HETEROGENEOUS CPU-GPU SOFTWARE FRAMEWORK FOR DAG'S IN HIGH PERFORMANCE COMPUTING

by

Abhishek Bagusetty

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Chemical Engineering

The University of Utah

August 2015

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Abhishek Bagusetty**

has been approved by the following supervisory committee members:

James C. Sutherland	, Chair	**07/30/2014** Date Approved
Matthew Might	, Member	**07/29/2014** Date Approved
Jeremy Thornock	, Member	**07/30/2014** Date Approved

and by **Milind Deo** , Chair/Dean of

the Department/College/School of **Chemical Engineering**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Recent advancements in High Performance Computing (HPC) infrastructure with traditional computing systems augmented with accelerators like graphic processing units (GPUs) and coprocessors like Intel Xeon Phi have successfully enabled predictive simulations specifically Computational Fluid Dynamics (CFD) with more accuracy and speed. One of the most significant challenges in high-performance computing is to provide a software framework that can scale efficiently and minimize rewriting code to support diverse hardware configurations. Algorithms and framework support have been developed to deal with complexities and provide abstractions for a task to be compatible with various hardware targets. Software is written in C++ and represented as a Directed Acyclic Graph (DAG) with nodes that implement actual mathematical calculations. This thesis will present an improved approach for scheduling and execution of computational tasks within a heterogeneous CPU-GPU computing system insulting application developers with the inherent complexity in parallelism. The details will be presented within a context to facilitate the solution of partial differential equations on large clusters using graph theory.

“For the Benefit of All Mankind!”

CONTENTS

ABSTRACT . . . . . **iii**

LIST OF FIGURES . . . . . **vii**

ACKNOWLEDGMENTS . . . . . **viii**

CHAPTERS

1. INTRODUCTION AND OVERVIEW **1**

1.1 Advantages with Simulations 2
1.2 Multicore and Many-core (GPUs) Architectures 3
1.3 Coprocessors and Massively Parallel Hardware 4
1.4 Recents Trends in Computational Architectures —Graphics Processing Units (GPUS) 4
1.5 Task Graph Approach Model 6
1.6 Uintah Computational Framework 6
1.7 Wasatch 7
1.8 Nebo, a Domain Specific Language (DSL) 8

2. SPATIAL FIELDS—FRAMEWORK AND EXTENSIONS **9**

2.1 Overview 9
2.2 Classification of Fields—Existence 9
2.2.1 Scratch (Temporary) Fields 10
2.2.2 Persistent Fields 10
2.3 Classification of Fields—Storage Mode 11
2.4 Spatial Fields—Multiple Locations 11
2.4.1 Overview 11
2.4.2 Classification of Field—Access Permissions 12
2.5 Characteristic Traits of a Field 14

3. TASK GRAPH EXECUTION AND SCHEDULING **15**

3.1 Overview 15
3.1.1 Dependency Task Graph 16
3.1.2 Execution Task Graph 16
3.1.3 Homogeneous Task Graph 18
3.1.4 Heterogeneous Task Graph 18
3.2 CUDA Resources 18
3.2.1 CUDA Streams 18
3.2.2 CUDA Kernels 20

3.3 Task Graph Scheduling 20
3.3.1 Task Scheduling on a Multithreaded System 21
3.3.2 Task Scheduling on Hybrid CPU-GPU Architecture 22
3.4 Task Graph Execution 23
3.4.1 Memory Resources Reuse 24
3.4.2 Memory Pools 25
3.4.3 ExprLib Task Execution Configuration 26
3.5 Device Selection for On-node Multi-GPUs 31
3.6 Optimization Strategies 31
3.6.1 Pinned and Pageable Memory 31

4. WASATCH 34

4.1 Overview 34
4.2 Wasatch Interface 35
4.2.1 Homogeneous Wasatch GPU Task Setup 36
4.2.2 Heterogeneous Wasatch CPU-GPU Task Setup 38
4.3 Wasatch GPU Task Execute 43
4.3.1 Homogeneous GPU Task Execute 43
4.3.2 Heterogeneous CPU-GPU Task Execute 43

5. RESULTS AND SUMMARY 45

5.1 Overview 45
5.2 Test Case—ExprLib 45
5.3 Test Case—Wasatch 51
5.3.1 Insight—Level of Concurrency 53
5.4 Future Work 55
5.5 Summary 57

REFERENCES 58

LIST OF FIGURES

2.1 Spatial field with multiple field locations 13
2.2 Characteristics of a field with multiple locations 13
3.1 Nodes of a task graph showing the dependency relation 17
3.2 Nodes of a task graph showing the flow of execution 17
3.3 Homogeneous task graph with the nodes having same hardware target 19
3.4 Heterogeneous task graph with the nodes having different hardware target . . . 19
3.5 Illustration of a heterogeneous task graph 25
3.6 Homogeneous GPU task execution using asynchronous mode 28
3.7 Heterogeneous GPU task execution using asynchronous mode 30
3.8 Performance analysis of pinned and pageable memory 33
4.1 Setup phase of a homogeneous GPU task graph 37
4.2 Flipping the GPU-runnable properties of expressions 39
4.3 Restoration of the GPU-runnable properties of the expressions 40
4.4 Setup phase of a heterogeneous CPU-GPU task graph 42
4.5 Execution phase of a heterogeneous CPU-GPU task graph 44
5.1 ExprLib-scalability test with diffusive flux and coupled source terms 47
5.2 ExprLib-scalability test showing multicore performance 48
5.3 ExprLib-scalability test showing GPU performance 50
5.4 Task graph used for scalability test in Wasatch 52
5.5 Computation time for the scalability test in Wasatch 54
5.6 Speedup analysis for the scalability test in Wasatch 54
5.7 Computation time for different modes of execution 56
5.8 Speedup analysis for different modes of execution 56

ACKNOWLEDGMENTS

Foremost, I express my sincere gratitude to my advisor Prof. James C. Sutherland for the continuous support with his patience, motivation, enthusiasm, and immense knowledge. Without his support, I wouldn't have travelled to the other side of the planet, which offered me an opportunity to contribute my share to human knowledge. His support has lead me to explore and understand a whole new world of programming that has changed the path of my career. I would like to thank especially Christopher Earl, Tony Saad, Alan Humphrey, and Qingyu Meng for the tremendous support they offered me in shaping my research.

I wish to thank my committee members, Matthew Might and Jeremy Thornock, for their time and patience. Finally, I would like to thank my parents for instilling a great sense of values and always supporting me to chase my dream.

CHAPTER 1

INTRODUCTION AND OVERVIEW

Predetermining the outcome or behavior of some physical system comes with prediction science. With rapidly increasing computational capability, modeling- and simulation-based design is taking on increased responsibility for the success of new engineered systems, in replacement of the present design practice that relies heavily on extensive testing of components and prototype systems [1]. Modeling and simulation are interrelated terms that help in mimicking a real system. Associated with this is an emerging interdisciplinary field of prediction science, which is the application of verified and validated computational simulations to predict the response of complex systems, particularly in cases where routine experimental tests are not feasible.

The advancement simulation and modeling is potentially critical to the design of new and complex engineered systems in a variety of applications across such diverse domains as microsystems, advanced materials, biological systems, energy generation and consumption, nuclear systems, and climate modeling. As such complex systems require the integration of a diverse set of disciplines and the sophisticated multiphysics simulations, there is a need for developing a new software framework model that can support the underlying new paradigm. Specifically, the future success requires unified software and algorithmic frameworks for integrating models and code from multiple disciplines.

In the earlier days, the tools and expertise required for such simulation were not available to anyone outside of government or large research institutions. However, as the availability and cost associated with high-performance computing hardware has reduced, the capabilities of commodity hardware has reached a point where conducting accurate simulations has become feasible to a diverse community.

Along with the rapid improvements in hardware and its availability, there have also been significant improvements in the general accessibility of software tools for utilizing computational resources. With the standardization of large-scale message passing standards, such as the message passing interface (MPI), the idea of using simulation to drive research and

development has become more feasible. This can be observed across a diverse range of applications ranging from the pharmaceutical industry [2], astronomical simulations [3], to computational chemistry simulations [4].

1.1 Advantages with Simulations

A major component of any development process is testing and verification, to ensure that a product or process functions in the desired fashion and poses no overt danger to the end user. The testing processes can present significant hazards; complex chemical reactions with toxic or flammable components, explosives testing, or other high-energy interactions all have inherent risks associated with them. Utilizing the proper simulation tools, many of these systems can be examined in a safe environment, before ever being tested in the lab.

This increases safety, reduces material costs, and in the event that a problem is detected during the testing process, a fix can be implemented and retested with significantly lesser than would have been possible in a more conventional testing lab. Another issue that is important to consider is information completeness. In any real system, we are limited in the amount of data we can collect due to constraints on sensor density and physical characteristics of the experiment itself. If we wish to examine some type of large-scale explosion or high-energy [5] behavior, then it is completely not feasible to experimentally capture the complete behavior of the explosive material and the resulting forces through the entire life cycle of the process. However, in simulating such a scenario, we can theoretically capture a complete data profile at all resolved scales of the simulation.

A pipeline of developing, fabricating and testing, and reiterating all the three steps again can be quite an expensive project for any process design. Most of the time the simulation testing is cheaper and faster than performing the multiple tests of the design each time. To simulate something physical, it is necessary to create a mathematical model that represents a physical domain. Models can take many forms including declarative, functional, constraint, spatial, or multimodel. A multimodel is a model containing multiple integrated models, each of which represents a level of granularity for the physical system. You can also execute (i.e., simulate) the program on a massively parallel computer.

The next biggest advantage of a simulation is the level of details that you can get from a simulation. A simulation can give you results that are not experimentally measurable with the current level of technology. Results such as surface interactions on an atomic level, flow at the exit of a microelectric thruster, or molecular flow inside of a star or simulating a boiler with various feeds are not measurable by any current devices or a feasible idea. A simulation can give these results when problems such as these are too small to measure, the

probe is too big and is skewing the results, or it is very costly to perform tests. You can set the simulation to run for as many time steps to any level of detail with the only restrictions being one's imagination, programming skills, and CPU resources.

1.2 Multicore and Many-core (GPUs) Architectures

Core clock rates for central processing units (CPUs) grew drastically, starting in the tens of MegaHertz and ending in the giga hertz range by the end of the decade. However, as design processes continued to shrink and CPU clock rates pushed higher, it became apparent that manufacturers were rapidly approaching a performance limit with traditional designs. Problems related to current leakage and heat generation began to scale more rapidly than any associated performance gains, and CPU manufacturers such as Intel and Advanced Micro Devices (AMD) began to look elsewhere for ways to improve performance.

As the speed of individual chips essentially plateaued, Moore's Law, an idea stating that the overall number of transistors on chip would double every 18 to 24 months, continued to hold [6]. Faced with ever-increasing on-chip real estate, CPU manufacturers began to fabricate chips with multiple processors or cores on a single die. The idea was that if it was not possible to improve the speed of serial computations, it was certainly still possible to increase the amount of concurrent work that could be performed on a single device.

GPUs had the potential for much more general computation.Realizing the potential of generic, programmable, vector hardware, which was capable of operating on massive amounts of data concurrently, GPU vendors such as NVIDIA began to develop and expose application programming interfaces (APIs), allowing software developers to more easily exploit the functionality of their hardware [7]. This has, in turn, created a significant need to reexamine software design practices related to high-performance computing, as in certain cases, specific computations may run one or even two orders of magnitude faster on GPU than would be possible on CPU. With this understanding, the ability of software to properly distribute workloads across a variety of hardware, keeping task on the device which gives the best performance, has become extremely important. Hence a new term GPGPU is introduced. GPGPU (general purpose computing on graphics processing units) is a methodology for high-performance computing that uses graphics processing units to crunch data. The characteristics of graphics algorithms that have enabled the development of extremely high-performance special purpose graphics processors show up in other HPC algorithms.

GPU properties lead to a very different processor architecture from traditional CPUs.

CPUs devote a lot of resources (primarily chip area) to make single streams of instructions run fast, including caching to hide memory latency and complex instruction-stream processing (pipelining, out-of-order execution and speculative execution). GPUs, on the other hand, use the chip area for hundreds of individual processing elements that execute a single instruction stream on many data elements simultaneously. Memory latency is hidden by very fast context switching; when a memory fetch is issued while processing one subset of data elements, that subset is set aside in favor of another subset that is not waiting on a memory reference.

1.3 Coprocessors and Massively Parallel Hardware

Unlike CPU cores, which have been traditionally designed to maximize serial performance, utilizing complicated circuitry for out of order execution and doing everything possible to avoid pipeline delays, there has long been the notion of vector-based processors. Vector processing is the idea of having hardware which is capable of operating on many pieces of data simultaneously, often executing a single instruction in parallel across each data element, a process known as single instruction multiple data (SIMD). However, with the exception of some multimedia extensions, such as streaming SIMD extensions (SSE), found within x86 processors, this has changed with the advent of discrete programmable graphics hardware, known as graphics processing units (GPUs). It was originally designed to accelerate tasks relating to computer graphics, such as geometry translation and coloring, which can be parallelized.

1.4 Recents Trends in Computational Architectures —Graphics Processing Units (GPUS)

This trend, requiring added concurrency from software algorithms, has introduced significant complications into the process of architecting quality, high-performance, simulation software. There are now (multi/many)-core CPUs, massively parallel coprocessor like devices such as general purpose GPUs (GPGPUs), all of which have their own memory, communication costs, and programing paradigms. Talking specifically about the GPUs, the key to its success has been due to its massive performance when compared to the traditional CPU [8].

GPUs are not the only type of accelerator core that has gained interest over the last decade. Other examples include field programmable gate arrays (FPGAs) and the Cell Broadband Engine (Cell BE), which have both been highly successful in many application

areas [9]. Today, however, these receive only a fraction of the attention of GPUs. There are three major GPU vendors for the PC market today, Intel being the largest. However, Intel is only dominant in the integrated and low-performance market. For high-performance and discrete graphics, AMD and NVIDIA are the sole two suppliers.

Heterogeneous computing is the new term that is more prominent in the field of scientific computing and graphics community. The term Heterogeneous computing is the coordination of two or more different processors, of different architecture types, to perform a computational task. Generally the CPUs that we are ware of have x86 architecture type, but you also have another processor (of a different architecture type). The other processor is an accelerator because it accelerates computations by assisting the CPU to get the work done. There is a sudden boom in using GPU computing because of its offered advantages over traditional systems in terms of computation, power consumption and inexpense. The software necessary to support Heterogeneous Computing is necessary to exploit the resources. The process for re-writing software involves an understanding of parallelism.

It is no longer enough to know a general purpose programming language and then express one's computational model as a self-contained program. Care must be taken with respect to taking advantage of the features available from underlying hardware and designing a program to not only run, but scale effectively on a large computing grid (often consisting of thousands if not hundreds of thousands of nodes). Additionally, as is the case with various GPGPU computing elements, more care must be taken with respect to algorithm correctness, as numeric rounding and floating point representations may not be entirely consistent between devices.

Generally there are different hardware options that act as augmentation for the traditional GPU and add to the term heterogeneous computing.

- Intel Xeon Phi - which generally has 60+ core on chip is refereed to as the coprocessor. It has a new processor architecture which are similar to the older x86 CPUs.
- Intel Integrated graphics - which is Intel's GPU and not as capable as NVIDIA or AMD's GPUs, but comes on the same chip as all Intel's CPUs and is probably the most ubiquitous GPU today for that reason.
- AMD FirePro and Radeon - which are the only other first-rate GPUs for desktops and servers.
- AMD APUs - which is AMD's merger of Radeon technology onto the same chip as the AMD CPU i.e, the merge of both CPU and GPU onto the same chip. The GPU

AMD that is put on APUs today is not as powerful as the full Radeon GPU, but it can still be used as an accelerator.

- Altera FPGAs - which are used to be restricted to very niche markets, but with the recent developments towards heterogeneous computing, Altera's FPGAs and FPGAs from other vendors will be considered as a viable option for many more applications.

Scientists and model developers should, ideally, be insulated from what would traditionally be considered engineering or computer science problems, and instead be allowed to focus upon their domain of specialization. Of course, this cannot always be the case, particularly given the pace of modern hardware development.

1.5 Task Graph Approach Model

Multiphysics simulation software is driven by complexity in data dependencies. In general, a software solving coupled system of partial differential equations (PDEs) representing a physical system mathematically expressed as transport equations is generally difficult in exploring low-level data dependencies. Mathematical expressions are represented in the form of nodes that directly represent the data dependencies in a task graph. The entire system of expressions forms a task graph, and the high level of layout is automated through standard graph algorithms [10]. In a multiphysics simulation, the task algorithm is properly ordered so that it mimics a discretized set of coupled Partial Differential Equations. This sort of design removes programmers from understanding the complex inner dependencies present in a multiphysics software. As the algorithms develop, parallelism options can be explored by task decomposition and using thread-based parallelism or GPU programming without the domain decomposition.

1.6 Uintah Computational Framework

Uintah [11], [12], [13] is a software framework designed for full-physics simulations on large-scale clusters. An important trend in the high-performance computing is to design a software framework that can scale well on varied architectures with peta-flop and eventually exascale performance. The novelty of this framework is that the formidable scalability and performance challenges associated while running in hybrid computing systems is insulated from the application developer [14]. Full physics means that Uintah supports simulations of both fluid dynamics and rigid body dynamics. Additionally, Uintah supports simulations of strong interactions between fluids and solids, such as temperature and velocity interactions as well as chemical and physical transformations. Since Uintah is a framework, each of its

software components implements the interactions and the numeric calculations underlying the simulations.

