# 論文 / 著書情報
# Article / Book Information

| | |
|---|---|
| 題目(和文) | |
| Title(English) | Performance Optimizations on a Many-core Processor |
| 著者(和文) | 林新華 |
| Author(English) | Lin Xinhua (James) |
| 出典(和文) | 学位:博士(理学),<br>学位授与機関:東京工業大学,<br>報告番号:乙第4151号,<br>授与年月日:2018年2月28日,<br>学位の種別:論文博士,<br>審査員:松岡 聡,渡辺 治,脇田 建,遠藤 敏夫,横田 理央 |
| Citation(English) | Degree:Doctor (Science),<br>Conferring organization: Tokyo Institute of Technology,<br>Report number:乙第4151号,<br>Conferred date:2018/2/28,<br>Degree Type:Thesis doctor,<br>Examiner:,,,, |
| 学位種別(和文) | 博士論文 |
| Type(English) | Doctoral Thesis |

# Performance Optimizations on a Many-core Processor

**Xinhua (James) Lin**

Supervisor: Prof. Satoshi Matsuoka

Department of Mathematical and Computing Science
Tokyo Institute of Technology

This dissertation is submitted for the degree of
*Doctor of Science*

December 2017

# Acknowledgements

First and foremost, I want to express my sincere gratitude to my advisor Prof. Satoshi Matsuoka, for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. To be competitive in a sophisticated game, such as academic research, without world-class mentorship is tough. I am so grateful for his top-notch guidance and cannot image having a better Ph.D advisor than him.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Osamu Watanabe, Prof. Ken Wakita, Prof. Toshio Endo, and Prof. Rio Yokota for their insightful comments and encouragement, but also for the insightful questions which incented me to widen my research from various perspectives.

I thank my associate advisor Prof. Akira Nukada for his continues advisor and help during my Ph.D study. I am also grateful to the following Matsuoka Laboratory staff: Rie Fukushima, Chisato Sato, Keiko Yoshida, and Reiko Yamamoto for their professional support and assistance in thesis submission, visa applications, and accommodation reservation. I thank the Matsuoka Laboratory members for their constant support throughout my stay in Japan: Jens, Arthur, Alex, Kevin, Xu, Jian, Haoyu, Bofang, Keisuke, Hoshino and others that I apologize for not mentioning, for their friendship.

In addition to my advisors and friends in Tokyo, I also would like to thank people in Shanghai. I greatly appreciate my superior Prof. Yizhong Gu, for his support me to apply the RONPAKU (Dissertation Ph.D) program and finish my Ph.D research work in recent five years. I also thank my Msc advisor Prof. Xinda Lu for setting me on the path of High Performance Computing (HPC) and encouraging me to pursue a Ph.D degree. I would like to thanks my staff in the HPC Center, Jianwen Wei, Minhua Wen, and Stephen Wang, for their support, especially they did an excellent job when I was in Tokyo.

I am happy to have many friends all over the world through my research work. I really enjoyed interaction with them. I would like to appreciate many suggestions and advice from: William Tang (Princeton), Naoya Maruyama (LLNL), Simon See (NVIDIA), Shuo Li (Intel), Stan Posey (NVIDIA), Puyong Wang (SSC), David Yuan (NVIDIA), Victor Lee (Intel), Hui Li (Intel), Ryan Sun (Intel). Xin Liu (WXSC), Bei Wang (Princeton), Alistair Rendell (ANU), Peter Strazdins (ANU), Filippo Spiga (ARM), Victor Eijkhout (TACC), Taisuke

Boku (U. Tsukuba), Kengo Nakajima (U.Tokyo), Sunita Chandrasekaran (Delware), Pavan Balaji (ANL), Ruud van der Pas (ORACLE), Wu Feng (VT), and Daniel S. Katz (UIUC).

I own great thanks to Japan Society for the Promotion of Science (JSPS) and China Scholarship Council (CSC) for providing me the RONPAKU Fellowship which allows me to carry out the full-time duty at Shanghai Jiao Tong University while at the same time study for my Ph.D degree at Tokyo Institute of Technology since 2013.

Last but not least, a special thanks to my family. Words cannot express how grateful I am to my mother, my mother-in-law, and father-in-law for their unwavering love and encouragements. Especially, I would like to dedicate this thesis to my beloved wife Dr. Ling Li for her supporting and encouraging me with love.

December, 2018

Xinhua (*James*) Lin

# Abstract

The landscape of computing is historically changing from single-core to many-core. Due to the end of Moore's Law in the recent decade, instead of improving the clock frequency, more cores are adding to a single chip, resulting in the emerging of a new breed of many-core processor architecture. Moreover, as the memory performance has been lagging behind the processor performance for a couple of decades, the increasing flops-to-byte ratio of many-core processors makes most application memory-bound. As a result, one promising solution to address this historic challenge is exploring on-chip data locality with an efficient inter-core communication. The inter-core communication has two basic schemas: the load/store and message-passing. The former, such as the cache-coherence protocol, is easy-to-program but less scalable; the latter, such as the register communication, is scalable but hard-to-program.

Despite of its strong scalability, the register communication brings the significant increasing programming challenge to many-core processors. Taking the China's home-grown 260-core SW26010 processor as an example, the register-level communication (RLC) adopted in SW26010 has two major constraints. First, RLC can only support row/column-based communications, which requires designing algorithms to only communicate among the cores in the same row/column. Second, RLC uses an anonymous producer-consumer protocol, which requires orchestrating message sequences manually so as to ensure a correct sending and receiving order.

This study, divided in three steps, aims to tackle the key programming challenges of the register communication. Taking the SW26010 processor as a research case, we first identified the programming challenges through a comprehensive evaluation of the processor, and then developed a systematic optimization solution. The research findings are envisioned to make a breakthrough in the performance optimizations and to inform researchers who face the same challenges.

The purpose of the first step in this study was to illuminate the uncharted area of the SW26010 processor. The inadequate public information of this processor's micro-architecture prevents global researchers from improving application performance on the TaihuLight supercomputer. To address this issue, we developed the micro-benchmark suite *swCandle*, mostly written in assemble language, to evaluate the key micro-architectural features of the

SW26010. The benchmark revealed some unanticipated findings on the processor micro-architecture beyond the publicly available data. For instance, the broadcast mode of RLC has the same latency as the peer-to-peer mode. According to this finding, we speculated that the broadcast mode might be implemented by default while the P2P mode might be implemented as a special case of the broadcast mode; the implementations could be similar to the mask operations in vector processing. These findings provide important information for performance optimizations in the following two steps.

Based on the findings reveled in the first step, we conducted the second one in which we optimized two compute-bound kernels. The first kernel is the direct N-body simulation. Due to the lack of efficient hardware support, the reciprocal square root (*rsqrt*) operations turned out to be the performance bottleneck of N-body on the SW26010. We applied the computation-oriented optimizations and achieved about 25% efficiency in a single core-group of the SW26010. The second kernel is double-precision general matrix-multiplication (DGEMM). We designed a novel algorithm for RLC and applied several on-chip communication-oriented optimizations. These endeavors improved the efficiency to up to 88.7% in a single core-group of the SW26010.

In contrast to the compute-bound kernels, due to the limited memory bandwidth of the SW26010, the single memory-bound kernel – such as Sparse matrix-vector multiplication (SpMV – cannot perform well on the processor, despite comprehensive optimizations. However, we anticipated, the overall performance of an algorithm can be effectively improved by overlapping multiple memory-bound kernels within the algorithm, and, thus, can provide a promising optimization approach for those multi-kernel memory-bound algorithms to achieve better performance on the SW26010. The aim of the third step in this study is to optimize the memory-bound algorithm Preconditioned Conjugate Gradient (PCG). First, in order to minimize the all_reduce communication cost of PCG, we developed a new algorithm RNPCG, a non-blocking PCG leveraging the on-chip register communication. Second, we optimized three key kernels of the PCG, including proposing a localized version of the Diagonal-based Incomplete Cholesky (LDIC) preconditioner. Third, to scale the RNPCG on TaihuLight, we designed the three-level non-blocking all_reduce operations. With these three steps, we implemented the RNPCG in the computational fluid dynamics software OpenFOAM. The experimental results on TaihuLight show that 1) compared with the default implementations of OpenFOAM, the RNPCG and the LDIC on a single-core group of SW26010 can achieve a maximum speedup of 8.9X and 3.1X, respectively; 2) the scalable RNPCG can outperform the standard PCG both in the strong and the weak scaling up to 66,560 cores.

