HAL
open science

# Evaluation of programming models for manycore and / or heterogeneous architectures for Monte Carlo neutron transport codes

Tao Chang

## To cite this version:

Tao Chang. Evaluation of programming models for manycore and / or heterogeneous architectures for Monte Carlo neutron transport codes. Computer science. Institut Polytechnique de Paris, 2020. English. NNT : 2020IPPAX099 . tel-03086536

**HAL Id: tel-03086536**

**https://theses.hal.science/tel-03086536v1**

Submitted on 22 Dec 2020

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2020IPPAX099

Thèse de doctorat

# Evaluation of programming models for manycore and / or heterogeneous architectures for Monte Carlo neutron transport codes

Thèse de doctorat de l'Institut Polytechnique de Paris

préparée à l'École polytechnique

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (ED IP Paris)

Spécialité de doctorat: Informatique, Données et Intelligence Artificielle

Thèse présentée et soutenue à Saclay, le 01/12/2020, par

**TAO CHANG**

Composition du Jury :

| | |
|---|---|
| Michael HEROUX<br>Professeur, Sandia National Laboratories and St. John's University | Rapporteur |
| Jean-François MEHAUT<br>Professeur, Université Grenoble Alpes | Rapporteur |
| Raymond NAMYST<br>Professeur, Université de Bordeaux | Examinateur |
| Marc VERDERI<br>Directeur de recherche, Laboratoire Leprince Ringuet de l'IPP | Président |
| Emmanuelle SAILLARD<br>Chargé de recherche, Le Centre de Recherche INRIA Bordeaux | Examinateur |
| Francieli ZANON BOITO<br>Maitre de Conférences, Université de Bordeaux | Examinateur |
| Christophe CALVIN<br>Expert International, CEA Saclay | Directeur de thèse |
| Emeric BRUN<br>Ingénieur de recherche, CEA Centre Marcoule | Encadrant |

*This thesis is dedicated to my family who have raised me to be the person I am today.*

*I hope that this achievement makes you happy.*

# Acknowledgements

First and foremost, I would like to express my deep and sincere gratitude to my thesis supervisor, Christophe CALVIN, for the continuous support of my Ph.D study throughout the past three years, for his motivation, enthusiasm and vision which have deeply inspired me every time I got stuck on research. I am extremely grateful for having the opportunity to study and work under his guidance.

I would like to thank my advisor, Emeric BRUN, for his patience, insightful comments and hardware support. I am grateful for his aids on helping me carrying out tests on the Cobalt-hybrid, Cobalt-v100 platforms. Moreover, I thank him and Philippe Thierry for providing me an Intel NUC machine to accomplish my research work.

I thank the rest of my thesis committee: Michael HEROUX and Jean-François MEHAUT, for being my reviewers and writing me the literature of review and Raymond NAMYST, Marc VERDERI, Emmanuelle SAILLARD, Francieli ZANON BOITO, for being the examiners at my thesis defense.

My sincere thanks also goes to Fausto MALVAGI, the director of my laboratory SERMA/LTSD in CEA Saclay and Edouard Audit, the director of Maison de la Simulation, for allowing me studying and working in two places throughout the research. I would also like to express my gratitude to my fellow labmates in these two laboratories, for their administrative, academical and hardware supports as well as insightful talks with me.

I would like to thank David Chamont for offering me access to the GridCL platform and being my academical advisor. I thank Pierre Kestener for giving his useful comments on the implementation of Kokkos in the Monte Carlo neutron transport code. I thank Serge G. Petiton, for guiding my master internship in Maison de la Simulation and enlightening me the first glance of research in the field of HPC.

I am very much thankful to my parents and twin brother for their love and sacrifices for supporting me continuously to chase my dream.

INSTITUT POLYTECHNIQUE DE PARIS

**Titre :** Évaluation de modèles de programmation pour les architectures manycore et / ou hétérogènes pour les codes de transport neutronique Monte Carlo

**Mots clés :** Architectures manycore, Architectures hétérogènes, Transport des particules

**Résumé :** Dans cette thèse nous nous proposons d'évaluer les différents modèles de programmation disponibles pour adresser les architectures de type manycore et / ou hétérogènes dans le cadre des codes de transport Monte Carlo. On considèrera dans un premier temps un cas test d'application simple mais représentatif pour couvrir un éventail assez large de solutions et les comparer en terme de performance, de portabilité de la performance, de facilité de mise en œuvre et de maintenabilité. Les architectures cibles sont les CPU 'classique', Intel Xeon Phi et GPU. Les modèles de programmation les plus pertinents seront ensuite mis en place dans un code de transport Monte Carlo.

**Title :** Evaluation of programming models for manycore and / or heterogeneous architectures for Monte Carlo neutron transport codes

**Keywords :** Manycore architectures, Heterogeneous architectures, Particles transport

**Abstract :** In this thesis we propose to evaluate the different programming models available for addressing manycore and / or heterogeneous architectures within the framework of the Monte Carlo transport codes. A simple but representative application test case will be considered in order to cover a fairly wide range of solutions and compare them in terms of performance, portability of performance, ease of implementation and maintainability. The target architectures are 'classic' CPU, Intel Xeon Phi and GPU. The most relevant programming models will then be set up in a Monte Carlo transport code.

**Institut Polytechnique de Paris**
91120 Palaiseau, France

# Résumé

La simulation du transport de neutrons de Monte-Carlo est une méthode stochastique largement utilisée dans le domaine nucléaire pour les calculs de référence. Au lieu d'introduire des approximations mathématiques et physiques pour simuler un système du monde réel, la méthode de Monte-Carlo regroupe les comportements des particules par échantillonnage statistique, ce qui entraîne de manière significative un coût de calcul énorme. Afin d'alléger ce coût, l'utilisation de supercalculateurs pour les simulations de Monte-Carlo est devenue une tendance pour de nombreux chercheurs et développeurs.

Cependant, comme de plus en plus d'architectures modernes (multi-core / manycore, architectures hétérogènes) émergent à un rythme assez rapide, il n'est pas anodin de développer des applications portables complètes sur toutes ces architectures, de plus, d'optimiser et de maintenir l'application en continu donc pour obtenir des performances préférables sur eux.

D'un côté, le développement et la durée de vie d'une application sont bien supérieurs au temps de développement d'architectures informatiques. Ainsi, concevoir une application pour une architecture donnée n'a pas de sens car après le développement de l'application, il y a de fortes chances que l'architecture ait déjà évolué. De l'autre côté, il est nécessaire que les applications soient évolutives et maintenables par le plus de personnes possible, permettant à ces personnes qui ne sont pas familiarisées avec le HPC de développer des codes capables de fournir de bonnes performances.

Pour répondre à l'évolution de la portabilité des performances, certains langages de programmation de haut niveau tels que OpenACC, OpenMP offload ont été proposés pour permettre à l'application de bien fonctionner sur une large gamme d'architectures sans de gros frais de développement et de maintenance. La plupart de ces modèles de programmation sont dédiés aux architectures basées sur des accélérateurs avec l'augmentation massive du parallélisme multi-niveaux qui représente la tendance sous-jacente des architectures de calcul. Dans cette thèse, nous nous intéressons au développement d'un code de transport de neutrons Monte-Carlo portable ciblant des architectures hybrides (CPU + GPU) avec l'utilisation de modèles de programmation hybrides.

Concernant l'évaluation de la portabilité des performances, certaines métriques ont déjà été proposées pour évaluer quantitativement le compromis entre performance et portabilité. Cependant, il existe peu de recherches traitant de l'évaluation des codes de transport de neutrons de Monte-Carlo sur les supercalculateurs en termes de portabilité des performances, en outre, de portabilité des performances productibles. Ainsi, nous avons l'intention

de donner une évaluation explicite en termes de portabilité, de performance, ainsi que de productivité pour un mini-benchmark de simulation de transport de neutrons de Monte-Carlo.

Le mini-benchmark slabAllNuclides a été implémenté sur un prototype de transport de neutrons de Monte-Carlo développé au CEA, appelé PATMOS. Une version de déchargement hétérogène du code a été développée via une méthode basée sur l'historique et une méthode basée sur des pseudo événements. Plusieurs modèles de programmation (thread OpenMP, OpenMP offload, OpenACC, CUDA, Kokkos et SYCL) ont été implémentés pour l'évaluation et les tests ont été réalisés sur différentes architectures (x86 ou OpenPower + GPU). Un ensemble de métriques a été adopté pour évaluer notre simulation Monte-Carlo de déchargement hétérogène en termes de portabilité, de performances et de productivité. L'analyse des métriques a été réalisée avec la proposition d'une métrique générique et son qui atténue les défauts des métriques d'origine, les rendant plus adaptables pour gérer la comparaison numérique des applications en termes de portabilité des performances et de portabilité de la productivité à travers une variété d'informatique architectures.

En un mot, l'objectif de la thèse est d'évaluer les métriques existantes en termes de portabilité, de performance et de productivité et de proposer des améliorations de ces métriques sur un exemple d'application réelle réalisant une simulation de transport de neutrons de Monte-Carlo. Les principaux objectifs sont les suivants:

1. proposer un nouvel algorithme de suivi des particules, la méthode pseudo event-based.
2. analyser et régler les performances de la version CUDA.
3. mettre en œuvre des codes de transport de neutrons Monte-Carlo portables via des modèles de programmation de haut niveau.
4. proposer et analyser une métrique générique et ses variantes.
5. évaluer les implémentations portables des codes de transport Monte-Carlo par la métrique générique.

# Contents

**List of Figures** iv

**List of Tables** vi

**Glossary** viii

**1 Introduction** 1

1.1 Motivations . . . 1

1.2 Objectives . . . 2

1.3 Outline . . . 2

**2 Parallel Computing** 4

2.1 General Terminology . . . 4

2.1.1 Von Neumann Architecture . . . 4

2.1.2 Flynn's Taxonomy . . . 5

2.1.3 Parallel Speedup . . . 8

2.1.4 Scalability . . . 10

2.2 Memory Architectures . . . 11

2.2.1 Shared Memory . . . 11

2.2.2 Distributed Memory . . . 12

2.2.3 Hybrid Memory . . . 13

2.3 Computing Architectures . . . 13

2.3.1 Hardware Characteristics . . . 13

2.3.2 CPU Architectures . . . 14

2.3.3 GPU Architectures . . . 18

2.3.4 Heterogeneous Architecture . . . 23

2.4 Parallel Programming Models . . . 23

2.4.1 Low-Level APIs . . . 24

2.4.2 Directive-based Programming Models . . . 27

2.4.3 Libraries . . . 29

2.4.4 Comparison of Implemented Programming Models . . . 32
2.5 Profiling Tools for Performance Analysis . . . 32
2.5.1 nvprof . . . 33
2.5.2 NVIDIA Visual Profiler . . . 33

**3 Monte Carlo Neutron Transport Simulation 34**
3.1 Nuclear Reactor Physics . . . 34
3.2 Neutron Transport Simulation . . . 35
3.2.1 Neutron Transport Equation . . . 35
3.2.2 Nuclear Cross Section . . . 37
3.2.3 Isotopic Depletion Calculations and Thermohydraulic Feedback . . . 41
3.3 Computational Methods . . . 41
3.3.1 Deterministic Method . . . 41
3.3.2 Monte Carlo Method . . . 42
3.4 HPC and Monte Carlo Neutron Transport Simulation . . . 44
3.4.1 State-of-the-art . . . 44
3.4.2 Parallelism . . . 45
3.4.3 History-based and Event-based Methods . . . 47
3.4.4 Cross Section Computing Algorithms . . . 49
3.4.5 The PATMOS Monte Carlo Prototype . . . 53

**4 Portable Implementation of on-the-fly Doppler Broadening in PATMOS 54**
4.1 Implementation of SIGMA1 on-the-fly Doppler Broadening Algorithm . . . 54
4.2 Implementations in the Different Programming Models . . . 57
4.2.1 Code Architecture . . . 58
4.2.2 OpenMP Thread Implementation . . . 61
4.2.3 CUDA Implementation . . . 61
4.2.4 OpenACC Implementation . . . 63
4.2.5 OpenMP Offload Implementation . . . 65
4.2.6 Kokkos Implementation . . . 67
4.2.7 SYCL Implementation . . . 69
4.3 Implementations of Particle Tracking Methods . . . 71
4.3.1 Heterogeneous Offloading History-based Method . . . 71
4.3.2 Heterogeneous Offloading Pseudo Event-based Method . . . 72
4.4 Numerical Results . . . 73
4.4.1 Description of the benchmark . . . 73
4.4.2 Compiler Support . . . 73
4.4.3 Execution Environment . . . 75
4.4.4 Performance Analysis in Host Mode . . . 76

4.4.5 Performance Analysis in Offload Mode . . . 80
4.4.6 Performance Evaluation . . . 84
4.5 Profiling and Optimization . . . 90
4.5.1 Profiling of HB and PEB . . . 91
4.5.2 Profiling of different programming models . . . 93
4.5.3 Optimizations of the CUDA version . . . 95

**5 Metrics Comparison among Programming Models 102**
5.1 Definitions of Metrics . . . 102
5.1.1 Generic Model . . . 104
5.1.2 Adaptability Analysis . . . 107
5.1.3 Metrics of Portability, Performance, Productivity . . . 111
5.1.4 Metrics of Performance Portability . . . 115
5.1.5 Metrics of Productivity Portability . . . 121
5.2 Evaluation of Benchmark . . . 122
5.2.1 Performance Portability Evaluation . . . 123
5.2.2 Productivity Portability Evaluation . . . 127

**6 Conclusion and Future Work 133**
6.1 Conclusion . . . 133
6.2 Future Work . . . 135

**Bibliography 136**

# List of Figures


2.1 Von Neumann Architecture. . . . . 5
2.2 Instruction and Data Stream. . . . . 6
2.3 SISD Organization. . . . . 6
2.4 SIMD Organization. . . . . 7
2.5 MISD Organization. . . . . 7
2.6 MIMD Organization. . . . . 8
2.7 Shared Memory Architecture. . . . . 11
2.8 Distributed Memory Architecture. . . . . 12
2.9 Hybrid Memory Architecture. . . . . 13
2.10 Modern CPU Architecture [90]. . . . . 14
2.11 Broadwell Die Diagram [4]. . . . . 15
2.12 POWER9 SMT Core Diagram [7]. . . . . 17
2.13 POWER9 SMT4 Chip Diagram where the cache hierarchy is illustrated [108]. . . . . 17
2.14 Modern GPU Architecture [90], where “green square” refers to GPU core, “yellow rectangle” is GPU control unit, “orange rectangle” signifies GPU cache or shared memory. . . . . 19
2.15 Intel Gen9.5 Diagram [5]. . . . . 20
2.16 Nvidia Pascal GP100 Streaming Multiprocessor Unit [87]. . . . . 21
2.17 Nvidia Volta GV100 Streaming Multiprocessor Unit [88]. . . . . 22
2.18 CUDA Thread Hierarchy [33]. . . . . 25
2.19 CUDA Memory Hierarchy [33]. . . . . 26
2.20 SYCL Implementations [8]. . . . . 31

3.1 Comparison of different types of cross sections for $^{238}$U via JANIS 4.0 [114] with the database set to ENDF/B-VII.0 library [30]. . . . . 38
3.2 Effect of temperature on cross sections [36]. . . . . 39
3.3 Neutron histories tracked from their birth to death through a set of volumes where green cycles refer to active neutrons and red cycles refer to desactived neutrons. . . . . 43
3.4 Serial Monte Carlo neutron transport simulation workflow. . . . . 44

4.1 Inheritance relationship between base class `Nuclide` and its derived classes. . . . . 58
4.2 Inheritance relationship between base class `NeutronMedia` and its derived classes. . . . . 59
4.3 Inheritance relationship between base class `NeutronMediaNavigator` and its derived classes. . . . . 59
4.4 Scaling performance of the slabAllNuclides using the history-based method in host mode (Simulation time, the lower the better; Strong scalability, the higher the better). . . . . 78
4.5 Scaling performance of the slabAllNuclides using the pseudo event-based method in host mode (Simulation time, the lower the better; Strong scalability, the higher the better). . . . . 78
4.6 Bank size analysis of the CUDA version of slabAllNuclides in offload mode, the higher the better. . . . . 82
4.7 Scaling performance of the CUDA version of slabAllNuclides in offload mode, the higher the better. . . . . 83
4.8 slabAllNuclides performance speedup on GridCL, the higher the better - Baseline for the performance is obtained in host mode with the use of pseudo event-based method and a fixed number of bank size, 100. . . . . 86
4.9 slabAllNuclides performance speedup on Cobalt-hybrid, the higher the better - Baseline for the performance is obtained in host mode with the use of pseudo event-based method and a fixed number of bank size, 100. . . . . 87
4.10 slabAllNuclides performance speedup on Cobalt-v100, the higher the better - Baseline for the performance is obtained in host mode with the use of pseudo event-based method and a fixed number of bank size, 100. . . . . 87
4.11 slabAllNuclides performance speedup on Gorgon, the higher the better - Baseline for the performance is obtained in host mode with the use of pseudo event-based method and a fixed number of bank size, 100. . . . . 88
4.12 slabAllNuclides performance speedup on Intel NUC, the higher the better - Baseline for the performance is obtained in host mode with the use of pseudo event-based method and a fixed number of bank size, 100. . . . . 89
4.13 Performance speedup of the CUDA version of slabAllNuclides adopting different optimization strategies tested on GridCL with a single Nvidia V100 and 20 CPU threads, the higher the better. . . . . 100

5.1 An example of 3P principle of the implemented programming models including OpenMP thread, OpenMP offload, OpenACC, Kokkos, SYCL. . . . . 103

# List of Tables


2.1 Comparison of implemented programming models. . . . . 32

4.1 Typical Monte Carlo neutron transport run time percentage in PATMOS. . . . . 57

4.2 Compiler support for different programming models. . . . . 74

4.3 List of machines to run tests of slabAllNuclides benchmark. . . . . 75

4.4 Configurations of processors composing accessible machines. . . . . 76

4.5 Performance of the OpenMP thread version of slabAllNuclides using the history-based method via different thread affinity policy. . . . . 77

4.6 Comparison of the performances of slabAllNuclides in offload mode adopting different thread affinity policies tested on GridCL with a single Nvidia V100. . . . . 81

4.7 Comparison of the performances of slabAllNuclides in offload mode adopting different thread affinity policies tested on GridCL with two Nvidia V100. . . . . 81

4.8 Particle tracking rate via different programming models on a set of platforms, the higher the better. . . . . 84

4.9 Profiling of the CUDA version of slabAllNuclides in offload mode on GridCL using nvprof. . . . . 91

4.10 Metric values of the computational kernel $sigma1DopplerBroadening()$ of the slabAllNuclides CUDA version using nvvp. . . . . 92

4.11 Metric values of the computational kernel $sigma1DopplerBroadening()$ of the OpenACC, OpenMP offload, Kokkos versions of slabAllNuclides using the PEB method. . . . . 94

4.12 Comparison of the performances of the Kokkos versions of slabAllNuclides tested on GridCL with a single Nvidia V100. . . . . 95

4.13 Granularity analysis of the CUDA version of slabAllNuclides performed on GridCL with a single Nvidia V100 and 20 CPU threads. . . . . 98

4.14 Comparison of the performances of the CUDA implementations of slabAllNuclides tested on GridCL with a single Nvidia V100 and 20 CPU threads. . . . . 99

5.1 Execution time of applications $\{a_1, a_2, a_3\}$ solving a given problem across a set of platforms $\{h_1, h_2\}$, the lower the better. . . . . 117

5.2 Portability of applications $\{a_1, a_2, a_3, a_4\}$ solving a given problem across a set of platforms $\{h_1, h_2, h_3, h_4, h_5, h_6\}$. . 118

5.3 Portability of slabAllNuclides implemented via different programming models across a set of platforms. . . . . 123

5.4 Metrics of performance of different implementations of slabAllNuclides across a set of platforms. . . . . 124
5.5 Evaluation of performance portability for different implementations of slabAllNuclides across a set of platforms via the generic model $\boldsymbol{F}$ and the absolute performance metric, particle tracking rate, the higher the better. . . . . 125
5.6 Evaluation of performance portability for different implementations of slabAllNuclides across a set of platforms via the generic model $\boldsymbol{F}$ and the relative performance metric, application efficiency, the higher the better. . . . . 126
5.7 Metrics of implicit productivity for implementations of slabAllNuclides across a set of platforms. . . . . 128
5.8 Implicit productivity portability evaluated by $\boldsymbol{F}$ for implementations of slabAllNuclides across a set of supported platforms. . . . . 129
5.9 Metrics of explicit productivity for implementations of slabAllNuclides across a set of platforms. . . . . 130
5.10 Evaluation of productive performance portability for different implementations of slabAllNuclides across a subset of platforms via the generic model $\boldsymbol{F}$, $\Psi_{\text{relative}}$, the higher the better. . . . . 131

# Glossary

$F_{am}$ Arithmetic Mean Operator. 106

$F_{gm}$ Geometric Mean Operator. 106

$F_{hm}$ Harmonic Mean Operator. 107

**3P** Portability, Performance and Productivity. 102

**COMA** Cache-Only Memory Access Model. 12

**DRAM** Dynamic Random-Access Memory. 15

**DRAM** Static Random-Access Memory. 15

**EPP** Explicit Productivity Portability. 122, 129

**HB** History-based Method. 77

**IPP** Implicit Productivity Portability. 121

**LOC** Lines of Code. 113

**MIMD** multiple instruction stream, multiple data stream. 6

**MISD** multiple instruction stream, single data stream. 6

**NUMA** Non Uniform Memory Access Model. 11

**NW** Number of Words. 113

**PEB** Pseudo Event-based Method. 79

**PS-LOC** Platform-Specific Lines of Code. 113

**PS-N** Platform-Specific Program Length. 114

**PS-NW** Platform-Specific Number of Words. 113

**RDTP** Relative Development Time Productivity. 115

**RPP** Relaxed Performance Portability. 118

**SIMD** single instruction stream, multiple data stream. 6

**SISD** single instruction stream, single data stream. 6

**SPMD** single program, multiple data. 8

**SPP** Strict Performance Portability. 117

**UMA** Uniform Memory Access Model. 11

# Chapter 1

# Introduction

## 1.1 Motivations

Monte Carlo neutron transport simulation is a stochastic method which is widely used in the nuclear field for reference calculations. Instead of introducing mathematical and physical approximations to simulate a real-world system, Monte Carlo method aggregates particles behaviors through statistical sampling which significantly leads to a huge amount of computational cost. In order to alleviate this cost, the use of supercomputers for Monte Carlo simulation has become a trend for many researchers and developers.

However, as more and more modern architectures (multi-core/manycore, heterogeneous architectures) emerge in a rather fast rhythm, it is non-trivial to develop full portable applications on all these architectures, furthermore, to optimize and maintain the application continuously so as to achieve preferable performances on them.

From one side, the development and life time of an application are much greater than the time for the development of computing architectures. Thus, it makes little sense to design an application for a given architecture since after the development of the application there is a good chance that the architecture has already evolved. From the other side, it is necessary for applications to be scalable and maintainable by as many people as possible, allowing those people who are not familiar with HPC to develop the codes capable of delivering good performance. Therefore, developing applications of high performance portability becomes a hotspot to address in the field of HPC.

To meet the challenges mentioned above and gain better performance portability, some high-level programming languages such as OpenACC, OpenMP have been proposed to allow the application performing well on a wide range of architectures without large development and maintenance effort. Most of these programming models is dedicated to accelerator-based architectures with the massive increase of multi-level parallelism which represents the underlying trend in computational architectures. In this thesis, we are interested in the development of portable Monte Carlo neutron transport code targeting to hybrid architectures (CPU + GPU) with the utilization of hybrid programming models.

Concerning the evaluation of performance portability, some metrics have already been proposed to evaluate quantitatively the tradeoff between performance and portability. However, there are few researches addressing the evaluation of Monte Carlo neutron transport codes on supercomputers in terms of performance portability. Thus, we intend to give an explicit evaluation in terms of performance, portability as well as productivity for a mini-benchmark of Monte Carlo neutron transport simulation.

## 1.2 Objectives

In a word, the objective of the thesis is to evaluate existing metrics in terms of performance, portability, productivity and to propose improvements of these metrics on an example of real application performing Monte Carlo neutron transport simulation. The entire process is described as follow:

1. We choose the application case, PATMOS, which is a Monte Carlo neutron transport prototype developed at CEA.
2. We implement a mini-benchmark, slabAllNuclides on PATMOS performing a fixed source Monte Carlo simulation which is representative of the application.
3. We develop a heterogeneous offload version of the code with the utilization of the conventional history-based method and a originally proposed pseudo event-based method.
4. We implement several programming models (OpenMP thread, OpenMP offload, OpenACC, CUDA, Kokkos and SYCL) targeting for hybrid architecture (CPU + GPU).
5. To evaluate the metrics as in any performance study, we set the OpenMP thread version of slabAllNuclides as the reference on CPU and the CUDA version of slabAllNuclides as the reference on GPU. The pseudo event-based method is adopted as reference both on CPU and GPU.
6. We perform evaluation of these implementations in terms of performance, portability, and productivity with the use of metrics that we have proposed and improved. It should be noted that we are not looking for an optimal implementation on a given architecture, but the best implementation in an application set which gives the best level of performance or productive performance across a set of architectures.

## 1.3 Outline

This dissertation is dedicated to implementing several programming models on the Monte Carlo neutron transport code, PATMOS, with a heterogeneous offloading strategy and evaluating these implementations in terms of performance, portability, productivity across a set of modern architectures with the utilization of some originally proposed or modified metrics.

Chapter 2 will give an introduction of parallel computing from the hardware and software perspectives. With reference to hardware perspective, the general terminology of parallel computer and the hardware architecture that are used in the thesis (x86/OpenPower + Nvidia/Intel GPUs) will be elaborated in detail. With reference to software perspective, programming models and profiling tools that are adopted in the thesis will be discussed.

Chapter 3 will cover background information about nuclear reactor physics, neutron transport simulation, Monte Carlo method and present the state-of-the-art progress of Monte Carlo neutron transport codes in the field of HPC. Cross section and on-the-fly Doppler broadening approach are addressed since they account for most of computational cost in the cases of isotopic depletion and thermal-hydraulic coupling. Two particle tracking methods, the history-based method and event-based method are discussed as well.

Chapter 4 expresses the implementation of portable Monte Carlo neutron transport codes adopting several programming languages or libraries including OpenMP thread, OpenMP offload, CUDA, OpenACC, Kokkos, as well as SYCL. The heterogeneous offloading strategy and the originally proposed particle tracking method, pseudo event-based method are introduced in detail. Numerical results and performance analysis of these implementations will be given along with some achieved optimizations of the CUDA version.

Chapter 5 focuses on the establishment of a generic model and the evaluation of implementations in terms of performance, portability, and productivity across a set of architectures with the use of this generic model. The best implementation obtaining the best level of performance or productive performance will be given.

Chapter 6 draws conclusions from the previous chapters and introduces some work for future development which may contain testifying our generic model with much more different cases widely used in the field of scientific computing, optimizing the Kokkos and SYCL versions of implementations as the future releases of Kokkos and SYCL may enable more support for our heterogeneous offloading strategy.

# Chapter 2

# Parallel Computing

The thesis work is based on the development of parallel code for Monte Carlo neutron transport simulation. Therefore, we begin with the literature of thesis by giving an introduction of parallel computing from both the hardware and software perspectives. It mainly consists of the basic information about parallel computing and some hardware architectures, programming models as well as profiling tools that are used in research work.

## 2.1 General Terminology

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously [50]. Instead of dividing a problem into a group of instructions and executing them sequentially, parallel computing firstly breaks the problem into several sub-problems and carries out them concurrently.

Nowadays, the demand of parallel computing has reached to a very high level. With the blowout of big data, numerical simulation needs a great amount of processing time which is far beyond the capacity of any single compute resource. Thus, more and more parallel computing are used in a wide range of scientific and industrial domains for the purpose of exploring the potential of underlying parallel hardware, making better use of dispersed compute resource, and eventually saving the cost of time and money. In what follows some general parallel terminology will be introduced to make better acknowledge of parallel computing.

### 2.1.1 Von Neumann Architecture

The *von Neumann architecture* is the fundamental design of modern parallel computers proposed by the Hungarian mathematician and physicist John von Neumann in his 1945 paper [123]. This stored-program design allows instruction and data being stored in electronic memory. Its main components are illustrated in Figure 2.1.

- **Central Processing Unit (CPU)**: An electronic circuitry which executes instructions of a computer program. It contains a processing unit and a control unit.

  * **Processing Unit (PU)**: It is comprised of an arithmetic logic unit (ALU) and a variety of registers. ALU allows for arithmetic and bitwise operations on input data and returns the result as output. Registers are fast storage areas which provide the fasted way to access data.
  * **Control Unit (CU)**: A component of CPU that fetches and decodes instructions and directs the operations of the processor. It is comprised of a current instruction register (CIR) and a program counter (PC). The CIR is responsible for holding the current instruction during processing. The PC contains the address of the next instruction to be executed and increments itself after fetching an instruction.

- **Memory Unit**: It consists of random access memory (RAM) which stores data and instructions.
- **Input/Output**: Input and output mechanisms.

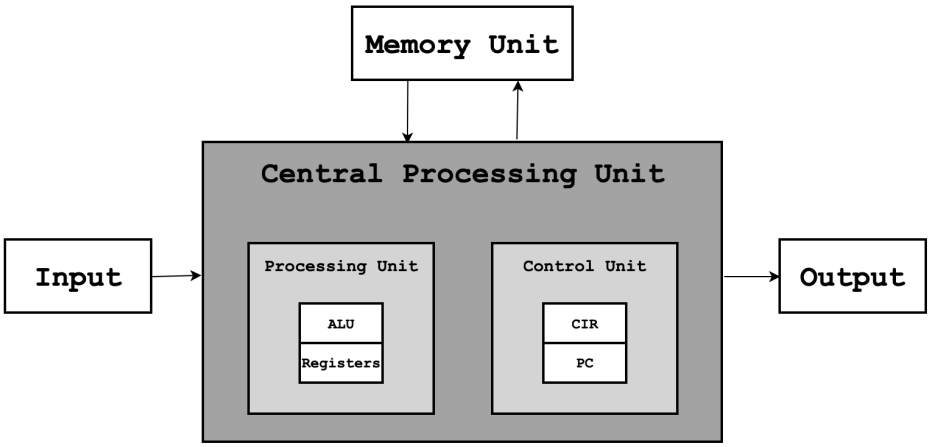



Figure 2.1: Von Neumann Architecture.

One of the key features of the *von Neumann architecture* is that both data and instructions are designed to share common bus to communicate between central processing unit and memory unit. Thus, data operation and instruction fetch cannot be executed simultaneously. This drawback is referred to as the *von Neumann bottleneck* and usually limits the performance of the system significantly [82]. Two strategies are mostly adopted by modern parallel computers to mitigate this bottleneck at memory hierarchy level, which are respectively adding internal memory layer such as cache to communicate with central processing unit and providing separate storage and pathways for data and instructions.

### 2.1.2 Flynn's Taxonomy

*Flynn's Taxonomy* is classification of parallel computing machines proposed by Michael J. Flynn in 1966 [46]. This classification is based on distinguishing the multiplicity (single/multiple) of instruction and data streams. As a result, there are four sub-categories according to Flynn's taxonomy:

- **SISD**: single instruction stream, single data stream
- **SIMD**: single instruction stream, multiple data stream
- **MISD**: multiple instruction stream, single data stream
- **MIMD**: multiple instruction stream, multiple data stream

Figure 2.2 show the workflow of instruction and data stream during program execution for the computer. Instruction stream refers to the flow of instructions fetched from main memory and operated by CPU. Data stream is the flow of operands loaded or stored between main memory and CPU. The multiplicity of stream has two states, single or multiple, indicating that if there is only one stream or multiple streams during any one clock cycle.

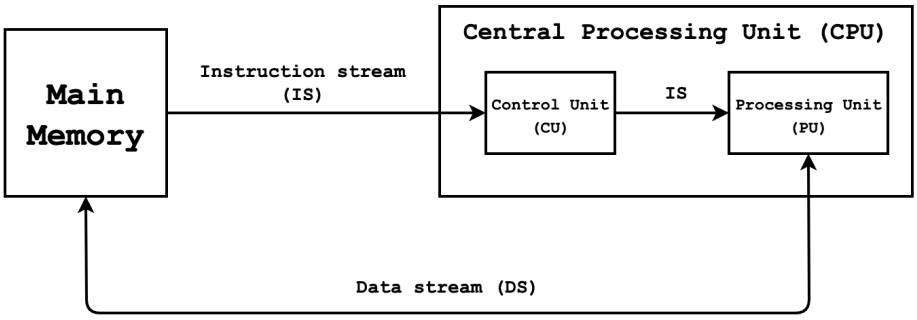



Figure 2.2: Instruction and Data Stream.

#### 2.1.2.1 SISD organization

SISD computing machines are serial computers which handles only one stream of instructions and data at any point in the execution. Some old type of computers such as CRAY1 and IBM360 adopt this organization. The diagram of SISD organization is illustrated in Figure 2.3.

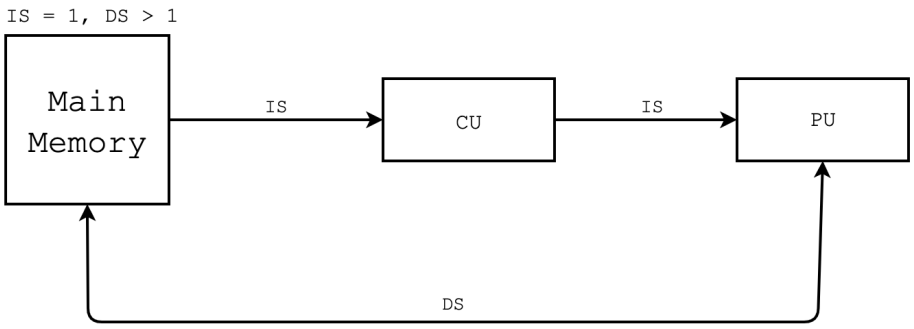



Figure 2.3: SISD Organization.

#### 2.1.2.2 SIMD organization

SIMD organization allows multiple processing units handling the same instruction broadcast from one control unit. Meanwhile, all processing units take different data streams to accomplish arithmetic operations simultaneously. The details of SIMD organization is depicted in Figure 2.4.

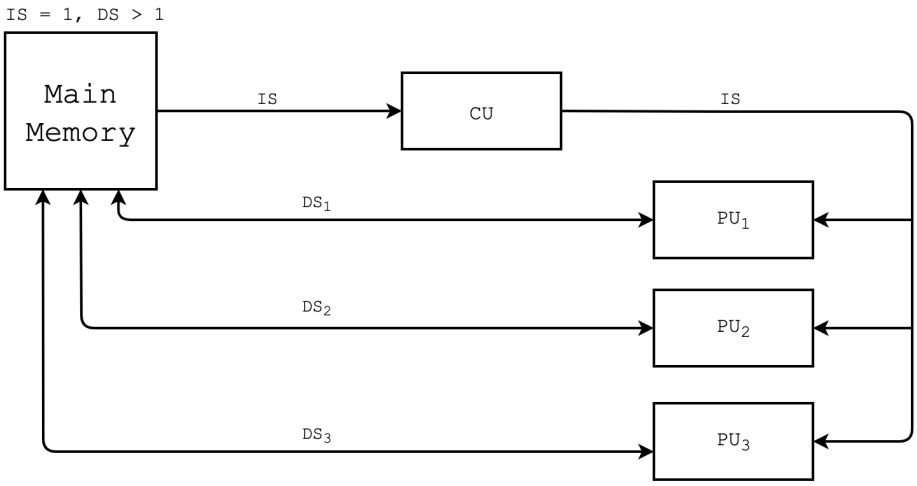



Figure 2.4: SIMD Organization.

Some parallel computers designed via SIMD organization are IBM 9000, Cray X-MP, ILLIAC-IV and Thinking Machines CM-2. Furthermore, SIMD organization is widely employed in most parallel computers as specific instructions and execution units to achieve SIMD vector operation such as vector processing unit (VPU) and graphics processing unit (GPU).

#### 2.1.2.3 MISD organization

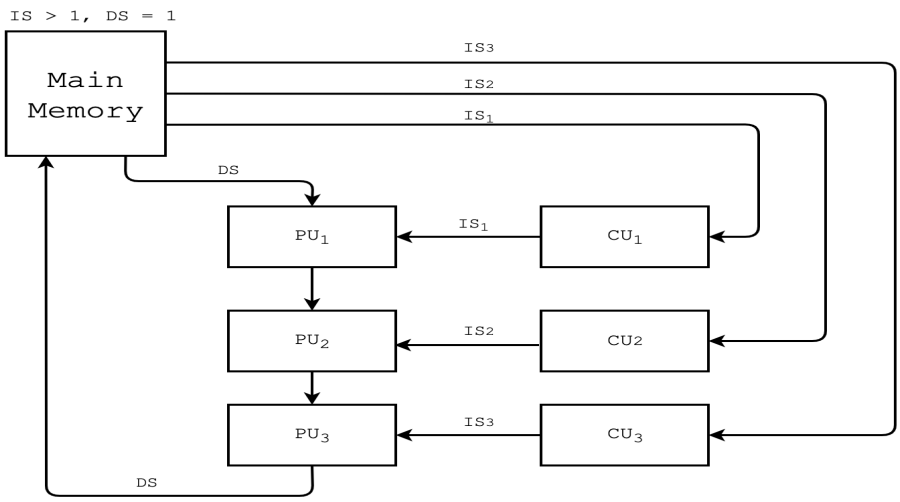



Figure 2.5: MISD Organization.

MISD machine operates multiple instruction streams and single data stream with multiple control units and processing units. Each control unit handles one instruction stream and sends it to the corresponding processing unit. All processing units share a single data stream to complete execution as shown in Figure 2.5. There are few examples of MISD machines that have ever existed.

#### 2.1.2.4 MIMD organization

MIMD organization is the most popular for a parallel computing machine. Most modern supercomputers, networked grids and all multiprocessor systems fall into this classification, such as IBM POWER5, Intel IA32, Cray XT3, etc. In contrast to MISD organization, in which all processing units are fed with only a single data stream, each processing unit in MIMD organization can handle a different data stream. Since there is no data and instruction dependencies among processors, executions on different processors can be asynchronous. The diagram of MIMD organization is denoted in Figure 2.6.

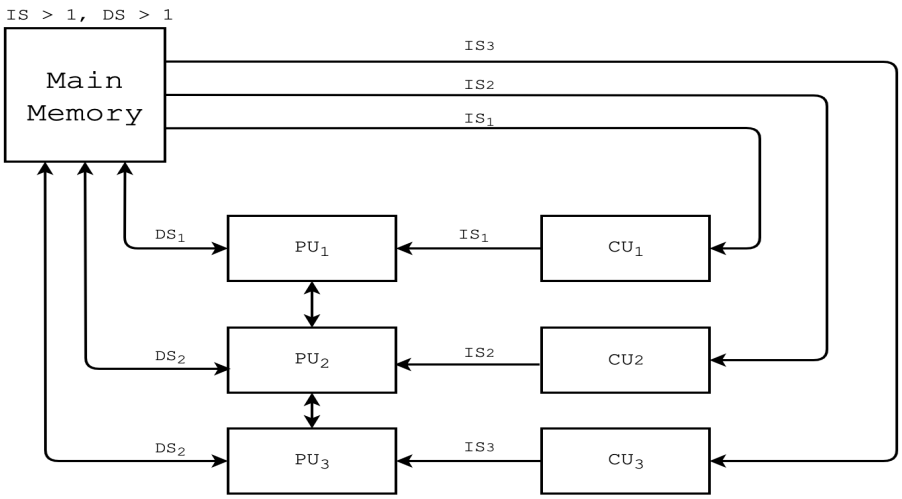



Figure 2.6: MIMD Organization.

Furthermore, MIMD contains a subcategory called as SPMD (single program, multiple data) [39]. It allows a single program to be executed simultaneously by multiple processors with different input, which usually refers message passing programming (section 2.4.1.1) on distributed memory architectures (section 2.2.2).

### 2.1.3 Parallel Speedup

#### 2.1.3.1 Amdahl's law

Amdahl's law [16] is a formula indicating the potential speedup of an algorithm which can be expected by parallel computing, as illustrated in Equation 2.1:

$$Speedup(P, N) = \frac{1}{\frac{P}{N} + 1 - P} \tag{2.1}$$

where:

- $P$: The proportion of run time of a given algorithm that can be parallelized.
- $N$: The number of processors performing the program.
- $P \in [0, 1]$:
  - if $P = 0$, then $Speedup = 1$, meaning that the algorithm has no parallel fraction and cannot be parallelized to achieve performance speedup.
  - if $P = 1$, then $Speedup = N$, signifying that the optimal speedup of algorithm is N. With the increase of number of processors, the speedup can be infinite in theory.
- $N \in [1, +\infty)$:
  - if $N = 1$, then $Speedup = 1$, the code is executed in serial mode.
  - if $N \to +\infty$, then $Speedup = \frac{1}{1-P}$. This is a derived version of Amdahl's law, assuming that the execution time of parallel fraction can be reduced to zero with multiple processors. This theoretical achievable speedup is considered as the upper performance limit of parallelization of the given algorithm.

Amdahl's law assumes the parallelizable part of an algorithm $P$ is a constant. However, this assumption does not always fit into real-world problems. In practice, the parallel fraction $P$ often varies with the change of problem size. Considering of this factor, we can have another derived version of Amdahl's law as shown in Equation 2.2:

$$Speedup(P(W), N) = \frac{1}{\frac{P(W)}{N} + 1 - P(W)} \tag{2.2}$$

where:

- $W$: The workload of a given problem.
- $P(W)$: The parallelizable proportion of execution time of a given problem with workload $W$.

Unfortunately, $P(W)$ is not a predictable function varying via specific rules. To overcome this shortcome, we can consider in another way. Instead of fixing the problem size and comparing the execution time, it is possible to firstly define a fixed execution time and then to calculate the speedup by comparing the achieved workload of program in parallel and serial modes.

#### 2.1.3.2 Gustafson's law

Gustafson's law was proposed by John L. Gustafson and his colleague Edwin H. Barsis in the article "Reevaluating Amdahl's Law" in 1988 [52]. As shown in Equation 2.3, Gustafson's law assumes that the total parallel workload varies linearly with the number of processors. It estimates the theoretical attainable speedup of a program by comparing the workload of the parallel version of the code $W_p$ with $N$ processors to the workload of the serial version of the code $W_s$ at a fixed execution time $T$.

$$\begin{aligned} W_s &= (1-P)W_s + PW_s; \\ W_p &= (1-P)W_s + NPW_s; \\ Speedup(N) &= \frac{W_p}{W_s} = 1 - P + NP; \end{aligned} \tag{2.3}$$

where:

- $W_s$: The workload of the code in serial mode at a fixed execution time $T$.
- $W_p$: The workload of the code in parallel mode with $N$ processors at a fixed execution time $T$.
- $P$: The parallelizable fraction of the code at a fixed execution time $T$.

### 2.1.4 Scalability

In the context of parallel computing, scalability refers to the property of a system to handle a growing amount of work by adding resources to the system [23]. There are two common types of scaling:

- **Strong scaling**: The strong scaling indicates the scalability tested under the condition that the time to solution varies with the number of processors for a fixed total problem size.
- **Weak scaling**: The weak scaling describes the scalability evaluated under the condition that the time to solution varies with the number of processors for a fixed problem size per processor.

There are many factors which influence the scalability of a parallel program. For example, the algorithm used in the program often contains some inherent limits which lead to performance degradation with the increase of compute resources. The memory capacity, memory bandwidth or network bandwidth of a parallel computing machine may also be the bottleneck of scalability.

## 2.2 Memory Architectures

Parallel computing machines can be separated into three types in the context of memory: shared memory architecture, distributed memory architecture and hybrid memory architecture. Processors in shared memory architecture share the global memory modules while processors in distributed memory architecture have their own local memory and communicate with each other via a network. Hybrid memory architecture is the mix of shared and distributed memory architectures. These memory architectures have their own advantages and disadvantages, we will dig into them in the following part.

### 2.2.1 Shared Memory

Shared memory is a global address space which can be accessed by all processors. The key advantage of shared memory architecture is that multiple processors have fast access to data in memory location, leading to high memory bandwidth. Meanwhile, such fast access to memory incurs two disadvantages. First, the resource contention and false sharing usually happen when several processors try to access the same or nearby memory locations. When adding more processors to shared memory architecture, the performance degradation will become more and more serious. Second, since several processors may handle data load/store operations simultaneously, data coherence must be kept to ensure that different processors always use the “correct” data that has just been updated by other processor. Such obligation for data coherence is quite a bottleneck to performance.

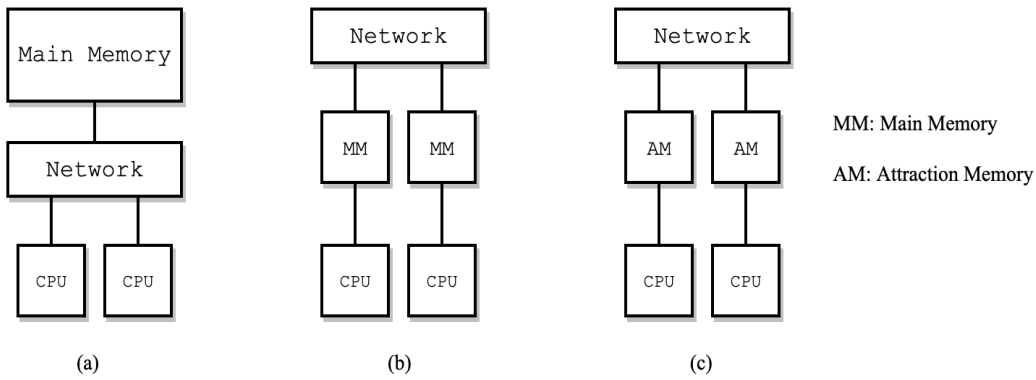



Figure 2.7: Shared Memory Architecture.

The shared memory architecture can further be divided into three models which are based on different manners of memory access, as listed below:

- **Uniform Memory Access Model (UMA)**: All processors in UMA are identical. They have equal access and equal access time to shared memory, as shown in Figure 2.7 (a).
- **Non Uniform Memory Access Model (NUMA)**: Global memory is not accessed uniformly by all processors. Some processors attached to the same local memory have equal access time. However, the access to a

non-local memory across link for these processors is slower. The NUMA model is described in Figure 2.7 (b).

- **Cache-Only Memory Access Model (COMA)**: As illustrated in Figure 2.7 (c), COMA model is a specific NUMA model where its local memories are used as cache (attraction memory) instead of main memory. The difference between main memory and attraction memory is that when a data is accessed by non-local processors, main memory keeps the space of data and gives a copy of data to processors to operate. On the contrary, attraction memory migrates the data to wherever it is needed.

### 2.2.2 Distributed Memory

In distributed memory architecture, processors have their own local memory instead of sharing a global address space. The separate memories are connected by message passing interconnection network such as Ethernet. When a processor needs to access remote data in another processor, it uses the network link to communicate with the remote processor. The diagram is depicted in Figure 2.8:

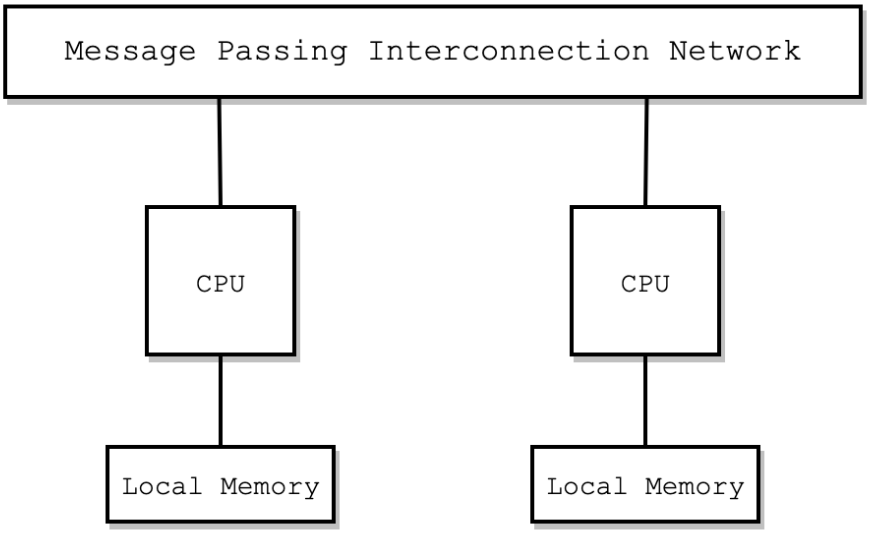



Figure 2.8: Distributed Memory Architecture.

Generally speaking, the key advantage of distributed memory architecture is that the problems of memory conflicts (for example memory contention, race condition, cache coherence) can be alleviated since each processor has its own private memory. However, the data access to remote memory consumes longer time than the access to local memory, which may degrade significantly the performance if it happens too much inter-communications during the execution of a program. Another shortcome of distributed memory is that programmers need to explicitly define the mechanism of communications and synchronizations of processors for a given code, which augments much programming complexity.

### 2.2.3 Hybrid Memory

Hybrid memory architecture, also called distributed-shared memory architecture, is widely employed by modern high performance computers. As Figure 2.9 shows, hybrid memory architecture consists of a group of shared memory machines which are connected to each other via inter-connection network. A parallel computing machine designed in hybrid memory architecture scales well with the increase of shared memory components, which makes it easier to exploit the computing resources at the cost of increased programming complexity.

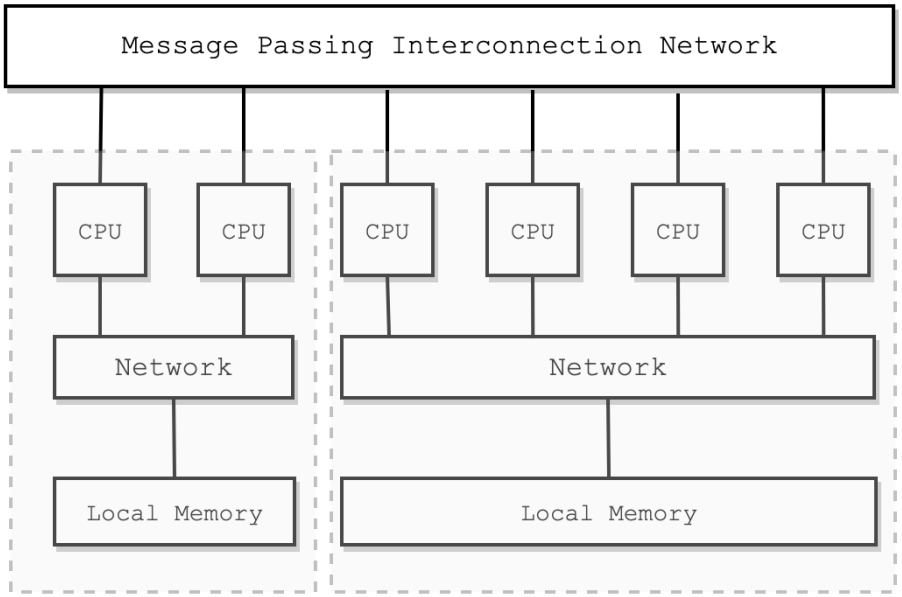



Figure 2.9: Hybrid Memory Architecture.