Uintah's main goal is to handle parallelism at a high level so that its components can focus on the numeric calculations underlying the simulations. Uintah exploits both of the major types of parallelism, task and data parallelism. For data parallelism, Uintah decomposes the physical simulation domain into patches. Uintah sets up the MPI process for each patch and handles all communication between patches. Additionally, Uintah provides support for automated load balancing of patches across processors. For task parallelism, Uintah employs a task graph model. Each node of the task graph specifies various tasks as well as the dependencies between tasks. Uintah handles scheduling these tasks and decides when, where, and how to run tasks in parallel. Furthermore, Uintah handles any communication that occurs between tasks. Thus, components of Uintah focus on what happens within a Uintah task on a single patch, rather than the interactions between tasks or patches.

Uintah provides support for other administrative duties for running simulations on large-scale clusters. Uintah supports adaptive mesh refinement, through which Uintah can increase or decrease resolution of individual patches based upon runtime flags [12]. For components that support GPU execution, Uintah can manage data transfer and kernel scheduling. All of the support Uintah provides to its components is centered around tasks that every simulation project must handle. Thus, component developers can focus their efforts on implementing models and numeric calculation, independent of resource concerns.

1.7 Wasatch

As stated previously, the goal of this work will be to extend a component of Uintah, Wasatch [10] for exploring the parallelism opportunities offered by GPUs. The main goal of this work is to extend the support to execute the computational fluid dynamics problems expressed in the form of a task graph so that parallelism offered from the device (GPU) is exploited in terms of graph scheduling algorithms rather than developing new numerical kernels for the GPUs. The Wasatch framework itself can be thought of as consisting of a variety of components such as SpatialOps and ExprLib, each of which exists at a different level and supports for various purposes. The Uintah Computational Framework (UCF) is primarily designed for providing MPI-based domain decomposition on adaptive mesh refinement. Expression Library (ExprLib) primarily purpose is to provide support for graph algorithms that can be used by the tasks defined in Wasatch component of Uintah with

the MPI interface parallelism and a lower level downstream library - Spatial Operations (SpatialOps), which implements details related to specific mathematical operators for the fields defined in the tasks.

1.8 Nebo, a Domain Specific Language (DSL)

High-performance computing applications are by definition very sensitive and susceptible to inefficient code. To avoid inefficiencies, most high-performance computing code is written at a very low level. However, with the rise of new architectures, such as multicore CPUs, many-core CPUs, and GPUs, all of this code must be rewritten for each new architecture the project is to use [15]. Rewriting all this code is an labor-intensive and error-prone process. To create code portable between multiple architectures, an efficient domain-specific language (DSL) embedded in C++ is introduced that is user-friendly for porting the code to varied architectures. Nebo's target domain is mostly focused on computational fluid dynamics (CFD) applications provides necessary abstractions for numerically solving partial differential equations. Nebo is a declarative DSL for numeric computation over arrays of any dimensionality. This DSL support of Nebo are reliably used for the projects like SpatialOps, ExprLib and Wasatch which are explained in further chapters.

CHAPTER 2

SPATIAL FIELDS—FRAMEWORK AND EXTENSIONS

2.1 Overview

Fields are generally created in Spatial Operation library (SpatialOps) that provides abstractions such as creation, storage, and destruction of objects, which represent the data fields and their generic interface to a set of mathematical operators. When attempting to simulate or describe a physical phenomena, it helps in discretizing spatially and temporally.

The most important concept related to the spatial fields is how to define it. The dimensions of the field like the size, information regarding the presence of physical boundary and the number of extra cells, etc are some of the parameters required to construct a spatial field. The detailed explanation of the above stated field properties is not in the scope of this context. But in the case of a heterogeneous CPU-GPU task graph where the fields can reside on CPU or GPU, certain parameters are necessary in constructing a spatial field. A device index is required for a field that defines the field's location on a hardware device target. A pointer to the memory address space specified by the device index and a field storage mode is also some important parameters to be considered while constructing a Spatial Field. Detailed information on the field storage mode is explained in §2.2.

2.2 Classification of Fields—Existence

Fields can be classified based on the life time of their existence. Some fields tends to stay for the entire duration of the simulation and others for a small instance until their purpose is served. For any simulation, a clear understanding of the life span of a field is necessary. Based on the lifetime of the fields, it can be classified into two types, scratch fields and persistent fields.

2.2.1 Scratch (Temporary) Fields

Scratch fields are used with a purpose of storing the values of field at a temporary location for performing some intermediate calculations. These fields are used for storing the field values for a short duration until they are used by another source. A scratch field can be created from a prototype field so that their field types and other properties can be mimicked. A scratch field can be created on any hardware device, and it is not necessary that the hardware device location is the same as that of prototype field. The primary purpose of a prototype field while creating a scratch field is to grab the information about the field's properties so that the scratch field can be constructed based off of that. Once a field's properties are available for a scratch field, a device index can be specified to create the field on the desired hardware target.

2.2.2 Persistent Fields

The name itself indicates that these fields are persistent for the entire duration of the simulation without discarding its memory like scratch fields. These fields are owned by the memory manager with a purpose that the field values get stored at each time step or at the check points so that they can be used for the other purposes like visualization. The term *memory manager* is very specific in its purpose and in a broad sense, these managers are responsible for the management of memory related to the field. Other cases where the persistent fields are required are when a field is necessary to be carried over to the subsequent time steps.

The persistent fields are activated by tagging the expression that they belong to. An expression is a software object that defines a mathematical expression and fields can be considered as analogous to the variables in mathematical expression. More details about the expressions are provided in the next chapter. When an expression is marked to be persistent, the fields underlying it get marked as persistent. A locking mechanism is assigned to these fields so that a deallocation is not to be performed for them, and this keeps them persistent. Currently for a task graph, all the topologic edge nodes are flagged to be persistent. Topologic edge nodes are nodes in a task graph that do not have any parents or children. When a node gets tagged as persistent, its corresponding expression and the fields associated with it gets tagged as persistent. This process gives exemption for the fields to remain dormant for the deallocation procedures.

2.3 Classification of Fields—Storage Mode

Storage mode is a very important property for a spatial field as it defines the ownership policy of a field that comes into existence. There are two modes for a field, Internal Storage and External Storage.

Internal Storage specifies that the values in a field are copied into an internal buffer managed by the spatial field. The ownership is managed by the spatial field and memory pools serves the memory requirements, which will be elaborately discussed in §3.4.2.

External Storage specifies that the values in a field are stored by an external entity. The ownership is generally handled by the external sources like Uintah [11], and a spatial field is created by wrapping the memory supplied by the external source. The external storage mode is good in efficacy as it avoids excessive copies of a field, and the internal storage mode is best in its sanity. This mode protects against the memory corruption and inadvertent deletion of the field's underlying memory.

Generally, memory allocated by an external sources has the storage mode as external for its ownership policy. Other fields that are created as scratch or temporary fields have their storage mode as internal with their allocation and deallocations managed by the spatial fields.

2.4 Spatial Fields—Multiple Locations

2.4.1 Overview

Spatial Fields can have field locations on the CPU or GPU or both. A field can be created on CPU or a GPU, and in a heterogenous CPU-GPU task graph, there could be a possibility that a field is requested on a different hardware target. In those circumstances, we designed a concept called consumer field. A field location that is created other than its actual field location is referred to as a *consumer field*. This process of adding a consumer field would increase the count on memory locations for a field across varied hardware targets. It was initially designed to add a consumer field on any given hardware target so that it is used as an input field for an expression. As the consumer fields were created to serve as input fields to the expression, their access is only restricted to read-only mode. For a instance, when a field is initially created on the CPU and has a consumer field on the GPU, only one location out of the two was designated to have a write access. A read-write access is only given for the primary field location, and the consumer fields are restricted for write access.

This design decision of consumer field to have read-only access is made to meet the purpose of executing a Nebo [15] statement. The read-only access limitation avoids any

confusion with the selection of field location for write access in case of a multiple field locations. A sample code snippet, `a <<= b+ c` , gives an example of a Nebo statement. The field (a) on the left hand side is the one to which the values are written from the calculations, and hence a write access is desired. The fields (b, c) on the right hand side of the Nebo statement serves as an input field for the calculations and hence, it is desired to have a read-only access. If a consumer field is added on a location for the field (a), then it would be difficult choice for the Nebo to decide which field out of the many is available for the write access. Hence we deny access to any field with an exception that has consumers to participate as a left hand term for a Nebo statement.

If a write access is provided to a desired field location, the other would get invalidated from the updates made to the other field location. Hence there is a need for the synchronization between multiple fields locations, so a design limitation has been imposed. This limitation states that the fields that take part on the left hand side of a Nebo statement should not add any consumer fields, but for the right hand side, multiple field locations are allowed with consumer fields as they are restricted to have read-only access. To prevent an explicit synchronizations that triggers the data transfer across all the field locations, a read-only access limitation has been imposed.

2.4.2 Classification of Field—Access Permissions

The model of consumer fields have restricted the use of multiple fields, and this limitation has been discarded with a new model of assigning traits to each of the field locations for a spatial field that has multiple field location. With this new model on the spatial fields, it can have multiple field locations on diverse hardware targets. An illustration in Figure 2.1 shows a spatial field with multiple field locations. A new model allows a spatial field to have multiple field locations with the traits describing about the access permission bestowed on each field location.

Recent advancement to the spatial fields depricates the use of consumer fields and creates two traits that describe the state of any field location for a spatial field given by *active* and *valid*. Active field locations are eligible for both read and write access, and valid field locations are restricted to only read access. An invalid field location is either inaccessible or contains invalid field values for read or write access. Different modes of access permissions for a field location is illustrated in Figure 2.2. This type of field location can be validated by copying data from a valid field source. A valid field location becomes invalid if the values in the field are modified during a calculation. A valid location can be flagged as an active field location. There is a limitation imposed which states that no two field locations can

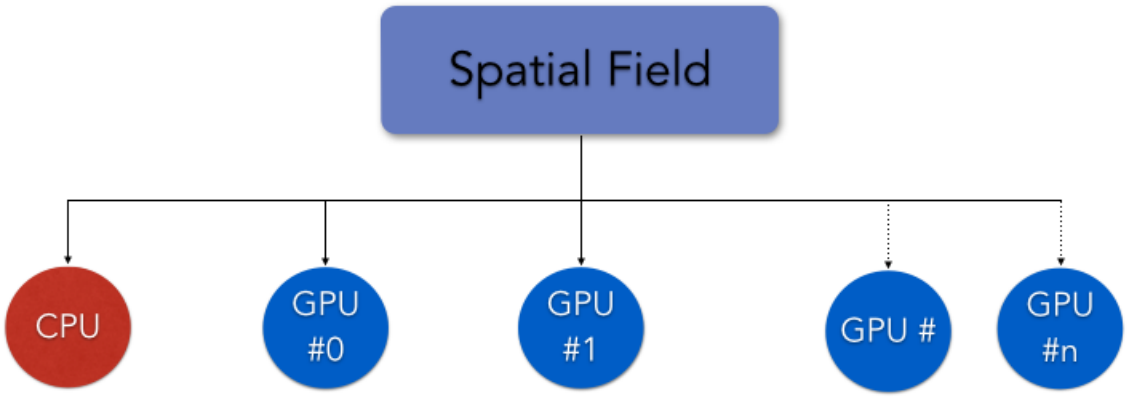

Figure 2.1: Spatial field with multiple field locations. A field can have multiple hardware targets like a CPU location and multiple GPU locations with varying indices.

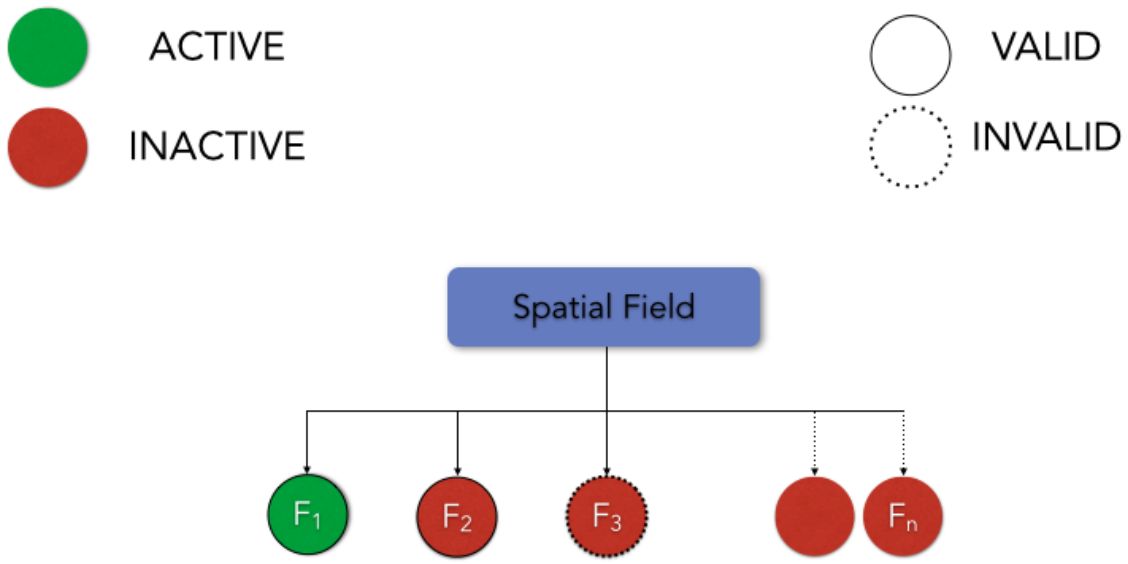

Figure 2.2: Characteristics of a field with multiple locations. Field locations colored in green are ACTIVE and in red are INACTIVE. A solid boundary indicates that a field location is VALID and a dotted boundary is INVALID.

be active at the same time, as there is a potential danger of getting modified at multiple locations, which breaks the correctness. This new model has expanded the scope and usage of the fields with multiple device locations.

2.5 Characteristic Traits of a Field

To maintain validity and correctness of a spatial field with multiple field locations, a field is characterized by two different states, *active* and *valid*. The active state of a field location gives information about the write access permissions for a particular field location out of multiple field locations. Any assignment to a field occurs on the active location of the field. No more than one field location can be active at the same time, which means that inactive field locations are restricted for write access. This is because the write access is performed with the help of iterators and indexing operators and that could possibly invalidate the field values resulting in loosing the correctness.

The other characteristic state of a field is valid, which gives information about the validness of the data within a field at a particular device location. When a field is valid, constant iterators and indexing operators are used for providing read-only access to the field values. Possible traits for a field are summarized in Table 2.1

Table 2.1: Characteristic traits for a field

States	**Valid**	**Active**	**Summary**
Case 1	✓	✓	A field can be in a state of both valid and active.
Case 2		✓	A field can never be in a state of active without being valid.
Case 3	✓		A field can be in a state of valid without being active.
Case 4			A field is neither active nor valid and needs a validation.

CHAPTER 3

TASK GRAPH EXECUTION AND SCHEDULING

3.1 Overview

Task graphs are employed in many areas of parallel computing [16] — [17], distributed computing [18], hardware task scheduling [19], [20]. A task graph is used for representing a task with communication, computations and dependency structure analysis of the program. The task graph considered in the Expression Library (ExprLib) is a simple directed acyclic graph (DAG), which is the primary graph model used for task scheduling. A DAG is a specialized case of a task graph that will not have circular dependencies within the nodes.

Task graph is a visualization in which nodes or vertices are connected by the edges pointing in the direction based on the flow of execution or dependency relation. The task graph representation gives an effective model to solve a multiscale, multiphysics problem on high-performance computing architectures. This idea can be used effectively in scheduling a task graph on Hybrid computing systems and multicore computing systems.

The main abstraction of ExprLib is a task graph. A task is a calculation often written in Nebo assignments that produces a field (or fields) and requires other fields. Each task advertises the fields it requires and the fields it produces. The field dependencies between tasks create a directed, acyclic graph (DAG) and can depend upon runtime information.

ExprLib provides functionality that allows a developer to completely focus on writing code that reflects mathematical expressions while removing much of the complexity associated with discretization and algorithm development. ExprLib provides heuristics to generate algorithms based on graph theory so that a task graph can be scheduled and executed on multicore, many-core (GPUs) computing systems. It also supplies a few different explicit time-integration schemes. This would allows one to create complex algorithms simply by specifying the dependency among various expressions taking part in the task graph. A task graph that describes complex multiphysics problems can be executed on computing system capable of parallelizing the operation on a hybrid computing system with GPUs.

Some of the key functionality that the ExprLib achieves

- Supports multithreading, GPGPU for task-based parallelism.
- Automates the process of memory management for fields on multiple hardware targets.

3.1.1 Dependency Task Graph

Dependency graph, the name itself suggests that it describes the dependency relation of each node connected by edges. The nodes in a graph are responsible for the computations and edges provides information about the dependency between nodes. This graph is constructed by introspecting the final resultant expression and querying for its dependencies. Figure 3.1 shows an illustration of a dependency graph with relation between nodes.

Introspection of the graph is necessary to determine the properties of a graph before it is used for task scheduling and execution. A count of expressions, in-edges, out-edges, and field size are some of the important parameters to be taken into account while introspecting a task graph. The number of expression determines the graph density, and the field size gives an estimate of node density. A root node is the starting point and generally a resultant expression for constructing a dependency task graph. The bottom most node in a dependency graph does not have any dependencies and serves as the starting point for the task execution, as discussed in §3.2.

3.1.2 Execution Task Graph

An inverse or transpose of edges of a dependency graph would result in a execution graph. This means that the root nodes that are at the at the top of the dependency graph will be the last set of tasks to be executed. The expressions computed from the bottom of the dependency graph reach to the top node of the graph meeting the dependencies and performing computations. The dependency graph represents our intuitive understanding of how a series of operations forming an algorithm are connected and will serve as an initial step in the process of translating a real model description into its graph representation. Figure 3.2 shows an illustration for flow of execution for a execution task graph.