# Table of contents

**List of figures** **xi**

**List of tables** **xv**

**1 Introduction** **1**
1.1 Motivation . . . 1
1.1.1 Many-core Processor Shift . . . 1
1.1.2 Inter-core Communication . . . 4
1.2 Problem Statement . . . 6
1.3 Proposal . . . 7
1.4 Contributions . . . 8
1.5 Outline . . . 9

**2 Background and Related Work** **11**
2.1 Data Motions in Many-core Processors . . . 11
2.1.1 Vertical and Horizontal Data Motions . . . 11
2.1.2 NVIDIA V100 GPU . . . 13
2.1.3 Intel Knights Landing . . . 13
2.1.4 Intel SCC Chip . . . 13
2.1.5 STI CELL . . . 14
2.1.6 Sunway SW26010 . . . 14
2.2 On-chip Network . . . 15
2.2.1 Inter-core Communication Shift . . . 15
2.2.2 Topology . . . 15
2.2.3 Routing . . . 16
2.2.4 Switching . . . 17
2.3 Register-communication Processors . . . 18
2.3.1 The Multi-ALU Processor . . . 18

2.3.2 The TILE64 Processors . . . 18
2.3.3 A 16-core Embedded Academic Processor . . . 19
2.3.4 The SW26010 Processor . . . 19
2.4 Related Work . . . 20
2.4.1 Benchmarking Many-core Processors . . . 20
2.4.2 Compute-bound Kernels Optimizations . . . 21
2.4.3 PCG Optimizations . . . 21
2.5 Summary . . . 22

**3 Evaluations with Micro-benchmarks 23**
3.1 The CPE Pipeline . . . 23
3.1.1 Instruction Latencies . . . 24
3.1.2 Instruction Issue Orders . . . 24
3.2 Register Communications . . . 26
3.2.1 P2P RLC Latencies . . . 26
3.2.2 Broadcast RLC Latencies . . . 30
3.2.3 RLC Bandwidths . . . 31
3.2.4 Routing Modes . . . 32
3.3 Memories . . . 35
3.3.1 On-chip SPM . . . 35
3.3.2 Off-chip Memory . . . 36
3.4 Implications for Performance Optimizations . . . 39
3.4.1 Applying the Roofline Model to the SW26010 . . . 39
3.4.2 Case study 1: Instruction Scheduling . . . 39
3.4.3 Case study 2: Designing the Reduction Mode of RLC . . . 41
3.5 Summary . . . 42

**4 Optimizing Compute-bound Kernels 45**
4.1 Direct N-Body Simulation . . . 45
4.1.1 Basic Algorithm . . . 45
4.1.2 Initial Solution . . . 46
4.1.3 Model-guided Optimizations . . . 47
4.1.4 Results and Analysis . . . 49
4.2 DGEMM . . . 51
4.2.1 Basic Algorithm . . . 51
4.2.2 Initial Solution . . . 52
4.2.3 RLC-friendly Algorithm . . . 52

4.2.4 Results . . . 57
4.3 Summary . . . 58

**5 Optimizing the Memory-bound PCG 59**
5.1 PCG Optimizations . . . 59
5.1.1 Basic Algorithm . . . 59
5.1.2 Initial Solution . . . 60
5.1.3 RLC-friendly Algorithm . . . 61
5.1.4 Optimizations on the Key Kernels . . . 64
5.2 Scaling RNPCG on TaihuLight . . . 67
5.2.1 Introduction to TaihuLight . . . 67
5.2.2 The Scalable RNPCG . . . 68
5.3 Implementing RNPCG in OpenFOAM . . . 69
5.3.1 Introduction to OpenFOAM . . . 69
5.3.2 Compiling the Mixed C/C++ Code . . . 71
5.3.3 Linking Static Libraries . . . 72
5.3.4 Scaling Standard Solvers . . . 72
5.4 Performance Evaluation . . . 72
5.4.1 Experimental Setup . . . 72
5.4.2 Evaluation Methods and Results . . . 74
5.5 Summary . . . 77

**6 Discussions 79**
6.1 A Performance Guideline for the SW26010 . . . 79
6.1.1 CPE Level . . . 79
6.1.2 Inter-CPE Level . . . 80
6.1.3 Core-group Level . . . 81
6.2 The Fall of the CELL and the Rise of the SW26010 . . . 81
6.2.1 Companions of the Two Processors . . . 81
6.2.2 The Reasons Caused the CELL Failing . . . 83
6.2.3 The Reasons caused the SW26010 Rising . . . 83
6.3 Implications For the Next Generation SW Processor . . . 84
6.3.1 Design Goal . . . 85
6.3.2 Design Details . . . 85

**7 Conclusion and Future Work 89**
7.1 Thesis Statement Revisited . . . 89

7.2 Summary of Research Contributions . . . . . 91
7.3 Future Work . . . . . 91

**References** **93**

**Appendix A Peer-reviewed Publications** **105**
A.1 International Conferences or Journal Articles . . . . . 105
A.2 International Workshops or Domestic Conferences . . . . . 106

# List of figures


1.1 The generic architecture of many-core processors. Lightweight cores are connected together with an on-chip network. The horizontal data motion refers to moving data between cores; the vertical data motion refers to moving data between the main memory and cores. . . . . 3
1.2 The SW26010 processor can support register communication among the 64 CPEs of a single core-group over a 2D on-chip mesh network. . . . . 6

2.1 An illustration of the two data motions in the memory hierarchy . . . . . 12
2.2 At the register level, each CPE can communicate with the remaining 7 CPEs in the same row or column via the 2D on-chip mesh network. . . . . 19

3.1 The absolute and pipelined latencies of the instructions. . . . . 25
3.2 The instruction issue order in between the in-order dual-issue pipeline of CPEs. (a) the micro-benchmark written in assembly language; (b) if the issue order is in-order, the latency of a single iteration of (a) would be *26 cycles*; (c) if the issue order is out-of-order, the latency would be *21 cycles*. . . . . 25
3.3 RLC uses the three local buffers of each CPE and the 2D on-chip network to transfer register data among 64 CPEs of a single core-group. . . . . 26
3.4 Measuring the latency of a single P2P RLC. (a) an illustration of the CPE(n,0) sending its register data to the CPE(n,j); (b) a micro-benchmark to measure the latency. . . . . 27
3.5 Measuring the latency of a round-trip P2P RLC (a) an illustration of the *ping-pong* test for the CPE(n,i) and CPE(n,j); (b) a micro-benchmark to measure the latency. . . . . 28
3.6 Measuring the latency of RLC RECEIVE instructions. (a) an illustration of sending multiple RLCs from CPE(n,i) to CPE(n,j); (b) a micro-benchmark to measure the latency. . . . . 29

3.7 Measuring the latency of the broadcast RLC. (a) an illustration of the CPE(n,0) broadcasting its register data to the remaining 7 CPEs in the same row *n*; (b) a micro-benchmark to measure the latency. . . . . 30
3.8 Measuring the latency of consecutive RLCs. (a) an illustration of sending consecutive RLCs from CPE(n,i) to CPE(n,j); (b) a micro-benchmark to measure the latency. . . . . 31
3.9 The static routing mode of RLC. (a) an illustration of a 3-hop static routing; (b) a micro-benchmark to measure the latency of the 3-hop static routing. . 33
3.10 The dynamic routing mode of RLC. (a) an illustration of a dynamic routing. (b) a micro-benchmark to measure the latency of the dynamic routing. . . . 34
3.11 The relative latency rate of the dynamic routing mode. The baseline is the latency when 100% hit to default branch A. . . . . 35
3.12 The RANK mode of DMA can gather/scatter data blocks in the row-major order from/to the 8 CPEs in the same row. . . . . 37
3.13 The STREAM bandwidth results for the DMA PE mode. (a) the maximum Copy memory bandwidth is 27.9 GB/s; (b) the maximum Scale memory bandwidth is 24.1 GB/s; (c) the maximum Add memory bandwidth is 23.4 GB/s; (d) the maximum Triad memory bandwidth is 22.6 GB/s. . . . . 37
3.14 Bandwidth comparison of the DMA *Rank* mode and the DMA *PE* mode. . 38
3.15 An illustration of applying the roofline model on the SW26010 processor. As can seen, using the RLC to explore on-chip data locality can significantly reduce the processor's arithmetic-density from 33.84 Flops/bytes to only 1.2 Flops/bytes. . . . . 40
3.16 Designing a row/column-based reduction operation with the logP model. (a) the optimal reduction tree for $l$=8, $g$=7, $o$=1, and $p$=8. (b) the activity of each CPE over time. The total latency is *51 cycles*. . . . . 41
3.17 The summary of the key findings revealed by the *swCandle* benchmark suite. 42

