Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Τεχνικές Αποδοτικής Εικονικοποίησης Επιταχυντών για Νεφοϋπολογιστικά Περιβάλλοντα

Διδακτορική Διατριβή

Στέφανος Σ. Γεράγγελος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών

Αθήνα, Ιούλιος 2019

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Τεχνικές Αποδοτικής Εικονικοποίησης Επιταχυντών για Νεφοϋπολογιστικά Περιβάλλοντα

Διδακτορική Διατριβή

Στέφανος Σ. Γεράγγελος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών

Συμβουλευτική Επιτροπή: Νεκτάριος Κοζύρης
Παναγιώτης Τσανάκας
Νικόλαος Παπασπύρου

Εγκρίθηκε από την επταμελή επιτροπή τη 17η Ιουλίου 2019.

Νεκτάριος Κοζύρης Καθ. ΕΜΠ	Παναγιώτης Τσανάκας Καθ. ΕΜΠ	Νικόλαος Παπασπύρου Καθ. ΕΜΠ
Γεώργιος Γκούμας Επίκ. Καθ. ΕΜΠ	Δημήτριος Σούντρης Καθ. ΕΜΠ	Άγγελος Μπίλας Καθ. Παν. Κρήτης

Στέργιος Αναστασιάδης
Αναπλ. Καθ. Παν. Ιωαννίνων

Αθήνα, Ιούλιος 2019

Στέφανος Σ. Γεράγγελος
Διδάκτωρ Εθνικού Μετσοβίου Πολυτεχνείου

Στη σύντροφό μου, Ιωάννα

Περίληψη

Καθώς ο ρυθμός με τον οποίο δημιουργούνται δεδομένα στο σύγχρονο κόσμο αυξάνεται συνεχώς, υπάρχει ολοένα και πιο έντονη η ανάγκη για αύξηση της διαθέσιμης υπολογιστικής ισχύος. Παρ' όλα αυτά, η προϋπάρχουσα τάση κλιμάκωσης των πυρήνων στις διάφορες γενιές αρχιτεκτονικών παρουσιάζει ένα εγγενές όριο για το άμεσο μέλλον. Σε αυτό το πλαίσιο, οι επιστήμονες και ερευνητές εκτιμούν ότι ο τρόπος επεξεργασίας των δεδομένων θα προσαρμοστεί σε αυτές τις νέες συνθήκες και τα μελλοντικά κέντρα δεδομένων (data centers) σταδιακά θα μετακινηθούν σε πιο ετερογενή περιβάλλοντα υιοθέτωντας υπολογιστικούς πόρους επιταχυντών.

Την ίδια στιγμή, το cloud computing διαδραματίζει εξαιρετικά σημαντικό ρόλο σε πολλά κέντρα δεδομένων προσφέροντας πλεονεκτήματα τόσο για τους τελικούς χρήστες όσο και για τους παρόχους, όπως για παράδειγμα, ευελιξία, ενοποίηση των διαθέσιμων φυσικών κόμβων, μείωση του κόστους και πιο αποτελεσματική χρησιμοποίηση των πόρων. Με την προαναφερθείσα δυναμική που εμφανίζουν οι ετερογενείς αρχιτεκτονικές και πιο συγκεκριμένα οι επιταχυντές, δημιουργείται μία αυξανόμενη ανάγκη για ένταξη αυτών στα υπάρχοντα cloud περιβάλλοντα. Ουσιαστικά, οι υποδομές εικονικοποίησης χρειάζεται να ενσωματώσουν τα συστήματα επιταχυντών λαμβάνοντας υπόψη τα εξειδικευμένα χαρακτηριστικά αυτού του τύπου των συσκεύων.

Σε αυτή την εργασία, εξερευνούμε τις συνέπειες της ένταξης των συσκευών επιτάχυνσης στα συστήματα εικονικοποίησης. Αναγνωρίζουμε τις βασικές δυσκολίες και προκλήσεις της εικονικοποίησης επιταχυντών και σκιαγραφούμε τους λόγους για τους οποίους οι παραδοσιακές μέθοδοι εικονικοποίησης Ε/Ε δεν είναι οι πλέον κατάλληλες για αυτές τις εξειδικευμένες συσκευές. Εξερευνούμε τεχνικές εικονικοποίησης με βάση δύο δημοφιλείς οικογένειες επιταχυντών, τις NVIDIA GPUs και τους συνεπεξεργα-

στές Intel Xeon Phi, ακολουθώντας διαφορετικές προσεγγίσεις με βάση τη φύση κάθε ενός από τους προαναφερθέντες επιταχυντές.

Σχετικά με την προσέγγισή μας για εικονικοποίηση GPU, προτείνουμε τη χρησιμοποίηση ενός εργαλείου απομακρυσμένης επιτάχυνσης στο πλαίσιο του ίδιου φυσικού κόμβου συνδυάζοντάς το με ένα αποδοτικό σύστημα για επικοινωνία εντός του ίδιου κόμβου. Αυτό έχει ως αποτέλεσμα σημαντικά καλύτερη επίδοση κατά τη διαδικασία offloading εφαρμογών σε εικονικοποιημένα περιβάλλοντα. Σύμφωνα με τη σχεδίαση του συστήματος επικοινωνίας, το μονοπάτι δεδομένων περιλαμβάνει την εμπλοκή του υπερεπόπτη (hypervisor) ως δικτυακού μέσου, αντί για το προνομιούχο driver domain. Επιπλέον, αξιολογούμε το σύστημα επικοινωνίας κάνοντας τόσο χρήση δικτυακών μικρο-μετροπρογραμμάτων, όσο και ενός κοινού GPU πυρήνα, δείχνοντας ότι η συνολική υποδομή αποτελεί μία βιώσιμη πρόταση για τα λογισμικά επιταχυντών που κυκλοφορούν με κλειστές άδειες. Πιο συγκεκριμένα, τα πειραματικά αποτελέσματα δείχνουν ότι η προσέγγισή μας βελτιώνει τη μεταφορά δεδομένων έως και 6.3 φορές σε σύγκριση με το απομακρυσμένο προκαθορισμένο μονοπάτι, ενώ προσθέτει ένα κόστος 15% κατά την εκτέλεση του GPU υπολογιστικού πυρήνα σε σχέση με την περίπτωση απευθείας ανάθεσης συσκευής.

Σύμφωνα με όσα γνωρίζουμε, η προσέγγισή μας για εικονικοποίηση Xeon Phi συνεπεξεργαστών είναι η πρώτη και η μοναδική αυτή τη στιγμή που υποστηρίζει το διαμοιρασμό ενός Xeon Phi επιταχυντή μεταξύ πολλαπλών εικονικών μηχανών που εκτελούνται στον ίδιο φυσικό κόμβο. Ακολουθούμε την τεχνική της παραεικονικοποίησης στοχεύοντας το χαμηλό επίπεδο μεταφοράς της αντίστοιχης στοίβας λογισμικού. Με αυτό τον τρόπο, οι βασικές σχεδιαστικές άρχες της προσέγγισής μας μπορούν να εφαρμοστούν στις μελλοντικές τεχνολογίες επιταχυντών. Η πειραματική αποτίμηση του πρωτοτύπου μας δείχνει ότι μπορεί να επιφέρει καλύτερη χρησιμοποίηση του επιταχυντή όταν αυτός χρησιμοποιείται από πολλαπλές εικονικές μηχανές, αυξάνοντας το συνολικό throughput έως 3.56x σε σχέση με μία host εφαρμογή, η οποία

αναπαριστά την περίπτωση της απευθείας ανάθεσης συσκευής.

Λέξεις-κλειδιά: Εικονικοποίηση Επιταχυντών, Διαμοιρασμός Επιταχυντών, Εικονικές Μηχανές, Νεφοϋπολογιστικά Περιβάλλοντα

Abstract

As data creation worldwide in today's world keeps growing with remarkable rates, the processing power needs to be increased proportionally. However, the multicore scaling trend presents a limit in the foreseeable future, which is referred in the literature as the dark silicon era. In this context, computer scientists and professionals estimate that the way data are processed will adapt to the new conditions and future data centers will gradually move from the scale-up paradigm to more heterogeneous architectures embracing accelerating resources.

At the same time, cloud computing has been established in many data center infrastructures offering benefits both for the end users as well as the service providers, such as flexibility, server consolidation, cost reduction and better resource utilization among others. With the aforementioned potential that heterogeneous computing and accelerators appears to develop, there is a growing need for integration in the current cloud stacks. In essence, virtualization-aware systems need to embrace accelerators by adapting their components into the specialized nature of this kind of hardware.

In this thesis, we explore the implications of integrating accelerator devices into the virtualization ecosystem. We identify the key challenges of virtualizing accelerator resources and we outline the reasons that the traditional I/O virtualization methods are not adequate for this kind of specialized devices. We explore virtualization techniques targeting two popular accelerator families, NVIDIA GPUs and Intel Xeon Phi coprocessors, following different approaches based on the nature of each accelerator environment.

In our GPU virtualization approach, we propose the use of a remote acceleration framework in a single-node virtualization platform combined with a low overhead intra-node framework which results in efficient application offloading in virtualized en-

vironments. The data path in the design of our intra-node framework is realized through the hypervisor as the network medium, instead of the driver domain. Furthermore, we evaluate our prototype using both network microbenchmarks and analyzing a common GPU stencil, showing that it is a viable approach for accelerator software stacks that are released in a closed manner. Specifically, evaluation results show that our approach boosts the transfer throughput by a factor of up to 6.3 compared to the remote default path, while it adds an overhead of 15% in terms of GPU execution compared to direct device assignment configuration.

Regarding our Xeon Phi virtualization approach, to the best of our knowledge it is the first and currently the only solution that enables sharing of a Xeon Phi device by multiple virtual machines running on the same physical node. We follow the paravirtualization technique targeting the low-level transport layer of the software stack. In this way, the design principles of our approach can be applied to future accelerator technologies as well. Evaluation shows that our prototype can enable better accelerator utilization when it is used by multiple VMs, increasing up to 3.56x the total throughput versus a single host application, which represents the direct device assignment configuration.

Keywords: Accelerator Virtualization, Accelerator Sharing, Virtual Machines, Cloud Environments

Contents

Contents **vii**

List of figures **ix**

List of tables **xi**

Αντί προλόγου **xiii**

Γλωσσάριο τεχνικών όρων **xvii**

Εκτεταμένη περίληψη **1**

1 Introduction **69**
1.1 Outline 72

2 Background **73**
2.1 Hardware acceleration 73
2.1.1 Graphics Processing Units 74
2.1.2 Intel Xeon Phi 77
2.2 Hardware virtualization 79
2.2.1 Evolution of virtualization 80
2.2.2 I/O virtualization techniques 81
2.2.3 Xen virtualization platform 85
2.2.4 KVM virtualization platform 88

3 Accelerator virtualization challenges **91**

4 Optimizing intra-node communication for remote GPU task offloading 97
4.1 Design of V4VSockets 98
4.2 Implementation details 100
4.3 Performance evaluation 102
4.3.1 Microbenchmark evaluation 103
4.3.2 GPU-enabled VMs 105
4.4 Summary 108

5 Enabling sharing of Xeon Phi accelerators in virtualized environments 109
5.1 Design and implementation of vPHI 110
5.2 Performance evaluation 117
5.2.1 Experimental setup 117
5.2.2 Microbenchmark performance 117
5.2.3 Application performance in native mode of execution . 120
5.2.4 Application performance in offload mode of execution 122
5.2.5 VM sharing 126
5.3 Summary 130

6 Related work 133
6.1 Intra-node communication 133
6.2 Accelerator virtualization 134

7 Conclusions and future directions 137

Bibliography 141

List of figures

1.1 Cisco forecasts 49 exabytes per month of mobile data traffic by 2021 (Source: Cisco VNI Mobile, 2017) 70

2.1 Typical offload path for accelerator devices 75
2.2 CUDA software stack 75
2.3 rCUDA use case 76
2.4 Communication between host and Xeon Phi 79
2.5 Direct device assignment 82
2.6 IOV-enabled device 83
2.7 I/O emulation 83
2.8 Paravirtualization 84
2.9 Xen software architecture (Source [3]) 86
2.10 Xen rings structure (Source [3]) 86
2.11 Generic intra-node communication in Xen 87
2.12 Virtio transport mechanism 89

4.1 rCUDA over V4VSockets 98
4.2 TCP/IP and V4VSockets 99
4.3 Generic intra-node communication in Xen 102
4.4 V4VSockets overview 102
4.5 V4VSockets latency 104
4.6 V4VSockets throughput 104
4.7 V4VSockets aggregate throughput 105
4.8 Matrix product total time of execution 106
4.9 Matrix product transfer throughput 107

5.1 vPHI sharing scheme 111
5.2 vPHI Architecture (data and control Path) 112
5.3 Send-receive communication latency 118
5.4 Remote memory access throughput 120
5.5 Launch and execution of `dgemm` using 56 threads 121
5.6 Launch and execution of `dgemm` using 112 threads 121
5.7 Launch and execution of `dgemm` using 224 threads 122
5.8 GEMM benchmark (problem size `s=1`) 123
5.9 GEMM benchmark (problem size `s=2`) 123
5.10 SPMV benchmark (problem size `s=1`) 125
5.11 SPMV benchmark (problem size `s=2`) 126
5.12 Reduction benchmark 127
5.13 VM sharing (compute phase - raw results) 128
5.14 VM sharing (compute phase - normalized results) 128

List of tables

2.1 Comparison between typical I/O virtualization techniques 85

5.1 Total operations completed after 5 min of execution 130

Αντί προλόγου

Η παρούσα διδακτορική διατριβή εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου. Περιλαμβάνει την έρευνα και τα αποτελέσματα των μεταπτυχιακών μου σπουδών στη ΣΗΜΜΥ του ΕΜΠ. Καθώς κλείνει αυτός ο κύκλος, αισθάνομαι την ανάγκη να ευχαριστήσω τους ανθρώπους που συνέβαλαν στην ολοκλήρωση αυτής της διατριβής αλλά και εκείνους που διαδραμάτισαν σημαντικό ρόλο στην εξέλιξή μου ως ερευνητή και γενικότερα ως άνθρωπο.

Αρχικά, θέλω να ευχαριστήσω ιδιαίτερα τον επιβλέποντά μου, Καθηγητή Νεκτάριο Κοζύρη, για την υποστήριξη και την εμπιστοσύνη που επέδειξε στο πρόσωπό μου καθ' όλη τη διάρκεια εκπόνησης της παρούσας εργασίας. Μου έδωσε τη δυνατότητα να συμμετέχω σε ένα καινοτόμο και υγιές ερευνητικό περιβάλλον, στο οποίο επικρατεί πνεύμα συνεργασίας και κατανόησης. Έτσι, ήλθα σε επαφή με συναρπαστικές τεχνολογίες και εργάστηκα σε ένα κλίμα γόνιμης ελευθερίας και δημιουργικότητας.

Στη συνέχεια, θα ήθελα να ευχαριστήσω τον Επίκουρο Καθηγητή Γιώργο Γκούμα, ο οποίος υπήρξε πολύτιμος αρωγός σε αυτή μου την πορεία. Ο Γιώργος με βοήθησε ουσιαστικά σε διάφορα επίπεδα κατά τα χρόνια αυτά και ήταν παρών για να με στηρίξει και να με καθοδηγήσει τόσο σε επιτυχίες και θετικές στιγμές όσο και σε αποτυχίες και απογοητεύσεις, που πάντοτε συνυπάρχουν στην ερευνητική πρόοδο μίας διδακτορικής διατριβής.

Επιπλέον, θα ήθελα να εκφράσω τις ευχαριστίες μου στον Καθηγητή Άγγελο Μπίλα, ο οποίος με βοήθησε στα τελευταία βήματα της διατριβής με τα ουσιαστικά και εποικοδομητικά του σχόλια. Θέλω, ακόμα, να ευχαριστήσω

τους Καθηγητές Παναγιώτη Τσανάκα και Νικόλαο Παπασπύρου, μέλη της τριμελούς συμβουλευτικής μου επιτροπής για τη βοήθειά τους καθ' όλη τη διάρκεια των σπουδών μου. Ευχαριστώ τον Καθηγητή Δημήτριο Σούντρη και τον Αναπληρωτή Καθηγητή Στέργιο Αναστασιάδη για τη συμμετοχή τους στην επιτροπή εξέτασης της διδακτορικής μου διατριβής.

Επιπροσθέτως, ευχαριστώ από την καρδιά μου το Δρα. Τάσο Νάνο, ο οποίος, κυρίως στα αρχικά βήματα αυτής της πορείας, αλλά και αργότερα, συνετέλεσε καθοριστικά στην εξέλιξη της διατριβής μου με το δημιουργικό του πνεύμα, τις ανεξάντλητες ιδέες του και την τεχνική του καθοδήγηση. Εξαιρετικά χρήσιμες στην αρχή του διδακτορικού υπήρξαν και οι τεχνικές αναλύσεις του Δρα. Βαγγέλη Κούκη που με βοήθησαν να σκεφτώ και να δοκιμάσω εναλλακτικά μονοπάτια.

Όλα αυτά τα χρόνια της παρουσίας μου στο Εργαστήριο Υπολογιστικών Συστημάτων συνυπήρξα με αξιόλογους ανθρώπους, παλιότερα και νεότερα μέλη του εργαστηρίου, η παρουσία των οποίων στο χώρο αυτό συνεισφέρει στη δημιουργία ενός υψηλού επιστημονικού και πνευματικού επιπέδου και στην καλλιέργεια αγαστής συνεργασίας. Στο πλαίσιο αυτό, επομένως, επιθυμώ να ευχαριστήσω τα μέλη της ερευνητικής μας ομάδας, Δημήτρη Σιακαβάρα, Χλόη Αλβέρτη, Αθηνά Ελαφρού, Στράτο Ψωμαδάκη, Δρα. Βασίλη Καρακώστα, Δρα. Νικέλα Παπαδοπούλου, Δρα. Βασίλη Καρακάση, Δρα. Κωστή Νίκα, Τάσο Κατσιγιάννη, Κωστή Παπαζαφειρόπουλο, Ορέστη Λάγκα Νικολό, Χριστίνα Γιαννούλα, Αλέξανδρο Χαριτάτο, Βαγγέλη Αγγέλου, Γιάννη Παπαδάκη, Νίκο Τριανταφύλλη, Πέτρο Αναστασιάδη καθώς και όλους τους συναδέλφους του εργαστηρίου, παλαιότερα και νεότερα μέλη, υποψήφιους διδάκτορες, μεταδιδακτορικούς ερευνητές και διοικητικούς υπαλλήλους του εργαστηρίου για τη γενικότερη συνύπαρξη, την ανταλλαγή ιδεών, τη συμβολή τους σε ερευνητικά, τεχνικά ή γραφειοκρατικά ζητήματα και τις διάφορες συζητήσεις που κάναμε εντός και εκτός του εργαστηρίου. Ακόμα, ευχαριστώ τον Βασίλη, τον Στράτο, την Χλόη, τον Κωστή καθώς και τον Χρήστο Κατσακιώρη που αφιέρωσαν χρόνο στην ανάγνωση και την αποσφαλμάτωση αυτής της διατριβής.

Ως υποψήφιος διδάκτωρ υπήρξα συνεπιβλέπων διπλωματικών εργασιών και μέσω αυτών μου δόθηκε η ευκαιρία να εξερευνήσω εναλλακτικά ερευνητικά μονοπάτια που σχετίζονται με το αντικείμενο της παρούσας διατριβής και να συνεργαστώ με αξιόλογους και ενδιαφέροντες φοιτητές. Έτσι, επιθυμώ να ευχα-

ριστήσω ιδιαίτερα τους Αντώνη Καρκατσούλη, Δημήτρη Βάσιλα, Κατερίνα Κουκίου, Μιχάλη Ροζή, Αιμίλιο Τσαλαπάτη, Κωνσταντίνο Φερτάκη και Δημήτρη Καλογερόπουλο. Επίσης, ευχαριστώ την Ιωάννα Αλιφιεράκη, που συνεισέφερε με τη διπλωματική της στη δημοσίευση εργασίας.

Κλείνοντας, δε θα μπορούσε να λείπει από αυτές τις γραμμές η αναφορά στην οικογένειά μου. Οφείλω ευγνωμοσύνη στον πατέρα μου, Σωτήρη, στη μητέρα μου Αλεξάνδρα και στην αδελφή μου Αθηνά, οι οποίοι από τα παιδικά μου χρόνια συνέβαλαν καθοριστικά στη διαμόρφωση της προσωπικότητάς μου με την ουσιαστική τους συμπαράσταση. Τέλος, ευχαριστώ το δικό μου άνθρωπο, τη σύντροφό μου, Ιωάννα, η οποία με την αγάπη της και την αστείρευτη υπομονή και επιμονή της κατάφερνε να με βοηθήσει να αντιμετωπίσω τις δύσκολες στιγμές σε αυτή την πορεία. Χωρίς την παρουσία της στη ζωή μου, δε θα μπορούσα να συνεχίσω.

Γλωσσάριο τεχνικών όρων

Αγγλικός όρος	**Ελληνικός όρος**
Accelerator	Επιταχυντής
Benchmark	Μετροπρόγραμμα
Bus	Δίαυλος
Central Processing Unit (CPU)	Κεντρική Μονάδα Επεξεργασίας (ΚΜΕ)
Cloud Environments	Νεφοϋπολογιστικά Περιβάλλοντα
Coprocessor	Συνεπεξεργαστής
Device Driver	Οδηγός Συσκευής
Microbenchmark	Μικρο-μετροπρόγραμμα
Performance	Επίδοση
Virtual Machine (VM)	Εικονική Μηχανή
Virtualization	Εικονικοποίηση

Εκτεταμένη περίληψη

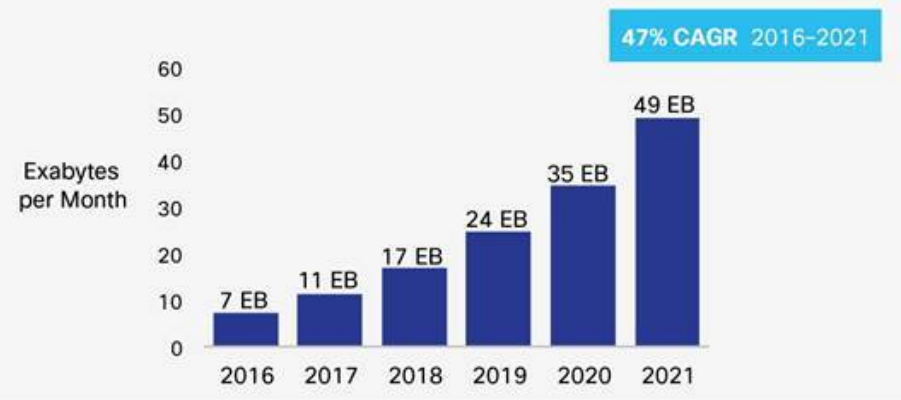
Εισαγωγή

Στο σύγχρονο διασυνδεδεμένο κόσμο, τα δεδομένα παράγονται με ταχύτατους ρυθμούς και σύμφωνα με τις προβλέψεις [35] αναμένεται ένα πιο έντονο περιβάλλον στο άμεσο μέλλον (Σχήμα 1). Σε αυτό το πλαίσιο, δημιουργείται μία ολοένα και αυξανόμενη ανάγκη για επεξεργαστική ισχύ, προκειμένου να συμβαδίσει με τον ρυθμό παραγωγής δεδομένων. Την ίδια στιγμή παρατηρείται μία επιβράδυνση στην κλιμάκωση των πολυπύρηνων αρχιτεκτονικών, η οποία αναφέρεται στη βιβλιογραφία ως η εποχή του *dark silicon* (dark silicon era) [16, 17]. Εξαιτίας αυτού του ενεργειακού προβλήματος, δημιουργείται η απαίτηση για αυξημένη επίδοση στα κέντρα δεδομένων (data centers) διατηρώντας την κατανάλωση ισχύος όσο το δυνατόν χαμηλότερη. Σε αυτή τη συνεχή προσπάθεια, η έλευση των πλατφορμών επιτάχυνσης (accelerator platforms) ή επιταχυντών (accelerators), όπως οι GPGPUs, Intel Xeon Phi, FPGAs κλπ., έχει μετασχηματίσει το υπολογιστικό τοπίο, καθώς αυτού του είδους οι συσκευές παράγουν ως αποτέλεσμα ένα πολύ καλύτερο λόγο επίδοσης ανά watt σε σχέση με τις παραδοσιακές υπολογιστικές μονάδες.

Επιπλέον, αρκετές μελέτες [4, 5] έχουν δείξει ότι, παρά την άφθονη υπολογιστική ισχύ που διατίθεται στα σύγχρονα κέντρα δεδομένων, μόλις ένα μικρό ποσοστό αυτής χρησιμοποιείται από τους παρόχους. Ένας από τους βασικούς λόγους αυτού του φαινομένου είναι η αλληλεπίδραση των διαφορετικών εφαρμογών που εκτελούνται στον ίδιο κόμβο μειώνοντας δραστικά τη συνολική επίδοση μέσω διαφόρων τρόπων, παρά το γεγονός ότι οι διαθέσιμοι υπολογιστικοί πόροι είναι υπεραρκετοί ώστε οι διαφορετικές εφαρμογές να

συνυπάρχουν [45, 13]. Επομένως, οι πάροχοι υπηρεσιών τείνουν να υποχρησιμοποιούν κάποιους από τους πόρους τους υπέρ μίας πιο σταθερής κατάστασης με μικρές μεταβολές επίδοσης για τους πελάτες τους. Οι ερευνητές έχουν προτείνει ότι μία σταδιακή μετάβαση προς ετερογενείς πλατφόρμες μπορεί να δημιουργήσει τις συνθήκες προκειμένου να ελαχιστοποιηθούν οι επιπτώσεις αυτού του προβλήματος [29, 40].

Εν τω μεταξύ, οι νεφοϋπολογιστικές τεχνολογίες ή υπολογιστικά νέφη (cloud computing) μπορούν να λειτουργήσουν ως μία πλεονεκτική προσέγγιση ως προς τη μείωση του κόστους για τους παρόχους υπηρεσιών. Καθώς η δημοφιλία των υπολογιστικών νεφών είχε αρχίσει να αναπτύσσεται τις τελευταίες δεκαετίες, υπήρξαν προσπάθειες προκειμένου τα διάφορα είδη πόρων να ενταχθούν σταδιακά στα περιβάλλοντα εικονικοποίησης (virtualization). Ερευνητές και μηχανικοί λογισμικού (software) και υλικού (hardware) άρχισαν να συσχεδιάζουν τα διάφορα υποσυστήματα εικονικοποίησης αρχικά για ΚΜΕ (CPU) και μνήμη και αργότερα για υψηλής επίδοσης συσκευές Εισόδου/Εξόδου (Ε/Ε), καθώς και αυτός ο τομέας έμοιαζε ελκυστικός από την οπτική γωνία της εικονικοποίησης.



Σχήμα 1: Η Cisco προβλέπει 49 exabytes κίνησης δεδομένων από κινητές συσκευές ανά μήνα μέχρι το 2021 (Πήγη: Cisco VNI Mobile, 2017)

Ακολουθώντας αυτή την κατεύθυνση της υπερ-σύγκλισης (hyper-convergence) σε συνδυασμό με τις προαναφερθείσες παρατηρήσεις από τα πεδία της αρχιτεκτονικής και των υπολογιστικών υποδομών, προέκυψε η ανάγκη

ένταξης των πλατφορμών επιτάχυνσης στο νεφοϋπολογιστικό οικοσύστημα. Σε αυτό το πλαίσιο, υπάρχουν σημαντικές προσπάθειες προκειμένου οι δυνατότητες των επιταχυντών να μπορούν να αξιοποιηθούν από τις εικονικές μηχανές. Τα προηγούμενα χρόνια δημοσιεύθηκε πλήθος αξιόλογων σχετικών εργασιών με επίκεντρο κυρίως τις GPUs [25, 26, 81, 71, 76, 41, 69, 62]. Επιπλέον, στον επιχειρηματικό τομέα, οι εταιρείες έχουν αρχίσει να προσφέρουν GPU-ως-υπηρεσία (GPU-as-a-service) σε πλαίσια υπολογιστικών νεφών [77, 50]. Παρ' όλα αυτά, οι τρέχουσες πρακτικές στον τομέα της παραγωγής περιλαμβάνουν συνήθως τη χρήση στατικών και λιγότερο ευέλικτων μεθόδων, όπως την *απευθείας ανάθεση συσκευής* ή τις προσεγγίσεις *IOV*.

Σε αυτή τη διατριβή, διερευνούμε τις δυνατότητες ένταξης των συσκευών επιτάχυνσης στα συστήματα εικονικοποίησης στοχεύοντας στις ακόλουθες ιδιότητες: στο διαμοιρασμό του επιταχυντή μεταξύ εικονικών μηχανών, στην ευελιξία (π.χ. στη δυνατότητα για migration ή για εφαρμογή πολιτικών χρονοδρομολόγησης), στη μείωση του συνολικού κόστους μίας νεφοϋπολογιστικής υποδομής, στη διαφάνεια, στη μικρότερη δυνατή επεμβατικότητα, στη μείωση του κόστους εικονικοποίησης και στην πιο αποτελεσματική χρησιμοποίηση των πόρων του επιταχυντή. Σε αυτό το πλαίσιο, προτείνουμε δύο συστήματα [51, 21, 22] στοχεύοντας δύο δημοφιλείς συσκευές επιτάχυνσης, τις NVIDIA GPUs και το Intel Xeon Phi, προκειμένου να αξιολογήσουμε τις προσεγγίσεις μας σε διαφορετικά περιβάλλοντα επιτάχυνσης. Εντοπίζουμε τις βασικότερες προκλήσεις εικονικοποίησης των διαφόρων υπολογιστικών πόρων επιτάχυνσης και σκιαγραφούμε τους λόγους για τους οποίους οι παραδοσιακές μέθοδοι εικονικοποίησης Ε/Ε δεν επαρκούν για την κάλυψη των εξειδικευμένων αυτών συσκευών. Επιπρόσθετα, εξετάζουμε τις δύο πλέον διαδεδομένες πλατφόρμες εικονικοποίησης (Xen και QEMU-KVM) στην ερευνητική κοινότητα, προκειμένου να διερευνήσουμε τις δυνατότητες εφαρμογής τεχνικών εικονικοποίησης επιταχυντών σε υπερεπόπτες (hypervisors) τύπου-1 και τύπου-2.

Στην πρώτη κατεύθυνση, σχεδιάζουμε το *V4VSockets* [51], ένα συμβατό με sockets, υψηλής επίδοσης σύστημα επικοινωνίας μεταξύ κόμβων (ενδοκομβικό (intra-node)) που αφορά εικονικές μηχανές στον hypervisor Xen και το χρησιμοποιούμε σε συνδυασμό με ένα δημοφιλές απομακρυσμένο σύστημα εκτέλεσης GPU εργασιών (rCUDA [62]), με στόχο να επιτύχουμε επιτάχυν-

ση εφαρμογών εντός εικονικών μηχανών με χαμηλό κόστος. Το μονοπάτι δεδομένων στο σχεδιασμό του ενδοκομβικού συστήματός μας διέρχεται μέσω του hypervisor ως δικτυακού μέσου αντί για το driver domain. Σε αντίθεση με τις αντίστοιχες προσεγγίσεις εικονικοποίησης GPU, η συνολική μας μέθοδος παρέχει τα ακόλουθα χαρακτηριστικά: επίδοση, διαφάνεια (transparency) στις εφαρμογές και συμβατότητα μεταξύ διαφορετικών εκδόσεων βιβλιοθηκών και runtimes επιταχυντών, καθώς δεν εξαρτάται από συγκεκριμένη έκδοση του συστήματος rCUDA. Τα πειραματικά αποτελέσματα δείχνουν ότι η προσέγγισή μας βελτιώνει τη μεταφορά δεδομένων έως και 6.3 φορές σε σύγκριση με το απομακρυσμένο προκαθορισμένο μονοπάτι, ενώ προσθέτει ένα κόστος 15% κατά την εκτέλεση ενός GPU υπολογιστικού πυρήνα σε σχέση με την περίπτωση απευθείας ανάθεσης συσκευής.

Η δεύτερή μας προσέγγιση, το *vPHI* [21, 22], αποτελείται από ένα σύστημα παραεικονικοποίησης (paravirtualization) που επιτρέπει στις εικονικές μηχανές να κάνουν offload εργασίες σε συσκεύες Intel Xeon Phi με διαφανή για τις εφαρμογές τρόπο. Σύμφωνα με όσα γνωρίζουμε, το vPHI είναι η πρώτη και επί του παρόντος η μοναδική προσέγγιση που παρέχει δυνατότητες διαμοιρασμού ενός συνεπεξεργαστή (coprocessor) Xeon Phi μεταξύ πολλαπλών εικονικών μηχανών που εκτελούνται στο ίδιο φυσικό μηχάνημα. Υλοποιούμε το vPHI χρησιμοποιώντας το QEMU-KVM ως τον hypervisor και προχωρούμε σε εικονικοποίηση του επιπέδου μεταφοράς της στοίβας λογισμικού συστήματος του επιταχυντή. Σε αυτό το πλαίσιο, οι βασικές αρχές της παρούσας εργασίας μπορούν να εφαρμοστούν και σε στοίβες λογισμικού μελλοντικών επεξεργαστών στην περίπτωση κατά την οποία η εν λόγω συσκευή παρέχει προγραμματιστική διεπαφή επιπέδου μεταφοράς, καθώς και σε διαφορετικές πλατφόρμες εικονικοποίησης, δεδομένου ότι ο αντίστοιχος hypervisor υποστηρίζει δυνατότητες παραεικονικοποίησης. Η πειραματική αποτίμηση δείχνει ότι το vPHI μπορεί να επιφέρει καλύτερη χρησιμοποίηση του επιταχυντή όταν αυτό χρησιμοποιείται από πολλαπλές εικονικές μηχανές, αυξάνοντας το συνολικό throughput έως 3.56x σε σχέση με μία host εφαρμογή, η οποία αναπαριστά την περίπτωση της απευθείας ανάθεσης συσκευής.

Η συνεισφορά αυτής της διατριβής συνοψίζεται στα ακόλουθα:

- Κατηγοριοποιούμε και περιγράφουμε τις παραδοσιακές μεθόδους εικονικοποίησης Ε/Ε και παρουσιάζουμε τα πλεονεκτήματα και τα μειονε-

κτήματα κάθε μιας από αυτές.

- Εντοπίζουμε τις βασικότερες προκλήσεις εικονικοποίησης των διαφόρων υπολογιστικών πόρων επιτάχυνσης και σκιαγραφούμε τους λόγους για τους οποίους οι παραδοσιακές μέθοδοι εικονικοποίησης Ε/Ε δεν επαρκούν ώστε να καλύψουν τις ανάγκες των εφαρμογών σε περιβάλλοντα διαμοιρασμού επιταχυντών.
- Παραθέτουμε τις σχετικές εργασίες στο αντίστοιχο πεδίο και σχολιάζουμε τα οφέλη αλλά και τους περιορισμούς των βασικότερων προσεγγίσεων.
- Παρουσιάζουμε τις προσεγγίσεις μας, το V4VSockets και το vPHI, διερευνώντας δύο κυρίαρχους επιταχυντές, καλύπτοντας ταυτόχρονα τις πιο δημοφιλείς πλατφόρμες εικονικοποίησης στην ερευνητική κοινότητα.
- Εκτελούμε πειράματα προκειμένου να αξιολογήσουμε τα συστήματά μας και να αναλύσουμε τη συμπεριφορά τους σε διάφορα σενάρια.
- Με βάση την πειραματική αποτίμηση, εντοπίζουμε την επιρροή των σχεδιαστικών μας επιλογών για τα προαναφερθέντα συστήματά, οι οποίες έγιναν προκειμένου να επιτυγχάνονται τα επιθυμητά χαρακτηριστικά.
- Συζητάμε τις προϋποθέσεις που τα συστήματα επιτάχυνσης επόμενης γενιάς χρειάζεται να πληρούν προκειμένου οι ερευνητές και οι σχεδιαστές συστημάτων να τα εντάξουν πιο ομαλά στις μελλοντικές στοίβες λογισμικού εικονικοποίησης.

Υπόβαθρο

Σε αυτή την ενότητα παρέχουμε συνοπτικά το απαραίτητο υπόβαθρο σχετικά με τις τεχνολογίες τις οποίες χρησιμοποιούμε στις προσεγγίσεις μας. Αρχικά, δίνουμε μία γενική εικόνα των επιταχυντών υλικού και πώς αυτή εισήγαγε την ετερογένεια στα παραδοσιακά υπολογιστικά περιβάλλοντα. Συγκεκριμένα, εστιάζουμε στις συσκευές επιτάχυνσης και στις αντίστοιχες στοίβες λογισμικού που χρησιμοποιούνται συνήθως. Στη συνέχεια, επισημαίνουμε τις δύο βασικές πλατφόρμες εικονικοποίησης εντός της ερευνητικής και ακαδημαϊκής κοινότητας και περιγράφουμε εν συντομία τις τεχνικές εικονικοποίη-

σης Ε/Ε και ποιες από αυτές είναι οι καταλληλότερες ανά περίπτωση. Με βάση αυτή την προσέγγιση, στην επόμενη ενότητα σκιαγραφούμε τις βασικές προκλήσεις ενσωμάτωσης των επιταχυντών στα περιβάλλοντα εικονικοποίησης καθώς και τους λόγους για τους οποίος οι προαναφερθείσες μέθοδοι δεν μπορούν να χρησιμοποιηθούν με τον ίδιο τρόπο με τον οποίο εφαρμόζονται στις παραδοσιακές τεχνικές εικονικοποίησης Ε/Ε.

Επιτάχυνση Υλικού

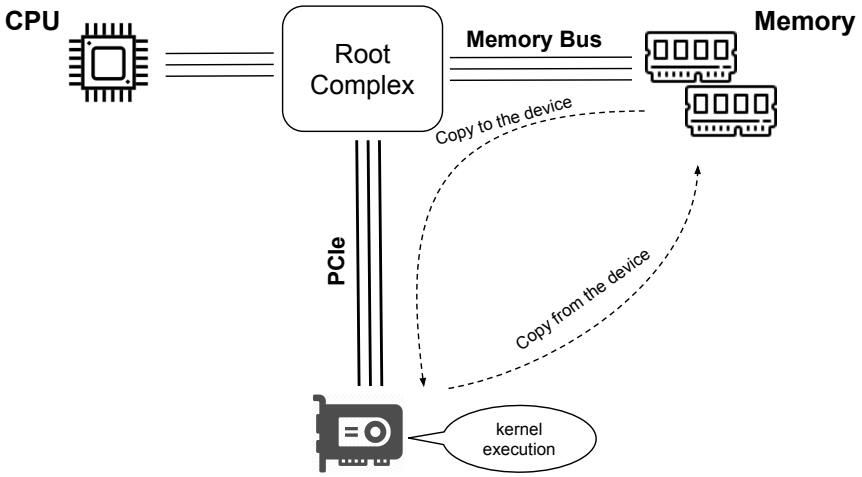
Σε ένα τυπικό υπολογιστικό σύστημα ο βασικός υπεύθυνος εκτέλεσης ενός προγράμματος είναι η ΚΜΕ (Κεντρική Μονάδα Επεξεργασίας). Η ΚΜΕ στα σύγχρονα συστήματα αποτελείται από εξαρτήματα υλικού γενικού σκοπού υποστηρίζοντας ένα ευρύ φάσμα εντολών και έτσι μπορεί να χρησιμοποιηθεί σε διαφορετικών τύπων υπολογιστικές εφαρμογές. Από την άλλη πλευρά, η επιτάχυνση υλικού επιτρέπει την εκτέλεση από εξειδικευμένες συσκευές λειτουργιών λογισμικού, οι οποίες ονομάζονται *πυρήνες* (*kernels*), συγκεκριμένων απαιτήσεων. Με βάση αυτά τα χαρακτηριστικά, οι πυρήνες αυτοί έχουν σχεδιαστεί ώστε να είναι εξαιρετικά αποδοτικοί για αντίστοιχου είδους λειτουργίες. Αυτό επιτυγχάνεται καθώς συνήθως οι επιταχυντές παρέχουν μαζικό παραλληλισμό δεδομένων για συγκεκριμένες λειτουργίες σε αντίθεση με μία, ή και περισσότερες, γενικού σκοπού ΚΜΕ. Στη συνέχεια, περιγράφουμε συνοπτικά δύο διαδομένες πλατφόρμες επιτάχυνσης στις οποίες βασιζόμαστε σε αυτή τη διατριβή.

Μονάδες Επεξεργασίας Γραφικών

Οι Μονάδες Επεξεργασίας Γραφικών (*Graphics Processing Units* (*GPUs*)) αποτελούν τυπικό παράδειγμα επιταχυντών που εκτελούν αποδοτικά λειτουργίες οι οποίες περιλαμβάνονται στους τομείς εφαρμογών επεξεργασίας γραφικών. Ιστορικά, με τις ολοένα και αυξανόμενες απαιτήσεις της αναπτυσσόμενης βιομηχανίας ηλεκτρονικών παιχνιδιών, οι μηχανικοί άρχισαν να παράγουν εξαιρετικά ισχυρές συσκευές, προκειμένου να αντιμετωπίσουν την επεξεργασία μεγάλου όγκου γραφικών δεδομένων σε σχετικά περιορισμένο χρόνο [12, 47]. Οι σχεδιαστές συστημάτων, εμπνευσμένοι από αυτές τις εξελίξεις, δημιούργησαν επιταχυντές στοχεύοντας επίσης και το HPC πεδίο κα-

θώς και κάποιες από τις επιστημονικές εφαρμογές έντασης δεδομένων (data-intensive). Έτσι, περισσότερο από μία δεκαετία πριν, οι εταιρείες ξεκίνησαν να κυκλοφορούν Γενικού Σκοπού GPUs (*General Purpose GPUs* (*GPGPUs*)) με την αντίστοιχη στοίβα λογισμικού.

Συνήθως, η τυπική offload σημασιολογία που διαφαίνεται στον προγραμματιστή εφαρμογών συμπεριλαμβάνει (Σχήμα 2) i) τη μεταφορά δεδομένων από την κύρια μνήμη στη μνήμη του επιταχυντή πάνω από το δίαυλο περιφερειακών (π.χ. PCIe), στον οποίο η συσκευή είναι συνδεδεμένη, ii) τον υπολογιστικό πυρήνα που πρόκειται να εκτελεστεί από τη συσκευή επιτάχυνσης και iii) τη μεταφορά των αποτελεσμάτων πίσω στην κύρια μνήμη προκειμένου η ΚΜΕ να έχει πρόσβαση σε αυτά. Όπως είναι αναμενόμενο, αυτές οι δύο μεταφορές δεδομένων έχουν ένα σημαντικό κόστος επίδοσης και χρειάζεται να λαμβάνονται υπόψη όταν γίνεται σύγκριση της επίδοσης κάποιας εφαρμογής που έχει εκτελεστεί στην ΚΜΕ και στη GPU αντίστοιχα.

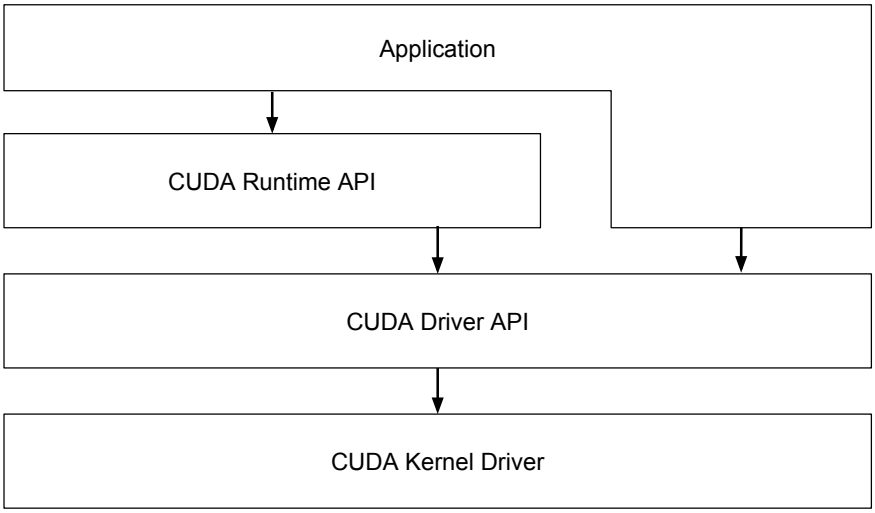


Σχήμα 2: Τυπικό offload μονοπάτι για συσκευές επιτάχυνσης

CUDA

Σε αυτό το πλαίσιο, η NVIDIA παρέχει τη δική της προγραμματιστική διεπαφή, που ονομάζεται *CUDA* [53] προκειμένου οι NVIDIA GPUs να χρησιμοποιηθούν για offload εργασιών. Στην πρώτη από τις προσεγγίσεις μας βα-

σιζόμαστε στο CUDA, καθώς πρόκειται για μία από τις κυρίαρχες πλατφόρμες στο πεδίο offload εργασιών. Παρ' όλα αυτά, η βασική ιδέα μπορεί να εφαρμοστεί και σε αντίστοιχες στοίβες λογισμικού που ακολουθούν αυτό το μοντέλο offload εκτέλεσης. Μία εικόνα της στοίβας λογισμικού CUDA φαίνεται στο Σχήμα 3.



Σχήμα 3: Στοίβα λογισμικού CUDA

rCUDA

Το rCUDA [62] είναι ένα ενδιάμεσο σύστημα το οποίο επιτρέπει υπολογιστικούς κόμβους που δεν είναι εξοπλισμένοι με GPUs να έχουν απομακρυσμένη πρόσβαση σε GPUs πάνω από δίκτυο (Σχήμα 4). Με αυτό τον τρόπο, μία εφαρμογή γραμμένη σε CUDA μπορεί να εκκινήσει σε ένα υπολογιστικό κόμβο χωρίς συσκευές GPU και τελικά να εκτελεστεί σε έναν απομακρυσμένο κόμβο εξοπλισμένο με GPU. Σε αυτή τη διαδικασία, δεν απαιτούνται αλλαγές στον κώδικα, αλλά ούτε και στο εκτελέσιμο αρχείο. Σε αυτό το πλαίσιο, ο αρχικός κόμβος λειτουργεί ως rCUDA πελάτης (client), ενώ ο κόμβος που είναι εξοπλισμένος με GPU λειτουργεί ως rCUDA εξυπηρετητής (server).

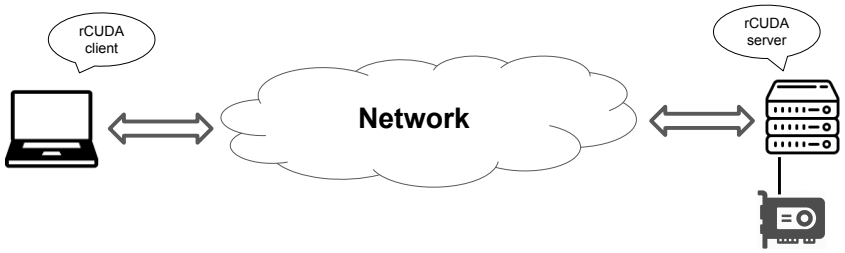
Στην πρώτη μας προσέγγιση, χρησιμοποιούμε το rCUDA σε εικονικοποιημένο περιβάλλον με σκοπό να παρέχουμε CUDA συμβατότητα στις υπάρ-

χουσες προμεταγλωττισμένες εφαρμογές. Οι συγγραφείς του rCUDA [62] παρουσιάζουν μία ανάλυση επίδοσης, σύμφωνα με την οποία η παραδοσιακή δικτυακή TCP/IP στοίβα συνεπάγεται ένα υψηλό κόστος για τις εφαρμογές, μετατρέποντας τη χρήση απομακρυσμένων GPU σε μη βιώσιμη λύση για τα δίκτυα TCP/IP/Ethernet. Έτσι, σχεδίασαν το σύστημά τους με ένα δομοστοιχειωτό τρόπο (modular) ως προς το μηχανισμό επικοινωνίας και προτείνουν τη χρήση διασυνδέσεων (interconncets) χαμηλού latency και υψηλού throughput, όπως το Infiniband [36], προκειμένου να μειώσουν αισθητά το δικτυακό κόστος. Στην πρώτη μας προσέγγιση, κάνουμε χρήση του rCUDA σε εικονικοποιημένο περιβάλλον και μειώνουμε το μονοπάτι επικοινωνίας μεταξύ των εικονικών μηχανών χρησιμοποιώντας το V4VSockets ως ένα χαμηλού κόστους ενδοκομβικό μηχανισμό επικοινωνίας, προκειμένου οι εικονικές μηχανές να κάνουν offload σε GPUs με βιώσιμο τρόπο.

Intel Xeon Phi

Στις αρχές της τρέχουσας δεκαετίας, η Intel ανακοίνωσε την κυκλοφορία της δικής της συσκευής επιτάχυνσης, την οποία ονόμασε Xeon Phi [39]. Το Xeon Phi ουσιαστικά αναφέρεται σε μία οικογένεια επεξεργαστών οι οποίοι κάνουν χρήση της αρχιτεκτονικής MIC (Many Integrated Core) της Intel. Αποτελείται από μία σειρά από πολυπύρηνους (manycore) επεξεργαστές και μπορεί να χρησιμοποιηθεί ως συνεπεξεργαστής για να επιταχύνει ένα σύστημα, ή ακόμα και ως βασικό επεξεργαστής (π.χ. Knights Landing).

Το βασικό ανταγωνιστικό πλεονέκτημα από την οπτική γωνία του προγραμματιστή εφαρμογών σε σχέση με τις GPUs της NVIDIA είναι ότι η Intel παρέχει x86 συμβατότητα και έτσι εφαρμογές οι οποίες έχουν γραφτεί για ένα



Σχήμα 4: Σενάριο χρήσης rCUDA

x86 επεξεργαστή μπορούν να εκτελεστούν χωρίς αλλαγές στον κώδικα σε ένα Xeon Phi συνεπεξεργαστή. Στην πράξη, πάρα ταύτα, οι προγραμματιστές χρειάζεται προσεκτικά να μεταβάλουν ελαφρώς τις εμπλεκόμενες εφαρμογές ή βιβλιοθήκες, προκειμένου να αξιοποιήσουν πλήρως τον παραλληλισμό που ο συνεπεξεργαστής προσφέρει και έτσι να επιτύχουν ανάλογη επίδοση [11, 65, 68, 15].

