularity of "shell quartets" which group the basis set information of four shells. The number of ERIs computed varies for each shell quartet according to the number of Basis functions contained in each shell. Time to solution of the Hatree-Fock (HF) method scales as O(N4), where N is the number of Gaussian basis functions.

The computational effort required to compute a shell quartet depends on the angular momenta of the quartet shells. An (ff—ff) quartet includes many more Gaussian functions than an (ss—ss) quartet, and each integral of the former type is generally more expensive to compute. This complicates the efficient distribution of shell quartets across computer processes. Various algorithms are available to compute ERIs such as the Obara-Saika, Rys Quadrature, and McMurchie-Davidson schemes. The most computationally efficient algorithm depends on the angular momenta of the shell quartet, and the extent of Gaussian primitive contraction.

5.3.1 Integral Evaluation in GAMESS

The general routine for evaluation of two-electron integrals in the FORTRAN 77 GAMESS program is discussed here. A pseudocode representation of the main two-electron integral evaluation driver TWOEI is shown in Figure 5.1.

```
DO ISHELL=1, NSHELL
    DO JSHELL=1, ISHELL
        DO KSHELL=1, JSHELL
            DO LSHELL=1, KSHELL
                1. SCREEN INTEGRAL SHELL
                IF (NOT SCREENED) THEN
                    2. COMPUTE ERIs OVER UNIQUE SHELLS
                        (IJ|KL)
                        (IK|JL)
                        (IL|JK)
                    3. UPDATE FOCK MATRIX
                END IF
            END DO
        END DO
    END DO
END DO
```

Figure 5.1: Pseudocode representation of GAMESS Integral Driver Subroutine TWOEI

For each iteration over the inner loop, up to three symmetry-unique integral batches are computed. This is a blocking technique called "triple sort", which reduces the number and size of data transfer messages compared to canonical ordering . MPI parallelization is implemented over the ISHELL and JSHELL loops, with dynamic load balancing implemented after the JSHELL loop. The innermost loop passes unscreened shell quartets to the SHELLQUART subroutine. Depending on the angular momenta of the quartet shells, an ERI computation algorithm is chosen among the ERIC , rotated axis , and Rys Quadrature [25, 26] methods in SHELLQUART. Cartesian Gaussians within shells are arranged into groups with descending powers of Cartesian products, with each group arranged in alphabetical order (e.g. X_3, Y_3, Z_3, X_2Y, X_2Z, Y_2X, Y_2Z, Z_2X, Z_2Y, XYZ). The computed integrals are immediately used to compute a partial contribution to the Fock matrix.

SIMINT Integral Evaluation Library The SIMINT integral library TODOcite 34. is an implementation of the Obara-Saika . (OS) method written in the C programming language. SIMINT was written to take advantage of single-instruction, multiple-data (SIMD) vectorization capabilities of computer processors. SIMD instructions apply a single operation to multiple data points at the same time. Vectorization is becoming increasingly important as high performance computing hardware trends toward larger vector register lengths.

While software compilers can automatically vectorize code in limited cases, careful manual restructuring of algorithms is usually required to maximize vectorization. The SIMINT library is built with a C++ code generator, which provides flexibility to easily modify the library. For example, the generator can be configured to optimize SIMINT code for the SIMD vector length of a target hardware system, or to change the ordering of computed integrals (including GAMESS ordering). The ability to generate complex code that is customized for hardware targets and/or software interfaces is an extremely powerful tool for performance portability and software interoperability. Several code generators have been widely adopted in computational chemistry, including other code generators for ERI evaluation . Instead of the canonical four-index loop over shells presented Figure 5.1, the loop structure in SIMINT is implemented as a two-index loop over pairs of shells. Data corresponding to pairs of shells (e.g. coordinates of Gaussian centers, primitives and contraction coefficients) are stored in shell pair data structures. A shell pair combines two shells corresponding to the bra or ket part of an integral quartet. Data corresponding to multiple shell pairs with the same angular momentum can be stored in a single shell pair data structure. This scheme presents two main advantages. First, several prefactors required for integral evaluation can be computed from pairs of shells in advance of the main loops over shells and primitives. Second, the integral evaluation function can operate on multiple shell quartets in one function call. SIMD registers can be efficiently utilized by filling vector lanes with primitives from different contracted shell quartets. Further details regarding the SIMINT implementation and OS method can be found elsewhere. In the present work, only the key differences between SIMINT and GAMESS that impact integration of the codes are discussed further.

5.3.2 GAMESS-SIMINT Integration

Supporting the loop structure over shell pairs in SIMINT requires significant modification of the GAMESS SCF driver. Because the SCF algorithm is relatively straightforward, the strategy for SIMINT-GAMESS integration is to encapsulate an entire SCF kernel in

a C++ interface with GAMESS. First, the call to the GAMESS SCF driver was replaced with a conditional option to call a FORTRAN wrapper subroutine that directs execution to the GAMESS-SIMINT SCF code. The wrapper routine imports all of the input data required for the SCF routine that was initialized by GAMESS (e.g. basis set data, nuclear charges, SCF convergence tolerances) and passes the information as function parameters to the GAMESS-SIMINT SCF driver. The parameters are matched to equivalent C++ data types and specified as const to avoid data modifications which might impact post-Hartree-Fock routines. Next, an initialization function is called to copy the GAMESS Gaussian shell data into simint_shell data structures. The simint_shell structures are stored in a two-dimensional C++ vector for convenience, with shells of the same angular momentum grouped into rows. GAMESS sp shells (pairs of s-type and p-type primitives with shared exponent values) are separated into individual s and p simint_shell structures. Next, arrays are allocated to store overlap integrals and the core Hamiltonian, and the one-electron integral driver is called. The OED one-electron integral library was interfaced with GAMESS-SIMINT for this task. OED was developed concurrently with ERD, and the function arguments are almost identical to ERD. The one-electron integral driver is a four-fold loop that iterates over pairs of simint_shell vector rows and columns. Overlap, kinetic, and nuclear attraction integrals are computed for all unique pairs of shells. As with ERD, the optimum integer and floating point memory requirement are computed before each call to an integral evaluation function. For most of the OED function parameters, members of the simint_shell structs are passed directly. One exception is the basis set contraction coefficients and exponents for shell pairs, which must first be copied into a single array. Once all one-electron integrals are computed, core Hamiltonian and overlap integrals are returned to the GAMESS-SIMINT SCF driver. The typical SCF routine follows, with an initial Fock matrix formed using the core Hamiltonian as a guess, and construction of an initial density matrix.

Prior to the start of the SCF iterations, the two-electron integrals are computed with SIMINT and stored in memory. First, a two-fold loop iterates over the simint_shell vec-

tor and initializes simint_multi_shellpair (SMS) structures (corresponding to the integral bra or ket pairs). All shell pairs of a given type are grouped into the same SMS structure, but the shell pairs could potentially be distributed into separate SMS structures if desired (for example, to distribute the work into multiple function calls during a parallel run). The SIMINT integral evaluation routine is then called within a second two-fold loop over symmetry-unique pairs of SMS structures. The output buffer of computed integrals typically contains integral values corresponding to multiple shell quartets. A final loop over the output buffer determines the i,j,k,l indices for each value, and stores the quantities accordingly in a one-dimensional array containing all computed integral values. Once all two-electron integrals are computed, a conventional SCF iterative procedure is executed. CBLAS/LAPACK are used for all low-level linear algebra routines throughout the GAMESS-SIMINT code, as provided by Intel MKL.

5.4 Field Programmable Gate Arrays (FPGAs)

Field Programmable gate arrays (FPGAs) were introduced by Xilinx in 1985. FPGAs contain a set of programmable logic that can be arranged to build an array of computing resources resulting in an application dependent structure [3]. In commercial FPGA boards, resources such as cache memory can be directly found on the chip.

Programmable interconnections that connect the logic blocks can reduce the performance of FPGAs. Vendors provide frequently used modules as hard macros to overcome this problem, as is the case in Intel Arria® 10 FPGAs. Programmable interconnections are only available between processing elements as hard macros on the chip. Those devices are made upon a set of hard macros (8-bit, 16-bit or even a 32-bit ALU), usually called processing elements (PEs).

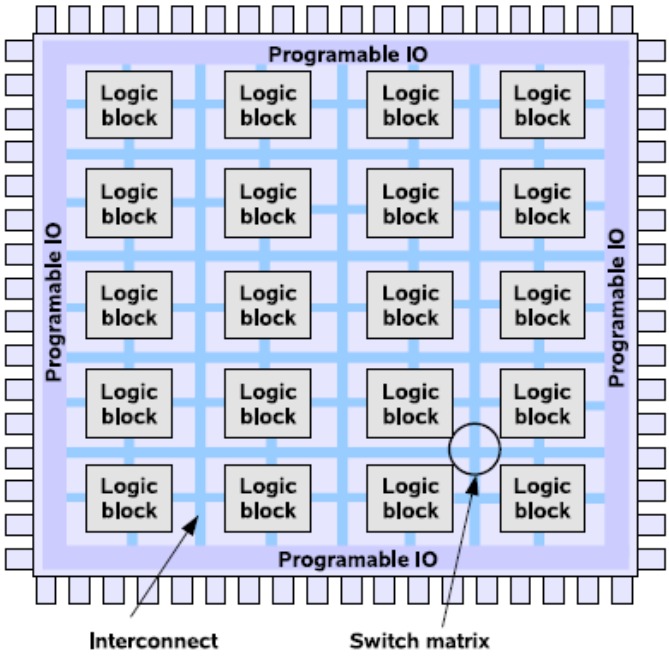

Figure 5.2: Structure of an FPGA [3]

5.4.1 FPGA Design Flow

FPGA implementation does not sequentially generate an instruction set sequentially but hardware components are mapped at different time on the available resources. The hardware required by the application for computation is built as components to be downloaded to device at runtime. Hardware generation is called logic synthesis. In the technology mapping step, application is mapped to the FPGA resources.

The design flow of a dynamic reconfigurable system identifies parts of the code to be executed on the reconfigurable device, the parts of the code to be executed on the host processor and implements the interface between the processor and the reconfigurable device. For the data-dominated parts efficient dataflow computation modules are required. The implementation of the control part on the CPU is usually done with C-program with system calls to access the device for reconfiguration. The usual steps stated in Figure 5.3 concern only the parts of the application that is identified by the hardware/software co-design

process to be executed on the FPGA.

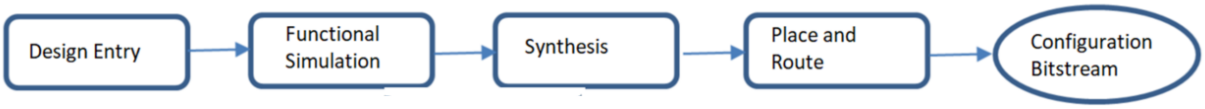

Figure 5.3: FPGA design flow

5.4.2 FPGA Design Tools

CAD tools perform the design entry, the functional simulation and the logic synthesis. Vendor tools such as Quartus are used for the placement, routing and the generation of configuration data.