4.1 The software *rsqrt* routine . . . . . 49
4.2 Performance impact of the optimizations for N-body . . . . . 51
4.3 The pseudocode of the initial implementation for DGEMM . . . . . 52
4.4 Visual presentation of the RLC-friendly algorithm for DGEMM . . . . . 53
4.5 Using RLC to compute the register kernels of the 64 CPEs in total 8 rounds 54
4.6 The pseudocode of DMA double-buffering . . . . . 56

5.1 Scheduling instructions to overlap the RLC_allreduce with the SpMV kernel. (a) the pseudo assembly code for the RLC_allreduce on the CPE(4,0); (b) the pseudo assembly code for the SpMV kernel. . . . . 64
5.2 An illustration of LDU and Sliced ELLPACK (strip size is 2) formats for a sparse matrix. . . . . 67
5.3 The breakdown of the performance impact of RNPCG optimizations on a single core-group. . . . . 75
5.4 Strong Scaling of Scalable RNPCG on TaihuLight . . . . . 77
5.5 Weak Scaling of Scalable RNPCG on TaihuLight . . . . . 77

6.1 Two design options of the next generation Sunway processor . . . . . 86

# List of tables


2.1 The approaches for data motions in many-core processors . . . . . 12

3.1 The results of P2P RLC latencies . . . . . 28
3.2 The latencies and gaps of a single iteration of consecutive RLCs. . . . . 32

4.1 The semi-empirical performance model for N-body . . . . . 48
4.2 Performances and efficiencies of DGEMM on the SW26010 . . . . . 57

5.1 Comparison of the two compilers available on TaihuLight . . . . . 71
5.2 Specifications of the two test nodes. We chose the Intel E5-2695v3 CPU to compare with the SW26010, as the two processors were released in the same year. . . . . 73
5.3 Breakdown latencies of the RLC_allreduce and the DMA_allreduce on the single core-group. . . . . 74
5.4 Evaluating the three preconditioners with two test cases on a single core-group. Our LDIC preconditioner is the fastest in the solution time for solving a linear system. . . . . 76

6.1 Comparisons of the CELL processor and the SW26010 processor . . . . . 82
6.2 Real-world applications developed on TaihuLight . . . . . 82
6.3 RLC vs. DMA for the all_reduce operation in terms of latencies on 64 CPEs of the SW26010 processor . . . . . 84

# Chapter 1

# Introduction

This chapter provides the overview of this thesis. The motivation of this research is presented in **Section 1.1**. Due to the end of Moore's Law in the recent decade, instead of improving the clock frequency, more cores are adding to a single chip, resulting in the emerging of a new breed of many-core processor architecture. As the memory performance has been lagging behind the processor performance for a couple of decades, the increasing flops-to-byte ratio of many-core processors makes most application memory-bound. One promising solution to address this challenge is exploring on-chip data locality with an efficient inter-core communication. The inter-core communication has two basic schemas: the load/store and message-passing. The former is easy-to-program but less scalability; the latter is scalable but hard-to-program. **Section 1.2** discusses the three key programming challenges of the message-passing schemas of the inter-core communication. Taking the SW26010 processor as a study case, **Section 1.3** illustrates our approaches to solve these programming challenges in the three steps. **Section 1.4** lists the key contributions of these three steps, respectively. This chapter ends with a thesis outline in **Section 1.5**.

## 1.1 Motivation

### 1.1.1 Many-core Processor Shift

#### Many-core Processor History

For the past four decades, Moore's Law [1] [2] dominates the semiconductor industry. It predicted the trend that the transistors density could double approximately every two years and the processors clock speed could doubled every 18-24 months. Furthermore, other key characteristics of processors such as operating voltage, switching speed, and energy

efficiency could continue improving. In a word, Moore's Law was the synchronizing force for all levels of the semiconductor industry.

However, in the last decade this trend has slowed. The continuing scaling process of the past four decades has reached physical limits in the processors clock speed and power consumption [3]. Especially, the end of voltage scaling [4] poses limits on frequency scaling, and highlights the power as the dominant limiting factor. As a result, the clock speed of high-end processors has been flat at less than 4GHz for several years. To mitigate the growing crisis of power consumption, the traditional doubling of clock speeds every processor generation has been replaced by a doubling of cores. A new breed of many-core processor has thus emerged.

Many-core processors trade the single core performance with many lightweight cores. These lightweight cores can achieve higher overall throughput under any specific power budgets. Both industry experts [5] and academia [6] agreed that this many-core design is the only practical approach to continue increasing processor performance. In 2007, Intel released the first many-core processor in the world, Teraflops Research Chip [7] [8]. The many-core chip consists of 80 lightweight cores on a single chip, and achieves 1.81 TeraFlops of peak performance within 265 Watts of power consumption. This proof-of-concept chip was a wake-up call to major processor venders to adopt the many-core architecture as the de facto industry standard. Since then, a amount of many-core processors were brought to market, e.g., NVIDIA V100 GPUs, Intel Knights Corner (KNC) and Knight Landing (KNL).

### Many-core Processor Architecture

A many-core processor consists of a mount of lightweight cores on a single chip, as shown in Fig. 1.1. Each core employs Single Instruction Multiple Data (SIMD), which enables the same instruction to operate on multiple pieces of data at the same time. Additionally, these cores are connected together by using on-chip network, which allows communications between these cores. These cores also share the same DRAM memory controller, and thus provides a shared memory space abstraction.

This many-core architectural design has two major benefits. First, The many-core processors use the simple, lightweight cores, instead of conventional, complex cores, for better performance efficiency. Second, these simple cores can be reused to various market segments with low cost. For example, NVIDIA reuses the same GPU core design across its four major products segments: the embedded processor Tegra, the consumer processor Geforce, the professional graphic processor Quadro, and the high-end compute processor Tesla.

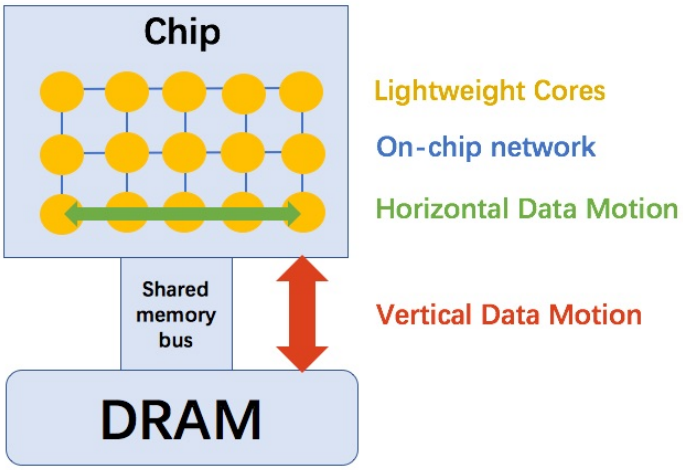



Fig. 1.1 The generic architecture of many-core processors. Lightweight cores are connected together with an on-chip network. The horizontal data motion refers to moving data between cores; the vertical data motion refers to moving data between the main memory and cores.

### Many-core Processor Challenge and Solutions

Due to these two major benefits, the many-core architecture is considered as the favorable design choice for the foreseeable future. With the growing number of cores, the many-core processor demands high memory bandwidth to perform well. Unfortunately, memory bandwidth has been lagging processor performance for a couple of decades. No significant improvements have been made in memory bandwidth, which partially resulted from the physical limitation of power consumption. For example, putting 10 TeraFlops on a single chip only requires about 100 Watts, however, to supply those floating point units at a modest memory bandwidth to floating point ratio of 0.2, the memory require about 2 KW of power consumption, which is unachievable on a single node. This is known as *Memory Wall* [6]. The memory wall results in the growing flops-to-bytes ratio of many-core processors.

The increasing flops-to-bytes ratio makes data motions as the most daunting challenge for many-core processors [9]. There are two types of data motions (See Fig. 1.1) in many-core processors: *horizontal* and *vertical*. The former refers to moving data at the same level, while the latter refers to moving data across different levels, e.g., moving data between the off-chip memory and the on-chip memory of cores. The vertical data motion is much more costly than the computing flops due to the high flops-to-bytes ratio.

