Use Authorization

In presenting this dissertation in partial fulfillment of the requirements for an advanced degree at Idaho State University, I agree that the Library shall make it freely available for inspection. I further state that permission to download and/or print my dissertation for scholarly purposes may be granted by the Dean of the Graduate School, Dean of my academic division, or by the University Librarian. It is understood that any copying or publication of this dissertation for financial gain shall not be allowed without my written permission.

Signature_____

Date _____

GPU Oriented Approach to Finite Element Multiphysics Simulation

By

Dawid L. Krol

A Dissertation submitted in partial fulfillment of the
Requirement for the degree of

Doctor of Philosophy
in
Engineering and Applied Science

IDAHO STATE UNIVERSITY

May 2015

To the Graduate Faculty:

The members of the committee appointed to examine the dissertation of DAWID KROL find it satisfactory and recommend that it be accepted.

_____

Dr. Steve Chiu, Major Advisor

_____

Dr. Dawid Zydek, Committee Member

_____

Dr. Eugene Stuffle, Committee Member

_____

Dr. Hossein Mousavinezhad, Committee Member

_____

Dr. Jason Harris, Graduate Faculty Representative

Dedication

I dedicate this thesis to my beloved wife Joanna Bieganska. Back in 2012 she decided to go through this wonderful but also demanding journey with me asking for nothing in return. She is the most the most charming and kind person I have ever meet. She believe in me unconditionally more than anyone else including myself. Thanks to her support and help in almost every area on my life I was able to complete the Ph.D. program and to finish the dissertation – I am sure that would not be able to do that without Joanna.

I would also like to extend the dedication to my parents – Grazyna and Kazimierz Krol – for their support, helping hand, and weekly Sunday Skype calls. Knowing that you have a place that you can call *Home* and to which you can come back no matter what happens is one of the most comforting thing on Earth. I would also like to thank my parents for being a *role models* for me a long before I knew what it means.

Acknowledgement

Firstly I would like to thank Dr. Dawid Zydek – my first advisor at Idaho State University and a person that made my Ph.D. program possible. Throughout all of these years, and even after Dr. Zydek decided to follow other career path, he always had time to answer my questions regarding research, address my concerns, or simply have a casual conversation. Dr. Zydek has extraordinary and positive influence on my research and thesis; he helped me greatly with all my publications. Every conversation we made was very motivating and encouraging to make further effort. Aside from academia, Dr. Zydek helped me a lot with making first step, or actually first few hundreds steps, in United States of America. I will never forget how much Dr. Zydek did for me and I will be always thankful for that and for his friendship.

I would also like to thank Dr. Steve Chiu who agreed to be my advisor for final year of program. I am sure it was not an easy decision to take a responsibility for student in final stage of research. Without his help I would never be able finish Ph.D. program. I am also very grateful for all the time Dr. Chiu spent on answering my countless emails, helping hand in number of academia and non-academia situations, and valuable advices regarding

my future career in United States of America. At last but not at least I want to express my gratitude for great influence on the direction and form of my research.

A special thanks Committee Members (listed in alphabetical order) for their time and commitment: Dr. Jason Harris, Dr. Hossein Mousavinezhad, and Dr. Gene Stuffle. I am very thankful for valuable input and inspiration they shared with me throughout 3 years of my studies. I am also grateful for all their help and support in non-academia field. It was a pleasure to meet and work along with such great researchers and brilliant minds – I will never forget that experience.

I would also like to extend my personal thanks to Polish community from Pocatello: Monika and Dawid Zydek, Eliza and Lukasz Borzadek, and Agnieszka and Piotr Lotowki. Living thousands of miles away from home is never easy however thanks to all aforementioned persons, holidays that we spent together, and time we shared I felt in Idaho like at home. Thank you for being there all the time.

Finally I would like to thank Dr. Leszek Koszalka and Dr. Iwona Pozniak-Koszalka - my advisors at the Wroclaw University of Technology. Their phone call on February 12, 2012 in which they asked “did I ever think about doing Ph.D. in United States of America” was so alike scenario of Hollywood movie in which a dreams come truth within a minute. I would never be able to full express how thankful I am for turning my life in that direction. I am so glad and proud that Polish students have a privilege to meet such wonderful persons like you.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

- **AMR -** Adaptive Mesh Refinement
- **BLAS** – Basic Linear Algebra Subprograms
- **BLT** – Bioluminescence Tomography
- **CAD** – Computer-Aided Design
- **CUDA** – Compute Unified Device Architecture
- **CPU** – Central Processing Unit
- **DCC** – Digital Content Creation
- **FE** – Finite Element
- **flops** – floating point operations per second
- **GPGPU** – General Purpose Graphical Processing Unit
- **GPU** – Graphical Processing Unit
- **GUI** – Graphical User Interface
- **HID** – Human Interface Device
- **HPC** – High Performance Computing
- **MOOSE** – Multiphyscis Object Oriented Simulation Environment
- **MPI** – Message Passing Interface

- **PDE** – Patrial Differential Equation
- **PETSc** – Portable, Extensible Toolkit for Scientific Computations
- **SIMT** – Single Instruction Multiple Threads
- **SM** – Streaming Multiprocessor
- **STL** – Standard Template Library

Abstract

Multiphysics simulations, which involves a mathematical model of various physics phenomena expressed using partial differential equations, are integral part of projects and research conducted in number fields of science. It allows to gather knowledge about model, predict future condition, and saves a lot of time and money.

One of the most popular method of performing multiphysics simulation is a Finite Element method. As a numerical method it can be run by computers. Unfortunately large scale simulations require humongous amounts of computational resources. Yet, even then simulation process may take many days or even weeks.

General Purpose Graphic Processing Unit (GPGPU) is a new approach in high-performance computing that favorites highly parallel execution using hundreds of thousands of low-performance GPU cores over classic thousands of high-performance CPU executors. As the related work shows this approach can be very beneficial when applied to certain category of problems.

The goal of the research was to enhance the performance of existing multiphysics simulator by applying GPGPU model to it. It was important to preserve the functionality of existing simulator and to keep the interfaces unchanged so that applications built on it would not have to be modified.

To accomplish that goal three approaches were tested. In first of them it was assumed that whole library will be redesigned and reimplemented to be executed entirely by GPU. The approach led to failure because of highly object oriented design of existing simulator, extensive usage of Standard Template Library (STL) containers, and numerous branching instructions; all of these are poorly supported by GPGPU. In the second approach hybrid GPU/CPU implementation was proposed. All highly-parallel algebraic operators used by simulator were reimplemented to run on GPU; as a result custom GPU BLAS library was created. Results show that GPU/CPU approach was 10% faster than classic CPU approach. In third approach custom BLAS library and STL containers were replaced by *cuBLAS* library and *Thrust* containers, included in new release of CUDA programming model, respectively. Results show further improvement in terms of performance.

The research proves that reimplementing the existing multiphysics simulator to run on GPU is possible and results in enhancing the performance of simulation. Both the research and implementation are good starting point to evaluate the cost effectiveness, energy efficiency, and fault tolerance of proposed approach.

Preface

"Mathematics is the language of nature" – this sentence was frequently said by mathematics prodigy, gifted scientist and a brilliant mind – Maximilian Cohen – the main protagonist of movie Pi from 1998 [1]. Quoted phrase is an entry point to his research, which goal is to discover a universal pattern that describes the universe. Although the concept of a comprehensive pattern or equation can be classified as math fiction, assumption that "mathematics is the language of nature" might be quite probable thesis. To support it one can refer to number of various papers and essays written throughout the centuries and especially to texts from XX century. Galileo Galilei, an Italian physicist from XVI century, wrote [2] that "philosophy [nature] … is written in mathematical language." Few hundred years later Sir James Jeans, British physicist, suggested in [3] that "God is a Mathematician." The most famous and influential essay, however, was written in 1960 by Eugene Wigner – Noble Prize awarded physicist and mathematician. In the paper commonly known as *Unreasonable Effectiveness* [4] author expressed the special role of mathematics in modern fields of study and, what is the most important, pointed out how mathematical concepts developed for certain case, are often applicable to problems that are far from original context.

As an example author referred to Newton's law of universal gravitation which origins in observation of free falling object on the surface of the Earth. The same law can be easily adapted to describe forces generated between the planets. Later in the text Eugene Wigner brought up Maxwell's equations that originally meant to model the magnetic and electric phenomena. Almost 30 years later the same equations were used by David Hughes to describe his new discovery [5] radio waves.

Traces of mathematics can be found everywhere. High school students apply simple physic formulas, and therefore mathematics, calculate distance, velocity, and acceleration. Undergraduate Sociology students data frequently refer to Gaussian distribution since surprisingly often statistical data generated from surveys perfectly match the curve of Gaussian function. Gaussian distribution as a tool was invented decades before the modern social science [6]. Moreover the function contains number π which origins are in Ancient Egypt and was used in completely different context [7]. Analogous situation can be observed in Structural Engineering. Concepts like bending moments, compressions, or slope and deflections were for centuries intuitively understated by firsts masons or builders. When physical background of these concepts were finally discovered by Leonardo da Vinci in late XV century and developed by his successors, like Leonhard Euler [8], in XVIII century it appeared that mathematical background already exists. Following that path even further one can observe similar tendency in quantum mechanics and string theory. First of them uses complex numbers which in 1545 century were just a trick to find real roots of certain polynomial equations [9], second is based on created in 1813 non-Euclidean geometry which was treated like a joke until 1914 – the beginning of string theory era.

All of these examples allow to presume that mathematics impersonates reversed visionary. In thesis author's personal opinion visionaries, rather that foreseeing future, are setting a flag that is later pursued by scientists and engineers around the world. If so then mathematics may be compared to ready solution that is waiting for a visionary to foresee an application for it. This scenario is also applicable to partial differential equations and multiphysics simulation.

Research theses

Throughout last two decades an attention of researchers from various fields of science is turned at mathematical models and multiphysics simulation. Modern computer simulation and Computer Aided Design allows engineers from different fields to increase the effectiveness of their work and research. This trend is not surprising since multiphysics simulation is crucial for almost every civil engineer project, automobile design facility, chemical laboratory, and nuclear power plant.

Simulation helps to improve the understanding of functionality and behavior of a model. Using models allows prediction of future conditions and foreseeing possible issues. Very often simulating a model is the only potion of a problem that can be solved. Furthermore, multiphysics simulation, saves a lot of time, energy, and subsequently, money.

Unfortunately when one is willing to apply multiphysics simulation to very complex model then he may quickly hit the performance barrier. In fact increasing complexity of model causes rapid growth in computational resources consumption. As a result complex simulations are a domain of research facilities equipped with extremely expensive

supercomputers. Although manufactures release more and more advanced supercomputers the “appetite” for computational power of scientists is unsatisfied. The answer for this problem may be seek in modern technologies that break up with old approaches to computations. Such an approach is General Purpose Graphical Processing Unit approach.

The idea of GPGPU assumes using modern Graphical Processing Units to perform general purpose computing. GPU is composed of a number of independent multiprocessor units called Streaming Multiprocessors. SMs execute in parallel thousands of instances of code called kernels. Although a single processor within a SM does not provide high performance and the single thread is not executed as fast as it would be on a modern Central Processing Unit, the ability to execute a massive number of threads in parallel gives GPU exceptional performance. This observation and assumption lead to the hypothesis that:

Applying GPGPU to multiphysics simulation may results in shortening simulation time while keeping the same results.

Hypothesis stated in that way is generic and there are many ways to confirm it or to prove it wrong. To start witch one can choose from different GPU manufacturers which may and will have an impact on performance, power consumption, and accurateness. Moreover brand of device can dictate the programming model – GPUs produced by *AMD* can execute code written in *OpenCL* whereas *NVidia* GPUs can, as for today, execute both. To continue, underlying hardware enforces developers to focus on different architecture-specific aspects crucial to use computational resources of GPU in full. Secondly one may want to work on a specialized simulator that can be applied to a small domain – e.g. heat

transfer – or develop framework that can be applied to wider spectrum of simulations. Finally GPGPU approach may be a starting point to implement new multiphysics simulator or applied to existing software. In first scenario program can be tailored to underlying architecture and therefore very effective; in second scenario framework design may compromise the performance however all application already build on the framework will remain functional. To narrow the scope of research three questions were asked:

- *Is it possible to re-implement existing multiphysics simulation framework without affecting existing software build on the framework?*
- *Will the GPGPU approach results in better performance of multiphysics simulation framework?*
- *Would the potential performance gain be worth of time and money required to re-implement and test thoroughly the framework and provide required hardware?*

These research questions point the direction in which research will be leaded and endorse the goal of the research.

Enhance the performance of existing multiphysics simulator by applying General Purpose Graphical Processing Unit model to it.

Scope of the research specified in that way makes it accomplishable in reasonable period of time and, in the same time, does not affect the generality of consideration.

Dissertation Overview

a. Content of the Dissertation

Following dissertation is divided into six main chapters. The order of chapters and sub-chapters reflect the order in which the study and the research were conducted. This manuscript covers both successful ideas but also concepts that failed. The dissertation consist of:

Chapter 1: Literature review – this section is divided into two sub-chapters. First one covers the topic of multiphysics simulation with emphasis put on Finite Element method concept and implementations. Second part is focused on GPGPU technology related to CUDA and CUDA-enabled hardware. Each sub-chapter consist of introduction to basic concepts and literature review of the corresponding topic. Literature review part presents other research in which concepts touched in this thesis were considered, describes benefits and limitations of these approaches, and exhibit related work which is goal similar to the one set in this research.

Chapter 2: Examined approaches – in this section all examined approaches to speed-up simulation time were explained. In first sub-chapter holistic approach is presented. In

this approach it was assumed that the complete process, that single CPU thread executes, can be translated to CUDA and launched by a single GPU thread. The design problems, architecture limitations, and major obstacles to accomplish the goal in this way were presented. Second sub-chapter is focused on hybrid CPU/GPU approach in which heavy computational parts of code were ported to CUDA and tailored into multiphysics framework as a layer between framework and hardware. The concept, advantages, and drawbacks of the approach were discussed. In last part of the chapter hybrid approach modified to use in-build libraries from the newest CUDA release was presented. New CUDA programming model includes number of libraries which may be very beneficial, in terms of performance and code robustness, to implement especially that it would not affect the idea of hybrid approach. It is enough to modify mentioned earlier layer between multiphysics framework and hardware. Therefore no further modifications in multiphysics framework itself would be required.

Chapter 3: Algorithms – in this chapter all methods designed and implemented were presented. Each method, and approaches to implement it, is considered in separate sub-chapter in which appropriate description, implementation details, and CUDA code is presented. Performance of parallel GPU application relies heavily on used technique and problem it is applied to – mechanisms that work great for certain problems may have terrible performance for others. Therefore every method was implemented in more than one way in order to examine different concepts and techniques. Different implementations are considered in sub-chapter specific to the method they implement.

Chapter 4: Experiment environment – this chapter describes the environment in which experiment were carried out. The detailed hardware configuration of machines used in the research was presented; the differences between them was described and detailed explanation on key differences was provided. In this section crucial concerns that follows every performance tests were discussed. Among others special attention was paid to external nondeterministic factors that may have negative influence in results. Because of that steps to reduce the noise and techniques to filter out remaining noise were proposed. The general plan of experiment was expressed using simple mathematical formulas. The details of test cases – like size of matrices that are about to be multiplied – are provided together with results in next chapter.

Chapter 5: Results – in this section obtained results were presented and briefly discussed. Performance evaluation was grouped in four sections. In first two of them hybrid CPU/GPU approach was considered. Firstly effectiveness of CUDA-enabled algorithms were evaluated and referred to performance of corresponding CPU algorithms; second the algorithms were introduced to simulation framework and performance of whole system was inspected. In second two subsections analogous scenario was used. First CUDA--enabled algorithms were implemented using features of new CUDA programming model are implemented into simulation software. Their efficiency was referred to previous results. Second multiphysics simulator was modified and compared to previous version of the application. Also in this chapter accurateness of GPU algorithm was assessed and compared to accurateness of CPU algorithm.

Chapter 6: Discussion – in this chapter results obtained from the experiments are discussed. Performance of GPU-enabled algorithms were assessed by comparing efficiency of different versions of the same method to each other and to effectiveness of classic CPU method. Later on the performance of proposed algorithms were compared to performance corresponding algorithms from commercial library supplied with new version of CUDA framework. Furthermore the behavior of every algorithm is discussed together explaining the reasons of why some algorithms are better than others. In this section it is also discussed how selection of CUDA algorithms may affect overall multiphysics simulation performance. It is also described why in certain cases hybrid GPU/CPU approach results in performance loss and what requirements have to be fulfilled in order to enhance the performance.

Conclusion – in last section of this dissertation a brief summary of conducted work was presented. In this section advantages and disadvantages of proposed approach are presented and the answers for research questions are formulated. In this part guidelines regarding future work are presented, aspects that were omitted in this manuscript but are worth or even must be considered are pointed out, and limitations are discussed. Also the thesis of the dissertation is reevaluated in terms of initial expectations, observations, results, and limit of proposed approach.

b. Self –Citations

The following papers played important role and contributed to research presented in this dissertation and contributed:

- *Solving PDEs in Modern Multiphysics Simulation Software* [10]: presents the concept of multiphysics simulation in HPC. It focuses on the architecture of multuphysics simulators and underlying libraries. In this paper code analysis of *libMesh* was conducted.

- *Matrix Multiplication in Multiphysics Systems Using CUDA* [11]: first implementation of CUDA algorithm that later was composed, together with other methods, custom GPU BLAS library were presented and evaluated in terms of efficiency.

- *Hybrid GPU/CPU Approach to Multiphysics Simulation* [12]: first iteration of GPU-enabled libMesh library is proposed. Instead of one-to-one translation from CPU thread to GPU thread hybrid approach is proposed.

- *Effectiveness evaluation of cuBLAS and Thrust CUDA 6.5 libraries* [13]: alternative implementation of previously proposed GPU-enabled BLAS operator was presented in this paper. It was an entry point for proposing second iteration of *libMesh* library implementation that uses *cuBLAS* operators and *Thrust* data structures

- *Problem-Independent Approach to Multiprocessor Dependent Task Scheduling* [14]: points out importance of parallel processing and importance of proper job management in HPC environment

Chapter 1
Literature review

Chapter 1.1. Multiphysics Simulation

Multiphysics simulation is a complex process that requires significant time and computational resources. It involves a number of physical phenomena usually described by PDEs – the process of multiphysics simulation is therefore equivalent of solving system of PDEs. Currently one of the most popular approaches to solve PDEs is the FE method. The method was originally developed in 1943 by A. Hrennikoff and R. Courant, it gained popularity in 1960 when applied to the problem of electromagnetic wave propagation [15]. Simulations are widely applied in most major fields of science and business like nuclear power plants, civil engineering, and automotive. Very soon after FE-based simulation become popular market responded with wider variety of multiphysics simulation software like *COMSOL Multiphysics* or *ANSYS Multiphysics*, but also open source software like the, *MOOSE Framework*.

Engineers and scientists are eager to use new technologies that may offer promising possibilities for performing simulations faster and easier. Although these aforementioned applications can be easily used on a typical modern personal computer, more advanced

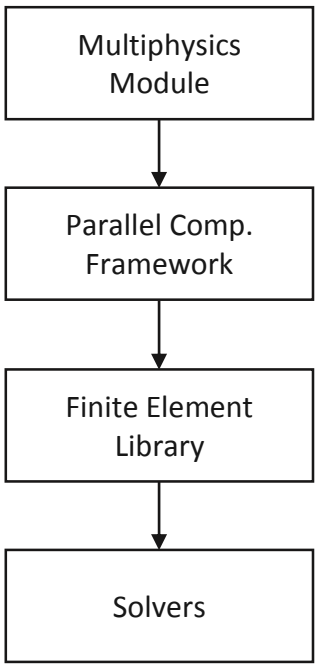


Figure 1. Generic multiphysics simulation system

simulations that incorporate a significant number of physics phenomena requires very precise results. These simulations are performed on meshes assembled from millions of shapes and require an enormous amount of computational resources. Resources are, as usual, limited and because of that researchers search for simulation framework that suits or can be accustomed to suit their needs.

a. Generic approach to multiphysics simulation

The general simulation approach used in most of FE-based simulators can be presented by diagram in Figure 1 [10]. The process starts with defining mathematical model of multiphysics phenomena problem with corresponding properties and expressing it in manner required in underlying framework. This step is performed in first layer of

generic simulator diagram called Multiphysics Model. Top level layer provides user a set of helper functions, methods to import data from CAD applications, or very often a graphical interface in which user can assemble model and set appropriate properties. Input data is the passed to Parallel Computational Framework. By definition each framework is a set of more or less compound wrapper functions that group calls of generic methods from underlying libraries in order to provide easy access to problem specific system. In this scenario second layer of simulation system is responsible for preprocessing of input data and parameters; base on that appropriate simulation plan, data structures, monitoring procedures, and thread pool are created. Finally the simulation is started and, from this point, is performed by FE library and FE solver [10]. Another role of Parallel Computational Framework is maintaining the contact with user and informing about status of the simulation. A popular approach to handle communication is through console log. The next layer, FE library, handles the simulation. This layer supplied the framework with utilities to perform FE based computations like input/output mechanics for meshes, error handling protocols, and interfaces to solver packages. It is also responsible for managing parallel processing across multiple computational nodes. Last layer is a Solver layer which consists of set methods that solves systems of differential equations, BLAS operators, and basic data structures with associated primitives [16]. The layer is responsible for solving an actual set of PDE provided by FE library in efficient way; because of its proximity to hardware it often provides implementation MPI and profiling tools. Results generated by Solvers are passed back to FE library which proceeds to next step of simulation or prepares the output and transmit it to higher layers.

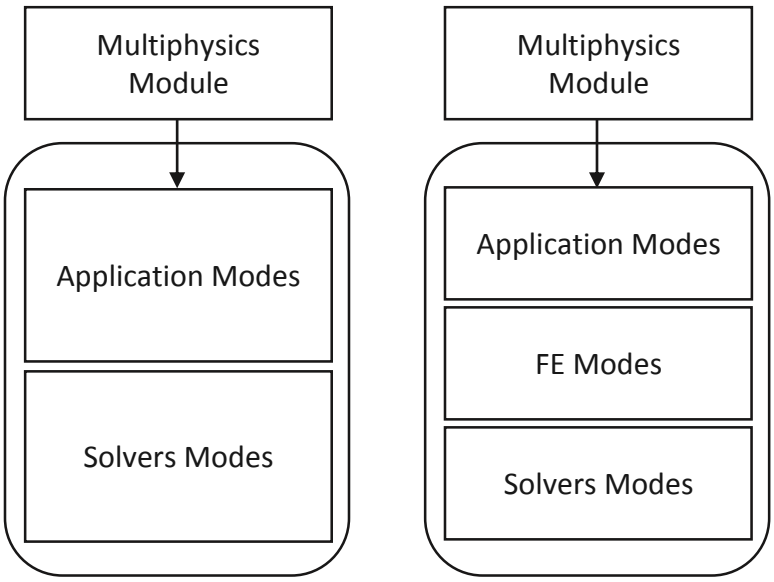


Figure 2. Architecture of commercial simulators (ANSYS Multiphysics on the left, COMSOL Multiphysics on the right)

b. Commercial multiphysics simulators

Currently there are two leading commercial multiphysics simulation systems: *COMSOL Multiphysics* and *ANSYS Multiphysics*. Both are an interactive environment for modeling and solving various scientific problems based on PDEs using FE method. Applications offer a user friendly interface – multiphysics module – that speeds up defining of problem and simulation parameters. Both applications provide number of API and plugins that allow to integrate simulator with custom applications written in most major programming languages including *MATLAB* or to import data from CAD applications. *COMSOL* and *ANSYS* are widely used in areas like acoustics, heat transfer, photonics, and structural mechanics [17] [18]. Moreover, as the documentation points out, their architecture – Figure 2 – has very close resemblance to generic model presented in

Figure 1. Both simulators provides satisfying performance and wide spectrum of appliances, however there are cases when more flexible, problem specific, and scalable solution is required – this is when custom frameworks like *MOOSE* becomes handy.

c. MOOSE Multiphysics

MOOSE is created and developed at INL. It is the multiphysics parallel computational framework used for solving computational engineering problems. Application was designed to reduce time and expense, required to develop new software, and to perform simulation in organized, manageable, and coordinated manner. MOOSE can be used, similar to its commercial competitors, in areas like heat conduction, fluid flow, solid mechanics, thermo-mechanics, and many others [19]. Originally a government application, it went open source on March 21st, 2014 [20]. Since then popularity of packages MOOSE is built on and MOOSE itself is growing in academia.