## 2.3 Computing Architectures

The current computer market is full of different hardware architectures produced by many vendors such as INTEL, NVIDIA. These architectures are designed for solving different problems and each product has its advantages and defects. In order to explore maximal computing capacity of a targeting architecture and learn about the performance bottlenecks on this architecture for a given code, it is required to get better knowledge of the targeting architecture. Therefore, we will express in detail several computing architectures used in the thesis work.

### 2.3.1 Hardware Characteristics

We start from highlighting some features which are widely used to characterize hardware capacity.

- **Latency**: refers to the time it takes for the execution of an operation. Its conventional unit of measurement is `microseconds`.

- **Frequency**: indicates the number of electronic pulses generated by a processor to synchronize operations of its components per second. It can also be called "clock rate" with the units of measurement `MHz` or `GHz`.
- **Bandwidth**: signifies the amount of data which can be handled per unit of time. `gigabytes/sec`, `megabytes/sec`, or `bytes/cycle` are widely used as units of measurement.
- **Throughput**: denotes the amount of floating-point operations which can be processed per unit of time. `GFLOPS` is a common unit of measurement.

In a word, the evolution of computing architectures is mainly to design a hardware with lower latency, higher frequency, higher bandwidth and higher throughput.

### 2.3.2 CPU Architectures

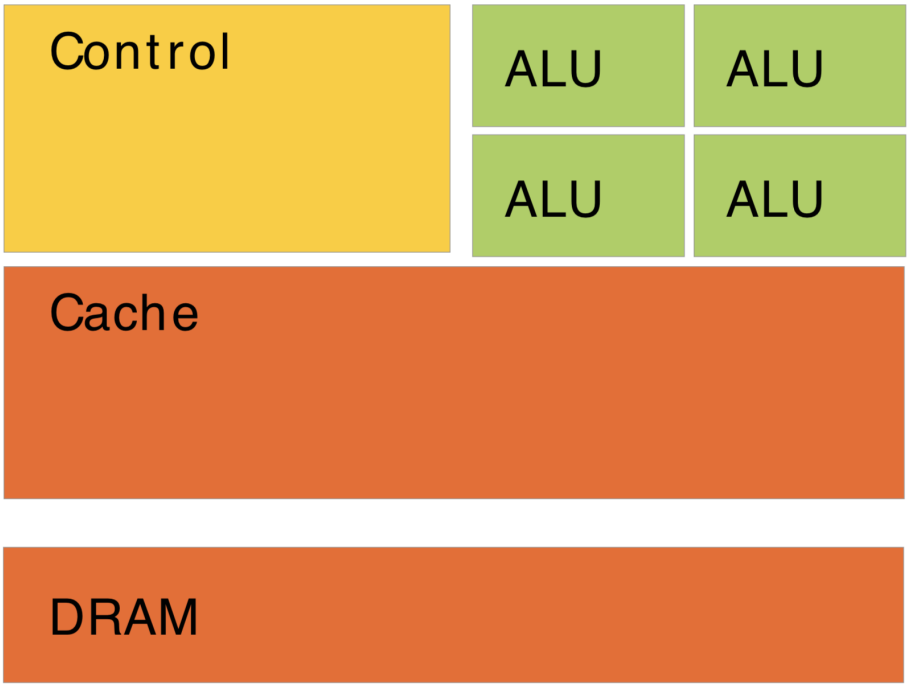



Figure 2.10: Modern CPU Architecture [90].

CPU is the electronic circuitry within a computer that executes a computer program by performing the basic arithmetic, logical, control and input/output(IO) operations specified by instructions [130]. It is a general purpose processor designed to perform a bunch of operations integrating the arithmetic and control operations while its performance of these tasks is not good enough. Unlike the traditional structure of CPU as we illustrated in section 2.1.1, where its components refer to processing unit and control unit, modern CPUs include memory units such as cache and DRAM as depicted in Figure 2.10:

- **Cache**: An intermediate memory layer made of SRAM which has higher bandwidth and lower latency compared with main memory.
- **DRAM**: Dynamic random-access memory is primarily used as main memory. It is comprised of an array of memory cells in which a transistor-capacitor circuit is used to store one bit of data based on its state of charge. Due to capacitor leakage, DRAM needs to perform refresh process, which is a key difference to SRAM and makes DRAM much slower than SRAM.
- **DRAM**: Static random-access memory uses multiple transistors in a latching circuit to store each bit of data. Since it makes no use of transistors, SRAM does not need to be refreshed and it has higher efficiency on data transfer comparing to DRAM.

#### 2.3.2.1 Intel Broadwell

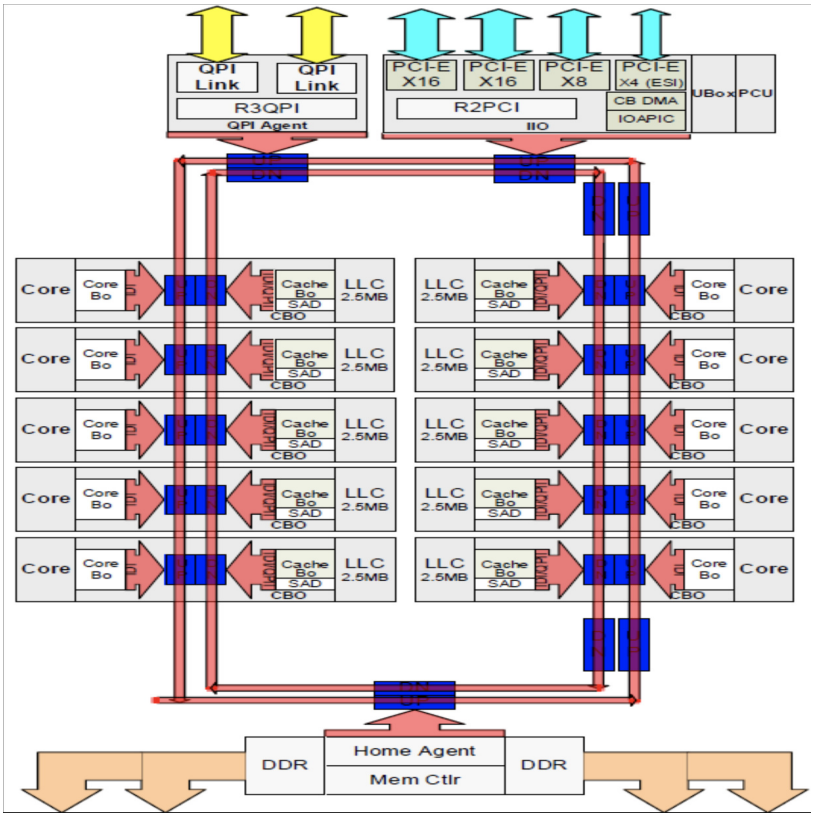



Figure 2.11: Broadwell Die Diagram [4].

Intel Broadwell is the fifth model generation of Intel Processor released in 2015 with 14nm die shrink of Haswell microarchitecture [86]. Based on 14nm process technology, Broadwell reduces 49% feature neutral area and gains better power efficiency due to improved Vmin (minimum operating voltage). Comparing to Haswell microarchitecture, Broadwell introduces some new instruction set extensions including `RDSEED`, `PREFETCHW`, `SMAP` and `Intel ADX`. There

are several instructions that are improved for lower latency in Broadwell such as `FP MUL` instructions and `CLMUL` instructions. Broadwell also enlarges instruction queue, scheduler and second-level TLB for higher throughput.

Figure 2.11 denotes the die diagram of a Broadwell microarchitecture with 10 cores. As we can see, the diagram can be divided into two main areas: *Uncore* and *Core*. *Uncore* is comprised of multiple modules designed specifically for the server and workstation market space [62], such as LLC (Last Level Cache), Cbo (Caching Agent, including Cache Box and Core Box), QPI (Intel QuickPath Interconnect), HA (Home Agent), iMC (Integrated Memory Controller), IIO (Integrated I/O module), PCU (Power Control Unit) and UBox (Config Agent). *Core* contains mainly the cores executing instructions and operations as well as two levels of caching hierarchy (L1 and L2).

#### 2.3.2.2 Intel Skylake

Skylake is the $6^{th}$ generation of Intel microarchitecture succeeding the Broadwell microarchitecture with 14nm process (client) or 14nm+ process (server) technologies. The modified 14nm+ process has promoted transistor channel strain which leads to higher frequency of processor ($100 \sim 300$ MHz) and better performance. Unlike Skylake (client) which adopts ring topology on-die interconnect firstly introduced in Sandy Bridge architecture [137], Skylake (server) implements a new networking topology called mesh interconnect architecture that mitigates the scaling bottlenecks of latency and bandwidth as the number of cores increases in ring interconnect architecture. These two types of Skylake machines are both used in the thesis work.

Note that Skylake introduces a number of new instructions in which the advanced vector extension using 512-bit SIMD techniques (AVX-512) [101] is one of the most important feature contributing to parallel computing. The AVX-512 instruction set in Skylake is composed of five separate sets:

- **AVX512F**: The fundamental of the 512-bit SIMD instruction extensions which is required for any implementations of AVX-512 extensions on microarchitectures. It is the extensions to AVX/AVX2 instruction sets using EVEX prefix so as to support 512-bit registers, arithmetic operations, bitwise operations, data management, parameter broadcasting, etc.
- **AVX512CD**: AVX512CD instruction set is designed specifically for conflict detection. Loops without address conflict can be detected and then vectorized to improve performance.
- **AVX512BW**: BW refers to bytes (8-bit) and words (16-bit). This instruction set extends AVX-512 extensions to cover 8-bit and 16-bit integer operations [102].
- **AVX512DQ**: DQ represents doubleword (32-bit) and quadword (64-bit). AVX512DQ instruction set introduces some new 32-bit and 64-bit operations such as conversion and transcendental support.
- **AVX512VL**: One of an additional AVVX-512 instruction set dedicated to the compatibility of AVX-512 operations on registers with different vector length. For instance, XMM (128-bit SSE) registers and YMM (256-bit AVX) registers.

#### 2.3.2.3 IBM POWER9

IBM POWER9 is a 14nm microarchitecture based on Power ISA (instruction set architecture) v3.0[61] firstly released in 2017. The key component of POWER9 architecture is the 4-way or 8-way simultaneous multithreading core, as SMT4 core and SMT8 core.

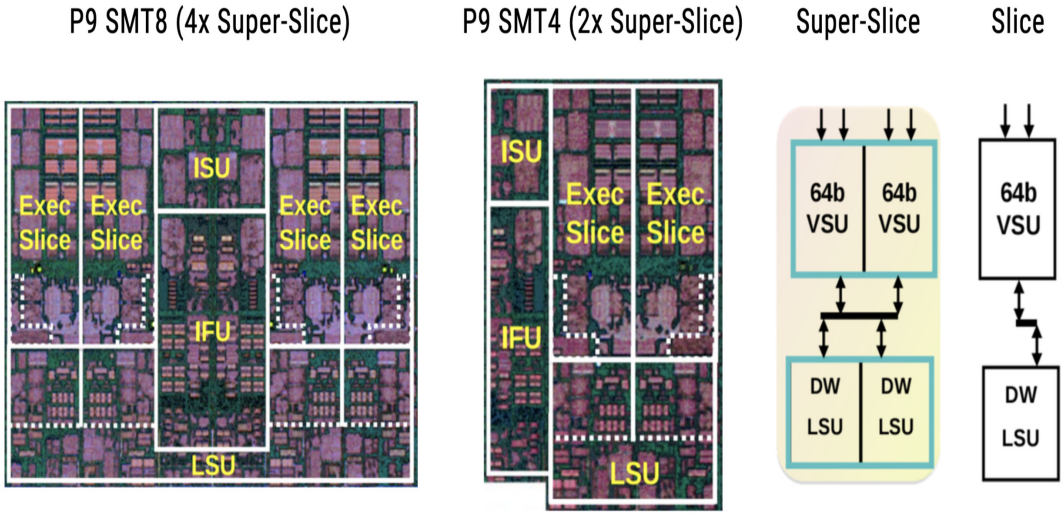



Figure 2.12: POWER9 SMT Core Diagram [7].

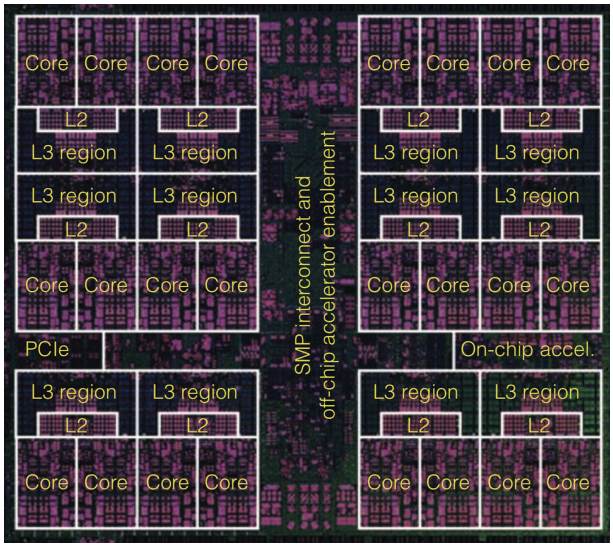



Figure 2.13: POWER9 SMT4 Chip Diagram where the cache hierarchy is illustrated [108].

Figure 2.12 shows the building blocks of POWER9 core. As we can see, a SMT core is composed of 1 or 2 super-slices (SMT4 or SMT8), along with an instruction fetch unit (IFU) and an instruction sequencing unit (ISU). Each super-slice combines 2 slices together where each slice consists of a 64-bit vector and scalar unit (VSU) coupled with load/store unit (LSU). Each slice is able to handle an address generation (AGEN) operation and a 64-bit scalar operation including either a computational operation (fixed-point/floating-point) or a load/store operation

at any given cycle. A super slice can be considered as a vector processing unit issuing two AGEN operations and a 128-bit vector operation every cycle. Besides, a POWER9 SMT core also contains 4 double-precision floating-point units (DFUs), which allows for issuing double-precision floating-point multi-add/divide/square root instructions every cycle. This design increases POWER9 CPU's double-precision throughput.

The POWER9 processor has a 10MB eDRAM-based L3 cache chunk per SMT8 core (or shared between 2 SMT4 cores), as denoted in Figure 2.13. Different L3 cache chunks are linked by fabric interconnect to allow cores sharing one L3 cache chunk accessing data residing in other L3 chunks. The POWER9 L2 cache is similar to L3 cache as each 512KB L2 cache is associated with a SMT8 core or two SMT4 cores. Moreover, the POWER9 processor has a 32KB L1 instruction cache in conjunction with a 32KB L1 data cache per SMT4 core.

### 2.3.3 GPU Architectures

GPU (graphics processing unit) is originally a specialized electronic circuit dedicated to rapidly handle image processing computations for output to a display device. Back then, it was widely used in personal computers, workstations or game consoles, only dealing with graphical operations. However, with more and more emergence of programmable GPUs and programming interfaces, GPU's highly parallel architecture has been found more performance-efficient than CPU architecture performing compute-intensive and data parallel calculations. Thus, a new software concept called "general-purpose computing on graphics processing unit" has been proposed and largely developed which allows GPUs handling general-purpose applications in different domains such as geometric computing, bioinformatics, computational finance, scientific computing and so on [34].

As denoted in Figure 2.14, GPU contains thousands of GPU cores settled massively in parallel so as to hide latency and maximize throughput. Comparing GPU core with CPU core, GPU core is designed for data-parallel computations with simple control logic whereas CPU core is rather heavy-weight, aimed at handling efficiently complex control logic such as an algorithm composed of multiple conditional operations (`if...else`).

Regarding to GPU caches, state-of-the-art GPUs are being designed with sizable hardware-managed multi-level caches, such as L1 cache, L2 cache and texture cache [85]. The cache memory has been enlarged with the evolution of modern GPUs. For example, the Nvidia Tesla Volta GPU [88] has 6MB last level cache while the latest released Nvidia Ampere Tensore Core GPU [92] has 40MB last level cache. Note that the last level cache size of Nvidia Ampere architecture is competitive with that of Intel Skylake architecture containing up to around 30 cores. But if we consider the average cache memory which can be used for each core, Nvidia Ampere has 5KB last level cache per core and that of Intel Skylake is 1.375MB per core. We can say that GPU is composed of tens of hundreds of cores with relatively small caches. On the contrary, CPU is composed of tens of cores with relatively large caches.

Furthermore, modern GPUs consume significant amount of power as the cost of performance optimization. For example, the Nvidia Tesla Pascal GPU [87] has a TDP (thermal design power) level of 300W for maximum

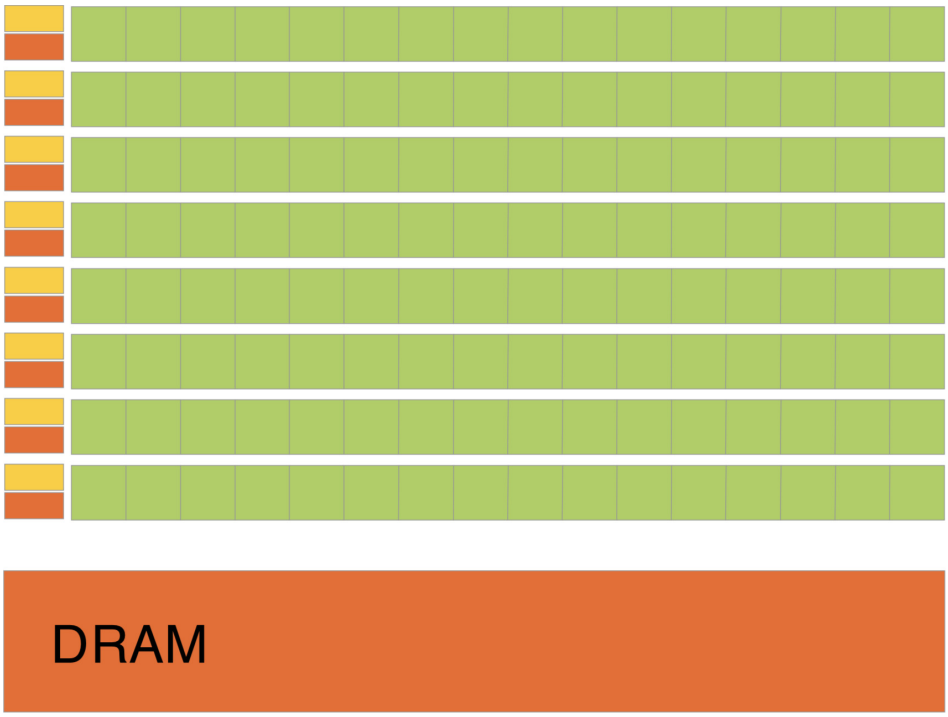



Figure 2.14: Modern GPU Architecture [90], where “green square” refers to GPU core, “yellow rectangle” is GPU control unit, “orange rectangle” signifies GPU cache or shared memory.

performance. In contrast, Intel Xeon Gold 6148 has a maximum power consumption of 150W. Since more and more GPUs have been widely used in high-performance computing (HPC) machines, the optimization of GPU power efficiency has become a research hotspot to make supercomputers in future more friendly in economy and energy. There are some research work that has been carried out to optimize power efficiency of GPUs from a hardware perspective, such as saving static energy in L1 and L2 caches [128], using filter-cache to reduce accesses to instruction cache [72] and so on. From a developer perspective, enhancing power efficiency of GPUs for a specific application is mainly to optimize codes so as to make maximal use of computing resources.

In the following part of this section, we will introduce several GPU microarchitectures related to our research work.

#### 2.3.3.1 Intel Gen9.5

Intel Gen9.5 is the 14nm+ microarchitecture of Intel’s integrated graphics processing unit (IGPU) used in several Intel’s CPU microarchitecture such as Kaby Lake, Coffee Lake, etc. Unlike dedicated graphics cards which have their own RAM, IGPUs share the system RAM with the CPU.

Figure 2.15 shows the main components of Intel Gen9.5. Overall, Intel Gen9.5 is comprised of three parts: unslice, display, as well as slice. Among them, slice is responsible for calculations. It is a cluster of subslices. Each subslice consists of 6 or 8 EUs which are dedicated to execute 3D shaders and computational kernels. Based on different scaling models, Intel Gen9.5 is allowed to have maximum 72 EUs and minimum 12 EUs. Each EU is

composed of 7 threads sharing the same instruction fetch unit and piplined to 4 instruction issue units.

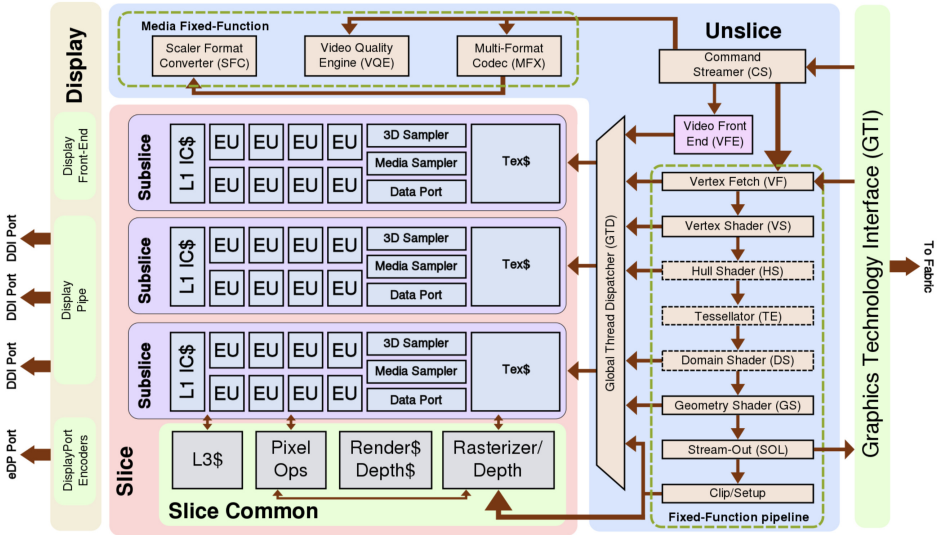



Figure 2.15: Intel Gen9.5 Diagram [5].

### 2.3.3.2 Nvidia Pascal

Nvidia Pascal microarchitecture was firstly introduced in 2016 with the release of Tesla P100 card. As the successor to Nvidia Maxwell microarchitecture, GP100 architecture provides some improvements such as enhanced peak throughput (For Tesla P100, 5.3 TFLOPS in double-precision, 10.6 TFLOPS in single-precision). The number of FP32 (single-precision floating-point) cores and FP64 (double-precision floating-point) cores in GP100 architecture are respectively 3840 and 1920. Thus, the ratio of FP32 units to FP64 units for GP100 is 2. By comparison, the ratio of FP32 units to FP64 units in Maxwell architecture is 32. Furthermore, GP100 can handle FP16 (half-precision floating-point) operations which deliver great speedups for many deep learning algorithms.

With respect to memory, the overall memory size of GP100 is enlarged including register, L2 cache and shared memory. HBM2, as the second generation of High Bandwidth Memory, is firstly adopted in GP100 architecture which offers three times higher memory bandwidth compared with the Maxwell GM200 GPU, in which GDDR5 (graphics double data rate type 5) memory is used. GP100 also use a new high-speed interconnect technology called NVLink to increase performance for both CPU-to-GPU and GPU-to-GPU inter-communications as a substitution of PCI Express (PCIe) [9]. The NVLink implementation in GP100 (NVLink 1.0) delivers up to 160GB/s bidirectional bandwidth with maximum 4 links. Besides, GP100 adds two hardware features to improve unified memory functionality: 49-bit virtual addressing support and page faulting mechanism. Due to these features, GP100 unified memory offers a system-wide virtual address space and it can be accessed simultaneously by CPU and GPU.

A full GP100 is composed of 60 streaming multiprocessors (SMs), a 4096KB L2 cache, eight 512-bit memory controllers, four 4096-bit HBM2 stacks and 4 NVLinks. The number of SMs may vary for different GP100 products.

For example, the Tesla P100 only enables 56 SMs. Each SM contains an instruction cache, a texture/L1 cache, 4 texture units, a 64KB shared memory and 2 processing blocks. Each processing block is comprised of 32 FP32 cores, 16 FP64 cores, an instruction buffer, a warp scheduler, 2 dispatch units and a 128KB register file, as illustrated in Figure 2.16.

Figure 2.16: Nvidia Pascal GP100 Streaming Multiprocessor Unit [87].

GP100 and earlier Nvidia GPUs employ a SIMT (single instruction, multiple threads) model which resembles in the SIMD model allowing multiple execution units fetching the same instruction and processing with multiple data. The SIMT model groups a set of 32 threads as a "warp" in which all threads execute the same instruction at a given cycle for parallel processing. A key feature of SIMT model which SIMD doesn't have is that all threads in a warp are allowed to have divergent execution paths. This feature enables Nvidia GPUs to deal with thread-level branch divergences at the cost of performance degradation. Moreover, because all threads in a GP100 warp shares the same program counter and call stack, the execution of divergent branches are serialized. During the execution, one portion of threads in the warp become active to complete `if` branch, then the other portion of threads within a warp change their state from inactive to active and handle `else` branch. After the end of `else` branch, the warp reconverges to the original state. Such behavior may yield false result or deadlock for several active threads in `if` branch requiring data which should have been updated or guarded by other threads in `else` branch. Hence, GP100 is unfriendly to handle thread-aware algorithms with data dependency among threads in a warp.

#### 2.3.3.3 Nvidia Volta

Nvidia Volta is the microarchitecture succeeding Pascal architecture firstly released in 2017 with the Tesla V100 product. A full GV100 consists of 84 SMs, a 6144KB L2 cache, eight 512-bit memory controllers, four 4096-bit HBM2 stacks and 6 NVLinks. Note that the Tesla V100 uses 80 SMs.

As we can see in Figure 2.17, each SM is composed of a L1 instruction cache, a 128KB combined L1 data cache and shared memory subsystem where shared memory size can be set up to 96KB, 4 texture units as well as 4 processing blocks. Each processing block is comprised of 16 FP32 cores, 8 FP64 cores, 16 INT32 cores, 2 mixed-precision tensor cores (capable of handling FP64, FP32, FP16 operations), a L0 instruction cache, a warp scheduler, a dispatch unit and a 64KB register file. In total, a Tesla V100 contains 5120 FP32 cores, 2560 FP64 cores, 5120 INT32 cores, 640 tensor cores, 320 texture units and 20MB register file size.

Figure 2.17: Nvidia Volta GV100 Streaming Multiprocessor Unit [88].

Comparing to GP100 architecture, GV100 has many advances from both hardware and software perspectives. Since L1 data cache is merged with shared memory, L1 cache in GV100 benefits from the low latency and high bandwidth of shared memory. Therefore the performance gap between a code with and without the use of shared memory is significantly narrowed. GV100 supports 6 NVLinks 2.0 delivering in total 300 GB/s bidirectional bandwidth which is around $2\times$ higher than the interconnect bandwidth of GP100. The HBM2 memory used in GV100 can yield maximum 900 GB/s memory bandwidth which is higher than 700 GB/s in GP100. Besides, GV100 add a new

"independent thread scheduling" feature to SIMT model, where each thread within a warp maintains its own program counter and call stack so as to keep an independent execution state. Due to this new feature, statements from `if` branch can be interleaved with statements from `else` branches and vice versa, making GV100 capable of carrying out thread-aware algorithms where data-dependent executions are required among threads in a warp. With respect to unified memory in GV100, it is improved by the introduction of a new Access Counter and the Address Translation Services (ATS) support over NVLink.

### 2.3.4 Heterogeneous Architecture

Heterogeneous architecture refers to a hybrid architecture combined through memory bus (PCIe, NVLink) with CPUs as the host and some processing units as the devices, including GPUs, DSPs (digital signal processors) [48], FPGAs (field-programmable gate arrays) [35] and so on. A host is responsible for initializing the environment and launching the application. A device, also called as an accelerator, is dedicated to handle the compute-intensive parts of the application so as to enhance the performance of application. The calculations running on a heterogeneous architecture is called heterogeneous computing, which is a popular trend in HPC domain. The most widely used heterogeneous architecture is abstracted as CPUs + GPUs, including many different combinations such as Intel Broadwell CPUs + 2 Nvidia P100 and IBM Power9 CPUs + 4 Nvidia V100.

## 2.4 Parallel Programming Models

A parallel programming model refers to an abstraction of underlying hardware architecture that allows for the expression of both algorithms and data structures. It is different from programming languages with the invocation of an abstract execution model which cannot be understood by programming languages. In other words, a parallel programming model exists independently of the choice of the programming language. It bridges the gap between hardware and software.

There are mainly two classes of parallel programming models, which are individually called process interaction and problem decomposition. With regards to process interaction, it denotes that the parallel processes are able to communicate with each other. The most common forms of process interaction are shared-memory and message-passing. There is also implicit interaction in which the process interaction is invisible to the programmer, such as domain-specific languages and functional programming languages.

As for problem decomposition, it specifies the way in which the child processes of a program are formulated. Task parallelism is one of the problem decomposition which focuses on processes. Data parallelism describes a way of problem decomposition by performing operations in parallel on a data set. There is also implicit parallelism which reveals nothing to the programmer, such as automatic parallelization support in compilers.

### 2.4.1 Low-Level APIs

#### 2.4.1.1 Message Passing Interface

Message Passing Interface (MPI) is a message-passing library interface specification designed for functioning on a wide variety of parallel computing architectures, allowing convenient C, C++, Fortran bindings [47]. It addresses principally the message-passing parallel programming model, in which multiple processes communicate by calling library routines to send and receive messages to other processes.

MPI is a standard, not an implementation. There are many implementations of MPI, such as OpenMPI (an open source MPI-2 implementation) [51] or MPICH2 (a freely available, portable implementation of MPI developed originally by Argonne National Laboratory) [77]. In most MPI implementations, a fixed set of processes is created at initialization of program. Moreover, these processes may execute different programs, which is a big distinction from SPMD model where every process executes the same program.

MPI has been designed by a community of parallel computing vendors, computer scientists, and application developers. Nowadays MPI is still the most widely used model in HPC domain.

#### 2.4.1.2 CUDA

CUDA is a general-purposed parallel computing platform and programming model created by NVIDIA [89] that leverages Nvidia GPUs to solve complex computational problems. It comes with a software environment that allows developers to use C, C++, Fortran, Python or other languages. This accessibility makes it easy for developers to manipulate Nvidia GPUs in parallel programming.

The basic CUDA processing flow follows the pattern: 1. Copy data from host (CPU memory) to device (GPU memory); 2. Invoke kernel to do operations on device data; 3. Copy back data from device to host. Since CUDA programming model is primarily asynchronous, during the execution of kernel, the host CPU can also execute other tasks simultaneously.

CUDA programming model provides three key abstractions: a hierarchy of thread groups, a hierarchy of memory groups and barrier synchronization. These three abstractions works together to explore maximal computing power of accelerators.

When a kernel is launched from the host side, a large number of device threads are generated and each thread executes the statements specified by the kernel [33]. All threads invoked by this kernel composes a grid. A grid is consisted of many thread blocks. Each thread block contains a group of threads which can cooperate with each other by using block-local synchronization and block-local shared memory.

Figure 2.18 show the CUDA thread hierarchy. Theoretically, CUDA provides the organization of threads in three dimension. But in practice, a grid is usually organized as a 2D arrays of blocks and a block is composed of a 3D array of threads.

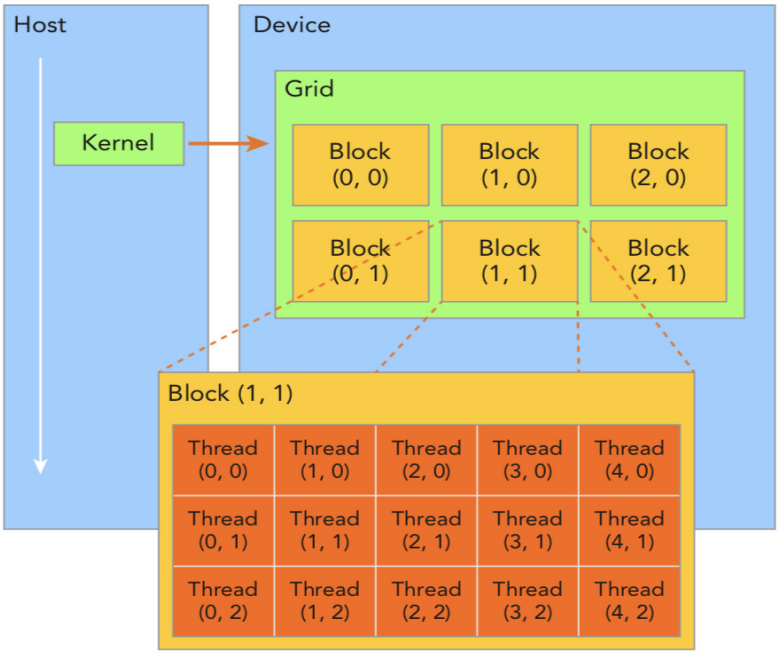



Figure 2.18: CUDA Thread Hierarchy [33].

Comparing to CPU memory hierarchy, CUDA memory hierarchy exposes a group of memory types which are programmable, such as registers, shared memory, local memory, constant memory, texture memory and global memory. The programmable types of memory makes CUDA programming model a very efficient tool for numerical calculation.

As shown in Figure 2.19, global memory, constant memory and texture memory resides in device memory with high latency and low bandwidth. Global memory is both readable and writable, while constant memory and texture are only readable. Constant memory must be declared with the attribute `__constant__`. Local memory is allocated for variables in a kernel that are eligible for registers but cannot fit into the register space. Local memory has high latency and low bandwidth because it actually resides in the same physical location as global memory. Shared memory is on-chip with much lower latency and higher bandwidth comparing to global memory. To get use of shared memory, variables in kernel function need to be decorated with `__shared__` attribute. Shared memory is visible among threads in a thread block. In other words, it is a tool for inter-thread communication. The `__syncthreads()` function must be used to access to shared memory without producing potential data hazard. Registers are specified with the highest bandwidth and the lowest latency since they are thread private. Variables declared in a kernel function without any other type qualifiers is generally stored in a register residing on each thread. However, register are scarce resources, there is a limitation of registers per thread depending on different GPU devices. Thus, if too much registers are used, there will be few thread blocks executing concurrently, leading to an extreme low occupancy and bad performance.

CUDA programming model provides two levels of barrier synchronization: system-level and block-level. System-

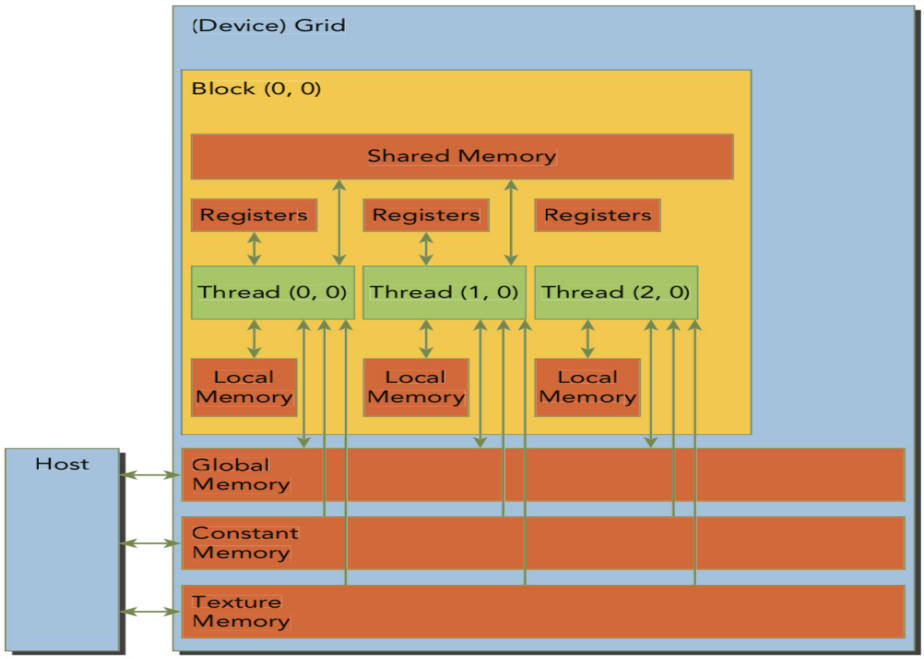



Figure 2.19: CUDA Memory Hierarchy [33].

level synchronization refers to the synchronization between host side and device side. `cudaDeviceSynchronize()` function can be used to block the host process until all CUDA operations have been completed. Block-level synchronization aims to barrier all threads in a thread block until they reach the same point in execution. It can be achieved by using the function `__syncthreads()`, as we've mentioned above.

Besides, CUDA provides asynchronous multi-streams to enable concurrency of CUDA operations such as kernel launches and memory copies. Operations within the same stream are ordered and unable to overlap. By comparison, operations marked with different streams can overlap to achieve concurrency [78].

#### 2.4.1.3 OpenCL

OpenCL is an open industry standard and framework for programming a heterogeneous collection of CPUs, GPUs, and other processors or hardware accelerators. Unlike CUDA only supporting Nvidia GPUs, OpenCL targets to a wide range of accelerators which are considered as OpenCL devices. OpenCL device is, in fact, an abstract term describing a physical architecture. Each OpenCL devices is made up of many compute units which can be further divided into a group of processing elements.

With regard to execution model, two terms "work-group" and "work-item" are used to describe OpenCL kernel where computations occur. As an example, if we choose a Nvidia Pascal card as OpenCL device, then its compute unit is equal to CUDA streaming processor and its processing element is mapped into CUDA thread. The OpenCL work-group refers to as CUDA thread block, and the OpenCL work-item equals to the CUDA threads in the thread block.

OpenCL memory model characterizes two fundamental memory regions: host memory and device memory. The device memory is divided into four named address spaces: global memory, constant memory, local memory and private memory. If we map them into CUDA terms, global memory and constant memory are same, local memory refers to as CUDA shared memory and private memory equals to CUDA registers.

Due to the abstracted memory and execution model, the key feature of OpenCL is portability. However, OpenCL is not performance portable across platforms, especially comparing to CUDA programs running on Nvidia GPUs. A study at Delft University from 2011 that compared CUDA programs and their straightforward translation into OpenCL C found CUDA to outperform OpenCL by at most 30% on the Nvidia implementation. The performance differences could mostly be attributed to differences in the programming model (especially the memory model) [45].

### 2.4.2 Directive-based Programming Models

Apart from the basic programming models introduced above, which are relatively at low-level, making them difficult to learn and to maintain, there are some directive-based programming models which are much more simple and easy to use, such as OpenMP, OpenACC, XMP [3][12], etc. In what follows we introduce several directive-based programming models that we use in our thesis.

#### 2.4.2.1 OpenMP

OpenMP is an application programming interface that supports portable parallel programming in C/C++ and Fortran across a wide range of architectures including multicore nodes and chips, NUMA systems, GPUs, FPGAs, etc. The most useful component of OpenMP API is called compiler directives. The structure of compiler directive is shown as follow:

```
Fortran:!$OMP PARALLEL LOOP PRIVATE(A)
C/C++:  #pragma omp parallel loop private(a)
```

Besides compiler directives, OpenMP also provides runtime library routines to set and query the number of threads in parallel region or even initialize the parallel loop. Furthermore, certain environment variables can be set manually to either specify loop schedule or to set number of threads, etc. Note that other programming models such as Intel Cilk+ [1] and TBB [71] also support data and task parallelism as OpenMP.

The execution model of OpenMP is called as the fork-join model. When an initial thread encounters a parallel region defined implicitly or explicitly by OpenMP directives, it generates a team of threads to accomplish the work inside this region concurrently. At the end of the parallel region, all members of the team join together and the program resumes being as a single thread of execution.

OpenMP 4.0 starts to provide a set of directives to perform offload computing. Such functionality is achieved by using the directive `target`. If a region is described by `target`, the data and code in this region may be offloaded to an available device. It should be noted that the host device may offload target regions to multiple target devices and OpenMP provides an interface to explicitly choose the target device. If there is no available target device, all target regions will be executed on the host device. In general, The OpenMP API defines two types `target` which are respectively dedicated to data mapping and code parallelization. The first one is composed of `target data`, `target enter/exit data`, `target update` and `declare target` which specify that variables or functions are mapped to or from a device. The second one consists `teams`, `distribute (simd)`, `distribute parallel for (simd)`. There are already some implementations of OpenMP API allowing for `target` directive. For instance, the IBM XL C/C++ V13.1.5 and XL Fortran V15.1.5 compilers are one of the first compilers that provide support for Nvidia GPU offloading using OpenMP 4.5 specification.

The OpenMP API provides a relaxed-consistency, shared-memory model where all OpenMP threads have access to a common place to store and retrieve data and each thread is allowed to have its own temporary view of the memory which allows the thread to cache variables and thereby to avoid going to memory for reference to a variable [10]. However, such relaxed-consistency model may cause data races and thus lead to incorrect result of the program. In order to avoid the memory consistency issue, OpenMP provides flush operations to enforce consistency between the temporary view and memory which can be specified using the `flush` directive.

#### 2.4.2.2 OpenACC

OpenACC is a user-driven performance-portable accelerator programming model supporting implicit offload computing by uses of compiler directives, library routines, and environment variables in C/C++ and Fortran [94]. In contrast to CUDA, it is much easier for OpenACC to port CPU-based original code to multiple architectures without any significant structural changes, which leads to lower portability and more complexity for code maintenance. Tetsuya Hoshino and his group studied the performance implications of OpenACC for two microbenchmarks and one real-world CFD application. The evaluations indicate that the current OpenACC compilers (PGI, HMPP, Cray) achieve approximately 50% of performance of the CUDA versions [59].

As a OpenMP-like directive-based programming model, OpenACC gives a basic directive format shown as follow:

```
Fortran:!$acc directive-name [clause-list]
C/C++:  #pragma acc directive-name [clause-list]
```

The compiler directives are used to accomplish data transfers between host and device, kernel execution. `data`

or `enter data/exit data` are adopted to achieve structured and unstructured data mapping. `parallel loop` or `kernel` directives invoke parallel region executed on devices. OpenACC also provides a set of runtime library routines as OpenMP. These routines can accomplish a large amount of works such as getting the number of devices or retrieving the device pointer by a host pointer.

The execution model of OpenACC is similar to that of OpenMP, which is host-directed execution with attached parallel accelerators, such as GPUs [13]. In the case of treating the multicore CPU as a device, the initial host thread may initiate a team of threads and perform parallel execution for code regions decorated by `parallel loop` and `kernel` directives.

OpenACC exposes coarse-grained, fine-grained and vector parallelism via `gang`, `worker`, and `vector` abstractions. In terms of CUDA terminology, a number of gangs is equal to a set of thread blocks. Each gang (thread block) has one or more workers, referring to as warps. Vector parallelism is for SIMD or vector operations within a worker (warp), which equals to CUDA threads in a warp.

OpenACC also supports concurrent activity queues for a targeting device where the host thread may enqueue operations of data transfer or kernel execution onto multiple device queues. The host thread may continue execution as the enqueued operations are handled asynchronously with the utilization of `async` directive clause or wait for all operations in a queue to complete with the use of `wait async` directive.

The memory model of OpenACC manages implicitly data movement between the host and device based on specific OpenPACC compiler directives. If a device shares memory with the host thread, new copies of the data will be created for the device and there is no data movement occurring. If a device has a discrete memory with the host thread, the space will be allocated in device memory and the data will be copied between the host memory and device memory. Moreover, OpenACC does not provide an interface for explicit management of shared memory or hardware caches. On the contrary, they are managed implicitly by the compiler with hints from the programmer in the form of directives [13].

### 2.4.3 Libraries

There are many programming libraries which have been developed and used for parallel computing such as Kokkos, SYCL, RAJA, etc. We intend to introduce Kokkos and SYCL which have be implemented in our thesis work in detail.

#### 2.4.3.1 Kokkos

Kokkos is a C++ based programming model for writing performance portable applications targeting complex node architectures with N-level memory hierarchies and multiple types of execution resources. It currently can use OpenMP, Pthreads and CUDA as backend programming models. The principal abstractions of Kokkos are parallel execution and data structure [43].

Concerning about parallel execution, Kokkos uses “execution space” term to describe the targeting architecture where code can actually be executed. It also provides three execution patterns which are called respectively `parallel_for`, `parallel_reduce` and `parallel_scan` to invoke parallelism. “Execution policy” is adopted to determine the manner how a parallel function is executed, such as setting the number of threads of kernel.

Kokkos provides a set of abstractions to manage data. Firstly, it uses “memory space” term to specify physical location of data and certain default memory access patterns. Secondly, a term named “memory layout” is used to express the mapping style of data array. Finally, Kokkos uses “memory trait” term to tell compiler how to access a data structure. There are different traits like atomic access, random access, etc.

The key advantage of Kokkos is that Kokkos gives abstractions for data layout management, making it a strong programming model to obtain high performance portability, while other programming models such as OpenMP and OpenACC are not able to address memory access pattern. However, Kokkos currently only supports 1 MPI process using at most one manycore device, which means that it is impossible to use OpenMP directives invoking multiple parallel threads targeting to multiple devices. In other words, Kokkos considers OpenMP and CUDA as same level execution space. Programmers may either choose OpenMP or CUDA to do parallel computing. But they cannot use OpenMP and CUDA at the same time.

#### 2.4.3.2 SYCL

SYCL is a C++ standard library (precisely, an open industry standard) that enables single-source heterogeneous programming under OpenCL backend (CUDA backend support is under development). It aims at providing the portability and efficiency of an application across a wide range of architectures, meanwhile, mitigating the complexity of heterogeneous programming by using higher-level C++ abstraction features such as templates, classes, and lambda functions. There are several available implementations of SYCL developed by different vendors including DPC++, ComputeCPP, triSYCL and hipSYCL as shown in Figure 2.20:

The SYCL execution model is comprised of the SYCL application execution model and the SYCL kernel execution model which targets respectively to define the execution of a SYCL program on host and device. The application execution model defines the basic characteristics of “accelerators” attached to a host such as `platform`, `context` and `device`. It also manages the launch and execution order of kernels by defining `command queue`, `command group` and `kernel function`. The kernel execution model defines the coarse-grained and fine-grained index space of a kernel function by using `work-group` and `work-item`.

With reference to SYCL memory model, SYCL use mainly two classes `buffer` and `accessor` to handle memory operations between host and device such the allocation and data transfer of memories. Based on different accessors, SYCL allows users to access different memories on device including `global memory`, `constant memory`, `local memory` and `private memory`.

Besides, SYCL offers several data parallel functions such as `parallel_for`, `parallel_work_in_group` and

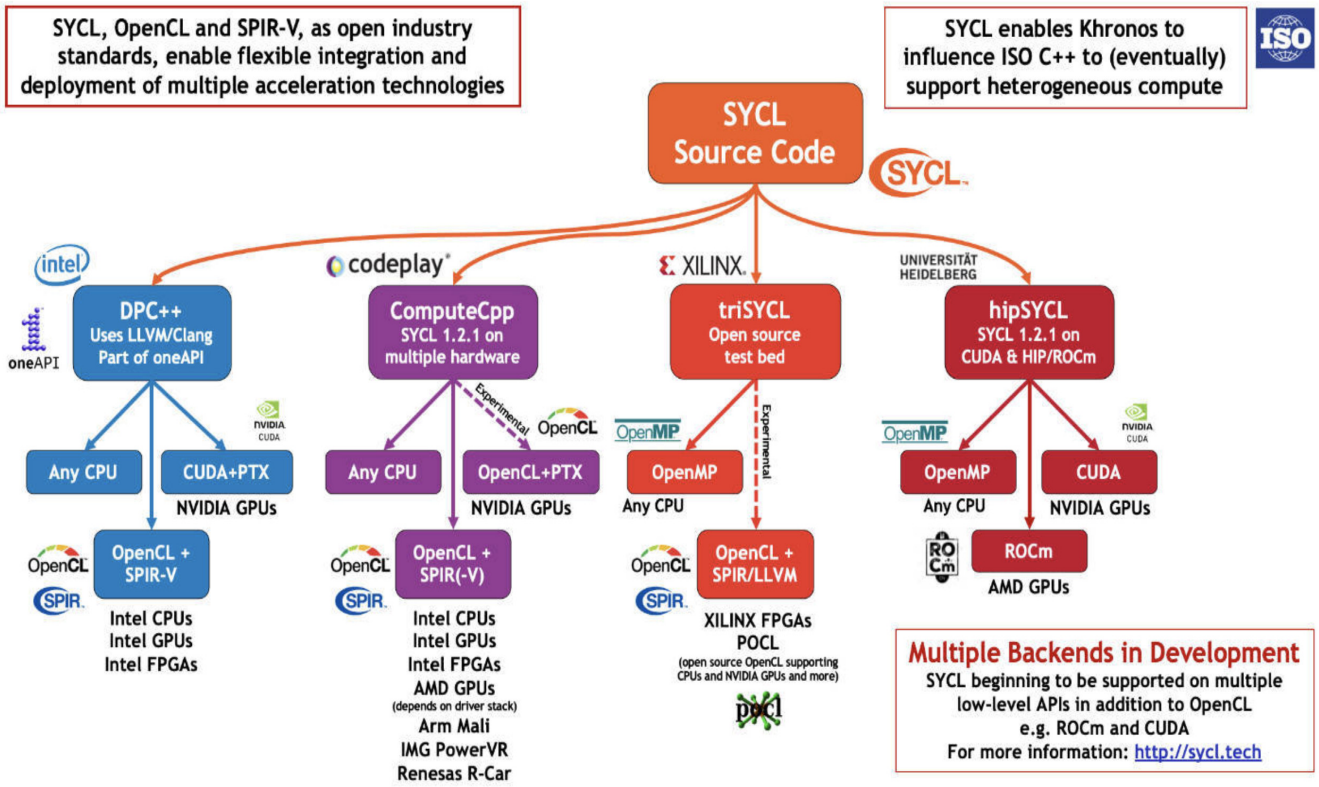



Figure 2.20: SYCL Implementations [8].

`parallel_work_in_item` to achieve hierarchical parallelism for a kernel as an outer loop for `work-group` and an inner loop for `work-item`. The synchronization between `work-items` within a `work-group` can be realized by a `work-group` barrier. On the contrary, there is no mechanism for synchronization between `work-groups` [69].

#### 2.4.3.3 Others

RAJA [58] is a collection of C++ software abstractions, being developed at Lawrence Livermore National Laboratory (LLNL), that enables architecture portability for HPC applications. StarPU is a task programming library for heterogeneous multi-core architectures which offers either the StarPU's high level GCC plugin pragmas, StarPU's rich C/C++ API, or OpenMP pragmas to handle task scheduling and data transfer on available CPUs and GPUs [11]. ArrayFire is a high performance software library for parallel computing with an easy-to-use API. Its array based function set makes parallel programming more accessible [135]. OCCA (oca-rina) is an open-source library which aims to provide a unified API for interacting with backend device APIs (e.g. OpenMP, CUDA, OpenCL) targeting to different architectures (CPUs, GPUs, FPGAs) [84]. DLB (Dynamic Load Balancing) library is dedicated to improve the load balance of the outer level of parallelism (e.g. MPI) by redistributing the computational resources at the inner level of parallelism (e.g. OpenMP) with LeWI (Lend When Idle) algorithm [2].

### 2.4.4 Comparison of Implemented Programming Models

Table 2.1 shows the comparison of several programming models implemented in our thesis work, including CUDA, Kokkos, OpenMP (thread + offload), OpenACC, and SYCL. From the perspective of thread hierarchy, all programming models provide a number of abstract levels of parallelism, enabling themselves to map to hardware with appropriate features except for CUDA, which only provides three levels of parallelism targeting specifically to the Nvidia GPUs.