Attempts to solve this vertical data motion challenge could be mostly categorized to two directions: 1) using the emerging technologies to increase memory bandwidths; 2) exploring on-chip data locality to minimize vertical data motions. One example of the first direction is

3D memory packing, which packages the memory on top of the processor cores, and thus allows each core to have higher bandwidth access to the cache banks directly stacked on top of it. The second direction, on the other hand, suggests the increasing value of an efficient on-chip inter-core communications for horizontal data motions.

### 1.1.2 Inter-core Communication

The inter-core communication becomes increasingly important for many-core processors due to the limited off-chip memory bandwidth. According to different communication schemas, the inter-core communication can be classified into two types: the load/store schema and the message-passing schema. In the former, data is put to shared on-chip caches by one core and retrieved by other cores later. While in the latter, data is sent directly from one core to another.

#### Load/Store schema

The most typical load/store schema is cache-coherence, which is a mechanism to ensure the consistency of data stored in local caches of a shared resource. Two protocols can be used to implement the cache-coherence, namely the snooping and the directory-based.

**Snooping Protocol:** The snooping protocol is used with a shared bus. With this shared bus, cores can read shared data without any coherence problems; in contract, in order to write, cores require the exclusive access to the bus. Due to its limited scalability of the shared bus, the snooping protocol cannot scale well thus has never been used in many-core processors.

**Directory-based Protocol:** The directory-based protocol provides more scalability than the snooping protocol. The shared data is placed in a common directory that maintains the coherence between caches. Cores must ask permission from the directory to load shared data from main memory to its local cache. Once the shared data is modified, the directory sends either updates or invalidations to other cores those have copies of the shared data.

Although it has more scalability, the directory-based protocol still fails to provide sufficient scalability to many-core processors. This is because the protocol may be able to scale to dozens of cores, but the data movement overhead would become high. The overhead per core grows with the core count and eventually exceeds the value of adding more cores [10]. For example, Intel Knights Corner (KNC) has 64 cores which are connected with a bidirectional ring bus. Each core has a private 512 kb L2 unified cache which is kept coherent by a distributed tag directory system (DTDs). The cache-coherence protocol on KNC is implemented with an extended MESI protocol [11] [12]. The performance evaluation and

modeling [13] showes the direct-based cache-coherence protocol provided less performance and predictability than the message-passing schema.

### Message-passing Schema

The message-passing schema can provide more scalability than Load/Store schema. Two approaches can be used to implement the message-passing schema, namely the direct memory access (DMA) and register communication.

**DMA:** DMA was used to enable a peripheral device control a processor's memory bus directly. In a DMA transfer, a DMA controller requests a permit from a CPU, then transfers data directly between the peripheral and the main memory. Despite normally for vertical data motions between peripherals and main memories, DMA can be also used for the horizontal data motions between cores on a single chip. One example is the STI CELL processor [14] [15], which consists of a PowerPC element (PPE) and 8 synergistic processing elements (SPEs). SPEs use DMA for both vertical data motions (main memory-to-local memory communications) and horizontal data motions (SPE-to-SPE communications). However, due to its high overhead, using DMA for the SPE-to-SPE communication was proved to be the key performance bottleneck of CELL [16].

**Register Communication:** Another, and more promising, message-passing approach is register communication. An example is the Sunway SW26010 processor, which was used to build the TaihuLight supercomputer [17] [18]. The SW26010 is composed of 4 core-groups, connected via an on-chip network, each of which includes a Management Processing Element (MPE) and 64 Computing Processing Elements (CPEs) arranged in an $8 \times 8$ mesh. As shown in Fig.1.2, register communication uses the three local buffers in each CPE and the 2D on-chip network to transfer register data. To send, the CPE puts the register data (with the `putr` instruction for the row network and `putc` for the column network) into the unified sender buffer. The unified sender buffer connects to both the row and column networks and can hold up to 6 vector elements. Notably, if the unified sender buffer is full, the `putr` and `putc` instructions will be blocked. To receive, the CPE moves the register data into the vectors register (with the `getr` instruction from the row receiver buffer and `getc` from the column receiver buffer). Each receiver buffer can only connect to the network in the same direction and hold up to 4 vector elements.

In addition to the SW26010, other processors using register communication are the Multi-ALU Processor (MAP) processor [19], the TILE64 processor [20], the TILE-GX processor [21], and the NI-VISA embedded processors [22].

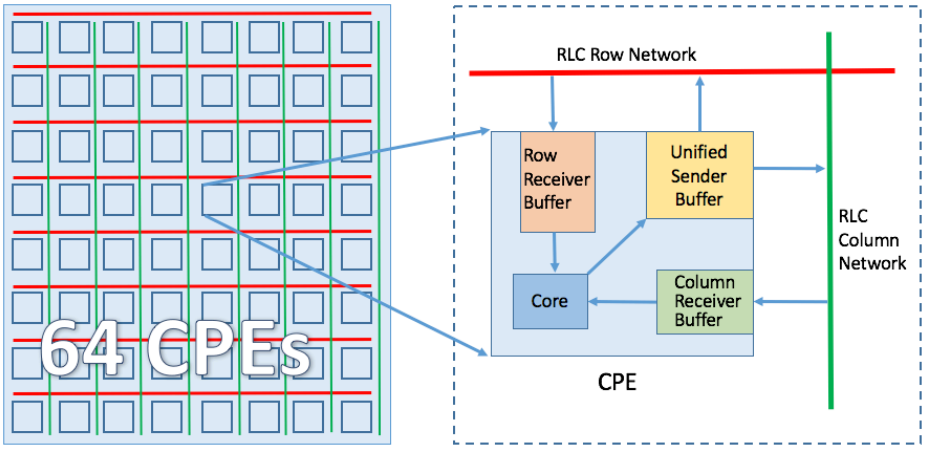



Fig. 1.2 The SW26010 processor can support register communication among the 64 CPEs of a single core-group over a 2D on-chip mesh network.

## 1.2 Problem Statement

The evolution of many-core processor was recently at an inflection point where the inter-core communication become more important. The register communication is one of the promising inter-core communications for future many-core processors. This inflection point of using register communications on many-core processors has caused a disruption in the performance optimizations. The conventional performance optimizations are no longer sufficient to achieve the ultimate performance on many-core processors. Applications and algorithms need to change – exploring on-chip data locality – in order to efficiently utilize the register communication among cores.

We discuss the key performance optimization challenges caused by the register communications in three perspectives. First, to reduce the complexity of routing and switching, the register commutations usually only provide limited communication patterns. For example, the SW26010 cannot support all-to-all communication among its 64 cores. Instead, the processor only allows communication among the 8 CPEs in the same row/column with two communication modes, the peer-to-peer and row/column-based broadcast. Another example is the TILE64 processor [20] [23]. The register communication on this processor can only support communications between two adjacent tiles. Second, the register communications use an anonymous producer-consumer protocol, which renders the RLC message “contains no ID tag". Addressing this constraint requires orchestrating the message sequences manually, so as to ensure the correct sending and receiving order. Third, most of the register communications lack support of efficient programming language and compiler. For example, the SW26010 provides no high-level language, such as C++/C/FORTRAN, support for the

register communication. The only programming approach for the register communication is assemble language.

Yet, these three key programming challenges of register communication have not been well addressed. The prior studies [18] [19] [20] [21] focused on the implementation of the register communications from computer architects' perspective, instead of programmers' perspective.

## 1.3 Proposal

This thesis aims to tackle the key programming challenges of the register communications. Taking the SW26010 processor as a research vehicle, we identified the key programming challenges through a comprehensive evaluation of the processor, and then addressed the challenges by developing a systematic optimization solution. The research findings are envisioned to make a breakthrough in the performance optimizations and to inform researchers who face the same challenges. We conducted our research in the following three steps:

First, the inadequate public information of the China's home-grown SW26010 processor's micro-architecture prevents global researchers from improving the performance of applications on the TaihuLight supercomputer. The purpose of the first study was to to illuminate the uncharted area of the SW26010 processor in order to provide important information for performance optimizations. First, we developed a micro-benchmark suite, mostly written in assemble language, to evaluate the key micro-architectural features of the SW26010 processor. The benchmark revealed some unanticipated findings beyond the publicly available data. For instance, the instruction issue order in between the in-order dual-issue pipeline is out-of-order; the broadcast mode of register communications has the same latency as the peer-to-peer mode. Second, we applied the roofline model, with the key parameters that we obtained from measuring the processors using the benchmark suite, to identify the key programming challenge of the SW26010 processor. Third, the methodology we developed in this study, that infers a processor's micro-architecture designs from benchmark results, can also be applied on other processors lacking of public information.

