Costa Rica Institute of Technology

Transmogrifying Performance Analysis: Data Analytics on GPU Application Codes

Diego Jiménez Vargas

Advisor:
Esteban Meneses, PhD

A thesis submitted in partial fulfillment
of the requirements for the degree of

Magister Scientiae in Computer Science

School of Computing

August 2022

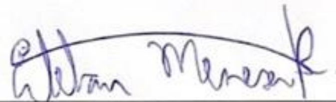
Instituto Tecnológico de Costa Rica
Escuela de Computación
Unidad de Posgrado

ACTA DE APROBACION DE TESIS

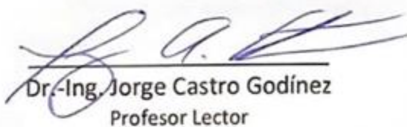
Transmogrifying Performance Analysis: Data Analytics on GPU Application Codes

Por: Diego Jiménez Vargas

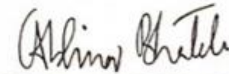
TRIBUNAL EXAMINADOR



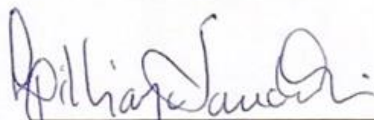
Dr. Esteban Meneses Rojas
Profesor Asesor



Dr.-Ing. Jorge Castro Godínez
Profesor Lector



Dr. Abhinav Bathele
Lector Externo



Dra.-Ing. Lilliana Sancho Chavarría
Presidente, Tribunal Evaluador Tesis
Programa Maestría en Computación

31 de Agosto, 2022

transmogrify verb

trans·mog·ri·fy

:to change somebody/something completely, especially in a surprising way

Oxford Learner’s Dictionaries

Abstract

High Performance Computing (HPC) is now reaching exascale capabilities. Modern supercomputers are catalyzing scientific research and have become central tools in topics like big data analysis and machine/deep learning. However, the road to extreme-scale computing is not without its challenges. Energy efficiency as well as power and cooling are some of the hardware concerns in this quest. On the other hand aspects like application scaling, the cost of scientific code development including new programming models and portability issues are some examples of challenges in the software spectrum.

This project is focused on one particular challenge that is also crucial in achieving next-generation compute capabilities: application performance analysis and optimization. Many of the leading HPC systems are powered by heterogeneous compute nodes, which integrate Graphic Processing Units (GPUs) as hardware accelerators. Adapting modern applications to leverage such systems effectively is of great importance. The performance evaluation process is key in enabling algorithms to scale on these modern massively parallel clusters. Although modern tools allow for the analysis of parallel applications, they usually limit the user to proprietary data formats and data visualization interfaces, effectively restricting the kinds of analysis that can be done.

In this project, we implemented a data transformation and manipulation workflow that enables the creation of context-aware hierarchical performance data for GPU applications profiled with NVIDIA's NSight Tools. This information can then be loaded into a tool like Hatchet, a Python-based library, to enable programmatic performance analysis. Through a series of case studies, we showcase how this newly implemented workflow in hand with a data analytics approach can help users identify bottlenecks and implement custom and reproducible analysis of GPU-accelerated performance data.

Acknowledgments

I want to acknowledge Prof. Esteban Meneses, for his constant guidance, not only on this project but through my academic life on the last 7 years. His input has always been valuable and appreciated.

Special thanks to Prof. Abhinav Bhatele and his research group who have constantly help this project move in the right direction.

This research is partially supported by a machine allocation on Kabré supercomputer at the Costa Rica National High Technology Center (CeNAT). It is also partially supported by a machine allocation on Raven HPC System at the Max Planck Computing and Data Facility (MPCDF).

To Cristi, my wife, thanks for the support and patience.
To the memory of my beloved grandfather, Tata.

Contents

Contents 7

List of Figures 10

List of Tables 12

1 Introduction 17

1.1 Contributions . . . 18
1.2 Problem Definition . . . 19
1.3 Justification . . . 20
1.4 Objectives . . . 21
1.4.1 Overall Objective . . . 21
1.4.2 Specific Objectives . . . 22

2 Background 23

2.1 Performance Measurement and Analysis . . . 23
2.1.1 Performance Monitoring . . . 24
2.1.2 Performance Analysis . . . 25
2.2 Parallel Computing with GPUs . . . 27
2.2.1 GPU Hardware Model . . . 27
2.2.2 GPUs in High-Performance Clusters . . . 30
2.2.3 GPU Programming Models . . . 31
2.2.4 NVIDIA NSight Performance Analysis Tools . . . 39

2.3 Hatchet: Data Analysis of Performance Data 42
2.4 GPU Accelerated Applications 45
2.4.1 Tensorflow Keras Model 46
2.4.2 BS-SOLCTRA Plasma Physics Simulator 46
2.4.3 LULESH Shock Hydrodynamics Simulator 47

3 From NVIDIA NSight Tools to Hatchet: Transmogrifying Trace Data and GPU Metrics for Programmatic Analysis 48
3.1 Generating Performance Data with NVIDIA NSight Tools 48
3.1.1 NVIDIA NSight Systems Time-Based Data 49
3.1.2 NVIDIA NSight Compute Kernel Specific GPU Metrics 52
3.2 Annotating GPU-Accelerated Applications 54
3.2.1 Tensorflow Keras Model: GPU-Accelerated Framework considerations . . 55
3.2.2 BS-SOLCTRA: OpenMP considerations 55
3.2.3 LULESH: CUDA considerations 57
3.3 Calling Context Trees Construction 59
3.3.1 NVTX Trace Processing 59
3.3.2 NSight Compute GPU Metrics Processing 61
3.3.3 Multi-GPU Executions 62
3.3.4 Hatchet Compliant Data Format 63

4 Results 65
4.1 Experimental Setup 65
4.2 Loading Reconstructed CCTs with Hatchet 66
4.3 Identifying Bottlenecks with Hatchet: Tensorflow Keras Model Case Study 68
4.4 Data Analytics on GPU Performance Metrics: BS-SOLCTRA Case Study 70
4.4.1 Descriptive Implementation Performance Overhead 70
4.4.2 Comparing Implementations through GPU Metrics 71
4.4.3 Multi-GPU Performance Analysis 74
4.5 Roofline Model in Hatchet: LULESH Case Study 76

5 Conclusions 82
5.1 Summary 82
5.2 Contributions 83
5.3 Limitations 83

5.4 Future Work 84

Bibliography **85**

List of Figures

1.1 NVIDIA's NSight Systems performance analysis GUI showing results of a profiling analysis for Lulesh-GPU 20

2.1 Simplified block diagram of a traditional NVIDIA GPU hardware organization. 28
2.2 Simplified block diagram of the main components of an NVIDIA Streaming Multiprocessor 29
2.3 Heterogeneous computing system model 30
2.4 CUDA CPU-GPU kernel offloading model 32
2.5 CUDA parallel kernel execution model 34
2.6 Threading behavior in device when using target teams directive 37
2.7 Threading behavior in device when nesting a parallel construct in a target teams directive 38
2.8 NSight Systems timeline trace view for a GPU-accelerated application. 40
2.9 Example SpeedOfLight Section of Nsight Compute GUI report 41
2.10 Example of Hatchet's GraphFrame data structure using a code-segment of LULESH . 43
2.11 MultiIndexed DataFrame component of a GraphFrame in Hatchet 43
2.12 Hatchet enables on-node scaling analysis through the GraphFrame structure and its associated algebra operators. Example using LULESH 44

3.1 NSight Systems Trace GUI for a LULESH execution 49
3.2 Top-Down view of the profiled LULESH-GPU code 50
3.3 Summary statistics generated by NSight Systems for the LULESH-GPU application . . 51
3.4 Sample of trace data for an NVTX-annotated version of LULESH-GPU 52
3.5 Example of output segment from NSight Compute for one of LULESH's kernels . . . 52

3.6 Extract from an NSight Compute generated report 54
3.7 Subset of NVTX annotated code regions in LULESH-GPU 58
3.8 Call Path Tree construction from NVTX Trace process 60
3.9 NSight Compute metrics file processing and kernel metrics apportioning 62
3.10 Multiple CCTs are created when Multi-GPU trace data is provided 63

4.1 Resulting CCT for LULESH loaded into Hatchet 67
4.2 Execution profile comparison for the three different precision policies tested on the Tensorflow Keras model 69
4.3 Programmatic computation of Speedups in Hatchet 70
4.4 Execution profile comparison for both versions of BS-SOLCTRA 71
4.5 Execution profile comparison for both versions of BS-SOLCTRA after optimizing descriptive implementation 74
4.6 Execution profiles for each MPI rank in a multi-GPU BS-SOLCTRA execution 75
4.7 Roofline model main components and interpretation 77
4.8 Example Roofline chart from the NSight Compute GUI 78
4.9 Roofline plot for the CalcVolumeForceForElems_kernel in LULESH 80
4.10 Roofline comparison of our implementation through programmatic analysis and the chart generated by NSight Compute 81

List of Tables

2.1 CUDA Software - Hardware Mapping 35

3.1 Main Sections from the detailed set analysis on NSight Compute 53

4.1 Base software stack on both testbed systems 66
4.2 NSight Compute metrics needed to compute Roofline model for GPU kernels 79

List of Listings

1 Example of sequential element-wise vector addition 32
2 Example CUDA element-wise vector addition 33
3 Example OpenMP target element-wise vector addition 36
4 Example of NVTX Push-Pop Ranges annotations 42
5 GraphFrame substract operation in Hatchet 45
6 Example command line used with NSight Systems 49
7 Command line used to generate NVTX trace report from NSight Systems-generated SQLite database 51
8 Command line used to generate kernel-specific metrics with NSight Compute . . . 53
9 Tensorflow Keras Model annotated with NVTX extract 56
10 NVTX-annotated OpenMP implementation of the particle trajectory computation. The two approaches are added to exemplify the difference 57
11 Example JSON file created with our implemented transmogrifying tool 64
12 NSight performance data transformation process and JSON hierarchy file loading in Hatchet 66
13 Extract from the Tensorflow Keras model showing how we can control the precision policy 68
14 Computing the achieved speedup per function when moving from double-precision to single-precision 69
15 Compilation command used by both prescriptive and descriptive implementations of BS-SOLCTRA 70
16 Comparing GPU-kernel metrics for both BS-SOLCTRA implementations 72

17 OpenMP bind clause added to descriptive implementation to change kernel launch configuration . . . 73
18 Compilation command used by multi-GPU prescriptive implementation of BS-SOLCTRA . . . 74
19 Index dropping on multi-GPU DataFrame, computing both the mean and maximum execution time for each node across all processes . . . 75
20 Computing load imbalance across processes in a multi-GPU execution of BS-SOLCTRA . . . 76
21 Roofline generation process pseudo-code . . . 78
22 Roofline components computation with NSight Compute data loaded into Hatchet . 79

Acronyms

AI Arithmetic Intensity. 77–80

AI Artificial Intelligence. 45, 46, 55

BS-SOLCTRA Biot-Savart Solver for Computing and Tracing Magnetic Field Lines. 46, 55, 70, 74

CCT Calling Context Tree. 11, 61–68

CLI Command Line Interface. 49, 52, 53

CPU Central Processing Unit. 18, 27, 28, 30–32, 35, 37–39, 41, 45, 48, 52, 55, 65, 66

CUDA Compute Unified Device Architecture. 31–35, 37, 38, 49, 57, 58, 73, 84

FGPA Field Programmable Gate Array. 30

GPGPU General-Purpose GPU. 27, 31

GPU Graphics Processing Unit. 10, 11, 17–22, 27–32, 34–41, 45–49, 52–57, 59, 61–68, 70, 71, 74–76, 78, 80–84

GUI Graphical User Interface. 10, 19, 20, 49, 50, 53, 78, 80

HBM2 High Bandwidth Memory. 28, 65

HPC High Performance Computing. 17, 19–21, 24, 27, 30, 31, 35, 41, 45–47, 49, 59, 65, 66, 74, 77, 82, 84

ILP Instruction Level Parallelism. 29

LULESH Livermore Unstructured Lagrange Explicit Shock Hydrodynamics. 47, 50, 57–59, 63, 67, 80, 82

MPI Message Passing Interface. 31, 62, 74

NGC NVIDIA GPU Cloud. 55

NVTX NVIDIA Tools Extension Library. 41, 49, 51, 58, 62, 63, 65

PAPI Performance Application Programming Interface. 24

SIMT Single Instruction Multiple Thread. 29, 34

SM Streaming Multiprocessor. 29, 38, 73, 78

SP Streaming Processor. 29, 38

TLP Thread Level Parallelism. 29

Chapter 1

Introduction

Supercomputers are instrumental in the work of scientists and engineers dealing with complex and large scale problems. In the upcoming years, billion-dollar investments are expected, from governments around the world, in a set of new computer systems that are targeted at breaking the exascale computing barrier [1], i.e being able to execute 10^{18} floating-point operations per second. These spearheading systems will not only power the traditional scientific modeling and simulation workloads associated to High Performance Computing (HPC), but they also reflect the broadening scope of science projects that rely on trending topics like big data analysis and machine learning.

However, achieving extreme-scale computing has been a long time coming due to numerous and diverse challenges that have been identified in several studies [2, 3]. Putting aside the necessity for energy-efficient circuits, power and cooling systems, performant interconnects and memory systems, the software spectrum of HPC is also faced with crucial tasks. Four broad categories of software challenges have been identified [4] in relation to several aspects. These categories involve issues like: scaling and complexity of modern architectures, the cost of developing next generation scientific codes including new programming models, portability issues, improving tools needed throughout the software cycle, managing the increasing amount of data that must be handled and ensuring software sustainability.

This project is particularly focused on one salient challenge that must be resolved to attain exascale computing. All three computer systems announced by the Exascale Computing Initiative, led by the Department of Energy of the United States, to become operative in the next three years will include acceleration based on Graphics Processing Units (GPUs) [5]. The ability to adapt modern scientific codes to leverage the power of heterogeneous compute nodes is then fundamental

in this process. Performance evaluation, in particular monitoring and analysis tools, will play a key role not only in enabling algorithms to scale massively onto these parallel systems but also on project productivity and reducing code optimization costs.

Modern performance analysis tools allow for the identification of performance bottlenecks in parallel applications. To do so, numerous profiling, instrumentation and sampling techniques are used to gather performance data. Each tool usually relies on its own unique data format and data visualization viewer, restricting the kinds of analysis that can be done. In general, tools that enable *programmatic analysis* and visualization of performance data are very limited. This is motivating the creation of programmable-oriented tools like the Hatchet Python-library, created in Lawrence Livermore National Laboratory in 2019 [6].

Hatchet is a Python-based library that is motivated by the existence of modern data analysis environments like the Pandas library. Hatchet provides a set of techniques to select, filter and aggregate performance datasets with structured indices, derived from a group of profiling and tracing tools, enabling the analysis of parallel applications' performance data programmatically. Currently, Hatchet provides support for datasets obtained from CPU-specific performance profiling and tracing tools but has not incorporated a way to read in and analyze performance data from GPU application codes profiled with NVIDIA NSight tools. This project is focused on the creation of a workflow that would enable GPU performance data, obtained from the NVIDIA NSight Systems and NSight Compute performance analysis tool, to be loaded and analyzed using programmable approaches like the one provided by Hatchet. This workflow could be used to feed NSight data into other programmable tools or custom-made analysis mechanisms.

1.1 Contributions

The contributions of this investigation are:

- A data processing and manipulation workflow that enables the generation of hierarchical call path information for GPU-accelerated applications from NVIDIA's NSight Systems and Compute output reports. This workflow would unlock programmatic performance analysis of said data for its users.
- A data format capable of storing metrics and maintaining hierarchical relationships of execution profiles derived from NVIDIA NSight tools of profiled GPU high-performance workloads.

- A set of performance case studies for three existing GPU-accelerated applications that illustrate how the implemented solution enables programmatic performance analysis.

1.2 Problem Definition

The next frontier of high performance computing is expected to be achieved in the next couple of years. Exascale computing systems will be comprised of massive numbers of heterogeneous compute nodes assembled from conventional multicore CPUs coupled with massively parallel GPU accelerators [7]. This integration is not only driven by the need of better power efficiency in supercomputers but also by the broadening landscape of applications that run on these systems, in particular the intrusion of AI and big-data workloads. On the latest Top500 ranking, from June 2022, 7 out of the top 10 supercomputers in the world included GPU-based acceleration [8], four of them being NVIDIA powered. The Green500 ranking also reflects how engraved GPUs have become in HPC with 9 out of the top 10 systems using GPUs, five of them relying on NVIDIA accelerators [9].

The road to exascale will then require the adaption of scientific codes to take advantage of this billion-way heterogeneous parallelism through the redesign of algorithms [2]. One of the main challenges that have been identified in achieving extreme scale is efficiently scaling applications to the level of concurrency of modern supercomputers [10]. Identifying issues like load imbalances or excessive synchronization events is crucial in preventing Amdahl fractions that would preclude applications from scaling on to hundreds of thousands of processors. To this end, performance analysis tools should incorporate data-analytics techniques that can facilitate and power structured programmatic processing of multiple hierarchical performance profiles.

Currently, there are several parallelism-oriented performance analysis tools that can capture an application's runtime performance data [11, 12, 13, 14, 15]. However, most of them rely on their unique data format to store performance data and usually provide either text based reports or a tool-specific graphical user interface to visualize performance information. One particular set of performance analysis tools of interest to the HPC community, given the common use of NVIDIA GPU accelerators, is their newest profiling toolset composed of: NVIDIA NSight Systems and NVIDIA NSight Compute. Like those previously mentioned, these system-wide and GPU-specific performance analysis tool generate their own proprietary opaque output data format (.nsys-rep and .ncu-rep) and provide GUI tools to analyze such data. Figure 1.1 shows a view of how NSight Systems' GUI presents performance data for analysis.

NSight performance analysis tools do not enable the user to programmatically analyze their

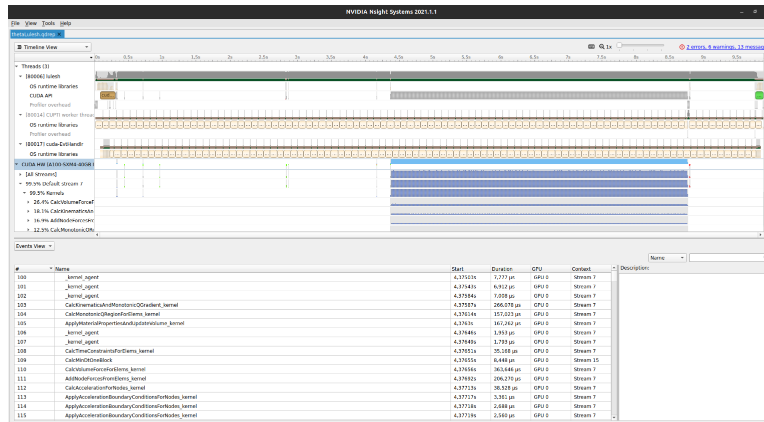


Figure 1.1: NVIDIA's NSight Systems performance analysis GUI showing results of a profiling analysis for Lulesh-GPU

captured data, limiting the types of studies that can be performed. As such, Hatchet's data model and operators could potentially help scientists in creating automated and reproducible performance analysis studies for GPU-accelerated applications. However, Hatchet currently does not have a mechanism to read in performance data gathered with NVIDIA NSight tools and a well-defined workflow to work with this type of data.

Given the state of affairs, it is the **hypothesis** of this thesis that extending Hatchet to work with NVIDIA NSight performance analysis data would enable programmatic performance data analytics capabilities of GPU-based application codes, allowing for the identification of performance bottlenecks and hotspots in their execution.

1.3 Justification

Development of parallel applications not only inherits the difficulties of traditional sequential programming but introduces a range of potential issues arising from parallelism like communication, synchronization and load imbalances. Furthermore, given the wide-spread inclusion of heterogeneous compute nodes, architectural details have to be taken into account when optimizing codes for a specific system. All of these characteristics make parallel software development error-prone. As such, tools for debugging and performance analysis play a key role in enabling world-leading applications and simulations to scale on to billion-fold concurrent supercomputers. These tools should allow for the automation and reproducibility of performance analysis studies in a way that makes the performance optimization process a productive one. In doing so, the end goal of HPC of

powering next-generation science and engineering discoveries would be one step closer.

Even though current tools for performance analysis are able to capture and report on performance statistics and metrics, they usually rely on proprietary data formats and tool-specific data viewers that constrain the types and automation of performance studies that the end-user can do. The primary innovation of this thesis will be the creation of a Hatchet-based performance analysis workflow that would potentially help users analyze GPU-derived hierarchical performance data in an open, programmable and reproducible manner through modern data analytics techniques.

Through the results of this thesis, the Hatchet performance analysis library would be capable of allowing end users to analyze hierarchical performance data generated by the popular NVIDIA NSight Systems tool. This library is currently free and open-source and was created at Lawrence Livermore National Laboratory, United States. Users of this performance analysis library include scientists and engineers of the main national laboratories in the United States that are looking to optimize their applications for GPUs and multi-node scaling. Results of this thesis will be of interest to the broad High Performance Computing community given that next-generation supercomputers are expected to rely heavily on GPU devices for acceleration.

The work on this thesis encompasses several aspects of Computer Science. Knowledge on data structures like trees and graphs is important when dealing with structured performance data. Furthermore, as we are dealing with GPU devices, parallelism and advanced computer architecture knowledge is required to implement a solution to the formulated problem and hypothesis. In terms of higher abstraction abilities, being able to apply the developed tool to analyze performance issues is also part of this thesis. Being able to pinpoint and justify bottlenecks or scaling problems in an application requires not only technical skills but also having an integral view of the applications, their parallelism and the architectural features of the underlying platform that is being used to analyze performance.

1.4 Objectives

1.4.1 Overall Objective

Extend the Python-based Hatchet library to allow for NVIDIA GPU hierarchical performance data to be manipulated and analyzed through programmatic data analytics techniques, enabling the identification of performance bottlenecks and hotspots.

1.4.2 Specific Objectives

- **SO1**: Generate a Hatchet-compliant GPU performance data format from NVIDIA NSight Systems tools profiling reports.
- **SO2**: Design and implement a performance analysis workflow to manipulate large-scale GPU call path profiling data using Hatchet.
- **SO3**: Apply the newly added Hatchet features on at least three GPU-application codes to identify possible performance issues.

Chapter 2

Background

2.1 Performance Measurement and Analysis

Throughout the evolution of computers, the need of hardware and software developers to understand the performance of these systems has remained a constant. As such, performance evaluation has become a part of the whole development cycle of new devices and applications. When applied to computer science and engineering, performance analysis can be defined as a process that combines measurement, interpretation and communication of a system's performance metrics [16]. This definition can be quite broad as a system could be seen as any collection of hardware and/or software components. Furthermore, the metrics used could be quite different depending on the system being studied, e.g. execution/response time, throughput, utilization, availability.

Given the wide range of systems and measurement criteria, performance analysis includes two key steps: selecting an evaluation technique and a set of performance metrics. Historically, the three techniques used are analytical modeling, simulation and measurement [17]. Analytical models are mathematical characterizations of a system that provide insights into its behavior. They often rely on simplifications and assumptions and as such, they usually provide much less accurate results than the other techniques.

On the other hand, simulation is the development of a program that models the key features of the system being studied. Simulations can then be easily modified to understand the effect of varying parameters on performance. When compared to analytical models, simulations require less assumptions and thus, provide somewhat more accurate results. However, developing and executing these simulations takes time, making this technique more costly than simple analytical models.

The third technique, which is the focus of this project, is measurement. Contrary to the two previous approaches, measurements are only possible if the system being studied, or at least a prototype of it, already exists. In terms of results, measurement provides the best accuracy as no assumptions or simplifications need to be made. However, measurement is the most costly option as it not only requires the system to have been already created but also relies on equipment, instruments and time.

2.1.1 Performance Monitoring

In the scope of HPC, performance analysis tools help application scientists tune their codes for a specific architecture. In particular, applications go through an iterative tuning process in which runtime behavior is measured to identify portions of the code that consume the largest percentage of the total execution time, i.e. bottlenecks, and then developers optimize those code segments. Performance analysis tools enable the first two tasks of that cycle.

The first component of a performance analysis tool is the monitoring of performance data. Modern tools rely on what is known as the event model to do that. In this abstraction, an event is a predefined change in the system state that happens at some specific time in a process or thread [16, 17]. An event can be seen as an instance of an action that triggers a program interruption. Computation is stopped and either attributes of the event are recorded or statistics are collected [18]. Examples of events may be the entering or exiting of a user level function, the start/finish of a communication operation in an MPI program or an L2 cache miss. Three major event monitoring techniques are commonly used: hardware monitoring, sampling and instrumentation [19].