Table 5.1: Available tools by vendor [3]

Manufacturer	**Tool**	**Description**
Synopsys	FPGA Compiler	Synthesis
Mentor	FPGA	Synthesis, place and route
Synplicity	Sinplify	Synthesis, place and route
Xilinx	ISE	Synthesis, place and route (only Xilinx products)
Altera	Quartus II	Synthesis, place and route (only Altera products)
Actel	Libero	Synthesis, place and route (only Actel products)
Atmel	Figaro	Synthesis, place and route (only Atmel products)

5.4.3 FPGA Programming

Maturity of compilers as well as the Von Neumann paradigm that allows any sequential program to be executed on the underlying hardware is the success behind microprocessor programming. A vast pool of algorithms exists in high-level languages such as Fortran, C,

C++ along with a community of programmers for conventional microprocessor programming.

Despite all the possible performance benefits, having to write code for FPGAs in a given hardware description language such as VHDL or Verilog makes it hard to decouple the programming process from the hardware. High-level synthesis enables describing the electronic system using a portable language such as OpenCL that can be compiled down to the hardware without manual intervention of the software developer.

5.4.4 Programming FPGAs on HPC Systems

After two decades since FPGAs were introduced, industry has merged the high performance of ASICs (Application Specific Integrated Circuits) with the flexibility of microprocessors onto current FPGA models. Recently these reprogrammable chips containing many registers, configurable logic gates and interconnections have started to become popular as computation accelerators. FPGA fabric can be made to mimic any processing unit, and enables users to tailor the machine to the problem in contrast to ASICs, whose logic is fixed at fabrication time. FPGAs allow large and complex SoC (System on Chip) for exact application need and the users are limited only by available resources rather than vendor specific architectures. But developers need to be aware of both hardware and software issues to exploit the programming opportunities with reconfigurable logic with the available technology. For obtaining full performance and resource utilization, FPGAs need to be programmed using low level hardware description languages. The level of expertise needed for programming these devices is limited. Writing scientific code using hardware description languages (HDL) is too complex, since programming HDLs require extensive knowledge in design automation tools, knowledge of logic design and low level details of FPGA architectures.

Altera Stratix® V and Arria® 10 boards are shipped with a supporting FPGA SDK for OpenCL that abstracts lower level details, targeting heterogeneous platforms. The Intel OpenCL SDK for FPGAs converts OpenCL code to an FPGA reconfigurable. The OpenCL

programming model covers the programming gap of FPGAs extending its portability from other devices such as GPUs and APUs. However, the current API lacks support for common mathematical libraries such as MKL.

Table 5.2: FPGA specifications

Altera Stratix® V FPGA specification
Host Interface - 8-lane PCI-Express upto Gen 3.0 Four 10G LAN/WAN/FC Ethernet channels accessed via 4 SFP+ ports Four banks of mini-UDIMM, each bank with 8GByte, x72, DDR3 SDRAM running at 1066 MT/s Configurator device for FPGA - either on-chip 512Mb FLASH or USB connection
Intel Arria® 10 FPGA specification
Host Interface - 16-lane PCI-Express Gen 3.0 DDR4 SDRAM Memory - Eight banks of DDR4 SDRAM x 72 bits / Four 4GB banks per FPGA (32GB total) Transfer rate - 2133 MT/s peak single precision performance - Up to 3 TFLOPS Peak Aggregate Memory Bandwidth - Up to 150 GBytes/s Two Arria 10 1150 GX FPGAs 75GB/s Peak DDR4 Memory Bandwidth per FPGA (4 Banks per FPGA)

OpenARC [15], [14], developed at Oak Ridge National Lab, provides better FPGA programmability by source to source translation of OpenACC directive based code into OpenCL, optimized for FPGAs. OpenARC takes the OpenACC C program as input and generates a hardware configuration file to execute on a FPGA device. The hardware specification is further compiled into a FPGA program by Altera offline OpenCL compiler. The same OpenARC compiler supports both OpenCL and CUDA back ends.

5.5 Workflow 1: Porting GAMESS-SIMGMS Kernel to FPGAs using OpenARC

5.5.1 Steps of Workflow 1

1. Setup GAMESS code on HPC system with FPGA devices.

2. Setup the HPC system to support GAMESS-SIMGMS module

3. Convert SIMGMS kernel into pure C code
    - Redirect output to the terminal
    - Compare for correctness and debug
4. Setup OpenARC compiler with FPGA environment settings
5. Configure GAMESS-SIMGMS setup for OpenARC compiler for FPGA environment
6. Add OpenACC directives and compile with OpenARC compiler to generate code to run on FPGAs
    - OpenARC generates optimized OpenCL device code for the FPGA
    - GNU compiler generates the non-FPGA executable.
    - Run the executable on FPGA device using Altera Quartus.

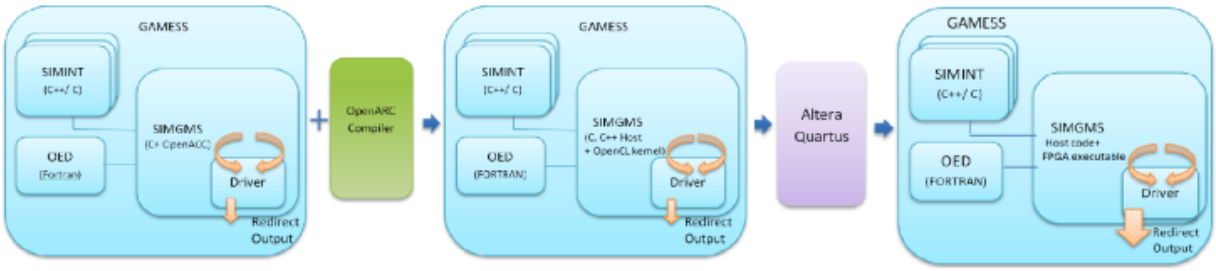

Figure 5.4: Application setup of porting computational chemistry kernel to FPGA

5.6 Workflow 2: Porting GAMESS-SIMGMS Computational Chemistry Kernel to GPUs using OpenARC

1. Setup GAMESS code on the HPC system with GPU devices.
2. Setup the GPU enabled HPC system to support GAMESS-SIMGMS module
3. Convert GAMESS-SIMGMS module into pure C code by hand

- Redirect output to the terminal
- Debug and verify output

4. Setup OpenARC compiler with GPU environment settings
5. Configure GAMESS-SIMGMS setup for OpenARC compiler for GPU environment
6. Add OpenACC directives to GAMESS-SIMGMS kernel and compile to generate executable to run on GPU device

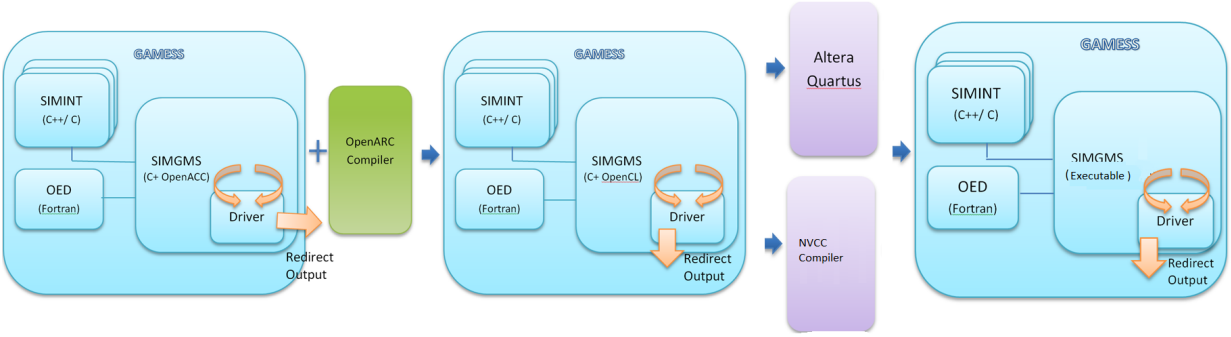

Figure 5.5: Application setup of porting computational chemistry kernel to FPGA vs. GPU

The GAMESS-SIMGMS kernel consists of several modules written in C, C++ and Fortran. Since OpenARC supports only C, we translated the code to C before inserting OpenACC directives as input. We evaluate the programmability by comparing the SLOC counts for the two cases to represent writing code for FPGA programming using OpenARC vs. by hand. OpenARC implements performance optimizations that expert programmers would do by hand.

5.7 Results: GAMESS-SIMGMS Computational Chemistry Kernel on FPGA vs. GPU Architectures

We compare execution times of the GAMESS-SIMGMS kernel with increasing problem sizes on Altera Stratix® V FPGA, Intel Arria® 10 FPGA, NVIDIA Tesla® C2050 CPU

and Intel Xeon® E5520 CPU.

5.7.1 Performance on Altera Stratix® V Board

Figure 5.6 shows the runtime of the kernel on an Altera Stratix® V FPGA vs. an Intel Xeon® E5520 CPU on a log scale. The problem size is the number of Gaussian basis set functions varying from 50 to 100.

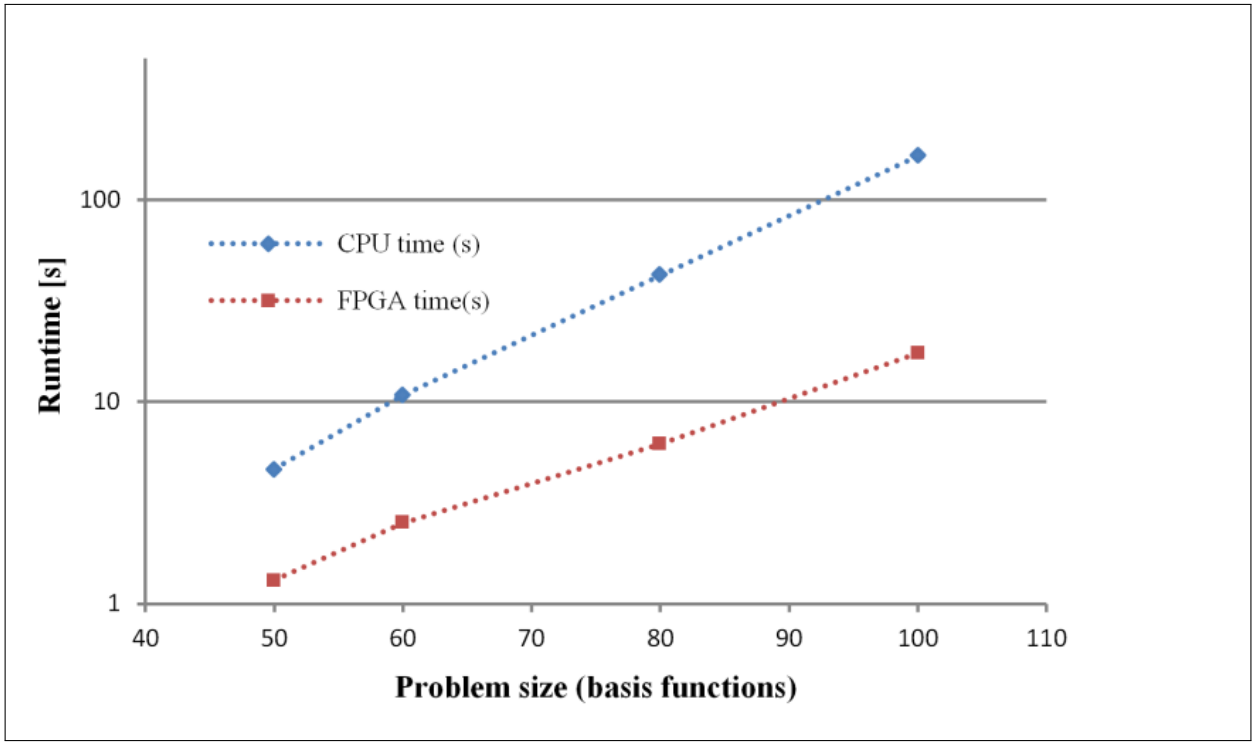

Figure 5.6: Runtime on Stratix® V FPGA vs. Intel Xeon CPU (logarithmic scale)