MOOSE allows performing up to 3D analysis. System is capable of using unstructured mesh that is built from shapes such as triangular, quadrilateral, tetrahedral, prism, and others. Framework also provides developers huge variety of post processing options [19]. All MOOSE functions can be performed in parallel in CPU cluster, e.g. at INL system the framework is used on supercomputer that number of cores is counted in thousands. Currently MOOSE supports Jacobian-free Newton–Krylov method with Physics-Based Preconditioning for solving tightly coupled multiphysics modules [21].

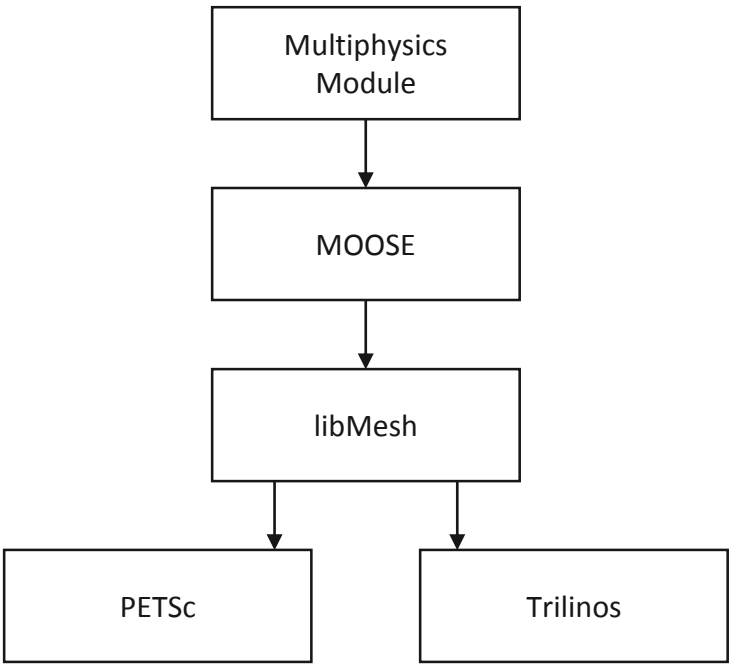


Figure 3. MOOSE Multiphysics architecture

MOOSE is an object oriented FE-based framework. This means that framework does not enforce use of integrated FE libraries or solvers (system can use any FE library available and best solvers peculiar to the selected library). In MOOSE, interfaces to one FE library (*libMesh*) and two solvers (*PETSc* and *Trilinos*) are provided; however, the framework is capable to use any other FE-based software. The structure to solve multiphysics problems using *MOOSE* is shown in Figure 3 [10].

As it can be seen MOOSE architecture matches generic ideally. Multiphysics Model layer of MOOSE framework is an application written by user based on methods provided by MOOSE. In this layer input data like mesh of an object and list of properties has to be specified. MOOSE layer is an interface between user code and FE library; it initializes all underlying data structures, preprocess and validates input, and starts the simulation.

LibMesh, as the method of weighted residuals, processes the FE computations after receiving the weak form of PDEs from *MOOSE*. Solver, as a tool finding PDEs' solutions, can be subset of FE Library for providing large scale parallel computing resources [10].

LibMesh is frequently used, especially after *MOOSE* went open source, in academia and commercial researches from vast variety of field of science. In [22] author simulated random initiation and subsequent propagation of interacting thermal cracks in a ceramic nuclear fuel pellet using coupled mechanics, heat conduction, and fracturing. The simulation allowed to precisely demonstrate the formation of cracks during the initial power rise and power ramp downs. A problem of estimating a hydrogen behavior and distribution in nuclear fuel rod was considered in [23]. *LibMesh*, as a part of *MOOSE* framework, was applied to model composed of diffusion under concentration gradient and temperature gradient. The simulations predicted that hydrogen tends to accumulate on colder areas right before it precipitates and as a result degrade the cladding ductility.

d. libMesh

The *libMesh* is a framework that uses arbitrary unconstructed discretization for numerical simulations of partial PDEs. First version of application was developed at the University of Texas at Austin in 2002. However, a major contribution in developing *libMesh* throughout ages came from INL, MIT, and PECOS Center [24].

The *libMesh* library was designed and implemented to simply parallel, adaptive, and multiscale multiphysics FE simulations. Designers wanted to achieve that by centralizing physics independent technology to support parallel and adaptive unstructured mesh-based

simulations. This approach allows users to focus on the specifics of a given application without considering the complexities of parallel and adaptive computing. Thanks to that *libMesh* proved a robust environment for a wide range of physical applications [10].

The library uses AMR in FE simulations. AMR can produce efficient meshes for refining solution through the coarsely resolved base-level regular Cartesian grid [25]. Framework supports 1D, 2D, and 3D simulations on big variety of geometric and FE types. The library provides interfaces to libraries that perform linear algebra computations, meshing, and partitioning.

Although the simulation is already performed in parallel with MPI on multiprocessor supercomputers, large scale simulations of models represented by a multimillion cell mesh still calls for more computational power. Since the frequency boundary of the CPU was almost reached, the only solution is to increase the number of computational nodes. However, upgrading existing supercomputers by adding new CPUs is very pricey and the increase of performance may be relatively small compared to the cost of modernization. IT seems that using GPGPU might be beneficial in this case however according to documentation no GPU support is provided and is not planned to be provide in the nearest future [24] [12].

e. PETSc

PETSc is a suite of data structures and methods designed for the scientific applications modeled by PDE. Library supplies developers with building blocks for the implementation of large-scale application executed, in parallel as well as in series, by

computers [26]. *PETSc* includes a suite of parallel linear, nonlinear equation solvers and time integrators. Components may be used in custom applications written in most major programming languages like FORTRAN, C/C++, Python, or MATLAB[1]. Through implemented MPI standard for all message-passing communication, *PETSc* provides many of the mechanisms needed within parallel application [27]. The library is organized hierarchically and by that enables users to employ the level of abstraction that is most appropriate for a particular problem [10].

PETSc consists of a variety of libraries. Each library implements certain family of objects and methods related to that object. Object form a hierarchy that enforces user to follow specified order of execution and to use only classes required by the simulation. Modules provided by *PETSc* are as follows: index sets for indexing into vectors, renumbering, etc.; vectors and matrices as a basic data storage with basic operators and subroutines; managing interactions between mesh data structures and vectors and matrices; over fifteen Krylov subspace methods; number of preconditioners, including multigrid, block solvers, and sparse direct solvers; nonlinear solvers; time steppers for solving time-dependent nonlinear PDEs including support for differential algebraic equations [28].

f. Trilinos

The developer of *Trilinos* – Sandia – historically did a work in area of developing scalable solver algorithms and software. Their software, however, was often enclosed within single context of a specific application code, providing a good robust solver that

[1] MATLAB supports only sequential execution of *PETSc*

specifically meets the needs of that application. The best example is *Aztec*, solver developed for *MPSalsa* project and only later extracted for use with other applications. Despite robustness of the solutions, applications were rarely reused in different projects and for different purposes. Therefore at some point developers decided to create a library of various project that would have a wide spectrum of appliances in different areas of scientific experiments – that decision set the cornerstone for *Trilinos* project [29].

Trilinos is the library of packages for solving the large-scale and complex multiphysics problems [30]. Application supports number of linear, nonlinear, and eigenvalue problems. *Trilinos* differs from *PETSc*, which has independent packages. In fact, Trillions could easily use *PETSc* to provide a variety of capabilities through the documented abstract interfaces without modifying their source code [31]. In addition, *Trilinos* supports also external solvers, like *PETSc*, by supplying users with interfaces. Therefore application can be used also as a framework placed between FE library and solver. Each *Trilinos'* package is the independent unit implemented using a particular algorithm [10].

Trilinos provides packages like nonlinear solvers like *NOX*, *LOCA*, and *GlobiPack*; linear solvers e.g. *AztecOO*, *Belos*, and *Komplex*; eigensolver Anasazi; preconditioners like Meros, *ML*, *Ifpack*; Basic Linear Algebra module *Epetra* and *Jpetra* implemented in C++ and Java respectively; common services package *Teuchos*.

g. Conclusion

As the documentation points out both *Trilinos* and *PETSc* support MPI, shared memory pthreads, and GPUs through CUDA or OpenCL. *PETSc* supports also MPI-GPU parallelism. Therefore FE solvers layer is rather unlikely to be an obstacle to overcome when porting the simulator to GPU. Because of that these packages will not be considered in this thesis, however their GPU capabilities will be used in experiment that assess the effectiveness of redesigned multiphysics simulator.

Chapter 1.2. General Purpose Graphic Processing Unit

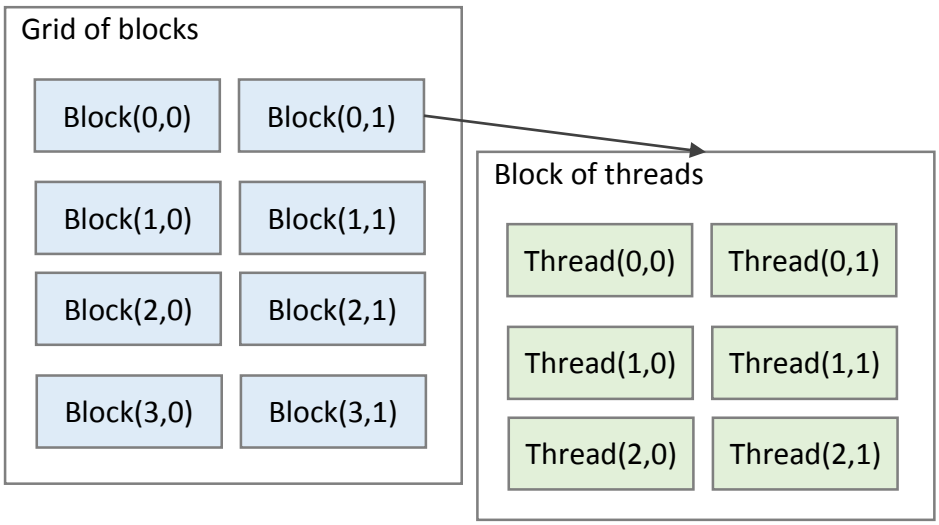
Throughout last couple years the area of HPC experienced a rapid and effective changes in field of both software and hardware. According to TOP 500 ranking the most powerful machine, ranked in June 2013 [32], has twice as much computational power as previous, ranked in November 2012, leader. Such a humongous progress in terms of computational resources originates from current trend to relay on highly parallel processing rather that relatively small cluster of high-frequency CPUs. In 2012 a version 3 of MPI standard [33], used by nearly every supercomputer, was approved. A year later, in June 2013, new version of another notable API - *OpenMP* - was released [34]. *OpenMP* focuses strongly on shared memory multicore processing and fill the niche left by MPI. A milestone in HPC, however, was introducing General Purpose Graphical Processing Unit. Concept assumes that parallel execution by thousands of low performance cores gives better result that executing the same program by just few high performance CPU. The cornerstone of idea is the limitations that hardware engineers approached. Moore's Law states that the

number of transistors located in a dense integrated circuit doubles every two years [35]. However increasing quantity of transistors placed on the same area eventually cause major growth of temperature. This lead to malfunctions or even irreversible damage. Therefore the answer to demand for computational power was to process in parallel.

In GPGPU SMs, that GPU consists of, are capable of executing enormous number of threads and therefore act as a cluster of low performance cores. The idea appeared to be so good that in 2010, soon after *NVidia* released parallel computing platform named CUDA, GPGPU revolutionized and dominated the market of HPC. As the TOP500 ranking shows in June 2010 only one supercomputer ranked in TOP 5 of the ranking was using GPU whereas in November 2010 three machines from TOP 5 (including the leader that doubled performance of his predecessor) relied on GPU [32]. Market responded almost immediately and main *NVidia* competitors on hardware market – AMD (that acquired ATI in 2006) and *Intel* - released their solutions – *AMD Radeon* GPU with *OpenCl* and Intel *Xeon Phi*. Both companies quickly caught up nevertheless NVidia still remains the leader due to new cutting-edge devices, extensive support, and new releases on CUDA framework.

a. CUDA programming model

From developer point of view kernel is a single, usually short, function that is executed by GPU threads. Thread is the most basic execution structure in GPU environment. Kernels are written in CUDA language which is a C++ with minimal



extensions; nonetheless it is important to remember that not all C++ features are supported by GPU kernels.

When kernel is called one has to explicitly specify how many threads are assigned to it; each thread assigned to kernel will execute exactly the same code. Threads are grouped in one-, two-, or three- dimension blocks. Each thread within a block of threads can be identified by properties$threadIdx.x$, $threadIdx.y$, and $threadIdx.z$; these properties are coordinates of thread. The property is always defined; when block is one-dimensional only then $threadIdx.y$, and $threadIdx.z$ will be 0. Block x-, y-, z-dimension may be retrieved by calling $blockDim.x$, $blockDim.y$, and $blockDim.z$ respectively. Logical

Figure 4. CUDA thread logical structure [54]

structure of thread mesh is presented in Figure 4. Maximal possible dimensions of block, as well as maximum number of threads within a block, are dictated by CUDA capability of used GPU [11].

Blocks of threads forms another structure called grid. Similar to block, grids may have up to three dimensions. Each block is identified by its coordinates: $blockIdx.x$, $blockIdx.y$, and $blockIdx.z$; maximal dimensions of block and maximal allowed number of threads within a block are related to CUDA capabilities of GPU. Threads within one block share one SM and one portion of memory. It allows to cooperate with each other to speed up execution of kernels.

As it was aforementioned each thread executes exactly the same part of code. Therefore the factor that distinguish the obtained result is where the thread is located. Base on block coordinates, block dimensions, and thread coordinates it is possible to calculate absolute coordinates of thread within an entire execution logic structure. Traditionally the absolute coordinates of thread are named idX, idY, and idZ can be calculated from equations (1), (2), and (3). The coordinates, distinct for different threads, differentiates the context under which thread runs.

$$idX = blockIdx.x \cdot blockDim.x + threadIdx.x \quad (1)$$

$$idY = blockIdx.y \cdot blockDim.y + threadIdx.y \quad (2)$$

$$idZ = blockIdx.z \cdot blockDim.z + threadIdx.z \quad (3)$$

To conclude the developer point of view, the general idea of parallel programming is to visualize the problem as a mesh of smaller sub problems and to overlay a logical structure of threads onto it so that threads are mapped to sub problems. The general rule that can be applied to most cases is that to divide the problem as much as possible and

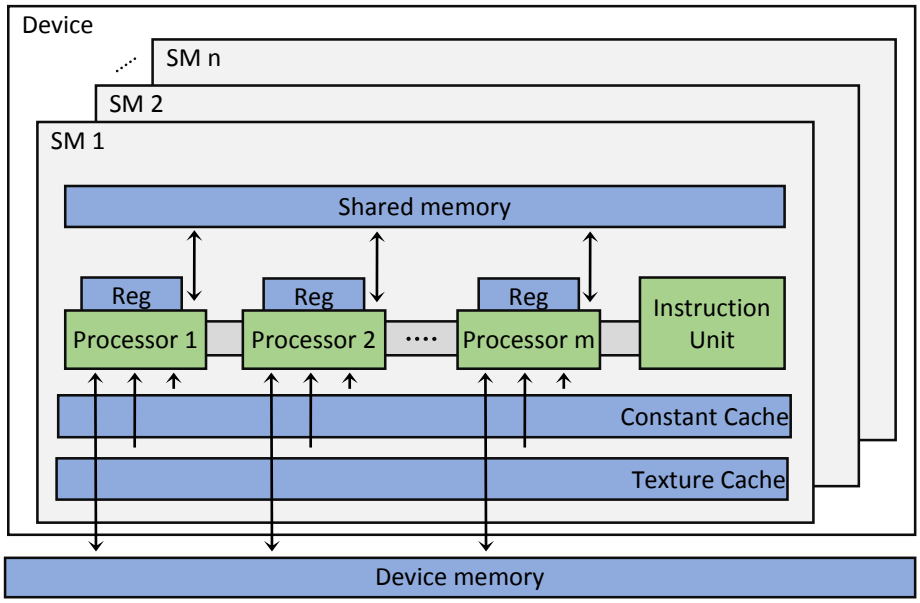


Figure 5. GPU architecture [54]

employ as many threads as feasible in order to increase performance and level of parallelism.

b. GPU architecture

Figure 5 presents the hardware architecture of GPU. GPU consists of set of Streaming Multiprocessors and global memory; each SM is built on set of interconnected processors, instruction unit, and additional levels of memory.

Each block of threads is assigned to a single SM and is executed entirely within a context of that SM. Although one SM may host more than one block of threads, one block cannot be split among multiple SM. Each SM is equipped with scheduler. It is responsible for scheduling bathes of threads, called warps, to processors within the SM. Size of warp is number of threads that will be executed in parallel by SM. Currently size of warp is 32,

nonetheless it is very likely to be changed in oncoming generations of GPU [11]. Each thread is executed by a single processor, therefore size of warp is also number of processors, sometimes called CUDA cores, within one SM. It allows to calculate number of SM in GPU by simply dividing number of processors, provided by GPU datasheet, by warp size.

GPU executes threads in SIMT manner. It means that an instruction unit, shared by all processors within SM, issues one and the same instruction at a time to each processor.

In CUDA programming model, presented in Figure 5, three types of memory are distinguished. First of them is global device memory – the main memory of the device. Usually it is implemented using DDR5 RAM. It can be accessed by every thread from every block. Because of large number of threads that can access this storage at Global memory is slow and therefore it is not recommended to store frequently accesses data there [36].

Second memory type is shared memory. This memory resides in SM and some part of the memory is assigned to a single block executed by SM. This part of memory can be accessed only by threads from block to which memory was assigned. It is very fast and efficient so the most frequently used data, especially the one that is used by many threads in the same block, should be placed here [11]. Third type of memory is local memory that can be used only by a thread that is assigned to this memory.

c. CUDA 6.5

New CUDA framework brings up a number of improvements including resolved previous version issues, support for CUDA Fortran, profiling tool interface, and number of libraries including *Thrust* and *cuBLAS*. Both of these libraries were an open source project a long before CUDA 6.5 [37]. *Thrust* is a GPU enabled equivalent of famous STL. STL was first introduced in '*90* and since then is a crucial part of many large projects. STL offers a set of flexible and optimized data structures together with corresponding optimized methods like sorting, searching, data sets merging etc. STL greatly improved the development process since programmers did not have to implement their own versions of common container-specific methods. *Thrust* library is based on the same idea like STL – even containers and methods kept the same names – and simplifies the process of CUDA code development. In addition *Thrust* allows to use GPU to store and manipulate data in parallel, and therefore more efficient, manner (e.g. sorting a list in parallel). *CuBLAS* library is GPU enabled BLAS which is a set of subroutines that performs common linear algebra operations in parallel. Origins of BLAS can be found in 1979 when it was a platform independent Fortran library that supplied developers with basic linear algebra operators that could be used as a blocks in bigger applications. *CuBLAS* offers a set of carefully designed and very well optimized linear algebra operators [38] which are encapsulated in easy to use wrapper function. Because of that *cuBLAS* not only simplifies the development process but also offers cutting-edge performance. Both *Thrust* and *cuBLAS*, being a response for developers (used to program in C++) needs, earned huge popularity eventually were approved by *NVidia* and include in CUDA 6.5 framework.

Possibilities and potential performance of *Thrust* and *cuBLAS* libraries are very promising therefore incorporating them in multiphysics simulation framework like

MOOSE may bring significant performance enhancement. Equally important is the simplicity and flexibility offered by mentioned libraries. Because of that eventual refactor of existing *MOOSE* implementation may be much easier that reimplementing it in plain CUDA. The goal of the paper is to evaluate the performance and usability of *Thrust* and *cuBLAS* libraries and, as a result, asserting their usefulness in multiphysics simulation software.

Although both *cuBLAS* and *Thrust* are relatively young part of CUDA framework the libraries itself exist for a few year and were able to attract significant attention of researchers and developers. In [39] authors evaluated the effectiveness of three level 3 *cuBLAS* methods – SGEMM, SSYRK, and STRSM – and proposed improved versions of these. In the experiment version 1.0 of *cuBLAS* was used; all kernels were executed by *NVidia GeForce 8800*. Authors started with evaluating the impact of size of input matrices on performance of *cuBLAS* operators and then moved to examining the relation between performance and matrix processing operator itself. As a result authors confirmed the effectiveness of algorithms and the improvements they proposed focus on combining *cuBLAS* methods with each other, dividing the work between GPU and CPU (which normally is idle when GPU executes the kernel), and resizing input matrices to maximize number of threads executed in one cycle.

Similar problem was also investigated in [40]. Authors made an effort to evaluate performance and accurateness of *cuBLAS* matrix multiplication method. As the reference point they used analogous algorithm from *Intel Matrix Kernel Library* and *ATLAS BLAS*. The experiments were carried out on computer equipped with *NVidia Tesla T10* and 8 core

Intel Xenon Nehalem. As results show GPU approach significantly outperforms CPU approach when single precision floating point numbers were considered. For double precision both solutions were equally effective. The disadvantage of *cuBLAS* method was, however, slightly smaller accurateness of results.

In [41] problem of WZ Factorization was considered. Authors evaluated the performance of matrix factorization algorithm implemented using CUDA library by comparing it to custom CPU and standard CPU BLAS version WZ factorization algorithm. In their work they used methods from level 1 and level 3 *cuBLAS* library. Experiment was carried out on *NVidia Tesla C2050*. Results show that algorithm written in CUDA outperforms CPU algorithms even by 6 times for large matrices. Authors also point out that CUDA algorithm reached almost 20 times bigger performance expressed in *Gflops*.

CuBLAS library was used in [42] to accelerate adaptive Finite Element framework for BLT. The BLT is a sensitive and accurate probing method that uses certain enzymes to mark biological entities like tumor cells or compounds of drugs. As a result in biochemical reaction part of energy is transformed into bioluminescent light which can be monitored. Obtained readings may be processed using FE method can be used to recover 3D image. FE simulation is time consuming process therefore authors decided to redesign their FE application to use *cuBLAS* library to obtain better performance. Experiments were conducted on *NVidia GT240* and results were compared to results obtained from CPU version of FE application executed on machine equipped with 8 core *Intel Xeon*. Research showed that FE application that used *cuBLAS* executed matrix inversion 20 times faster and matrix multiplication over 200 times faster than CPU version of FE application.

Thrust library was used in research presented in [43] where authors used *Thrust* data structures to search the effective variable-length string sorting algorithm. As they mentioned string sorting was a major issue even when fixed-length strings were used. Algorithm that were proposed simplifies the variable-length string sorting problem to series of short fixed-length string sorting problems: first CPU extract few character long prefixes, second *Thrust* data structure sort operator is used to sort an array of extracted prefixes. Experiments were carried out on *NVidia GTX 580* and *NVidia K20*. On both devices algorithm obtained much shorter execution time than the serial version of the same algorithm. Surprisingly the difference in execution time between *NVidia K20* and *NVidia GTX 580* was not significant, however authors did not covered this case.

In [44] *Thrust* library was used in Discrete Event Simulation – a technique that allows to study the dynamic behavior of complex systems – problem. Authors used a node of supercomputer located in Ohio Supercomputing Center that is built on two *Intel Xeon* cores and *NVidia Tesla M2070*. Similar to other researches in which CPU and GPU versions of the same algorithm are compared when problem scales in small both version has similar effectiveness or CPU slightly outperforms GPU. When the problem scale got larger the execution time of CPU algorithm starts to grow rapidly. Authors observed that CUDA algorithm was almost 60 times faster that its competitor for a large scale problem.

Data structures from *Thrust* library were also used in [45]. Author explored how *Thrust* library can be used to enhance performance of sound simulation and jitter analysis algorithms with minimal changes in already existing serial C++ application. In the experiment machine equipped with *NVidia GTX 480* GPU was used whereas *Intel Core2*

Quad CPU was used to generate the reference point results. As the results show GPU approach allowed to obtain better performance for each considered input. For the smallest input data set (1 Million of samples) GPU enabled algorithm was 2 times faster than CPU algorithm; for the largest data set (16 Million of samples) GPU version was over 9 times faster.

Chapter 2
Examined approaches

Chapter 2.1. Pure GPU approach

LibMesh is FE Library and parallel computational framework. It can be run in parallel on thousands CPU cores thanks to it architecture, optimized code, employed MPI, and other functionalities that support parallel execution. Currently *libMesh*, as a part of *MOOSE Multiphysics*, is run on over 20 thousands cores at Idaho National Laboratory [10]. Furthermore it was shown that the applications scales incredibly well. Consequently the first approach to port *libMesh* to GPU assumed that one-to-one translation from CPU thread to CUDA thread will be kept – as such flow of each single GPU thread would be exactly the same as flow of CPU thread in original approach. Perspectives for this approach were extraordinary since even a single GPU would be able to handle hundreds thousands of threads, at cost of lower single thread performance of course, compared to just dozens thousands of threads currently used.