Η δεύτερη προσέγγιση που παρουσιάζουμε σε αυτή τη διατριβή βασίζεται στο Xeon Phi της Intel ως συσκευή επιτάχυνσης. Προκειμένου να γίνουν κατανοητά τα ζητήματα σχεδιασμού και υλοποίησης της εργασίας μας, στη συνέχεια αναφέρουμε συνοπτικά τις βασικές ιδέες και τα κύρια εσωτερικά χαρακτηριστικά του μοντέλου εκτέλεσης του Xeon Phi καθώς και της αντίστοιχης στοίβας λογισμικού.

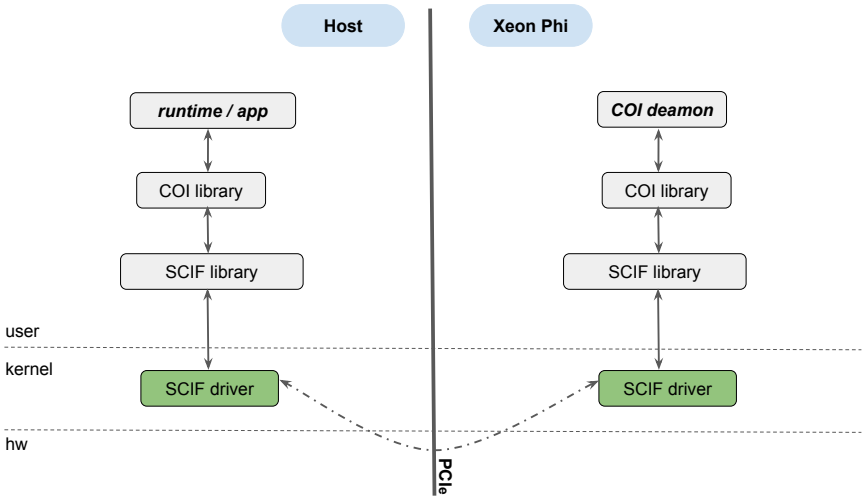
Μοντέλο εκτέλεσης του Xeon Phi

Η Intel ορίζει ένα μοντέλο εκτέλεσης το οποίο υποστηρίζει τρία σχήματα προγραμματισμού προκειμένου να ικανοποιήσει τις αντίστοιχες ανάγκες ανάλογα με το εκάστοτε σενάριο χρήσης: το *γηγενές* (*native*) σχήμα, το *offload* και το *συμμετρικό* (*symmetric*). Στο native σχήμα ο χρήστης παρέχει το εκτελέσιμο αρχείο απευθείας στην κάρτα Xeon Phi. Το offload σχήμα επιτρέπει στο χρήστη να εκτελέσει μία εφαρμογή στον host και να κάνει offload στο συνεπεξεργαστή κάποια απαιτητικά ως προς τον υπολογισμό κομμάτια κάνοντας χρήση των αντίστοιχων κατευθυντήριων οδηγιών (directives) σε κάποιο περιβάλλον, π.χ. *OpenMP*. Τέλος, στο συμμετρικό σχήμα το Xeon Phi μπορεί να αντιμετωπιστεί ως ένας ανεξάρτητος κόμβος και με αυτό τον τρόπο ο χρήστης μπορεί να φορτώσει κάποιες διεργασίες από την ίδια παράλληλη εφαρμογή στην πλευρά του host και κάποιες άλλες στον επιταχυντή, χρησιμοποιώντας, για παράδειγμα, το *MPI*.

Στοίβα λογισμικού συστήματος του Xeon Phi

Οι συσκευές Xeon Phi είναι διασυνδεδεμένες με το σύστημα μέσω του διαύλου PCIe. Η Intel παρέχει το *SCIF* (Symmetric Communication Interface), ένα αφαιρετικό στρώμα χαμηλού επιπέδου πάνω από το PCIe, προκειμένου να επιτρέπει στα ανώτερα στρώματα να αξιοποιήσουν τις δυνατότητες DMA του Xeon Phi χωρίς να εμπλακούν απευθείας με τις λειτουργίες PCIe. Με το vPHI,

ουσιαστικά παρέχουμε ένα σχήμα εικονικοποίησης του SCIF, προκειμένου να ενεργοποιήσουμε την ίδια λειτουργικότητα για εικονικές μηχανές.



Σχήμα 5: Επικοινωνία μεταξύ host και Xeon Phi

Το Σχήμα 5 απεικονίζει τη γενική αρχιτεκτονική της στοίβας λογισμικού συστήματος η οποία χρησιμοποιείται σε ένα τυπικό μη-εικονικοποιημένο περιβάλλον Xeon Phi. Χρησιμοποιώντας το SCIF, οι εφαρμογές που τρέχουν τόσο στον host όσο και στη συσκευή μπορούν να επικοινωνήσουν μεταξύ τους με την ίδια προγραμματιστική διεπαφή. Προκειμένου να επιτευχθεί κάτι τέτοιο, το Xeon Phi από την πλευρά του εκκινεί ένα ελαφρύτερο λειτουργικό σύστημα (*uOS*), το οποίο αποτελείται από έναν τροποποιημένο πυρήνα Linux, που περιέχει έναν οδηγό (driver) SCIF. Επίσης, για να διαφανεί η διεπαφή SCIF, γίνεται χρήση μιας βιβλιοθήκης SCIF (`libscif`) από τις ανώτερες βιβλιοθήκες ή runtimes. Επιπλέον, η στοίβα λογισμικού συστήματος του Xeon Phi περιλαμβάνει έναν εξομοιωμένο οδηγό δικτύου ως μέρος του uOS, ο οποίος κάνει χρήση του SCIF, και επιτρέπει στους χρήστες να κάνουν χρήση δικτυακών εργαλείων (π.χ. `ssh`) και να συνδέονται στη συσκευή Xeon Phi. Με αυτό τον τρόπο, μπορούν να εκτελούν εφαρμογές στο συνεπεξεργαστή κάνοντας χρήση ενός φλοιού (shell). Αντίστοιχα με τη Xeon Phi κάρτα, τα αντίστοιχα υποσυστήματα, όπως η libscif και ο οδηγός SCIF, έχουν υλοποιηθεί και στην πλευρά του host.

Εικονικοποίηση υλικού

Η εικονικοποίηση υλικού αναφέρεται στη διαδικασία που έχει στόχο την παρουσίαση μίας πλατφόρμας υλικού η οποία υπάρχει και λειτουργεί εσωτερικά σε επίπεδο λογισμικού. Η παραγόμενη εικονικοποιημένη οντότητα ονομάζεται *εικονική μηχανή* (*virtual machine* (VM)), *domain* ή *επισκέπτης* (*guest*), ανάλογα με το αναφερόμενο πλαίσιο, και η αντίστοιχη οντότητα λογισμικού που είναι υπεύθυνη για την υλοποίηση της διαδικασίας εικονικοποίησης λέγεται *υπερεπόπτης* (*hypervisor*), *ελεγκτής εικονικών μηχανών* (*virtual machine monitor* (*VMM*)) ή ευρύτερα *ξενιστής* (*host*). Έτσι σε έναν hypervisor μπορούν να εκτελούνται πολλαπλές εικονικές μηχανές.

Σε αυτή τη διατριβή, χτίζουμε τα συστήματά μας για δύο από τους πιο γνωστούς hypervisors στην ερευνητική κοινότητα, δηλαδή τον *Xen* [3] και το *KVM* [43]. Αυτοί οι hypervisors είναι βασισμένοι σε διαφορετικές αρχιτεκτονικές λογισμικού και σχεδιαστικές αρχές. Σε εννοιολογικό επίπεδο, ο Xen είναι ένας hypervisor *τύπου-1*, το οποίο σημαίνει ότι εκτελείται απευθείας στο φυσικό υλικό, ενώ το KVM ανήκει στην κατηγορία hypervisors *τύπου-2*, το οποίο σημαίνει ότι εκτελείται ως μέρος του λειτουργικού συστήματος του host. Πέραν της δημοφιλίας τους, επιλέξαμε να εργαστούμε με διαφορετικούς hypervisors προκειμένου να διαπιστώσουμε τις προκλήσεις της ενσωμάτωσης της λογικής των επιταχυντών σε ποικιλόμορφες πλατφόρμες εικονικοποίησης. Στη συνέχεια, αναφερόμαστε σχηματικά στην αρχιτεκτονική κάθε ενός από αυτά τα δύο περιβάλλοντα εικονικοποίησης καθώς και σε κάποιες βασικές αρχές λειτουργίας τους. Πριν από αυτό, παραθέτουμε περιληπτικά το ουσιώδες υπόβαθρο σχετικά με τις διαφορετικές προσεγγίσεις εικονικοποίησης υλικού, εστιάζοντας κατά κύριο λόγο στην εικονικοποίηση Ε/Ε. Με βάση αυτή τη συζήτηση, επισημαίνουμε σε επόμενη ενότητα τις προκλήσεις εμπλουτισμού των περιβαλλόντων εικονικοποίησης με σημασιολογία επιταχυντών.

Εξέλιξη της εικονικοποίησης

Η ιστορία εικονικοποίησης υλικού περιλαμβάνει ένα ευρύ σύνολο από εφαρμοσμένες τεχνικές και προσεγγίσεις με σκοπό να διατηρηθεί η διαφάνεια και να μεγιστοποιηθεί η επίδοση για τις εφαρμογές. Σε αυτό το πλαίσιο, εστιάζουμε σε τρία υποσυστήματα εικονικοποίησης: την ΚΜΕ, την κύρια μνήμη και

την Ε/Ε και σχολιάζουμε τι είδους πρακτικές είναι αυτές που τελικά επικράτησαν σε κάθε ένα από αυτά, ενώ συζητάμε επίσης τις ομοιότητες και τις διαφορές μεταξύ τους καθώς και τους αντίστοιχους συμβιβασμούς, που χρειάζεται να γίνουν σχεδόν πάντοτε στις διαδικασίες σχεδιασμού υπολογιστικών συστημάτων.

Ιστορικά, οι πρώτες απόπειρες εικονικοποίησης υλικού στράφηκαν στην ΚΜΕ, καθώς προέκυπταν ζητήματα επίλυσης των μη-εικονικοποιήσιμων χαρακτηριστικών [20] συγκεκριμένων ISA (π.χ. x86). Τέτοιες προσεγγίσεις περιλαμβάνουν την *πλήρη εξομοίωση επεξεργαστή* (*full processor emulation*), τη *μετάφραση του binary* (*binary translation*) και την *υποβοηθούμενη από το υλικό εικονικοποίηση* (*hardware-assisted virtualization*). Οι πρώτες δύο, όντας προσεγγίσεις λογισμικού, συνεπάγονται μεγάλο επεξεργαστικό κόστος εξαιτίας του μεγάλου αριθμού από traps όταν εκτελείται κώδικας guest (emulation) ή κατά το συνεχή έλεγχο και μετάφραση συγκεκριμένων κομματιών από εντολές (binary translation). Το πλεονέκτημα τους είναι ότι παλαιότερες ΚΜΕ που δε γνωρίζουν ότι υπόκεινται εικονικοποίηση μπορούν με τη χρήση κάποιας από αυτές τις τεχνικές να χρησιμοποιηθούν σε περιβάλλοντα εικονικοποίησης. Από την άλλη πλευρά, με την υποβοηθούμενη από το υλικό εικονικοποίηση, η ΚΜΕ πρέπει βάσει σχεδιασμού να έχει γνώση της ύπαρξης οντοτήτων εικονικοποίησης. Με αυτό τον τρόπο, αυτή η τεχνική συνεπάγεται εξαιρετική επίδοση, αλλά απαιτεί την ύπαρξη επεκτάσεων εικονικοποίησης στην ΚΜΕ. Καθώς αρκετές εταιρείες επεξεργαστών (π.χ. Intel, AMD) πλέον ενσωματώνουν σε κάθε περίπτωση αυτά τα χαρακτηριστικά στα chips τους, η υποβοηθούμενη από το υλικό εικονικοποίηση έχει επικρατήσει ως η πλέον διαδεδομένη λύση για νεφοϋπολογιστικές υπηρεσίες προσπερνώντας τις προσεγγίσεις λογισμικού.

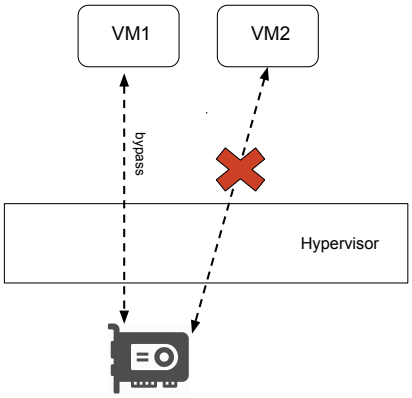
Συνεχίζοντας την αναδρομή στην εξέλιξη της εικονικοποίησης του υλικού, το επόμενο βήμα για τους σχεδιαστές συστημάτων ήταν η εικονικοποίηση της διαχείρισης μνήμης για πολλαπλές εικονικές μηχανές μειώνοντας ταυτόχρονα το κόστος της τροποποίησης του πίνακα σελίδων για κάθε εικονική μηχανή. Ακολουθώντας το παράδειγμα της ΚΜΕ, οι τεχνικές λογισμικού προϋπήρξαν των λύσεων σε επίπεδο υλικού. Οι *σκιώδεις πίνακες σελίδων* (*shadow page tables*) επιτρέπουν στον ελεγκτή εικονικών μηχανών να εποπτεύει τις αλλαγές στους πίνακες σελίδων υλοποιώντας τις αντίστοιχες δομές σε επίπεδο

λογισμικού. Αργότερα, οι εταιρείες εισήγαγαν ενισχυμένες τεχνικές σε επίπεδο υλικού προσθέτοντας γνώση στη Μονάδα Διαχείρισης Μνήμης (Memory Management Unit (MMU)) σχετικά με την ύπαρξη ξεχωριστών πινάκων σελίδων ανά εικονική μηχανή (π.χ. *εμφωλευμένοι* (*nested*)/*εκτεταμένοι* (*extended*) πίνακες σελίδων (page tables)). Αντίστοιχα με την περίπτωση της ΚΜΕ, οι προσεγγίσεις σε επίπεδο υλικού θεωρούνται πλέον οι de-facto μηχανισμοί στις σύγχρονες νεφοϋπολογιστικές υποδομές.

Τεχνικές εικονικοποίησης Ε/Ε

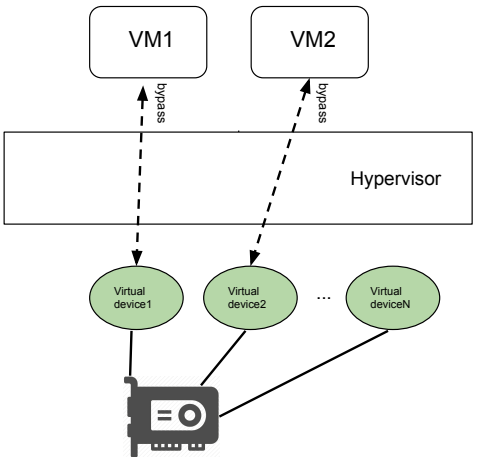
Ενώ τα μοντέλα εικονικοποίησης της ΚΜΕ και της ΜΜU προσφέρονται για εφαρμογές που είναι απαιτητικές σε υπολογισμό, με την έλευση των νεφοϋπολογιστικών υπηρεσιών, προέκυψε η ανάγκη για εικονικοποίηση πόρων Ε/Ε με ένα αποδεκτό άνω όριο κόστους. Σε αυτό το πλαίσιο, αρκετές τεχνικές προτάθηκαν ισορροπώντας μεταξύ αφενός του κόστους εικονικοποίησης και αφετέρου της πολυπλοκότητας και της διαφάνειας του κώδικα. Παρακάτω, αναφέρουμε επιγραμματικά τις υπάρχουσες προσεγγίσεις για εικονικοποίηση συσκευών Ε/Ε και παρουσιάζουμε τα πλεονεκτήματα και τα μειονεκτήματα κάθε μεθόδου. Έτσι, κάνουμε την ακόλουθη κατηγοριοποίηση:

- **Απευθείας ανάθεση συσκευής** (**Direct device assignment**) (Σχήμα 6): η συσκευή ανατίθεται απευθείας και αποκλειστικά σε μία εικονική μηχανή, αποτρέποντας τις υπόλοιπες εικονικές μηχανές να έχουν πρόσβαση σε αυτή τη συσκευή.

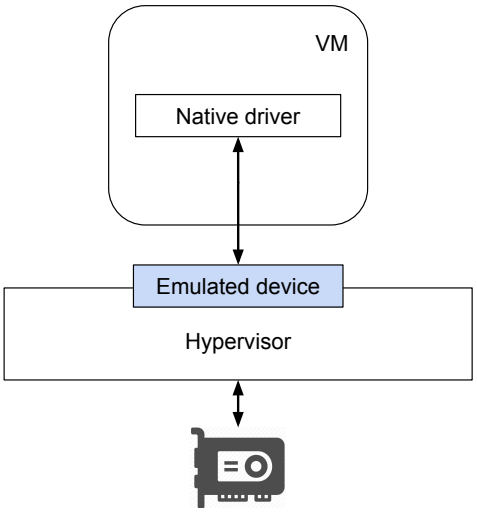


Σχήμα 6: Απευθείας ανάθεση συσκευής (Direct device assignment)

- **Συσκευές IOV** (**IOV-enabled devices**) (Σχήμα 7): η φυσική συσκευή καθεαυτή διαθέτει ένα μέγιστο αριθμό από εικονικές συσκευές και κάθε μία από αυτές μπορεί να ανατεθεί απευθείας σε κάποια εικονική μηχανή.



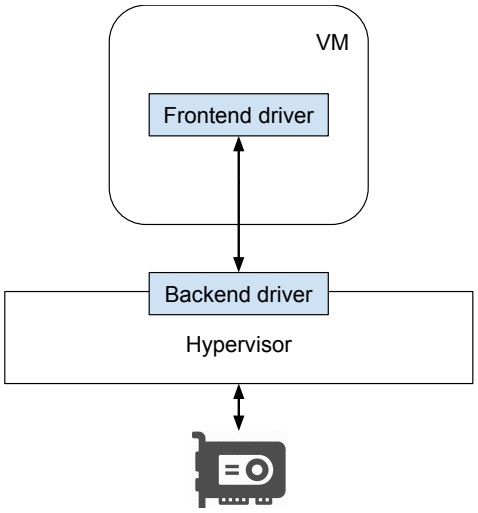
Σχήμα 7: Συσκευές IOV (IOV-enabled device)



Σχήμα 8: Εξομοίωση Ε/Ε (I/O emulation)

- **Εξομοίωση Ε/Ε** (**I/O emulation**) (Σχήμα 8): ο οδηγός στην πλευρά του guest παραμένει αμετάβλητος, ενώ υλοποιείται ένα backend στον host, το οποίο εξομοιώνει τη λειτουργικότητα που αναμένει ο guest.
- **Παραεικονικοποίηση** (**Paravirtualization**) (Σχήμα 9): ένας οδηγός (frontend)

με γνώση ότι υπόκειται εικονικοποίηση εισάγεται στον guest και ο αντίστοιχος οδηγός backend εκτελείται στην πλευρά του host.



Σχήμα 9: Παραεικονικοποίηση (Paravirtualization)

Μία συμπτυγμένη σύγκριση των πλεονεκτημάτων και των μειονεκτημάτων των προαναφερθεισών τεχνικών εικονικοποίησης Ε/Ε συνοψίζεται στον Πίνακα 1.

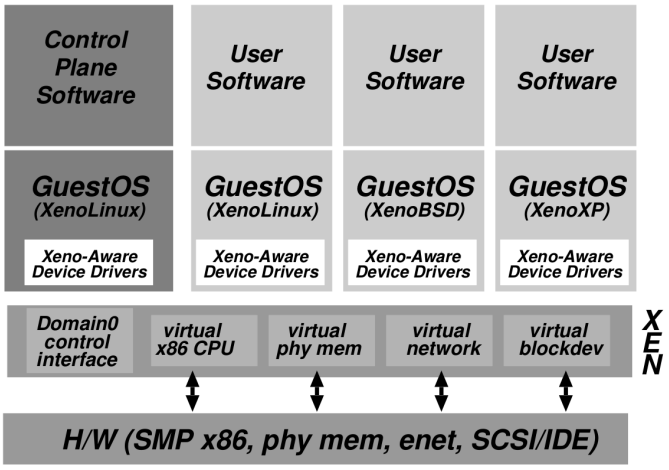
Πίνακας 1: Σύγκριση μεταξύ των τυπικών τεχνικών εικονικοποίησης Ε/Ε

	Πλεονεκτήματα	**Μειονεκτήματα**
Απευθείας ανάθεση συσκευής	- Παρόμοια επίδοση με τη native περίπτωση	- Δεν υποστηρίζεται διαμοιρασμός συσκευών - Ζητήματα ελαστικότητας και κλιμακωσιμότητας - Περιορισμένη υποστήριξη για migration
Συσκευή IOV	- Παρόμοια επίδοση με τη native περίπτωση - Υποστηρίζεται διαμοιρασμός	- Απαιτείται εξειδικευμένο υλικό - Ζητήματα ελαστικότητας και κλιμακωσιμότητας - Περιορισμένη υποστήριξη για migration
Εξομοίωση Ε/Ε	- Μη τροποποιημένο guest λειτουργικό σύστημα - Υποστηρίζεται διαμοιρασμός	- Χαμηλή επίδοση
Παραεικονικοποίηση	- Σημαντικά χαμηλότερο κόστος εικονικοποίησης - Υποστηρίζεται διαμοιρασμός	- Απαιτείται τροποποίηση στο guest λειτουργικό σύστημα

Στη συνέχεια, περιγράφουμε συνοπτικά από την οπτική γωνία της αρχιτεκτονικής λογισμικού τις δύο πλατφόρμες εικονικοποίησης τις οποίες χρησιμοποιούμε σε αυτή τη διατριβή.

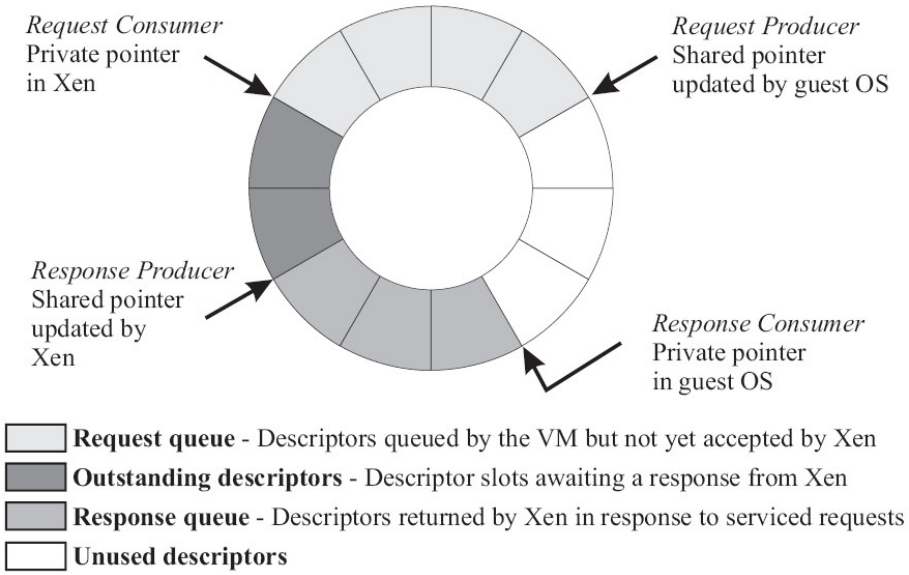
Πλατφόρμα εικονικοποίησης Xen

Η γενική αρχιτεκτονική του hypervisor Xen [3] απεικονίζεται στο Σχήμα 10. Ο Xen βασίζεται στην έννοια της *παραεικονικοποίησης* (*paravirtualization* (PV)) ειδικά για το μονοπάτι Ε/Ε. Σύμφωνα με αυτή τη μέθοδο, ο Xen κάνει χρήση ενός μοντέλου split-driver, σύμφωνα με το οποίο ένας οδηγός *frontend* εκτελείται στη guest εικονική μηχανή παρέχοντας στο χώρο χρήστη ή στο χώρο πυρήνα μία διεπαφή ανά κλάση συσκευών και ο αντίστοιχος οδηγός *backend* ο οποίος εκτελείται από προνομιούχες guest εικονικές μηχανές που ονομάζονται *driver domains*. Με αυτό τον τρόπο, τα driver domains χειρίζονται την πρόσβαση σε δεδομένα και έτσι ουσιαστικά επιτρέπουν στις εικονικές μηχανές να αλληλεπιδρούν με το υλικό. Συνήθως, στα περιβάλλοντα Xen υπάρχει ένα driver domain, που ονομάζεται *dom0*. Οι μη-προνομιούχες εικονικές μηχανές, τα *domUs*, επικοινωνούν με το dom0 για να χρησιμοποιήσουν το υλικό.



Σχήμα 10: Αρχιτεκτονική λογισμικού Xen (Πηγή [3])

Στον Xen η μνήμη εικονικοποιείται προκειμένου να παρέχει συνεχόμενες περιοχές στα λειτουργικά συστήματα που εκτελούνται στα guest domains. Αυτό επιτυγχάνεται προσθέτοντας ένα αφαιρετικό επίπεδο μνήμης ανά domain, το οποίο ονομάζεται *ψευδο-φυσική μνήμη* (*pseudo-physical memory*). Επομένως, στον Xen, η *μνήμη μηχανής* (*machine memory*) αναφέρεται στη φυσική μνήμη ολόκληρου του συστήματος, ενώ η ψευδο-φυσική μνήμη αναφέρεται στη φυσική μνήμη την οποία αντιλαμβάνεται το κάθε guest λειτουργικό σύστημα.



Σχήμα 11: Δομή δακτυλίων Xen (Πήγη [3])

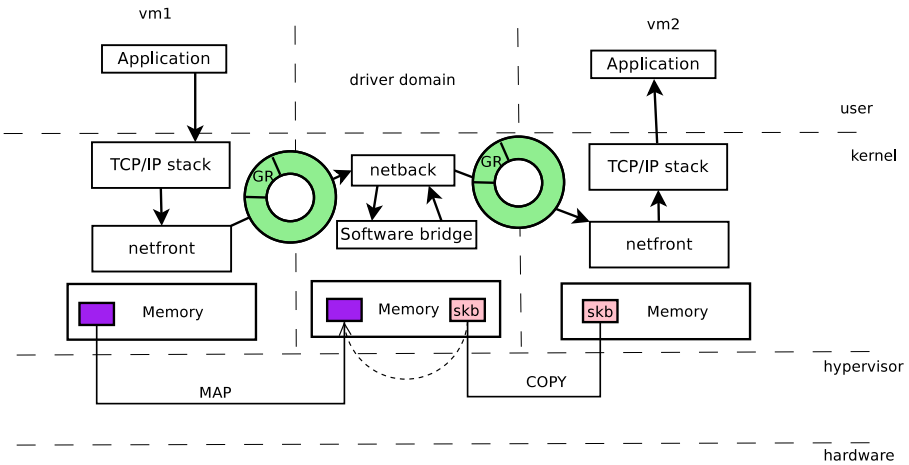
Προκειμένου τα guest domains να διαμοιράζονται σελίδες, ο Xen εξάγει έναν μηχανισμό *παραχώρησης* (*grant*). Τα grants του Xen αποθηκεύονται σε *πίνακες παραχώρησης* (*grant tables*) και παρέχουν ένα γενικό μηχανισμό διαμοιρασμού μνήμης μεταξύ των domains. Οι οδηγοί συσκευών δικτύου βασίζονται σε αυτόν το μηχανισμό για να ανταλλάσσουν πληροφορίες ελέγχου και δεδομένα. Δύο guests εγκαθιστούν μεταξύ τους ένα *κανάλι συμβάντων* (*event channel*), το οποίο προκαλεί την εκτέλεση των αντίστοιχων χειριστών. Οι *δακτύλιοι E/E* (*I/O rings*) (Σχήμα 11) είναι κυκλικοί buffers, που αποτελούν μία τυπική δομή χωρίς ανάγκη κλειδώματος για επικοινωνία παραγωγού-καταναλωτή. Μέσω των δακτυλίων E/E, ο Xen παρέχει μία απλή αφαιρετική

δομή ανταλλαγής μηνυμάτων πάνω από τους μηχανισμούς των grant και event channel. Καθώς με το V4VSockets βελτιστοποιούμε την επικοινωνία μεταξύ εικονικών μηχανών εντός του ίδιου κόμβου, περιγράφουμε εν συντομία στις επόμενες παραγράφους το προκαθορισμένο δικτυακό μονοπάτι για την ενδοκομβική επικοινωνία.

Δικτυακή Ε/Ε παραεικονικοποίησης (PV) στον Xen

Η πιο κοινή μέθοδος επικοινωνίας των εικονικών μηχανών στον Xen είναι μέσω της δικτυακής αρχιτεκτονικής παραεικονικοποίησης (PV). Στις guest εικονικές μηχανές εκτελείται ο αντίστοιχος οδηγός *netfront*, ο οποίος εξάγει μία γενική διεπαφή Ethernet στο χώρο πυρήνα. Στο driver domain εκτελείται ο αντίστοιχος εξειδικευμένος οδηγός υλικού και ο οδηγός *netback*, ο οποίος επικοινωνεί με το frontend μέσω του μηχανισμού event channel και εισάγει πλαίσια (frames) σε μία γέφυρα λογισμικού (software bridge).

Τα δεδομένα εισέρχονται και εξέρχονται από την εικονική μηχανή κάνοντας χρήση του μηχανισμού grant, ενώ οι ειδοποιήσεις υλοποιούνται με τη χρήση των event channels, του εικονικού μηχανισμού IRQ που παρέχει ο Xen. Οι λιγότερο κρίσιμες λειτουργίες υλοποιούνται από το Xenstore (αρίθμηση διεπαφών, ανταλλαγή χαρακτηριστικών κλπ.).



Σχήμα 12: Γενικός μηχανισμός ενδοκομβικής επικοινωνίας στον Xen

Προκειμένου οι εικονικές μηχανές που συνυπάρχουν στο ίδιο φυσικό μη-

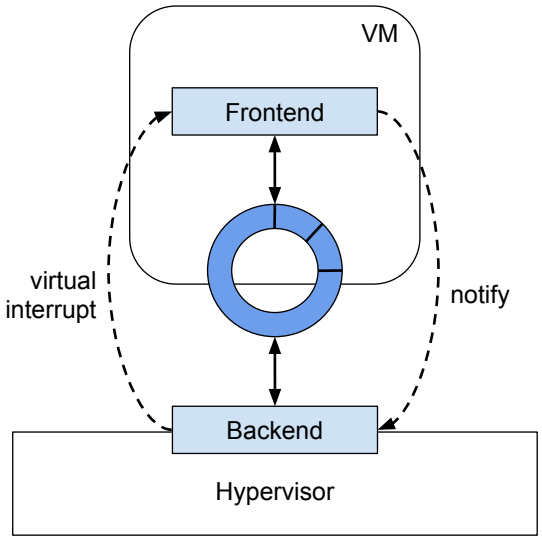
χάνημα να επικοινωνούν μεταξύ τους, χρειάζεται να περάσουν από το software bridge του driver domain. Στο Σχήμα 12 απεικονίζεται το μονοπάτι δεδομένων δύο τέτοιων εικονικών μηχανών που ανταλλάσσουν δεδομένα. Η κίνηση των δεδομένων πραγματοποιείται με τη χρήση διαμοιραζόμενων σελίδων που εγκαθίστανται χρησιμοποιώντας το μηχανισμό grant. Κάθε αίτηση μετάδοσης περιέχει μία αναφορά grant και ένα offset εντός της διαμοιραζόμενης σελίδας. Αυτό επιτρέπει αποσταλθέντες και ληφθέντες buffers να επαναχρησιμοποιηθούν, αποτρέποντας συχνές ανανεώσεις του TLB (Translation Lookaside Buffer). Κατά τη λήψη πακέτων, το guest domain εισάγει μία αίτηση λήψης στο δακτύλιο, υποδεικνύοντας πού να αποθηκευτεί το πακέτο, και το driver domain τοποθετεί εκεί τα αντίστοιχα περιεχόμενα.

Πλατφόρμα εικονικοποίησης KVM

Όπως αναφέραμε πρωτύτερα, ο hypervisor KVM [43] εκτελείται σε ένα λειτουργικό σύστημα Linux. Έτσι, αυτό το λειτουργικό σύστημα μπορεί να υποστηρίξει τόσο διεργασίες όσο και πλήρεις εικονικές μηχανές. Σε αυτό το πλαίσιο, το KVM αποτελείται από ένα module πυρήνα, με το οποίο δημιουργείται μία εικονική συσκευή (συγκεκριμένα η `/dev/kvm`) και εξάγει έναν αριθμός από εντολές `ioctl` για αυτή τη συσκευή. Για παράδειγμα, εκτελώντας την ioctl κλήση συστήματος `KVM_CREATE_VM` στη συσκευή KVM, δημιουργείται ένα νέος guest, ο οποίος από την πλευρά του host δεν είναι τίποτα περισσότερο από μία διεργασία χρήστη με το δικό της εικονικό χώρο διευθύνσεων. Από την οπτική γωνία του guest, αυτός ο χώρος διευθύνσεων μεταφράζεται σε αυτό που αποκαλείται φυσικός χώρος διευθύνσεων του guest και αντιμετωπίζεται ως τυπική φυσική μνήμη. Το KVM κάνει χρήση των δυνατοτήτων εικονικοποίησης του υλικού (π.χ. επεκτάσεις Intel VT, AMD-V) προκειμένου να εικονικοποιήσει τις λειτουργίες της ΚΜΕ και της MMU. Το KVM χρησιμοποιείται κυρίως μαζί με το QEMU [6], το οποίο αποτελεί έναν εξομοιωτή που εκτελείται στον host και προκαλεί κλήσεις συστήματος στο `/dev/kvm`.

Στο vPHI χρησιμοποιούμε το QEMU-KVM ως τον hypervisor μαζί με το *virtio* ως το μηχανισμό επικοινωνίας μεταξύ του host και των εικονικών μηχανών. Το virtio [67] είναι μία τυποποιημένη διεπαφή που χρησιμοποιείται για την ανάπτυξη εικονικοποιημένων συσκευών ακολουθώντας την προσέγγιση

της παραεικονικοποίησης. Όπως αναφέραμε νωρίτερα, η παραεικονικοποίηση επιτρέπει χαμηλό κόστος εικονικοποίησης Ε/Ε εγκαθιστώντας ένα αποδοτικό κανάλι επικοινωνίας μεταξύ του host και της εικονικής μηχανής (guest). Κάνοντας χρήση αυτής της μεθόδου, το εικονικό υλικό παρέχει μία διεπαφή λογισμικού στον αντίστοιχο οδηγό στον guest, ο οποίος έχει γνώση ότι υπόκειται εικονικοποίηση και έτσι μειώνει την όποια περιττή κίνηση Ε/Ε που ένας οδηγός χωρίς αντίστοιχη γνώση θα παρήγαγε.



Σχήμα 13: Μηχανισμός μεταφοράς virtio

Αντίστοιχα με τη μέθοδο Ε/Ε του Xen, το virtio ακολουθεί την προσέγγιση του μοντέλου split-driver, σύμφωνα με την οποία ένας παραεικονικοποιημένος frontend οδηγός εισάγεται στον guest, επικοινωνώντας με το αντίστοιχο backend στην πλευρά του host. Για να επιτευχθεί η επικοινωνία, μία δομή κοινού δακτυλίου εγκαθιδρύεται μεταξύ του guest και του host (Σχήμα 13). Ο οδηγός frontend υποβάλλει αιτήσεις Ε/Ε τοποθετώντας τους αντίστοιχους buffers στον κοινό δακτύλιο και ειδοποιεί το backend. Στη συνέχεια, το backend επεξεργάζεται το συμβάν, εξομοιώνει την Ε/Ε που του ζητήθηκε και παράγει ένα αντίστοιχο αποτέλεσμα. Τοποθετεί την απάντηση στο δακτύλιο και ειδοποιεί την πλευρά του guest μέσω μίας εικονικής διακοπής (virtual interrupt). Ο guest περιμένει είτε κάνοντας busy-wait στον κοινό δακτύλιο, καταναλώνοντας κύ-

κλους ΚΜΕ, είτε μπλοκάροντας έως ότου η αίτηση ολοκληρωθεί. Στην τελευταία περίπτωση, η διακοπή που παράγεται από τον host ξυπνάει τον guest, ο οποίος προωθεί προς τα πάνω την απάντηση στα ανώτερα στρώματα. Σε αυτό το σημείο επισημαίνουμε ότι κατά την επικοινωνία μεταξύ του guest και του host δεν προκαλείται καμία αντιγραφή, καθώς χρησιμοποιείται μία περιοχή κοινής μνήμης και ο host μπορεί να έχει πρόσβαση στο φυσικό χώρο διευθύνσεων του guest και να απεικονίσει (map) τους αντίστοιχους buffers στο δικό του χώρο διευθύνσεων.

Προκλήσεις κατά την εικονικοποίηση επιταχυντών

Οι επιταχυντές αποτελούν ένα εξειδικευμένο υποσύνολο συσκευών Ε/Ε, καθώς πέραν των τυπικών χαρακτηριστικών Ε/Ε προσφέρουν και υπολογιστική ισχύ. Επομένως, για να εικονικοποιήσει κάποιος έναν επιταχυντή, χρειάζεται να επιλύσει, πέραν των προαναφερθεισών δυσκολιών της εικονικοποίησης Ε/Ε, επιπλέον αρκετές περισσότερες προκλήσεις εξαιτίας της ειδικής φύσης των συσκευών αυτών. Σε αυτή την ενότητα, εντοπίζουμε τις κυριότερες προκλήσεις εικονικοποίησης των συσκευών επιτάχυνσης και προχωρούμε στην ακόλουθη κατηγοριοποίηση.

Πολυπλοκότητα συσκευής

Μία παραδοσιακή συσκευή Ε/Ε συνήθως περιλαμβάνει ένα σύνολο από καταχωρητές Ε/Ε και μνήμη. Κάποιες πιο πολύπλοκες συσκευές (όπως οι προσαρμογείς δικτύου 10Gbps) είναι εξοπλισμένες επίσης με μικροεπεξεργαστή, ώστε να πραγματοποιούν offload ειδικών λειτουργιών, όπως TCP offload για τις μηχανές TSO (TCP Segmentation Offload). Από την άλλη πλευρά, η αρχιτεκτονική των σύγχρονων επιταχυντών είναι άκρως πολύπλοκη σε σύγκριση ακόμα και με τις πιο σύγχρονες, χαμηλού latency και υψηλού bandwidth συσκευές δικτύου. Οι επιταχυντές είναι εξοπλισμένοι με πολλούς πυρήνες (cores), μπορούν να χειριστούν εκατοντάδες νήματα (threads) και διαθέτουν τη δική τους πολύπλοκη μονάδα διαχείρισης μνήμης. Σε κάποιες περιπτώσεις δύνανται ακόμα και να εκκινήσουν ένα ανεξάρτητο λειτουργικό σύστημα, όπως συμβαίνει με το Intel Xeon Phi, το οποίο μπορεί να εκληφθεί ως ένας κατανεμημένος κόμβος εντός του ίδιου συστήματος. Συνεπώς, ο διαχωρισμός της

λογικής συσκευής από τη φυσική της υλοποίηση σε αυτό το πλαίσιο παρουσιάζει μεγάλες προκλήσεις.

Σημασιολογία χρονοδρομολόγησης

Δεδομένων των ανωτέρω, οι συσκευές επιτάχυνσης επιπλέον υπακούν σε εντελώς διαφορετική σημασιολογία χρονοδρομολόγησης σε σχέση με την παραδοσιακή διαδικασία της χρονοδρομολόγησης ΚΜΕ. Για παράδειγμα, τουλάχιστον στις παραδοσιακές GPUs δεν υπάρχει το χαρακτηριστικό του διαμοιρασμού χρόνου και έτσι οι GPU εργασίες δεν μπορούν να διακοπούν σε επίπεδο εντολών υλικού. Επομένως, στην περίπτωση που υπάρχει συναγωνισμός μεταξύ μίας μικρής και μίας μεγάλη εργασίας και συμβεί η μεγάλη εργασία να προλάβει να δεσμεύσει τη GPU, τότε η μικρότερη εργασία θα χρειαστεί να περιμένει την ολοκλήρωση της μεγάλης, γεγονός που αυξάνει το συνολικό χρόνο αναμονής των εργασιών. Κάτι τέτοιο δυσκολεύει τη διαδικασία πολυπλεξίας των αιτημάτων Ε/Ε και κατά συνέπεια το διαμοιρασμό της αντίστοιχης συσκευής μεταξύ πολλαπλών πελατών. Πρόσφατες εξελίξεις στην αρχιτεκτονική των GPUs υποστηρίζουν τη διακοπή μίας εργασίας μέσω ενός τρόπου διαμοιρασμού χρόνου προσφέροντας έναν πιο λεπτομερή έλεγχο [57]. Παρά ταύτα, δεν υπάρχει, τουλάχιστον δημόσια, διαθέσιμη πληροφορία που να δείχνει την ύπαρξη μηχανισμού ελέγχου σε επίπεδο λογισμικού [84]. Επιπλέον, στις παραδοσιακές GPUs δεν υπάρχει υποστήριξη ούτε για διαμοιρασμό χώρου, γεγονός που συνεπάγεται τη σειριακή εκτέλεση στη συσκευή, ακόμα και σε περίπτωση που οι αντίστοιχες εργασίες θα μπορούσαν να εκτελεστούν ταυτόχρονα στη GPU. Προκειμένου να μετριαστούν οι επιπτώσεις αυτής της αδυναμίας, η NVIDIA έχει κυκλοφορήσει το Multi-Process Service (MPS) [56] για τις νεότερες GPUs της, μέσω του οποίου παρέχει το χαρακτηριστικό του διαμοιρασμού χώρου σε πολλαπλές εργασίες. Το MPS ουσιαστικά αποτελείται από έναν proxy/δαίμονα ο οποίος υποβάλλει αιτήσεις στη συσκευή εκ μέρους των αρχικών GPU εργασιών. Με αυτό τον τρόπο, οι αιτήσεις υποβάλλονται εντός του ίδιου GPU πλαισίου (GPU context), το οποίο εγκυμονεί επιπτώσεις ως προς την ασφάλεια σε ένα εικονικοποιημένο περιβάλλον διαμοιρασμού GPU. Παρ' όλα αυτά, με την κυκλοφορία της αρχιτεκτονικής Volta [58], η NVIDIA ανανέωσε το υποσύστημα MPS επιτρέποντας στις GPU εργασίες να εκτελούνται απευθείας στον επιταχυντή και δημιουργώντας απομόνωση μεταξύ αυτών μέ-

σω ανάθεσης ξεχωριστών χώρων διευθύνσεων. Η μελέτη των πολιτικών χρονοδρομολόγησης εικονικών μηχανών και η αποτίμηση των αντίστοιχων αλγορίθμων είναι εκτός του πλαισίου αυτής της διατριβής. Παρ' όλα αυτά, καθώς με τη συγκεκριμένη διατριβή παρέχουμε συστήματα για διαμοιρασμό επιταχυντών χρησιμοποιώντας αποδοτικές μεθόδους σε τέτοια περιβάλλοντα, καθιστούμε δυνατή την αντίστοιχη μελλοντική έρευνα που θα λαμβάνει υπόψη τη χρονοδρομολόγηση μεταξύ των εικονικών μηχανών. Αρκετές από τις σχετικές εργασίες της ερευνητικής κοινότητας που αναφέρουμε στην αντίστοιχη ενότητα σχετικά με την εικονικοποίηση GPU συσκευών εξετάζουν ζητήματα χρονοδρομολόγησης στα συστήματά τους. Σχετικά με τα χαρακτηριστικά του Xeon Phi στο πλαίσιο της χρονοδρομολόγησης επιταχυντών, αυτό υποστηρίζει το διαμοιρασμό τόσο του χρόνου όσο και του χώρου και διαθέτει αυτά τα χαρακτηριστικά μέσω του μικρολειτουργικού συστήματός του, το οποίο εκτελείται σε έναν αποκλειστικό πυρήνα (core).

Επιπρόσθετα, όπως αρκετές από τις εργασίες στη GPU ερευνητική κοινότητα αναφέρουν [72, 60, 44, 10], το context switch στις αρχιτεκτονικές SIMT είναι τάξης-μεγέθους πιο αργό. Σε αυτό το κόστος χρονοδρομολόγησης, πρέπει επίσης να λάβουμε υπόψη το κόστος της επαναφόρτωσης στη μνήμη του επιταχυντή μέσω του διαύλου PCIe των δεδομένων μίας προηγουμένως διακεκομμένης εργασίας. Καθώς είναι κοινά αποδεκτό ότι η αντιγραφή δεδομένων από και προς τη συσκευή μέσω του PCIe αποτελεί την κυριότερη στενωπό (bottlekneck) κατά τις λειτουργίες offload προς τη συσκευή, αυτό το κόστος είναι σχετικά υψηλό. Τέλος, θεωρώντας το γεγονός ότι, όπως σε κάθε μέθοδο εικονικοποίησης Ε/Ε, ένα επιπλέον επίπεδο χρονοδρομολόγησης προστίθεται στην όλη διαδικασία, καθίσταται σαφές ότι η προσφορά υψηλής ποιότητας υπηρεσίας είναι εξαιρετικά δύσκολη.

Λογισμικό συστήματος κλειστού κώδικα

Ένα επιπλέον εμπόδιο ειδικά για την περίπτωση της εικονικοποίησης GPU προκύπτει από το γεγονός ότι οι βασικές εταιρείες κατασκευής GPU παρέχουν τους αντίστοιχους οδηγούς με έναν κλειστό τρόπο. Αυτό καθιστά αρκετά δύσκολο για τους μηχανικούς και τους ερευνητές να κατανοήσουν πλήρως τη βαθύτερη αρχιτεκτονική του συστήματος προκειμένου να παρέχουν μία διαχωρισμένη εικονική συσκευή. Προς αυτό το στόχο, έχουν πραγματοποιηθεί

αξιοσημείωτες προσπάθειες για την καλύτερη κατανόηση και διερεύνηση της εσωτερικής λειτουργίας της κλειστής GPU αρχιτεκτονικής και των αντίστοιχων οδηγών συσκευής και για την αντίστοιχη παροχή προς την ερευνητική κοινότητα εναλλακτικών ανοιχτού κώδικα ακολουθώντας το δύσκολο δρόμο του reverse engineering [86, 61, 49]. Αυτές οι δουλειές έχουν θέσει τα θεμέλια για τη μετέπειτα έρευνα στον τομέα της εικονικοποίησης GPU. Παρ' όλα αυτά, οι τεχνικές reverse engineering θυσιάζουν σε κάποιο βαθμό την αξιοπιστία, την επίδοση ακόμα και ένα υποσύνολο χαρακτηριστικών με σκοπό το άνοιγμα της στοίβας λογισμικού της εκάστοτε συσκευής. Από την άλλη πλευρά, μία διαφορετική προσέγγιση εικονικοποίησης πλατφορμών επιτάχυνσης είναι να παρέχει κάποιος συμβατότητα με υψηλότερου επιπέδου προγραμματιστικές διεπαφές ανακατευθύνοντάς τις σε ένα άλλο backend με τη χρήση διαφόρων τεχνικών. Το βασικό μειονέκτημα αυτής της προσέγγισης είναι ότι εξαρτάται από συγκεκριμένες εκδόσεις προγραμματιστικών διεπαφών και έτσι, τα προκύπτοντα συστήματα δεν παραμένουν συμβατά με μελλοντικές εκδόσεις [31].

Στην προσέγγισή μας, προτείνουμε μία έμμεση υψηλότερου επιπέδου μέθοδο, σύμφωνα με την οποία η χρήση ενός απομακρυσμένου GPU συστήματος βελτιστοποιείται μέσω ενός αποδοτικού μηχανισμού ενδοκομβικής επικοινωνίας. Με αυτό τον τρόπο, η προσέγγισή μας μπορεί να παραμένει συμβατή με μελλοντικές εκδόσεις της αντίστοιχης βιβλιοθήκης, καθώς αυτή η μέθοδος βασίζεται σε ένα απομακρυσμένο σύστημα που παραμένει ενήμερο με τις host βιβλιοθήκες παρέχοντας συχνά τις αντίστοιχες νέες εκδόσεις. Επιπλέον, όπως δείχνουμε στην επόμενη ενότητα, η μέθοδός μας συμβάλλει σε σημαντική αύξηση της επίδοσης σε σύγκριση με το προκαθορισμένο δικτυακό μονοπάτι επικοινωνίας και συνολικά μπορεί να αποτελέσει μία βίωσιμη λύση προκειμένου πολλαπλές εικονικές μηχανές να μπορούν να έχουν αποδοτική πρόσβαση σε μία GPU.

Μη γνώση εικονικοποίησης

Από την άλλη πλευρά, ακόμα και στην περίπτωση της στοίβας επιταχυντών ανοιχτού λογισμικού, η μείωση του κόστους εικονικοποίησης παραμένει ένας απαιτητικός στόχος, ακόμα και για εικονικές μηχανές που δεν υποβάλλουν ταυτόχρονες αιτήσεις και δεν προκύπτει ανάγκη για χρονοδρομολόγηση στις συγκεκριμένες περιπτώσεις. Ένας από τους βασικούς λόγους για αυτό

είναι ότι το native λογισμικό συστήματος αγνοεί εγγενώς ότι οι πόροι που παρέχει υπόκεινται εικονικοποίηση. Για παράδειγμα, στην περίπτωση της εικονικοποίησης του Xeon Phi, το λογισμικό συστήματος που εκτελείται στον επιταχυντή και διαχειρίζεται τη μνήμη της συσκευής χρειάζεται να έχει γνώση σε κάποιο βαθμό ότι υπόκειται εικονικοποίηση ή τουλάχιστον να παρέχει κάποια διεπαφή για διαχείριση μνήμης σε επίπεδο σελίδας, προκειμένου να μειωθεί το κόστος εικονικοποίησης, καθώς κατά την εικονικοποίηση εμπλέκονται περισσότερα επίπεδα χώρων διευθύνσεων, τα οποία στη γενική περίπτωση βρίσκονται διάσπαρτα στη μνήμη.