Ideally, the monitoring of an application should be minimally intrusive to its performance. In general, software monitors are usually competing with the executed program for hardware resources having varying degrees of impact on its performance (*probe effect* [20]). Modern processors are equipped with specific hardware monitoring registers, known as hardware event counters to reduce that impact [21] . These registers enable the collection of data for a wide range of performance metrics like cache misses or instruction counts. This type of monitoring is required for frequent events. The use of such hardware event counters has become foundational to modern performance analysis tools. The Performance Application Programming Interface (PAPI) has become the de-facto middleware component that other tools use to read data from hardware counters [22].

Given the amount of events that occur during program execution on extreme-scale systems, logging all of them is highly impractical. Tools then rely on statistical performance data extracted through sampling. In this monitoring technique, processor interruptions are used at a given sampling

rate. The interrupt routine logs the current instruction and relevant information about it from the program counter and adds whatever performance metric is being measured to the accumulated total value. For example, if interested in execution time, the sampling interval's length would be added to the sampled function's accumulated execution time. A hybrid sampling technique known as Event-Based sampling can be used such that hardware counter units trigger a sampling event when a hardware counter has passed certain threshold [23]. Sampling may provide low measurement overhead depending on the sampling frequency. The downside to this technique is that only statistical information is gathered. Historically, tools like *gprof* [15] relied on interval timers to sample program execution and determine which parts of a the application consumed the most time. However, with modern hardware counters, other events like instructions executed, cycles, cache misses and hits or stalls can also be measured.

Instrumentation on the other hand, is based on the addition of code to mark specific events or regions of interest in an application. Contrary to sampling, more precise information of an application's behavior can be obtained through instrumentation. This process can take place on different stages of a program: source-code specification, compiler or linker instrumentation and even binary/dynamic instrumentation [23]. On the first type of instrumentation, the developer is tasked with marking specific code regions that are of interest with high-level procedure call statements. For the other types of instrumentation, either the compiler, a source-to-source transformation tool or the monitoring software are responsible of inserting instrumentation to specific code regions. However, instrumentation can distort application performance due to added overhead [11].

2.1.2 Performance Analysis

Performance monitoring is the first step in the process of understanding an application's behavior. Performance analysis tools are then usually classified depending on the level of detail they use to gather and analyze data. In particular, two types of performance tools dominate the landscape: profilers and tracers.

Profiling tools are based on summary statistics of a program's execution. When an event occurs, raw performance data for the application is aggregated, most commonly over time, for the entire execution or for specific regions of interest (region profiles). Some profiling tools are also able to keep an updated summary of metrics per process and/or thread. In doing so, these tools help developers pinpoint which parts of an application's code consume the most time. Profilers can be built using either sampling or instrumentation to monitor performance data. Traditionally, profilers like *gprof*[15] help attribute execution time to specific functions or code statements. Modern

profiling tools like HPCToolkit[11], Caliper[24] and TAU[13] provide information on time spent in different calling contexts. This last type of profiling is specially useful for parallel programs where function invocation can be derived from different calling threads/processes. These tools collect data providing two types of information: *contextual information* like process ID, file name, line of code; and *performance metrics* like cache hits/misses, floating point operations and others[6]. Depending on how sampled data is aggregated, profiles can be categorized as *call path profilers* or *call graph profilers*.

Call path profilers allow for the attribution of a function's cost to context-dependent invocations [23]. This is achieved by recording the stack of procedures present in a thread/process at a certain point in time. So basically, the calling context of an event is the set of procedures that were present in the call stack when the event was triggered[11]. *Call graph profilers* like *gprof* on the other hand don't keep information about the stack but rather just apportion the total execution time of each function among its callers. Data from *call path profilers* can be used to create a **Calling Context Tree (CCT)**. This data structure is a prefix tree in which the path from the root node to an arbitrary node relays information on the call path that led to that given node[25].

The second common type of performance analysis tools are tracers. Even though profilers allow for a quick interpretation of the overall execution of a program, they don't take into account the time-ordering of events [18]. A trace on the other hand, provides the most detailed data as they store information about every individual event. Traces are logs of time-stamped events for which relevant associated metrics are also stored. By relying on traces, a timeline view of a program's execution can be created and profile-like views can be obtained from traced data [26]. Because of their nature, the main concern of trace generation is the volume of data that is produced by these tools.

Performance tools are used to focus on a particular data analysis method. However, given that profiling might hide time-dependent performance anomalies and tracing might have scalability issues derived from high data volumes, modern performance analysis suites like HPCToolkit [11], TAU [13], Intel's VTune Amplifier [12] and ompP [14] support both profiling and tracing. These tools have become popular thanks to this property, given that either profiling or tracing may be the best technique for a specific situation.

2.2 Parallel Computing with GPUs

This project is focused on the analysis of performance data extracted from GPU-accelerated applications, in particular from NVIDIA powered systems. As such, understanding how these devices interplay with the CPU, how they are organized and how the different programming models expose their parallel capabilities is important. We will start by reviewing the major concepts of NVIDIA GPUs device organization and architecture. Then, we will shift our focus on to how GPUs are currently incorporated into high-performance computing clusters where through the offloading model programmers can accelerate computation. Finally, we will discuss how two major frameworks CUDA and OpenMP provide different levels of GPU abstraction to parallelize HPC applications.

2.2.1 GPU Hardware Model

During the early 2000s, when faced with prospective heat dissipation and energy-consumption issues on CPUs, the semiconductor industry started diverging on two different paths of microprocessor design: *multicore* and *many-core* systems [27]. Up to that point, advances in manufacturing hardware technologies had regularly provided programmers with increasing speed for their applications with every new processor generation. *Multicore* processors incorporate multiple high complexity (out-of-order execution, multi-instruction issue) cores focused on maintaining sequential application performance. *Many-core* systems on the other hand, focus on increasing parallel throughput and to do so, rely on a large number of much simpler cores [28]. This latter approach has been used in the development of GPUs for years. Current NVIDIA devices like the V100 and A100 GPUs have core counts of 5376 and 6912 respectively (single-precision floating point cores) [29, 30].

Amid this shift in processor trends, the scientific and high performance computing community started exploring the usage of Graphic Processing Units as a complementing processing element to the CPU. GPUs became attractive because of their floating-point processing and high throughput capabilities [31]. At that time, the term General-Purpose GPU (GPGPU) was coined to describe how their usage was changing from a purely specialized-purpose (graphics and image rendering) to a more general-purpose case where these devices started powering classical scientific applications like matrix operations or computational fluid dynamics [32, 33].

Figure 2.1 shows a simplified block diagram of a typical NVIDIA GPU hardware organization. GPUs are interconnected to a host machine through a high-speed bus, usually PCI-Express. This interface is used to receive commands from CPU, like memory transfers or computation launches.

These commands are then dispatched to the appropiate execution units inside the device. Another crucial aspect of GPU development is memory. The GPU exposes another memory hierarchy separate from the CPU's. At the lowest level we find the High Bandwidth Memory (HBM2) controllers. This is the dynamic random access memory technology used by NVIDIA GPUs, which in modern devices like the V100 and A100 provide memory bandwidths of 900 `GB/s` and 1555 `GB/s` [29, 30] respectively. Although these values may seem high, if all device cores where to be used these bandwidths would not be enough to service all of them at once. This would effectively idle computational resources thus precluding an application from achieving peak performance [34]. For this reason, modern GPUs have also incorporated extra layers of memory hierarchy like a unified shared L2 and L1 caches, as well as a local register files. Each of these levels has memory bandwidths which are orders of magnitude different, thus providing different latency values to an application. For example, the register file, shown in Figure 2.2 runs at the same speed as the processing elements, meaning there would be zero latency for data access in this chunk of memory [35]. Effectively programming for GPUs, requires an understanding of how to utilize this complex hierarchy in order to maximize resource utilization.

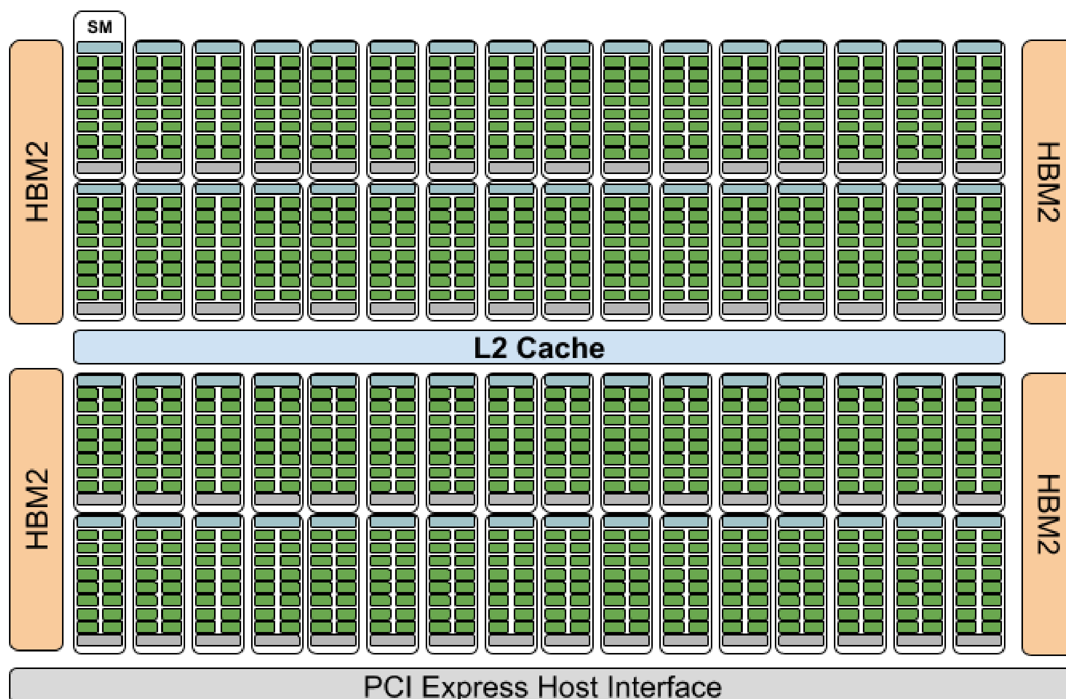


Figure 2.1: Simplified block diagram of a traditional NVIDIA GPU hardware organization.

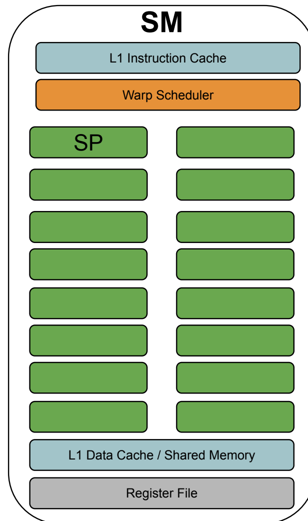


Figure 2.2: Simplified block diagram of the main components of an NVIDIA Streaming Multiprocessor

Now, at the heart of modern NVIDIA GPUs we usually find a collection of Streaming Multiprocessor (SM)s. Figure 2.2 shows how each SM in turn is made up of several Streaming Processors SP. The number of SMs and SPs per SM in an NVIDIA GPU varies across generations. This feature has enabled GPUs to scale considerably over time.

Each SM is able to execute hundreds of threads simultaneously thanks to what NVIDIA has termed the Single Instruction Multiple Thread (SIMT) architecture. In this model, multiple threads issue the same instruction applying it to different data. This feature is what enables GPUs to have such massive data parallel throughput capabilities. SIMT pipelines instructions to power Instruction Level Parallelism (ILP) within a single thread and relies on hardware multithreading to provide Thread Level Parallelism (TLP) on each SP. These SPs are simple in-order execution cores with no branch prediction or speculative execution [36]. Each thread that executes on top of these SPs, has their own instruction address counter and register state so independent execution is possible. This model enables programmers to develop code following both a thread-level or a data-parallel approach. In the following sections we will explore how the different software models expose this hardware to the programmer and what considerations must be taken when analyzing and optimizing performance.

2.2.2 GPUs in High-Performance Clusters

In the context of clusters and supercomputers, heterogeneous computing is defined as a scheme in which the system is composed of compute nodes with distinct mechanisms or models of instruction execution [37]. Graphics Processing Units have become the most popular technology to be used as accelerators in such heterogeneous systems. However other platforms like Field Programmable Gate Arrays (FGPA) are also being used as parallel processors. It is worth noting that GPUs and FGPAs are not substitutes of the CPU but rather are exposed as computational aids with extreme parallelism capabilities, thus they are usually referred to in this context as accelerators.

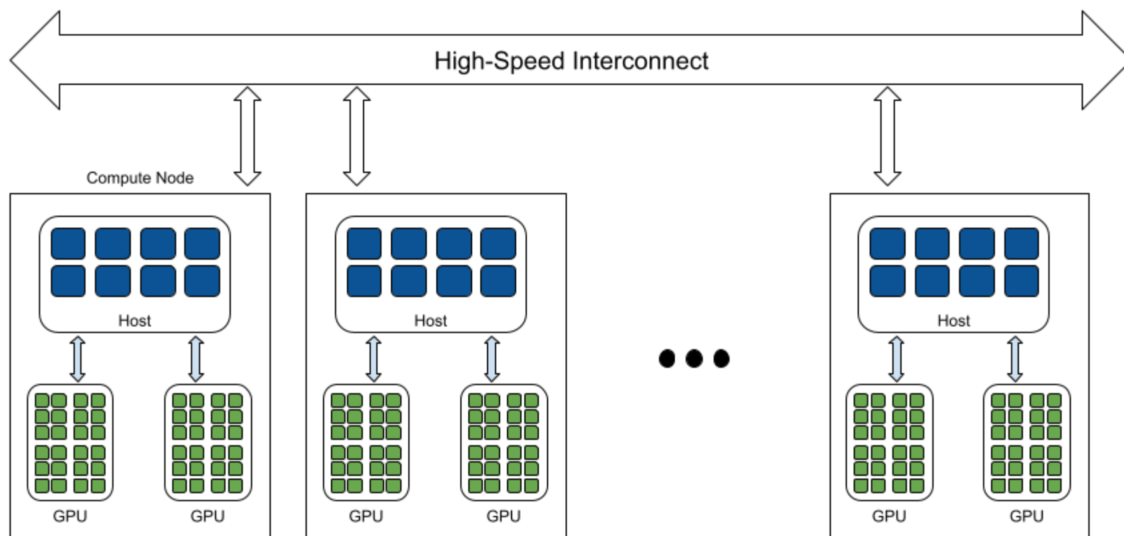


Figure 2.3: Heterogeneous computing system model

In the HPC domain, an heterogeneous compute cluster is composed of a set of distributed nodes made up of traditional shared memory multicore processors and attached GPUs. As mentioned in section 2.2.1, the CPU and GPUs in each node are connected through PCI-Express and nodes in the system are interconnected through some high-speed network fabric for inter-node communication, such as Ethernet or Infiniband. Figure 2.3 shows a high level overview of said heterogeneous clusters.

In heterogeneous systems, there are at least three possible levels of interaction that can be described [38]. First, there's inter-node interactions usually through processes and a communication

mechanism like the popular Message Passing Interface (MPI) [39]. The second level can be found at the node level. Each host component is typically a multicore system capable of threading. This type of mechanism can be exploited through threading libraries and APIs like the popular OpenMP [40]. And finally, heterogeneous systems exhibit CPU-GPU interactions in which data transfers going back and forth between host and devices are used to offload some of the computation on to the accelerator. A fourth level of interest that arises when analyzing application performance is single GPU application behavior, at which we are interested in how an application leverages the different hardware and architectural features of a Graphics Processing Unit. Thus, effectively analyzing the performance of a scientific application that runs on a supercomputer and relies on GPU-acceleration requires the ability to factor in all of these different possible levels of complexity.

2.2.3 GPU Programming Models

In terms of GPU programming and its interaction with the CPU several models have been used in HPC, such as OpenCL[41], HIP[42], OpenACC[43] and OpenMP[40]. The choice of model depends on several factors like programmer experience, portability concerns, programming language and of course target platform [44]. For the sake of this project, we will focus solely on two of the most popular models: Compute Unified Device Architecture (CUDA) [45] and OpenMP.

2.2.3.1 CUDA

Compute Unified Device Architecture (CUDA) is NVIDIA's flagship programming model for General-Purpose GPU development. It was first introduced in 2007 so that developers could take advantage of device hardware without having to deal with complex graphics pipelines as was necessary before its introduction. CUDA development framework includes compilers, language extensions, libraries and development tools focused on facilitating GPU computing. In particular, CUDA provides C language extensions that enable the programmer to target potentially parallel portions of an application to be executed on the GPU.

Figure 2.4 shows the fundamental idea behind CUDA: the kernel offloading model. In this scheme, there are two co-existing processors that ideally work together. The CPU is usually referred to as the *host* while the GPU is termed the *device*. The *host* is where program execution starts and where *device* functions are called. When developing a CUDA-based application, compute intensive tasks, named *kernels*, are identified and marked with language extensions. When executing, the host sets up and transfers the necessary data on to the device and then launches the kernel, which executes operations on the GPU and then finalizes, returning control to the CPU.

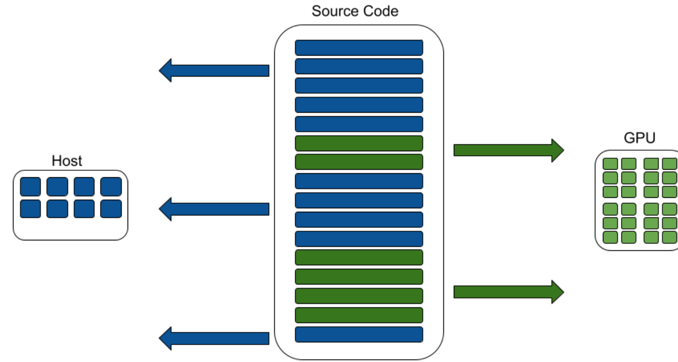


Figure 2.4: CUDA CPU-GPU kernel offloading model

CUDA-implemented applications consist of a unified source code that includes both *host* and *device* code. The NVIDIA compiler then separates it into two, having the *host* compiler handle CPU code and dealing with *gpu* code itself. We will use a vector addition operation as example to review the major concepts in CUDA. Code Listing 1 shows how this operation is usually implemented when executed on a CPU. The for loop iterates n times, where n is the size of the vectors that are being added.

Listing 1 Example of sequential element-wise vector addition

```
1  // A for loop is used to iterate over the positions of the input arrays and stored in the
   ↪  resulting vector
2  void vecAdd(double *a, double *b, double *c, int n)
3  {
4      for(int i=0; i<n; i++){
5          c[i] = a[i] + b[i];
6      }
7  }
```

Code Listing 2 shows a simplified version of this vector addition operation implemented with CUDA. This source code shows some of the major components of a GPU-accelerated application. Notice that the `vecAdd` function has been marked with the `__global__` declaration specifier. This prefix indicates to the compiler to generate *device* code for this function and not *host* code. This is how *kernel* functions are specified in CUDA. Now, before going into the changes inside the `vecAdd` function, notice that in the `main` function, extra operations have to be performed for this application to run. In particular, memory related operations like allocation and data movement.

As mentioned in Section 2.2.1, the GPU has a separate memory hierarchy from the CPU.

Listing 2 Example CUDA element-wise vector addition

```
// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    if (id < n)  c[id] = a[id] + b[id];
}

void main( int argc, char* argv[] )
{
    // Size of vectors
    int n = 100000;
     // Host vectors
    double *h_a, *h_b, *h_c;
     // Device input vectors
    double *d_a,*d_b,*d_c;
     // Allocate memory for each vector on host
    h_a, h_b, h_c = (double*)malloc(size);
     // Allocate memory for each vector on GPU
    cudaMalloc(&d_a, size); cudaMalloc(&d_b, size); cudaMalloc(&d_c, size);
    //Initialize vectors on CPU
    vectInit(h_a, h_b, h_c);
    // Copy host vectors to device
    cudaMemcpy( d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, h_b, size, cudaMemcpyHostToDevice);

    int blockSize, gridSize;
    // Number of threads in each thread block
    blockSize = 1024;
    // Number of thread blocks in grid
    gridSize = (int)ceil((float)n/blockSize);

    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

    // Wait for GPU to complete and copy array back to host
    cudaDeviceSynchronize()
    cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

    //Do something with resulting array
    // Release device memory
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
}
```

This separation must be taken into account in the CUDA programming model. From the software perspective, the *host* and *devices* have independent memory address spaces. Kernels execute based on data inside *device* memory and as such, the model provides operations to allocate, deallocate and transfer data between address spaces. Lines 19 and 41 in Code Listing 2 show how memory is allocated and freed in the device and lines 23, 24 and 37 show the basic data transfer operations provided by CUDA. Notice that when transferring data, the direction must be specified as a parameter

to the `cudaMemcpy` operations.

Going back to the changes inside the `vecAdd` kernel, these are related to the Single Instruction Multiple Thread architecture mentioned in Section 2.2.1. In CUDA, the thread is the fundamental unit of its execution model. When a kernel is *launched*, a large number of threads should ideally be generated in the GPU to exploit data parallelism. Figure 2.5 shows the execution model of a CUDA-based application. When invoked, kernel execution is moved to the target *device* where a group of blocks of threads are created to handle execution. In CUDA, a group of threads is labeled a *block* and the total group of blocks used to execute a kernel is termed a *grid*. When launching a kernel, both the desired size of the *grid* and *blocks* must be specified. All blocks are created with the same amount of threads.

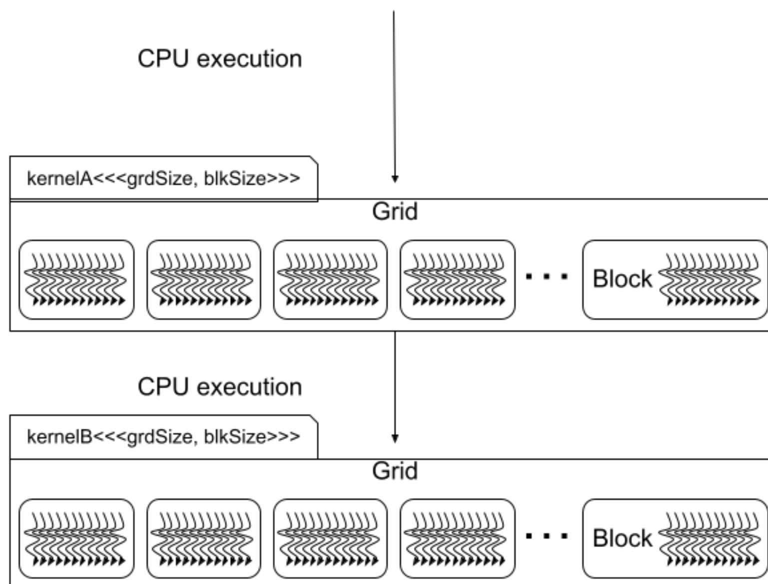


Figure 2.5: CUDA parallel kernel execution model

In the vector addition example (Code Listing 2), the kernel launch is specified in line 33. The `<<<>>>` keyword signals the compiler that this is a *device* function call. We now have a large number of threads to handle the operation so instead of relying on an iteration loop, we assign different threads to each i-th addition, effectively parallelizing this computation. To do this, we take advantage of the thread hierarchy in CUDA. When a kernel is launched, each *block* is assigned a sequential numerical identifier. Each *thread* inside each block is also sequentially numbered so, to identify a thread in the total *grid* the operation `blockIdx.x*blockDim.x+threadIdx.x`

is used. In the case where we launch more threads than the size of the input arrays, a conditional prevents the application from executing invalid memory accesses. Another important aspect of the execution model in CUDA is synchronization. By design and to enable simultaneous execution of CPU and GPU, kernel calls are asynchronous, meaning that the host can keep its execution flow while the GPU is executing. Line 36 in Code Listing 2 shows one of multiple available synchronization operations available in CUDA. This operation blocks CPU execution until the *device* function finishes executing. This aspect of CUDA execution will become relevant when analyzing performance and measuring kernel execution time.

One of CUDA's advantages in relation to higher abstraction models is that it has been designed to provide a tight mapping between software and the underlying hardware [34]. Table 2.1 synthesizes the existing relationship between CUDA software components and NVIDIA GPU hardware discussed so far in this section. On the other hand, this characteristic of CUDA can also be viewed as a disadvantage in some cases. In particular, programmers have to deal with mapping their application to low-level details of the architecture and hardware affecting productivity and most importantly portability [46]. In response to this, models like OpenACC, OpenMP and more recently Kokkos [47] provide higher level abstraction of GPU programming looking to provide portability and freeing the programmer from requiring deep hardware knowledge. In the following subsection we discuss how OpenMP can be used in developing accelerated applications.