In the first iteration top-to-bottom approach of redesigning was employed. The selected starting point was *libMesh::ParallelObject* class. This effort approached several

issues soon after immediate children were considered. First of all CUDA[2] does not fully support class inheritance and polymorphism. In order to make member function accessible by GPU it has to be at least properly annotated; in certain cases it has to be also reimplemented to match mechanisms supported by CUDA. The problem is possible to solve however it would result in more complicated and less readable code. Furthermore some children classes, like *libMesh::System* and *libMesh::MeshBase*, have pointer based member variables; when such objects are passed to GPU memory, the member pointer still points to host memory. Since GPU operates on different memory address space each reference to pointer variable that points to host memory results in runtime error. To solve that problem all pointer member variables would have to be copied to GPU memory. Additional data transfers impact the performance, not to mention significant additional effort to provide implementation of hundreds post-allocation methods unique to each class with pointer-based member variables. Finally libMesh is highly object oriented application and therefore most of immediate children classes of *libMesh::ParallelObject* are also parent classes. A common practice in such scenario is to use so called virtual methods. These methods are not defined for parent classes since parent class may be too generic to implement any legitimate functionality of some methods. Children classes are more specialized and therefore supply some functionality related to these methods. This design pattern works great on CPUs but is not supported by GPUs and therefore code cannot be ported without complete redesign of library which is complicated using top-to-bottom approach.

[2] as of December 2012 – May 2013

Observations form first iteration led to conclusion that bottom-to-top design tactic may be more beneficial. This iteration started with *libMesh::Parallel::Sort* class. Unfortunately this time also major issues were encountered. Examined class, as well as many of libMesh objects, uses data containers from STL library. STL is a famous C++ library that provides wide selection of data structures with associated very efficient manipulation methods. Library is highly optimized and easy to use therefore it is crucial element in many C++ projects. Unfortunately CUDA does not support STL. In order to solve that issue the content of container would have to be copied to traditional array and then passed to GPU; result returned by GPU would be a classic array also, so content would have to be populate in STL container again. Another issue faced in this iteration was the frequency of branch statements in libMesh code. As mentioned in Chapter 1, GPU executes kernels in SIMT mode. It means that when even thread follows different and "longer" execution path, other threads are forced to stall until the next instruction common for all of them is reached. Although both of these issues are not critical to porting process, the performance of final result would be disappointing and most likely worse that the original one.

Although both design methodologies assessed in this chapter failed, it does not prove that one-to-one translation from CPU thread to GPU thread is impossible. It would require to completely redesign the library architecture and therefore enormous amount of time and work, which exceed the scope of this dissertation and capabilities of one person, would be needed. It is likely that Conversation with Cody Permann – one of *MOOSE Multiphysics* developers – seems to prove the observations. Moreover, significant change like that would result in altering existing API and therefore applications build atop of *libMesh* would have

to be modified. Because of that different porting idea, unfortunately less sophisticated, was proposed.

Chapter 2.2. Hybrid GPU/CPU approach

Second approach takes a step back from idea of enclosing very complex algorithms and complex data structures and turns to basic concepts behind GPGPU. CPU and GPU works in a host – device manner. It means that every GPU-enabled application is initialized and handled by CPU. CPU thread or threads follows a normal flow of application and periodically, when some part of algorithm is highly parallel, invokes a GPU kernels that execute selected operation in parallel. Figure 6 presents an example flow of program that performs SAXPY operation.

As it can be seen application starts with a single CPU thread initializes basic parameters and allocates necessary memory. Next CPU thread invokes GPU kernel that fills data structures with data which is later used to perform SAXPY operation. At the end CPU thread displays generated input and result of vector-scalar multiplication. It can be seen that kernels are intended to by highly specialized operation that produces a single element which is a part of bigger result. In serial application kernel would be the operation that is executed within a loop or block of nested loops. This encourages to use this approach to alter libMesh. In [10] it was showed that one of the most time consuming operations are basic linear algebra operations like matrix multiplication or vector addition – all of them executed by CPU in series using multiply nested loops. This approach would, basically, introduce another level of parallelism. Currently simulation starts with a single CPU thread

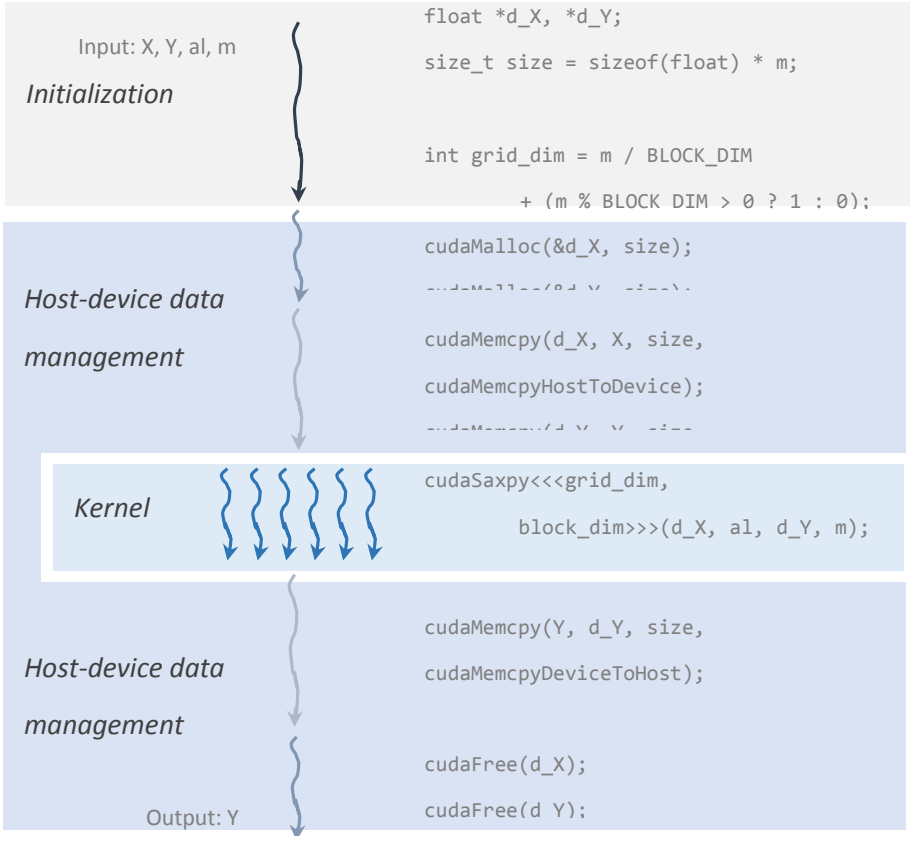


Figure 6. Standard CUDA program flow

that initializes simulation, set all required parameters, and spawns required number of child threads that executes simulation process in parallel. Finally parent gathers the results from child threads and produces the output. In the new approach child threads will have capability of invoking CUDA kernels in order to execute heavy parallel tasks – like BLAS operations – by GPU. In theory this concept may result in enhanced performance of overall multiphysics simulator, since the most time consuming part of simulation will be improved. Nonetheless it is important to remember that effectiveness of kernel depends on two factors: overhead related to data allocation and quality and level of parallelism of kernel.

Allocating and populating data structures in GPU memory is not an instant and resource-free operation. Therefore it is an overhead that has to be considered when comparing the execution time of CPU and GPU versions of the same operation. Kernel itself is a single operation, in this example generating and multiplying of floating point number, that are executed in parallel by GPU and in series (e.g. in a loop) by CPU. Depending on how time consuming is the operation and on how many operations have to be executed parallel approach can be more or less efficient. Because of that detailed evaluation of all implemented CUDA algorithm is required.

Chapter 2.3. CUDA 6.5 enhanced hybrid GPU/CPU approach

On August 2014 *NVidia* has released new CUDA programming model versioned with 6.5. Among number of improvements, like new code profiler and extended support to other programming languages, the most significant for research performed in this dissertation was including two libraries, namely *Thrust* and *cuBLAS*, to standard CUDA release. As mentioned in Chapter 1, *Thrust* library is GPU-enabled substitute of STL and *cuBLAS* is GPU version of BLAS. Both, as documentation and research done in this area, are very efficient and stabile libraries. Therefore it was reasonable to assess the performance of these libraries and decide on their usability in multiphysics simulation.

This approach is very similar to the one presented in previous subchapter: heavy computational and highly-parallel algebraic operators were replaced by *cuBLAS* operators in an analogous way they were replaced by custom operators presented in Chapter 2.1.

Complete list of used *cuBLAS* methods together with description of notions they use is presented in Algorithms chapter.

New feature of third approach to redesign *libMesh* is presence of *Thread* library. *LibMesh* relies heavily on STL containers and *Thrust* library extends the functionality of STL. The naming convention and majority of object and method names were kept in *Thrust* library. It allows to easily switch from one library to another simply by changing declaration of variable from STL container to *Thrust* container. New library provides also parallel execution on GPU of certain primitives related to container, like search and sort. It may be very beneficial for overall performance of *libMesh* because it operates on some container primitives. *Thrust* offers also possibility to access containers stored on GPU directly. It is very likely that host operations on data in GPU memory would be followed by the overhead related to data transfers over PCIe port, however there are cases when it may be irrelevant. When a container is frequently processed by GPU and rarely accessed by CPU then the overhead of retrieving data from GPU memory may be marginal when compared to overhead of frequent allocation from host to GPU memory. Potential improvements appeared to be so promising that in this approach each occurrence of STL library was replaced by corresponding container from *Thrust* library.

Chapter 2.4. Limitations

GPGPU was invented with intention to speed up processing in research facilities equipped with HPC. Obviously one GPU, even from the cutting-edge segment, would be unable to replace thousands of high-performance modes that consist of number of modern

CPU. Therefore graphic cards has to be used “in bulk”. This exposes engineers to several hardware design problems that hast to be overcome in order to provide not only better performance but also comparable fault-tolerance, resilience, and survivability of already existing classic high-performance clusters.

HPC systems keep scaling in volume together with more and more complex computations they perform. It is also accompanied, unfortunately, with escalation of failure frequency which may forfeit hours of computing of long-running applications. In classic supercomputer environments so called checkpoint/restart technique is used. The idea is to periodically save a state of execution on one or more reliable storage systems and in case of failure restore it and resume a normal flow. When classic CPU system is considered checkpoint/restart mechanism can be accomplished at three levels: kernel, library and application levels [46]. At kernel level the operating system, like *V-System* [47] or Charlotte [48], can spawn a process transparent to developer that can construct the state. At the library level library itself is responsible for providing checkpoint/recovery functionality. One of the examples is *Berkeley Lab Checkpoint/Restart* library which uses system calls to save registers content onto the stack [49]. Finally when the checkpointing and restarting mechanism is delivered at application level then developers need to implement such functionality into their product [50].

Chapter 3
Algorithms

Chapter 3.1. Proposed algorithms

a. Memory Allocation

CUDA framework offers various methods that allocate memory in device and pass data between host and device. Some of them are just wrapper functions around original CUDA allocation and transmission methods, other offers preprocessing steps or align data in memory to maximize performance of kernel. As it can be seen first memory is allocated using *cudaMalloc*, then one can copy specified amount of data from host memory (pointed by pointer to host memory) to device memory (pointed by pointer to device memory). At the end device memory is released using *cudaFree* method.

CuBLAS library offers its own methods that handle data allocation. Basically these method are just a wrapper functions that eventually uses standard *cudaMalloc*. However the advantage of *cuBLAS* methods is that data can be preprocessed before it is transferred to device. Thanks to the preprocessing user may for example transpose matrix before passing it to GPU within one method call.

The last considered method of handling the memory comes from *Thrust* library. The undisputed advantage of this approach is flexibility. Developer does not have to remember to manually allocate the memory and copy data because everything is done by simply assigning *thrust::host_vector* to *thrust::device_vector*. This functionality simplifies the process of developing and refactoring existing code.

b. AXPY

Level 1 of every BLAS library is reserved for vector operations like dot product or vector norms. In this research a classic AXPY operation – which is a simple vector-scalar multiplication summed with second vector – is considered. Operator is defined as $y = \alpha x + y$ where α is a scalar, x and y are vectors of equal dimension.

In standard CPU approach, presented in Algorithm 1, for every loop iteration equation (4) is calculated. Complexity of this algorithm is $O(n)$.

$$y[i] = \alpha \cdot x[i] + y[i] \tag{4}$$

Second considered algorithm was implemented in plain CUDA and is presented in Algorithm 2. Each element of output vector y is calculated by a single GPU thread so that i-th element is calculated by a thread given by $threadIdx.x$ and $blockIdx.x$ where $i = threadIdx.x + blockIdx.x \cdot blockDim.x$ using formula (4). Complexity of this algorithm, assuming parallel execution, is $O(1)$.

c. Vector Swap

Swap operator, also included in level 1 of BLAS library, switches the places of corresponding elements in two vectors. In CPU approach each pair of elements would be swapped in a single loop (Algorithm 3) using procedure (5). The complexity of this algorithm is $O(n)$.

$$v = y[i] \rightarrow y[i] = x[i] \rightarrow x[i] = v \tag{5}$$

CUDA equivalent of swap operator in presented in Algorithm 4. A grid of threads, equal in size to vectors, is overlaid on vectors and each GPU thread swaps a pair of elements from input vectors: i-th element is swapped using formula (5) by thread $threadIdx.x$ and $blockIdx.x$ where $i = threadIdx.x + blockIdx.x \cdot blockDim.x$ Complexity of this parallel algorithm is $O(1)$.

d. Vector Addition and Vector-Scalar Multiplication

In the simulation process very often a special cases of AXPY operator are used. Two of them are sum of two vectors and vector-matrix multiplication. Although the same result can be obtained using AXPY, this two operators perform twice less floating point operations and therefore is, in theory, twice faster. The output of vector-vector addition is defined by $y = x + y$. Serial approach performs the operation in a single loop; each iteration performs following operation (6). Scalar-vector multiplication is also executed in one loop; in each iteration (7). Algorithms are presented by Algorithm 5 and Algorithm 7. Complexity of both operators is $O(n)$.

$$y[i] = x[i] + y[i] \tag{6}$$

$$y[i] = \alpha \cdot y[i] \tag{7}$$

In corresponding GPU version of vector addition and vector-scalar multiplication, presented in Algorithm 6 and Algorithm 8 respectively. Each thread performs the same operation as single iteration of corresponding CPU algorithm; i-th element is calculated by thread $threadIdx.x + blockIdx.x \cdot blockDim.x$. Complexity of these algorithms is $O(1)$.

e. DOT Product

Another operator frequently used by FE libraries is DOT Product operation. DOT operator is here understand in algebraic manner. Operator takes two vectors and calculates sum of multiplied corresponding elements. Result is given by formula (8). CPU algorithm executes the operation in a loop in which it takes two corresponding elements from input vectors, multiplies them, and increment the result (Algorithm 9). Complexity of this algorithm is $O(n)$.

$$d = \sum_{i=1}^{n} x[i] \cdot y[i] \tag{8}$$

CUDA version of DOT Product is a little bit less straightforward. To calculate the result of DOT Product is necessary to traverse entire vector, therefore it is highly serial execution scenario; GPUs does not handle this type of problems, called gather operation, very well since parallel execution is rather enforced that applied. It is, however, possible to distribute the work among threads – in this case each thread will calculate multiplication

result of a single pair of element so that i-th pair will be processed by thread $threadIdx.x + blockIdx.x \cdot blockDim.x$. After that threads will atomically add the results to variable that will represent DOT Product value. Algorithm is presented in Algorithm 10; complexity of this algorithm, assuming parallel execution, is $O(1)$.

f. GEMV

Level 2 of BLAS library is responsible for matrix-vector operations including matrix-vector multiplication operator (GEMV) defined as $y = \alpha Ax + \beta y$. Serial GEMV (Algorithm 11) operator performs multiplication in doubly nested loop, therefore complexity of this algorithm is $O(n^2)$. Each iteration of outer loop is responsible calculating single element of output vector using (9) where sigma sign represents inner loop.

$$y[i] = \alpha \sum_{j=1}^{n} (A[i][j] \cdot x[j]) + \beta y[i] \tag{9}$$

Second algorithm, presented in Algorithm 12, is implemented in plain CUDA. In this method each element of output vector is calculated collectively by a number of threads. First each thread calculates (10), where $i = threadIdx.x + blockIdx.x \cdot blockDim.x$ and $j = threadIdx.y + blockIdx.y \cdot blockDim.y$, then results from a treads mapped to a single row in matrix A are summed which produces the final result (11).

$$y[i]_j = \alpha A[i][j] \cdot x[j] \tag{10}$$

$$y[i] = \sum_{j=1}^{n} y[i]_j + \beta y[i] \tag{11}$$

In theory algorithm would be executed in parallel (including summation of partial results) so its complexity is $O(1)$.

g. Matrix Swap

Another level 3 BLAS operator used frequently in FE libraries is swap operator. Similar to vector swap operator, operator takes two matrices and swap corresponding elements. Serial algorithm uses two nested loops; in inner loop corresponding elements in two matrices are swapped. Inner loop performs operation given by (12). Complexity of this method is $O(n^2)$; code is presented in Algorithm 13.

$$v = A[i][j] \rightarrow A[i][j] = B[i][j] \rightarrow B[i][j] = v \tag{12}$$

CPU algorithm performs the same operation with complexity of $O(1)$. Each pair of elements is swapped using single thread so that (i-th j-th) elements are swapped by thread given by $i = threadIdx.x + blockIdx.x \cdot blockDim.x$ and $j = threadIdx.y + blockIdx.y \cdot blockDim.y$. Code is presented by Algorithm 14.

h. Matrix-Scalar Multiplication and Matrix Addition

Analogous to SAXPY operator, scalar-matrix multiplication and matrix-matrix addition – a special case of GEMM method – are proposed. It this situation the complexity expressed in number of floating point operations is significantly smaller that complexity of GEMM operator because a matrix-matrix multiplication step is omitted. This reduces the number of floating point operations by 4 orders of magnitude. Serial algorithm performs the operations in doubly nested loop. Each loop travers matrix in one dimension; inner loop

calculate (13) for matrix addition and (14) for matrix-scalar multiplication. Complexity of both algorithms is $O(n^2)$. Algorithm 15 and Algorithm 17 presents matrix addition and matrix-scalar multiplication respectively.

$$A[i][j] = A[i][j] + B[i][j] \tag{13}$$

$$A[i][j] = \alpha \cdot A[i][j] \tag{14}$$

Corresponding CUDA algorithms perform the same operations as CPU algorithms. Each thread calculates single element in output matrix; (i-th j-th) elements are added or multiplied by thread given by $(i = threadIdx.x + blockIdx.x \cdot blockDim.x, j = threadIdx.y + blockIdx.y \cdot blockDim.y)$ thread using (13) and (14) respectively. Complexity of both CUDA algorithms, Algorithm 16 and Algorithm 18, is given by $O(1)$.

i. Matrix Transposition

Matrix transposition is very important operator in FE library. The operator reorder the elements so that i-th column is rewritten as i-th row. Classic CPU algorithm swaps element between two matrices within two nested loops; the inner loop assign (i-th j-th) element from input matrix to (j-th i-th) element of output matrix (15). Complexity of this algorithm is $O(n^2)$.

$$B[j][i] = A[i][j] \tag{15}$$

GPU algorithm – Algorithm 20 – performs the operation in very similar way: each singe thread is responsible for rearranging one element. Thread $(i = threadIdx.x +$

$blockIdx.x \cdot blockDim.x, j = threadIdx.y + blockIdx.y \cdot blockDim.y$ transpose (i-th j-th) element to (j-th i-th) element in output matrix (15). Complexity of this CUDA algorithm is $O(1)$.

j. GEMM

Level 3 of BLAS is a set of matrix-matrix operators. This level contains matrix-matrix multiplication called GEMM. Operator is given by formula $C = \alpha AB + \beta C$. Classic CPU algorithm consists of three nested loops what makes it algorithm of complexity $O(n^3)$. Inner loop calculates the value of (i-th j-th) element using formula (16), where i and j are number of iteration two outer loops are in Algorithm 21.

$$C[i][j] = \alpha \cdot \sum_{k=0}^{l}(A[i][k] \cdot C[k][j]) + \beta \cdot C[i][j] \quad (16)$$

In this experiment two CUDA matrix multiplication algorithms, presented in [11], are used. First of them, CUDA GEMM presented by Algorithm 22, is a simple parallel algorithm in which each element of output matrix is calculated by one thread. Element (i, j) is calculated using (16) by single thread where $i = threadIdx.x + blockIdx.x \cdot blockDim.x$ and $j = threadIdx.y + blockIdx.y \cdot blockDim.y$.

Second CUDA algorithm, Tiled CUDA GEMM, also designate single thread per one element in output matrix, however this version make a use of fast shared memory. Algorithm is presented by Algorithm 23. Threads are grouped in so called tiles and threads within a single tile cooperates to speedup execution. Problem of matrix multiplication is divided into sum of sub-matrices multiplication. Threads within one file copy data from

matrices in global memory to sub-matrices in shared memory; the sub-matrices are later multiplied. Each single thread within one iteration copies (i-th j-th) element, where $i = threadIdx.x + blockIdx.x \cdot blockDim.x$ and $j = threadIdx.y + blockIdx.y \cdot blockDim.y$, from input matrices in global memory to (k-th l-th) element of input submatrices in shared memory; when submatrices are populated with data thread calculates (k-th l-th) element in output submatrix using (17).

$$C[k][l] = \alpha \cdot \sum_{t=0}^{TILE\_DIM} A[k][t] \cdot B[t][l] \tag{17}$$

The output submatrices are later summed what gives a submatrix within a result of multiplication of matrix A and B. In theory complexity of both CUDA algorithms in $O(n)$.

k. Triangular GEMM

A special case of matrix multiplication is multiplication of two triangular matrices. Triangular matrix is a square matrix with zeros located above the main diagonal or below the low diagonal; such matrices are called left triangular and right triangular respectively. Linear algebra reports numerous properties of these matrices including, important for this research, result of multiplication of two left triangular or right triangular matrices is also left or right triangular matrix. This property simplifies the multiplication algorithm because almost half of the elements do not have to be calculated. Therefore serial CPU operator takes a form of Algorithms. It is very similar to CPU GEMM algorithm however first inner loop iterates through reduced number of elements. Although it does not affect overall Big-O complexity – it still remains $O(n^3)$ – the actual execution time will be shorter.

Algorithms, both left and right triangular version, are presented in Algorithm 24 and Algorithm 27 respectively.

Two CUDA triangular matrix multiplication operators are considered in this research. First of them, Simple CUDA Triangular GEMM (used in two versions that supports left- and right-triangular matrices), works in a way similar to CUDA GEMM – each element of output matrix is calculated by a single thread. The difference is that when elements comes from part of matrix filled with zeros thread terminates at that point. Both left and right triangular versions of algorithm are presented in Algorithm 25 and Algorithm 28. The complexity of these algorithms is $O(n)$.

Tiled CUDA Triangular GEMM also employs single thread per one element in output matrix and similar to Tiled CUDA GEMM uses of fast shared memory. Since triangular matrices are multiplied some operations can be skipped. In this case when whole tile of threads is supposed to calculate elements from segment of matrix that will be 0 then execution of these threads can be terminated immediately. Two version of this algorithm are presented in Algorithm 26 and Algorithm 29. The complexity of both CUDA algorithms is $O(n)$.

Chapter 3.2. cuBLAS operator

CuBLAS library offers a wide spectrum of operators including all three levels of classic BLAS and custom methods that implements special cases of more generic original BLAS methods. All of them are carefully designed and optimized by team of experienced

developers and researchers [13]. Presents *cuBLAS* methods that corresponds to algorithms designed and implemented in this dissertation.