Επίπεδο διαφάνειας

Ένα επιπλέον ζήτημα για την πολύπλοκη αρχιτεκτονική Ε/Ε των επιταχυντών είναι ο ορισμός κατάλληλης σημασιολογίας για τις εικονικές συσκευές και τις αντίστοιχες διεπαφές καθώς και το επίπεδο της διαφάνειας για τις εφαρμογές. Λαμβάνοντας υπόψη αυτό, η ερευνητική κοινότητα έχει προσφέρει προσεγγίσεις για εικονικοποίηση επιταχυντών ακολουθώντας κάποιους συμβιβασμούς. Μεταξύ αυτών, υπάρχουν παραλλαγές της πλήρους εικονικοποίησης (full virtualization) σε επίπεδο λογισμικού, κατά την οποία μία εικονική συσκευή με ακριβώς την ίδια διεπαφή με τη φυσική συσκευή εξάγεται στο χώρο χρήστη. Όπως αναφέραμε νωρίτερα, με αυτή την προσέγγιση η εικονική μηχανή αγνοεί ότι ο αντίστοιχος επιταχυντής είναι εικονικοποιημένος. Αυτό έχει το πλεονέκτημα της διαφάνειας, καθώς δεν απαιτούνται αλλαγές σε κανένα επίπεδο της στοίβας λογισμικού, αλλά επιφέρει αυξημένο κόστος επίδοσης, διότι περιλαμβάνει την εξομοίωση μίας αρκετά πολύπλοκης συσκευής. Με την προσέγγιση της πλήρους εικονικοποίησης συνήθως ο εικονικός οδηγός συσκευής παραμένει άθικτος και εξομοιώνονται οι αντίστοιχες λειτουργίες παγιδεύοντας (trapping) προς τον hypervisor κάθε φορά που ο guest προβαίνει σε μία αίτηση Ε/Ε ή σε μία αντίστοιχη πρόσβαση. Ο αυξημένος όγκος αυτών των traps συνεισφέρει στο συνολικό κόστος, καθώς το κάθε trap έχει αποδειχθεί κοστοβόρο. Στο άλλο άκρο των προτεινόμενων λύσεων, παρέχεται μία εικονική συσκευή με εξ ολοκλήρου νέα διεπαφή, σχεδιασμένη να είναι απλή και αποδοτική, θυσιάζοντας τη συμβατότητα. Κάπου μεταξύ αυτών των άκρων βρίσκονται οι προσεγγίσεις παραεικονικοποίησης, σύμφωνα με τις οποίες ένας οδηγός frontend εισάγεται στον guest με γνώση ότι ο επιταχυντής

υπόκειται εικονικοποίηση. Με αυτή τη γνώση, το frontend κομμάτι ανακατευθύνει τη ροή ελέγχου και δεδομένων και επικοινωνεί αποδοτικά με το αντίστοιχο backend μέρος, το οποίο βρίσκεται σε ένα προνομιούχο επίπεδο (εικονική μηχανή ή hypervisor). Όπως περιγράφουμε παρακάτω, το vPHI αποτελεί προσέγγιση παραεικονικοποίησης σε συνδυασμό με τη στόχευση εικονικοποίησης του χαμηλού επιπέδου στρώματος μεταφοράς, με στόχο τη συμβατότητα με μελλοντικά υψηλού επιπέδου runtimes και βιβλιοθήκες καθώς και με διαφορετικές εκδόσεις. Τέλος, προκειμένου να ξεπεραστούν τα ζητήματα τόσο της διαφάνειας όσο και της επίδοσης, οι εταιρείες προσθέτουν υποστήριξη υλικού στις φυσικές συσκευές ακολουθώντας το μοντέλο της απευθείας ανάθεσης συσκευής. Με αυτό τον τρόπο, επιτρέπουν στους επιταχυντές να προσαρτηθούν απευθείας στην εικονική μηχανή προσφέροντας επίδοση πολύ κοντά στη native, αλλά αποκλείοντας το διαμοιρασμό της συσκευής και την κλιμακωσιμότητα. Σε αυτό το πλαίσιο, η NVIDIA έχει προσπαθήσει να αμβλύνει τα ζητήματα κλιμακωσιμότητας και να προσαρμοστεί προς την IOV προσέγγιση. Έτσι, σε κάποιες από τις κάρτες της παρέχει στο χρήστη τη δυνατότητα να ορίσει στατικά ένα μέγιστο αριθμό από εικονικές GPUs στο επίπεδο του υλικού και να αναθέσει απευθείας κάθε μία από αυτές σε μία πιθανά διαφορετική εικονική μηχανή [78]. Όπως περιγράψαμε προηγουμένως, τα κύρια μειονεκτήματα αυτής της προσέγγισης είναι η ανελαστικότητα και η δυσκολία κλιμακωσιμότητας σε δυναμικά νεφοϋπολογιστικά περιβάλλοντα, όπου το migration και η δυναμική πρόσθεση/αφαίρεση (hot plugging/unplugging) συσκευών συμβαίνει κατά τακτά χρονικά διαστήματα.

Βελτιστοποίηση της ενδοκομβικής επικοινωνίας για απομακρυσμένο offloading GPU εργασιών

Σε αυτή την ενότητα παρουσιάζουμε την προσέγγισή μας για την ενεργοποίηση επιτάχυνσης GPU για εικονικές μηχανές που εκτελούνται στον ίδιο φυσικό κόμβο. Εστιάζουμε σε NVIDIA GPUs και προτείνουμε μία φορητή μέθοδο με διαφορετικές εκδόσεις βιβλιοθηκών της NVIDIA CUDA. Επιτυγχάνουμε αυτή την ιδιότητα κάνοντας χρήση του rCUDA, ενός απομακρυσμένου συστήματος πρόσβασης σε GPU. Όπως αναφέραμε σε προηγούμενη ενότητα, το σημαντικότερο μειονέκτημα του rCUDA ως προς το κόστος εκτέλεσης εί-

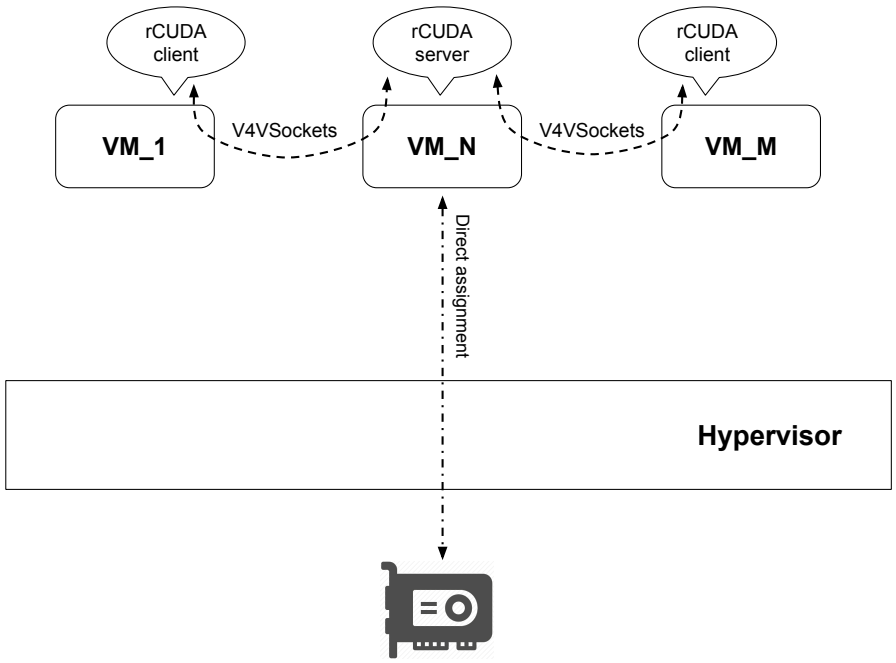
ναι ότι παρουσιάζει χαμηλή επίδοση όταν εφαρμόζεται σε διατάξεις με τυπικές στοίβες δικτύου. Αντ' αυτού, οι συγγραφείς του rCUDA προτείνουν τη χρήση δικτύων διασύνδεσης χαμηλού latency και υψηλού throughput ώστε να καταστεί εφικτή με βιώσιμο τρόπο η επιτάχυνση εφαρμογών για κόμβους που δεν είναι εξοπλισμένοι με GPUs.

Σε αυτό το πλαίσιο, χρησιμοποιούμε το rCUDA σε έναν φυσικό κόμβο ο οποίος φιλοξενεί πολλαπλές εικονικές μηχανές και βελτιστοποιούμε την ενδοκομβική επικοινωνία μεταξύ αυτών των εικονικών μηχανών. Πιο συγκεκριμένα, ακολουθώντας το σχήμα του rCUDA, υπάρχουν δύο ειδών κόμβων σε ένα HPC cluster: οι GPU κόμβοι, οι οποίοι λειτουργούν ως rCUDA servers, και οι μη-GPU κόμβοι, οι οποίοι λειτουργούν ως rCUDA clients και προωθούν τις CUDA αιτήσεις στους αντίστοιχους servers μέσω δικτύου. Στο Σχήμα 14 απεικονίζεται η τη διάταξη που χρησιμοποιούμε. Εγκαθιστούμε ένα φυσικό κόμβο εξοπλισμένο με μία GPU συσκευή και την αναθέτουμε απευθείας σε μία εικονική μηχανή, η οποία λειτουργεί ως rCUDA server. Έπειτα, εκκινούμε μία εικονική μηχανή που διαδραματίζει το ρόλο του rCUDA client και έτσι έχει πρόσβαση στη GPU μέσω του rCUDA συστήματος. Παρ' όλα αυτά, μειώνουμε αισθητά το δικτυακό κόστος υλοποιώντας και χρησιμοποιώντας το V4VSockets, ένα εξαιρετικά αποδοτικό και διαφανές σύστημα ενδοκομβικής επικοινωνίας μεταξύ εικονικών μηχανών που εκτελούνται στον ίδιο φυσικό κόμβο. Παρά το γεγονός ότι σε αυτή τη διατριβή επικεντρωνόμαστε σε ένα σενάριο χρήσης με GPU, το V4VSockets μπορεί να χρησιμοποιηθεί και σε άλλα σενάρια εφαρμογών sockets μεταξύ συνυπαρχουσών εικονικών μηχανών, προκειμένου να μειωθεί αισθητά το κόστος επικοινωνίας.

Στη συνέχεια περιγράφουμε συνοπτικά το σχεδιασμό του V4VSockets κάνοντας χρήση του Xen ως πλατφόρμα εικονικοποίησης και παρουσιάζουμε τα αποτελέσματα αξιολόγησης της προσέγγισής μας.

Σχεδιασμός του V4VSockets

Το V4VSockets είναι ένα εξαιρετικά αποδοτικό σύστημα ενδοκομβικής επικοινωνίας στη Xen πλατφόρμα. Η προσέγγισή μας είναι χτισμένη στο V4V, το οποίο αποτελεί μέρος του έργου Xenclient [80]. Το V4V αποτελεί έναν αφαιρετικό μηχανισμό που παρέχεται από τον hypervisor Xen υποστηρίζοντας

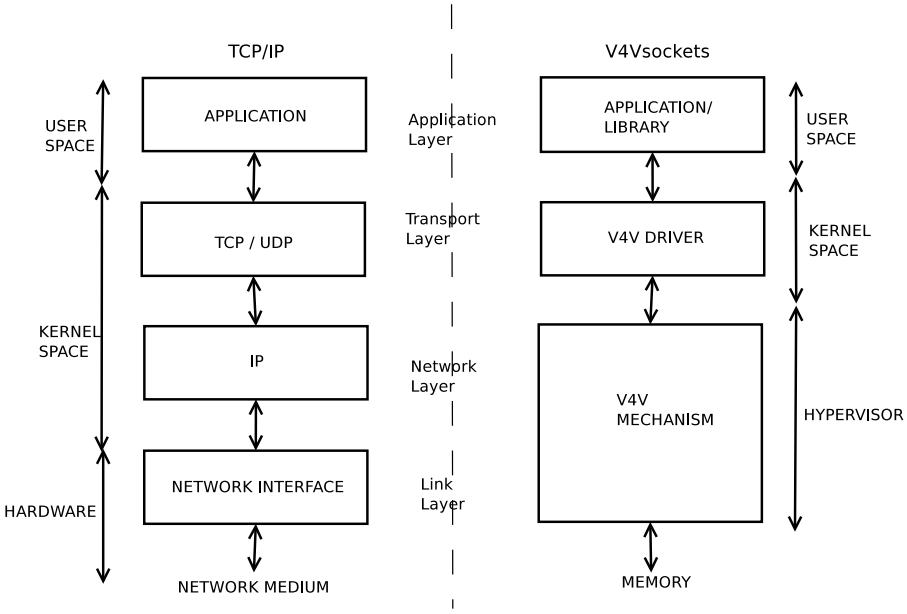


Σχήμα 14: rCUDA πάνω από V4VSockets

πρωτογενείς λειτουργίες επικοινωνίας μεταξύ συνυπαρχουσών εικονικών μηχανών.

Το V4VSockets αποτελεί ουσιαστικά ένα γενικό στρώμα socket για το V4V μηχανισμό μεταφοράς και επιτρέπει εφαρμογές που εκτελούνται στο χώρο χρήστη μίας εικονικής μηχανής να επικοινωνούν με άλλες συνυπάρχουσες εικονικές μηχανές που εκτελούνται στο ίδιο φυσικό μηχάνημα. Το V4VSockets αποτελείται από έναν οδηγό συσκευής, που παρέχει μία socket προγραμματιστική διεπαφή στο χώρο χρήστη, και το V4V μηχανισμό μεταφοράς, ο οποίος παρέχεται ως επέκταση του hypervisor Xen. Μία αναλογία με πρωτοκόλλου TCP/IP απεικονίζεται στο Σχήμα 15.

Το V4VSockets είναι υλοποιημένο ως ένα σύστημα με πρωτόκολλο πλήρους-στοίβας και υποστηρίζει επικοινωνία peer-to-peer μεταξύ συνυπαρχουσών εικονικών μηχανών. Σε αντίθεση με τη συνήθη πρακτική της αποδέσμευσης της επικοινωνίας σε μία προνομιούχο εικονική μηχανή αφήνοντας τον hypervisor να διαχειρίζεται μόνο τα ζητήματα ασφάλειας, επιλέγουμε να παρακάμψουμε το προαναφερθέν ενδιάμεσο στρώμα και να χρησιμοποιήσουμε



Σχήμα 15: TCP/IP και V4VSockets

τον hypervisor ως επίπεδο ελέγχου *και* δεδομένων. Έτσι, τα δεδομένα ρέουν μεταξύ δύο εικονικών μηχανών χωρίς την παρέμβαση τρίτης εικονικής μηχανής, παρέχοντας με αυτό τον τρόπο καλύτερη απομόνωση και κλιμακωσιμότητα. Επιπλέον, αποφεύγουμε την τεχνική διαμοιρασμού σελίδων ανάμεσα στην εικονική μηχανή πηγής και στην εικονική μηχανή προορισμού που μπορεί να οδηγήσει σε διάφορα ζητήματα ασφάλειας (π.χ. διαρροή δεδομένων μεταξύ των δύο εικονικών μηχανών) και αντί αυτού χρησιμοποιούμε αντιγραφές μνήμης στις αντίστοιχες φάσεις του μονοπατιού δεδομένων. Παρέχουμε μία επισκόπηση της αρχιτεκτονικής αυτής στις επόμενες παραγράφους, περιγράφοντας συνοπτικά τις λειτουργίες κάθε επιπέδου.

Επίπεδο Εφαρμογής: Μία από τις πιο σημαντικές πτυχές του σχεδιασμού είναι η συμβατότητα με κάποια γενική προγραμματιστική διεπαφή και συγκεκριμένα τη διεπαφή socket. Ειδικότερα, στοχεύουμε στην παροχή ενός χαμηλού-κόστους συστήματος socket επικοινωνίας προς τις εφαρμογές που εκτελούνται σε συνυπάρχουσες εικονικές μηχανές χωρίς την ανάγκη αναδόμησης, επαναϋλοποίησης ή επαναμεταγλώττισης αυτών. Έτσι, στο V4VSockets, το επίπεδο εφαρμογής αναφέρεται στις κοινές κλήσεις επιπέδου socket (`socket()`,

`bind()`, `connect()` κλπ.), που προωθούν τις αντίστοιχες λειτουργίες και τα αντίστοιχα ορίσματα στο επίπεδο μεταφοράς.

Στην προσέγγισή μας το *επίπεδο Μεταφοράς* βρίσκεται στον πυρήνα της εικονικής μηχανής. Ουσιαστικά, υλοποιεί τις κλήσεις socket και τις αρχές επικοινωνίας του πρωτοκόλλου επικοινωνίας πάνω από το δικτυακό μέσο (στην περίπτωσή μας τον hypervisor Xen). Πιο συγκεκριμένα, το επίπεδο μεταφοράς χειρίζεται τη σημασιολογία των εικονικών συνδέσεων μεταξύ των εικονικών μηχανών που χρειάζεται να επικοινωνήσουν, είναι υπεύθυνο για να κατατμήσει και να στείλει τα πακέτα στα ανώτερα επίπεδα καλώντας τις αντίστοιχες υπερκλήσεις (hypercalls) στον hypervisor (επίπεδο δικτύου), και παρέχει ένα μηχανισμό ειδοποίησης του χώρου χρήστη της εικονικής μηχανής για τη λήψη των πακέτων, όπως επίσης και για έλεγχο λαθών.

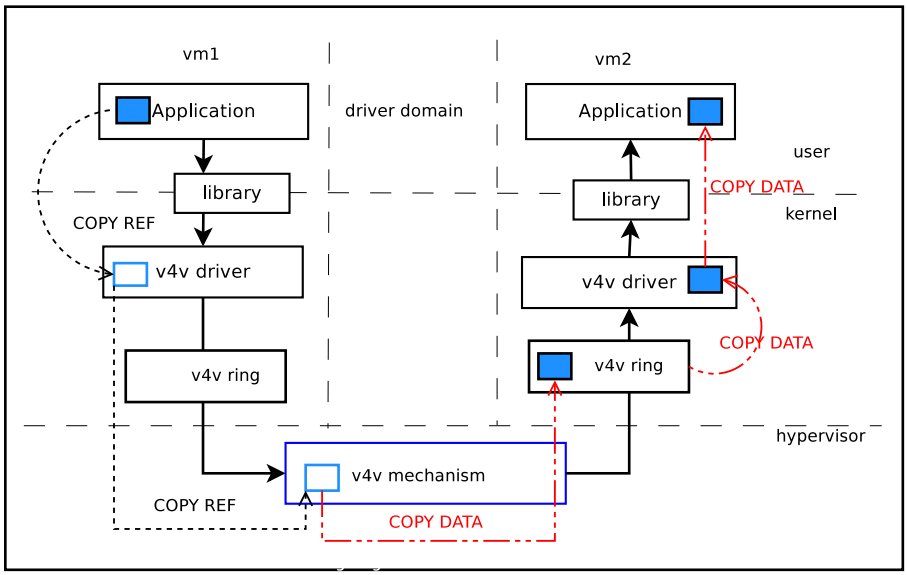
Το *επίπεδο Δικτύου/Συνδέσμου* βρίσκεται στον hypervisor, παρέχει ενθυλάκωση των μηνυμάτων των ανώτερων επιπέδων σε πακέτα που θα αποσταλούν στον προορισμό τους, σύμφωνα με τη σημασιολογία του V4V, καθώς και παράδοση των πακέτων αυτών. Το επίπεδο αυτό είναι υπεύθυνο για την αποστολή του εκάστοτε πακέτου στον προορισμό του, που στην περίπτωσή μας συνιστά μία αντιγραφή μνήμης. Επομένως, στην περίπτωση αποστολής ενός πακέτου, ο hypervisor τοποθετεί τα δεδομένα στον αντίστοιχο χώρο μνήμης του παραλήπτη και ειδοποιεί το επίπεδο μεταφοράς σχετικά με το εισελθέν πακέτο.

Ένα παράδειγμα ανταλλαγής δεδομένων με χρήση του V4VSockets παρουσιάζεται στο Σχήμα 16. Σε αυτό το σημείο, ο αναγνώστης μπορεί να ανατρέξει στο Σχήμα 12 σε προηγούμενη ενότητα, προκειμένου να συγκρίνει οπτικά τα δύο αντίστοιχα μονοπάτια δεδομένων.

Αξιολόγηση της επίδοσης του V4VSockets

Σε αυτή την ενότητα αρχικά περιγράφουμε τα πειράματα που εκτελέσαμε προκειμένου να αναλύσουμε τη συμπεριφορά του V4VSockets κάνοντας χρήση δικτυακών μετροπρογραμμάτων και στη συνέχεια παρουσιάζουμε την αξιολόγηση του συστήματός μας σε σενάρια που περιλαμβάνουν εικονικές μηχανές με πρόσβαση σε GPU.

Εγκαθιστούμε ένα φυσικό μηχάνημα με 2x Intel X5650 (Chipset 5520)



Σχήμα 16: Συνοπτική εικόνα του V4VSockets

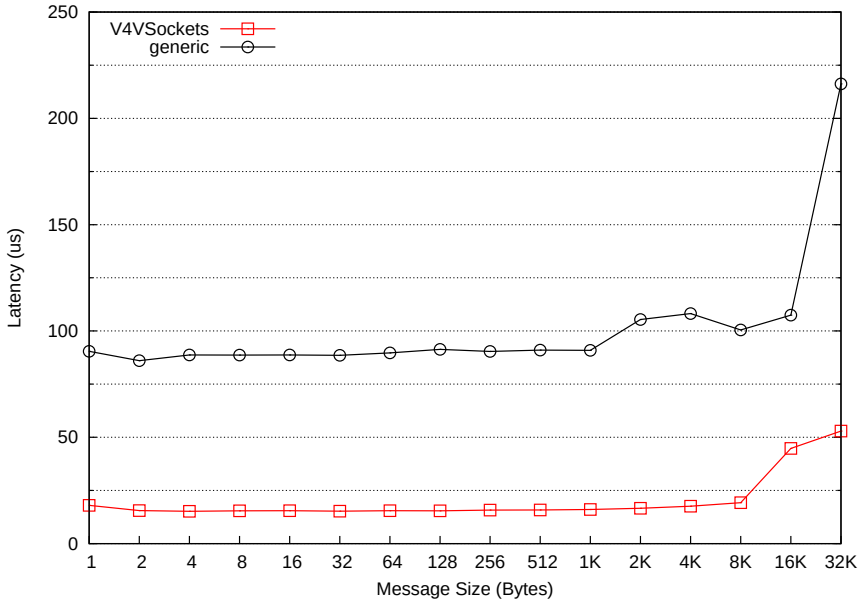
και 48GB RAM (@1333MHz) και εκτελούμε δύο βασικά πειράματα με χρήση μικρο-μετροπρογραμμάτων, προκειμένου να παρουσιάσουμε τα προτερήματα και τις αδυναμίες της προσέγγισής μας χωρίς το θόρυβο που θα προσέθεταν μοτίβα επικοινωνίας μεγαλύτερων εφαρμογών. Τέλος, παρουσιάζουμε και ένα τρίτο πιο ρεαλιστικό πείραμα χρησιμοποιώντας μία CUDA εφαρμογή από το πεδίο του GPGPU.

Αξιολόγηση του V4VSockets μέσω μικρο-μετροπρογραμμάτων

Εγκαθιστούμε ένα φυσικό κόμβο ως host εικονικών μηχανών και εκκινούμε έως 16 μονοπύρηνες εικονικές μηχανές (VM_1,VM_2, ...,VM_{16}). Χρησιμοποιούμε το NetPIPE [52] ως ένα μικρο-μετροπρόγραμμα, προκειμένου να συγκρίνουμε το V4VSockets με το προκαθορισμένο μονοπάτι TCP/IP πάνω από netfront/netback. Εφαρμόζουμε το μικρο-μετροπρόγραμμα μεταξύ των εικονικών μηχανών (16 διαφορετικά στιγμιότυπα, το VM_1 με το VM_2, το VM_3 με το VM_4 και ούτω κάθε εξής).

Στο Σχήμα 17 και στο Σχήμα 18 απεικονίζονται οι αντίστοιχες μετρήσεις όταν 2 εικονικές μηχανές (το VM_1 με το VM_2) ανταλλάσσουν μηνύματα. Το Σχήμα 17 δείχνει ότι το latency που επιτυγχάνει το V4VSockets για ένα μήνυμα

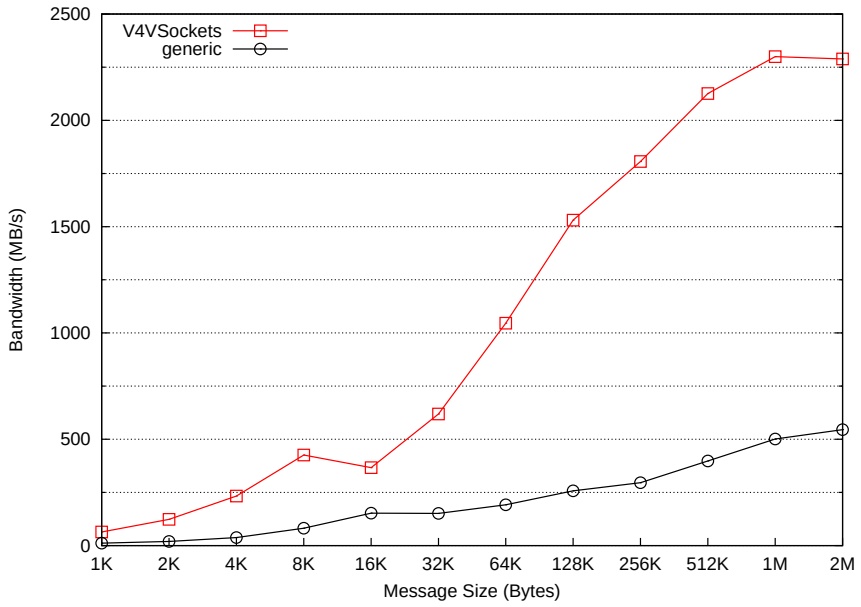
2 Bytes βελτιώνεται κάτα 81% σε σχέση με τη γενική περίπτωση. Πιο συγκεκριμένα, το latency στο μοντέλο split-driver είναι 86 us, ενώ το V4VSockets ολοκληρώνει το ίδιο έργο σε 16 us. Αυτό οφείλεται κυρίως στο κόστος επεξεργασίας της στοίβας TCP/IP, καθώς επίσης και στο μη αποδοτικό μονοπάτι δεδομένων μέσω του driver domain, το οποίο παρακάμπτεται στο δικό μας βελτιστοποιημένο μηχανισμό μεταφοράς.



Σχήμα 17: Latency του V4VSockets

Εξετάζοντας στη συνέχεια το throughput (Σχήμα 18), το V4VSockets υπερισχύει της προκαθορισμένης περίπτωσης και με βάση αυτή τη μετρική. Το μέγιστο throughput του V4VSockets είναι 2299 MB/s, 4.59x καλύτερο από το μοντέλο split-driver, το οποίο αποδίδει χαμηλότερα στα 501 MB/s για μηνύματα 1 MB.

Προκειμένου να διαπιστώσουμε τον τρόπο με τον οποίο το V4VSockets κλιμακώνει με βάση κυμαινόμενο αριθμό από εικονικές μηχανές που ανταλλάσσουν μηνύματα, μετράμε το throughput του συστήματος για 2, 4, 8 και 16 εικονικές μηχανές που επικοινωνούν σε ζεύγη (Σχήμα 19). Το αθροιστικό throughput αυξάνεται ανάλογα με τον αριθμό των εικονικών μηχανών που επικοινωνούν. Για παράδειγμα, δύο εικονικές μηχανές είναι ικανές να ανταλλάσ-

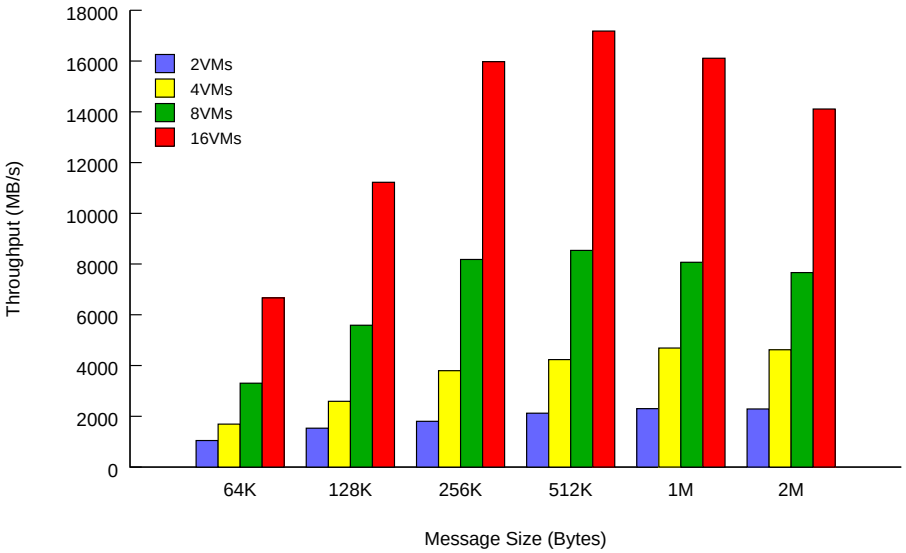


Σχήμα 18: Throughput του V4VSockets

σουν μηνύματα 512 KB σε ≈ 2 GB/s, ενώ 16 εικονικές μηχανές επιτυγχάνουν αθροιστικό throughput 8x μεγαλύτερο (≈ 16 GB/s) για το ίδιο μέγεθος μηνύματος.

Με βάση την υλοποίησή μας, το V4VSockets πραγματοποιεί τρεις αντιγραφές δεδομένων κατά τη μεταφορά μηνυμάτων: από το VM_1στον Xen, από τον Xen στον πυρήνα του VM_2 και από τον πυρήνα του VM_2 στο χώρο χρήστη του VM_2. Αυτό αποτελεί μία σχεδιαστική επιλογή, προκειμένου να αποφευχθεί η εναλλακτική επιλογή της κοινής μνήμης, η οποία μπορεί δυνητικά να παρουσιάσει επιπτώσεις ασφάλειας σε διάφορα σενάρια χρήσης. Σύμφωνα με την εν λόγω σχεδιαστική επιλογή: το VM_1 ειδοποιεί μέσω μίας κλήσης συστήματος και μίας υπερκλήσης ότι υπάρχει ένα μήνυμα για το VM_2. Ο Xen αντιγράφει τα δεδομένα από το χώρο χρήστη του VM_1 στο VM_2 και ειδοποιεί τον πυρήνα. Όταν ξυπνήσει ο πυρήνας, τα δεδομένα βρίσκονται ήδη στην cache του επεξεργαστή και έτσι τα δεδομένα μεταφέρονται απευθείας στο χώρο χρήστη του VM_2. Ως αποτέλεσμα, με χρήση του V4VSockets μπορούμε να φτάσουμε πε-

ρισσότερο από το μισό του bandwidth της μνήμης του συστήματος[1], επιτρέποντας ουσιαστικά bandwidth σαν αντιγραφή μνήμης σε μετρήσεις ανταλλαγής μηνυμάτων μεταξύ εικονικών μηχανών.



Σχήμα 19: Αθροιστικό throughput του V4VSockets

Προκειμένου να επιβεβαιώσουμε ότι το σύστημα συμπεριφέρεται με αποδεκτή επίδοση και με ένα βιώσιμο τρόπο όταν μεγάλος αριθμός από εικονικές μηχανές ασκούν πίεση στο δίαυλο μνήμης, εξετάζουμε τι αποτέλεσμα έχει στο latency ο μηχανισμός ανταλλαγής δεδομένων. Κατά την ανταλλαγή μικρών μηνυμάτων από 16 εικονικές μηχανές σε ζεύγη, το round trip latency παραμένει στα 16 us, επιβεβαιώνοντας την κλιμακωσιμότητα της προσέγγισής μας.

Εικονικές μηχανές με πρόσβαση σε GPU

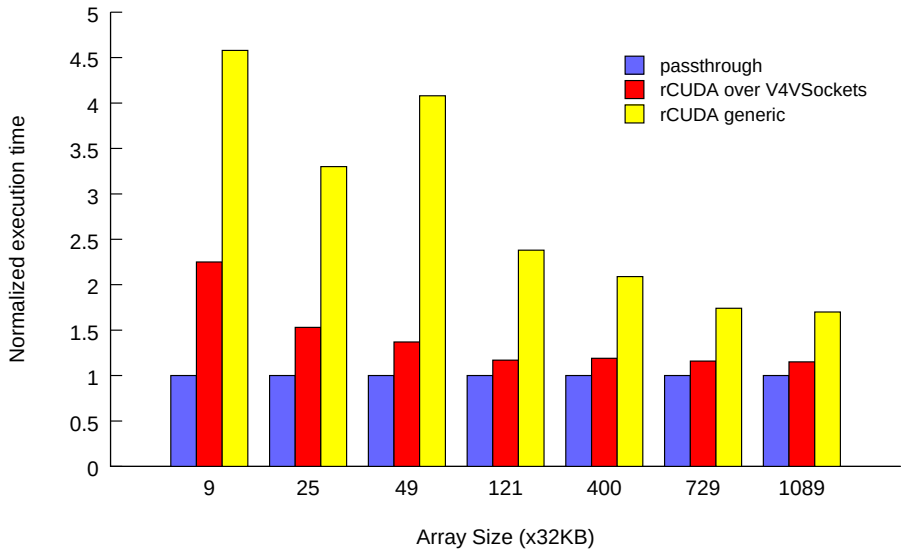
Σε αυτή την ενότητα, δείχνουμε τα προτερήματα του V4VSockets σε ένα πιο ρεαλιστικό μετροπρόγραμμα από το πεδίο των εφαρμογών GPU. Όπως αναφέραμε νωρίτερα, εφαρμόζουμε στο rCUDA τον αποδοτικό μας μηχανισμό μεταφοράς. Χτίζοντας σε ένα υπάρχον σύστημα απομακρυσμένης επιτάχυνσης σε συνδυασμό με το V4VSockets, επιτρέπουμε στις εικονικές μηχανές

[1] Εκτελέσαμε ένα stream μικρο-μετροπρόγραμμα και μετρήσαμε μέγιστο bandwidth μνήμης 27 GB/s

να ωφεληθούν από μία εικονική μηχανή εξοπλισμένη με GPU η οποία εκτελείται στον ίδιο φυσικό κόμβο, αποφεύγοντας πολύπλοκες διατάξεις ή ακριβές τεχνικές που διαταράσσουν τις κοινές υποδομές, όπως το IOV.

Κάνουμε χρήση δύο εικονικών μηχανών, του VM_1 που λειτουργεί ως rCUDA server, και του VM_2 που λειτουργεί ως rCUDA client. Προκειμένου να παρέχουμε GPU πρόσβαση στο VM_1, αναθέτουμε τη συσκευή GPU σε αυτή την εικονική μηχανή χρησιμοποιώντας την τεχνική του PCIe passthrough. Τελικά, θεωρούμε δύο περιπτώσεις: το γενικό μηχανισμό μεταφοράς με χρήση του TCP/IP πάνω από το μοντέλο split-driver (*rCUDA generic*) και το V4VSockets (*rCUDA over V4VSockets*). Ως βάση σύγκρισης, εκτελούμε ακριβώς το ίδιο πείραμα χωρίς την παρέμβαση του rCUDA, απευθείας στο VM_1 (*passthrough*)[2].

Χρησιμοποιούμε μία τυπική HPC εφαρμογή (stencil), τον απλής ακρίβειας πολλαπλασιασμό πίνακα-με-πίνακα, που παρέχει η CUDA στα samples [55] της NVIDIA.



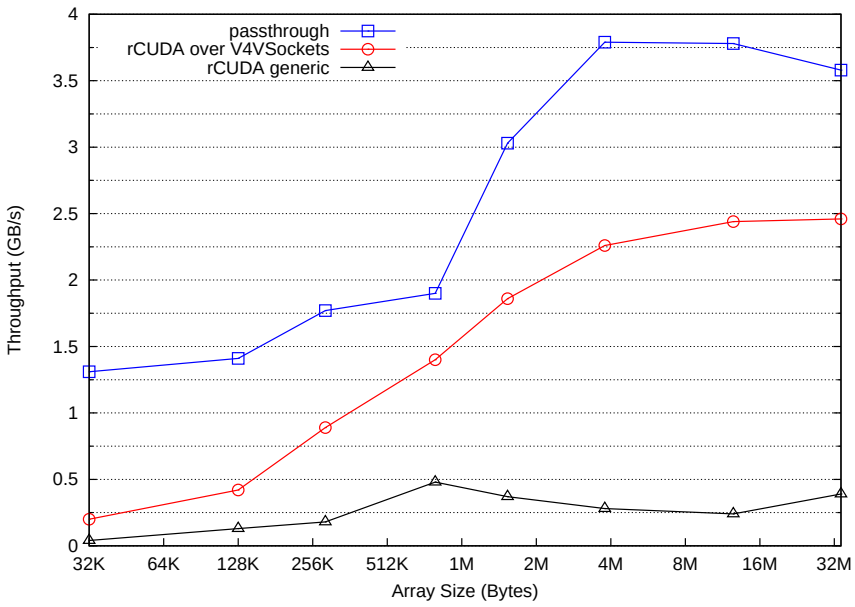
Σχήμα 20: Συνολικός χρόνος εκτέλεσης πολλαπλασιασμού πινάκων

Το εν λόγω πείραμα περιλαμβάνει την ακόλουθη διαδικασία: δύο αντι-

[2] Επιβεβαιώνουμε τις μετρήσεις και σε μη-εικονικοποιημένο περιβάλλον με ίδια συσκευή GPU. Δεν παρατηρήσαμε κάποια σημαντική διαφορά ως προς την επίδοση σε σχέση με την passthrough εκτέλεση σε εικονική μηχανή.

γραφές των πινάκων εισόδου από την κύρια μνήμη του κόμβου στη μνήμη της συσκευής GPU, την εκτέλεση του πολλαπλασιασμού στη GPU και τελικά μία αντιγραφή του παραχθέντος πίνακα πίσω στην κύρια μνήμη. Ο κανονικοποιημένος συνολικός χρόνος εκτέλεσης του μετροπρογράμματος πολλαπλασιασμού πίνακα-με-πίνακα απεικονίζεται στο Σχήμα 20. Ο άξονας των X αναπαριστά το μέγεθος του πίνακα πολλαπλασιασμένο επί 32 KB.

Παρατηρούμε ότι το V4VSockets αποδίδει πολύ κοντά στη βάση σύγκρισης. Για έναν πίνακα εισόδου των 1089 x 32 KB (2112 x 4224 στοιχεία τύπου `float`) το V4VSockets προσθέτει ένα 15% κόστος σε σύγκριση με την εκτέλεση τοπικά, ενώ η γενική περίπτωση προσθέτει ένα 71% κόστος. Ουσιαστικά αυτό αποτελεί το στόχο μας: μέσω του V4VSockets και του rCUDA, οι εικονικές μηχανές ενός φυσικού κόμβου να μπορούν να διαμοιράζονται απρόσκοπτα μία συσκευή GPU με ένα πολύ μικρό κόστος σε σχέση με τη γενική περίπτωση. Δεδομένου, επίσης, ότι οι πλήρεις HPC εφαρμογές χρησιμοποιούν πίνακες μεγάλου μεγέθους, το σύστημά μας μπορεί να παρέχει το αναγκαίο bandwidth προκειμένου να πραγματοποιείται offload εκτέλεση σε GPU με μικρό κόστος εξαιτίας της απομακρυσμένης εκτέλεσης.



Σχήμα 21: Throughput μεταφοράς πολλαπλασιασμού πινάκων

Για να εξετάσουμε διεξοδικότερα τον αντίκτυπο του V4VSockets στη βελτίωση του χρόνου εκτέλεσης, απεικονίζουμε το throughput που επιτυγχάνεται κατά την αντιγραφή ενός από τους πίνακες εισόδου από την κύρια μνήμη του κόμβου στη μνήμη της συσκευής GPU (ουσιαστικά πρόκειται για μία κλήση `cudamemcpy()`) στο Σχήμα 21. Από τα μεγέθη των πινάκων του Σχήματος, παρατηρούμε ότι το πείραμα βάσης σύγκρισης επιτυγχάνει 3.79 GB/s σε throughput και αυτό μπορεί να αυξηθεί μέχρι 4.42 GB/s για ολόκληρο το πείραμα (για μεγαλύτερα μεγέθη τα οποία δε φαίνονται στο Σχήμα), ενώ το μέγιστο throughput στην απομακρυσμένη περίπτωση V4VSockets είναι 2.46 GB/s. Παρ' όλα αυτά, για έναν πίνακα μεγέθους 34 MB (2112 x 4224 στοιχεία τύπου `float`) το V4VSockets υπερισχύει της γενικής περίπτωσης κατά συντελεστή 6.3 (0.39 GB/s).

Σύνοψη

Συνοψίζοντας, παρουσιάσαμε το V4VSockets, ένα εξαιρετικά αποδοτικό σύστημα επικοινωνίας για εικονικές μηχανές που εκτελούνται στον ίδιο φυσικό κόμβο. Το V4VSockets είναι συμβατό με socket API και μπορεί να χρησιμοποιηθεί με διάφορες socket εφαρμογές που εκτελούνται σε συνυπάρχουσες εικονικές μηχανές, όπως web servers, servers βάσεων δεδομένων ή δικτυακές λειτουργίες (firewall, εξισορροπητές φόρτου κλπ.). Σε αυτή τη διατριβή, επικετρωθήκαμε στο σενάριο χρήσης GPU και συνδυάσαμε το V4VSockets με το rCUDA, ένα σύστημα απομακρυσμένης εκτέλεσης GPU εφαρμογών. Με αυτό τον τρόπο, καταλήξαμε σε μία υψηλού επιπέδου προσέγγιση αποφεύγοντας την *πολυπλοκότητα* μίας πλήρως εικονικοποιημένης GPU και ξεπερνώντας τόσο το *κλειστού κώδικα* λογισμικό συστήματος της GPU όσο και τη *μη γνώση εικονικοποίησης* του επιταχυντή. Επιπλέον, αυτή η προσέγγιση παρέχει *διαφάνεια* σε CUDA εφαρμογές αλλά όχι σε αντίστοιχες εφαρμογές γραμμένες σε διαφορετικές προγραμματιστικές διεπαφές. Σχετικά με τη διαδικασία *χρονοδρομολόγησης*, παρέχουμε το μηχανισμό για εφαρμογή γενικών πολιτικών, παρ' όλα αυτά η εξερεύνηση τεχνικών χρονοδρομολόγησης σε αυτό το πεδίο είναι εκτός του πλαισίου αυτής της διατριβής. Με βάση την εμπειρία μας, θεωρούμε ότι είναι αναγκαίες αλλαγές σε επίπεδο υλικού για τις μελλοντικές τεχνολογίες επιταχυντών, προκειμένου να εφαρμοστούν αποδοτικές μέθοδοι χρονοδρομολόγησης.

Διαμοιρασμός επιταχυντών Xeon Phi σε εικονικοποιημένα περιβάλλοντα

Σε αυτή την ενότητα, περιγράφουμε το vPHI, ένα σύστημα εικονικοποίησης του Xeon Phi χαμηλού κόστους, που επιταχύνει τις εικονικές μηχανές επιτρέποντάς τις να κάνουν offload εργασίες σε μία κάρτα Xeon Phi. Σύμφωνα με όσα γνωρίζουμε, το vPHI αποτελεί την πρώτη προσέγγιση που επιτρέπει το διαμοιρασμό του Xeon Phi της Intel μεταξύ εικονικών μηχανών που εκτελούνται στον ίδιο φυσικό κόμβο. Το σύστημά μας είναι συμβατό με εκτελέσιμα που έχουν προκύψει από προμεταγλωττισμένες εφαρμογές, εξαλείφοντας την ανάγκη για τροποποίηση, αλλά ακόμα και για επαναμεταγλώττιση του υπάρχοντα πηγαίου κώδικα. Επίσης, υποστηρίζει και τα τρία σχήματα που ορίζονται στο μοντέλο εκτέλεσης του Xeon Phi, δηλαδή του *γηγενούς* (*native*), του *offload* και του *συμμετρικού* (*symmetric*).

Κάνουμε χρήση της αρχιτεκτονικής του Xeon Phi υπό μορφή συσκευής επιτάχυνσης ως σενάριο χρήσης και επιτρέπουμε το διαμοιρασμό του συνεπεξεργαστή μεταξύ εικονικών μηχανών που εκτελούνται στον ίδιο host μέσω εικονικοποίησης του επιπέδου μεταφοράς της στοίβας λογισμικού του επιταχυντή. Συγκρίνοντας αυτή την προσέγγιση με την προαναφερθείσα τεχνική εικονικοποίησης GPU, σε αυτή την εργασία επιλέγουμε τον επιταχυντή Xeon Phi, καθώς η αντίστοιχη στοίβα λογισμικού παρέχεται ως ανοιχτός κώδικας και έτσι μπορούμε να εικονικοποιήσουμε τα χαμηλότερα επίπεδα της στοίβας και να εξερευνήσουμε τα οφέλη αλλά και τους περιορισμούς μίας τέτοιας προσέγγισης.

Προκειμένου να αξιοποιηθούν οι δυνατότητες του συνεπεξεργαστή Xeon Phi σε εικονικοποιημένα περιβάλλοντα, η Intel προσφέρει κάποιες λύσεις για τον hypervisor KVM [23] και για τον hypervisor Xen [24] κάνοντας χρήση της μεθόδου *PCIe passthrough*. Επιπλέον, η VMware υποστηρίζει το Xeon Phi με το ESXi χρησιμοποιώντας την ίδια τεχνική [79]. Αυτές οι προσεγγίσεις εικονικοποίησης προσφέρουν επίδοση πολύ κοντά στο native σε βάρος της δυνατότητας διαμοιρασμού ενός επιταχυντή μεταξύ πολλών εικονικών μηχανών, η οποία δεν υποστηρίζεται με αυτές τις μεθόδους, καθώς η κάρτα Xeon Phi ανατίθεται απευθείας σε μία μοναδική εικονική μηχανή. Με το vPHI καταφέρνουμε να υπερκεράσουμε αυτό το εμπόδιο επιτρέποντας το διαμοιρασμό μίας

συσκευής Xeon Phi μεταξύ συνυπαρχουσών εικονικών μηχανών.

Στις επόμενες παραγράφους, περιγράφουμε συνοπτικά το σχεδιασμό του vPHI και στη συνέχεια παρουσιάζουμε την αξιολόγηση των αποτελεσμάτων μέσω εκτέλεσης διαφόρων πειραμάτων.

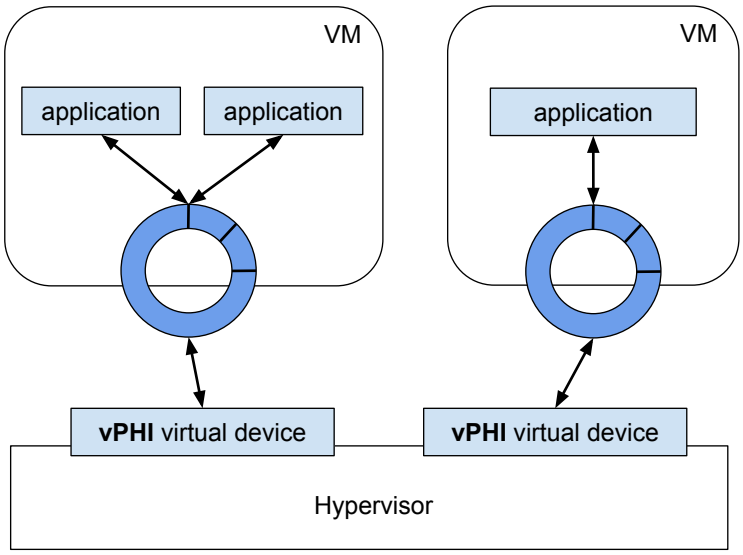
Σχεδιασμός του vPHI

Μία από τις βασικές προϋποθέσεις κατά το σχεδιασμό του vPHI είναι η ενεργοποίηση του διαμοιρασμού ενός συνεπεξεργαστή Xeon Phi μεταξύ εικονικών μηχανών που συνυπάρχουν στον ίδιο φυσικό κόμβο και της δυνατότητας ταυτόχρονου offload εργασιών από τις πολλαπλές εικονικές μηχανές προς τον επιταχυντή. Υλοποιούμε το vPHI κάνοντας χρήση του QEMU-KVM ως hypervisor και του virtio ως διεπαφή παραεικονικοποίησης. Όπως φαίνεται στο Σχήμα 23, το vPHI αποτελείται από έναν οδηγό guest πυρήνα και μία backend συσκευή QEMU υλοποιημένη στο χώρο χρήστη του host. Κατά τη φάση της υλοποίησης, προχωρήσαμε σταδιακά σε δύο εκδόσεις του πρωτοτύπου μας για λόγους επίδοσης. Στην πρώτη έκδοση του vPHI, χρειάστηκε να κάνουμε μία μικρή τροποποίηση στο KVM module του πυρήνα του host, προκειμένου να ανακατευθύνουμε καταλλήλως τα σφάλματα σελίδας στον guest, ενώ στη δεύτερη έκδοση, τροποποιήσαμε τον οδηγό Xeon Phi στον host. Παρά το ότι ο στόχος μας είναι να είμαστε όσο το δυνατόν λιγότερο παρεμβατικοί, η τροποποίηση της πρώτης έκδοσης είναι απαραίτητη προκειμένου να υποστηρίξουμε την απεικόνιση των περιοχών μνήμης του χώρου χρήστη του guest στη μνήμη της συσκευής Xeon Phi μέσω του `scif_mmap()`, ενώ οι τροποποιήσεις της δεύτερης έκδοσης στον οδηγό του host βοηθούν στη βελτίωση της επίδοσης του vPHI, καθώς μέσω αυτής καθιστούμε γνωστό στον οδηγό του host την ύπαρξη διεσπαρμένης μνήμης του guest και έτσι μειώνουμε το αντίστοιχο κόστος.