Node hardware execution targets are assigned based on the properties of the underlying expressions, and it is an important property that determines the nature of a task graph described in §3.1.3 and §3.1.4. Each node is examined for a count on number of in-edges and out-edges. The out-edges determines a count on number of consumers that require field or fields produced for their respective calculations. In-edges determines a count on number of dependency relations to achieve for a given node to start the computations.

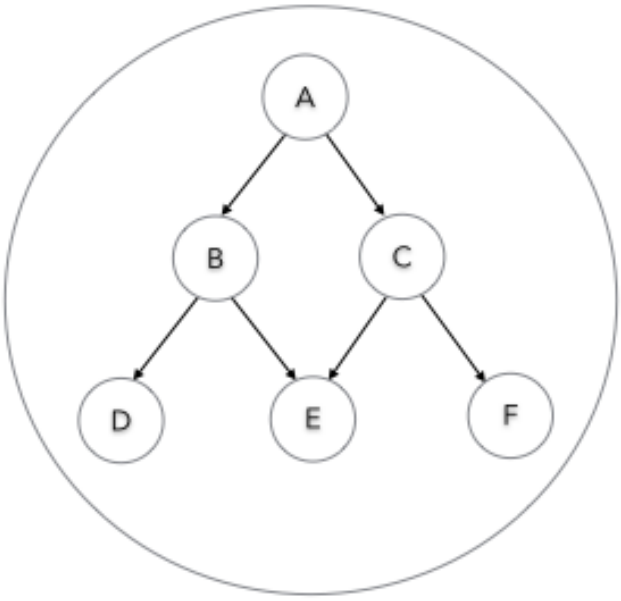

Figure 3.1: Nodes of a task graph showing the dependency relation. The arrows connecting the nodes shows the direction of dependence.

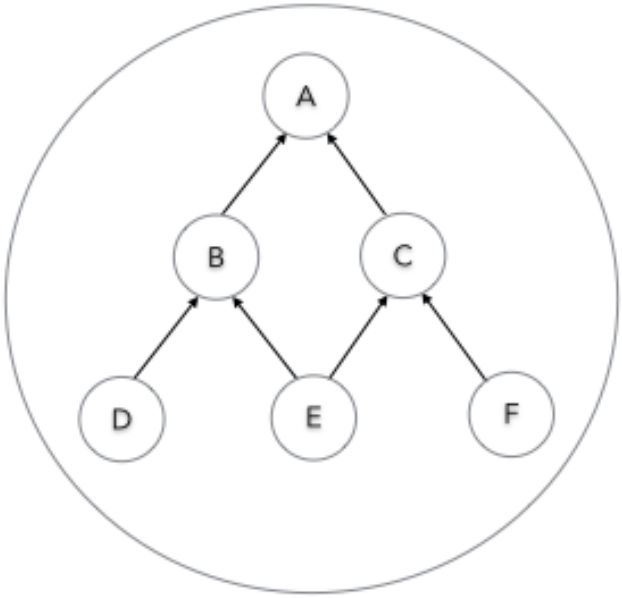

Figure 3.2: Nodes of a task graph showing the flow of execution. The arrows connecting the nodes shows the direction of execution.

3.1.3 Homogeneous Task Graph

Task graph can be classified into two types based on their node execution hardware target information. The first type is homogeneous task graph, in which all the nodes, including topologic edge nodes, have the same hardware targets assigned to it. These hardware targets can be single or multicore architecture without any accelerators designated as CPU. Figure 3.3 shows an illustration of a homogeneous task graph where each node is assigned the same hardware target.

3.1.4 Heterogeneous Task Graph

The second type is a heterogeneous task graph in which only some nodes can be executed on GPU, whereas others cannot. This is practically encountered for cases in which the code performing computations is not suitable for parallelization, which implies that some nodes have execution hardware target as GPU and others as CPU only. Figure 3.4 shows an illustration of a heterogeneous task graph where each node is assigned with a different hardware target.

3.2 CUDA Resources

When a expression is created, it is associated with unique CUDA resource handles, *cudaStream*. CUDA streams are necessary for an expression in managing computations and data transfer operations on a device and are discussed in §3.2.1.

3.2.1 CUDA Streams

A stream is a resource handle of sequence of instructions that execute in order. Different streams, on the other hand, may execute their instructions out of order with respect to one another. A stream is defined by creating a CUDA stream object and specifying it as the stream parameter to a sequence of kernel executions or for host-to-device or device-to-host asynchronous data transfers. In case of a NULL stream, the operations will not be performed until all the preceding operations on the GPU have been completed. Streams are helpful to launch as many kernels as the device capability supports that can run concurrently with a host-to-device and a device-to-host asynchronous copy before there is a need for synchronization.

CUDA streams play an important role for expressions that can perform calculations on GPU. Expressions has its own CUDA stream, and it ensures the performance of sequential execution of calculations written in the form of Nebo statements [15]. Every expression has it individual CUDA stream, and they can be executed out of order to explore task

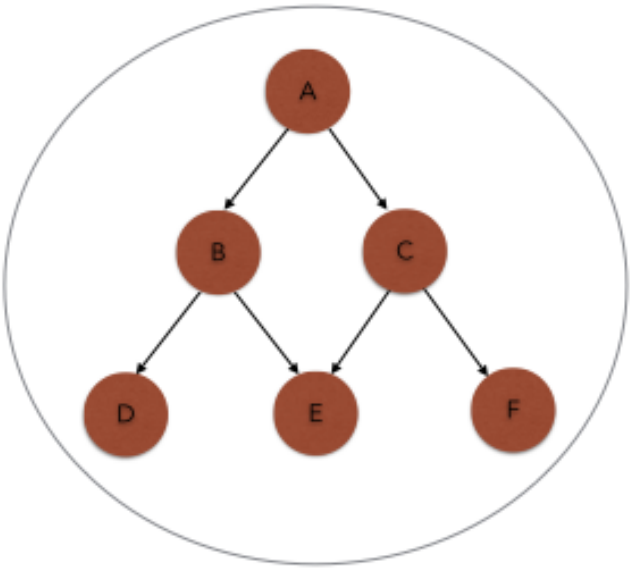

Figure 3.3: Homogeneous task graph with the nodes having same hardware target. This implies that there is no data-transfer between any other hardware targets.

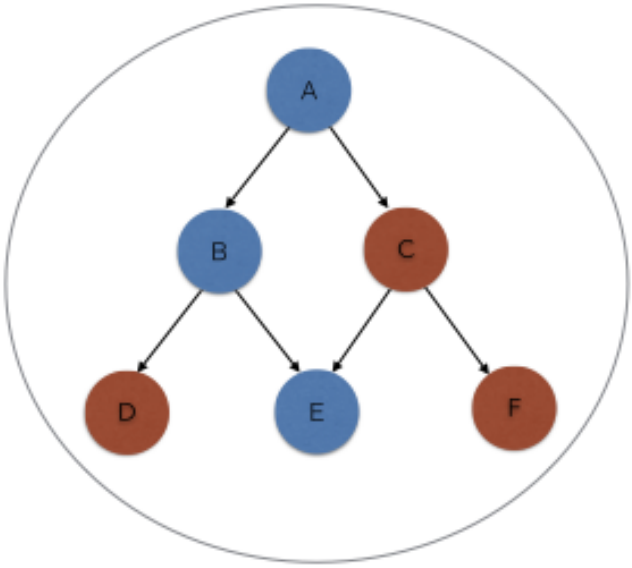

Figure 3.4: Heterogeneous task graph with the nodes having different hardware target. This implies that there is a need to perform data-transfer between the nodes of different colors indicating different hardware targets.

parallelism and concurrency on the device. Usage of CUDA stream for an expression helps in triggering a asynchronous execution with respect to host thread and enables the host thread to launch all the other expressions that are ready in the queue to begin execution. Detailed usage of CUDA stream with a task graph is explained in §3.4

3.2.2 CUDA Kernels

The CUDA kernels are analogous to functions in C / C++, and it is important to know about the launch, execution characteristics, and resource management considerations. The kernels are asynchronous functions that can concurrently be executed with GPU. The meaning of asynchronous is that the control of the thread returns to the CPU before the GPU has actually completed its requested operations. The main purpose of having the asynchronous state for the kernel is to hide the driver overhead when performing multiple kernel launches consecutively.

A CUDA kernel is launched by specifying an array of threads, stream ID, and block ID, which are often passed as parameters to the kernel. There is nothing special about passing parameters to a kernel. The triple angle-bracket <<< >>> syntax or *execution configuration* is used to define a CUDA kernel. Each thread that is responsible to execute the kernel is given a unique ID that is readily accessible to the kernel through a built-in threadIdx variable. The runtime system takes care of any complexity introduced by the fact that these parameters need to get from the host to the device. A CUDA kernel has to be defined using a `global` keyword declaration specifier so that it is qualified to be invoked from a host (CPU) or a device (GPU). If a kernel gets launched with a zero or default stream ID, the operations are performed in the same order as that of the launch sequence.

When a kernel gets launched, it runs with the help of a grid of block of threads. It is not necessary that all the blocks run concurrently. Each block is assigned to a streaming multiprocessor (SM), and each SM maintains the context for multiple blocks. The CUDA programming model does not guarantee for the order of execution or the concurrency of the blocks or threads. It is the responsibility of the developer to ensure that race condition is avoided by maintaining order of execution decisions. The maximum number of kernel launches that a device can execute concurrently is 32 on devices with compute capability 3.5 and 16 on devices with lower compute capability [7].

3.3 Task Graph Scheduling

ExprLib uses a scheduler, a priority queue, and a threadpool to exploit task parallelism. When all the dependencies of a task are completed, the scheduler adds the task to the

priority queue. The threads in the threadpool repeatedly pull tasks from the priority queue. When a thread finishes a task, the thread informs the scheduler so that other tasks that depend on the just finished task can be scheduled.

For memory management, ExprLib uses memory pools as discussed in §3.4.2, one for each type of memory (CPU or GPU). When a task begins execution, the appropriate memory memory management procedures serves the memory requirement for the scratch fields and persistent fields as discussed in §2.2. After a field is used by all the tasks that depend upon it, the field's memory is returned to the appropriate memory pool. Thus, the same memory can be used for multiple fields produced from different tasks in the same graph.

With this view of task-graph, ExprLib has the appropriate information to determine where to deploy a task on a heterogeneous system and manages memory transfers between host or device. Each task is either tagged as runnable on CPU or GPU, with tasks that are written using Nebo syntax explicitly tagged as GPU runnable by default. Tasks that contain handwritten C++ code are tagged as CPU only, and tasks that contain only Nebo assignments are tagged as runnable on GPU because of its offered support. Depending upon the hardware location of the fields, ExprLib's scheduler performs the necessary procedures for memory management and scheduling of tasks.

A GPU-enabled task is likely to run on CPU if its input fields are produced by CPU-only tasks and its output fields are used by CPU-only tasks. The reason is to avoid the overhead obtained from data transfer of fields between host and device. On the other hand, a chain of GPU-enabled tasks is likely to run on the GPU. In both the above stated cases, the decision was based on avoiding expensive data transfers between CPU and GPU. Currently, we are working on heuristics for minimizing the overall execution time of a heterogeneous task graph with smart scheduling heuristics.

Finally, for a GPU task execution, it bears mentioning that each task is assigned a different CUDA stream. This stream manages high-level concurrency between CUDA kernels as explained in §3.2.1 and §3.2.2. Each Nebo assignment corresponds to a single CUDA kernel, and each ExprLib task can contain multiple Nebo assignments. Since each task has a unique stream, multiple Nebo assignments within a task result in sequential kernel execution, maintaining order of execution within a task, but allowing out-of-order execution among tasks, moderated by the scheduler.

3.3.1 Task Scheduling on a Multithreaded System

The precise semantics of threads differ from one operating system to another, but in general the threads of a single program are akin to multiple processes except that they

share the same address space. This means that they can all examine and modify the same variables. On the other hand, each thread has its own registers and execution stack and perhaps private memory. Multithreading execution refers to an application with multiple threads running within a process supplied by the operating system. A thread is a stream of instructions that is an entity in a process. Each thread is process/task specific and has its own registers and stack memory. The virtual address space is process specific or common to all threads within a process. So the data on a heap are shared or can be accessed by all the threads within a process. Multithreading can also be applied to a single process to enable parallel execution on a multiprocessing system. Threads require mutually exclusive operations often implemented in semaphores in order to prevent common data from being simultaneously modified or read while in execution of a process.

The Expression Library framework is capable of exploiting the task level parallelism by decomposing the task and its corresponding expressions. A thread pool is maintained, and each individual thread is assigned with a set of expressions to be executed. The inherent parallelism at the Nebo operator level uses a first in, first out (FIFO) thread pool [15]. As the Nebo operator thread pool is accessible to the task scheduler, higher flexibility is attained to address computational block necks.

A thread pool is maintained so that the tasks are assigned to an available number of active threads. Each task is assigned to a thread and executed sequentially on a particular thread. Every task exposes its fields for the data parallelism available by sub dividing the fields such that each subfield is assigned with a thread at the Nebo level. Threads at task level and at the Nebo level share computational work such that idle time for a thread is minimized. This can be easily determined by providing the scheduler with the total number of processing resources that are allocated from a thread pool. More details on the multithreaded scheduling of tasks is not under the scope of this work.

3.3.2 Task Scheduling on Hybrid CPU-GPU Architecture

An important emerging trend in high performance computing is to use both the CPU and GPUs to reach the desired exascale computing limits. This would require a software framework that manages complexities in handling task scheduling, memory management, and computations on these hybrid architectures. Such a framework must address the above stated challenges to meet scalability and performance. Scheduling a task to the computing systems augmented with graphics processing units (GPUs) is a complex task that involves efficient memory management on both the CPU and GPUs on-node. For computing systems with on-node multiple GPUs, the scheduler must additionally manage a CUDA context for

each device.

A heterogeneous task graph with complex dependencies can be scheduled using a hybrid CPU-GPU scheduler present in ExprLib. In this case, a scheduler should manage task ordering along with computations and data transfers. The scheduler iterates from its root nodes and makes a query for all its dependency expressions. During this phase, the scheduler collects information about consumer and dependent expressions to maintain a counter for each node in the task graph.

Every vertex is tagged as CPU or GPU depending upon the properties of the expressions within it. The hardware execution target is assigned based on the property of expressions in any given vertex. If an expression in a vertex is capable of running on the GPU, the node is assigned to be GPU runnable and for other cases as CPU runnable. After hardware targets are assigned to the vertices, suitable memory manager tags are determined to manage memory for the fields within a given vertex. These tags aids the memory managers with mechanisms such as locking a field's memory, tagging them as persistent for other subsequent operations, and managing the memory appropriately that is owned by other sources.

3.4 Task Graph Execution

After the task scheduling phase, execution of a task depends on the nature of the task parallelism involved in it. Details on task execution with multithreading is not in the scope of this work. When an expression reports its completion, a callback mechanism is triggered that updates consumer and dependency vertices as discussed in §3.3.2. If a vertex attains all its dependencies, the task is populated to the queue for execution. A thread launches the execution of the vertex based on its assigned hardware target.

For a heterogeneous graph introduced in §3.1.4, there is a need for moving fields from one hardware target to another so that the vertex attains all the fields on the appropriate hardware target assigned for computations. There can be a case where a field/fields produced on the CPU is required on the GPU or vice-versa, so a data transfer is performed based on the source and destination hardware targets for each vertex. In case of a densely populated task graph, memory associated with the fields that are produced and have been used by the other vertices stay idle and not being used. Memory can be reclaimed and reused from these dormant fields, and the details are explained in §3.4.1.

While executing a heterogeneous task graph, there is a chance that computations and data transfer can be overlapped on a device with the help of a stream and also currently perform work on the CPU as well. This asynchronous mode is of vital importance for a

heterogenous task graph execution, and details regarding asynchronous mode are explained in §3.4.3.

3.4.1 Memory Resources Reuse

For a heterogeneous or homogeneous task graph model, a better understanding of memory usage is necessary for improving performance and reducing memory requirements. When a field/fields are produced as a result from computations, it is supplied as an input for the upstream dependency expressions. After the computations in all the consumer vertices or expressions are completed, their input fields are no longer necessary and remain dormant for the rest of the time step. The memory associated with these fields can be deallocated so that the same can be reused for other fields. This process will prevent any unnecesary memory management heuristics from being performed on other fields. Before releasing memory for the dormant fields, a check is performed to ensure that a field is not locked for persistence requirements. Unless a field is marked as persistent, which means that the memory associated with the field should exists at each step and not be deallocated. The persistency requirements are enforced on to the fields so that they can be used for the time-stepping purposes, and hence memory deallocation is not desired.

Consider a task graph shown in Figure 3.5, which has 10 nodes named from A till J, out of which A, B, E, H, and J are tagged for GPU execution and C, D, F, G, and I for CPU execution. Topologic edge nodes in a task graph like A, G, H, I, and J are tagged to be persistent. During execution, the nodes G, H, I, and J act as starting nodes for a graph execution. As the computations in the starting nodes (G,H,I, and J) are finished with their computations, their field values are made available for their consumer expressions. The fields G, H, I, and J are made available for vertices D, E, and F based on their dependency relation. Once the computations in the vertices D, E, and F are complete, input fields G, H, I, and J are no longer required. These fields can be deallocated, but as they are tagged to be persistent for their memory to reuse.