Software Entity	Hardware Component
CUDA Thread	Streaming Processor (SP)
CUDA Block	Streaming Multiprocessor (SM)
Kernel Grid	GPU Device

Table 2.1: CUDA Software - Hardware Mapping

2.2.3.2 OpenMP

The Open Multi-Processing (OpenMP) API is a widely adopted standard for shared-memory parallel programming in the HPC community [48]. Starting from 2013, the standard incorporated *device constructs* for heterogeneous computing and has since been updating and adding functionalities to enhance accelerator programming following a directive based approach [40]. Conceptually, OpenMP allows programmers to develop single-source applications that can be executed either on multicore CPUs or on GPUs. The main advantage in comparison to an architecture specific approach like CUDA is that it is platform-agnostic thus, making code implemented in this standard portable.

OpenMP is a compiler directive based API that provides runtime functions and environment variables that enable programmers to control parallelization. It is the responsibility of the programmer to identify code sections that could be executed concurrently and add the appropiate constructs to ensure application correctness. Through this model both *task* and *data* parallelism can be exploited. The latter being the most relevant for GPU acceleration.

In the traditional multicore execution model, OpenMP applications start their execution sequentially. A *master thread* that runs throughout the lifetime of the program handles the serial sections. At different points of the application where the programmer has identified and specified *parallel regions* using compiler directives, named *pragmas*, additional threads are created. At this point, with multiple active threads, program execution is carried out in parallel. Thread synchronization constructs are part of OpenMP and allow the programmer to coordinate the different threads. When a *parallel region* completes, the master thread regains control of the application and continues with its execution.

Listing 3 Example OpenMP target element-wise vector addition

```
1  // OpenMP target region. Data elements are distributed among teams and threads
2  void vecAdd(double *a, double *b, double *c, int n)
3  {
4      #pragma omp target teams distribute parallel for
5      for(int i=0; i<n; i++){
6          c[i] = a[i] + b[i];
7      }
8  }
9
10 void main( int argc, char* argv[] )
11 {
12      // Host vectors
13     double *h_a, *h_b, *h_c;
14     // Allocate memory for each vector on host
15     h_a, h_b, h_c = (double*)malloc(size);
16     //Initialize vectors on CPU
17     vectInit(h_a, h_b, h_c);
18
19     // GPU data transfers and allocation
20     #pragma omp target enter data map(to:h_a[0:size], h_b[0:size]) map(alloc:
       ↪  h_c[0:size])
21
22     // Execute the operation on GPU
23     vecAdd(d_a, d_b, d_c, n);
24
25     // GPU transfer result back to host
26     #pragma omp target exit map(from:h_c[0:size])
27     //Do something with resulting array
28 }
```

The execution scheme followed by OpenMP is called the *fork-join model* [49]. This model is

complemented with additional constructs to handle accelerator programming. Similarly to CUDA, OpenMP defines a separation between the *host device* and the *target device*, the latter being some sort of accelerator, like GPUs. Again, Code Listing 3 shows the same vector addition operation we used as example in Section 2.2.3.1, but now implemented with OpenMP.

Execution in this example application starts with a master thread executing on the *host*. The necessary data structures are allocated and initialized on the CPU. As we intend to parallelize the vector addition for loop, the required data must be transferred or allocated on the *target device* data environment. Line 20 shows how data movement is specified in OpenMP through compiler directives. The `#pragma omp target` directive is used to establish a *target* region, which implies execution on the accelerator. The `map` clause is used in conjunction with `target` to specify a list of data variables that need to be created or transferred to and from the accelerator. When a variable might be needed across multiple *target* regions, constantly copying it to and from the device might be detrimental to performance. In these cases, the `target enter data` and `target exit data` provide continuous access to variables across different *target* regions.

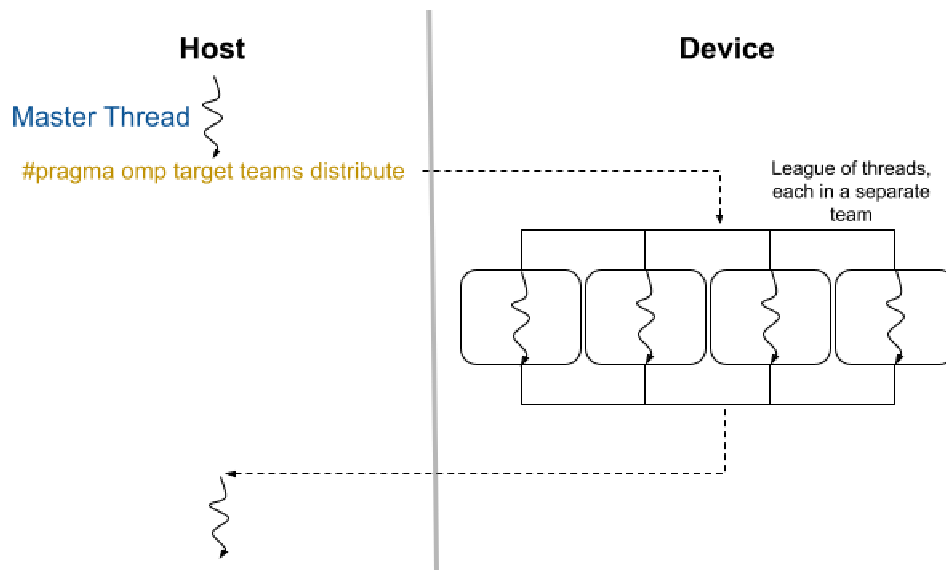


Figure 2.6: Threading behavior in device when using target teams directive

After dealing with memory management and data movement, we must deal with specifying parallelism inside the *device*. The directive on Line 4 of Code Listing 3 is a composite construct. Figure 2.6 shows how execution occurs when we solely rely on the `#pragma omp target teams distribute` portion. Execution starts on the host with a master thread. When the

aforementioned pragma is encountered, execution flow is transferred on to the accelerator where a league of threads is created. However, these threads are distributed across different *teams*, which in terms of hardware, means each Streaming Multiprocessor will execute only one thread. The `target teams` combination on its own specifies that the subsequent code block should be executed in parallel. So, in Code Listing 3 each thread would execute the for loop entirely, which is not the desired behavior. The idea of using multiple threads on a for loop is to apportion the iterations across the different threads, which is achieved by adding the worksharing `distribute` clause to the directive. When finished executing, control returns to the *host* master thread. One major difference with respect to CUDA is that in OpenMP the activation of a *target region* is synchronous, so execution flow in the CPU is stalled until the *accelerator* completes its execution.

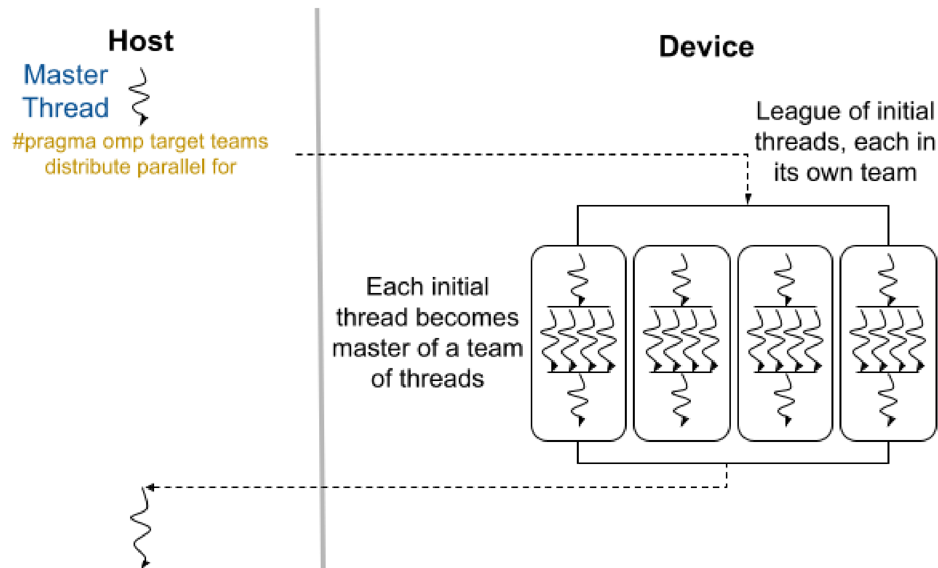


Figure 2.7: Threading behavior in device when nesting a parallel construct in a target teams directive

One major downside of relying on the `target teams distribute` is that parallelism is restricted to just one thread per team. This greatly limits the amount of SPs we use inside each GPU SM. OpenMP allows us to create a second level of parallelism inside the *device*, at the team level. Figure 2.7 shows how this is done. The `parallel` clause further initializes multiple threads per team instead of having just one thread per team. This is complemented with the `for` clause which then distributes the team-apportioned iterations of the loop across the different team threads. Just like before, when each team of threads completes its execution, control returns to the master thread on the *host*.

As previously mentioned, one of the attractive features of OpenMP is its promise of portability. As a way to further increase productivity, the standard has incorporated two different modes of parallelism specification for accelerators. Developers can then choose between one of two implementation decisions: *prescriptive* and *descriptive* parallelism [50]. The first of these approaches is what we used in Code Listing 3. Under this model, the programmer dictates to the compiler and runtime where and how parallelization should occur. In this example, we explicitly tell the system to create two levels of parallelism inside the device and to distribute the iterations of the loop accordingly. When using the *descriptive* approach, the programmer is responsible for just hinting at what code sections should be parallelized but not how, thus leaving this decision to the compiler. The main motivation behind the *descriptive* parallelism approach is productivity, so the `#pragma omp target teams loop` directive is used to indicate sections that should be parallelized by the compiler. This reduces the amount of directives and clauses that are needed to parallelize an application and frees the programmer from having to understand how their choices are being mapped on to the underlying hardware.

2.2.4 NVIDIA NSight Performance Analysis Tools

Now that we have had an overview of the major concepts of performance analysis and the different aspects of GPU development the importance of providing an integral perspective of all them should be clear. This is, we would like to analyze an application as a whole, considering inter-node communication, intra-node parallelism, CPU-GPU kernel offloading and specific kernel operation metrics that describe overall GPU architecture utilization. This kind of comprehensive analysis would enable applications to truly grasp all of the available computing power that modern hardware architectures offer. As part of this effort, NVIDIA created a product family under the NSight name [51] of which this project is particularly interested in NSight Systems [52] and NSight Compute [53].

2.2.4.1 NVIDIA NSight Systems

NSight Systems is a performance analysis tool for system-wide analysis developed by NVIDIA. In particular, it's this corporation's newest tool created to aid developers in tuning and scaling GPU-accelerated applications. Targeted as a system-wide tool, NSight Systems allows for the identification of performance issues on both the CPU and the GPU as well as their interplay, like excessive synchronization operations, numerous data transfers or low GPU usage due to starvation as well as large kernel launch latencies across an application.

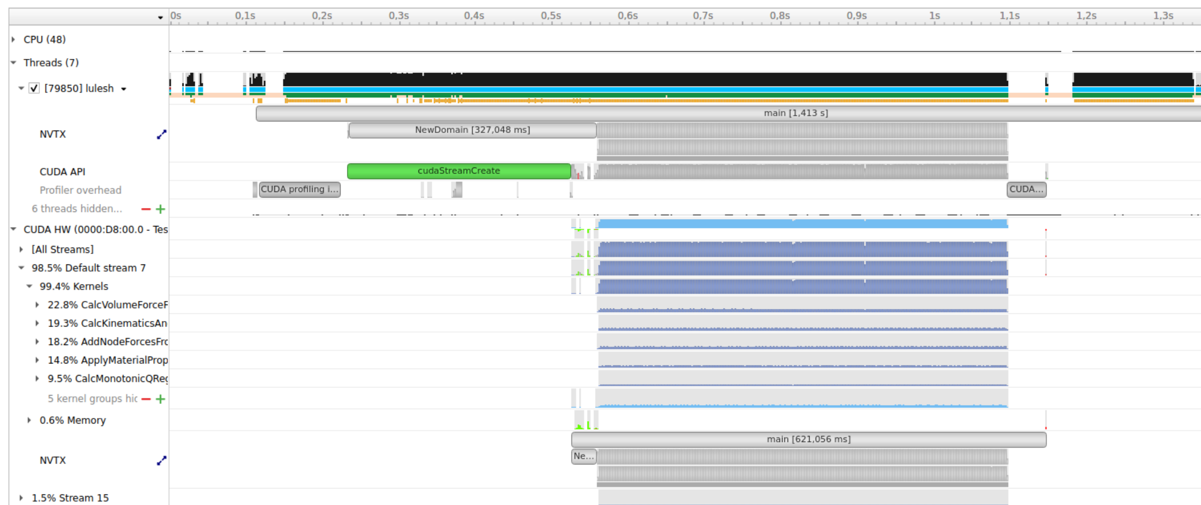


Figure 2.8: NSight Systems timeline trace view for a GPU-accelerated application.

In terms of data monitoring and collection, NSight Systems uses statistical sampling to generate traces [54]. This tool's main feature is its tracing capabilities to understand events in a time-ordered manner. As part of the data collection process, NSight Systems periodically interrupts the application to gather statistical measurements of performance metrics. It also provides several backtracing algorithms to gather data on the active functions on the stack of a specific thread. Through this data, NSight provides a timeline-based view of events in a profiled GPU-acccelerated application, like Figure shows.

2.2.4.2 NVIDIA NSight Compute

NSight Systems provides a helpful view of system wide issues and could be used to analyze application runs that utilize multiple MPI ranks, GPUs and GPU streams. It is also helpful in identifying particularly troublesome kernels that are not behaving as expected. However, it does not provide GPU-architecture and hardware specific metrics that could help understand why that unexpected behavior is happening. This is precisely the end-goal of NSight Compute, an interactive profiler that collects performance metrics relevant to kernel execution.

Data capture by NSight Compute is achieved through the injection of measurement libraries that enable the interception of CUDA driver data as well as hardware performance counters and software patching of kernel instructions. This tool provides a quite comprehensive list of possible metrics that could be collected: from instruction statistics that inform the user of executed low-level

assembly instructions, to what NVIDIA calls *SpeedOfLight* metrics that give a high level summary of achieved percentage of utilization with respect to the theoretical maximum. Figure 2.9 shows a portion of the graphical report that NSight Compute provides. The tool does not only show raw metric values but provides insights through some analysis rules and provides recommendations on possible performance optimization actions.

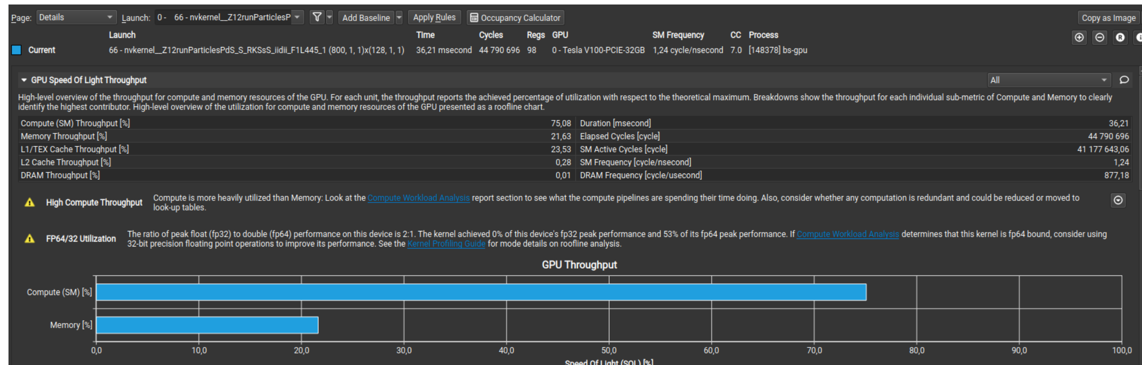


Figure 2.9: Example SpeedOfLight Section of Nsight Compute GUI report

An important aspect of NSight Compute operation is that, depending on which metrics the user wants to collect, kernel execution must be repeated one or more times. Hardware and reproducibility constraints preclude the system from collecting all hardware performance counters in one single pass [55]. This requirement and the number of metrics that the user wants to obtain can incur different levels of overhead on the application.

Both NSight Tools have a clear role in the performance optimization cycle of GPU-accelerated applications. This robust tool ecosystem has helped NVIDIA cement itself as the major accelerator vendor in the HPC domain so far.

2.2.4.3 NVTX Instrumentation

Of particular interest to this thesis, NSight Systems provides tracing of user specified code regions. To achieve this, the NVIDIA Tools Extension Library (NVTX) is used as an instrumentation mechanism to mark regions of interest [56]. NVTX, a C-based API, is used to annotate the execution time line with CPU events by marking time ranges with meaningful names. Code listing 4 shows an example of how NVTX annotations are specified in code, particularly the proxy app LULESH [57]. In this case, the `parentFunction` marks the `CalcPositionAndVelocityForNodes` event with a Push-Pop range. This specific type of annotation is useful when working with nested

time ranges that start and end in the same execution context (thread or process). In this example, the Push-Pop ranges inside the `CalcPositionAndVelocityForNodes` function would be identified by the profiler as children ranges of the original `parentFunction` range annotation.

Listing 4 Example of NVTX Push-Pop Ranges annotations

```
1  void CalcPositionAndVelocityForNodes(const Real_t u_cut, Domain* domain)
2  {
3      nvtxRangePushA("CalcPositionAndVelocityForNodes");
4      Index_t dimBlock = 128;
5      Index_t dimGrid = PAD_DIV(domain->numNode,dimBlock);
6
7      nvtxRangePushA("CalcPositionAndVelocityForNodes_kernel");
8      CalcPositionAndVelocityForNodes_kernel<<<dimGrid, dimBlock>>>
9          (domain->numNode,domain->deltatime_h,u_cut,
10          domain->x.raw(),domain->y.raw(),domain->z.raw(),
11          domain->xd.raw(),domain->yd.raw(),domain->zd.raw(),
12          domain->xdd.raw(),domain->ydd.raw(),domain->zdd.raw());
13     nvtxRangePop();
14
15     nvtxRangePop();
16 }
17 void parentFunction(){
18      nvtxRangePushA("ParentFunction");
19      CalcPositionAndVelocityForNodes(u_cut, domain);
20      nvtxRangePop();
21 }
```

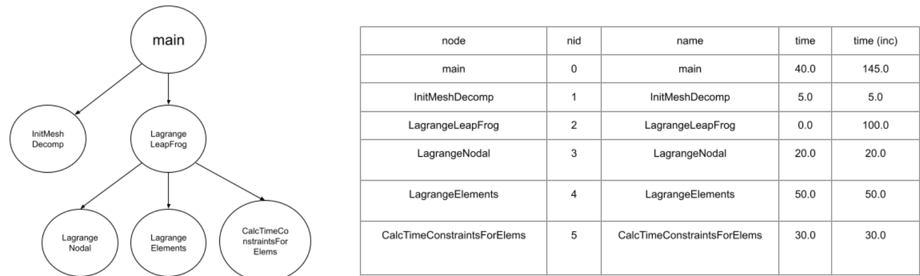
Both NSight Systems and Compute are capable of profiling NVTX annotated portions of an application to provide concise data of user-interest regions. This annotation tool is also used to generate output data that is later used to reconstruct aggregated profiles from time-based performance data. This process will be discussed in further sections of this document.

2.3 Hatchet: Data Analysis of Performance Data

Although modern performance analysis tools are able to monitor and collect diverse data and metrics, they usually rely on their own unique data formats and graphical user interfaces (GUI) to visualize that data. This lack of standardization for performance data and inter-tool compatibility constrains the kinds of analyses that end-users can do. Furthermore, as most tools are GUI-based they offer limited functionality to the user in terms of programmable performance analysis making this task often very tedious. This is the motivation behind the recently developed performance analysis library Hatchet [6].

Hatchet is a Python-based library that builds on the features of data-science oriented libraries

to provide programmatic analysis of structured performance data. In particular, Hatchet's strength is its capability to handle call path and calling context tree data generated by different performance monitoring tools, mainly HPCToolkit[11] and Caliper[24]. Hatchet is built on top of the open-source Pandas library [58], using its DataFrame data structure to store performance data.



node	nid	name	time	time (inc)
main	0	main	40.0	145.0
InitMeshDecomp	1	InitMeshDecomp	5.0	5.0
LagrangeLeapFrog	2	LagrangeLeapFrog	0.0	100.0
LagrangeNodal	3	LagrangeNodal	20.0	20.0
LagrangeElements	4	LagrangeElements	50.0	50.0
CalcTimeConstraintsForElems	5	CalcTimeConstraintsForElems	30.0	30.0

Figure 2.10: Example of Hatchet's GraphFrame data structure using a code-segment of LULESH

Pandas DataFrames are a mechanism to store potentially mixed-datatype information in a two-dimensional tabular format. This type of data structure is common when working with multidimensional data. In a DataFrame each column can be seen as a one-dimensional homogeneously-typed array and some of the columns of the structure can be used as the index for the DataFrame (multi-indexing is also supported). The great power of the Pandas DataFrame is the set of operations, inspired from datasheets and SQL queries, that can be performed on the data: subsetting, slicing, inserting and deleting columns and even aggregating or grouping data.

```
>>> gf.dataframe.head(30)
                                                      time (inc)      time  nid                            name
node                                            rank
{'name': 'main', 'type': 'region'}              0      5882425.0  121489.0    0                            main
                                                1      5902349.0  105528.0    0                            main
                                                2      5898577.0  110799.0    0                            main
                                                3      5882996.0  113830.0    0                            main
                                                4      5905595.0  118953.0    0                            main
                                                5      5877613.0  133256.0    0                            main
                                                6      5870933.0  114035.0    0                            main
                                                7      5898724.0  137098.0    0                            main
{'name': 'LagrangeLeapFrog', 'type': 'region'}  0      5342467.0     528.0    1                LagrangeLeapFrog
                                                1      5584419.0     499.0    1                LagrangeLeapFrog
                                                2      5616143.0    3520.0    1                LagrangeLeapFrog
                                                3      5445647.0     511.0    1                LagrangeLeapFrog
                                                4      5571039.0     513.0    1                LagrangeLeapFrog
                                                5      5402024.0     517.0    1                LagrangeLeapFrog
                                                6      5333089.0     525.0    1                LagrangeLeapFrog
                                                7      5761086.0     543.0    1                LagrangeLeapFrog
{'name': 'CalcTimeConstraintsForElems', 'type':... 0    137012.0   21493.0    9     CalcTimeConstraintsForElems
                                                1        24826.0    2745.0    9     CalcTimeConstraintsForElems
                                                2        24111.0    2758.0    9     CalcTimeConstraintsForElems
                                                3        40633.0   10194.0    9     CalcTimeConstraintsForElems
                                                4        24116.0    2321.0    9     CalcTimeConstraintsForElems
                                                5        29700.0    2828.0    9     CalcTimeConstraintsForElems
                                                6        68123.0   14203.0    9     CalcTimeConstraintsForElems
                                                7        24741.0    2970.0    9     CalcTimeConstraintsForElems
{'name': 'CalcCourantConstraintForElems', 'type... 0     85866.0   85866.0   10   CalcCourantConstraintForElems
                                                1        17558.0   17558.0   10   CalcCourantConstraintForElems
                                                2        16695.0   16695.0   10   CalcCourantConstraintForElems
                                                3        25658.0   25658.0   10   CalcCourantConstraintForElems
                                                4        17195.0   17195.0   10   CalcCourantConstraintForElems
                                                5        21244.0   21244.0   10   CalcCourantConstraintForElems
```

Figure 2.11: MultiIndexed DataFrame component of a GraphFrame in Hatchet

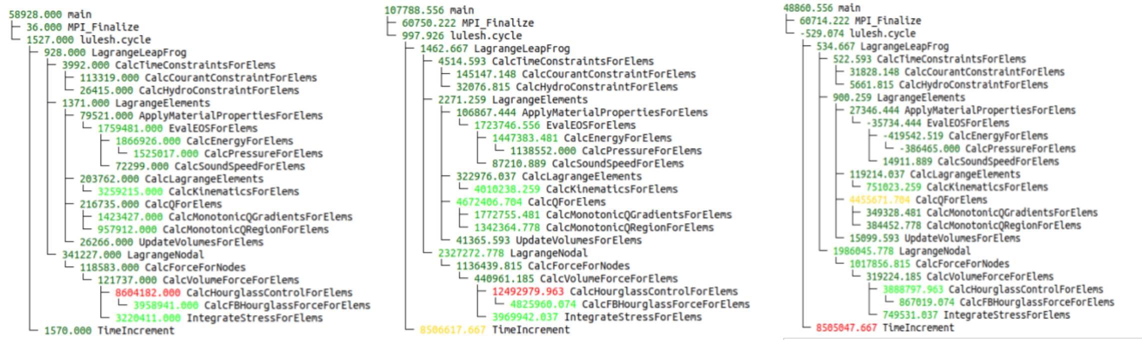


Figure 2.12: Hatchet enables on-node scaling analysis through the GraphFrame structure and its associated algebra operators. Example using LULESH

In terms of indexing DataFrames, Pandas allows indexes of only certain types, e.g. numbers, text or dates. However, to be able to deal with structured performance data with nonlinear data, Hatchet provides data structures that enable DataFrames to be indexed using nodes in a tree or graph. As such, Hatchet proposes its primary data structure named *GraphFrames*, shown in Figure 2.10. *GraphFrames* combine a structured index graph with a pandas DataFrame in which nodes of the graph can be inserted as indexes of the tabular data. Through this structure, data analytics techniques can be used on the DataFrame whilst preserving tree-based relationships. Through this powerful concept, profile data in Hatchet can be high-dimensional when metrics for each function are gathered per-MPI process and/or per-OpenMP thread. In such cases, a MultiIndex can be created consisting of the index of the node and contextual information (MPI rank or thread ID). This is illustrated in Figure 2.11 where a MultiIndex consisting of the function node and the MPI rank is used to differentiate different call paths of a parallel execution of LULESH. The DataFrame component of the GraphFrame can store any meaningful metric that would be relevant to the analysis, typically execution time but other measurements could be used.