Ουσιαστικά, το vPHI πιάνει τα αρχικά αιτήματα μεταφοράς SCIF και τα ανακατευθύνει μέσα από τη backend συσκευή QEMU. Με την λήψη αυτών των αιτημάτων, ο οδηγός backend τα προωθεί στον host οδηγό SCIF, ο οποίος ελέγχει τη φυσική συσκευή. Μετά την ολοκλήρωση ενός αιτήματος Ε/Ε, τα αποτελέσματα προωθούνται πίσω στη στοίβα ακολουθώντας την αντίθετη κατεύθυνση, καταλήγοντας τελικά στον αρχικό αιτούντα, ο οποίος συνήθως

είναι ένα επίπεδο μεταφοράς κάποιου runtime. Ταυτόχρονα πολυ-νηματικά αιτήματα εκτέλεσης από διαφορετικές εικονικές μηχανές μπορούν να καταλήξουν να εκτελούνται παράλληλα στη συσκευή Xeon Phi εξαπλωμένα στους διαθέσιμους πυρήνες (cores) της κάρτας. Στην περίπτωση που υπάρχει υπερκάλυψη (oversubscription) ως προς το λόγο των αιτηθέντων νημάτων προς τους φυσικούς πυρήνες, τότε η πολυπλεξία των πόρων πραγματοποιείται από το χρονοδρομολογητή του uOS που εκτελείται σε ένα απομονωμένο πυρήνα του Xeon Phi. Στο Σχήμα 22 απεικονίζεται σε μία υψηλού-επιπέδου επισκόπηση ο τρόπος με τον οποίο το vPHI κάνει χρήση του virtio προκειμένου να επιτρέψει το διαμοιρασμό της συσκευής επιτάχυνσης τόσο μεταξύ εφαρμογών εντός της ίδιας εικονικής μηχανής όσο και μεταξύ διαφορετικών εικονικών μηχανών που εκτελούνται στον ίδιο φυσικό κόμβο. Σε πιο λεπτομερές επίπεδο, στο Σχήμα 23 θεωρούμε ένα αίτημα SCIF που προκαλείται από μία εφαρμογή μέσα σε μία εικονική μηχανή και δείχνουμε το αντίστοιχο μονοπάτι Ε/Ε. Πρόκειται για ένα αντιπροσωπευτικό σενάριο που συντελείται σε οποιοδήποτε από τα τρία προαναφερθέντα σχήματα εκτέλεσης του Xeon Phi. Οι συνεχείς γραμμές αναπαριστούν το μονοπάτι ελέγχου, ενώ οι διακεκομμένες γραμμές αναπαριστούν το μονοπάτι δεδομένων. Στις επόμενες παραγράφους,αναφερόμαστε σε κάθε φάση του Σχήματος 23 καθώς περγράφουμε κάθε υποσύστημα και τις λειτουργίες που επιτελεί.

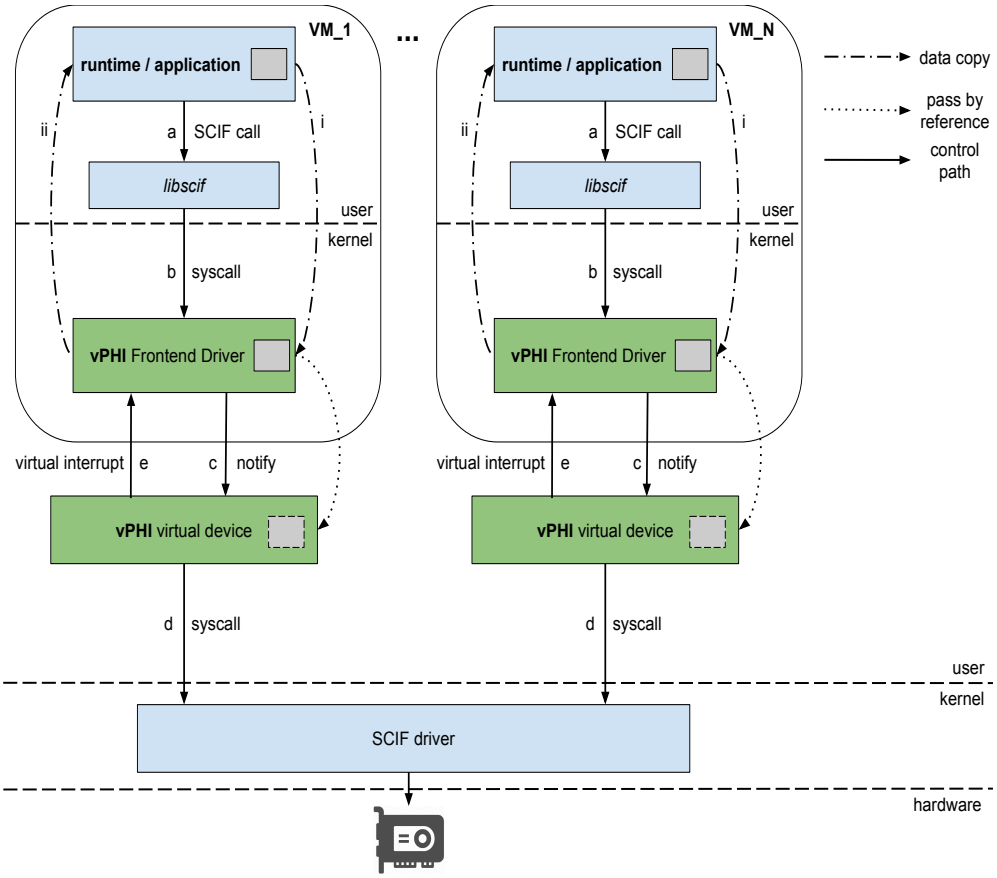
Με βάση το Σχήμα 23, περιγράφουμε ένα παράδειγμα μίας εφαρμογής που εκτελείται σε μία εικονική μηχανή και πραγματοποιεί offload υπολογιστικών πυρήνων στην κάρτα Xeon Phi. Όπως αναφέραμε προηγουμένως, αυτό το σενάριο μπορεί να συμβεί παράλληλα με διαφορετικές εφαρμογές εντός της ίδιας εικονικής μηχανής ή ακόμα και από πολλαπλές εικονικές μηχανές που κάνουν χρήση του vPHI, γεγονός το οποίο επιτρέπει το διαμοιρασμό της συσκευής και στα δύο επίπεδα. Στο Σχήμα 23, δείχνουμε ένα παράδειγμα δύο εικονικών μηχανών (`VM_1` και `VM_N`) που πραγματοποιούν offload υπολογιστικών εργασιών μοιραζόμενες τη μοναδική συσκευή Xeon Phi. Ακολούθως, επικεντρώνουμε την περιγραφή μας σε δύο εικονικές μηχανές, αλλά η διαδικασία είναι η ίδια και για οποιαδήποτε άλλη εικονική μηχανή ακολουθεί αυτό το παράδειγμα. Το runtime κάνει αίτημα για λειτουργίες DMA με το Xeon Phi ως προορισμό για το εκτελέσιμο, τις βιβλιοθήκες κλπ., μέσω της βιβλιοθήκης (`libscif`) (βήμα a). Στη συνέχεια, η `libscif` υποβάλει (βήμα b) την αντίστοιχη κλή-



Σχήμα 22: Σχήμα διαμοιρασμού vPHI

ση συστήματος (`open()`, `close()`, `ioctl()`, `poll()`, `mmap()`) ανάλογα με τη λειτουργία που ζητήθηκε. Η πλειονότητα της λειτουργικότητας του SCIF παρέχεται στο χώρο χρήστη μέσω διαφορετικών εντολών `ioctl()`. Εφόσον το vPHI υλοποιεί λειτουργίες SCIF, τόσο το runtime/εφαρμογή, όσο και η `libscif` παραμένουν άθικτες και δεν απαιτείται ούτε καν επαναμεταγλώττιση. Συνεπώς, το υποβληθέν αίτημα πιάνεται από τον vPHI οδηγό frontend.

vPHI οδηγός frontend: Υλοποιούμε τον vPHI οδηγό frontend ως ένα module για πυρήνα Linux το οποίο εισάγεται δυναμικά στο χώρο πυρήνα του guest. Ο οδηγός λειτουργεί ως μία "κόλλα" μεταξύ της `libscif`, η οποία αγνοεί την ύπαρξη εικονικοποίησης, και της υπόλοιπης στοίβας, προωθώντας τις λειτουργίες που ζητήθηκαν στη vPHI συσκευή backend μέσω των καναλιών επικοινωνίας virtio. Μεταξύ των καθηκόντων του οδηγού frontend είναι να πολυπλέκει τα αιτήματα και να ενορχηστρώνει τα νήματα ή τις διεργασίες του χώρου χρήστη που περιμένουν απάντηση από το συνεπεξεργαστή. Σε αυτό το σημείο, είχαμε δύο σχεδιαστικές επιλογές: θα μπορούσαμε είτε να υλοποιήσουμε μία μέθοδο βασισμένη σε polling είτε μία μέθοδο βασισμένη σε διακοπές (interrupts). Καθώς η πραγματοποίηση busy-waiting σε κοινούς πόρους καταναλώνει κύκλους KME, επιλέξαμε την προσέγγιση που βασίζεται σε διακοπές,



Σχήμα 23: Αρχιτεκτονική του vPHI (μονοπάτι δεδομένων και ελέγχου)

προσθέτοντας ένα κόστος κατά την εγκατάσταση του μηχανισμού αναμονής από τον οδηγό, υπέρ της επίδοσης στις περιπτώσεις: i) που αυξάνεται ο αριθμός από παράλληλα αιτήματα και ii) η όλη διαδικασία διαρκεί αρκετά περισσότερο από την εγκατάσταση του μηχανισμού. Έτσι, ο οδηγός τοποθετεί μία αναφορά για κάποιον buffer στη δομή κοινού δακτυλίου, στη συνέχεια ειδοποιεί τη συσκευή backend (c) ότι υπάρχει ένα εκκρεμές αίτημα και εγκαθιστά τον μηχανισμό αναμονής μέχρι να προκληθεί ένα συμβάν αφύπνισης. Όταν συμβεί το τελευταίο, ο χειριστής διακοπών ελέγχει την τελευταία απάντηση στον κοινό δακτύλιο και αφυπνίζει την αντίστοιχη οντότητα, η οποία συνεχίζει και προωθεί τα δεδομένα προς τα πάνω στη στοίβα. Κατά τη διάρκεια όλης αυτής της διαδικασίας, οι μόνες αντιγραφές που πραγματοποιούνται είναι εκείνες

μεταξύ του χώρου χρήστη και του χώρου πυρήνα σε κάθε κατεύθυνση του μονοπατιού (i, ii). Οποιαδήποτε άλλη ανταλλαγή δεδομένων πραγματοποιείται μέσω αναφορών μειώνοντας το κόστος εικονικοποίησης ειδικά για μεταφορές μεγάλων δεδομένων.

vPHI συσκευή backend: Σχεδιάζουμε τη vPHI συσκευή backend ως μία εικονική συσκευή PCI και την υλοποιούμε ως επέκταση του QEMU. Όπως αναφέραμε προηγουμένως, το backend ειδοποιείται από το frontend (c) όταν ένα νέο αίτημα έχει προστεθεί στο δακτύλιο virtio. Τότε, το backend ελέγχει τον κοινό δακτύλιο και απεικονίζει τον buffer στο δικό του χώρο διευθύνσεων αποφεύγοντας ξανά κάποια αντιγραφή μνήμης. Το backend έχει πρόσβαση στις απεικονίσεις μνήμης φυσικού χώρου διευθύνσεων του guest προς χώρο χρήστη του host, καθώς εκείνο είναι που εγκαθιστά τη μνήμη του guest όταν εκκινεί η εικονική μηχανή. Στη συνέχεια, το backend πραγματοποιεί την αντίστοιχη κλήση συστήματος (d) στον οδηγό SCIF του host και περιμένει το αποτέλεσμα. Όταν η κλήση συστήματος επιστρέψει, προωθεί το αποτέλεσμα στον κοινό δακτύλιο και ειδοποιεί τον guest μέσω μίας εικονικής διακοπής (e). Ακολουθώντας αυτή την προσέγγιση, κάθε εικονική μηχανή στο ίδιο φυσικό μηχάνημα αναπαρίσταται από μία διαφορετική διεργασία QEMU στον host. Έτσι, καθίσταται δυνατός ο διαμοιρασμός του Xeon Phi, καθώς από την οπτική γωνία του οδηγού του host, πολλαπλές εικονικές μηχανές που υποβάλλουν αιτήματα SCIF αποτελούν ουσιαστικά πολλαπλές διεργασίες στον host που εκτελούν παράλληλα κλήσεις συστήματος στον οδηγό SCIF.

Αξιολόγηση της επίδοσης του vPHI

Πειραματική διάταξη

Σε αυτή την ενότητα περιγράφουμε τις μεθόδους πειραματικής αξιολόγησης που χρησιμοποιήσαμε για να αναλύσουμε τη συμπεριφορά του vPHI σε διάφορα σενάρια και παρουσιάζουμε τα αποτελέσματα που προέκυψαν. Εγκαθιστούμε ένα φυσικό μηχάνημα με 1x Intel Xeon E5-2695 v2, 64GB RAM (DDR3-1600Mhz), εξοπλισμένο με ένα συνεπεξεργαστή Intel Xeon Phi 3120P. Εγκαθιστούμε το μηχάνημα ως host εικονικών μηχανών κάνοντας χρήση του QEMU-KVM ως hypervisor.

Αρχικά, υλοποιούμε ένα σύνολο από μικρο-μετροπρογράμματα για να αξιο-

λογήσουμε τη raw επίδοση SCIF της πρώτης έκδοσης του πρωτοτύπου μας. Έπειτα, πραγματοποιούμε ένα υψηλότερου επιπέδου πείραμα χρησιμοποιώντας το `dgemm` από τα samples [38] της Intel για πολλαπλασιασμό πινάκων. Για το πείραμα `dgemm` ακολουθούμε το *native* σχήμα εκτέλεσης σύμφωνα με το μοντέλο εκτέλεσης της Intel.

Στο native σχήμα εκτέλεσης υπάρχουν δύο επιλογές. Ο χρήστης μπορεί είτε να κάνει `ssh` στον επιταχυντή και να εκτελέσει τοπικά την εφαρμογή, είτε να φορτώσει το εκτελέσιμο MIC απευθείας από τον host. Στην πρώτη περίπτωση ο χρήστης χρειάζεται να αντιγράψει ο ίδιος στο συνεπεξεργαστή τα εκτελέσιμα, τις βιβλιοθήκες και τις άλλες εξαρτήσεις και ύστερα να εκτελέσει την εφαρμογή. Σε ένα εικονικοποιημένο περιβάλλον, αυτό καθίσταται δυνατό ρυθμίζοντας ένα δικτυακό bridge στον host μεταξύ της εξομοιωμένης διεπαφής δικτύου `mic0` και της διεπαφής δικτύου που είναι ρυθμισμένη στην εικονική μηχανή. Παρ' όλα αυτά, μία τέτοια ρύθμιση δεν προσφέρεται ιδιαίτερα για νεφοϋπολογιστικά περιβάλλοντα. Τέτοιου είδους ρυθμίσεις μπορούν να καταλήξουν σε ένα σενάριο με πολλούς συνδεδεμένους χρήστες σε ένα κοινό περιβάλλον επιταχυντή καταστρέφοντας τα χαρακτηριστικά απομόνωσης των νεφοϋπολογιστικών υποδομών. Έτσι, δοκιμάζουμε το native σχήμα κάνοντας χρήσης της τελευταίας περίπτωσης που περιγράψαμε, την οποία καθιστά δυνατή το vPHI.

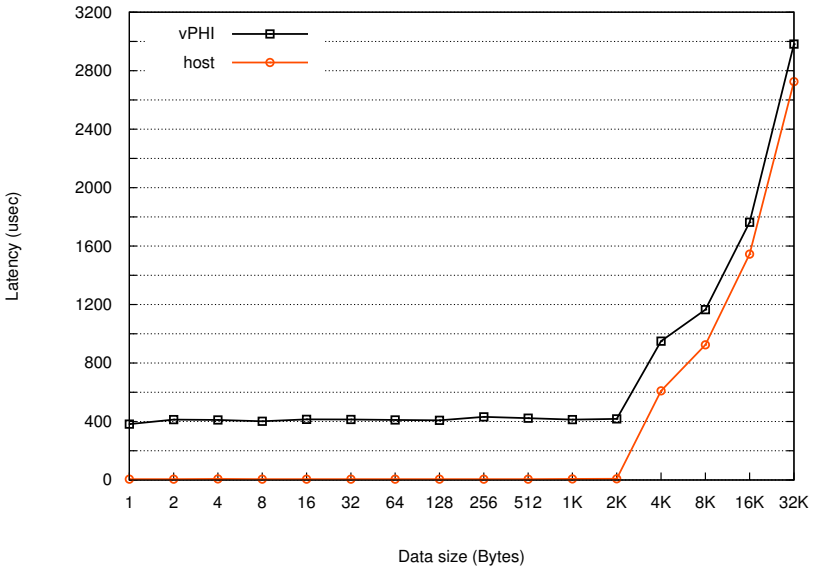
Στη συνέχεια, αξιολογούμε τη δεύτερη (βελτιωμένη) έκδοση του vPHI και εκτελούμε ένα σύνολο από υψηλού-επιπέδου πειράματα βασισμένα στη Σουίτα Μετροπρογραμμάτων Κλιμακώσιμων Ετερογενών Υπολογισμών (Scalable Heterogeneous Computing Benchmark Suite (SHOC) [75]) κάνοντας χρήση του *offload* σχήματος εκτέλεσης. Τέλος, χρησιμοποιούμε ένα από τα μετροπρογράμματα SHOC για να αναδείξουμε το χαρακτηριστικό διαμοιρασμού του vPHI και πώς αυτό συμπεριφέρεται όταν πολλαπλές εικονικές μηχανές κάνουν χρήση ενός συνεπεξεργαστή Xeon Phi.

Επίδοση μικρο-μετροπρογραμμάτων

Υλοποιούμε ένα σύνολο από μικρο-μετροπρογράμματα τα οποία ανταλλάσσουν δεδομένα πάνω από το PCIe μεταξύ του host και του Xeon Phi κάνοντας χρήση του SCIF. Εκτελούμε αυτά τα μετροπρογράμματα με σκοπό να

εκτιμήσουμε το κόστος εικονικοποίησης του vPHI. Αναλύουμε την επίδοση του vPHI χρησιμοποιώντας τόσο two-way επικοινωνία αποστολής-λήψης όσο και απομακρυσμένες λειτουργίες μνήμης. Αρχικά, εκτελούμε το μετροπρόγραμμα στον host, προκειμένου να λάβουμε τη βάση σύγκρισης ως προς την επίδοση. Στη συνέχεια, εκκινούμε μία μονοπύρηνη εικονική μηχανή με το vPHI και εκτελούμε το μετροπρόγραμμα στο εικονικοποιημένο περιβάλλον. Και στις δύο περιπτώσεις, ο αντίστοιχος server εκτελείται στο συνεπεξεργαστή, προκειμένου να εξυπηρετήσει αιτήματα αποστολής SCIF (στην περίπτωση αποστολής-λήψης) ή να εγκαταστήσει καταλλήλως μνήμη συσκευής (στην περίπτωση λειτουργίας απομακρυσμένης μνήμης).

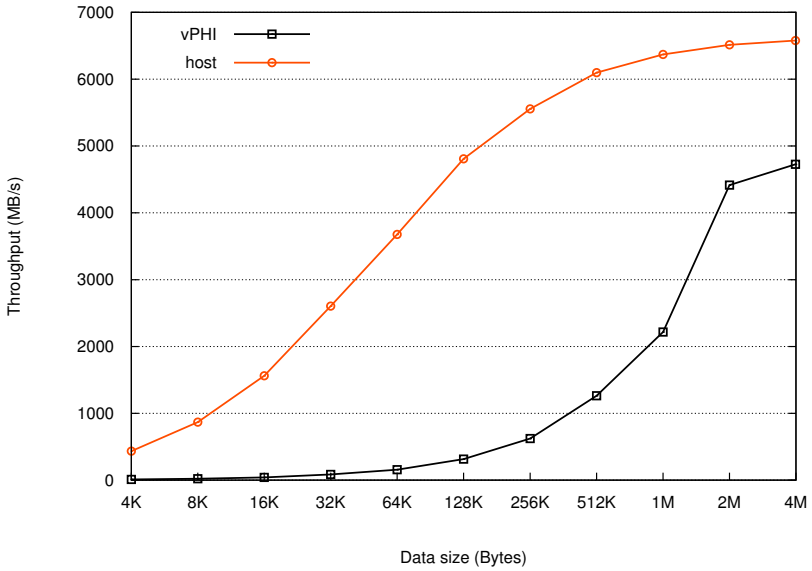
Προκειμένου να μετρήσουμε το latency, χρησιμοποιούμε το μετροπρόγραμμα αποστολής-λήψης, σύμφωνα με το οποίο ένας SCIF server εκκινείται στον επιταχυντή, ακούει για αιτήματα σύνδεσης και όταν εγκαθιδρυθεί μία σύνδεση, ο server μπλοκάρει στη `scif_recv()`, περιμένοντας να σερβίρει δεδομένα στον αντίστοιχο client. Σε αυτό το πλαίσιο, ένας SCIF client εκτελείται στον host (ή στην εικονική μηχανή), ο οποίος συνδέεται με τον server και στέλνει έναν αριθμό από δεδομένα. Απεικονίζουμε το αντίστοιχο μετρηθέν latency τόσο για τον host όσο και για το vPHI στο Σχήμα 24.



Σχήμα 24: Latency επικοινωνίας αποστολής-λήψης

Για την native (`host`) εκτέλεση το latency για αποστολή 1 Byte είναι 7 us, ενώ για την εικονικοποιημένη περίπτωση, το αντίστοιχο latency ανεβαίνει στα 382 us. Έτσι, το κόστος εικονικοποίησης του vPHI είναι 375 us (=382−7). Καθώς πρόκειται για μία αισθητή διαφορά, προχωρήσαμε σε εκτενέστερες breakdown μετρήσεις για να εξετάσουμε περαιτέρω την αιτία αυτού του κόστους. Με βάση αυτή τη breakdown ανάλυση, καταλήγουμε ότι το 93% αυτού του κόστους αποδίδεται στο σχήμα αναμονής του vPHI εντός του οδηγού frontend. Πιο συγκεκριμένα, όταν ο οδηγός frontend υποβάλει ένα αίτημα SCIF στον κοινό δακτύλιο, η αντίστοιχη διεργασία τοποθετείται σε μία ουρά αναμονής, έως ότου ικανοποιηθεί το αίτημα. Όταν το backend τελειώσει την εκτέλεση του αιτήματος, προκαλεί μία εικονική διακοπή και ο χειριστής διακοπών στον guest αναλαμβάνει να ξυπνήσει όλες τις διεργασίες που περιμένουν στην ουρά, οι οποίες ελέγχουν τον κοινό δακτύλιο για να διαπιστώσουν αν η εν λόγω απάντηση είναι για αυτές. Ο μηχανισμός αναμονής και αφύπνισης αποτελεί την κύρια πηγή για τη μείωση της επίδοσης για εργασίες ευαίσθητες ως προς το latency. Όπως αναφέραμε σε προηγούμενες ενότητες, αυτό το σχήμα είναι προτιμητέο για μεταφορές μεγαλύτερου όγκου δεδομένων προκειμένου να μειώσουμε τη χρησιμοποίηση της KME που θα ήταν αυξημένη στην εναλλακτική μέθοδο busy-wait. Μπορούμε να χρησιμοποιήσουμε μία πιθανή υβριδική προσέγγιση η οποία θα άλλαζε σχήμα αναμονής ανάλογα με την εφαρμογή. Πιο συγκεκριμένα, θα μπορούσε να χρησιμοποιεί κάθε φορά το καλύτερο από τα δύο διαθέσιμα σχήματα ανάλογα με το αιτούμενο μέγεθος δεδομένων, επιτρέποντας με αυτό τον τρόπο latency κοντά στο native για μικρά μεγέθη δεδομένων και διατηρώντας αποδεκτό ρυθμό μεταφοράς για μεγαλύτερα μεγέθη. Τέλος, στο Σχήμα 24 παρατηρούμε ότι το προαναφερθέν κόστος παραμένει σταθερό όσο αυξάνεται το μέγεθος των δεδομένων και έτσι προκύπτει ένα σταθερό offset με όρους latency σε σύγκριση με τις μετρήσεις βάσης σύγκρισης.

Στη συνέχεια, εκτελούμε ένα άλλο μετροπρόγραμμα κάνοντας χρήση του SCIF μοντέλου απομακρυσμένης πρόσβασης μνήμης, το οποίο είναι πιο κατάλληλο για μεταφορές μεγαλύτερων δεδομένων, προκειμένου να εκτιμήσουμε το μέγιστο throughput που το vPHI μπορεί να επιτύχει. Σε αυτό το πείραμα, εκκινούμε ένα εκτελέσιμο στο Xeon Phi, το οποίο ακούει ομοίως για εισερχόμενες συνδέσεις και μετά "καρφιτσώνει" (pin) μία περιοχή μνήμης της συσκευ-



Σχήμα 25: Throughput απομακρυσμένης πρόσβασης στη μνήμη

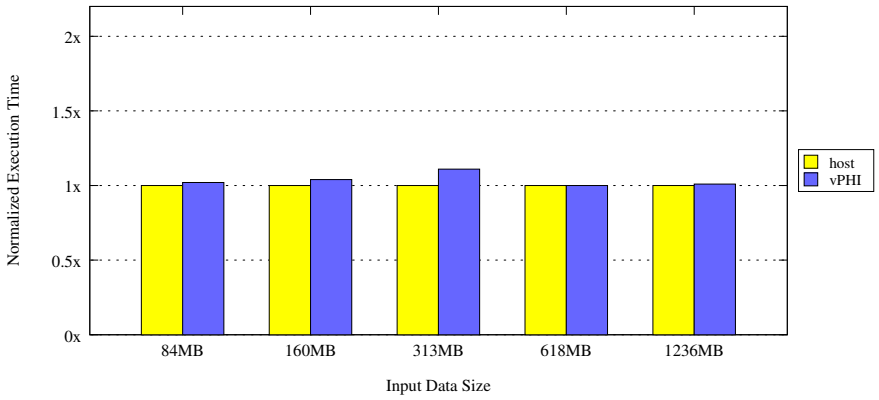
ής με βάση το μέγεθος που ζητήθηκε κάνοντας χρήση της `scif_register()`. Στην πλευρά του host (ή της εικονικής μηχανής) το μετροπρόγραμμα υποβάλει ένα αίτημα σύνδεσης και ύστερα πραγματοποιεί μία απομακρυσμένη ανάγνωση από τη συσκευή επιτάχυνσης. Τα αποτελέσματα απεικονίζονται στο Σχήμα 25. Παρατηρούμε ότι η απομακρυσμένη ανάγνωση του host φτάνει τα 6.4GB/s, ενώ το αντίστοιχο throughput του vPHI είναι 4.6GB/s, το οποίο ισοδυναμεί με το 72% της περίπτωσης του host.

Επίδοση εφαρμογών σε native σχήμα εκτέλεσης

Σε αυτή την ενότητα δείχνουμε τα αποτελέσματα μίας υψηλότερου επιπέδου εφαρμογής που χρησιμοποιήσαμε με την πρώτη έκδοση του vPHI. Μετρήσαμε την εκτέλεση του πολλαπλασιασμού πινάκων `cblas_dgemm` από τα samples [38] της Intel, το οποίο κάνει χρήση της βιβλιοθήκης MKL [37]. Χρησιμοποιούμε το `micnativeloadex`, ένα εργαλείο που παρέχει η Intel για να επιτρέπει τη φόρτωση εκτελέσιμων MIC στο συνεπεξεργαστή απευθείας από τον host, ακολουθώντας την προσέγγιση του native σχήματος. Όπως περιγράψαμε πρωτύτερα, το `micnativeloadex` κάνει χρήση της βιβλιοθήκης COI

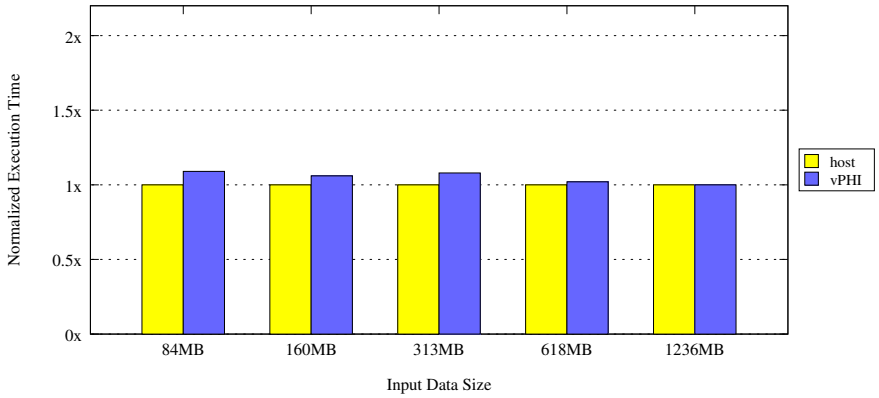
και επικοινωνεί χρησιμοποιώντας το πρωτοκόλλο SCIF με το `coi_daemon` να εκτελείται στο συνεπεξεργαστή. Ο ρόλος του `micnativeloadex` είναι να ρυθμίσει κατάλληλα το περιβάλλον, να φορτώσει τις απαραίτητες βιβλιοθήκες και εκτελέσιμα και να εκκινήσει τον αιτούμενο αριθμό από νήματα.

Σε αυτό το πείραμα εκτελούμε το `micnativeloadex` με παρεχόμενο εκτελέσιμο το `dgemm` στον host και στην εικονική μηχανή. Αφού έχει φορτωθεί το εκτελέσιμο `dgemm` στο Xeon Phi και καθώς εκτελείται ως μία ολότητα χωρίς την παρέμβαση του vPHI, παρατηρούμε ότι δεν υπάρχει μείωση της επίδοσης για το vPHI σε σύγκριση με τον host όσον αφορά στο χρόνο εκτέλεσης στη συσκευή. Όμως, προκειμένου να εκτιμήσουμε το κόστος του vPHI καθ' όλη τη διαδικασία offloading, μετράμε επίσης το συνολικό χρόνο εκτέλεσης από τη στιγμή που το `micnativeloadex` φορτώνεται στον host (ή στην εικονική μηχανή) μέχρι να παραχθούν τα τελικά αποτελέσματα και το εργαλείο να τερματίσει την εκτέλεσή του. Μεταβάλλουμε τον αριθμό των νημάτων όπως επίσης και το μέγεθος των πινάκων. Ο επιταχυντής διαθέτει ένα σύνολο από 57 πυρήνες, από τους οποίους οι 56 είναι διαθέσιμοι προς επεξεργασία, καθώς το uOS χρησιμοποιεί τον εναπομείναντα πυρήνα. Επίσης, παρέχει 4 νήματα ανά πυρήνα προσφέροντας ένα σύνολο από 224 νήματα στο χρήστη. Έτσι, εκτελούμε το μετροπρόγραμμα για 56, 112 και 224 νήματα. Απεικονίζουμε τα αποτελέσματα στο Σχήμα 26, στο Σχήμα 27 και στο Σχήμα 28 για 56, 112 και 224 νήματα αντίστοιχα.

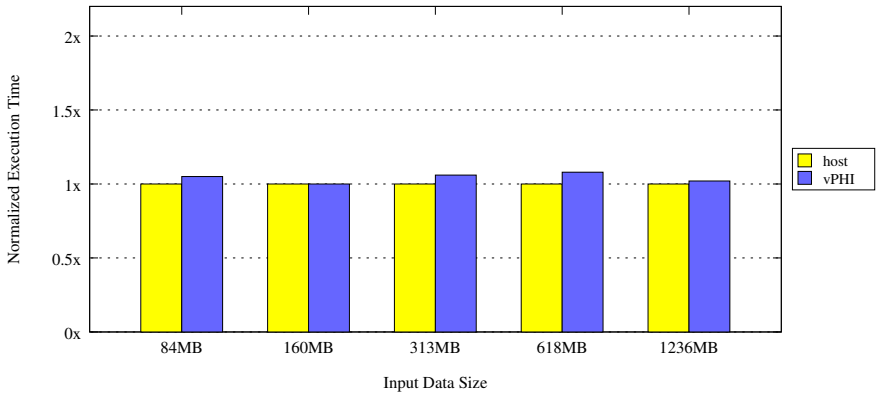


Σχήμα 26: Φόρτωση και εκτέλεση του `dgemm` χρησιμοποιώνας 56 νήματα

Ο άξονας των Ψ αναπαριστά τον κανονικοποιημένο χρόνο εκτέλεσης ο



Σχήμα 27: Φόρτωση και εκτέλεση του `dgemm` χρησιμοποιώνας 112 νήματα



Σχήμα 28: Φόρτωση και εκτέλεση του `dgemm` χρησιμοποιώνας 224 νήματα

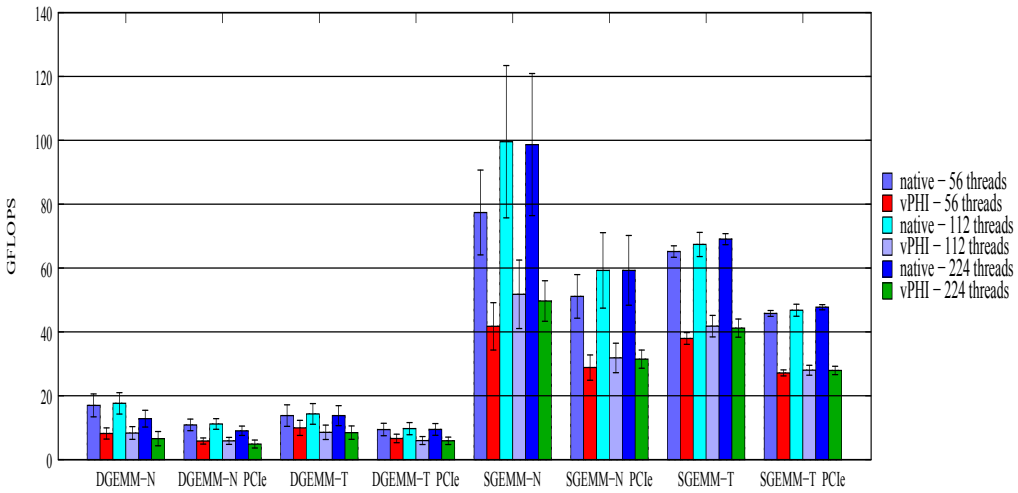
οποίος περιλαμβάνει τη φόρτωση των απαραίτητων binary αρχείων κάνοντας χρήση του `micnativeloadex` από τον host (ή την εικονική μηχανή) και την πραγματική εκτέλεση στον επιταχυντή. Ο άξονας των X αναπαριστά το συνολικό μέγεθος των δύο πινάκων εισόδου. Προσπαθούμε να πραγματοποιήσουμε και να παρουσιάσουμε ένα πείραμα που θα έχει νόημα ως προς το συνολικό μέγεθος των δύο αρχείων εισόδου. Από τα ανωτέρω σχήματα βγάζουμε το συμπέρασμα ότι για μεγαλύτερα πειράματα (τάξης μεγέθους δευτερολέπτων), τα οποία περιλαμβάνουν βρόχους μεγαλύτερης εκτέλεσης όπως επίσης και μεταφορά binary αρχείων (βιβλιοθήκες/εκτελέσιμα) πάνω από το PCIe, το κόστος εικονικοποίησης αποσβένεται και το σχετικό κόστος σε σχέση με το συνολικό χρόνο εκτέλεσης είναι αμελητέο. Αντίθετα, όσο μειώνεται το μέγεθος

των δεδομένων που μεταφέρονται, το κόστος εικονικοποίησης του vPHI έχει μεγαλύτερο αντίκτυπο, όπως τα προηγηθέντα πειράματα ως προς το latency δείχνουν. Παρ' όλα αυτά, δεδομένου του κόστους μίας PCIe λειτουργίας, ένα τυπικό σενάριο στο οποίο εμπλέκεται ένας συνεπεξεργαστής συνήθως περιλαμβάνει τη φόρτωση μίας σημαντικής ποσότητας δεδομένων και ακολούθως μίας βαριάς υπολογιστικής φάσης, καθώς σε αντίθετη περίπτωση δεν αξίζει τον κόπο η πραγματοποίηση offload για μία μικρή ποσότητα δεδομένων και μίας αντίστοιχης εκτέλεσης ενός ελαφρού υπολογισμού που θα μπορούσε να πραγματοποιηθεί στην τοπική ΚΜΕ με μικρό κόστος.

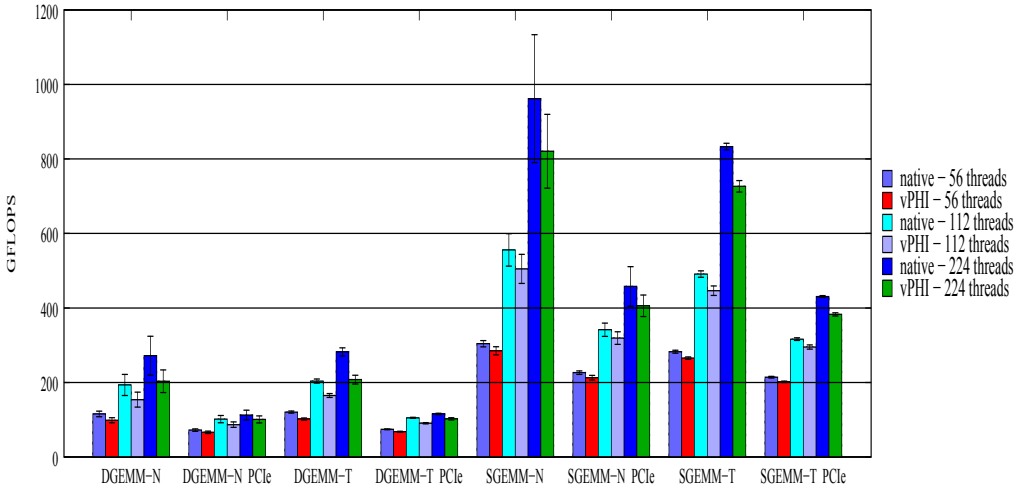
Επίδοση εφαρμογών σε offload σχήμα εκτέλεσης

Σε αυτή την ενότητα παρουσιάζουμε αποτελέσματα από υψηλότερου επιπέδου μετροπρογράμματα που πραγραμματοποιούν offload δημοφιλών υπολογιστικών φορτίων στο Xeon Phi. Χρησιμοποιούμε τη σουίτα SHOC [75] και εκτελούμε μετροπρογράμματα i) στο native host και ii) εντός μίας εικονικής μηχανής σε συνδυασμό με τη δεύτερη έκδοση του vPHI, προκειμένου να προσδιορίσουμε το κόστος και τη στιβαρότητα του συστήματος. Επίσης, μεταβάλλουμε τα νήματα που εκτελούνται στο Xeon Phi και εκτελούμε κάθε μετροπρόγραμμα εκκινώντας 56, 112 και 224 νήματα στον επιταχυντή. Εκτελούμε τα μετροπρογράμματα GEMM και SPMV 10 φορές και απεικονίζουμε τη μέση τιμή και την τυπική απόκλιση.

Αρχικά παρουσιάζουμε το GEMM, το οποίο μετράει την επίδοση πολλαπλασιασμού πίνακα-με-πίνακα χρησιμοποιώντας τη βιβλιοθήκη της Intel MKL (Math Kernel Library) [37]. Εκτελούμε μονής ακρίβειας (single precision (`SP`)) και διπλής ακριβείας (double precision (`DP`)) εκτελέσεις σε native και εικονικοποιημένο περιβάλλον. Χρησιμοποιούμε δύο διαφορετικά μεγέθη προβλημάτων (`s=1` και `s=2`) και επίσης κάνουμε χρήση της λειτουργίας transpose της εφαρμογής. Τα αποτελέσματα απεικονίζονται στο Σχήμα 29 για μέγεθος `s=1` και στο Σχήμα 30 για μέγεθος `s=2`. Πιο συγκεκριμένα, το μέγεθος προβλήματος `s=1` αντιστοιχεί σε 98304 στοιχεία πίνακα για εκτελέσεις `SP` (`float`) και 32768 στοιχεία πίνακα για εκτελέσεις `DP` (`double`), ενώ το μέγεθος προβλήματος `s=2` αντιστοιχεί σε 1179648 στοιχεία πίνακα για εκτελέσεις `SP` (`float`) και 327680 στοιχεία πίνακα για εκτελέσεις `DP` (`double`). Κάθε υπολογιστικός πυ-



Σχήμα 29: Μετροπρόγραμμα GEMM (μέγεθος προβλήματος `s=1`)



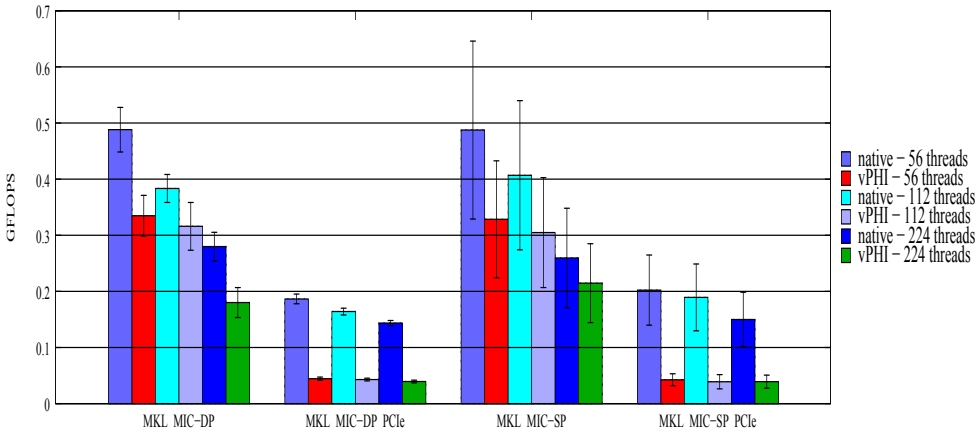
Σχήμα 30: Μετροπρόγραμμα GEMM (μέγεθος προβλήματος `s=2`)

ρήνας εκτελείται 4 φορές. Τέλος, το `_PCIe` επίθεμα υποδεικνύει ότι τα αποτελέσματα περιλαμβάνουν τόσο το χρόνο εκτέλεσης όσο και το χρόνο μεταφοράς, ενώ στα υπόλοιπα αποτελέσματα περιλαμβάνεται αποκλειστικά ο χρόνος εκτέλεσης. Αυτό εξηγεί τη χαμηλότερη επίδοση των `_PCIe` αποτελεσμάτων σε σύγκριση με τα μη `_PCIe` τόσο στη native περίπτωση όσο και στην περίπτωση του vPHI. Σχετικά με τη διαφορά στην επίδοση μεταξύ του native και του vPHI

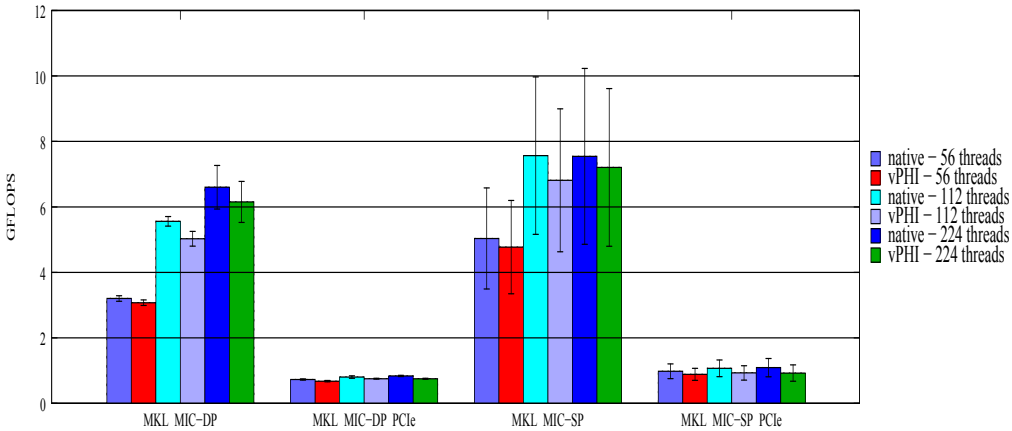
σεναρίου για τις φάσεις υπολογισμού, παρατηρούμε ότι η επίδοση εικονικοποίησης είναι 47%-72% της native περίπτωσης (ανάλογα με τον αριθμό των νημάτων) για το μέγεθος προβλήματος `s=1` και 74%-94% της native περίπτωσης για `s=2`. Το αυξημένο κόστος επίδοσης για `s=1` αποδίδεται στο γεγονός ότι ο υπολογιστικός πυρήνας είναι μικρός και έτσι το κόστος εικονικοποίησης συμμετέχει περισσότερο στο συνολικό χρόνο. Σχετικά με τον αριθμό των νημάτων, τόσο στην εικονικοποιημένη όσο και στη native περίπτωση η επίδοση αυξάνεται σημαντικά μόνο για το μεγαλύτερο (`s=2`) μέγεθος προβλήματος όσο αυξάνουμε τον αριθμό των νημάτων. Πιο συγκεκριμένα, στη native περίπτωση, η επίδοση αυξάνεται κατά 67%-83% από 56 σε 112 νήματα (ανάλογα με τη λειτουργία) και κατά 38%-73% από 112 σε 224 νήματα, ενώ για την εικονικοποιημένη περίπτωση η επίδοση αυξάνεται κατά 56%-77% από 56 σε 112 νήματα και κατά 26%-63% από 112 σε 224 νήματα. Σχετικά με τις `_PCIe` μετρήσεις, η εικονικοποιημένη περίπτωση αποδίδει στο 53%-70% της native περίπτωσης για `s=1` και 85%-94% της native περίπτωσης για μέγεθος προβλήματος `s=2`. Στην πρώτη έκδοση του vPHI (δε φαίνεται στα Σχήματα) υπήρχε μία αισθητή αύξηση του κόστους εικονικοποίησης για τα `_PCIe` αποτελέσματα σε σύγκριση με τις φάσεις υπολογισμού. Αυτή η αύξηση ήταν αναμενόμενη για τα `_PCIe` αποτελέσματα, καθώς αναδεικνύεται ο χρόνος μεταφοράς πάνω από το PCIe, κατά τη διάρκεια του οποίου το SCIF επίπεδο εγκατάστασης (registration) (και κατά συνέπεια το vPHI) χρησιμοποιείται πιο έντονα. Όμως, στη δεύτερη (βελτιωμένη) έκδοση του vPHI παρατηρούμε ότι δεν υπάρχει πια αυτή η αύξηση, καθώς πλέον το κόστος εικονικοποίησης του registration έχει μειωθεί αισθητά. Σχετικά με τον αριθμό των νημάτων για τα `_PCIe` αποτελέσματα με μέγεθος προβλήματος `s=2`, η επίδοση για τη native περίπτωση αυξάνεται κατά 41%-51% από 56 σε 112 νήματα και κατά 10%-36% από 112 σε 224 νήματα, ενώ για την εικονικοποιημένη περίπτωση η επίδοση αυξάνεται κατά 31%-50% από 56 σε 112 νήματα και κατά 13%-30% από 112 σε 224 νήματα. Όπως περιγράφουμε παρακάτω, το προαναφερθέν κόστος εικονικοποίησης μπορεί να αποσβεστεί κατά την εκτέλεση εφαρμογών που μεταφέρουν μία φορά τα δεδομένα τους πάνω από το PCIe και ύστερα εκτελούν πολλαπλούς υπολογισμούς στον επιταχυντή.

Στη συνέχεια, παρουσιάζουμε τα μετροπρογράμματα SPMV από την ίδια σουίτα. Το Σχήμα 31 και το Σχήμα 32 απεικονίζουν τα αποτελέσματα του πει-

ράματος SPMV για s=1 και s=2 αντίστοιχα για μονή και διπλή ακρίβεια χρησιμοποιώντας την Intel MKL. Για το μετροπρόγραμμα SPMV, το μέγεθος προβλήματος s=1 αντιστοιχεί σε 1024 γραμμές, ενώ το μέγεθος προβλήματος s=2 αντιστοιχεί σε 8192 γραμμές του πίνακα. Κάθε πυρήνας SPMV εκτελείται 100 φορές. Παρατηρούμε ότι για s=2 το MKL_MIC-DP με vPHI αποδίδει στο 90%-96% της native περίπτωσης. Αντίστοιχα, η σχετική επίδοση για το MKL_MIC-DP_PCIe είναι 90%-94% της native περίπτωσης (για μέγεθος s=2). Σχετικά με τον αριθμό των νημάτων, παρατηρούμε μία πτώση για μικρότερο μέγεθος προβλήματος τόσο για το vPHI όσο και για τη native περίπτωση, ενώ για s=2 η αύξηση του αριθμού των νημάτων ωφελεί τις εφαρμογές.



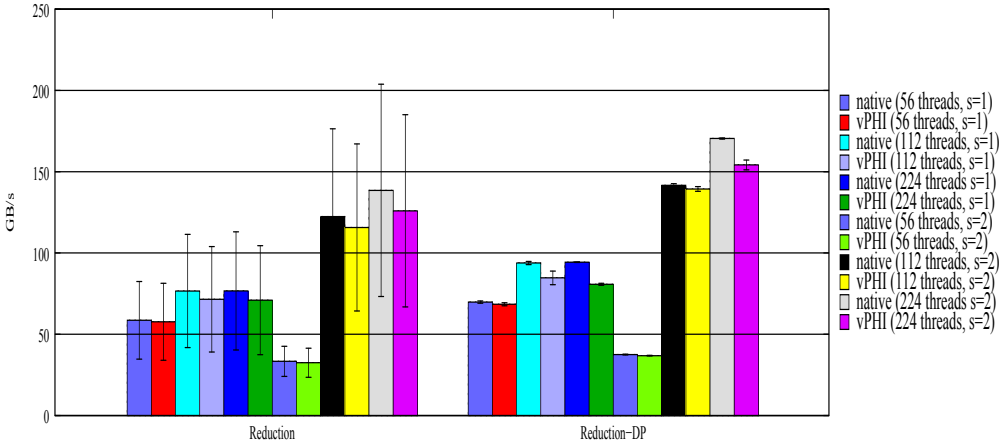
Σχήμα 31: Μετροπρόγραμμα SPMV (μέγεθος προβλήματος s=1)



Σχήμα 32: Μετροπρόγραμμα SPMV (μέγεθος προβλήματος s=2)

Διαμοιρασμός μεταξύ εικονικών μηχανών

Όπως παρατηρήσαμε στα προηγούμενα πειράματα, η επίδοση της μεταφοράς δεδομένων μειώνεται στην εικονικοποιημένη περίπτωση. Για αυτό το λόγο, σε αυτή την ενότητα χτίζουμε σταδιακά ένα σενάριο χρήσης στο οποίο η επιρροή των μεταφορών πάνω από το PCIe είναι μειωμένη. Αυτό επιτυγχάνεται είτε πραγματοποιώντας αυτές τις μεταφορές όταν μία άλλη εικονική μηχανή εκτελεί υπολογισμούς στο συνεπεξεργαστή είτε προφορτώνοντας τα δεδομένα και έπειτα εκτελώντας στη συσκευή λειτουργίες που αφορούν μεγαλύτερης χρονικής διάρκειας πειράματα. Βασίζουμε την ανάλυσή μας στο μετροπρόγραμμα Reduction από τη σουίτα SHOC, το οποίο μετράει την επίδοση της λειτουργίας αθροίσματος από μειώσεις σε αριθμούς κινητής υποδιαστολής.