Table 1. *cuBLAS* methods and corresponding operators

cuBLAS method	*Implements*	*Description*
cublasSaxpy	AXPY Vector-Scalar Multiplication[1] Vector Addition[2]	Method required following parameters: • *handle* – to handle library context • *alpha* – scalar value • *n* – size of vectors • *x* – first vector container • *incx* – stride between consecutive elements in vector *x* • *y* – second vector container • *incy* – stride between consecutive elements in vector *y*
cublasSswap	Vector Swap	Method required following parameters: • *handle* – to handle library context • *n* – size of vectors • *x* – first vector container • *incx* – stride between consecutive elements in vector *x* • *y* – second vector container • *incy* – stride between consecutive elements in vector *y*
cublasSdot	DOT Product	Method required following parameters: • *handle* – to handle library context • *n* – size of vectors • *x* – first vector container • *incx* – stride between consecutive elements in vector *x* • *y* – second vector container • *incy* – stride between consecutive elements in vector *y* • result – stores result of DOT operator
cublasSgemv	GEMV	Method required following parameters: • *handle* – to handle library context

		• *trans* – indicates whether matrix *A* will be transposes • *m* – first dimension of matrix *A* • *n* – second dimension of matrix *A* • *lda* – stride between consecutive elements in matrix *A* • *x* – vector container • *incx* – stride between consecutive elements in vector *x* • *y* – second vector container • *incy* – stride between consecutive elements in vector *y*
cublasSgemm	Matrix Transposition[3] Matrix-Scalar Multiplication[4] Matrix Addition[5]	Method required following parameters: • *handle* – to handle library context • *transa* – indicates whether matrix *A* will be transposes • *transb* – indicates whether matrix *B* will be transposes • *m* – first dimension of matrix • *n* – second dimension of matrix • *A* – first matrix • *lda* – stride between consecutive elements in matrix *A* • *B* – second matrix • *ldb* – stride between consecutive elements in matrix *B* • *C* – output matrix • *ldc* – stride between consecutive elements in matrix C
cublasSgemm	GEMM	Method required following parameters: • *handle* – to handle library context • *transa* – indicates whether matrix *A* will be transposes • *transb* – indicates whether matrix *B* will be transposes

	Triangular GEMM	• *m* – first dimension of matrix • *n* – second dimension of matrix • *A* – first matrix • *lda* – stride between consecutive elements in matrix *A* • *B* – second matrix • *ldb* – stride between consecutive elements in matrix *B* • *C* – output matrix • *ldc* – stride between consecutive elements in matrix C

[1] vector *x* needs to be filled with zeros
[2] *alpha* need to be set to 1
[3] matrix *B* needs to be filled with zeros, *alpha* needs to be 1, *transa* need to be set to transpose
[4] matrix *B* needs to be filled with zeros
[5] *alpha* and *beta* need to be set to 1

Chapter 4
Experiments

Chapter 4.1. Hardware configuration

Quadro line is a GPU brand designed by *NVidia* and developed by *PNY* and *NVidia*. This product line was designed for professional CAD and DCC market segment – as opposed to *GeForce* line which is designed almost exclusively for gaming [51]. The idea behind the *Quadro* line was to reduce the functionality of *GeForce* GPU that is important for gaming, like quality of textures and number of shaders, in favor of features crucial to CAD/DCC industry like high performance anti-aliased lines and two-sided lighting. In addition custom firmware and drivers, which support CAD applications better, for *Quadro* line were developed. *NVidia Quadro 5000,* which was used in this research, is built on introduced in March 2010, *Fermi* architecture [52].

Tesla line, named by Nikola Tesla, is a dedicated streaming and general purpose GPU brand designed and manufactured by *NVidia*. These devices are highly specialized hardware that are intended to be used in HPC centers. As of January 2015, TOP500 ranking

points out that three supercomputers out TOP 10 are built on *NVidia Tesla* GPUs [32]. Even though it is technically a graphical processing unit it was never meant to display images – *Tesla* GPUs, despite the most recent products, are not even equipped in display port. Used in this research *NVidia Tesla K40* is built on *Kepler* architecture – presented to public in 2012 [53].

Table 2. *NVidia Tesla K40* and *NVidia Quadro 5000* specification

Property	*NVidia Tesla K40*	*NVidia Quadro 5000*
Base clock	745 *MHz*	513 *MHz*
Processor cores	2880	352
Memory clock	3.0 *GHz*	750 *MHz*
Memory bandwidth	288 *GB/sec*	120 *GB/sec*
Interface	384-bit	320-bit
Total board memory	12 *GB* DDR5	2.5 *GB* DDR5
Board power	235 *W*	152 *W*
CUDA capability	3.5	2.0
Max dimension of grid of thread blocks	3	
Max *x*-dimension of a grid of thread blocks	2^{31}-1	65535
Max *y*-, *z*- dimensions of a grid of thread blocks	65535	
Max dimensionality of thread block	3	
Max x- or y-dimension of a block	1024	
Max z-dimension of a block	64	
Max number of threads per block	1024	

Warp size	32	
Max number of resident blocks per multiprocessor	16	8
Max number of resident warps per multiprocessor	64	48
Max number of resident threads per multiprocessor	2048	1536
Max amount of shared memory per multiprocessor	112 *kB*	48 *kB*
Amount of local memory per thread	512 *kB*	

The main difference, from technology point of view, is significantly bigger frequency of both base and memory clock and greater number of processor cores. Obviously it translates to higher performance of calculations; especially important is memory clock frequency since slow access to memory was a major factor that slows the execution in *NVidia Quadro 5000* GPU. It can be also seen that amount of memory was increased almost 5 times; yet the amount of memory per single core was decreased. Unfortunately the improvements caused higher power consumption [52] [53].

Worth to point out is the difference in CUDA capability between *Tesla K40* and *Quadro 5000*. Although the technology leap between 2.0 and 3.5 is not as significant as between 2.0 and 1.3 [52], the changes are significant. First and foremost the maximal *x*-dimension of block was enormously increased. This allows to map the logical structure of threads to big scale problems. It can be also noticed that the amount of fast shared memory assigned to multiprocessor was doubled. It may be related to doubling the number

of blocks that can resides at single SM at any time which; if this prediction is true then amount of memory per block remains almost the same. Nonetheless it will encourage developer to use this level of storage more often and therefore improve overall kernel execution efficiency [52] [53].

Chapter 4.2. Plan of experiment

a. Metric

One of the goals of this thesis is to evaluate the benefits of implementing modern GPGPU technology in multiphysics simulation software. Therefore an obvious step toward, was to assess the performance of proposed approach. Traditionally performance is measured I number of floating point operations per one second – *flops*. This metric is very popular in various brochures advertising new hardware solutions and in most cases it is valid way to evaluate capabilities of device.

In research however, including this thesis, flops metric is not the most fortunate way of describing performance mainly because it does not translate directly to time. The performance of GPU algorithm depends on a number of factors like frequency of synchronizations within a kernel, type of memory data is located in, alignment of this data in the memory, number of threads that execute the kernel and many others. Following example illustrates the disadvantage of *flops* metric.

Example: *Disadvantage of flops metric*

The task is to multiply two square matrices of size 2×2 filled with floating point numbers from range $[0, 1]$ using simple CUDA matrix multiplication algorithm. It can be observed that it is enough to run the kernel on a thread grid of size 2×2 which gives four threads encapsulated in one block. As a result each thread will calculate the value of one element in result matrix and perform four floating point operations. The task will be completed in time t_1. The performance expressed in flops is given by $\frac{4 \cdot 4}{t_1} = \frac{16}{t_1}$ *flops*.

GPU is, however, capable of running much more threads in parallel so one may consider increasing the number of threads assigned to that task. The common practice in this situation is to resize the input matrices to math desired dimensions of thread grid and fill it with zeros. Assuming that a grid of 32×32 threads will be assigned to perform that task, then both matrices will be resized to 32×32, each thread will calculate one element in result 32×32 matrix, and finally result will be trimmed back to size of 2×2 by removing previously added columns and rows. It can be calculated that each thread will execute 64 floating point operations so the performance is given by $\frac{32 \cdot 32 \cdot 64}{t_2} = \frac{65536}{t_2}$ *flops*. For such small scale problem $t_1 < t_2 < 1.01 t_1$ and the performance is humongous greater. That leads to contradiction since algorithm with better performance needs more time to execute the same task.

Multiphysics simulation software is just a tool for scientists and engineers and what matters for them is how soon and how accurate will be the results they get. The performance expresses in flops also does not indicate how well the algorithm will perform in different system with different hardware configuration. Because of that in this dissertation elapsed time would be that major evaluation criterions that describes the performance of proposed approach. Nevertheless flops metric, together with memory access metric, would also be used as a supporting factor that may allow to clarify aspects that are unable to be interpreted with time metric only.

b. Precision issue

As mentioned in previous paragraph [54] concerns regarding floating point operations precision were reported. GPU, as well as CPU, uses the same standard of notation and calculation – IEEE 754. Within this standard two types are distinguished: single precision numbers (*float*) and double precision numbers (*double*). No matter what type is used, general notation is very similar: first bit encodes sign, followed by exponent bits encoding the exponent offset[3] and bits encoding the fraction[4]. The architecture of IEEE 754 that tries to "fit infinite number of numbers onto finite number of bits" has to use some limitations; one of the limitations is the rounding step performed at the end of every arithmetic operation – result is rounded to the nearest number feasible to encode using designate number of bits. It is guaranteed than the result of any single basic arithmetic

[3] 8 bits for float and 11 bits for double
[4] 23bits for float and 52 bits for double

operation[5] on two numbers encoded using IEEE 754 will be exactly the same no matter what hardware accomplished it. Nevertheless the error related to frequent rounding step may appear since it is strongly related to level of parallelism and the order of operations. *NVidia* took a number of steps to obtain as good precision as possible including implementing into the hardware support for double precision type in devices with CUDA capability of 1.3 and so called fused multiply-add operation (multiplication operation performed on a result of addition is accompanied by a single rounding step) for GPUs that have CUDA capability of 2.0. Despite the improvements a common practice, applied also to this research, is to calculate the error by referring GPU result to expected result (usually calculated by CPU). In in this research the error is calculated for each repetition of experiment; errors are used to calculate Mean Square Root Error.

c. Experimental approach

The experiments $E =< A; S >$ carried out in this research can be described by following pattern: each entity A – which can be an algorithm, method, or a simulation – has to execute given set of experiment scenario S; execution of each experiment will be described with amount of time expressed in ms required to complete it $p_t(E)$ and also, if applicable, performance boost compared to CPU version, and MRSE metric. Scenarios were designed to uniformly cover the selected scope of interest, e.g. when two vectors are

[5] add, subtract, multiply, divide, square root, fused-multiply-add, remainder, conversion operations, scaling, sign operations, and comparisons

added the size of vectors changes from n_1 to n_2 with step $s = \frac{|n_2 - n_1|}{200}$. Details on each scenario are presented in section in which corresponding results are presented.

Whenever the subject of the research is to assess the performance of a hardware solution or a new algorithm and experiments are not carried out in dedicated server class environment, then results may be affected by nondeterministic occurrences or, simply speaking, noise. Typical sources of noise are all concurrently running applications, since they rivals with the experiment thread for resources, and operation system related processes that are meant to provide additional features which may not be required by the experiment threads, like maintaining GUI and some HID. Therefore a good practice is to disable all known and unused sources or possible noise. In this research whenever experiments were performed on regular desktop computer, the machine was running operating system in terminal mode, disabled network adapter, and all unnecessary services, like apache and database servers, turned off.

Even though careful preparation of experimental environment can greatly help with reducing separating results from some sources of interferences, it is crucial to remember that not all sources are known or can be deactivated. In this type of situations repeating a single experiment multiple times and applying statistic methods to obtained results may become very valuable. As such the following experimenting plan was used in the research:

- each experiment E was repeated at least n times
- results from ever experiments, namely experiment execution time $t(E)$ were gathered and formed a set RE

- 3% of the most extreme results from RE were discarded
- The average value of observed metrics and standard distribution were calculated and used as a representative experiment results

To select optimal values of parameters n a small experiment were performed: three CUDA algorithms with various complexity were executed for the same scenario for each $n \in \{10, 11, \dots, 1000\}$. The results showed that for n greater than 300 standard distribution of RE curve flattens and further increase in number of repetitions becomes cost ineffective. Therefore in this research the value of n was set to 300.

Chapter 5
Results

Chapter 5.1. Performance of Linear Algebra Operators

a. Memory Allocation

In this section performance of memory allocation primitives is evaluated. Set of algorithms A considered in this experiment is given by $\in \{Plain$ CPU Allocator, cuBLAS Allocator, Thrust Allocator$\}$; set of scenarios is given by $S = (A, B, C \in [0,1]^{n \times n} : n \in \{100, 150, \dots, 10\,000\})$.

The results presented in Figure 7 show that:

- together with growing size of data structures time required to allocate grows
- for each allocation methods linear growth of allocation time can be observed, however for matrices size of 5200×5200 slope gets steeper
- the shortest execution time is observed for Plain CUDA Allocator

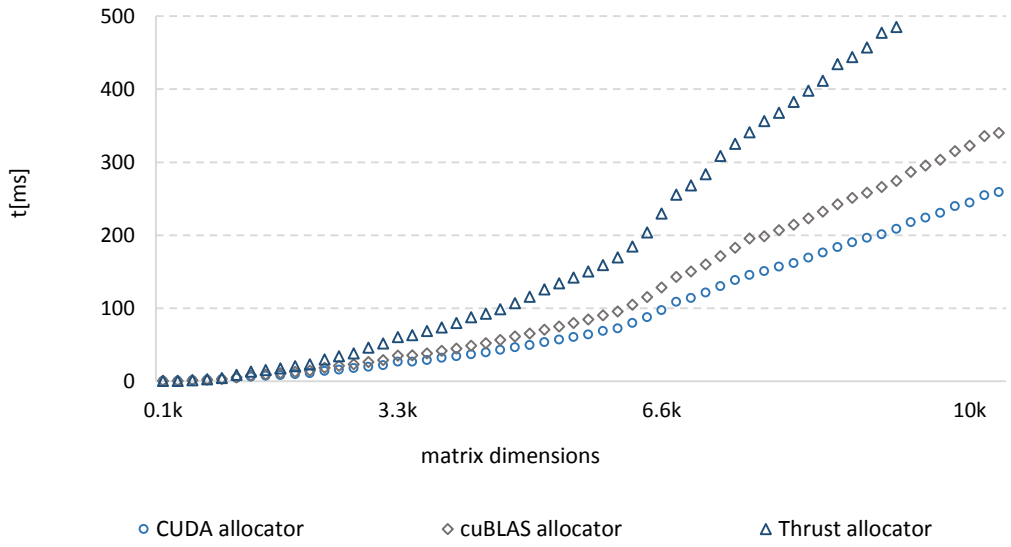


Figure 7. Allocation method execution time

- second best allocator is *cuBLAS* Allocator; it can be observed, however, that the difference in execution time between Plain CUDA Allocator and *cuBLAS* Allocator grows with size of problem

- *Thrust* allocation is the slowest and the execution time grows in faster pace than two other allocators

b. AXPY Operator

In this section performance of AXPY operator is evaluated. Set of algorithms A considered in this experiment is given by $\in \{\text{CPU AXPY}, \text{CUDA AXPY}, \text{cuBLAS AXPY}\}$; set of scenarios is given by $S = (\alpha \in [0,1]; x, y \in [0,1]^n : n \in \{100, 125, \dots, 15\,000\})$.

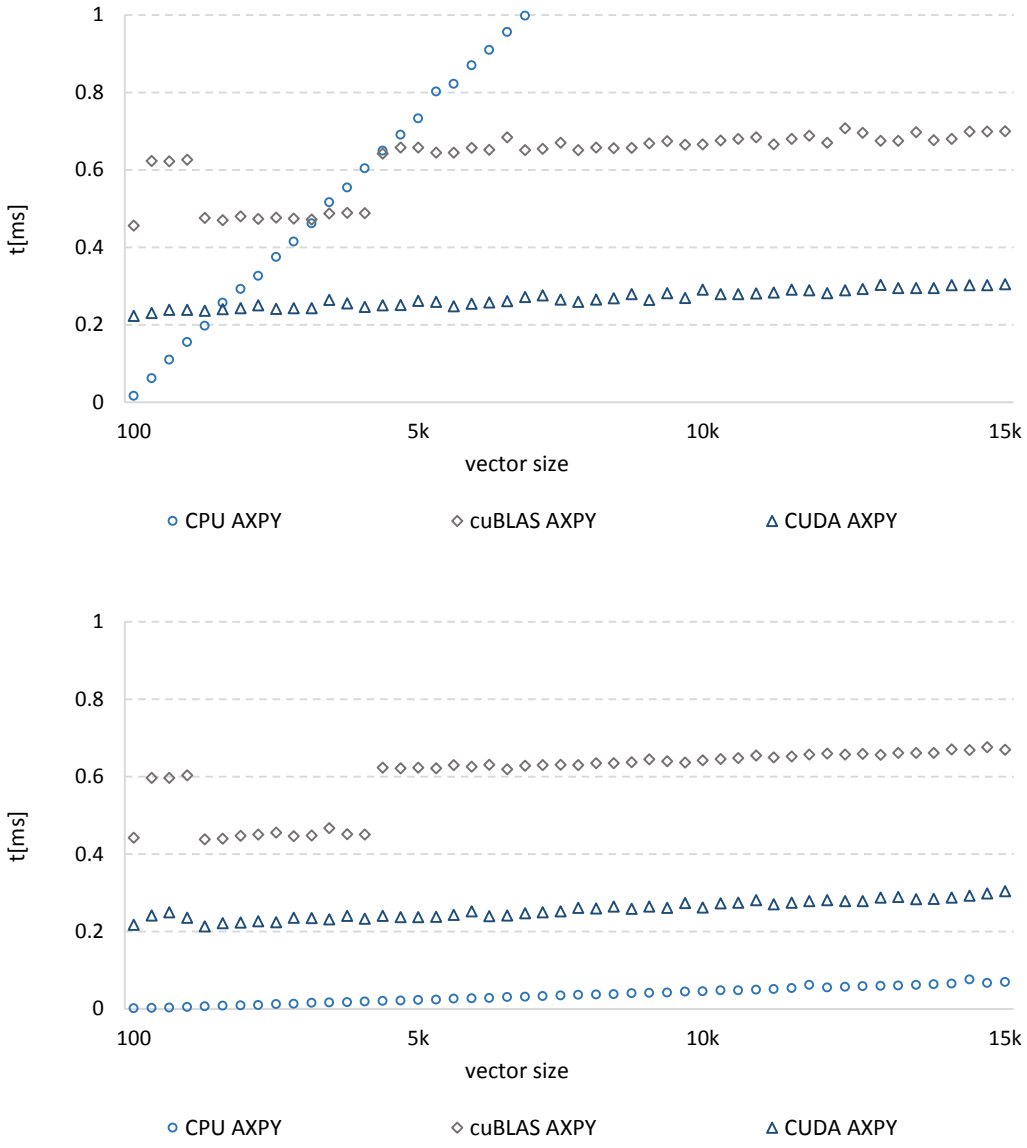


Figure 8. AXPY operator performance (*Thrust* containers on the top, array containers on the bottom)

The results, showed in Figure 8, show that for standard array container:

- there is small impact of problem size on algorithm performance; execution time grows for all algorithms, however the slope is rather gentle

- CPU algorithm is the most effective among all tested operators; second best was CUDA algorithm; *cuBLAS* operator was the worst
- for vector size of 1300 an execution time of *cuBLAS* algorithms decrease; later for vector size of 4300 it gets worse and follows the original trend
- the execution time of CPU method for vector size equals to 100 is 69 times greater than execution time for problem size of 15000
- for execution time of *cuBLAS* method for problem size is 1.51 times greater than execution time for problem size of 15000
- for execution time of CUDA method for problem size is 1.4 times greater than execution time for problem size of 15000

The results, showed in Figure 8, show that for standard *Thrust* container:

- GPU AXPY has slightly worse performance - about 3% - than the same algorithms using array of numbers
- CPU AXPYs suffers huge performance drop when *Thrust* container is used; the execution time grows proportionally to problem size
- CUDA AXPY is the most efficient algorithm; *cuBLAS* AXPY is worse than CUDA AXPY but it outperforms CPU approach for bigger problem sizes
- for vector size of 1300 an performance of *cuBLAS* algorithm increase however for vector size of 4300 it gets back to previous trend

- the execution time of CPU method for vector size equals to 100 is over 140 times greater than execution time for problem size of 15000
- for execution time of *cuBLAS* method for vector size equals to 100 is 1.54 times greater than execution time for problem size of 15000
- for execution time of CUDA method for vector size equals to 100 is 1.37 times greater than execution time for problem size of 15000

c. Vector Swap

In this section performance of Vector Swap operator is evaluated. Set of algorithms A considered in this experiment is given by $\in \{\text{CPU Vector Swap}, \text{CUDA Vector Swap}, \text{cuBLAS Vector Swap}\}$; set of scenarios is given by $S = (x, y \in [0,1]^n : n \in \{100, 125, \dots, 15\,000\})$.

The results, showed in Figure 9, show that for standard array container:

- there is minimal impact of problem size on algorithms performance
- execution time increase all algorithms with very slow peace
- CPU algorithm is the most effective operator in this experiment; second best was CUDA algorithm and third one was *cuBLAS* method
- for problem size of an execution time of GPU algorithms bumped up and after a while came back to follow the original trend

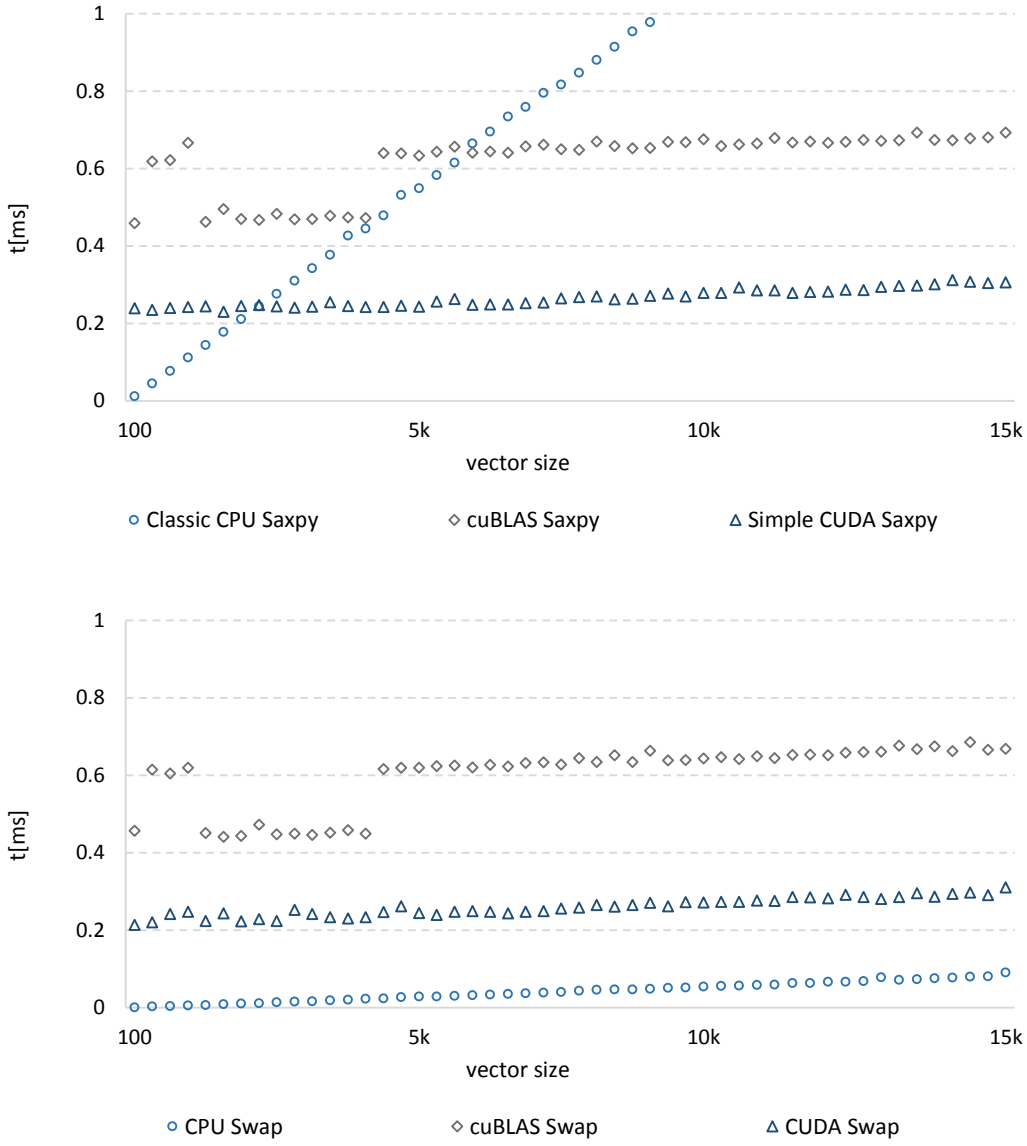


Figure 9. Vector Swap operator performance (*Thrust* containers on the top, array containers on the bottom)