Aside from the data structure, Hatchet provides a sort of GraphFrame algebra through which structured performance data can be filtered, aggregated and even pruned. This operations enable interesting and common performance analysis tasks like identifying and understanding load imbalance in multi-node executions or even understanding how scaling hardware resources affect performance. Figure 2.12 shows a case study in which Hatchet is used to understand how varying the number of cores impacts the execution time of certain functions on the LULESH proxy app. This study is performed by loading into Hatchet performance data gathered through Caliper for a run on 1 core and a run on 27 cores.

Figure 2.12 shows three profiles for LULESH. The one of the left is the performance data for a 1 core run and the middle profile corresponds to the same code executed in parallel on 27 cores. The profile on the right is the result of using Hatchet's substract operation, as stated in code listing 5. This study reveals that the `TimeIncrement` function shows the biggest increase in execution time when scaling the application to multiple cores. This might indicate some sort of synchronization overhead issue in this function that might need to be examined for the code to fully scale. Notice that this quick analysis can be easily described through Hatchet's structures and operators.

Listing 5 GraphFrame substract operation in Hatchet

```python
import hatchet as ht
filename1 = 'lulesh-annotation-profile-1core-nompi.json'
filename2 = 'lulesh-annotation-profile-27cores-nompi.json'
gf1 = ht.GraphFrame.from_caliper_json(filename1)
gf1.drop_index_levels()
gf2 = ht.GraphFrame.from_caliper_json(filename2)
gf2.drop_index_levels()
gf3 = gf2 - gf1
print(gf3.tree())
```

Hatchet is a performance analysis library that advocates for the creation of easily scripted automated and reproducible performance analysis studies. It doesn't include any performance monitoring capabilities but in turn is working towards being able to integrate as many performance data formats as possible. Currently, Hatchet is able to analyze CPU-based codes but GPU applications are becoming of increasing interest to the HPC community.

2.4 GPU Accelerated Applications

GPU acceleration powers diverse workloads in HPC. Motivated by this fact we chose three different applications ranging from Artificial Intelligence (AI) to classical High Performance Computing simulation domains like plasma physics and hydrodynamics. These applications will be used throughout the rest of this document to discuss how a Hatchet-extension was designed and implemented as well as understand how this new capability enables programmatic identification of performance bottlenecks. Furthermore, they also help illustrate some of the limitations of our current implementation and the complexities of dealing with all levels of interaction existing in this type of applications.

2.4.1 Tensorflow Keras Model

In recent years, there has been wide discussion about the convergence of AI and HPC [59]. This convergence has been motivated in part by the common reliance on GPU acceleration, but a more symbiotic relationship has been established as AI is now helping research in traditional HPC applications like high energy physics[60], materials science [61] and cosmology[62]. Because of this, we decided to select a deep learning model application that could be used to understand the main challenges in profiling this sort of codes based on modern AI frameworks like Tensorflow [63] or PyTorch [64] with NSight Tools and integrating resulting data into Hatchet.

In this application, a simple model is created and trained against the popular MNIST dataset [65]. The model consists of two large dense layers based on a ReLU activation function and a smaller dense layer connected to a the final softmax activation layer. This example application was extracted from the Tensorflow Keras Performance Documentation. It will enable us to discuss about how mixed precision can be used to optimize a performance bottleneck in this type of GPU-accelerated application.

2.4.2 BS-SOLCTRA Plasma Physics Simulator

The Biot-Savart Solver for Computing and Tracing Magnetic Field Lines (BS-SOLCTRA) is a C++ based application that simulates plasma confinement inside the Stellarator of Costa Rica 1 (SCR1) [66]. BS-SOLCTRA relies on the field-line tracing technique to provide physicists with information about the vacuum magnetic field generated by the modular coils of the SCR1. To do so, this simulator tracks a set of input particles over a time-integration loop on which a fourth-order Runge-Kutta method is used in conjunction with Biot-Savart's Law to understand the magnetic structure inside the reactor.

Particle trajectories are the end result of BS-SOLCTRA. The computation of such paths is independent, meaning that particle interactions are not taken into account. As such, the computation of individual particle trajectories can be carried out completely in parallel. This application had been previously parallelized for HPC execution with MPI + OpenMP for multicore systems and ported to other parallel programming models to deal with load imbalances [67]. Recently, an effort was made to port this application to leverage GPU-acceleration with OpenMP. An initial *prescriptive* implementation was created yielding significant speedups when compared to the original MPI+OpenMP implementation. A second *descriptive* version was implemented resulting in significant performance degradation. In the following sections of this project we discuss how the

added Hatchet capabilities were used to pinpoint the performance hot spot with NSight profiling data.

2.4.3 LULESH Shock Hydrodynamics Simulator

The final code we use in this thesis is the Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) proxy application [57]. This is a hydrodynamics simulation that solves the Sedov blast wave test problem in three dimensions using a Lagrangian approach. Lagrangian methods rely on a mesh to model the problem domain into different material elements and boundaries. A simulation based on such methods, follows the evolution of the materials across the mesh through space and time [68]. The application is coded to simulate one octant of the spherical Sedov blast. Different nodes in the intersections of the mesh cells are used to store kinematic values like positions and velocities. The application then uses a time stepping leapfrog algorithm to the evolution of the blast wave.

LULESH has been used to test different traditional and emerging parallel programming models like MPI, MPI+OpenMP, Charm++, Chapel and of particular interest for this thesis, CUDA [69, 70]. This latter version includes multiple GPU kernels that execute during the different phases of the simulation.

These three applications are used across the following sections of this document to detail how NSight performance data was used to extend Hatchet's capabilities. They also serve as performance case studies to demonstrate how Hatchet can be used to identify bottlenecks across HPC domains.

Chapter 3

From NVIDIA NSight Tools to Hatchet: Transmogrifying Trace Data and GPU Metrics for Programmatic Analysis

3.1 Generating Performance Data with NVIDIA NSight Tools

Hatchet's functionality and attractiveness are derived from its core concept: *GraphFrames*. As mentioned in Section 2.3, this is a composite data structure consisting of both a Pandas dataframe and an indexed tree or graph. This latter component is used to represent hierarchical performance data like an application's call graph or calling context tree. Therefore, the first step in this project was determining how NSight tools gather and report performance data from GPU accelerated applications to obtain said hierarchical information.

Initially, this project was focused on using NSight Systems data only, given that this application is the one providing information on both CPU and GPU execution. Midway through the development of this project, a design decision taken in the interest of providing additional GPU performance data resulted in the incorporation of NSight Compute as an supplementary and optional input to the NSight-to-Hatchet transformation pipeline. We will start by reviewing the process that is used to generate the necessary data on both NVIDIA tools.

The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics proxy application was used as a test workload to guide both the tools exploration and the implementation of a data transformation pipeline.

3.1.1 NVIDIA NSight Systems Time-Based Data

As mentioned in Section 2.2.4.1, the NSight Systems tool is a performance analysis software with a focus on tracing. It's main use is providing the user a time-based view of the different levels of execution involved in a GPU-accelerated application. Execution in HPC environments is carried out through the Command Line Interface (CLI) executable `nsys`, to which the user provides a series of parameters depending on the desired analysis. When running the `nsys` tool, an output file with `.nsys-rep` extension is generated. This is a proprietary and opaque data format that can't be easily understood or manipulated. This report file is then usually loaded by the user to the GUI application to analyze the application's trace. Figure 3.1 shows an example of the view the user gets of performance data for an application profiled with NSight Systems.

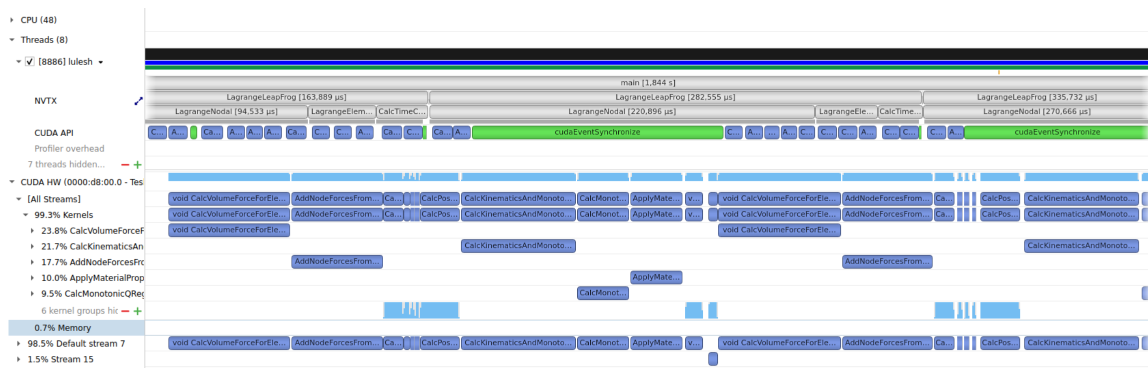


Figure 3.1: NSight Systems Trace GUI for a LULESH execution

The `nsys` tool is capable of tracing multiple common APIs like CUDA, OpenACC, MPI, OpenMP and of particular interest, NVTX. Code Listing 6 shows how a profiling analysis is specified. The *–trace* parameter is used to indicate what APIs should be traced during the execution. In this particular example we are tracing NVTX and CUDA calls.

Listing 6 Example command line used with NSight Systems

```
1 nsys profile --stats=true --trace=cuda,nvtx -o lulesh_nvtx_metrics ./lulesh -s 200
```

Although when the `nsys-rep` file is loaded into NVIDIA's NSight Systems GUI a hierarchical top-down view of the application call path can be seen, no mechanism was found to extract this information. Figure 3.2 shows this hierarchical-like view of the performance data. Moreover, this view seems to show only partial data as no information is given of the GPU kernels that were executed.

Top-Down View — Process [239802] lulesh (7 of 7 threads)

Filter... — 62 927 samples are used.

Symbol Name	Self, %	Total, %	Module Name
_start	·	82,96	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
__libc_start_main	·	82,96	/usr/lib64/libc-2.17.so
main	·	82,96	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
NewDomain(char**, int, int, int, int, int, int, bool, int, int, int)	2,53	37,33	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
thrust::detail::vector_base<int, std::allocator<int> >::operator[](unsigned long)	0,49	18,14	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
thrust::detail::vector_base<double, std::allocator<double> >::operator[](unsigned long)	0,20	7,13	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
Domain::BuildMesh(int, int, int, int, int, Vector_h<double>&, Vector_h<double>&, Vector_h<double>&, Vector_h<int...	0,18	3,78	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
SetupConnectivityBC(Domain*, int)	0,11	2,33	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
Domain::CreateRegionIndexSets(int, int)	0,10	1,80	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
Vector_h<int>::Vector_h(int)	·	1,21	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
LagrangeLeapFrog(Domain*)	·	31,29	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
LagrangeNodal(Domain*)	·	30,49	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
CalcForceForNodes(Domain*)	·	29,91	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
TimeIncrement(Domain*)	·	29,64	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
cudaEventSynchronize	·	29,64	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
cudart::cudaApiEventSynchronize(CUevent_st*)	0,00	29,63	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
LagrangeElements(Domain*)	·	0,51	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
VerifyAndWriteFinalOutput(double, Domain&, int, int, int, bool)	·	13,46	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
write_solution(Domain*)	0,03	13,46	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
fprintf	0,03	12,72	/usr/lib64/libc-2.17.so
thrust::detail::vector_base<double, std::allocator<double> >::operator[](unsigned long)	0,00	0,68	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh
cuda_init(int)	·	0,68	/work/djimenez/thesis/lulesh_gpu/cuda/src/lulesh

Figure 3.2: Top-Down view of the profiled LULESH-GPU code

The *–stats=true* parameter in Code Listing 6 instructs the tool to generate summary statistics after the data collection. By specifying this flag, an additional SQLite database is also generated storing all raw performance data gathered by NSight Systems. Figure 3.3 shows a view of the summary statistics that NSight generated for the same LULESH-GPU profiled run. Notice that even though it provides insight into the usage of different CUDA API calls, data movement and timing information of the different kernels in the code, all of this data is generated in a flat-view way. No hierarchical data or call path information can be inferred from this generated report. Furthermore, NVIDIA ships with this software a series of Python report scripts that the user can use to extract specific data from the SQLite database. However, none of the possible reports that NSight provides "out-of-the-box" gives an integral view of the application, i.e host and device performance data in a hierarchical manner, as required by Hatchet.

As mentioned before, an additional SQLite database is generated for which NVIDIA provides a basic reference documentation. This exported database stores different data depending on the kind of event measured, however it is currently not well documented and requires an advanced understanding of the low-level performance events and their correlation.

```
 Time (%)  Total Time (ns)  Num Calls    Avg (ns)       Med (ns)      Min (ns)     Max (ns)    StdDev (ns)            Name
 --------  ---------------  ---------  ------------  ------------  -----------  -----------  ------------  ----------------------
     38,0      395 316 157         32  12 353 629,0       4 545,0        4 152  394 692 967  69 768 987,0  cudaStreamCreate
     32,0      338 581 305      1 662     203 719,0     208 167,0        3 601      589 705      21 708,0  cudaEventSynchronize
     14,0      145 671 838     20 013       7 278,0       6 990,0        6 227      122 073       1 563,0  cudaLaunchKernel
     12,0      130 273 779          1 130 273 779,0 130 273 779,0  130 273 779  130 273 779           0,0  cudaDeviceReset
      0,0        9 380 625         31     302 600,0     168 420,0        7 449    1 371 734     362 875,0  cudaMemcpyAsync
      0,0        7 091 605         60     118 193,0      28 460,0        9 820      396 534     112 132,0  cudaMalloc
      0,0        1 887 604      1 662       1 135,0       1 094,0        1 012        5 359         278,0  cudaEventRecord
      0,0        1 753 883        100      17 538,0       4 686,0        2 475      100 730      25 698,0  cudaStreamSynchronize
      0,0        1 615 670          4     403 917,0      12 907,0       10 218    1 579 638     783 816,0  cudaHostAlloc
      0,0          265 924          1     265 924,0     265 924,0      265 924      265 924           0,0  cudaMemGetInfo
      0,0          221 719          1     221 719,0     221 719,0      221 719      221 719           0,0  cudaDeviceSynchronize
      0,0          160 501          2      80 250,0      80 250,0       48 790      111 711      44 491,0  cudaMemcpy
      0,0           10 149          1      10 149,0      10 149,0       10 149       10 149           0,0  cudaEventCreateWithFlags
      0,0            9 871          1       9 871,0       9 871,0        9 871        9 871           0,0  cuCtxSynchronize

Running [/work/djimenez/nvhpc_2022_221_Linux_x86_64_cuda_11.5/installdir/Linux_x86_64/22.1/profilers/Nsight_Systems/target-linux-x64/reports/gpukernsum.py lulesh_nvtx_nsys.sqlite]...

 Time (%)  Total Time (ns)  Instances  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)                                                  Name
 --------  ---------------  ---------  --------  --------  --------  --------  -----------  ----------------------------------------------------------------------------------------------------
     23,0      115 313 368      1 662  69 382,0  69 087,0    64 639    82 399      1 669,0  void CalcVolumeForceForElems_kernel<(bool)1>(const double *, const double *, const double *, const …
     21,0      105 293 684      1 662  63 353,0  63 295,0    60 800    70 751        987,0  CalcKinematicsAndMonotonicQGradient_kernel(int, int, double, const int *, const double *, const dou…
     17,0       85 702 095      1 662  51 565,0  51 519,0    50 495    53 568        457,0  AddNodeForcesFromElems_kernel(int, int, const int *, const int *, const int *, const double *, cons…
     10,0       48 267 260      1 662  29 041,0  29 535,0    24 608    32 319      1 661,0  ApplyMaterialPropertiesAndUpdateVolume_kernel(int, double, double, double, double *, double *, doub…
      9,0       46 178 077      1 662  27 784,0  27 776,0    25 408    32 192        887,0  CalcMonotonicQRegionForElems_kernel(double, double, double, double, double, int, int *, int *, int …
      7,0       35 653 435      1 662  21 452,0  21 472,0    17 568    23 552        649,0  CalcPositionAndVelocityForNodes_kernel(int, double, double, double *, double *, double *, double *,…
      3,0       17 058 300      1 662  10 263,0  10 240,0     7 968    12 416        518,0  CalcAccelerationForNodes_kernel(int, double *, double *, double *, double *, double *, double *, do…
      3,0       14 979 690      1 662   9 013,0   8 992,0     8 224    10 048        304,0  void CalcTimeConstraintsForElems_kernel<(int)128>(int, double, double, int *, double *, double *, d…
      1,0        8 848 254      4 986   1 774,0   1 728,0     1 375     3 488        241,0  ApplyAccelerationBoundaryConditionsForNodes_kernel(int, double *, int *)
      1,0        7 453 177      1 662   4 484,0   4 384,0     4 192     5 537        212,0  void CalcMinDtOneBlock<(int)1024>(double *, double *, double *, double *, int)
      0,0          115 680         40   2 892,0   2 304,0     1 281    11 712      2 370,0  void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__parallel_for::ParallelForAgent<thrus…
      0,0           36 542         17   2 149,0   1 664,0     1 088     8 928      1 885,0  void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__parallel_for::ParallelForAgent<thrus…
      0,0           23 744         12   1 978,0   1 920,0     1 888     2 207         94,0  void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__parallel_for::ParallelForAgent<thrus…

Running [/work/djimenez/nvhpc_2022_221_Linux_x86_64_cuda_11.5/installdir/Linux_x86_64/22.1/profilers/Nsight_Systems/target-linux-x64/reports/gpumemtimesum.py lulesh_nvtx_nsys.sqlite]...

 Time (%)  Total Time (ns)  Count  Avg (ns)   Med (ns)  Min (ns)  Max (ns)   StdDev (ns)      Operation
 --------  ---------------  -----  ---------  --------  --------  ---------  -----------  ------------------
     91,0        3 158 291     27  116 973,0  43 199,0     1 248  1 070 385    269 412,0  [CUDA memcpy HtoD]
      8,0          293 755      6   48 959,0  43 279,0     1 983    103 070     51 521,0  [CUDA memcpy DtoH]

Running [/work/djimenez/nvhpc_2022_221_Linux_x86_64_cuda_11.5/installdir/Linux_x86_64/22.1/profilers/Nsight_Systems/target-linux-x64/reports/gpumemsizesum.py lulesh_nvtx_nsys.sqlite]...

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)      Operation
 ----------  -----  --------  --------  --------  --------  -----------  ------------------
 20,338         27  0,753     0,500     0,000     4,001     1,005        [CUDA memcpy HtoD]
 3,204           6  0,534     0,541     0,000     1,061     0,578        [CUDA memcpy DtoH]
```

Figure 3.3: Summary statistics generated by NSight Systems for the LULESH-GPU application

3.1.1.1 NVIDIA NSight Systems NVTX Trace Reports

After a joint meeting with Hatchet and NVIDIA NSight System developers where the objective of this project was laid out, we were suggested using NVTX Push-Pop ranges annotations to mark regions of interest in the code for the profiler to monitor. By doing so, one particular NVIDIA-generated report could provide hierarchical data or a way to reconstruct an application's call path. At the time of implementing our solution, those reports hadn't yet shipped to the general public but we were granted early access to them. In particular the mentioned report is capable of generating a trace for NVTX annotated code, from here on referred to as the nvtxpptrace report. Code Listing 7 shows how the report is used to generate a `csv` file with the resulting trace information.

Listing 7 Command line used to generate NVTX trace report from NSight Systems-generated SQLite database

```
1  nsys stats --report nvtxpptrace --format=csv --output lulesh_nvtx_metrics
   ↪  lulesh_nvtx_metrics.sqlite
```

Figure 3.4 shows a sample of the results that the NVTX report generates. This is trace data with start and end timestamps for the different NVTX annotated ranges. Notice that each record of this trace provides a RangeId and a ParentId. Through timing data and this information, we can

reconstruct a tree representing caller-callee relationships for the executed application. This process will be described in Section 3.3.

```
Start (ns),End (ns),Duration (ns),DurChild (ns),DurNonChild (ns),Name,PID,TID,Lvl,NumChild,RangeId,ParentId,RangeStack,NameTree
353095791,10447707292,10094611501,10163561276,-68949775,main,30684,30684,0,8673,1,,:1,main
658788506,659047010,258504,0,258504,InitMeshDecomp,30684,30684,1,0,2,1,:1:2,--InitMeshDecomp
659052234,1173650599,514598365,0,514598365,NewDomain,30684,30684,1,0,3,1,:1:3,--NewDomain
1177992885,1178008817,15932,0,15932,cudaDeviceSetCacheConfig,30684,30684,1,0,4,1,:1:4,--cudaDeviceSetCacheConfig
1178066657,1182499401,4432744,4431394,1350,LagrangeLeapFrog,30684,30684,1,3,5,1,:1:5,--LagrangeLeapFrog
1178067175,1179145572,1078397,1075573,2824,LagrangeNodal,30684,30684,2,4,6,5,:1:5:6,----LagrangeNodal
1178067760,1179100567,1032807,1031040,1767,CalcForceForNodes,30684,30684,3,2,7,6,:1:5:6:7,------CalcForceForNodes
1178068194,1179087232,1019038,24692,994346,CalcVolumeForceForElems,30684,30684,4,2,8,7,:1:5:6:7:8,--------CalcVolumeForceForElems
1179061080,1179074538,13458,0,13458,CalcVolumeForceForElems_kernel,30684,30684,5,0,9,8,:1:5:6:7:8:9,----------CalcVolumeForceForElems_kernel
1179075349,1179086583,11234,0,11234,AddNodeFocesFromElems_kernel,30684,30684,5,0,10,8,:1:5:6:7:8:10,----------AddNodeFocesFromElems_kernel
1179088044,1179100046,12002,0,12002,TimeIncrement,30684,30684,4,0,11,7,:1:5:6:7:11,--------TimeIncrement
1179101487,1179111203,9716,9001,715,CalcAccelerationForNodes,30684,30684,3,1,12,6,:1:5:6:12,------CalcAccelerationForNodes
1179101958,1179110959,9001,0,9001,CalcAccelerationForNodes_kernel,30684,30684,4,0,13,12,:1:5:6:12:13,--------CalcAccelerationForNodes_kernel
1179111639,1179135611,23972,22367,1605,ApplyAccelerationBoundaryConditionsForNodes,30684,30684,3,3,14,6,:1:5:6:14,------ApplyAccelerationBoundaryConditionsForNodes
1179112159,1179120186,8027,0,8027,ApplyAccelerationBoundaryConditionsForNodes_kernel,30684,30684,4,0,15,14,:1:5:6:14:15,--------ApplyAccelerationBoundaryConditionsForNodes_kernel
1179120502,1179127024,6522,0,6522,ApplyAccelerationBoundaryConditionsForNodes_kernel,30684,30684,4,0,16,14,:1:5:6:14:16,--------ApplyAccelerationBoundaryConditionsForNodes_kernel
1179127372,1179135190,7818,0,7818,ApplyAccelerationBoundaryConditionsForNodes_kernel,30684,30684,4,0,17,14,:1:5:6:14:17,--------ApplyAccelerationBoundaryConditionsForNodes_kernel
1179136162,1179145240,9078,8415,663,CalcPositionAndVelocityForNodes,30684,30684,3,1,18,6,:1:5:6:18,------CalcPositionAndVelocityForNodes
1179136576,1179144991,8415,0,8415,CalcPositionAndVelocityForNodes_kernel,30684,30684,4,0,19,18,:1:5:6:18:19,--------CalcPositionAndVelocityForNodes_kernel
```

Figure 3.4: Sample of trace data for an NVTX-annotated version of LULESH-GPU

3.1.2 NVIDIA NSight Compute Kernel Specific GPU Metrics

Modern GPUs are complex hardware platforms and several multi-level factors can affect application performance. Although NSight Systems data could provide information on CPU-GPU interaction issues like excessive data movement or device synchronization, architecture specific data for GPU-kernel execution is lacking. During the development of this project, we decided on incorporating NSight Compute as an extra source of performance data that could give Hatchet users more data to work with during performance analysis.