Second, based on the findings from the first step, we conducted the second one in which we adopted two compute-bound kernels to identify the potential programming challenges. The first kernel that we used is the direct N-body simulation. Due to the lack of efficient hardware support, the reciprocal square root (*rsqrt*) operations turned out to be the performance bottleneck of N-body on the SW26010. Guided by a semi-empirical performance model developed by us, we applied the computation-oriented optimizations including strength reduction and instruction mix. The optimizations achieved about 25% efficiency in one core

group of the SW26010. The second kernel is double-precision general matrix-multiplication (DGEMM). The initial implementation of DGEMM had much lower arithmetic intensity than the SW26010. We thus designed a novel algorithm for RLC to reuse the data that already reside in 64 CPEs, and applied several communication-oriented optimizations. These endeavors improved the efficiency to up to 88.7% in one core group of the SW26010.

Third, in contrast to the compute-bound kernels, due to the limited memory bandwidth of the SW26010, the single memory-bound kernel – such as Sparse matrix-vector multiplication (SpMV – cannot perform well on the processor, despite comprehensive optimizations. However, we anticipated, the overall performance of an algorithm can be effectively improved by overlapping multiple memory-bound kernels within the algorithm, and, thus, can provide a promising optimization approach for those multi-kernel memory-bound algorithms to achieve better performance on the SW26010. The aim of the third step in this study is to optimize the memory-bound algorithm Preconditioned Conjugate Gradient (PCG). First, in order to minimize the all_reduce communication cost of PCG, we developed a new algorithm RNPCG, a non-blocking PCG leveraging the on-chip register communication. Second, we optimized three key kernels of the PCG, including proposing a localized version of the Diagonal-based Incomplete Cholesky (LDIC) preconditioner. Third, to scale the RNPCG on TaihuLight, we designed the three-level non-blocking all_reduce operations. With these three steps, we implemented the RNPCG in the computational fluid dynamics software OpenFOAM. The experimental results on TaihuLight show that 1) compared with the default implementations of OpenFOAM, the RNPCG and the LDIC on a single-core group of SW26010 can achieve a maximum speedup of 8.9X and 3.1X, respectively; 2) the scalable RNPCG can outperform the standard PCG both in the strong and the weak scaling up to 66,560 cores.

## 1.4 Contributions

This thesis consists of three main work: the first work is to benchmark the SW26010 processor; the second and third are to optimize two compute-bound kernels and a memory-bound algorithm, respectively. In concise, the three main work have the following key contributions:

- The first work has three key contributions. 1) We developed a micro-benchmarks suite in assembly language for the SW26010 processor. To our knowledge, this is the first benchmark that reveals the micro-architectural features of this processor. 2) The benchmark suite quantified the key micro-architectural features of the SW26010 processor, and revealed some unanticipated findings beyond the publicly available data. For instance, the instruction issue order in between the in-order dual-issue pipeline is

out-of-order; the broadcast mode of register communications has the same latency as the peer-to-peer mode. 3) We applied the roofline model [24], with the key parameters that we obtained from measuring the processors using the benchmark suite, to identify the key programming challenges of the SW26010 processor.

- The second work has two key contributions. 1) We found that *rsqrt* is the bottleneck that hampers the performance of the N-body kernel, and we thus optimized the *rsqrt* with a software routine. 2) We designed RLC-friendly algorithms for DGEMM and implemented in the assembly with manual instruction scheduling to achieve about 90% efficiency on the single core-group of SW26010.

- The third work has three key contributions. 1) We developed a non-blocking PCG solver with the speedy on-chip register communication for the SW26010. It is noteworthy that our RNPCG is independent from the in-house implementation that was used in TaihuLight's HPCG test [25]. To our knowledge, this is the first paper that systematically reports PCG optimizations on TaihuLight. 2) We proposed a localized version of the DIC preconditioner on the SW26010. Although it requires more iterations to convergence, we proved, the local DIC preconditioner can outperforms the standard DIC preconditioners on a single-core group of the SW26010. 3) We are the first to scale up the PCG solver of OpenFOAM on TaihuLight up to 66,560 cores.

## 1.5 Outline

This thesis is organized into 3 parts and 7 chapters. *Part I* encompasses the first two chapters, providing a general introduction of this research. **Chapter 1** briefly summarizes the motivation, research problems, our approach, and key contributions of this study. **Chapter 2**illustrates the context of this research by presenting the background knowledge of data motions schemas, on-chip network, register communications. In addition, this chapter conducts a critical review of the research on benchmarking and performance optimizations on many-core processors.

*Part II* presents the three main work of this study in three chapters. The first work in **Chapter 3** evaluates the key architectural features of SW26010 processor with a micro-benchmark suite. The second work in **Chapter 4** optimizes the two compute-bound kernels, namely N-body and DGEMM. The third work in **Chapter 5** optimizes the memory-bound PCG and implements the optimized PCG solver in OpenFOAM.

*Part III* provides a systematic discussion on the research findings. **Chapter 6** proposes a performance guideline for the SW26010 processor based on the benchmark results and

optimization experience gained in the three main work. In addition, this chapter compares the STI CELL processor and the SW26010 processor, and discuss the implications for the next generation of Sunway processor. This thesis is concluded in **Chapter 7** with a possible agenda of future research.

# Chapter 2

# Background and Related Work

This chapter aims to provide a broad context of the research presented in this thesis. In **Section 2.1**, we first discuss the two data motion schemas in many-core processors: the vertical data motion and horizontal data motion. Then we compare the different data motion approaches used in five many-core processors. In **Section 2.2**, we introduce three essential basic characteristics of on-chip networks: topology, routing, and switching. In **Section 2.3**, we introduce four processors using register communications. In **Section 2.4**, we compare our three major work in this thesis with prior work.

## 2.1 Data Motions in Many-core Processors

### 2.1.1 Vertical and Horizontal Data Motions

Memory hierarchy is important to a computer system. Data should be moved from the slower (far) memory up to the faster (near) memory. Figure 2.1 shows the three levels of the memory hierarchy: register (fastest), local memory of processor cores (on-chip, fast), and the main memory (slow). There are two kinds of data motions in the memory hierarchy: the vertical and the horizontal data motions. The former refers to move data across different levels; the latter refers to the move data at the same level.

We compared the five typical many-core processor (Tab. 2.1) in terms of the two data motions. To move data vertically, due to using caches, both the Intel KNL and Intel SSC chip can move data implicitly from the main memory to on-chip caches. Caches can virtualize the notion of on-chip and off-chip memory, and are therefore invisible to current programming models. However, the cost of moving data off-chip is substantial. As a result, the other three processors adopt a software management memory, scratchpad memory (SPM) [26], and explicitly move the data on/off chip. For instance, both the STI CELL processor and the

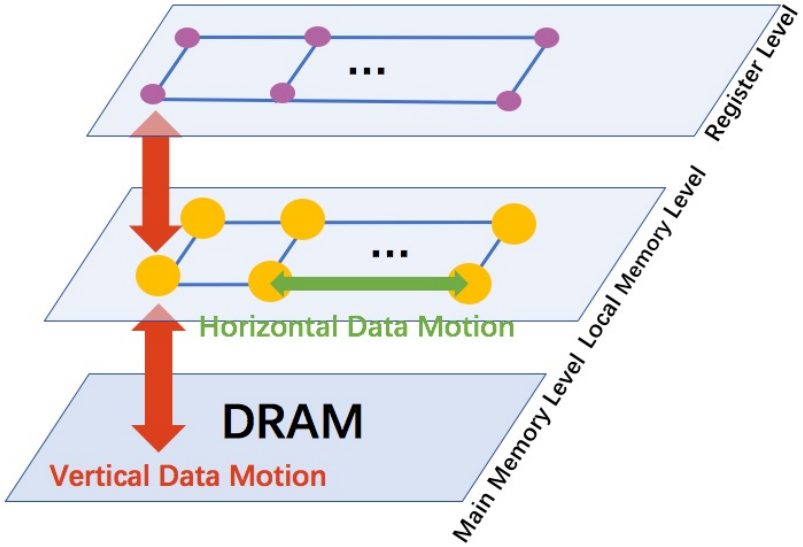



Fig. 2.1 An illustration of the two data motions in the memory hierarchy

SW26010 adopt DMA to transfer data between main memory and local memory of compute cores.