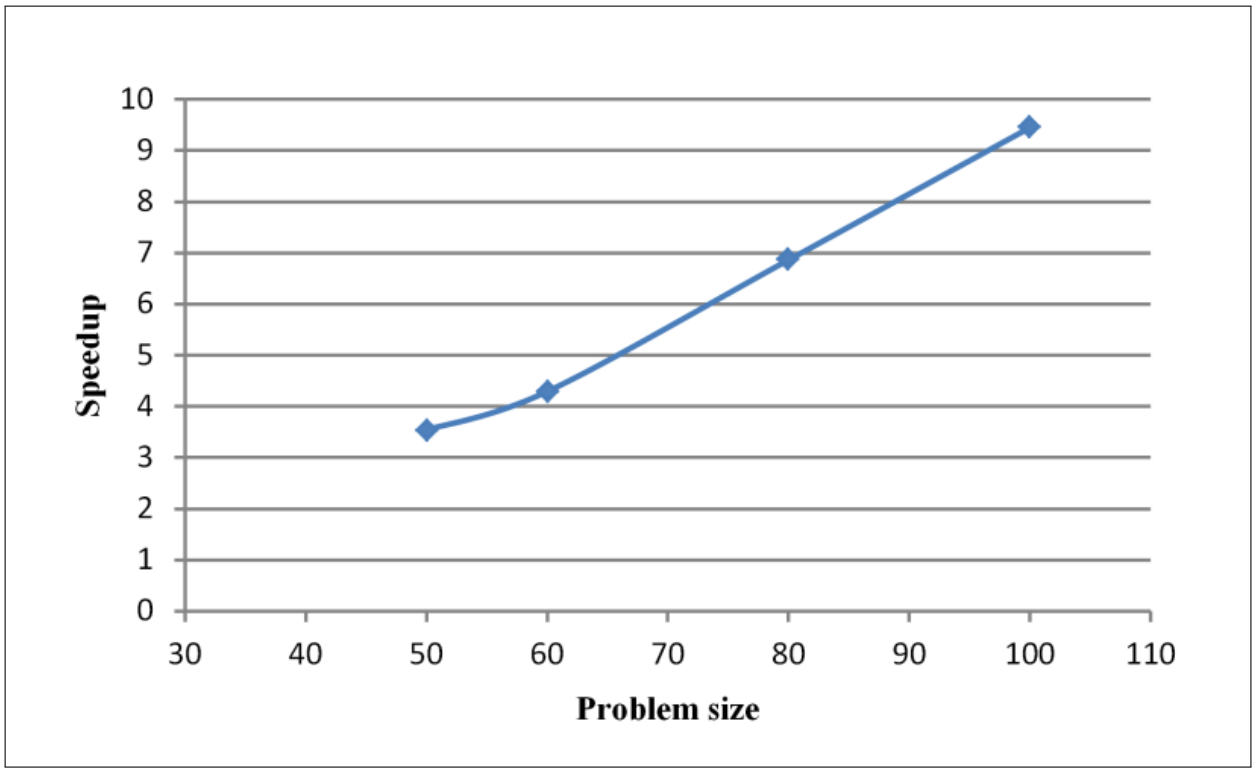

Figure 5.7: Speedup on Stratix® V FPGA vs. Intel Xeon CPU

The speedup achieved with different problem sizes on the Altera Stratix® V FPGA vs. Intel Xeon E5520 CPU is shown in Figure 5.7. The maximum speedup we achieved on the Stratix V FPGA board is 9.5X for the problem size of 100.

$$Speedup = \frac{Runtime\ on\ the\ CPU}{Runtime\ on\ the\ accelerator}$$

The tested FPGA (Stratix® V) does not contain dedicated floating point cores but the floating point units are synthesized from existing building blocks.

5.7.2 Performance on Intel Arria® 10 Board

Figure 5.8 shows the comparison of the execution times of the kernel with increasing problem sizes, on an Intel Arria® 10 FPGA accelerator board and an Intel Xeon® E5520 CPU.

Figure 5.9 indicates the speedup achieved for different problem sizes for the same problem set. We achieve up to 64X speedup on the FPGA.

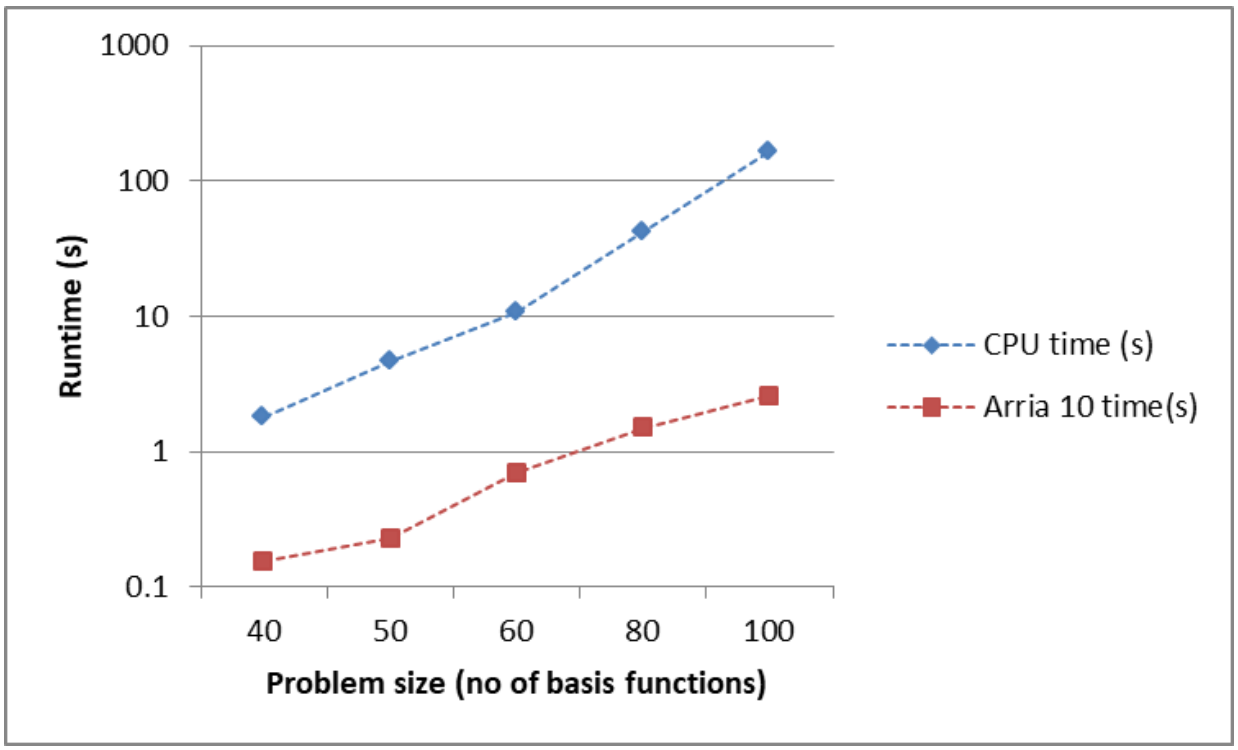

Figure 5.8: Runtime on Arria® 10 FPGA vs. Intel Xeon CPU (logarithmic scale)

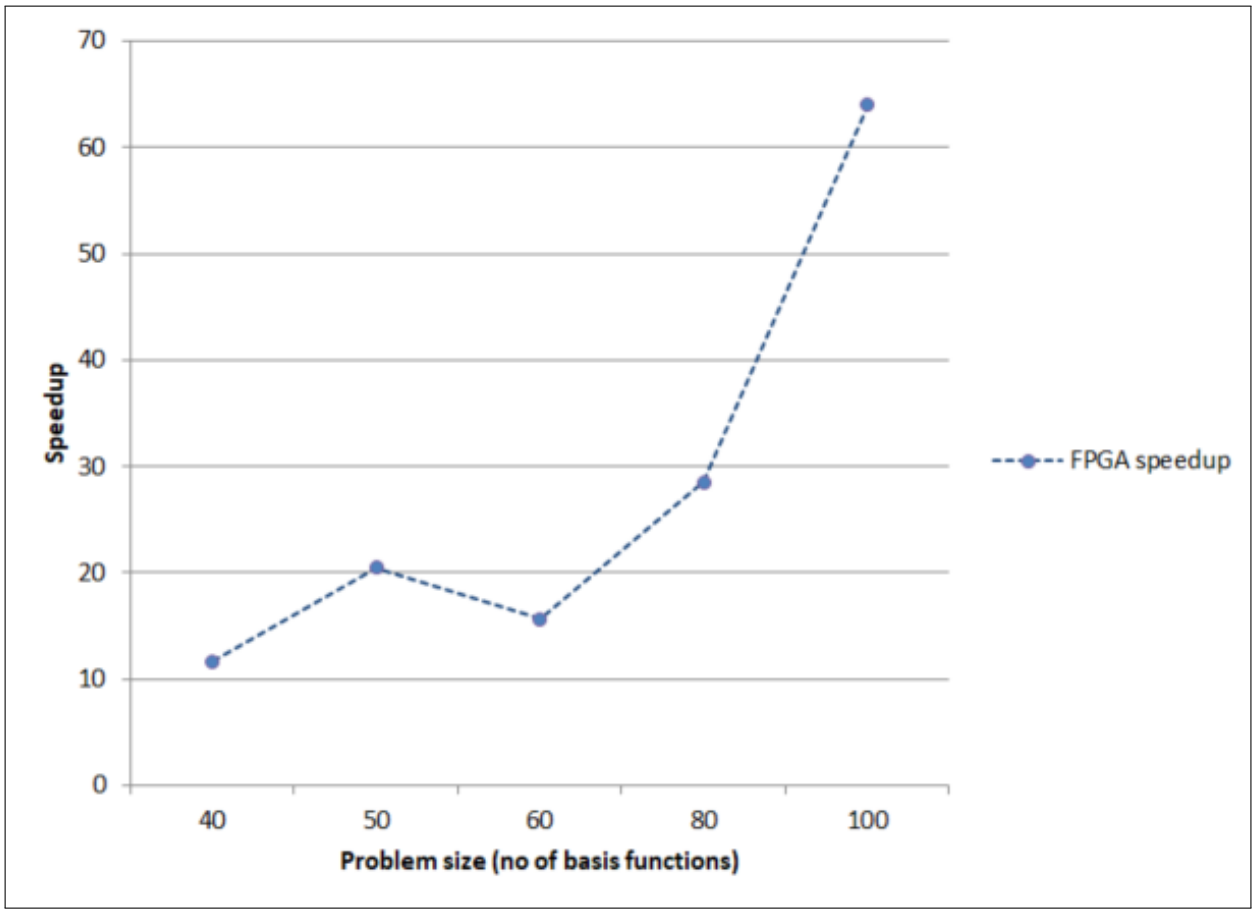

Figure 5.9: Speedup on Arria® 10 FPGA vs. Intel Xeon CPU

5.7.3 Performance on Altera Stratix® V vs. Intel Arria® 10 FPGA

The line graph of Figure 5.10 indicates the execution times of the kernel with increasing problem size on Altera Stratix® V FPGA board vs. Intel Arria® 10 FPGA board vs. Intel Xeon CPU for problem sizes varying from 50 to 100. Column graph shows the comparison of the execution times between the Altera Stratix® V and Intel Arria® 10 FPGAs.