Just like NSight Systems, NSight Compute is executed through the CLI `ncu` tool. NSight Compute provides two possible default outputs after data collection. An output `.ncu-rep` file is generated if the `-export/-o` parameter is used. If not specified, per-kernel data is printed out in

```
CalcAccelerationForNodes_kernel(int, double *, double *, double *, double *, double *, double *, double *), 2022-Aug-19 11:03:27, Context 1, Stream 7
  Section: GPU Speed Of Light Throughput
  ---------------------------------------------------------------------- --------------- ------------------------------
  DRAM Frequency                                                          cycle/usecond                         764,23
  SM Frequency                                                            cycle/nsecond                           1,08
  Elapsed Cycles                                                                  cycle                          4 259
  Memory [%]                                                                          %                           0,23
  DRAM Throughput                                                                     %                           0,23
  Duration                                                                      usecond                           3,94
  L1/TEX Cache Throughput                                                             %                           6,97
  L2 Cache Throughput                                                                 %                           0,19
  SM Active Cycles                                                                cycle                          67,81
  Compute (SM) [%]                                                                    %                           0,02
  ---------------------------------------------------------------------- --------------- ------------------------------
  WRN   This kernel grid is too small to fill the available resources on this device, resulting in only 0.0 full
        waves across all SMs. Look at Launch Statistics for more details.

  Section: Launch Statistics
  ---------------------------------------------------------------------- --------------- ------------------------------
  Block Size                                                                                                       128
  Function Cache Configuration                                                                 cudaFuncCachePreferNone
  Grid Size                                                                                                          2
  Registers Per Thread                                                  register/thread                             22
  Shared Memory Configuration Size                                                 byte                              0
  Driver Shared Memory Per Block                                             byte/block                              0
  Dynamic Shared Memory Per Block                                            byte/block                              0
  Static Shared Memory Per Block                                             byte/block                              0
  Threads                                                                        thread                            256
  Waves Per SM                                                                                                    0,00
  ---------------------------------------------------------------------- --------------- ------------------------------
```

Figure 3.5: Example of output segment from NSight Compute for one of LULESH's kernels

the CLI, as Figure 3.5 shows. Again, the `ncu-rep` is a proprietary and opaque data format, that can be opened with the associated GUI but, which for the sake of inputting data to Hatchet can not be used.

NSight Compute is able to extract a large amount of GPU metrics. The tool provides pre-defined groups of metrics referred to as *Sections* [71]. Each *section* conveys information about logically associated metrics, for example the `LaunchStats` *section* summarizes the grid and block size configuration used to launch a specific kernel. There are also metric *sets* which include one or more *sections*. When using the full *set* of *sections*, NSight Compute version 2021.3.0.0, provides 506 different GPU-specific metrics for each kernel. Table 3.1 shows a description of some of the main metrics *sections* gathered when profiling with the `detailed` set on NSight Compute.

Section Name	Description
Compute Workload Analysis	SM compute resources associate metrics. This includes achieved instructions per cycle (IPC) and pipeline utilization.
Instruction Stats	Information on assembly instructions used during execution, for example: Double-precision Fused Multiply Add, Single-precision Multiply, Non-coherent Global Memory Load. This hints at the most utilized pipelines in the GPU.
Launch Stats	Provides information of kernel launch configuration: kernel grid size, block size.
Memory Workload Analysis	Memory resources metrics like throughtput, cache level hit rates and achieved bandwidth.
Occupancy	Information on the ratio of active thread-warps per SM in relation to the maximum number of possible active warps the device is capable of. High occupancy might help hide memory latencies.
Scheduler Stats	Summarizes data on the warp scheduling level. This might provide information on stall-issues and resource idling.
Speed of Light	This section gives a high-level overview of compute and memory throughput. This is presented as an achieved percentage with respect to the theoretical maximum of the used platform.
Speed of Light Roofline	NSight Compute extracts metrics that can be used to construct a Roofline model of the executed kernels. This section contains the necessary metrics to construct Roofline charts.

Table 3.1: Main Sections from the detailed set analysis on NSight Compute

Unlike NSight Systems however, the `ncu` tool provides several parameters to control how performance data is aggregated and saved. Code Listing shows the specific command line we use to generate kernel metrics in a data format that can then be used to complement the hierarchical call path constructed from NSight Systems.

Listing 8 Command line used to generate kernel-specific metrics with NSight Compute

```
1  ncu --nvtx --print-summary=per-nvtx  --call-stack --print-kernel-base function --set
↪  detailed --page=raw --csv ./lulesh -s 200 > lulesh_ncu_metrics_200.csv
```

The `--nvtx` and `--print-summary=per-nvtx` options instruct the tool to filter the application kernels by NVTX Push/Pop ranges and to output the associated kernel metrics per NVTX Range context. This last feature is important so that a kernel that is invoked from two different calling contexts is reported twice (one per context) instead of aggregating metrics just by kernel name. This information is used in conjunction with the `--call-stack` parameter which makes the tool output the call-stack for each output kernel. This will later enable us to

apportion GPU-metrics to the correct kernel in the case of multiple context invocations. The `-set detailed` parameter specifies the metrics to be extracted by the tool, as described on Table 3.1.

Finally, NSight Compute enables the user to output the data in two different data-grouping formats. The `details` page format provides information in the format shown in Figure 3.5, while the `raw` page format provides a record-like description of each kernel and its resulting metrics. This parameters in hand with the `--csv` flag, result in a GPU kernel metrics report like the one Figure 3.6 shows.

```
"Range:PL_Type:PL_Value:CLR_Type:Color:Msg_Type:Msg","Kernel Name","Block Size","Grid Size","Device Id","Invocations","Metric Name","Metric Unit","Minimum","Maximum","Average"

" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","dram__bytes.sum.per_second","byte/second","1441340782.122905","2428571428.571429","2064056056.218551"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","dram__cycles_active.avg.pct_of_peak_sustained_elapsed","%","0.120552","0.360034","0.301536"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","dram__cycles_elapsed.avg.per_second","cycle/second","460893854.748603","1735930735.930736","678246949.533312"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","fbpa__dram_sectors.avg.pct_of_peak_sustained_elapsed","%","0.090245","0.268383","0.225710"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","gpc__cycles_elapsed.avg.per_second","cycle/second","649586824.953445","2449990981.240981","956272576.646230"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","gpc__cycles_elapsed.max","cycle","6717.000000","18872.000000","7656.132588"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","gpu__compute_memory_access_throughput.avg.pct_of_peak_sustained_elapsed","%","0.299208","8.157104","2.931747"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","gpu__compute_memory_request_throughput.avg.pct_of_peak_sustained_elapsed","%","0.318466","2.755886","1.943464"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","gpu__compute_memory_throughput.avg.pct_of_peak_sustained_elapsed","%","0.371699","8.157104","2.966963"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","gpu__dram_throughput.avg.pct_of_peak_sustained_elapsed","%","0.120552","0.360034","0.301536"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","gpu__time_duration.sum","nsecond","7232.000000","11456.000000","8007.421634"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","idc__request_cycles_active.avg.pct_of_peak_sustained_elapsed","%","0.000000","0.000000","0.000000"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","inst_executed","inst","8414.000000","8414.000000","8414.000000"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","l1tex__data_bank_reads.avg.pct_of_peak_sustained_elapsed","%","0.012719","0.035739","0.031820"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","l1tex__data_bank_writes.avg.pct_of_peak_sustained_elapsed","%","0.012719","0.035739","0.031820"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","l1tex__data_pipe_lsu_wavefronts.avg.pct_of_peak_sustained_elapsed","%","0.159124","0.447112","0.398082"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","l1tex__data_pipe_tex_wavefronts.avg.pct_of_peak_sustained_elapsed","%","0.000000","0.000000","0.000000"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","l1tex__f_wavefronts.avg.pct_of_peak_sustained_elapsed","%","0.015899","0.044674","0.039775"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","l1tex__lsu_writeback_active.avg.pct_of_peak_sustained_elapsed","%","0.030606","0.085997","0.076567"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","l1tex__lsuin_requests.avg.pct_of_peak_sustained_elapsed","%","0.136335","0.383079","0.341071"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","l1tex__m_l1tex2xbar_req_cycles_active.avg.pct_of_peak_sustained_elapsed","%","0.019344","0.054353","0.048393"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","l1tex__m_xbar2l1tex_read_sectors.avg.pct_of_peak_sustained_elapsed","%","0.033918","0.095304","0.084853"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","l1tex__t_sector_hit_rate.pct","%","0.000000","0.000000","0.000000"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","l1tex__tex_writeback_active.avg.pct_of_peak_sustained_elapsed","%","0.000000","0.000000","0.000000"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","l1tex__texin_sm2tex_req_cycles_active.avg.pct_of_peak_sustained_elapsed","%","0.016562","0.046535","0.041432"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","l1tex__throughput.avg.pct_of_peak_sustained_active","%","36.327889","40.711864","39.995385"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","launch__block_dim_x","block","1024.000000","1024.000000","1024.000000"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","launch__block_dim_y","block","1.000000","1.000000","1.000000"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","launch__block_dim_z","block","1.000000","1.000000","1.000000"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","launch__block_size","","1024.000000","1024.000000","1024.000000"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","launch__context_id","","1.000000","1.000000","1.000000"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","launch__device_id","","0.000000","0.000000","0.000000"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","launch__function_pcs","","139990146905344.000000","139990146905344.000000","139990146905344.000000"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","launch__grid_dim_x","","2.000000","2.000000","2.000000"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","launch__grid_dim_y","","1.000000","1.000000","1.000000"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","launch__grid_dim_z","","1.000000","1.000000","1.000000"
" ""main""  ""LagrangeLeapFrog""  ""CalcTimeConstraintsForElems""  ""CalcMinDtOneBlock"" ","CalcMinDtOneBlock","1024","2","0","7248","launch__grid_size","","2.000000","2.000000","2.000000"
```

Figure 3.6: Extract from an NSight Compute generated report

Notice that the first component of each row in this report is the call stack of the profiled kernel. Thus the `CalcMinDtOneBlock` kernel operation is reached through the `main` $\rightarrow$ `LagrangeLeapFrog` $\rightarrow$ `CalcTimeConstraintsForElems` $\rightarrow$ `CalcMinDtOneBlock` callpath. Each record in this file correspond to one specific kernel metric for which the tool reports its name, its units and the minimum, maximum and average values measured during execution. Furthermore, the report includes the `Device Id` which is a sequential identifier for the GPU on which the measure kernel was executed. This particular datum can be used to differentiate execution context when a multi-GPU analysis is being performed.

3.2 Annotating GPU-Accelerated Applications

Having determined how to use both NSight Systems and NSight Compute to generate both NVTX trace data and GPU-specific metrics per NVTX context, we needed to annotate each of the chosen applications with NVTX Push/Pop ranges. This process implies identifying important regions, in particular host functions and device kernels in each of the applications. As each of

these codes is based on different GPU-acceleration mechanisms there are some considerations that developers should take when analyzing this type of codes. A public code repository has been created to hold the different NVTX-annotated versions of each application and can be accessed under `https://gitlab.com/diegojv/nvtx_annotated_applications`.

3.2.1 Tensorflow Keras Model: GPU-Accelerated Framework considerations

The two most popular modern AI frameworks, Tensorflow and PyTorch, are Python-based. Each of these frameworks provide deep learning developers with high-level implementation abstractions that enable the development of complex neural network architectures without dealing with low-level hardware complexities. As such, execution details and GPU-acceleration are hidden from the programmer up to some degree. When implementing a model in any of these frameworks, no explicit CPU-GPU data transfers or kernel launches are specified. These details are handled backstage by the framework which greatly hinders our approach to Hatchet analysis of GPU-powered AI applications. The main reason being the complexity of capturing GPU-related events that happen at the framework layer with NVTX and obtaining GPU metrics with NSight Compute for such applications.

NVIDIA provides through their NVIDIA GPU Cloud (NGC) catalog [72] a set of software packages shipped as containers. These are NVIDIA-optimized applications, libraries and their required dependencies. An NVIDIA Tensorflow container is available and includes a mechanism to activate NVTX annotations inside the framework. However, the integration of a container inside the cluster environment used in this project resulted in non-conformant NVTX trace reports. This left a gap between a high-level NVTX trace for user specified code and the GPU-related metrics extracted with NSight Compute, as there is no data on the intermediate layer that handles data movement and kernel launches. Because of this, we decided on providing analysis capabilities based solely on the user-defined model, which could still hint at possible performance hot-spots in this sort of application. Code Listing 9 shows an extract of the annotated model we use in this project. The different stages of the application are identified by the developer, annotating those of particular interest. Tensorflow operations that execute on the GPU like the `model.fit` operation are fully synchronous and as such, the execution time reported by this NVTX Push/Pop range will correspond to the actual execution time of the GPU operation.

3.2.2 BS-SOLCTRA: OpenMP considerations

The Biot-Savart Solver for Computing and Tracing Magnetic Field Lines simulator implements a time-integration loop in which on each iteration, particle positions (x, y, z values) are updated.

Listing 9 Tensorflow Keras Model annotated with NVTX extract

```
1  import nvtx
2  import tensorflow as tf
3  from tensorflow import keras
4
5  @nvtx.annotate("main()")
6  def main():
7      ...
8      with nvtx.annotate("layers.Dense"):
9          dense1 = layers.Dense(num_units, activation='relu', name='dense_1')
10         ...
11         outputs = layers.Activation('linear', dtype='float32')(outputs)
12     with nvtx.annotate("keras.Model"):
13        model = keras.Model(inputs=inputs, outputs=outputs)
14     with nvtx.annotate("model.compile"):
15        model.compile(loss='sparse_categorical_crossentropy',
16                optimizer=keras.optimizers.RMSprop(),
17                metrics=['accuracy'])
18     with nvtx.annotate("mnist.load_data"):
19         ...
20     with nvtx.annotate("model.get_weights"):
21         initial_weights = model.get_weights()
22     with nvtx.annotate("model.fit"):
23         history = model.fit(x_train, y_train,
24                             batch_size=8192,
25                             epochs=10,
26                             validation_split=0.2)
27     with nvtx.annotate("model.evaluate"):
28         test_scores = model.evaluate(x_test, y_test, verbose=2)
```

As mentioned on Section 2.4.2, particle trajectories are completely independent so the process of updating each particle position can be performed completely in parallel. Recent implementations of this application perform said process using GPU-acceleration with OpenMP, using both prescriptive and descriptive approaches. We annotated both versions of this application with NVTX to determine performance bottlenecks and understand the main differences between both OpenMP programming approaches. Code Listing 10 shows the main portion of this application and the applied NVTX annotations. There are two important considerations when annotating OpenMP code for Hatchet analysis:

1. Data transfers with `target enter/exit data map` and OpenMP *target* regions are synchronous. This means again, that the reported time for those NVTX ranges is truly the time each data transfer and GPU kernel execution took.

2. When compiled, whatever code falls inside a `#pragma omp target` directive is converted into a GPU kernel, including functions that are called inside the *target* region. In the case of

this application, notice that on line 14, the computeIteration function is called. This is where the actual magnetic field and fourth-order Runge-Kutta methods are computed. However, because these functions are now device code, it is impossible to annotate them with NVTX. When extracting GPU performance metrics with NSight Compute, the metrics will correspond to the whole process that is encased in the *target* region.

Listing 10 NVTX-annotated OpenMP implementation of the particle trajectory computation. The two approaches are added to exemplify the difference

```
1    nvtxRangePushA("memcpyH2D");
2  #pragma omp target enter data map(to:coils[0:size_3D], e_r[0:size_3D],
   ↪  leng_segment[0:size_2D], particles[0:size_particles])
3    nvtxRangePop();
4              for (int i = 1; i <= steps; i++){
5                  nvtxRangePushA("runParticles_kernel");
6                  #pragma omp target teams distribute parallel for //Prescriptive version
7                  #pragma omp target teams loop //Descriptive version
8                  for(int p=0; p < particle_count ; p++){
9                      int base = p*DIMENSIONS;
10                     if((particles[base] == MINOR_RADIUS) && (particles[base+1] ==
                       ↪  MINOR_RADIUS) && (particles[base+2] == MINOR_RADIUS)){
11                         continue;
12                     }
13                     else{
14                         diverged = computeIteration(coils, e_r, leng_segment,
                           ↪  &particles[base], step_size, mode, divergenceCounter);
15                     }
16                 }
17                 nvtxRangePop();
18             }
19     nvtxRangePushA("memcpyD2H");
20     #pragma omp target exit data map(release:coils[0:size_3D], e_r[0:size_3D],
       ↪  leng_segment[0:size_2D], particles[0:size_particles])
21     nvtxRangePop();
22     nvtxRangePop();
```

3.2.3 LULESH: CUDA considerations

Before moving into the process of reconstructing hierarchical data for the profiled application, we had to also annotate the CUDA LULESH version with NVTX Push-pop ranges. To guide this process we relied on Hatchet's example data, in particular the Caliper [24] annotated results for LULESH. Caliper developers have created a public repository holding example annotated applications [73], one of them being the CPU version of the LULESH code Hatchet developers used for their example. Even though the application structure is not the same given the GPU optimizations on our version of LULESH, this Caliper annotated version of LULESH did provide a baseline as

to what regions of code where relevant and as such, we annotated following the same standard as Caliper for annotations.

```
static inline
void LagrangeLeapFrog(Domain* domain)
{
    nvtxRangePushA("LagrangeLeapFrog");
   /* calculate nodal forces, accelerations, velocities, positions, with
    * applied boundary conditions and slide surface considerations */
   LagrangeNodal(domain);

   /* calculate element quantities (i.e. velocity gradient & q), and update
    * material states */
   LagrangeElements(domain);

   CalcTimeConstraintsForElems(domain);

   nvtxRangePop();

}
```

(a) Host function with several function calls

```
static inline
void CalcAccelerationForNodes(Domain *domain)
{
    nvtxRangePushA("CalcAccelerationForNodes");
    Index_t dimBlock = 128;
    Index_t dimGrid = PAD_DIV(domain->numNode,dimBlock);

    nvtxRangePushA("CalcAccelerationForNodes_kernel");
    CalcAccelerationForNodes_kernel<<<dimGrid, dimBlock>>>
        (domain->numNode,
         domain->xdd.raw(),domain->ydd.raw(),domain->zdd.raw(),
         domain->fx.raw(),domain->fy.raw(),domain->fz.raw(),
         domain->nodalMass.raw());
    cudaDeviceSynchronize();
    nvtxRangePop();

    nvtxRangePop();
}
```

(b) Host function containing a GPU-kernel launch

Figure 3.7: Subset of NVTX annotated code regions in LULESH-GPU

Figure 3.7 shows two examples of the NVTX Push-Pop ranges annotations that were added to the LULESH-GPU code. In particular, Figure 3.7a shows a host function that contains several other host function calls. This relationship would then result in a hierarchy where the `LagrangeLeapFrog` function is the parent node of the `LagrangeNodal`, `LagrangeElements` and `CalcTimeConstraintsForElems` functions. Likewise, Figure 3.7b shows how NVTX annotations work for GPU-kernel launches. As NVTX is a host side library, the nvtxRange must be specified inside the host function launching the kernel.

Again, for our Hatchet analysis to make sense, there are some considerations when analyzing CUDA codes with NSight tools:

1. CUDA kernel launches are asynchronous, so once the *host* has posted the launch operation it continues with program execution. To obtain true timing information for a kernel when using NSight Systems, a `cudaDeviceSynchronize` operation should be added right after the kernel launch and inside the NVTX Push/Pop range. Figure 3.7b exemplifies this modification. However, constant device synchronization operations will add significant overhead to your total application so this should be used with caution and only when profiling. This same precaution should be taken with asynchronous data transfers.

2. CUDA supports the execution of multiple concurrent streams that could be used by an application to overlap data transfers with computation. However, the `nvtxpptrace` report

shipped by NVIDIA does not include a *"Stream ID"* data column that could be used to differentiate said streams. Because of this, reconstructing a hierarchy from a multiple stream application would be impossible and would result in incorrect hierarchical information.

3.3 Calling Context Trees Construction

Now that we have reviewed both the data generation process and the annotation of representative HPC applications we can proceed to the reconstruction of a GPU-accelerated application's calling context tree. Furthermore, we need to discuss the creation of a Hatchet-compliant data format that could enable developers to programmatically analyze NVIDIA NSight-derived performance data with Hatchet. Again, we will use LULESH as the guiding application to present this process.

3.3.1 NVTX Trace Processing

After executing the NSight Systems profiler over the newly NVTX annotated LULESH-GPU code and generating the SQLite database, we used the nvtxpptrace report provided by NVIDIA developers to generate a trace of NVTX events. Figure 3.4 shows a sample of the data that was obtained. Through the nvtxpptrace report we obtained a `csv` file containing all of the NVTX events that happened during program execution. Each record of this `csv` file contains data such as the start and end timestamps of the event, as well as the function name, *RangeId* and the *ParentId* that refers to the parent NVTX Push-Pop range that spawned the current function.

Given that data in this trace file is entirely sequential it would then be possible to reconstruct the hierarchy of function calls from this data, in particular the start and end timestamps. Based solely on the name, the usage of the ParentId column seems to be the obvious choice. However, this identifier does not correspond to the actual application call path relationship but is an identifier of the Push/Pop range. So for a given kernel, if called by the same function two different times, the *ParentId* on each case would differ, instead of pointing to the same parent node in a call path tree.

Figure 3.8 shows the process and data structures involved in reconstructing the call path tree from the NVTX trace for LULESH. A trace-parsing process is executed and stack structure is used to keep track of the current active function while a tree is built as the process is executed. Each tree node stores different information that helps identify it like the function name and a reference to its parent node. Each node keeps a list of its children which helps figure out if a function has already been processed before and if so, apportion the corresponding data accordingly. Each node

Start	End	Name	PID
1	12	main	30876
2	3	InitMeshDecomp	30876
4	11	LagrangeLeapFrog	30876
5	6	LagrangeNodal	30876
7	8	LagrangeElements	30876
9	10	CalcTimeConstraintsForElems	30876

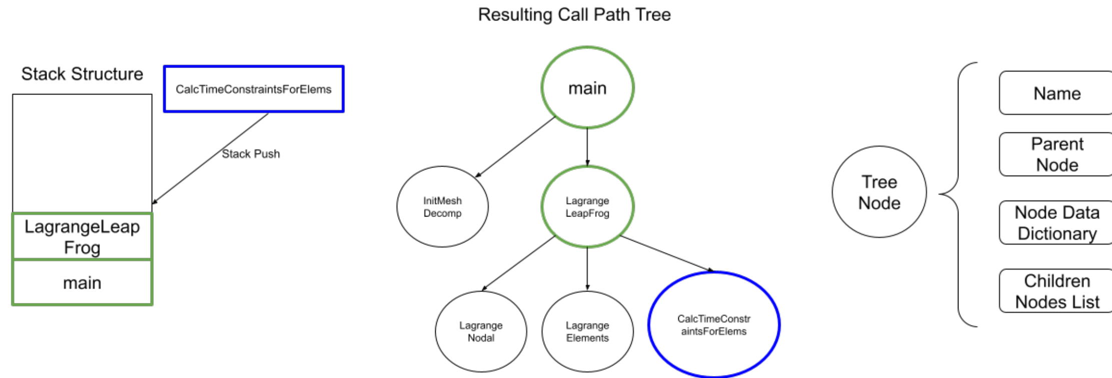


Figure 3.8: Call Path Tree construction from NVTX Trace process

also keeps a dictionary of function-related data like duration, process identifier and, as it will be discussed later on, GPU-related metrics.