As the nodes D, E, and F completes their computations and made their fields available as inputs for vertices B and C, which are its consumers. Once the vertices D, E and F complete their calculations, the memory associated with their input fields can be deallocated as they are no longer required and also not tagged to be persistent. Similarly, nodes B and C have made their fields available for performing computations on node A, and after completion, memory for fields B and C can also be released. It is very important to note that node A, which is a final resultant topologic edge node should not be deallocated. When releasing the memory of a dormant field that is no longer required, it is important to note that the

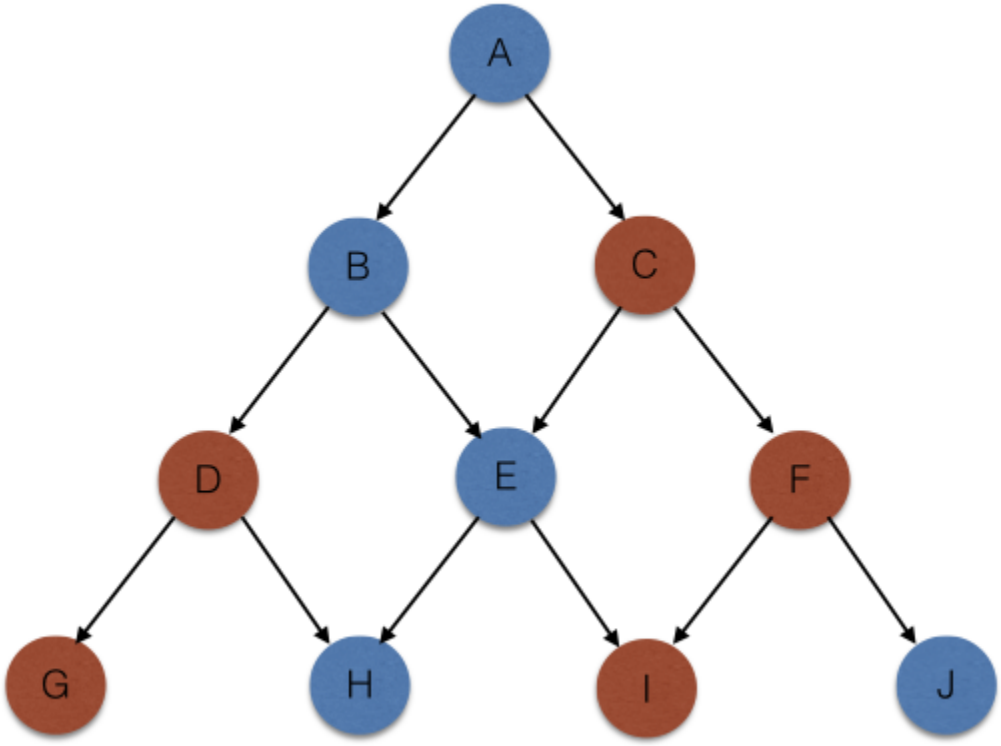

Figure 3.5: Illustration of a heterogeneous task graph. The nodes colored in blue are GPU runnable, and nodes in red are CPU runnable. The memory resources allocated for the nodes can be reused for a better memory management.

dormant fields does not release the memory until the computations in the expressions using these fields as inputs are complete.

3.4.2 Memory Pools

The motivation behind using a memory pool is to minimize the memory latency overhead when performing repeated dynamic memory allocation and deallocation procedures. Fields that are no longer required are deallocated, and the memory is released to the operating system so that the new fields can get the memory. Instead of deallocating and performing an allocation when memory is required for a field, the memory can be put into a pool so that a new field can reuse the memory by pulling it out of the pool. This will avoid the overhead generated from performing repeated allocation and deallocation.

A memory pool is a preallocated memory space assigned with a fixed size, which is a high water level. The high water level mark is the safest maximum allowable memory requirement for running an application without starving the application during runtime. The high water mark can be approximately estimated by the parameters like the number

of fields actively participating at each time step, problem size, etc., for the simulation. As the number of fields created and destroyed at each time step remains same, estimating the high water mark can be approximated with ease. If the memory consumption increases with each time step, then it is a clear indication that a potential memory leak exists in the simulation as the field count should not change for every time step.

The dynamic memory allocation and deallocation procedures on both CPU and GPU would add a significant overhead because of its repeated calls to the operators `new`/ `cudaMalloc` or `delete`/`cudaFree`. To avoid this, separate memory pools for the CPU and GPU are maintained from which memory is pooled based on the requirements for the fields. These memory pools are destroyed at the end of the simulation, freeing the memory to the operating system. The memory for a new field object that is required as an internal-storage mode described in §2.3 is allocated and discarded after their lifetime. The CPU memory requirements from the pool are served by CUDA pinned memory for a GPU-build targeting heterogeneous CPU-GPU tasks. Further details on pinned memory support for the heterogenous tasks are explained in §3.6.

The memory for fields on a device is provided by a GPU memory pool. Memory allocation for a device is accomplished by CUDA API `cudaMalloc` and when a pool is destroyed, its memory is released to the operating system by using CUDA API `cudaFree`. Returning memory to an incorrect memory pool would result in an undefined behavior because of the difference in memory management procedures. As shown in Figure 3.5, the memory associated with fields B, C, D, E, and F can be deallocated so that it can be reassigned to a new field. Fields B and E have memory on GPU, managed by GPU memory pools and fields C, D, and F are managed by CPU memory pools.

3.4.3 ExprLib Task Execution Configuration

The task execution configuration depends on many factors, such as nature of task graph, whether it is homogeneous or heterogeneous, node hardware targets and the dependency relation between them. In this section, the execution configuration for a homogeneous GPU task and heterogeneous CPU-GPU task graph is elaborately described. A single host thread is used for the task execution. Every node in the task graph is assumed to have enough computational complexity. Nature of a task graph is introspected at the scheduling phase where all the nodes in a task graph are traversed. All the nodes are queried for the GPU executable properties of the expressions within it and based on that, a decision is made whether a task is a homogeneous or heterogeneous in nature.

Based on the nature of the nodes in a graph, the task can be classified as follows

- Homogeneous GPU task
- Heterogeneous CPU-GPU task

3.4.3.1 Homogeneous GPU Task

Homogeneous GPU task execution is explained using Figure 3.6 for a single threaded execution on host and using a single GPU. As shown in Figure 3.6, there are six nodes, each consisting of expressions capable of running on a GPU. To explore the asynchronous capability on the device, it is necessary to use `cudaStream` and hence, every expression is assigned with a stream. A single host thread is used and hence, the launch sequence is as follows: node D is launched first, followed by E and F. As the configuration is asynchronous, the host thread launches the calculations on the device and returns without waiting for the calculations to actually complete on the device. With an assumption of having enough computational complexity, these nodes are capable of executing concurrently on the device as they have independent streams. If an expression is tagged as GPU executable, each Nebo statement within the node is treated as a separate CUDA kernel that shares the same stream assigned for the expression. Kernels in the expression are in-order execution as queued up by the hardware thread. Hence the execution of Nebo statements within an expression is synchronous with respect to the device, but still it is asynchronous with respect to the host. With the help of the above mentioned functionality, both the copy and compute engines on the GPU can be concurrent.

Once all nodes D, E, and F in the lower level are computed, the host thread is ready to launch their respective consumers. For executing node B, the calculations in the nodes D and E have to be complete and their fields made available. As the calculations are asynchronous with respect to host, there is a potential danger of launching new expressions whose dependencies are met as seen by the host thread, but the device-side work might still be not complete. The status of work on the `cudaStream` is checked using CUDA API `cudaStreamQuery`, which returns `cudaSuccess` when there are no pending instructions. If the stream is still performing the work, an explicit synchronization is performed by using CUDA API `cudaStreamSynchronize`. This process ensures that any race conditions are avoided.

It is necessary to check if the `cudaStreams` of the respective dependencies (node D, E, and F) have completed the set of instructions assigned to them. A check on `cudaStream` is performed before the consumers (node B / C) is launched for the computations to exploit the advantages of avoiding explicit synchronizations. For the last node, the calculations performed for vertex A are also asynchronous and hence, a synchronization is performed to

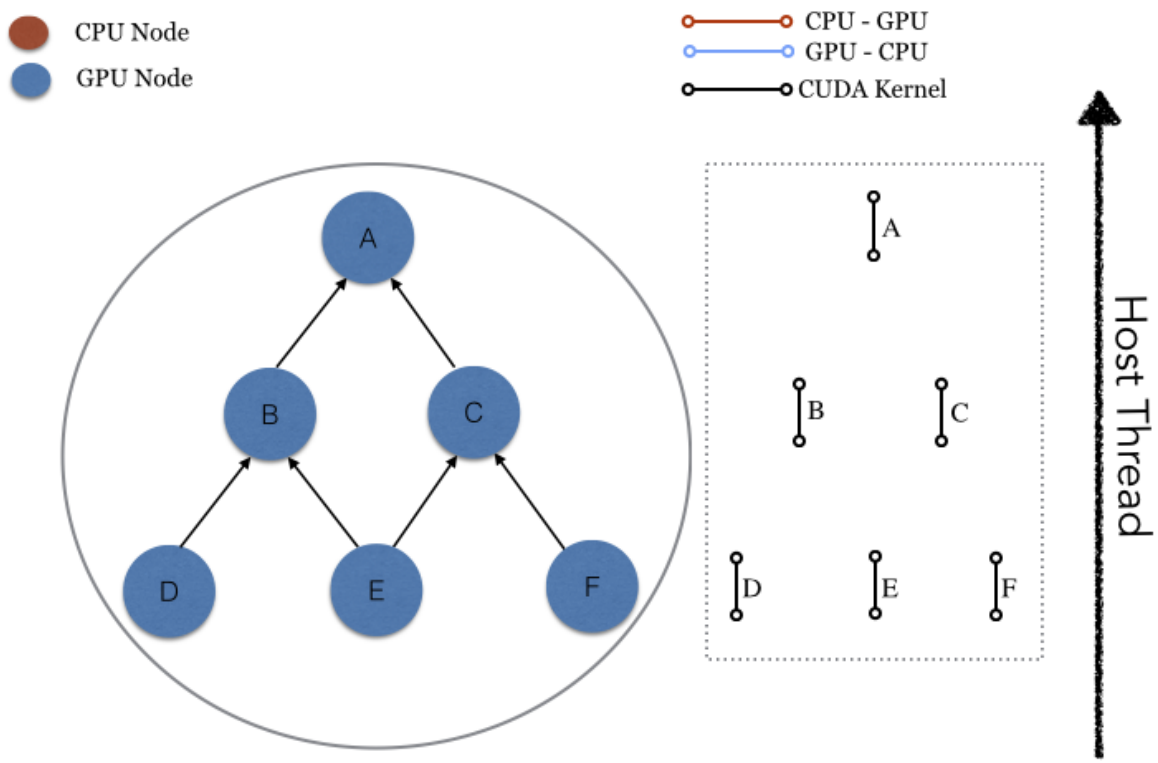

Figure 3.6: Homogeneous GPU task execution using asynchronous mode. Configurations of hardware targets, thread resources and the order of kernel execution is shown.

ensure that the task execution is complete. The host thread is made to wait until the work on the stream associated with the expression A is complete on the device.

3.4.3.2 Heterogeneous CPU-GPU Task

Heterogeneous CPU-GPU task execution is explained using Figure 3.7 for a single threaded execution on host and using a single GPU. The node hardware targets are already assigned, as explained in §3.3.2. The distinct feature that is involved in heterogeneous graphs is that a data transfer can be performed along with the computations. The main purpose of using asynchronous task execution is to overlap data transfer with computation kernels wherever possible. All the expressions whether they are tagged for CPU or GPU execution are assigned with a stream.

A single host thread is used and hence, the launch sequence is as follows: node D is launched first, followed by nodes E and F. Calculations are performed on the CPU when the host thread invokes node D. Once the calculations for node D are complete, data transfer from host-to-device is also performed, as it is required for one of its consumer (node B) to have the field on GPU. The data transfer is asynchronous with respect to the host and hence, the control of the host thread returns immediately after launching it. Node E is tagged as GPU executable and a data transfer from device to host is also required, as one of its consumers (node C) needs to have the field produced from E on CPU. The calculations and data transfer associated with an expression are asynchronous with respect to host and share the same stream. This ensures that the instructions are in-order so that data transfer is performed only after the calculations are complete. There are some synchronous instructions on the device but still asynchronous with respect to host. The host thread after launching the work for node E can start executing the calculations in node F on the CPU. At this point, there can be a data transfer from device to host performed for node D, calculations on device, and device to host data transfer for node E, and calculations for node F on host can go concurrently. This is illustrated in Figure 3.7, using lines with filled circles. As the configuration is asynchronous, a check has to be performed before launching a new expression just as explained in §3.4.3.1.

The concurrency can also be attained with the help of multithreaded parallelism at task level in which threads would start launching multiple expressions that are ready to be executed and in turn launching its kernels underneath the expressions. Using multithreading for launching tasks, multiple CPU and GPU expressions can be launched as opposed to single-threaded host task execution. Further details regarding the multithreaded host-GPU execution model is not in the scope of this discussion.

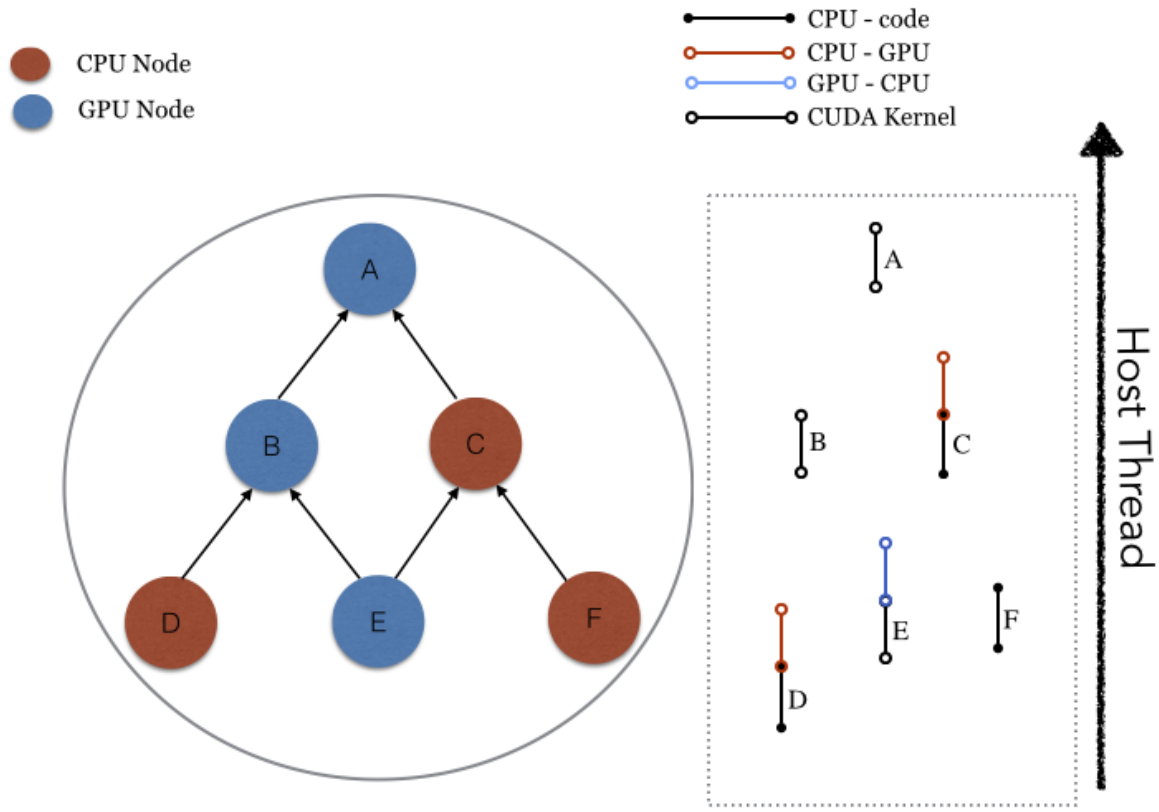

Figure 3.7: Heterogeneous GPU task execution using asynchronous mode. Configurations of hardware targets, thread resources, and the order of data-transfer and kernel execution is shown.

3.5 Device Selection for On-node Multi-GPUs

Device selection and assignment is an important task while working with a heterogeneous CPU-GPU task graph or a Homogeneous GPU task graph. The host thread can set the device context by making a call to a CUDA API `cudaSetDevice`. The device context is set before using any of the CUDA APIs for device operations. This ensures a proper device context is assigned before invoking any CUDA API functions. Maintaing device context is important while using CUDA resource handles like `cudaStream` and `cudaEvent` for operations such as status query, synchronization, creation, and deletion of resources. Using the resource handles with an invalid device context would lead to undefined behavior.

The selection and assignment of a device to a task graph is operated by different entities depending upon the environment of the simulations. In a brief summary, for a simulation outside of Wasatch, the hybrid scheduler operating within ExprLib handles the device selection and assignment for a given task graph by round robin fashion. For Wasatch, assigning a device to a task depends upon the nature of the task graph and the controlling entity. For a homogeneous GPU task graph, the controlling entity to assign device index for a task is bestowed with Uintah. Uintah's Unified Scheduler supports a dynamic, round robin assignment of device per task [14], and further details are not in the scope of this discussion. For a heterogeneous task graph, the controlling entity to assign the device index is with Wasatch, as information about the GPU runnable properties of a task graph are not known to Uintah.

3.6 Optimization Strategies

The most important optimization technique used to perform the asynchronous operations is to use CPU/GPU concurrency, which hides the memory management overhead and the memory transfer overhead of CUDA runtime API launches. The CUDA driver takes valuable CPU time to write instructions to the GPU, and hence, overlapping that CPU execution with the GPU processing can improve the performance.