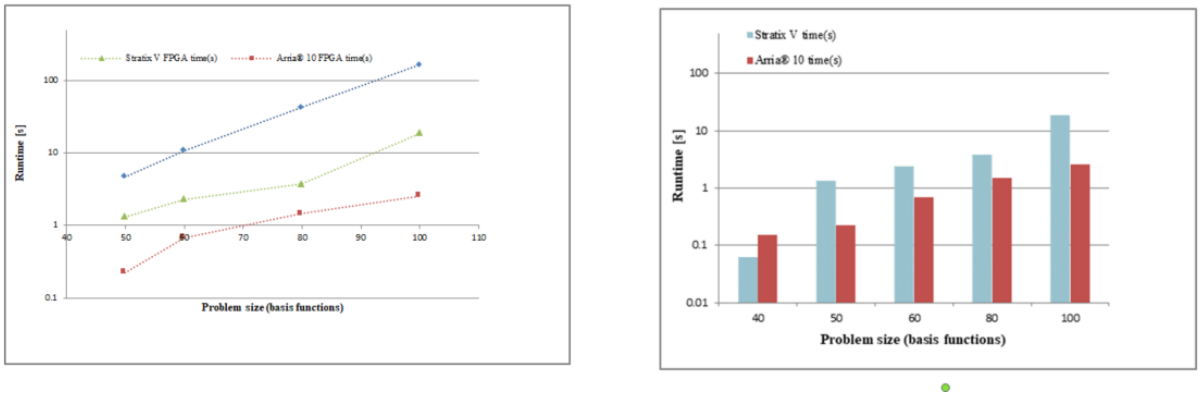

Figure 5.10: Runtimes on FPGAs (Stratix® V & Arria® 10) vs. CPU (log scale).

5.7.4 Performance on a Nvidia Tesla® P100 GPU

Figure 5.11 compares execution times of the kernel with increasing problem size on Nvidia Tesla® P100 GPU vs. Intel Xeon® E5520 CPU. Figure 5.12 indicates the speedup achieved for on the GPU. The kernel was implemented using 64 workers.

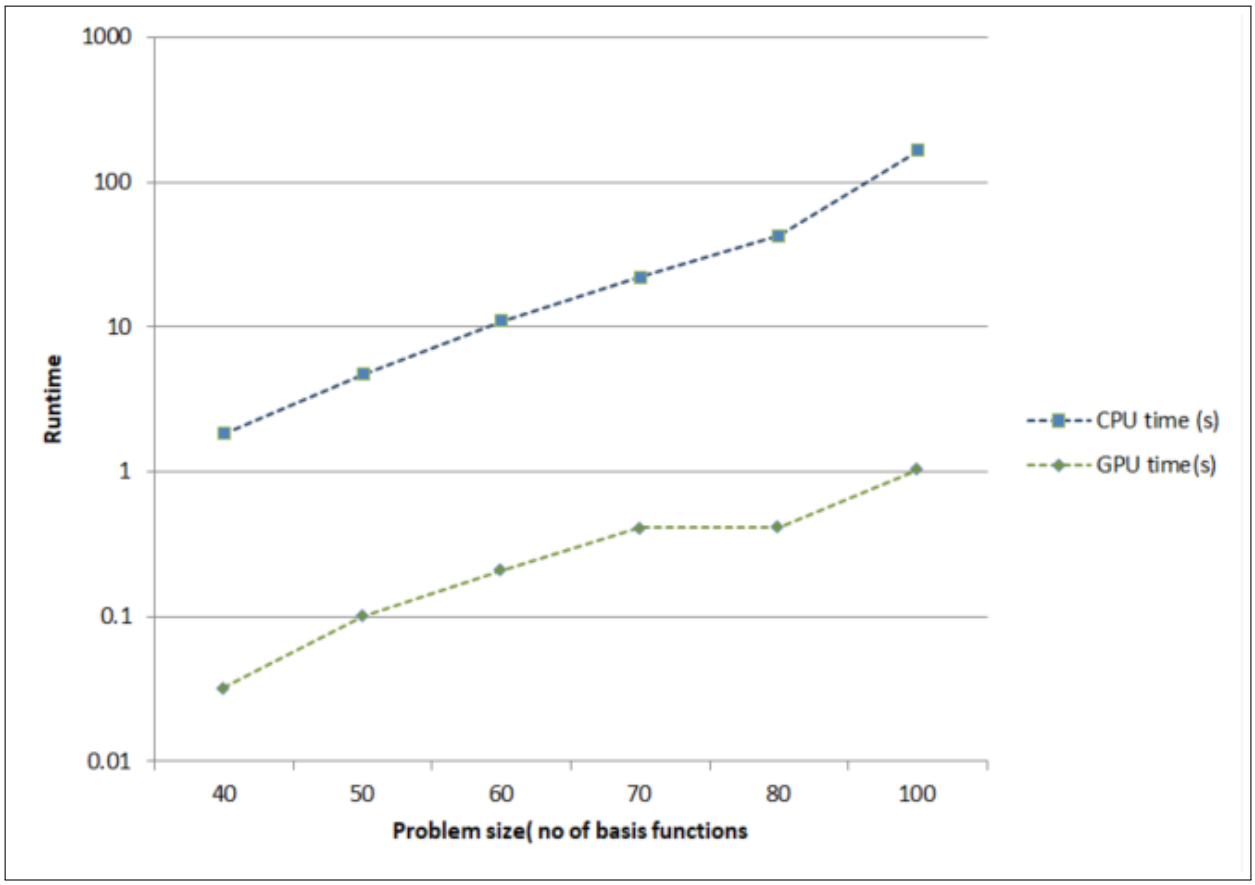

Figure 5.11: Runtime on Nvidia Tesla® GPU vs. Intel Xeon® CPU (logarithmic scale).

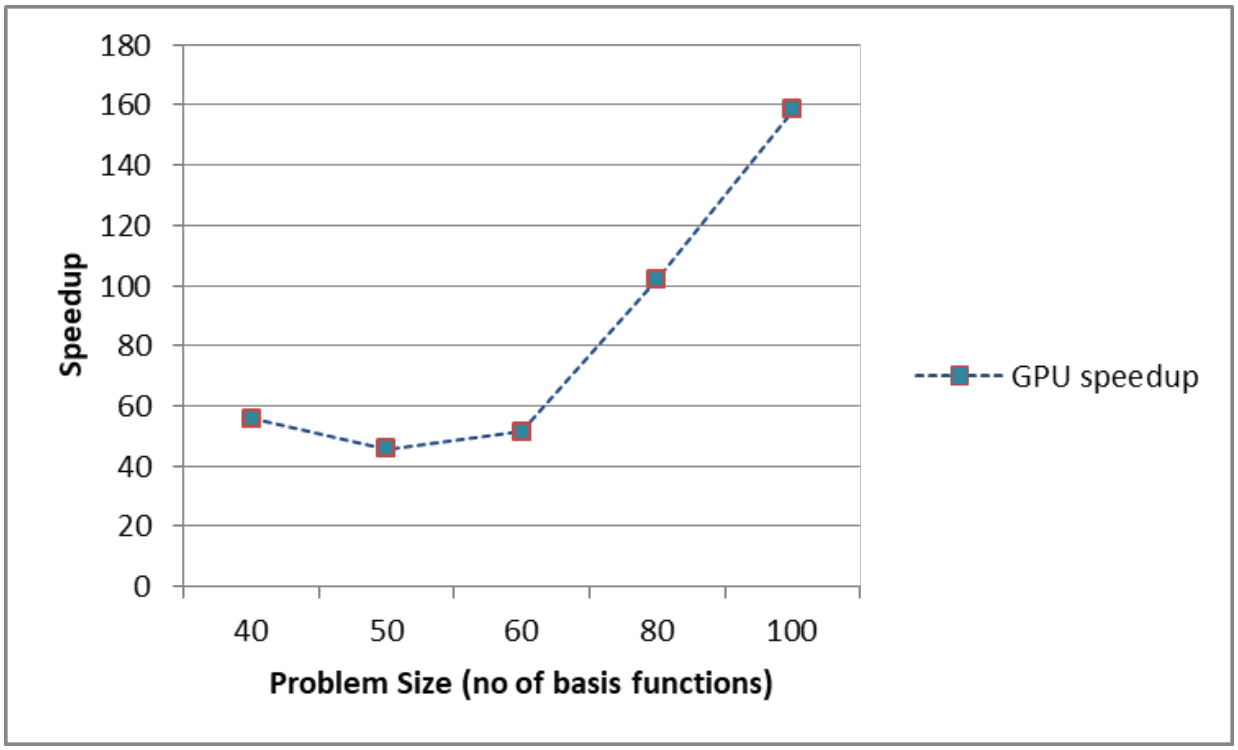

Figure 5.12: Speedup on Nvidia Tesla® GPU vs. Intel Xeon CPU.

5.7.5 Performance on Nvidia Tesla® GPU vs. Intel Arria® 10 FPGA

Figure 5.13 indicates the runtime achieved on a Nvidia Tesla® GPU compared to the Intel Arria® 10 FPGA for different problem sizes. Figure 5.14 presents speedup on the GPU compared to the Arria® 10 FPGA for different problem sizes. For this problem we get a better speedup on the GPU than on the FPGA.

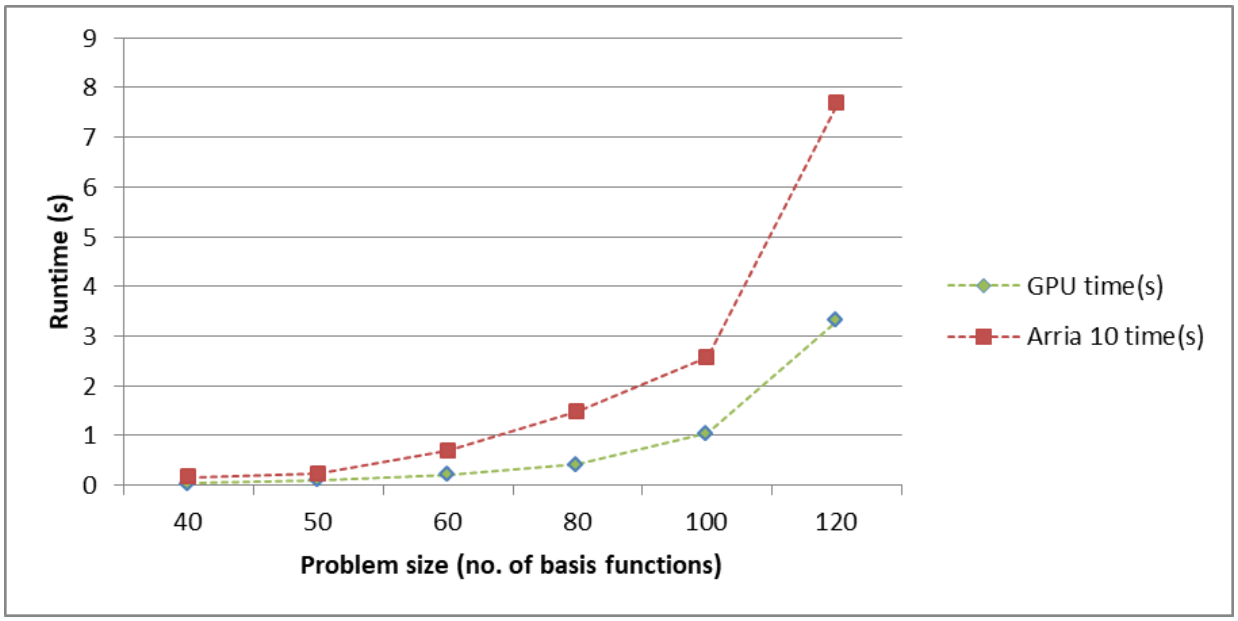

Figure 5.13: Runtime on Nvidia GPU vs. Intel Arria® 10 FPGA.