To compensate for the limited off-chip memory bandwidth, many-core processors share on-chip data among cores to avoid costly vertical data motions. The five many-core processors use two schemas to communicate among cores: the load/store and message-passing. The NVIDIA GPUs use the shared memory to exchange data among the streaming multi-processors. The Intel Knights Landing (KNL) adopts a shared L2 cache for the two cores in the same tile. Both of them use the load/store schema which requires data to be put somewhere by one core first and retrieved by other cores later. In contrast, the rest of the three processors adopt the message-passing communication schema. The Intel SCC Chip uses the message passing to send cache data. The STI CELL uses DMA to transfer data between core memories. The SW26010 uses a light-weight register-register communication to share register data among its 64 CPEs of a single core-group. In the following, we will

Table 2.1 The approaches for data motions in many-core processors

| Processors | Vertical Data Motions | Horizontal Data Motions |
|---|---|---|
| NVIDIA GPUs | explicit | load/store (shared memory) |
| Intel KNL | implicit | load/store (cache coherence) |
| Intel SCC Chip | implicit | message-passing |
| STI CELL | explicit (DMA) | message-passing (DMA) |
| SW26010 | explicit (DMA) | message-passing (RLC) |

illustrate the architecture and data motion schemas of these five many-core processors in details.

### 2.1.2 NVIDIA V100 GPU

NVIDIA V100 GPU [27] consists of 6 graphics processing clusters, 42 texture processing clusters, 84 Volta streaming multiprocessors (SMs). Each SM has 64 FP32 Cores, 64 INT32 Cores, 32 FP64 Cores, and 8 new Tensor Cores. As a result, in total the GV100 GPU can achieve 7.8 TFlops in DP, 15.7 TFlops in SP, and 125 TFlops in Tensor performance. Compared with the previous generation P100, the V100 has several innovative architectural features, such as new streaming multiprocessor architecture optimized for deep learning, the second-generation NVLink, and the HBM2 memory.

### 2.1.3 Intel Knights Landing

The Intel Knights Landing (KNL) [28] [29] provides up to 72 enhanced Silvermont cores. Each core has a private 32 KB L1 data and 32 KB L1 instruction cache. Cores are arranged in tiles. Each tile holds two cores and a shared 1 MB L2 cache (private to the tile). Tiles are connected into a 2D mesh that provides cache coherence between the L2 caches. The mesh incorporates the memory controllers and the I/O connections.

KNL presents an heterogeneous memory hierarchy with near and far memory. The near memory consists of 16 GB of integrated MCDRAM; the far memory consists of DDR4 slots, accessible through two memory controllers. Regarding the horizontal data motion, cache-coherence is provided between the L2 caches across tiles using a MESIF protocol.

### 2.1.4 Intel SCC Chip

The 48-core Intel Single Chip Cloud (SCC) processor [30] is the second processor developed by the Intel TeraScale Research program. The first processor from the same program was the 80-core TeraFlops Chip [7] – the first many-core processor in the world. The SCC chip consists of 24 dual-IA-core tiles connected by a 2D-grid on-die network. These 24 tiles are organized in a $6 \times 4$ mesh and each tile contains two blocks of a P54C core [31], a mesh interface unit (MIU), a 16 KB message passing buffer (MPB). The MPB provides a fast, on-die shared SRAM, as opposed to the bulk memory accessed through four DDR3 channels.

While it does not provide any hardware-managed memory coherence, the SCC chip features a new memory type to enable efficient communication between the cores. To move data horizontally, the MIU catches cache misses and decodes the 32-bit memory addresses

from the core into a system address to access up to 64 GB of memory. On the other hand, to move data vertically, the SCC chip uses four DDR3 memory controllers to transfer data between the MPB and main memory.

### 2.1.5 STI CELL

The CELL processor [14] [15] is a joint design effort by Sony, Toshiba, and IBM (STI). The processor adopts the heterogenous many-core architecture; it consists of a conventional high performance PowerPC core (PPC) and 8 simple synergistic processing elements (SPEs). Each SPE has four SP 6-cycle pipelined FMA data paths and one DP half-pumped 9-cycle pipelined FMA data path with 4 cycles of overhead for data movement. In addition, each SPE has its own local memory from which it fetches code and reads and writes data. All loads and stores issued from the SPE can only access the SPE's local memory.

To move data vertically, the CELL processor depends on explicit DMA operations to move data from main memory to the local store of the SPE. Dedicated DMA engines allow multiple concurrent DMA loads to run concurrently with the SIMD execution unit, thereby mitigating memory latency overhead via double-buffered DMA loads and stores. On the other hand, as the cache coherent PowerPC core, the eight SPEs, the DRAM controller, and I/O controllers are all connected via 4 data rings. To move data horizontally, the CELL processor also adopts DMA. Simultaneous DMA transfers on the same ring are possible and all transfers need to be orchestrated by the PPC.

### 2.1.6 Sunway SW26010

The SW26010 processor [32] is the 4th generation of the China's independently-developed Sunway processor. The previous generation, SW1600, is a 16-core processor [33]. In order to achieve high energy-efficiency, the SW26010 processor shifted from multi-core to many-core architecture. The SW26010 processor has four core-groups with each of them consisting of a manage processing element (MPE) and 64 compute processing elements (CPEs). The 64 CPEs are connected by a 2D on-chip mesh network. Both MPE and CPE cores are 64-bit RISC single-threaded cores with 32 vector registers, working at 1.45 GHz and supporting 256 bit vectors (i.e. 4 double-precisions). Each CPE has a in-order dual-issue pipeline and performs 8 double-precisions flops per cycle (4 data paths with fused multiply-add instructions), reaching the peak performance of 11.6 GFlops.

On a single core-group of the SW26010, the MPE and the 64 CPEs are connected via an 2D-mesh on-chip network and a memory control (MC) is used to connect a 8GB DDR3 main memory. DMA is adopted to move data vertically between the main memory and the CPE

local memories. Instead of using the load/store scheme such as a cache-coherence protocol, a lightweight register level communication (RLC) is employed to move data horizontally among the 64 CPEs.

## 2.2 On-chip Network

### 2.2.1 Inter-core Communication Shift

As the core count on a single chip increases, a scalable and high-bandwidth communication fabric to connect them becomes critically important. In the recent years, the architecture of inter-core communications are shifting from buses and crossbars to on-chip networks (OCNs) [34] [35]. The buses can only scale to a modest number of processors. The crossbars provides more scalability than the buses, and can be used for a small number of cores. However, both of them scale poorly for many-core processor. Compared with the buses and crossbars, OCNs have two main advantages [34]: 1) The OCNs can supply scalable bandwidth at low area and power overheads that correlate sub-linearly with the node numbers; 2) The OCN are very efficient in their use of wiring, multiplexing different communication flows on the same links allowing for high bandwidth.

An OCN consists of three essential components: topology, routing, and switching. The topology determines the physical layout between nodes. The routing can balance traffic. The switching determines how resources are allocated to messages. We will explain these three components in more depth as the following.

### 2.2.2 Topology

The topology determines the physical layout of connections among the main component, such as nodes, switches, and links. The topology is vital for OCNs for two reasons. First, it determines the number of hops a message must traverse, thus influencing network latency significantly. Second, it dictates the total number of alternate paths between nodes, affecting how well the network can spread out traffic.

The topologies can be categorized into direct and indirect. In the former, each node is connected to at least one core; in the latter, a subset of nodes are not connected to any core. The typical OCN topologies [36] include the ring, 2D mesh, fat tree, and torus. Among them, the first two are the most popular topologies used in many-core processors.

#### Ring

In a ring topology, all routers connect to a ring and each router connects to two neighbor routers and one core. Hence, the routers number is equal to core number. The design of the ring topology is simple, and the power consumption is low. It was adopted in the STI CELL processor [14], where four unidirectional rings are used, and each ring is 16 bytes wide and can support three concurrent transfers.

#### 2D Mesh

The 2D mesh topology consists of a grid of routers with interconnecting cores placed along with the routers. Each router, except those at the edges, is connected to four neighbor routers and one core. Therefore, the router number is equal to the core number. The mesh topology is popular in many-core processor, such as Intel 80-core TeraFlops Chip [8], Intel 48-core SSC Chip [30], TILE64 processor [20], and the SW26010 processor [18]. Taking the TILE 64-core processor for an example, the iMesh of TILE64 consists of five 8 x 8 meshes, each channel consisting of two 32 bit unidirectional links, leveraging the wealth of wiring available on-chip.