Table 2.1: Comparison of implemented programming models.

| Programming Models | Thread Hierarchy | Memory Hierarchy | Targeting Architectures |
|---|---|---|---|
| CUDA | block, warp, thread | Programmable shared memory | Nvidia GPUs |
| Kokkos | team, thread, vector | | CPUs, Nvidia GPUs |
| SYCL | work-group, work-item | | CPUs, Intel GPUs |
| OpenMP | team, thread, simd | Lack of interface to on-chip memory | CPUs, Nvidia GPUs |
| OpenACC | gang, worker, vector | | CPUs, Nvidia GPUs |

From the perspective of memory hierarchy, CUDA, Kokkos, and SYCL provide an interface to explicitly manage shared memory while OpenMP, OpenACC both manage on-chip memory implicitly with the utilization of compiler directives or high-level functions, giving users no interface to programme shared memory explicitly. Note that CUDA uses `__shared__` to declare variables on shared memory. Kokkos makes use of the member function $team_shmem().get_shmem()$ to explicitly get an allocation on the scratch pad memory. SYCL provides access to allocated shared memory via local memory if a SYCL accessor has the access target `target::local`.

From the perspective of targeting architectures, CUDA is dedicated to targeting to Nvidia GPUs. Kokkos supports multicore/manycore processors (x86, OpenPower, ARM, etc) and Nvidia GPUs. SYCL is capable of targeting to any CPU (x86, OpenPower, ARM), Nvidia/AMD/Intel GPUs as well as Intel/Xilinx FPGAs based on different implementations. OpenMP and OpenACC may port codes to multicore CPUs and Nvidia GPUs.

## 2.5 Profiling Tools for Performance Analysis

Performance analysis is significant for parallel programming to mitigate bottlenecks. There are many profiling tools which are widely used for programmers to optimize their codes. In what follows several tools related to our thesis work will be introduced.

### 2.5.1 nvprof

nvprof is one of NVIDIA profiling tools that enables users to collect and view profiling data from the command-line. As a light-weight command-line profiler, nvprof is very handy to profile CUDA applications for quick checks such as collecting a summary of run time consumption for all the kernels and memory transfer operations.

Besides, nvprof also offers users a way to perform more detailed analysis collecting a list of available events and metrics during kernel execution where an event refers to a hardware counter describing a specific countable activity, action or occurrence on a CUDA device such as `active_cycles_pm` (number of cycles a multiprocessor has at least one active warp) and a metric is a property of a CUDA application characterized by one or more event values such as `achieved_occupancy` (ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor).

Furthermore, nvprof is capable of remote profiling by enabling the command-line option `-output-profile` so as to obtain a data file which may be later imported into nvprof and the NVIDIA Visual Profiler (nvvp) for analysis.

### 2.5.2 NVIDIA Visual Profiler

The NVIDIA Visual Profiler (nvvp) offers different views including the Timeline View, the Analysis View, the Source-Disassembly View and so on to allow users to analyze potential performance bottlenecks of their CUDA applications and thus find a way to mitigate those bottlenecks and eventually optimize the performance [91].

The Timeline View consists of a set of timeline rows where each row shows the start and end times of the activities corresponding to the type of this row which may represent a process for the profiled application, a CPU thread performing either a CUDA driver or CUDA runtime API call as well as OpenACC and OpenMP directives, an estimate of the compute utilization of a device over time, a context summarizing all memcpys of a type performed during the profiling (device-to-host, host-to-device, device-to-device, and peer-to-peer), a CUDA default or created stream undergoing a series of memcpys and kernel executions.

The Analysis View provides users a straightforward analysis which may be performed in guided or unguided mode exploring some detailed performance information of a CUDA application which indicates the kernel performance limiter, the kernel latency, the PC sampling as well as the memory statistics during the execution.

The Source-Disassembly View shows the hotspots and profiling data analyzed and aggregated for a kernel at the source and assembly instruction level where the hotspots indicate the lines of source code that may generate the performance bottlenecks and they are colored according to different level of importance (low, medium or high).

In addition to the results displayed in the timeline, analysis and source-disassembly views, users may also collect the specific event and metrics values of a memory copy or kernel execution in the GPU Details View as what they can do with the command-line profiler, nvprof. Moreover, there are some other views such as CPU Details/Source View, OpenACC Details View, OpenMP Details View which are provided in nvvp to allow users to perform a complete performance analysis.

# Chapter 3

# Monte Carlo Neutron Transport Simulation

In this chapter, we will elaborate some aspects of nuclear reactor physics which are considered in Monte Carlo neutron transport simulation along with the context of Monte Carlo transport codes for the HPC.

## 3.1 Nuclear Reactor Physics

A nuclear reactor is a system which generates heat from nuclear fission where a self-propagating nuclear chain reaction happens due to the collision of a neutron with an atomic nucleus. Each collision produces energy and releases neutrons which may be used to trigger more collisions and free more neutrons to form the chain reaction. Based on different characteristics as the types of fuel or coolant and so on, there are many types of nuclear reactor such as Pressurized Water Reactor (PWR), Boiling Water Reactor (BWR), Advanced Gas-cooled Reactor (AGR), Fast Breeder Reactor (FBR), etc. The main components of fuel are $UO_2$ and $PuO_2$ that consists uranium ($^{233}U$, $^{235}U$) and plutonium ($^{239}Pu$) nucleus. The coolants of reactors include $H_2O$, $D_2O$, $CO_2$, Na and so on.

Nuclear reactor physics is the physics of neutron fission chain reacting systems. It encompasses those applications of nuclear physics and radiation transport and interaction with matter that determine the behavior of nuclear reactors [115]. As a relatively mature discipline which has originated in the middle of the twentieth century and evolved for tens of years, nuclear reactor physics has now become the fundament directing the uses of nuclear energy for electricity production, neutron physics research, radiation therapy and so on. Nowadays, nuclear energy provides 10% of the world's electricity from about 440 nuclear reactors. In France, there is about 75% of its electricity derived from nuclear power plants.

The study of nuclear reactor physics is intended to understand and predict the behavior of nuclear reactor which is determined by the transport of neutrons and their interaction with nuclides composing the materials within a reactor. In other words, it can be summarized as the study of neutron transport, calculating macroscopic physical quantities such as the density of particles, the reaction rates, the heat power. Mathematically, this process can be

abstracted as to solve the neutron transport equation derived from the Boltzmann transport equation that is firstly proposed by Ludwig Boltzmann in 1872 [116].

## 3.2 Neutron Transport Simulation

### 3.2.1 Neutron Transport Equation

The neutron transport equation is a time-dependent linear equation which indicates a balance statement (neutron production equals to neutron losses) in the phase space element $drd\Omega dE$ for the angular neutron density $\psi(\vec{r}, \hat{\Omega}, E, t)$ at time $t$ and kinetic energy $E$, with the position vector $\vec{r}$ and the unit vector in travel direction $\hat{\Omega}$ (equals to $\frac{\vec{v}(E)}{|\vec{v}(E)|}$, where $\vec{v}(E)$ signifies the neutron velocity vector), as described in Equation 3.1. It should be noted that we are interested in the stationary form of the following equation without consideration of the first term $\partial n/\partial t$ in the thesis work.

$$\begin{aligned}
&\overbrace{\frac{1}{|\vec{v}(E)|}\frac{\partial}{\partial t}\psi(\vec{r}, \hat{\Omega}, E, t)}^{\partial n/\partial t} + \\
&\overbrace{\hat{\Omega}\cdot\nabla\psi(\vec{r}, \hat{\Omega}, E, t)}^{\text{streaming}} + \\
&\overbrace{\Sigma_t(\vec{r}, E)\psi(\vec{r}, \hat{\Omega}, E, t)}^{\text{collision}} \\
&= \underbrace{\int_0^\infty dE' \int_{4\pi} d\Omega' \Sigma_s\left(\vec{r}, E' \to E, \hat{\Omega}' \to \hat{\Omega}\right)\psi\left(\vec{r}, \hat{\Omega}', E', t\right)}_{\text{scattering}} \\
&+ \underbrace{\frac{\chi(E)}{4\pi}\int_0^\infty dE' \int_{4\pi} d\Omega' \nu\left(E'\right)\Sigma_f\left(\vec{r}, E'\right)\psi\left(\vec{r}, \hat{\Omega}', E', t\right)}_{\text{fission}} \\
&+ \underbrace{S_{ext}(\vec{r}, \hat{\Omega}, E, t)}_{\text{external source}}
\end{aligned} \tag{3.1}$$

On the left hand side:

- $\partial n/\partial t$: The partial derivative of the number of neutrons with respect to time $t$ inside the phase space element.
- *streaming*: The time variation of neutrons in the phase space element caused by travel without nuclear reactions.
- *collision*: The number of neutrons inside the phase space element that encounters collisions (*radiative capture* + *scattering*) at time $t$.

On the right hand side:

- *scattering*: The number of scattered neutrons at time $t$ moved inside the space area from other areas.

- *fission*: The prompt fission neutrons produced inside the space area due to fission at time $t$ where $\chi(E)$ refers to the energy spectrum of the fission neutrons and $\nu(E')$ means the average number of neutrons generated per fission.
- *external source*: The neutrons provided by external source that is independent of the neutron flux at time $t$ inside the phase space element.

With reference to $\Sigma_t$, $\Sigma_s$ and $\Sigma_f$ denoted in Equation 3.1, they are respectively the macroscopic total cross section, the macroscopic scattering cross section and the macroscopic fission cross section which characterize the probability of a given reaction that a neutron may experience per unit path length traveled in the material. The details of cross section will be elaborated in section 3.2.2.

Note that the *scattering*, *radiative capture* and *fission* mentioned above are different types of nuclear reactions that happens during the bombardment-driven process. Scattering describes the interaction without changing the nature of the target nuclide. It can be further divided into two types: elastic scattering and inelastic scattering. In an elastic scattering process, the kinetic energy and momentum of the "incident neutron-target nucleus" system is conserved and there is no energy transferred into nuclear excitation. By contrast, in an inelastic scattering reaction, the momentum of the system is preserved while the kinetic energy of the system is not conserved due to the fact that some energy of the incident neutron is transferred into nuclear excitation. Such energy transfer leaves the target nucleus in a short-lived energy state and ends up with the emission of one or more gamma rays turning the nucleus from an excited state to the ground state. In a radiative capture process, the target nucleus forms a compound nucleus in an excited state of energy by absorbing the incident neutron and is de-excited by emitting gamma rays. As for fission, the target nucleus captures the incident neutron and splits into lighter nuclei along with the production of several neutrons and the emission of gamma rays, as well as the release of a large amount of energy.

Solutions to the neutron transport equation can be divided into fixed source problems and eigenvalue problems according to whether an external neutron source exists [133]. Eigenvalue problems can be further classified into ($k$, $\lambda$, $\gamma$, $\alpha$)-eigenvalue problems [122][14][140][29] based on different ways to reach criticality of a steady-state multiplying system in the absence of external source as described in Equation 3.2.

$$\boldsymbol{M} \cdot \psi(\vec{r}, \hat{\Omega}, E) = \boldsymbol{F} \cdot \psi(\vec{r}, \hat{\Omega}, E) \tag{3.2}$$

where:

- $\boldsymbol{M}$: The transport operator ($\hat{\Omega} \cdot \nabla + \Sigma_t(\vec{r}, E) - \int_0^\infty dE' \int_{4\pi} d\Omega' \Sigma_s \left(\vec{r}, E' \to E, \hat{\Omega}' \to \hat{\Omega}\right)$).
- $\boldsymbol{F}$: The fission operator ($\frac{\chi(E)}{4\pi} \int_0^\infty dE' \int_{4\pi} d\Omega' \nu\left(E'\right) \Sigma_f\left(\vec{r}, E'\right)$).

If we perform a $k$-eigenvalue calculation for Equation 3.2, the problem is reduced to determine the effective multiplication factor $k_{eff}$ and its corresponding eigenfunction $\psi_{eff}(\vec{r}, \hat{\Omega}, E)$.

$$\boldsymbol{M} \cdot \psi(\vec{r}, \hat{\Omega}, E) = \frac{\boldsymbol{F}}{k_{eff}} \cdot \psi(\vec{r}, \hat{\Omega}, E) \tag{3.3}$$

The effective multiplication factor indicates the ratio of the neutron production to the neutron loss which is used to characterize the subcritical, critical and supercritical states of a multiplying system listed as follow:

- $k_{eff} < 1$: Subcritical state, the neutron population of system decreases in time.
- $k_{eff} = 1$: Critical state, the chain reaction is self-sustaining as the neutron population remains unchanged in time.
- $k_{eff} > 1$: Supercritical state, the neutron population increases in an exponential way with time.

### 3.2.2 Nuclear Cross Section

Nuclear cross sections are the key ingredients of the neutron transport equation as they represent the probability of the neutron to interact with the crossed material.

#### 3.2.2.1 Macroscopic Cross Section

The macroscopic cross section characterizes the probability of the neutron to interact with the material per unit path length traveled in the material. It is the sum of the microscopic cross section of all the nuclides present in the material weighted by their atomic density, as described in Equation 3.4.

$$\Sigma_r(E, T) = \sum_i N_i \sigma_{r,i}(E, T) \tag{3.4}$$

where:

- $r$: The nuclear reaction type.
- $i$: The index of a nuclide that resides in the material.
- $E$: The energy of the incident neutron.
- $T$: The temperature of the material.
- $N_i$: The atomic density of the nuclide $i$.
- $\sigma_{r,i}(E, T)$: The $r$-type microscopic cross section of the nuclide $i$ at a given energy $E$ and temperature $T$.

#### 3.2.2.2 Microscopic Cross Section

The microscopic cross section of a nucleus is denoted in a simple form as $\sigma_r$ with the units of measurement barn ($10^{-28}$m$^2$) or $10^{-24}$cm$^2$ where $r$ refers to one sort of nuclear reactions such as elastic scattering ($\sigma_s$), inelastic scattering ($\sigma_i$), radiative capture ($\sigma_\gamma$) and fission ($\sigma_f$). The total microscopic total cross section $\sigma_t$ is used to characterize the probability that any sort of nuclear reaction will occur for a nucleus as described in Equation 3.5.

$$\sigma_t = \sigma_s + \sigma_i + \underbrace{\sigma_\gamma + \sigma_f}_{\text{absorption}} + \dots \tag{3.5}$$

Figure 3.1 depicts the cross sections of several reaction types for $^{238}$U varying with the incident energy of the neutron. As we can see, the total cross section is typically dominated by the elastic scattering cross section plus absorption cross section. Furthermore, the energy-dependent cross section can be generally divided into three energy ranges including low-energy region, resonance region and high-energy region [57]. The full range of cross section for a nuclide may be comprised of one or more regions. For example, $^{238}$U contains all three regions while $^{1}$H is only represented in the way of low-energy region.

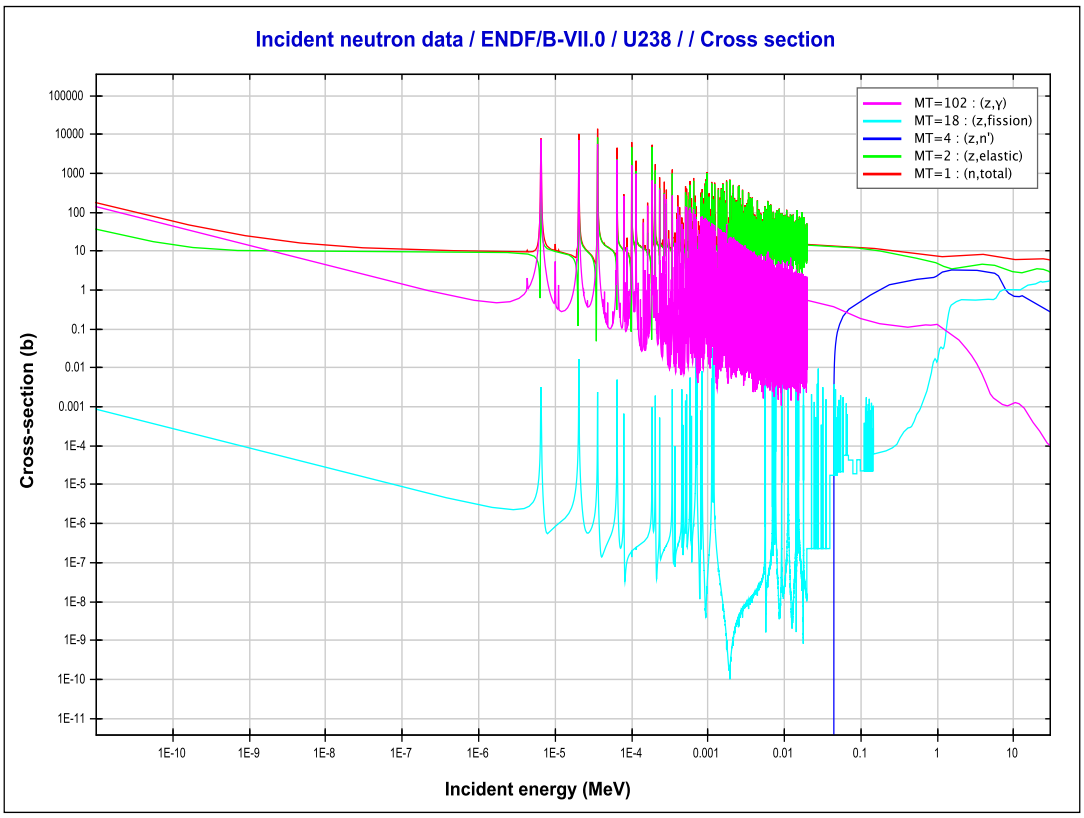



Figure 3.1: Comparison of different types of cross sections for $^{238}$U via JANIS 4.0 [114] with the database set to ENDF/B-VII.0 library [30].

In the low-energy region (typically $E < 1$eV), the incident energy is well below the resonance and the $\frac{1}{v}$ behavior ($v$ the velocity of incident neutron) is observed instead for both the scattering cross section and the absorption

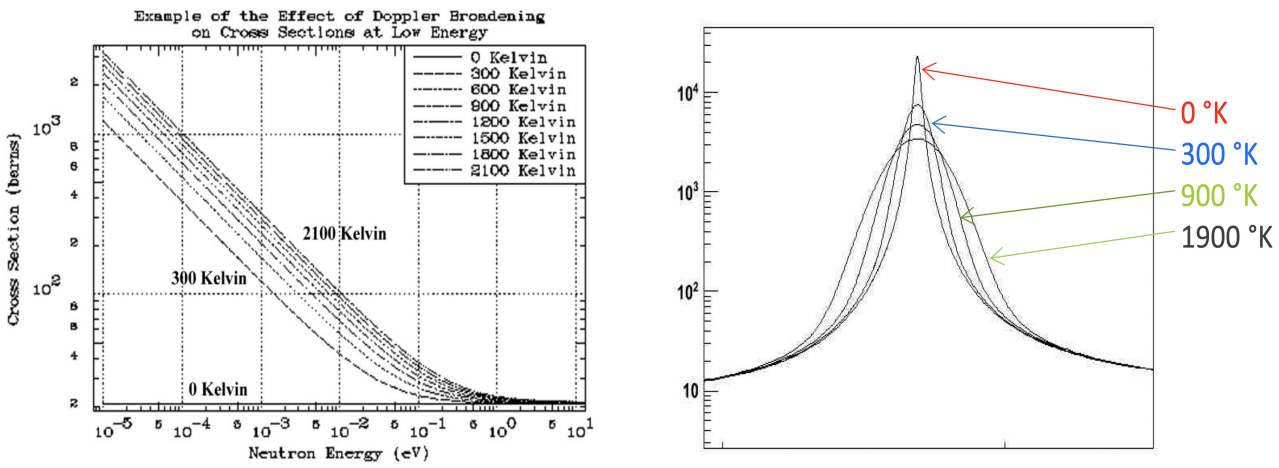
cross section when the effects of temperature and Doppler broadening is taken into account (mainly discussed in section 3.2.2.3). According to this behavior, the cross sections are tabulated as smooth functions of $\frac{1}{v}$ or $\frac{1}{\sqrt{E}}$.

The resonance region is one characterized by large fluctuations of several orders of magnitude. It can be divided into two regions which are respectively the resolved and unresolved resonance region. In the resolved resonance region, resonances can be seen and distinguished using methodologies such as the R-matrix formalism [30]. In the unresolved resonance region, there are so many resonances that they cannot be distinguished with each other and a statistical representation is adopted.

In the high-energy region, the cross sections become smooth again such as in the low-energy region. Moreover, we can find from the Figure 3.1 that at high energy the reactions involving neutron emission such as the inelastic scattering and fission become more significant comparing to ones without neutron emission such as the elastic scattering and radiative capture.

#### 3.2.2.3 Effects of Temperature and Doppler Broadening

The effect of temperature on cross sections is caused by the thermal motion of atoms which adds the dependence of cross sections on the relative energy between the incident neutron and the target nucleus. Such effect leads to different phenomena across energy regions.




(a) (b)

Figure 3.2: Effect of temperature on cross sections [36].

In the low-energy region, when the temperature augments from $0$ Kelvin, the energy-dependent scattering cross section function changes from a quasi-constant function to a decreasing function in $\frac{1}{v}$ shape, as depicted in Figure 3.2a.

In the resonance region, resonances are flattened with the increase of temperature. This process is called the Doppler broadening of resonances. The peak-to-peak amplitudes of resonances become smaller and the widths of resonances become broader. Resonances may eventually be smoothed out if the temperature is enhanced to a

high level. For example, due to the Doppler effect, we can easily see that the increase of temperature (from $0$ Kelvin to $300$ Kelvin) significantly weakens the resonance, as shown in Figure 3.2b.

In the high-energy region, the effect of temperature is negligible since the thermal motion of target nucleus is much less dominant than the motion of incident neutron. The target nucleus is considered at rest as it behaves at $0$ Kelvin.

Mathematically, the Doppler broadening equation, as described in Equation 3.6, has been proposed by Cullen and Weisbin to calculate Doppler broadened cross sections along with the SIGMA1 algorithm which performs the convolution in this equation by piecewise-linear exact integration [38].

$$\sigma\left(y,T\right)=\frac{1}{y^2}\left(\frac{1}{\pi}\right)^{1/2}\int_0^\infty x^2\sigma\left(x,T_0\right)\left[e^{-(x-y)^2}-e^{-(x+y)^2}\right]dx \tag{3.6}$$

where $y^2=\alpha E$, $x^2=\alpha E_r$, $\alpha=\dfrac{M}{k_B(T-T_0)}$:

- $E$: The incident neutron energy.
- $E_r$: The relative energy between incident neutron and target nucleus.
- $M$: The target mass.
- $k_B$: The Maxwell-Boltzmann constant equaling to $1.38064852\times10^{-23}$ m$^2\cdot$kg$\cdot$s$^{-2}\cdot$K$^{-1}$.
- $T_0$: The base temperature of cross sections to be broadened.

In addition to the SIGMA1 method, there are many methods that introduces approximations to the Doppler broadening equation. For instance, the Gauss-Legendre quadrature and Gauss-Hermite quadrature approaches approximate the integral in Equation 3.6 to a form of Gaussian quadrature which contributes to higher computational efficiency comparing to the SIGMA1 method due to the avoidance of complementary error function evaluations [107][76][41][65].

The Doppler broadening algorithms mentioned above are widely used in cross section processing codes such as NJOY [81], AMPX [42], PREPRO [37] to prepare cross sections at the desired temperatures from 0 Kelvin cross sections. These codes also build 0 Kelvin cross sections from more elementary parameters such as resonances parameters.

Note that the simulation performed in our thesis work is based on thermohydraulic coupling which takes the effect of Doppler broadening into account for temperature feedback. As the temperature varies during the execution of simulation, a large number of temperature-dependent cross section data is required and the on-the-fly Doppler broadening method becomes essential as the conventional pretabulated method needs too much memory to store broadened cross section data.

### 3.2.3 Isotopic Depletion Calculations and Thermohydraulic Feedback

One of the main challenges for Monte-Carlo neutron transport code in reactor physics is the realization, at the fuel pin scale, of an irradiation cycle at a fuel nuclear reactor core. To perform this kind of calculation in realistic conditions, it is necessary to couple a neutron transport code with:

- An isotopic depletion solver to take into account the creation and disappearance of nuclides in the material under irradiation. This is governed by the Boltzmann-Bateman non-linear coupled system [17]. This kind of calculation induces a large number of nuclides in the fuel material.
- The coupling with a thermohydraulic solver to take into account the temperature and/or density change in the fuel and the coolant. This kind of calculation needs a large number of temperature to be considered. To avoid a prohibitive memory foot print (about 1GB per temperature for the cross section), on-the-fly Doppler broadening techniques become mandatory.

## 3.3 Computational Methods

Two major approaches are conventionally used to solve Equation 3.1, including the deterministic method and the Monte Carlo method. The deterministic methods consist in applying numerical discretizations to the variables of the neutron transport equation. The Monte Carlo method is to transform the integral form of the neutron transport equation into the average of $N$ values of particle histories collected through a large number of iterations [80] which is given as Equation 3.7:

$$X \approx \frac{1}{N} \sum_{n=1}^{N} \left( \sum_{k=1}^{\infty} x_{nk} \right) \tag{3.7}$$

where $X$ is a macroscopic quantity of interest such as a reaction rate, $x_{nk}$ is the observation of this quantity at the very position of $k^{th}$ iteration for particle $n$. According to the central limit theorem and the law of large numbers, $X$ converges to the exact value of the integral when $N \to \infty$.

### 3.3.1 Deterministic Method

The deterministic method introduces discretizations and approximations to simplify and obtain numerical solutions of the neutron transport equation. The discretizations are typically made with respect to the variables such as energy, angular, and spatial [124][134][121].

- Energy discretization: The multigroup cross section library is used in substitution of the continuous-energy cross section data where the energy range is divided into multiple groups and the cross sections in each group become a constant.

- Angular discretization: The integral of angular is transformed to a sum of weighted neutron flux function following discretized direction vector.
- Spatial discretization: The geometry is discretized into a mesh of points and the integral of space becomes a sum of neutron flux at each point.

The introduction of discretizations makes the deterministic method a compute-efficient solution to neutron transport simulation at the expense of precision. Moreover, due to the limits of discretizations under complex conditions, the deterministic method may be unable to handle problems with specific features.

### 3.3.2 Monte Carlo Method

The Monte Carlo method is a stochastic approach to obtain the statistical properties of real-world problems [110]. The key feature of Monte Carlo method is the use of random sampling and statistical analysis. For example, to solve a mathematical equation, unlike conventional deterministic method which introduces numerical discretizations to achieve approximations, Monte Carlo method aggregates the results of a large amount of samples generated from a probability distribution which is constructed by (pseudo-)random numbers to simulate the reality corresponding to the equation.

Due to the fact that Monte Carlo method solves the equations abstracted from real-world problems with minimal mathematical approximations, Monte Carlo method is widely used to simulate complex systems with many coupled degrees of freedom in many domains such as statistical physics [19], computational biology [93], artificial intelligence [26] and so on. When it is applied to solve the neutron transport problem as described in Equation 3.7, a neutron transport problem will be turned into the tracking of a large number of particle histories from their birth to their death in conjunction with the aggregated estimates obtained by tallying.

A history (random walk process) of neutron $n$ consists of a sequence of iterations $(I_{n0}, I_{n1}, I_{n2}, \ldots, I_{nK_n})$ in which the neutron moves a distance from its initial position to next position and then interact with the material it is flying through (Figure 3.3). $K_n$ is the total number of iterations contained in the history of neutron $n$ and after the final iteration the neutron is desactived by leakage, absorption or energy cut. Each iteration $I_{nk,k\in K_n}$ can be denoted by a triple $(\vec{P}_{nk}, \vec{D}_{nk}, C_{nk})$ where $\vec{P}_{nk}$ is the position of neutron $n$ at the beginning of $k^{th}$ iteration, $\vec{D}_{nk}$ is the displacement of neutron $n$ at $k^{th}$ iteration. $C_{nk}$ is one type of interactions for neutron $n$ at $k^{th}$ iteration including scattering, absorption as well as none of collisions if the neutron moves across boundaries.

In the context of nuclear reactor physics, especially taking isotopic depletion into account, the most computational work during each iteration is the calculations of cross sections which are required for random samples of the distance of movement, the nuclide to be collided, the sort of interaction and the update of kinetic state of the neutron.

As for the tallying part, the observation $x_{nk}$ given in Equation 3.7 of a random variable $X$ varies with the different choices of estimators. For example, in order to tally the reaction rate for reaction $r$ within a volume without

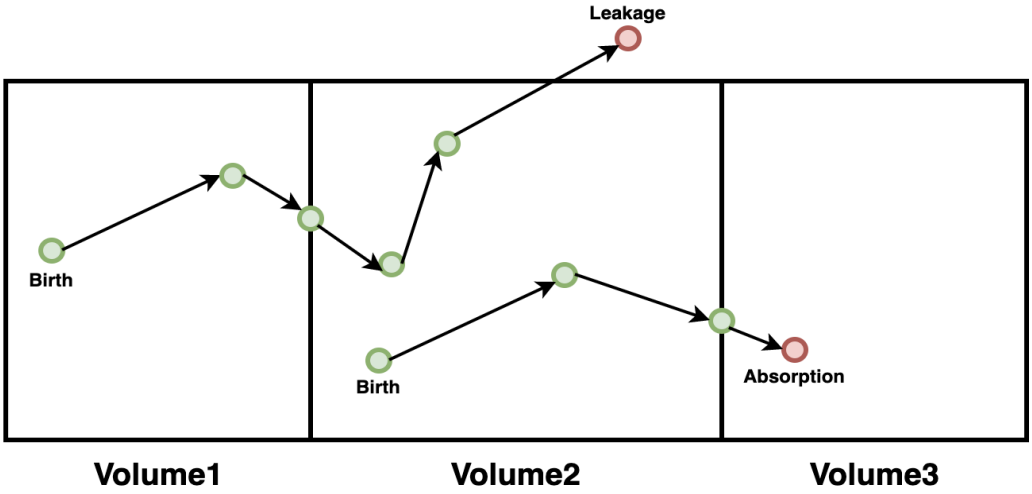



Figure 3.3: Neutron histories tracked from their birth to death through a set of volumes where green cycles refer to active neutrons and red cycles refer to desactived neutrons.

considering weights, we can use several estimators listed as follows:

- Analog estimator: observations of reaction $r$ are accumulated. $x_{nk}$ equals to $1$ if reaction $r$ happens, $0$ otherwise.
- Collision estimator: observations resulting in any type of collisions are scored with the ratio of macroscopic cross section for reaction $r$ to total macroscopic cross section. $x_{nk}$ becomes $\frac{\Sigma_r}{\Sigma_t}$ if there is a collision, $0$ otherwise.
- Track-length estimator: observations resulting in any type of collisions are scored with the multiplication of corresponding distance of movement and macroscopic cross section for reaction $r$. $x_{nk}$ equals to $|\vec{D}_{nk}|\Sigma_r$ for a colliding iteration, $0$ otherwise.

Furthermore, the error estimation of a tally is basically achieved by the use of variance by which we know the statistical error of the corresponding estimate. The comparison among estimates can be handled by the adoption of variance of the mean of estimates from which we can analyze the order of number of particles that makes the errors of several estimators tending to an arbitrary low level.

Overall, the very few approximations introduced make the Monte Carlo method a precise approach which is capable of handling a large number of neutron transport problems under complex conditions and performing reference calculations. However, it incurs a much higher computational cost comparing to the deterministic method widely used in the industry. For this reason, researchers and developers of many Monte Carlo transport solvers have turned to solutions by porting Monte Carlo codes to modern architectures so as to reduce time to solution and/or model more complex systems.

## 3.4 HPC and Monte Carlo Neutron Transport Simulation

The development of Monte Carlo neutron transport simulation in HPC mainly focus on the parallelization of Monte Carlo codes so as to benefit from high throughput of parallel computing machines and eventually yield improving performance. In what follows an explicit introduction of the state-of-the-art in the development of Monte Carlo neutron transport in HPC will be elaborated.

### 3.4.1 State-of-the-art

The general workflow of a serial Monte Carlo neutron transport simulation is illustrated in Figure 3.4. It is mainly comprised of three steps including initialization, particle tracking and finalization. The initialization step is responsible for setting up materials and generating particles from source. The particle tracking step is devoted to tracking particle histories one by one in which the cross section calculations account for the most consuming part of Monte Carlo neutron transport simulation. The finalization step is in charge of tally computation in order to retrieve estimates of physical quantities.

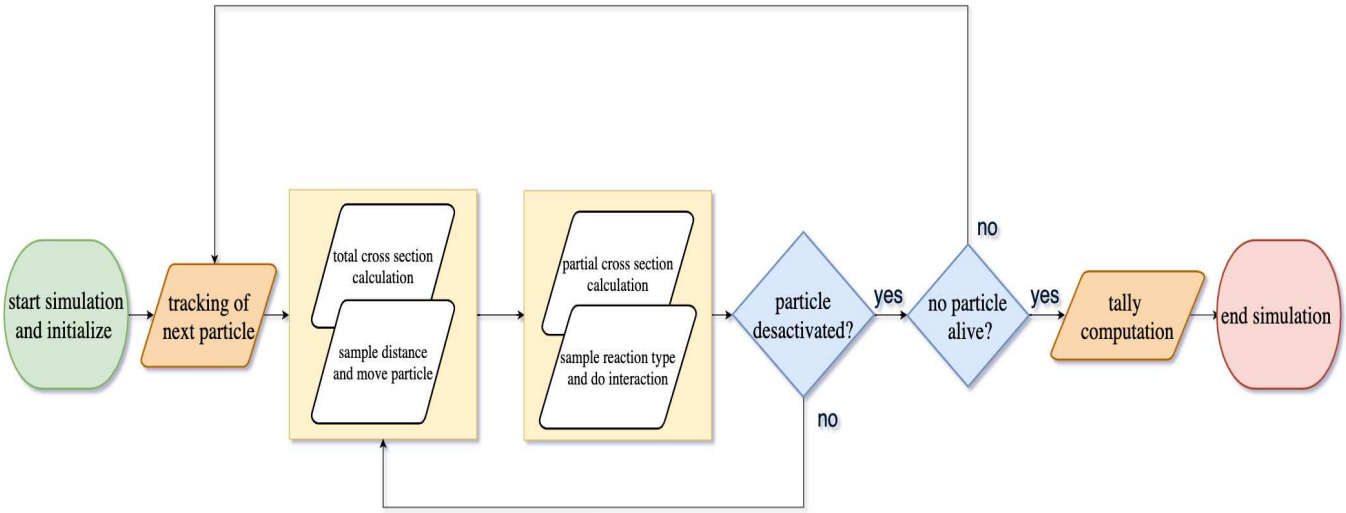



Figure 3.4: Serial Monte Carlo neutron transport simulation workflow.

Due to the embarrassingly parallel nature of Monte Carlo transport problems where particles are simulated independently during their lifetime, Monte Carlo simulation is an ideal candidate for data parallelism at coarse grain level. Such parallelization is achieved by the use of distributed/shared memory parallel programming model like MPI and OpenMP for all three sections mentioned above. It decomposes the serial codes into a set of identical processes in parallel fed with different data including dispatched particles, replicated tallies and geometries (or splitted geometries).

The distributed memory parallelism reached in Monte Carlo transport simulation needs to arrive at a trade-off between memory footprint on each rank and communication workloads across ranks. For example, if the geometries

of Monte Carlo simulation are divided into separate domains across ranks, communications among ranks will be significantly augmented because each rank needs to address the domain-crossing issue by sending and receiving the data of out-domain and in-domain particles. By contrast, if distributed memory parallelism is only responsible for dispatching particles, there will be much less inter-communications among ranks at the expense of larger memory footprints since each rank is required to hold entire data of geometries [20].

Similarly, there is always a compromise between memory footprint and achievable bandwidth for shared memory process of Monte Carlo transport simulation. For instance, the data of nuclides is usually stored as one copy and used for all multithreading processes on a shared memory system. Such strategy highly reduces memory footprints but degrades the efficiency of memory access due to the avoidance of data races and extra overheads of non-local memory access via links and so on. The corresponding performance penalty varies a lot with hardware properties such as cache hierarchy, memory access model (NUMA or UMA). It's up to developers to find a way the most adaptive to the coarse-grained parallelism of Monte Carlo transport simulation on a specific working machine taking memory footprint and bandwidth into consideration.

In addition to coarse-grained data parallelism, the parallelization at fine-grained level has also been developed for Monte Carlo transport simulation and it has become a major aspect to dig into as modern architectures have added more features of vectorization such as SIMD and SIMT techniques. Typically, after the parallelization of coarse granularity, each processing thread is assigned to track histories of a batch of particles. To further parallelize the Monte Carlo transport codes, one may either vectorize the inner-loops in particle tracking kernel injected by cross section calculations or divide particles within a batch into vectors based on different events and execute a vector of particles in parallel once a time. The first strategy in which particles are tracked in sequence is actually called as history-based method. The second strategy where particles of a sorted event type are simulated in parallel is instead event-based method.

### 3.4.2 Parallelism

The parallelism of history-based Monte Carlo neutron transport codes used in our thesis is showed in Algorithm 1. At the distributed memory level, each MPI rank is in charge of performing an independent replication of the simulation (the first loop at line 1). The inter-communications among MPI ranks occurs only at the end of each simulation to compute the mean and variance of all the tallies. However, each MPI rank duplicates the global data such as the cross section data, the isotopic description of the materials, the description of the geometry and above all the tallies data structure, which lead to large memory footprints. For depletion calculation, tally data structure and materials description may represent several tens of gigabytes. Hence, for these kind of calculation one MPI rank is typically attached to one node.

The second loop (line 3) is executed in sequence to handle all particles in the current process batch by batch. The third loop (line 5) is achieved by the shared memory level of parallelism as this loop contains the largest

trip count (between $10^5$ and $10^6$). All threads are responsible for tracking the histories of a number of particles assigned to themselves in parallel. It is obvious that with the increase of the number of threads, the scalability will always degrade due to the common obstacles occurred at shared memory level such as memory contention, NUMA systems. Besides, since in our case we use the on-the-fly Doppler broadening method which contributes to a large amount of computational cost, the workloads of each thread may be significantly unbalanced. In order to mitigate this load balancing issue, we adopt the strategy of dynamic scheduling to assign the untracked particles to idle threads.

The `while` loop (line 6) simulates a particle from its birth to its death. This type of tracking approach is called history-based because we won't go to the next particle until the disappearance of the current particle.

The fourth loop (line 8) is dedicated to calculating macroscopic cross sections sequentially in which the calculations of microscopic cross section for each nuclide can be vectorized or parallelized by SIMD and SIMT techniques employed in our Monte Carlo neutron transport codes. This level of parallelism is the key point to be studied in this thesis work.

**Algorithm 1:** Parallelism of History-based Monte Carlo neutron transport codes.

```
1  foreach process do                                          /* MPI level */
2      distribute all particles in the current process into a number of batches;
3      foreach batch do
4          ...;
5          foreach particle in the current batch do            /* Shared memory parallelism */
6              while particle is alive do
7                  ...;
8                  foreach nuclide in the material do
9                      calculate microscopic cross section;    /* SIMD, SIMT */
10                     sum up macroscopic cross section;
11                 end
12                 ...;
13             end
14         end
15         ...;
16     end
17 end
```

### 3.4.3 History-based and Event-based Methods

Most of the MC transport codes are based on the history-based method, but in the 80's to make the best use of vector processors the event-based algorithm was developed which is brought up to exploit GPUs and vectorization.

Algorithm 2 shows the general procedure of history-based Monte Carlo transport simulation at the processing thread level. The outermost `for` loop (line 1) is executed in serial containing a `while` loop (line 2) tracking a particle from its birth to death. In each iteration, cross sections are firstly calculated (line 3-6) to sample distance as well as collided nuclide and interaction type (line 7). The displacement and collision take place afterward as illustrated from line 8 to line 12. The vectorizable parts reside in macroscopic cross section calculations including an outer loop for total cross section sum-up, an inner loop of energy lookup and an inner loop of integral in addition if on-the-fly Doppler broadening method is adopted. The details of these cross section algorithms are expressed in section 3.4.4.

**Algorithm 2:** History-based Monte Carlo transport algorithm.

```
1  foreach particle distributed to the processing thread do
2      while particle is alive do
3          calculation of macroscopic cross section, with several vectorizable parts;
4              • outer loop for calculating macroscopic cross section;
5              • inner loop of energy lookup to find E_low and E_up;
6              • inner loop of integral injected by on-the-fly Doppler broadening method;
7          sample distance to collision in material;
8          if new position is still inside material then
9              move particle to new position, and do collision;
10         else
11             move particle across boundary;
12         end
13     end
14 end
```

As an alternative solution, the event-based method was proposed by regrouping particles according to their different event types and undertaking the simulation of banked particles in parallel [25]. The general procedure of event-based method is described in Algorithm 3. The outermost `while` loop (line 1) ensures that all particles dispatched to a given processing thread will be tracked until they are desactived. The first inner `for` loop (line 2) is devoted to fill identified particles to vectors of different event types. The second inner `for` loop (line 5) is responsible for processing particles of an event type vector by vector.

**Algorithm 3:** Event-based Monte Carlo transport algorithm.

```
1 while any particles are alive on the processing thread do
2     foreach active particle do
3         Identify next event type and move it to the vector of corresponding event type;
4     end
5     foreach vector of event type do
6         perform event for all particles in vector;
7     end
8 end
```

The classification of event types may be of fine or coarse granularity which is totally up to developers to define. For example, Bleile developed an event-based Monte Carlo transport algorithm where only three event types are used: collision processing, material interface crossing, and cell boundary crossing [22][21]. Bergmann and Vujić proposed an event-based Monte Carlo transport algorithm adapting to GPUs in which more than a dozen of event types are defined [18]. Fine-grained event types introduce less branch divergence for each event kernel while it incurs extra workload due to shuffle operations so as to regroup particles into different vectors. On the contrary, coarse-grained event types reduce particle regrouping need but make branch divergence a major point to degrade kernel performance. If we define three events including migration, scattering and absorption, the event-based Monte Carlo transport algorithm can be unfolded as shown in Algorithm 4.

Initially, all particles on the processing thread are stored in the vector of migration. In each iteration of `while` loop (line 2), three `for` loops are performed successively to process particles of different event type. The loop on the migration vector (line 3) takes responsibility of calculating cross sections, sampling distance and moving particles to collision sites. If a particle is going to undertake a collision, it will be moved to the vector of scattering or absorption and be processed in one of the following two loops. The loop of scattering (line 13) carries out scattering interactions for all particles in the vector and then shuffles them back to the vector of migration to prepare for next iteration. The loop of absorption (line 17) processes all particles of absorption, kills these particles afterward and adds newly born particles (production of fission) to the vector of migration. All three inner `for` loops are candidates for vectorization using SIMD or SIMT techniques.

Overall, compared with history-based method, event-based approach is better suited for vectorization on modern computers to benefit from data-level parallelism. At the same time, it introduces extra workload of consolidating surviving particles during the simulation and requires restructuring the control flow and redesigning data structure, which makes it difficult to implement with full-physics capabilities.

Recently, a significant number of studies have explored using Intel MIC and Nvidia GPU for Monte Carlo transport solvers and micro-benchmarks [18][95][55][54]. Both history-based method and event-based method are im-

plemented and the results are quite informative and promising. They have proved that history-based method is far more straightforward to implement than event-based method and event-based method may outperform history-based method with specific tuning strategies such as remapping data references, use of intrinsic functions and design of trivial kernel. Romano has developed a simple model to theoretically estimate the efficiency of event-based method and observed that the good performance is yielded when the ratio of the particle bank size (e.g. migration vector size) to the vector length (e.g. AVX-512 SIMD-vector is able to store 8 `double`) is over 20 [106].

**Algorithm 4:** (Migration, scattering, absorption)-based Monte Carlo transport algorithm.

```
1  initially, store all particles in the vector of migration;
2  while any particles are alive on the processing thread do
3      foreach particle in the vector of migration do
4          calculation of macroscopic cross section;
5          sample distance to collision in material;
6          if new position is still inside material then
7              move particle to new position;
8              shuffle particle to the vector of corresponding collision;
9          else
10             move particle across boundary;
11         end
12     end
13     foreach particle in the vector of scattering do
14         do scattering;
15         shuffle particle to the vector of migration;
16     end
17     foreach particle in the vector of absorption do
18         do absorption;
19         add produced particles (due to fission) to the vector of migration;
20     end
21 end
```

### 3.4.4 Cross Section Computing Algorithms

For a given material comprised of a set of nuclides, each nuclide has its own energy grid stored in a continuous array mapping to corresponding cross section data stored in another array of the same length. The macroscopic cross sections are summed up as described in Equation 3.4. As for microscopic cross section calculations, there

are mainly two approaches, which are called respectively pre-tabulated method and on-the-fly Doppler broadening method.

With reference to pre-tabulated method, an energy lookup is employed to find the nearest energy lower-upper bound in cross section tables of given temperatures and the required cross section is retrieved by performing an interpolation based on this bound. The cross sections are prepared for all the temperatures occurring in the simulated system and then loaded into memory. This kind of method induces a lot of memory operations for little computation and is therefore memory-bound.

With reference to on-the-fly Doppler broadening method, instead of storing all cross section data in memory, only the cross section data of a baseline temperature is stored and the temperature dependence is calculated on-the-fly during the simulation. This kind of method consumes less memory but is much more compute intensive.

#### 3.4.4.1 Macroscopic Cross Section Lookup Algorithm

As shown in Algorithm 5, the macroscopic cross section of a given material accumulates with the multiplications of microscopic cross sections and atomic densities of the nuclides composing the material. This algorithm has well vectorization potential since the loop size is large enough (from tens of to hundreds of nuclides) and the workload assigned to each processing unit is equal to others which gets rid of the performance penalty caused by imbalance. Bergmann [18] implemented a macroscopic cross section kernel where in addition to the total macroscopic cross section, other values related to it are computed as well, including interaction distance, collided nuclide and so on. These extra calculations are equally distributed to all processing units and thus the workload balance is nicely maintained.

**Algorithm 5:** Macroscopic cross section lookup.

**Input:** a group of $M$ couples of microscopic cross sections, atomic densities, $\{\sigma_i, N_i\}_{i \in M}$, corresponding to $M$ nuclides which compose a given material

**Result:** calculated macroscopic cross section $\Sigma$ of the material

```
1 Σ = 0;
2 foreach composing nuclide of the material, indexed with i, i ∈ M do
3     Σ += N_i σ_i;
4 end
```

#### 3.4.4.2 Pre-tabulated Microscopic Cross Section Calculation Algorithm

Pre-tabulated approach mainly consists two steps, an energy lookup and an interpolation. Typically, the continuous-energy cross section data is stored as a long table of cross sections (cross section grid) which is bijective to another

long table of energy points (energy grid). In this case, an energy lookup with input energy $E$ is indeed to find the lower-upper bound $E_k$ and $E_{k+1}$ with the relation $E_k \leqslant E \leqslant E_{k+1}$. The corresponding microscopic cross section is then retrieved by one of linear-linear interpolations as described in Equation 3.8.

$$\begin{aligned} \sigma(E) &= \sigma(E_k) + \tfrac{E-E_k}{E_{k+1}-E_k}[\sigma(E_{k+1}) - \sigma(E_k)] \\ \sigma(E) &= \sigma(E_{k+1}) - \tfrac{E_{k+1}-E}{E_{k+1}-E_k}[\sigma(E_{k+1}) - \sigma(E_k)] \end{aligned} \tag{3.8}$$

The key nature of such linearly-interpolable cross section data is that energy grid sizes highly depend on their corresponding nuclides. Some nuclides have hundreds of energy points (e.g. $^{1}$H, $^{3}$H) whereas other nuclides have more than a million energy points (e.g. $^{238}$U). This feature leads to considerable variations of workloads between energy lookup processes through different nuclides and makes efficiency of energy lookup algorithms nuclide-dependent via different search schemes.

With respect to workload imbalance, some methods such as unionized energy grid [74], fractional cascading [79], hash map [24][125], N-ary map [127] have been proposed to address this issue by reconstructing energy grids and supplementing additional mapping tables to effectively reduce iteration path. The basic logic of these methods is either to expand energy tables of various lengths to a same length or to compress searching ranges of long energy tables into narrower ones.

With respect to search schemes, binary search and linear search are two major approaches commonly used to perform energy lookups. The computational complexity of binary search is $O(log_2 N)$ while linear search is $O(N)$, $N$ is the number of elements in table. Comparing to linear search, binary search has optimized computational complexity which gains much better performance with a large problem size. However, the random memory access pattern of binary search leads to bad memory coalescing and high cache misses, giving it few chances to be vectorized with SIMD or SIMT optimizations [111].

Previous work done by Wang [129] has tested the search schemes, reconstruction and mapping techniques mentioned above with 512-bit SIMD optimizations. The results shows that the threshold table length for choosing between vectorized linear search and binary search is 200. Such behavior is informative and indicates that with uses of mapping techniques such as hashing and N-ary mapping, it may contribute to performance improvement if the vectorized linear search is adopted for energy lookups through narrow bins. Besides, the vectorized binary search, also called as N-ary search, yield worse performance compared with binary search by reason of higher cache misses that are caused by extra non-coalesced memory access. As for reconstruction and mapping techniques, unionized energy grid method contributes to the best performance improvement whereas N-ary mapping the worst. In general, there are few opportunities to exploit vectorization for Monte Carlo algorithms based on pre-tabulated cross section where latency-bound energy lookups take up most of computing time. As a solution, the compute-intensive FLOP work between frequent memory loads introduced by on-the-fly Doppler broadening may mitigate the latency-bound bottleneck mainly induced by the binary search in the pre-tabulated approach [119].

**Algorithm 6:** Basic pre-tabulated algorithm with binary search scheme.

**Input:** a given energy $E$, the energy grid $eg[N]$ and cross section grid $csg[N]$ of a nuclide with grid size equaling to $N$

**Result:** calculated microscopic cross section $\sigma(E)$ of the nuclide

1 $k = getLowerBound(E, eg)$; /* $0 \leqslant k < N-1$ and $eg[k] < E \leqslant eg[k+1]$ */

2 $\sigma(E) = csg[k] + \frac{E - eg[k]}{eg[k+1] - eg[k]}(csg[k+1] - csg[k])$; /* linear-linear interpolation */

Now we only consider a basic pre-tabulated method using binary search scheme without the introduction of reconstructed cross section data and mapping techniques as a comparison of on-the-fly Doppler broadening algorithm expressed in section 3.4.4.3. The general procedure of this method is illustrated in Algorithm 6 where $getLowerBound()$ is a binary search function retrieving the last element which is lower than the input energy in the input energy grid.

#### 3.4.4.3 On-the-fly Doppler Broadening Microscopic Cross Section Calculation Algorithm

On-the-fly Doppler broadening algorithms calculate temperature-dependent cross sections during the Monte Carlo transport simulation which reduces memory footprints at the expense of much higher computational cost. In addition to the SIGMA1 on-the-fly Doppler broadening method which performs piecewise-linear exact integration, some other methods have also been proposed to reconstruct cross sections by various multipole representations and regression models [60][66][136].