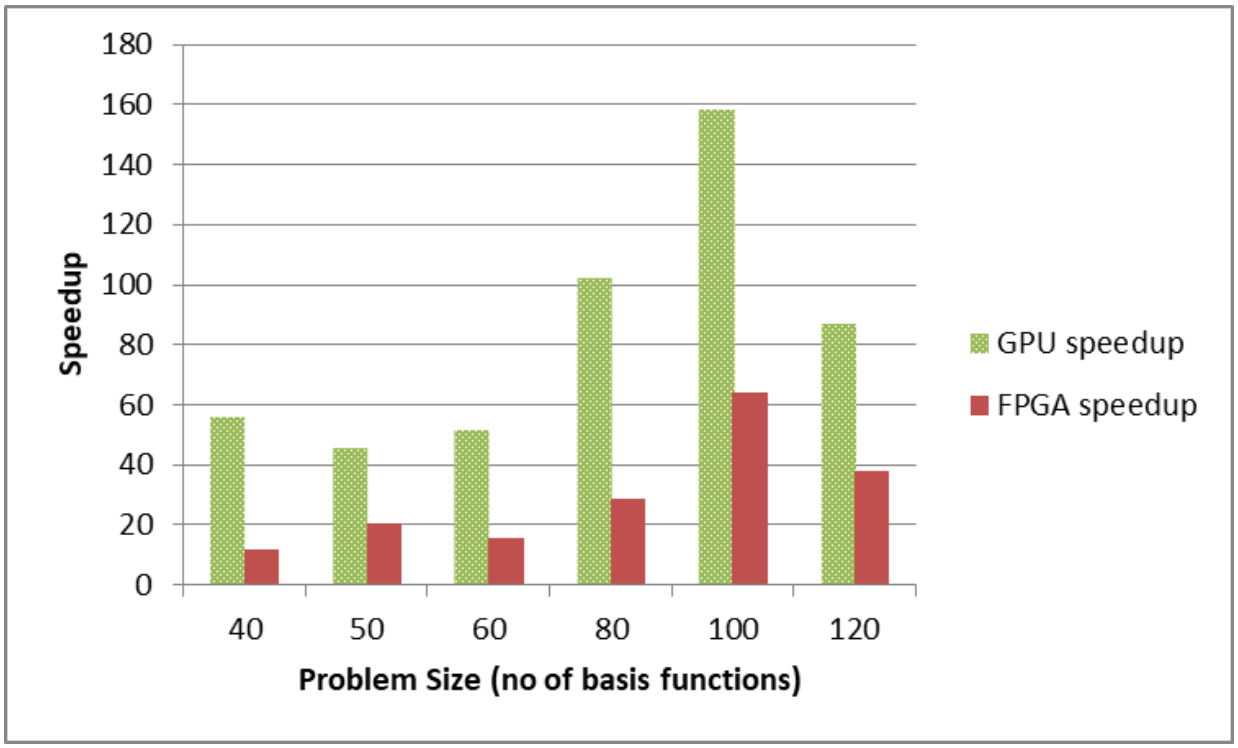

Figure 5.14: Speedup obtained on Nvidia GPU vs. Arria® 10 FPGA.

Table 5.3: Runtimes and Speedup achieved on FPGAs (Stratix® V and Arria® 10) vs. Nvidia GPU vs. CPU.

Problem size [basis functions]	CPU time [s]	GPU time [s]	Stratix® V time [s]	Arria® 10 time [s]	GPU speedup [1]	Stratix® V speedup [1]	Arria® 10 speedup [1]
40	1.806 348	0.032 363		0.154 401	55.815 221		11.699 069
50	4.638 461	0.101 401	1.300 248	0.226 698	45.743 740	3.567 366	20.460 970
60	10.768 384	0.209 356	2.310 464	0.686 642	51.435 755	4.660 702	15.682 676
80	42.272 370	0.414 016	3.765 446	1.480 493	102.103 228	11.226 391	28.552 901
100	164.550 018	1.039 194	18.759 409	2.572 125	158.343 888	8.771 599	63.974 347

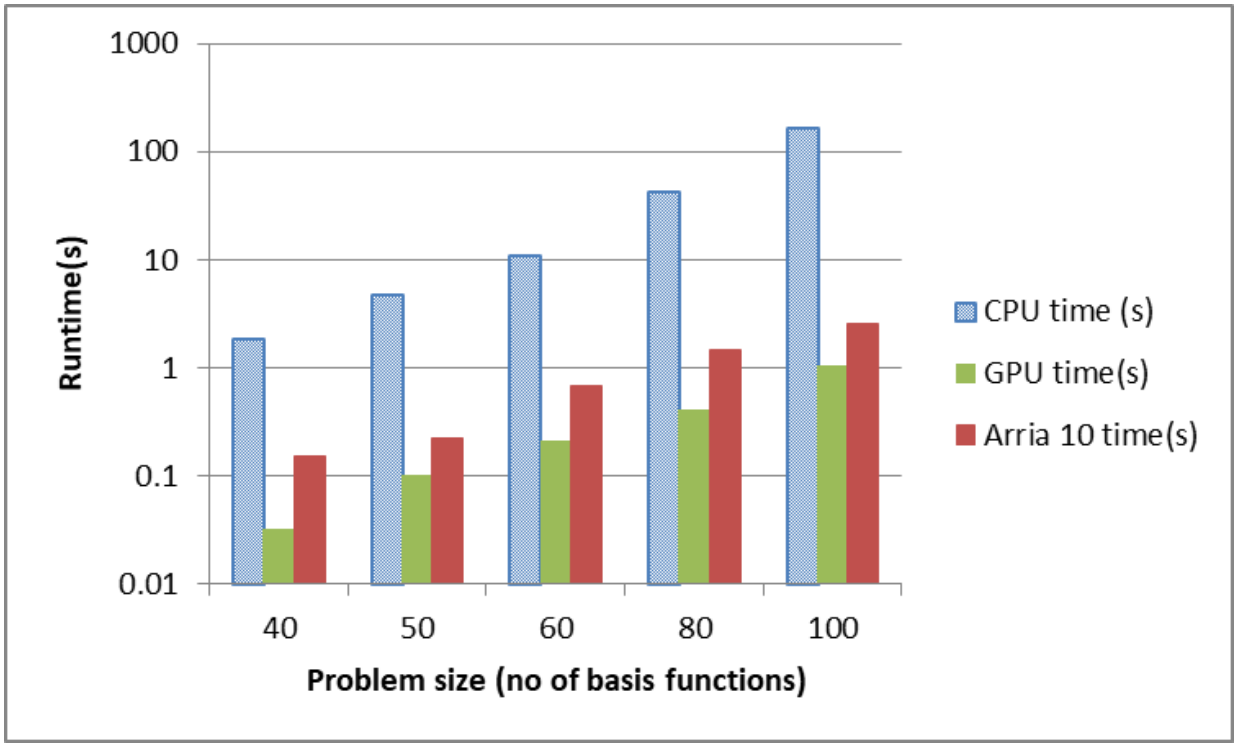

Figure 5.15: Runtime on Nvidia GPU vs. Intel Arria® 10 FPGA vs. Xeon® CPU

5.8 Evaluating two Workflows: porting GAMESS-SIMGMS Computational Chemistry Kernel to FPGA vs. GPU Architectures using OpenARC Compiler

Porting the GAMESS-SIMGMS computational chemistry kernel is a long term development process, where the completion time may range from 3 to 6 months for each workflow. Compared to the classroom study we discussed in chapter 3, in this scenario, workflow steps need to encapsulate multiple atomic steps. Having high level steps of the problem solving steps delivers clarity to the overall picture when we represent a complex problem.

In this problem our goal is to evaluate the reusability of the OpenARC compiler using the workflows – 1) porting GAMESS-SIMGMS to FPGAs, and 2) porting GAMESS-SIMGMS to GPUs.

5.8.1 Workflow Representation

We recorded the steps of the workflows and the time we spent on each task throughout the porting process and analyzed where the two workflows differed from each other. We use colored arrows to distinguish common paths in the two workflows.

5.8.2 Workflow 1: Porting GAMESS-SIMGMS to Altera Stratix® V FPGA

Table 5.4: Porting the kernel to Stratix® V FPGA

Step	Task	Time (weeks : days)
	Start	
1	Setup GAMESS code on the HPC system with FPGA.	3:0
2	Setup the HPC system to support GAMESS-SIMINT module (Map libraries with environment. This includes miner code changes.)	2:3
3	Convert GAMESS-SIMINT module to pure C.	12:0
4	Debug code and verify.	3:0
5	Setup OpenARC compiler with FPGA environment settings.	1:0
6	Test OpenARC compiler on available benchmarks.	0:4
7	Configure GAMESS-SIMINT setup for OpenARC compiler for FPGA environment. This includes getting familiar with the compiler using available benchmark.	4:0
8	Add OpenACC directives.	0:3
9	Convert to OpenCL and compile and Generate FPGA executable using OpenARC.	1:0
	End	
	Total Time	27:3

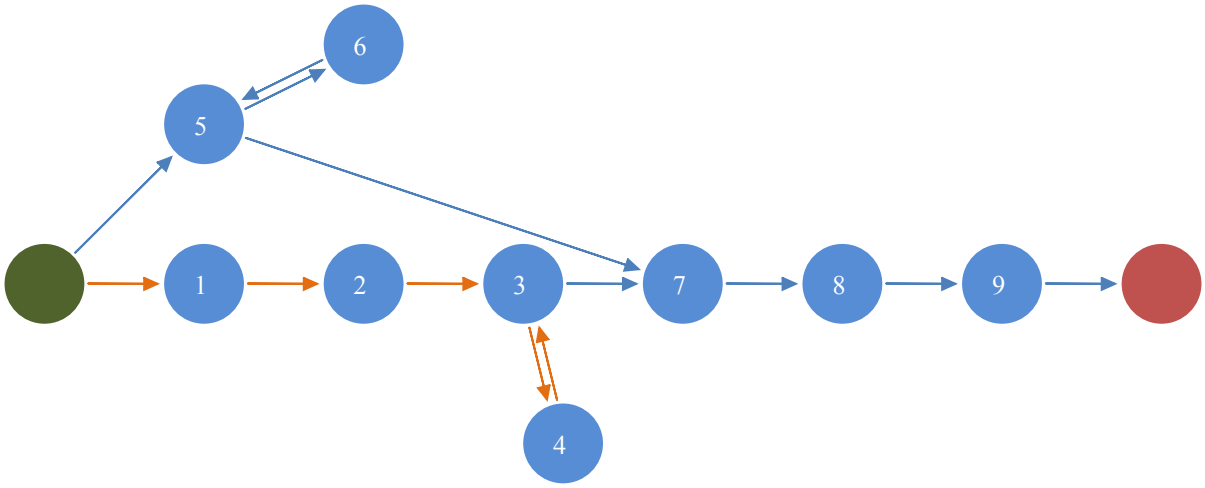

Figure 5.16: Workflow diagram of porting SIMINT kernel to Stratix V FPGA

5.8.3 Workflow 2: Porting GAMES-SIMGMS Kernel to GPU using OpenARC

Table 5.5: Porting the kernel to Nvidia Tesla® GPU

Step	Task	Time (weeks : days)
	Start	
1	Setup GAMESS code on the HPC system with GPU	2:3
2	Setup the GPU enabled HPC system to support GAMESS-SIMINT module (Map libraries with environment. This includes miner code changes.)	4:0
3	Convert GAMESS-SIMGMS module into pure C code and redirect output.	12:0
4	Debug code and verify.	3:0
5	Setup OpenARC compiler with GPU environment settings.	1:0
6	Test OpenARC compiler on available benchmarks.	1:0
7	Configure GAMESS-SIMINT setup for OpenARC compiler for GPU based system.	1:0
8	Add OpenACC directives.	0:3
9	Compile the code using OpenARC and generate GPU executable.	0:3
	End	
	Total Time	25:2