### 2.2.3 Routing

Routing algorithms decide the path a message takes from a source to a destination over the network. The routing algorithm is used to distribute traffic evenly among the paths supplied by the network topology, so as to avoid hotspots and minimize contention. Due to the small areas on chip, lightweight routing algorithms should be designed specific for many-core processors, instead of complex routing algorithms used in off-chip networks. The routing algorithms have three kinds: deterministic, oblivious and adaptive.

#### Deterministic Routing

In deterministic routing, the path is determined only by the source and destination; all messages from node A to B will always traverse the same path. One example of the deterministic routing is dimension-ordered routing (DOR) [37]. DOR was adopted by the TILE64 processor.

#### Oblivious Routing

To provide higher throughput than the deterministic routing, in oblivious routing, messages traverse different paths from A to B, but the path is selected randomly without regard to

network congestion. A router could randomly choose among alternative paths prior to sending a message. One example of an oblivious routing algorithm is Valiant's randomized routing algorithm [38].

#### Adaptive Routing

One weakness of the oblivious routing is the paths are chosen randomly without regard to the state of the network. This may cause to send data to the congestion path. To address this issue, the adaptive routing takes the traffic information into account to chooses the path. However, the adaptive routing can raise potential deadlocks and change inter-message orders. Therefore, the dead-lock free and re-order mechanisms should be implemented in the adaptive routing.

### 2.2.4 Switching

Compared with both topologies and routing algorithms, the switching has a more significant impact on OCN performance. It determines when buffers and links are assigned to messages, how these resources are shared among the many messages using the network, and the granularity at which they are allocated. Data granularity has three levels: message, packet, and flit. According to these granularities, the switching can be classified into the message-based, packet-based, and flit-based switching.

#### Message-based Switching

Message-based switching, such as the circuit switching [39], reserves the link from the source to destination until entire data is transmitted. The message-based switching has four steps. 1) A probe is sent into network and reserves the links from the source to the destination. 2) Once the probe reaches the destination, an acknowledgement message will be sent back to the source. 3) Once the source receives the acknowledgement message, it will begin to transmit the entire message. 4) When the entire message has received at the destination, the links are released.

#### Packet-based Switching

Unlike the message-based switching, packet-based switching breaks down the messages into multiple packets and each packet is handled independently by the network. There are two types of packet-based switching: store-and-forward [40] and cut-through switching [41]. In store-and-forward switching, routers store the complete packet and forward it based on the

information within the header. In other words, each packet is individually routed from source to destination. On the other hand, the cut-through switching allows transmission of a packet to proceed to the next node before the entire packet is received at the current router.

#### Flit-based Switching

Unlike the packet-based switching, the flit-based switching allocates storage and bandwidth to flits rather than entire packets. This allows relatively small flit- buffers to be used in each router, even for large packet sizes. One example of the flit-based switching is the wormhole switching [42] and it was adopted in the TILE64 [20] processor. As using the finest data granularity, the wormhole switching requires much smaller buffers than both the message-based and packet-based switchings, thus making it as the most popular switching for many-core processors.

## 2.3 Register-communication Processors

Based on OCNs, register communications are adopted by many processors. In this section we select four processors as case studies to illustrate the register communication.

### 2.3.1 The Multi-ALU Processor

The Multi-ALU Processor (MAP) [19] provides two approaches to quickly communicate among its three on-chip processors: the register communication and shared on-chip cache. For the register communication, an OCN allows a message to be transmitted from a register to another. Moreover, a thread executing on one processor can directly write to a remote register. The benchmark results showed the register communication is 10X faster than the shared on-chip cache.

### 2.3.2 The TILE64 Processors

The TILE64 processor [20] [23] is a many-core processor targeting the high-performance demands of a wide range of embedded applications. The 64 cores of this processor are connected with a 2D mesh and arranged in an $8 \times 8$ array. Each pair of two adjacent cores can communicate their register data with a 2-cycle latency.

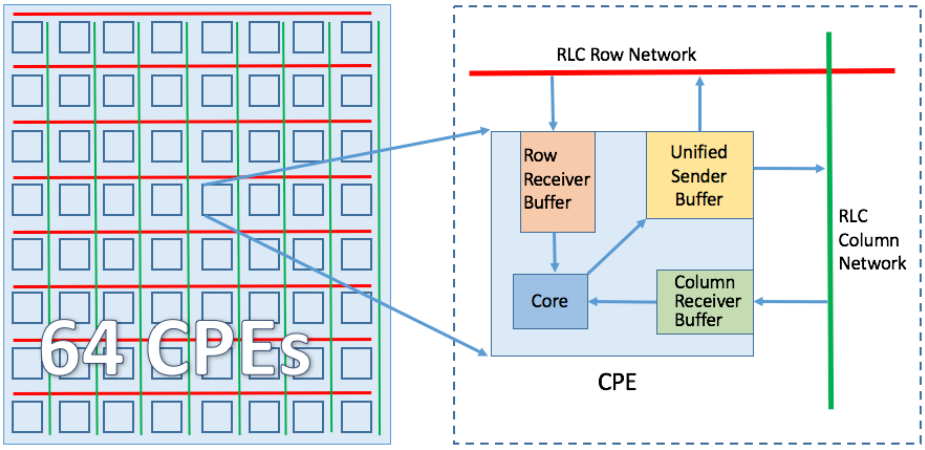



Fig. 2.2 At the register level, each CPE can communicate with the remaining 7 CPEs in the same row or column via the 2D on-chip mesh network.

### 2.3.3 A 16-core Embedded Academic Processor

The 16-core embedded academic processor [43] adopts hybrid inter-core communication schemas with both shared-memory and register inter-core communications. The processor has two on-chip clusters and each cluster comprises eight Processor Cores and one Memory Core. Shared memory is only allowed within the same cluster; however, the register communication can communicate all Processor Cores across different clusters.

### 2.3.4 The SW26010 Processor

The SW26010 processor can support register communication among the 64 CPEs of a single core-group. Fig.2.2 shows that register communication uses the three local buffers in each CPE and the 2D on-chip network to transfer register data. To send, the CPE puts the register data (with the `putr` instruction for the row network and `putc` for the column network) into the unified sender buffer. The unified sender buffer connects to both the row and column networks and can hold up to 6 vector elements. Notably, if the unified sender buffer is full, the `putr` and `putc` instructions will be blocked. To receive, the CPE moves the register data into the vectors register (with the `getr` instruction from the row receiver buffer and `getc` from the column receiver buffer). Each receiver buffer can only connect to the network in the same direction and hold up to 4 vector elements.

However, the register communication of the SW26010 has two constraints. The first is RLC does not support all-to-all communication among the 64 CPEs. Instead, RLC can only allow communication among the 8 CPEs in the same row or column with the two

communication modes, the peer-to-peer mode and the row/column-based broadcast mode. The latencies of the two modes are the same [44]. Due to the asynchronous communication, the RLC latency can be hidden by the double-buffering optimization. The second is RLC uses an anonymous producer-consumer protocol, which renders the register communication message “contains no ID tag" that shows the senders’ information. Addressing this constraint requires orchestrating the message sequences manually, so as to ensure the correct sending and receiving order. This manual approach significantly increases the programming challenges of the SW26010.

## 2.4 Related Work

This section provides a comparison of the three major work presented in this thesis with prior work. Section 2.4.1 compares the benchmarks used on other many-core processors, such as NVIDIA GPU and Intel Xeon Phi. Section 2.4.2 compares the optimization work on DGEMM and N-body on other platforms. Section 2.4.3 compares the non-blocking PCG optimizations.

### 2.4.1 Benchmarking Many-core Processors

Many benchmark suites have been developed for commodity many-core processors (such as the STI CELL processor, NVIDIA GPUs, and the Intel Xeon Phi processor). For example, Y. Dou *et al.* [45] evaluated impact factors of the P2P DMA mode on the STI Cell processor. In contrast, we conducted a comprehensive evaluation on three DMA modes of the SW26010: the *PE* mode, the *BCAST* mode, and the *RANK* mode. We revealed the *BCAST* mode has 16X more bandwidth than the P2P mode, and the *RANK* mode can outperform the *PE* mode substantially when the data block size was small. In addition,

H. Wong *et al.* [46] used a set of CUDA micro-benchmarks to illustrate the architecture of the arithmetic processing cores and the memory hierarchies of the NVIDIA GT200 GPU. Due to the divergence of the core designs and the memory hierarchies between GPUs and the SW26010, we developed our own micro-benchmarks, written in assembly language, to reveal the micro-architectural features of the CPE pipelines and LDMs.