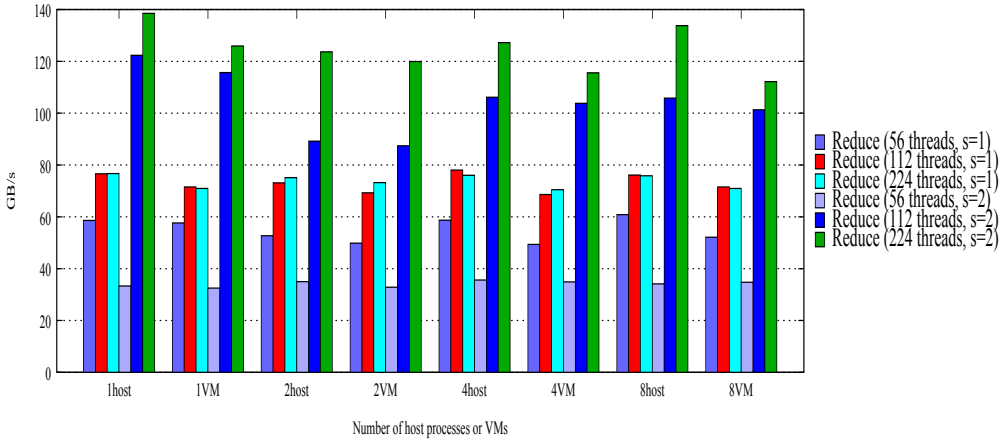


Σχήμα 33: Μετροπρόγραμμα reduction

Αρχικά, δείχνουμε το κόστος εικονικοποίησης μίας εικονικής μηχανής που εκτελεί το μετροπρόγραμμα Reduction. Εκτελούμε το μετροπρόγραμμα 5 φορές και υπολογίζουμε τη μέση τιμή και την τυπική απόκλιση. Απεικονίζουμε τα αποτελέσματα στο Σχήμα 33. Σχετικά με τα μεγέθη προβλημάτων αυτού του μετροπρογράμματος, το μέγεθος προβλήματος `s=1` αντιστοιχεί σε 1048576 στοιχεία για εκτελέσεις SP (`float`) και 524288 στοιχεία για εκτελέσεις DP (`double`), ενώ το μέγεθος προβλήματος `s=2` αντιστοιχεί σε 2097152 στοιχεία για εκτελέσεις SP (`float`) και 1048576 στοιχεία για εκτελέσεις DP (`double`). Κάθε

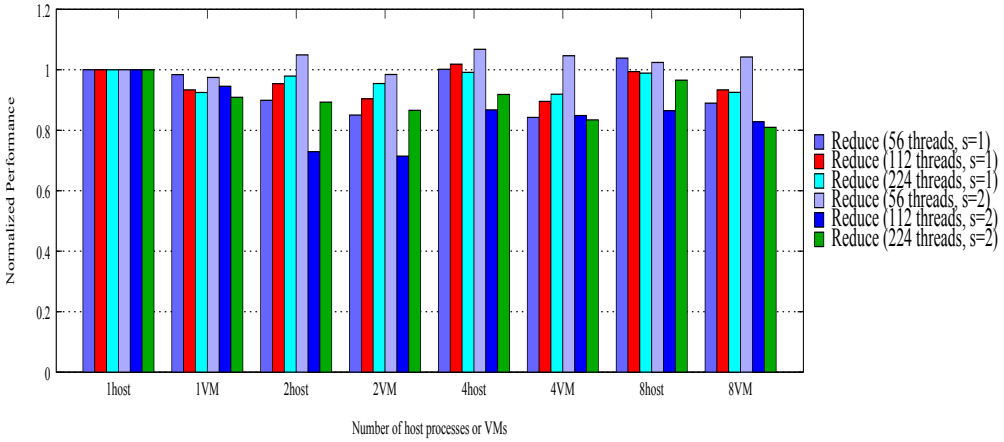
πυρήνας reduction εκτελείται 256 φορές. Σχετικά με το κόστος εικονικοποίησης, το vPHI αποδίδει στο 90%-98% της native περίπτωσης στο `Reduction-DP` για μέγεθος προβλήματος `s=2`. Σχετικά με τον αριθμό των νημάτων, η επίδοση του `Reduction-DP` για `s=2` αυξάνεται σημαντικά από τα 56 στα 112 νήματα (κατά 277% για τη native περίπτωση και κατά 278% για το vPHI).

Στη συνέχεια, δείχνουμε την αποτελεσματικότητα του vPHI όταν χρησιμοποιείται παράλληλα από πολλαπλές εικονικές μηχανές. Σε αυτό το πλαίσιο, εκκινούμε 1, 2, 4 και 8 διεργασίες στον host και αντίστοιχα 1, 2, 4 και 8 μονοπύρηνες εικονικές μηχανές και εκτελούμε το μετροπρόγραμμα Reduction από τη σουίτα SHOC για κάθε εικονική μηχανή (ή host διεργασία) παράλληλα. Σε κάθε εικονική μηχανή (ή host διεργασία), εκτελούμε το μετροπρόγραμμα Reduction 5 φορές και λαμβάνουμε τη μέση τιμή αυτών των εκτελέσεων για κάθε εικονική μηχανή (ή host διεργασία). Επίσης, μεταβάλλουμε τον αριθμό των Xeon Phi νημάτων όπως προηγουμένως. Το Σχήμα 34 και το Σχήμα 35 απεικονίζουν τα αποτελέσματα αυτού του πειράματος για δύο διαφορετικά μεγέθη προβλήματος (`s=1` and `s=2`). Πιο συγκεκριμένα, στο Σχήμα 34 απεικονίζουμε τις raw μετρήσεις, ενώ το Σχήμα 35 δείχνει τα κανονικοποιημένα αποτελέσματα. Τα αποτελέσματα του Σχήματος 35 είναι κανονικοποιημένα ως προς την εκτέλεση 1 host διεργασίας για κάθε ρύθμιση νημάτων.



Σχήμα 34: Διαμοιρασμός μεταξύ εικονικών μηχανών (φάση υπολογισμού - raw αποτελέσματα)

Το Σχήμα 35 δείχνει ότι για το μέγεθος προβλήματος `s=2` και για 56 νή-



Σχήμα 35: Διαμοιρασμός μεταξύ εικονικών μηχανών (φάση υπολογισμού - κανονικοποιημένα αποτελέσματα)

ματα το vPHI επιτυγχάνει επίδοση κοντά στη native για 2 εικονικές μηχανές, 4 εικονικές μηχανές και 8 εικονικές μηχανές σε σύγκριση με την εκτέλεση μίας host διεργασίας. Η αντι-διαισθητική παρατήρηση της χαμηλής πτώσης της επίδοσης ακόμα και όταν κάθε εικονική μηχανή κάνει χρήση όλων των διαθέσιμων νημάτων (224) αποδίδεται στη φύση της εκτέλεσης, καθώς υπάρχουν επικαλυπτόμενες φάσεις τις οποίες ο χρονοδρομολογητής του uOS μπορεί να εκμεταλλευτεί προκειμένου να χρονοδρομολογήσει νήματα από διαφορετικές εικονικές μηχανές. Θεωρούμε ότι για επιταχυντές όπως ο Xeon Phi, στους οποίους προσφέρεται ένα επίπεδο μεταφοράς και επίσης είναι εφικτές δυνατότητες χρονοδρομολόγησης στη συσκευή, υπάρχει αρκετός χώρος για τους νεφοϋπολογιστικούς παρόχους να υιοθετήσουν τα ετερογενή χαρακτηριστικά των υποδομών που προκύπτουν και να παρέχουν δυνητικά δυνατότητες διαμοιρασμού στους χρήστες.

Στο τελευταίο πείραμα, διαμορφώνουμε ένα περιβάλλον προκειμένου να δείξουμε τη χρησιμότητα του συστήματός μας προσομειώνοντας ένα πιο δυναμικό, ρεαλιστικό σενάριο. Πιο συγκεκριμένα, θεωρούμε ένα σενάριο χρήσης με πολλαπλές εικονικές μηχανές σε ένα νεφοϋπολογιστικό περιβάλλον, στο οποίο καθώς εκτελούν μία εφαρμογή στην KME, χρειάζονται να δεσμεύσουν τους πόρους του επιταχυντή για να υπολογίσουν ένα αποτέλεσμα όσα πιο άμεσα γίνεται και ύστερα συνεχίζουν την εκτέλεσή τους στην KME. Με

αυτή την προσέγγιση προσπαθούμε να προσομοιώσουμε δαίμονες (deamons) που εκτελούνται για μεγάλα χρονικά διαστήματα, έχουν προφορτώσει τα δεδομένα τους στον επιταχυντή και κάποια χρονική στιγμή χρειάζονται ένα γρήγορο υπολογισμό. Αυτό επιτυγχάνεται μέσω πολλαπλών παράλληλων εικονικών μηχανών που χρησιμοποιούν τον επιταχυντή κάποια τυχαία στιγμή και με ένα συνεχή και δυναμικό τρόπο. Η ιδέα πίσω από αυτό είναι ότι η πιθανότητα ταυτόχρονης συνεκτέλεσης πυρήνων (kernels) στον επιταχυντή ακριβώς την ίδια στιγμή είναι αισθητά μειωμένη. Επιπλέον, γίνεται πιο αποτελεσματική χρήση των πόρων του επιταχυντή σε σύγκριση με ένα μη-διαμοιραζόμενο σενάριο, σύμφωνα με το οποίο ο επιταχυντής θα μπορούσε να παραμένει υποχρησιμοποιούμενος για εκτενέστερες φάσεις.

Προκειμένου να προσομειώσουμε το προαναφερθέν σενάριο, τροποποιούμε το μετροπρόγραμμα Reduction από τη σουίτα SHOC ως εξής: όταν το μετροπρόγραμμα εκκινεί σε μία εικονική μηχανή, πρώτα αντιγράφει τα δεδομένα πάνω από το PCIe, πραγματοποιεί μία εκτέλεση ζεστάματος (warm-up) και στη συνέχεια συγχρονίζεται μέχρις ότου όλες οι εικονικές μηχανές φτάσουν σε αυτή τη φάση. Ύστερα από αυτό το σημείο, εκκινούμε το χρονιστή, καθώς η προφόρτωση των δεδομένων θεωρείται σε αυτό το πείραμα εκτός του κρίσιμου μονοπατιού. Στη συνέχεια, εκτελεί ένα βρόγχο, σύμφωνα με τον οποίο κοιμάται για μία τυχαία χρονική περίοδο (από το σύνολο [0..3]δευτερόλεπτα) και όταν ξυπνήσει, εκτελεί τον πυρήνα reduction για 1024 επαναλήψεις. Μετά την εκτέλεση του πυρήνα, κοιμάται ξανά. Η εφαρμογή συνεχίζει αυτό τον κύκλο με τα δεδομένα του μετροπρογράμματος ήδη φορτωμένα και τερματίζει μετά από μία συγκεκριμένη χρονική διάρκεια. Αναθέτουμε σε κάθε εφαρμογή 224 νήματα και εκτελούμε αυτούς τους βρόγχους στις εικονικές μηχανές για 5 λεπτά συνολικά. Θεωρούμε την ολοκλήρωση μίας λειτουργίας ως το σύνολο των 1024 επαναλήψεων του πυρήνα reduction.

Αυτό το πείραμα συνεπάγεται μία δυναμική εκτέλεση πολλαπλών εικονικών μηχανών που χρησιμοποιούν τον επιταχυντή, όταν τον χρειάζονται με τα δεδομένα τους φορτωμένα για όλο αυτό το χρονικό διάστημα. Η μετρική σε αυτό το πείραμα είναι το άθροισμα των λειτουργιών όλων των εικονικών μηχανών που ολοκληρώνουν την εκτέλεσή τους στο σύστημα. Επίσης, εκτελούμε το ίδιο μετροπρόγαμμα ως μία host διεργασία και συγκρίνουμε τα αποτελέσματα. Η ιδέα πίσω από αυτή την επιλογή είναι, ότι παρά το γεγονός ότι μία

host εφαρμογή μπορεί να έχει ως αποτέλεσμα καλύτερη επίδοση σε σύγκριση με μία εικονική μηχανή, από την άλλη πλευρά όταν εκτελούνται πολλαπλές εικονικές μηχανές, αξιοποιούν τις άεργες φάσεις της εφαρμογής και έτσι το συνολικό throughput αυξάνεται. Η εκτέλεση σε μία host εφαρμογή αναπαριστά την απευθείας ανάθεση του επιταχυντή σε μία μόνο οντότητα. Σε αντίθεση με τα προηγούμενα πειράματα SHOC, στα οποία χρησιμοποιούσαμε affinity για τα νήματα του επιταχυντή για να εξαπλωθούν στους πυρήνες (cores) της συσκευής, σε αυτό το πείραμα θέτουμε μόνο τον αριθμό των νημάτων χωρίς τη χρήση affinity προσπαθώντας να προσεγγίσουμε ένα πιο ρεαλιστικό σενάριο, σύμφωνα με το οποίο ο χρήστης της εικονικής μηχανής δε γνωρίζει τα affinities των άλλων εικονικών μηχανών και με αυτό τον τρόπο επιτρέπει στο χρονοδρομολογητή του επιταχυντή να διαχειριστεί αυτή την εργασία. Απεικονίζουμε τα αποτελέσματα στον Πίνακα 2.

	Συνολικές λειτουργίες
1 host διεργασία	199
1 εικονική μηχανή	188
2 εικονικές μηχανές	370
4 εικονικές μηχανές	709
8 εικονικές μηχανές	412

Πίνακας 2: Σύνολο λειτουργιών μετά από 5 λεπτά εκτέλεσης

Παρατηρούμε ότι το συνολικό throughput του συστήματος αυξάνεται έως και 3.56x για 4 εικονικές μηχανές και στη συνέχεια, αυξάνοντας τον αριθμό των εικονικών μηχανών, το συνολικό throughput αρχίζει να μειώνεται εξαιτίας του αυξημένου ανταγωνισμού στον επιταχυντή.

Σύνοψη

Σε αυτό το κεφάλαιο, παρουσιάσαμε το vPHI, ένα σύστημα παραεικονικοποίησης για επιταχυντές Xeon Phi, το οποίο επιτρέπει το διαμοιρασμό της συσκευής μεταξύ εικονικών μηχανών που εκτελούνται στον ίδιο κόμβο. Σχεδιάσαμε και υλοποιήσαμε ένα στρώμα εικονικοποίησης του χαμηλού επιπέδου μεταφοράς, καθώς τα αντίστοιχα μέρη του λογισμικού παρέχονται ως *ανοι-*

χτού κώδικα. Χρησιμοποιώντας αυτή την τεχνική, αποφεύγουμε την *πολυπλοκότητα* μίας πλήρως εικονικοποιημένης συσκευής και παρέχουμε *διαφάνεια* στα ανώτερα επιπέδα του λογισμικού που κάνουν χρήση του SCIF. Επιπλέον, μειώνουμε το κόστος εικονικοποίησης λόγω της αντίστοιχης *μη γνώσης* του επιταχυντή της διάσπαρτης φυσικής μνήμης του guest μεταβάλλοντας τον οδηγό του host. Κατά την πειραματική αποτίμηση της μεθόδου μας, προσπαθήσαμε να δημιουργήσουμε και να παρουσιάσουμε ένα πείραμα που εκτελείται για μεγάλο χρονικό διάστημα σε ένα loop με τις εικονικές μηχανές να πραγματοποιούν ένα γρήγορο υπολογισμό στον επιταχυντή με προφορτωμένα δεδομένα και στη συνέχεια να κοιμούνται για μία τυχαία χρονική διάρκεια. Με αυτό τον τρόπο, η πιθανότητα ταυτόχρονης συνεκτέλεσης πυρήνων (kernels) στον επιταχυντή ακριβώς την ίδια στιγμή μειώνεται αισθητά και κάθε εικονική μηχανή μπορεί να γίνει "schedule in" στον επιταχυντή με μικρότερο κόστος εξαιτίας της ύπαρξης των δεδομένων της στη μνήμη της συσκευής. Με το vPHI παρέχουμε το μηχανισμό για εφαρμογή αλγορίθμων *χρονοδρομολόγησης*, αλλά η εκτενής μελέτη και εξερεύνηση σχετικών μεθόδων είναι εκτός του πλαισίου αυτής της διατριβής. Προκειμένου να εφαρμοστούν πιο αποτελεσματικές πολιτικές χρονοδρομολόγησης και να υπάρξει πιο αποδοτικός διαμοιρασμός τέτοιων συσκευών, θεωρούμε ότι οι επιταχυντές νέας γενιάς χρειάζεται να σχεδιαστούν με γνώση της δυνητικής ύπαρξης εικονικοποιημένων οντοτήτων στα συστήματα στα οποία είναι προσαρτημένοι.

Σχετικές εργασίες

Σε αυτή τη διατριβή εξετάζουμε το πρόβλημα της εικονικοποίησης συσκευών επιτάχυνσης και σε αυτό το πλαίσιο έχουμε διερευνήσει διάφορες σχετικές τεχνικές από προηγούμενες εργασίες. Στη συνέχεια, περιγράφουμε τις σχετικές εργασίες και τονίζουμε τα βασικά χαρακτηριστικά κάθε μιας σε σχέση με τις δικές μας προσεγγίσεις.

Ενδοκομβική επικοινωνία

Σε προηγούμενη ενότητα, περιγράψαμε τη δική μας προσέγγιση χρήσης του rCUDA πάνω από το V4VSockets, το βελτιστοποιημένο ενδοκομβικό σύστημά μας για επικοινωνίας στον hypervisor Xen. Σε αυτή την ενότητα περι-

γράφουμε τις προηγούμενες εργασίες για βελτιστοποίηση εικονικών μηχανών που εκτελούνται στον ίδιο φυσικό κόμβο.

Στο [82] ο συγγραφέας παρέχει μία εκτενή μελέτη των διαθέσιμων μηχανισμών ενδοκομβικής επικοινωνίας μεταξύ εικονικών μηχανών. Σε αυτό το πλαίσιο, αναφέρει ότι η βασική αιτία κόστους ενδοκομβικής επικοινωνίας είναι το πολύπλοκο μονοπάτι δεδομένων μεταξύ των συνυπαρχουσών εικονικών μηχανών. Η δικτυακή κίνηση μεταξύ των εικονικών μηχανών που επικοινωνούν μεταξύ τους ανακατευθύνεται μέσω του driver domain, γεγονός που συνεπάγεται σημαντική μείωση της επίδοσης. Η αποστολή και λήψη πακέτων περιλαμβάνει τη διάσχιση της δικτυακής στοίβας TCP/IP και την εκτέλεση πολλαπλών υπερκλήσεων Xen. Αρκετές βελτιστοποιήσεις έχουν προταθεί σχετικά με αυτό τον περιορισμό, όπως η χρήση τεχνικών κοινής μνήμης, τις οποίες παρέχει ο hypervisor Xen, για την απλοποίηση της ανταλλαγής δεδομένων μεταξύ των εικονικών μηχανών. Η χρήση μίας δεξαμενής από μοιραζόμενες σελίδες για απευθείας ανταλλαγή πακέτων φαίνεται πολύ πιο αποδοτική σε σχέση με τη διάσχιση του δικτυακού μονοπατιού επικοινωνίας μέσω του driver domain. Το XenSocket [85] και το IVC [33] παρέχουν ένα βασικό, one-way κανάλι επικοινωνίας με χρήση σημασιολογίας socket, εισάγοντας ένα νέο τύπο οικογένειας διευθύνσεων. Το XenLoop [83], από την άλλη πλευρά, πιάνει τις κλήσεις στις τοπικές εικονικές μηχανές μέσω του Linux μηχανισμού netfilter και εγκαθιστά ένα πλήρως-αμφίδρομο κανάλι δεδομένων μεταξύ των εικονικών μηχανών για αποδοτική ανταλλαγή δεδομένων. Στο XWay [42] οι συγγραφείς ορίζουν μία νέα εικονική συσκευή που εγκαθιστά απευθείας επικοινωνία μεταξύ των εικονικών μηχανών, παρακάμπτοντας εντελώς το driver domain. Επιπρόσθετα, στο [88] οι συγγραφείς προτείνουν ένα βελτιστοποιημένο μονοπάτι μέσω του hypervisor Xen με χρήση κοινής μνήμης, ενώ το SChannel [32] παρέχει ένα αμφίδρομο κανάλι κοινής μνήμης μέσω του driver domain. Το MMNet [64] αποτρέπει τις αντιγραφές απεικονίζοντας ολόκληρο το χώρο διευθύνσεων του πυρήνα μίας εικονικής μηχανής στην αντίστοιχη εικονική μηχανή με την οποία επικοινωνεί. Επιπλέον, το YASMIN [66] παρέχει ένα βελτιστοποιημένο ενδοκομβικό μονοπάτι επικοινωνίας μέσω του hypervisor, αλλά με χρήση της λιγότερο ασφαλούς τεχνικής της κοινής μνήμης. Στο [27] οι Guan et al. προτείνουν μία τεχνική χρονοδρομολόγησης μεταξύ συνυπαρχουσών εικονικών μηχανών που λαμβάνει υπόψη

την αντίστοιχη ενδοκομβική επικοινωνία, προκειμένου να μειωθεί το δικτυακό latency.

Πέραν των προαναφερθεισών μεθόδων που εκμεταλλεύονται τη συνύπαρξη των εικονικών μηχανών που επικοινωνούν μεταξύ τους, έχουν γίνει σημαντικές προσπάθειες κάνοντας χρήση λιγότερο παρεμβατικών προσεγγίσεων με στόχο τη βελτιστοποίηση της υπάρχουσας δικτυακής στοίβας του Xen. Για παράδειγμα, οι Menon et al. [48] βελτιώνουν τη δικτυακή επίδοση εισάγοντας αντιγραφές αντί για επαναπεικονίσεις σελίδων και κάνοντας χρήση προηγμένων χαρακτηριστικών μνήμης, όπως οι superpages και οι καθολικές απεικονίσεις σελίδων.

Αντίστοιχες προσεγγίσεις με τις προαναφερθείσες εργασίες έχουν προταθεί επίσης και για άλλους hypervisors [14, 28]. Πιο συγκεκριμένα, στο [14] οι Diakhate et al. έχουν προτείνει μία προσέγγιση για το KVM με χρήση τεχνικών κοινής μνήμης μέσω μίας εικονικής συσκευής ανταλλαγής μηνυμάτων στοχεύοντας εικονικοποιημένα πλαίσια εφαρμογών MPI.

Στην προσέγγισή μας, χτίζουμε σε αυτό το σκέλος της βιβλιογραφίας και αντί να βελτιστοποιήσουμε το μονοπάτι δεδομένων μέσω του driver domain, το παρακάμπτουμε, κάνοντας χρήση του hypervisor ως δικτυακού μέσου. Επιπλέον, αποφεύγουμε την τεχνική της κοινής μνήμης, ενισχύοντας τις ιδιότητες της απομόνωσης και της ασφάλειας για τις εικονικές μηχανές που επικοινωνούν μεταξύ τους.

Εικονικοποίηση επιταχυντών

Από το πεδίο της εικονικοποίησης επιταχυντών, οι περισσότερες προσεγγίσεις που έχουν προταθεί επικεντρώνονται κυρίως στις GPUs. Στη νεφοϋπολογιστική βιομηχανία, πολλοί πάροχοι, όπως η Amazon [77], η Microsoft [50] κλπ. έχουν αρχίσει να προσφέρουν υπολογιστικούς πόρους GPU ως υπηρεσία. Επιπλέον, μία κλάση από προτεινόμενες λύσεις περιλαμβάνει τη χρήση της τεχνολογίας passthrough για να παρέχει σε μία εικονική μηχανή απευθείας πρόσβαση σε μία συσκευή του host. Επιπρόσθετα, υπάρχουν λύσεις [78, 41] οι οποίες παρέχουν ένα στατικό αριθμό από εικονικές GPUs, κάθε μία από τις οποίες ανατίθεται απευθείας σε μία εικονική μηχανή. Άλλες εργασίες υπάγονται στην κατηγορία της πλήρους εικονικοποίησης [76], σύμφωνα με την

οποία δεν πραγματοποιείται κάποια τροποποίηση στο guest λειτουργικό σύστημα. Κάποιες προσεγγίσεις που έχουν προταθεί κάνουν χρήση της τεχνικής παραεικονικοποίησης [25, 26, 81], ενώ άλλες εφαρμόζουν μεθόδους και πλήρους εικονικοποίησης και παραεικονικοποίησης [71].

Η τεχνική της ανακατεύθυνσης της προγραμματιστικής διεπαφής (API redirection) με μία προσέγγιση client-server έχει χρησιμοποιηθεί από κάποιες εργασίες προκειμένου να προσφέρουν χαρακτηριστικά εικονικοποιήσης GPU [69]. Σε αυτό το πλαίσιο, στο rCUDA [62] οι συγγραφείς καθιστούν δυνατή την απομακρυσμένη εκτέλεση εργασιών GPU, όπως έχουμε αναφέρει. Αυτές οι εργασίες βασίζονται σε ένα μοντέλο client-server με επικοινωνία πάνω από ένα επίπεδο μεταφοράς. Παρά το γεγονός ότι αυτό κάνει αυτές τις προσεγγίσεις ανεξάρτητες από τον hypervisor, χρειάζεται να εφαρμοστούν βελτιστοποιήσεις για το συγκεκριμένο επίπεδο μεταφοράς με βάση το διασυνδεδεμένο περιβάλλον. Σε αυτή τη διατριβή, εφαρμόσαμε το V4VSockets ως το μέσο μεταφοράς του rCUDA, προκειμένου να αξιοποιήσουμε τη συνύπαρξη των εικονικών μηχανών στο πλαίσιο μίας GPU η οποία διαμοιράζεται από πολλαπλές εικονικές μηχανές. Τέλος, πρόσφατα, στο [19] χρησιμοποιήθηκε επίσης η τεχνική απομακρυσμένης επιτάχυνσης για το συνεπεξεργαστή Xeon Phi.

Όπως αναφέραμε σε προηγούμενη ενότητα, οι βασικές εταιρείες κατασκευής GPU παρέχουν τον πηγαίο τους κώδικα με έναν κλειστό τρόπο. Λαμβάνοντας υπόψη αυτό, οι αντίστοιχες προσεγγίσεις που εικονικοποιούν το στρώμα χαμηλού επιπέδου της στοίβας είναι μη πρακτικές, καθώς βασίζονται κυρίως σε μεθόδους reverse engineering, οι οποίες καθίστανται μη χρήσιμες σε νέες εκδόσεις συσκευών GPU στην περίπτωση που οι αντίστοιχες εταιρείες εισάγουν σημαντικές τροποποιήσεις. Επιπλέον, οι εργασίες που υλοποιούν μία wrapper υψηλού επιπέδου βιβλιοθήκη η οποία ανακατευθύνει το μονοπάτι σε τροποποιημένα υποσυστήματα χρειάζεται να ενημερώνονται συνεχώς όταν μία νέα λειτουργικότητα προστίθεται από τους κατασκευαστές GPU. Σε αυτή τη διατριβή, βασίζουμε την προσέγγισή μας σε ένα middleware σύστημα (rCUDA) το οποίο παραμένει ενημερωμένο παρέχοντας σταθερά νέες εκδόσεις.

Σχετικά με την εικονικοποίηση του Xeon Phi, σύμφωνα με όσα γνωρίζουμε, η προσέγγισή μας με το vPHI είναι η πρώτη και αυτή τη στιγμή η μοναδική λύση που καθιστά δυνατό το διαμοιρασμό μίας συσκευής Xeon Phi με-

ταξύ πολλαπλών εικονικών μηχανών. Παρ' όλα αυτά, υπάρχουν κάποιες προσεγγίσεις που σχετίζονται με αυτό το πρόβλημα μέσω μίας διαφορετικής οπτικής γωνίας. Σε αυτό το πλαίσιο, η Intel παρέχει ένα σύνολο από patches τόσο για τον Xen [24], όσο και για το KVM [23], για απευθείας ανάθεση (*PCIe passthrough*) του συνεπεξεργαστή σε μία μόνο εικονική μηχανή. Η ίδια τεχνική χρησιμοποιείται και από τη VMware [79] στο δικό της hypervisor (ESXi). Παρ' όλα αυτά, οι μέθοδοι που βασίζονται στο PCIe passtrough παρουσιάζουν ένα βασικό μειονέκτημα, όπως περιγράψαμε σε προηγούμενη ενότητα. Παρά το γεγονός ότι μία εφαρμογή που εκτελείται σε μία τέτοια εικονική μηχανή μπορεί να αποδώσει σε ρυθμό κοντά στο native, δεν υπάρχει η δυνατότητα διαμοιρασμού μίας φυσικής συσκευής μεταξύ πολλαπλών εικονικών μηχανών.

Στο πλαίσιο εικονικοποίησης πόρων του Xeon Phi, η ScaleMP [1] παρέχει μία λύση που δημιουργεί ένα επίπεδο αφαίρεσης της μνήμης και των υπολογιστικών πόρων παρέχοντάς το στο host περιβάλλον. Με αυτό τον τρόπο, μία εφαρμογή με υψηλές απαιτήσεις σε πυρήνες (cores) ή μνήμη μπορεί να χρησιμοποιήσει αυτή την πλατφόρμα με ένα διαφανή τρόπο. Αυτή η προσέγγιση παρέχει τελικά μία μεγάλη SMP υποδομή στο χρήστη αντί για ένα περιβάλλον που κάνει χρήση του μοντέλου εκτέλεσης offload σε εικονικές μηχανές.

Επιπλέον, υπάρχουν κάποιες προσεγγίσεις που παρέχουν ή βελτιώνουν τεχνικές για εικονικοποίηση ή διαμοιρασμό FPGAs, χρησιμοποιώντας ένα παραδοσιακό FPGA ως εικονικό επαναρυθμίσιμο υλικό [34], μειώνοντας την πολυπλοκότητα επικοινωνίας μεταξύ των εφαρμογών και των FPGA πόρων [46], προσφέροντας FPGA-ως-Υπηρεσία [93], ενσωματώνοντας τις FPGAs στις νεφοϋπολογιστικές στοίβες [7, 9, 18, 8, 2, 73, 91, 74, 63], επικεντρώνοντας στις ιδιότητες της χρονοδρομολόγησης [92, 59], της εμπιστευτικότητας [87] και της ασφαλούς εκτέλεσης [30] σε περιβάλλοντα εικονικοποίησης FPGA ή στοχεύοντας σε εικονικές δικτυακές εφαρμογές [90].

Τέλος, οι Yu et al. [89] προτείνουν μία προσέγγιση για αυτόματη εικονικοποίηση APIs διαφορετικών επιταχυντών. Παρ' όλα αυτά, προκειμένου να κατασκευάσουν τα εικονικοποιημένα στρώματα, απαιτείται ως είσοδος μία προδιαγραφή του API. Στη συνέχεια, όπως υποστηρίζουν οι συγγραφείς, ένας προγραμματιστής μπορεί να εικονικοποίησει το νέο API σε διάστημα λίγων ημερών.

Συμπεράσματα και μελλοντικές κατευθύνσεις

Η ετερογενής υπολογιστική επεξεργασία έχει αρχίσει να παίζει σημαντικό ρόλο στα υπολογιστικά συστήματα υψηλής επίδοσης και εξαιτίας των προβλημάτων ισχύος-πυκνότητας διαφαίνεται ότι έχει μεγάλη δυναμική για τα μελλοντικά κέντρα δεδομένων. Την ίδια στιγμή, μία σημαντική ποσότητα HPC εργασιών εκτελούνται σε νεφοϋπολογιστικά περιβάλλοντα, λόγω των πλεονεκτημάτων που προσφέρουν τόσο στους χρήστες όσο και στους παρόχους των υποδομών. Σε αυτό το πλαίσιο, υπάρχει μία αυξανόμενη ανάγκη να γεφυρωθεί το χάσμα μεταξύ του οικοσυστήματος των επιταχυντών και του κόσμου της εικονικοποίησης, ο οποίος αποτελεί το δομικό στοιχείο των νεφοϋπολογιστικών περιβαλλόντων.

Με βάση αυτές τις παρατηρήσεις, αυτή η διατριβή επικεντρώθηκε σε τεχνικές εικονικοποίησης συσκευών επιτάχυνσης στοχεύοντας στη διαφάνεια των εφαρμογών, στη φορητότητα σε μελλοντικά περιβάλλοντα και στην ύπαρξη αποδεκτού κόστους λόγω των επιπρόσθετων επιπέδων εικονικοποίησης. Για το σκόπό αυτό, περιγράψαμε συνοπτικά και κατηγοριοποιήσαμε τις παραδοσιακές μεθόδους εικονικοποίησης Ε/Ε και βάσει αυτών εντοπίσαμε τις βασικές προκλήσεις της εικονικοποίησης επιταχυντών σε σύγκριση με τις παραδοσιακές συσκευές Ε/Ε.

Περιγράψαμε εναλλακτικά μονοπάτια για μεταφορές δεδομένων μέσω διαφόρων επιπέδων εικονικοποίησης και σκιαγραφήσαμε τους κύριους σχεδιαστικούς συμβιβασμούς για αποδοτικά συστήματα εικονικοποίησης επιταχυντών. Σε αυτό το πλαίσιο, προτείναμε δύο προσεγγίσεις για δύο διαφορετικά οικοσυστήματα επιταχυντών, κάθε ένα από τα οποία έχει τα δικά του χαρακτηριστικά. Με αυτές τις προσεγγίσεις, προσπαθήσαμε να ξεπεράσουμε ή να παρακάμψουμε τις προκλήσεις εικονικοποίησης επιταχυντών που περιγράψαμε σε προηγούμενη ενότητα, παρά το γεγονός ότι για ορισμένες από αυτές χρειάζεται δυνητικά υποστήριξη από το υλικό, προκειμένου να αντιμετωπιστούν αποτελεσματικά. Επιπλέον, επικεντρωνόμαστε στις επιθυμητές ιδιότητες που περιγράψαμε στην εισαγωγική ενότητα. Πιο συγκεκριμένα, καθώς προτείνουμε προσεγγίσεις βασισμένες στο λογισμικό, ενεργοποιούμε το διαμοιρασμό του επιταχυντή, την ευελιξία (π.χ. τη δυνατότητα για migration ή για εφαρμογή πολιτικών χρονοδρομολόγησης) και μειώνουμε το συνολικό κόστος μί-

ας cloud υποδομής, το οποίο θα αυξανόταν στην περίπτωση χρήσης ακριβών IOV λύσεων. Σχετικά με τη διαφάνεια, η πρώτη μας προσέγγιση είναι συμβατή με CUDA Runtime εφαρμογές, ενώ το δεύτερο framework μας διατηρεί τη συμβατότητα με όλα τα στρώματα του λογισμικού που χρησιμοποιούν το επίπεδο μεταφοράς του επιταχυντή. Τα συστήματά μας είναι επίσης όσο το δυνατόν λιγότερο παρεμβατικά, εκτός από μία μετρίου μεγέθους τροποποίηση που πραγματοποιήσαμε στον host οδηγό του Xeon Phi για λόγους βελτιστοποίησης. Σε αυτό το πλαίσιο, προσπαθήσαμε να μειώσουμε το κόστος εικονικοποίησης των προσεγγίσεών μας σε σύγκριση με native εκτελέσεις. Τέλος, διαμορφώσαμε ένα πείραμα με ένα μη-batch μετροπρόγραμμα για να δείξουμε την πιο αποτελεσματική χρησιμοποίηση του επιταχυντή.

Αρχικά, επικεντρωθήκαμε στις συσκευές NVIDIA GPU και προτείναμε τη χρήση ενός απομακρυσμένου συστήματος επιτάχυνσης σε πλατφόρμα εικονικοποίησης ενός κόμβου σε συνδυασμό με ένα ενδοκομβικό σύστημα επικοινωνίας χαμηλού κόστους που επιτρέπει στις εφαρμογές σε εικονικοποιημένα περιβάλλοντα να πραγματοποιούν αποδοτικά offloading. Περιγράψαμε το σχεδιασμό του ενδοκομβικού μας συστήματος, που ονομάζεται V4VSockets, κάνοντας χρήση του hypervisor ως δικτυακού μέσου, αντί για το driver domain. Παρά το γεγονός ότι σε αυτή τη διατριβή επικεντρωθήκαμε στη χρήση του V4VSockets σε σενάρια χρήσης GPU, το ενδοκομβικό μας σύστημα επικοινωνίας μπορεί να χρησιμοποιηθεί και σε συνδυασμό με άλλες socket εφαρμογές, καθώς παρέχει socket συμβατότητα. Επιπλέον, αξιολογήσαμε την προσέγγισή μας τόσο χρησιμοποιώντας δικτυακά μικρο-μετροπρογράμματα όσο και αναλύοντας ένα σύνηθες GPU stencil. Πέραν των πλεονεκτημάτων από πλευράς επίδοσης, δείξαμε ότι η έμμεση προσέγγισή μας για εικονικοποίηση GPU συμβαδίζει με επιταχυντές που είναι βασισμένοι σε κλειστές στοίβες λογισμικού, διότι σε αυτές δεν είναι πρακτική η τεχνική εικονικοποίησης χαμηλότερων επιπέδων του λογισμικού συστήματος. Επίσης, η προσέγγισή μας διατηρεί τη φορητότητα με μελλοντικές εκδόσεις βιβλιοθηκών και οδηγών συσκευής, διότι κάνει χρήση ενός απομακρυσμένου συστήματος που ενημερώνεται σταθερά ως προς τις host βιβλιοθήκες παρέχοντας συνεχώς νέες εκδόσεις. Παρέχουμε το V4VSockets ως λογισμικό ανοιχτού κώδικα στο https://github.com/HPSI/V4VSockets.

Σε ανοιχτές στοίβες λογισμικού επιταχυντών, όπως αυτή που παρέχεται

με το Intel Xeon Phi, προτείναμε τη χρήση τεχνικών παραεικονικοποίησης στοχεύοντας τα χαμηλότερα επίπεδα μεταφοράς της στοίβας. Σε αυτό το πλαίσιο, περιγράψαμε τους συμβιβασμούς στις σχεδιαστικές επιλογές του συστήματός μας, που ονομάζεται vPHI. Σε σύγκριση με μεθόδους εικονικοποίησης υψηλότερων επιπέδων, η προσέγγισή μας παρέχει συμβατότητα με περισσότερα επίπεδα της στοίβας ενώ την ίδια στιγμή παραμένει διαφανής σε διάφορες τροποποιήσεις της αρχιτεκτονικής της συσκευής, εφόσον η τελική πλατφόρμα βασίζεται στο ίδιο επίπεδο μεταφοράς. Σε αυτό το πλαίσιο, υποστηρίζουμε ότι για τις μελλοντικές συσκευές επιταχυντών μπορεί να καταβληθεί προσπάθεια ώστε να υποστηρίζεται αυτό το μοντέλο ενός γενικού και κοινού επιπέδου μεταφοράς. Με αυτό τον τρόπο, η μέθοδός μας θα μπορεί να εφαρμοστεί σε αυτούς τους επιταχυντές και να προσφέρει διαφανή πρόσβαση σε εικονικοποιημένες οντότητες δίνοντας τη δυνατότητα στα νεφοϋπολογιστικά περιβάλλοντα να υιοθετήσουν πιο εύκολα την εξειδικευμένη φύση αυτών των συσκευών. Επιπλέον, αξιολογήσαμε εκτενώς την προσέγγισή μας σε έναν εικονικοποιημένο server και δείξαμε ότι πολλαπλές εικονικές μηχανές μπορούν να διαμοιραστούν μία συσκευή Xeon Phi με αποδεκτό κόστος. Ειδικά σε νεφοϋπολογιστικά περιβάλλοντα στα οποία πολλαπλές συνυπάρχουσες εικονικές μηχανές κάνουν χρήση ενός διαμοιραζόμενου επιταχυντή για συγκεκριμένες χρονικές περιόδους με προφορτωμένα σύνολα δεδομένων (datasets), το vPHI μπορεί να αποτελέσει μία επωφελή προσέγγιση προς την κατεύθυνση ενός αυξημένου συνολικού throughput του συστήματος οδηγώντας σε καλύτερη χρησιμοποίηση του επιταχυντή. Παρέχουμε το vPHI ως λογισμικό ανοιχτού κώδικα στο https://github.com/sgerag/vphi.

Με βάση την προηγούμενη εμπειρία, καταλήγουμε ότι για συσκευές επιτάχυνσεις που αγνοούν ότι υπόκεινται εικονικοποίηση, οι τεχνικές παραεικονικοποίησης μπορούν να οδηγήσουν σε σημαντικά οφέλη. Μία κρίσιμη παράμετρος για να αποφασιστεί το επίπεδο της στοίβας που θα εικονικοποιηθεί είναι αν υπάρχουν υποσυστήματα λογισμικού του περιβάλλοντος του επιταχυντή που παρέχονται με κλειστό τρόπο και σε ποιο επίπεδο της στοίβας υφίστανται. Στην περίπτωση ενός οικοσυστήματος λογισμικού σχεδιασμένο με ανοιχτό κώδικα, η εικονικοποίηση των χαμηλότερων επιπέδων της στοίβας μπορεί να καταστήσει δυνατή τη συμβατότητα με διαφορετικά runtimes και βιβλιοθήκες, χαρακτηριστικό το οποίο είναι μη πρακτικό σε ιδιόκτητο λογι-

σμικό.

Στο πλαίσιο των μελλοντικών υπολογιστικών συστημάτων, υποστηρίζουμε ότι οι νέας γενιάς πλατφόρμες επιτάχυνσης θα χρειαστεί να σχεδιαστούν λαμβάνοντας υπόψη τις ιδιότητες της εικονικοποίησης και το διαμοιρασμό των συσκευών επιτάχυνσης σε εικονικοποιημένα περιβάλλοντα. Έτσι, οι μηχανικοί συστημάτων θα είναι σε θέση να ενσωματώσουν τους επιταχυντές στο νεφοϋπολογιστικό οικοσύστημα με μικρότερη προσπάθεια.

Σε τέτοια περιβάλλοντα, η διαχείριση των πόρων λαμβάνοντας υπόψη τα τρέχοντα υπολογιστικά φορτία αποτελεί ένα βασικό παράγοντα βέλτιστης λειτουργίας. Σε αυτό το πλαίσιο, μελλοντική έρευνα μπορεί να στραφεί σε νέους τρόπους λήψης δυναμικών αποφάσεων σχετικά με τη χρήση επιτάχυνσης σε συγκεκριμένα υπολογιστικά φορτία με βάση πολιτικές προτεραιοτήτων, όπως η ενεργειακή απόδοση, η δικαιοσύνη, ή η Ποιότητα Υπηρεσίας (Quality of Service (QoS)).

1

Introduction

In today's interconnected world data are growing with remarkable rates and the projections [35] forecast a more intense environment for the near future (Figure 1.1). In this context, there is an ever growing need for processing power to keep up with the data creation speed. However, at the same time there is a slowdown in the multicore scaling trend, which is referred in the literature as the dark silicon era [16, 17]. Due to this power wall problem, there is a demand to provide increased performance in the data centers while keeping the power consumption as low as possible. In this ongoing effort, the advent of accelerator platforms, such as GPGPUs, Intel Xeon Phi, FPGAs etc, has transformed the computing landscape, since this kind of devices provide a much better performance per watt ratio than traditional computing elements.

Additionally, numerous studies [4, 5] have shown that, despite the large amount of processing power available in modern data centers, only a small portion of it is being utilized by the providers. One of the main reasons is that the interference between different workloads running on the same machine can in many ways severely hurt performance, despite the fact that the available processing resources are more than sufficient for the workloads to co-exist [45, 13]. Hence, providers prefer to underutilize some of their resources in favor of a more stable behavior with minimum performance variations for their clients. Researchers have proposed that moving to heterogeneous platforms can form a path to mitigate the consequences of this problem [29, 40].

Meanwhile, cloud computing has proven to be a beneficial approach for service providers in terms of cost reduction. As the popularity of cloud computing during the last decades started to grow, there have been efforts to gradually integrate different kinds of resources into the virtualization ecosystem.

Software and hardware researchers and engineers started to co-design virtualization components for CPU and memory and later for high performance I/O devices, as they appear more attractive from a virtualization perspective.

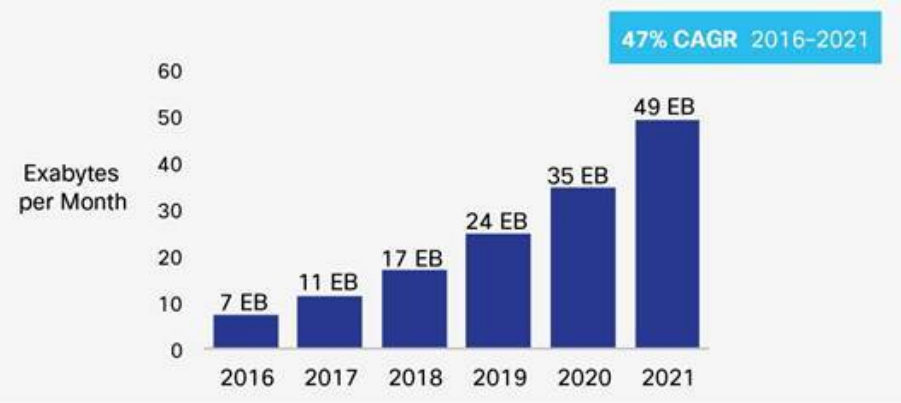


Figure 1.1: Cisco forecasts 49 exabytes per month of mobile data traffic by 2021 (Source: Cisco VNI Mobile, 2017)

Following this direction towards hyper-convergence combined with the aforementioned observations from the fields of architecture and computing infrastructure, a need for consolidation of accelerator platforms into the cloud ecosystem has emerged. In this regard, there are significant efforts to enable accelerator capabilities in virtual machines (VMs). In the previous years, plenty of remarkable works have been published mostly targeting GPUs [25, 26, 81, 71, 76, 41, 69, 62]. Additionally, in the enterprise section, companies have started to offer GPU-as-a-service in a cloud context [77, 50]. However, the state of practice approaches in industry most commonly utilize static and less flexible solutions, such as *direct device assignment* or *IOV* approaches.

In this dissertation, we explore the potential of integrating accelerator devices into the virtualization ecosystem aiming at the following properties: sharing of the accelerator between VMs, flexibility (e.g. ability to migrate or to apply scheduling policies), reduction of total cost of ownership in a cloud infrastructure, transparency, less intrusiveness, reduced virtualization overhead and more effective accelerator utilization. In this context, we propose two frameworks [51, 21, 22] targeting two popular acceleration devices, NVIDIA GPUs

and Intel Xeon Phi, to validate our approaches on different accelerator environments. We identify the key challenges of virtualizing accelerator resources and we outline the reasons that the traditional I/O virtualization methods are not adequate for this kind of specialized devices. Additionally, we examine the two most commonly used virtualization platforms (Xen and QEMU-KVM) by the research community, in order to investigate the potential of applying accelerator virtualization techniques to both type-1 and type-2 hypervisors.

In the first approach, *V4VSockets* [51], we design a socket-compliant, high-performance intra-node communication framework for co-located VMs in the Xen hypervisor and we utilize it with a popular remote GPU execution framework (rCUDA [62]) to enable low-overhead acceleration of applications running inside VMs. The data path in the design of our intra-node framework is realized through the hypervisor as the network medium, instead of the driver domain. Contrary to previous GPU virtualization approaches, the overall method provides a combination of all of the following features: performance, application transparency and portability across different versions of accelerator runtimes and libraries, since it is not dependent on a specific version of the rCUDA framework. Evaluation results show that our approach boosts the transfer throughput by a factor of up to 6.3 compared to the remote default path while it adds an overhead of 15% in terms of GPU execution compared to a direct device assignment configuration.

Our second approach, *vPHI* [21, 22], consists of a paravirtualization framework that enables VMs to offload tasks to Intel Xeon Phi devices in a transparent to the applications way. To our knowledge, vPHI is the first and currently the only approach that provides sharing capabilities of a Xeon Phi coprocessor to multiple virtual machines inside the same physical host. We implement vPHI using QEMU-KVM as the hypervisor and we virtualize the transport layer of the accelerator's software stack. In this context, the main principles of our work can be applied to future accelerator stacks in case the target device provides a transport layer API, as well as different virtualization platforms, provided that the respective hypervisor supports paravirtualization capabilities. Evaluation shows that vPHI can enable better accelerator utilization when it is used by multiple VMs, increasing up to 3.56x the total throughput versus a single host application, which represents the direct device assignment configuration.

The contribution of this thesis can be summarized as follows:

- We categorize and describe the traditional I/O virtualization methods and present the advantages and disadvantages of each one of them.
- We identify the key challenges of virtualizing accelerator devices and outline the reasons that the traditional I/O virtualization techniques are not adequate to meet workloads needs in an accelerator sharing environment.
- We describe the related work in this field and comment on the benefits and limitations of the major approaches.
- We present our approaches, V4VSockets and vPHI, exploring two dominant accelerator devices and covering the most popular virtualization platforms in the research community. We compare these frameworks with existing approaches and mention the key differences between them.
- We execute experiments to evaluate our frameworks and to analyze their behavior in various scenarios.
- Based on the experimental evaluation, we identify the impact of the design choices for our frameworks in order to comply with the desired features.
- We discuss what next generation accelerated systems should meet in order for system researchers and engineers to embrace them in future virtualization software stacks.

1.1 Outline

The rest of this dissertation is organized as follows: in Chapter 2 we provide the necessary background. We discuss the challenges of virtualizing accelerator devices in Chapter 3. Chapter 4 presents our approach on optimizing intra-node communication for remote GPU task offloading, while in Chapter 5 we describe our approach on enabling sharing of Xeon Phi accelerators in virtualized environments. We discuss related work in Chapter 6 and finally, in Chapter 7, we conclude and present directions of future work.

2

Background

In this chapter we provide background information regarding the technologies that we use in our approaches. Initially, we give an overview of hardware acceleration and how it introduced heterogeneity into traditional computing environments. More specifically, we focus our description on device accelerator platforms and the relevant software stacks that are commonly used. Next, we mention the two major virtualization platforms inside the research and academic communities and we describe the I/O virtualization techniques explaining which one is more appropriate in various use cases. Based on this discussion, in the following chapter, we outline the major challenges for embracing accelerators into virtualization environments and the reasons that the previously described methods cannot be used in the same way that traditional I/O virtualization is based on.