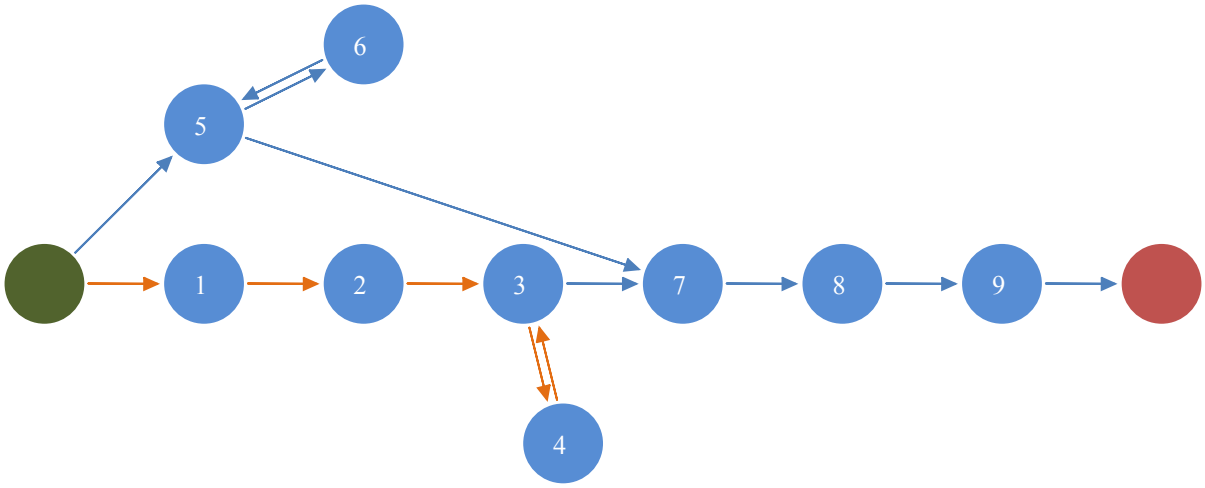

Figure 5.17: Workflow diagram of porting SIMINT kernel to Nvidia Tesla® GPU

5.8.4 Analysis

We compared the workflow diagrams of the porting the GAMESS-SIMGMS kernel to Altera Stratix® V FPGA with the porting of GAMESS-SIMGMS kernel to Nvidia Tesla® GPU using the OpenARC compiler. We look for the deviation of the branches in the two diagrams. Our subject of interest is in evaluating the portability of the code when OpenARC compiler is used. These different paths were assigned evaluation metrics for the predefined criteria in order to obtain a productivity measurement for evaluation.

Overall Development Time

Development time can be negatively affected using OpenARC by the spending more time on a given step. We consider total time taking all the steps into account.

Portability

The additional work involved to make the same application work on a different platform makes the application less portable. If the configuration time on the compared platform is high, the portability is less. In this scenario the portability of the compiler can be negatively

affected by the number of additional lines of codes for the source to source translation in the following steps:

5- Setup OpenARC compiler with environment settings.

6- Test OpenARC compiler on available benchmarks.

7- Configure GAMESS-SIMGMS setup for OpenARC compiler for environment.

8- Convert to OpenCL and compile and Generate executable.

Code Reusability

The following steps affect code reusability negatively in this particular problem:

2- Setup the HPC system to support GAMESS-SIMGMS module with minor code change

3- Convert GAMESS-SIMINT module to pure C

8- Add OpenACC directives

We can show the comparison using SLOC count and the time spent on the tasks for each workflow for this criterion.

Productivity Vector

The presented vector of measurement for evaluation takes development time of obtaining the end-to-end results, code portability and code reusability into consideration.

$$[\textit{Development-time Portability Maintainability}]$$

Evaluating the Workflow of Porting the Kernel to FPGA against Porting the Kernel to GPU

Since the two workflows complement each other in each task that we carefully abstracted, we can compare each step against its equivalent for all.

Development-time – Total time is, 27 weeks and 3 days for the FPGA workflow, against the total time of 25 weeks and 2 days for the GPU workflow, which gives us $-1W_d$ in the

vector.

Portability – In order to get a quantifiable value for portability, we consider the additional time spent on the tasks 5,6,7,8 in workflow 1 vs. workflow 2.For Tasks 5and 8 time spent was the same for both workflows. Total time difference is 6 weeks for workflow 1 against 3 weeks and 3 days for workflow 2.

The time difference gives us a negative value$-1W_p$ for the FPGA workflow compared to the GPU workflow.

Code Reusability – There were 0 Additional lines of code used for porting the FPGA vs. porting the GPU in the steps of 2,3,8, except the SLOC count for the configuration, which is a negligible. We assign 0 as the value for reusability.

$$[\textit{Development-time Portability Reusability}]$$

$$[-1W_d \quad -1W_p \quad 0]$$

We assign weights for the vector values as $W_d = 2$ and $W_p = 1$.

$$[-2 \quad -1 \quad 0]$$

We observe that the development time is a negative value for the FPGA porting process in comparison to the GPU. The negative portability value suggests that the portability is a negative value for the FPGA workflow, which indicates that it is harder to port the code to FPGA compared to porting to the GPU. Depending on the weights assigned for portability, and development metrics, the most productive workflow might be porting the code to GPUs compared to FPGAs. But the compiler enables the user to reuse the code from one workflow to the other. The workflow 1 preceded the workflow 2 and the user was familiar with the compiler and the process. This study would need to be repeated with two groups of programmers with similar backgrounds, one group porting to FPGAs and the other porting to GPUs, to get a reliable comparison.

5.9 Programming Effort Analysis: Porting Computational Chemistry to FPGAs using OpenARC vs. Porting by Hand

Table 5.6: SLOC comparison

Type of File/s	Source lines of code (SLOC)
FPGA accelerator code – Without OpenARC	
C++	495
OpenCL	56
FPGA accelerator code – With OpenARC	
C++ & OpenACC	338
Translated module size	
C, C++	5423

Looking at the SLOC table we observe that the code translation has the highest SLOC count, with 5423 lines of code. The SLOC count for writing accelerator code with OpenARC is 338 lines of code to 551 lines of code. If we consider the number of lines alone, the developer writes 24% more lines of code to write OpenCL than using OpenACC. We also need to consider that using OpenARC instead of manually writing the code using OpenCL also eliminated the need to understand the underlying hardware in detail.

The translation step was required since the SIMGMS module was written in C++, C and some Fortran but the compiler needs a C program as input. The chart in Figure 5.18 shows to the times spent on different tasks in the first workflow, porting the computational chemistry kernel to Altera Stratix® V FPGA with the OpenARC compiler.

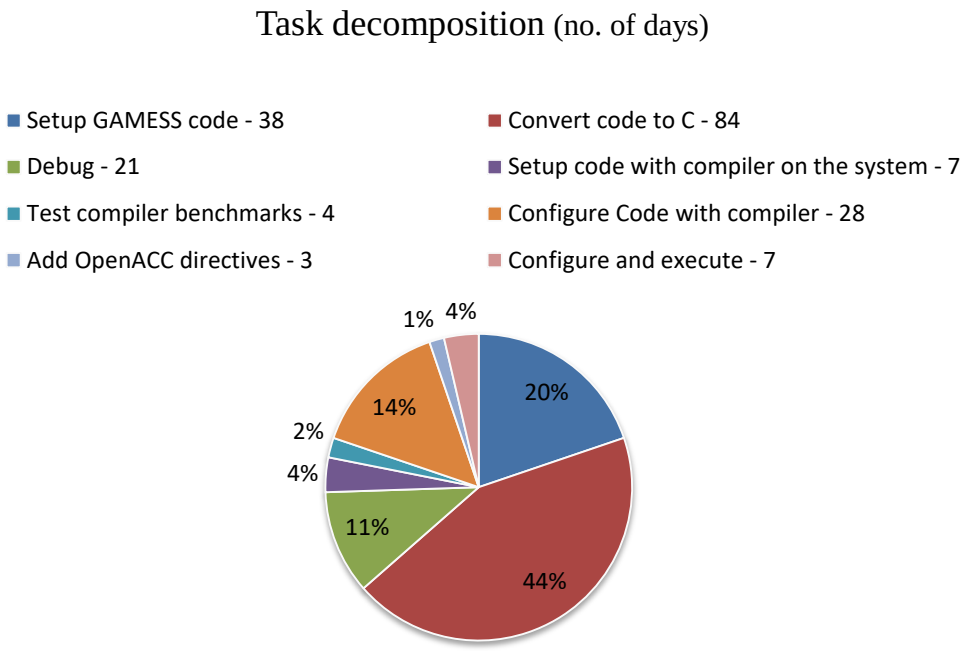

Figure 5.18: Task Decomposition of Workflow 1

Figure 5.19 compares the percentages of time spent on workflow 1 if there wasn't a code conversion involved with the actual workflow. This is an example of the time spent on a typical scientific programming workflow on programming vs. configuration tasks on the new architecture.

The analysis below is an example to show how a typical porting process in the scientific domain differs from mainstream software development.