3.6.1 Pinned and Pageable Memory

Understanding pageable and pinned memory is critical for asynchronous memory copies using `cudaMemcpyAsync`. When memory is allocated on the CPU (host) by using standard `new` or `malloc` operator, the memory is pageable by default. The GPU (device) cannot access this pageable memory directly and hence, when a host-to-device or device-to-host memory transfer is invoked, the CUDA runtime driver allocates a page-locked or a temporary buffer so that host data structure is copied to the staging area. Once the data are on

the staging area, the data are copied to the device memory. This procedure involves two stages of data transfer from host memory to a staging buffer and then moving the data from the staging buffer to the device memory. This limits the bandwidth and increases the time for completing a data transfer.

With the use of `cudaMallocHost` or `cudaAllocHost` API provided by CUDA, memory is allocated on the host with pinned privileges. There is certainly an overhead on using these methods. However, the overhead can be hidden if the simulation occurs over a prolonged time scale with fairly higher problem sizes. Pinned memory is a bit expensive procedure for allocating and deallocating but provides higher transfer throughputs by effectively utilizing the available bandwidth. The pinned memory allocation occurs only at the first time step when the memory pool is constructed and deallocated when the memory pool is destructed and hence, it minimizes the calls to this API.

Figure 3.8 gives information of a test case that is run over problem sizes varying from 16^3 to 128^3, and it can be clearly seen that there is a significant advantage with using the pinned memory over pageable memory. The computations involved in both versions are the same and the only difference is the effect of using pinned memory over pageable memory while performing data transfer. At a given problem size of 128^3, the percentage of computation for both the cases is the same, and the data transfer is about 57% of total computation time while using the pageable memory, and a percentage of data transfer is about 43% while using pinned memory. Using a lot of pinned memory can degrade the overall performance of the system and hence, the pinned memory should be limited in its usage. The pinned memory can be limited depending upon the system's memory and hence, a failure in allocating pinned memory can be expected due to the shortage of requested memory. In these circumstances, dynamic memory allocation is performed, and the memory is tracked for appropriate deallocation procedures.

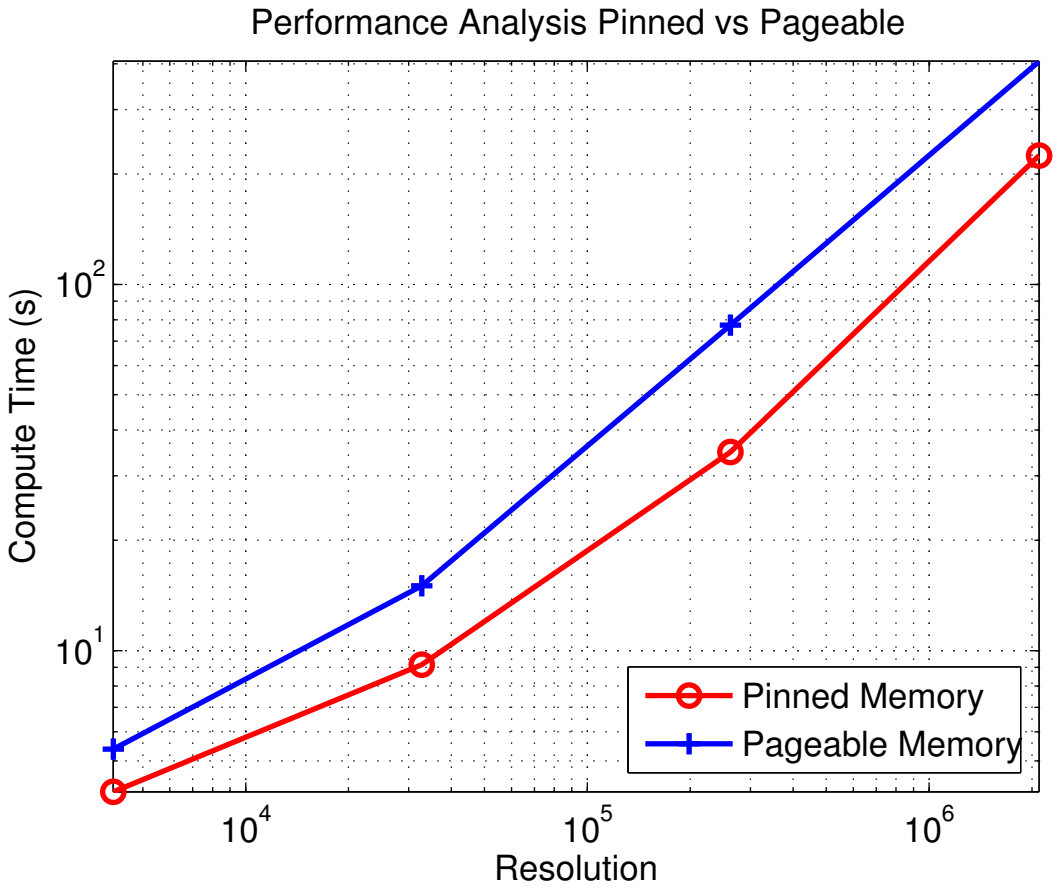

Figure 3.8: Performance analysis of pinned and pageable memory. Inherent data transfer is performed for the problem sizes 16^3, 32^3, 64^3, and 128^3

CHAPTER 4

WASATCH

4.1 Overview

Wasatch [10] is a component of Uintah that interfaces with Expression Library and SpatialOps downstream libraries to facilitate the rapid development of time dependent partial differential equation solvers and upstream with Uintah. Wasatch is designed to be a highly flexible simulation component for solution of transient partial differential equations (PDEs). Based on the same lines of major computational framework like Uintah and a downstream project like Expression Library, Wasatch is also designed based on Graph Theory. However, it allows dynamic construction and introspection of graphs. Each node in a graph defines its direct dependencies, and the graph is constructed recursively with the support of Expression Library. This is in contrast to Uintah where the user defines the graph statically.

The dynamic graph generation support of Wasatch allows very easy definition of complex algorithms and allows a great deal of flexibility when modifying models. Wasatch interfaces with Expression Library for the graph construction and introspection and SpatialOps for the field operations. Similar to Uintah, Wasatch decomposes the simulation model and numeric calculations into a task graph. Each Wasatch task calculates one or more terms of a partial differential equation. Wasatch provides a collection of expressions that can be used to define each term for implementing partial differential equations for a variety of models. Thus end users can quickly and easily design and run simulations on a variety of models. Every Wasatch task is created by Uintah. Unlike Uintah, Wasatch's task graph is inferred at runtime. Every Wasatch task is defined with the fields that it computes and fields that are required for the computation of the task. At runtime, Wasatch recursively constructs the task graph by adding tasks that compute needed dependencies.

4.2 Wasatch Interface

In the current work, the discussion is related to providing support for running a Wasatch task on GPU. The motivation for exploring the Wasatch GPU implementation is the ability for Uintah to run asynchronous, out-of-order scheduling of both CPU and GPU computational tasks. Uintah framework is already in a state to be deployed on traditional systems augmented with the graphic processing units (GPUs) and Intel Xeon Phi.

A Wasatch task is a simple task graph that is exposed to Uintah for the execution. A thin layer of interface is present in Wasatch that shares information regarding a given task with Uintah upstream and to the ExprLib downstream. When a task is created, information about which fields are required to perform calculations in the task, which fields are produced as a result from the task, and which fields act as modifies for a given task is defined. Hence a field mode is defined that tags a given field as REQUIRES, COMPUTES, or MODIFIES for defining a task. When the information about the abstraction of fields is exposed to Uintah, it can be scheduled to the Uintah computational framework (UCF) for task execution. On the other hand, when the Uintah performs the task execution, ExprLib is responsible for the actual task execution with the dynamic graph algorithms present in it. Before the task execution on GPU is elaborated, it is necessary to understand the setup process that is necessary so that Wasatch task complies with the support available from Uintah and ExprLib.

For any given Wasatch task, the fields that are owned and managed by Uintah are tagged as persistent fields, and others that are managed by ExprLib are tagged as scratch fields. These are some of the cases where fields are tagged as persistent and managed by Uintah.

- When a field is required to be saved for I/O purpose, the Uintah manages the field so that the values associated with the field is stored in a proper format to be compatible for a visualization tool.
- Fields that are topologic edge nodes of a task graph are tagged as persistent for time-stepping scheme.

All the other fields are internally managed by ExprLib. These are scratch fields that are discarded to the memory pool after their brief existence as explained in §2.2. Certain limitations have been imposed by Uintah when executing task graphs on GPU. It is required that all the expressions should be GPU runnable and every field associated with it should have GPU memory. To use the support of UCF GPU task scheduling and execution, all the fields should have their memory on GPU so that the fields that are managed by Uintah can register them. This enforces a homogeneous GPU task graph model and exposes its

limitation on task graphs with nodes that cannot run on a GPU. Hence, heterogeneous CPU-GPU task graphs are its limitations, and a support for this is explained in the further section.

For configuring a task to support GPU execution, a simple yet effective analysis is performed to meet the above stated conditions imposed by Uintah for running a GPU task graph. Before a task is exposed to Uintah for scheduling and execution, an introspection is necessary to determine the nature of the task-graph as either homogeneous GPU or heterogeneous CPU-GPU task graph. Depending upon the nature of the task graph, a given task is marked as Uintah's GPU task.

There are two important phases to prepare a task for GPU execution,

- Wasatch Task Setup
- Wasatch Task Execute

4.2.1 Homogeneous Wasatch GPU Task Setup

This is a special case of a Uintah GPU task where all the expressions are GPU runnable and the task graph is said to be a homogeneous GPU task, which meets the conditions imposed by Uintah. A task can be marked as Uintah's GPU task after introspection is performed on all nodes. Once a task is flagged, Uintah's Unified Scheduler [14] performs a setup process for scheduling and execution. When a task is marked as Uintah's GPU task, a `cudaStream` is assigned to it from Uintah. Before the actual task execution begins, configuring the task for GPU runs is important so that the ExprLib is informed about the nature of the fields. An illustration for the setup process on homogeneous GPU tasks is illustrated in Figure 4.1.

At the interface of Wasatch and ExprLib, the fields owned by Uintah are wrapped as spatial fields so that a better field abstractions are used that are compatible for performing operations using Nebo. During the setup phase, ExprLib's Hybrid Scheduler prepares the nodes of a task graph with the assigned hardware targets. Depending upon the ownership of a given field classified as Uintah field or a scratch field, appropriate memory management techniques are adopted. Field mode (REQUIRES, COMPUTES, MODIFIES) is assigned to all the fields managed by Uintah. A GPU data warehouse is defined in Uintah that acts as a container to keep track of all the variables managed by Uintah. When a field mode is defined as requires, the variable is obtained from the data warehouse. For the field mode as computes, the memory is allocated for the variable, and it is registered with the data warehouse and the same follows for field mode modifies. For each field mode, the memory is provided by Uintah and it is wrapped as spatial field.

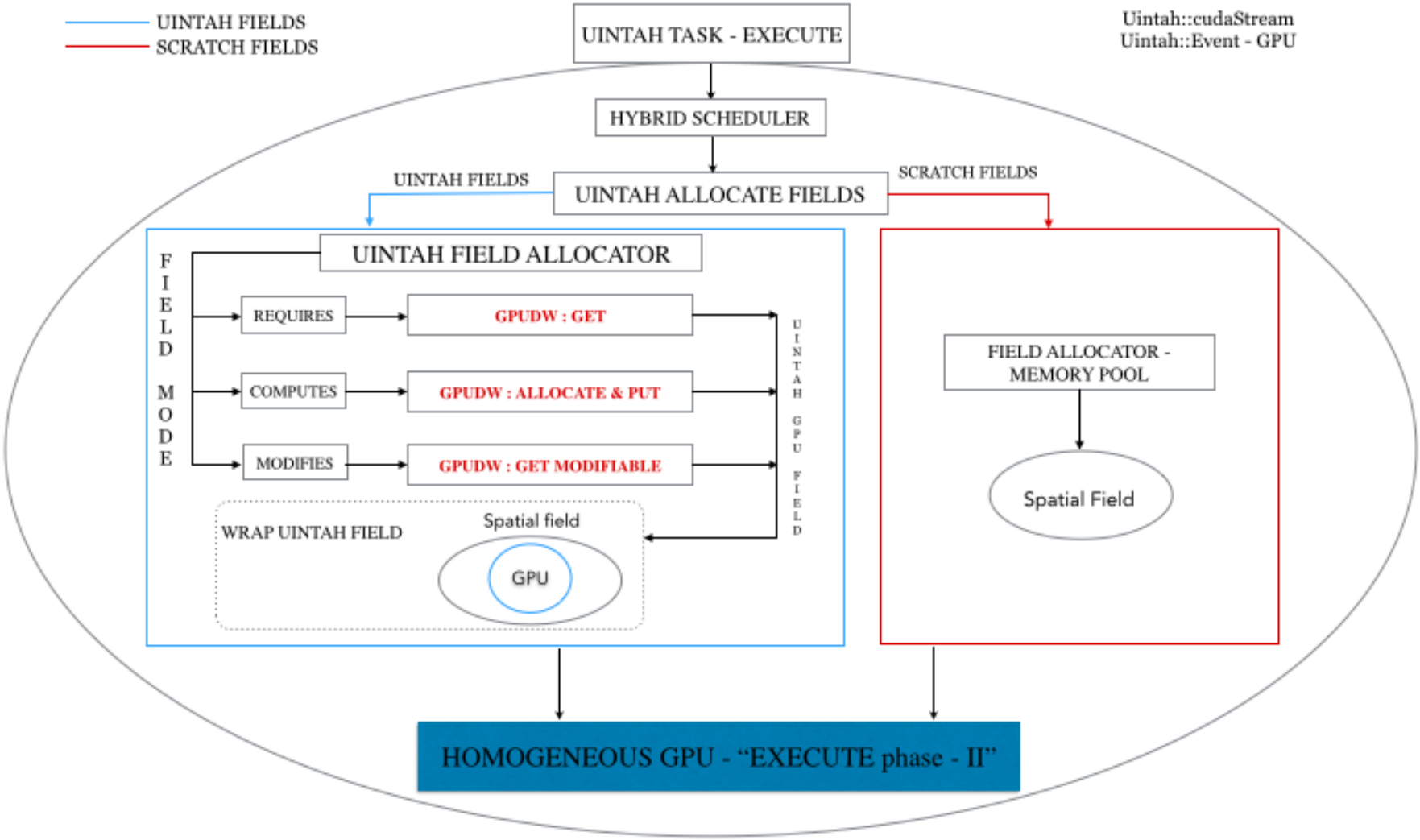

Figure 4.1: Setup phase of a homogeneous GPU task graph. The schematics shows the control flow during a setup phase for a homogeneous GPU task graph.

During wrapping of a Uintah field as a spatial field, memory on CPU is necessary to obtain information regarding the field properties. This information is crucial while performing operations on fields using Nebo. The CPU field supplied by Uintah is temporary and is not registered to a data warehouse. A GPU field is obtained from Uintah as a raw pointer with a valid memory on GPU. Using the information obtained from the CPU field and raw memory of the GPU field, a spatial field is created. The expression library scheduler is responsible for initiating and setting up the memory for scratch fields. This initialization procedure provides an opportunity to explore the low-level task and data parallelism opportunities by ExprLib and SpatialOps projects.

A stream with proper device context is made available for Uintah GPU tasks during execution. The fields owned by Uintah have their memory associated with a particular device context, and the same device context has to be used in downstream projects while creating memory for scratch fields, wrapping Uintah field as a spatial field, assigning node hardware targets to the GPU nodes and creating streams with appropriate device context.

There is certainly a possibility that a device context ID can change during the task execution, and as the control for assigning a device index is vested with Uintah, dynamic assignment of device context occurs for an on-node multi-GPU system. For this condition, the memory associated with the fields has to be deallocated and a new memory with appropriate device index has to be allocated. Streams associated with the expressions are destroyed and recreated. The scheduler has to perform the setup process again to assign appropriate node hardware targets for the GPU nodes.

4.2.2 Heterogeneous Wasatch CPU-GPU Task Setup

Heterogeneous CPU-GPU task graphs are always possible when an expression is written in standalone C++, or the operators that are not supported for a hardware device, or the code is not yet written using Nebo [15] syntax always forces the task computations only on the CPU. This enables the task graph to become a heterogeneous CPU-GPU task in nature. Uintah GPU task scheduling and execution support is limited only to the homogeneous GPU tasks and because of this limitation imposed by Uintah, heterogeneous CPU-GPU tasks would face a potential problem.

When a heterogenous task is found from the introspection for the GPU runnable properties on the expressions, the task is masked to Uintah as a complete CPU only task. Before a task is scheduled for execution, the GPU runnable properties of all the eligible expressions are converted to CPU only as shown in Figure 4.2. Once a task is exposed to Uintah, the GPU runnable properties can later be restored as illustrated in Figure 4.3.