The `csv` trace file is processed event by event according to the following procedure:

1. Check if the stack is empty
    a) If empty, push function into the stack and create tree node
    b) If not empty, verify if current start time is greater than the end time of the function at the top of the stack
        i. While the start time of the current function is greater than the end time of the function on the top of the stack: pop the function at the top of the stack. Once the start time is smaller than the end time of the function at the top of the stack: push the current function. Add current node to tree as node child of the function at the top of the stack

ii. If the start time of the current function is less than the end time of the function at the top of the stack: add current function as node child of the function at the top of the stack. Push the current function to to the top of the stack

As Figure 3.8 shows, when the `CalcTimeConstraintsForElems` event is processed, the only functions on the stack are `main` and `LagrageLeapFrog`. At this point, the start time for `CalcTimeConstraintsForElems` is less than the end time of `LagrageLeapFrog` which is the function at the top of the stack. As such, `CalcTimeConstraintsForElems` is added to the call path tree as a node child of `LagrangeLeapFrog`. At this point, `CalcTimeConstraints-ForElems` is pushed to the top of the stack and subsequent events are analyzed. If the following hypothetical event had a start time greater than the end time of `CalcTimeConstraintsForElems` then this function would be popped of the stack and the new event would become the new top of the stack. When no events remain to parse, the process is complete and the resulting CCT is complete. If no NSight Compute kernel metrics file is provided, the process finishes and a Hatchet-compliant file is created.

3.3.2 NSight Compute GPU Metrics Processing

If an NSight Compute GPU kernel metrics file is provided to the transformation script, then the next phase of the application begins. This next stage relies on the constructed CCT and the Python Pandas library to read in and pre-process the metrics file as a DataFrame. The following process is then applied. Initially, a list of unique kernels is extracted from the initialized DataFrame. This list is constructed by identifying each unique call path in the *"CallStack"* column. The following process is then applied:

1. Read-in metrics file and load it as a Pandas DataFrame (`df`)
2. Create a list of unique kernel call paths from the *"CallStack"* column (`df["CallStack"].unique()`)
3. For each kernel in the unique kernels list:
   a) Locate kernel function node (`kernelNode`) in CCT searching by name and by parent. This information is part of the *"CallStack"* column
   b) Subset the original DataFrame (`df`) extracting only the current kernel GPU metrics (`kernel_metrics`)

c) Add a new dictionary boolean item to `kernelNode`'s data dictionary, marking it as a GPU-kernel. This will help filter information in Hatchet once the data is loaded

d) For every metric in the `kernel_metrics` DataFrame:

   - Add metric and its average value as new items to `kernelNode`'s data dictionary

Figure 3.9 illustrates this process. The `CalcTimeConstraintsForElems_kernel` kernel is being processed as part of the unique kernel call path list. Its associated records in the original DataFrame (`df`) are subsetted and using the *"CallStack"* column, the node is located in the previously constructed CCT. Each metric is then added to the node's data dictionary by saving its name and average value.

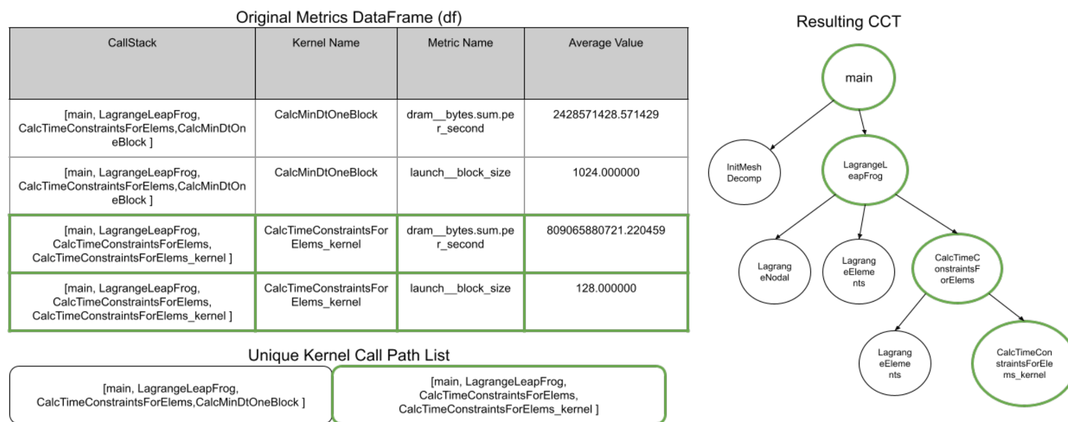


Figure 3.9: NSight Compute metrics file processing and kernel metrics apportioning

3.3.3 Multi-GPU Executions

It is possible to generate Multi-GPU performance data with NSight Systems. Using the Message Passing Interface, an application could utilize multiple *devices* simultaneously. Figure 3.10 illustrates how our transformation tool deals with such situations. In particular, how a multi-rank trace file is processed. The CCT construction process is identical to the one explained in Section 3.3.1. However, there's a previous step that must be executed and that is highlighted in Figure 3.10 through colors. Instead of simply reading in the input NVTX trace file, it is loaded as a Pandas DataFrame that then is used to filter the trace events by their *"PID"* column. Each MPI rank is a separate operating system process and thus, this column enables us to distinguish events based on

Start	End	Name	PID
1	12	main	30876
1.5	11	main	30877
2	3	InitMeshDecomp	30876
4	11	LagrangeLeapFrog	30876
5	6	InitMeshDecomp	30877
7	8	LagrangeElements	30876
9	10	LagrangeLeapFrog	30877

Figure 3.10: Multiple CCTs are created when Multi-GPU trace data is provided

their process identification number. The CCT process is executed for each unique *"PID"* present in the NVTX trace report.

3.3.4 Hatchet Compliant Data Format

One of the main objectives of this projects was generating a Hatchet-compliant data format that could be used to store the reconstructed CCT and its associated metrics. However, once we had reconstructed the hierarchy and analyzed our options, we decided on using a JSON file that follows the same format that Caliper-data uses to be read into Hatchet. This decision enables us to reuse the Caliper Hatchet reader and will facilitate creating a new Hatchet NSight reader based of this existing implementation.

This file is composed of a high-level JSON object that is made up of four different key-value fields: *i)* data, *ii)* columns, *iii)* column_metadata and *iv)* nodes. Code Listing 11 illustrates these components with a subset of LULESH's function calls. The nodes key stores the different function names and their hierarchical relationships through the parent element. For each of these nodes, a record in the data JSON key is stored. These data-records are ordered sequentially following the

nodes key ordering. The columns key is a list of the names of each component of the data values. Finally, the column_metadata key stores a boolean indicating how a data value should be interpreted when loaded into Hatchet.

Listing 11 Example JSON file created with our implemented transmogrifying tool


```
{
 "data": [
  [9,0,9,null],
  [138.472091324,0,-5.285117875,0],
  [2.5975e-05,0,2.5975e-05,1],
  [1.779597704,0,1.779597704,2],
  [1.6283e-05,0,1.6283e-05,3],
  [0.023146529,0,7.65e-07,4],
  [0.003685577,0,1.185e-06,5],
  [0.003642114,0,6.34e-07,6],
  [0.003627609,0,0.00360372,7],
  [1.436e-05,0,1.436e-05,8],
  [9.529e-06,0,9.529e-06,9]
 ],
 "columns": ["inclusive#sum#time.duration","mpi.rank","sum#time.duration","path"],
 "column_metadata": [{"is_value": true},{"is_value": true},{"is_value": true},{"is_value":
 ↪  false}
 ],
 "nodes": [{"column": "path","label": "main"},{"column": "path","label":
 ↪  "InitMeshDecomp","parent": 0},
  {"column": "path","label": "NewDomain","parent": 0},{"column": "path","label":
  ↪  "cudaDeviceSetCacheConfig","parent": 0},
  {"column": "path","label": "LagrangeLeapFrog","parent": 0},{"column": "path","label":
  ↪  "LagrangeNodal","parent": 4},
  {"column": "path","label": "CalcForceForNodes","parent": 5},{"column": "path","label":
  ↪  "CalcVolumeForceForElems","parent": 6},
  {"column": "path","label": "CalcVolumeForceForElems_kernel","parent": 7},{"column":
  ↪  "path","label": "AddNodeFocesFromElems_kernel","parent": 7
  }
 ]
}
```


By following this transformation process we were able to process input NVIDIA NSight performance data and generate hierarchical profile data for GPU accelerated applications from diverse domains. The resulting CCTs are exported in a Hatchet-compliant JSON file which can be easily loaded into this library by means of an already existing reader. In the following section we discuss our main results and review how through this newly added functionality, Hatchet users are now capable of processing and analyzing GPU performance data extracted with NVIDIA performance tools.

Chapter 4

Results

The final challenge in this project was determining whether or not through our NVIDIA NSight performance data transformation pipeline, Hatchet users can really identify performance bottlenecks in GPU-accelerated applications. We have reviewed the performance data collection process and how through an NVTX-annotation approach, we manage to reconstruct Calling Context Trees from traces and add GPU-specific metrics to kernel functions in those hierarchical structures. In this section we discuss the main results of this project.

4.1 Experimental Setup

The development of this project and the performance data collection process for all three applications was carried out in two different HPC platforms. Experimental data for single-GPU runs was obtained from Kabré HPC system [74] at the Costa Rica National High Technology Center (CeNAT). Multi-GPU experiments were done using the Raven HPC system [75] at the Max Planck Computing and Data Facility (MPCDF).

- **CeNAT's Kabré HPC system**: single-GPU performance data capturing was carried out using the *Nukwä* partition. This partition is composed of 8 single-GPU nodes. Four of these nodes rely on Tesla K40 NVIDIA GPUs while the other four nodes are powered by Tesla V100 NVIDIA GPUs. We specifically used the latter nodes. Each of these nodes has an Intel Xeon Silver 4214R CPU host processor connected through PCI-Express to one Tesla V100 GPU with 32 GB HBM2.

- **MPCDF's Raven HPC system**: Raven is comprised of 1592 compute nodes powered by Intel Xeon IceLake-SP Processors Platinum 8360Y. Additionally, there are 192 GPU-accelerated nodes, each one with the same Intel Xeon IceLake-SP CPU host connected to 4 Ampere A100-SXM4 NVIDIA GPUs (40 GB HBM2).

In terms of the software environment, Table 4.1 gives a description of the base software stack used on each of the systems. On the following sections, a description of the specific software requirements and compilation commands for each application will be given.

Software Component	**Kabré**	**Raven**
O.S	CentOS Linux 7	SUSE Linux Enterprise Server 15-SP3
Compiler	nvcc v 11.5.119	nvcc v 11.7.64
NSight Systems	2021.5.1.118-f89f9cd	2022.2.1.31-5fe97ab
NSigt Compute	2021.3.0.0	2022.2.0.0

Table 4.1: Base software stack on both testbed systems

4.2 Loading Reconstructed CCTs with Hatchet

Wrapping up the on-going example we used in Chapter 3, the final step was validating that our generated CCTs and output JSON files could be loaded into Hatchet for their manipulation. Code Listing 12 shows the process we use to read the hierarchical performance data. Figure 4.1 shows the resulting output for Hatchet commands on lines 11 and 12.

Listing 12 NSight performance data transformation process and JSON hierarchy file loading in Hatchet

```
1  # 1. Apply transformation script, feeding in NSight Systems trace and optional NSight
   ↪  Compute metrics file
2
3  $ python src/transmogrifier.py -t input_files/LULESH_200_nvtx_trace_nvtxpptrace.csv -m
   ↪  input_files/LULESH_200_metrics.csv
4
5  # 2. Using the resulting JSON file, open an interactive Python session and load it using
   ↪  Hatchet
6  $ python
7
8  >>> import hatchet as ht
9  >>> filename = 'hierarchy.json'
10 >>> gf = ht.GraphFrame.from_caliper_json(filename)# Use Caliper reader to load JSON file
11 >>> print(gf.tree(metric_column='time (inc)')) # Display CCT using inclusive time column
12 >>> gf.dataframe # Display dataframe
```

Given that we rely on the same data format the Hatchet Caliper reader, no extra code changes were necessary to load in the data. Figure 4.1a shows the resulting call path for LULESH. This resulting call path was validated against the Caliper annotated version for the CPU version of LULESH [73] in terms of structure. However, this is not a 1:1 mapping given the code optimizations made for the GPU version we are profiling. Some time discrepancies can also be observed in the total time integration for the main function, however these are derived from no-NVTX annotated code regions. Figure 4.1 also shows the associated dataframe including GPU-specific metrics from NSight Compute

(a) LULESH Call Path Tree created from NSight Systems NVTX trace data

node	rank	time (inc)	time	nid	...	smsp__thread_inst_executed_per_inst_executed.ratio	smsp__thread_inst_executed_pred_on_per_inst_executed.ratio	name
					...			
{'name': 'main', 'type': 'region'}	0	141.846032	-3.267699e+00	0	...	NaN	NaN	main
{'name': 'InitMeshDecomp', 'type': 'region'}	0	0.000023	2.253800e-05	1	...	NaN	NaN	InitMeshDecomp
{'name': 'LagrangeLeapFrog', 'type': 'region'}	0	129.523025	-1.088100e-05	4	...	NaN	NaN	LagrangeLeapFrog
{'name': 'CalcTimeConstraintsForElems', 'type':...	0	0.115609	8.329033e-03	25	...	NaN	NaN	CalcTimeConstraintsForElems
{'name': 'CalcMinDtOneBlock', 'type': 'region'}	0	0.044882	2.405500e-05	28	...	31.889470	10.951509	CalcMinDtOneBlock
{'name': 'CalcTimeConstraintsForElems_kernel', ...	0	0.040234	1.111800e-05	27	...	31.737214	30.866325	CalcTimeConstraintsForElems_kernel
{'name': 'cudaFuncSetCacheConfig', 'type': 'reg...	0	0.007292	6.393000e-06	26	...	NaN	NaN	cudaFuncSetCacheConfig
{'name': 'LagrangeElements', 'type': 'region'}	0	0.156963	1.186214e-02	18	...	NaN	NaN	LagrangeElements
{'name': 'ApplyMaterialPropertiesAndUpdateVolum...	0	0.044514	7.020000e-07	23	...	NaN	NaN	ApplyMaterialPropertiesAndUpdateVolume
{'name': 'ApplyMaterialPropertiesAndUpdateVolum...	0	0.040679	1.303500e-05	24	...	30.817324	29.474956	ApplyMaterialPropertiesAndUpdateVolume_kernel
{'name': 'CalcKinematicsAndMonotonicQGradient',...	0	0.048063	1.661000e-06	19	...	NaN	NaN	CalcKinematicsAndMonotonicQGradient
{'name': 'CalcKinematicsAndMonotonicQGradient_k...	0	0.042829	1.791400e-05	20	...	31.964988	31.693136	CalcKinematicsAndMonotonicQGradient_kernel
{'name': 'CalcMonotonicQRegionForElems', 'type'...	0	0.045104	8.070000e-07	21	...	NaN	NaN	CalcMonotonicQRegionForElems
{'name': 'CalcMonotonicQRegionForElems_kernel',...	0	0.041161	1.910700e-05	22	...	31.469929	30.671609	CalcMonotonicQRegionForElems_kernel
{'name': 'LagrangeNodal', 'type': 'region'}	0	129.243582	1.385300e-05	5	...	NaN	NaN	LagrangeNodal
{'name': 'ApplyAccelerationBoundaryConditionsFo...	0	0.128595	1.359000e-06	14	...	NaN	NaN	ApplyAccelerationBoundaryConditionsForNodes
{'name': 'ApplyAccelerationBoundaryConditionsFo...	0	0.118376	8.665000e-06	15	...	31.994064	27.423484	ApplyAccelerationBoundaryConditionsForNodes_ke...
{'name': 'CalcAccelerationForNodes', 'type': 'r...	0	0.057818	6.440000e-07	12	...	NaN	NaN	CalcAccelerationForNodes
{'name': 'CalcAccelerationForNodes_kernel', 'ty...	0	0.053576	1.463900e-05	13	...	31.999977	30.358946	CalcAccelerationForNodes_kernel
{'name': 'CalcForceForNodes', 'type': 'region'}	0	129.007400	1.135000e-06	6	...	NaN	NaN	CalcForceForNodes
{'name': 'CalcVolumeForceForElems', 'type': 're...	0	0.094072	4.345920e-03	7	...	NaN	NaN	CalcVolumeForceForElems
{'name': 'AddNodeFocesFromElems_kernel', 'type'...	0	0.040707	2.538600e-05	9	...	31.795387	30.613812	AddNodeFocesFromElems_kernel
{'name': 'CalcVolumeForceForElems_kernel', 'typ...	0	0.042308	2.397600e-05	8	...	32.000000	31.691561	CalcVolumeForceForElems_kernel
{'name': 'TimeIncrement', 'type': 'region'}	0	128.907014	6.345000e-06	10	...	NaN	NaN	TimeIncrement
{'name': 'cudaEventSynchronize', 'type': 'region'}	0	128.892709	9.214000e-06	11	...	NaN	NaN	cudaEventSynchronize
{'name': 'CalcPositionAndVelocityForNodes', 'ty...	0	0.042141	7.230000e-07	16	...	NaN	NaN	CalcPositionAndVelocityForNodes
{'name': 'CalcPositionAndVelocityForNodes_kerne...	0	0.039067	1.204100e-05	17	...	31.999976	30.666638	CalcPositionAndVelocityForNodes_kernel
{'name': 'NewDomain', 'type': 'region'}	0	2.346042	2.346042e+00	2	...	NaN	NaN	NewDomain
{'name': 'checkErrors', 'type': 'region'}	0	0.001507	8.780000e-07	29	...	NaN	NaN	checkErrors
{'name': 'cudaDeviceSetCacheConfig', 'type': 'r...	0	0.000031	3.112200e-05	3	...	NaN	NaN	cudaDeviceSetCacheConfig

(b) LULESH's GraphFrame DataFrame component including GPU-specific metrics

Figure 4.1: Resulting CCT for LULESH loaded into Hatchet

In the following sections we discuss three different case studies that levarage this newly added Hatchet functionality. For all three case studies and different versions of the code, we apply that was just described to transform NSight performance data into CCTs that can be loaded into Hatchet.

4.3 Identifying Bottlenecks with Hatchet: Tensorflow Keras Model Case Study

At its most basic level, the possibility of visualizing the hierarchical structure of an application and the associated function times enable developers to identify the most time-consuming sections of an application and understand the impact of performance optimization modifications. In this case study, the Tensorflow Keras model helps us illustrate how programmers can now follow this analysis-driven optimization process for GPU-accelerated applications.

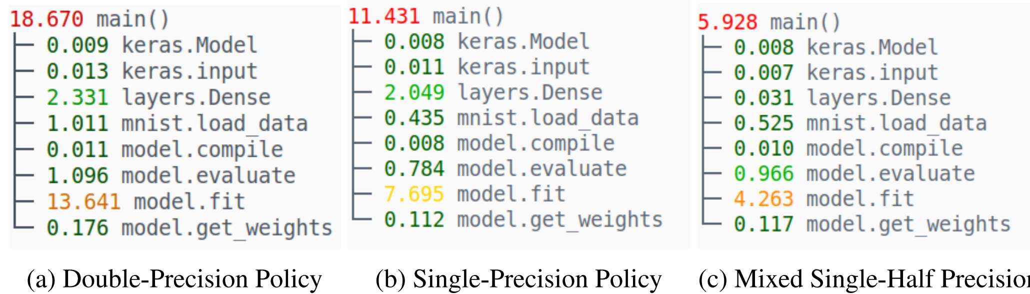
Modern NVIDIA GPUs are capable of executing floating operations in different precision levels. This type of functionality is particularly useful for large deep learning models that require more memory and compute resources to train. Numerous studies have proven that relying on lower-precision computation and data representation can incur little degradation in resulting classification accuracy [76, 77]. By relying on Hatchet analysis, we can quantify the impact on performance of varying the precision policy used across the layers of the model. The Keras API enables programmers to use mixed precision (float16 and float32) operations across the network. Code Listing 13 shows how we control the precision policy on our model to analyze performance.

Listing 13 Extract from the Tensorflow Keras model showing how we can control the precision policy

```
1  import nvtx
2  import tensorflow as tf
3  from tensorflow import keras
4  from tensorflow.keras import layers
5  from tensorflow.keras import mixed_precision
6
7  print("TensorFlow version:", tf.__version__)
8  policy = tf.keras.mixed_precision.experimental.Policy('mixed_float16')
9  tf.keras.mixed_precision.experimental.set_policy(policy)
```

We tested three different levels of precision, starting from `float64` for both layer computation and the type of the variables. Figure 4.2a shows the CCT for an execution of the Tensorflow Keras model based on double-precision computation and data. Hatchet's tree view allowed us to quickly recognize that the training phase of our model was the most time consuming section.

As such, we then tested both single-precision (`float32`) and a mix of single and half precision (`mixed_float16`) for the layer computations. Variables were kept on float32 in both cases for numeric stability. Figures 4.2b and 4.2c show the performance difference as we reduced the computation precision for the different layers in the model.



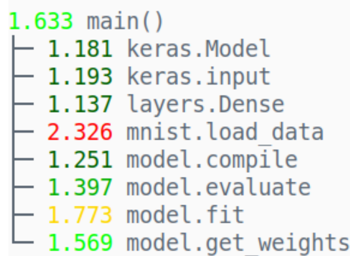
(a) Double-Precision Policy (b) Single-Precision Policy (c) Mixed Single-Half Precision

Figure 4.2: Execution profile comparison for the three different precision policies tested on the Tensorflow Keras model

Hatchet's programmatic capabilities allow us to easily quantify the achieved speedups per optimization. To do so, we can use Hatchet's GraphFrame algebra to compute the division of two GraphFrames. Then, it is possible to show the execution profile using the achieved speedup per function as measure. An alternative method is using the DataFrame components an apply the same division operation to create a new data column called Speedup. Both processes are shown in Code Listing 14 and Figure 4.3 shows the resulting outputs.

Listing 14 Computing the achieved speedup per function when moving from double-precision to single-precision

```
1  >>> gf_64 = ht.GraphFrame.from_caliper_json("keras_64.json")
2  >>> gf_32 = ht.GraphFrame.from_caliper_json("keras_32.json")
3
4  # Graphically displaying speedups
5  >>> gf_speedup_32 = gf_64/gf_32
6  >>> print(gf_speedup_32.tree(metric_column="time (inc)"))
7
8  # Creating a Speedup column in the dataframe is an alternative method
9  >>> gf_64.dataframe['Speedup_32'] = gf_64.dataframe['time
   ↪  (inc)'].div(gf_32.dataframe['time (inc)'])
10 >>> sorted_df = gf_64.dataframe.sort_values(by=['Speedup_32'], ascending=False)
11 >>> print(sorted_df[["name","Speedup_32"]])
```



(a) Graphical Speedup Profile

node	rank	name	Speedup_32
{'name': 'mnist.load_data', 'type': 'region'}	0	mnist.load_data	2.326408
{'name': 'model.fit', 'type': 'region'}	0	model.fit	1.772775
{'name': 'main()', 'type': 'region'}	0	main()	1.633289
{'name': 'model.get_weights', 'type': 'region'}	0	model.get_weights	1.569385
{'name': 'model.evaluate', 'type': 'region'}	0	model.evaluate	1.397375
{'name': 'model.compile', 'type': 'region'}	0	model.compile	1.250544
{'name': 'keras.input', 'type': 'region'}	0	keras.input	1.192574
{'name': 'keras.Model', 'type': 'region'}	0	keras.Model	1.180777
{'name': 'layers.Dense', 'type': 'region'}	0	layers.Dense	1.137284

(b) DataFrame representation with newly added Speedup_32 column

Figure 4.3: Programmatic computation of Speedups in Hatchet

4.4 Data Analytics on GPU Performance Metrics: BS-SOLCTRA Case Study

Now that we have seen that the ability to load NSight performance data on Hatchet effectively allows us to identify performance bottlenecks, we will review a case study in which we go deeper and use GPU-specific metrics to determine the cause of an overhead. We will then discuss how our implementation can also help analyze multi-GPU executions.