2.1 Hardware acceleration

In a typical computing system the basic component that is in charge of program execution is the CPU (Central Processing Unit). The CPU in modern commodity systems consists of a general purpose hardware component that supports a rich set of instructions and thus can be used for various types of computing applications. Contrary to this, with hardware acceleration, specific computational workloads, called *kernels*, can be executed by specialized devices, that have been designed to be extremely efficient for this kind of tasks. They accomplish this by usually providing massive data parallelism for specific operations compared to a single, or even multiple, general purpose CPUs. In

the following, we describe two well-known accelerator platforms that we use as target systems in this thesis.

2.1.1 Graphics Processing Units

Graphics Processing Units (*GPUs*) is a typical example of accelerators that perform efficiently for operations that are involved in a graphics application domain. Historically, with the growing demands inside the gaming industry, vendors started to produce extremely powerful devices in order to process large amounts of graphical data in a relatively small period of time [12, 47]. Inspired by this evolution, system designers created accelerators targeting also the HPC domain and some of the data-intensive scientific applications. Thus, more than a decade ago, vendors started to release General Purpose GPUs (or *GPGPUs*) and the relevant software stacks.

Typically, the common offload semantics that are exposed to the application programmer include (Figure 2.1) i) the data transfer from main memory to the accelerator's memory over the peripheral bus (e.g. PCIe), to which the device is attached, ii) the compute kernel to be executed by the accelerator device and iii) the transfer of results back to main memory to be accessed by the CPU. As it is expected, these two data transfers have a significant performance overhead and should be accounted when comparing application's performance executed on a CPU and on a GPU respectively.

CUDA

In this context, NVIDIA provides an API, called *CUDA* [53], for the application developers in order to utilize NVIDIA's GPUs for task offloading. We base our approach to CUDA, since it is one of the dominant platforms in the GPU offloading domain. However, the key idea can be applied to similar stacks that follow the offload execution model.

A CUDA program can be written using one of the two APIs that are exposed to the application (Figure 2.2), either i) the *Driver* API or ii) the *Runtime* API. The Driver API is a low-level interface that provides an additional level of control and exposes lower level details to the application. The Runtime API provides higher level semantics in order to hide some of the Driver API's de-

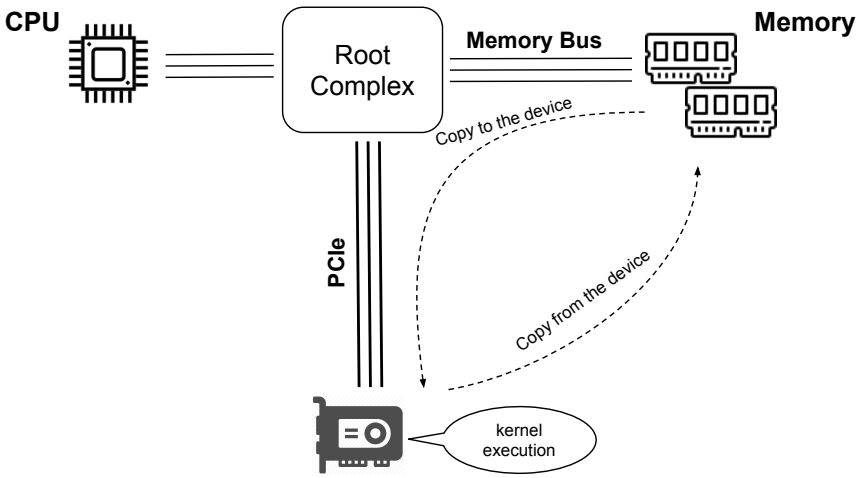


Figure 2.1: Typical offload path for accelerator devices

tails. In this way, with the Runtime API, NVIDIA aims to ease the development process and lessen the debugging effort.

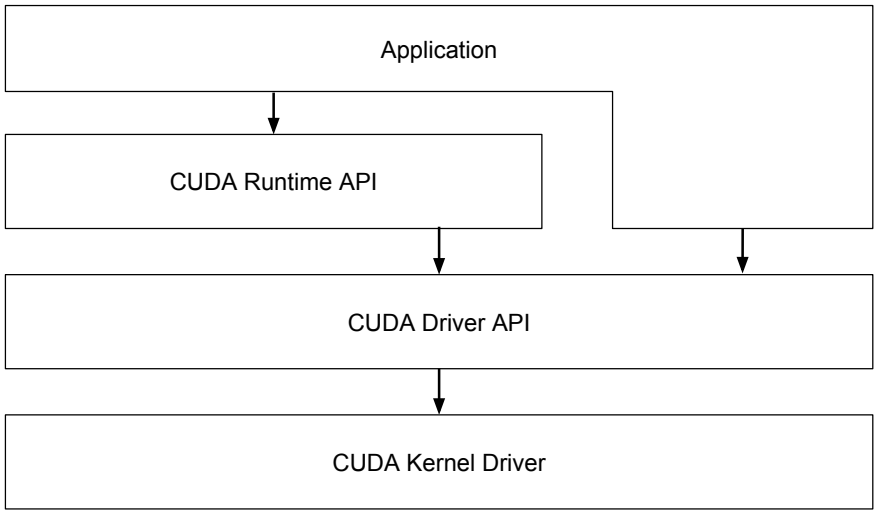


Figure 2.2: CUDA software stack

Among other components, CUDA software stack also includes a proprietary compiler, `nvcc` [54], that is used to produce the corresponding binaries. One of the operations that `nvcc` implements is the translation and mapping of

the symbols that are used by the CUDA Runtime API (e.g. for kernel execution) into the final binary. Being a closed-source compiler, `nvcc` applies techniques for this purpose that are difficult to unveil for virtualization frameworks that aim to provide CUDA Runtime compatibility.

rCUDA

rCUDA [62] is a middleware framework that enables computer nodes that are not equipped with GPUs to remotely access GPUs over the network (Figure 2.3). Hence, an application that is written in CUDA can be launched in a computing host environment without GPU devices and executed on a remote GPU-equipped node without any modifications neither to the source code nor to the executable binary. In this context, the former node act as the rCUDA client, while the GPU-equipped one acts as the rCUDA server.

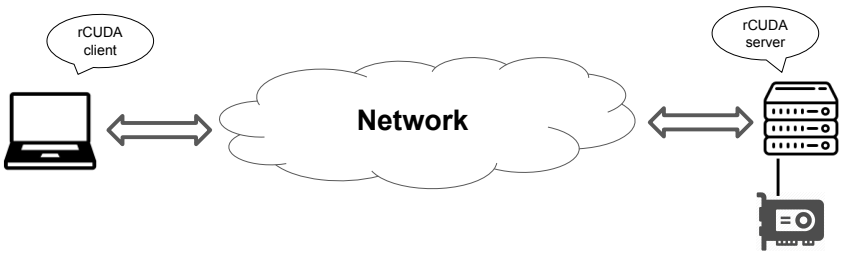


Figure 2.3: rCUDA use case

In our first approach, we utilize rCUDA in a virtualized environment in order to provide CUDA compatibility to existing precompiled applications. The authors of rCUDA [62] present a performance analysis according to which the traditional TCP/IP network stack implies a high overhead for applications, making the use of remote GPU not a viable approach for TCP/IP/Ethernet networks. Instead, they modularize their framework regarding the communication mechanism and propose the use of low latency and high throughput interconnects, such as Infiniband [36], in order to significantly reduce the network overhead. In our first approach, we use rCUDA in a virtualization environment and we optimize the communication path between virtual machines using V4VSockets as a low overhead intra-node communication mechanism,

in order VMs to be able to offload applications to GPUs in a viable manner. We explain this concept in greater detail in Chapter 4.

2.1.2 Intel Xeon Phi

In the beginning of this decade, Intel announced the release of its accelerator device, namely Xeon Phi [39]. Xeon Phi is a product family of processors that employ Intel's MIC (Many Integrated Core) architecture. It consists of a series of massively-parallel manycore processors and it can be used to accelerate a system as a coprocessor, or even as a host processor (e.g. Knights Landing).

From the application programmer's point of view, the major competitive advantage over NVIDIA's GPUs is that Intel provides x86 compatibility, hence applications that have been written for an x86 processor can be executed unmodified on a Xeon Phi coprocessor. In practice, however, developers have to carefully tune the involved applications or libraries, in order to fully exploit the parallelism that the coprocessor offers and achieve adequate performance [11, 65, 68, 15].

The second approach which we present in this thesis targets Intel Xeon Phi as the accelerator device. In order to understand the design aspects and the implementation details of our framework, we briefly describe in the following the basic concepts and internals of the Xeon Phi execution model and the corresponding software stack.

Xeon Phi model of execution

Intel defines a model of execution that supports three modes to meet different use case needs: *native*, *offload* and *symmetric*. In *native* mode the user supplies the executable directly on the Xeon Phi card. *Offload* mode permits the user to execute the application on the host CPU and offload some compute-intensive workloads to the coprocessor using the corresponding directives of a framework, e.g. *OpenMP*. Finally, in *symmetric* mode Xeon Phi can be viewed as an independent node and in that way a user can launch some processes of the same parallel application on the host side and some other processes on the accelerator, using, for example, *MPI*.

Xeon Phi system software stack

Xeon Phi devices are connected to the system through the PCIe bus. Intel provides *SCIF* (Symmetric Communication Interface), a low-level abstraction layer over PCIe, in order to enable higher level components to exploit DMA capabilities of Xeon Phi without messing directly with PCIe transactions. With vPHI, we essentially provide a virtualization scheme of SCIF to enable the same functionality for VMs. Figure 2.4 depicts the general system architecture of the software stack that is used in a typical non-virtualized Xeon Phi environment. By using SCIF, applications running both on the host and the device can communicate with the same API. In order to accomplish that, Xeon Phi itself boots a micro operating system (*uOS*), which consists of a modified Linux kernel, that contains a SCIF driver. Also, to expose the SCIF API, a SCIF library (`libscif`) is used. Furthermore, the Xeon Phi software stack includes an emulated network driver as part of the uOS, that uses SCIF, and enables users to utilize network tools (e.g. `ssh`) and connect to the Xeon Phi device. In this way, they can execute applications on the coprocessor using a shell. Similar to the Xeon Phi card, the respective components, libscif and the SCIF driver, have been implemented for the host side.

The library exposes the SCIF API to the application-side and communicates with the SCIF driver by performing system calls to a character device, namely `/dev/mic/scif`. The SCIF API provides both one-way and two-way communication semantics. There is a family of socket-like SCIF calls (`scif_bind()`, `scif_listen()`, `scif_accept()`, `scif_connect()`, `scif_send()`, `scif_recv()`) that supports traditional send-receive communication and another set of calls (`scif_register()`, `scif_unregister()`, `scif_(v)readfrom()`, `scif_(v)writeto()`, `scif_mmap()`, `scif_munmap()`), that exposes read/write RDMA semantics. RDMA is a common communication pattern used in high performance interconnects domain. In such contexts, developers frequently use a combination of RDMA and polling as an alternative to blocking methods, in order to notify the client of an I/O completion event. Similarly, SCIF provides `scif_poll()` to inform the caller that a subsequent operation to a specific endpoint can be performed without blocking, which for example could mean that some data have been received. Finally, there is another set of

SCIF calls (`scif_fence_*()`) that act as synchronization barriers. vPHI provides the same interface to the applications running on VMs by redirecting the traffic through the host.

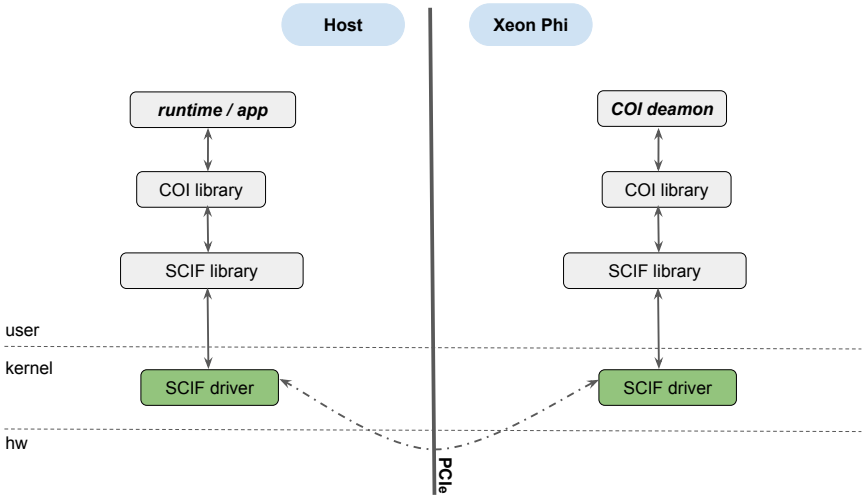


Figure 2.4: Communication between host and Xeon Phi

However, for certain use cases even SCIF exposes unnecessary details that many runtime systems or libraries do not need to be aware of. Hence, Intel provides a higher level library which uses SCIF as the transport layer and abstracts the low-level details (Figure 2.4). This library is called *COI* (Coprocessor Offload Infrastructure) and can be used to build runtime frameworks in order to query and control the state of Xeon Phi devices that are available in the system or to offload computational workloads to the coprocessor, by loading the appropriate libraries and executables, transferring the data over PCIe. The Xeon Phi device receives the respective requests from the host through a COI daemon that is launched after uOS has booted. By virtualizing the SCIF transport layer, vPHI remains compatible with higher level frameworks, such as COI.

2.2 Hardware virtualization

Hardware virtualization refers to the process of presenting a view of a hardware platform that internally exists and operates purely in software. The result-

ing virtualized entity is called *virtual machine* (VM), *domain* or *guest*, depending on the context, and the corresponding software entity that is in charge of implementing the virtualization process is called *hypervisor*, *virtual machine monitor* (VMM) or more broadly *host*. Thus a single hypervisor can host multiple virtual machines.

In this thesis, we build our frameworks for two of the most well known hypervisors inside the research community, namely *Xen* [3] and *KVM* [43]. These hypervisors are based on different software architecture and design principles. At a conceptual level, Xen is a *type-1* hypervisor, which means that it runs directly on the physical hardware, while KVM belongs to the category of *type-2* hypervisors, which means that it runs as part of a host operating system. Besides their popularity, we choose to work with different hypervisors in order to examine the challenges of integrating accelerator logic into diverse virtualization platforms. Later in this chapter, we briefly describe the architecture of each one of them as well as some basic operation principles. Before that, we provide the essential background regarding the different approaches for virtualizing hardware, focusing primarily on I/O virtualization. Based on this discussion, we mention the challenges of enhancing virtualization environments with accelerator semantics in the next chapter.

2.2.1 Evolution of virtualization

History of hardware virtualization includes a large set of applied techniques and approaches in order to retain transparency and maximize performance for applications. In this context, we focus on three areas of virtualization: CPU, memory and I/O and comment on what kind of practises eventually are established in each one of them, while also discussing commonalities and differences as well as tradeoffs that almost always arise in the process of designing computing systems.

Historically, the first approaches towards hardware virtualization targeted CPU, while trying to overcome non-virtualizable features [20] of certain ISA (e.g. x86). Such approaches include *full processor emulation*, *binary translation* and *hardware-assisted virtualization*. The first two, being software approaches, induce a large processing overhead due to large numbers of traps when exe-

cuting guest code (emulation) or constantly monitoring and translating specific blocks of instructions (binary translation). The advantage is that legacy virtualization-unaware CPUs can be used in a virtualization environment utilizing one of these techniques. On the other hand, with hardware-assisted virtualization, the CPU is by design aware of the existence of virtualized entities. In this way, this technique comes with great performance boost, but requires the existence of CPU virtualization extensions. Since many processor vendors (e.g. Intel, AMD) integrate these features in their chips by default nowadays, hardware-assisted virtualization has become by far the mainstream solution in cloud services outweighing software solutions.

In the roadmap of hardware virtualization, the next step for system designers was how to virtualize memory management for several virtual machines and at the same time reduce the cost of page table manipulation. Following the CPU paradigm, software techniques preceded the hardware solutions. *Shadow page tables* enable the VMM to monitor page table updates by implementing associated structures at the software level. Later, vendors introduced enhanced hardware techniques by adding awareness inside the MMU about the existence of separate page tables per virtual machine (e.g. *nested/extended page tables*). Similar to the CPU case, hardware approaches are considered the de-facto mechanisms on commodity infrastructures for Cloud.

2.2.2 I/O virtualization techniques

While compute intensive applications fitted well in the paradigm of CPU/MMU virtualization, with the advent of cloud computing, the need for virtualizing I/O resources at an acceptable overhead has emerged. In this context, several techniques have been proposed balancing between virtualization overhead on the one hand and code complexity and transparency on the other. In this section we briefly describe the existing approaches for virtualizing an I/O device and present the advantages and disadvantages of each method. Hence, we proceed to the following categorization:

- **Direct device assignment** (Figure 2.5): the device is directly assigned exclusively to one virtual machine (VM), preventing other VMs to have access to that device.

- **IOV-enabled devices** (Figure 2.6): the physical device per se exposes a maximum number of virtual devices, each one of them can be directly attached to a VM.
- **I/O emulation** (Figure 2.7): the guest-side driver remains intact, while there is a backend implementation in the host.
- **Paravirtualization** (Figure 2.8): a virtualization-aware driver is inserted in the guest and the associated backend driver runs at the host side.

Direct device assignment (or *PCIe passthrough*) offers near-native performance at the cost of limited scalability, since only one VM can have exclusive access to the real device, hurting the sharing use case. In order to mitigate this scalability issue, SR-IOV [70] was proposed as an alternative targeting mostly high-speed interconnection networks. SR-IOV is a hardware extension to the PCIe specification and it consists of an **IOV-enabled device** solution. In a few words, it defines a number of Virtual Functions (VFs) for each physical device and assigns each VF to an individual virtual machine. In this way, it can achieve near-native performance for each of the VMs, the maximum number of which still remains static regarding the virtualized device. The major downside of IOV-enabled approaches is that special I/O devices are needed, which increase the total ownership cost in a cloud infrastructure. Additionally, both of the methods described present flexibility problems and this can be a major issue for cloud environments, which inherently feature a dynamic and elastic

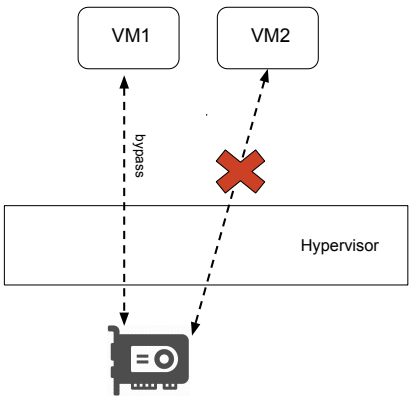


Figure 2.5: Direct device assignment

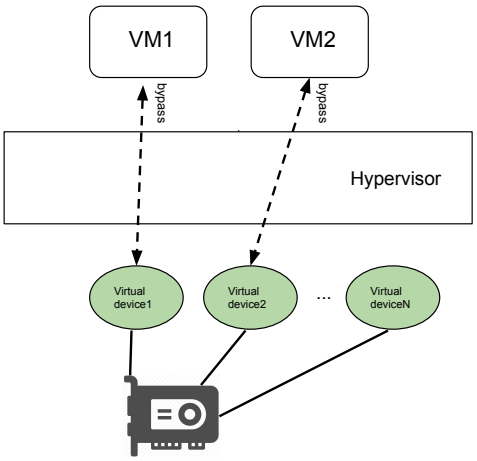


Figure 2.6: IOV-enabled device

management of resources. For example, both methods have limited support for migration, since in this case it is required that the current hardware configurations and features to be available on the destination host as well.

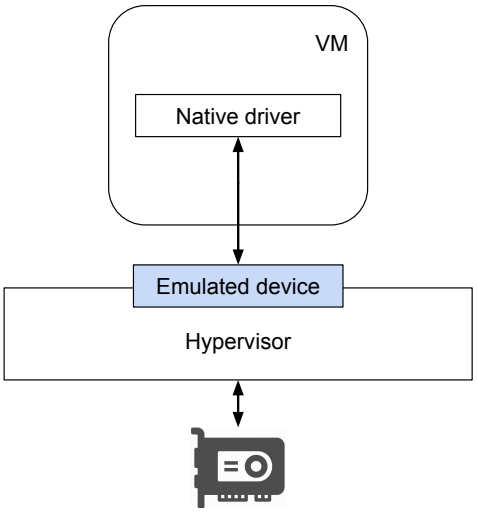


Figure 2.7: I/O emulation

From the two remaining non-static approaches the less intrusive one is the **I/O emulation**, according to which the guest device driver remains intact and

traps the privileged instructions to the hypervisor which further proceeds and performs the completion of the I/O operation. Since the VM is unaware that the access to the device is being virtualized, the emulation method comes with a relatively high performance degradation. To reduce this overhead without any dependency on modern hardware features, **paravirtualization** approaches have been proposed. According to this method, a virtualization-aware device driver is being loaded on the guest side, which batches the operations of an I/O request and traps to the hypervisor which satisfies the request. Essentially, it tries to minimize the exits to the hypervisor which constitute the major factor of overhead in this model. Depending on the specific approach, a paravirtualization solution may even perform at a near-native rate, offering a major advantage in terms of performance. The downside stems from the fact that parts of the guest operating system (usually drivers) have to be modified in order to take full advantage of guest virtualization awareness.

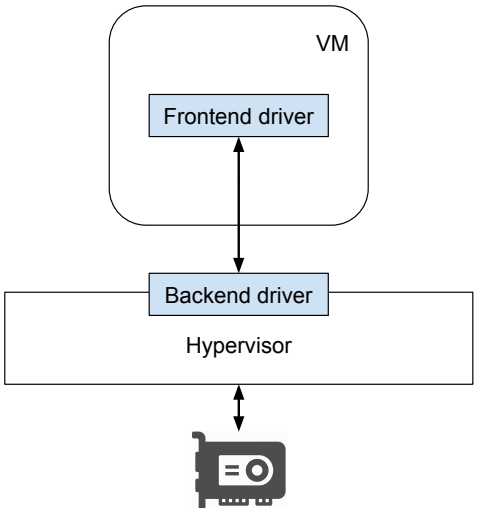


Figure 2.8: Paravirtualization

A summarized comparison between the aforementioned I/O virtualization techniques is depicted in Table 2.1.

Table 2.1: Comparison between typical I/O virtualization techniques

	Advantages	**Disadvantages**
Direct device assignment	- Near-native performance	- No device sharing - Flexibility and scalability issues - Limited support for migration
IOV-enabled device	- Near-native performance - Sharing feature	- Special hardware needed - Flexibility and scalability issues - Limited support for migration
I/O emulation	- Unmodified guest OS - Sharing feature	- Low performance
Paravirtualization	- Significantly lower virtualization overhead - Sharing feature	- Modifications to the guest OS

In the following, we briefly describe the two virtualization platforms that we use in this thesis from a software architectural point of view.

2.2.3 Xen virtualization platform

The general architecture of the Xen [3] hypervisor is depicted in Figure 2.9. Xen is based on the *paravirtualization* (PV) concept especially for the I/O path. According to this method, Xen utilizes a split-driver model, according to which a *frontend* driver runs inside the guest VM exposing a per-device class API to guest user space or kernel-space and the corresponding *backend* driver that is executed by privileged guests called *driver domains*. In this way, data access is handled by driver domains that essentially enable VMs to interact with the hardware. In most setups, there is a single driver domain that is called *dom0*. The unprivileged domains, which are called *domUs*, communicate with dom0 to access the hardwrare.

In Xen, memory is virtualized in order to provide contiguous regions to OSs running on guest domains. This is achieved by adding a per-domain memory abstraction called *pseudo-physical memory*. Thus, in Xen, *machine memory* refers to the physical memory of the entire system, whereas *pseudo-physical memory* refers to the physical memory that the OS in any guest domain is aware of.

To efficiently share pages across guest domains, Xen exports a *grant* mechanism. Xen's grants are stored in *grant tables* and provide a generic mechanism

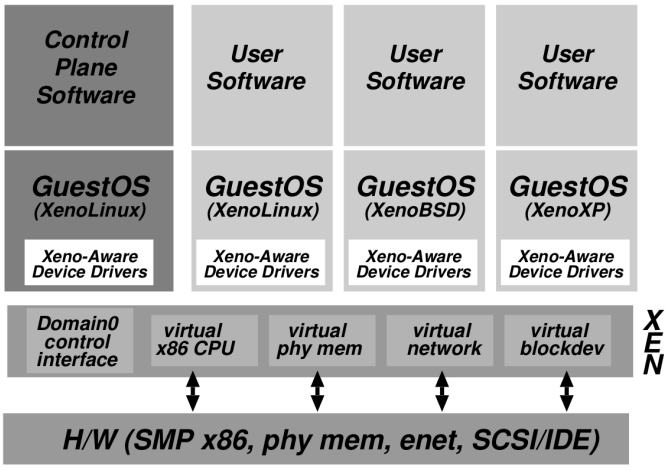


Figure 2.9: Xen software architecture (Source [3])

for memory sharing between domains. Network device drivers are based on this mechanism in order to exchange control information and data. Two guests setup an *event channel* between them and exchange events that trigger the ex-

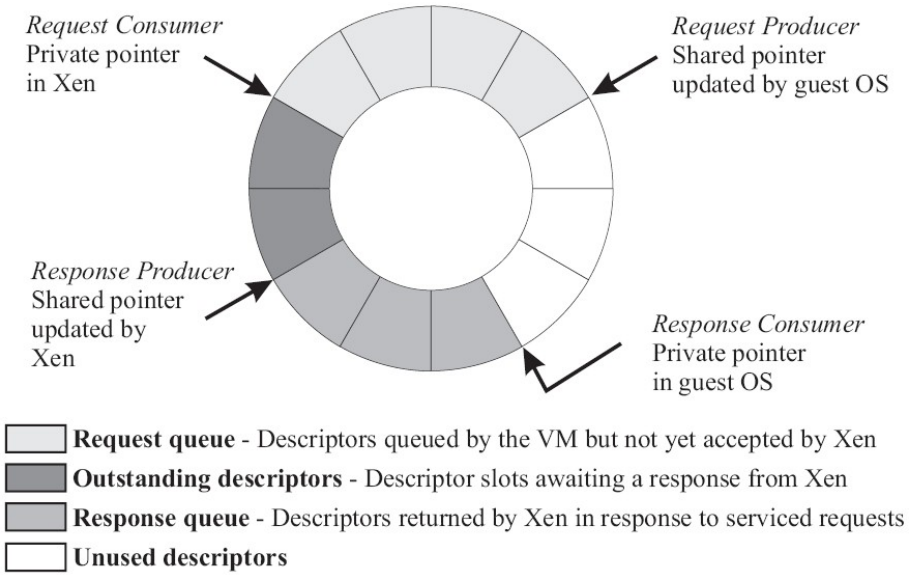


Figure 2.10: Xen rings structure (Source [3])

ecution of the corresponding handlers. *I/O rings* (Figure 2.10) are ring buffers, a standard lock-less data structure for producer-consumer communication. Through I/O rings, Xen provides a simple message-passing abstraction built on top of the grant and event channel mechanisms. Since with V4VSockets we optimize the communication between VMs running on a same node, we briefly describe the default network path for intra-node communication in the next section.

Xen Paravirtualization (PV) Network I/O

The common method of VM communication in Xen is through the Paravirtualization (PV) network architecture. Guest VMs host the *netfront* driver, which exports a generic Ethernet API to kernel-space. The driver domain hosts a hardware specific driver and the *netback* driver, which communicates with the netfront using the event channel mechanism and injects frames to a software bridge.

Data flow in and out of the VM using the grant mechanism, while notifications are implemented using event channels, the virtual IRQ mechanism that Xen provides. Less-critical operations are carried out by Xenstore (interface numbering, feature exchange etc.).

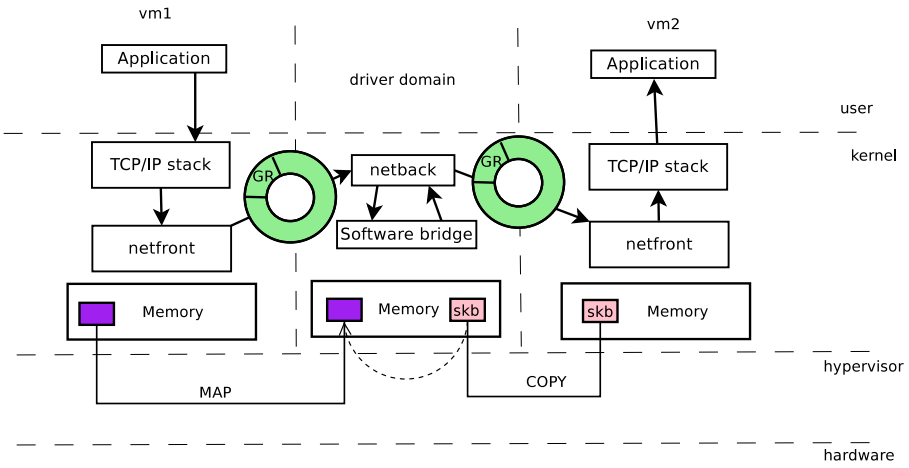


Figure 2.11: Generic intra-node communication in Xen

To communicate with each other, VMs that co-exist in the same VM host

have to cross through the software bridge in a driver domain. Figure 2.11 presents the data path of two VMs exchanging data. Data movement is realized using shared pages that are set up using the grant mechanism. Each transmission request contains a grant reference and an offset within the granted page. This allows transmit and receive buffers to be reused, preventing the TLB (Translation Lookaside Buffer) from needing frequent updates. To receive packets, the guest domain inserts a receive request into the ring, indicating where to store a packet, and the driver domain places the contents there.

2.2.4 KVM virtualization platform

As we mentioned previously, KVM [43] hypervisor is executed on a Linux host operating system. This operating system can support both host processes and full virtual machines. In this context, KVM consists of a loadable kernel module, which creates a virtual device (namely `/dev/kvm`) and exports a number of `ioctl` commands to this device. For example, by issuing a `KVM_CREATE_VM` ioctl system call to the KVM device, a new guest is created, which from the host perspective is nothing more than just a user process with its own virtual address space. From the guest perspective this address space is translated to what is called guest physical address space and is being treated as ordinary physical memory. KVM utilizes the hardware virtualization capabilities (e.g. Intel VT, AMD-V extensions) in order to virtualize CPU and MMU operations. KVM is mostly used together with QEMU [6], a user space emulator which runs on the host and issues system calls to `/dev/kvm`.

In vPHI we use QEMU-KVM as the hypervisor with *virtio* as the communication mechanism between the host and the virtual machines. Virtio [67] is a standardized interface used for development of virtualized devices following the paravirtualization approach. As we mentioned earlier, paravirtualization enables low overhead I/O virtualization by establishing an efficient communication channel between the host and the VM (guest). Using this method, the virtual hardware exposes a software interface to the respective driver in the guest, which is aware that is being virtualized and thus reduces any unnecessary I/O traffic that a virtualization-unaware driver would produce.

Similarly to the Xen I/O method, virtio follows the split-driver model ap-

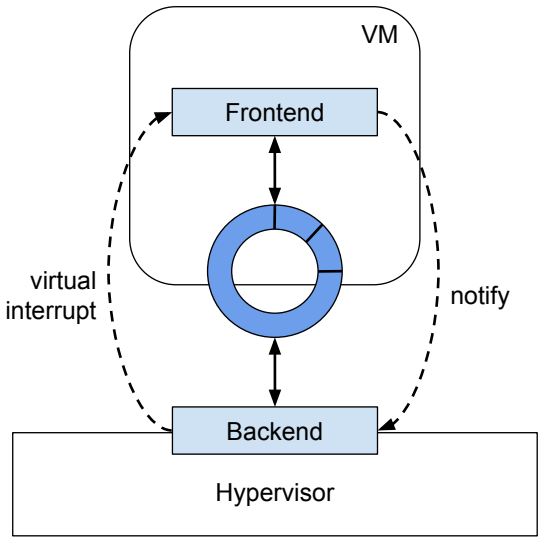


Figure 2.12: Virtio transport mechanism

proach, according to which a paravirtualized frontend driver is inserted into the guest, communicating with the respective backend on the host side. To realize the communication, a shared ring structure is registered between the guest and the host (Figure 2.12). The frontend driver submits I/O requests by posting the respective buffers in the shared ring and notifies the backend. Afterwards, the backend processes the event, emulates the I/O that is requested and produces a corresponding response. It posts the response in the ring and notifies the guest side via a virtual interrupt. The guest can either busy-wait on the shared ring, consuming CPU cycles, or block until the request is fulfilled. In the latter case, the interrupt produced by the host wakes up the guest, which pushes up the response to the upper layers. We point out here that no copies are involved during the communication between the guest and the host, since a shared memory area (ring) is used and the host can access the guest's physical address space and map the corresponding buffers to its own address space.

3

Accelerator virtualization challenges

Accelerators belong to a special part of I/O devices which offer computational power besides traditional I/O characteristics. Therefore, in order to virtualize an accelerator, one has to resolve all the aforementioned obstacles of I/O virtualization plus many more challenges due to their special nature. In this chapter, we identify the major challenges of virtualizing accelerator devices and we proceed to the following categorization.

Device complexity

A traditional I/O device usually includes a set of I/O registers and memory. Some more complicated devices (such as 10Gbps network adapters) are equipped with a microprocessor as well, in order to offload specific operations, like TCP offload for TSO (TCP Segmentation Offload) engines. On the other hand, modern accelerators' architecture is highly complex compared even with high end, low latency and high bandwidth network devices. They are equipped with many cores, can handle hundreds of threads and they have their own complicated memory management unit. They sometimes even boot an independent operating system, as Intel Xeon Phi does, which can be viewed as a distributed node in the same system. Decoupling the logical device from its physical implementation in this context presents major challenges.

Scheduling semantics

Given the background, accelerator devices obey to completely different scheduling semantics compared to the traditional CPU scheduling process. For example, there is no time-sharing features at least on legacy GPUs and

GPU tasks cannot be preempted at instruction level. Therefore, if there is contention between a short task and a long task and happens the long one to acquire the GPU, then the short job will have to wait for the completion of the longer one, increasing the total waiting time of the tasks. This can hinder the process of multiplexing I/O requests and consequently device sharing between multiple clients. Recent advances in GPU architectures enable devices to support preemption in a time-sharing manner offering a more fine grained control [57]. Nevertheless, there is no publicly available information that shows the availability of software level preemption control [84]. Additionally, there is no space-sharing support on legacy GPUs, which results in serialized execution on the device, even if the corresponding jobs could have been executed simultaneously on the GPU. In order to mitigate the implications of this inability, NVIDIA has released Multi-Process Service (MPS) [56] for its newer GPUs, which provides the feature of space-sharing to multiple tasks. MPS is essentially a proxy/daemon which submits requests to the device on behalf of the original GPU tasks. In this way, requests are submitted in the same GPU context, which poses security implications in a GPU-shared virtualization environment. However, with the release of the Volta architecture [58], NVIDIA updated its MPS subsystem allowing multiple GPU tasks to be directly executed on the accelerator and enabling isolation between them through separate GPU address space assignment. Exploring VM scheduling policies and evaluating algorithms in accelerator virtualization systems is beyond the scope of this thesis. However, as we provide frameworks for accelerator sharing using flexible techniques in such environments, we enable the mechanisms for future research considering scheduling between virtual machines. Many of the works that we mention in Chapter 6 regarding GPU virtualization also consider scheduling issues in their frameworks. Regarding Xeon Phi characteristics in the context of accelerator scheduling, it supports both the features of time and space-sharing and exposes them through its micro operating system which is executed on a dedicated core.

Furthermore, as many works in the GPU research community mention [72, 60, 44, 10], context switch in SIMT architectures is orders of magnitude slower. In this scheduling overhead we have to also consider the cost of reloading data of a previously preempted task in accelerator's memory through the PCIe bus.

Since it is commonly accepted that copying data back and forth to the device through PCIe is the major bottleneck when offloading operations to the device, this overhead is relatively high. Finally, considering the fact that, as in every I/O virtualization method, another layer of scheduling is added to the process, it becomes profound that offering high quality of service is extremely difficult.

Closed-source system software

An additional obstacle specifically for the GPU virtualization case arises from the fact that the major GPU vendors provide drivers in a closed manner. This makes difficult for engineers and researchers to fully understand the underlying system's architecture in order to provide a decoupled virtualized device. To this end, remarkable efforts have been made to better understand and explore the internals of closed GPU architectures and device drivers and provide open-source alternatives to the research community by following the hard path of reverse engineering [86, 61, 49]. These works have built the ground for follow-up research on GPU virtualization. However, reverse engineering techniques sacrifice at some level the robustness, the performance and even a subset of features with the purpose of opening the software stack of the device. On the other hand, a different approach for virtualizing acceleration platforms, is to provide compatibility with higher level APIs by redirecting them to another backend using various techniques. The major downside of this approach is that it is dependent on specific API versions and in this way, the resulting frameworks do not remain portable to future versions [31].

In our approach, we propose an indirect higher level method, according to which the use of a remote GPU framework is optimized through a highly efficient intra-node communication mechanism. In this way, our approach can be compatible with future library versions, since this method is based on a remote framework that is constantly up-to-date against the host libraries by continuously providing new releases. Additionally, as we show in Chapter 4, our method results in signifigant performance improvement compared to the default network communication path and overall it can be a sustainable solution for multiple virtual machines in order to be able to access efficiently a single

GPU.

Virtualization-unawareness

On the other side, even in the open-source accelerator software case, reducing the virtualization overhead still remains a challenging task, even when VMs do not issue simultaneous requests and there is no need for scheduling at these times. One of the main reasons is that the native system software is inherently unaware that the resources it provides are being virtualized. For example, in the case of Xeon Phi virtualization, the system software that is executed on the accelerator and manages device memory needs to have knowledge of virtualization at some degree or at least to provide interfaces for managing memory at a page-granularity level, in order to reduce the virtualization overhead, since more levels of address spaces are involved that are generally scattered accross memory. We will discuss this performance issue in greater detail in Chapter 5.

Level of transparency

Another challenge for the complex I/O architecture of accelerators is to define appropriate semantics for virtual devices and interfaces and to decide the level of transparency for the applications. Considering this, research community has provided approaches for virtualizing accelerator following some tradeoffs. Among these, there are variations of full virtualization in software, where a virtual device with exactly the same interface as the physical one is exposed to user space. As we mentioned previously, with this approach the VM is unaware that the corresponding accelerator is a virtualized one. This has the advantage of transparency, because there is no need for changes in any level of the software stack, but comes at the cost of increased overhead, since it involves the emulation of a fairly complex device. A full virtualization approach usually leaves intact the virtual device driver and emulates the corresponding operations by trapping at the hypervisor each time the guest driver proceeds to an I/O request or access. This increased amount of traps contributes to the total overhead, since the cost of each trap has proven to be expensive. On the other end of proposed solutions, a virtual device with a completely new interface is provided, designed to be simple and efficient, sacrificing compatibility.

Somewhere in the middle paravirtualization approaches lie, according to which some kind of a frontend driver is inserted to the guest with knowledge that the accelerator is being virtualized. With this knowledge, the frontend part redirects the control and data flow and communicates efficiently with its sibling backend component, which lives in a privilleged layer (VM or hypervisor). As we discuss later in more detail, vPHI belongs to a paravirtualization approach combined with the concept of targeting the low level transport layer, in order to be compatible with future high level runtimes and libraries and be portable across different software versions. Finally, to overcome both the transparency and the performance issues, vendors have added hardware support to physical devices following the direct device assignment model. In this way, they enable accelerators to be directly attached in a virtual machine offering near-native performance but eliminating device sharing and scalability. In this context, NVIDIA has tried to alleviate these scalability issues and adapt to the IOV-enabled approach. Thus, in some of its cards it provides to the user the ability to statically define a maximum number of virtual GPUs in hardware and directly attach each one of them to a possibly different virtual machine [78]. As we described in the previous chapter, the major drawbacks of this approach is the flexibility and the scalability in dynamic cloud environments, where migration and device hot plugging/unplugging take place at frequent periods of time.

4

Optimizing intra-node communication for remote GPU task offloading

In this chapter, we present our approach for enabling GPU acceleration for virtual machines running on a physical node. We target NVIDIA GPUs and we propose a portable method across different versions of NVIDIA's CUDA libraries. We achieve this property by utilizing rCUDA, a remote GPU acceleration framework. As we mentioned in Chapter 2, the major drawback of rCUDA in terms of execution overhead is that it performs poorly when it is applied in configurations with commodity network stacks. Instead, the authors of rCUDA propose the use of low latency and high throughput interconnects in order to enable acceleration for nodes that are not equipped with GPUs in a viable way.

In this context, we use rCUDA in a single environment that hosts multiple virtual machines and we optimize the intra-node communication between these VMs. More specifically, following the rCUDA scheme, there are two types of nodes in an HPC cluster: the GPU nodes, which act as rCUDA servers, and the non-GPU nodes, which act as rCUDA clients and forward CUDA requests to the respective servers through the network. Figure 4.1 depicts our the configuration that we use. We setup a physical node equipped with a GPU device and directly assign this device to a single VM, which acts as the rCUDA server. Subsequently, we launch a VM that acts as the rCUDA client and can access the GPU through the rCUDA framework. However, we significantly alleviate the network overhead by implementing and utilizing V4VSockets, a higly efficient and transparent intra-node communication framework between VMs running on the same physical node. Despite the fact that in this thesis we focus on a GPU use case, V4VSockets can be used with other scenarios of socket applications between co-existing VMs to significantly reduce the communication

overhead.

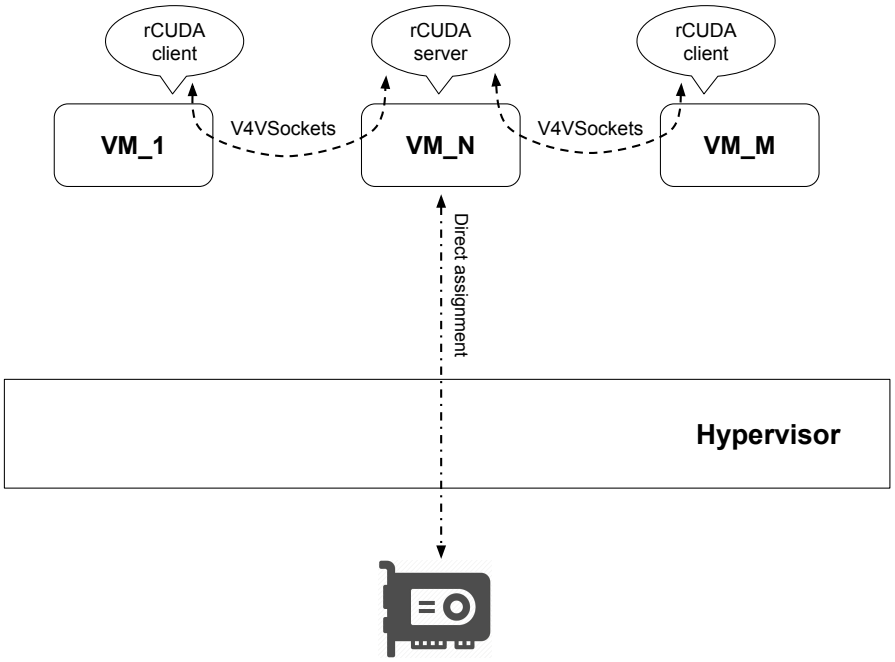


Figure 4.1: rCUDA over V4VSockets

In the following, we describe the design of V4VSockets using Xen as the virtualization platform. Afterwards, we delve into the implementaion details and in the final section, we present the evaluation results of our approach.

4.1 Design of V4VSockets

V4VSockets is a highly efficient intra-node communication framework in the Xen platform. Our approach builds on V4V, part of the XenClient project [80]. V4V is an abstract mechanism provided by the Xen hypervisor that supports basic communication primitives between co-located VMs.

V4VSockets is essentially a generic socket layer for the V4V transport mechanism that enables applications running on VMs' user space to communicate with other VMs co-existing on the same node. V4VSockets consists of a device driver to expose the socket API to user space and the V4V transport

mechanism provided as an extension to the Xen hypervisor. An analogy to the TCP/IP protocol suite is shown in Figure 4.2.

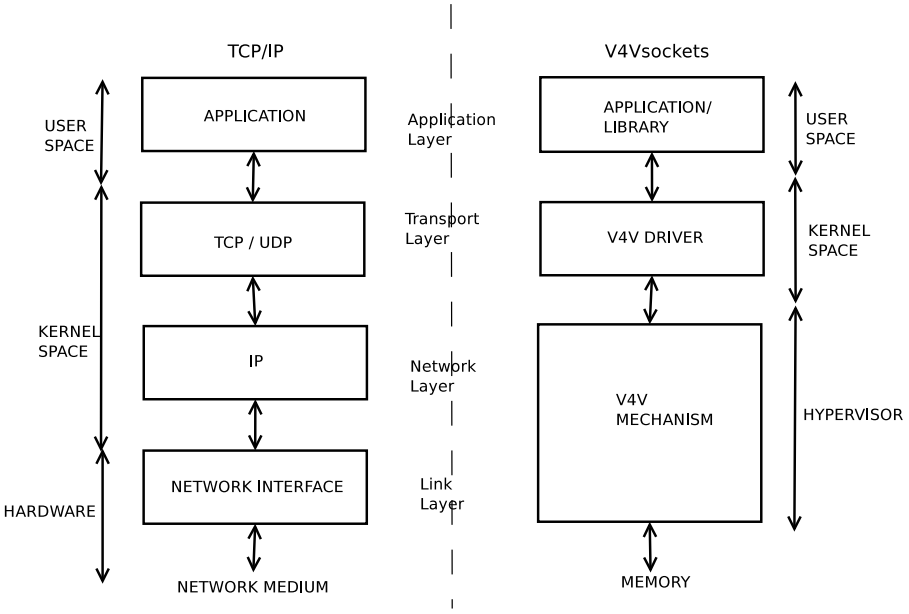


Figure 4.2: TCP/IP and V4VSockets

V4VSockets is built as a full-stack protocol framework that supports peer-to-peer communication between co-located VMs. Contrary to the common approach of decoupling communication to a privileged VM, leaving only security issues to be handled by the hypervisor, we choose to bypass the intermediate layer and use the hypervisor as the control *and* the data plane. Data flow between two VMs without the intervention of a third VM, providing better isolation and scalability. Additionally, we avoid the page sharing technique between the source VM and the destination VM which can lead to various security issues (e.g. data leakage between the two VMs) and instead we use memory copies in the corresponding phases of the data path. To provide an architectural overview, we briefly describe how the operations are realized in each layer.

Application layer: One of the most important aspects of our design is the API compatibility with generic concepts, namely the socket interface. Specifically, we aspire to provide a low-overhead socket communication framework

to applications running in co-located VMs without the need to refactor, re-implement or recompile them. Thus, in V4VSockets, the application layer refers to the common socket-layer calls (`socket()`, `bind()`, `connect()` etc.) which forward the relevant actions and arguments to the transport layer.

The *Transport layer* in our approach resides in the VM kernel. Essentially, it implements the socket calls and the communication primitives of the communication protocol over the network medium (Xen in our case). Specifically, the transport layer handles the virtual connection semantics between peer VMs that need to communicate, is in charge of fragmenting and sending upper-layer packets by issuing hypercalls to the hypervisor (network layer), and provides a notification mechanism to the VM's user space for receiving packets, as well as error control.

The *Network/Link layer* resides in the hypervisor, providing encapsulation of upper-layer messages to packets that will be transmitted to their destination, according to V4V semantics, and packet delivery. This layer is in charge of transmitting the packet to its destination, which in our case consists of a memory copy. Thus, in the case of a packet send, the hypervisor places data into the relevant memory space of the receiver and notifies the transport layer about an incoming packet.

In the following section we describe the V4VSockets framework in detail and present essential parts of the implementation, focusing mainly on the data exchange mechanism.

4.2 Implementation details

The core part of our framework is the transport layer, implemented as a VM kernel module, where all calls from user space are translated into `V4V` hypercalls and issued to the hypervisor (the network/link layer). `V4V` provides basic support for communication through the following hypercalls:

- `register`/`unregister`: The `register` call is used when a new socket is created and provides the necessary memory space to transfer data.
- `send`: This call is used when a send call is issued, providing the relevant structures to the hypervisor that, in turn, realizes the transfer.

- `notify`: When data is placed correctly, or when there's a notification that needs attention, the VM kernel issues this call and the hypervisor handles all necessary steps to complete the operation (e.g. receive calls).

The interesting part of V4VSockets is mainly focused on the data path. We base our framework on the `v4v_ring` structure, containing a static, pre-allocated ring buffer that essentially simulates the network medium. This buffer follows the generic producer-consumer concept, with two pointers `rx` and `tx` that are altered by the VM and hypervisor respectively.

This buffer is allocated in the VM kernel and registered to the hypervisor with the `bind()` system call. Essentially, this translates into a `register` hypercall and, thus, the machine frames that comprise the ring buffer are stored and mapped in Xen, forming a shared memory region between the VM kernel and the hypervisor.

The `accept()` system call initializes a receive operation: the application listens to a specific port for incoming packets.Once data have been written to the ring, the hypervisor updates the `tx` pointer. Following a `recvmsg()` system call, the VM kernel copies data from the ring space to a local staging buffer. Additionally, it updates the `rx` pointer (to free up space in the ring) and copies the received packet to user space.

The `sendmsg()` system call initiates a send operation: the VM kernel creates an `iovector` from the user space arguments, packs the data into a `V4V` message and issues the `send` hypercall. The hypervisor copies the data into the ring of the receiver VM, updating the `tx` pointer.

An example of a data exchange between two peer VMs is shown in Figure 4.4. At this point, we repeat the Figure 4.3 from Chapter 2, in order to visually compare the two data paths.

In V4VSockets, user space applications issue generic socket calls using a custom address family constant instead of `AF_INET`. To keep compatibility for applications that have this constant hard-coded, we wrap the initial `socket()` system call around a library that re-issues the call with our custom address family. The rest of the calls (e.g. `bind()`, `listen()`, `accept()` etc.) use the socket descriptor provided by the initial call so the API remains intact.