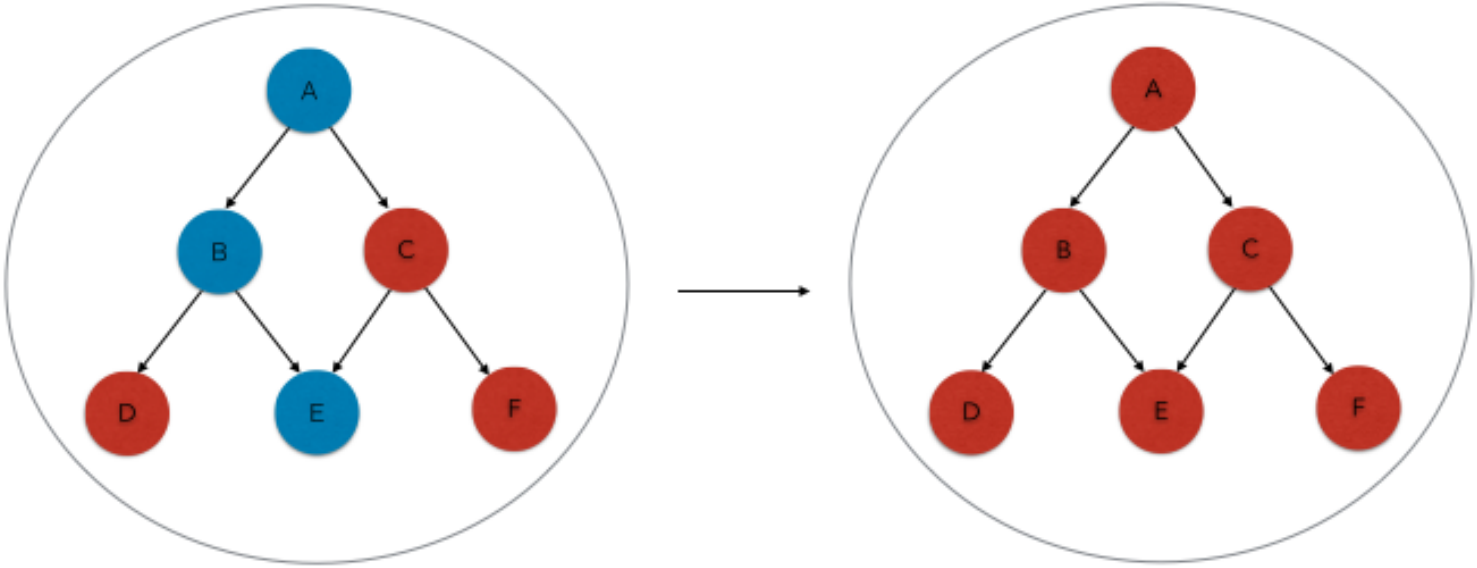

Figure 4.2: Flipping the GPU-runnable properties of expressions. GPU runnable expressions (colored in blue) are converted to CPU runnable expressions (colored in red) before a task is exposed to Uintah framework for a heterogeneous CPU-GPU task graph execution.

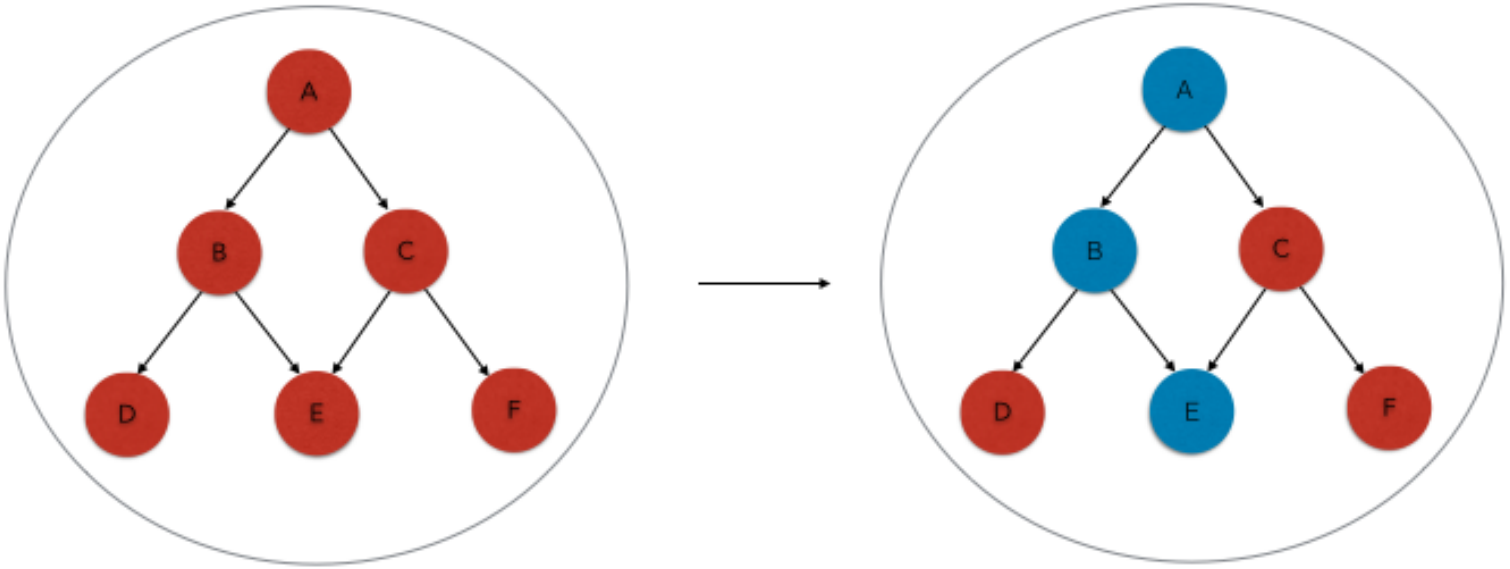

Figure 4.3: Restoration of the GPU-runnable properties of the expressions. Properties that were flipped for processing a heterogeneous CPU-GPU task graph are restored.

The setup process involved in configuring a heterogeneous CPU-GPU task is almost the same as that of a homogeneous task graph. As explained in §4.2.1, there are two types of fields—Uintah field and scratch field. Once a mode (COMPUTES / REQUIRES / MODIFIES) is assigned to the fields, they can be wrapped as a spatial field. The Uintah framework is not aware that some of the expressions and its fields are eligible for executing on GPU, the GPU data warehouse is not created; instead, a CPU data warehouse is made available. This CPU data warehouse keeps track of all the Uintah based fields. The memory obtained for wrapping a Uintah field is only on the CPU. Hence, all the eligible GPU fields will create a memory location on GPU with appropriate device context using the information provided from their respective CPU mirror fields, as illustrated in Figure 4.4.

For all the eligible Uintah based GPU fields, a field location on the CPU, owned and managed by Uintah already exists. Along with that, an additional field location on the GPU is managed as a scratch field. Uintah is not aware of the additional field location that has been added for heterogeneous task graphs. The other scratch fields have their memory allocated and managed with the information provided by the ExprLib's Hybrid Scheduler. As the task is marked as a CPU only task for Uintah, neither a stream nor a device context is assigned by Uintah for execution. Hence, a GPU load balancer is designed for handling the process of selecting and assigning GPU device contexts for a heterogeneous CPU-GPU task graph. More details on the GPU-load balancer are explained in §4.2.2.1

4.2.2.1 Device Index Selection and Assignment

Uintah cannot process a heterogeneous CPU-GPU task graph and this is a reason why a device index is not assigned. The GPU-load balancer is designed that assigns a device context per task in a round-robin fashion. The GPU load balancer does static assigning of device context per task unlike the Uintah's device assignment, which is dynamic. Assigning a particular device context has to be made at the earlier stages of execution. Information about the device context is informed to the hybrid scheduler, which is responsible for creating `cudaStreams` for expressions and also assigning proper hardware targets for the GPU nodes. This process unifies and ensure that the memory management and wrapping of Uintah's fields and scratch fields have the same device context.

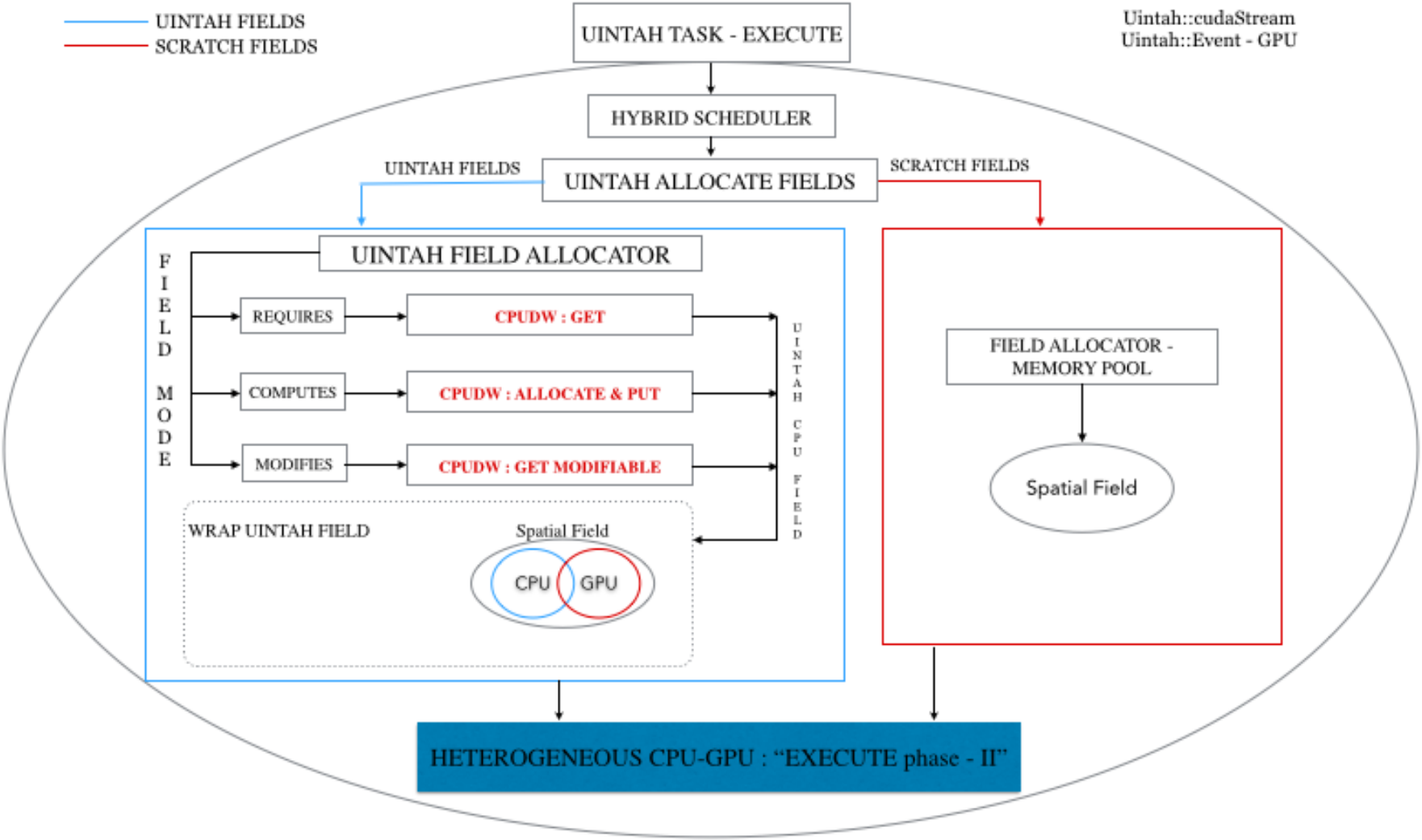

Figure 4.4: Setup phase of a heterogeneous CPU-GPU task graph. The schematics shows the control flow during a setup phase for a heterogeneous CPU-GPU task graph.

4.3 Wasatch GPU Task Execute

4.3.1 Homogeneous GPU Task Execute

When a task gets to a phase of the execution, a `cudaStream` with its device context is made available by the scheduler. A `cudaStream` keeps a track of the progress and reports the status to Unified Scheduler of the Uintah, which manages the task scheduling. The execution work flow is the same as explained in §3.4.3.1.

In reference to Figure 3.6, the nodes A, D, E, and F are managed by Uintah and hence, their memory is not deallocated until to the end of the simulation, whereas B and C, which act as scratch fields, manage their memory from the memory pool. It is very important to note that the `cudaStream` supplied by Uintah should not report task completion to the Unified Scheduler before a task is actually completed. As already stated in §3.4.3.1, the resultant node A performs calculations asynchronously with respect to the host; hence, a synchronization is performed so that the host thread is made to wait till all the work assigned to the stream is complete. This way the Uintah's stream does not prematurely report the completion status of a task to the scheduler. Any changes to the device index by the Uintah during a task execution phase is handled as explained in §4.2.1.

4.3.2 Heterogeneous CPU-GPU Task Execute

A heterogeneous CPU-GPU task graph is viewed by the Uintah as the CPU only task, as explained in §4.2.2. A static device index is assigned by the GPU-load balancer for the heterogeneous CPU-GPU tasks. The execution work flow is the same as explained in §3.4.3.2. The illustration in Figure 4.5 shows the nodes A, D, E, and F are managed by Uintah, whereas B and C are managed as as scratch fields. The Uintah's fields A and E, tagged as GPU runnable, are required to have their memory on an appropriate device for performing calculations on GPU. This is the reason an additional field location on the GPU is added. This GPU field location is allocated as a scratch field and it is indicated by a circular marker with a square enclosure in Figure 4.5. It is critical that the original Uintah based CPU fields are informed regarding any changes because of calculations done on their GPU counterparts. As calculations are performed on the expressions A and E, its associated GPU fields are updated with the recent field values as a result from the calculations. As the field associated with a variable is updated, a validation of the actual Uintah CPU fields are preformed, which invokes a device to host asynchronous data transfer from its valid source GPU fields. This process ensures that all the Uintah based CPU fields valid at each time step. A sanity check is performed to ensure that all the device-side work is completed before releasing the host thread.

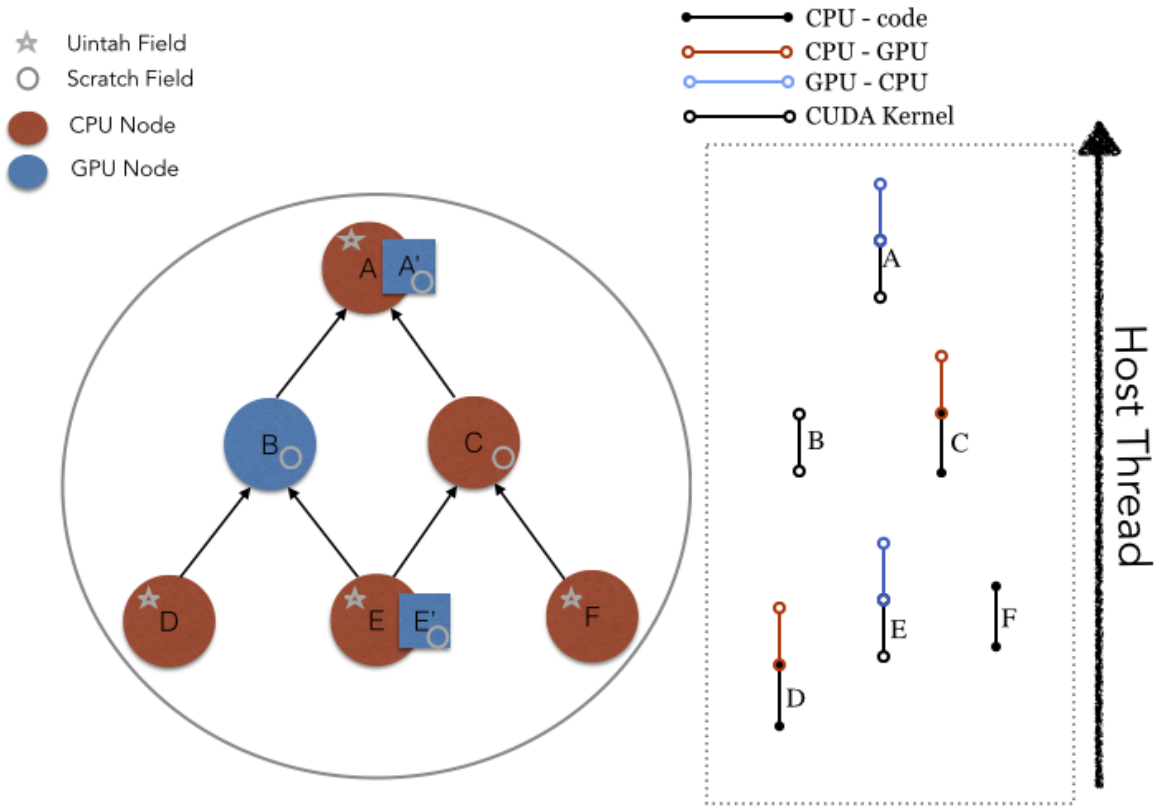

Figure 4.5: Execution phase of a heterogeneous CPU-GPU task graph. The schematics shows the control flow during the task execution process.

CHAPTER 5

RESULTS AND SUMMARY

5.1 Overview

All the test cases were chosen to demonstrate the abilities of the projects to scale on trending and emerging architectures with providing the support that has not been explored before. Earlier, the capability of running GPU was limited to the test cases in the Expression Library (ExprLib) and with the recent developments in Uintah and its component Wasatch, supporting of fields on multiple hardware targets has led to exploration of the GPU capabilities. Wasatch is capable of supporting both the homogeneous GPU as well as heterogeneous CPU-GPU tasks, and the performance constraints are reported in the further sections.

The GPU add-on extension for the Wasatch component to support multiple hardware targets and also to provide the flexibility in scheduling and executing any nature of task graph has been demostrated. Unless specified, all the tests were run on a 12-core Intel Xeon E5-2620 (2x6 cores at 2.00 GHz) with 16 GB RAM and a NVidia GeForce GTX 680.

5.2 Test Case—ExprLib

We set up several tests of different computational intensity in ExprLib that evaluates diffusion and source term expressions to compute solution variables. The source terms involved in these tests are similar to the calculations performed in a detailed chemical kinetics simulation. Each of these tests evaluate 30 partial differential equations (PDEs) arranged in the form of a task graph for 100 iterations. Each of the 30 PDEs has an expression that computes the diffusive flux and the source term. A right side expression performs the divergence of the fluxes and adds all the source terms if applied.