4.4.1 Descriptive Implementation Performance Overhead

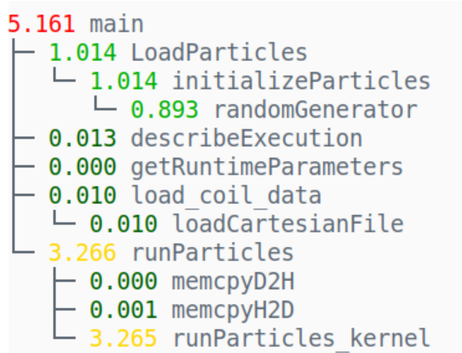
As mentioned in Section 2.4.2, the BS-SOLCTRA simulator has been recently ported to OpenMP for GPU-acceleration. Two different versions have been implemented to test the difference between *prescriptive* and *descriptive* approaches. Under the prescriptive approach the programmer must identify the *target* parallel regions and specify how the computation must be distributed across *teams* and *threads*. The *descriptive* approach requires the programmer identifying the parallelizable *target* region, but leaves to the compiler the decision of how to map the computation to *teams* and *threads*. Code Listing 10 in Section 3.2.2, shows the different pragmas used by both approaches.

Listing 15 Compilation command used by both prescriptive and descriptive implementations of BS-SOLCTRA

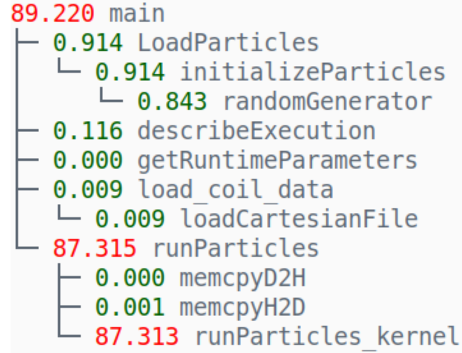
```
1 $ nvc++ -O3 -mp=gpu -gpu=pinned,fastmath -Minfo=mp -o bs-gpu solctra_multinode.cpp
  ↪  main_multinode.cpp utils.cpp -lnvToolsExt
```

Code Listing 15 shows how both versions of the application are compiled. Furthermore, all BS-SOLCTRA executions reported in this section were carried out using a problem size of 102 400 particles and 1000 iteration steps. However, a considerable performance overhead is

measured for the *descriptive* implementation. Figure 4.4 shows the Hatchet profiles for both versions of the application. Notice that a an approximate 27× slowdown is measured for the `runParticles_kernel` GPU operation, being this also the most time consuming operation in the application. This suggests that there is some issue with the compiler parallel distribution decisions.



(a) Prescriptive implementation execution profile



(b) Descriptive implementation execution profile

Figure 4.4: Execution profile comparison for both versions of BS-SOLCTRA

4.4.2 Comparing Implementations through GPU Metrics

Unlike the Tensorflow Keras model, there is no framework-provided automatic optimization techniques we can test with BS-SOLCTRA. This is the type of scenario where NSight Compute GPU performance metrics come into play. Code Listing 16 shows how through Hatchet's programmatic operations, we can easily compare the metrics for the two implementations.

We start by loading each hierarchical profile from the generated JSON files. Then, because this application only has one GPU kernel operation, we have two possible ways to filter the DataFrames and extract only the kernel-related row. We can rely on the *"isGPUKernel"* column that was added during the transformation process to indicate a node has GPU metrics data (lines 9 and 10 in Code Listing 16). Alternatively, when there are multiple kernels in an application, filtering can be performed based on other columns like the kernel name (lines 13 and 14). Now, as we are trying to pin-point the main differences in the metrics to determine what could be causing the slowdown, we can use the Pandas compare operation to generate a new DataFrame that includes both versions metrics side by side. This is precisely what we

are doing in line 16. Then we can query for specific metrics to visualize the difference between both implementations. In this particular case, we query the difference DataFrame for the `smsp__sass_thread_inst_executed_op_dfma_pred_on.sum.per_cycle_elapsed` metric. This value describes the total amount of double precision fused-multiply operations that are executed per cycle and we noticed that the descriptive version is executing approximately 27 times less instructions per cycle than the prescriptive implementation.

Listing 16 Comparing GPU-kernel metrics for both BS-SOLCTRA implementations

```
1   >>> import hatchet as ht
2   >>> filename_slow = "hierarchy-bs-gpu-descriptive-slow.json"
3   >>> filename_prescriptive = "hierarchy-bs-gpu-prescriptive.json"
4   # Load NSight performance data as GraphFrames
5   >>> gf_slow = ht.GraphFrame.from_caliper_json(filename_slow)
6   >>> gf_prescriptive = ht.GraphFrame.from_caliper_json(filename_prescriptive)
7
8   # Filter each dataframe to extract only the relevant kernel data
9   >>> kernel_slow = gf_slow.dataframe.loc[gf_slow.dataframe["isGPUKernel"]==True]
10  >>> kernel_prescriptive =
    ↪   gf_prescriptive.dataframe.loc[gf_prescriptive.dataframe["isGPUKernel"]==True]
11
12  # Alternative filtering based on kernel name
13  >>> kernel_slow = gf_slow.dataframe.loc[gf_slow.dataframe["name"]=="runParticles_kernel"]
14  >>> kernel_prescriptive =
    ↪   gf_prescriptive.dataframe.loc[gf_prescriptive.dataframe["name"]=="runParticles_kernel"]
15
16  >>> difference = kernel_prescriptive.compare(kernel_slow)
17  >>> difference["smsp__sass_thread_inst_executed_op_dfma_pred_on.sum.per_cycle_elapsed"]
18                                                              self      other
19  node                                                  rank
20  {'name': 'runParticles_kernel', 'type': 'region'} 0     800.90085  29.609158
21
22  >>> difference["launch__grid_size"]
23                                                              self     other
24  node                                                  rank
25  {'name': 'runParticles_kernel', 'type': 'region'} 0     800.0    102400.0
26
27  >>> difference["launch__block_size"]
28                                                              self    other
29  node                                                  rank
30  {'name': 'runParticles_kernel', 'type': 'region'} 0     128.0     1.0
```

The difference in operations per cycle hints at a difference in the way the kernel was launched in each implementation. Thus, we confirm this suspicion by querying both the `launch__grid_size` and `launch__block_size` dimensions. As we can see, the *descriptive* implementation is launching the kernel with 102 400 OpenMP *teams* and only one thread per team. The *prescriptive* implementation on the other hand, initializes a total of 800 OpenMP *teams* each with 128 threads. The total amount of threads in both cases is 102 400, however, there's a semantic difference and

more importantly, this distinction does have an effect in how threads are scheduled for execution. Although this code is developed with OpenMP, it executes on top of the CUDA execution model, in which each Streaming Multiprocessor schedules threads in groups of 32, called *warps*. This basic execution unit enables the model to hide memory access latencies by scheduling a different *warp* in an SM when then current *warp* is stalled by a memory access. However, as the *descriptive* implementation is scheduling only one thread per *team* (each one mapped to a different SM), the system is presumably unable to hide memory access latencies and as such, when stalled, numerous hardware resources are unused on every cycle. The *prescriptive* implementation, on the other hand, has 4 possible warps per SM and is capable of utilizing more resources and thus achieving a high instructions per cycle rate.

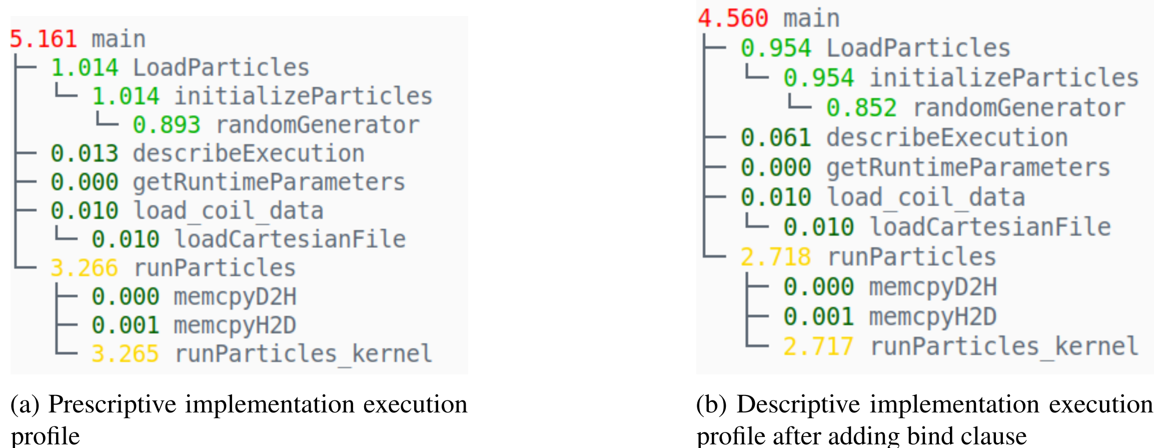
Having identified this crucial difference, we decided on giving an extra hint to the compiler in the *descriptive* implementation. The OpenMP standard [40] includes the `bind` clause that enables programmers to provide extra directions as to how to distribute iterations of a loop. Code Listing 17, line 6, shows how the kernel specifying `pragma` was modified to change its launching configuration.

Listing 17 OpenMP bind clause added to descriptive implementation to change kernel launch configuration

```
1   ...
2           for (int i = 1; i <= steps; i++){
3               nvtxRangePushA("runParticles_kernel");
4               #pragma omp target teams loop bind(teams, parallel)
5               for(int p=0; p < particle_count ; p++){
6                   int base = p*DIMENSIONS;
7                   if((particles[base] == MINOR_RADIUS) && (particles[base+1] ==
                    ↪  MINOR_RADIUS) && (particles[base+2] == MINOR_RADIUS)){
8                       continue;
9                   }
10                  else{
11                      diverged = computeIteration(coils, e_r, leng_segment,
                        ↪  &particles[base], step_size, mode, divergenceCounter);
12                  }
13              }
14              nvtxRangePop();
15          }
16  ...
17  nvtxRangePop();
```

Again, we can use Hatchet's graphical representation to understand the effect of a performance optimization code change. Figure 4.5 shows the resulting performance profile for the *descriptive* implementation with the newly added `bind` clause. Execution time for the *descriptive* implementation

was accelerated approximately by a 27× factor. Even more, it now seems to be slightly faster than the *prescriptive* implementation, presumably due to some compiler optimization. This difference is out of the scope of this project but following the same GPU metric comparison procedure, some reason might be discovered for it.



(a) Prescriptive implementation execution profile

(b) Descriptive implementation execution profile after adding bind clause

Figure 4.5: Execution profile comparison for both versions of BS-SOLCTRA after optimizing descriptive implementation

4.4.3 Multi-GPU Performance Analysis

The BS-SOLCTRA simulator allowed us to test another of Hatchet functionalities: understanding load imbalance across parallel process, in our case multi-GPU executions. This simulator, in its *prescriptive* implementation, is capable of using MPI processes to distribute the input particles across GPUs to accelerate the computation. For this case study, we relied on the Raven HPC system at MPCDF. As described at the start of this chapter, each Raven node has 4 NVIDIA A100 GPUs connected to the host. Thus, we use OpenMPI version 4.0.7 in conjunction with the NVIDA compiler to build this version of the application. Code listing 18 shows the compilation command used on Raven for this version of the code.

Listing 18 Compilation command used by multi-GPU prescriptive implementation of BS-SOLCTRA

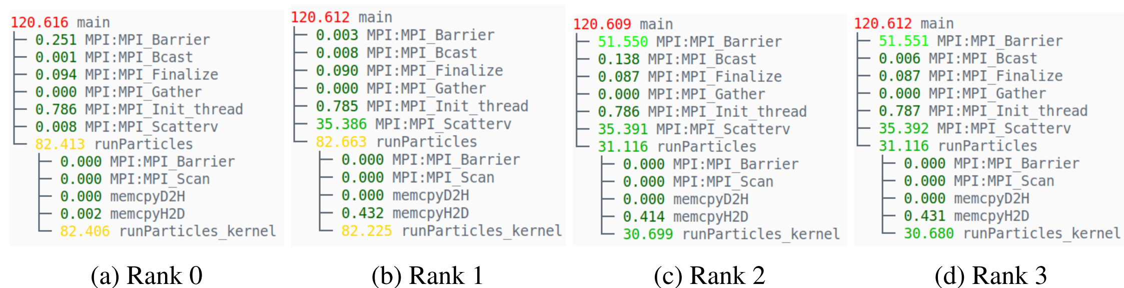
```
1   $ nvc++ -O3 -mp=gpu -gpu=pinned,fastmath -Minfo=mp -I$(MPI_DIR)/include -o bs-mpi-gpu
    ↪  solctra_multinode.cpp main_multinode.cpp utils.cpp -lmpi -lnvToolsExt
```

Before loading any data into Hatchet, we artificially injected a load imbalance scenario into the simulation. This imbalance is achieved by redistributing the load of the parallel processes according

to Equation 4.1. So for a 4 GPU execution and a problem size of 1 024 000 particles and 1000 iterations, ranks 0 and 1 will each compute 384 000 particles, while ranks 2 and 3 will be responsible of 128 000 particles each.

$$rankShare = \begin{cases} \frac{totalParticles}{numberRanks} + 0.5 * \frac{totalParticles}{numberRanks}, & \text{if } rank < \frac{numberRanks}{2} \\ \frac{totalParticles}{numberRanks} - 0.5 * \frac{totalParticles}{numberRanks}, & \text{otherwise} \end{cases} \tag{4.1}$$

Figure 4.6 shows execution profile for each MPI rank. Notice that both ranks 0 and 1 have considerably higher execution times for the `runParticles_kernel` routine. This was to be expected because of the injected imbalance which amounts to a factor of 3 times as much work.



(a) Rank 0 (b) Rank 1 (c) Rank 2 (d) Rank 3

Figure 4.6: Execution profiles for each MPI rank in a multi-GPU BS-SOLCTRA execution

Now, having loaded the different profiles into Hatchet *GraphFrames*, we can use this tool's load imbalance computing methodology. Code listing 19 shows the first stage of this process for the 4 GPU execution.

Listing 19 Index dropping on multi-GPU DataFrame, computing both the mean and maximum execution time for each node across all processes

```
1  >>> import hatchet as ht
2  >>> filename = "hierarchy-multi-gpu.json"
3  >>> gf = ht.GraphFrame.from_caliper_json(filename)
4
5  # We create a copy of GraphFrame
6  >>> gf2 = gf.copy()
7
8  # We need to drop all index levels (rank) in the DataFrame, except node, using the mean
   ↪  time value across all processes for each node
9  >>> gf.drop_index_levels(function=np.mean)
10
11 # Drop all index levels on gf copy DataFrame, excep node, preserving the max time value
   ↪  across all processes for each node
12 >>> gf2.drop_index_levels(function=np.max)
```

We are preserving two different GraphFrames, one with the mean execution time per function across all processes and one with the maximum execution time per function across all processes. This is done because imbalance is measured as: $\Lambda = \frac{T_{max}}{T_{avg}} - 1$, where T_{max} is the execution time of the most loaded process and T_{avg} is the average execution time of all processes. Code Listing 20 then shows the second half of this process, where Λ is computed for all tree nodes. The resulting imbalance value for the `runParticles_kernel` in this artificial imbalance scenario ($\Lambda = 1.45 - 1$) reveals that the most loaded GPU is doing around 45% more work than the average.

Listing 20 Computing load imbalance across processes in a multi-GPU execution of BS-SOLCTRA

```
1  # Using the div operator, we apply the Load Imbalance formula, saving the results in a
   ↪ new column named imbalance
2  >>> gf.dataframe['imbalance'] = gf2.dataframe['time (inc)'].div(gf.dataframe['time
   ↪ (inc)'])
3
4  >>> sorted_functions = gf.dataframe.sort_values(by=['imbalance'], ascending=False)
5
6  >>> print(sorted_functions[["name","imbalance"]])
7                                                                   name  imbalance
8  node
9  {'name': 'MPI:MPI_Bcast', 'type': 'region'}              MPI:MPI_Bcast   3.593435
10 {'name': 'MPI:MPI_Gather', 'type': 'region'}            MPI:MPI_Gather   2.020589
11 {'name': 'MPI:MPI_Barrier', 'type': 'region'}          MPI:MPI_Barrier   2.012709
12 {'name': 'MPI:MPI_Barrier', 'type': 'region'}          MPI:MPI_Barrier   1.995102
13 {'name': 'runParticles_kernel', 'type': 'region'}  runParticles_kernel   1.458454
14 {'name': 'runParticles', 'type': 'region'}                runParticles   1.454647
15 {'name': 'memcpyD2H', 'type': 'region'}                      memcpyD2H   1.360891
16 {'name': 'memcpyH2D', 'type': 'region'}                      memcpyH2D   1.352125
17 {'name': 'MPI:MPI_Scatterv', 'type': 'region'}        MPI:MPI_Scatterv   1.333310
18 {'name': 'MPI:MPI_Scan', 'type': 'region'}                MPI:MPI_Scan   1.151653
19 {'name': 'MPI:MPI_Finalize', 'type': 'region'}        MPI:MPI_Finalize   1.052869
20 {'name': 'MPI:MPI_Init_thread', 'type': 'region'}  MPI:MPI_Init_thread   1.000713
21 {'name': 'main', 'type': 'region'}                                main   1.000032
```

Again, we have proven that Hatchet is now capable of providing its users programmatic analysis of NVIDIA GPU-accelerated applications for another common cause of performance issues, load imbalance. This process can be turned into a script and be re-used to understand the impact that load balancing strategies may have on simulation behavior.

4.5 Roofline Model in Hatchet: LULESH Case Study

So far we have reviewed cases in which the pre-existing Hatchet infrastructure and built-in operations have allowed us to analyze and understand performance for GPU-accelerated applications. However, through the development of this project and thanks to the new data that is available

for analysis, novel Hatchet functionalities can be devised and implemented. In this section we discuss how by relying on Hatchet's DataFrame capabilities and NSight Compute metrics we have implemented a proof-of-concept for the construction of Hatchet based Roofline performance models [78].

The Roofline model is a popular performance analysis mechanism in HPC. The model provides an insightful visualization of an application's usage of computational resources, memory bandwidth and data locality all in one chart [79]. The focus of the model is conveying throughput information, in particular the ratio of floating operations to data moved from memory by one particular function. This relation is named Arithmetic Intensity (AI) and is basically a measure of data reuse by a function, once the data is loaded from memory. The resulting chart is able to communicate a function's floating point performance, memory performance and AI in a two-dimensional chart.

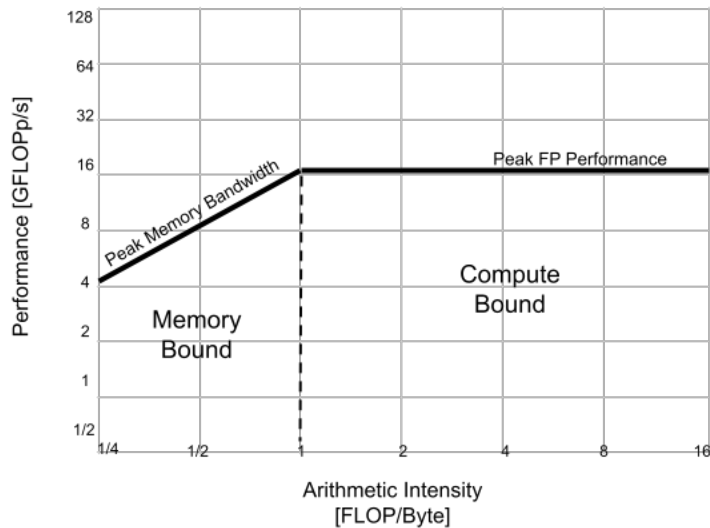


Figure 4.7: Roofline model main components and interpretation

Figure 4.7 shows the main components of a Roofline chart. These are usually on a log-log scale and the Y-axis represents the floating point performance attained by an operation. X-axis on the other hand provides information on the Arithmetic Intensity of the function. Typically, there are two different *ceilings* representing the hardware platform's floating-point peak performance and peak memory bandwidth values. These serve as a reference as to how close a function is to achieving peak performance or is saturating memory bandwidth. This is precisely the power of

the Roofline model, it provides a *bound and bottleneck* analysis [78]. Notice that we have marked two main regions falling under the peak ceilings. Depending on where a function is located in the chart, according to its achieved floating-point performance and arithmetic intensity, a performance limiter could be identified. If for example, a GPU-kernel falls under the memory-bound region, developers could focus on modifying the memory access layout to improve the Arithmetic Intensity. If on the other hand, a kernel falls in the compute bound region but lower than the peak performance ceiling, developers must explore aspects such as occupancy, launch configurations and more detailed complex architectural features.

NSight Compute provides a Roofline analysis section that results in a report that when visualized on the GUI can plot a Roofline chart per kernel invocation. Figure 4.8 shows an example of one such visualization for a test application. This chart shows two horizontal ceilings representing both: peak single-precision (upper limit) and peak double-precision (lower limit) performance. This particular example kernel falls under the compute bound region and thus would require looking into SM, hardware pipelines and scheduling metrics.

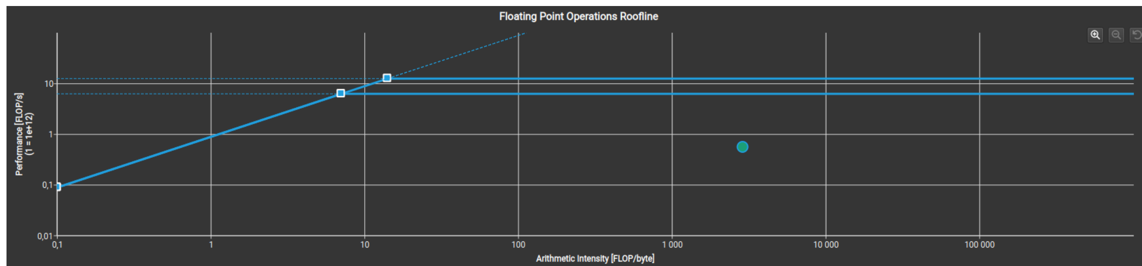


Figure 4.8: Example Roofline chart from the NSight Compute GUI

Now, as stated before, with GPU-specific metrics available in Hatchet, we can also offer users the possibility of constructing such charts programmatically. As this is currently a proof-of-concept implementation, the chart construction function has not been integrated into Hatchet's code base.

Listing 21 Roofline generation process pseudo-code

```
1 >>> import hatchet as ht
2 >>> filename = "lulesh_100_detailed.json"
3 >>> gf = ht.GraphFrame.from_caliper_json(filename)
4 >>> kernels = gf.dataframe.loc[gf.dataframe["isGPUKernel"]==True]
5 >>> for kernel in kernels:
6         computeRooflineMetrics(kernel)
7         chartRoofline(kernel["name"], kernel["peak_traffic"], kernel["peak_work"],
        ↪  kernel["dp_flops"], kernel["arithmetic_intensity"])
```

Code listing 21 shows a pseudo-code description of how easily a Roofline chart can be plotted once the relevant metrics and derived metrics have been computed. Now, the most important aspect here is utilizing the correct metrics to compute the achieved floating-point and memory performance as well as the achieved Arithmetic Intensity.

Roofline component	Metrics Involved	Description
Peak FP64 Performance	peak_fp = derived__sm__sass_thread_inst_executed_op_dfma_pred_on_x2 * sm__cycles_elapsed.avg.per_second	Total number of thread-level executed FP64 fused-multiply add operations of the application times the operation frequency of the GPU
Peak Memory Bandwidth	peak_bw = dram__bytes.sum.peak_sustained * dram__cycles_elapsed.avg.per_second	Total number of bytes loaded by the application times the operation frequency of the GPU memory
FP64 FLOP/S	flops = (smsp__sass_thread_inst_executed_op_dadd_pred_on.sum.per_cycle_elapsed + smsp__sass_thread_inst_executed_op_dmul_pred_on.sum.per_cycle_elapsed + smsp__sass_thread_inst_executed_op_dfma_pred_on.sum.per_cycle_elapsed*2) * smsp__cycles_elapsed.avg.per_second	Total number of double-precision floating point operations per second. FMA operations count as two instructions, thus the multiplication of 2
Achieved Bandwidth	bw = dram__bytes.sum.per_second	Total number of bytes loaded by second
Peak Arithmetic Intensity	peak_ai = peak_fp/peak_bw	Ratio of PEAK FLOP/S to Peak GB/S
Achieved Arithmetic Intensity	achieved_ai = flops/bw	Ratio of Achieved FLOP/S to Achieved GB/S

Table 4.2: NSight Compute metrics needed to compute Roofline model for GPU kernels

Table 4.2 shows the required NSight Compute metrics and their relations to attain the main Roofline components. Each of this components must be computed per kernel so that a Roofline plot can be created for each operation.Code Listing 22 shows this process as implemented in a Python script to automate this process.