**Algorithm 7:** Basic SIGMA1 on-the-fly Doppler broadening algorithm.

**Input:** a given energy $E$ and temperature $T$, the energy grid $eg[N]$ and cross section grid $csg[N]$ of a nuclide with grid size equaling to $N$, the elements in $csg$ are of a baseline temperature $T_0$

**Result:** calculated microscopic cross section $\sigma(E)$ of the nuclide

1 calculate $E_1$ and $E_2$ based on input parameters;

2 $lb = getLowerBound(E_1, eg)$, $ub = getLowerBound(E_2, eg) + 1$; /* $0 \leqslant lb < ub \leqslant N-1$ */

3 $\sigma(E) \mathrel{+}= compute_integral(lb, ub, eg, csg, \ldots)$; /* integral computation between $[lb, ub]$ */

A basic SIGMA1 on-the-fly Doppler broadening approach can be abstracted as a process to retrieve lower-upper bound of integral range along with another process of integral computation. In Algorithm 7, $lb$ and $ub$ are calculated by the same binary search function $getLowerBound()$ described in Algorithm 6 with inputs $E_1$, $E_2$ and $eg$. The $compute_integral()$ function is an integral computation kernel approximated from Equation 3.6 which only considers calculations inside the range $-4 \leqslant x - y < 4$ since the term $e^{-(x-y)^2}$ becomes negligible when $|x - y|$ is large [38].

### 3.4.5 The PATMOS Monte Carlo Prototype

There are many simulation codes developed to solve Monte Carlo transport problems, such as MCNP [118], TRIPOLI [27], Serpent [75], RMC [126], OpenMC [105], Shift [96] as well as the code used in our thesis, PATMOS [28].

PATMOS is a prototype of Monte Carlo neutron transport under development at CEA dedicated to the testing of algorithms for high-performance computations on modern architectures [28]. One of the goals is to perform pin-by-pin full core depletion calculations for large nuclear power reactors with realistic temperature fields. Numerical results have been verified by comparisons with TRIPOLI-4® [27] and MCNP5 [118].

The physics of PATMOS is simplified with two types of particles (mono-kinetic pseudo-particles and neutrons). Four types of physical interactions including elastic scattering, discrete inelastic scattering, absorption and simplified fission have been implemented. The scoring part is encapsulated into a scorer class which deals with tally computation during the simulation and gathers statistical results afterwards.

PATMOS relies on a hybrid parallelism based on MPI for distributed memory parallelism and OpenMP, C++ native threads or Intel TBB for shared memory parallelism. It is entirely written in C++, with a heavy use of polymorphism in order to always allow the choice between competing algorithms such as the mix of nuclides with pre-computed Doppler-broadened cross sections and on-the-fly Doppler broadening.

Several energy lookup algorithms have been implemented in PATMOS with much vectorization effort to improve performance especially for pre-tabulated method dominated by energy lookup process. The SIGMA1 on-the-fly Doppler broadening method used in NJOY as the reference method [81] has also been implemented in PATMOS with some effort to reorganize the main loop of the SIGMA1 algorithm which in turn yields the reconstructed main loop a better potential for vectorization. This on-the-fly Doppler broadening algorithm has been ported to Nvidia GPUs as a rewritten CUDA kernel which allows CPU threads offloading the compute-intensive workloads of cross section calculations to multiple GPUs and copying back results for further simulation.

In this chapter, we give a brief introduction about nuclear reactor physics, neutron transport simulation, as well as Monte Carlo method. The state-of-the-art progress of HPC and Monte Carlo neutron transport codes are discussed with respect to parallelism, particle tracking methods, cross section computing algorithms.

# Chapter 4

# Portable Implementation of on-the-fly Doppler Broadening in PATMOS

Chapter 2 and Chapter 3 have presented the context of this thesis work. In this chapter, the implementation of the SIGMA1 Doppler Broadening algorithm in PATMOS with the chosen programming models are described. Then, the performance results of these different implementations are analyzed on a benchmark executed on several platforms [31]. At last, performance profiling and optimization are presented.

## 4.1 Implementation of SIGMA1 on-the-fly Doppler Broadening Algorithm

This section presents the SIGMA1 Doppler broadening algorithm which allows, from a cross section at a base temperature (typically 0 Kelvin), to compute the cross section at a given temperature and a given energy.

As we have illustrated in Algorithm 7, our SIGMA1 on-the-fly Doppler broadening algorithm mainly consists in computing the indexes corresponding to the lower and upper bounds of the interval of integration (function $getLowerBound()$) and then compute the integral of Equation 3.6 (function $compute_integral()$) in which the cross section has the following form:

$$\sigma(x) \quad = \quad \sigma_i + s_i(x^2 - x_i^2) \quad = \quad \sigma_{i+1} - s_i(x_{i+1}^2 - x^2) \tag{4.1}$$

where $s_i = \frac{\sigma_{i+1} - \sigma_i}{x_{i+1}^2 - x_i^2}$.

According to Equation 3.6 and Equation 4.1, we have:

$$\begin{aligned}
\sigma(y,T) &= \sigma^*(y,T) - \sigma^*(-y,T) \\
\sigma^*(y,T) &= \tfrac{1}{y^2}\left(\tfrac{1}{\pi}\right)^{1/2}\int_0^\infty x^2\sigma(x,T_0)\,e^{-(x-y)^2}dx \\
&\approx \tfrac{1}{y^2}\left(\tfrac{1}{\pi}\right)^{1/2}\sum_i\int_{x_i}^{x_{i+1}} x^2\sigma(x,T_0)\,e^{-(x-y)^2}dx \\
&\approx \tfrac{1}{y^2}\left(\tfrac{1}{\pi}\right)^{1/2}\sum_i\left[\int_{x_i}^{x_{i+1}} x^2\left(\sigma_i - s_i x_i^2\right)e^{-(x-y)^2}dx + \int_{x_i}^{x_{i+1}} x^4 s_i e^{-(x-y)^2}dx\right]
\end{aligned} \tag{4.2}$$

Assume $z = x - y$, we have:

$$\begin{aligned}
\sigma^*(y,T) &= \sum_i\left(\tfrac{1}{\pi}\right)^{1/2}\int_{x_i-y}^{x_{i+1}-y}\left(\tfrac{1}{y^2}z^2 + \tfrac{2}{y}z + 1\right)\left(\sigma_i - s_i x_i^2\right)e^{-z^2}dz \\
&\quad + \sum_i\left(\tfrac{1}{\pi}\right)^{1/2}\int_{x_i-y}^{x_{i+1}-y}\left(\tfrac{1}{y^2}z^4 + \tfrac{4}{y}z^3 + 6z^2 + 4yz + y^2\right)s_i e^{-z^2}dz \\
&= \sum_i\left\{A(x_i-y)\left[\sigma_i - s_i x_i^2\right] + B(x_i-y)\,s_i\right\} \\
&\quad - \sum_i\left\{A(x_{i+1}-y)\left[\sigma_{i+1} - s_i x_{i+1}^2\right] + B(x_{i+1}-y)\,s_i\right\} \\
&= A(x_0-y)\left[\sigma_0 - s_0 x_0^2\right] + B(x_0-y)\,s_0 \\
&\quad + \sum_{i=1}^{N}\left[s_i - s_{i-1}\right]\left\{B(x_i-y) - A(x_i-y)\,x_i^2\right\} \\
&\quad - A(x_{N+1}-y)\left[\sigma_{N+1} - s_N x_{N+1}^2\right] + B(x_{N+1}-y)\,s_N
\end{aligned} \tag{4.3}$$

where

$$\begin{aligned}
A(a) &= \tfrac{1}{y^2}F_2(a) + \tfrac{2}{y}F_1(a) + F_0(a) \\
B(a) &= \tfrac{1}{y^2}F_4(a) + \tfrac{4}{y}F_3(a) + 6F_2(a) + 4yF_1(a) + y^2F_0(a)
\end{aligned} \tag{4.4}$$

with $F_n(a) = (\frac{1}{\pi})^{1/2}\int_a^\infty z^n e^{-z^2}dz$, which can be expanded as follows:

$$\begin{aligned}
F_0(a) &= \frac{1}{2}erfc(a) \\
F_1(a) &= \frac{1}{2\sqrt{\pi}}exp(-a^2) \\
F_2(a) &= \frac{1}{4}erfc(a) + \frac{1}{2\sqrt{\pi}}exp(-a^2)a \\
F_3(a) &= \frac{1}{2\sqrt{\pi}}exp(-a^2)(1+a^2) \\
F_4(a) &= \frac{3}{8}erfc(a) + \frac{1}{2\sqrt{\pi}}exp(-a^2)(\frac{3}{2}a + a^3)
\end{aligned} \tag{4.5}$$

where $erfc$ and $exp$ are respectively the complementary error function (see Equation 4.6) and the base $e$ exponential function.

$$erfc(a) = \frac{2}{\sqrt{\pi}}\int_a^\infty e^{-z^2}dz \tag{4.6}$$

**Algorithm 8:** Function of $compute_integral()$.

**Input:** a given energy $E$, a coefficient $\alpha$, a lower-upper bound $lb$, $ub$ and an energy grid $eg[N]$, a cross section grid $csg[N]$

**Result:** accumulated cross section $\sigma$ after integral computation

1 $\sigma = 0$, $y = \sqrt{\alpha E}$;

`/* Accumulation of the first element */`

2 $index = lb$, $ea = eg[index], eb = eg[index + 1]$;

3 $x = \sqrt{\alpha \cdot ea}, a = x - y$;

4 calculate $F_0(a)$, $F_1(a)$, $F_2(a)$, $F_3(a)$, $F_4(a)$; `/* According to Equation 4.5 */`

5 calculate $A$ and $B$ based on $F_n(a)$; `/* According to Equation 4.4 */`

6 $siab = \dfrac{csg[index + 1] - csg[index]}{\alpha(eb - ea)}$;

7 $\sigma \mathrel{+}= A(csg[index] - siab \cdot \alpha \cdot ea) + B \cdot siab$;

`/* Accumulation from the second element to the second last element */`

8 **for** $index \leftarrow lb + 1$ **to** $ub - 1$ **by** $1$ **do**

9 $\quad ea = eg[index - 1], eb = eg[index], ec = eg[index + 1]$;

10 $\quad x = \sqrt{\alpha \cdot eb}, a = x - y$;

11 $\quad$ calculate $F_0(a)$, $F_1(a)$, $F_2(a)$, $F_3(a)$, $F_4(a)$; `/* According to Equation 4.5 */`

12 $\quad$ calculate $A$ and $B$ based on $F_n(a)$; `/* According to Equation 4.4 */`

13 $\quad siab = \dfrac{csg[index] - csg[index - 1]}{\alpha(eb - ea)}$;

14 $\quad sibc = \dfrac{csg[index + 1] - csg[index]}{\alpha(ec - eb)}$;

15 $\quad \sigma \mathrel{+}= (sibc - siab)(B - A \cdot \alpha \cdot eb)$;

16 **end**

`/* Accumulation of the last element */`

17 $index = ub$, $ea = eg[index - 1], eb = eg[index]$;

18 $x = \sqrt{\alpha \cdot eb}, a = x - y$;

19 calculate $F_0(a)$, $F_1(a)$, $F_2(a)$, $F_3(a)$, $F_4(a)$; `/* According to Equation 4.5 */`

20 calculate $A$ and $B$ based on $F_n(a)$; `/* According to Equation 4.4 */`

21 $siab = \dfrac{csg[index] - csg[index - 1]}{\alpha(eb - ea)}$;

22 $\sigma \mathrel{-}= A(csg[index] - siab \cdot \alpha \cdot eb) + B \cdot siab$;

Based on Equation 4.3, we have the algorithm of the function $compute_integral()$ with a given energy, lower and upper bounds of the interval of integration, a coefficient $\alpha = \frac{M}{k_B(T - T_0)}$ and cross section grids as inputs (see Algorithm 8).

The calculations of $F_n(a)$ fucntions call $erfc$ and $exp$ functions multiple times which accounts for a large part of the computing time. A CPU-based profiling of a typical Monte Carlo neutron transport benchmark, slabAllNuclides, implemented in PATMOS in terms of run time percentage is illustrated in Table 4.1 in which all results were retrieved by `C++ native clock` and `perf` (more detail about slabAllNuclides is elaborated in section 4.4.1).

Table 4.1: Typical Monte Carlo neutron transport run time percentage in PATMOS.

| **Processing Step** | **Run Time Percentage** (%) |
|---|---|
| Total Cross Section | 95.4 |
| $exp$ | 17.6 |
| $erfc$ | 49.4 |
| $getLowerBound$ | 2.4 |
| $compute_integral$ | 79.2 |
| Partial Cross Section | 1.7 |
| $exp$ | 0.2 |
| $erfc$ | 0.6 |
| $getLowerBound$ | 0.1 |
| $compute_integral$ | 1.4 |
| Initialization | 1.8 |
| $buildMedium$ | 1.5 |
| Others | 1.1 |

The total cross section calculations account for up to 95% of total run time where the most computational cost comes from the functions $erfc$ and $exp$ which take up respectively 49% and 18% of total run time. Besides, comparing to the run time percentage of $compute_integral$ function which gains around 79% of total run time, the function $getLowerBound$ is quite negligible with only 2% run time percentage.

Thus, we adopt a heterogeneous offloading strategy for our portable implementations of Monte Carlo code where only the cross section calculation is offloaded on the accelerator. The rest of the calculation remains on the host. Each accelerator can be considered as a cross section server that can be queried during the particle tracking process, by one or several host threads, to compute cross sections at a given energy and temperature. To achieve this heterogeneous offloading strategy, the main development effort is to rewrite codes which are dedicated to porting nuclide data and microscopic cross section calculations to devices via implementations of several programming models. The next section gives more details about the difference.

## 4.2 Implementations in the Different Programming Models

PATMOS allows two levels of parallelism via MPI + Multi-thread libraries (OpenMP, C++ threads). Each MPI rank performs history tracking for a batch of particles and the average scores of simulation are calculated via inter-node communications. In the shared memory parallellism, each CPU thread is in charge of tracking a group of particles. It allows all threads to share the non-mutable data and scores which are concurrently updated through atomic memory

operations.

In order to address architectures where multiple accelerators are associated with a host upon a single node, we may use either “Multiple Threads, Single Accelerator (MTSA)” or “Multiple Threads, Multiple Accelerators (MTMA)” strategies. The main difference between them is that MTSA requires one MPI process using a single accelerator while MTMA allows multiple threads of a MPI rank to target to multiple accelerators. MTMA avoids extra launch latency and memory footprint on a single node, which makes it a memory-friendly way for our heterogeneous offloading strategy.

Since the offloading part of our MC simulation introduces no inter-node communication, we do not consider MPI in this thesis work but only `OpenMP thread + {X}`, where `{X}` can be any languages or libraries which are capable of parallel programming on modern accelerators. In this thesis work, we have considered CUDA, OpenACC, OpenMP offload, Kokkos, SYCL or none at all (called OpenMP thread implementation in the following).

### 4.2.1 Code Architecture

The rewritten work consists in developing a set of derived classes inheriting from three base classes that include `Nuclide`, `NeutronMedia` and `NeutronMediaNavigator`.

The base class `Nuclide` engages in storing basic information related to a given nuclide such as its atomic weight (AW), energy and cross section grids of a particular base temperature as well as its table length. The derived classes of `Nuclide` are implemented via different programming languages or libraries to accomplish the nuclide data transfer from host to devices. The UML (unified modeling language) diagram of class `Nuclide` along with its derived classes is depicted in Figure 4.1.

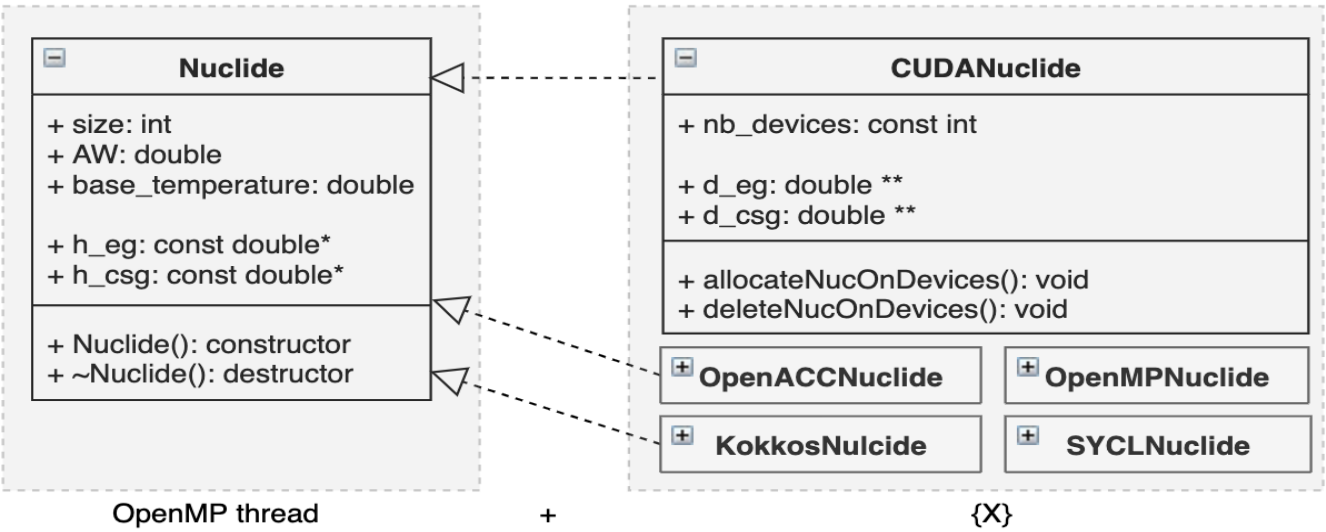



Figure 4.1: Inheritance relationship between base class `Nuclide` and its derived classes.

The base class `NeutronMedia` is designed to store material and nuclide data which is required to perform a Monte Carlo neutron transport simulation as illustrated in Figure 4.2. It contains a function $makeNavigator()$

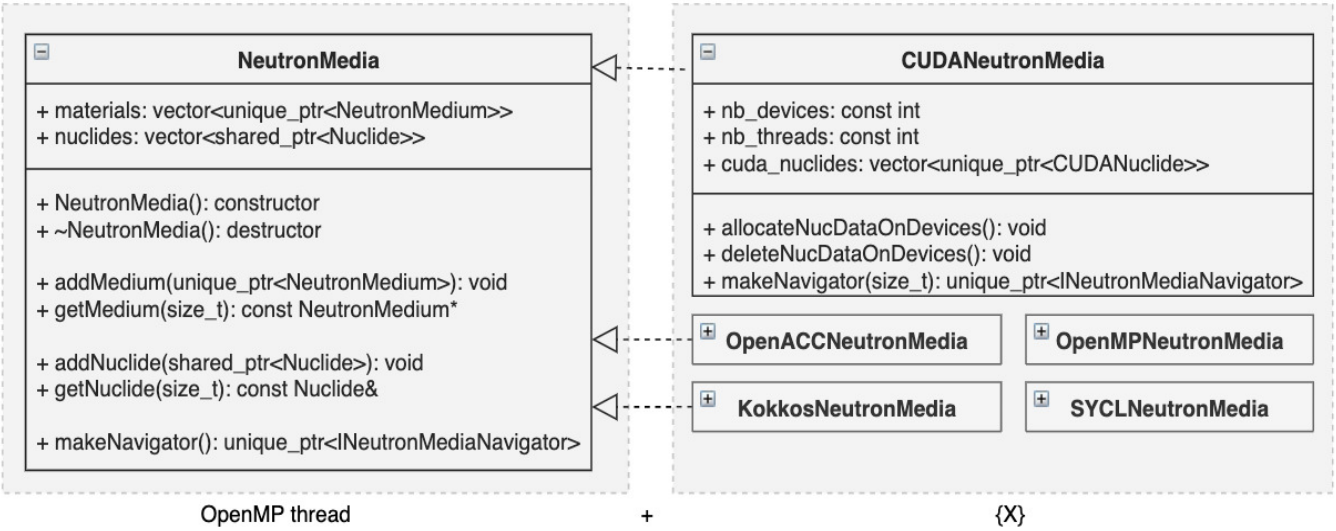



Figure 4.2: Inheritance relationship between base class `NeutronMedia` and its derived classes.

which returns a pointer referring to an instance of the class `NeutronMediaNavigator` or its derived classes such as `CUDANeutronMediaNavigator`. Each pointer is distributed to a CPU thread and responsible for undergoing iterations of particle history tracking. The derived classes of `NeutronMedia` are dedicated to storing the total number of devices and threads, creating all objects of derived class of `Nuclide`, transfering them from host to devices with the call of the function $allocateNucDataOnDevices()$ and deallocating them from devices with the invocation of the function $deleteNucDataOnDevices()$. The function $makeNavigator()$ of the derived classes of `NeutronMedia` is parameterized by a value of type `size_t` indicating the thread ID of the corresponding instance of the class `NeutronMediaNavigator`.

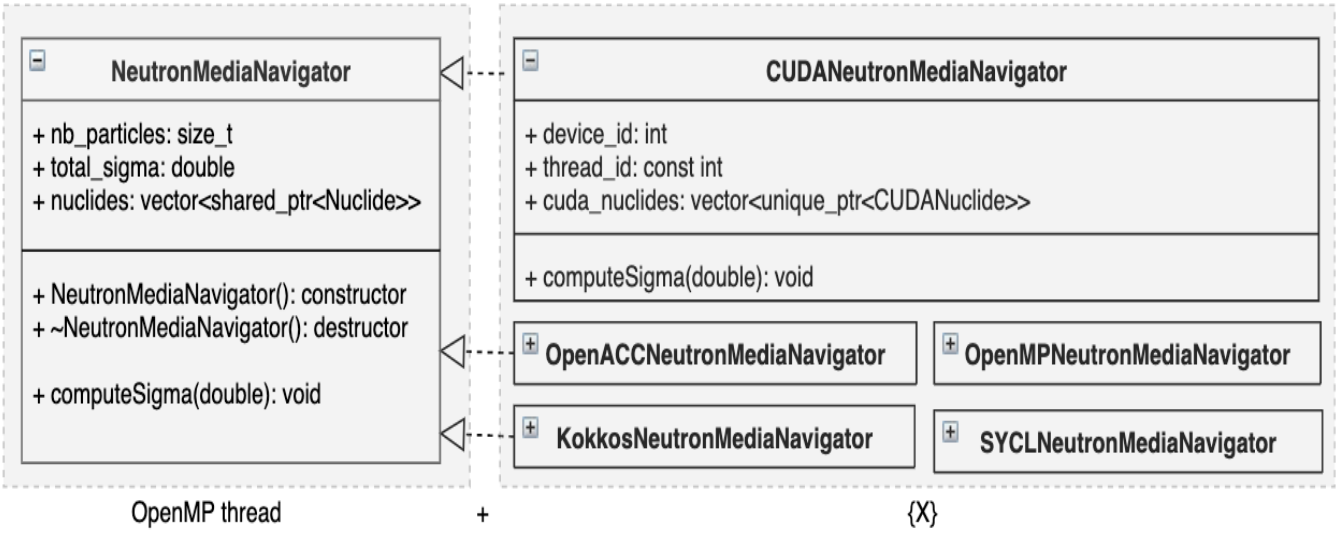



Figure 4.3: Inheritance relationship between base class `NeutronMediaNavigator` and its derived classes.

At last, the base class `NeutronMediaNavigator` handles the iterations of particle history tracking which include

mainly the total cross section calculations, the random samples of distance, collided nuclide, interaction type as well as kinetic state of the tracked particle after intereaction. The core part, total cross section calculation, is put in the function $computeSigma()$ which differs a lot between its derived classes. The attribute `device_id` is retrieved by the `thread_id` in cooperation with the total number of devices `nb_devices`, as described in the formula `device_id = thread_id % nb_devices` where % is the operator to obtain the remainder. Figure 4.3 shows the diagram of the class `NeutronMediaNavigator`.

**Algorithm 9:** Pseudo-code of OpenMP thread implementation.

```
1  void NeutronMediaNavigator::computeSigma(...){
2    total_sigma = 0;
3    for(int t=0; t<N; ++t){                    /* calculate microscopic cross sections of N nuclides */
4      sigma1DopplerBroadening(t, h_macroscopic_cs, ...);
5    }
6    for(int t=0; t<N; ++t){                     /* calculate macroscopic cross section of N nuclides */
7      total_sigma += h_macroscopic_cs[t] * h_concentrations[t];
8    }
9  }
10 void sigma1DopplerBroadening(...){
11   h_macroscopic_cs[i] = 0;
12   E1 = ...; E2 = ...;
13   lb = getLowerBound(E1, ...); ub = getLowerBound(E2, ...) + 1;
14   h_macroscopic_cs[i] += compute_integral(lb, ub, ...);
15 }
16 double compute_integral(...){
17   double sigma = 0;
18   int index = lb; sigma += ...;
19   #pragma omp simd reduction(+: sigma)
20   for(int i=lb+1; i<ub; ++i){
21     sigma += ...;
22   }
23   index = ub; sigma -= ...;
24   return sigma;
25 }
```

In general, the parallelization effort is made with the function $sigma1DopplerBroadening()$, as a part of the function $computeSigma()$ in the class `NeutronMediaNavigator` as well as its derived classes to vectorize the SIGMA1

on-the-fly Doppler broadening algorithm on host or accelerators. Furthermore, since the function $getLowerBound()$ takes up a rather negligible run time percentage comparing to that of the function $compute_integral()$, we basically focus on the parallelization of $compute_integral()$, more specifically, the parallelization of the `for` loop in the integral computation.

### 4.2.2 OpenMP Thread Implementation

The OpenMP thread version does not induce nuclide data transfer between host and devices, thus the single point to address is the OpenMP SIMD directive that we adopt to vectorize the `for` loop in the function $compute_integral()$, as shown in Algorithm 9 (at line 19). Besides, this loop can be vectorized by auto-vectorization of Intel compiler since version 16. It should be noted that the OpenMP thread version is the reference implementation on CPU.

### 4.2.3 CUDA Implementation

With respect to other portable implementations which allow for offloading microscopic total cross section calculations to accelerators, the rewritten parts are designed to accomplish the nuclide data transfer between host and devcies as well as the microscopic cross section calculations on devices. The data management of a particular nuclide is coded in the functions $allocateNucOnDevices()$ and $deleteNucOnDevices()$ of derived classes inherited from the base class `Nuclide`. Furthermore, the functions $allocateNucDataOnDevices()$ and $deleteNucDataOnDevices()$ of child classes inherited from parent class `NeutronMedia` perform a `for` loop to recursively allocate, copy and remove data of all nuclides. The microscopic cross section calculations are offloaded by parallelizing the `for` loop in the function $computeSigma()$ of derived classes inherited from the base class `NeutronMediaNavigator` (Algorithm 9 at lines 3–5).

The parallelization is achieved by distributing each call of a function $sigma1DopplerBroadening()$ in the outer-loop to a work-group of device where each work-group is composed of a set of work-items which are responsible for vectorizing the inner-loop in the function $compute_integral()$. For example, the default strategy for Nvidia GPUs is to assign the workload of microscopic cross section calculation of each nuclide to a CUDA warp so as to achieve a task-level parallelism. And all CUDA threads within a warp are used to vectorize the inner-loop of integral computation as a completion of data-level parallelism. Our reference implementation on GPU achieved by CUDA is shown in Algorithm 10.

The functions $allocateNucOnDevices()$ and $deleteNucOnDevices()$ in the class `CUDANuclide` calls the functions of the CUDA runtime API $cudaMalloc()$, $cudaMemcpy()$ as well as $cudaFree()$ to allocate, copy and delete the data of a given nuclide on multiple devices (at lines 1–9). Note that the function $cudaSetDevice()$ is used to set the current device associated to the corresponding host thread for kernel execution.

**Algorithm 10:** Pseudo-code of CUDA implementation.

```
1  void CUDANuclide::allocateNucOnDevices(...){
2    for(int idev=0; idev<nb_devices; ++idev){        /* allocate and copy the nuclide data to devices */
3      cudaSetDevice(idev); cudaMalloc(...); cudaMemcpy(..., cudaMemcpyHostToDevice);
4    }
5  }
6  void CUDANuclide::deleteNucOnDevices(...){
7    for(int idev=0; idev<nb_devices; ++idev)                 /* remove the nuclide data from devices */
8      cudaSetDevice(idev); cudaFree(...);
9  }
10 void CUDANeutronMediaNavigator::computeSigma(...){
11   dim3 blocksize(32, 1); int gridsize = N;                                  /* set CUDA kernel dimension */
12   cudaMalloc(...); cudaMemcpyAsync(..., cudaMemcpyHostToDevice, stream);
13   kernel_launch<<<gridsize, blocksize, 0, stream>>>(...);                          /* launch CUDA kernel */
14   cudaMemcpyAsync(..., cudaMemcpyDeviceToHost, stream); cudaFree(...);
15   total_sigma = 0; ...                                     /* macroscopic cross section calculation */
16 }
17 __global__ void kernel_launch(...){
18   int t = blockIdx.x;                                                                  /* ensure t < N */
19   sigma1DopplerBroadening(t, ...);
20 }
21 __device__ inline double compute_integral(...){
22   double sigma = 0; double sigma_tmp = 0;
23   int index = lb; sigma += ...;
24   for(int i=lb+1+threadIdx.x; i<ub; i+=32)
25     sigma_tmp += ...;
26   for(int mask=16; mask>0; mask/=2)
27     sigma_tmp += __shfl_xor_sync(...);
28   sigma += sigma_tmp;
29   index = ub; sigma -= ...;
30   return sigma;
31 }
```

As for the outer-loop in the function $computeSigma()$ of class `CUDANeutronMediaNavigator` calling a function $sigma1DopplerBroadening()$ during each iteration, a CUDA kernel $kernel_launch()$ is replaced to parallelize this loop along with two asynchronous memory copies $cudaMemcpyAsync()$ where all three functions are enabled with

CUDA multi-stream support owning a single stream associated to the corresponding host thread (at lines 12–14). This CUDA asynchronous multistreaming technique allows for concurrent CUDA calls initiated by different host threads. By contrast, operations (data transfer and kernel calculation) with the same stream called by a single host thread are executed sequentially and cannot overlap. With reference to the computational kernel $kernel_launch()$ (at lines 17–20), its default dimension is set to `N` threadblocks, where each threadblock contains one CUDA warp. Note that `N` is the gridsize of kernel (at line 11). Each warp is assigned to calculate the microscopic cross section of a single nuclide recorded by the task index `t = blockIdx.x`.

At last, the inner-loop in the function $compute_integral()$ is vectorized by CUDA threads within a warp. Each CUDA thread handles a part of the workload and store the partial result in `sigma_tmp`. A sum reduction is then operated by resorting to a warp shuffle instruction $__shfl_xor_sync()$ to aggregate the total value of `sigma_tmp` among CUDA threads within the warp.

Generally, from the thread hierarchy perspective, our CUDA implementation makes use of two levels of work units `warp-thread` to achieve the task-level and data-level parallelism of the SIGMA1 on-the-fly Doppler broadening algorithm. From the memory hierarchy point of view, the programmable memory including registers and global memory are adopted to accomplish data management such as load/store and warp shuffle operations. Shared memory is not considered for the initial design as other programming models such as OpenACC and OpenMP offload do not yet offer programming interface to shared memory.

### 4.2.4 OpenACC Implementation

The OpenACC implementation is quite similar to the CUDA version as the CUDA runtime instructions are replaced by OpenACC directives to achieve nuclide data transfer between host and devices and offload microscopic cross section calculations to accelerators.

In order to handle the data management of memory allocation, movement and removal between host and devices, the directives `#pragma acc {enter/exit} data {copyin/copyout/delete}` are adopted as shown in Algorithm 11 (at lines 4, 10, 14, 18). The OpenACC runtime function $acc_set_device_num()$ is used to set current device which may be CPU host, Nvidia GPU and so on. Moreover, the device pointers that are associated with the related host pointers by the directive `#pragma acc enter data copyin` can be retrieved by another OpenACC runtime function $acc_deviceptr()$.

Concerning the parallelization of SIGMA1 on-the-fly Doppler broadening method implemented in the function $computeSigma()$ of class `OpenACCNeutronMediaNavigator`, our OpenACC version adopts the directive `#pragma acc parallel loop gang` to parallelize the outer-loop and the directive `#pragma acc loop vector` the inner-loop. The number of parallel gangs for the outer-loop is set to `N` by clause `num_gangs(N)` which corresponds to the gridsize `N` in the CUDA implementation. The length of vector unit for the inner-loop is defined by clause `vector_length(32)` which can be interpreted as 32 CUDA threads within a warp under the condition that the device type is set to Nvidia.

**Algorithm 11:** Pseudo-code of OpenACC implementation.

```
1  void OpenACCNuclide::allocateNucOnDevices(...){
2    for(int idev=0; idev<nb_devices; ++idev){
3      acc_set_device_num(idev, device_type);                    /* set current device */
4      #pragma acc enter data copyin(...)                    /* copy nuclide data to device */
5    }
6  }
7  void OpenACCNuclide::deleteNucOnDevices(...){
8    for(int idev=0; idev<nb_devices; ++idev){
9      acc_set_device_num(idev, device_type);                    /* set current device */
10     #pragma acc exit data delete(...)                  /* remove nuclide data from device */
11   }
12 }
13 void OpenACCNeutronMediaNavigator::computeSigma(...){
14   #pragma acc enter data copyin(...) async(stream)
15   #pragma acc parallel loop gang num_gangs(N) vector_length(32) async(stream) present(...)
16   for(int t=0; t<N; ++t)
17     sigma1DopplerBroadening(t, ...);
18   #pragma acc exit data copyout(...) async(stream)
19   total_sigma = 0; ...                          /* macroscopic cross section calculation */
20 }
21 inline double compute_integral(...){
22   double sigma = 0; double sigma_tmp = 0;
23   int index = lb; sigma += ...;
24   #pragma acc loop vector reduction(+: sigma_tmp)
25   for(int i=lb+1; i<ub; ++i)
26     sigma_tmp += ...;
27   sigma += sigma_tmp;
28   index = ub; sigma -= ...;
29   return sigma;
30 }
```

The clause `present(...)` ensures that the data required by the function $sigma1DopplerBroadening()$ is already copied to the device. The clause `async(stream)` where `stream` may be a nonnegative scalar integer expression makes the host multi-threads possible to process kernel computations or data operations in parallel, which equals to the CUDA asynchronous multi-stream mechanism. Besides, the reduction operation can be achieved by using

the clause `reduction(operator:vars)` on a loop to perform one type of operation for one or more variables. In our case, the variable `sigma_tmp` is involved in a sum reduction that spans the inner-loop with the `vector` clause and the aggregated value of this variable undergoing the sum reduction will be updated at the exit of the inner-loop.

### 4.2.5 OpenMP Offload Implementation

Like the OpenACC version, the OpenMP offload implementation also relies on the utilization of directives and runtime functions to execute some portion of the code on the device. However, there are several differences between them which need to be highlighted.

Firstly, although OpenMP target specification offers the directives to manage data transfer between host and devices including `#pragma omp target {enter/exit} data`, as far as we know, it doesn't give any interface to access the device addresses allocated by these directives of data management as what the function $acc_deviceptr()$ does in OpenACC specification. Since these device pointers are required to be stored in the data structure that will be transferred between host and device during each call of the function $computeSigma()$, we directly use some OpenMP runtime functions dedicated to performing data management such as $omp_target_alloc()$, $omp_target_memcpy()$ and $omp_target_free()$ where $omp_target_alloc()$ allocates memory in a device data environment and returns the corresponding device address (at lines 3, 4, 9 in Algorithm 12).

Secondly, the directives adopted in the function $computeSigma()$ which are responsible for data transfer and kernel launch, `#pragma omp target {data map/teams distribute` have no clause specifying the asynchronous stream. Despite the fact that OpenMP offload does offer a couple of directives `#pragma omp task`, `#pragma omp taskwait` as well as a clause widely used to specify a set of directives denoted as `nowait`, such techniques only allow the corresponding code region being executed asynchronously with respect to the current host thread. Thus, OpenMP offload is not able to provide CUDA asynchronous multistreaming support as OpenACC which achieves this functionality by using the clause `async(stream)`.

It should be noted that the map-type of the directive `#pragma omp target data map` that we use in our implementation is `tofrom` or `to`. The map-type `tofrom` signifies the mapping of data from host to device at the start of the region and the update of data from device to host at the end of the region while the map-type `to` refers to barely the mapping of variable to the device data environment. Furthermore, the outer-loop parallelized by the directive `#pragma omp target teams distribute` are specified by the clauses `num_teams(N)` and `thread_limit(32)` which define respectively `N` parallel teams and the upper limit of 32 threads per team.

As for the parallelization of the inner-loop in the function $compute_integral()$, the directive `#pragma omp parallel for` is used along with the clause `reduction(+:sigma_tmp)` so as to perform a sum reduction for the variable `sigma_tmp`. Besides, the clause `device(idev)` utilized in directives `#pragma omp target [data]` creates the data environment on the device of ID `idev` and the clause `if(target [data]:device_enabled)` makes the target region in the host data environment when the scalar expression represented by `device_enabled` evaluates to zero. The

variable `device_enabled` is set at runtime depending on the variable `nb_devices` that is returned by the OpenMP API runtime function $omp_get_num_devices()$ as `device_enabled = 0`, if `nb_devices = 0`; `device_enabled = 1`, otherwise.

**Algorithm 12:** Pseudo-code of OpenMP offload implementation.

```
1  void OpenMPNuclide::allocateNucOnDevices(...){
2    for(int idev=0; idev<nb_devices; ++idev){
3      auto this->d_eg[idev] = omp_target_alloc(..., idev);          /* allocate memory on device */
4      omp_target_memcpy(..., idev, ihost);                          /* copy nuclide data to device */
5    }
6  }
7  void OpenMPNuclide::deleteNucOnDevices(...){
8    for(int idev=0; idev<nb_devices; ++idev){
9      omp_target_free(..., idev);                                   /* remove nuclide data from device */
10   }
11 }
12 void OpenMPNeutronMediaNavigator::computeSigma(...){
13   #pragma omp target data map(tofrom: ...) map(to: ...) device(...) if(target data: ...)
14   {
15     #pragma omp target teams distribute num_teams(N) thread_limit(32) device(...) if(target: ...)
16     for(int t=0; t<N; ++t)
17       sigma1DopplerBroadening(t, ...);
18   }
19   total_sigma = 0; ...                                            /* macroscopic cross section calculation */
20 }
21 inline double compute_integral(...){
22   double sigma = 0; double sigma_tmp = 0;
23   int index = lb; sigma += ...;
24   #pragma omp parallel for reduction(+: sigma_tmp)
25   for(int i=lb+1; i<ub; ++i)
26     sigma_tmp += ...;
27   sigma += sigma_tmp;
28   index = ub; sigma -= ...;
29   return sigma;
30 }
```

### 4.2.6 Kokkos Implementation

**Algorithm 13:** Pseudo-code of Kokkos implementation.

```
1  void KokkosNuclide::allocateNucOnDevices(...){
2    Kokkos::View<double *, ...> d_eg_view(...);          /* allocate a view inside a device memory space */
3    Kokkos::View<double *, ...>::HostMirror h_eg_view = Kokkos::create_mirror_view(d_eg_view);
4    for(int i=0; i<size; ++i){
5      h_eg_view(i) = h_eg[i];                                                /* fill up the mirror view */
6    }
7    Kokkos::deep_copy(d_eg_view, h_eg_view);       /* deep copy from mirror view to its original view */
8  }
9  void KokkosNeutronMedia::deleteNucDataOnDevices(...){
10   for(int i=0; i<N; ++i)
11     this->kokkos_nuclides[i].reset(nullptr);     /* explicitly delete objects of the class KokkosNuclide */
12 }
13 void KokkosNeutronMediaNavigator::computeSigma(...){
14   Kokkos::TeamPolicy<> policy(N, Kokkos::AUTO());                        /* set team policy of kernel */
15   Kokkos::deep_copy(...);
16   Kokkos::parallel_for(policy, functor);   /* functor is a C++ class defining sigma1DopplerBroadening() */
17   Kokkos::deep_copy(...);
18   total_sigma = 0; ...                                     /* macroscopic cross section calculation */
19 }
20 KOKKOS_INLINE_FUNCTION double compute_integral(..., Kokkos::TeamPolicy<>::member_type& vector){
21   double sigma = 0; double sigma_tmp = 0;
22   int index = lb; sigma += ...;
23   Kokkos::parallel_reduce(Kokkos::ThreadVectorRange(vector,ub-lb-1), KOKKOS_LAMBDA (const int& j,
     double& t_sigma){
24     t_sigma += ...;
25   }, sigma_tmp);
26   sigma += sigma_tmp;
27   index = ub; sigma -= ...;
28   return sigma;
29 }
```

Firstly, since Kokkos implementation does not support multiple devices, our Kokkos implementation is not able to be processed on a heterogeneous architecture of a CPU associated to multiple GPUs.

In our Kokkos implementation, the data management between host and device is operated by the use of the

class `Kokkos::View` and the Kokkos runtime function $Kokkos::deep_copy()$. A Kokkos view is an array of zero or more dimensions specified by a type of entries, an execution space, a memory space as well as a memory layout. In our scenario, we utilize views of a single dimension allocated in the default execution space such as `Serial`, `Cuda` and manually set different memory spaces including `Kokkos::CudaSpace` and `Kokkos::HostSpace` with their default memory layouts such as `Kokkos::LayoutLeft`. The Kokkos function $deep_copy()$ is used to manage data placement between views in different memory spaces. In order to avoid the limitations that only those views with an identical memory layout can be performed by $deep_copy()$ correctly, the `Kokkos::View::HostMirror` is used as a type of view mapped to the host memory space with a compatible memory layout comparing to its original type `Kokkos::View`.

It should also be noted that the execution space `OpenMP` is not taken into account since our programming model explicitly makes use of OpenMP thread directives to accomplish task-level parallelism and Kokkos is only responsible for serial execution inside a host memory space. Moreover, like OpenACC and OpenMP offload, we use the $data()$ method defined in Kokkos View class to obtain the device pointer of a given view conserving nuclide data which is put in other views whose entries are of a C++ struct and these views will be used in the function $computeSigma()$. As for the function $deleteNucOnDevices()$ of parent class `Nuclide`, it is not explicitly implemented in the child class `KokkosNuclide` due to the fact that all views produced in our Kokkos implementation are stored in either a C++ struct or a C++ function and Kokkos provides a reference-counting mechanism to automatically manage deallocation of Views as once the reference count of a view reaches zero, this view will be deallocated from its memory space (constructor and copy increase the reference count, destructor and assignment decrease the reference count). Alternatively, to ensure that before the call of $Kokkos::finalize()$, reference counts of all views are returned to zero, we implement the function $deleteNucDataOnDevice()$ of the class `KokkosNeutronMedia` to forcefully call the destructor of the class `KokkosNuclide` in which views of nuclide data are conserved.

The computational kernel is launched by calling the function `Kokkos::parallel_for()` parameterized by a Kokkos execution policy setting its league size and team size along with a user-defined functor encapsulating the $sigma1DopplerBroadening()$ function. In our case, the `Kokkos::TeamPolicy` is set to `N` leagues (handling cross section calculations of `N` nuclides in parallel) and `Kokkos::AUTO` threads per team (`Cuda` execution space equaling to a multiple of 32 and `Serial` equaling to the corresponding vector length of a given host processor). Moreover, Kokkos adopts a mechanism to verify if a host thread generated by Kokkos already exists in a thread pool, which may sometimes cause runtime error with the utilization of our programming model `OpenMP thread + Kokkos::Serial`. For the purpose of avoiding this bug, we manually remove the verification part implemented in the Kokkos source file `Kokkos_HostThreadTeam.cpp`.

The $compute_integral()$ function is implemented in the functor as a `KOKKOS_INLINE_FUNCTION` where the inner loop is parallelized by the `Kokkos::parallel_reduce()` function which accepts a parameter of an instance of `Kokkos::ThreadVectorRange()` defining the loop range and a parameter of `KOKKOS_LAMBDA` function encapsulating the calculation part as illustrated in Algorithm 13 (at lines 23–25). Although Kokkos provides an interface to pro-

gramme “scratched pad” memory shared by threads in a team, we don’t use explicitly this functionality in order to make our Kokkos version in accordance with other implementations.

It should be noted that Kokkos provides indeed the interoperability of CUDA runtime API functions. By instantiating an object of the class `Kokkos::Cuda` with its constructor fed with a parameter of the class `cudaStream_t` which is initialized by the CUDA runtime function $cudaStreamCreate()$, the kernels and memory operations invoked by different host threads can be overlapped as an achievement of CUDA asynchronous multistreaming in the latest version of Kokkos. Nevertheless, such technique has not been implemented in our basic Kokkos version since for all versions implemented via high-level programming languages or libraries, we intend to hide low-level APIs completely and offer users a pure high-level interface to programme.

### 4.2.7 SYCL Implementation

As shown in Algorithm 14 (at lines 1–8), the memory copies of nuclide data are invoked as memory copy kernels in the function $allocateNucOnDevices()$ of the class `SYCLNuclide`. All nuclide data are stored in the `cl::sycl::buffer` class of one dimension as an input parameter of a command group which is submitted to a queue by invoking its member function $submit()$. The access of SYCL buffer is achieved by the class `cl::sycl::accessor` which provides an entry to data either on host or within a command group enqueued to SYCL queues. The constructor of SYCL queues adopted in our implementation accepts an instance of the class `cl::sycl::default_selector` which may select a host device or accelerators. The memory copy operation is undertaken by the member function $copy()$ of the `cl::sycl::handler` class which copies the host data into the memory object accessed by SYCL accessors.

The parallelization of outer-loop in the function $computeSigma()$ is accomplished by the invocation of an hierarchical kernel which calls the function $parallel_for_work_group()$ of the class `cl::sycl::handler` with two values of the class `cl::sycl::range` representing respectively the number of work-groups and the number of work-items per work-group as well as a user-defined functor encapsulating the function $sigma1DopplerBroadening()$ in its `operator()` function. In our case, the number of work-groups is set to `N` where each work-group is assigned to a task of microscopic cross section calculation of a nuclide. The number of work-items per work-group denoted as `work_group_size` is hardware-dependent which needs developers to manually set its value. Besides, since by default all variables declared inside the `parallel_for_work_group` scope are allocated in local memory which are shared by all work-items within the group, we declare these variables of the type `cl::sycl::private_memory<T>` as a substitution of the type `T` for the purpose of explicitly declaring them in private memory and avoiding the use of shared memory. In this way, the SYCL version can be in accordance with other implementations.

After the execution of the computational kernel, a member function $update_host()$ of SYCL handler for memory operation is enqueued to the SYCL queue to copy the calculated data back to host.

Note that each call of the function $submit()$ invokes a single kernel or memory operation by convention. It should be avoided to put multiple kernels and memory operations in the same scope of a function $submit()$.

**Algorithm 14:** Pseudo-code of SYCL implementation.

```
1  void SYCLNuclide::allocateNucOnDevices(...){
2    cl::sycl::queue sq(cl::sycl::default_selector{});  /* create a SYCL queue of a particular device type */
3    cl::sycl::buffer<...> buffer_eg(h_eg, ...);                    /* wrap the host data in a SYCL buffer */
4    sq.submit([&] (cl::sycl::handler& cgh){                                                /* Enqueue a kernel */
5      auto d_eg_accessor = buffer_eg.get_access<...>(cgh);              /* request access to the buffer */
6      cgh.copy(h_eg, d_eg_accessor);       /* copy h_eg to the memory object accessed by d_eg_accessor */
7    });
8  }
9  void SYCLNeutronMediaNavigator::computeSigma(...){
10   this->sq.submit([&] (cl::sycl::handler& cgh){
11     cgh.parallel_for_work_group(N, work_group_size, functor(...));
12   });
13   this->sq.submit([&] (cl::sycl::handler& cgh){
14     cgh.update_host(...);
15   });
16   total_sigma = 0; ...                                  /* macroscopic cross section calculation */
17 }
18 cl::sycl::private_memory<double> compute_integral(..., cl::sycl::group<1>& work_items){
19   size_t num_items = work_items.get_local_range(0); double[num_items] item_sigmas = {0};
20   cl::sycl::private_memory<double> sigma(work_items);
21   cl::sycl::private_memory<int> index(work_items);
22   work_items.parallel_for_work_item([&](cl::sycl::h_item<1> item){
23     sigma(item) = 0; sigma_tmp = 0;
24     index(item) = lb(item); sigma(item) += ...;
25     for(int i=lb(item)+1; i<ub(item); i+=item.get_get_local_range(0))
26       sigma_tmp += ...;
27     item_sigmas[item.get_local_id(0)] = sigma_tmp;
28     for(int mask=item.get_local_range(0); mask>0; mask/=2){
29       if(item.get_local_id(0)<mask)
30         item_sigmas[item.get_local_id(0)] += item_sigmas[thread.get_local_id(0) + mask];
31     }
32     sigma(item) += item_sigmas[0];
33     index(item) = ub(item); sigma(item) -= ...;
34   });
35   return sigma;
36 }
```

Since it is possible to call the function $parallel_work_for_item()$ of the class `cl::sycl::group` multiple times within the `parallel_work_for_group` scope, the inner-loop in the $compute_integral()$ function can be vectorized by this function (at lines 25–26). However, due to the fact that SYCL of version 1.2.1 does not provide a pre-defined function performing the sum reduction, we are obligated to manually implement a reduction loop to summarize the final result with the utilization of shared memory (at lines 28–31). Although the latest version of SYCL released in 2020 offers the function $parallel_reduce()$ of the class `cl::sycl::handler` along with the `cl::sycl::reduction` interface and `cl::sycl::reducer` class to perform reduction over all work items, this function has not been expanded to the class `cl::sycl::group` so as to be invoked in the `parallel_work_for_group` scope [69]. Therefore, it is not suitable to our coding strategy of hierarchical parallelism and we hope such functionality can be implemented in the future release.

Additionally, SYCL allows the host to target to multiple devices. However, unlike CUDA, OpenACC and OpenMP offload which offer users the interface to explicitly associate a particular device with a host thread, the function $select_device()$ of the class `device_selector` set up the device based on the highest score gained by all available SYCL devices and it will randomly select a device if more than one device obtains the same high score. As for the kernel execution order in our SYCL implementation, since all command groups are submitted to different queues where each queue is owned by a particular host thread, the order of execution is determined by the SYCL runtime. Moreover, because all command group objects use almost the same nuclide data as the contents of requirement sets, they depend on each other and cannot be overlapped as concurrent kernels.

## 4.3 Implementations of Particle Tracking Methods

### 4.3.1 Heterogeneous Offloading History-based Method

**Algorithm 15:** Heterogeneous offloading history-based Monte Carlo neutron transport algorithm.

```
1  foreach particle distributed to the processing thread do
2      while particle is alive do
3          calculation of macroscopic cross section, with several vectorizable parts;
4              • do microscopic cross section lookups ⟹ offloaded;
5                  ∗ binary search to find lower-upper bounds;
6                  ∗ integral computation;
7              • sum up macroscopic cross section;
8          sample distance, move particle, do interaction;
9      end
10 end
```

Algorithm 15 shows the procedure of history-based method implemented in PATMOS adopting our heterogeneous offloading strategy. Microscopic cross section lookup using the SIGMA1 on-the-fly Doppler broadening algorithm is the only part offloaded to accelerators. Once the required data for cross section lookups are transferred from host to device, calculations on device begin and the host summarizes macroscopic total cross section after the results being transferred back. It is obvious that our design brings in too many back-and-forth data transfers between host and device. The size of data movement for each calculation (a group of nuclides in one material) is quite small but the large number of memcpy calls induces many launch overheads which may degrade performance in an overwhelming way.

### 4.3.2 Heterogeneous Offloading Pseudo Event-based Method

As one of general guidelines for mitigating the performance bottleneck due to host-device data transfers is to batch many small transfers into one large transfer. Thus, we developed another tracking algorithm in which one host thread treats several particles at the same time so as to request the GPU to compute the microscopic cross sections for all of these particles at the same time, in one bigger kernel. This is achieved by banking multiple particles into one group and offloading microscopic cross section lookups for all these particles. In this way, the number of data transfers can be reduced and the amount of work for each kernel is increased.