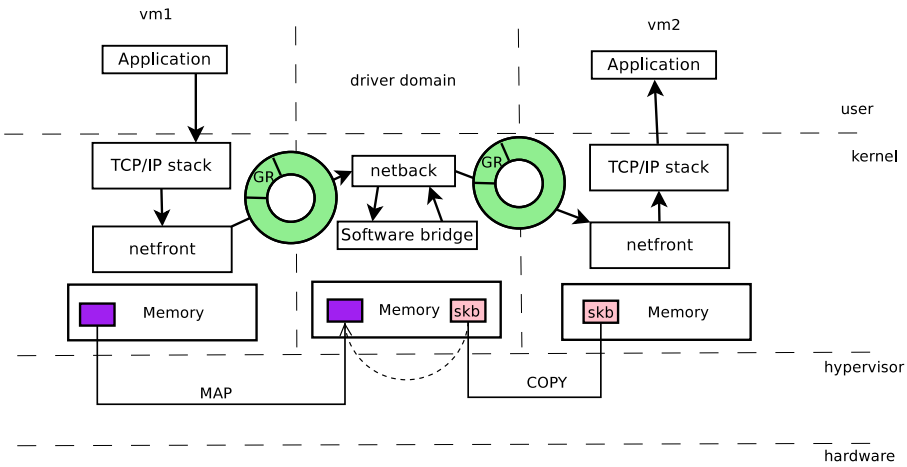


Figure 4.3: Generic intra-node communication in Xen

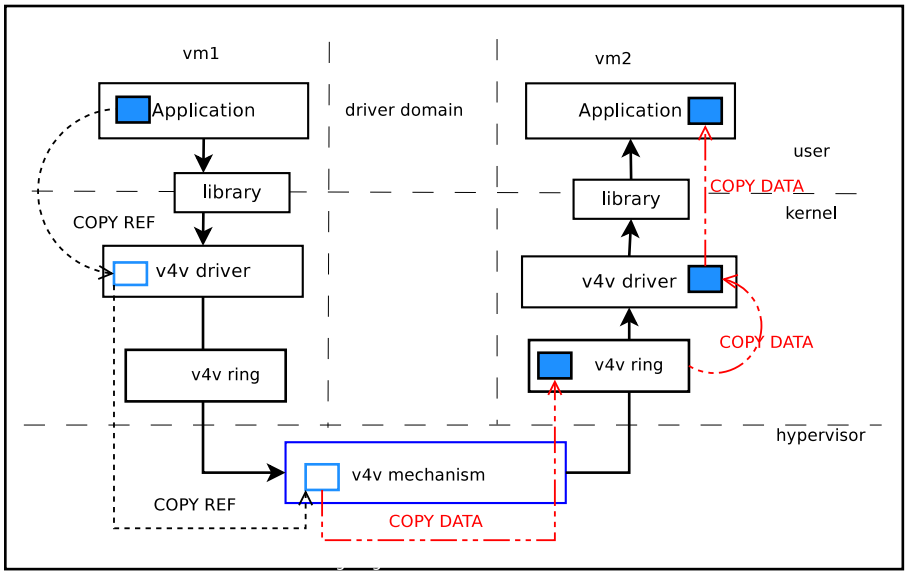


Figure 4.4: V4VSockets overview

4.3 Performance evaluation

In this section we first describe the experiments we performed to analyze the behavior of V4VSockets using network benchmarks and subsequently we present the evaluation of our framework in scenarios with GPU-enabled VMs.

We setup a host machine with 2x Intel Xeon X5650 (Chipset 5520) and

48GB RAM (@1333MHz) and perform two basic experiments using microbenchmarks, in order to illustrate the merits and shortcomings of our approach without the noise of application-specific communication patterns. We also perform a third real-life experiment using a CUDA application from the GPGPU domain.

4.3.1 Microbenchmark evaluation

We setup the physical machine as a VM host and spawn up to 16 single core VMs (VM_1,VM_2, ...,VM_{16}). We use NetPIPE [52] as a microbenchmark, in order to compare V4VSockets to the default TCP/IP over netfront/netback case. We deploy the microbenchmark between VMs (16 separate instances, VM_1 to VM_2, VM_3 to VM_4 and so on).

Figure 4.5 and Figure 4.6 plot the respective measurements when two VMs (VM_1 to VM_2) exchange messages. Figure 4.5 shows that the latency achieved by V4VSockets for a 2 Bytes message is improved by 81% compared to the generic case. Specifically, the latency of the split driver model is 86 us, while V4VSockets completes the same task at 16 us. This is mainly due to the processing overhead of the TCP/IP stack, as well as the inefficient data path through the driver domain (Section 2.2.3), which is bypassed in our optimized transport mechanism.

In terms of throughput (Figure 4.6), V4VSockets outperforms the default case as well. V4VSockets peaks a maximum throughput of 2299 MB/s, 4.59x better than the split-driver, which performs poorly at 501 MB/s for 1 MB messages.

To examine how V4VSockets scale with a various number of VMs exchanging messages, we measure the system's throughput for 2, 4, 8 and 16 VMs communicating in pairs (Figure 4.7). The aggregate throughput increases proportionally to the number of communicating VMs. For instance, two VMs are able to exchange 512 KB messages at $\approx$ 2 GB/s, while 16 VMs achieve 8x aggregate throughput for the same message size ($\approx$ 16 GB/s).

Based on our approach (Section 4.1), V4VSockets performs three data copies when transferring messages across: VM_1–to–Xen, Xen–to–VM_2-kernel, VM_2-kernel–to–VM_2-user space. This is a design choice in order to avoid the

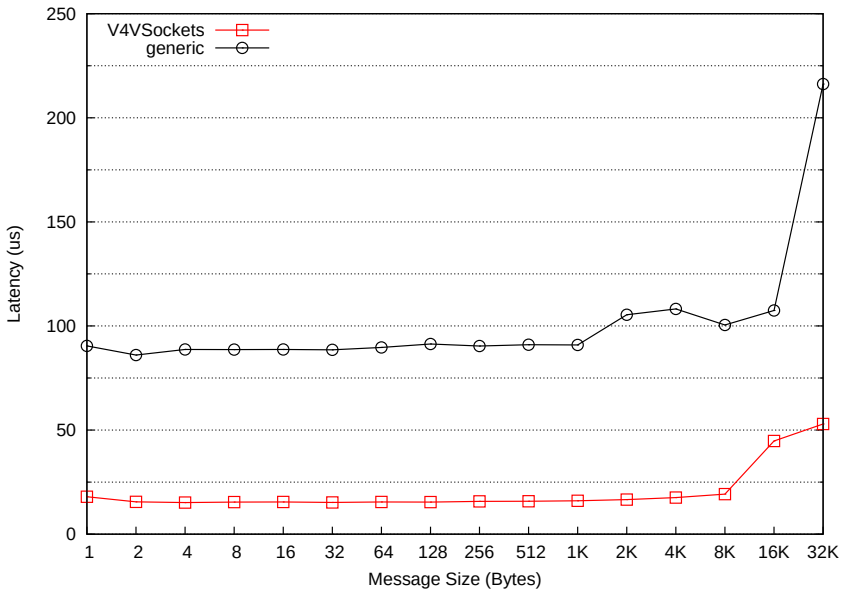


Figure 4.5: V4VSockets latency

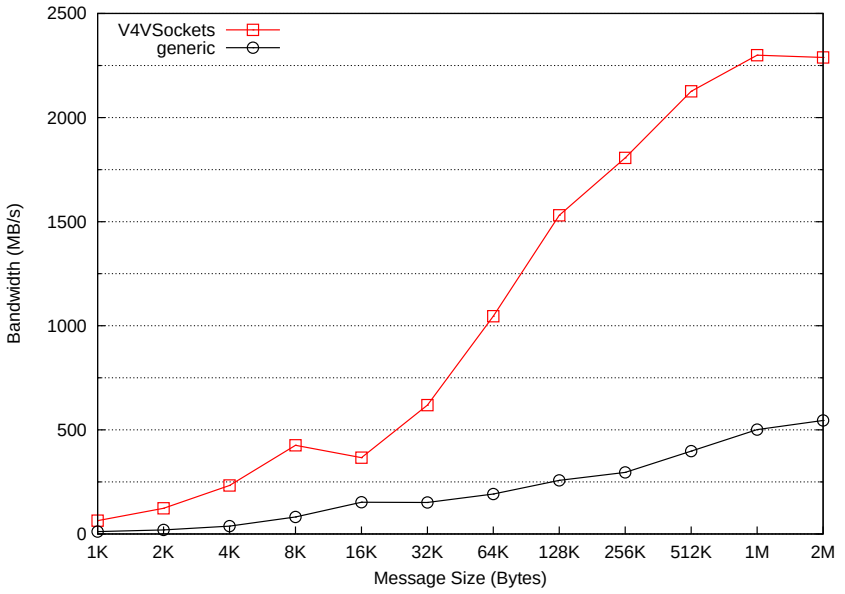


Figure 4.6: V4VSockets throughput

shared memory alternative, which has potential security implications for various use cases. According to this design: VM_1 notifies through a system call and

a hypercall that there is a message for VM_2. Xen copies data from VM_1 user space into VM_2 and notifies the kernel. When the kernel wakes up, data are already in the processor's cache, and thus, data flow directly to VM_2 user space. As a result, we are able to reach more than half of the system's memory bandwidth[1], bringing memory-copy-like bandwidth measurements to VM–to–VM message exchange.

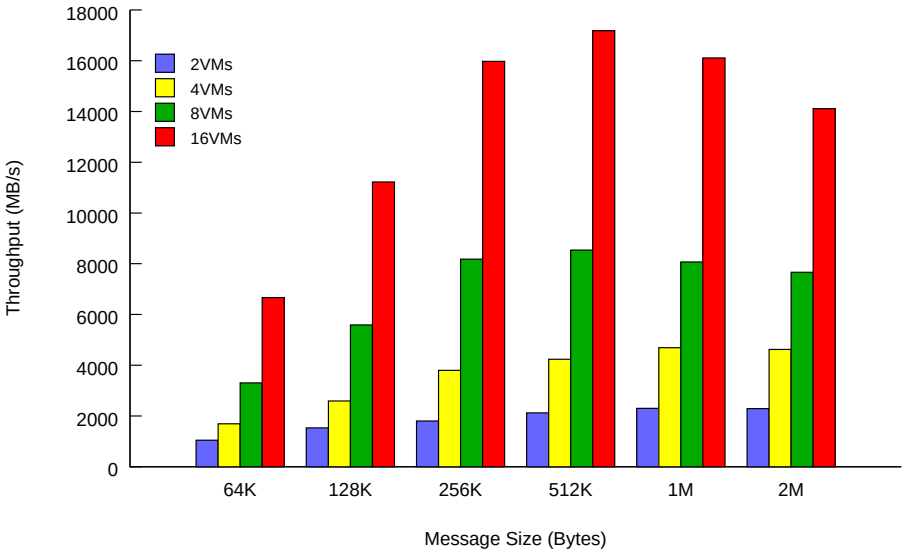


Figure 4.7: V4VSockets aggregate throughput

To validate that the system sustains acceptable performance when a large number of VMs put pressure on the memory bus, we examine the effect that our data exchange mechanism has on latency. When 16 VMs exchange small messages in pairs, the round trip latency remains as low as 16 us, verifying the scalability of our approach.

4.3.2 GPU-enabled VMs

In this section, we demonstrate the merit of V4VSockets on a real-life benchmark from the GPU applications domain. As we previously described, we apply our efficient transport mechanism to rCUDA. Building on existing

[1] We performed a stream microbenchmark and measured 27 GB/s as the maximum memory bandwidth.

remote acceleration frameworks and V4VSockets, we enable VMs to benefit from a GPU-equipped VM residing in the same host, without complicated setups or disruptive and expensive techniques such as IOV.

We use two VMs, VM_1 acting as the rCUDA server, and VM_2 as the client. In order to provide GPU access to the rCUDA server domain, we assign the GPU device to this VM using PCIe passthrough. Finally, we consider two cases: the generic transport mechanism using TCP/IP over the split-driver model (*rCUDA generic*) and V4VSockets (*rCUDA over V4VSockets*). As a baseline, we perform the exact same experiment without the intervention of rCUDA, directly on VM_1 (*passthrough*)[2].

We use a common HPC application stencil, the single precision matrix-matrix product, provided in CUDA by the NVIDIA samples [55].

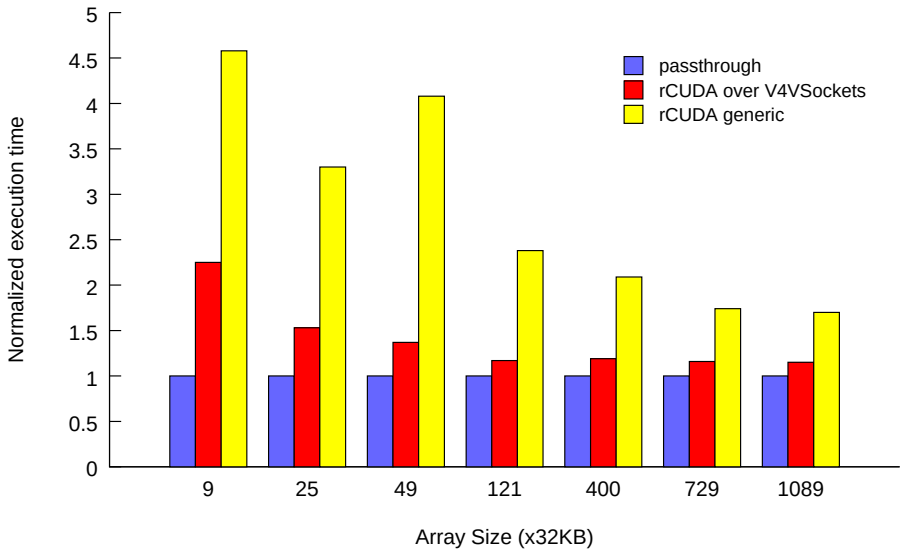


Figure 4.8: Matrix product total time of execution

This experiment includes the following procedure: two copies of the input matrices from node's main memory to GPU device memory, the product execution on the GPU and finally one copy of the output matrix back to main memory. The normalized total time of execution of the matrix-matrix product

[2] We validate the measurements in a non-virtualized environment with an identical GPU device. We did not observe significant difference in terms of performance compared to the passthrough VM execution.

benchmark is depicted in Figure 4.8. The X axis presents the array size multiplied by 32 KB.

We observe that V4VSockets performs really close to the baseline case. For an input matrix size of 1089 x 32 KB (2112 x 4224 `float` type elements) V4VSockets adds a 15% overhead compared to running locally, whereas the generic case adds a 71% overhead. This is essentially our goal: through V4VSockets and rCUDA, VMs can seamlessly share a GPU device in a single VM host with a minimum overhead compared to the generic case. Given that full stack HPC applications use large matrix sizes, our framework can provide the necessary bandwidth to offload GPU execution with the minimum overhead due to remote execution.

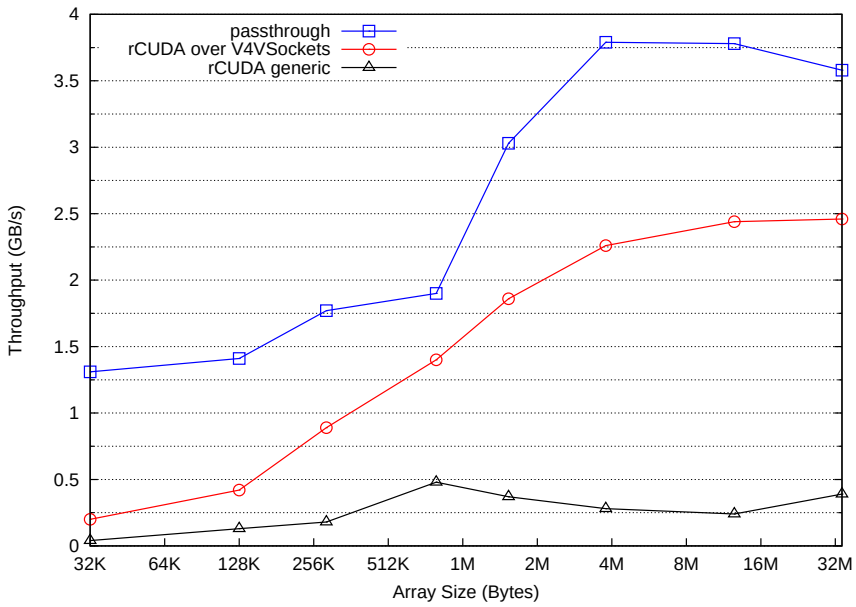


Figure 4.9: Matrix product transfer throughput

To elaborate more on the impact of V4VSockets to the improvement on the execution time, we plot the throughput achieved when copying one of the input matrices from the machine's main memory to the GPU device memory (essentially this is a `cudamemcpy()` call) in Figure 4.9. From the array sizes presented here, the baseline experiment achieves 3.79 GB/s in terms of throughput and it can reach 4.42 GB/s for the whole benchmark (for larger sizes not shown

in the Figure), while the peak throughput in the remote V4VSockets case is 2.46 GB/s. However, for a matrix size of 34 MB (2112 x 4224 `float` type elements) V4VSockets outperforms the generic case by a factor of 6.3 (0.39 GB/s).

4.4 Summary

Summarizing, we presented V4VSockets, a highly efficient communication framework for VMs running on the same physical node. V4VSockets is compatible with socket API and can be used with various socket applications executing on co-located VMs, such as web servers, database servers or network functions (firewall, load balancers etc.). In this thesis, we focused on the GPU use case and we combined V4VSockets with rCUDA, a remote GPU execution framework. In this way, we ended up with a high-level approach avoiding the *complexity* of a fully virtualized GPU and overcoming both the *closed-source* GPU system software and the *virtualization-unaware* accelerator. Additionally, this approach provides *transparency* to CUDA applications but not to GPU kernels written in different APIs. Regarding the *scheduling* process, we provide the mechanism for applying coarse-grained policies, although exploring scheduling techniques in this context is beyond the scope of this thesis. Based on our experience, we argue that modifications to the hardware are necessary for future accelerator technologies, in order to apply efficient scheduling methods.

5

Enabling sharing of Xeon Phi accelerators in virtualized environments

In this chapter, we describe vPHI, a low overhead Xeon Phi virtualization framework, that accelerates virtual machines by enabling them to offload tasks on a Xeon Phi card. To our knowledge, vPHI is the first approach that enables Intel's Xeon Phi sharing between multiple VMs running on the same physical node. Our framework is binary-compatible with precompiled applications, alleviating the need for porting or even recompiling existing source code. It also supports all three modes that are defined in the Xeon Phi model of execution, i.e. *native*, *offload* and *symmetric*.

We use Xeon Phi architecture in the form of device accelerator as our use case and enable coprocessor sharing between virtual machines on the same physical host by virtualizing the transport layer of the accelerator's software stack. Compared to our previously described approach of GPU virtualization, in this work we choose Xeon Phi accelerator, since the corresponding software stack is provided as open-source and thus we are able to virtualize lower levels of the stack and explore the benefits and limitations of such an approach.

In order to leverage Xeon Phi coprocessor capabilities in virtualized environments, Intel offers a couple of solutions with KVM [23] and Xen hypervisor [24] using the *PCIe passthrough* method. Additionally, VMware supports Xeon Phi with ESXi using the same technique [79]. These virtualization approaches offer near-native performance at the expense of the inability to share a single accelerator device to many VMs, since the Xeon Phi card is being directly assigned exclusively to a single VM. With vPHI we manage to enable sharing of a Xeon Phi device by multiple co-located virtual machines.

In the following, we describe the design and implementation of vPHI and

afterwards we present the evaluation results executing various experiments.

5.1 Design and implementation of vPHI

One of the basic requirements in the design of vPHI is to enable sharing of a single Xeon Phi coprocessor between co-located virtual machines on the same physical node and to provide the ability to multiple VMs to concurrently offload workloads to the accelerator. We implement vPHI using QEMU-KVM as the hypervisor and virtio as the paravirtualization interface. As shown in Figure 5.2, vPHI consists of a guest kernel driver and a QEMU backend device implemented in host user space. During the implementation phase, we gradually proceeded with two versions of our prototype due to performance implications. In the first version of vPHI, we needed to make a slight modification to the KVM host kernel module, in order to properly redirect page faults to the guest, while in the second version, we modified the host Xeon Phi driver. Despite that our goal is to be as less intrusive as possible, the modification of the first version is necessary in order to support mapping of guest user memory areas to Xeon Phi device memory through `scif_mmap()`, while the second version's modifications in the host driver benefit vPHI in terms of performance. We further analyze these modifications in the following paragraphs.

Essentially, vPHI intercepts original SCIF transport requests and redirects them through the backend QEMU device. Upon receiving these requests, the backend driver forwards them to the host SCIF driver, which controls the physical device. After the completion of an I/O request, the results are pushed back to the stack following the opposite direction eventually reaching the original requester, which usually is a runtime's transport layer. Simultaneous multi-threaded execution requests from different VMs can end up running in parallel on the Xeon Phi device spread across the available cores of the card. If there is an oversubscription regarding requested threads to physical cores ratio, then the resource multiplexing is accomplished by the scheduler of the uOS which runs on a dedicated Xeon Phi core. Figure 5.1 demonstrates at a high-level view the way that vPHI utilizes virtio to enable sharing of the accelerator device both between applications inside a single VM as well as between different VMs running on the same physical host. At a more detailed level, in Figure 5.2 we con-

sider a SCIF request triggered by an application inside a VM and show the corresponding I/O path. This is a representative scenario that occurs in any of the three aforementioned Xeon Phi modes of execution. Solid lines represent the control path, while dashed lines represent the data path. We refer to each phase of Figure 5.2 in the subsequent paragraphs, as we describe each component and its operation.

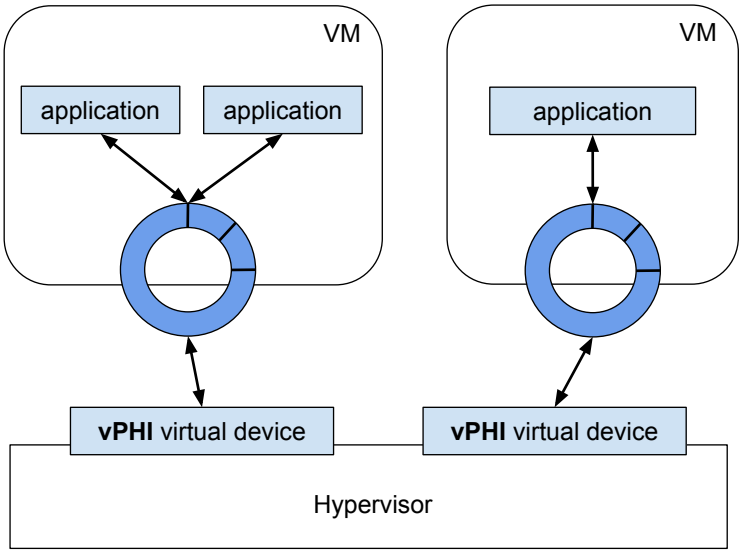


Figure 5.1: vPHI sharing scheme

Based on Figure 5.2, we describe the example of an application executing on a VM that offloads computational kernels to the Xeon Phi card. As we mentioned previously, this scenario can occur in parallel with different applications inside a VM or even by multiple VMs using vPHI, which enables sharing at both levels. In Figure 5.2, we show an example of two VMs (`VM_1` and `VM_N`) that concurrently offload computational tasks sharing the single Xeon Phi decice. In the following, we focus our description on one of the two VMs, but the procedure is the same for every other VM that follows this example. The runtime has to request DMA transactions with Xeon Phi as the destination for the binary, libraries etc., through (`libscif`) library (step a). Afterwards, `libscif` issues (step b) the corresponding system call (`open()`, `close()`, `ioctl()`, `poll()`, `mmap()`) depending on the operation requested. Most of the SCIF

functionality is exposed to user space through different `ioctl()` commands. Since vPHI implements SCIF operations, both the runtime/application as well as `libscif` remain intact and no recompilation is even needed. Hence, the issued system call is intercepted by the vPHI frontend driver.

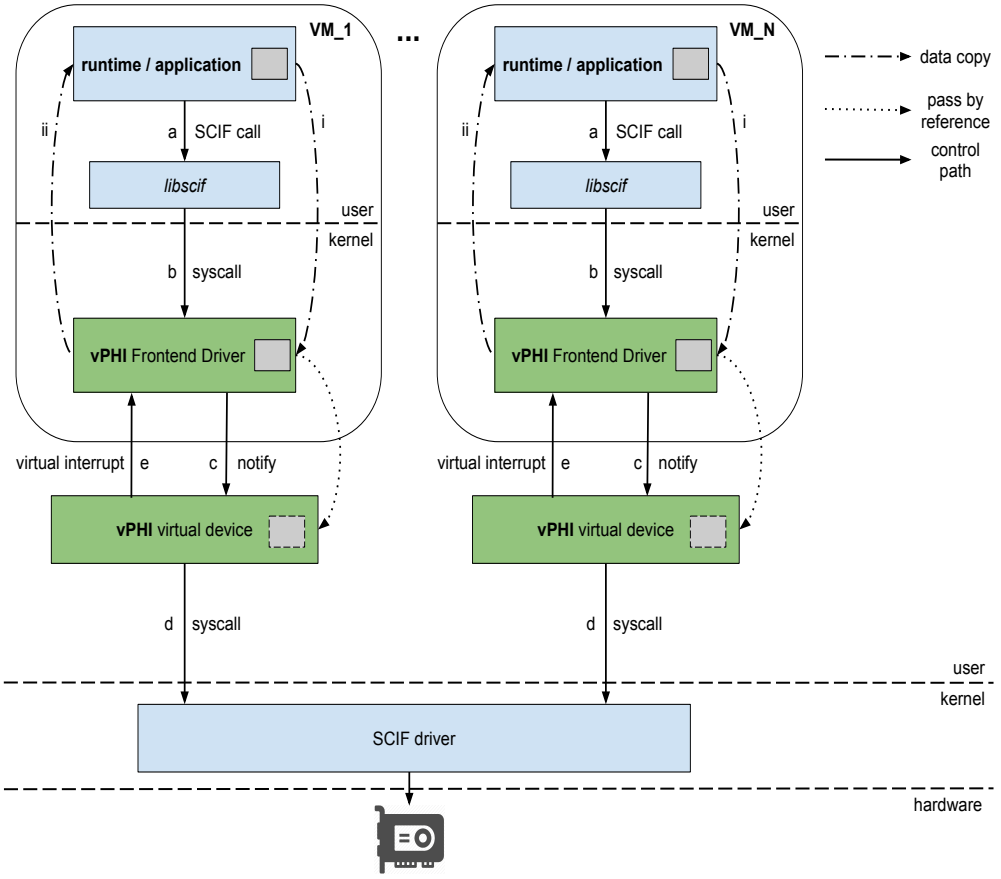


Figure 5.2: vPHI Architecture (data and control Path)

vPHI frontend driver: We implement vPHI frontend driver as a Linux kernel module which is inserted dynamically at guest kernel space. The driver acts as a "glue" between virtualization-unaware `libscif` and the rest of the stack by forwarding the operations requested to the vPHI backend device through virtio communication channels. Among its duties, the frontend driver multiplexes requests and orchestrates the user space threads or processes that are waiting for a response from the coprocessor. We had two design choices in this

step: we can either implement a polling-based method or an interrupt-based one. Since busy-waiting on a shared resource consumes CPU cycles, we chose the interrupt-based approach, adding up some extra overhead when the driver sets up the sleeping mechanism, in favor of better performance: i) when the number of parallel requests increases and ii) when the total procedure lasts much more than the registration of the sleeping mechanism. Thus, the driver places a reference to a buffer in the shared ring structure, then notifies the back-end device (c) that there is a pending request and registers the waiting mechanism until a wakeup event arrives. When this happens, the interrupt handler checks the last response in the shared ring and wakes up the respective entity to continue and push the data up to the stack. Throughout this procedure the only copies that occur are the ones between user space and kernel space at each direction of the path (i, ii). Every other data exchange is realized through references reducing the virtualization overhead especially for large data transfers.

vPHI backend device: We design vPHI backend device as a virtual PCI device and implement it as a QEMU extension. As we mentioned previously, the backend is notified by the frontend (c) when a new request has been pushed to the virtio ring. Then, the backend checks the shared ring and maps the buffer to its address space avoiding again any copies. The backend has access to the memory mappings of guest physical address space to host user space, since it registers guest memory when the VM boots. Afterwards, the backend performs the relevant system call (d) to the host SCIF driver and waits for the result. When the system call returns, it pushes the result to the shared ring and notifies the guest via a virtual interrupt (e). Following this approach, every VM on the same physical machine, is represented by a different QEMU host process. Thus, Xeon Phi sharing is enabled, as from the host driver's perspective, multiple VMs issuing SCIF requests are essentially multiple host processes that execute system calls to the SCIF driver in parallel.

Blocking vs non-blocking mode: QEMU has been implemented based mainly on an event-driven approach. As such, QEMU handles events as they are produced and during that time the whole VM is in blocking mode. Any previously running entity inside the guest pauses. This model prevents race conditions and avoids many synchronization points at the cost of suspending the execution of the virtual machine. That way, event handling should complete as

early as possible to prevent noticeable pauses of the guest. In a few cases, when this is not possible, QEMU follows a threading model, according to which it spawns a worker thread that executes the long-running handling of the event, and falls back to the event-driven mode unfreezing the VM.

In designing vPHI functionality, we had to decide between these two modes for SCIF operations. Following QEMU's approach, we chose the blocking mode for most SCIF operations and a non-blocking mode for operations that otherwise would potentially block the virtual machine for an unacceptable period of time. For example, SCIF defines a connection-oriented model between two endpoints before data transfers begin and in this context it implements `scif_listen()` and `scif_accept()` with similar logic to the POSIX sockets API. Hence, we implement `scif_accept()` in a non-blocking way, since we do not know beforehand when a corresponding `scif_connect()` request will arrive. For data transfer requests, we follow the blocking model, although one can argue that the blocking cost increases proportionally with the data size. The performance tradeoff to consider here arises on the one hand from the blocking cost that prevents any other threads inside the guest to make progress, while on the other hand from the overhead of creating and eventually destroying the worker thread. As the data size increases, the non-blocking method appears more appealing. Hence, compared to our first prototype, we implement an additional non-blocking operation for send-receive operations and integrated into the framework.

Guest memory registration and MMIO: Apart from two-way send-receive communication semantics, SCIF also supports remote memory access for exchanging data between host and device memory, exposing the relevant API (`scif_(v)readfrom()`, `scif_(v)writeto()`). For a buffer to be involved in a set of remote memory operations, the relevant memory pages have to be *pinned*. Memory pinning refers to a procedure according to which a page or a set of pages are marked in a way that prevents the operating system from swapping them out. In this way, a subsequent remote memory **read** from these pages would load valid data to the remote node. If the respective pages are not pinned, and happens to have been swapped out, the read operation will acquire invalid data, without any chance to produce a page fault and bring back the original data from the disk. Likewise, a remote memory **write** operation

could overwrite data of some other process in case of a previous swap out. SCIF exposes the memory pinning functionality through `scif_register()`/`scif_unregister()` calls. In vPHI's implementation of memory pinning we first pin the pages that the user requested in the guest operating system. These pages correspond to the user-supplied buffer which refers to the guest user space. However, the buffers which are pushed to the shared ring always refer to the guest kernel space and subsequently, virtio translates them to guest physical space in order to be later accessed by QEMU backend through the mapping to its own host user space. Thus, before it uses the shared ring, vPHI first maps the respective pinned page to a kernel address, and then it pushes this buffer to the virtio ring for further processing. This operation must be performed for each individual page, since a contiguous buffer is required by the host driver and this is not guaranteed at guest physical (or equally host virtual) address space. At this point, we perform an optimization compared to the first version of our prototype in order to further reduce the virtualization overhead of memory registration. Specifically, we modify the host driver by adding another `ioctl()` operation, namely `SCIF_REGV`, in order to support input of non-contiguous buffers inside an array. As a result, the backend performs a single `ioctl()` reducing the total registration overhead of the first version due to the unawareness by the host kernel driver of the existence of scattered guest memory.

Apart from remote read/write operations with the pinned buffers, SCIF supports memory mapping of remote buffers to the local address space through `scif_mmap()`. After a successful call to `scif_mmap()`, in order to access Xeon Phi memory, the user can simply dereference (load/store) the relevant `mmap`'ed memory address without any intermediate library or system call. This memory access would either page fault to the host operating system, which will confirm the validity of the mapping and fetch the corresponding frame to main memory or retrieve the data from the main memory that a previous memory access has forced to fetch. Inside `scif_mmap()` the host properly setups the corresponding memory management structures pointing to device memory. In vPHI, we perform a two-level mapping, one from the user-supplied address to a guest physical frame and a second from the guest physical frame to the host physical frame, which corresponds to Xeon Phi memory. The problem with this

approach lies in the fact that if an application running inside a virtual machine performs e.g. a pointer dereference of a previously successfully mapped buffer, then it will fault into the KVM host kernel module, which will try to examine the situation based on the address that faulted. However, this address will be interpreted by the host driver as a reference to its own address space leading to an invalid memory area. In order to overcome this problem, we have to make a slight modification to the host driver as well as the KVM. Linux kernel separates different mappings by defining different *vmas* (virtual memory areas). We therefore tag every vma that has been created by vPHI during `scif_mmap()` using a new label (`VM_PFNPHI`) and store the relevant physical frame number. Then, in every fault that is triggered by a vPHI `mmap`'ed area, KVM spots the frame number that corresponds to the respective Xeon Phi memory region.

Implementation details: During a SCIF data transfer using vPHI, the guest frontend driver first allocates a buffer and copies the user-supplied data (for the send/write case) or the received data (for the receive/read case). In this step we need to allocate guest physically contiguous pages, so we use kernel's `kmalloc()` API, since this set of pages will be later used for I/O between guest and host through virtio ring. However, the Linux memory subsystem imposes a limitation on the maximum set of physically contiguous pages that can be allocated. This upper limit is defined in the kernel (`KMALLOC_MAX_SIZE`) and depends on the architecture. Specifically, for `x86_64` architecture the limit is 4MB. Hence, if the requested data size is greater than this value, we implement the data transfer breaking up the allocation to `KMALLOC_MAX_SIZE` elements and proceed with each one of them.

The host Xeon Phi driver exposes a set of information related to the Xeon Phi, such as the family codename of the accelerator, through the `sysfs` filesystem. Some of Intel's software runtimes and tools rely on this information to operate as intended. Thus, we implement the necessary functionality, in order to be able to offload workloads on the Xeon Phi, and we expose the same information that is provided in the host.

5.2 Performance evaluation

5.2.1 Experimental setup

In this section we describe the experimental evaluation methods we used to analyze the behavior of vPHI in several scenarios and present the results that emerged. We setup a host machine with 1x Intel Xeon E5-2695 v2, 64GB RAM (DDR3-1600Mhz), equipped with an Intel Xeon Phi 3120P coprocessor. We configure the node as a VM host using QEMU-KVM as the hypervisor.

At first, we implement a set of microbenchmarks to evaluate raw SCIF performance of the first version of our prototype and we show the results in Section 5.2.2. Next, we conduct a higher level experiment using `dgemm` from Intel samples [38] for matrix multiplication. For the `dgemm` experiment we follow the *native* mode of execution according to Intel's execution model.

In native mode of execution there are two choices. The user can either `ssh` to the accelerator and execute the application locally, or launch the MIC executable directly from the host. In the first case the user should explicitly copy the executables, libraries and other dependencies on the coprocessor and then execute the application. In a virtualized environment, this can become possible by configuring a network bridge on the host between the emulated `mic0` network interface and the interface that is attached to the VM. However, this configuration is not well-suited for cloud environments. Such setups can end up with many users logged in a shared accelerator environment ruining the isolation characteristics of cloud computing. Hence, we test the native mode using the latter case described, which is enabled by vPHI.

Afterwards, we evaluate the optimized version of vPHI and we conduct a set of high-level experiments based on the Scalable Heterogeneous Computing Benchmark Suite (SHOC) [75] using the *offload* mode of execution. Finally, we use one of the SHOC benchmarks to demonstrate the sharing feature of vPHI and how it behaves when multiple VMs use the single Xeon Phi coprocessor.

5.2.2 Microbenchmark performance

We implement a set of microbenchmarks that exchange data over the PCIe between the host and Xeon Phi using SCIF. We execute these benchmarks in

order to estimate the virtualization overhead of vPHI. We analyze vPHI performance using send-receive two-way communication as well as remote memory operations. First, we execute the benchmark on the host, in order to obtain the baseline performance. Then, we spawn a single-core VM with vPHI and execute the benchmark in the virtualized environment. In both cases, a corresponding server is executed on the coprocessor, in order to serve SCIF send request (in the send-receive case) or properly register device memory (in the remote memory operation case).

In order to measure latency, we use the send-receive benchmark, according to which a SCIF server is launched on the accelerator, listens for connection requests and when a connection is established, it blocks on `scif_recv()`, waiting to serve data to the respective client. In this context, a SCIF client is executed on the host (or on the VM), which connects to the server and sends a number of data. We show the corresponding latency measured for the host as well as for the vPHI in Figure 5.3.

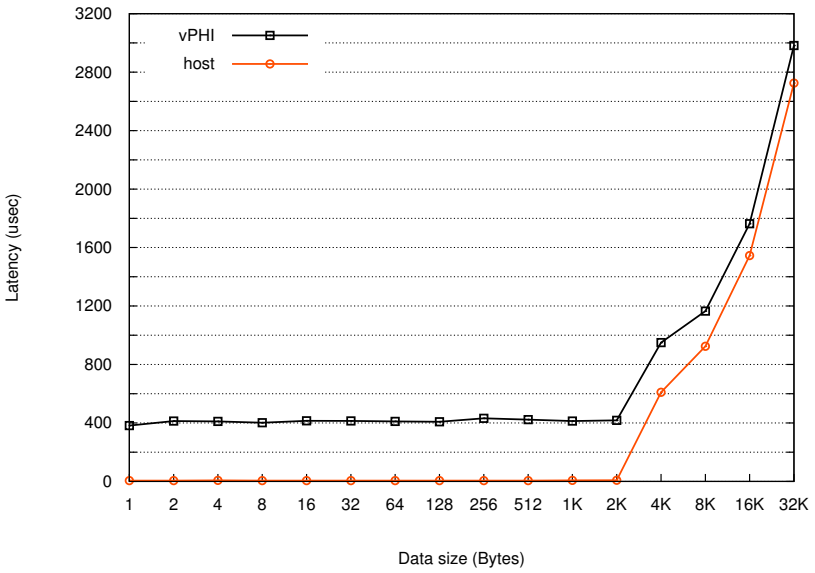


Figure 5.3: Send-receive communication latency

For the native (`host`) execution the latency for sending 1 Byte is 7 us, while for the virtualized one, the respective latency climbs up to 382 us. Hence, the

virtualization overhead of vPHI is 375 us (=382−7). Since this is a notable increase, we performed deeper breakdown measurements to further investigate the cause of this overhead. Based on the breakdown analysis, we conclude that 93% of this overhead is attributed to the waiting scheme of vPHI inside the frontend driver. More specifically, when the frontend driver issues a SCIF request to the shared ring, the relevant process is placed on a waiting queue, until the request is fulfilled. When the backend has finished the execution of the request, it triggers a virtual interrupt and the interrupt handler in the guest wakes up all sleeping processes, which check the shared ring to determine if the reply is for them. The mechanism of sleeping and waking up is the main source of performance degradation for latency-sensitive workloads. As we mentioned in the previous sections, this scheme is necessary for larger data transfers in order to reduce the CPU utilization of an alternative busy-wait method. A possible hybrid approach can be used that can switch modes depending on the application. More specifically, it can use each time the best of the two available schemes depending on the requested data size, so we can enable near-native latency for small data sizes, while retaining acceptable transfer rate for larger ones. Finally, in Figure 5.3 we can observe that the previously mentioned overhead remains constant as data size increases, so there is a constant offset in terms of latency compared to the baseline measurement.

Next, we execute another benchmark using the SCIF remote memory access model, which is more suitable for larger data transfers, in order to estimate the maximum throughput that vPHI can attain. In this experiment, we launch an executable on Xeon Phi, that again listens for incoming connections and then pins a device memory area based on the requested size using `scif_register()`. In the host (or VM) side the benchmark requests a connection and afterwards it performs a remote read from the accelerator device. The results are depicted in Figure 5.4. We can observe that the host's remote read can reach 6.4GB/s, while vPHI's respective throughput is 4.6GB/s, which equals to 72% of the host case.

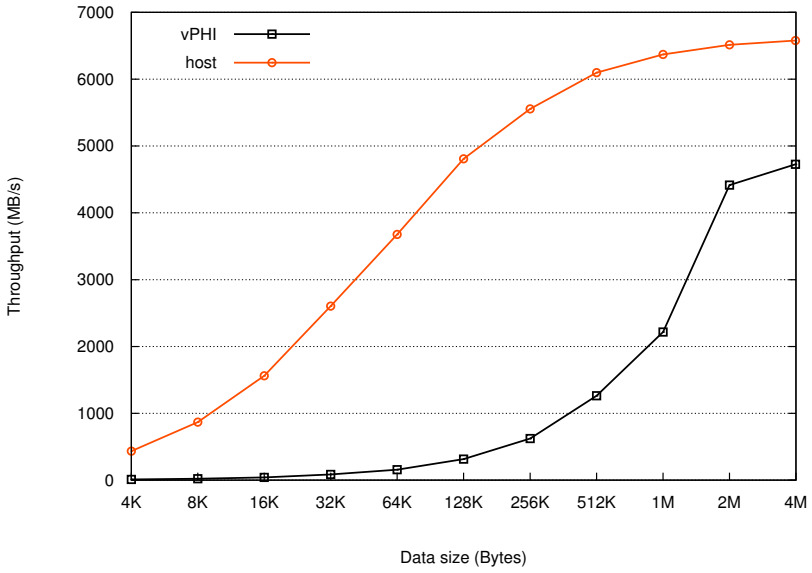


Figure 5.4: Remote memory access throughput

5.2.3 Application performance in native mode of execution

In this subsection we show the results of a higher level application that we used with the first version of vPHI. We measured the execution of `cblas_dgemm` matrix multiplication from Intel samples [38], which uses the MKL [37] library. We use `micnativeloadex`, a tool that Intel provides to enable launching of MIC executables to the coprocessor directly from the host, following the native mode approach. As we previously described, `micnativeloadex` uses the COI library and communicates using the SCIF protocol with `coi_daemon` executed on the coprocessor. `Micnativeloadex`'s role is to properly setup the environment, launch the necessary libraries and executables and spawn the requested number of threads.

In this experiment we execute `micnativeloadex` with `dgemm` as the supplied binary on the host and on the VM. After the moment that the `dgemm` executable has been launched on Xeon Phi and since it is executed as a whole without vPHI intervention, we observed no performance degradation for the vPHI compared to the host concerning actual execution time on the device. In order to estimate the overhead of vPHI in the entire offloading procedure,

however, we also measure the total time of execution from the moment that `micnativeloadex` is launched on the host (or the VM) until the final results are produced and the tool finishes execution. We vary the number of threads as well as the size of the matrices. The accelerator has a total of 57 cores, of which 56 are available for processing, since the uOS uses a dedicated core. It also provides 4 threads per core exposing a total of 224 threads for the user. Hence, we execute the benchmark with 56, 112 and 224 threads. We plot the results in Figure 5.5, Figure 5.6 and Figure 5.7 for 56, 112 and 224 number of threads respectively.

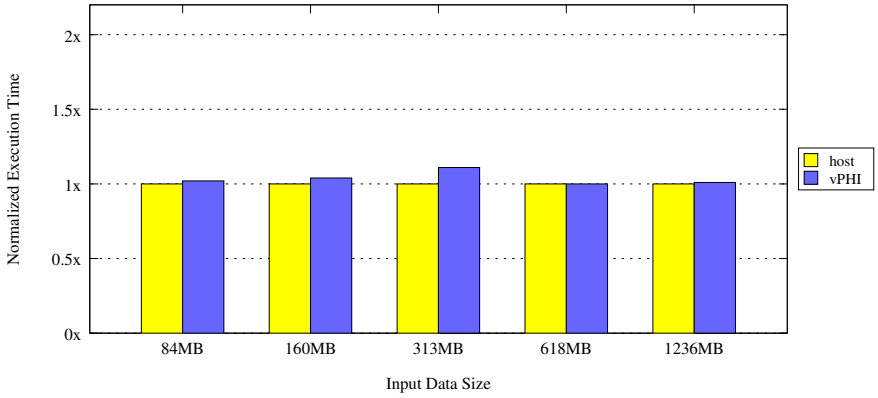


Figure 5.5: Launch and execution of `dgemm` using 56 threads

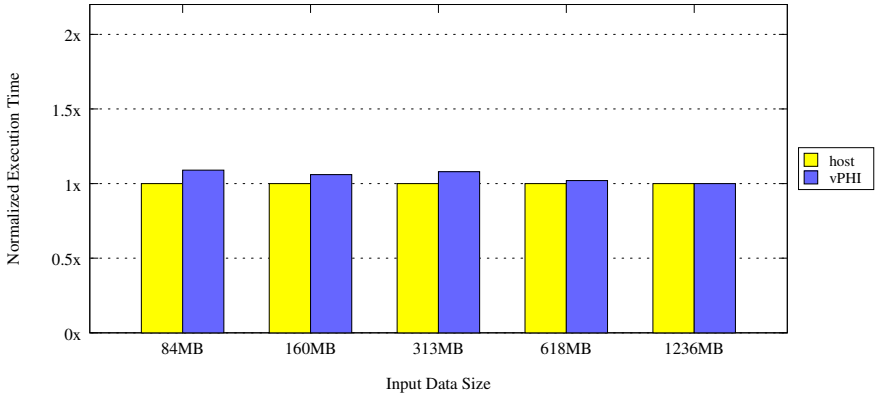


Figure 5.6: Launch and execution of `dgemm` using 112 threads

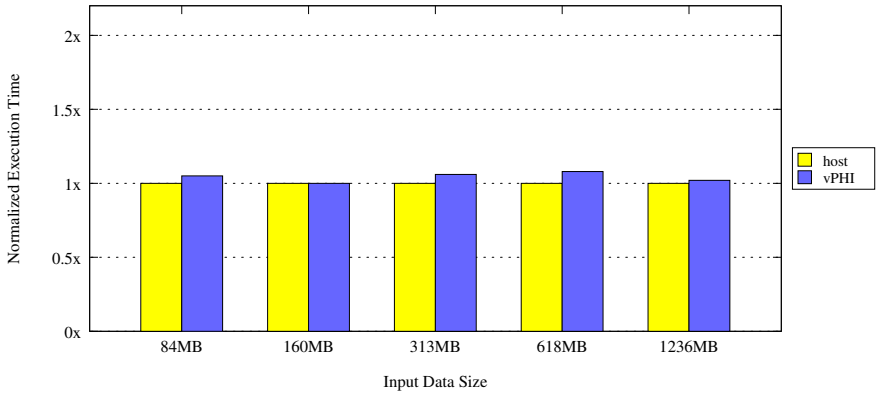


Figure 5.7: Launch and execution of `dgemm` using 224 threads

The Y axis represents the normalized total time of execution that includes the launching of the necessary binaries using `micnativeloadex` from the host (or the VM) and the actual execution on the accelerator. The X axis represent the total size of the two input arrays. We try to setup and present a meaningful experiment in terms of input data size. From the above figures we can draw the conclusion that for larger experiments (in the order of seconds), which include longer-running loops as well as transferring sizable binaries (libraries/executables) over the PCIe, the virtualization cost of vPHI is amortized and the relative overhead compared to the total execution time is negligible. In contrast, as the size of transfered data decreases, vPHI's virtualization overhead has a greater impact, as the previous latency experiments prove. However, given the cost of a PCIe transaction, a typical scenario involving a coprocessor usually consists of loading a substantial amount of data followed by a heavy computational phase, because otherwise it may not worth the effort of offloading a small amount of data and perform a light computation that could be carried out on the local CPU with less overhead.

5.2.4 Application performance in offload mode of execution

In this subsection, we present results from high-level benchmarks that offload popular computational workloads to Xeon Phi. We use the SHOC suite [75] and execute benchmarks i) natively and ii) inside a virtual machine combined with the second version of vPHI, in order to determine the overhead and

the robustness of the system. We also vary the threads that run on the Xeon Phi and execute each benchmark spawning 56, 112 and 224 threads on the coprocessor. We execute the GEMM and SPMV benchmarks 10 times and we plot the mean and standard deviation.