The mathematical expression governing these tests is as follows:

$$\frac{\partial \phi_i}{\partial t} = \frac{\partial}{\partial x}\Big(- \Gamma_i \frac{\partial \phi_i}{\partial x}\Big) + \frac{\partial}{\partial y}\Big(- \Gamma_i \frac{\partial \phi_i}{\partial y}\Big) + \frac{\partial}{\partial z}\Big(- \Gamma_i \frac{\partial \phi_i}{\partial z}\Big) + s_i \tag{5.1}$$

$$s_i = \sum_{j=1}^{n} exp(\phi_j) \tag{5.2}$$

$$s_i = \sum_{i=1}^{n} exp(\phi_i) \tag{5.3}$$

where s_i are source terms. For example, if the source terms are expected to operate as coupled as shown in (5.2), there will be all-to-all coupling of terms, and the independent source terms as shown in (5.3) have one-to-one coupling of terms.

The coupled-source terms are used to signify the importance of complex dependencies to be attained before calculating an expression. For 20 PDEs, each source term has to read in the inputs from all the 20 input fields every time. In this process, each term has to wait until all the 20 input fields are initialized and available for computation. Therefore, the calculation of solution variables for the PDEs are interdependent. For the calculations involving independent source terms, each source term expression has a one-to-one coupling and independent with other source term expressions. Hence, the calculation of one PDE does not have any effect on the calculation of another.

The first test case calculates the divergence of diffusive flux without any source terms along with taking divergence of fluxes as indicated mathematically in (5.4).

$$\frac{\partial \phi_i}{\partial t} = \frac{\partial}{\partial x}\Big(- \Gamma_i \frac{\partial \phi_i}{\partial x}\Big) + \frac{\partial}{\partial y}\Big(- \Gamma_i \frac{\partial \phi_i}{\partial y}\Big) + \frac{\partial}{\partial z}\Big(- \Gamma_i \frac{\partial \phi_i}{\partial z}\Big) \tag{5.4}$$

The second test case focuses on calculating diffusive flux with the independent source term along with the divergence of fluxes. Equation (5.1) is the governing equation and (5.3) is the source term that governs this test case. The third test case is regarding the calculation of diffusive flux along with coupled source terms. Equation (5.1) along with (5.2) governs the third test. In Figure 5.1, a prototype for the third test that involves 2 PDEs that solves diffusive flux along with coupled source terms is illustrated. Diffusive fluxes along with evaluating source terms in separate expressions are performed. The right hand side term does the task of calculating the divergence of fluxes and adding the source term. These tests are in order of increasing computational intensity.

Figure 5.2 shows the speedup of multithreaded performance over single-threaded performance for all three test cases with problem size varying from 64^3 and 128^3. It can be inferred from Figure 5.2 that coupled source with diffusion test scales well when compared to other test cases. This states that the performance of single threaded to that of multithreaded is good for the coupled-source test, which is because of computationally intensive calculations and interdependencies among the fields while coupling the source terms. It can be analyzed that the calculation of diffusive flux is common in all the three test variants. The only difference among the tests that can contribute and cause a difference is the way the source terms are being computed.

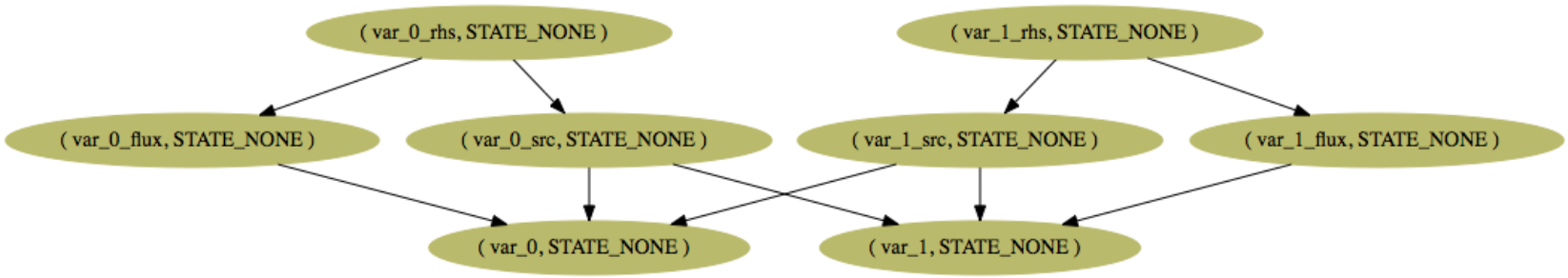

Figure 5.1: ExprLib-scalability test with diffusive flux and coupled source terms. Note that all the nodes in this test case are GPU runnable and 2 PDEs are solved.

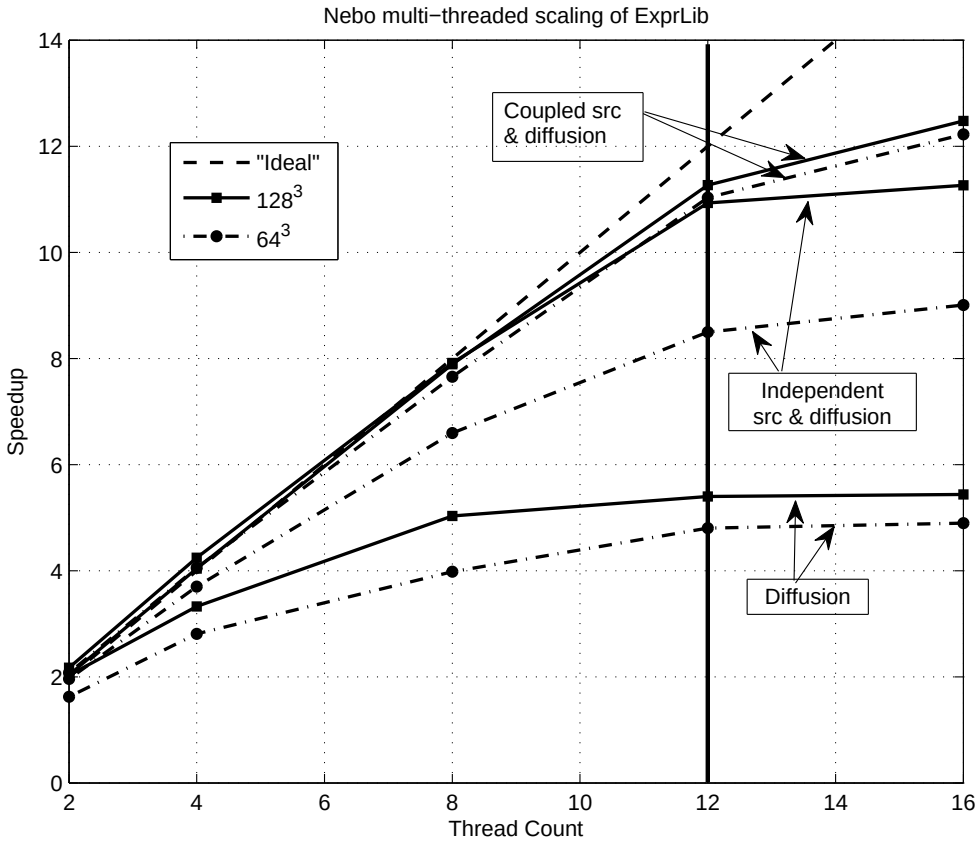

Figure 5.2: ExprLib-scalability test showing multicore performance. Thread counts are varied from 2–16 in the x-axis and speedup on y-axis. The problem size under study is 16^3 and 128^3.

As with the multithread backend tests, the diffusion with a coupled source term test case scales the best, and the diffusion-only test scales the worst. The general trend of these tests is that more computationally dense calculations (coupled source terms with diffusion) perform better than less computationally dense (diffusion only) calculations. The reason for this trend is that memory latency is being hidden while performing the intensive calculations and more pronounced in case of less dense calculations. Finally, it is interesting to note that Nebo's multithread backend with 16 threads improves over 12 threads for most tests on a system with 12 cores. This limited improvement comes from hyperthreading.

It is very evident that the diffusive flux calculation alone without the source terms does not scale well after a certain thread count. This is because there are more thread resources available for computation than available work. It is also observed that after 8 threads the performance tends to get stable. There is also a gradual decline in the performance because of launching more threads from the pool that is causing overhead for creating or pulling active worker threads out of a pool. For the independent source term and diffusion calculations, the performance curves are in between to the coupled source terms test and diffusion only test. This is because the computational intensity involved is intermediate when compared with the other tests. It is seen that any resources that are greater than 12 threads have a negative effect on the performance. The scaling of the tests degrade because of the same reason for not having enough work to the available computational resources. The diffusion-only test does not scale well for thread count more than 8, and the diffusive flux with independent source terms test case does not scale well for threads greater than 12. The most computationally intensive of all is the diffusive flux with the coupled-source terms. The coupled source terms calculation is performed over all the coupled dependencies and hence, there is also a significant memory caching overhead expected for the coupled-source terms. The diffusion with independent source term test does not scale as well, especially for the smaller problem size; moreover, the diffusion-only test does not scale well, particularly beyond 4 cores.

Likewise, Figure 5.3 shows speedup of many-core (GPU) performance over single-thread performance for the same tests and varied problem sizes. These tests are performed with synchronous task execution and does not involve any data-transfers. All the tests can be considered as homogeneous tasks. Each expression has Nebo expressions that are capable of performing computations on GPU. Figure 5.3 shows that at almost every problem size, the many-core GPU execution performance is more than 10x faster than the single thread execution, with the fastest being just over 140x at an increased thread count.

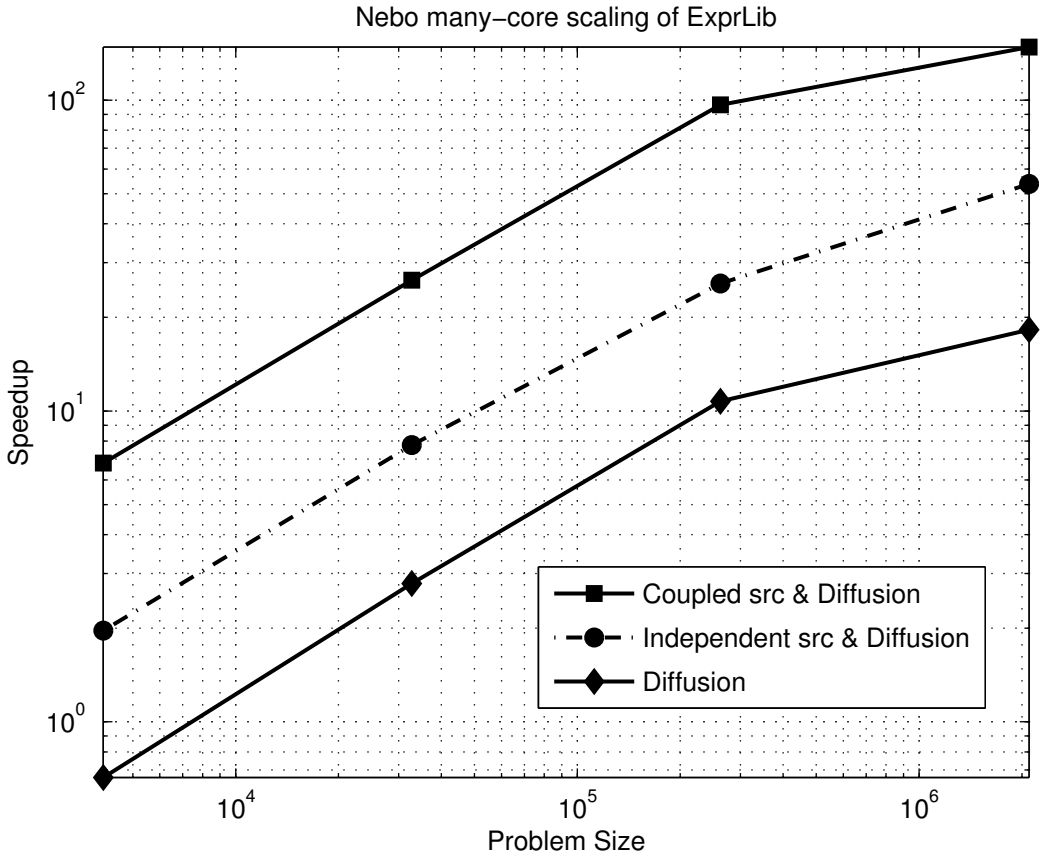

Figure 5.3: ExprLib-scalability test showing GPU performance. The problem size is varied from 16^3 to 128^3 in the x-axis and speedup on y-axis. Speedup comparison is for GPU execution over serial performance.

5.3 Test Case—Wasatch

The test case under consideration is very similar to the one used in §5.2. The diffusive flux and coupled source terms test is used to perform scaling analysis for its role in doing some intensive calculations for testing Wasatch framework. This test is carried out on a framework that is preliminarily in the stages of Wasatch development for supporting on GPU and hence, there is still a scope for optimization.

Figure 5.4 shows a prototype for a test case that involves calculation of diffusive flus in all three directions, x, y, and z, with source terms along with time stepping integration. Figure 5.4 shows the prototype equation with a single PDE and the test case under study involves 20 similar PDEs arranged in the form of a task graph for 50 time steps. Each of the 20 PDEs has an expression that computes the diffusive flux and source terms, which form the right hand side of the equation and stores the result in a solution variable. After computing the diffusive flux and source terms, divergence on the diffusive flux is calculated and the source terms are added to the result. Once the right hand side of the expression is computed, a time stepping procedure is invoked to advance the solution variable for a time dependent PDE. The time stepping procedure is the only additional task that is performed in the Wasatch, which makes these test cases differ from ExprLib test cases.

The complexity involved with the time stepping scheme are the time stepping variables - `time`, `dt`, `rkstage`, and `timestep` indicated in the graph within a node colored in grey. These nodes are tagged only to execute on the CPU. Time stepping scheme is implemented in Uintah and the underlying type of the time stepping variables are not yet supported for GPU execution. These variables forces the graph and makes it a heterogeneous CPU-GPU task graph. There is a particular nomenclature that indicates whether a node is GPU runnable or CPU runnable, as shown in Figure 5.4. The nodes colored in green and blue diamond shapes are GPU runnable, and the nodes with a grey color are only CPU runnable. This makes the graph to be heterogeneous in nature for execution on multiple hardware targets and forces the data transfer to be performed on all the Uintah managed variables to be synchronized with their mirror GPU fields at each time step.

As seen from Figure 5.4, all the three components x, y, and z of the diffusive fluxes are computed with a source term that gets added along with the diffusive fluxes. A divergence operation on the diffusive flux is performed in a seperate task and the source terms gets added to the resultant expression of divergence. Once the right hand side of the expression is computed, time stepping scheme is invoked with the help of time stepping variables as discussed above. As there are several variables that can only be computed on the CPU, the

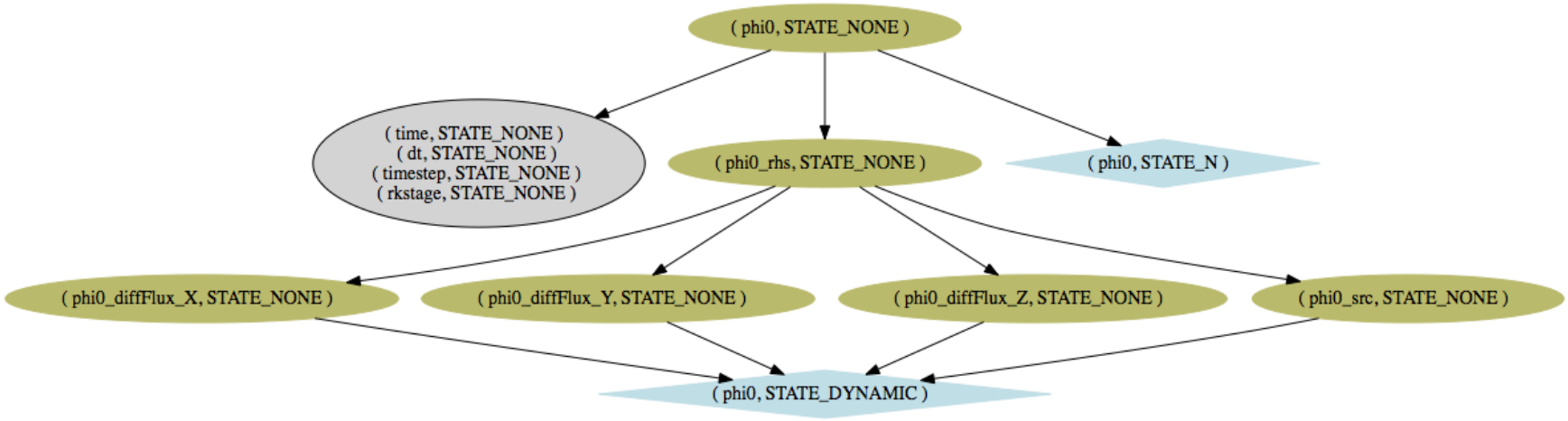

Figure 5.4: Task graph used for scalability test in Wastch. The test case shows a single PDE with diffusive flux and coupled source terms. The nodes colored in green oval and blue diamond shapes are GPU runnable, and the node in grey is only CPU runnable.

nature of this task graph is inherently tagged for heterogeneous CPU-GPU task execution.

Figure 5.5 shows the total computation time for a test case in Wasatch for three different modes of execution. The first mode is about a task graph executed for a completely serial run on the CPU with a single core. The second mode is about scheduling and execution of a task graph in the form of a homogeneous GPU version, where all the nodes compute on GPU with the support from Uintah, as explained in §4.3.1. The last mode is a heterogeneous CPU-GPU task graph, that involves a different execution pattern, as explained in §4.3.2. As seen in Figure 5.5, the total computation time is more for the serial CPU version than its counterpart's homogeneous and heterogeneous GPU execution. When comparing the performance between the homogeneous and heterogeneous test runs, the homogeneous performs better than the heterogeneous task execution. This is because of the explicit data transfers in the form of synchronization of the CPU field location for all the Uintah-managed variables in a heterogeneous task graphs that slows the performance.