**Algorithm 16:** Heterogeneous offloading pseudo event-based Monte Carlo neutron transport algorithm.

```
1  foreach bank of particles distributed to the processing thread do
2      while particles remain in bank do
3          foreach remaining particle in bank do                    /* First event */
4              bank required data for microscopic cross section lookups;
5          end
6          • do microscopic cross section lookups ⟹ offloaded;     /* Second event */
7            ∗ binary search to find lower-upper bounds;
8            ∗ integral computation;
9          foreach remaining particle in bank do                    /* Third event */
10             • sum up macroscopic cross section;
11             sample distance, move particle, do interaction;
12         end
13     end
14 end
```

To fulfill this tuning strategy, our history-based method is redesigned to a new “pseudo event-based” approach. The details of this new method are described in Algorithm 16. The overall procedure of history tracking is reorganized into three events including a `for` loop of data banking, a kernel of microscopic cross section calculations and another `for` loop of macroscopic cross section calculations along with the distance sampling, particle movement and interaction.

## 4.4 Numerical Results

In this section, the numerical results of a benchmark implemented in PATMOS via different programming languages and libraries as well as two particle tracking approaches (history-based and pseudo event-based methods) executed across a set of architectures will be elaborated.

### 4.4.1 Description of the benchmark

The benchmark, called slabAllNuclides, consists in a fixed source Monte Carlo simulation of a slab discretized in 10,000 volumes. Each volume is made with the same material containing 388 nuclides, which corresponds to the number of nuclides available in the ENDFB-VII.0 nuclear data library. The number of nuclides in the material has a big impact on the macroscopic cross section calculation time. That is why we chose a material composition representative of what is encountered in depletion calculation. Moreover, $^{1}$H and $^{238}$U are predominant so as to obtain a neutronic behavior representative of a pressurized water reactor. The temperature of the material is 900 Kelvin. We simulate 10 batches of 500,000 particles.

In the following, all the tests are performed with the SIGMA1 on-the-fly Doppler broadening algorithm. The base temperature from which the cross section are Doppler broadened is 0 Kelvin. Performances of the different implementations (programming models and particle tracking method) are evaluated by comparing the simulation time (the run time per cycle) or the particle tracking rate (the number of particles treated per second).

### 4.4.2 Compiler Support

Since the programming languages or libraries implemented in the research (OpenMP thread, CUDA, OpenACC, OpenMP offload, Kokkos and SYCL) do not cover all architectures and each one has different degrees of maturity achieved by different compilers, it is of significance that we make a global summary of the compiler support for these programming models.

From Table 4.2, we find that OpenMP thread can be compiled by most of compilers except NVCC where NVCC is Nvidia’s CUDA Compiler based on LLVM compiler infrastructure which is dedicated to compiling CUDA code.

Concerning OpenACC, PGI starts to support OpenACC 1.0 specification from its 2012 release and the full features of directives-based parallel programming specified in OpenACC 2.6 as well as some features of OpenACC

3.0 specification such as deep copy directives have been implemented in recent releases. Besides, GCC provides implementation of OpenACC specification from GCC 5 and its latest release GCC 10 starts to support some features of OpenACC 2.6 specification.

With respect to OpenMP offload, GCC offers support for offload of OpenMP 4.5 and 5.0 specifications targeting to Intel MIC, Nvidia GPUs and AMD GCN architectures. IBM XLC 16.1 supports OpenMP 4.5 specification offloading computations to Nvidia GPUs with the requirements of little endian operating system and CUDA 9.2 backend. Clang fully supports OpenMP 4.5 specification and some features of OpenMP 5.0 specification with a set of offload targets including x86, OpenPower, Arm and Nvidia GPU architectures. Moreover, Intel C++ compiler next generation code generator provided in the Intel oneAPI HPC toolkit support OpenMP 4.5/5.0 `TARGET` offload feature for Intel GPUs (Gen9 through Gen11). It should also be noted that the OpenMP offload support targeting to host (multicore CPU) is provided by PGI 2019 or later releases, the development of GPU target offload is on the roadmap.

Table 4.2: Compiler support for different programming models.

| Compilers | Programming Models | | | | | |
|---|---|---|---|---|---|---|
| | OpenMP thread | CUDA | OpenACC | OpenMP offload | Kokkos | SYCL |
| GCC | ✓ | | ✓ | ✓ | | |
| PGI | ✓ | | ✓ | IN FUTURE | | |
| XLC | ✓ | | | ✓ | | |
| Clang | ✓ | | | ✓ | | |
| Intel C++ | ✓ | | | ✓ | | |
| NVCC | ✓ | ✓ | | | | |
| DPC++ | ✓ | | | | | ✓ |
| hipSYCL | ✓ | ✓ | | | | ✓ |
| ComputeCpp | ✓ | | | | | ✓ |
| Kokkos wrapper | ✓ | ✓ | | | ✓ | |

With reference to Kokkos, GCC, PGI, XLC, Clang, Intel C++, NVCC compilers can be wrapped into a Kokkos compiler for the compilation of Kokkos code on x86, OpenPower, Intel MIC, ARM architectures. The support of Kokkos for AMD and Intel GPU backend is on the roadmap and should be developed by the year of 2021.

As for SYCL, DPC++, hipSYCL and ComputeCpp provide support for compiling SYCL code with the offload target of Intel CPUs, Intel GPUs, Intel FPGAs, AMD GPUs, Nvidia GPUs and so on. DPC++ is a LLVM-based high-performance compiler for the Data Parallel C++ language of C++ and SYCL standards provided in the Intel oneAPI Base Toolkit [103]. It mainly offloads code across OpenCL platforms such as Intel CPUs, GPUs and FPGAs. A DPC++ backend implementation in collaboration with Codeplay enabling Nvidia GPU offloading support has been released recently [104]. This implementation should work on Nvidia GPUs of capability 5.0 or higher with CUDA

10.2 installed. hipSYCL [15] is a LLVM-based compiler building on CUDA, HIP and OpenMP backends instead of OpenCL which allows it to target multicore CPUs, Nvidia GPUs and AMD GPUs. It is wrapped with a clang plugin to provide support for SYCL code and thus interoperable with CUDA and HIP codebases. ComputeCpp is a LLVM-based device compiler capable of compiling SYCL implementations on OpenCL platforms which mainly include Intel CPUs, Intel GPUs, Intel FPGAs, AMD GPUs with SPIR(-V) support and Nvidia GPUs with experimental PTX support [113]. Except for SYCL implementations mentioned above, there are another two incomplete LLVM-based implementations, triSYCL [68] and sycl-gtx [120] which both build on OpenCL backend to provide offload support for OpenCL devices.

### 4.4.3 Execution Environment

All intra-node tests of slabAllNuclides benchmark expressed in this chapter are carried out on five machines which are listed in Table 4.3:

Table 4.3: List of machines to run tests of slabAllNuclides benchmark.

| Machines | Details | | | |
|---|---|---|---|---|
| GridCL | 2× 20-core Intel Xeon Gold 6138, | HT | + | 2× Nvidia V100 |
| Cobalt-hybrid | 2× 14-core Intel Xeon E5-2680 v4, | HT | + | 2× Nvidia P100 |
| Cobalt-v100 | 2× 20-core Intel Xeon Gold 6148, | HT | + | 4× Nvidia V100 |
| Gorgon | 2× 20-core IBM Power9, | SMT4 | + | 4× Nvidia V100 |
| Intel NUC | 1× 4-core Intel Core i7-8705G, | HT | + | 1× Intel HD Graphics 630 |

where HT refers to hyper-threading, SMT refers to simultaneous multi-threading.

In summary, all five machines listed above compose a set of architectures including x86 (Broadwell, Skylake, Kaby Lake G), OpenPower (Power9), Nvidia GPU (P100, V100) and Intel GPU (Intel HD Graphics 630). The first four are computing center machines while the last one is a desktop machine.

The basic configurations of these architectures are illustrated in Table 4.4. Note that the cores and threads of Nvidia GPUs listed in Table 4.4 signify respectively the streaming multiprocessors (SMs) and CUDA threads. As for the Intel Gen9.5 architecture, the number of cores refers to its number of execution units.

The compilers installed on GridCL include GCC 7.1, PGI 20.7, Intel C++ 17.0, Clang 11.0.0 and NVCC 9.2/10.1, which offer support for OpenMP thread, CUDA, OpenACC, OpenMP offload and Kokkos implementations. On Cobalt-hybrid and Cobalt-v100, GCC 7.1, PGI 18.7, Intel C++ 17.0, and NVCC 9.0 are provided to achieve the compilation of OpenMP thread, CUDA, OpenACC, Kokkos. Gorgon uses GCC 7.3, XLC 16.1, PGI 19.4 and CUDA 9.2/10.1 to compile OpenMP thread, CUDA, OpenACC, OpenMP offload and Kokkos. At last, Intel NUC is installed with GCC 7.4, PGI 19.10, ComputeCpp 1.1.6, DPC++ 2021.1-beta03 to provide support for OpenMP thread,

OpenACC, Kokkos as well as SYCL implementations.

Table 4.4: Configurations of processors composing accessible machines.

| Processors | Cores | Threads | Frequency GHz | Memory Bandwidth GB/s | FP64 TFLOPS |
|---|---|---|---|---|---|
| | | | Skylake | | |
| Intel Xeon Gold 6138 | 40 | 80 | 2.00 | 207.4 | 2.6 |
| Intel Xeon Gold 6148 | 40 | 80 | 2.40 | 207.4 | 3.1 |
| | | | Broadwell | | |
| Intel Xeon E5-2680 v4 | 28 | 56 | 2.40 | 76.8 | 1.1 |
| | | | Kaby Lake G | | |
| Intel Core i7-8705G | 4 | 8 | 3.10 | 37.5 | 0.2 |
| | | | Power9 | | |
| Power9 SMT4 | 40 | 160 | 3.20 | 340 | 1.0 |
| | | | Nvidia Pascal | | |
| P100 | 56 | 1792 | 1.33 | 720 | 5.3 |
| | | | Nvidia Volta | | |
| V100 | 80 | 2560 | 1.30 | 900 | 7.8 |
| | | | Intel Gen9.5 | | |
| Intel HD Graphics 630 | 24 | 168 | 0.35 | 37.5 | 0.03 |

### 4.4.4 Performance Analysis in Host Mode

Due to the fact that our implementations in PATMOS adopts a heterogeneous offloading strategy, distributing particle tracking tasks to a group of CPU threads where the microscopic cross section calculations required in all tasks may be offloaded to devices or performed on host, the performance analysis should be classified into two modes: host and offload. We start from the study of the effects of thread binding on performance in host mode.

#### 4.4.4.1 Thread Affinity Analysis

The test is carried out on the machine GridCL where the SYCL version is not able to be executed successfully. The benchmark is compiled by Intel C++ 17.0 with the auto-vectorization functionality enabled. The configuration of GridCL can be checked from Table 4.3 and Table 4.4.

OpenMP mainly provides three levels of thread affinity, `sockets`, `cores` and `threads` so as to bind a thread to a hardware socket, core or thread. In order to deliver better performance for a given number of threads, it is a priority to bind these threads to as much as hardware cores rather than hardware sockets or threads since the latter two choices may end up with many cores bound by more than one thread. Therefore, we prefer to set the thread affinity level to `cores` by using the command `export OMP_PLACES=cores` instead of `sockets` and `threads`. As for the thread affinity policy, we consider two cases, `close` and `spread` which bind the threads respectively by a close and sparse

distribution across the available places.

From Table 4.5, we find that the `spread` policy outperforms the `close` policy. The performance gap becomes greater when the number of threads is less than the total number of cores on the machine (In this case, GridCL contains two sockets of 20 cores). The reason for this performance gap is that binding all threads to the cores in a single socket as what the `close` policy does gives each thread limited bandwidth. By contrast, binding all threads over two sockets gives each thread a higher available bandwidth and mitigates the memory-bound issue.

Table 4.5: Performance of the OpenMP thread version of slabAllNuclides using the history-based method via different thread affinity policy.

| Number of threads | cores/close | cores/spread |
|---|---|---|
| 10 | 1269.89 | 1113.39 |
| 20 | 732.26 | 639.34 |
| 30 | 472.89 | 462.95 |
| 40 | 374.05 | 373.86 |
| 50 | 352.12 | 351.85 |
| 60 | 331.73 | 329.19 |
| 70 | 310.20 | 307.66 |
| 80 | 294.50 | 291.38 |

The performance is evaluated by the simulation time (s/cycle), the lower the better.

Note that the performance of slabAllNuclides using the pseudo event-based method is similar to that using the history-based method in host mode. The `spread` policy is also better than the `close` policy. Thus, we set the thread affinity in host mode to `cores/spread` for all tests undergone in the following.

#### 4.4.4.2 Scaling Analysis

Now, we carry out a series of tests for scaling analysis (strong scaling) on the machine GridCL. For host mode, the scaling performance is mainly influenced by overheads to avoid memory contention when shared resources such as nuclide data are accessed by multiple threads simultaneously.

Figure 4.4 depicts the run time per cycle (the lower the better) and strong scalability (the higher the better) of slabAllNuclides benchmarks using the history-based method (denoted as HB). In terms of simulation time, all curves are in a form of $\frac{1}{x}$ as the run time per cycle decreases with the increase of threads. For example, the OpenMP thread version gains a factor of approximate 28x performance speedup with 40 threads comparing to that with a single thread. In terms of strong scalability, all versions resemble in each other. With the increase of threads from 10 threads to 20 threads and finally to 40 threads, their scalability drops from around 90% to 80%, and then to 70%. It can be concluded that we obtain 10% drop of strong scalability as the number of threads doubles.

Furthermore, when the hyper-threading is enabled, the scalability drops from 70% to around 45% as the number of threads varies from 40 to 80, which corresponds to the normal behavior of hyper-threading.

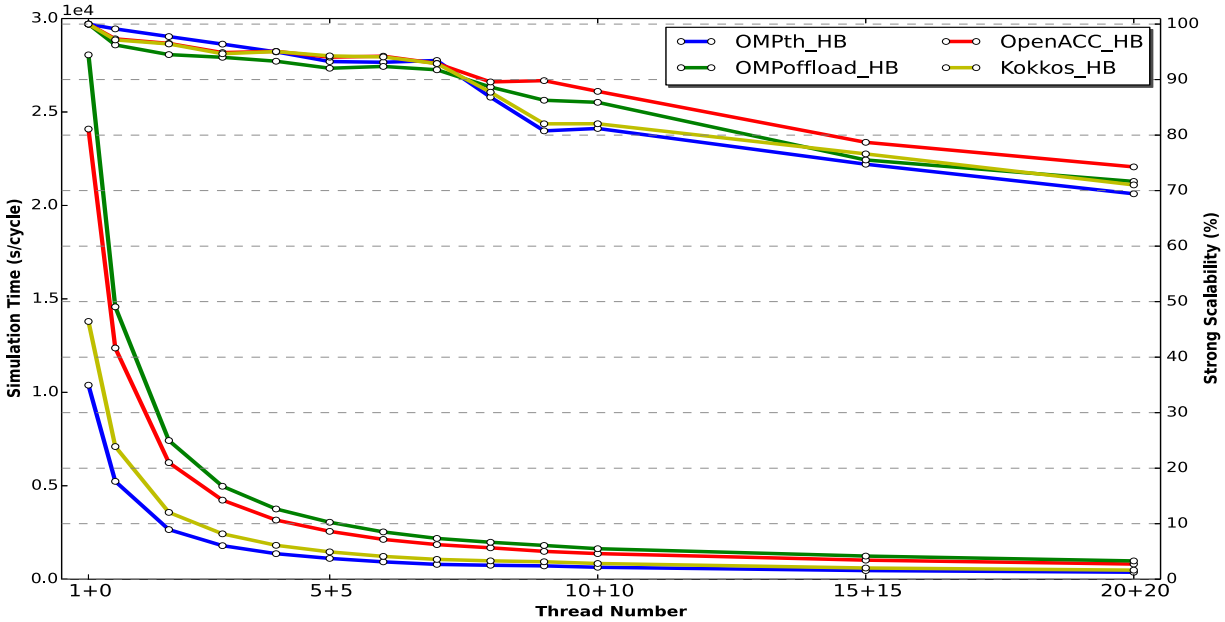



Figure 4.4: Scaling performance of the slabAllNuclides using the history-based method in host mode (Simulation time, the lower the better; Strong scalability, the higher the better).

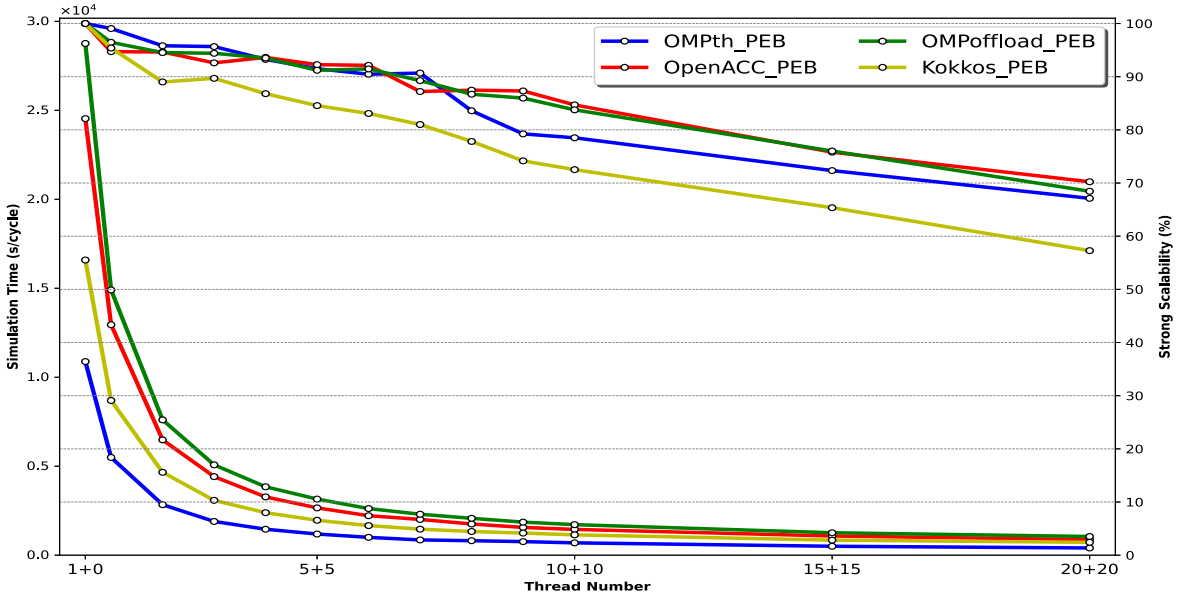



Figure 4.5: Scaling performance of the slabAllNuclides using the pseudo event-based method in host mode (Simulation time, the lower the better; Strong scalability, the higher the better).

Similarly, Figure 4.5 shows the simulation time and strong scalability of slabAllNuclides benchmarks using the pseudo event-based method (denoted as PEB). Note that the bank size of pseudo event-based method is set to 100 as there is no need to perform a bank size analysis in host mode. From the perspective of performance, the simulation time of different implementations of slabAllNuclides using the pseudo event-based method is generally a little higher than the one using the history-based method. This is caused by the data banking and additional loop overheads of the pseudo event-based method. From the perspective of strong scalability, the scalability of all versions decreases in the similar way of the history-based method.

#### 4.4.4.3 Vectorization

The performance gaps between different programming models are caused by different compilers. For instance, by using the options `-Wa,-adhlng` and `-fopt-info` with GCC compiler to generate assembly code along with the source code and show optimization information.

**OpenMP directive enabled**

```
LOOP BEGIN at sigma1xse.cxx(88,3)
  remark #15305:  vectorization support:  vector length 2
  remark #15301:  OpenMP SIMD LOOP WAS VECTORIZED
  remark #15475:  -- begin vector cost summary --
  remark #15476:  scalar cost:  450
  remark #15477:  vector cost:  195.000
  remark #15478:  estimated potential speedup:  2.280
  remark #15488:  -- end vector cost summary --
LOOP END
```

**Auto-vectorization enabled**

```
LOOP BEGIN at sigma1xse.cxx(88,3)
  remark #15305:  vectorization support:  vector length 4
  remark #15475:  -- begin vector cost summary --
  remark #15476:  scalar cost:  419
  remark #15477:  vector cost:  91.750
  remark #15478:  estimated potential speedup:  4.320
  remark #15488:  -- end vector cost summary --
LOOP END
```

We find that the `for` loop in the function $compute_integral()$ is vectorized with a vector length 2 for holding

two 64-bit double-precision floating point variables simultaneously as only the SSE (XMM0-XMM15) registers are used in the instructions such as `vmulsd %xmm4, %xmm7, %xmm0`. In summary, the OpenMP SIMD directive and GCC compiler options both fail to make the GCC compiler fully exploiting the hardware feature (Skylake processor with AVX512 extensions).

The Intel C++ compiler manages to parallelize the `for` loop in the function $compute_integral()$ with a 256-bit vectorization (vector length 4 for holding four variables of `double` type simultaneously) achieved by the use of AVX2 (YMM0-YMM15) registers, as shown in the optimization report listed above which is enabled by adding the `-qopt-report=5` into compiler options. It indicates that the OpenMP SIMD directive also has no effect on Intel C++ compiler for the utilization of AVX2 extensions on GridCL as it acts for the GCC compiler. In order to accomplish a higher level of vectorization, the compiler options for auto-vectorization must be added. However, all compilers, including GCC, Intel C++, as well as PGI and LLVM Clang, fail to vectorize the inner-loop with AVX512 extensions on GridCL.

### 4.4.5 Performance Analysis in Offload Mode

#### 4.4.5.1 Thread Affinity Analysis

Due to the GPU topology, the CPU affinity may affect the performance. By using the command `nvidia-smi topo -m`, we find that on GridCL, a single GPU is connected closely to a socket including 20 cores, 40 threads via PCIe. A comparison of the performances of slabAllNuclides adopting different thread affinity policies in offload mode tested on GridCL shows that for slabAllNuclides using the pseudo event-based method PEB, both the OpenMP `cores/spread` and `cores/close` policies introduces few performance gap between each other, as illustrated in Table 4.6. The maximal ratio of the performance gap to the greater performance for a given number of threads is 4.57% achieved by 10 threads while the average percentage of performance gap equals to 1.56%.

On the contrary, the performances of slabAllNuclides HB adopting different thread affinity policies differentiate quite a lot with each other. From Table 4.6 we find that the maximal percentage of performance gap reaches up to 24.06% and the average ratio of performance gap to the higher performance is 12.66%. It indicates that comparing to pseudo event-based method, history-based method is much more influenced by the choice of thread affinity policy related to GPU topology. Note that the tests listed in Table 4.6 are performed with a single device by setting the environment variable `CUDA_VISIBLE_DEVICES` to a single number, 0 or 1. We can conclude that the `cores/close` policy works better than the `cores/spread` policy for slabAllNuclides HB in offload mode targeting to a single device.

As for multi-threads targeting to multi-GPUs, the effect of CPU affinity is similar to that of a single device, as described in Table 4.7. With respect to history-based method, the `cores/close` policy is more suitable than the `cores/spread` policy, improving around 16.33% performance in average. With respect to pseudo event-based method, the `cores/spread` policy outperforms a little the `cores/close` with a maximal 8.94% performance improvement and an average percentage of performance gap equaling to 2.50%.

Overall, in what follows we set the thread affinity to `cores/spread` for all tests carried out in offload mode to accord with the thread affinity set for tests executed in host mode, as we discussed in the section 4.4.4.

Table 4.6: Comparison of the performances of slabAllNuclides in offload mode adopting different thread affinity policies tested on GridCL with a single Nvidia V100.

| Number of threads | cores/close | | cores/spread | |
|---|---|---|---|---|
| | HB | PEB | HB | PEB |
| 2 | 1168.85 | 393.20 | 1278.35 | 399.46 |
| 4 | 725.82 | 215.61 | 955.73 | 207.73 |
| 6 | 1383.32 | 153.42 | 1794.62 | 149.67 |
| 8 | 1731.11 | 124.94 | 2004.87 | 122.95 |
| 10 | 1845.44 | 112.84 | 1592.36 | 107.69 |
| 20 | 1905.26 | 96.90 | 1868.24 | 96.87 |
| 40 | 2475.58 | 96.69 | 2556.41 | 96.83 |
| 60 | 3047.89 | 97.16 | 4006.39 | 96.96 |
| 80 | 3980.08 | 141.11 | 4062.55 | 143.19 |

where HB and PEB are respectively the abbreviations of history-based and pseudo event-based methods. The performance is evaluated by the simulation time (s/cycle), the lower the better.

Table 4.7: Comparison of the performances of slabAllNuclides in offload mode adopting different thread affinity policies tested on GridCL with two Nvidia V100.

| Number of threads | cores/close | | cores/spread | |
|---|---|---|---|---|
| | HB | PEB | HB | PEB |
| 2 | 1087.49 | 388.09 | 1071.43 | 377.84 |
| 4 | 587.34 | 202.77 | 632.48 | 195.39 |
| 6 | 442.70 | 141.48 | 483.95 | 137.02 |
| 8 | 397.23 | 110.35 | 864.31 | 107.81 |
| 10 | 594.39 | 99.27 | 969.56 | 90.39 |
| 20 | 621.76 | 62.15 | 933.36 | 57.77 |
| 40 | 927.63 | 49.20 | 943.89 | 49.15 |
| 60 | 1669.81 | 49.39 | 1699.01 | 49.85 |
| 80 | 1971.01 | 49.83 | 1966.18 | 49.99 |

where HB and PEB are respectively the abbreviations of history-based and pseudo event-based methods. The performance is evaluated by the simulation time (s/cycle), the lower the better.

#### 4.4.5.2 Bank Size Analysis

Moreover, when it comes to the pseudo event-based method, the bank size is also a key factor that influences the performance as well as the optimal thread number. In what follows we perform a bank size analysis for the CUDA

version of slabAllNuclides on GridCL. If the bank size is set to 1, the program will run in a history-based way. If it is superior to 1, a pseudo event-based way. The tests are carried out with certain thread numbers and a single Nvidia V100 card, as depicted in Figure 4.6.

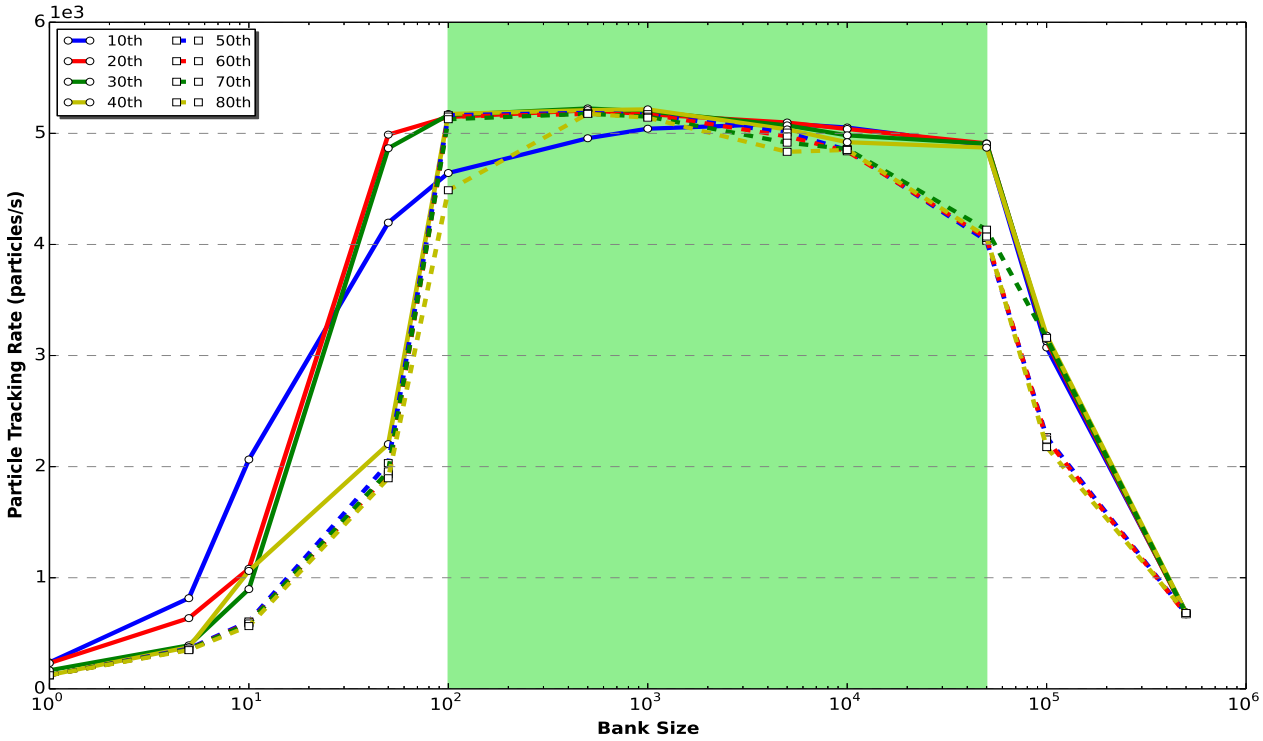



Figure 4.6: Bank size analysis of the CUDA version of slabAllNuclides in offload mode, the higher the better.

We find that too small and too large bank size cannot maximize performance. For the region of bank size ranging from 1 to 100, with the increase of bank size, the performance of benchmark improves significantly, gaining a factor of approximately 20x speedup. When the bank size surpasses 100, most of performances executed by different thread numbers varies little and becomes stable, until the bank size exceeds a rather large number (in our case, $5 \times 10^4$), as the green region shown in Figure 4.6. After this green region, the performance of benchmark drops down which is due to the CPU idle threads caused by the lack of tasks to be distributed to all available threads (number of banks is smaller than number of threads). For example, when the bank size equals to $5 \times 10^5$, there is only a single bank to be assigned to a CPU thread and all performances executed by multiple threads tend to the same value.

It is obvious that choosing a bank size residing in the green region is a better choice to exploit the optimal performance of our CUDA implementation in offload mode. Thus for the following tests carried out via the pseudo event-based method, we intend to fix all their bank sizes to 100.

#### 4.4.5.3 Scaling Analysis

Concerning the offload mode, the overheads of kernel launch and memory transfer become a major point to affect scaling performance. Unlike host mode where its performance improves with the increase of threads continuously, the performance of offload mode is more like having a peak value at a given number of threads. If we use more threads, the benefits obtained from multithreading will be uncompetitive to the overheads of kernel launch and memory transfer, leading to a significant performance degradation.

In particular, for Nvidia GPUs with multi-stream support, too many host threads where each one targets to a CUDA stream make some overheads and latency of kernels and memory transfers unable to be hidden due to the total multi-stream number limited by compute resources. Therefore, it is significant to perform a scalability analysis that concentrates on finding the optimal thread number delivering the best performance.

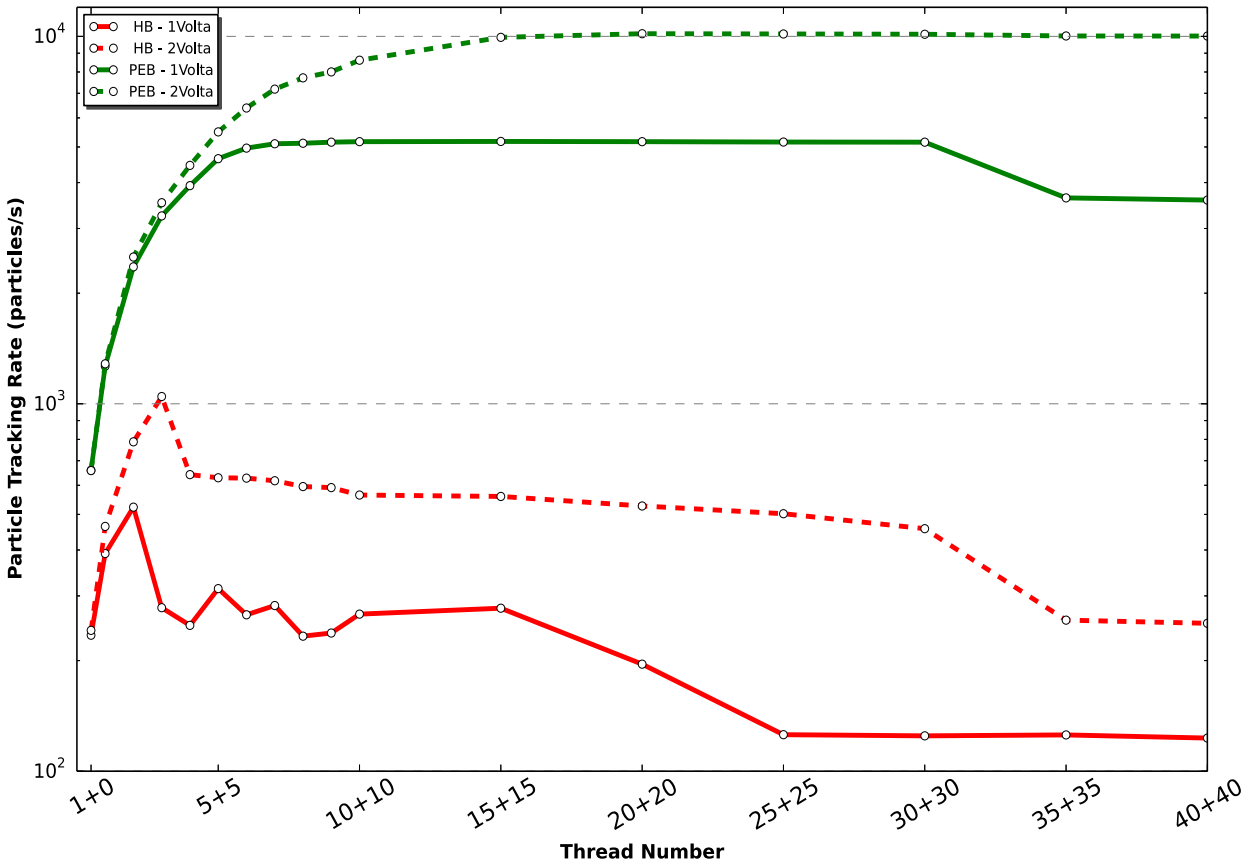



Figure 4.7: Scaling performance of the CUDA version of slabAllNuclides in offload mode, the higher the better.

The tests are carried out with the CUDA version of slabAllNuclides on the machine GridCL in offload mode. Both the history-based method and the pseudo event-based method are taken into account. The bank size of the pseudo event-based method is set to 100 which has already been proved as a good choice to exploit the best performance.

Figure 4.7 shows that the pseudo event-based method generally exceeds a magnitude of performance achieved by the history-based method. With reference to the pseudo event-based method, when the number of threads surpasses a value (15 for a single GPU, 30 for two GPUs), its performance reaches up to a high value and becomes quite stable. With reference to the history-based method, there is a peak value obtained by a specific number of threads (4 for a single GPU, 6 for two GPUs). When the number of threads increases over the peak number, its performance degrades little by little. This scalability analysis indicates that the history-based method is a bad candidate for our heterogeneous offloading strategy while the pseudo event-based method exploits more computational capability of devices and delivers much more improving performance.

### 4.4.6 Performance Evaluation

A series of tests have been carried out on the platform set introduced in section 4.4.3 for the purpose of evaluating the performance of the different programming models. Numerical results of peak performance are illustrated in Table 4.8.

Table 4.8: Particle tracking rate via different programming models on a set of platforms, the higher the better.

| Machine | | Programming Model | slabAllNuclides ($\times 10^2$ particles/s) | |
|---|---|---|---|---|
| | | | HB | PEB |
| GridCL | CPU (40 cores, HT2) | OMPth | 17.2 | 16.9 |
| | | OMPth+ACC | 7.7 | 6.9 |
| | | OMPth+offload | 6.5 | 5.5 |
| | | OMPth+Kokkos | 12.8 | 8.1 |
| | 1V100 | OMPth+CUDA | 5.2 | 51.8 |
| | | OMPth+ACC | 3.5 | 34.7 |
| | | OMPth+offload | 0.4 | 2.6 |
| | | OMPth+Kokkos | 0.4 | 0.7 |
| | 2V100 | OMPth+CUDA | 10.5 | 101.7 |
| | | OMPth+ACC | 7.0 | 59.8 |
| | | OMPth+offload | 0.6 | 4.5 |
| Cobalt-hybrid | CPU (28 cores, HT2) | OMPth | 10.2 | 9.5 |
| | | OMPth+ACC | 4.5 | 4.4 |
| | | OMPth+Kokkos | 7.1 | 5.9 |

| | | | | |
|---|---|---|---|---|
| | 1P100 | OMPth+CUDA | 4.7 | 25.7 |
| | | OMPth+ACC | 3.3 | 21.8 |
| | | OMPth+Kokkos | 0.3 | 2.4 |
| | 2P100 | OMPth+CUDA | 10.7 | 51.0 |
| | | OMPth+ACC | 6.2 | 42.8 |
| Cobalt-v100 | CPU (40 cores, HT2) | OMPth | 14.6 | 13.6 |
| | | OMPth+ACC | 6.6 | 6.4 |
| | | OMPth+Kokkos | 7.9 | 7.2 |
| | 1V100 | OMPth+CUDA | 4.2 | 55.7 |
| | | OMPth+ACC | 2.5 | 21.8 |
| | | OMPth+Kokkos | 0.3 | 2.8 |
| | 2V100 | OMPth+CUDA | 11.0 | 84.7 |
| | | OMPth+ACC | 5.2 | 42.0 |
| | 4V100 | OMPth+CUDA | 21.7 | 140.4 |
| | | OMPth+ACC | 10.5 | 70.4 |
| Gorgon | CPU (40 cores, SMT4) | OMPth | 7.0 | 6.8 |
| | | OMPth+ACC | 7.7 | 7.4 |
| | | OMPth+offload | 1.7 | 1.5 |
| | | OMPth+Kokkos | 6.7 | 6.0 |
| | 1V100 | OMPth+CUDA | 4.4 | 55.0 |
| | | OMPth+ACC | 2.4 | 36.6 |
| | | OMPth+offload | 2.5 | 6.5 |
| | | OMPth+Kokkos | 0.31 | 1.2 |
| | 2V100 | OMPth+CUDA | 8.2 | 104.4 |
| | | OMPth+ACC | 4.8 | 62.9 |
| | | OMPth+offload | 4.9 | 12.8 |
| | 4V100 | OMPth+CUDA | 15.4 | 147.0 |
| | | OMPth+ACC | 8.2 | 80.6 |
| | | OMPth+offload | 8.9 | 25.0 |
| Intel NUC | CPU (4 cores, HT) | OMPth | 3.2 | 3.1 |
| | | OMPth+ACC | 1.2 | 1.1 |

| | | | |
|---|---|---|---|
| | OMPth+Kokkos | 2.3 | 2.2 |
| | OMPth+SYCL | 0.2 | 0.1 |
| 1Gen9.5 | OMPth+SYCL | 0.3 | 0.6 |

where OMPth refers to OpenMP thread, OMPth+offload means OpenMP host and offload functionalities.

We find that among the implemented programming models, the CUDA and OpenACC versions allow to obtain better performance. For example, at maximal $101.7 \times 10^2$ and $59.8 \times 10^2$ particles/s on GridCL, $147.0 \times 10^2$ and $80.6 \times 10^2$ particles/s on Gorgon). The OpenMP offload version is not competitive with the OpenACC version, for instance, merely leading to a $5-10\%$ tracking rate of the CUDA version on GridCL and Gorgon. The Kokkos version obtains even much worse performance than the OpenMP offload version, with the peak performances of $0.7 \times 10^2$ particles/s on GridCL and $1.2 \times 10^2$ particles/s on Gorgon. As for the SYCL version, it is the single version capable of offloading computational kernel to Intel Graphics while there is a lack of compiler support to successfully compile and execute the SYCL version on other machines equipped with Nvidia GPUs. The peak performance of the SYCL version in offload mode is $0.6 \times 10^2$ particles/s on Intel NUC.

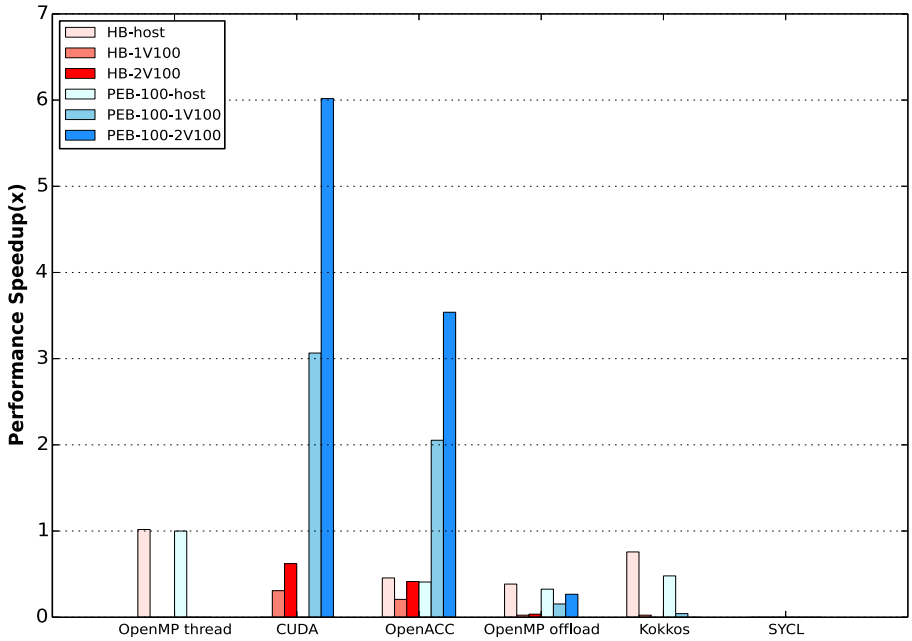



Figure 4.8: slabAllNuclides performance speedup on GridCL, the higher the better - Baseline for the performance is obtained in host mode with the use of pseudo event-based method and a fixed number of bank size, 100.

Figure 4.8 provides a straightforward view for comparison of performance speedup among different programming models on GridCL. From the figure we find that in general the pseudo event-based method (PEB) outperforms the history-based method (HB) in offload mode. Comparing to the baseline performance obtained by the pseudo event-based method fixing the bank size to 100, the CUDA version may reach up to approximately 6.0x speedup and the

OpenACC version may gain a factor of 3.5x speedup. Other implementations including OpenMP offload, Kokkos as well as SYCL are not competitive to the baseline performance in host mode.

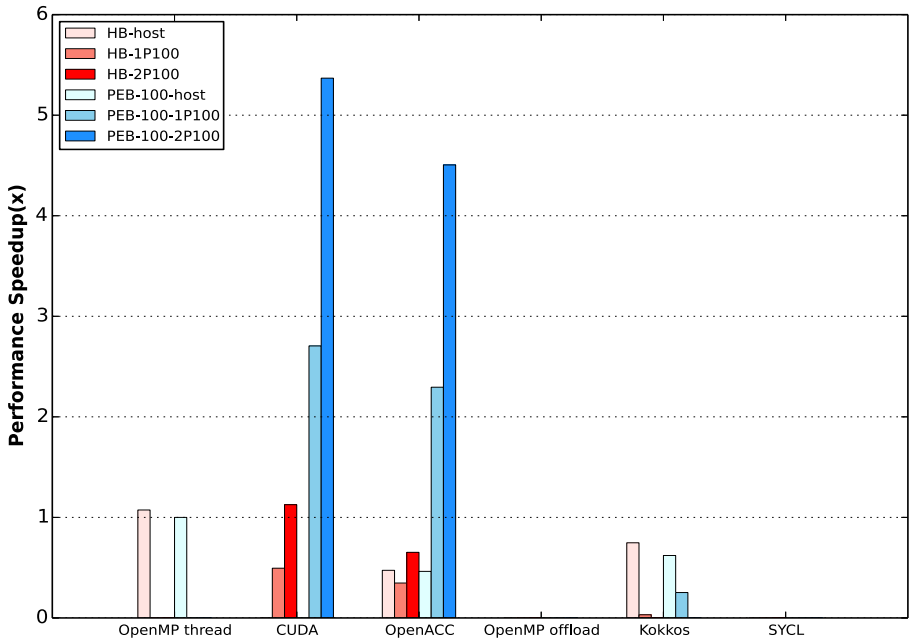



Figure 4.9: slabAllNuclides performance speedup on Cobalt-hybrid, the higher the better - Baseline for the performance is obtained in host mode with the use of pseudo event-based method and a fixed number of bank size, 100.

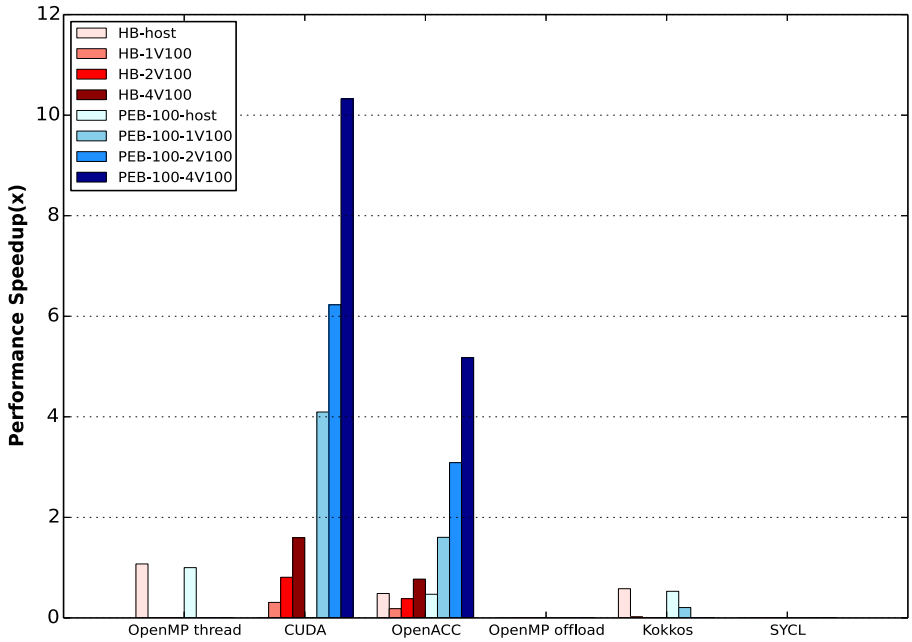



Figure 4.10: slabAllNuclides performance speedup on Cobalt-v100, the higher the better - Baseline for the performance is obtained in host mode with the use of pseudo event-based method and a fixed number of bank size, 100.

Figure 4.9 shows the performances of different programming models on Cobalt-hybrid. In comparison with the baseline performance in host mode, the CUDA version may obtain at maximal a factor of 5.5x speedup and the OpenACC version reaches up to a factor of 4.5x speedup with the utilization of two Nvidia P100s. If we only target to a single accelerator, the CUDA version gains around 2.6x performance speedup and the OpenACC version gets a factor of 2.3x speedup. Except for these two implementations, the Kokkos version in offload mode obtains worse performance than its performance in host mode.

Figure 4.10 depicts the performance speedups of different programming models on Cobalt-v100. By using 4 Nvidia GPUs, the CUDA and OpenACC versions may respectively reach up to at maximal 10.2x speedup and 5.2x speedup compared with the baseline performance in host mode. At the same time, they may obtain approximately 6x and 3x speedups with the utilization of 2 Nvidia GPUs. When targeting to a single device, the CUDA version gains a factor of 4.1x speedup and the OpenACC version a factor of 1.7x speedup. The OpenMP offload and SYCL versions are unable to be compiled and executed on Cobalt-v100. As for the Kokkos version, its performance in offload mode is similar to that on GridCL and Cobalt-hybrid, which is much worse than the performances of the CUDA and OpenACC versions.

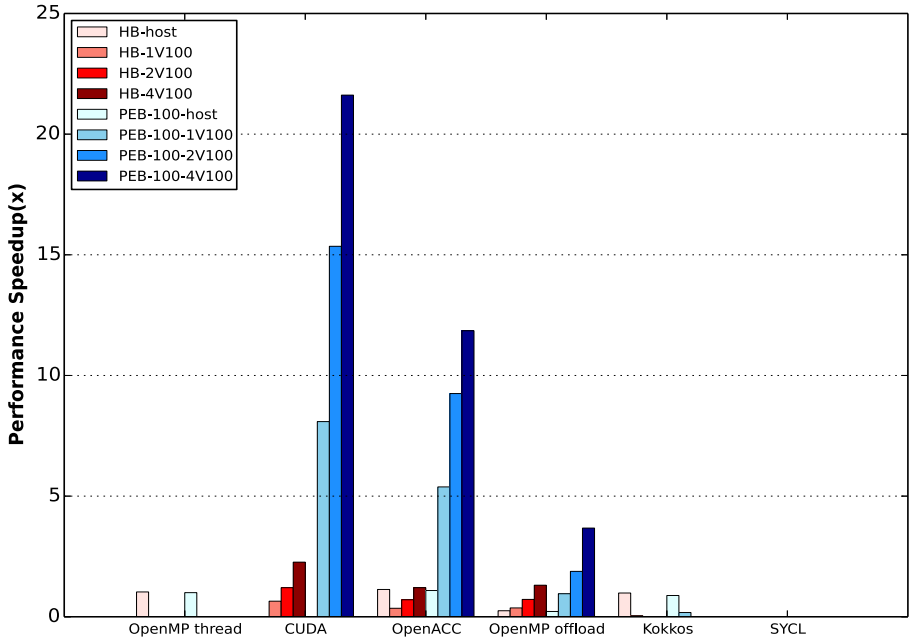



Figure 4.11: slabAllNuclides performance speedup on Gorgon, the higher the better - Baseline for the performance is obtained in host mode with the use of pseudo event-based method and a fixed number of bank size, 100.

The performance speedups tested on the OpenPower-based machine, Gorgon are illustrated in Figure 4.11. Except for the SYCL version, other implementations of different programming languages and libraries can be compiled successfully and the performance gaps among them are similar to those obtained on GridCL. However, the performance speedups of the CUDA, OpenACC, OpenMP offload and Kokkos versions in offload mode in compar-

ison with the OpenMP thread version in host mode are much higher than those tested on GridCL as well as other platforms. For example, the CUDA version may gain a factor of around 22x speedup comparing to the baseline performance. This is due to the fact that the baseline performance in host mode tested on Gorgon via the pseudo event-based method because all compilers achieves no vectorization for the inner-loop in the SIGMA1 on-the-fly Doppler broadening algorithm on OpenPower architecture while the peak performance tested on x86 architecture is obtained by Intel C++ compiler which manages to vectorize the inner-loop with AVX2 extensions.

Besides, the best performance of the OpenMP offload version obtained on Gorgon is better than its peak performance tested on GridCL while the performances of the CUDA and OpenACC versions gained on Gorgon are similar to the peak values tested on GridCL. For instance, the peak performance of the OpenMP offload version tested on GridCL in offload mode with a single device gains $2.6 \times 10^2$ particles/s while its peak performance obtained on Gorgon reaches up to $6.5 \times 10^2$ particles/s. This may be caused by the OpenMP offload functionalities achieved by different compilers including LLVM Clang and XLM C++.