- the execution time of CPU method for vector size equals to 100 is 91 times greater than execution time for problem size of 15000
- for execution time of *cuBLAS* method for vector size equals to 100 is 1.46 times greater than execution time for problem size of 15000

- for execution time of CUDA method for vector size equals to 100 is 1.45 times greater than execution time for problem size of 15000

The results, showed in Figure 9, show that for standard *Thrust* container:

- *Thrust* containers caused relatively small drop in performance for GPU-based algorithms; for both of them performance is about 4% worse
- CPU operator is significantly less effective when *Thrust* container is used; performance gets smaller linearly with growing problem size
- CUDA operator is the best method in this experiment; second place is taken by *cuBLAS* operator
- the execution time of CPU method for vector size equals to 136 is 137 times greater than execution time for problem size of 15000
- for execution time of *cuBLAS* method for vector size equals to 100 is 1.51 times greater than execution time for problem size of 15000
- for execution time of CUDA method for vector size equals to 100 is 1.28 times greater than execution time for problem size of 15000

d. Vector-Scalar Multiplication

In this section performance of Vector-Scalar Multiplication operator is evaluated. Set of algorithms A considered in this experiment is given by $\in \{\text{CPU Vector} -$

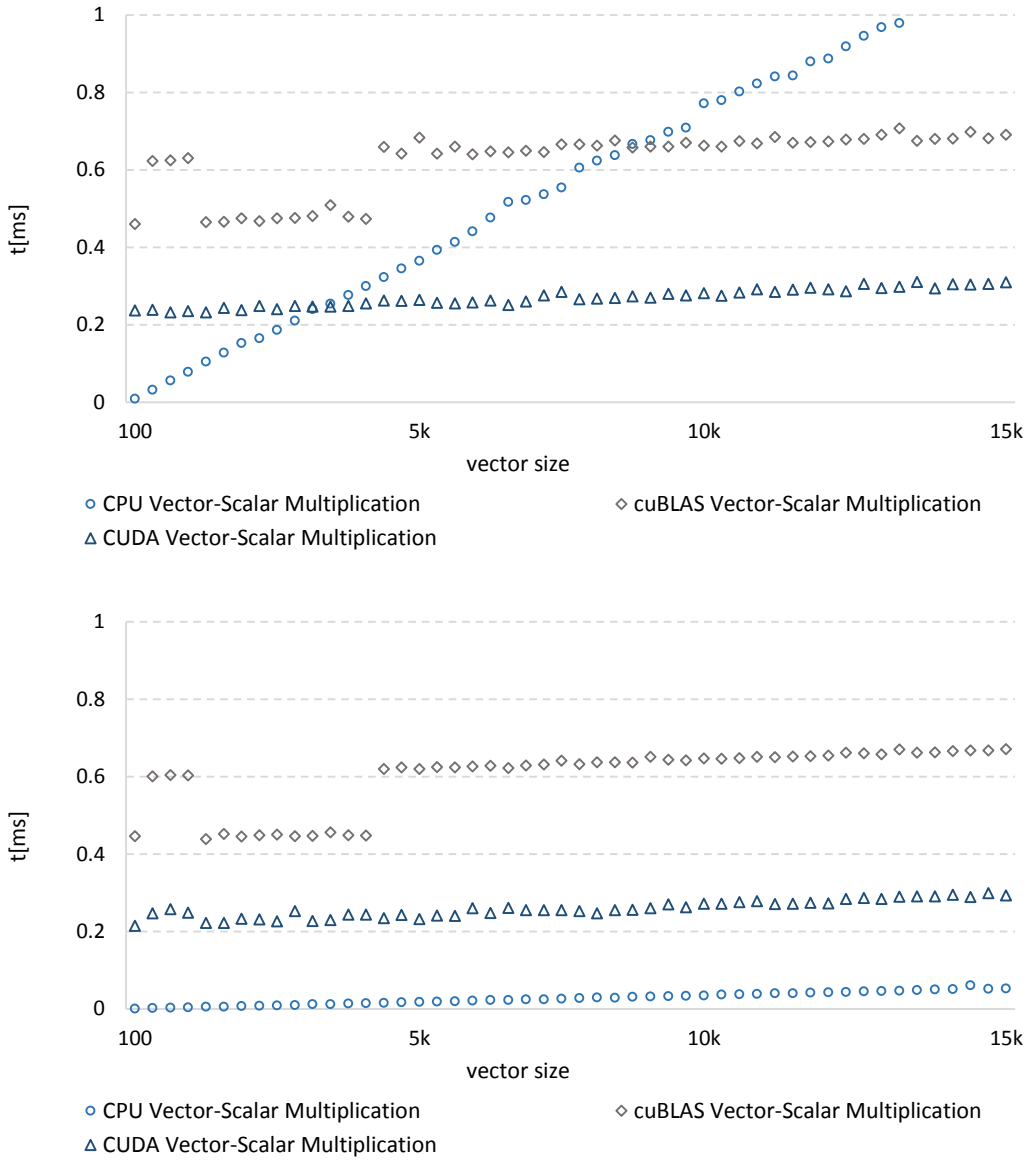


Figure 10. Vector-Scalar Multiplication operator performance (*Thrust* containers on the top, array containers on the bottom)

Scalar Multiplication, CUDA Vector − Scalar Multiplication, cuBLAS AXPY} ; set of scenarios is given by$S = (\alpha \in [0,1]; x \in [0,1]^n : n \in \{100, 125, \ldots, 15\,000\})$.

The results, showed in Figure 10, show that for standard array container:

- there is a small impact of problem size on algorithm performance however execution time of all methods increase linearly with increase in problem size

- CPU Matrix-Scalar Multiplication method is the most effective operator in this experiment; second is CUDA Matrix-Scalar Multiplication and the last one is *cuBLAS* Matrix-Scalar Multiplication

- for problem size of 1300 an execution time of GPU algorithms drops down and after a while came back to follow the original trend

- the execution time of CPU method for vector size equals to 100 is 53 times greater than execution time for problem size of 15000

- for execution time of *cuBLAS* method for vector size equals to 100 is 1.5 times greater than execution time for problem size of 15000

- for execution time of CUDA method for vector size equals to 100 is 1.36 times greater than execution time for problem size of 15000

The results, showed in Figure 10, show that for standard *Thrust* container:

- *Thrust* containers caused slower execution time; GPU-based algorithms were 6% less effective

- CPU operator is significantly less effective when *Thrust* container is used; performance gets smaller linearly with growing problem size

- CUDA Matrix-Scalar Multiplication is the best method in this experiment; second place is taken by *cuBLAS* Matrix-Scalar Multiplication
- the execution time of CPU method for vector size equals to 136 is 125 times greater than execution time for problem size of 15000
- for execution time of *cuBLAS* method for vector size equals to 100 is 1.5 times greater than execution time for problem size of 15000
- for execution time of CUDA method for vector size equals to 100 is 1.3 times greater than execution time for problem size of 15000

e. Vector Addition

In this section performance of Vector Addition operator is evaluated. Set of algorithms A considered in this experiment is given by $\in \{\text{CPU Vector} - \text{Scalar Addition}, \text{CUDA Vector} - \text{Scalar Addition}, \text{cuBLAS AXPY}\}$; set of scenarios is given by $S = (x, y \in [0,1]^n : n \in \{100, 125, \dots, 15\,000\})$.

The results, showed in Figure 11, show that for standard array container:

- problem size has marginal impact on performance of all methods; execution time increase together with increase of vector size for all algorithms
- CPU Vector Addition is the most effective operator in this experiment; second best was CUDA Vector Addition algorithm and third one was *cuBLAS* Vector Addition method

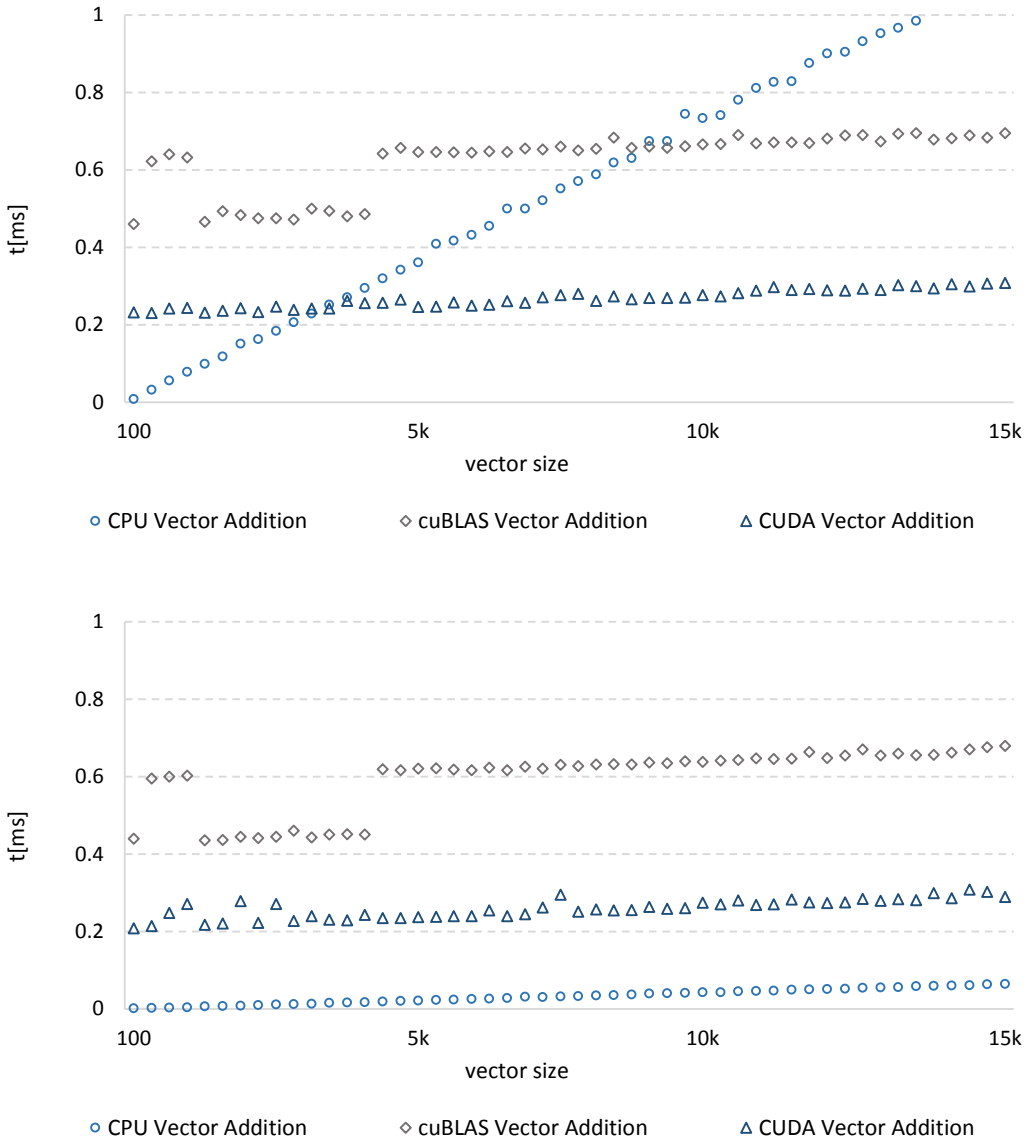


Figure 11. Vector Addition operator performance (*Thrust* containers on the top, array containers on the bottom)

- for problem size of an execution time of GPU algorithms bumped up and after a while came back to follow the original trend

- the execution time of CPU method for vector size equals to 100 is 64 times greater than execution time for problem size of 15000

- for execution time of *cuBLAS* method for problem size is 1.54 times greater than execution time for problem size of 15000
- for execution time of CUDA method for problem size is 1.39 times greater than execution time for problem size of 15000

The results, showed in Figure 11, show that for standard *Thrust* container:

- switching to *Thrust* containers resulted in approximately 5% worse performance of all CUDA algorithms
- CPU operator is significantly less effective when *Thrust* container is used; performance gets smaller linearly with growing problem size
- CUDA operator is the best method in this experiment; second place is taken by *cuBLAS* operator
- the execution time of CPU method for vector size equals to 100 is 137 times greater than execution time for problem size of 15000
- for execution time of *cuBLAS* method vector size equals to 100 is 1.51 times greater than execution time for problem size of 15000
- for execution time of CUDA method vector size equals to 100 is 1.32 times greater than execution time for problem size of 15000

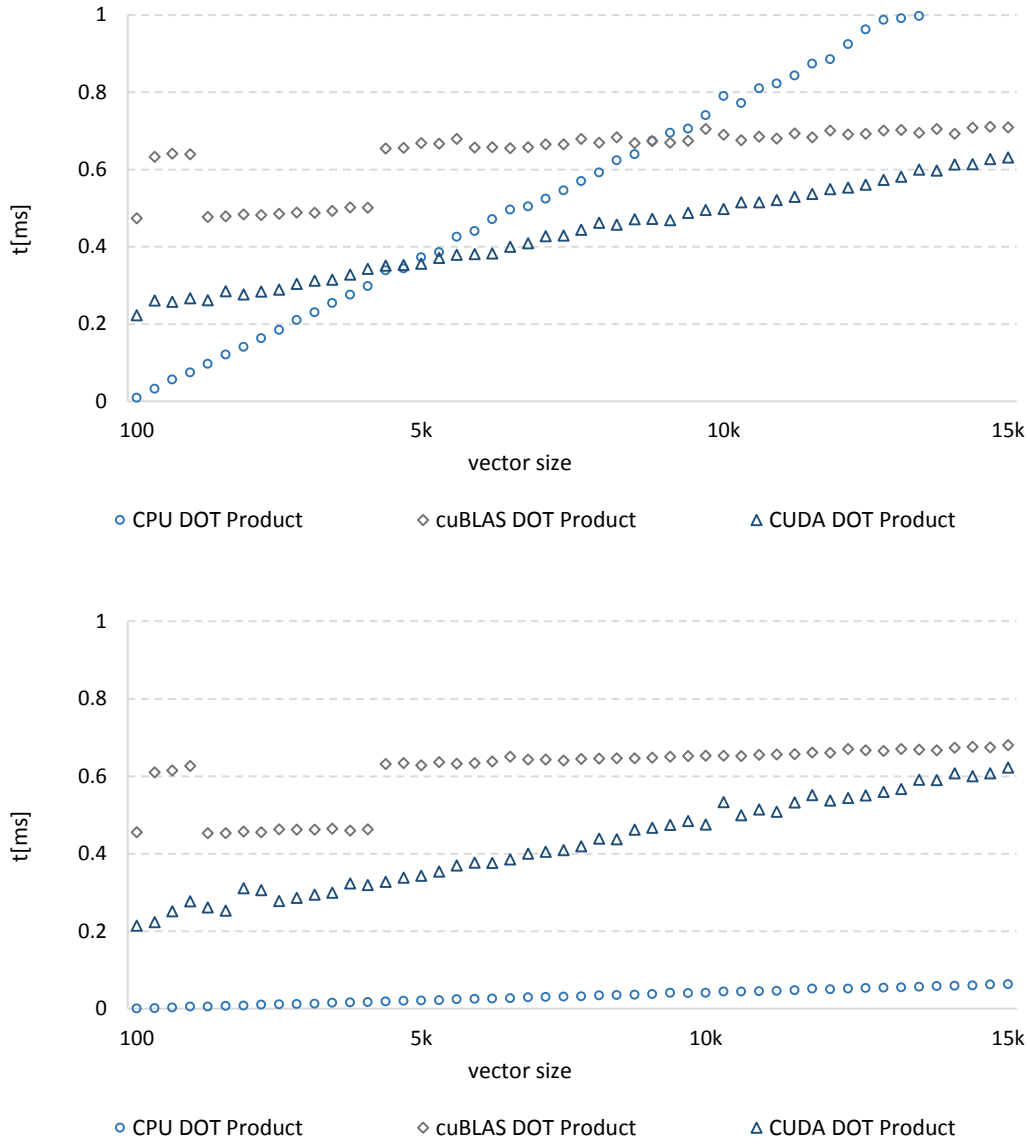


Figure 12. DOT Product operator performance (*Thrust* containers on the top, array containers on the bottom)

f. DOT Product

In this section performance of Dot Product operator is evaluated. Set of algorithms A considered in this experiment is given by $A \in \{\text{CPU DOT}, \text{CUDA DOT}, \text{cuBLAS DOT}\}$.

Scenarion examined in this research are given by $S = (x, y \in [0,1]^n : n \in \{100, 125, \dots, 15\,000\})$.

The results, showed in Figure 12, show that for standard array container:

- the most efficient algorithm is CPU DOT Product; execution time grows with time, however the slope is rather gentle: the difference between first and the last test case is 0.08 ms
- two remaining operators are worse than CPU operator for over 15 times for larger size matrices
- CUDA DOT operator execution time grows linearly with stepper slope that CPU DOT execution time; the difference between execution time of these two methods for vector size of 100 is 0.21 ms, whereas for vector size of 15 000 the difference is 0.07 ms
- the execution time of *cuBLAS* DOT operator also grows linearly for most of the experimental sets; it has the slowest increase pace: the difference in execution time for vector size equal to 400 and 15 000 is 0.22 ms
- for vector size of 1300 an execution time of *cuBLAS* algorithms decrease; later for vector size of 4300 it gets worse and follows the original trend

The results, showed in Figure 12, show that for standard *Thrust* container:

- switching to *Thrust* container has a little influence, about 2%, on performance of *cuBLAS* DOT and CUDA DOT operator

- the difference in execution time of *cuBLAS* DOT operator for vector size between 100 and 15 000 is 0.076 ms

- switching to *Thrust* data structures has significant impact on CPU DOT operator

- execution time of CPU algorithm grown linearly; for vector size of 4800 it gets outperformed by CUDA DOT operator; for vector size equals to 9000 *cuBLAS* DOT method becomes more efficient

- for vector size of 1300 an execution time of *cuBLAS* algorithm decrease; it gets back to previous trend when vector size reach size of 4300

g. GEMV

In this section performance of GEMV operator is evaluated. Set of algorithms A considered in this experiment is given by $A \in \{\text{CPU GEMV}, \text{Atomic CUDA GEMV}, \text{Plain CUDA GEMM}, \text{Tiled CUDA GEMM}, \text{cuBLAS GEMV}\}$; GEMV is a special case of GEMM operator when second matrix is a vector; set of scenarios is given by $S = (\alpha, \beta \in [0,1]; x \in [0,1]^n : n \in \{100, 125, \dots, 10\,000\}; A \in [0,1]^{n \times n} : n \in \{100, 125, \dots, 7\,000\})$.

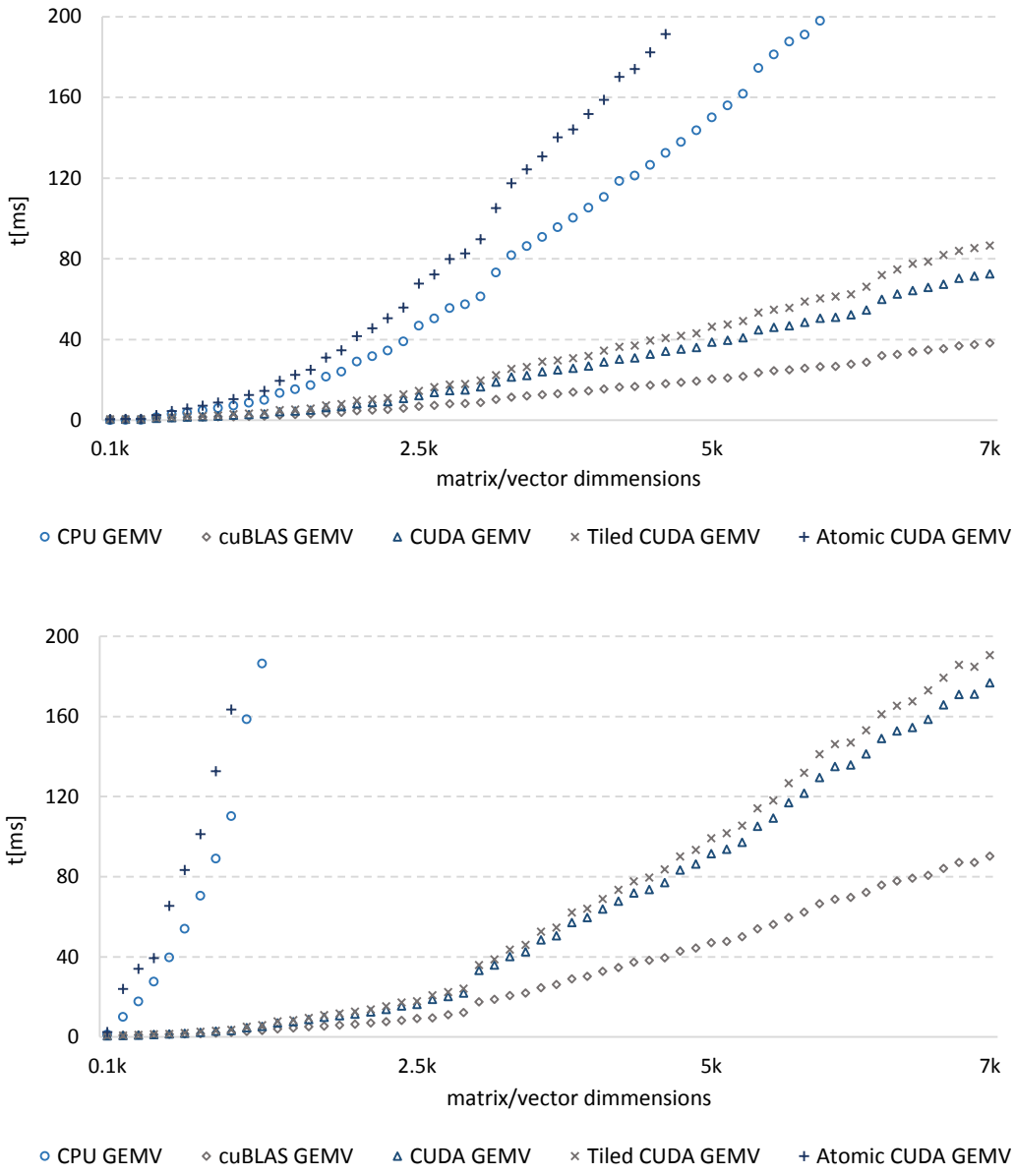


Figure 13. GEMV operator performance (*Thrust* containers on the top, array containers on the bottom)

The results, showed in Figure 13, show that for standard array container:

- the most efficient algorithms is *cuBLAS* GEMV operator

- second best operator is Plain CUDA GEMM operator; it is almost equally efficient as Tiled CUDA GEMM operator, yet Tiled CUDA GEMM is about 2% less effective

- execution time of *cuBLAS* GEMV, Plain CUDA GEMM, and Tiled CUDA GEMM can be described by polynomial

- Atomic CUDA GEMV and CPU GEMV operators are the least effective; execution time of these operators grows almost exponentially together with growing size of input matrix/vector

- for *cuBLAS* GEMV the difference in execution time between problem size of 25×25 and 7000×7000 is about 89 ms

- for Plain CUDA GEMM the difference in execution time between problem size of 25×25 and 7000×7000 is about 176 ms

- for Tiled CUDA GEMM the difference in execution time between problem size of 25×25 and 7000×7000 is about 190 ms

- for Atomic CUDA GEMM the difference in execution time between problem size of 25×25 and 7000×7000 is about 7.8 s

- for CPU GEMM the difference in execution time between problem size of 25×25 and 7000×7000 is about 5.5 s

The results, showed in Figure 13, show that for standard *Thrust* container:

- switching to *Thrust* containers resulted in significantly worse performance of CPU GEMV operator
- for CPU GEMM the difference in execution time between problem size of 25×25 and 7000×7000 is about 995 ms
- *Thrust* data structures did not cause big performance drop when applied to CUDA operators; a 4% increase of execution time can be observed

h. Matrix Swap

In this section performance of Matrix Swap operator is evaluated. Set of algorithms A considered in this experiment is given by $A \in \{\text{CPU Matrix Swap}, \text{CUDA Matrix Swap}\}$; set of scenarios is given by $S = (A, B \in [0,1]^{n \times n} : n \in \{25, 50, \dots, 5\,000\})$.