Listing 22 Roofline components computation with NSight Compute data loaded into Hatchet

```
1   def peak_work(kernel):
2     peak_work =  kernel["derived__sm__sass_thread_inst_executed_op_dfma_pred_on_x2"] *
    ↪   kernel["sm__cycles_elapsed.avg.per_second"]
3     return peak_work
4   def peak_traffic(kernel):
5     peak_traffic =  kernel["dram__bytes.sum.peak_sustained"] *
    ↪   kernel["dram__cycles_elapsed.avg.per_second"]
6     return peak_traffic
7   def dp_flops(kernel):
8     dp_flops =
    ↪   ((kernel["smsp__sass_thread_inst_executed_op_dadd_pred_on.sum.per_cycle_elapsed"]+
    ↪   kernel["smsp__sass_thread_inst_executed_op_dmul_pred_on.sum.per_cycle_elapsed"]+
9     kernel["smsp__sass_thread_inst_executed_op_dfma_pred_on.sum.per_cycle_elapsed"]*2)*
    ↪   kernel["smsp__cycles_elapsed.avg.per_second"])
10    return dp_flops
11  def achieved_traffic(kernel):
12    traffic = kernel["dram__bytes.sum.per_second"]
13    return traffic
14  def peak_ai(kernel):
15    ai_peak = kernel["peak_work"]/kernel["peak_traffic"]
16    return ai_peak
17  def arithmetic_intensity(kernel):
18    ai_value = kernel["dp_flops"]/kernel["achieved_traffic"]
19    return ai_value
```

Figure 4.9 shows a resulting plot generated from a Hatchet NSight Compute data analysis. Through these plots the developer could identify performance issues as related either to low AI

or computation issues. There is an important clarification that must be made about the ceiling values shown in these charts. These LULESH data collection experiments were performed in Kabré's NVIDIA V100 GPUs. This model has a theoretical peak performance of 7.8 TFLOP/s and a theoretical memory bandwidth of 900 GB/s. These performance values are achieved when the GPU executes at maximum device operation frequency. However, the experimental roofline shown here (same applies for NSight Compute GUI), compute the peak performance based on a lower value of GPU frequency. This lower value corresponds to a design decision of the NSight Compute data capturing process. As a way to achieve consistent profiling results, this software limits the GPU clock frequency to its base value [80].

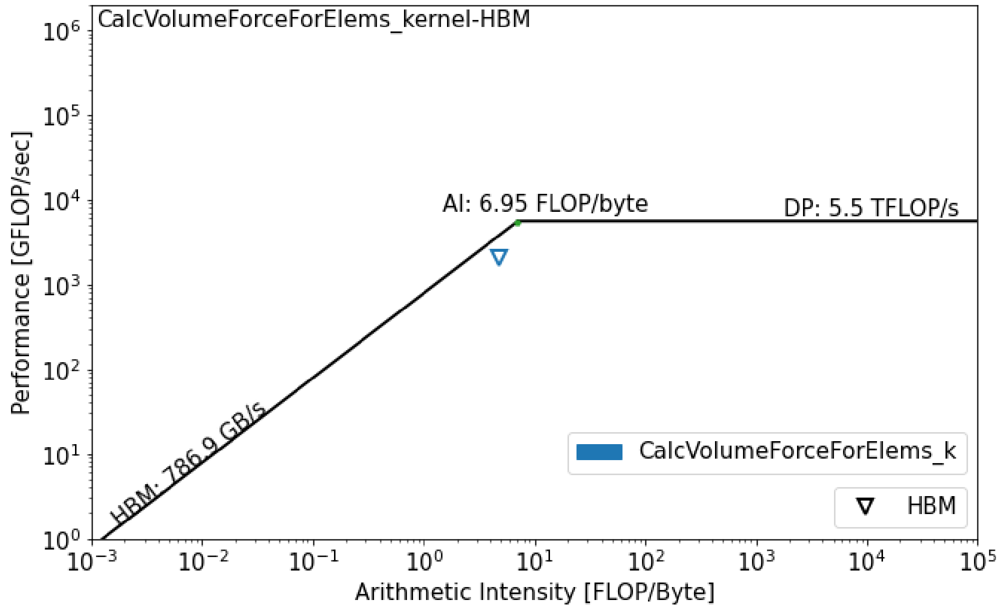
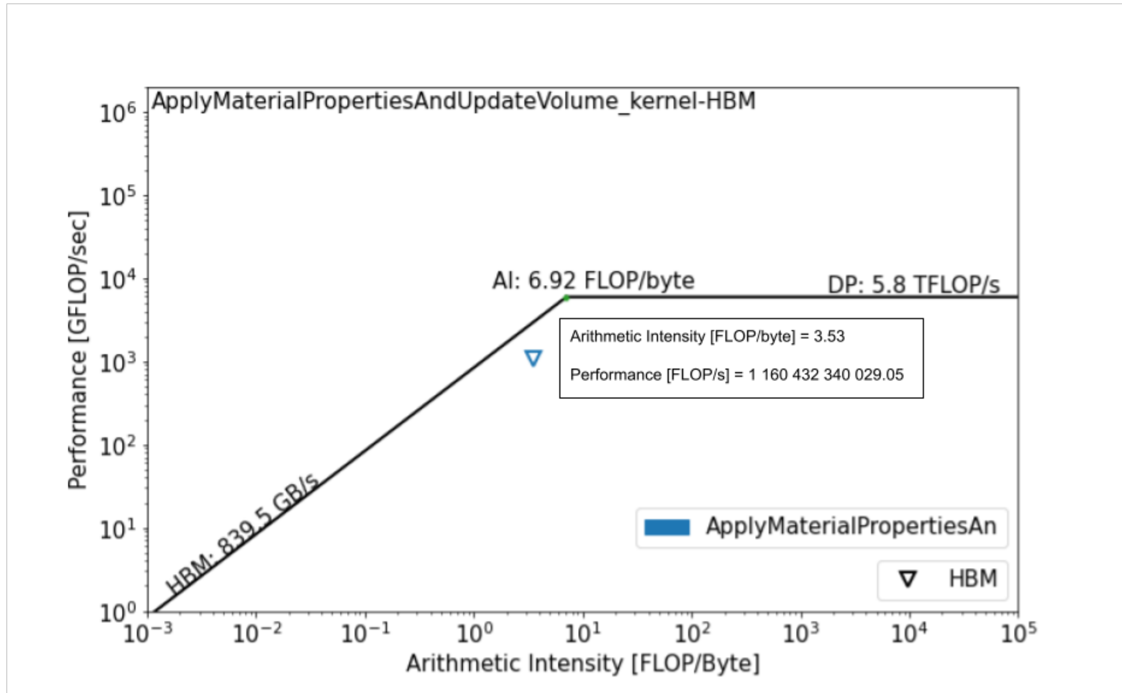


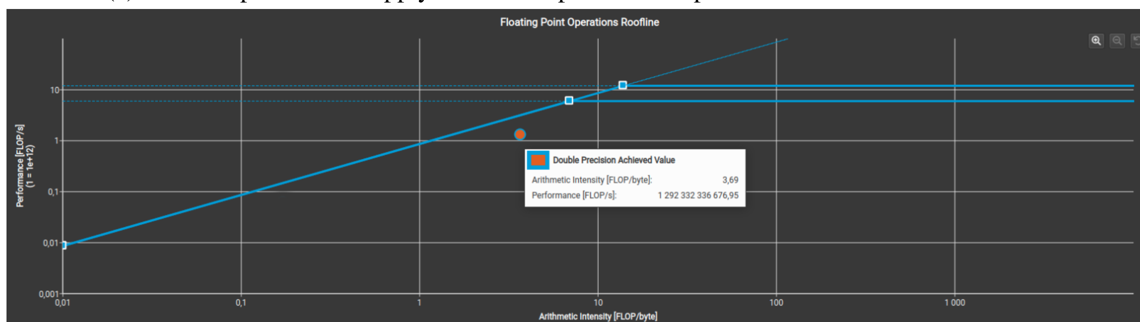
Figure 4.9: Roofline plot for the CalcVolumeForceForElems_kernel in LULESH

Figure 4.10 shows a comparison between one of our generated rooflines and an NSight Compute roofline chart for the same kernel. There are slight differences in the achieved Arithmetic Intensity and total FLOP/s that are explained by the fact the NSight Compute plots a roofline for every kernel invocation, while we are using the average metrics for a kernel to compute those values. In conclusion, through NVIDIA NSight data, Hatchet can now provide its user with a new visualization functionality for performance analysis. The three different case studies we have reviewed show

that indeed it is possible to identify performance bottlenecks in GPU-accelerated applications with our new data transformation workflow. Furthermore, they show the breadth of possibilities that programmatic performance analysis offers and that now can be applied to NVIDIA NSight tools data.



(a) Roofline plot for the ApplyMaterialPropertiesAndUpdateVolume_kernel in LULESH



(b) NSight Compute Roofline plot for the ApplyMaterialPropertiesAndUpdateVolume_kernel in LULESH

Figure 4.10: Roofline comparison of our implementation through programmatic analysis and the chart generated by NSight Compute

Chapter 5

Conclusions

5.1 Summary

This project focused on the implementation of a data processing and manipulation workflow to unlock programmatic performance analysis of NVIDIA NSight Tools-generated data. This allows users to implement programmatic performance analysis pipelines and offers a set of functionalities to facilitate the manipulation of hierarchical profiles throught the Hatchet library.

Through the work developed in this thesis, we identified and integrated the necessary components to offer Hatchet users a comprehensible and accessible way to analyze GPU-accelerated applications profiled with NSight Tools. An NVTX code annotation methodology and programming model-specific considerations were established to generate meaningful performance data on both NSight Systems and NSight Compute. Resulting application traces and GPU-specific metrics can the be fed to a Python software capable of integrating both reports into hierarchical profiles like call graphs and calling context trees. This software can even process multi-GPU performance data, creating a per-process tree.

Furthermore, an already Hatchet-compliant data model was exploited to facilitate the loading of NSight performance data into Hatchet for analysis. This model is capable of maintaining hierarchical profiles and whatever amount of GPU-metrics are provided by the NSight Compute report. This annotation-profiling-transformation pipeline was illustrated through the use of a well-known HPC application like LULESH.

Three different case studies coming from three typical HPC domains were used to exemplify how the implemented pipeline now enables users to identify the main performance bottlenecks in

their GPU powered applications. Through Hatchet's simple hierarchy visualizations a Tensorflow Keras model was optimized through the use of the framework's mixed precision capabilities. GPU kernel metrics extracted with NSight Compute and their usage within Hatchet, allowed us to identify a difference in the launch configuration of a plasma physics simulation kernel implemented under two different OpenMP programming approaches.

Furthermore, using the BS-SOLCTRA simulation, we showed how Hatchet user can now identify and quantify load imbalances in multi-GPU executions. Finally, the Roofline model capability prototyped with Hatchet analysis is able to provide insightful information of a kernel's main performance limiters be it memory access or floating point performance. In conclusion, thanks to the work developed in this project, programmatic performance analysis of NVIDIA NSight performance data is now possible in Hatchet, giving its users the freedom to implement their own custom analysis of GPU executions.

5.2 Contributions

The main contribution of this project is a data processing and manipulation workflow capable of generating hierarchical call path information for GPU-accelerated applications from NVIDIA's NSight Systems and Compute output reports. This functionality is currently being integrated into the actual Hatchet code base to become publicly available.

We proposed and implemented a mechanism to generate Caliper-like JSON files that are capable of storing hierarchical profiles and numerous GPU-related metrics for those functions identified as kernels.

Finally, we presented three different real-world applications and the process a user follow to carry out typical performance analysis procedures like bottleneck identification, code version comparisons, speedup calculations, load imbalance identification in multi-GPU executions and the creation of Roofline charts to quickly visualize what is bounding an application's performance.

5.3 Limitations

The main limitation of this project and the implemented pipeline is the necessity to perform the trace extraction and the GPU metrics collection in two different application executions. The design decision by NVIDIA, although comprehensible, to have users rely on two different tools to optimize performance might make this process more cumbersome than with other performance analysis tools.

Furthermore, given that the trace and GPU-metrics correspond to two different executions, there might be differences that could potentially affect the legitimacy of an analysis.

As our hierarchy reconstruction process depends on the available data in the NVTX trace report, there are some CUDA features that can not be used, for example: the usage of multiple concurrent streams which is used to overlap data movement and computation. The NVTX trace report does not include information on which stream an operation occurred and as such, it is impossible to differentiate them and reconstruct a correct context-aware call path.

In terms of our NVTX annotation methodology, there are some CUDA capabilities that have not been explored and could be troublesome. For example, measuring the total execution time of asynchronous data movement operations without having to synchronize right after the function invocation, which would render the whole idea of asynchronicity invalid or the use of unified memory management which "hides" data transfers from the programming layer.

5.4 Future Work

As previously mentioned, the implemented transformation procedure is currently being integrated into Hatchet. As future work, the Roofline construction feature also needs to be included into this library to make this freely available. Related to the Roofline model, although available in the GPU metrics, we did not explore the creation of Hierarchical Roofline charts that not only show the compute-to-main memory relation but also the relation between computation and other levels of the memory hierarchy like register files or cache levels.

We are also currently looking for an HPC application accelerated with NVIDIA GPUs to perform a real-need application optimization study that could be published.

Bibliography

[1] Jack Dongarra, Steven Gottlieb, and William TC Kramer. Race to exascale. *Computing in Science & Engineering*, 21(1):4–5, 2019.

[2] Daniel A Reed and Jack Dongarra. Exascale computing and big data. *Communications of the ACM*, 58(7):56–68, 2015.

[3] Robert Lucas, James Ang, Keren Bergman, Shekhar Borkar, William Carlson, Laura Carrington, George Chiu, Robert Colwell, William Dally, Jack Dongarra, et al. Doe advanced scientific computing advisory subcommittee (ascac) report: top ten exascale research challenges. Technical report, USDOE Office of Science (SC)(United States), 2014.

[4] Al Geist and Robert Lucas. Major computer science challenges at exascale. *The International Journal of High Performance Computing Applications*, 23(4):427–436, 2009.

[5] Douglas Kothe, Stephen Lee, and Irene Qualters. Exascale computing in the united states. *Computing in Science & Engineering*, 21(1):17–29, 2018.

[6] Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. Hatchet: pruning the overgrowth in parallel profiles. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–21, 2019.

[7] M Usman Ashraf, Fathy Alburaei Eassa, Aiiad Ahmad Albeshri, and Abdullah Algarni. Performance and power efficient massive parallel computational model for hpc heterogeneous exascale systems. *IEEE Access*, 6:23095–23107, 2018.

[8] Top500.org. Top500 june 2022, 2022. data retrieved from: `https://www.top500.org/lists/top500/2022/06/`.

[9] Top500.org. Green500 november 2022, 2022. data retrieved from: `https://www.top500.org/lists/green500/2022/06/`.

[10] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[11] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[12] Intel Corporation. Intel® vtune™ profiler, 2021. data retrieved from: `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html#gs.3m4ujq`.

[13] Sameer S Shende and Allen D Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[14] Karl Fuerlinger, Michael Gerndt, and Jack Dongarra. On using incremental profiling for the performance analysis of shared memory parallel applications. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par 2007 Parallel Processing*, pages 62–71, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[15] Susan L Graham, Peter B Kessler, and Marshall K McKusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982.

[16] David J Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge university press, 2005.

[17] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. john wiley & sons, 1990.

[18] Sameer Shende. Profiling and tracing in linux. In *Proceedings of the Extreme Linux Workshop*, volume 2. Citeseer, 1999.

[19] David Padua, editor. *Encyclopedia of Parallel Computing: Performance Measurement*, pages 1522–1522. Springer US, Boston, MA, 2011.

[20] Jason Gait. A debugger for concurrent programs. *Software: Practice and Experience*, 15(6):539–554, 1985.

[21] Jack Dongarra, Kevin London, Shirley Moore, Phil Mucci, and Dan Terpstra. Using papi for hardware performance monitoring on linux systems. In *Conference on Linux Clusters: The HPC Revolution*, volume 5. Citeseer, 2001.

[22] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

[23] Tjerk P Straatsma, Katerina B Antypas, and Timothy J Williams. *Exascale scientific applications: Scalability and performance portability*. CRC Press, 2017.

[24] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. Caliper: performance introspection for hpc software stacks. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 550–560. IEEE, 2016.

[25] Glenn Ammons, Thomas Ball, and James R Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices*, 32(5):85–96, 1997.

[26] Judit Giménez, Jesús Labarta, F Xavier Pegenaute, Hui-Fang Wen, David Klepacki, I-Hsin Chung, Guojing Cong, Felix Voigtländer, and Bernd Mohr. Guided performance analysis combining profile and trace tools. In *European Conference on Parallel Processing*, pages 513–521. Springer, 2010.

[27] Wen-mei Hwu, Kurt Keutzer, and Timothy G Mattson. The concurrency challenge. *IEEE Design & Test of Computers*, 25(4):312–320, 2008.

[28] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[29] NVIDIA Corporation. Nvidia tesla v100 gpu architecture whitepaper. data retrieved from: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[30] NVIDIA Corporation. Nvidia a100 tensor core gpu architecture whitepaper. data retrieved from: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[31] Richard Vuduc and Jee Choi. A brief history and introduction to gpgpu. In *Modern Accelerator Technologies for Geographic Information Science*, pages 9–23. Springer, 2013.

[32] Chris J Thompson, Sahngyun Hahn, and Mark Oskin. Using modern graphics architectures for general-purpose computing: A framework and analysis. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings.*, pages 306–317. IEEE, 2002.

[33] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *SC'04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, pages 47–47. IEEE, 2004.

[34] Jaegeun Han and Bharatkumar Sharma. *Learn CUDA Programming: A beginner's guide to GPU programming and parallel computing with CUDA 10. x and C/C++*. Packt Publishing Ltd, 2019.

[35] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[36] NVIDIA Corporation. Cuda c++ programming guide pg-02829-001_v11.7. data retrieved from: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[37] Mohamed Zahran. Heterogeneous computing: Here to stay. *Communications of the ACM*, 60(3):42–45, 2017.

[38] Allen D Malony, Scott Biersdorff, Sameer Shende, Heike Jagode, Stanimire Tomov, Guido Juckeland, Robert Dietrich, Duncan Poole, and Christopher Lamb. Parallel performance measurement of heterogeneous parallel systems with gpus. In *2011 International Conference on Parallel Processing*, pages 176–185. IEEE, 2011.

[39] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Message Passing Interface Forum, 2012.

[40] OpenMP Architecture Review Board. Openmp application programming interface. data retrieved from: `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf`.

[41] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[42] AMD. C++ heterogeneous-compute interface for portability hip. `https://github.com/ROCm-Developer-Tools/HIP`.

[43] OpenACC-Standard.org. The openacc application programming interface version 3.1. data retrieved from: `https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf`.

[44] George S Markomanolis, Aksel Alpay, Jeffrey Young, Michael Klemm, Nicholas Malaya, Aniello Esposito, Jussi Heikonen, Sergei Bastrakov, Alexander Debus, Thomas Kluge, et al. Evaluating gpu programming models for the lumi supercomputer. In *Asian Conference on Supercomputing Frontiers*, pages 79–101. Springer, Cham, 2022.

[45] NVIDIA Corporation. Cuda toolkit. data retrieved from: `https://developer.nvidia.com/cuda-toolkit`.

[46] Abdulla Mohamed. *Enhancing CMS DAQ Systems Performance using Performance Profiling of Parallel Programs on GPGPUS*. PhD thesis, Bahrain U., 2020.

[47] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing*, 74(12):3202–3216, 2014.

[48] Jose Monsalve Diaz, Swaroop Pophale, Oscar Hernandez, David E Bernholdt, and Sunita Chandrasekaran. Openmp 4.5 validation and verification suite for device offload. In *International Workshop on OpenMP*, pages 82–95. Springer, 2018.

[49] Ruud Van der Pas, Eric Stotzer, and Christian Terboven. *Using OpenMP# The Next Step: Affinity, Accelerators, Tasking, and SIMD*. MIT press, 2017.

[50] Guray Ozen and Michael Wolfe. Performant portable openmp. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pages 156–168, 2022.

[51] NVIDIA Corporation. Nvidia® developer tools. data retrieved from: https://developer.nvidia.com/tools-overview.

[52] NVIDIA Corporation. Nvidia® nsight™ systems. data retrieved from: https://developer.nvidia.com/nsight-systems.

[53] NVIDIA Corporation. Nvidia® nsight™ compute. data retrieved from: https://developer.nvidia.com/nsight-compute.

[54] NVIDIA Corporation. Nvidia® nsight™ systems documentation - installation guide. data retrieved from: https://docs.nvidia.com/nsight-systems/InstallationGuide/index.html#overview.

[55] NVIDIA Corporation. Nvidia® nsight™ compute kernel replay. data retrieved from: https://docs.nvidia.com/nsight-compute/2022.2/ProfilingGuide/index.html#replay.

[56] NVIDIA Corporation. Nvidia® tools extension library (nvtx). data retrieved from: https://docs.nvidia.com/nsight-visual-studio-edition/2020.1/nvtx/index.html#nvtx-library.

[57] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, Lawrence Livermore National Laboratory, August 2013.

[58] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[59] Eliu A Huerta, Asad Khan, Edward Davis, Colleen Bushell, William D Gropp, Daniel S Katz, Volodymyr Kindratenko, Seid Koric, William TC Kramer, Brendan McGinty, et al. Convergence of artificial intelligence and high performance computing on nsf-supported cyberinfrastructure. *Journal of Big Data*, 7(1):1–12, 2020.

[60] Dan Guest, Kyle Cranmer, and Daniel Whiteson. Deep learning and its application to lhc physics. *arXiv preprint arXiv:1806.11484*, 2018.

[61] Logan Ward, Ben Blaiszik, Ian Foster, Rajeev S Assary, Badri Narayanan, and Larry Curtiss. Machine learning prediction of accurate atomization energies of organic molecules from low-fidelity quantum chemical calculations. *MRS Communications*, 9(3):891–899, 2019.

[62] Asad Khan, EA Huerta, Sibo Wang, Robert Gruendl, Elise Jennings, and Huihuo Zheng. Deep learning at scale for the construction of galaxy catalogs in the dark energy survey. *Physics Letters B*, 795:248–258, 2019.

[63] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[64] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[65] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[66] Diego Jiménez, Luis Campos-Duarte, Ricardo Solano-Piedra, Luis Alonso Araya-Solano, Esteban Meneses, and Iván Vargas. Bs-solctra: Towards a parallel magnetic plasma confinement simulation framework for modular stellarator devices. In *Latin American High Performance Computing Conference*, pages 33–48. Springer, 2019.

[67] Diego Jiménez, Esteban Meneses, and V.I. Vargas. Adaptive plasma physics simulations: Dealing with load imbalance using charm++. In *Practice and Experience in Advanced Research Computing*, PEARC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[68] R Hornung, JA Keasler, and MB Gokhale. Hydrodynamics challenge problem, lawrence livermore national laboratory. *Tech. Rep. LLNL-TR-490254*, 2011.

[69] Ian Karlin. Lulesh programming model and performance ports overview. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2012.

[70] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, et al. Exploring traditional

and emerging parallel programming models using a proxy application. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 919–932. IEEE, 2013.

[71] NVIDIA Corporation. Nvidia® nsight™ compute sets and sections. data retrieved from: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#sections-and-rules.

[72] NVIDIA Corporation. Nvidia gpu cloud. data retrieved from: https://docs.nvidia.com/ngc/index.html.

[73] LLNL Caliper Developers. Caliper examples, 2020. data retrieved from: https://github.com/LLNL/caliper-examples.

[74] CeNAT. Kabré: Guía de usuario. https://kabre.cenat.ac.cr/guia-usuario/.

[75] Max Planck Computing and Data Facility. Raven user guide. https://docs.mpcdf.mpg.de/doc/computing/raven-user-guide.html.

[76] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International conference on machine learning*, pages 1737–1746. PMLR, 2015.

[77] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[78] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[79] Charlene Yang. Hierarchical roofline analysis: How to collect data using performance tools on intel cpus and nvidia gpus. *arXiv preprint arXiv:2009.02449*, 2020.

[80] NVIDIA Corporation. Nsight compute documentation - clock control. data retrieved from: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#clock-control.