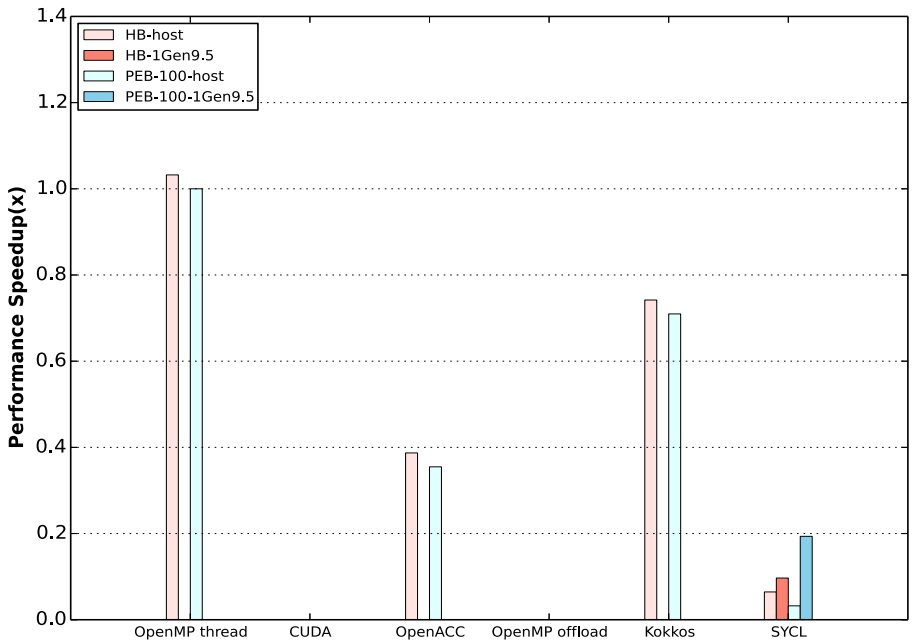



Figure 4.12: slabAllNuclides performance speedup on Intel NUC, the higher the better - Baseline for the performance is obtained in host mode with the use of pseudo event-based method and a fixed number of bank size, 100.

At last, Figure 4.12 shows the performance speedups of different programming models executed on Intel NUC equipped with an integrated graphical card (Intel Gen9.5). The SYCL version is the single implementation capable of offloading computational kernel to Intel Graphics. However, the corresponding performance in host and offload modes is much worse than the baseline performance obtained by the OpenMP thread version executed in host mode, with a factor of 0.1-0.2x speedup. Concerning other programming models such as OpenMP thread, OpenACC and Kokkos which are successfully processed in host mode, the OpenMP thread and Kokkos versions are

compiled by Intel compiler with vectorization support enabled. This is the reason that their performances outperform the performance of the OpenACC version which is compiled by PGI compiler.

Overall, based on the peak performances expressed above, we end up with several conclusions listed as follows:

1. Pseudo event-based method (PEB) surpasses significantly history-based method (HB) with a magnitude of performance speedup in offload mode while its performance is a little less than the performance of history-based method in host mode.
2. In host mode, the implementation that is able to be compiled by Intel compiler with the vectorization options enabled such as the OpenMP thread and Kokkos versions gain better performance than those implementations requiring to be compiled by other compilers such as the OpenACC, OpenMP offload versions.
3. In offload mode (Nvidia P100, V100, Intel Gen9.5):
   - The CUDA version renders the highest performance via PEB comparing to other programming models.
   - The OpenACC version generally gains approximately 50%-60% of the CUDA performance via both HB and PEB.
   - The OpenMP offload version provokes a huge performance degradation via either HB or PEB which is caused by underdeveloped support of CUDA multi-streams for our OpenMP offload implementation achieved by LLVM-Clang and IBM XLC compilers comparing to the CUDA and OpenACC versions.
   - Our Kokkos version is limited to offload computational kernels to a single device and does not support CUDA multistreaming. Its performance in offload mode is less competitive to the baseline performance in host mode which is completely unsatisfactory.
   - Our SYCL version can only be compiled on Intel NUC associated with Intel Graphics. However, its performance in offload mode is much worse than the baseline performance with a factor of 0.1-0.2x speedup. More optimizations and tests are required with the use of more architectures and the next version of SYCL compilers since SYCL is a preliminary programming model and its implementations achieved by different compilers are immature.

## 4.5 Profiling and Optimization

In this section, we intend to perform a series of profiling to understand better about the conclusions drew in the previous section. The analysis is carried out on GridCL focusing on the offload mode by using the profiling tools such as nvprof and nvvp to ensure that most of implementations of different programming models can be analyzed. Unfortunately, the SYCL version is not able to be included in the following analysis. We start the analysis from the comparison between the history-based method and the pseudo event-based method.

### 4.5.1 Profiling of HB and PEB

The number of threads is set to 10 and the bank size for the pseudo event-based method is fixed to 100. The total number of particles is reset to $5 \times 10^3$. We use nvprof to perform an analysis of the CUDA version of slabAllNuclides using both the history-based method and the pseudo event-based method in default summary mode.

Table 4.9: Profiling of the CUDA version of slabAllNuclides in offload mode on GridCL using nvprof.

| GPU activities | Time Percentage(%) | | Execution Time (s) | |
|---|---|---|---|---|
| | HB | PEB | HB | PEB |
| sigma1DopplerBroadening | 85.81 | 72.41 | 9.17 | 1.62 |
| CUDA memcpy HtoD | 9.36 | 22.33 | 1.00 | 0.50 |
| CUDA memcpy DtoH | 4.84 | 5.27 | 0.52 | 0.12 |

The performance is evaluated by the execution time (s), the lower the better.

From Table 4.9 we find that the execution of the CUDA kernel function $sigma1DopplerBroadening()$ and memcpys (HtoD and DtoH) via PEB saves around 90% run time comparing to those via HB. This is in accordance with the performance results introduced in the previous section. To get more detailed information about the CUDA kernels of both HB and PEB methods, we use nvvp to perform a complete profiling analysis which is illustrated in Table 4.10.

Several metrics used in the table above are described as:

- Global memory load/store throughput.
- Global memory load efficiency: ratio of requested global memory load throughput to required global memory load throughput expressed as percentage.
- Global memory store efficiency: ratio of requested global memory store throughput to required global memory store throughput expressed as percentage.
- Achieved occupancy: ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor.
- Branch efficiency: ratio of branch instruction to sum of branch and divergent branch instruction.
- Warp execution efficiency: ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor.
- Double-precision floating-point efficiency: ratio of achieved to peak double-precision floating-point operations.
- Execution dependency: An input required by the instruction is not yet available.

Table 4.10: Metric values of the computational kernel $sigma1DopplerBroadening()$ of the slabAllNuclides CUDA version using nvvp.

| Metrics | HB | PEB |
|---|---|---|
| Grid size | [388,1,1] | [25996,1,1] |
| Block size | [32,1,1] | [32,1,1] |
| Registers/Thread | 69 | 69 |
| Shared Memory/Block | 0B | 0B |
| Theoretical Block Limit | 28 | 28 |
| Device Block Limit | 32 | 32 |
| Global Memory Load Throughput | 33.03GB/s | 262.20GB/s |
| Global Memory Store Throughput | 0.36GB/s | 2.87GB/s |
| Global Memory Load Efficiency | 50.39% | 65.24% |
| Global Memory Store Efficiency | 25.00% | 25.00% |
| Achieved Occupancy | 6.22% | 31.86% |
| Double-precision Floating-point Efficiency | 4.42% | 21.94% |
| Branch Efficiency | 98.94% | 98.68% |
| Warp Execution Efficiency | 94.92% | 94.72% |
| Memory Dependency | 33.11% | 17.97% |
| Execution Dependency | 48.57% | 37.79% |

- Memory dependency: A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding.

With reference to the analysis of the history-based method, the results indicate that the performance of the kernel $sigma1DopplerBroadening()$ is most likely limited by the latency of arithmetic or memory operations which delivers low compute throughput and memory bandwidth comparing to the peak performance of V100, as described in Table 4.10 (compute throughput and global memory load/store throughput). The branch efficiency and warp execution efficiency show that the issues of divergent branches and idle threads per warp for the kernel function $sigma1DopplerBroadening()$ is not the key bottleneck limiting the performance. Besides, the grid size of the kernel is too small to keep the GPU busy so as to hide compute and memory latency. As a result, the achieved occupancy of the kernel via the HB method gains 6.22% while the theoretical occupancy equals to 43.8% as the number of blocks per SM is limited by the usage of registers per thread (69, where 64 is the maximal registers per thread that can be used to allow 32 blocks executing concurrently on each SM). The execution dependency and memory dependency are the most important stall reasons that limit the performance where the values corresponding to these two types of stalls are 48.57% and 33.11%, indicating that the latency of the kernel is mainly introduced by execution and memory dependency stalls and it may be potentially reduced by increasing instruction-level parallelism and optimizing memory alignment as well as access patterns. The memory dependency stall can also be reflected by

the metrics of global memory load and store efficiency, which are respectively 50.39% and 25.00%, meaning that for each memory transaction call, only a half or a quarter of bus volume (32 bytes in total) can be actually used due to memory misalignment and inadequate access patterns.

With reference to the analysis of the pseudo event-based method, although the compute throughput and memory bandwidth improve quite a lot comparing to the metric values of the HB method (64-bit floating-point efficiency varies from 4.42% to 21.94%, global memory load throughput increases from 33.03GB/s to 262.20GB/s, global memory store throughput augments from 0.36GB/s to 2.87GB/s), its performance is still limited by the latency of arithmetic or memory operations. The branch efficiency and warp execution efficiency remain the same as the HB method. The execution dependency and memory dependency are still the most two important reasons causing the performance degradation, taking up respectively 37.79% and 17.97% stalls during the kernel execution, which contribute to less stalls relative to the HB method. Moreover, since the grid size of the kernel for the PEB method is much larger than that of the HB method, the achieved occupancy via the PEB method gains 31.86% which surpasses the value of the HB method significantly and leads to a considerable performance improvement.

In summary, the PEB method has significantly improved the compute throughput and memory bandwidth in comparison with the HB method. The achieved occupancy of the PEB method gains a factor of 5 enhancement comparing to the HB method so as to hide more compute and memory latency. However, even for the PEB method, the utilization of compute resources of Nvidia V100 is still at the medium level and the kernel performance is bound by instruction and memory latency. From one side, the occupancy of the kernel may be improved by reducing the number of registers used for each CUDA thread or by restricting the maximum number of registers per thread with the flag `-maxrregcount`. From the other side, the execution dependency and memory dependency stalls may be reduced by adding instruction-level parallelism and optimizing memory alignment and access patterns. In what follows we continue to perform a series of performance analysis of different implementations of slabAllNuclides (OpenACC, OpenMP offload, Kokkos) using the PEB method to know better about the performance bottlenecks of all these implementations.

### 4.5.2 Profiling of different programming models

The following performance analysis has been carried out with the OpenACC, OpenMP offload, Kokkos versions of slabAllNuclides via the PEB method. The input parameters are set to $5 \times 10^3$ particles, 10 threads, and the bank size is fixed to 100.

#### 4.5.2.1 OpenACC Analysis

From Table 4.11, we can see that the computational kernel of the OpenACC version uses 110 registers for each thread which leads to a smaller theoretical block limit and a lower achieved occupancy comparing to the CUDA

Table 4.11: Metric values of the computational kernel $sigma1DopplerBroadening()$ of the OpenACC, OpenMP offload, Kokkos versions of slabAllNuclides using the PEB method.

| Metrics | OpenACC | OpenMP offload | Kokkos |
|---|---|---|---|
| Grid size | [25996,1,1] | [25996,1,1] | [25996,1,1] |
| Block size | [32,1,1] | [32,1,1] | [1,32,1] |
| Registers/Thread | 110 | 200 | 70 |
| Shared Memory/Block | 276B | 0B | 0B |
| Theoretical Block Limit | 16 | 10 | 28 |
| Global Memory Load Throughput | 195.51GB/s | 57.31GB/s | 62.24GB/s |
| Global Memory Store Throughput | 2.19GB/s | 0.63GB/s | 0.31GB/s |
| Global Memory Load Efficiency | 64.94% | 63.82% | 25.00% |
| Global Memory Store Efficiency | 25.00% | 25.00% | 25.00% |
| Achieved Occupancy | 18.02% | 12.59% | 20.16% |
| Double-precision Floating-point Efficiency | 14.13% | 5.61% | 5.22% |
| Branch Efficiency | 93.16% | 97.50% | 100.00% |
| Warp Execution Efficiency | 39.96% | 27.00% | 100.00% |

version. The memory bandwidth of the OpenACC version is also worse than the CUDA version. Furthermore, the OpenACC version makes use of shared memory automatically and each block contains 276 bytes shared memory. The warp execution efficiency of the OpenACC version gains around 40% which is much less than the CUDA version (around 100%), indicating that the OpenACC kernel generates more intra-warp divergence than the CUDA kernel.

#### 4.5.2.2 OpenMP offload Analysis

The OpenMP offload kernel is similar to the OpenACC kernel. From the aspect of the usage of registers, it uses 200 registers per thread which makes the theoretical block limit decreasing to 10 and results in a lower achieved occupancy in comparison with the OpenACC version. The memory bandwidth of the OpenMP offload kernel degrades significantly due to the lack of CUDA multi-stream support. It should also be noted that the OpenMP offload version makes no use of shared memory as the CUDA version. The warp execution efficiency of the OpenMP offload kernel is 27% which is even less than the OpenACC version, indicating that the OpenMP offload version also has much higher intra-warp divergence than the CUDA version.

#### 4.5.2.3 Kokkos Analysis

The Kokkos version use 70 registers for each thread which is in accordance with the CUDA version. As a results, its achieved occupancy is almost equal to that of the CUDA version. The memory bandwidth of the Kokkos kernel is similar to that of the OpenMP offload kernel as they both provide no support of CUDA multi-streaming. However,

the global memory load efficiency of the Kokkos is 25% which is less than other versions, leading to a larger number of memory transactions for memory load operation. This may be the reason that the performance of the Kokkos version is not competitive with the OpenMP offload version. Besides, the Kokkos kernel also makes no use of shared memory as the CUDA kernel. As for the warp execution efficiency of the Kokkos version, it is equal to 100%, meaning that there is no intra-warp divergence occurring during the execution of the Kokkos kernel.

Table 4.12: Comparison of the performances of the Kokkos versions of slabAllNuclides tested on GridCL with a single Nvidia V100.

| Particle Tracking Method | Simulation Time (s/cycle) | |
|---|---|---|
| | Kokkos basic | Kokkos optimized |
| HB | 426.95 | 393.26 |
| PEB | 97.63 | 87.60 |

The performance is evaluated by the simulation time (s/cycle), the lower the better.

Note that Kokkos also provides the interoperability of CUDA runtime API functions to enable the support of CUDA asynchronous multistreaming. We intend to implement this technique in our Kokkos version as an optimization and the timeline view of nvvp shows that the optimized Kokkos version indeed makes use of CUDA multi-streams and the performance of the optimized Kokkos version has improved 10% relative to the performance of the basic Kokkos version using the PEB method, as showed in Table 4.12.

### 4.5.3 Optimizations of the CUDA version

According to the performance analysis carried out in the previous section, we intend to accomplish several optimizations for the purpose of mitigating the bottlenecks of the kernel of slabAllNuclides. In order to make fully use of CUDA's feature set, the optimization effort has been done on the CUDA version of slabAllNuclides. The input parameters are set to $5 \times 10^5$ particles, 20 threads. The bank size for the PEB method is still set to 100.

#### 4.5.3.1 Zero Copy

Since our design of algorithms, especially for the history-based method, requires many small data transfers during the simulation, the zero copy functionality can be used to obtain higher bandwidth between the host and the device by using pinned memory. Instead of allocating data in pageable host memory which need to be firstly copied to a temporary pinned memory and then transferred to device memory from the pinned memory once a data transfer between host and device is invoked, the zero copy functionality manages to directly allocate the data in pinned memory so as to avoid the cost of data transfer between pageable and pinned memory with the utilization of CUDA runtime API $cudaMallocHost()$ or $cudaHostAlloc()$ and $cudaFreeHost()$.

#### 4.5.3.2 Register Usage

Both the HB and PEB methods own the problem of the overuse of registers per thread which may prevent the kernel from fully utilizing the device. Since in our case, the maximum number of registers that a thread can use to execute 32 blocks concurrently on a SM is 64, we may use the `-maxrregcount=64` flag to manually set the maximum number of registers for each thread.

#### 4.5.3.3 Unified Memory

The CUDA Unified Memory is a single memory address space accessible from both host and device which allows the data to be read and written from CPUs or GPUs without explicit invocation of memory transfer operations. In order to retrieve a pointer accessible from any processor, the CUDA runtime API function $cudaMallocManaged()$ need to be called. In our case, the input data required for kernel execution may be stored in managed memory.

#### 4.5.3.4 Multiple Kernels

Because the computational kernel of the function $sigma1DopplerBroadening()$ consists mainly two parts, a procedure of precalculation to retrieve lower-upper bound and increment several cross section values to the final result and a procedure of integral computation which handles the execution of several inner-loops. When a warp deals with the cross section calculation of a nuclide, all CUDA threads within this warp can only execute the same workloads of the precalculation part simultaneously which makes no contribution to parallelize this procedure so as to gain improved performance.

As a solution, we may separate the original kernel into two small kernels which are dedicated to handling the precalculation part and integral computation part respectively. Each CUDA thread in the precalculation kernel is responsible for precalculating the data of a set of nuclides required in the integral computation kernel. Each warp in the integral computation kernel is responsible for summarizing the final cross section values of a set of nuclides from inner-loops.

#### 4.5.3.5 Fine-grained Kernel

As the performance bottlenecks of the kernel using the PEB method may be alleviated by adding instruction-level parallelism and optimizing memory access patterns, we may redesign our computational kernel into a more fine-grained one where a set of CUDA threads within a warp are assigned to calculate the cross section of a nuclide. For example, we can use 4 or 8 threads to calculate the cross section value of a nuclide and thus a warp can handle 8 or 4 tasks at the same time. In this case, it is preferable to perform the sum reduction for the inner-loops with the utilization of shared memory. The newly designed function of sum reduction is shown above where `NUCS_PER_WARP`

refers to the number of nuclides assigned to a warp and `sigma_tmp` is a table of size 32 allocated in shared memory for data storage.

**Original sum reduction**

```
for (int i = 16; i > 0; i >>= 1)
  sigma += __shfl_xor_sync(0xffffffff, sigma, i);
```

**Newly designed sum reduction**

```
if (threadIdx.x < NUCS_PER_WARP) {
  for (int i = 0; i < 32 / NUCS_PER_WARP; ++i)
    sigma += sigma_tmp[threadIdx.x + i * NUCS_PER_WARP];
}
```

**Original process**

```
int lb = getLowerBound(E1, ...);
int ub = getLowerBound(E2, ...) + 1;
```

**Newly designed process**

```
int task_id = threadIdx.x % NUCS_PER_WARP;
double ee;
if (threadIdx.x < NUCS_PER_WARP) {
  ee = ...;
} else {
  ee = ...;
}
res[threadIdx.x] = getLowerBound(ee, ...);
int lb, ub;
lb = res[task_id];
if (threadIdx.x < NUCS_PER_WARP) {
  ub = res[task_id + NUCS_PER_WARP] + 1;
} else {
  ub = res[threadIdx.x] + 1;
}
```

Furthermore, since the precalculation part includes two invocations of the function $getLowerBound()$, we may also execute these two functions concurrently with the utilization of shared memory. The idea is to divide all 32

threads within a warp into two groups where the two groups are responsible for retrieving respectively the lower and upper bounds of all tasks assigned to this warp.

The value of `NUCS_PER_WARP` can be set to {1, 2, 4, 8, 16}. If we extend this value to 32, meaning that each thread calculates the cross section of a nuclide on its own and there is no vectorization for the inner-loops, we will have no more need to use shared memory since the intra-warp communication among threads is reduced to zero.

A series of tests of slabAllNuclides concerning all these cases have been executed and the corresponding results are listed in Table 4.13. It is obvious that the performance of both the HB and PEB methods improves firstly with the increase of the number of tasks for each warp. After it reaches up to the peak value obtained with `NUCS_PER_WARP` equaling to 4, the performance degrades little by little.

Table 4.13: Granularity analysis of the CUDA version of slabAllNuclides performed on GridCL with a single Nvidia V100 and 20 CPU threads.

| NUCS_PER_WARP | Simulation Time (s/cycle) | |
|---|---|---|
| | HB | PEB |
| 1 | 2257.97 | 93.12 |
| 2 | 2086.65 | 74.95 |
| 4 | 1927.87 | 73.39 |
| 8 | 2228.79 | 85.98 |
| 16 | 2740.79 | 109.98 |
| 32 | 3080.32 | 159.34 |

The performance is evaluated by the simulation time (s/cycle), the lower the better.

We can say that there is always a tradeoff between the increase of instruction-level parallelism and the increase of branch divergence. The performance of the CUDA version using the PEB method under the condition that `NUCS_PER_WARP = 4` gains approximately 21% improvement comparing to the one given by `NUCS_PER_WARP = 1`.

#### 4.5.3.6 Comparison of performances of the CUDA version via different optimizations

According to all optimizations discussed above, we denote them in a simple way to prepare for a thorough performance evaluation.

- *warp*: the original implementation of the CUDA version.
- *zero-copy*: an optimized version with the zero copy functionality.
- *maxrregcount-64*: an optimized version with the maximum usage of registers per thread set to 64.
- *maxrregcount-32*: an optimized version with the maximum usage of registers per thread set to 32.

- *unified-memory*: an optimized version with the use of unified memory.
- *multi-kernels*: an optimized version dividing the original kernel into two small kernels.
- *4-nucs-per-warp*: an optimized version of a fine-grained kernel where each warp handles 4 computational tasks simultaneously.

The corresponding results are illustrated in Table 4.14. We can find that the *zero-copy* strategy improves significantly the performance via the HB method with a factor of 2x speedup while it degrades around 50% of the performance via the PEB method.

Table 4.14: Comparison of the performances of the CUDA implementations of slabAllNuclides tested on GridCL with a single Nvidia V100 and 20 CPU threads.

| Optimizations | Simulation Time (s/cycle) | |
|---|---|---|
| | HB | PEB |
| warp | 1868.24 | 96.87 |
| zero-copy | 933.20 | 198.48 |
| maxrregcount-64 | 1600.47 | 91.69 |
| maxrregcount-32 | 1664.74 | 222.70 |
| unified-memory | 4775.12 | 237.65 |
| multi-kernels | 2121.47 | 80.99 |
| 4-nucs-per-warp | 1927.87 | 73.39 |

The performance is evaluated by the simulation time (s/cycle), the lower the better.

As for the strategy of setting up maximum registers per thread, the *maxrregcount-64* enhances a little performance for both the HB and PEB methods comparing to the original *warp* version while the *maxrregcount-32* improves the performance adopting the HB method and weakens around 40% of the performance relative to the *warp* version using the PEB method. It is recommended to make fully use of available registers for each thread without limiting the theoretical maximum occupancy a kernel may achieve since the local memory of a much lower memory bandwidth comparing to registers will be used to accomplish memory operations for spilled data.

With reference to the *unified-memory* optimization, the results shows that our programming strategy is not a good candidate for the utilization of CUDA Unified Memory since the performance of both the HB and PEB methods using unified memory obtains approximately 60% performance degradation in comparison with the original *warp* version.

In addition to the optimizations discussed above, the implementations optimized by *multi-kernels* and *4-nucs-per-warp* manage to outperforms the original one with a factor of 1.2-1.3x speedup for the PEB method. However, these two optimizations make no contribution to achieve the performance improvement for the HB method, as depicted in Figure 4.13.

Overall, we can draw several conclusions listed as follows:

- With respect to the history-based method, the *zero-copy* strategy is the most useful way to optimize the kernel and improve the performance.
- With respect to the pseudo event-based method, the *multi-kernels* and *4-nucs-per-warp* strategies become the most favorable ways to mitigate performance bottlenecks and achieve optimizations.
- The *maxrregcount* optimization should be used carefully so as to fully exploit the compute resources of the device and avoid too much usage of local memory.
- The CUDA Unified Memory is inappropriate to be used in our case with the heterogeneous offloading strategy.

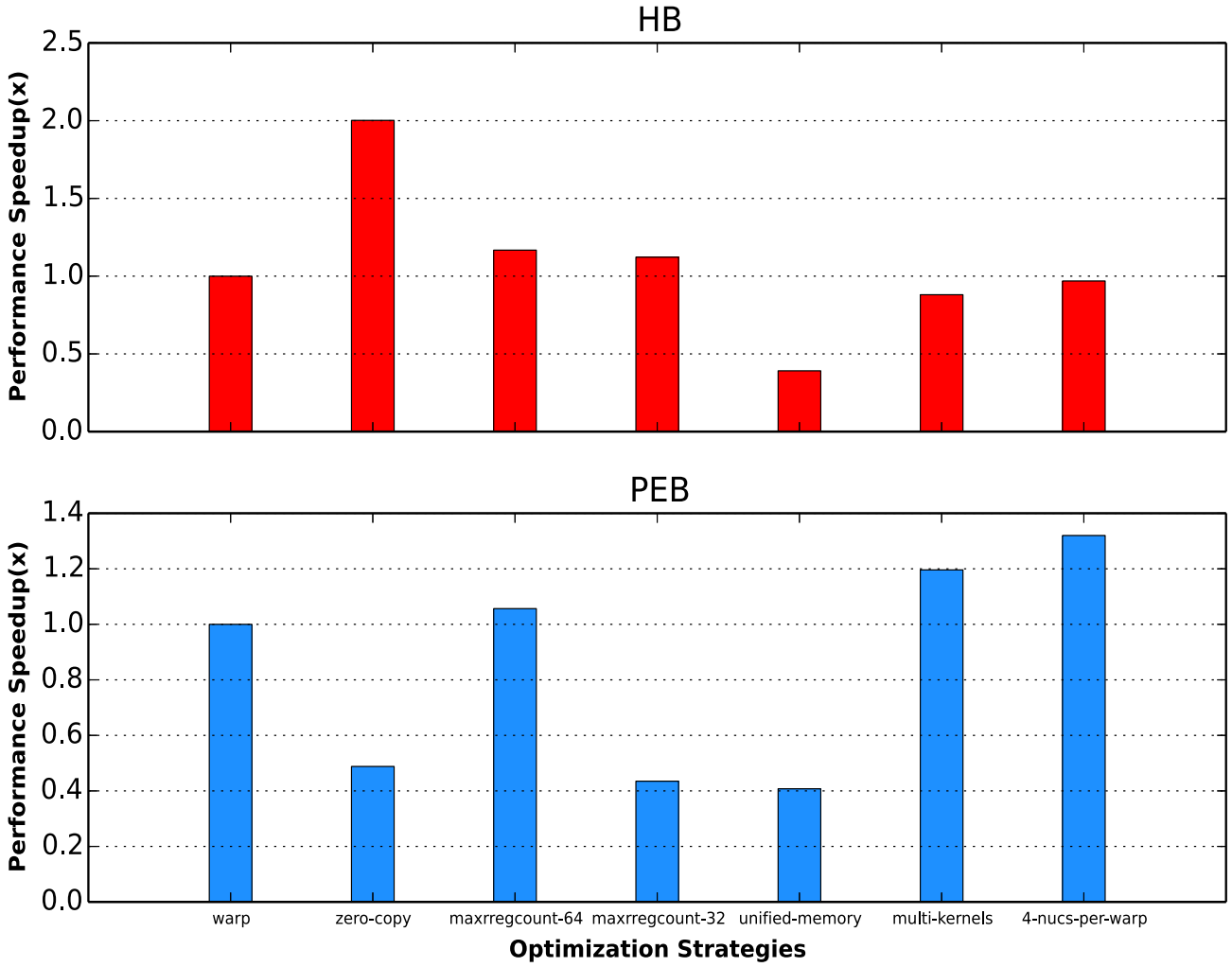



Figure 4.13: Performance speedup of the CUDA version of slabAllNuclides adopting different optimization strategies tested on GridCL with a single Nvidia V100 and 20 CPU threads, the higher the better.

In conclusion, this chapter introduces explicitly the portable implementations of Monte Carlo neutron transport

code with the utilization of a heterogeneous offloading strategy via different programming models. The basic performance evaluation and profiling have been done for the purpose of better understanding the bottlenecks of our portable implementations and achieving several optimizations.

# Chapter 5

# Metrics Comparison among Programming Models

Based on the portable implementations of a set of programming languages or libraries on the Monte Carlo neutron transport code PATMOS, we need to find ways to evaluate them so as to choose the most suitable one to our Monte Carlo code and accessible machines.

Till now, there are few researches addressing the evaluation of Monte Carlo neutron transport codes on supercomputers with reference to 3P (performance, portability, productivity), not to mention the evaluation of Monte Carlo neutron transport codes adopting a partial offloading strategy. To our knowledge, the only work related to the implementation of MC transport solver with concern of portability is carried on by Bleile and his group [21] where they developed a monoenergetic event-based MC neutron transport solver relying on the Nvidia Thrust library and discovered that the Thrust version can only obtain a maximum of 36% performance comparing to the CUDA version.

In order to obtain the correct feedback and take it as reference for future development of next generation Monte Carlo neutron transport code, it is significant to perform an evaluation of our Monte Carlo neutron transport codes in terms of 3P. Thus, in this chapter, we intend to give an explicit evaluation of portable implementations of programming models on PATMOS in respect of performance, portability, and productivity which are leveraged by metrics proposed by ourselves or other researchers. A part of work has already be presented in the conference SNA+MC and will be published recently [32].

## 5.1 Definitions of Metrics

According to the 3P Principle, a programming language or library cannot satisfy performance, portability and productivity at the same time and there are always trade-offs among them. For example, Figure 5.1 intuitively showed the relation among 3P for the programming models implemented in our research. The right arrow indicates that

after several explicit optimizations, the implementation of a programming model on an application may gain better performance but become less productive.

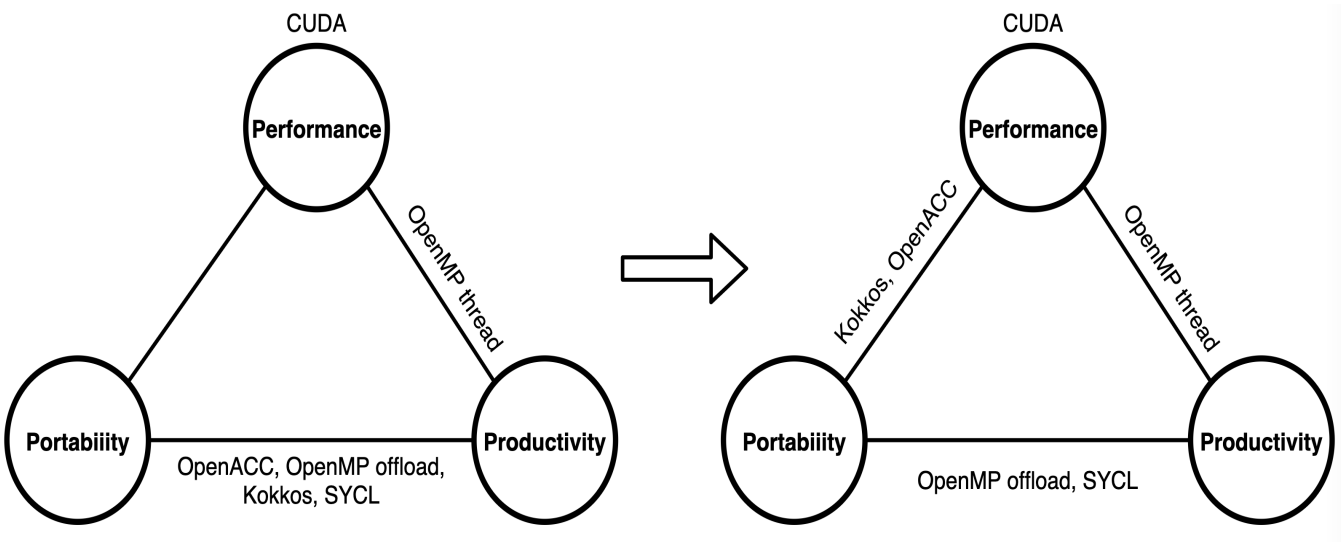



Figure 5.1: An example of 3P principle of the implemented programming models including OpenMP thread, OpenMP offload, OpenACC, Kokkos, SYCL.

However, 3P Principle only states the trade-offs among performance, portability as well as productivity. The lack of measures to accurately capture the 3P properties of an application across different architectures makes it an unsatisfactory candidate for quantitative evaluation with explicitness and conciseness.

From the perspective of quantitative evaluation, some researchers have already done impressive work in terms of performance portability and productivity. For example, Pennycook and his group have proposed a definition for performance portability which highlights the objectivity and measurability as well as a metric to quantify the performance portability of an application across a set of computing architectures [97][98]. This metric has been used in many studies of performance portability [100][40][73][139]. The definition is given as:

**Definition 5.1.1** *Performance portability is a measurement of an application's performance efficiency for a given problem that can be executed correctly on all platforms in a given set.*

Its corresponding metric measures the performance efficiency of an application for a case set $(a, p, H)$ where performance efficiency includes application efficiency, architectural efficiency and so on. The metric is defined as follows:

$$\boldsymbol{\mathcal{P}}(a,p,H) = \begin{cases} \dfrac{|\mathrm{H}|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} & \text{if } i \text{ is supported} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

where $e_i(a, p)$ signifies the performance efficiency of the application $a$ solving a problem $p$ on a platform $i$ in the platform set $H$.

The reason of $e_i(a, p)$ referring to the performance efficiency is that performance efficiency is objective enough to describe the ability of an application exploiting the computing power of a platform for the purpose of solving a problem. However, the use of $\mathcal{P}$ always involves comparisons of estimated values between different applications or problems such as $\mathcal{P}(a_1, p, H)$ against $\mathcal{P}(a_2, p, H)$. It is meaningless that we only compute an aggregated value by $\mathcal{P}$ without doing any comparison. In our opinion, the metrics of absolute performance like execution time or throughput may also be used by $\mathcal{P}$.

Furthermore, $\mathcal{P}$ performs the harmonic mean to ensure that the estimated result is directly proportional to the sum of scores across $H$. Nonetheless, this argument remains to be discussed. Although Smith has proved in his paper [112] that harmonic mean should be used for summarizing performance expressed as a rate (details will be expressed in section 5.1.2.1), which is corresponding to the performance efficiency, such correct aggregation is used to capture an intrinsic property of a particular platform. When it comes to the aggregation of values across a platform set, it is not precise to say that harmonic mean summarizes correctly the measured values to reflect an intrinsic property of a platform set. In fact, whether such intrinsic property of a platform set exists is itself a question to ask. Thus, $\mathcal{P}$ may also adopt other aggregating methods such as arithmetic and geometric means to estimate the data set.

According to the two arguments addressed above, we have come up with an original idea to establish a generic model which defines a variety of metrics with regard to 3P, allowing the utilization of different aggregating methods performing from multiple dimensions such as an application set, a problem set, a platform set.

### 5.1.1 Generic Model

Before the establishment of the generic model, we first give several crucial notions which are repeatedly used for the following definitions.

**Definition 5.1.2** *An application is a program which adopts specific algorithms and programming models to handle one or more problems and produce an output.*

**Definition 5.1.3** *A problem is actually a suite of input parameters that describes a real problem.*

Note that in our case, slabAllNuclides fed with different number of threads, number of particles, number of volumes, bank size are considered as different problems.

**Definition 5.1.4** *A platform is a hardware and software infrastructure that supports the execution of an application (architecture, operating system, compiler, etc.).*

**Definition 5.1.5** *A characteristic is a value describing a property such as performance, productivity, portability, maintainability, energy consumption and so on. This value is obtained by other metrics with the following formula:*

$$c = C(a, p, h) \tag{5.2}$$

*where $C(a, p, h)$ is a metric parameterized by an application, a problem and a platform.*

Given a characteristic $c$, an application set $A = \{a\}$, a problem set $P = \{p\}$ and a platform set $H = \{h\}$, we have three functions parameterized by three fixed variables along with a variable set:

- $\mathcal{A}(c, A, p, h)$: A generic function that evaluates the characteristic $c$ of a set of applications $A$ solving the problem $p$ running on the platform $h$.
- $\mathcal{P}(c, a, P, h)$: A generic function that evaluates the characteristic $c$ of the application $a$ solving a set of problems $P$ running on the platform $h$.
- $\mathcal{H}(c, a, p, H)$: A generic function that evaluates the characteristic $c$ of the application $a$ solving the problem $p$ across a set of platforms $H$.

$\mathcal{A}(c, A, p, h)$ is generally used to describe an intrinsic property of a given problem and platform. For example, we can set OpenMP thread version of slabAllNuclides via the history-based and pseudo event-based methods as two elements in $A$ and fix a particular problem $p$. Then we compare the results of $\mathcal{A}(c, A, p, h_1)$ and $\mathcal{A}(c, A, p, h_2)$ where $h_1$ and $h_2$ differentiate between each other on the type of compiler. In this way, the general influence of compilers for history-based and pseudo-event based applications on an expected property can be evaluated numerically.

$\mathcal{P}(c, a, P, h)$ indicates a general property of a given application and platform solving a set of problems. For instance, if we set $P = \{p_i\}_{i \in 1,2,\ldots,10}$ as ten problems with different number of threads, $\mathcal{P}(c, a, P, h)$ will become a value reflecting the scalability of the application $a$ running on the platform $h$. The comparison between $\mathcal{P}(c, a_1, P, h)$ and $\mathcal{P}(c, a_2, P, h)$ shows the corresponding scalabilities of the applications $a_1$ and $a_2$ on a particular platform.

Similarly, $\mathcal{H}(c, a, p, H)$ is calculated to highlight a property of a specific application and problem across a platform set. Let $H$ be a group of architectures, then $\mathcal{H}(c, a, p, H)$ leverages a property relevant to portability for a particular application and problem.

Now we transfer $\mathcal{A}$, $\mathcal{P}$, $\mathcal{H}$ to three operators $\boldsymbol{F_A}$, $\boldsymbol{F_P}$, $\boldsymbol{F_H}$ as described in Equation 5.3 which signify the computations of a data set $\boldsymbol{C}(a, p, h)$ retrieved from one of three dimensions including application, problem and platform so as to eventually calculate an aggregated metric.

$$\begin{aligned} \mathcal{A}(c, A, p, h) &= \boldsymbol{F_A} \cdot \boldsymbol{C}(a, p, h) \\ \mathcal{P}(c, a, P, h) &= \boldsymbol{F_P} \cdot \boldsymbol{C}(a, p, h) \\ \mathcal{H}(c, a, p, H) &= \boldsymbol{F_H} \cdot \boldsymbol{C}(a, p, h) \end{aligned} \tag{5.3}$$

Since our objective is to establish a generic model which applies to all three dimensions, the mathematical expressions should be independent of each dimension. Thus we define $\boldsymbol{F}$ as a simplified notation of $\{\boldsymbol{F_A}, \boldsymbol{F_P}, \boldsymbol{F_H}\}$ and we obtain:

$$\boldsymbol{F} \cdot \boldsymbol{C}(a,p,h) = \boldsymbol{F} \cdot \{C_i(a,p,h)\}_{i=1,2,\ldots,N} = \begin{cases} \mathcal{C}(\{a_i\},p,h) & = & \mathcal{C}(A,p,h) & \text{if } \forall a_i \in A \\ \mathcal{C}(a,\{p_i\},h) & = & \mathcal{C}(a,P,h) & \text{if } \forall p_i \in P \\ \mathcal{C}(a,p,\{h_i\}) & = & \mathcal{C}(a,p,H) & \text{if } \forall h_i \in H \end{cases} \tag{5.4}$$

Note that the distribution of all elements in the data set $\{C_i(a,p,h)\}_{i=1,2,\ldots,N}$ is better one-dimensional because it is quite meaningless to aggregate values diversifying on multiple dimensions (e.g. summarization of three values $C(a_2,p_1,h_1)$, $C(a_1,p_2,h_1)$ and $C(a_1,p_1,h_2)$ does not help us to identify the exact influence of each dimension on the property). Therefore, we exclude possibilities of complicate interactions between different dimensions in our generic model.

There are a variety of mathematical models which are extensively used to perform statistical computations on data sets. In what follows we define our generic operator $\boldsymbol{F}$ by Pythagorean means (arithmetic, geometric, harmonic) [99][44] which are respectively denoted as $\boldsymbol{F_{am}}$, $\boldsymbol{F_{gm}}$ and $\boldsymbol{F_{hm}}$.

#### 5.1.1.1 Arithmetic Mean

**Definition 5.1.6** *The metric $\boldsymbol{F_{am}}$ is an operator computing the arithmetic mean of a data set of $N$ values $\{C_i(a,p,h)\}_{i=1,2,\ldots,N}$, defined by Equation 5.5:*

$$\boldsymbol{F_{am}} \cdot \boldsymbol{C}(a,p,h) = \begin{cases} \dfrac{1}{N}\displaystyle\sum_{i=1}^{N} C_i(a,p,h) & (a) \\ \sum_{i=1}^{N} \omega_i C_i(a,p,h) & (b) \end{cases} \tag{5.5}$$

where Equation 5.5(a) is the standard version of arithmetic mean metric using equal weightings while Equation 5.5(b) is the weighted version. Each $C_i(a,p,h)$ is multiplied by a weight $\omega_i$ and the sum of $\{\omega_i\}_{i\in N}$ equals to 1. Note that the standard version is a special case of weighted version with $\omega_i = \dfrac{1}{N}$.

#### 5.1.1.2 Geometric Mean

**Definition 5.1.7** *The metric $\boldsymbol{F_{gm}}$ is an operator computing the geometric mean of a data set of $N$ values $\{C_i(a,p,h)\}_{i=1,2,\ldots,N}$, defined by Equation 5.6:*

$$\boldsymbol{F_{gm}} \cdot \boldsymbol{C}(a,p,h) = \begin{cases} \displaystyle\prod_{i=1}^{N} C_i(a,p,h)^{\frac{1}{N}} & (a) \\ \prod_{i=1}^{N} C_i(a,p,h)^{\omega_i} & (b) \end{cases} \tag{5.6}$$

where Equation 5.6(a) is the standard version of geometric mean metric using equal weightings while Equation 5.6(b) is the weighted version. Each $C_i(a,p,h)$ is raised to the power of a weight $\omega_i$ and the sum of $\{\omega_i\}_{i\in N}$ equals to 1. The standard version is a special case of weighted version with $\omega_i = \dfrac{1}{N}$.

#### 5.1.1.3 Harmonic Mean

**Definition 5.1.8** *The metric $\boldsymbol{F_{hm}}$ is an operator computing the harmonic mean of a data set of $N$ values $\{C_i(a,p,h)\}_{i=1,2,\ldots,N}$, defined by Equation 5.7:*

$$\boldsymbol{F_{hm}} \cdot \boldsymbol{C}(a,p,h) = \begin{cases} \dfrac{N}{\sum_{i=1}^{N} \dfrac{1}{C_i(a,p,h)}} & (a) \\ \dfrac{\frac{1}{\omega_i}}{\sum_{i=1}^{N} \dfrac{}{C_i(a,p,h)}} & (b) \end{cases} \tag{5.7}$$

where Equation 5.7(a)(b) are respectively the standard and weighted versions of harmonic mean metric using equal weightings. Each $\frac{1}{C_i(a,p,h)}$ is multiplied by a weight $\omega_i$ and the sum of $\{\omega_i\}_{i \in N}$ equals to 1. The standard version is also a special case of weighted version with $\omega_i = \dfrac{1}{N}$.

Generally, the arithmetic mean metric $\boldsymbol{F_{am}}$ calculates the sum of a collection of numbers divided by the count of numbers in the collection [63]. The geometric mean metric $\boldsymbol{F_{gm}}$ is often used to aggregate the values of a data set containing different numeric ranges [6] so as to indicate the central tendency by using the products of these values. The harmonic mean metric $\boldsymbol{F_{hm}}$ is appropriate for situations when the average of rates is desired by calculating the inverse of the sum of the inverses of a collection of values. In order to apply these metrics of Pythagorean means correctly for the purpose of accurate capture of a property, it is significant to exploit conditions that these metrics are applicable to.

### 5.1.2 Adaptability Analysis

We say a metric is adaptive to a data set $\boldsymbol{C}(a,p,h)$ if it is capable of evaluating a set of elements $\{C_i(a,p,h)\}_{i=1,2,\ldots,N}$ with correctness. Otherwise, the observed result may be overestimated or underestimated. Based on several studies [112][70] addressing this issue, we arrive at the following definitions.

**Definition 5.1.9** *A single-platform property is characterized by $\boldsymbol{F}$ summarizing a data set $\boldsymbol{C}(a,p,h)$ which varies on the dimension of application or problem. By contrast, a property captured by $\boldsymbol{F}$ with its data set $\boldsymbol{C}(a,p,h)$ varying on the dimension of platform is called across-platform property.*

A single-platform property indicates an intrinsic characteristic of the platform whereas an across-platform property expresses an extrinsic property among platforms. Thus, the adaptabilities of arithmetic, geometric and harmonic mean metrics dependent on the property to be characterized. Before further discussions, we firstly define the form of property used in our model.

**Definition 5.1.10** *A property characterized by $C(a,p,h)$, no matter it is single-platform or across-platform, is generally represented by a base-unit quantity or its inverse as well as a ratio of one base-unit quantity to another base-unit quantity.*

#### 5.1.2.1 Adaptability to Single-platform Properties

Since a single-platform property reflects an intrinsic characteristic of the platform, in order to estimate it correctly, it is important to summarize the measured values for each quantity. Hence, we define the adaptability of $\boldsymbol{F}$ as follow:

**Definition 5.1.11** *Adaptability of the metric $\boldsymbol{F}$ towards a set of single-platform data $\boldsymbol{C}(a,p,h)$ is maintained only if all measured values of each base-unit quantity are accumulated to a total value and the result of $\boldsymbol{F} \cdot \boldsymbol{C}(a,p,h)$ is proportional or inversely proportional to these total values.*

Now we consider the first case, $C(a,p,h)$ is represented by a base-unit quantity such as time and we have a set of data $\boldsymbol{C}(a,p,h) = \{t_i\}_{i=1,2,\ldots,N}$. Based on Equation 5.5, it is obvious to find that the standard version of arithmetic mean metric $\boldsymbol{F_{am}}$ performs well for this data set, since the estimated value is proportional to the sum of measured time.

$$\boldsymbol{F_{am}} \cdot \boldsymbol{C}(a,p,h) = \frac{1}{N}\sum_{i=1}^{N} t_i \quad \propto \quad \textstyle\sum_{i=1}^{N} t_i \tag{5.8}$$

Alternatively, if $C(a,p,h)$ is denoted as the inverse of time, the standard version of harmonic mean metric $\boldsymbol{F_{hm}}$ will be adaptive towards the data set $\boldsymbol{C}(a,p,h) = \{\frac{1}{t_i}\}_{i=1,2,\ldots,N}$ because the final result is inversely proportional to the sum of measured time, as expressed below:

$$\boldsymbol{F_{hm}} \cdot \boldsymbol{C}(a,p,h) = \frac{N}{\sum_{i=1}^{N} t_i} \quad \propto \quad \frac{1}{\sum_{i=1}^{N} t_i} \tag{5.9}$$

As for the third case, we consider $C(a,p,h)$ a ratio such as floating point operations per second (GLOPS) with $\boldsymbol{C}(a,p,h)$ becoming $\{\frac{p_1}{t_1}, \frac{p_2}{t_2}, \ldots, \frac{p_N}{t_N}\}$. If we adopt a strategy of measurement which fixes the execution time $T$, and measures the corresponding floating point operations of $N$ cases, $\{\frac{p_1}{T}, \frac{p_2}{T}, \ldots, \frac{p_N}{T}\}$, the estimate of $\boldsymbol{F_{am}} \cdot \boldsymbol{C}(a,p,h)$ will be similar to Equation 5.8:

$$\boldsymbol{F_{am}} \cdot \boldsymbol{C}(a,p,h) = \frac{1}{N}\sum_{i=1}^{N} \frac{p_i}{T} = \frac{1}{NT}\sum_{i=1}^{N} p_i \quad \propto \quad \textstyle\sum_{i=1}^{N} p_i \tag{5.10}$$

Since this estimate is proportional to $\sum_{i=1}^{N} p_i$, we can say that $\boldsymbol{F_{am}}$ is an accurate measurement of floating point operations per second when all values in $\boldsymbol{C}(a,p,h)$ are measured with the same execution time. For example, an application counting the total single-precision floating point operations within the same time step. However, if we adopt another strategy of measurement by which each $C(a,p,h)$ is evaluated with fixed number of floating point operations, denoted as $P$, then we will have:

$$\boldsymbol{F_{am}} \cdot \boldsymbol{C}(a,p,h) = \frac{1}{N}\sum_{i=1}^{N} \frac{P}{t_i} = \frac{P}{N}\sum_{i=1}^{N} \frac{1}{t_i} \quad \propto \quad \textstyle\sum_{i=1}^{N} \frac{1}{t_i} \tag{5.11}$$

The result is proportional to the sum of the inverses of $t_i$ instead of the inverse of the sum of $t_i$. Therefore,

$\boldsymbol{F_{am}}$ is not suitable to handle this case accurately. But if we apply $\boldsymbol{F_{hm}}$ to the data set, the result resembles in Equation 5.9 indicating that $\boldsymbol{F_{hm}}$ is adaptive to $\boldsymbol{C}(a,p,h)$ retrieved by second strategy of measurement.

$$\boldsymbol{F_{hm}} \cdot \boldsymbol{C}(a,p,h) = \frac{N}{\sum_{i=1}^{N} \frac{t_i}{P}} = \frac{PN}{\sum_{i=1}^{N} t_i} \quad \propto \quad \frac{1}{\sum_{i=1}^{N} t_i} \tag{5.12}$$

Note that the conclusions drawn above expands the ones drawn in Smith's previous work [112] where he concludes that arithmetic mean is a bad candidate for summarizing floating point operations per second and this quantity should be estimated by harmonic mean instead. The reason he arrived at such conclusions is that in his paper, each case is assumed to perform the same number of floating point operations during the execution time. This assumption is actually our second strategy of measurement proposed above.

In fact, the standard versions of $\boldsymbol{F_{am}}$ and $\boldsymbol{F_{hm}}$ adopting two strategies of measurement can be considered as weighted versions of $\boldsymbol{F_{am}}$ and $\boldsymbol{F_{hm}}$ applying to $\boldsymbol{C}(a,p,h)$ with $\boldsymbol{C}(a,p,h) = \{\frac{p_1}{t_1}, \frac{p_2}{t_2}, \ldots, \frac{p_N}{t_N}\}$ and the formulas resemble in Equation 5.10 and Equation 5.12, which are expressed as

$$\boldsymbol{F} \cdot \boldsymbol{C}(a,p,h) = \left\{ \begin{array}{ccccccc} \sum_{i=1}^{N} \omega_i \frac{p_i}{t_i} & = & \sum_{i=1}^{N} \left( \frac{t_i}{\sum_{i=1}^{N} t_i} \right) \frac{p_i}{t_i} & = & \frac{1}{N} \frac{\sum_{i=1}^{N} N p_i}{\sum_{i=1}^{N} t_i} & = & \frac{1}{NT} \sum_{i=1}^{N} p'_i \\ \frac{1}{\sum_{i=1}^{N} \omega_i \frac{t_i}{p_i}} & = & \frac{1}{\sum_{i=1}^{N} \left( \frac{p_i}{\sum_{i=1}^{N} p_i} \right) \frac{t_i}{p_i}} & = & \frac{N \sum_{i=1}^{N} p_i}{\sum_{i=1}^{N} N t_i} & = & \frac{PN}{\sum_{i=1}^{N} t'_i} \end{array} \right\} = \frac{\sum_{i=1}^{N} p_i}{\sum_{i=1}^{N} t_i} \tag{5.13}$$

where:

- $\omega_i = \frac{t_i}{\sum_{i=1}^{N} t_i}$ or $\frac{p_i}{\sum_{i=1}^{N} p_i}$
- $p'_i = N p_i$
- $t'_i = N t_i$
- $T = \sum_{i=1}^{N} t_i$
- $P = \sum_{i=1}^{N} p_i$

According to Equation 5.13, we know that for a data set $\boldsymbol{C}(a,p,h) = \{\frac{p_1}{t_1}, \frac{p_2}{t_2}, \ldots, \frac{p_N}{t_N}\}$ characterizing an intrinsic single-platform property, its optimal estimate is equaling to $\frac{\sum_i^N p_i}{\sum_i^N t_i}$ and we can always find a weighted version of $\boldsymbol{F_{am}}$ or $\boldsymbol{F_{hm}}$ to obtain it. However, for the standard versions of $\boldsymbol{F_{am}}$ or $\boldsymbol{F_{hm}}$, it is not safe to directly apply them to any data set in a form like $\boldsymbol{C}(a,p,h) = \{\frac{p_1}{t_1}, \frac{p_2}{t_2}, \ldots, \frac{p_N}{t_N}\}$. We better fix one quantity and measure the other.