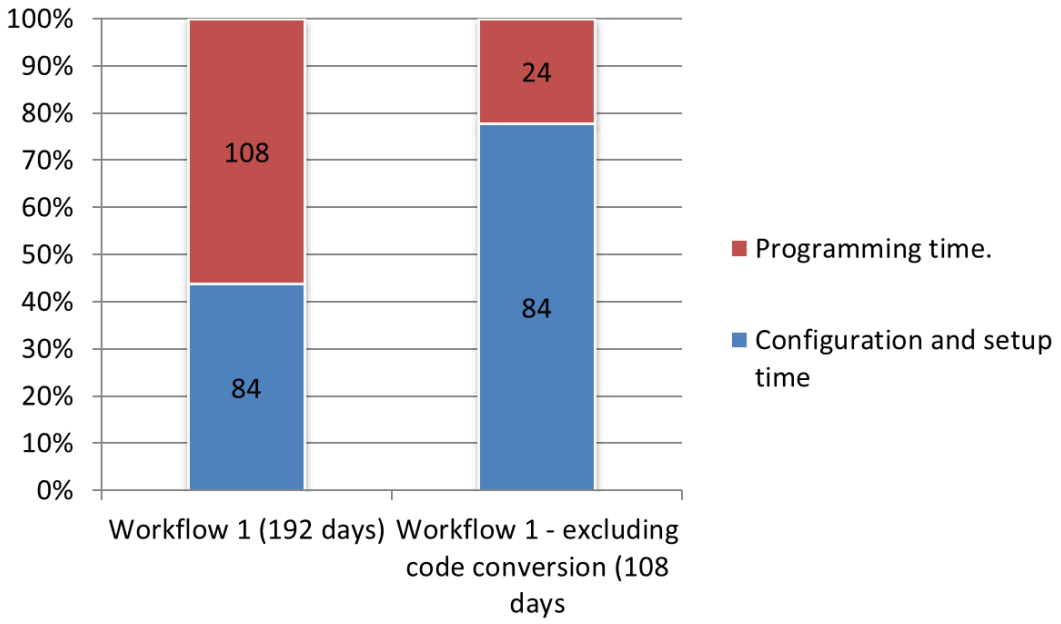

Figure 5.19: Programming and Setup time distribution

In the first scenario, 84 days or 44% of time of the development life time is spent on setup and configurations mainly due to variable programming environment and 108 days or 56% of the time is spent on programming on the workflow 1. The second scenario we exclude the code conversion step. If the code was already written in C++ language, 84 days or 77% of the time would be taken by configurations and programming environment setup and only 24 days or 22% of the time is spent on programming.

Chapter 6: Conclusions and Future Work

In order to get the best possible performance from new and state-of-the-art accelerators that become available, porting new and existing scientific applications to evolving supercomputer environments, is an inevitable task in the scientific application development life-cycle. The HPC community has an inherent need to optimize the porting process to get the most productivity out of available resources. Well-defined workflows are a useful analysis tool for studying the scientific application life cycle.

In chapter 3 of this thesis, we presented a graphical scheme as a means to compare workflows and presented a measure for productivity using workflow-specific criteria. Our evaluation criteria can be modified for different objectives and constraints and the productivity vector can be used with few variations. For our preliminary case studies we selected efficiency, accuracy, portability and maintainability as the criteria of evaluation. These criteria have to be selected related to the case study we intend to evaluate. By changing the criteria of the productivity vector, the methodology can be adapted for a different problem domain. Focusing on the end-to-end solution of the problem is more relevant when evaluating the productivity. A process is productive when it meets the requirements of the user accurately and completely and efficiently in terms of time and resources. It is beneficial to specify requirements according to the specific problem. The productivity vector should evaluate the end-to-end productivity related to overall time and resources rather than technical attributes needed to achieve the immediate execution performance. In the third chapter, our approach is to compare novice user productivity to that of an expert user, using the expert workflow as a baseline. Productivity depends on the user perception of the problem and on knowledge of the methods of solution. Novice users generally prefer graphical user interfaces (GUIs) and find them easier to use. In our second case study using the Lighthouse tool, we found that Lighthouse does not support the most time-consuming task of pre-processing the input files. Since the Lighthouse developers

state that they have tested their tool with matrices from Matrix Market, it should be easy for them to extend their tool to support conversion from the Matrix Market (.mtx) and Harwell-Boeing formats used by Matrix Market to the PETSc binary format. The runtime efficiency and accuracy criteria are also not well-supported by the Lighthouse tool, since it gives the novice user no help in choosing an appropriate solver and preconditioner for the problem at hand, or even in determining whether or not convergence to a solution occurred. The Lighthouse tool is very helpful to novice users in generating working code that makes proper use of the PETSc routines. What is lacking is help with options for running the code that select an appropriate solver and preconditioner. The developers of Lighthouse state that they have tested their tool with large numbers of sparse matrices from Matrix Market and used the Lighthouse tool to successfully solve them. However, the developers of Lighthouse are also expert users of PETSc and have the expert knowledge to specify the appropriate options. We presented our results[23] at a conference that was also attended by one of the Lighthouse developers, and they have proceeded to implement some of our suggestions.

New hardware platforms that might provide better speed-up to applications may fail to be adopted by the community, if the programmability is poor on those architectures. In chapter 4 of the thesis, we compared two common low-level programming models, namely, CUDA and OpenCL, with the directive based OpenACC programming model using known code complexity measures.[18] Cyclomatic complexity based measures have been found to be indicators of likelihood of errors and difficulty of maintaining a software project by software engineering researchers [10], [11]. Software engineering researchers have also shown that SLOC counts account for only 30-35% of code development and maintenance costs. We found that CUDA and OpenCL programs do generally have much higher SLOC counts than the corresponding serial and OpenACC versions. So far the high performance computing community has mainly used SLOC counts as the quantitative metric for code complexity, as in [6]. However, none of the metrics appear to capture the widely held belief that code complexity is higher for CUDA and OpenCL programs than for OpenACC

implementations. Portability and maintainability are not black and white - that is, a code is not either portable or not portable, but more or less portable. A direct quantitative metric of how portable and maintainable a GPU or CPU+GPU code is has not yet been devised. It should be possible to structure a code so that the less portable portions of it are isolated and easily identified. Another possibility is to use libraries that implement commonly used functions efficiently using low-level code so that the application code can be written at a higher level of abstraction.

We ported a GAMESS computational chemistry kernel to an Altera Stratix® V FPGA, [13] an Intel Arria® 10 FPGA and a Nvidia Tesla® P100 GPU using the OpenARC compiler. OpenARC translates OpenACC directives into optimized OpenCL for FPGAs and optimized CUDA for GPUs. We compared two workflows – 1) porting a computational chemistry kernel to Nallatech FPGAs and 2) porting the same computational kernel to Nvidia Tesla® GPUs using the OpenARC compiler. We evaluated both workflows using the predefined evaluation criteria of development time, portability and code reusability in the form of productivity matrix vectors. We observe that using OpenARC the code is 100% reusable on two architectures, making the combination of OpenACC and OpenARC compiler a highly portable option for programming FPGAs. By analyzing the scientific programming process of porting the computational chemistry kernel to Stratix® FPGAs we observe that the 44% of the time was spent on configuring the programming environment settings. This number even went to 77% of the time when we excluded the code conversion time - which turned the problem into a porting only task, which may be a common scenario if the application needs to be only programmed for acceleration. These observations illustrate that the scientific programming lifecycle differs from the typical software development life cycle mainly due to the fact that the programing environment itself becomes a variable in many scientific computing tasks.

We achieve up to 9.5X speedup on the Stratix® V and up to 64X speedup on newer Arria® 10 FPGA. We observed the best speedup for the kernel on the Nvidia Tesla® GPU which was 160 times that of the Intel Xeon® CPU. We presented our results at the

2018 American Chemical Society meeting. Our presentation was attended by the author of the SIMINT code who thinks that a pipelined implementation of a recursive version of the code could give better results on the FPGAs. This future work will be carried out in collaboration with the SIMINT author.

The majority of the time for the FPGA porting using OpenARC was spent in getting familiar with the GAMESS code and getting it to run on our cluster, and on converting the code from C++ to pure C by hand. The first effort would be much less if done by a domain scientist familiar with the GAMESS code. The second effort would be much less using a source-to-source optimizing compiler that can take Fortran and C++ as input. The OpenARC project is moving towards using the LLVM infrastructure, the most recent version of which supports C++ and Fortran in addition to C. When this language support becomes available, we plan to re-evaluate the productivity of using OpenARC, or its successor, for porting more GAMESS kernels to FPGAs.

Another aspect of exascale computing is the power bound of 20-30 Megawatts that will be imposed on the exascale machine. It will become important to use the accelerator technology that yields not only the best performance in terms of runtime, but also the best energy efficiency. GPUs are more energy-efficient than CPUs for codes that run well on them. Because they are specialized to the code that is running on them, FPGAs are potentially the most energy-efficient option for codes that have the right characteristics for porting to FPGAs. Our future work will involve evaluating energy-efficiency as part of overall productivity.

Quantum computing is being explored as a possible option for computational chemistry. Because a quantum computer can effectively represent an exponential number of states in a polynomial number of qubits, until a measurement is taken, quantum computing may become viable for carrying out computations that are currently too expensive on classical computers. The current research uses a hybrid classical-quantum approach such as the Variational Quantum Eigensolver (VQE) method. Until now, quantum computers have been programmed at a very low level using a gate language specific to the underly-

ing quantum hardware. The XACC project at Oak Ridge National Laboratory serves as a bridge between high level expression of quantum algorithms in C++ and Python and hardware-specific code, using an LLVM intermediate representation to do transformations and optimizations [16]. Both the XACC and GAMESS developers are interested in exploring the integration of GAMESS as an XACC plug-in to test the productivity of using quantum computing to accelerate computationally intensive calculations, and we plan to collaborate with them on this future work.

References

[1] Matrixmarket. `http://math.nist.gov/MatrixMarket/`.

[2] Petsc user manual, revision 3.6. `http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf`, 2015.

[3] Mayank Daga, Zachary S Tschirhart, and Chip Freitag. Exploring parallel programming models for heterogeneous computing systems. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 98–107. IEEE, 2015.

[4] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[5] H Carter Edwards and Christian R Trott. Kokkos: Enabling performance portability across manycore architectures. In *Extreme Scaling Workshop (XSW), 2013*, pages 18–24. IEEE, 2013.

[6] Ronak Etemadpour, Matthew Bomhoff, Eric Lyons, Paul Murray, and Angus Forbes. Designing and evaluating scientific workflows for big data interactions. In *Big Data Visual Analytics (BDVA), 2015*, pages 1–8. IEEE, 2015.

[7] Stuart Faulk, Eugene Loh, Michael L Van De Vanter, Susan Squires, and Lawrence G Votta. Scientific computing's productivity gridlock: How software engineering can help. *Computing in science & engineering*, 11(6):30–39, 2009.

[8] M. S. Gordon and M. W. Schmidt. *Advances in electronic structure theory: GAMESS a decade later*, pages 1167–1189. Elsevier, Amsterdam, 2005.

[9] Lorin Hochstein, Victor R Basili, Uzi Vishkin, and John Gilbert. A pilot study to compare programming effort for two parallel programming models. *Journal of Systems and Software*, 81(11):1920–1930, 2008.

[10] Lorin Hochstein, Victor R. Basili, Marvin V. Zelkowitz, Jeffrey K. Hollingsworth, and Jeff Carver. Combining self-reported and automatic data to improve programming effort measurement. *SIGSOFT Softw. Eng. Notes*, 30(5):356–365, September 2005.

[11] Watts Humphrey. The personal software process (psp). Technical Report CMU/SEI-2000-TR-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2000.

[12] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[13] WK Umayanganie Klaassen and Shirley V Moore. Porting a gamess computational chemistry kernel to fpgas. In *Heterogeneous High-performance Reconfigurable Computing (H2RC'17) (SC17), 2017 Third International Workshop on on*, 2017.

[14] Seyong Lee and Jeffrey Vetter. Openarc: Extensible openacc compiler framework for directive-based accelerator programming study. In *WACCPD: Workshop on Accelerator Programming Using Directives in Conjunction with SC'14*, November 2014.

[15] Seyong Lee and Jeffrey S Vetter. Openarc: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *HPDC '14: Proceedings of the ACM Symposium on High-Performance Parallel and Distributed Computing, Short Paper*, 2014.

[16] Alexander J McCaskey. Xacc-extreme-scale accelerator programming framework. Technical report, Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States), 2016.