Figure 5.6 is about the speedup comparisons for the homogeneous GPU and heterogeneous GPU task execution over a serial CPU version. Around 42X speedup is observed for the homogeneous task graph and about 22X for heterogeneous task graph at a problem size of $128^3(128, 128, 128)$. Heterogeneous tasks perform explicit data transfer, and the field size is considered to be huge at this problem size of 128^3, and along with 20 PDEs over 50 iterations makes the performance drop to about 22X. All the tests are performed at an increasing computational intensity with a problem size varying from $16^3(16, 16, 16)$ to $128^3(128, 128, 128)$ with 20 PDEs for 50 iterations.

5.3.1 Insight—Level of Concurrency

Level of concurrency is an important issue to be considered while performing asynchronous execution of a task graph. In a heterogeneous architecture environment, level of concurrency is a parameter that defines the parallelization efficiency. We focus on elaborating the use of `cudaStream` in exploiting concurrency on GPU. CUDA resource handles are created for each expression to enable concurrent execution as discussed in §3.3.2.

A test case is carried out to check the level of concurrency that is expected out of a synchronous task execution and asynchronous task execution for a problem size solving for 20 PDEs and a single PDE. Synchronous mode is created by enforcing a single `cudaStream` to the entire tree, that performs computations and data transfer in an orderly fashion they are queued into the stream with respect to host thread. In case of an asynchronous mode, separate stream handles provide concurrent computations and data transfer for the independent tasks.

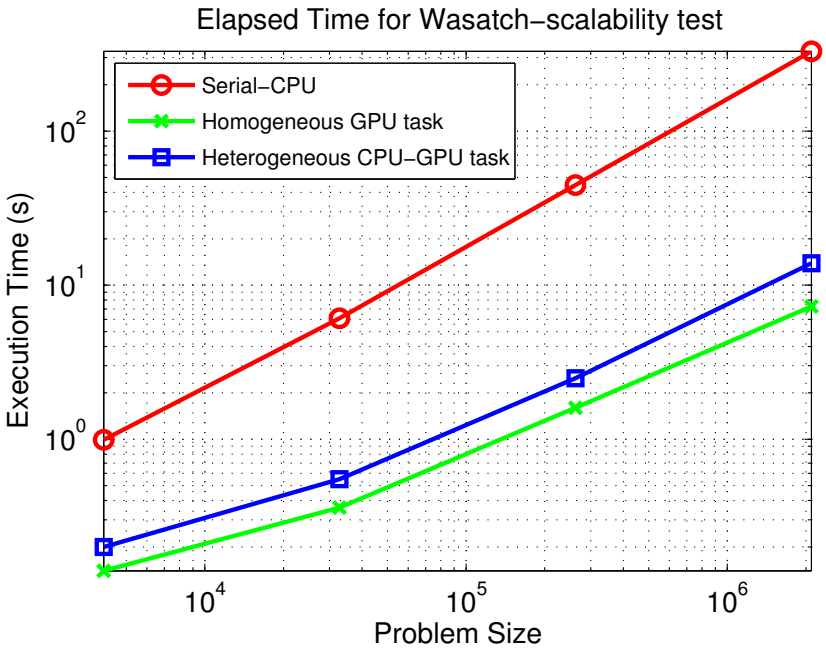

Figure 5.5: Computation time for the scalability test in Wasatch. The problem size is varied from 16^3 to 128^3 for serial, homogeneous, and heterogeneous task graph

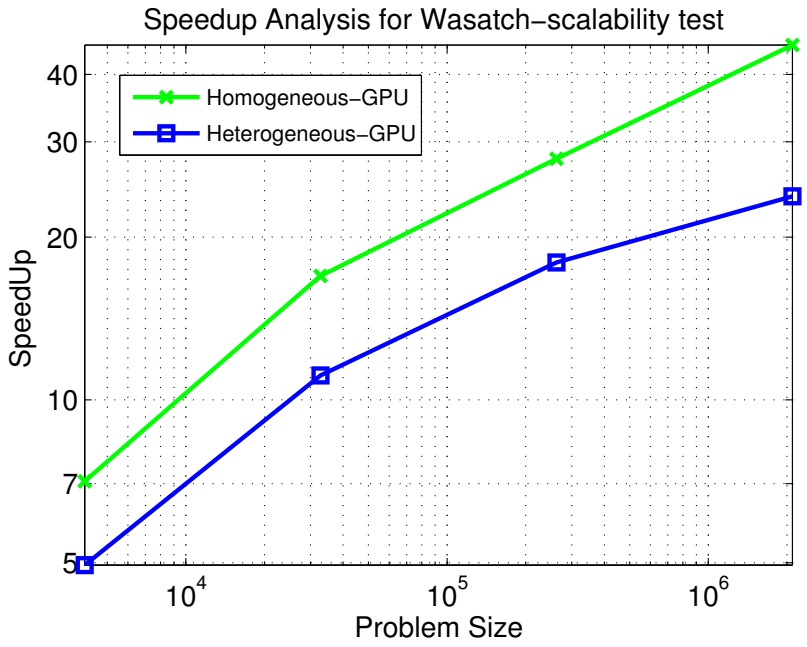

Figure 5.6: Speedup analysis for the scalability test in Wasatch. The problem size is varied from 16^3 to 128^3 for homogeneous GPU and the heterogeneous CPU-GPU task graph

As shown in Figure 5.7, the total computation time for solving a single PDE in shown for asynchronous and synchronous mode of execution. The reason for not plotting data seperately in solving a single PDE in an asynchronous and sychronous mode is because the task graph is not dense enough to show a significant difference. The test cases involve, calculating diffusive flux with coupled source terms along with divergence of fluxes that evaluates over 20 PDEs and 1 PDE for 10 iterations over a problem size varying from 16^3 to 128^3. Data is also shown for solving 20 PDEs in asynchronous and synchronous modes in the same figure, and there is a noticeable difference in the computation time. In terms of speedup, the asynchronous version is faster up to 8.5X at a problem size of 16^3 and about 2.75X faster at a problem size of 128^3 when compared to the synchronous version, as shown in Figure 5.8.

Figure 5.8 shows the speedup comparison of the asynchronous mode over the synchronous mode for 20 PDEs. There is a gradual decrease in the speedup with the increase in problem size, and this is because of the computational intensity. At a lower problem size of 16^3, the speedup is about 8.5X and at 128^3 grid size, the speedup is just 2.5X. The field sizes at 128^3 involve a heavy kernel computation, data transfer, and some necessary synchronizations done to avoid any race conditions performed for every expression.

This shows that the behavior of the asynchronous task execution tends to gradually move towards synchronous task execution because of the intense work involved in it. It is still commendable that a performance speedup of 2.5X is observed at a problem size of 128^3 for 20 PDEs.

5.4 Future Work

- Nebo supports implementation of boundary conditions for GPU and modifications are certainly necessary for graph algorithms to get the GPU execution support. Currently, all the extra work (modifiers, boundary conditions) that is assigned to the expression has been turned to CPU-only, and this has to be addressed for GPU execution.
- Edge and node weights are an important property to be considered while examining a expression tree. This can provide insight in understanding critical parameters for assigning a node hardware target for a heterogeneous CPU-GPU task graph.
- GPU execution with host multithreaded support is necessary for performing concurrent task execution on hybrid computing architectures. This goal is to achieve execution of the tasks that can be concurrently executed on multi-GPUs and also use Nebo multithreaded backend for CPU expressions.

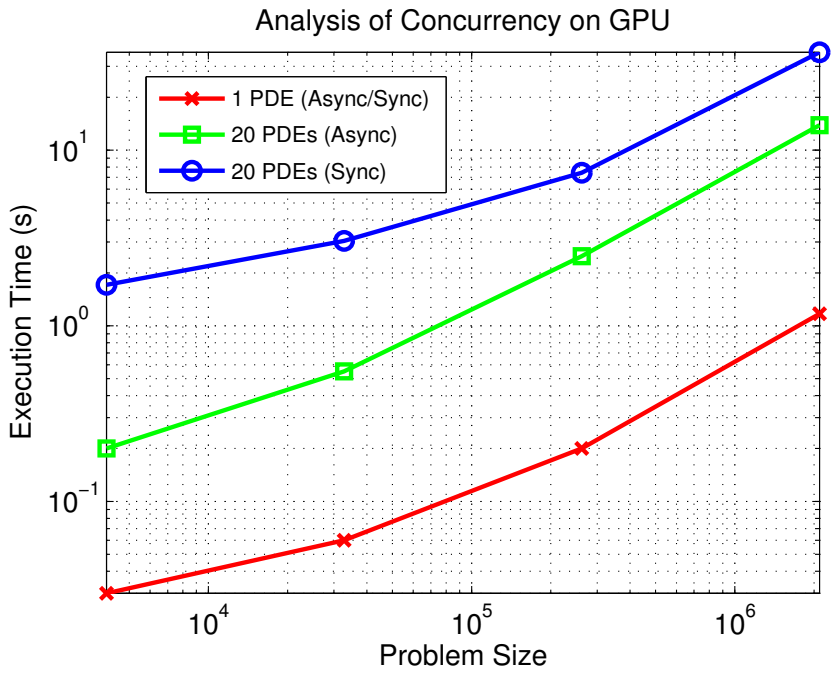

Figure 5.7: Computation time for different modes of execution. The Wasatch scalability test with problem size varying from 16^3 to 128^3 is analyzed for the total execution time for asynchronous mode and synchronous mode task execution. Please note that 20 PDEs and 1 PDE are used for this study.

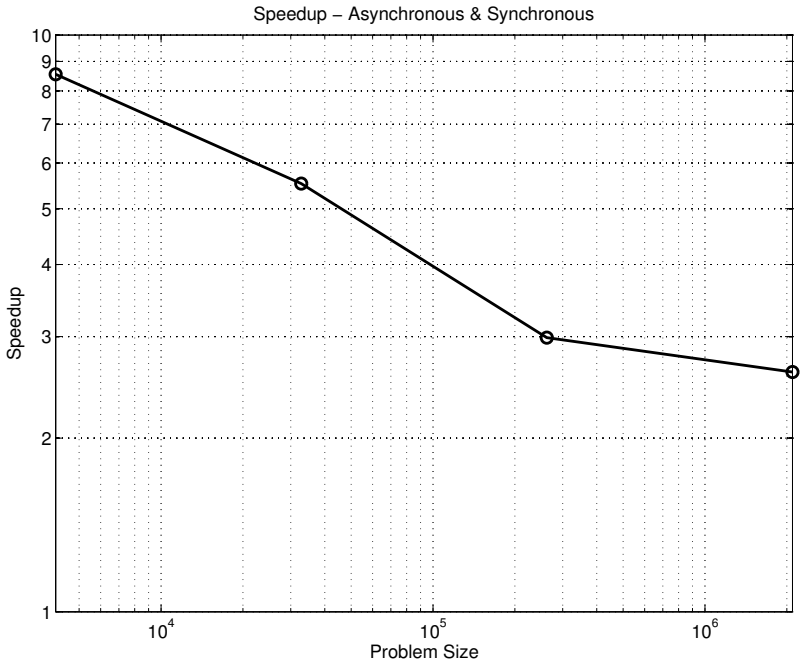

Figure 5.8: Speedup analysis for different modes of execution. The Wasatch scalability test with problem size varying from 16^3 to 128^3 is tested for the performance of asynchronous mode over synchronous mode task execution.

- CUDA 6.0 provides support for Unified Memory Addressing (UVA), which can be adopted to ease the memory management heuristics and improving the code readability. With the use of UVA, the same pointer can be used for accessing the memory both on host and device that removes the complexities that arise with separating memory management procedures and drastically reduces code.

5.5 Summary

Addressing issues on task graph scheduling and execution on heterogeneous and emerging architectures are important for better performance and improved accuracy. This work majorly focus on providing support for directed acyclic task graph based approach for solving partial differential equations to use heterogeneous computing systems augmented with GPUs by exploiting much of the task-based parallelism. This is an initial effort towards achieving the goal of extreme scale computing for computational science and engineering applications.

- The primary objective of this work is to addresses the infrastructure abilities for carrying out Wasatch tasks on diverse hardware targets in conjunction with the Uintah Computational Framework. More details can be found in §4.2
- With the support of task graph execution on heterogeneous computing systems, various other projects like Lattice Based Multiscale Simulation (LBMS), One-Dimensional Turbulence (ODT), etc., can use the opportunity to explore the new areas of high performance computing.
- Robust memory management heuristics are designed for better organizing memory on CPU and GPU. More details regarding this can be found in §3.4.2
- The concept on consumer field which restricts permissions on write access is deprecated as elaborated in §2.4.
- There has been major advancements with respect to graph algorithms to efficiently schedule tasks when using multiple hardware targets. A synchronous task execution model is changed to an asynchronous task execution model that was enabled to explore task-based parallelism on GPUs with the help of `cudaStream`. More details can be found in §3.4

REFERENCES

[1] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. London: Academic Press, 2000.

[2] R. M. D'Souza, M. Lysenko, S. Marino, and D. Kirschner, "Data-parallel algorithms for agent-based model simulation of tuberculosis on graphics processing units," in *Proceedings of the 2009 Spring Simulation Multiconference*, ser. SpringSim '09. San Diego, CA, USA: Society for Computer Simulation International, 2009, pp. 21:1–21:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1639809.1639831

[3] V. Springel, S. D. M. White, A. Jenkins, C. S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, J. A. Peacock, S. Cole, P. Thomas, H. Couchman, A. Evrard, J. Colberg, and F. Pearce, "Simulations of the formation, evolution and clustering of galaxies and quasars," *Nature*, vol. 435, no. 7042, pp. 629–636, 2005. [Online]. Available: http://dx.doi.org/10.1038/nature03597

[4] J. Yang, Y. Wang, and Y. Chen, "Gpu accelerated molecular dynamics simulation of thermal conductivities," *Journal of Computational Physics*, vol. 221, no. 2, pp. 799–804, 2007. [Online]. Available: http://dx.doi.org/10.1016/j.jcp.2006.06.039

[5] J. Guilkey, T. Harman, and B. Banerjee, "An eulerian-lagrangian approach for simulating explosions of energetic devices," *Computers & Structures*, vol. 85, no. 11-14, pp. 660–674, 2007. [Online]. Available: http://dx.doi.org/10.1016/j.compstruc.2007.01.031

[6] F. P. Miller, A. F. Vandome, and J. McBrewster, *Moore's Law: History of Computing Hardware, Integrated Circuit, Accelerating Change, Amdahl's Law, Metcalfe's Law, Mark Kryder, Jakob Nielsen (Usability Consultant), Wirth's Law*. Eastbourne: Alpha Press, 2009.

[7] (2014) Nvidia developer zone. [Online]. Available: http://http://docs.nvidia.com/cuda/cuda-runtime-api/modules.htmlmodules

[8] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, "Graphics processing unit (gpu) programming strategies and trends in gpu computing," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 4–13, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2012.04.003

[9] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in heterogeneous computing," *Scientific Programming*, vol. 18, no. 1, pp. 1–33, 2010. [Online]. Available: http://dx.doi.org/10.1155/2010/540159

[10] P. K. Notz, R. P. Pawlowski, and J. C. Sutherland, "Graph-based software design for managing complexity and enabling concurrency in multiphysics pde software,"

ACM Trans. Math. Softw., vol. 39, no. 1, pp. 1–21, 2012. [Online]. Available: http://dx.doi.org/10.1145/2382585.2382586

[11] M. Berzins, Q. Meng, J. Schmidt, and J. C. Sutherland, *DAG-Based Software Frameworks for PDEs*, ser. Euro-Par 2011: Parallel Processing Workshops. Springer Science + Business Media, 2012, pp. 324–333.

[12] S. G. Parker, "A component-based architecture for parallel multi-physics pde simulation," *Future Generation Computer Systems*, vol. 22, no. 1-2, pp. 204–216, 2006. [Online]. Available: http://dx.doi.org/10.1016/j.future.2005.04.001

[13] M. Berzins, J. Luitjens, Q. Meng, T. Harman, C. A. Wight, and J. R. Peterson, "Uintah: A scalable framework for hazard analysis," in *Proceedings of the 2010 TeraGrid Conference*, ser. TG '10. New York, NY, USA: ACM, 2010, pp. 3:1–3:8. [Online]. Available: http://doi.acm.org/10.1145/1838574.1838577

[14] Q. Meng, A. Humprey, and M. Berzins, "The uintah framework: a unified heterogeneous task scheduling and runtime system," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 11 2012, pp. 2441–2448. [Online]. Available: http://dx.doi.org/10.1109/SCC.2012.6674233

[15] C. Earl, "Introspective pushdown analysis and nebo," Ph.D. dissertation, Univ. of Utah, Salt Lake City, 2014.

[16] E. B. Fernandez and B. Bussell, "Bounds on the number of processors and time for multiprocessor optimal schedules," *IEEE Trans. Comput.*, vol. C-22, no. 8, pp. 745–751, 1973. [Online]. Available: http://dx.doi.org/10.1109/TC.1973.5009153

[17] O. Sinnen, *Task Scheduling for Parallel Systems*. New Jersey: Wiley-Interscience, 2007.

[18] *Cilk Reference Manual*, Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science, 1998, release 5.4.3. [Online]. Available: http://supertech.lcs.mit.edu/cilk/manual-5.4.3.pdf

[19] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *ACM SIGPLAN Notices*, vol. 33, no. 5, pp. 212–223, 1998. [Online]. Available: http://dx.doi.org/10.1145/277652.277725

[20] K. Randall, "Cilk: Efficient multithreaded computing," Ph.D. dissertation, Massachusetts Institute of Technology, 1998.