#### 5.1.2.2 Adaptability to Across-platform Properties

When it comes to address an across-platform property, it is not easy to decide which metric is the most suitable one for a given data set. From the one side, since each measured value indicates the expected property of the corresponding platform, we don't need paying attention to correctly summarize these values like what we have

discussed in previous section. From the other side, how to choose a metric and properly use it highly depends on the objective of users and the property to be aggregated. Therefore, we need to know more about the nature of three metrics, $\boldsymbol{F_{am}}$, $\boldsymbol{F_{gm}}$, $\boldsymbol{F_{hm}}$.

In general, $\boldsymbol{F_{am}}$, $\boldsymbol{F_{gm}}$ and $\boldsymbol{F_{hm}}$ have a relation described below, the equality among them is achieved under the condition that all elements in the data set $\boldsymbol{C}(a,p,h)$ are equal.

$$\boldsymbol{F_{hm}} \leqslant \boldsymbol{F_{gm}} \leqslant \boldsymbol{F_{am}} \tag{5.14}$$

Concerning about the relation between the percentage of change of the metric $\boldsymbol{F}$ and the amount of change for an element in the data set, we assume that $\boldsymbol{C}(a,p,h) = \{c_1, c_2, \ldots, c_N\}$, $\boldsymbol{C'}(a,p,h) = \{c_1 + M, c_2, \ldots, c_N\}$ and $M$ is a constant signifying the amount of change for $c_1$. Let $K_1 = \sum_{i=1}^{N} c_i$, $K_2 = \prod_{i=1}^{N} c_i$, $K_3 = \sum_{i=1}^{N} \frac{1}{c_i}$ and we have:

$$\frac{\boldsymbol{F_{am}} \cdot (\boldsymbol{C'}(a,p,h) - \boldsymbol{C}(a,p,h))}{\boldsymbol{F_{am}} \cdot \boldsymbol{C}(a,p,h)} = \frac{K_1 + M - K_1}{K_1} = \frac{M}{K_1} \tag{5.15}$$

$$\frac{\boldsymbol{F_{gm}} \cdot (\boldsymbol{C'}(a,p,h) - \boldsymbol{C}(a,p,h))}{\boldsymbol{F_{gm}} \cdot \boldsymbol{C}(a,p,h)} = \frac{K_2^{\frac{1}{N}}(1 + \frac{M}{c_1})^{\frac{1}{N}} - K_2^{\frac{1}{N}}}{K_2^{\frac{1}{N}}} = (1 + \frac{M}{c_1})^{\frac{1}{N}} - 1 \tag{5.16}$$

$$\frac{\boldsymbol{F_{hm}} \cdot (\boldsymbol{C'}(a,p,h) - \boldsymbol{C}(a,p,h))}{\boldsymbol{F_{hm}} \cdot \boldsymbol{C}(a,p,h)} = \frac{K_3}{K_3 + \frac{1}{c_1+M} - \frac{1}{c_1}} - 1 = \frac{M}{K_3 c_1 (c_1 + M) - M} \tag{5.17}$$

Equation 5.15 signifies that the amount of change introduced by any element in the data set leads to the same percentage of change for the metric $\boldsymbol{F_{am}}$. On the contrary, Equation 5.16 and Equation 5.17 indicate that the percentage of change for $\boldsymbol{F_{gm}}$ and $\boldsymbol{F_{hm}}$ depends on the value of the changing element in the data set.

If we set $\rho_1 = \dfrac{M}{c_1}$ as the percentage of change of the changing element, the percentages of change for $\boldsymbol{F_{am}}$ and $\boldsymbol{F_{hm}}$ will turn into $\frac{c_1 \rho_1}{K_1}$ and $\frac{1}{K_3 c_1 (1 + \frac{1}{\rho_1}) - 1}$. It means that given a fixed percentage of change for one of elements $C_i(a,p,h)$, the corresponding percentages of change for $\boldsymbol{F_{am}}$ and $\boldsymbol{F_{hm}}$ depend on $c_i$, in other words, on the choice of element. By contrast, the percentage of change for $\boldsymbol{F_{gm}}$ becomes $(1 + \rho_1)^{\frac{1}{N}} - 1$, indicating that the percentage of change introduced by any element contributes to the same percentage of change for $\boldsymbol{F_{gm}}$. Based on the analysis above, we arrive at two definitions:

**Property 5.1.1** *Given an amount of change for the value of any case in $\boldsymbol{C}(a,p,h)$, the percentage of change for $\boldsymbol{F_{am}}$ is case-independent.*

**Property 5.1.2** *Given a percentage of change for the value of any case in $\boldsymbol{C}(a,p,h)$, the percentage of change for $\boldsymbol{F_{gm}}$ is case-independent.*

In addition, $\boldsymbol{F_{am}}$ and $\boldsymbol{F_{gm}}$ accept both positive numbers and zero while $\boldsymbol{F_{hm}}$ only applies to positive number. Note that $\boldsymbol{F_{gm}} = 0$ if any element in $\boldsymbol{C}(a,p,h)$ equals to zero. However, since $\boldsymbol{F_{gm}}$ usually cooperates with logarithm

to convert the product of values to the sum of logarithms of values, one may need to pay more attention to the use of zero value.

$\boldsymbol{F_{gm}}$ also has a property that $\boldsymbol{F_{am}}$ and $\boldsymbol{F_{hm}}$ do not have, that is, the geometric mean of the ratio of two data sets (the geometric mean of a normalized data set) equals to the ratio of the geometric mean of one data set to the geometric mean of the other data set.

$$\boldsymbol{F_{gm}} \cdot \left(\frac{\boldsymbol{C'}(a,p,h)}{\boldsymbol{C}(a,p,h)}\right) = \frac{\boldsymbol{F_{gm}} \cdot \boldsymbol{C'}(a,p,h)}{\boldsymbol{F_{gm}} \cdot \boldsymbol{C}(a,p,h)} \tag{5.18}$$

This feature allows $\boldsymbol{F_{gm}}$ being capable of giving consistent results for a normalized data set. However, if we want to aggregate performance values such as time, floating point operations, it is recommended to avoid the use of the geometric mean since multiplications of times or floating point operations are meaningless [112].

### 5.1.3 Metrics of Portability, Performance, Productivity

#### 5.1.3.1 Metrics of Portability

After the establishment of generic model $\boldsymbol{F}$, now we take a look at the metrics of portability. We give a classical definition of portability in HPC domain.

**Definition 5.1.12** *Portability refers to the ability of an application $a$ to solve a problem $p$ correctly on a given set of platforms $H$.*

Thus, the generic metric of portability is $\boldsymbol{F}$ applying to a specific data set $\boldsymbol{C}(a,p,h)$ and we have three different sub-types, $\boldsymbol{F_{am}}$, $\boldsymbol{F_{gm}}$ and $\boldsymbol{F_{hm}}$. Since the portability of an application solving a problem on a platform can be described by a boolean value, 0 or 1, referring to non-portable or portable, the values of elements $\{C_i(a,p,h)\}_{i=1,2,\ldots,N}$ can be expressed as:

$$\forall i = 1,2,\ldots,N, \quad C_i(a,p,h) = \begin{cases} 0 & \text{if } (a,p) \text{ fails to run correctly on } h \\ 1 & \text{if } (a,p) \text{ runs correctly on } h \end{cases} \tag{5.19}$$

According to Equation 5.19, we have:

$$\begin{aligned} \boldsymbol{F_{gm}} \cdot \boldsymbol{C}(a,p,h) = \boldsymbol{F_{hm}} \cdot \boldsymbol{C}(a,p,h) &= \begin{cases} 0 & \text{if } \exists h \in H, h \text{ doesn't support } (a,p) \\ 1 & \text{if } \forall h \in H, h \text{ supports } (a,p) \end{cases} \\ \boldsymbol{F_{am}} \cdot \boldsymbol{C}(a,p,h) &= \frac{|S|}{|H|} \end{aligned} \tag{5.20}$$

where $S$ is the sub-set of $H$ containing all supported platforms in $H$ for $(a,p)$. $|S|$ is the total number of supported platforms and $|H|$ is the total number of platforms.

Note that the harmonic mean metric $\boldsymbol{F_{hm}}$ only applies to positive number. In this scenario, we use limits $\frac{1}{0} \to \infty$ and $\frac{1}{\infty} \to 0$ to allow calculations of zero values for $\boldsymbol{F_{hm}}$. We can say that both $\boldsymbol{F_{gm}}$ and $\boldsymbol{F_{hm}}$ accurately describe the portability of a couple $(a, p)$ on a set of platforms $H$. On the contrary, $\boldsymbol{F_{am}}$ fails to address the feature of portability across multiple platforms.

#### 5.1.3.2 Metrics of Performance

Generally, performance is a key property of interest to HPC community. Here we a generic definition of performance for a triple $(a, p, h)$.

**Definition 5.1.13** *Performance refers to any measurable property reflecting the running efficiency of an application $a$ to solve a problem $p$ correctly on a platform $h$.*

There are a variety of metrics capable of characterizing the performance, commonly including time-based metrics (e.g. execution time) and energy-based metrics (e.g. energy consumption). We list several time-based metrics as follows:

- **Time/Duration**: A basic metric describing the performance. It is the easiest metric to be measured, with `sec` as its unit of measurement.
- **Throughput**: A metric denoting the total amount of floating point operations which can be processed per unit of time. It can be measured by some profiling tools such as Intel Advisor and nvprof. `GFLOPS` and `MFLOPS` are common units of measurement.
- **Bandwidth**: A metric signifying the total amount of data which can be handled per unit of time. It can also be measured by profiling tools like Intel Advisor and nvprof. `gigabytes/sec`, `megabytes/sec` are widely used as units of measurement.
- **Particle Tracking Rate**: A particular metric targeting to describe the performance of any particles codes such as MC codes. Its unit of measurement is `particles/sec`.
- **Speedup**: A metric representing a ratio of a measured value to a baseline value on the platform. This metric has no unit.
- **Application Efficiency**: A metric describing the achieved performance as a fraction of the best observed performance on the platform [109].
- **Architectural Efficiency**: A metric representing the ratio of achieved performance to the peak theoretical performance on the platform.

#### 5.1.3.3 Metrics of Productivity

The term "productivity" is a broad concept involving many aspects in HPC community. One may be of interest to assess the productivity of a HPC center by measuring the ratio of scientific output to total costs of ownership [131][132] or to measure the productivity of an individual by computing the performance of an application that he has been devoted to developing as a fraction of his development effort [138]. When it comes to the productivity of an application, it may be assessed by the LOC (lines of code) required by this application which adopts a particular programming model or algorithm to achieve an expected behavior.

Due to the confounding factors of productivity, the correctness of flattening it into a metric has been challenged [64] and it seems that to make a metric of productivity more convincing, one should narrow the concept of productivity down as much as possible, only focusing on a single factor to study.

The first level of classification is based on the subject. Productivity of HPC's sites, developers, applications differentiates among each other. For example, we use LOC as a measurement of productivity. For a developer, it's a subjective metric since LOC highly depends on his skillset. But for an application, LOC is quite objective because the basic mechanisms required by this application of a specific programming model or algorithm to perform an expected behavior remain almost unchangeable.

The second level of classification divides productivity into explicit productivity and implicit productivity where explicit productivity bridges the relat on between output and input with a formula like $\frac{\texttt{output}}{\texttt{input}}$ while implicit productivity only focuses on the judgement of input. The input may be development effort and the output is usually considered as performance.

In our case, we firstly study the implicit productivity of an application $a$ solving a problem $p$ on a platform $h$, $(a, p, h)$. The definition is given by:

**Definition 5.1.14** *Implicit productivity of a case $(a, p, h)$ refers to the development effort devoted to $a$ for solving a problem $p$ on a platform $h$.*

Note that the time dependence of development effort is ignored because this quantity is heavily influenced by developer's skillset. Thus, from the perspective of applications, time should not be considered as a parameter in the metrics of implicit productivity.

The typical metrics of implicit productivity are LOC and NW (number of words), which has already been used in many productivity studies [117][83]. Since the code size of an application is relevant to the development style of developers, comparisons of LOC or NW among a set of applications should be done under the condition that all applications are developed in a same coding style, adopting same algorithms.

We propose another metrics of implicit productivity, PS-LOC (platform-specific lines of code) and PS-NW (platform-specific number of words) which are used to measure the lines of code or number of words of a platform-specific code path for an application which allows it executing correctly on a particular platform.

The Halstead complexity metric [53] is also a choice to characterize implicit productivity of a case $(a, p, h)$. Instead of counting lines of code, we separate each line into a group of operators and operands and then obtain the program length $N$ by summarizing $N_1$ and $N_2$ where $N_1$ and $N_2$ are the total number of occurrences of operators and operands. Similarly, the vocabulary $n$ is aggregated by $n_1$ and $n_2$ where they represent respectively the total number of distinct operators and operands. Based on $N$ and $n$, the program volume $V$ and program difficulty $D$ are defined as $V = N \log_2 n$ and $D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$. The program effort $E$ is calculated by $V$ and $D$ with the formula $E = V \cdot D$. Parameters mentioned above mainly including $N$, $V$ and $E$ can be used as metrics to evaluate implicit productivity. Moreover, in order to address the program length of a platform-specific code path for an application, a new metric PS-N (platform-specific program length) is proposed the same as PS-LOC and PS-NW.

Besides, Harrell et al. [56] proposed a metric $D(A)$ called "code divergence" to quantify the difference of LOC between distinct applications targeting different platforms in $A$. The formula is given by:

$$\begin{aligned} D(A) &= \begin{pmatrix} |A| \\ 2 \end{pmatrix}^{-1} \sum_{\{a_i, a_j\} \subset A} d\left(a_i, a_j\right) \\ d\left(a_i, a_j\right) &= \frac{|LOC(a_i) - LOC(a_j)|}{min(LOC(a_i, LOC(a_j))} \end{aligned} \tag{5.21}$$

where $d\left(a_i, a_j\right)$ represents the pairwise distance which is quantified by normalized change in LOC between the application $a_i$ targeting to a particular platform and the application $a_j$ targeting to the same or different platform.

In fact, the pairwise distance $d\left(a_i, a_j\right)$ can be used to calculate the corresponding distance of an application between its different platform-specific code paths. We have:

$$d_{i,j}(a, p, h) = d\left[(a, p, h_i), (a, p, h_j)\right] = \frac{|LOC(a, p, h_i) - LOC(a, p, h_j)|}{min(LOC(a, p, h_i), LOC(a, p, h_j))} \tag{5.22}$$

Furthermore, for applications implemented by high level programming model, their code paths of different platforms may differ little between each other despite some addition and removals of lines. For example, an application implemented by OpenACC may use a single line to set device type, such as `acc_device_nvidia` and `acc_device_host`. In order to change its device type from Nvidia GPU to host, it is required to delete the line of `acc_device_nvidia` and add a new line targeting to `acc_device_host`. The total number of LOC of two code paths remain the same. Therefore, for the purpose of characterizing this feature, a measure called churn is defined [56]:

$$\begin{aligned} chn_{i,j}(a, p, h) &= \frac{\text{\# lines of add/del}}{l} \\ l &= \begin{cases} 1, & \Delta LOC = 0 \\ \Delta LOC & \Delta LOC \neq 0 \end{cases} \\ \Delta LOC &= |LOC(a, p, h_i) - LOC(a, p, h_j)| \end{aligned} \tag{5.23}$$

Several relative metrics of measurements mentioned above may also be specified with respect to baseline

values as reference, such as relative LOC, relative program effort and so on. Overall, we have a set of implicit productivity metrics described in the following equation:

$$C(a,p,h) = \begin{cases} d_{i,j}(a,p,h) \text{ and } chn_{i,j}(a,p,h) \\ \text{program volume/effort} \\ \text{N (program length) and PS-N} \\ \text{LOC and PS-LOC, NW and PS-NW} \\ \text{relative metrics derived from items above} \\ \ldots \end{cases} \tag{5.24}$$

As for explicit productivity, it is in fact a hybrid definition of "performance-implicit productivity" which is described as follow:

**Definition 5.1.15** *Explicit productivity of a case $(a,p,h)$ refers to the ratio of the performance of an application $a$ solving a problem $p$ on a platform $h$ to the corresponding development effort devoted to $a$.*

A classical measure of explicit productivity is called RDTP (relative development time productivity), which is shown in Equation 5.25. This metric is originally developed to evaluate parallel code development where the relative speedup and effort are ratios of performance/development effort of parallel code to that of serial code [49].

$$\Psi_{\text{relative}} = \frac{\text{relative speedup}}{\text{relative effort}} \tag{5.25}$$

As a matter of fact, the reference version required to compute relative speedup and relative effort in Equation 5.25 may not limited to a serial application. In our scenario, we may set a parallel code as reference, such as OpenMP version, and calculate relative metrics of a set of parallel applications implementing different programming models.

Furthermore, the relative effort in $\Psi_{\text{relative}}$ is mainly characterized by LOC, although other attributes can also be considered such as development time, function points [138][67]. We prefer some metrics of implicit productivity such as PS-LOC to account for development effort in $\Psi_{\text{relative}}$ and we eventually have:

$$C(a,p,h) = \frac{\text{relative speedup}}{\text{relative metric of implicit productivity}} \tag{5.26}$$

It should be noted that if the value of relative speedup or metric of implicit productivity can't be zero, otherwise, $C(a,p,h)$ becomes meaningless.

### 5.1.4 Metrics of Performance Portability

The metric $\boldsymbol{\mathcal{P}}(a,p,H)$ proposed by Pennycook which is described in Equation 5.1 can actually be represented by the standard version of portability metric $\boldsymbol{F_{hm}}$ applying to a data set $\boldsymbol{C}(a,p,h)$ retrieved by a performance efficiency

metric with $C_i(a, p, h) = e_i(a, p)$, and we have:

$$\boldsymbol{F_{hm}} \cdot \boldsymbol{C}(a, p, h) = \boldsymbol{\mathcal{P}}(a, p, H) \tag{5.27}$$

With reference to discussions above, the metric $\boldsymbol{\mathcal{P}}$ can be expanded from adopting the metrics of relative performance to other metrics of absolute performance with the utilization of multiple aggregating methods. We arrive at a new definition of performance portability along with a new metric of performance portability.

**Definition 5.1.16** *Performance portability is a measurement adopting various aggregating methods which estimates the performance of an application $a$ running correctly on a platform set $H$ to solve a given problem $p$ with the intention of comparison.*

In this way, $\boldsymbol{F}$ becomes the generic metric of performance portability when it applies to a data set $\boldsymbol{C}(a, p, h)$ retrieved by a metric of performance.

#### 5.1.4.1 Adaptability of $F$ towards performance metrics

Now that we have several metrics of performance portability and a set of performance metrics including execution time, application efficiency and so on, a study of adaptability of $\boldsymbol{F}$ towards these performance metrics is going to be of great significance.

Because the objective of summarization of values is no longer to reflect an intrinsic property, but to simply aggregate measured data for comparison, the common rule that we should follow is that each platform in the platform set should be treated independently and equally. In other words, since each platform has its own scale of performance for an application to solve a problem, the performance improvement with reference to the maximal performance on platforms $h_1$ and $h_2$ should make the same contribution to the estimated value of $\boldsymbol{F}$. Therefore, we have the following definition:

**Definition 5.1.17** *Adaptability of a performance portability metric $\boldsymbol{F}$ towards a data set $\boldsymbol{C}(a, p, h)$ retrieved by a performance metric is maintained only if the contribution made by any platform in a platform set $H$ is treated independently and equally.*

For instance, consider an application set $A = \{a_1, a_2, a_3\}$ solving a given problem $p$ across a platform set $H = \{h_1, h_2\}$ as illustrated in Table 5.1. We assume that comparing to the application $a_1$, $a_2$ and $a_3$ respectively gains a $25\%$ performance improvement on platforms $h_1$ and $h_2$. According to Definition 5.1.17, they should make same contribution to the estimated value of $\boldsymbol{F}$, thus $\boldsymbol{F} \cdot \boldsymbol{C}(a_2, p, h) = \boldsymbol{F} \cdot \boldsymbol{C}(a_3, p, h)$.

Based on Definition 5.1.17, we need to find out in which way the contribution of performance on platforms is expressed by different performance metric. In general, there are two ways which are individually described as a percentage of change and an amount of change.

Table 5.1: Execution time of applications $\{a_1, a_2, a_3\}$ solving a given problem across a set of platforms $\{h_1, h_2\}$, the lower the better.

| Applications | Platforms | | Performance Improvement | Expected $\boldsymbol{F} \cdot \boldsymbol{C}(a, p, h)$ |
|---|---|---|---|---|
| | $h_1$ | $h_2$ | | |
| $a_1$ | 10 | 100 | - | - |
| $a_2$ | 7.5 | 100 | 25% | equal |
| $a_3$ | 10 | 75 | 25% | equal |

The first type mainly applies to several performance metrics indicating the absolute performance, including time, throughput, bandwidth as well as particle tracking rate. Because the absolute performance is unable to reflect the scale of performance on a particular platform, we can use the percentage of change to implicitly express the scaled contribution on this platform.

On the contrary, the second type is for metrics measuring the relative performance such as application efficiency and architectural efficiency. Since these metrics represent itself a scaled value on a particular platform, we can directly use the amount of change to present the scaled contribution on this platform. Overall, with respect to Property 5.1.1 and Property 5.1.2, we have a few properties listed as follows:

**Property 5.1.3** *$\boldsymbol{F_{gm}}$ is the most adaptive metric applying to a data set $\boldsymbol{C}(a, p, h)$ retrieved by a metric indicating the absolute performance since in this scenario the percentage of change for $\boldsymbol{F_{gm}}$ is platform-independent.*

**Property 5.1.4** *$\boldsymbol{F_{am}}$ is the most adaptive metric applying to a data set $\boldsymbol{C}(a, p, h)$ retrieved by a metric indicating the relative performance since in this scenario the percentage of change for $\boldsymbol{F_{am}}$ is platform-independent.*

**Property 5.1.5** *$\boldsymbol{F_{hm}}$ is the least adaptive metric in $\boldsymbol{F}$ to evaluate performance portability since the percentage of change for $\boldsymbol{F_{hm}}$ is not platform-independent in all cases and in turn fails to meet the requirement described in Definition 5.1.17.*

#### 5.1.4.2 Strict and Relaxed Performance Portability

One of key features of $\mathcal{P}$ is that it highlights the property of portability which can be expressed as a boolean value. In this way, if an application is not supported by all platforms in $H$, then the metric value will equal to zero. Similarly, $\boldsymbol{F_{gm}}$ and $\boldsymbol{F_{hm}}$ applying to a data set $\boldsymbol{C}(a, p, h)$ retrieved by a performance metric also keep this feature. We call the performance portability maintaining this feature a strict performance portability, which can be defined as:

**Definition 5.1.18** *Strict performance portability (SPP) inherits the entire feature of portability saying that for an application $a$ solving a problem $p$, if this application is not supported by all platforms in the platform set $H$, the estimated value will be zero.*

However, strict performance portability addresses significantly the state of portability at the cost of lack of expressiveness for the metric to indicate which application can hit the highest score value in terms of performance portability when a large number of applications may not be supported by all platforms in $H$.

For example, in Table 5.2, if we calculate the strict performance portability of a set of applications $\{a_1, a_2, a_3, a_4\}$ solving a problem across a platform set $H = \{h_1, h_2, h_3, h_4, h_5, h_6\}$, the estimated values of all applications will be zero since each application is not supported by all platforms in $H$. In fact, we can't retrieve non-zero values of $\{a_1, a_2, a_3\}$ and $\{a_4\}$ at the same time. Moreover, in order to obtain non-zero values of $\{a_1, a_2, a_3\}$, the maximal platform set is limited to $H = \{h_3, h_5\}$. Such behavior is impractical and requires users to cautiously choose the platform set so as to attain comparable results from metrics of performance portability. After all, it is meaningless that most of applications gain the same score, zero. In pursuance of exploiting complete information, it is obligated to apply the metric of strict performance portability to a variety of sub-sets of platforms which makes the procedure of evaluation redundant and complicated.

Table 5.2: Portability of applications $\{a_1, a_2, a_3, a_4\}$ solving a given problem across a set of platforms $\{h_1, h_2, h_3, h_4, h_5, h_6\}$.

| Applications | Platforms | | | | | |
|---|---|---|---|---|---|---|
| | $h_1$ | $h_2$ | $h_3$ | $h_4$ | $h_5$ | $h_6$ |
| $a_1$ | 1 | | 1 | 1 | 1 | |
| $a_2$ | | 1 | 1 | 1 | 1 | |
| $a_3$ | 1 | 1 | 1 | | 1 | |
| $a_4$ | | | | | | 1 |

As an alternative, we propose a new type of performance portability which is called as relaxed performance portability. The definition is given as follow:

**Definition 5.1.19** *Relaxed performance portability (RPP) limits the boolean feature of portability and addresses more about performance, which means that for an application $a$ solving a problem $p$, the estimated value will be zero only if this application is blocked by all platforms in the platform set $H$.*

With respect to Definition 5.1.18 and Definition 5.1.19, we know that $\boldsymbol{F_{am}}$ is a natural metric calculating relaxed performance portability while $\boldsymbol{F_{gm}}$ and $\boldsymbol{F_{hm}}$ are metrics aggregating strict performance portability. We need to redesign $\boldsymbol{F_{gm}}$ and $\boldsymbol{F_{hm}}$ in order to make them capable of characterizing relaxed performance portability. But for convenience, we make modifications of $\boldsymbol{F_{gm}}$ and $\boldsymbol{F_{hm}}$ as well as $\boldsymbol{F_{am}}$.

#### 5.1.4.3 Variants of $F$

Generally, the criterion for design of variants of $\boldsymbol{F}$ can be summarized as follows:

1. Quantify both the performance and portability.

2. The metric equals to zero when all platforms in the set $H$ do not support $(a, p)$.

3. With the increase of non-supported platforms for an application, the metric value degrades.

4. Not to change the nature of $\boldsymbol{F_{am}}$ and $\boldsymbol{F_{gm}}$ as described in Property 5.1.1 and Property 5.1.2.

Let $S$ and $N$ the sub-sets of a platform set $H$ with $S \cup N = H$. All platforms in $S$ support an application $a$ solving a problem $p$ while all platforms in $N$ don't support the couple $(a, p)$. According to the criterion, we may firstly calculate $\boldsymbol{F} \cdot \boldsymbol{C}(a, p, h)_{h \in S}$, denoted as $F'$, the estimated value of $\boldsymbol{F}$ applying to a data set $\boldsymbol{C}(a, p, h)$ where each $h$ is in the platform sub-set $S$ and then make use of this value in variants of $\boldsymbol{F}$.

Here is our proposition of a transfer operator $\boldsymbol{T_1}$ to get the first variant of $\boldsymbol{F}$:

$$(\boldsymbol{T_1} \circ \boldsymbol{F}) \cdot \boldsymbol{C}(a, p, h) = \frac{|S|}{|H|} F' \tag{5.28}$$

where |S| is the total number of supported platforms and |H| is the total number of platforms. Based on Equation 5.20, we have $\boldsymbol{T_1} \circ \boldsymbol{F_{am}} = \boldsymbol{F_{am}}$.

$\boldsymbol{T_1} \circ \boldsymbol{F}$ directly use the estimated value of $\boldsymbol{F}$ applying to the data set retrieved from all supported platforms in the platform set $H$ and add a multiplier factor, the ratio of total number of supported platforms $S$ to the total number of platforms $H$, to weight it. Alternatively, we can also treat $F'$ as a part of terms for all elements in $\boldsymbol{C}(a, p, h)$ which are retrieved from non-supported platforms and we have:

$$\begin{aligned} \boldsymbol{T_2} \circ \boldsymbol{C}(a, p, h) &= \{C_i^{'}(a, p, h)\}_{\forall i=1,2,\ldots,|H|} \\ C_i^{'}(a, p, h) &= \begin{cases} f(F') & \text{if } C_i(a, p, h) = 0 \\ C_i(a, p, h) & \text{if } C_i(a, p, h) > 0 \end{cases} \end{aligned} \tag{5.29}$$

where $\boldsymbol{T_2}$ is the second transfer operator applying to the data set $\boldsymbol{C}(a, p, h)$. $f$ is a function parameterized by $F'$ with a positive value as output. $\boldsymbol{F_{am}}$, $\boldsymbol{F_{gm}}$ and $\boldsymbol{F_{hm}}$ have its corresponding function $f$ to meet the third criteria proposed above, saying that with the increase of non-supported platforms for an application, the metric value decreases. Thus, we can calculate their critical factors $k_{am}$, $k_{gm}$, $k_{hm}$ to learn about the maximal and minimal values of $f$.

Let $S$ and $N$ the sub-sets of a platform set $H$ containing respectively the supported and non-supported platforms in $H$ for a couple $(a, p)$, $K_1 = \frac{\sum_{i=0}^{|S|} C_i(a,p,h)}{|S|}$, $K_2 = \prod_{i=1}^{|S|} C_i(a, p, h)^{\frac{1}{|S|}}$, $K_3 = \frac{|S|}{\sum_{i=0}^{|S|} \frac{1}{C_i(a,p,h)}}$. The criticality equations of $\boldsymbol{F_{am}}$, $\boldsymbol{F_{gm}}$ and $\boldsymbol{F_{hm}}$ are described in Equation 5.30:

$$\begin{aligned}
\frac{\sum_{i=0}^{|S|} C_i(a,p,h) + |N|k_{am}}{|S|+|N|} &= K_1 \\
\prod_{i=1}^{|S|} C_i(a,p,h)^{\frac{1}{|S|+|N|}} \prod_{i=1}^{|N|} k_{gm}^{\frac{1}{|S|+|N|}} &= K_2 \\
\frac{|S|+|N|}{\sum_{i=0}^{|S|} \frac{1}{C_i(a,p,h)} + |N|k_{hm}} &= K_3
\end{aligned} \tag{5.30}$$

And we have:

$$\begin{aligned}
k_{am} &= F'_{am} \\
k_{gm} &= F'_{gm} \\
k_{hm} &= \frac{1}{F'_{hm}}
\end{aligned} \tag{5.31}$$

- if $f(F'_{am}) < k_{am}$: $\boldsymbol{F_{am}} \cdot (\boldsymbol{T_2} \circ \boldsymbol{C}(a,p,h))$ decreases with the increase of non-supported platforms for the couple $(a,p)$.
- if $f(F'_{gm}) < k_{gm}$: as above.
- if $f(F'_{hm}) > k_{hm}$: as above.

With the intention of highlighting the effect of the increase of non-supported platforms on $\boldsymbol{F}$, we choose $f(F'_{am}) = \frac{k_{am}}{|N|}$, $f(F'_{gm}) = \frac{k_{gm}}{|N|}$, $f(F'_{hm}) = \frac{|N|}{k_{hm}}$ for numerical evaluations afterward.

At last, we propose the third transfer operator along with the third variant of $\boldsymbol{F}$ described as follow:

$$(\boldsymbol{T_3} \circ \boldsymbol{F}) \cdot \boldsymbol{C}(a,p,h) = \texttt{Integer}[|S|].\text{Decimal}[\widetilde{F'}] \tag{5.32}$$

where $\texttt{Integer}[|S|]$ means that we use $|S|$ as the integer part of metric and $\texttt{Decimal}[\widetilde{F'}]$ as the decimal part of metric. $\widetilde{F'}$ is the normalized value of $F'$ with $\widetilde{F'} = \frac{F'}{F'_{max}}$. $F'_{max}$ is the maximal $F'$ obtained among a set of applications. Note that if $\widetilde{F'}$ equals to 100%, the decimal part will be noted as $0.\bar{9}$.

$\boldsymbol{T_3} \circ \boldsymbol{F}$ is a more intuitive formula which addresses more portability comparing to the other ones. It allows numerical comparison of performance portability among a set of supported/non-supported platforms where the performances of different applications are only compared when these applications have the same number of supported platforms.

Overall, the variants of $\boldsymbol{F}$ that we adopt enable the numerical comparison of performance portability among applications across different architectures when not all applications are supported by all architectures. This feature is more adaptive under the circumstances that the supported platforms for each application in an application set differentiates significantly among each other, resulting in small application and platform sets.

### 5.1.5 Metrics of Productivity Portability

As the definition of performance portability, we give the definition of productivity portability:

**Definition 5.1.20** *Productivity portability is a measurement adopting various aggregating methods which estimates the average productivity of an application $a$ solving a given problem $p$ across a platform set $H$ with the intention of comparison.*

In this way, $\boldsymbol{F}$ becomes the generic metric of productivity portability when it applies to a data set retrieved by a productivity metric.

#### 5.1.5.1 Metrics of Implicit Productivity Portability

When $\boldsymbol{F}$ applies to implicit productivity metrics, it characterizes the average development effort for a couple $(a, p)$ across a set of platforms. The metrics that indicate platform-specific code paths of an application should be adopted, such as PS-LOC. Since there is no meaning to aggregate the PS-LOC of a non-supported platform, the platform set should be limited to a sub-set of platforms supporting this application.

The adaptability of $\boldsymbol{F}$ towards metrics of implicit productivity portability (IPP) is unlike $\boldsymbol{F}$ towards performance portability metrics. Since metrics such as LOC, PS-LOC is not scaled on each platform, we are unable to choose the aggregating method (arithmetic, geometric, harmonic means) with reference to the platform-independent percentage of change. By contrast, we give all aggregating methods a chance to be examined.

Moreover, the metric of pairwise distances $d_{i,j}(a, p, h)$ may also be applied to $\boldsymbol{F}$ and we obtain the code divergence of the application of different platform-specific code paths as described in Equation 5.33 based on Equation 5.22. The number of combinations for taking $r$ things out of $n$ things is denoted as $Comb(n, r)$.

$$\boldsymbol{F} \cdot \boldsymbol{C}(a,p,h) = \begin{cases} \boldsymbol{F_{am}} \cdot \boldsymbol{C}(a,p,h) & = & \dfrac{\sum_{\{i,j\} \subset H} d_{i,j}\,(a,p,h)}{Comb(|H|,2)} \\ \\ \boldsymbol{F_{gm}} \cdot \boldsymbol{C}(a,p,h) & = & \prod_{\{i,j\} \subset H} d_{i,j}\,(a,p,h)^{\dfrac{1}{Comb(|H|,2)}} \\ \\ \boldsymbol{F_{hm}} \cdot \boldsymbol{C}(a,p,h) & = & \dfrac{Comb(|H|,2)}{\sum_{\{i,j\} \subset H} \dfrac{1}{d_{i,j}\,(a,p,h)}} \end{cases} \tag{5.33}$$

Similarly, according to Equation 5.23, we may apply $\boldsymbol{F}$ towards the metric of churn $chn_{i,j}(a, p, h)$, as described in Equation 5.34 where $chn_{i_0,i}$ refers to the churn distance between the reference code path $i_0$ and another code path $i$ for an implementation. This value expresses the average addition and removals of lines required for an implementation of a given coda path to be ported to another platform in a given platform set.

$$\boldsymbol{F} \cdot \boldsymbol{C}(a,p,h) = \begin{cases} \boldsymbol{F_{am}} \cdot \boldsymbol{C}(a,p,h) & = & \dfrac{\sum_{i=1}^{|H|-1} chn_{i_0,i}\,(a,p,h)}{|H|-1} \\ \boldsymbol{F_{gm}} \cdot \boldsymbol{C}(a,p,h) & = & \prod_{i=1}^{|H|-1} chn_{i_0,i}\,(a,p,h)^{\frac{1}{|H|-1}} \\ \boldsymbol{F_{hm}} \cdot \boldsymbol{C}(a,p,h) & = & \dfrac{|H|-1}{\sum_{i=1}^{|H|-1} \dfrac{1}{chn_{i_0,i}\,(a,p,h)}} \end{cases} \tag{5.34}$$

#### 5.1.5.2 Metrics of Explicit Productivity Portability

When $\boldsymbol{F}$ applies to explicit productivity metrics, it actually reflects the productive performance portability which represents the average performance improvement per program utility unit across a platform set. In order to obtain meaningful relative speedup or effort, it is required that we have to at least find an application supported by all platforms in the platform set and treat it as the reference implementation. Otherwise we may use different applications as reference on distinct platforms and the aggregation of results will be quite meaningless. Based on this limitation, the variants of $\boldsymbol{F}$ addressing relaxed performance portability cannot be directly used to capture productive performance portability. We have to find the application supported by a maximal number of platforms in the platform set and perform the evaluation of productive performance portability simply across the new sub-set of platforms.

The adaptability of $\boldsymbol{F}$ towards metrics of explicit productivity portability (EPP), also called as productive performance portability, is similar to that of performance portability. We should maintain the feature of platform-independent percentage of change. Since our productive performance portability metrics adopt speedup as performance metric, according to Property 5.1.1, $\boldsymbol{F_{am}}$ is the most adaptive metric to aggregate productive performance portability of an application solving a problem across a set of platforms.

## 5.2 Evaluation of Benchmark

The benchmark slabAllNuclides is used to perform further evaluations. SIGMA1 on-the-fly Doppler broadening approach and pseudo event-based method are chosen along with the input parameters fixed to $5 \times 10^5$ particles and $100$ bank size. Ten platforms are composed from five machines introduced in the previous chapter in the consideration of single CPU or CPU + 1GPU.

Portability of six versions of slabAllNuclides implemented via programming models of OpenMP thread, CUDA, OpenACC, OpenMP offload, Kokkos as well as SYCL across ten platforms is shown in Table 5.3.

Note that green cells refer to applications portable to the corresponding platform while gray cells represent the non-portability of applications towards the related platform. We find that the OpenMP version is only targeted to CPU platforms and it is non-portable to Nvidia GPUs and Intel GPU. The CUDA version is limited to be executed on heterogeneous architectures of CPU + Nvidia GPU. Both the OpenACC and Kokkos implementations are portable

to nine of ten platforms composed of Intel CPUs and Nvidia GPUs whereas the OpenMP offload implementation is only portable to the heterogeneous architecture of GridCL and Gorgon due to lack of compiler support on other platforms. The SYCL version is the single implementation portable to the architecture of Intel CPU+GPU, however, it is not supported by other platforms owing to lack of compilers to support the offload functionality.

Table 5.3: Portability of slabAllNuclides implemented via different programming models across a set of platforms.

| Platforms | | Programming Models | | | | | |
|---|---|---|---|---|---|---|---|
| | | OpenMP thread | CUDA | OpenACC | OpenMP offload | Kokkos | SYCL |
| GridCL | Skylake | 1 | 0 | 1 | 1 | 1 | 0 |
| | +1V100 | 0 | 1 | 1 | 1 | 1 | 0 |
| Cobalt-hybrid | Broadwell | 1 | 0 | 1 | 0 | 1 | 0 |
| | +1P100 | 0 | 1 | 1 | 0 | 1 | 0 |
| Cobalt-v100 | Skylake | 1 | 0 | 1 | 0 | 1 | 0 |
| | +1V100 | 0 | 1 | 1 | 0 | 1 | 0 |
| Gorgon | Power9 | 1 | 0 | 1 | 1 | 1 | 0 |
| | +1V100 | 0 | 1 | 1 | 1 | 1 | 0 |
| Intel NUC | Kaby Lake G | 1 | 0 | 1 | 0 | 1 | 1 |
| | +1Gen9.5 | 0 | 0 | 0 | 0 | 0 | 1 |

### 5.2.1 Performance Portability Evaluation

We use a metric of relative performance, application efficiency and a metric of absolute performance, particle tracking rate to perform the evaluation of performance portability with our generic model $\boldsymbol{F}$. The values of particle tracking rate are exactly those listed in Table 4.8. The data set of application efficiency are measured by dividing the performance (particle tracking rate) of an application on a given platform by the best performance achieved on this platform.

Table 5.4 shows all values of performance metrics that will be applied to $\boldsymbol{F}$ and its variants as the data set $\boldsymbol{C}(a, p, h)$. Based on these values, we can calculate the results of $\boldsymbol{F} \cdot \boldsymbol{C}(a, p, h)$ evaluating the performance portability of different implementations of slabAllNuclides which are illustrated in Table 5.5 and Table 5.6. Note that all blank cells represent that the implementation of a given programming model fails to run on a given platform. These blank cells are corresponding to the gray cells illustrated in Table 5.3 and will be considered as zero values for computation.

From Table 5.5 we can find that for the given platform set and application set, $\boldsymbol{F}_{am}$ is the single metric which renders meaningful results for comparison. By contrast, $\boldsymbol{F}_{gm}$ and $\boldsymbol{F}_{h}\boldsymbol{m}$ both obtain zero values for all implementations of benchmark. In order to make these two metrics capable of retrieving non-zero values, the groups of platform

Table 5.4: Metrics of performance of different implementations of slabAllNuclides across a set of platforms.

| Machines | #GPUs | Programming Models | | | | | |
|---|---|---|---|---|---|---|---|
| | | OpenMP thread | CUDA | OpenACC | OpenMP offload | Kokkos | SYCL |
| Particle Tracking Rate ($\times 10^2$ particles/s) | | | | | | | |
| GridCL | 0 | 16.9 | | 6.9 | 5.5 | 8.1 | |
| | 1 | | 51.8 | 34.7 | 2.6 | 0.7 | |
| Cobalt-hybrid | 0 | 9.5 | | 4.4 | | 5.9 | |
| | 1 | | 25.7 | 21.8 | | 2.4 | |
| Cobalt-v100 | 0 | 13.6 | | 6.4 | | 7.2 | |
| | 1 | | 55.7 | 21.8 | | 2.8 | |
| Gorgon | 0 | 6.8 | | 7.4 | 1.5 | 6.0 | |
| | 1 | | 55.0 | 36.6 | 6.5 | 1.2 | |
| Intel NUC | 0 | 3.1 | | 1.1 | | 2.2 | 0.1 |
| | 1 | | | | | | 0.6 |
| Application Efficiency (%) | | | | | | | |
| GridCL | 0 | 100 | | 41 | 33 | 48 | |
| | 1 | | 100 | 67 | 5 | 1 | |
| Cobalt-hybrid | 0 | 100 | | 46 | | 62 | |
| | 1 | | 100 | 85 | | 9 | |
| Cobalt-v100 | 0 | 100 | | 47 | | 53 | |
| | 1 | | 100 | 39 | | 5 | |
| Gorgon | 0 | 92 | | 100 | 20 | 81 | |
| | 1 | | 100 | 67 | 12 | 2 | |
| Intel NUC | 0 | 100 | | 35 | | 71 | 3 |
| | 1 | | | | | | 100 |

Particle tracking rate (particles/s), the higher the better; Application Efficiency (%), the higher the better.

and application should be reset to smaller ones where all applications in the application subset are supported by all platforms in the platform subset. Although we may draw graphs of performance portability varying over a sequence of subsets of platforms, as what Deakin has done in his paper [40], this requires a large amount of work for repeated calculations of performance portability over a range of architectures. As an alternative, we have used the variants of $\boldsymbol{F}$ to retrieve the values of performance portability.

All results retrieved by different aggregating methods are calculated as reference. According to Property 5.1.3, $\boldsymbol{F_{gm}}$ is the most adaptive metric applying to a data set $\boldsymbol{C}(a, p, h)$ of particle tracking rate since in this case the percentage of change for $\boldsymbol{F_{gm}}$ is platform-independent to meet the requirement of adaptability of performance portability expressed in Definition 5.1.17. Therefore, we mainly give a detailed analysis of results obtained by $\boldsymbol{F_{gm}}$ which are highlighted with green color.

With respect to the metric $\boldsymbol{T_1} \circ \boldsymbol{F_{gm}} \cdot \boldsymbol{C}(a, p, h)$, the highest value is obtained by the CUDA version, $18.0 \times 10^2$ particles/s, while the lowest value is retrieved by the SYCL version, $0.0$ particles/s by keeping one decimal place. The priorities of different programming models for better performance become CUDA $>$ OpenACC $>$ OpenMP thread $>$ Kokkos $>$ OpenMP offload $>$ SYCL.

Table 5.5: Evaluation of performance portability for different implementations of slabAllNuclides across a set of platforms via the generic model $\boldsymbol{F}$ and the absolute performance metric, particle tracking rate, the higher the better.

| Metrics | | Particle Tracking Rate ($\times 10^2$ particles/s) | | | | | |
|---|---|---|---|---|---|---|---|
| | | OpenMP thread | CUDA | OpenACC | OpenMP offload | Kokkos | SYCL |
| $\boldsymbol{F} \cdot \boldsymbol{C}(a,p,h)$ | $\boldsymbol{F_{am}}$ | 5.0 | 18.8 | 14.1 | 1.6 | 3.7 | 0.1 |
| | $\boldsymbol{F_{gm}}$ | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | $\boldsymbol{F_{hm}}$ | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| $\boldsymbol{F'} \cdot \boldsymbol{C}(a,p,h)$ | $\boldsymbol{F'_{am}}$ | 10.0 | 47.1 | 15.7 | 4.0 | 4.1 | 0.4 |
| | $\boldsymbol{F'_{gm}}$ | 8.6 | 44.9 | 9.9 | 3.4 | 3.1 | 0.2 |
| | $\boldsymbol{F'_{hm}}$ | 7.1 | 42.4 | 5.2 | 2.9 | 2.2 | 0.2 |
| $\boldsymbol{T_1} \circ \boldsymbol{F} \cdot \boldsymbol{C}(a,p,h)$ | $\boldsymbol{F_{am}}$ | 5.0 | 18.8 | 14.1 | 1.6 | 3.7 | 0.1 |
| | $\boldsymbol{F_{gm}}$ | 4.3 | 18.0 | 9.0 | 1.4 | 2.8 | 0.0 |
| | $\boldsymbol{F_{hm}}$ | 3.5 | 17.0 | 4.7 | 1.2 | 2.0 | 0.0 |
| $\boldsymbol{F} \cdot (\boldsymbol{T_2} \circ \boldsymbol{C}(a,p,h))$ | $\boldsymbol{F_{am}}$ | 6.0 | 23.5 | 15.7 | 2.0 | 4.1 | 0.1 |
| | $\boldsymbol{F_{gm}}$ | 3.8 | 15.3 | 9.9 | 1.2 | 3.1 | 0.0 |
| | $\boldsymbol{F_{hm}}$ | 2.4 | 10.6 | 5.2 | 0.7 | 2.2 | 0.0 |
| $\boldsymbol{T_3} \circ \boldsymbol{F} \cdot \boldsymbol{C}(a,p,h)$ | $\boldsymbol{F_{am}}$ | 5.21 | 4.99 | 9.33 | 4.09 | 9.09 | 2.01 |
| | $\boldsymbol{F_{gm}}$ | 5.19 | 4.99 | 9.22 | 4.08 | 9.07 | 2.01 |
| | $\boldsymbol{F_{hm}}$ | 5.17 | 4.99 | 9.12 | 4.07 | 9.05 | 2.00 |

where $\boldsymbol{T_1}$, $\boldsymbol{T_2}$ and $\boldsymbol{T_3}$ are three transfer operators defined in section 5.1.4.3 to obtain variants of $\boldsymbol{F}$. Note that the value of $\boldsymbol{T_3} \circ \boldsymbol{F} \cdot \boldsymbol{C}(a,p,h)$ is a score which has no unit.

With respect to the metric $\boldsymbol{F_{gm}} \cdot (\boldsymbol{T_2} \circ \boldsymbol{C}(a,p,h))$, the priority order is exactly the same as the metric $\boldsymbol{T_1} \circ \boldsymbol{F_{gm}} \cdot \boldsymbol{C}(a,p,h)$. The CUDA version gains the best result which surpasses the OpenACC version with a factor of 1.5x speedup.

With respect to the metric $\boldsymbol{T_3} \circ \boldsymbol{F} \cdot \boldsymbol{C}(a,p,h)$, the highest score is retrieved by the OpenACC version while the lowest score is obtained by the SYCL version. The OpenACC and Kokkos versions benefit from the largest integer part which represents the number of supported platforms for a given programming model. As for the comparison between the OpenACC and Kokkos version, the decimal part of the OpenACC version is larger than that of the Kokkos version, which means that the normalized aggregating result of the OpenACC version surpasses the one of the Kokkos version. Overall, this variant of $\boldsymbol{F_{gm}}$ renders a priority order as OpenACC $>$ Kokkos $>$ OpenMP thread $>$ CUDA $>$ OpenMP offload $>$ SYCL.

Now we consider the evaluation of performance portability for different implementations of slabAllNuclides via the generic model $\boldsymbol{F}$ and the relative performance metric, application efficiency, as illustrated in Table 5.6. Similarly, the values highlighted with green color are gained by the utilization of $\boldsymbol{F_{am}}$ which is the most adaptive metric for the evaluation of application efficiency according to Property 5.1.4. Since in this scenario the percentage of change

for $F_{am}$ is platform-independent to meet the requirement of adaptability of performance portability expressed in Definition 5.1.17.

$F_{am}$ manages to get meaningful values and the results shows that the priority order is OpenACC $>$ OpenMP thread $>$ CUDA $>$ Kokkos $>$ SYCL $>$ OpenMP offload where the OpenACC version gains the best application efficiency 53% and the OpenMP offload version renders the worst, 7%. $F_{gm}$ and $F_{hm}$ are still unable to evaluate the strict performance portability for the given application set and platform set in our case. All values expressing the relaxed performance portability are calculated by the variants of $F$ as reference.

Table 5.6: Evaluation of performance portability for different implementations of slabAllNuclides across a set of platforms via the generic model $F$ and the relative performance metric, application efficiency, the higher the better.

| Metrics | | Application Efficiency (%) | | | | | |
|---|---|---|---|---|---|---|---|
| | | OpenMP thread | CUDA | OpenACC | OpenMP offload | Kokkos | SYCL |
| $F \cdot C(a,p,h)$ | $F_{am}$ | 49 | 40 | 53 | 7 | 33 | 10 |
| | $F_{gm}$ | 0 | 0 | 0 | 0 | 0 | 0 |
| | $F_{hm}$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $F' \cdot C(a,p,h)$ | $F'_{am}$ | 98 | 100 | 59 | 18 | 37 | 52 |
| | $F'_{gm}$ | 98 | 100 | 55 | 14 | 16 | 17 |
| | $F'_{hm}$ | 98 | 100 | 52 | 11 | 5 | 6 |
| $T_1 \circ F \cdot C(a,p,h)$ | $F_{am}$ | 49 | 40 | 53 | 7 | 33 | 10 |
| | $F_{gm}$ | 49 | 40 | 50 | 6 | 15 | 3 |
| | $F_{hm}$ | 49 | 40 | 47 | 4 | 4 | 1 |
| $F \cdot (T_2 \circ C(a,p,h))$ | $F_{am}$ | 59 | 50 | 59 | 9 | 37 | 15 |
| | $F_{gm}$ | 44 | 34 | 55 | 5 | 16 | 3 |
| | $F_{hm}$ | 33 | 25 | 52 | 3 | 5 | 1 |
| $T_3 \circ F \cdot C(a,p,h)$ | $F_{am}$ | 5.98 | 4.99 | 9.59 | 4.18 | 9.37 | 2.52 |
| | $F_{gm}$ | 5.98 | 4.99 | 9.55 | 4.14 | 9.16 | 2.17 |
| | $F_{hm}$ | 5.98 | 4.99 | 9.52 | 4.11 | 9.05 | 2.06 |

where $T_1$, $T_2$ and $T_3$ are three transfer operators defined in section 5.1.4.3 to obtain variants of $F$. Note that the value of $T_3 \circ F \cdot C(a,p,h)$ is a score which has no unit.