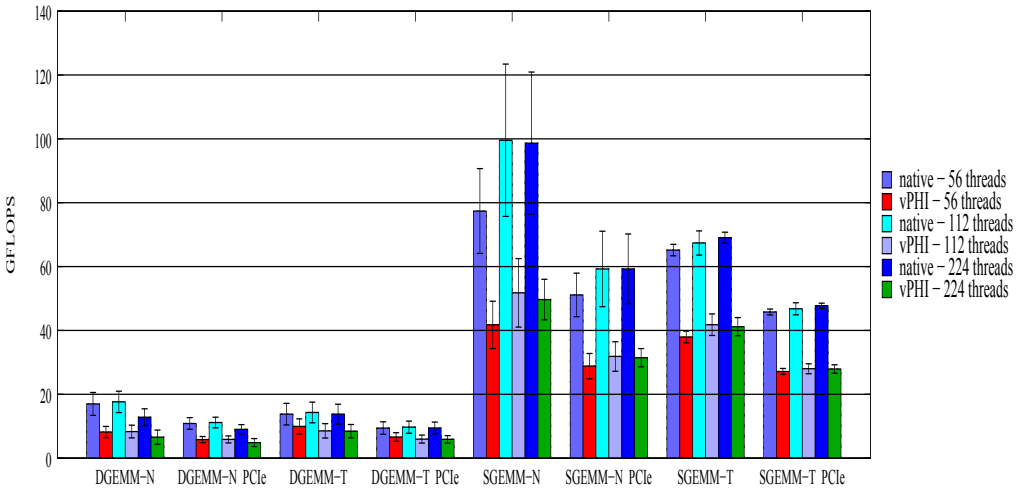


Figure 5.8: GEMM benchmark (problem size `s=1`)

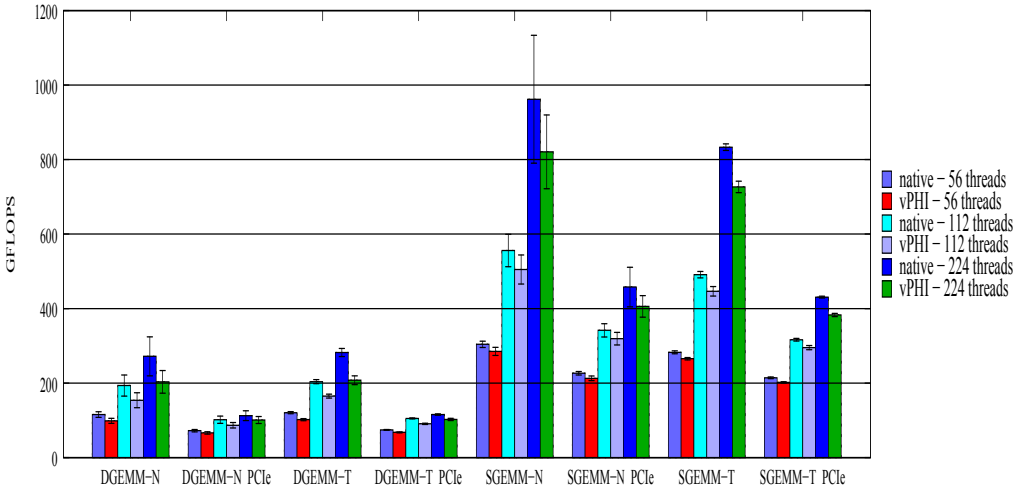


Figure 5.9: GEMM benchmark (problem size `s=2`)

First we present GEMM, which measures performance of matrix-matrix multiplications using Intel MKL (Math Kernel Library) [37]. We execute sin-

gle (SP) and double (DP) precision tests in native and virtualized environment. We use two different problem sizes (s=1 and s=2) and also use the transpose mode of the application. The results are depicted in Figure 5.8 for s=1 and in Figure 5.9 for s=2. Specifically, the s=1 problem size equals to 98304 matrix elements for SP (float) tests and 32768 matrix elements for DP (double) tests, while the s=2 problem size equals to 1179648 matrix elements for SP (float) tests and 327680 matrix elements for DP (double) tests. Each compute kernel is executed 4 times. Finally, the _PCIe suffix indicates that the results include both the execution as well as the transfer time, while the remainders consider solely the execution time. This explains the lower performance of the _PCIe results compared to the non _PCIe ones both for the native and the vPHI case. Concerning the difference in performance between the native and the vPHI scenario for the compute phases, we can observe that the virtualized performance is 47%-72% of the native case (depending on the number of threads) for s=1 problem size and 74%-94% of the native case for s=2. The increased performance overhead for s=1 is attributed to the fact that the computation kernel is small and thus the virtualization overhead participates more in the total time. Regarding the number of threads, in both the virtualized and the native case the performance is significantly increased only for the larger (s=2) problem size as we increase the number of threads. Specifically, for the native case, the performance is increased to 67%-83% from 56 to 112 threads (depending on the operation) and to 38%-73% from 112 to 224 threads, while for the virtualized case the performance is increased to 56%-77% from 56 to 112 threads and to 26%-63% from 112 to 224 threads. Regarding the _PCIe measurements, the virtualized case performs at 53%-70% of the native case for s=1 and 85%-94% of the native case for s=2 problem size. In the first version of vPHI (not shown in the Figures) there was a significant increase in virtualization overhead for the _PCIe results compared to the compute phases. This increase was expected for the _PCIe results, since they expose the transfer time over the PCIe, during which the SCIF registration layer (and consequently vPHI) is more heavily used. However, in the second (optimized) version of vPHI we observe that there is not such an increase, since the virtualization overhead of the registration operations has been significantly reduced. Regarding the number of threads for the _PCIe results in the s=2 problem size, the performance for

the native case is increased to 41%-51% from 56 to 112 threads and to 10%-36% from 112 to 224 threads, while for the virtualized case the performance is increased to 31%-50% from 56 to 112 threads and to 13%-30% from 112 to 224 threads. As we describe in the next subsection, the aforementioned virtualization overhead can be amortized when running applications that transfer their data over the PCIe once and then execute multiple computations on the accelerator.

Next, we present the SPMV benchmarks of the same suite. Figure 5.10 and Figure 5.11 depict the results of SPMV experiments for `s=1` and `s=2` respectively for single and double precision using Intel MKL. For the SPMV benchmark, the `s=1` problem size equals to 1024 rows, while the `s=2` problem size equals to 8192 rows for the matrix. Each SPMV kernel is executed 100 times. We can observe that for `s=2` the `MKL_MIC-DP` with vPHI performs at 90%-96% of the native case. Similarly, the corresponding performance for the `MKL_MIC-DP_PCIe` is at 90%-94% of the native case (for the `s=2` size). Regarding the number of threads, we observe a dropdown for the smaller problem size both for vPHI and the native case, while for `s=2` the increase in number of threads benefits the applications.

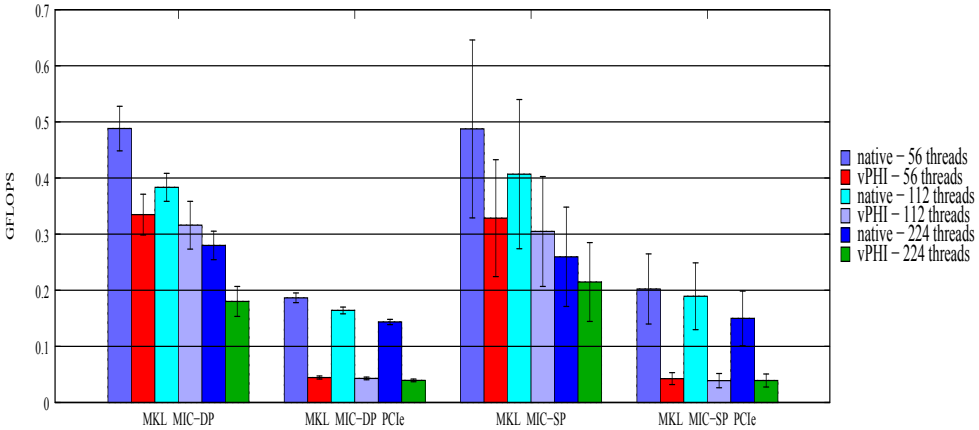


Figure 5.10: SPMV benchmark (problem size `s=1`)

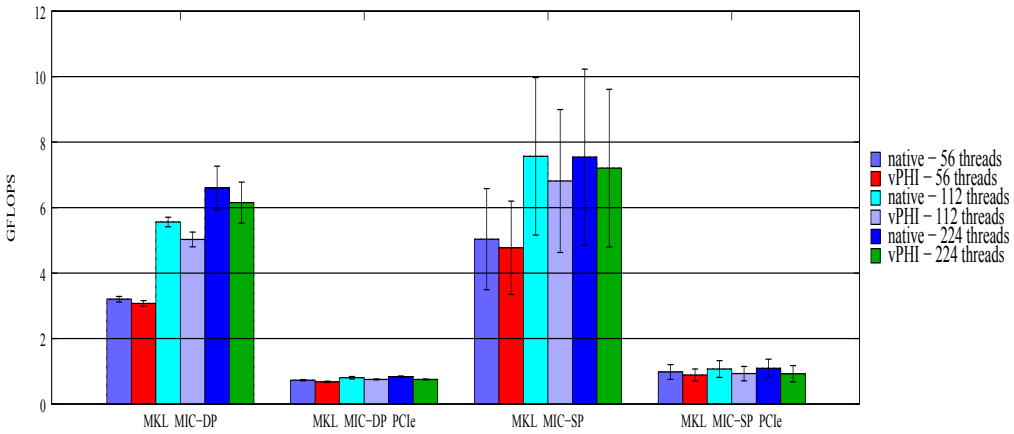


Figure 5.11: SPMV benchmark (problem size `s=2`)

5.2.5 VM sharing

As we observed in the previous experiments, the performance of the data transfer is reduced in the virtualized case. For this reason, in this subsection we gradually build a use case that reduces the significance of the transfers over the PCIe, by either performing them when another VM executes computations on the coprocessor or by preloading the data and then execute operations on the device for longer running experiments. We base our analysis on the Reduction benchmark from the SHOC suite, which measures the performance of sum reduction operation on floating point numbers.

First, we show the virtualization overhead of a single VM that executes the Reduction benchmark. We execute the benchmark 5 times and we calculate the mean and standard deviation. We plot the results in Figure 5.12. For the problem sizes of this benchmark, the `s=1` problem size equals to 1048576 elements for SP (`float`) tests and 524288 elements for DP (`double`) tests, while the `s=2` problem size equals to 2097152 elements for SP (`float`) tests and 1048576 elements for DP (`double`) tests. Each reduction kernel is executed 256 times. Regarding the virtualization overhead, vPHI performs at 90%-98% of the native case in the `Reduction-DP` for the `s=2` problem size. Regarding the number of threads, the performance of `Reduction-DP` for `s=2` is increased significantly from 56 to 112 threads (277% for the native case and 278% for vPHI).

In the following, we demonstrate the effectiveness of vPHI when it is uti-

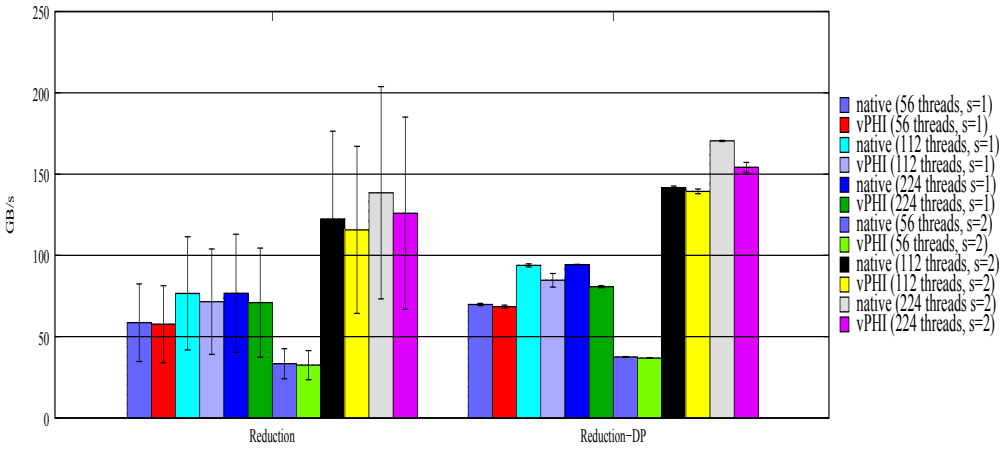


Figure 5.12: Reduction benchmark

lized in parallel by multiple virtual machines. In this context, we launched 1, 2, 4 and 8 processes on the host and 1, 2, 4 and 8 single-vCPU VMs respectively for each configuration and execute the Reduction benchmark from the SHOC suite for each VM (or host process) in parallel. Inside each VM (or host process), we execute the Reduction benchmark 5 times and consider the mean of these executions for every VM (or host process). We also vary the number of Xeon Phi threads as previously. Figure 5.13 and Figure 5.14 depict the results of this experiment for two different problem sizes (`s=1` and `s=2`). More specifically, in Figure 5.13 we plot the raw measurements, while Figure 5.14 shows the normalized results. The results in Figure 5.14 are normalized to the single host process execution for each thread configuration.

Figure 5.14 shows that for the `s=2` problem size and for 56 threads vPHI achieves near-native performance for 2 VMs, 4VMs and 8 VMs compared to the single host process execution. The counter-intuitive observation of the low performance degradation even when every VM uses all the available threads (224) is attributed to the nature of the execution, as there are overlapping phases that the scheduler of the uOS can take advantage of to schedule threads from different VMs. We believe that for accelerators like Xeon Phi, where a transport layer is exposed and also scheduling capabilities are feasible on the device, there is enough room for cloud providers to embrace the heterogeneous characteristics of the resulted infrastructures and expose the sharing potential to their

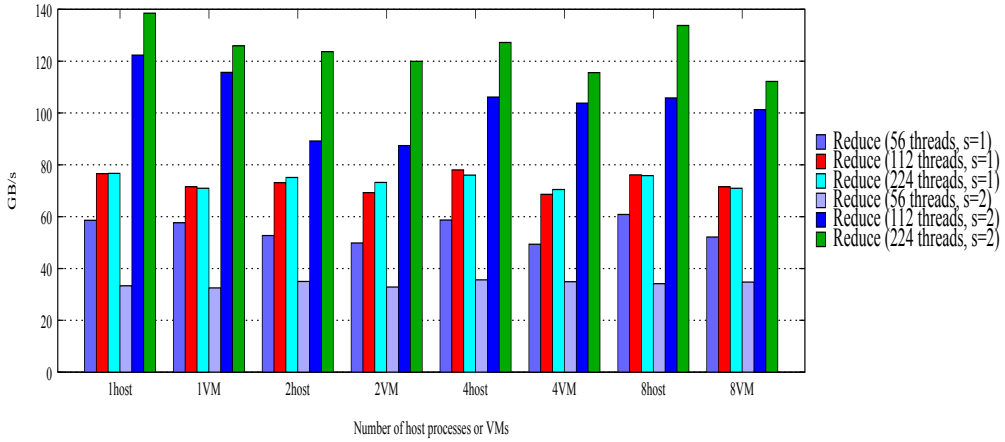


Figure 5.13: VM sharing (compute phase - raw results)

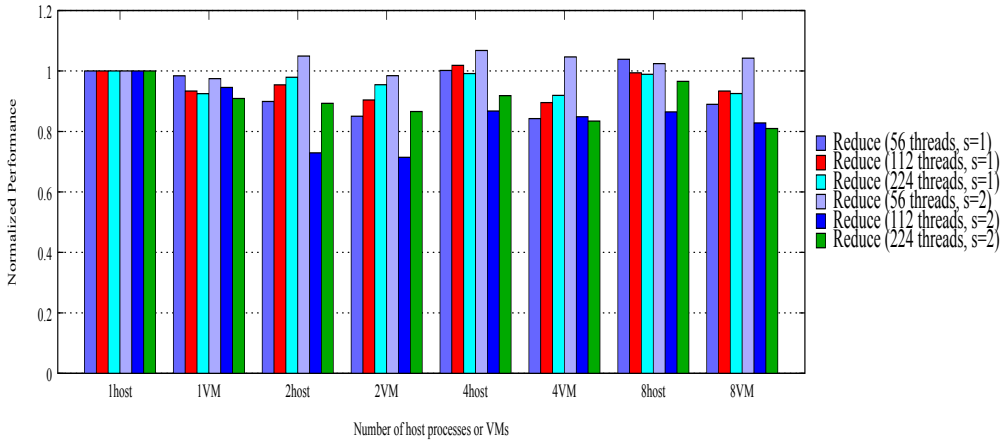


Figure 5.14: VM sharing (compute phase - normalized results)

users.

In the last experiment, we setup a configuration to demonstrate the usefulness of our framework simulating a more dynamic, real-world scenario. Specifically, we consider the use case of multiple VMs in a cloud environment, that while executing an application on the CPU, they need to instantly acquire the accelerator to compute a result as quickly as possible and then continue the execution on the CPU. With this approach we try to simulate long-running daemons that have preloaded their data on the accelerator and at some point in time require a fast computation. This is realized by multiple VMs in par-

allel that acquire the accelerator at some point randomly and in a continuous and dynamic manner. The idea is that the probability of simultaneously co-executing kernels on the accelerator at the exact same time is highly reduced. Additionally, the accelerator resources are utilized more effectively compared to a non-virtualized, non-sharing scenario, during which the accelerator could remain underutilized for much longer phases.

In order to simulate the previously described scenario, we modify the Reduction benchmark from the SHOC suite as follows: when the benchmark on a VM is launched, first it copies the data over the PCIe, it performs a warm-up execution and then it synchronizes until all of the VMs have reached this phase. After this point, we start the timer, since the preloading of data in this experiment is considerd out of the critical path. Next, it processes a loop, according to which it sleeps for a random period of time (from the [0..3]seconds set) and when it wakes up, it executes the reduction kernel for 1024 iterations. After the execution of the kernel, it sleeps again. The application continues this cycle with the benchmark's data already loaded and exits when a specific period of time elapses. We assign each application with 224 threads and execute these loops on the VMs for a total of 5 minutes. We consider a single operation as the total of the 1024 iterations of the reduction kernel.

The result is a dynamic execution of multiple VMs that utilize the single accelerator, when they need to with their data loaded for all this period. The metric in this experiment is the summary of the operations of all the VMs that are completed in the system. Also, we execute the same benchmark as a single host application and we compare the results. The idea behind this choice is that although a single application can result in greater performance compared to a single VM, on the other hand when multiple VMs are launched, they leverage the idle phases of the application and hence the overall throughput is increased. The single applications represents the direct assignment of the accelerator to a single entity. Contrary to the previous SHOC experiments, where we used affinity for the accelerator threads in order to be spreaded on the device cores, in this experiment we only set the number of threads without using affinity trying to approach a real-world scenario, according to which the VM user is not aware of the other VMs' affinities and hence it allows this task to be handled by the accelerator's scheduler. We plot the results in Table 5.1.

We observe that the total throughput of the system is increased by up to 3.56x for 4 VMs, compared to the single host process execution and thus vPHI enables a better utilization of the accelerator. The total throughput reaches its peak value at 4 VMs and beyond that, as we increase the number of virtual machines, the total throughput starts to degrade because of the increased contention on the accelerator.

	Total operations
1 host process	199
1 VM	188
2 VMs	370
4 VMs	709
8 VMs	412

Table 5.1: Total operations completed after 5 min of execution

5.3 Summary

In this chapter, we presented vPHI, a paravirtualization framework of Xeon Phi accelerators which enables sharing of the device between VMs running on the same node. We designed and implemented a virtualization layer of the low-level transport SCIF API, since the relevant software components are provided as *open-source*. Using this technique, we avoid the *complexity* of a fully virtualized accelerator and we provide *transparency* to the higher level software components that use SCIF. Furthermore, we reduce the virtualization overhead due to the accelerator's *virtualization-unawareness* of the scattered guest physical memory by modifying the host driver. In the evaluation section, we tried to setup and presented a long-running experiment in a loop with VMs requiring a quick computation on the accelerator with preloaded data and then sleeping for a random period of time. In this way, the probability of simultaneously co-executing kernels on the accelerator at the exact same time is highly reduced and each VM can be "scheduled in" the accelerator with less overhead due to the existence of its data in device memory. With vPHI we provide the mechanism for applying *scheduling* algorithms, but the thorough study and

exploration of such techniques are beyond the scope of this thesis. In order to apply more effective scheduling policies and to enable more efficient sharing of such devices, we believe that next generation accelerators need to be designed with awareness of the potential existence of virtualized entities on the systems they are attached to.

6

Related work

In this thesis we examine the problem of virtualizing accelerator devices and in this context we have explored various related techniques from previous work. In the following sections, we describe the related work and we mention the key points of each one compared to our approaches.

6.1 Intra-node communication

In Chapter 4, we described our approach of utilizing rCUDA over V4VSockets, our optimized intra-node communication framework in the Xen hypervisor. In this section we describe the previous work towards optimizing communication of virtual machines that reside on the same physical node.

In [82] the author provides an extensive survey of the available intra-node communication mechanisms between virtual machines. In this context, he mentions that a major source of intra-node communication overhead is the complex data path between co-existing VMs. Network traffic between peer VMs is redirected via the driver domain, resulting in a significant performance penalty. Packet transmission and reception involves traversal of the TCP/IP network stack and the invocation of multiple Xen hypercalls. Several optimizations have been proposed regarding this limitation, such as shared memory techniques that are provided by the Xen hypervisor and are exploited to facilitate data exchange between VMs. Using a pool of shared pages for direct packet exchange seems a lot more efficient than traversing the network communication path via the driver domain. XenSocket [85] and IVC [33] provide a basic, one-way communication channel using socket semantics, introducing a new address family type. XenLoop [83], on the contrary, intercepts calls to local VMs through the

Linux netfilter mechanism and establishes a full-duplex data channel between peers to efficiently exchange data. In XWay [42] the authors define a new virtual device that establishes direct communication between VMs, bypassing the driver domain completely. Additionally, in [88] the authors propose an optimized path through the Xen hypervisor using shared memory, while SChannel [32] provides a bidirectional shared memory channel via the driver domain. MMNet [64] eliminates copies by mapping the entire kernel address space of a VM into the address space of its communicating peer. Furthermore, YASMIN [66] provides an optimized intra-node communication path through the hypervisor, but using the less secure shared memory technique. In [27] Guan et al. propose a communication-aware scheduling technique between co-located VMs in order to reduce the corresponding network latency.

Apart from methods like the above that exploit the co-location of the communicating VMs, there has been great effort using less intrusive approaches for optimizing the existing network stack in Xen. For instance, Menon et al. [48] improve network performance by introducing copies instead of page remapping and using advanced memory features, such as superpages and global page mappings.

Similar approaches to the aforementioned works have been proposed concerning other hypervisors as well [14, 28]. Specifically, in [14] Diakhate et al. have proposed an approach for KVM using shared memory techniques via a virtual message passing device targeting MPI applications' virtualized context.

We build on this strand of the literature, but instead of optimizing the data path through the driver domain, we bypass it, using the hypervisor as the network medium. Additionally, we avoid the shared memory technique, reinforcing the isolation and security properties for the communicating VMs.

6.2 Accelerator virtualization

From the accelerator virtualization domain, most of the approaches that have been proposed target mainly GPUs. In the cloud industry, many providers, such as Amazon [77], Microsoft [50] etc. have started to offer GPU computing resources as a service. Furthermore, a class of proposed solutions includes the use of passthrough technology to provide to a VM direct access to a host device.

Additionally, there are solutions [78, 41] that expose a static number of virtual GPUs, each one of them directly assigned to a virtual machine. Other works fall into the full virtualization category [76], according to which no modification is performed to the guest operating system. Some approaches have been proposed using the paravirtualization technique [25, 26, 81], while others employ both full virtualization and paravirtualization methods [71].

API redirection technique with a client-server approach is used by some solutions in order to provide GPU virtualization features [69]. In this context, in rCUDA [62] the authors enable remote execution of GPU tasks, as we have mentioned previously. These works are based on a client-server model that communicate over a transport layer. Although this makes the approches hypervisor-independent, they need to apply transport-specific optimizations based on the interconnecting environment. In this dissertation, we applied V4VSockets as the transport medium of rCUDA, in order to exploit the co-location of VMs in the context of a single GPU that is shared by multiple VMs. Finally, recently, in [19] the remote acceleration technique is used as well targeting the Xeon Phi coprocessor.

As we have mentioned in Chapter 3, the major GPU vendors provide their source code in a closed manner. Considering this fact, the corresponding approaches that virtualize a low-level layer of the stack are impractical, since they are mostly based on reverse engineering methods, that become unusable in new versions of GPU devices in case the vendors introduce significant modifications. Additionally, the works that implement a high-level wrapper library which redirects the path to custom components need to continuously be updated when new functionality is introduced by GPU vendors. In this thesis, we base our approach to a middleware framework (rCUDA) which remains updated by constantly providing new releases.

Regarding Xeon Phi virtualization, to the best of our knoweledge, our approach with vPHI is the first and currently the only solution that enables sharing of a Xeon Phi device by multiple virtual machines. However, there are a few approaches that are related to this problem from a different point of view. In this context, Intel provides a set of patches for Xen [24], as well as KVM [23], to directly assign (*PCIe passthrough*) the coprocessor to a single virtual machine. The same technique is being utilized by VMware [79], as well, to its ESXi hyper-

visor. However, PCIe passtrough-based methods present a major disadvantage, as we described in Chapter 2. Despite the fact that an application running in such a virtual machine can perform at a near-native rate, sharing of a single physical device to multiple VMs is not possible.

In the context of virtualizing Xeon Phi resources, ScaleMP [1] provides a solution that abstracts memory and computing resources and exposes them in a host environment. In this way, an application with high demands on cores or memory can utilize this platform in a transparent manner. This approach eventually provides a large SMP configuration to the user rather than a setup that uses the offload model of execution to virtual machines.

Additionally, there are some approaches towards providing or optimizing virtualization or sharing characteristics for FPGAs, by realizing a traditional FPGA as a virtual reconfigurable hardware [34], reducing the communication complexity between the application and the FPGA resources [46], offering FPGA-as-a-Service [93], integrating FPGAs to cloud stacks [7, 9, 18, 8, 2, 73, 91, 74, 63], focusing on the properties of scheduling [92, 59], confidentiality [87] and secure execution [30] in FPGA virtualization environments or targeting virtual networking appliances [90].

Finally, Yu et al. [89] propose an approach to automatically virtualize arbitrary accelerator APIs. However, in order to construct the virtualized components, an input API specification is required. Then, as they authors state, a developer can virtualize the new API in a matter of days.

7

Conclusions and future directions

Heterogeneous processing has started to play an important role in high performance computing systems and due to power-density problems it appears to have a great potential for future data centers. At the same time, a large amount of HPC workloads is executed in cloud computing environments, due to the benefits that cloud offers both for the users and the infrastructure providers. In this context, there is a growing need to bridge the gap between the accelerator ecosystem and the world of virtualization, which is the building block of cloud computing environments.

Based on these observations, this thesis focused on techniques of virtualizing accelerator devices aiming for application transparency, portability into future environments and acceptable overhead due to the added virtualization layers. To this end, we outlined and categorized the traditional I/O virtualization methods and based on this ground we identified the key challenges of virtualizing accelerators compared to traditional I/O devices.

We described alternative paths for data transfers through the various virtualization layers and we outlined the major design tradeoffs for efficient accelerator virtualization frameworks. In this context, we proposed two approaches targeting two different accelerator ecosystems, each one with its own characteristics. With these approaches, we try to overcome or bypass the accelerator virtualization challenges described in Chapter 3, although for some of them potential support from the hardware is needed, in order to be addressed efficiently. Additionally, we focus on the desired properties that we mentioned in Chapter 1. Specifically, since we propose software approaches, we enable sharing of the accelerator, flexibility (e.g. ability to migrate or apply scheduling policies) and we reduce the total cost of ownership in cloud environments that

expensive IOV solutions would otherwise increase. Regarding transparency, our first approach is compatible with CUDA runtime applications, while our second framework retains compatibility with all components that utilize the accelerator's transport layer. Our frameworks are also as less intrusive as possible, except a modest modification that we performed in the Xeon Phi host driver for optimization purposes. In this context, we tried to reduce the virtualization overhead for our approaches compared to native executions. Finally, we setup an experiment with a non-batch benchmark to demonstrate the more effective utilization of the accelerator.

Regarding our approaches, at first, we targeted NVIDIA GPU devices and we proposed the use of a remote acceleration framework in a single-node virtualization platform combined with a low overhead intra-node framework which results in efficient application offloading in virtualized environments. We described the design and implementation of our intra-node framework, called V4VSockets, outlining the use of the hypervisor as the network medium, instead of the driver domain. Although, in this thesis we focused on the use of V4VSockets in a GPU use case, our intra-node communication framework can be combined with other socket applications as well, since it provides socket compatibility. V4VSockets optimizes the communication of the VMs running on the same node and besides GPU tasks it can boost socket applications executing on different co-located VMs, such as web servers, database servers or network functions (firewall, load balancers etc.). Furthermore, we evaluated our approach using both network microbenchmarks and analyzing a common GPU stencil. Besides performance benefits, we believe that our indirect GPU virtualization approach fits well in accelerators that are based on closed software stacks, as in these stacks the virtualization of lower level system components is impractical. Additionally, our approach retains portability with future library and driver versions, since it utilizes a remote framework that is constantly updated against host libraries by continuously providing new releases. We provide V4VSockets as open-source at https://github.com/HPSI/V4VSockets.

In open accelerator software stacks, such as the one that is released with Intel Xeon Phi, we proposed the use of paravirtualization techniques targeting the lower transport layers of the stack. In this regard, we described the tradeoffs for our design choices in our framework, called vPHI, and we mentioned the

key implementation details. Compared to higher level virtualization methods, our approach provides compatibility with more levels of the stack while at the same time remains transparent to various device architecture modifications, as long as the resulting platform is based on the same transport layer. In this context, we argue that effort can be made for future accelerator devices to support such a generic and common transport. In this way, our method can be applied to these accelerators and provide transparent access by virtualized entities enabling cloud environments to more easily adopt the specialized nature of these devices. Additionally, we thoroughly evaluated our approach in a virtualized server and we showed that multiple virtual machines can share a single Xeon Phi device with acceptable overhead. Especially, in cloud environments where multiple co-located VMs use a shared accelerator for specific periods of time with preloaded datasets, vPHI can be a beneficial approach towards increased overall system throughput leading to better utilization of the accelerator. We provide vPHI as open-source at https://github.com/sgerag/vphi.

Based on the previous experience, we conclude that for non-virtualization aware accelerator devices, paravirtualization techniques can result in major benefits. A critical parameter to decide the target level of the stack to be virtualized is if there are software components of the accelerator environment that are provided in a closed-manner and in which level of the stack they exist. In case of an open-source designed software ecosystem, virtualizing lower level layers of the stack can enable compatibility with various and diverged runtimes and libraries, a benefit which is impractical with proprietary-oriented software.

In the context of future computing systems, we argue that next generation accelerator platforms should be designed considering virtualization properties and sharing of acceleration devices in virtualized environments. Thus, systems engineers will be able to integrate accelerators into cloud ecosystem with minimal effort. This can enhance future cloud systems with an abstraction of heterogeneous resources that can be made available to the users in an elastic manner.

In such environments, the management of resources considering the currently running workloads is a major factor of optimal operation. In this context, future research can focus on new ways of taking dynamic decisions about using acceleration on specific workloads based on policy priorities, such as energy ef-

ficiency, fairness, or QoS (Quality of Service).

Bibliography

[1] Accelerated-Computing - ScaleMP. http://www.scalemp.com/solutions/accelerated-computing, Accessed 2018.

[2] Mikhail Asiatici, Nithin George, Kizheppatt Vipin, Suhaib A. Fahmy and Paolo Ienne. Virtualized Execution Runtime for FPGA Accelerators in the Cloud. *IEEE Access*, 5:1900–1910, 2017. ISSN 2169-3536. doi: 10.1109/ACCESS.2017.2661582.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian A. Pratt and Andrew Warfield. Xen and the Art of Virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177. ACM, 2003.

[4] Luiz Andre Barroso. Warehouse-Scale Computing: Entering the Teenage Decade. *SIGARCH Comput. Archit. News*, 39(3), June 2011. ISSN 0163-5964. doi: 10.1145/2024723.2019527.

[5] Luiz Andre Barroso and Urs Hoelzle. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009. ISBN 159829556X, 9781598295566.

[6] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Anaheim, CA, 2005. USENIX Association.

[7] Stuart Byma, J. Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia and Paul Chow. FPGAs in the Cloud: Booting Virtualized Hardware

Accelerators with OpenStack. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 109–116, May 2014. doi: 10.1109/FCCM.2014.42.

[8] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou and Doug Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016. doi: 10.1109/MICRO.2016.7783710.

[9] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang and Kun Wang. Enabling FPGAs in the Cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 3:1–3:10, Cagliari, Italy, 2014. ACM. ISBN 978-1-4503-2870-8. doi: 10.1145/2597917.2597929.

[10] Guoyang Chen, Yue Zhao, Xipeng Shen and Huiyang Zhou. EffiSha: A Software Framework for Enabling Effficient Preemptive Scheduling of GPU. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 3–16, Austin, Texas, USA, 2017. ACM. ISBN 978-1-4503-4493-7. doi: 10.1145/3018743.3018748.

[11] Tim Cramer, Dirk Schmidl, Michael Klemm and Dieter Mey. OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. In *Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, pages 38–44, 01 2012. ISBN 9783000395451.

[12] Thomas Scott Crow, Dr. Frederick and C. Harris. Evolution of the Graphical Processing Unit, 2004.

[13] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Program-*

ming Languages and Operating Systems, ASPLOS '13, pages 77–88, Houston, Texas, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451125.

[14] François Diakhaté, Marc Pérache, Raymond Namyst and Hervé Jourdren. Efficient shared memory message passing for inter-VM communications. In *4th Workshop on Virtualization in High-Performance Cloud Computing (VHPC '08), Euro-par 2008*, 2008.

[15] Jack Dongarra, Mark Gates, Azzam Haidar, Yulu Jia, Khairul Kabir, Piotr Luszczek and Stanimire Tomov. HPC Programming on Intel Many-integrated-core Hardware with MAGMA Port to Xeon Phi. *Sci. Program.*, 2015:9:9–9:9, January 2015. ISSN 1058-9244. doi: 10.1155/2015/502593.

[16] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, San Jose, California, USA, 2011. ACM. ISBN 978-1-4503-0472-6. doi: 10.1145/2000064.2000108.

[17] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam and Doug Burger. Power Challenges May End the Multicore Era. *Commun. ACM*, 56(2):93–102, February 2013. ISSN 0001-0782. doi: 10.1145/2408776.2408797.

[18] Suhaib A. Fahmy, Kizheppatt Vipin and Shanker Shreejith. Virtualized FPGA Accelerators for Efficient Cloud Computing. In *Proceedings of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, CLOUDCOM '15, pages 430–435. IEEE Computer Society, 2015. ISBN 978-1-4673-9560-1. doi: 10.1109/CloudCom.2015.60.

[19] Konstantinos Fertakis, Stefanos Gerangelos, Georgios Goumas and Nectarios Koziris. RACCEX: Towards Remote Accelerated Computing Environments. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 212–217, Dec 2018. doi: 10.1109/CloudCom2018.2018.00049.

[20] Robert P. Goldberg Gerald J. Popek. Formal requirements for virtualizable third generation architectures. *ACM*, 17:412–421, 1974.

[21] Stefanos Gerangelos and Nectarios Koziris. vPHI: Enabling Xeon Phi Capabilities in Virtual Machines. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1333–1340, May 2017. doi: 10.1109/IPDPSW.2017.110.

[22] Stefanos Gerangelos, Georgios Goumas and Nectarios Koziris. Efficient accelerator sharing in virtualized environments: A Xeon Phi use-case. *Journal of Systems and Software*, 150:37 – 50, 2019. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2018.12.029.

[23] Getting Kernel-Based Virtual Machine (KVM) to Work with Intel Xeon Phi Coprocessors. https://software.intel.com/en-us/articles/getting-kernel-based-virtual-machine-kvm-to-work-with-intel-xeon-phi-coprocessors, Accessed 2018.

[24] Getting Xen working for Intel Xeon Phi Coprocessor. https://software.intel.com/en-us/articles/getting-xen-working-for-intelr-xeon-phitm-coprocessor, Accessed 2018.

[25] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo and Giuseppe Coviello. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I*, EuroPar'10, pages 379–391, Ischia, Italy, 2010. Springer-Verlag. ISBN 3-642-15276-7, 978-3-642-15276-4.

[26] Mathias Gottschlag, Marius Hillenbrand, Jens Kehne, Jan Stoess and Frank Bellosa. LoGV: Low-Overhead GPGPU Virtualization. In *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 1721–1726, Nov 2013. doi: 10.1109/HPCC.and.EUC.2013.245.

[27] Bei Guan, Jingzheng Wu, Yongji Wang and Samee U. Khan. CIVSched: A Communication-Aware Inter-VM Scheduling Technique for Decreased

Network Latency between Co-Located VMs. *IEEE Transactions on Cloud Computing*, 2(3):320–332, July 2014. ISSN 2168-7161. doi: 10.1109/TCC.2014.2328582.

[28] Omid Kashefi Hamid Reza Mohebbi and Mohsen Sharifi. ZIVM: A Zero-Copy Inter-VM Communication Mechanism for Cloud Computing. In *Computer and Information Science*, pages 18–27, 2011.

[29] Nikos Hardavellas, Michael Ferdman, Babak Falsafi and Anastasia Ailamaki. Toward Dark Silicon in Servers. *IEEE Micro*, 31(4):6–15, July 2011. ISSN 0272-1732.

[30] Festus Hategekimana, Joel Mandebi Mbongue, Md Jubaer Hossain Pantho and Christophe Bobda. Secure Hardware Kernels Execution in CPU+FPGA Heterogeneous Cloud. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 182–189, Dec 2018. doi: 10.1109/FPT.2018.00035.

[31] Cheol-Ho Hong, Ivor Spence and Dimitrios S. Nikolopoulos. GPU Virtualization and Scheduling Methods: A Comprehensive Survey. *ACM Comput. Surv.*, 50(3):35:1–35:37, June 2017. ISSN 0360-0300. doi: 10.1145/3068281.

[32] Zang Hongyong, Gui Kuiyan, Li Yaqiong, Sun Yuzhong and Meng Dan. A Highly Efficient Inter-Domain Communication Channel. In *IEEE Ninth International Conference on Computer and Information Technology*, 2009.

[33] Wei Huang, Matthew J. Koop, Q. Gao and Dhabaleswar K. Panda. Virtual machine aware communication libraries for high performance computing. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Reno, Nevada, 2007. ACM. ISBN 978-1-59593-764-3. doi: 10.1145/1362622.1362635.

[34] Michael Hübner, Peter Figuli, Romuald Girardey, Dimitrios Soudris, Kostas Siozios and Juergen Becker. A Heterogeneous Multicore System on Chip with Run-Time Reconfigurable Virtual FPGA Architecture. In *2011 IEEE International Symposium on Parallel and Distributed*

Processing Workshops and Phd Forum, pages 143–149, May 2011. doi: 10.1109/IPDPS.2011.135.

[35] In Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update 20142019 White Paper.

[36] InfiniBand Architecture. InfiniBand Architecture Specification 1.3, 2012.

[37] Intel Math Kernel Library (Intel MKL). https://software.intel.com/en-us/intel-mkl, Accessed 2018.

[38] Intel Software Product Samples and Tutorials. https://software.intel.com/en-us/product-code-samples?value=20825&value=20802, Accessed 2018.

[39] Intel® Xeon Phi™ Coprocessor. https://software.intel.com/en-us/xeon-phi/mic, Accessed 2018.

[40] Christoforos Kachris, Georgi Gaydadjiev, Huy-Nam Nguyen, Dimitrios S. Nikolopoulos, Angelos Bilas, Neil Morgan, Christos Strydis, Vasilis Spatadakis, Dimitris Gardelis, Ricardo Jimenez-Peris and Alexandre Almeida. The VINEYARD project: Versatile integrated accelerator-based heterogeneous data centres. In *2016 5th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pages 1–4, May 2016. doi: 10.1109/MOCAST.2016.7495121.

[41] Shinpei Kato, Michael McThrow, Carlos Maltzahn and Scott Brandt. Gdev: First-class GPU Resource Management in the Operating System. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 37–37, Boston, MA, 2012. USENIX Association.

[42] Kangho Kim, Chei-Yol Kim, Sung-In Jung, Hyun-Sup Shin and Jin-Soo Kim. Inter-domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 11–20, Seattle, WA, USA, 2008. ACM. ISBN 978-1-59593-796-4. doi: 10.1145/1346256.1346259.

[43] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin and Anthony Liguori. KVM: the Linux Virtual Machine Monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07)*, 2007.

[44] Zhen Lin, Lars Nyland and Huiyang Zhou. Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 77:1–77:11, Salt Lake City, Utah, 2016. IEEE Press. ISBN 978-1-4673-8815-3.

[45] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron and Mary Lou Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 248–259, Porto Alegre, Brazil, 2011. ACM. ISBN 978-1-4503-1053-6. doi: 10.1145/2155620.2155650.

[46] Stelios Mavridis, Manolis Pavlidakis, Ioannis Stamoulias, Christos Kozanitis, Nikolaos Chrysos, Christoforos Kachris, Dimitrios Soudris and Angelos Bilas. VineTalk: Simplifying software access and sharing of FPGAs in datacenters. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2017. doi: 10.23919/FPL.2017.8056788.

[47] Chris McClanahan. History and evolution of GPU architecture. A Paper Survey. http://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf, 2010, Accessed 2018.

[48] Aravind Menon, Alan L. Cox and Willy Zwaenepoel. Optimizing network virtualization in xen. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 2–2, Boston, MA, 2006. USENIX Association.

[49] Konstantinos Menychtas, Kai Shen and Michael L. Scott. Enabling OS Research by Inferring Interactions in the Black-box GPU Stack. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*,

USENIX ATC'13, pages 291–296, San Jose, CA, 2013. USENIX Association.

[50] Microsoft Azure. https://azure.microsoft.com/en-us/, Accessed 2018.

[51] Anastassios Nanos, Stefanos Gerangelos, Ioanna Alifieraki and Nectarios Koziris. V4VSockets: Low-overhead Intra-node Communication in Xen. In *Proceedings of the 5th International Workshop on Cloud Data and Platforms*, CloudDP '15, pages 1:1–1:6, Bordeaux, France, 2015. ACM. ISBN 978-1-4503-3478-5. doi: 10.1145/2744210.2744215.

[52] NetPIPE - Network Protocol Independent Performance Evaluator. http://linux.die.net/man/1/netpipe, Accessed 2018.

[53] NVIDIA CUDA. https://developer.nvidia.com/cuda-zone, Accessed 2018.

[54] NVIDIA CUDA Compiler. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html, Accessed 2018.

[55] NVIDIA CUDA samples. https://developer.nvidia.com/cuda-downloads, Accessed 2018.

[56] NVIDIA Multi-Process Service (MPS). https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, Accessed 2018.

[57] NVIDIA Tesla P100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, Accessed 2018.

[58] NVIDIA Tesla V100. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, Accessed 2018.

[59] Julio Proaño Orellana, Blanca Caminero, Carmen Carrión, Luis Tomas, Selome Kostentinos Tesfatsion and Johan Tordsson. FPGA-Aware Scheduling Strategies at Hypervisor Level in Cloud Environments. *Sci. Program.*, 2016:3–, June 2016. ISSN 1058-9244. doi: 10.1155/2016/4670271.

[60] Jason Jong Kyu Park, Yongjun Park and Scott Mahlke. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support*

for Programming Languages and Operating Systems, ASPLOS '15, pages 593–606, Istanbul, Turkey, 2015. ACM. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694346.

[61] Pathscale. pscnv driver. https://github.com/pathscale/pscnv, 2012, Accessed 2018.

[62] Antonio J. Peña, Carlos Reaño, Federico Silla, Rafael Mayo, Enrique S. Quintana-Ortí and Josẽ Duato. A complete and efficient CUDA-sharing solution for HPC clusters. *Parallel Computing*, 40(10):574 – 588, 2014. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/j.parco.2014.09.011.

[63] Sébastien Pinneterre, Spyros Chiotakis, Michele Paolino and Daniel Raho. vFPGAmanager: A Virtualization Framework for Orchestrated FPGA Accelerator Sharing in 5G Cloud Environments. In *2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–5, June 2018. doi: 10.1109/BMSB.2018.8436930.

[64] Prashanth Radhakrishnan and Kiran Srinivasan. MMNet: An Efficient Inter-VM Communication Mechanism. In *XenSummit*, Boston, 2008.

[65] Rezaur Rahman. Application Performance Tuning on Xeon Phi. In *Intel® Xeon Phi™ Coprocessor Architecture and Tools: The Guide for Application Developers*, pages 153–170. Apress, 2013. ISBN 978-1-4302-5927-5. doi: 10.1007/978-1-4302-5927-5_10.

[66] Michalis Rozis, Stefanos Gerangelos and Nectarios Koziris. YASMIN: Efficient Intra-node Communication Using Generic Sockets. In *12th Workshop on Virtualization in High-Performance Cloud Computing (VHPC'17), Frankfurt, Germany, June 18-22, 2017*, pages 617–628, 2017. ISBN 978-3-319-67629-6.

[67] Rusty Russell. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008. ISSN 0163-5980. doi: 10.1145/1400097.1400108.

[68] Erik Saule, Kamer Kaya and Ümit V. Çatalyürek. Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi. In *Proceed-*

ings of the 10th International Conference Parallel Processing and Applied Mathematics (PPAM), 2013.

[69] Lin Shi, Hao Chen, Jianhua Sun and Kenli Li. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Transactions on Computers*, 61(6):804–816, June 2012. ISSN 0018-9340. doi: 10.1109/TC.2011.112.

[70] Single Root I/O Virtualization and Sharing Specification Revision 1.1.

[71] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada and Kenji Kono. GPUvm: Why Not Virtualizing GPUs at the Hypervisor? In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 109–120, Philadelphia, PA, 2014. USENIX Association. ISBN 978-1-931971-10-2.

[72] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro and Mateo Valero. Enabling Preemptive Multiprogramming on GPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 193–204, Minneapolis, Minnesota, USA, 2014. IEEE Press. ISBN 978-1-4799-4394-4.

[73] Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh, Alberto Leon-Garcia and Paul Chow. Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 237–246, Monterey, California, USA, 2017. ACM. ISBN 978-1-4503-4354-1. doi: 10.1145/3020078.3021742.

[74] Naif Tarafdar, Nariman Eskandari, Varun Sharma, Charles Lo and Paul Chow. Galapagos: A Full Stack Approach to FPGA Integration in the Cloud. *IEEE Micro*, 38(6):18–24, Nov 2018. ISSN 0272-1732. doi: 10.1109/MM.2018.2877290.

[75] The Scalable Heterogeneous Computing Benchmark Suite (SHOC) for Intel® Xeon Phi™. https://software.intel.com/en-us/blogs/2013/03/20/the-scalable-heterogeneous-computing-benchmark-suite-shoc-for-intelr-xeon-phitm, Accessed 2018.

[76] Kun Tian, Yaozu Dong and David Cowperthwaite. A Full GPU Virtualization Solution with Mediated Pass-through. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 121–132, Philadelphia, PA, 2014. USENIX Association. ISBN 978-1-931971-10-2.

[77] Amazon GPU Instances. http://aws.amazon.com/ec2/instance-types/, Accessed 2018.

[78] NVIDIA GRID Virtual GPU Technology. http://www.nvidia.com/object/grid-technology.html, Accessed 2018.

[79] Using the Intel Xeon Phi Compute Accelerator with ESX 6.0. https://cto.vmware.com/using-intel-xeon-phi-esx-6-0, Accessed 2018.

[80] V4V implementation. http://lists.xen.org/archives/html/xen-devel/2013-05/msg02711.html, Accessed 2018.

[81] Dimitrios Vasilas, Stefanos Gerangelos and Nectarios Koziris. VGVM: Efficient GPU capabilities in virtual machines. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 637–644, July 2016. doi: 10.1109/HPCSim.2016.7568395.

[82] Jian Wang. Survey of State-of-the-art in Inter-VM Communication Mechanisms. 2018.

[83] Jian Wang, Kwame-Lante Wright and Kartik Gopalan. XenLoop: a transparent high performance inter-vm network loopback. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 109–118, Boston, MA, USA, 2008. ACM. ISBN 978-1-59593-997-5. doi: 10.1145/1383422.1383437.

[84] Bo Wu, Xu Liu, Xiaobo Zhou and Changjun Jiang. FLEP: Enabling Flexible and Efficient Preemption on GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 483–496, Xi'an, China, 2017. ACM. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037742.

[85] Pankaj Rohatgi Xiaolan Zhang, Suzanne McIntosh and John Linwood Griffin. XenSocket: A High-Throughput Interdomain Transport for Virtual Machines. In *International Conference on Middleware*, 2007.

[86] X.Org Foundation. Nouveau Wiki. http://nouveau.freedesktop.org/wiki/, 2011, Accessed 2018.

[87] Sadegh Yazdanshenas and Vaughn Betz. The Costs of Confidentiality in Virtualized FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–12, 2019. ISSN 1063-8210. doi: 10.1109/TVLSI.2019.2919644.

[88] Lamia Youseff, Dmitrii Zagorodnov and Rich Wolski. Inter-OS Communication on Highly Parallel Multi-Core Architectures, 2008.

[89] Hangchen Yu, Arthur M. Peters, Amogh Akshintala and Christopher J. Rossbach. Automatic Virtualization of Accelerators. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 58–65, Bertinoro, Italy, 2019. ACM. ISBN 978-1-4503-6727-1. doi: 10.1145/3317550.3321423.

[90] Jose Fernando Zazo, Sergio Lopez-Buedo, Yury Audzevich and Andrew W. Moore. A PCIe DMA engine to support the virtualization of 40 Gbps FPGA-accelerated network appliances. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, Dec 2015. doi: 10.1109/ReConFig.2015.7393334.

[91] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen and Thomas Moscibroda. The Feniks FPGA Operating System for Cloud Computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 22:1–22:7, Mumbai, India, 2017. ACM. ISBN 978-1-4503-5197-3. doi: 10.1145/3124680.3124743.

[92] Qian Zhao, Masahiro Iida and Toshinori Sueyoshi. A Study of FPGA Virtualization and Accelerator Scheduling. In *Proceedings of the First Workshop on Emerging Technologies for Software-defined and Reconfigurable Hardware-accelerated Cloud Datacenters*, ETCD'17, pages 3:1–3:4, Xi'an,

China, 2017. ACM. ISBN 978-1-4503-4923-9. doi: 10.1145/3129457.3129503.

[93] Qian Zhao, Motoki Amagasaki, Masahiro Iida, Morihiro Kuga and Toshinori Sueyoshi. Enabling FPGA-as-a-Service in the Cloud with hCODE Platform. *IEICE Transactions*, 101-D(2):335–343, 2018. doi: 10.1587/transinf.2017RCP0004.

Publications

- S. Gerangelos, G. Goumas, N. Koziris. Efficient accelerator sharing in virtualized environments: A Xeon Phi use-case. *Journal of Systems and Software,* 2019.

- K. Fertakis, S. Gerangelos, G. Goumas, N. Koziris. RACCEX: Towards Remote Accelerated Computing Environments. *The 10th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2018),* Nicosia, Cyprus, 10-13 December, 2018.

- A. Tsalapatis, S. Gerangelos, S. Psomadakis, K. Papazafeiropoulos, N. Koziris. utmem: Towards Memory Elasticity in Cloud Workloads. *Proceedings of the 13th Workshop on Virtualization in High-Performance Cloud Computing (VHPC'18),* held in conjunction with International Supercomputing Conference - High Performance (ISC 2018), Frankfurt, Germany, 24-28 June, 2018.

- M. Rozis, S. Gerangelos, N. Koziris. YASMIN: Efficient Intra-Node Communication Using Generic Sockets. *Proceedings of the 12th Workshop on Virtualization in High-Performance Cloud Computing (VHPC'17),* held in conjunction with International Supercomputing Conference - High Performance (ISC 2017), Frankfurt, Germany, 18-22 June, 2017.

- S. Gerangelos, N. Koziris. vPHI: Enabling Xeon Phi Capabilities in Virtual Machines. *Proceedings of the 2017 IEEE 31th International Parallel and Distributed Processing Symposium Workshops (IPDRM 2017),* Orlando, Florida, USA, 29 May-02 June, 2017.

- D. Vasilas, S. Gerangelos, N. Koziris. Efficient GPU Capabilities in Virtual Machines. *Proceedings of the 2016 International Conference on High Performance Computing & Simulation (HPCS 2016),* Innsbruck, Austria, 18-22 April, 2016.

- A. Nanos, S. Gerangelos, I. Alifieraki, N. Koziris. V4VSockets: low-overhead intra-node communication in Xen. *Proceedings of the 5th International*

Workshop on Cloud Data and Platforms (CloudDP '15), held in conjunction with EuroSys 2015, Bordeaux, France, 21-24 April, 2015.