The results, showed in Figure 14, show that for standard array container:

- CUDA Matrix Swap was the most effective algorithm in this experiment
- execution time of both operators can be described by polynomial, however execution time of CPU operator grows with greater speed
- for CUDA Matrix Swap operator the difference in execution time between problem size of 25×25 and 5000×5000 is about 56 ms

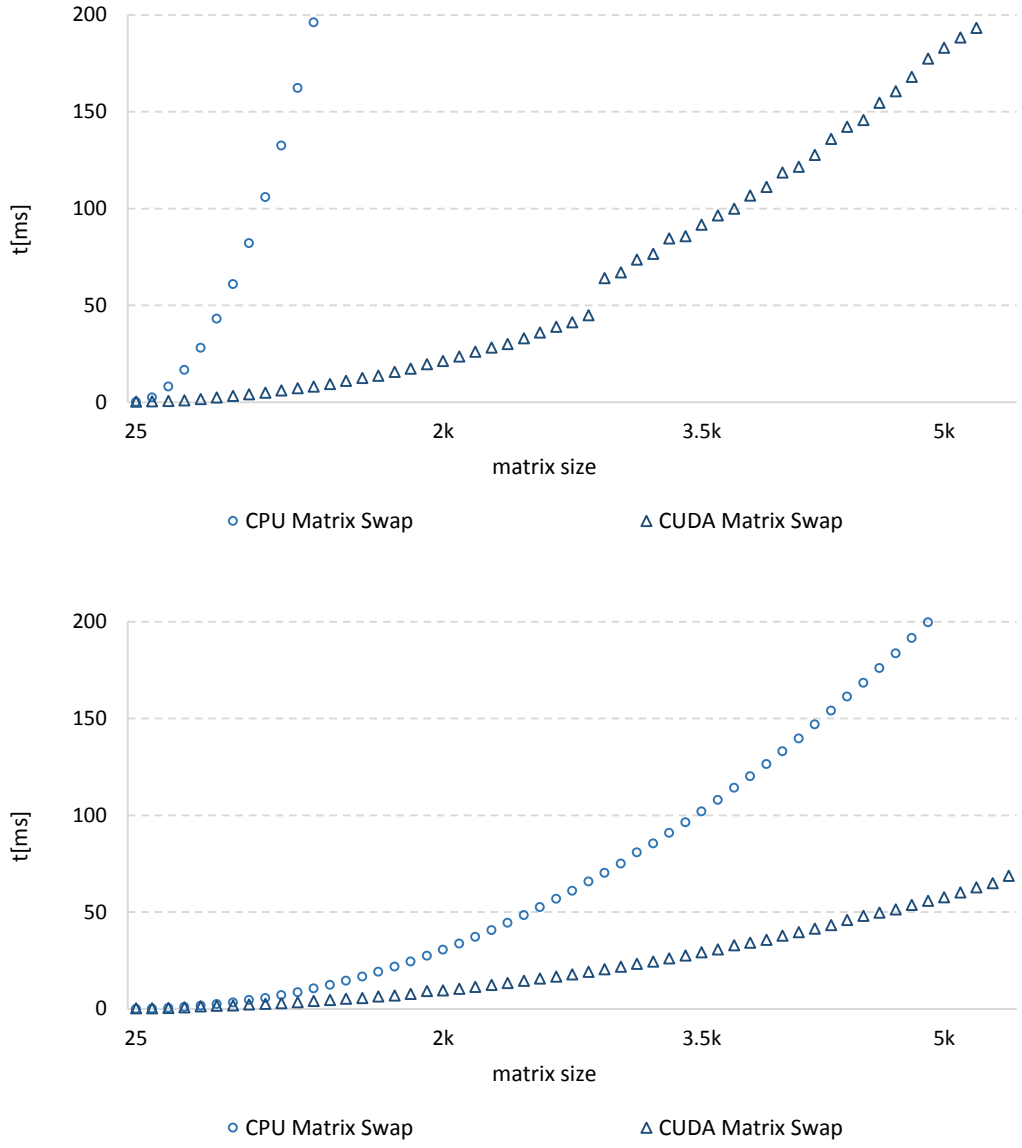


Figure 14. Matrix Swap operator performance (*Thrust* containers on the top, array containers on the bottom)

- for CPU Matrix Swap operator the difference in execution time between problem size of 25×25 and 5000×5000 is about 204 *ms*

The results, showed in Figure 14, show that for standard *Thrust* container:

- after *Thrust* container was applied to evaluated methods each of them experienced much longer execution time

- lower performance is especially visible for CPU Matrix Swap operator since its execution time grows almost exponentially

i. Matrix Addition

In this section performance of Matrix Addition operator is evaluated. Set of algorithms A considered in this experiment is given by $A \in \{\text{CPU Matrix Addition}, \text{CUDA Matrix Addition}, \text{cuBLAS GEAM}\}$; set of scenarios is given by $S = (A, B \in [0,1]^{n \times n} : n \in \{25, 50, \dots, 5\,000\})$.

The results, showed in Figure 15, show that for standard array container:

- all operators have similar performance until matrix size is smaller than 800×800

- when the matrices get bigger than 800×800 the execution time of all algorithms follows polynomial growth

- the nest performance was achieved by CUDA Matrix Addition operator; second best operator is *cuBLAS* operator; the least effective is CPU Matrix Addition method

- for CUDA Matrix Addition the difference in execution time between problem size of 25×25 and 5000×5000 is about 62 ms

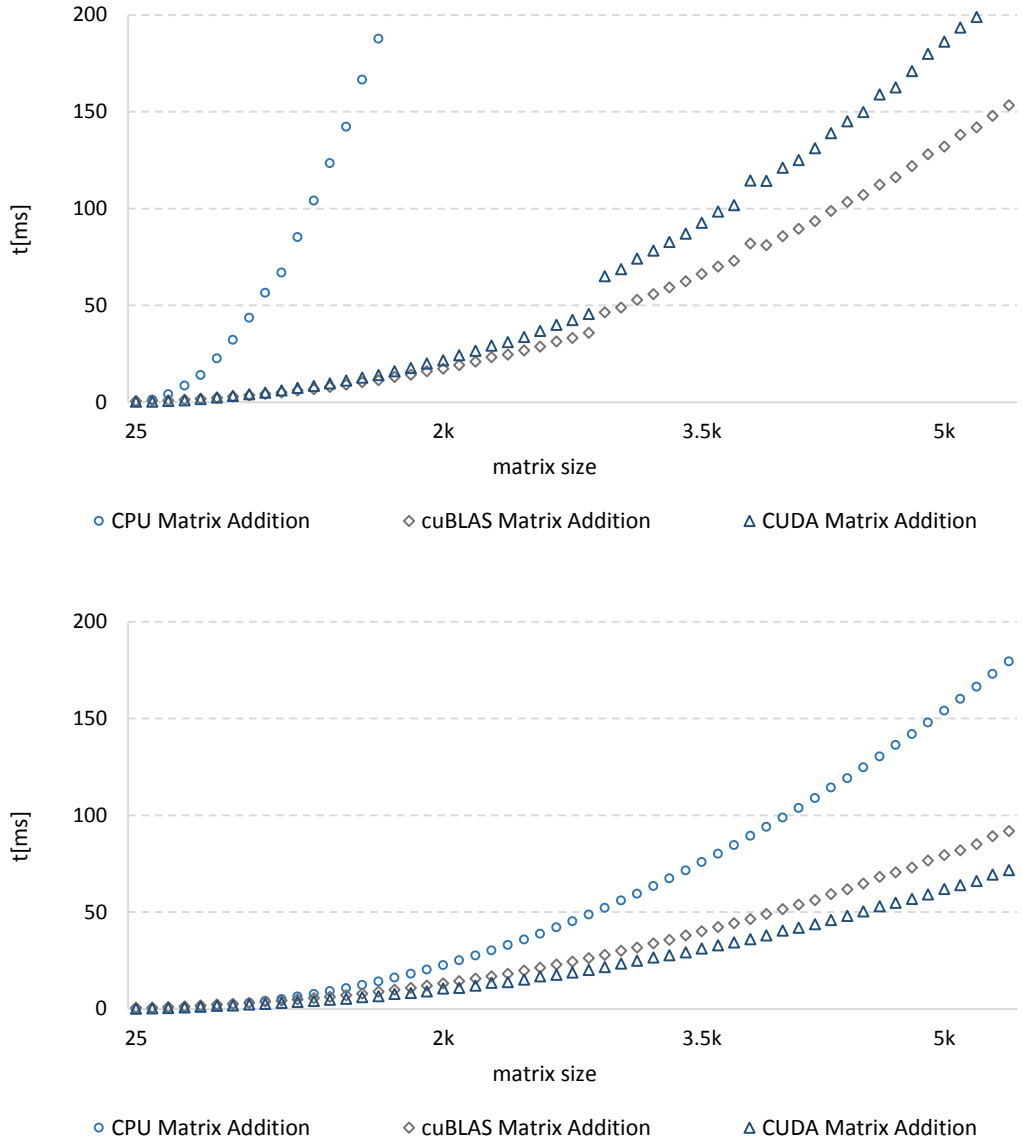


Figure 15. Matrix Addition operator performance (*Thrust* containers on the top, array containers on the bottom)

- for CPU Matrix Addition the difference in execution time between problem size of 25×25 and 5000×5000 is about 154 ms
- for *cuBLAS* Matrix Addition the difference in execution time between problem size of 25×25 and 5000×5000 is about 78 ms

The results, showed in Figure 15, show that for standard *Thrust* container:

- switching to *Thrust* containers had very negative influence on CPU operator; the execution time started to grow in almost exponential manner
- applying *Thrust* data structures to GPU-based operators caused 50% drop in performance; because of that *cuBLAS* operator outperformed CUDA Matrix Addition algorithm

j. Matrix-Scalar Multiplication

In this section performance of Matrix-Scalar Multiplication operator is evaluated. Set of algorithms A considered in this experiment is given by $A \in \{\text{CPU Matrix} - \text{Scalar Multiplication}, \text{CUDA Matrix} - \text{Scalar Multiplication}, \text{cuBLAS GEMM}\}$; set of scenarios is given by $S = (\alpha \in [0,1]; A \in [0,1]^{n \times n} : n \in \{25, 50, \dots, 5\,000\})$.

The results, showed in Figure 16, show that for standard array container:

- the execution time of all operators is similar especially for matrix sizes smaller than 725×725
- for bigger problem sizes execution size of all operators starts to grow in polynomial manner, but performance is still relatively similar
- the best algorithm evaluated in this experiment was CUDA Matrix-Scalar Multiplication method; *cuBLAS* GEAM operator was ranked second; the worst performance was achieved by CPU Matrix-Scalar Multiplication

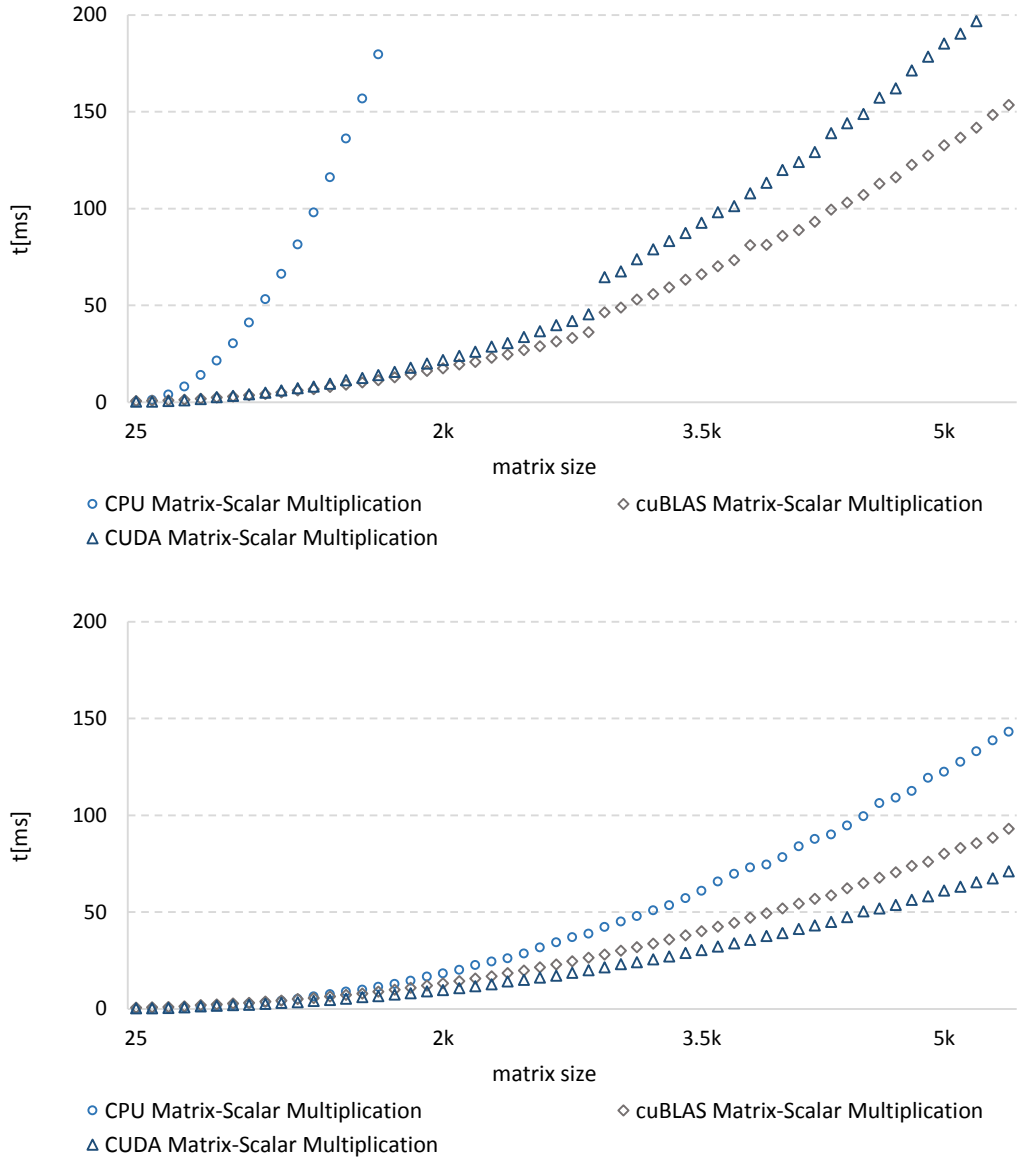


Figure 16. Matrix-Scalar Multiplication operator performance (*Thrust* containers on the top, array containers on the bottom)

- for CUDA Matrix-Scalar Multiplication the difference in execution time between problem size of 25×25 and 5000×5000 is about 59 ms
- for CPU Matrix-Scalar Multiplication the difference in execution time between problem size of 25×25 and 5000×5000 is about 120 ms

- for *cuBLAS* Matrix-Scalar Multiplication the difference in execution time between problem size of 25×25 and 5000×5000 is about 77 ms

The results, showed in Figure 16, show that for standard *Thrust* container:

- execution time of CPU algorithm is strongly affected by *Thrust* containers; it grows almost exponentially together with increasing matrices size
- for CPU Matrix-Scalar Multiplication the difference in execution time between problem size of 25×25 and 5000×5000 is about 2 s
- replacing classic arrays with *Thrust* containers had a smaller, but still significant, impact on execution time of CUDA operators; it was, however, sufficient to alter the ranking of operator: *cuBLAS* GEAM operator is more effective than CUDA Matrix-Scalar Multiplication

k. Matrix Transposition

In this section performance of Matrix Transposition operator is evaluated. Set of algorithms A considered in this experiment is given by $A \in \{\text{CPU Matrix Transposition}, \text{CUDA Matrix Transposition}, \text{cuBLAS Matrix Transposition}\}$; set of scenarios is given by $S = (A \in [0,1]^{n \times n} : n \in \{25, 50, \dots, 5\,000\})$.

The results show, showed in Figure 17, that for standard array container:

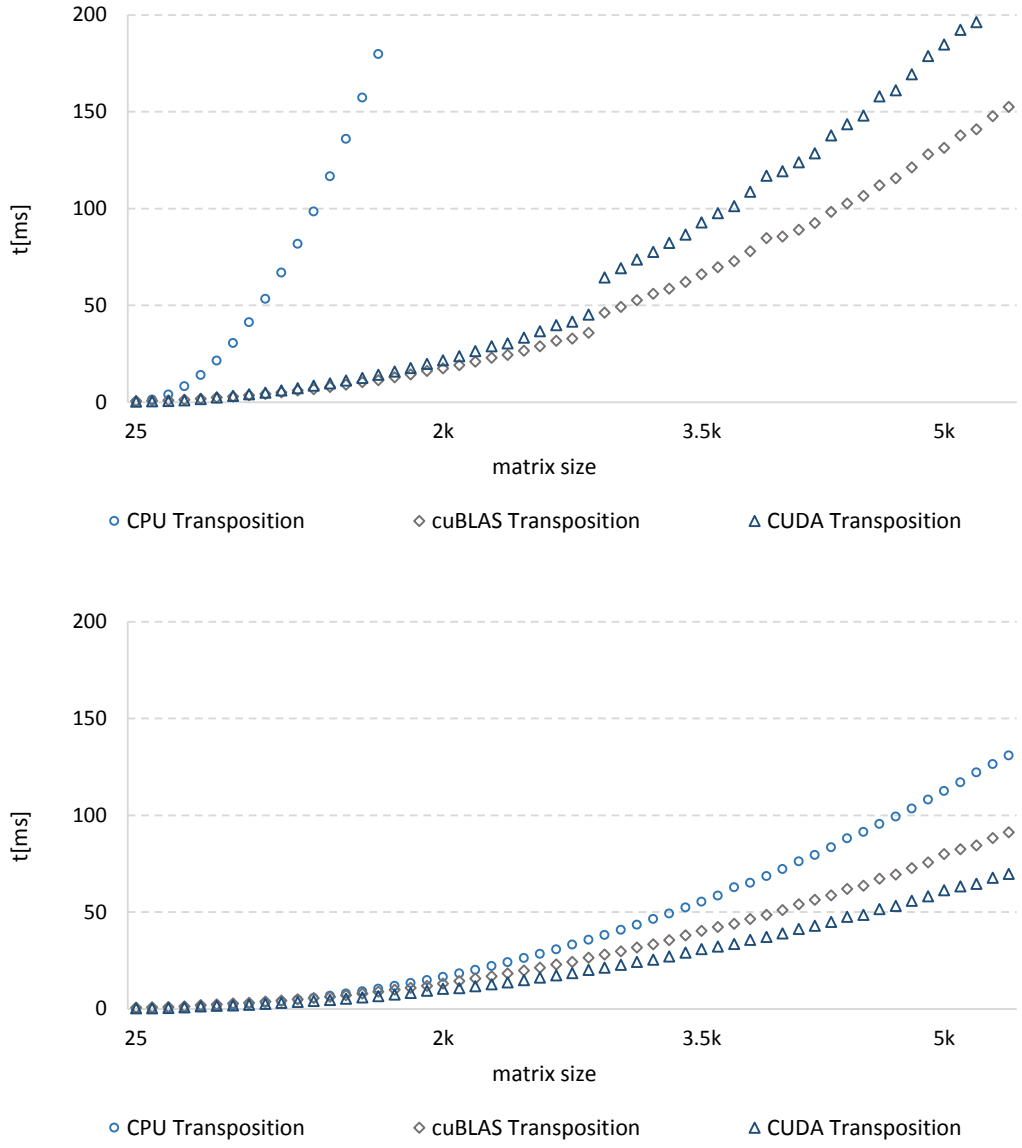


Figure 17. Matrix Transposition operator performance (*Thrust* containers on the top, array containers on the bottom)

- the most efficient operator examined in this experiment is CUDA Matrix Transposition operator; second best operator is *cuBLAS* Matrix Transposition; CPU operator is the least effective

- execution time of each operator grows in different pace however each of them follows polynomial trend
- for CUDA Matrix Transposition the difference in execution time between problem size of 25×25 and 5000×5000 is about 59 ms
- for *cuBLAS* Matrix Transposition the difference in execution time between problem size of 25×25 and 5000×5000 is about 76 ms
- for CPU Matrix Transposition the difference in execution time between problem size of 25×25 and 5000×5000 is about 111 ms

The results, showed in Figure 17, show that for standard *Thrust* container:

- after *Thrust* data structures were applied to methods each of them experiences significantly slower execution time
- longer execution time is especially visible for CPU Matrix Transposition operator since it follows exponential trend
- in this scenario *cuBLAS* operator is more efficient than CUDA Matrix Transposition method

I. GEMM

In this section performance of GEMM operator is evaluated. Set of algorithms A considered in this experiment is given by $A \in \{\text{CPU GEMM},$

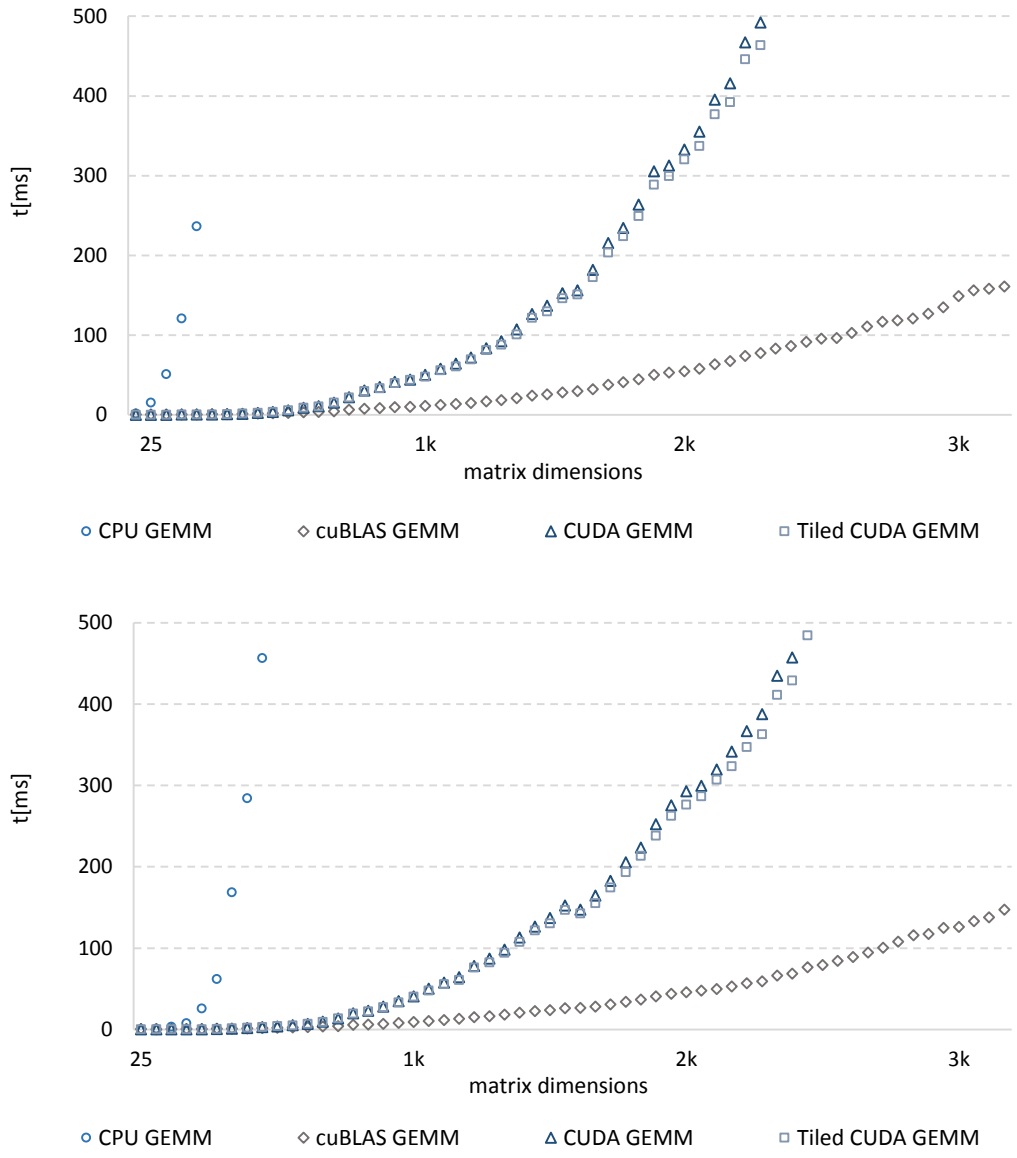


Figure 18. GEMM operator performance (*Thrust* containers on the top, array containers on the bottom)

Simple CUDA GEMM, Tiled CUDA GEMM, cuBLAS GEMM}; set of scenarios is given by $S = (\alpha, \beta \in [0,1]; A, B \in [0,1]^{n \times n} : n \in \{25, 50, \ldots, 3\,000\})$.