Similarly, we explain in detail the results retrieved by the variants of $F_{am}$. With reference to the metric $T_1 \circ F_{gm} \cdot C(a,p,h)$, the results are equal to those obtained by the original metric $F_{am}$, which corresponds to the equation $T_1 \circ F_{am} = F_{am}$. With reference to the metric $F_{gm} \cdot (T_2 \circ C(a,p,h))$, the results shows that the OpenMP thread and OpenACC versions gain the highest value, 59% while the OpenMP offload version obtains the lowest one, 9%. The priorities of different programming models for better application efficiency become OpenMP thread $=$ OpenACC $>$ CUDA $>$ Kokkos $>$ SYCL $>$ OpenMP offload. With reference to the metric $T_3 \circ F \cdot C(a,p,h)$, the priority order resembles in the one obtained in Table 5.5, which is OpenACC $>$ Kokkos $>$ OpenMP thread $>$

CUDA $>$ OpenMP offload $>$ SYCL.

In conclusion, the results listed in Table 5.5 and Table 5.6 prove that $\boldsymbol{F_{hm}} \leqslant \boldsymbol{F_{gm}} \leqslant \boldsymbol{F_{am}}$. For example, the values retrieved by the variant $\boldsymbol{T_1} \circ \boldsymbol{F_{hm}}$, $\boldsymbol{T_1} \circ \boldsymbol{F_{gm}}$, $\boldsymbol{T_1} \circ \boldsymbol{F_{am}}$ for the CUDA version in Table 5.5 increase from $17.0 \times 10^2$ particles/s to $18.8 \times 10^2$ particles/s. Excluding the variant of $\boldsymbol{F}$ transferred by the operator $\boldsymbol{T_3}$ which maintains its own scoring mechanism, the CUDA version aggregates the highest value of particle tracking rate and the OpenACC version gains the highest application efficiency when they are evaluated by the metric $\boldsymbol{F}$ as well as its variants with the utilization of two transfer operators $\boldsymbol{T_1}$ and $\boldsymbol{T_2}$. As for the metric $\boldsymbol{T_3} \circ \boldsymbol{F}$, the OpenACC version obtains the best performance in terms of particle tracking rate and application efficiency, since this variant addresses much more portability than performance, which makes the OpenACC and Kokkos versions the most competitive implementations of slabAllNuclides running across the given ten platforms in terms of performance portability.

### 5.2.2 Productivity Portability Evaluation

#### 5.2.2.1 Implicit Productivity Portability

Table 5.7 shows a variety of metrics of implicit productivity for implementations of slabAllNuclides across a set of platforms. The measured source files include the partial offloading part to transfer nuclide data to device and calculate microscopic cross sections on device.

With respect to LOC, the results are measured by using the command line `wc -l`. The CUDA version requires the largest number of LOC whereas the OpenMP thread version the smallest. The PS-LOC of each implementation for Intel x86-based, IBM Power-based, Intel GPU-based and Nvidia GPU-based platforms are same and thus the corresponding code distances $d_{i,j}$ between two of four types of platforms for all implementations are equal to zero. It indicates that the OpenACC, OpenMP offload, Kokkos and SYCL versions are of high abstraction which leads to little difference between distinct platform-specific code paths. The churns of different code paths for each implementation reflect that the OpenMP offload version share a single code path for both Intel x86-based, IBM Power-based and Nvidia GPU-based platforms and the largest difference between code paths is introduced by the Kokkos version where it is obliged to define the Kokkos execution space during the compilation. The differences of code paths for each implementation are summarized as follows:

- OpenMP thread: none.
- CUDA: none.
- OpenACC: set `acc_device_host` or `acc_device_nvidia`.
- OpenMP offload: `nb_devices = omp_get_num_devices()`, the program automatically decide whether it is executed on host or device according to `nb_devices`.

- Kokkos: set `Kokkos::CudaSpace` or `Kokkos::HostSpace` during the compilation.
- SYCL: set `sycl::cpu_selector{}` or `sycl::gpu_selector{}`.

Table 5.7: Metrics of implicit productivity for implementations of slabAllNuclides across a set of platforms.

| Metrics | Programming Models | | | | | |
|---|---|---|---|---|---|---|
| | OpenMP thread | CUDA | OpenACC | OpenMP offload | Kokkos | SYCL |
| Lines of code (LOC) | | | | | | |
| Total | 567 | 1517 | 711 | 699 | 705 | 832 |
| $h_1$ | 567 | - | 710 | 699 | 697 | 831 |
| $h_2$ | 567 | - | 710 | 699 | 697 | - |
| $h_3$ | - | 1517 | 710 | 699 | 697 | - |
| $h_4$ | - | - | - | - | - | 831 |
| Code distance | | | | | | |
| $d_{1,2}$ | 0 | - | 0 | - | 0 | - |
| $d_{1,3}$ | - | - | 0 | - | 0 | - |
| $d_{1,4}$ | - | - | - | - | - | 0 |
| $d_{2,3}$ | - | - | 0 | 0 | 0 | - |
| $d_{2,4}$ | - | - | - | - | - | - |
| $d_{3,4}$ | - | - | - | - | - | - |
| Churn | | | | | | |
| $chn_{1,2}$ | 0 | - | 0 | 0 | 0 | - |
| $chn_{1,3}$ | - | - | 2 | 0 | 16 | - |
| $chn_{1,4}$ | - | - | - | - | - | 2 |
| $chn_{2,3}$ | - | - | 2 | 0 | 16 | - |
| $chn_{2,4}$ | - | - | - | - | - | - |
| $chn_{3,4}$ | - | - | - | - | - | - |
| Halstead complexity metric | | | | | | |
| Program length | 2255 | 5570 | 2315 | 2698 | 2790 | 3816 |
| Program volume | 201912 | 59078 | 22211 | 25870 | 26866 | 36957 |
| Program difficulty | 11.6 | 11.4 | 11.8 | 11.4 | 11.3 | 11.5 |
| Program effort | 242706 | 675996 | 262814 | 295612 | 303432 | 425327 |

$h_1$ represents an Intel x86 architecture, $h_2$ an IBM Power architecture, $h_3$ a Nvidia GPU platform, $h_4$ an Intel GPU architecture. $d_{i,j}$ and $chn_{i,j}$ refer to individually the code distance and churn between the code paths of $h_i$ and $h_j$. If $i = j$, $d_{i,i} = 0$, $chn_{i,i} = 0$.

With respect to the Halstead complexity metrics, all operators and operands are counted manually with the use of `vimgrep` to search operator patterns. We find that the program difficulty of all implementations are at the same level. This is reasonable since all implementations adopt the same algorithms and they are coded in a single style. The program length, volume and effort of the CUDA version is the highest ones comparing to other implementations which means that the CUDA version is the most complicate implementation requiring much development effort. Moreover, it should be noted that the SYCL version differs quite a lot from the OpenMP thread, OpenACC, OpenMP offload and Kokkos versions in terms of program length/volume/effort by cause of its cumbersome way of declaring

variables in private memory and its user-defined function of sum reduction.

Table 5.8: Implicit productivity portability evaluated by $F$ for implementations of slabAllNuclides across a set of supported platforms.

| Metrics | Programming Models | | | | | |
|---|---|---|---|---|---|---|
| | OpenMP thread | CUDA | OpenACC | OpenMP offload | Kokkos | SYCL |
| Number of supported platforms $\|S\|$ & $\|H\| = 10$ | | | | | | |
| | 5 | 4 | 9 | 4 | 9 | 2 |
| Number of combinations of supported platforms | | | | | | |
| | 10 | 6 | 36 | 6 | 36 | 1 |
| Platform-specific lines of code (PS-LOC) | | | | | | |
| $F_{am}$ | 567 | 1517 | 710 | 699 | 697 | 831 |
| $F_{gm}$ | 567 | 1517 | 710 | 699 | 697 | 831 |
| $F_{hm}$ | 567 | 1517 | 710 | 699 | 697 | 831 |
| Code divergence | | | | | | |
| $F_{am}$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $F_{gm}$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $F_{hm}$ | 0 | 0 | 0 | 0 | 0 | 0 |
| Churn | | | | | | |
| $F_{am}$ | 0 | 0 | 1 | 0 | 8 | 0.25 |
| $F_{gm}$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $F_{hm}$ | 0 | 0 | 0 | 0 | 0 | 0 |

Now we may apply the metric $F$ towards the data sets of implicit productivity listed above with the intention of evaluating the implicit productivity portability for implementations of slabAllNuclides across a set of supported platforms. From Table 5.8 we find that the estimated values of PS-LOC for each implementation retrieved by $F_{am}$, $F_{gm}$, $F_{hm}$ are equal since the platform-specific code distances between different code paths of each implementation are equaling to zero. We can only observe several differences from results of the metric churn. For instance, with reference to the Kokkos version, the estimated value of platform-specific churn aggregated by $F_{am}$ is 8, meaning that given a code path as reference, it requires on average 8 lines of modification to make the Kokkos version successfully running on all supported platforms. It should be noted that $F_{gm}$ and $F_{hm}$ always return zeros if any value in its data set is equal to zero. Thus, both of them are not qualified to capture the feature of churns for each implementation of different code paths.

In general, for implementations which are portable to multiple architectures including the OpenACC, OpenMP offload, Kokkos, SYCL versions, their code divergences and churns are all at a low level, indicating that their maintenance effort is rather little. However, because the code paths of each implementation targeting to different platforms are similar to each other, the retrieved data sets are not suitable to examine different sub-types of $F$. We can only say that in our scenario, $F_{am}$ is the single metric capable of evaluating the churns of several implementations.

#### 5.2.2.2 Explicit Productivity Portability

Table 5.9: Metrics of explicit productivity for implementations of slabAllNuclides across a set of platforms.

| Machines | #GPUs | Programming Models | | | | | |
|---|---|---|---|---|---|---|---|
| | | OpenMP thread | CUDA | OpenACC | OpenMP offload | Kokkos | SYCL |
| | | | | Relative Speedup (x) | | | |
| GridCL | 0 | 2.45 | | 1.00 | 0.80 | 1.17 | |
| | 1 | | 1.49 | 1.00 | 0.07 | 0.02 | |
| Cobalt-hybrid | 0 | 2.16 | | 1.00 | | 1.34 | |
| | 1 | | 1.18 | 1.00 | | 0.11 | |
| Cobalt-v100 | 0 | 2.13 | | 1.00 | | 1.13 | |
| | 1 | | 2.56 | 1.00 | | 0.13 | |
| Gorgon | 0 | 0.92 | | 1.00 | 0.20 | 0.81 | |
| | 1 | | 1.50 | 1.00 | 0.18 | 0.03 | |
| Intel NUC | 0 | 2.82 | | 1.00 | | 2.00 | 0.09 |
| | | | | Relative PS-LOC (x) | | | |
| GridCL | 0 | 0.80 | | 1.00 | 0.98 | 0.98 | 1.17 |
| | 1 | | 2.14 | 1.00 | 0.98 | 0.98 | |
| Cobalt-hybrid | 0 | 0.80 | | 1.00 | 0.98 | 0.98 | 1.17 |
| | 1 | | 2.14 | 1.00 | 0.98 | 0.98 | |
| Cobalt-v100 | 0 | 0.80 | | 1.00 | 0.98 | 0.98 | 1.17 |
| | 1 | | 2.14 | 1.00 | 0.98 | 0.98 | |
| Gorgon | 0 | 0.80 | | 1.00 | 0.98 | 0.98 | |
| | 1 | | 2.14 | 1.00 | 0.98 | 0.98 | |
| Intel NUC | 0 | 0.80 | | 1.00 | 0.98 | 0.98 | 1.17 |
| | | | | $\Psi_{\text{relative}}$ | | | |
| GridCL | 0 | 3.06 | | 1.00 | 0.82 | 1.19 | |
| | 1 | | 0.70 | 1.00 | 0.07 | 0.02 | |
| Cobalt-hybrid | 0 | 2.70 | | 1.00 | | 1.37 | |
| | 1 | | 0.55 | 1.00 | | 0.11 | |
| Cobalt-v100 | 0 | 2.66 | | 1.00 | | 1.15 | |
| | 1 | | 1.20 | 1.00 | | 0.13 | |
| Gorgon | 0 | 1.15 | | 1.00 | 0.20 | 0.83 | |
| | 1 | | 0.70 | 1.00 | 0.18 | 0.03 | |
| Intel NUC | 0 | 3.52 | | 1.00 | | 2.04 | 0.08 |

Considering the evaluation of explicit productivity portability (EPP or productive performance portability), we first need to choose a subset of the original platform set $H$ so as to make at least one application in the application set capable of running across the platforms in the subset of $H$. Fortunately, we may exclude the platform of the machine Intel NUC associated with an Intel GPU and reset the platform set to $H'$ with $|H'| = 9$. In this way, both the OpenACC and Kokkos versions are good candidates for reference and the relative speedup and implicit productivity (PS-LOC) of different programming models are illustrated in Table 5.9 as we use the results of the OpenACC version as baseline values.

Based on the values of $\Psi_{\text{relative}}$ listed above, we can use the variants of $\boldsymbol{F}$ to evaluate the productive performance portability of different implementations of slabAllNuclides across the platform set $H'$. Note that the blank cells in Table 5.9 signify the meaninglessness and cannot be replaced by zero values. Thus, the original metric of $\boldsymbol{F_{am}}$ is also unable to aggregate meaningful results.

From Table 5.10, we find that $\boldsymbol{F_{am}}$, $\boldsymbol{F_{gm}}$ and $\boldsymbol{F_{hm}}$ retrieve the same priority order. According to Property 5.1.4, $\boldsymbol{F_{am}}$ is the most adaptive metric for the evaluation of relative value, so we give an explication of results obtained by $\boldsymbol{F_{am}}$ which are highlighted with green color.

Table 5.10: Evaluation of productive performance portability for different implementations of slabAllNuclides across a subset of platforms via the generic model $\boldsymbol{F}$, $\Psi_{\text{relative}}$, the higher the better.

| Metrics | | $\Psi_{\text{relative}}$ | | | | | |
|---|---|---|---|---|---|---|---|
| | | OpenMP thread | CUDA | OpenACC | OpenMP offload | Kokkos | SYCL |
| $\boldsymbol{F'} \cdot \boldsymbol{C}(a,p,h)$ | $\boldsymbol{F'_{am}}$ | 2.618 | 0.788 | 1.000 | 0.318 | 0.763 | 0.080 |
| | $\boldsymbol{F'_{gm}}$ | 2.454 | 0.754 | 1.000 | 0.213 | 0.311 | 0.080 |
| | $\boldsymbol{F'_{hm}}$ | 2.245 | 0.726 | 1.000 | 0.153 | 0.086 | 0.080 |
| $\boldsymbol{T_1} \circ \boldsymbol{F} \cdot \boldsymbol{C}(a,p,h)$ | $\boldsymbol{F_{am}}$ | 1.454 | 0.350 | 1.000 | 0.141 | 0.763 | 0.009 |
| | $\boldsymbol{F_{gm}}$ | 1.363 | 0.335 | 1.000 | 0.095 | 0.311 | 0.009 |
| | $\boldsymbol{F_{hm}}$ | 1.247 | 0.323 | 1.000 | 0.068 | 0.086 | 0.009 |
| $\boldsymbol{F} \cdot (\boldsymbol{T_2} \circ \boldsymbol{C}(a,p,h))$ | $\boldsymbol{F_{am}}$ | 1.745 | 0.438 | 1.000 | 0.176 | 0.848 | 0.018 |
| | $\boldsymbol{F_{gm}}$ | 1.325 | 0.308 | 1.000 | 0.087 | 0.311 | 0.013 |
| | $\boldsymbol{F_{hm}}$ | 0.962 | 0.225 | 1.000 | 0.048 | 0.086 | 0.011 |
| $\boldsymbol{T_3} \circ \boldsymbol{F} \cdot \boldsymbol{C}(a,p,h)$ | $\boldsymbol{F_{am}}$ | 5.99 | 4.30 | 9.38 | 4.12 | 9.29 | 1.03 |
| | $\boldsymbol{F_{gm}}$ | 5.99 | 4.31 | 9.41 | 4.09 | 9.13 | 1.03 |
| | $\boldsymbol{F_{hm}}$ | 5.99 | 4.32 | 9.45 | 4.07 | 9.04 | 1.04 |

The variants of $\boldsymbol{F_{am}}$ transferred by $\boldsymbol{T_1}$ and $\boldsymbol{T_2}$ obtain the similar results. The priorities of different programming models for better productive performance portability are both OpenMP thread $>$ OpenACC $>$ Kokkos $>$ CUDA $>$ OpenMP offload $>$ SYCL where the highest values gained by the OpenMP thread version of two variants are respectively $1.454$ and $1.745$ and the lowest values retrieved by the SYCL version of two variants are respectively $0.009$ and $0.018$.

As for the metric $\boldsymbol{T_3} \circ \boldsymbol{F_{am}} \cdot \boldsymbol{C}(a,p,h)$, the OpenACC version renders the highest score, 5.99, while the SYCL version obtains the lowest score, 1.03. The priority order retrieved by this variant resembles in the one attained by the same variant addressing performance portability, which is OpenACC $>$ Kokkos $>$ OpenMP thread $>$ CUDA $>$ OpenMP offload $>$ SYCL.

Overall, excluding the metric $\boldsymbol{T_3} \circ \boldsymbol{F} \cdot \boldsymbol{C}(a,p,h)$ which addresses more portability than productive performance, the results shows that the OpenMP thread version obtains the best productive performance across a set of platforms

which makes it the best candidate for implementation in terms of performance, portability and productivity with the given platform set. The OpenACC and Kokkos versions are more competitive to the CUDA version in terms of 3P which are mainly due to their much wider range of portable platforms and much higher abstraction of code. The OpenMP offload version gains bad scores since it achieves bad performance and covers a narrow range of platforms in the platform set. Lastly, the SYCL obtains bad results however these values may be biased since this programming model is preliminary and its compiler support is underdeveloped. In order to retrieve unbiased estimates, we need carry out more tests across more architectures for the SYCL version with the use of the next version of SYCL compilers.

It should also be addressed that the conclusions drawn above depend on the applications, problems and platforms (environments included) that we use in the thesis [31][32]. For example, it is safe to say that the OpenMP thread version of our Monte Carlo neutron transport codes obtains the best productive performance across a set of platforms to solve a fixed source Monte Carlo problem using SIGMA1 on-the-fly Doppler broadening approach. However, if we change the applications, the problems, or the platforms, other conclusions may be reached instead. Hence rather than giving a guide for users who want to make a choice of programming model and environment for their applications with the use of our generic metric, our interest lies in offering users a practical tool to analyze their special cases and help them making their own choice.

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

This thesis project depicts the proposal of a generic metric as well as its variants to evaluate Monte Carlo neutron transport code in terms of performance, portability and productivity through the implementation of a benchmark based on the Monte Carlo neutron transport prototype PATMOS via different programming languages or libraries.

The development work is based on the SIGMA1 on-the-fly Doppler broadening method which calculates cross section data on-the-fly during the simulation. Because this method has added more FLOP workloads and thus manages to mitigate the memory-bound issue caused by random memory accesses which are mainly incited by the binary search of the typical pretabulated method. Besides, we adopt a heterogeneous offloading strategy for our portable implementations of Monte Carlo codes, which performs all the particle tracking and scoring on host, and offloads the calculations of total microscopic cross sections of nuclides to accelerators since the cross section computation accounts for up to 95% of total run time in PATMOS. To achieve this heterogeneous offloading strategy, the main development effort is to rewrite codes which are dedicated to porting nuclide data and microscopic cross section calculations to devices via implementations of several programming models.

The first part of work in this thesis project focuses on the portable implementations of microscopic cross section computational kernel in PATMOS. Our general programming model can be abstracted as `OpenMP thread + {X}`, where `{X}` can be any languages or libraries which are capable of parallel programming on modern accelerators including CUDA, OpenACC, OpenMP offload, Kokkos, SYCL or none at all. For the purpose of reducing the large amount of overheads introduced by kernel launch and memory transfer operations with the utilization of conventional particle tracking approach, the history-based method, we have proposed another approach, called the pseudo event-based method, which reorganizes the procedure of history tracking into three events which are respectively assigned to handle data banking, microscopic cross section calculation and the rest of processes (macroscopic cross section summarization, distance sampling, particle movement and interaction).

The tests show that the pseudo event-based method generally exceeds a magnitude of performance achieved by the history-based method. It corresponds to the fact that the pseudo event-based method manages to batch small transfers into one larger transfer and merge small kernels into one larger kernel which eliminates significantly the data transfer and kernel launch overheads and exploits higher computational capability of accelerators. The numerical quantities such as the achieved occupancy have also proved that the pseudo event-based method achieves to improve the compute throughput and memory bandwidth of Nvidia GPUs in comparison with the history-based method. However, from the perspective of the utilization of compute resources of Nvidia GPUs, both the pseudo event-based and history-based methods are far from making full use the compute resources. The profiling analysis indicates that our heterogeneous offloading code may be mitigated by reducing the number of registers used for each CUDA thread, adding instruction-level parallelism and optimizing memory alignment and access patterns.

As for the comparison of performance achieved by different portable implementations of slabAllNuclides, the results show that in host mode, the OpenMP thread and Kokkos versions which can be compiled by Intel compiler with the vectorization options enabled gain better performance than other implementations. In offload mode, the CUDA and versions always renders better performance than other implementations via both the pseudo event-based and history-based methods.

The second part of work in this thesis project intends to evaluate our portable implementations of Monte Carlo neutron transport codes in terms of performance, portability and productivity. A generic metric $\boldsymbol{F}$ has been established based on the metric of performance portability $\boldsymbol{\mathcal{P}}(a, p, H)$ proposed by Pennycook and his group. Comparing to Pennycook's metric, our generic metric allows for the aggregation of values retrieved from an application set or a problem set. Besides, it integrates three aggregating methods which are respectively the arithmetic mean, the geometric mean and the harmonic mean. The adaptabilities of three aggregating methods applying to different types of data set have been analyzed theoretically. According to the definitions of performance, we have proved that $\boldsymbol{F_{gm}}$ is the most adaptive metric applying to a data set retrieved by a metric indicating the absolute performance and $\boldsymbol{F_{am}}$ is the most adaptive metric applying to a data set retrieved by a metric indicating the relative performance.

Furthermore, we have proposed several variants of $\boldsymbol{F}$ to make it capable of evaluating relaxed performance portability where the estimated value will be zero only if the given application is blocked by all platforms in the platform set. Such variants make users easier to obtain meaningful values without the obligation of finding the appropriate application or platform set where each application in the application set is supported by all platforms in the platform set.

We have used the generic metric $\boldsymbol{F} \cdot \boldsymbol{C}(a, p, h)$ and its variants to evaluate the performance portability and productive performance portability of different implementations of slabAllNuclides (OpenMP thread, CUDA, OpenACC, OpenMP offload, Kokkos and SYCL) across a set of platforms.

The main contributions of this research work can be concluded into several points:

- Proposition of a new particle tracking algorithm, pseudo event-based method for the heterogeneous offloading

strategy of SIGMA1 on-the-fly Doppler broadening method.

- Implementations of SIGMA1 on-the-fly Doppler broadening method via multiple programming models.
- Proposition and analysis of the generic metric and its variants $\boldsymbol{F} \cdot \boldsymbol{C}(a, p, h)$ for the quantitative evaluation of portable implementations of Monte Carlo neutron transport codes in terms of portability, performance and productivity.

## 6.2 Future Work

There is some work that we intend to do for future development. From evaluation metric side, the operator $\boldsymbol{F}$ need to be testified by much more different cases which are widely used in the domain of scientific computing. More aggregating methods may also be integrated into our generic metric to enrich the capability of $\boldsymbol{F}$ handling complex problems. Besides, more tests need to be done so as to cover a wider range of architectures (AMD, Intel GPUs, etc.) for the evaluation of our portable implementations of Monte Carlo neutron transport code in terms of performance, portability and productivity. We also intend to find an optimal version using portable programming model (on Nvidia GPUs) such as the OpenACC version and compare the cost the portability for moving it to other architectures.

From programming model side, we have interest to optimize the OpenMP offload, Kokkos and SYCL versions. For example, Kokkos may enable the support of MTMA strategy in future which makes it possible to allow multiple threads targeting to multiple devices at the same time. Several implementations of SYCL such as DPC++ may completely support Nvidia GPUs. All of these possible changes need to be tested and integrated in our portable implementations of Monte Carlo neutron transport code.

From Monte Carlo neutron transport simulation side, since we have demonstrated that porting total microscopic cross section calculation with Doppler broadening techniques to accelerators may contribute to a significant performance improvement, we can offload partial cross section calculation as well in order to make our implementation more adaptive to other complex cases. Moreover, we may also offload other methods of cross section calculation such as pre-tabulated and multipole methods to accelerators.

# Bibliography


[1] *Intel Cilk+*. URL https://www.cilkplus.org.

[2] *DLB User Guide*.

[3] *Xcalable Home Page*. URL http://www.xcalablemp.org.

[4] Broadwell - Microarchitectures - Intel. . URL https://en.wikichip.org/wiki/intel/microarchitectures/broadwell.

[5] Gen9.5 - Microarchitectures - Intel. . URL https://en.wikichip.org/wiki/intel/microarchitectures/gen9.5.

[6] TPC-D - Frequently Asked Questions (FAQ). URL https://web.archive.org/web/20111104224323/http://www.tpc.org/tpcd/faq.asp#anchor1140017.

[7] POWER9 - Microarchitectures - IBM. URL https://en.wikichip.org/wiki/ibm/microarchitectures/power9.

[8] *SYCL Overview*. URL https://www.khronos.org/sycl.

[9] NVLink, Pascal and Stacked Memory: Feeding the Appetite for Big Data. 2014. URL https://devblogs.nvidia.com/nvlink-pascal-stacked-memory-feeding-appetite-big-data.

[10] *OpenMP Application Programming Interface*, 11 2015.

[11] *StarPU Handbook*, 09 2018.

[12] *XcalableMP Language Specification*, 11 2018.

[13] *OpenACC Application Programming Interface*, 11 2019.

[14] G. Allaire and G. Bal. Homogenization of the Criticality Spectral Equation in Neutron Transport. *Mathematical Modelling and Numerical Analysis*, 33:721–746, 1999. doi: 10.1051/m2an:1999160.

[15] A. Alpay and V. Heuveline. SYCL beyond OpenCL: The Architecture, Current State and Future Direction of hipSYCL. In *Proceedings of the International Workshop on OpenCL*, pages 1–1, 2020. doi: 10.1145/3388333.3388658.

[16] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the AFIPS Conference*, volume 30, pages 483–485, 1967. doi: 10.1145/1465482.1465560.

[17] H. Bateman. The Solution of a System of Differential Equations Occurring in the Theory of Radio-active Transformations. *Proc. Cambridge Phil. Soc., 1908*, 15:423–427, 1908.

[18] R. M. Bergmann and J. L. Vujić. Algorithmic Choices in WARP–A Framework for Continuous Energy Monte Carlo Neutron Transport in General 3D Geometries on GPUs. *Annals of Nuclear Energy*, 77:176–193, 2015. doi: 10.1016/j.anucene.2014.10.039.

[19] K. Binder. *Applications of the Monte Carlo Methods in Statistical Physics*. Springer, 2013.

[20] R. C. Bleile. Thin-Threads: An Approach for History-Based Monte Carlo on GPUs. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2018.

[21] R. C. Bleile, P. S. Brantley, S. A. Dawson, M. J. O'Brien, and H. Childs. Investigation of Portable Event-based Monte Carlo Transport Using the Nvidia Thrust Library. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2016.

[22] R. C. Bleile, P. S. Brantley, M. J. O'Brien, and H. Childs. Algorithmic Improvements for Portable Event-based Monte Carlo Transport Using the Nvidia Thrust Library. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2016.

[23] A. B. Bondi. Characteristics of Scalability and their Impact on Performance. In *Proceedings of the 2th International Workshop on Software and Performance*, page 195, 2000. doi: 10.1145/350391.350432.

[24] F. B. Brown. New Hash-based Energy Lookup Algorithm for Monte Carlo Codes. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2014.

[25] F. B. Brown and W. R. Martin. Monte Carlo Methods for Radiation Transport Analysis on Vector Computers. *Progress in Nuclear Energy*, 14(3):269–299, 1984. doi: 10.1016/0149-1970(84)90024-6.

[26] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Travener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[27] E. Brun, F. Damian, C. Diop, E. Dumonteil, F. Hugot, C. Jouanne, Y. Lee, F. Malvagi, A. Mazzolo, O. Petit, et al. Tripoli-4®, CEA, EDF and AREVA Reference Monte Carlo Code. In *SNA+ MC 2013-Joint International Conference on Supercomputing in Nuclear Applications+ Monte Carlo*, page 06023. EDP Sciences, 2014. doi: 10.1051/snamc/201406023.

[28] E. Brun, S. Chauveau, and F. Malvagi. Patmos: A Prototype Monte Carlo Transport Code to Test High Performance Architectures. In *Proceedings of International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering, Jeju, Korea*, 2017.

[29] A. Carreño, A. Vidal-Ferràndiz, D. Ginestar, and G. Verdú. The Solution of the Lambda Modes Problem Using Block Iterative Eigensolvers. In *Computational Science - ICCS 2018. Lecture Notes in Computer Science*, volume 10861. Springer, Cham, 2018. doi: 10.1007/978-3-319-93701-4_67.

[30] M. B. Chadwick et al. ENDF/B-VII.0: Next Generation Evaluated Nuclear Data Library for Nuclear Science and Technology. *Nuclear Data Sheets*, pages 2931–3060, 2006. doi: 10.1016/j.nds.2006.11.001.

[31] T. Chang, E. Brun, and C. Calvin. Portable Monte Carlo Transport Performance Evaluation in the PATMOS Prototype. In *International Conference on Parallel Processing and Applied Mathematics*, pages 528–539. Springer, 2019.

[32] T. Chang, E. Brun, and C. Calvin. Performance Portability Analysis of Different Programming Models for High Performance Monte Carlo Neutron Transport. In *Joint International Conference on Supercomputing in Nuclear Applications + Monte Carlo 2020 (SNA+MC 2020)*, 2020. To be published.

[33] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. John Wiley & Sons, Inc., 2014.

[34] D. M. Chitty. A Data Parallel Approach to Genetic Programming using Programmable Graphics Hardware. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, pages 1566–1573. ACM, 2007.

[35] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011. doi: 10.1109/TCAD.2011.2110592.

[36] D. E. Cullen. POINT 2009: A Temperature Dependent ENDF/B-VII.0 Data Cross Section Library. Technical report, Lawrence Livermore National Laboratory, 2009.

[37] D. E. Cullen. PREPRO 2019: 2019 ENDF/B Pre-processing Codes. Technical report, IAEA-NDS-229, August 2019.

[38] D. E. Cullen and C. R. Weisbin. Exact Doppler Broadening of Tabulated Cross Sections. *Nuclear Science and Engineering*, 60(3):199–229, 1976. doi: 10.13182/NSE76-1.

[39] F. Darema. The SPMD Model: Past, Present and Future. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 1–1. Springer, 2001.

[40] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, and J. Salmon. Performance Portability across Diverse Computer Architectures. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 1–13. IEEE, 2019. doi: 10.1109/P3HPC49587.2019.00006.

[41] C. Dean, R. Perry, R. Neal, and A. Kyrieleis. Validation of Run-time Doppler Broadening in MONK with JEFF3. 1. *Journal of the Korean Physical Society*, 59(2):1162–1165, 2011.

[42] M. E. Dunn and N. M. Greene. AMPX-2000: a Cross-Section Processing System for Generating Nuclear Data for Criticality Safety Applications. *Trans. Am. Nucl. Soc*, 86:118–119, 2002.

[43] H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling Manycore Performance Portability through Polymorphic Memory Access Patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. doi: 10.1016/j.jpdc.2014.07.003.

[44] H. Eves. Means Appearing in Geometric Figures. *Mathematics magazine*, 76(4):292–294, 2003. doi: 10.1080/0025570X.2003.11953195.

[45] J. Fang, L. A. Varbanescu, and H. Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. *International Conference on Parallel Processing*, 2011.

[46] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C21(9):948–960, 1972. doi: 10.1109/TC.1972.5009071.

[47] M. P. I. Forum. *MPI: A Message-Passing Interface Standard*, 06 2015.

[48] G. Frantz. Digital signal processor trends. In *IEEE Micro*, volume 20, pages 52–59, 2000. doi: 10.1109/40.888703.

[49] A. Funk, V. Basili, L. Hochstein, and J. Kepner. Application of a Development Time Productivity Metric to Parallel Software Development. In *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 8–12, 2005. doi: 10.1145/1145319.1145323.

[50] A. Gottlieb and G. S. Almasi. Highly Parallel Computing. *Benjamin-Cummings Publishing Co., Inc.Subs. of Addison-Wesley Longman Publ.*, 1989.

[51] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine. Open mpi: A high-performance, heterogeneous mpi. In *2006 IEEE International Conference on Cluster Computing*, pages 1–9, 2006. doi: 10.1109/CLUSTR.2006.311904.

[52] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, 1988. doi: 10.1145/42411.42415.

[53] M. H. Halstead et al. *Elements of Software Science*, volume 7. Elsevier New York, 1977.

[54] S. P. Hamilton and T. M. Evans. Continuous-energy Monte Carlo neutron transport on GPUs in the Shift code. *Annals of Nuclear Energy*, 128:236–247, 2019. doi: 10.1016/j.anucene.2014.10.039.

[55] S. P. Hamilton, S. R. Slattery, and T. M. Evans. Multigroup Monte Carlo on GPUs: Comparison of History-and Event-based Algorithms. *Annals of Nuclear Energy*, 113:506–518, 2018. doi: 10.1016/j.anucene.2017.11.032.

[56] S. L. Harrell, J. Kitson, R. Bird, S. J. Pennycook, J. Sewall, D. Jacobsen, D. N. Asanza, A. Hsu, H. C. Carrillo, H. Kim, et al. Effective Performance Portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 24–36. IEEE, 2018. doi: 10.1109/P3HPC.2018.00006.

[57] M. Herman and A. Trkov. *ENDF-6 Formats Manual*, 2009.

[58] R. Hornung et al. *ASC Tri-lab Co-design Level 2 Milestone Report 2015*, 09 2015.

[59] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki. CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application. In *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2013.

[60] R. N. Hwang. A rigorous pole representation of multilevel cross sections and its practical applications. *Nuclear Science and Engineering*, 96(3):192–209, 1987. doi: 10.131182/NSE87-A16381.

[61] IBM. *Power ISA Version 3.0B*, 2019.

[62] Intel. *Intel Xeon Processor E7 v2 2800/4800/8800 Product Family*, 2014.

[63] H. K. Jacobs. *Mathematics: A Human Endeavor (Third ed.)*. W. H. Freeman, 1994.

[64] C. Jaspan and C. Sadowski. No Single Metric Captures Productivity. In *Rethinking Productivity in Software Engineering*, pages 13–20. Springer, 2019.

[65] Y. Jo and N. Z. Cho. Refinements of On-The-Fly Doppler Broadening via Gauss-Hermite Quadrature in Monte Carlo Reactor Analysis. *Trans. Kor. Nucl. Soc*, pages 18–19, 2017.

[66] C. Josey, P. Ducru, B. Forget, and K. Smith. Windowed Multipole for Cross Section Doppler Broadening. *Journal of Computational Physics*, 307:715–727, 2016. doi: 10.1016/j.jcp.2015.08.013.

[67] K. Kennedy, C. Koelbel, and R. Schreiber. Defining and Measuring the Productivity of Programming Languages. *The International Journal of High Performance Computing Applications*, 18(4):441–448, 2004. doi: 10.1177/1094342004048537.

[68] R. Keryell et al. triSYCL. `https://github.com/triSYCL/triSYCL`, 2014.

[69] Khronos. *SYCL Specification*, 2020.

[70] G. King. How Not to Lie with Statistics: Avoiding Common mistakes in Quantitative Political Science. *American Journal of Political Science*, pages 666–687, 1986. doi: 10.2307/2111095.

[71] A. KUKANOC and M. J. VOSS. The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks. *Intel Technology Journal*, 4, 11 2007.

[72] A. Lashgar, A. Baniasadi, and A. Khonsari. Inter-warp Instruction Temporal Locality in Deep-Multithreaded GPUs. In *Architecture of Computing Systems (ARCS)*, volume 7767, pages 134–146, 2013. doi: 10.1007/978-3-642-36424-2_12.

[73] T. Law, R. Kevis, S. Powell, J. Dickson, S. Maheswaran, J. Herdman, and S. Jarvis. Performance Portability of an Unstructured Hydrodynamics Mini-application. In *Proceedings of 2018 International Workshop on Performance, Portability, and Productivity in HPC (P3HPC). ACM, New York, NY, USA*, 2018.

[74] J. Leppänen. Two Practical Methods for Unionized Energy Grid Construction in Continuous-Energy Monte Carlo Neutron Transport Calculation. *Annals of Nuclear Energy*, 36(7):878–885, 2009. doi: 10.1016/j.anucene.2009.03.019.

[75] J. Leppänen. Serpent–A Continuous-energy Monte Carlo Reactor Physics Burnup Calculation Code. *VTT Technical Research Centre of Finland*, 4, 2013.

[76] S. Li, K. Wang, and G. Yu. Research on Fast-Doppler-Broadening of Neutron Cross Sections. In *PHYSOR 2012 – Advances in Reactor Physics – Linking Research*, 2012.

[77] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Proceedings of 18th International Parallel and Distributed Processing Symposium, 2004.*, pages 16–, 2004. doi: 10.1109/IPDPS.2004.1302922.

[78] J. Luitjens. *CUDA STREAMS*.

[79] A. L. Lund, A. R. Siege, B. Forget, C. Josey, and P. K. Romano. Using Fractional Cascading to Accelerate Cross Section Lookups in Monte Carlo Neutron Transport Calculations. 2015.

[80] I. Lux and L. Koblinger. *Monte Carlo Particle Transport Methods*. CRC press, 2018.

[81] R. Macfarlane, D. W. Muir, R. Boicourt, A. C. Kahler III, and J. L. Conlin. The NJOY Nuclear Data Processing System, Version 2016. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2017.

[82] J. D. Markgraf. The Von Neumann Bottleneck. 2007.

[83] M. Martineau, S. McIntosh-Smith, and W. Gaudin. Assessing the Performance Portability of Modern Parallel Programming Models Using TeaLeaf. *Concurrency and Computation: Practice and Experience*, 29(15):e4117, 2017. doi: 10.1002/cpe.4117.

[84] D. S. Medina, A. St-Cyr, and T. Warburton. OCCA: A unified approach to multi-threading languages. *arXiv preprint arXiv:1403.0968*, 2014.

[85] S. Mittal. A Survey of Techniques for Managingand Leveraging Caches in GPUs. *Journal of Circuits, Systems, and Computers*, 23(8), 2014. doi: 10.1142/S0218126614300025.

[86] A. Nalamalpu, N. Kurd, A. Deval, C. Mozak, J. Douglas, A. Khanna, F. Paillet, G. Schrom, and B. Phelps. Broadwell : A family of IA 14nm processors. *Symposium on VLSI Circuits*, pages C314–C315, 2015. doi: 10.1109/VLSIC.2015.7231304.

[87] Nvidia. *NVIDIA Tesla P100*, 2016.

[88] Nvidia. *NVIDIA TESLA V100 GPU ARCHITECTURE*, 2017.

[89] Nvidia. *CUDA C Programming Guide*, 10 2018.

[90] Nvidia. *CUDA C Programming Guide*, 2018.

[91] Nvidia. *Profiler User's Guide*, 2019.

[92] Nvidia. *NVIDIA A100 Tensor Core GPU Architecture*, 2020.

[93] P. Ojeda, E. Garcia, Martin, A. Londonõ, and N.-Y. Chen. Monte Carlo Simulations of Proteins in Cages: Influence of Confinement on the Stability of Intermediate States. *Biophysical Journal*, 96(3):1076–1082, 2009. doi: 10.1529/biophysj.107.125369.

[94] OpenACC-Standard.org. *The OpenACC Application Programming Interface*, 11 2017.

[95] D. Ozog, A. D. Malony, and A. R. Siegel. A Performance Analysis of SIMD Algorithms for Monte Carlo Simulations of Nuclear Reactor Cores. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 733–742. IEEE, 2015. doi: 10.1109/IPDPS.2015.105.

[96] T. M. Pandya, S. R. Johnson, T. M. Evans, G. G. Davidson, S. P. Hamilton, and A. T. Godfrey. Implementation, Capabilities, and Benchmarking of Shift, A Massively Parallel Monte Carlo Radiation Transport Code. *Journal of Computational Physics*, 308:239–272, 2016.

[97] S. J. Pennycook, J. D. Sewall, and V. W. Lee. A Metric for Performance Portability. *arXiv preprint arXiv:1611.07409*, 2016.

[98] S. J. Pennycook, J. D. Sewall, and V. W. Lee. Implications of a Metric for Performance Portability. *Future Generation Computer Systems*, 92:947–958, 2019. doi: 10.1016/j.future.2017.08.007.

[99] S. Pinelas, P. Narasimman, and K. Ravi. Stability of Pythagorean Mean Functional Equation. *Global Journal of Mathematics*, 1(4 (2015)):398–411, 2015.

[100] I. Z. Reguly and G. R. Mudalige. Productivity, Performance, and Portability for Computational Fluid Dynamics Applications. *Computers & Fluids*, 199:104425, 2020. doi: 10.1016/j.compfluid.2020.104425.

[101] J. Reinders. *AVX-512 Instructions*, 2013.

[102] J. Reinders. *Additional AVX-512 Instructions*, 2014.

[103] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian. Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL, 2020.

[104] R. Reyes. Codeplay Contribution to DPC++ brings SYCL Support for NVIDIA GPUs. 2020. URL `https://www.codeplay.com/portal/02-03-20-codeplay-contribution-to-dpcpp-brings-sycl-support-for-nvidia-gpus`.

[105] P. K. Romano and B. Forget. The OpenMC Monte Carlo Particle Transport Code. *Annals of Nuclear Energy*, 51:274–281, 2013. doi: 10.1016/j.anucene.2012.06.040.

[106] P. K. Romano and A. R. Siegel. Limits on the Efficiency of Event-based Algorithms for Monte Carlo Neutron Transport. *Nuclear Engineering and Technology*, 49(6):1165–1171, 2017. doi: 10.1016/j.net.2017.06.006.

[107] P. K. Romano and T. H. Trumbull. Comparison of Algorithms for Doppler Broadening Pointwise Tabulated Cross Sections. *Annals of Nuclear Energy*, 75:358–364, 2015.

[108] S. K. Sadasivam, B. W. Thomptp, R. Kalla, and W. J. Starke. IBM Power9 Processor Architecture. In *IEEE Micro*, volume 37, pages 40–51, 2017. doi: 10.1109/MM.2017.40.

[109] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications? In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 440–451. IEEE, 2012. doi: 10.1109/ISCA.2012.6237038.

[110] S. S. Sawilowsky. You think you've got trivials? *Journal of Modern Applied Statistical Methods*, 2(1):218–225, 2003. doi: 10.22237/jmasm/1051748460.

[111] A. Siegel, K. Smith, K. Felker, P. Romano, B. Forget, and P. Beckman. Improved Cache Performance in Monte Carlo Transport Calculations Using Energy Banding. *Computer Physics Communications*, 185(4):1195–1199, 2014. doi: 10.1016/j.cpc.2013.10.008.

[112] J. E. Smith. Characterizing Computer Performance with a Single Number. *Communications of the ACM*, 31(10):1202–1206, 1988. doi: 10.1145/63039.63043.

[113] C. Software. ComputeCpp. URL `https://codeplay.com/products/computesuite/computecpp`.

[114] N. Soppera, M. Bossant, O. Cabellos, E. Dupont, and C. J. Díez. JANIS: NEA JAva-based Nuclear Data Information System. *EPJ Web Conf.*, 146, 2017. doi: 10.1051/epjconf/201714607006.

[115] W. M. Stacey. *Nuclear Reactor Physics - Second Edition*. Wiley, 2007.

[116] H. Stewart. *An Introduction to the Theory of the Boltzmann Equation*. Dover Books, 1971.

[117] A. Tabuchi, M. Nakao, H. Murai, T. Boku, and M. Sato. Performance Evaluation for a Hydrodynamics Application in XcalableACC PGAS Language for Accelerated Clusters. In *Proceedings of Workshops of HPC Asia*, pages 1–10, 2018. doi: 10.1145/3176364.3176365.

[118] X.-. M. C. Team. MCNP—A General N-Particle Transport Code, Version 5, 2003.

[119] J. R. Tramm and A. R. Siegel. Memory bottlenecks and memory contention in multi-core monte carlo transport codes. In *SNA+ MC 2013-Joint International Conference on Supercomputing in Nuclear Applications+ Monte Carlo*, page 04208. EDP Sciences, 2014. doi: 10.1051/snamc/201404208.

[120] P. Žužek. sycl-gtx. `https://github.com/ProGTX/sycl-gtx`, 2014.

[121] P. Vaz. Neutron Transport Simulation (selected topics). *Radiation Physics and Chemistry*, 78(10):829–842, 2009. doi: 10.1016/j.radphyschem.2009.04.022.

[122] G. Velarde, C. Ahnert, and J. M. Aragonés. A Comparison of the Eigenvalue Equations in k, $\alpha$, $\lambda$ and $\gamma$ in Reactor Theory. In *JUNTA DE ENERGIA NUCLEAR*, 1977.

[123] J. von Neumann. First Draft of a Report on the EDVAC. 1945.

[124] J. C. Wagner, E. L. Redmond, S. P. Palmtag, and J. S. Hendricks. MCNP: Multigroup/Adjoint Capabilities. Technical report, Los Alamos National Lab., NM (United States), 1994.

[125] J. A. Walsh, P. K. Romano, B. Forget, and K. S. Smith. Optimizations of the Energy Grid Search Algorithm in Continuous-Energy Monte Carlo Particle Transport Codes. *Computer Physics Communications*, 196:134–142, 2015. doi: 10.1016/j.cpc.2015.05.025.

[126] K. Wang, Z. Li, D. She, Q. Xu, Y. Qiu, J. Yu, J. Sun, X. Fan, G. Yu, et al. RMC-A Monte Carlo Code for Reactor Core Analysis. In *SNA+ MC 2013-Joint International Conference on Supercomputing in Nuclear Applications+ Monte Carlo*. EDP Sciences, 2014.

[127] Y. Wang. *Optimization of Monte Carlo Neutron Transport Simulations with Emerging Architectures*. PhD thesis, 2017.

[128] Y. Wang, S. Roy, and R. N. Run-time Power-gating in Caches of GPUs for Leakage Energy Savings. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 300–303, 2012. doi: 10.1109/DATE.2012.6176483.

[129] Y. Wang, E. Brun, F. Malvagi, and C. Calvin. Competing Energy Lookup Algorithms in Monte Carlo Neutron Transport Calculations and their Optimization on CPU and Intel MIC Architectures. *Journal of Computational Science*, 20:94–102, 2017. doi: 10.1016/j.jocs.2017.01.006.

[130] M. H. Weik. A Third Survey of Domestic Electronic Digital Computing Systems. *Ballistic Research Laboratory*, 1961.

[131] S. Wienke. Productivity and Software Development Effort Estimation in HPC.

[132] S. Wienke. *Productivity and Software Development Effort Estimation in High-Performance Computing*. Apprimus Verlag, 2017.

[133] Y. Wu. *Fusion Neutronics*. Springer, 2017.

[134] Y. Wu, Z. Xie, and U. Fischer. A Discrete Ordinates Nodal Method for One-Dimensional Neutron Transport Calculation in Curvilinear Geometries. *Nuclear Science and Engineering*, 133(3):350–357, 1999.

[135] P. Yalamanchili, U. Arshad, Z. Mohammed, P. Garigipati, P. Entschev, B. Kloppenborg, J. Malcolm, and J. Melonakos. ArrayFire - A high performance software library for parallel computing with an easy-to-use API, 2015. URL `https://github.com/arrayfire/arrayfire`.

[136] G. Yesilyurt, W. R. Martin, and F. B. Brown. On-the-fly Doppler broadening for Monte Carlo Codes. *Nuclear science and engineering*, 171(3):239–257, 2012. doi: 10.13182/NSE11-67.

[137] M. Yuffe et al. A Fully Integrated Multi-CPU, GPU and Memory Controller 32nm Processor. In *Proceedings of the International Solid-State Circuits Conference*. IEEE, 2011. doi: 10.1109/ISSCC.2011.5746311.

[138] M. Zelkowitz, V. Basili, S. Asgari, L. Hochstein, J. Hollingsworth, and T. Nakamura. Measuring Productivity on High Performance Computers. In *Software Metrics, IEEE International Symposium on*, pages 6–6. Citeseer, 2005.

[139] T. Zhao, S. Williams, M. Hall, and H. Johansen. Delivering Performance-Portable Stencil Computations on CPUs and GPUs Using Bricks. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 59–70. IEEE, 2018. doi: 10.1109/P3HPC.2018.00009.

[140] A. Zoia, E. Brun, and F. Malvagi. Alpha Eigenvalue Calculations with TRIPOLI-4®. *Annals of Nuclear Energy (Oxford)*, pages 276–284, 2014. doi: 101016/janucene201307018.

INSTITUT POLYTECHNIQUE DE PARIS

**Titre:** Évaluation de modèles de programmation pour les architectures manycore et / ou hétérogènes pour les codes de transport neutronique Monte Carlo

**Mots clés:** Architectures manycore, Architectures hétérogènes, Transport des particules

**Résumé:** Dans cette thèse nous nous proposons d'évaluer les différents modèles de programmation disponibles pour adresser les architectures de type manycore et / ou hétérogènes dans le cadre des codes de transport Monte Carlo. On considèrera dans un premier temps un cas test d'application simple mais représentatif pour couvrir un éventail assez large de solutions et les comparer en terme de performance, de portabilité de la performance, de facilité de mise en œuvre et de maintenabilité. Les architectures cibles sont les CPU 'classique', Intel Xeon Phi et GPU. Les modèles de programmation les plus pertinents seront ensuite mis en place dans un code de transport Monte Carlo.

**Title:** Evaluation of programming models for manycore and / or heterogeneous architectures for Monte Carlo neutron transport codes

**Keywords:** Manycore architectures, Heterogeneous architectures, Particles transport

**Abstract:** In this thesis we propose to evaluate the different programming models available for addressing manycore and / or heterogeneous architectures within the framework of the Monte Carlo transport codes. A simple but representative application test case will be considered in order to cover a fairly wide range of solutions and compare them in terms of performance, portability of performance, ease of implementation and maintainability. The target architectures are 'classic' CPU, Intel Xeon Phi and GPU. The most relevant programming models will then be set up in a Monte Carlo transport code.

**Institut Polytechnique de Paris**
91120 Palaiseau, France