[17] Pate Motter, Kanika Sood, Elizabeth Jessup, and Boyana Norris. Lighthouse: an automated solver selection tool. In *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, pages 16–24. ACM, 2015.

[18] WK Umayanganie Munipala and Shirley V Moore. Code complexity versus performance for gpu-accelerated scientific applications. In *Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE), 2016 Fourth International Workshop on*, pages 50–50. IEEE, 2016.

[19] Boyana Norris, Sa-Lin Bernstein, Ramya Nair, and Elizabeth Jessup. Lighthouse: A user-centered web service for linear algebra software. *arXiv preprint arXiv:1408.1363*, 2014.

[20] Robert W Numrich, Lorin Hochstein, and Victor R Basili. A metric space for productivity measurement in software development. In *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 13–16. ACM, 2005.

[21] Sarat Sreepathi, M. L. Grodowitz, Robert Lim, Philip Taffet, Philip C. Roth, Jeremy Meredith, Seyong Lee, Dong Li, and Jeffrey Vetter. Application characterization using oxbow toolkit and pads infrastructure. In *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC '14, pages 55–63, Piscataway, NJ, USA, 2014. IEEE Press.

[22] Thomas Sterling and Chirag Dekate. Productivity in high-performance computing. In *Advances in Computers*, volume 72, pages 101–134. Elsevier, 2008.

[23] W. K. Umayanganie Munipala and Shirley V. Moore. An evaluation framework for scientific programming productivity: Position paper. In *Proceedings of the International Workshop on Software Engineering for Science*, SE4Science '16, pages 27–30, New York, NY, USA, 2016. ACM.

[24] Maciej Wielgosz, Ernest Jamro, and Kazimierz Wiatr. Hardware implementation of the exponent based computational core for an exchange-correlation potential matrix generation. In *International Conference on Parallel Processing and Applied Mathematics*, pages 115–124. Springer, 2009.

[25] Maciej Wielgosz, Grzegorz Mazur, Marcin Makowski, Ernest Jamro, Pawel Russek, and Kazimierz Wiatr. Analysis of the basic implementation aspects of hardware accelerated density functional theory calculations. *Computing and Informatics*, 29(6):989–1000, 2012.

[26] C. Wood, S. Sane, D. Ellsworth, A. Gimenez, K. Huck, T. Gamblin, and A. Malony. A scalable observation system for introspection and in situ analytics. In *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*, pages 42–49, Nov 2016.

[27] Min Zhang and Lorin Hochstein. Fitting a workflow model to captured development data. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 179–190. IEEE Computer Society, 2009.

[28] Min Zhang and Lorin Hochstein. Fitting a workflow model to captured development data. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 179–190. IEEE Computer Society, 2009.

Appendix 1

LAPACK Porting Guide for STAMPEDE

These guidelines are intended to help C and C++ application developers use LAPACK routines in a way that results in efficient, portable, and maintainable code for high performance computing systems.

1. Efficiency: In order for your code to be as efficient as possible, you should use the version of LAPACK that is tuned for your computer system. You should install and use the Netlib reference version only as a last resort if no tuned version is available. Look in your system documentation to determine what library to use. The library may not be called LAPACK even though it includes LAPACK. For example, on non-Cray Intel systems, the Intel Math Kernel Library (MKL) is usually installed and includes LAPACK. Likewise, on non-Cray AMD systems, the library is ACML. On IBM systems, the library is ESSL. On Cray systems, it is Cray libsci. Although the routine names and prototypes are a de facto standard and are the same across implementations, the commands for compiling and linking are different. Please refer to your system library documentation to determine the correct compile and link commands.

2. Portability:

   2.1 For maximum efficiency and portability for a C or C++ code that uses LAPACK, you should use the extended LAPACK (lapacke) interface if it is available. The LAPACKE routine names start with LAPACKE, followed by the usual LAPACK routine name, for example LAPACKE_dgesv, for the LAPACK routine DGESV that solves a linear system (SV) for a general matrix (GE) in double precision (D). Go to `www.netlib.org/lapack/explore-html/files.html` and expand the LAPACKE

section for documentation. In the src directory, you will find the prototypes for each routine. In the example directory, you will find examples of how to use an LAPACKE routine in a CC++ program.

2.2 - For maximum portability, you should use portable LAPACK data types for arguments to LAPACKE routines - for example, lapack_int instead of int. See the LAPACKE example programs for examples of how to do this.

2.3 - Some of the vendors name their header files differently from the Netlib LAPACK reference version. For example, the MKL LAPACKE main header file is named mkl_lapacke.h instead of lapacke.h, so you will need to include mkl_lapacke.h instead of lapacke.h if you are using MKL. This will make your program slightly less portable, so you should document this vendor-specific change.

3. Correctness:

   3.1 - For correctness, you should use the LAPACKE header files that come with the LAPACK implementation you are using, rather than downloading the Netlib reference version header files. Use the appropriate -I flag if necessary so that the compiler can find the header files for the LAPACK version you are using.

   3.2 - When you call an LAPACKE routine, make sure that you specify correctly whether your matrix is stored in row-major or column-major order. For C programs, the natural ordering is row-major order. See the LAPACKE example programs for how to do this.

Appendix 2

Sparse Linear Solver Code Generated by Lighthouse

```
/ * Program usage:  mpiexec ex1 [-help] [all PETSc options] */

static char help[] = "Solves a linear system with KSP.\n\n";

/*T
   Main operation: Solve a linear system
   Input file format: PETSc binary format (matrix and rhs in the
      same file)
   Processor: 1 (sequential)
   Output format: PETSc binary format
T*/

#include <petscksp.h>

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **args)
{
  Vec             x, b;      /* approx solution, RHS, exact solution
     */
  Mat             A;        /* linear system matrix */
  KSP             ksp;          /* linear solver context */
  PetscViewer     fd,viewer;
  PetscErrorCode ierr;
  PetscInt        its;
  PetscMPIInt     size;
  char            file[2][PETSC_MAX_PATH_LEN];     /* input file
     name */
  PetscBool       flg;

  PetscInitialize(&argc,&args,(char *)0,help);
  ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
  if (size != 1) SETERRQ(PETSC_COMM_WORLD,1,"This is a uniprocessor
      example only!");

  /*
     Determine files from which we read the linear system (matrix
        and right-hand-side vector).
  */
  ierr = PetscOptionsGetString(PETSC_NULL,"-f",file[0],
     PETSC_MAX_PATH_LEN,&flg);CHKERRQ(ierr);
  if (!flg) {
    SETERRQ(PETSC_COMM_WORLD,1,"Must indicate binary file with the
       -f option");
  }
```

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   - - -
        Compute the matrix and right-hand-side vector that define
        the linear system, Ax = b.
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      - - - */

/*
     Open binary file.  Note that we use FILE_MODE_READ to
        indicate
     reading from this file.
*/
ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD,file[0],
   FILE_MODE_READ,&fd);CHKERRQ(ierr);

/*
   Load the matrix and vector; then destroy the viewer.
*/
ierr = MatCreate(PETSC_COMM_WORLD,&A);CHKERRQ(ierr);
ierr = MatSetFromOptions(A);CHKERRQ(ierr);
ierr = MatLoad(A,fd);CHKERRQ(ierr);

ierr = VecCreate(PETSC_COMM_WORLD,&b);CHKERRQ(ierr);
ierr = VecSetFromOptions(b);CHKERRQ(ierr);
ierr = VecLoad(b,fd);CHKERRQ(ierr);

ierr = VecDuplicate(b,&x);CHKERRQ(ierr);

ierr = PetscViewerDestroy(&fd);CHKERRQ(ierr);

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   - - -
              Create the linear solver and set various options
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      - - - */
/*
   Create linear solver context
*/
ierr = KSPCreate(PETSC_COMM_WORLD,&ksp);CHKERRQ(ierr);

/*
   Set operators. Here the matrix that defines the linear system
   also serves as the preconditioning matrix.
*/
ierr = KSPSetOperators(ksp,A,A,DIFFERENT_NONZERO_PATTERN);CHKERRQ
   (ierr);

/*
  Set runtime options, e.g.,
      -ksp_type <type> -pc_type <type> -ksp_monitor -ksp_rtol <
         rtol>
  These options will override those specified above as long as
  KSPSetFromOptions() is called _after_ any other customization
  routines.
*/
```

```
  ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     - - -
                        Solve the linear system
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        - - - */
  /*
     Solve linear system
  */
  ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     - - -
                        Check solution and clean up
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        - - - */
  /*
     Check the error
  */

  ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
  ierr = PetscPrintf(PETSC_COMM_WORLD,"Iterations %D\n",its);
     CHKERRQ(ierr);
  //VecView(x,PETSC_VIEWER_STDOUT_WORLD);

  ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD,"solution.petsc",
     FILE_MODE_WRITE,&viewer);CHKERRQ(ierr);
  ierr = VecView(x, viewer);CHKERRQ(ierr);
  ierr = PetscViewerDestroy(&viewer);CHKERRQ(ierr);

  /*
     Free work space.  All PETSc objects should be destroyed when
        they
     are no longer needed.
  */
  ierr = VecDestroy(&x);CHKERRQ(ierr);
  ierr = VecDestroy(&b);CHKERRQ(ierr);
  ierr = MatDestroy(&A);CHKERRQ(ierr);
  ierr = KSPDestroy(&ksp);CHKERRQ(ierr);

  /*
     Always call PetscFinalize() before exiting a program.  This
        routine
       - finalizes the PETSc libraries as well as MPI
       - provides summary and diagnostic information if certain
          runtime
         options are chosen (e.g., -log_summary).
  */

  ierr = PetscFinalize();
  return 0;
}
```

Modified Linear Solver

```
ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   - - -
                    Solve the linear system
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      - - - */
/*
   Solve linear system
*/
ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   - - -
                    Check solution and clean up
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      - - - */
/*
   Check the error
*/

ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_WORLD,"Iterations %D\n",its);
   CHKERRQ(ierr);
//VecView(x,PETSC_VIEWER_STDOUT_WORLD);

ierr = PetscViewerASCIIOpen(PETSC_COMM_WORLD,"output/solution.
   petsc",&viewer);CHKERRQ(ierr);
ierr = VecView(x, viewer);CHKERRQ(ierr);
ierr = PetscViewerDestroy(&viewer);CHKERRQ(ierr);

/*
   Free work space.  All PETSc objects should be destroyed when
      they
   are no longer needed.
*/
ierr = VecDestroy(&x);CHKERRQ(ierr);
ierr = VecDestroy(&b);CHKERRQ(ierr);
ierr = MatDestroy(&A);CHKERRQ(ierr);
ierr = KSPDestroy(&ksp);CHKERRQ(ierr);

/*
   Always call PetscFinalize() before exiting a program.  This
      routine
     - finalizes the PETSc libraries as well as MPI
     - provides summary and diagnostic information if certain
        runtime
       options are chosen (e.g., -log_summary).
*/

ierr = PetscFinalize();
return 0;
```

```
}
```

Vita

W.K. Umayanganie Klaassen is a Ph.D. candidate in the Computational Science Program at the University of Texas at El Paso. She has more than five years of research experience in the area of performance and productivity analysis in high performance computing. She holds a MSc in computational Science from the University of Texas at El Paso and a BSc focused on mathematics, physics, computer science and a higher diploma in Information Technology from the University of Colombo, Sri Lanka. Her contact email address is umunipala@miners.utep.edu.