The results, showed in Figure 18, show that for standard array container:

- the most efficient algorithm is *cuBLAS* GEMM; the difference in execution time of *cuBLAS* GEMM and any other GEMM operator grows together with size of the problem

- linear growth of execution time for *cuBLAS* GEMM can be observed; the difference in execution time between problem size of 25×25 and 3000×3000 is about 160 ms

- next two best algorithms are Tiled CUDA GEMM and Plain CUDA GEMM; they were almost equally effective throughout the whole experiment; Tiled CUDA GEMM was better, however, for about 3%

- for Tiled CUDA GEMM the difference in execution time between problem size of 25×25 and 3000×3000 is about 995 ms

- for Plain CUDA GEMM the difference in execution time between problem size of 25×25 and 3000×3000 is about 1048 ms

- the CPU operator was the least efficient; its execution time grows almost exponentially

- for CPU GEMM the difference in execution time between problem size of 25×25 and 3000×3000 is about 5 min

The results show, showed in Figure 18, that for standard *Thrust* container:

- *Thrust* containers, when applied to GPU-based operators, resulted in about 3% increase of execution time
- it did not, however, affect the pace in which execution time grows with increasing problem size
- switching to *Thrust* containers caused humongous drop in performance for CPU GEMM operator; for large scale matrices, 3000×3000, the execution time is almost an hour

m. Triangular GEMM

In this section performance of Triangular GEMM operator is evaluated. Set of algorithms A considered in this experiment is given by $A \in \{\text{CPU Triangular GEMM}, \text{Simple CUDA Triangular GEMM}, \text{Tiled CUDA Triangular GEMM}, \text{cuBLAS GEMM}\}$; set of scenarios is given by $S = \{\alpha, \beta \in [0,1]; A, B \in [0,1]^{n \times n} : n \in \{25, 50, \dots, 3\,000\}; A[i][j] = B[i][j] = 0 : j < i < n\}$.

The results, showed in Figure 19, show that for standard array container:

- performance of CPU Triangular GEMM is very poor; execution time of this algorithm grow almost exponentially and therefore it makes it unusable for large scale matrices
- execution time of all GPU algorithm can be described by polynomial growth; for small size problems, up to matrix size of 525×525, it is very similar among

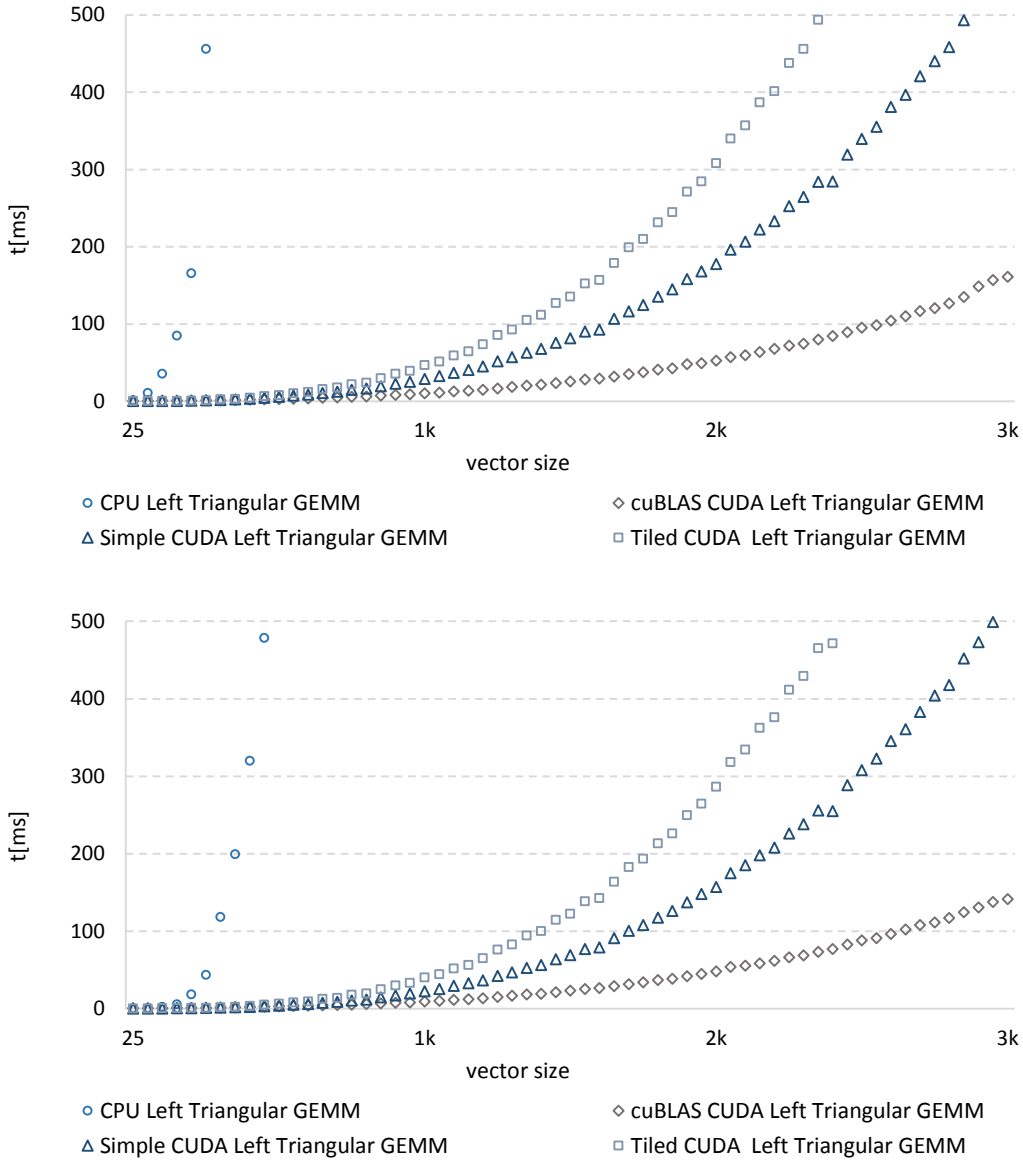


Figure 19. Triangular GEMM operator performance (*Thrust* containers on the top, array containers on the bottom)

all of these operators; after that size execution time starts to grow with different speed for different operators

- Tiled CUDA Triangular GEMM – the worst of GPU operators – needed about 8 ms to multiply matrices of size 512×512; to multiply matrices of size 3000×3000 it needed almost a 1 s

- CUDA Triangular GEMM operator is second best operator; it needed 3 ms to multiply two matrices of size 525×525; to multiply two matrices of size 3000×3000 it needed 567 ms

- the most efficient operator – *cuBLAS* GEMM – has the gentlest slope function that describes its execution time

- for 525×525 matrices *cuBLAS* GEMM needed 5.5 ms to complete operation; for 3000×3000 execution time was 161 ms

The results, showed in Figure 19, show that for standard *Thrust* container:

- switching to *Thrust* containers had humongous impact of CPU Triangular GEMM; its execution time grows even faster compared to standard array container

- using *Thrust* containers had almost on impact on GPU methods; this scenarios shows analogous behavior of GPU operators to the one presented for classic array containers

- for Tiled CUDA Triangular GEMM the difference in execution time for matrix size between 25×25 and 3000×3000 is 989 ms

- for Simple CUDA Triangular GEMM the difference in execution time for matrix size between 25×25 and 3000×3000 is 560 ms

- for *cuBLAS* GEMM the difference in execution time for matrix size between 25×25 and 3000×3000 is 158 ms

Chapter 5.2. Performance of Multiphysics simulation

To examine the effectiveness of the proposed hybrid approach, an experiment was performed. Performance of the proposed method was compared to the performance of the standard CPU approach. In the experiment two multiphysics heat conduction problems, diffusion and diffusion with convection were considered. A weak form of the PDE that describes the convection diffusion problem (and therefore diffusion only also) is given by the equations below.

$$-\nabla \cdot k\nabla u + \beta \cdot \nabla u = f \tag{18}$$

Where k is diffusivity, β is the vector field of velocity, $-\nabla \cdot k\nabla u$ is diffusion, and $\beta \cdot \nabla u$ is convection. After moving f to the left side of equation, multiplying both sides by the shape function ψ, and integrating the equation over the domain Ω, equation (19) is obtained [20].

$$-\int_{\Omega} \psi(\nabla \cdot k\nabla u) + \int_{\Omega} \psi(\beta \cdot \nabla u) - \int_{\Omega} \psi f = 0 \tag{19}$$

Applying the divergence theorem to equation (19) transforms it to equation (20).

$$\int_{\Omega} \nabla\psi \cdot k\nabla u - \int_{\partial\Omega} \psi(k\nabla u \cdot \hat{n}) + \int_{\Omega} \psi(\beta \cdot \nabla u) - \int_{\Omega} \psi f = 0 \qquad (20)$$

When represented in terms of multiphysics kernels and boundary conditions, equation (20) has the following form [20].

$$(\nabla\psi, k\nabla u) - \langle\psi, k\nabla n \cdot \hat{n}\rangle + (\psi, \beta \cdot \nabla u) - (\psi, f) = 0 \qquad (21)$$

Where $(\nabla\psi, k\nabla u)$, $(\psi, \beta \cdot \nabla u)$, and (ψ, f) are multiphysics kernels and $\langle\psi, k\nabla n \cdot \hat{n}\rangle$ is a boundary condition [20].

Set of algorithms A considered in this experiment is given by $A \in \{\text{CUDA Hybrid}, \text{Thrust Hybrid}, \text{cuBLAS Hybrid}, \text{Thrust \& cuBLAS Hybrid}\}$ where CUDA Hybrid represents *libMesh* with underlying algorithms that were designed in second approach, *Thrust* Hybrid represents *libMesh* in which all STL containers were replaced by *Thrust* containers, *cuBLAS* Hybrid represents *libMesh* with underlying *cuBLAS* operators, and finally *Thrust-cuBLAS* Hybrid represent *libMesh* with *Thrust* data structures and *cuBLAS* operators. Set of scenarios is given by $S = \{M \times P : M \in Mesh^{n\times n} : n \in \{32, 64, 128, 256, 512\}, P \in \{Diffusion, Diffusion \,\&\, Convection\}\}$ where M is sealed steel cylinder given by mesh of size $n \times n$ and P represents multiphysics simulation.

Table 3 presents the performance of proposed solutions expressed as ration of $\frac{t_{hybrid(P)}}{t_{libMesh}(P)}$ where $t_{libMesh}(P)$ is duration of simulation P executed by classic *libMesh* implementation, $t_{libMesh}(P)$ is duration of simulation P executed by one of proposed hybrid GPU/CPU *libMesh* implementation.

Table 3. Hybrid simulators performance

Mesh size	*Physics*	*Speed-up factor of Hybrid approach*			
		CUDA	*Thrust*	*cuBLAS*	*Thrust & cuBLAS*
32×32	Diffusion	0.974	0.921	0.989	0.903
64×64	Diffusion	0.986	0.782	0.992	0.778
128×128	Diffusion	1.012	0.698	1.019	0.712
256×256	Diffusion	1.025	0.612	1.031	0.698
512×512	Diffusion	1.036	0.562	1.049	0.612
32×32	Diffusion & Convection	0.979	0.928	0.992	0.915
64×64	Diffusion & Convection	0.994	0.798	0.998	0.773
128×128	Diffusion & Convection	1.007	0.721	1.013	0.753
256×256	Diffusion & Convection	1.018	0.676	1.027	0.713
512×512	Diffusion & Convection	1.031	0.598	1.042	0.682

The results shows that:

- for CUDA Hybrid and *cuBLAS* Hybrid greater performance gain follows bigger problem sizes
- for small size problems, 64×64 and below, none of the proposed modifications resulted in performance gain

- when problem size gets bigger than 64×64, CUDA Hybrid and *cuBLAS* Hybrid executes diffusion simulation faster than original *libMesh* implementation by 1.2% and 1.9% respectively

- when problem size gets bigger than 64×64, CUDA Hybrid and *cuBLAS* Hybrid executes diffusion & convection simulation faster than original *libMesh* implementation by 0.7% and 1.3% respectively

- when *Thrust* library was applied to *libMesh*, for both proposed approaches that involves new data containers significant drop in performance was observed; the performance of simulation were decreasing together with growing problem size

- for problem size of 512×512 Thrust Hybrid and *cuBLAS* & Thrust Hybrid are slower than original *libMesh* implementation by over 30%

Chapter 6
Discussion

a. Parallel BLAS algorithms

The results show that the overhead, related to highly object oriented architecture of *Thrust* data structures and wrapper nature of methods, has serious negative impact on overall performance of memory allocation operators. Both CPU and plain CUDA implementations outperforms *Thrust* operators. It can be also observed that effectiveness of algorithms (both serial and parallel) with *Thrust* data structures were worse than effectiveness of corresponding non-*Thrust* versions. Despite worse performance considered library provides extremely usable and flexible data structures. *Thrust* containers are easy to use, greatly increases code readability, simplifies code refactor, and speed up the development process. Therefore a small decrease in performance may be worth the robustness offered by *Thrust* library.

The research confirmed the enormous effectiveness of both CUDA and *cuBLAS* methods. The scenario was similar for all algorithms for which corresponding CPU operators have complexity of $O(n^2)$ and higher. First, when dimensions of

matrices/vectors were small, CPU outperform all GPU algorithms. It happens because of overhead related to data transfers between host and device and because of relatively low-performance of GPU cores. However increasing dimensions cause exponential growth of CPU algorithms execution time and therefore they are no longer competitive. In the same time execution time of CUDA algorithms grows linearly. When execution time of plain CUDA algorithms are compared to equivalent *cuBLAS* methods it can be observed that plain CUDA operators are more effective for smaller scale of problem. It is because of the overhead caused by wrapper nature of *cuBLAS* methods. Nevertheless the difference is getting smaller when matrix/vector dimensions are getting bigger and eventually *cuBLAS* operators outpace plain CUDA algorithms. As the experience shows in real life when developers decide to consider GPGPU the scale of problem is already big enough to not focus on corner cases when serial algorithm may be more efficient.

A different situation, however, can be observed for all CPU operators described by complexity of $O(n)$. The complexity of these algorithms is so insignificant that an effort to make it parallel seems to be pointless. As the research shows when a regular data structures are used CPU outperforms all GPU approaches. Furthermore even when two GPU approaches are compared it appears that the algorithm that the overhead may be more important than design of the algorithm itself. This observation points out how the performance of hardware and overhead related to design can affect the effectiveness of algorithm for relatively small scale of problem.

b. Multiphysics simulation

Results show that switching from STL container to *Thrust* containers has very negative influence on performance of proposed hybrid GPU/CPU multiphysics simulator. Even when efficient *cuBLAS* operators work atop of *Thrust* data structures the overall performance of simulator is very disappointing. Moreover together with growing problem size the performance drop gets bigger. It is related to overhead that accompanies every operation that involves any data manipulation within a *Thrust* data structures. Therefore it is not recommended to replace STL container with *Thrust* containers at least until additional research is done. Further research would have to evaluate how increasing the spectrum of used *Thrust* parallel operators affects the performance of simulation. Currently, when the only used *Thrust* operators are accessors to element enclosed in data structures, the negative impact on performance of simulation is unacceptable.

As it can be seen in cases in which the number of points was relatively small, CPU approach outperforms the proposed Hybrid approach (in each version). This is caused by an additional steps that are related to the GPU/CPU approach. The *libMesh* data structure has to be translated to standard arrays, memory on GPU has to be allocated, and data has to be copied. Therefore, although execution of the kernel may be faster than the multiplication process performed by CPU, together with the mentioned overhead, the overall performance is worse. Nevertheless, when the number of points grows, the proposed approach is more efficient. For meshes with 128×128 and more, the hybrid approach reduces the total simulation time by over 1.03 (3%) and 1.05(5%) for CUDA Hybrid and *cuBLAS* Hybrid respectively. It can be also observed that with the increase in

number of points the speed-up factor grows. Therefore it can be assumed that the performance increase would be even greater for a mesh that consists of more than 1024×1024 points. A reasonable idea to test this hypothesis would be to implement a mechanism that tracks simulation and record profiling. These outcomes could then be used to decide whether to use the hybrid or CPU approach.

The obtained results also show that the complexity of multiphysics phenomena has an influence on performance. For the same model the speed-up factor obtained for a simple diffusion phenomena is definitely larger than for the convection diffusion phenomena. Therefore, it may be assumed that for complex multiphysics problems, the difference in time consumption between matrix multiplication and other parts of the code is getting smaller. As a result, in the worst-case scenario, the speed-up factor will be very close to one. However it will never drop below one so the hybrid approach would be at worst as good as the CPU approach.

Conclusion

The research carried out and presented in this thesis focuses almost exclusively on the multiphysics simulation performance in the purest and the most literal meaning of that word. It is because there is still a little research done in this area and, especially, in porting existing CPU-based systems to GPU-enabled systems. Topic covered in the thesis was the obvious choice to assess the potential legitimacy of applying GPGPU technology and, in case of being wrong, invalidity of using it.

The most important part of the multiphysics simulation is to solve PDEs. For larger problems these equations have to be solved simultaneously on HPC in order to finish the calculation in reasonable time. In this paper, a strategy of dealing with multiphysics problems is presented by a hierarchical software framework. Examples of the systems are *MOOSE, COMSOL,* and *ANSYS*. In this research architecture of these systems was presented and described. The code analysis and documentation review allowed to observe that both commercial and academia solutions follow a generic pattern architecture that consist of multiphysics models, parallel computational framework, Finite Element library, and set of PDE solvers. In first layer multiphysics problem is defined, second layer

initializes simulation and preprocess input data, third layer runs solvers and processes corresponding mesh, finally last layer solves system of partial differential equations.

The simulation process is very time consuming and requires a lot of computational power. Because of that even when multimode high-performance supercomputer is applied the simulation may take even few days or weeks. Moreover the cost of upgrading the hardware is not proportional to performance gain. Therefore an attempt to use different computational technology, namely GPGPU, was made. Because *MOOSE Framework* and all of its layers are open source and because both *MOOSE* and underlying layers – namely *libMesh*, *PETSc*, and *Trilinos* – are widely used in academia and business, they were selected to the research.

Code analysis shows that the lowest layer – PDE solvers – are already efficient applications capable of being executed by classic CPU-based supercomputers and also machines that takes advantage of newer technologies like GPU and FPGA. FE Library layer however, despite capability of being executed in parallel by number of CPU nodes, is not ready to be ran by GPUs and for that reason it becomes a bottleneck. Because of that *libMesh* the most suitable candidate to be redesigned and reimplemented to run on GPU.

First approach to taken to move the simulation to GPU environment assumed that each CPU thread will be mapped one-to-one to GPU thread; as such Parallel Multiphysics Framework would spawn numerous GPU kernels instead of GPU threads. This approach would significantly increase parallelism of execution since even single GPU is able to run hundred thousands of threads simultaneously what exceeds capabilities, understood as number of parallel threads not performance, of most supercomputers. This approach lead

to dead end since there are some issues with porting the whole framework to GPU. The biggest of them is the architecture of *libMesh*. *LibMesh* is a highly object oriented application written in C++. Investigation of the code reveals that the library strongly relies on highly hierarchical abstract classes and polymorphism which were nor well supported by GPU. Furthermore *libMesh* code that is executed by single CPU thread is very complex in terms of branch operations. Because GPU execute kernels in SIMT pattern, in which each GPU thread enclosed in warp executes the same instruction at the same moment, branches have significant negative influence on performance. *LibMesh* uses also STL containers. The main problem is that STL containers use pointers extensively. When STL data structure is transferred to GPU memory addresses are copied instead. This results in error because GPU tries to refer to host memory which cannot be accessed directly by GPU. To solve the aforementioned issues *libMesh* architecture would have to be completely redesign the architecture of *libMesh*, however it would be an extremely time consuming and expensive project, not to mention that results still may be unsatisfying. Because of this, a hybrid approach that uses GPU and CPU is proposed in this paper.

Second approach to enhance multiphysics simulation performance assumed porting to GPU only the most time consuming and highly parallel parts of libMesh. In this approach, named hybrid GPU/CPU, the application flow is very similar to original one: single CPU thread initializes simulation and spawns child threads, child threads work together via MPI and when done they return results to parent thread. The difference is when ported to GPU part of code is approached. In this situation child thread calls a GPU kernel thread that executes heavy computational part of application. Code reviews and profiling reveals that the best candidates to be reimplemented in GPU-enabled manner are linear

algebraic operators. To assess validity and performance boost preliminary research was conducted – selected operators were implemented in CUDA and their effectiveness were referred efficiency of corresponding CPU algorithms. The results proved outstanding performance of GPU approach; parallel were multiple times more efficient than GPU algorithms and the difference grew with size of problem. When implemented to multiphysics simulator the time required to finish the simulation was almost 18% shorter than original version.

In third approach features from new CUDA programing model, namely *Thrust* and *cuBLAS*, were used. *Thrust* is the GPU version of STL whereas *cuBLAS* is GPU-enabled version of BLAS library. Since the work done in previous step resulted in custom GPU BLAS library, it seemed obvious that it should be referred to *cuBLAS*. All algorithms implemented in previous approach were reimplemented using *cuBLAS* methods and a number of experiments were conducted to evaluate the performance. As the results show *cuBLAS* was more efficient in certain cases especially for large scale problems. Therefore multiphysics simulator was updated to use more efficient *cuBLAS* operators instead of previously used GPU algorithms. In addition all instances of STL containers were replaced by *Thrust* containers so that some container-related methods, like sorting, could be executed by GPU. Upgrade to CUDA 6.5 brought even greater performance boots: the newest version of hybrid GPU/CPU approach was 5% faster than previous one and over 13% faster than original CPU version.

The investigation, research, and obtained results answer the research questions asked at the beginning of the dissertation: it is possible to redesigned existing multiphysics

simulator to run on GPU; changes does not affect existing applications that were build atop of it; furthermore GPU version may be more efficient than CPU version. Nevertheless stating that the new approach is undoubtedly beneficial and that it is safe to proceed with this approach may be risky. The main concern is relatively small performance gain which may be easily compromised by other factors that has to be considered before approach is implemented in large-scale and expensive supercomputer.

One of them is fault-tolerance and resilience of application. In current state any random failure of even one thread results in termination of kernel and, as a result, termination of parent CPU process. Depending on configuration of CPU layer of multiphysics framework, in best case scenario the CPU thread can be relaunched or even a thread context can be restored from a checkpoint; in worst case scenario, however, whole simulation would fail. No matter which scenario occurs any GPU layer failure results in loss in execution time and therefore in performance drop. Resolving that issue is highly important task because single failure may neglect days of computations which rerunning causes loses in time and money.

Second issue that is worth being investigated is cost effectiveness of new approach. GPUs have an opinion of being highly energy inefficient; supercomputers that consist of thousands of GPU consumes more power than equally efficient systems. This may lead to conclusion that each flops of GPU performance is more expensive. On the other hand, however, it is hard to achieve the performance of GPU using only CPU. Cost of adding new CPU nodes of performance equal to GPU modes is significantly higher. Furthermore one has to take into account context in which supercomputer is used. If multiphysics

simulation is the only or the main role of machine then switching to GPGPU may be reasonable. However when other applications are also ran on supercomputer than cost of redesign and implementation to support new hardware architecture has to be considered.

Further work in this area may include finding solutions and answers to issues: cost effectiveness and fault-tolerance. A good idea would be to design and implement a mechanism that tracks the parameters of simulation and record the used method and performance. Results would help to determine which approach, hybrid or standard, offers better performance. It is also reasonable to experiment with GPU features, parallel programing frameworks, or kernel execution properties like different structure of thread grids or even a dynamic structure which shape depends on input data structures. Another promising alternative is to investigate *libMesh* and other modules architecture to discover other highly parallel parts of code or even whole blocks that can be ported to GPU without compromising their performance. Moreover it might be profitable to consider a hybrid hardware architecture that takes advantage of GPU, CPU, and other technologies like FPGA. One can also focus on developing intelligent and automated detection of these parts in code that can be efficiently migrated to these accelerators.

References

[1] D. Aronofsky, Director, *Pi.* [Film]. United States: Protozoa Pictures, 1998.

[2] E. Burt, The Metaphysical Foundations of Modern Science, Chicago: Dover Publications, INC, 2003.

[3] J. Jeans, The Mysterious Universe, Cambridge University Press, 1930.

[4] E. Wigner, "The Unreasonable Effectiveness of Mathematics in the Natural Sciences," *Communications in Pure and Applied Mathematics,* vol. 13, no. 1, 1960.

[5] D. Hughes, "Prof. D. E. Hughes' Research in Wireless Telegraphy," *The Electrician,* vol. 43, pp. 40 - 41, 1899.

[6] C. F. Gauss, Theory of the Motion of the Heavenly Bodies Moving about the Sun in Conic Sections. A translation of Guass' Theoria Motus, Little, Brown and Company, 1857.

[7] J. Arndt and C. Haenel, π unleashed, Springer-Verlang, 2001.

[8] W. Gautschi, "Leonhard Euler: His Life, the Man, and His Work," *SIAM Review,* vol. 50, no. 1, pp. 3 - 33, 2008.

[9] G. Cardano, "Ars magna or The Rules of Algebra. A translation of Gerolamo's Ars magna.," 1993.

[10] D. Krol and D. Zydek, "Solving PDEs in Modern Multiphysics Simulation Software," in *2013 IEEE International Conference on Electro/Information Technology (EIT)*, Rapid City SD, 2013.

[11] D. Krol and D. Zydek, "Matrix Multiplication in Multiphysics Systems Using CUDA," in *Proceedings of the 18th International Conference on Systems Science (ICSS 2013), Advances in Intelligent Systems and Computing*, 2014.

[12] D. Krol, J. Harris and D. Zydek, "Hybrid GPU/CPU Approach to Multiphysics Simulation," in *Progress in Systems Engineering*, Las Vegas, 2014.

[13] D. Krol, S. Chiu, M. Liu and D. Zydek, "Effectiveness evaluation of cuBLAS and Thrust CUDA 6.5 libraries," *Journal of Distributed Computing,* 2015.

[14] D. Krol, D. Zydek and L. Koszalka, "Problem-Independent Approach to Multiprocessor Dependent Task Scheduling," *International Journal of Electronics and Telecommunications,* vol. 58, no. 4, pp. 369 - 379, 2013.

[15] W. Zimmerman, "Multiphysics Modeling with Finite Element Methods," *Series on Stability, Vibration and Control of Systems, Series A,* vol. 18, 2006.

[16] H. Achkar, F. Pennec, D. Peyrou, P. Pons and P. Pons, "Use the Reverse Engineering Technique to Link COMSOL and ANSYS Softwares," in *International Conference on Thermal, Mechanical and Multi-Physics Simulation and Experiments in Microelectronics and Micro-Systems, EuroSimE 2008.*, 2008.

[17] ANSYS Multiphysics, "ANSYS Multiphysics," [Online]. Available: www.ansys.com. [Accessed 12 2012].

[18] COMSOL Multiphysics, "COMSOL Multiphysics User's Guide, 3.5a ed.," 2008. [Online]. Available: http://wiki.crc.nd.edu/wiki/images/6/6a/Comsol_quick_3.5a.pdf. [Accessed 12 2012].

[19] Idaho National Laboratory, "MOOSE Workshop," Idaho National Laboratory, Idaho Falls, 2012.

[20] Idaho National Laboratory, "MOOSE Workshop," Idaho National Laboratory, Idaho Falls, 2014.

[21] D. Gaston, C. Newman, G. Hansen and D. Lebrun-Grandié, "MOOSE: A parallel computational framework for coupled systems of nonlinear equations," *Nuclear Engineering and Design,* vol. 239, pp. 1768 - 1778, 2009.

[22] H. Huang, B. Spencer and J. Hales, "Discrete element method for simulation of early-life thermal fracturing behavior in ceramic nuclear fuel pellets," *Nuclear Engineering and Design,* vol. 278, pp. 515 - 528, 2014.

[23] O. Courty, A. Motta and J. Hales, "Modeling and simulation of hydrogen behavior in Zircaloy-4 fuel cladding," *Journal of Nuclear Materials,* vol. 452, no. 1 - 3, pp. 311 - 320, 2014.

[24] libMesh, "libMesh," [Online]. Available: http://libmesh.github.io. [Accessed 10 2014].

[25] B. Kirk, J. Peterson and R. Stogner, "libMesh : a C++ library for parallel adaptive mesh refinement/coarsening simulations," *Engineering with Computers,* vol. 22, no. 3 - 4, pp. 237 - 254, 2006.

[26] J. Mukherjee and W. Gropp, "Performance Evaluation and Enhancement of Dendro," *ACM Transactions on Mathematical Software,* vol. 31, no. 3, pp. 397 - 423, 2005.

[27] Argonne National Laboratory, "PETSc," [Online]. Available: http://www.mcs.anl.gov/petsc/. [Accessed 12 2012].

[28] Argonne National Laboratory, "PETSc 3.5 User Manual," 2015.

[29] Sandia National Laboratories, "An Overview of Trilinos," 2003.

[30] Sandia National Laboratories, "The Trilinos Project," [Online]. Available: http://trilinos.sandia.gov. [Accessed 1 2013].

[31] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams and S. Kendall, "An Overview of the Trilinos Project," *ACM Transactions on Mathematical Software,* vol. 31, no. 3, pp. 397 - 423, 2005.

[32] TOP500, "TOP500 Ranking," 2014. [Online]. Available: http://www.top500.org/. [Accessed December 2014].

[33] MPI oficial forum, "www.mpi-forum.org," [Online]. Available: http://www.mpi-forum.org. [Accessed 11 2014].

[34] OpenMP, "OpenMP 4.0 Specification," 12 2014. [Online]. Available: http://openmp.org/wp/2013/07/openmp-40.

[35] G. E. Moore, "Cramming More Components onto," *Electronics Magazine,* 1965.

[36] NVidia, "CUDA Programming Guide," [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. [Accessed 1 2014].

[37] NVidia, "CUDA 6.5 release notes," 11 2014. [Online]. Available: developer.nvidia.com/cuda-zone.

[38] A. Abdelfattah, D. Keyes and H. Ltaief, "KBLAS: An Optimized Library for Dense Matrix-Vector Multiplication on GPU Accelerators," *ACM Transactions on Mathematical Software,* 2014.

[39] S. Barrachina, M. Castillo, F. Igual and R. Mayo, "Evaluation and tuning of the Level 3 CUBLAS for graphics processors," in *IEEE International Symposium on Parallel and Distributed Processing IPDPS 2008*, Miami, 2008.

[40] P. Estival and L. Giraud, "Performance and accuracy of the matrix multiplication routines: CUBLAS on Nvidia Tesla versus MKL and ATLAS on Intel Nehalem," 2012.

[41] B. Bylina and J. Bylina, "GPU-accelerated WZ Factorization with the Use of the CUBLAS Library," in *Federated Conference on Computer Science and Information Systems*, 2012.

[42] B. Zhang, X. Yang, F. Yang, X. Yang, C. Qin, D. Han, X. Ma, K. Liu and J. Tian, "The CUBLAS and CULA based GPU acceleration of adaptive finite element framework for bioluminescence tomography," *Optics Express,* vol. 18, no. 19, 2010.

[43] A. Deshpande and P. J. Narayanan, "Can GPUs Sort Strings Efficiently?," in *IEEE High Performance Computing (HiPC)*, 2013.

[44] J. Sang, C.-R. Lee, V. Rego and C.-T. King, "A Fast Implementation of Parallel Discrete-Event Simulation on GPGPU," in *International Conference on Parallel and Distributed Processing Techniques and Applications*, 2013.

[45] T. P. Loken, "A Comparison of Massively Parallel Programming Models Through Applications in Sound Propagation and Jitter Measurement," University of Nevada, Reno, 2014.

[46] D. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler and S. Zhou, "Process migration," *Journal of ACM,* vol. 32, no. 3, pp. 241 - 299, 2000.

[47] R. Gioiosa, J. C. Sancho, S. Jiang and F. Petrini, "Transparent, incremental checkpointing at," in *Proc. ACM/IEEE Intl. Conf. on Supercomputing kernel level: a foundation for fault tolerance for parallel computers*, Seattle, 2005.

[48] M. Bozyiğit, "User-level process checkpoint and restore for migration," *ACM SIGOPS Operating,* vol. 35, no. 2, pp. 86 - 96, 2001.

[49] P. Hargrove and J. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters," in *Proc. SciDAC*, Denver, 2006.

[50] J. P. Walters and V. Chaudhary, "Application-Level Checkpointing Techniques for Parallel Programs," in *Proc. Intl. Conf. on Distributed Computing and Internet Technology*, 2006.

[51] PNY, "PNY NVidia Quadro Specification," [Online]. Available: http://www.pny.eu/data/sitedynamic/Image/Quadro%205000%20SDI%20IO%20by%20PNY%20Datasheet.pdf. [Accessed December 2012].

[52] NVidia, "NVidia Fermi Architecture," [Online]. Available: http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf. [Accessed December 2012].

[53] NVidia, "NVidia Kepler Architecture," [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf. [Accessed September 2013].

[54] N. Whitehead and A. Fit-Florea, "Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs," NVidia, 2011.

[55] A. Nukada, H. Takizawa and S. Matsuoka, "NVCR: A Transparent Checkpoint-Restart Library for NVIDIA CUDA," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, Shanghai, 2011.

[56] H. Jiang, Y. Zhang, J. Jenness and L. Kuan-Ching, "A Checkpoint/Restart Scheme for CUDA Programs with Complex Computation States," *International Journal of Networked and Distributed Computing,* vol. 1, no. 4, pp. 196 - 212, 2013.

Appendix 1

input: Vectors *y*, *x*, both of size *m*, scalar *al*
output: Vector *y* of size *m*

```
1  for (int i = 0; i < m; i++)
2  {
3          y[i] += x[i] * al
4  }
```

Algorithm 1. CPU SAXPY

input: Vectors *y*, *x*, both of size *m*, scalar *al*
output: Vector *y* of size *m*

```
1  int id = blockIdx.x * blockDim.x + threadIdx.x;
2  if (id < m)
3  {
4          y[id] += x[id] * al;
5  }
```

Algorithm 2. CUDA SAXPY

input: Vectors *y*, *x*, both of size *m*
output: Vectors *y*, *x*, both of size *m*

```
1  for (int i = 0; i < m; i++)
2  {
3          float v = y[i];
4          y[i] = x[i];
5          x[i] = v;
6  }
```

Algorithm 3. CPU Vector Swap

input: Vectors *y*, *x*, both of size *m*
output: Vectors *y*, *x*, both of size *m*

```
1   int id = blockIdx.x * blockDim.x + threadIdx.x;
2   if (id < m)
2   {
3           float v = y[i];
4           y[i] = x[i];
5           x[i] = v;
6   }
```

Algorithm 4. CUDA Vector Swap

input: Vectors *y*, *x*, both of size *m*
output: Vector *y* of size *m*

```
1   for (int i = 0; i < m; i++)
2   {
3           y[i] += x[i];
4   }
```

Algorithm 5. CPU Vector Addition

input: Vectors *y*, *x*, both of size *m*
output: Vector *y* of size *m*

```
1   int id = blockIdx.x * blockDim.x + threadIdx.x;
2   if (id < m)
3   {
4           y[id] += x[id];
5   }
```

Algorithm 6. CUDA Vector Addition

input: Vectors *y* of size *m*, scalar *al*
output: Vector *y* of size *m*

```
1   for (int i = 0; i < m; i++)
2   {
3           y[i] = y[i] * al;
4   }
```

Algorithm 7. CPU Vector-Scalar Multiplication

input: Vectors *y* of size *m*, scalar *al*
output: Vector *y* of size *m*

```
1  int id = blockIdx.x * blockDim.x + threadIdx.x;
2  if (id < m)
3  {
4          y[id] = y[id] * al;
5  }
```

Algorithm 8. CUDA Vector-Scalar Multiplication

input: Vectors *y* of size *m*, scalar *al*
output: Scalar *d*

```
1  for (int i = 0; i < m; i++)
2  {
3          d = x[i] * y[id];
4  }
```

Algorithm 9. CPU DOT Product

input: Vectors *y*, *x*, both of size *m*
output: Scalar *d*

```
1  int id = blockIdx.x * blockDim.x + threadIdx.x;
2  if (id < m)
3  {
4          float v = x[id] * y[id];
5          atomicAdd(&d, v);
6  }
```

Algorithm 10. CUDA DOT Product

input: Matrix *A* of size *m*×*l*, vector *y* of size *m,* vector *x* of size *l*, scalars *al* and *be*
output: Vector *y* of size *m*

```
1  for (int i = 0; i < m; i++)
2  {
3          y[i] *= y[i] * be;
4          for (int j = 0; i < l; i++)
5          {
6                  y[i] += A[i * l + j] * x[j];
7          }
8  }
```

Algorithm 11. CPU GEMV

input: Matrix *A* of size *m*×*l*, vector *y* of size *m*, vector *x* of size *l*, scalars *al* and *be*
output: Vector *y* of size *m*

```
1  int idA = blockIdx.y * blockDim.y + threadIdx.y;
2  int idX = blockIdx.x * blockDim.x + threadIdx.x;
3  if (idA >= m || idX >= l)
4  {
5          return;
6  }
7  if (idX == 0)
8  {
9          y[idA] *= be;
10 }
11 __ syncthreads();
12 float result = A[idA * l + idX] * X[idX] * al;
13 atomicAdd(&Y[idA], result);
```

Algorithm 12. CUDA GEMV

input: Matrices *A*, *B*, both of size *m*×*l*
output: Matrices *A*, *B*, both of size *m*×*l*

```
1  for (int i  = 0; i < m; i++)
2  {
3          for (int j= 0; j < l; j++)
4          {
5                  float v = A[i * l + j];
6                  A[i * l + j] = B[i * l + j];
7                  B[i * l + j] = v;
8          }
9  }
```

Algorithm 13. CPU Matrix Swap

input: Matrices A, B, both of size $m \times l$
output: Matrices A, B, both of size $m \times l$

```
1  int idY = blockIdx.y * blockDim.y + threadIdx.y;
2  int idX = blockIdx.x * blockDim.x + threadIdx.x;
3  if (idA < m || idB < l)
4  {
5        idX += idY * l;
6        float temp = A[idX];
7        A[idX] = B[idX];
8        B[idX] = temp;
9  }
```

Algorithm 14. CUDA Matrix Swap

input: Matrices A, B, both of size $m \times l$
output: Matrix C of size $m \times l$

```
1  for (int i  = 0; i < m; i++)
2  {
3          for (int j= 0; j < l; j++)
4          {
5                  A[i * l + j] += B[i * l + j];
6          }
7  }
```

Algorithm 15. CPU Matrix Addition

input: Matrices A, B, both of size $m \times l$
output: Matrices A of size $m \times l$

```
1  int idY = blockIdx.y * blockDim.y + threadIdx.y;
2  int idX = blockIdx.x * blockDim.x + threadIdx.x;
3  if (idA < m || idB < l)
4  {
6        idX += idY * l;
7        A[idX] += B[idX];
8  }
```

Algorithm 16. CUDA Matrix Addition

input: Matrix A of size $m \times l$, scalar al
output: Matrix A of size $m \times l$

```
1  for (int i  = 0; i < m; i++)
2  {
3          for (int j= 0; j < l; j++)
4          {
5                  A[i * l + j] *= al;
6          }
7  }
```

Algorithm 17. CPU Matrix-Scalar Multiplication

input: Matrix A of size $m \times l$, scalar al
output: Matrix A of size $m \times l$

```
1  int idY = blockIdx.y * blockDim.y + threadIdx.y;
2  int idX = blockIdx.x * blockDim.x + threadIdx.x;
3  if (idA < m || idB < l)
4  {
5         idX += idY * l;
6         A[idX] *= al;
7  }
```

Algorithm 18. CUDA Matrix-Scalar Multiplication

input: Matrix A of size $m \times l$
output: Matrix B of size $m \times l$

```
1  for (int i  = 0; i < m; i++)
2  {
3          for (int j= 0; j < l; j++)
4          {
5                  B[j * l + i] = A[i * l + j];
6          }
7  }
```

Algorithm 19. CPU Matrix Transposition

input: Matrix *A* of size *m*×*l*
output: Matrix *B* of size *l*×*m*

```
1  int idY = blockIdx.y * blockDim.y + threadIdx.y;
2  int idX = blockIdx.x * blockDim.x + threadIdx.x;
3  if (idA < m || idB < l)
4  {
5        B[idX * l + idY] = A[idY * l + idX];
6  }
```

Algorithm 20. CUDA Matrix Transposition

input: Matrix *A* of size *m*×*l*, matrix *A* of size *l*×*n*, matrix *A* of size *m*×*n*, scalars *al*, *be*
output: Matrix *C* of size *m*×*n*

```
1  for (int i  = 0; i < m; i++)
2  {
3        for (int j= 0; j < n; j++)
4        {
5              C[i * n + j] *= C[i * n + j] * be;
6              for (int k = 0; k < l; k++)
7              {
8                    C[i * n + j] += A[i * l + k] * B[k * n + j] * al;
9              }
10       }
11 }
```

Algorithm 21. CPU GEMM

input: Matrix *A* of size *m*×*l*, matrix *A* of size *l*×*n*, matrix *A* of size *m*×*n*, scalars *al*, *be*
output: Matrix *C* of size *m*×*n*

```
1   int idB = blockIdx.x * blockDim.x + threadIdx.x;
2   int idA = blockIdx.y * blockDim.y + threadIdx.y;
3   if (idA >= m || idB >= n)
4   {
5           return;
6   }
7   int idC = idB + idA * n;
8   idA *= l;
9   float result = C[idC] * be;
10  for (int i = 0; i < l; i++)
11  {
12          result += A[idA] * B[idB] * al;
13          idA++;
14          idB += n;
15  }
16  C[idC] = result;
```

Algorithm 22. Simple CUDA GEMM

input: Matrix *A* of size *m*×*l*, matrix *A* of size *l*×*n*, matrix *A* of size *m*×*n*, scalars *al*, *be*
output: Matrix *C* of size *m*×*n*

```
1   int idB = blockIdx.x * blockDim.x + threadIdx.x;
2   int idA = blockIdx.y * blockDim.y + threadIdx.y;
3   if (idA >= m || idB >= n)
4   {
5           return;
6   }
7   int id_temp = threadIdx.x + threadIdx.y * blockDim.x;
8   int offset = 0;
9   float result = 0.0;
10  extern shared float shared[];
11  float *tA = &shared[0];
12  float *tB = &shared[blockDim.x * blockDim.x];
13  while (offset < l)
14  {
15          tA[id temp] = tB[id temp] = 0;
16          if (idA < m && offset + threadIdx.x < l)
17          {
18                  tA[id temp] = A[idA * l + offset + threadIdx.x];
19          }
20          if (idB < n && offset + threadIdx.y < l)
21          {
22                  tB[id temp] = B[idB + (offset + threadIdx.y) * n];
23          }
24          offset += blockDim.x;
25          __ syncthreads();
26          for (int j = 0; j < blockDim.x; j++)
27          {
28                  result += tA[threadIdx.y * blockDim.x + j] *
                            tB[threadIdx.x + j * blockDim.x] * al
29          }
30          __syncthreads();
31  }
32  if (idA < m && idB < n)
33  {
34          C[idB + idA * n] = C[idB + idA * n] * be + result;
35  }
```

Algorithm 23. Tiled CUDA GEMM

input: Matrix *A*, *B* both of size *m*×*m* scalars *al*, *be*
output: Matrix *C* of size *m*×*m*

```
1   for (int i  = 0; i < m; i++)
2   {
3           for (int j= 0; j < i; j++)
4           {
5                   C[i * m + j] *= C[i * m + j] * be;
6                   for (int k = 0; k < m; k++)
7                   {
8                           C[i * m + j] += A[i * m + k] * B[k * m + j] * al;
9                   }
10          }
11  }
```

Algorithm 24. CPU Left Triangular GEMM

input: Ma Matrix *A*, *B* both of size *m*×*m* scalars *al*, *be*
output: Matrix *C* of size *m*×*m*

```
1   int idB = blockIdx.x * blockDim.x + threadIdx.x;
2   int idA = blockIdx.y * blockDim.y + threadIdx.y;
3   if (idA >= m || idB >= idA)
4   {
5           return;
6   }
7   int idC = idB + idA * m;
8   idA *= m;
9   float result = C[idC] * be;
10  for (int i = 0; i < m; i++)
11  {
12          result += A[idA] * B[idB] * al;
13          idA++;
14          idB += m;
15  }
16  C[idC] = result;
```

Algorithm 25. Simple CUDA Left Triangular GEMM

input: Matrix *A* of size *m*×*l*, matrix *A* of size *l*×*n*, matrix *A* of size *m*×*n*, scalars *al*, *be*
output: Matrix *C* of size *m*×*n*

```
1   int idB = blockIdx.x * blockDim.x;
2   int idA = blockIdx.y * blockDim.y + threadIdx.y;
3   if (idA < idB)
4   {
5           return;
6   }
7   idB += threadIdx.x;
8   idA += threadIdx.y;
9   if (idA >= m || idB >= n)
10  {
11          return;
12  }
13  int id_temp = threadIdx.x + threadIdx.y * blockDim.x;
14  int offset = 0;
15  float result = 0.0;
16  extern shared float shared[];
17  float *tA = &shared[0];
18  float *tB = &shared[blockDim.x * blockDim.x];
19  while (offset < l)
20  {
21          tA[id temp] = tB[id temp] = 0;
22          if (idA < m && offset + threadIdx.x < l)
23          {
24                  tA[id temp] = A[idA * l + offset + threadIdx.x];
25          }
26          if (idB < n && offset + threadIdx.y < l)
27          {
28                  tB[id temp] = B[idB + (offset + threadIdx.y) * n];
29          }
30          offset += blockDim.x;
31          __ syncthreads();
32          for (int j = 0; j < blockDim.x; j++)
33          {
34                  result += tA[threadIdx.y * blockDim.x + j] *
                                tB[threadIdx.x + j * blockDim.x] * al
35          }
36          __syncthreads();
37  }
38  if (idA < m && idB < n)
39  {
40          C[idB + idA * n] = C[idB + idA * n] * be + result;
41  }
```

Algorithm 26. Tiled CUDA Left Triangular GEMM

input: Matrix *A*, *B* both of size *m*×*m* scalars *al*, *be*
output: Matrix *C* of size *m*×*m*

```
1   for (int i  = 0; i < m; i++)
2   {
3           for (int j= i; j < m; j++)
4           {
5                   C[i * n + j] *= C[i * n + j] * be;
6                   for (int k = 0; k < m; k++)
7                   {
8                           C[i * m + j] += A[i * m + k] * B[k * m + j] * al;
9                   }
10          }
11  }
```

Algorithm 27. CPU Right Triangular GEMM

input: Matrix *A*, *B* both of size *m*×*m* scalars *al*, *be*
output: Matrix *C* of size *m*×*m*

```
1   int idB = blockIdx.x * blockDim.x + threadIdx.x;
2   int idA = blockIdx.y * blockDim.y + threadIdx.y;
3   if (idA >= m || idB >= m || idB < idA)
4   {
5           return;
6   }
7   int idC = idB + idA * m;
8   idA *= m;
9   float result = C[idC] * be;
10  for (int i = 0; i < m; i++)
11  {
12          result += A[idA] * B[idB] * al;
13          idA++;
14          idB += m;
15  }
16  C[idC] = result;
```

Algorithm 28. Simple CUDA Right Triangular GEMM

input: Matrix *A* of size *m*×*l*, matrix *A* of size *l*×*n*, matrix *A* of size *m*×*n*, scalars *al*, *be*
output: Matrix *C* of size *m*×*n*

```
1   int idB = blockIdx.x * blockDim.x + blockDim.x;
2   int idA = blockIdx.y * blockDim.y + blockDim.y;
3   if (idA > idB)
4   {
5           return;
6   }
7   int idB = blockIdx.x * blockDim.x - blockDim.x + threadIdx.x;
8   int idA = blockIdx.y * blockDim.y - blockDim.y + threadIdx.y;
9   if (idA >= m || idB >= n)
10  {
11          return;
12  }
13  int id_temp = threadIdx.x + threadIdx.y * blockDim.x;
14  int offset = 0;
15  float result = 0.0;
16  extern shared float shared[];
17  float *tA = &shared[0];
18  float *tB = &shared[blockDim.x * blockDim.x];
19  while (offset < l)
20  {
21          tA[id temp] = tB[id temp] = 0;
22          if (idA < m && offset + threadIdx.x < l)
23          {
24                  tA[id temp] = A[idA * l + offset + threadIdx.x];
25          }
26          if (idB < n && offset + threadIdx.y < l)
27          {
28                  tB[id temp] = B[idB + (offset + threadIdx.y) * n];
29          }
30          offset += blockDim.x;
31          __ syncthreads();
32          for (int j = 0; j < blockDim.x; j++)
33          {
34                  result += tA[threadIdx.y * blockDim.x + j] *
                                tB[threadIdx.x + j * blockDim.x] * al
35          }
36          __syncthreads();
37  }
38  if (idA < m && idB < n)
39  {
40          C[idB + idA * n] = C[idB + idA * n] * be + result;
41  }
```

Algorithm 29. Tiled CUDA Right Triangular GEMM