J. Fang *et al.* [47] designed a micro-benchmark suite *MIC-Meter* to quantify the key architectural features of the Intel Xeon Phi 5100. Part of our latency measurement approach was based on their work. However, for RLC, which is the unique hardware feature of the SW26010, we developed micro-benchmarks to measure the latencies and memory bandwidths of RLC.

### 2.4.2 Compute-bound Kernels Optimizations

#### N-body

N-body optimizations have been studied on many-core processors well. The architecture-independent techniques include vectorization and loop unrolling on both the NVIDIA GPU [48] and the Intel KNC [49]; the architecture-specific optimizations include instruction mix on the STI CELL [50], data alignment on the Intel KNC [51], and Multi-Channel DRAM optimization on the Intel KNL [52]. The closest approach [53] to our study is the algorithm approximating the whole N-body kernel on an early IBM machine without hardware *rsqrt*. In contrast, due to the inefficient *rsqrt* hardware instructions on the SW26010, we used a software *rsqrt* routine to boost the performance.

#### DGEMM

Architecture-specific optimizations are required for DGEMM to obtain the near-peak performance on many-core processors, such as software pipelining on the NVIDIA GPU [54], Knights Corner (KNC)-friendly matrix format on the Intel KNC [55], DMA double-buffering on the STI CELL [16]. In contrast to the commercial many-core processors, the home-grown SW26010 is designed with a unique hardware feature, RLC, to reuse the data among 64 CPEs. Therefore, we designed a RLC-friendly algorithm to achieve 88.7% efficiency in a single core group of the SW26010. The novel optimization on RLC is critical to reach the ninja performance for DGEMM on this highly memory-bound processor.

### 2.4.3 PCG Optimizations

The key idea of the optimizations on PCG is the overlap the all_reduce communication with computations. These efforts include the communication-avoiding PCG [56], which is based on three-recurrence CG method [57], the pipelined PCG [58] and 2-iteration Pipelined2PCG [59]. However, these various optimized PCG need more memory storage and registers for extra vector operations. Due to the limited limited numbers of registers available in each CPE (32), our RLC-friendly Non-blocking RLC (RNPCG) is based on Non-blocking PCG (NBPCG) [59]. We use the RLC among 64 CPEs to overlap the two all-reduce operation with the SpMV preconditioner kernels and on the single-core group.

Parallel ILU preconditioners include two kinds: global and local. Global preconditioners [60] [61] can use the global coupling information from the matrix for the ill-conditioned matrices. The localized ILU preconditioners [62] [63] can perform on each processor independently, which can be viewed as an inaccurate block Jacobi preconditioner. Yang et.al

[64] mentioned they implemented a global ILU preconditioner with RLC for their implicit solver on TaihuLight, but did not provide any details. In our work, we propose the localized DIC preconditioner for the SW26010.

## 2.5 Summary

In this chapter we provided an in-depth context of many-core processors and inter-core communication. We started with an overview of the vertical and horizontal data motions in many-core processors. Next, we discussed the three important aspects of on-chip networks: topologies, routing, and switching. Afterward, we provided an extensive presentation of the four typical processors using register communications. Finally, we compared our three major work with prior work.

# Chapter 3

# Evaluations with Micro-benchmarks

The design of the SW26010 took a radical design departure from the conventional x86 multi-core processors in terms of its energy-efficiency designs. However, compared with commodity processors, such as Intel Xeon Phi processors and NVIDIA GPUs, the SW26010 was provided with rather limited public data about its micro-architecture. The lack of the knowledge of the SW26010 processor prevents global researchers from conducting comprehensive performance optimizations on the TaihuLight supercomputer. The existing data of the processor are mainly from three documents: J. Dongarra [17], H. Fu et al. [18], and D. Chen and X. Liu [65].

The purpose of this chapter to illuminate the uncharted area of the home-grown processor so as to provide important information for its performance optimizations. We developed a micro-benchmark suite *swCandle*, mostly written in assemble language, to evaluate three key architectural features that impact the performance: 1) the pipeline of the processor cores, 2) the register-register communication among cores, and 3) the approaches to access the explicit memory hierarchy. In **Section 3.1**, we evaluate the CPE pipelines. Then in **Section 3.2**, we measure the latencies and bandwidths of register communication. Afterward, we evaluate the on-chip SPM and main memory in **Section 3.3**. Based on the results from these three evaluations, we discuss the implications on performance optimizations in **Section 3.4**. We summarize the chapter in **Section 3.5**.

## 3.1 The CPE Pipeline

Each CPE has an in-order dual-issue pipeline that allows the vectorized floating-point instructions to co-issue with the data motion instructions in the same cycles. This type of in-order dual-issue pipeline design was also adopted in the STI CELL processor [14] and the Intel Knight Corner (KNC) co-processor [13] for energy-efficiency.

### 3.1.1 Instruction Latencies

**Publicly available data**: no public information was available about the instructions latencies of the SW26010.

**Benchmark approaches**: we measured the absolute and pipelined latencies of four types of instruction: the vectorized arithmetic instructions, the scratchpad memory (SPM) access instructions, the permutation instructions, and the synchronization instructions (used in register communications).

We executed a long sequence of the same instructions with read after write (RAW) data dependencies to measure the absolute latencies. Taking the `vmad` instruction (vectorized fused multiply-add) as an example, we created a long sequence of `vmad $0, $1, $2` with `$0` as a register used to store results, and executed the sequence in sequential due to the RAW data dependency.

On the other hand, we executed a long sequence of the same instructions without RAW data dependencies to examine the pipelined latencies. For example, we created a long sequence of the `vmad` instruction using different registers and executed the sequence in a pipelined behavior due to the data independency.

**Results**: Fig. 3.1 shows the the absolute and pipelined latencies of the instructions measured by our micro-benchmarks. The instruction prefix 'v' represents the vector operations. The instruction suffix 'd'/'s' and 'w'/'f' indicate double/single precision and word/floating-point vectors respectively; 'c'/'r' mean synchronizations in column/row. The permutation instructions 'ins'/'ext'/'shf' correspond to insertion/extraction/shuffle. The `sync`/`synr` instructions can synchronize register communications in the same **c**olumn/**r**ow. The normal vectorized arithmetic instructions are fully pipelined, thus requiring *7 cycles* for the absolute latency and *1 cycle* for the pipelined latency. On the other hand, due to the lack of the efficient hardware support (e.g. the extended math unit), the strong arithmetic instructions (such as square root and division) are not fully pipelined and require long latencies.

### 3.1.2 Instruction Issue Orders

**Publicly available data**: the in-order dual-issue CPE pipeline (P0 and P1) can support in-order issue for scalar and vectorized computing operations of both floating-point and integer instructions in P0, and in-order issue for data motion, compare, jump and scalar integer operations in P1 [18]. However, the instruction issue order between P0 and P1 was unknown.

**The benchmark approach**: as shown in Fig. 3.2a, we developed a micro-benchmark to evaluate the instruction issue order between P0 and P1. The three dependent `vldd` instructions

| Category | Instructions | Absolute Latency (*cy*) | Pipelined Latency (*cy*) |
|---|---|---|---|
| Arithmetic | vmuld, vmuls, vaddd, vadds, vsubd, vsubs, vmad, vmas | 7 | 1 |
| | vsqrtd/vsqrts<br>vdivd/vdivs | 32/18<br>34/19 | 28/14<br>30/15 |
| SPM Access | vldd, vlds | 4 | 1 |
| Permutation | vinsw, vinsf, vextw, vextf, vshff, vshfw | 1 | 1 |
| Synchronization | sync, synr | 14 | 14 |

Fig. 3.1 The absolute and pipelined latencies of the instructions.

(each requiring *4 cycles*) follow the three dependent `vmld` instructions (each requiring *7 cycles*), and have the write after read (WAR) data dependency with the first `vmuld` instruction only.

If the issue order is in-order, the first `vldd` can be co-issued with the third `vmuld` in the same cycle (Fig. 3.2b). The total latency of the 6 instructions would be $7 \times 2 + 4 \times 3 = 26$ *cycles*. While if the issue order is out-of-order, the first `vldd` can be issued one cycle after the first `vmuld` (Fig. 3.2c). Then the total latency would be *21 cycles*.

**The result**: as the measured total latency is *21 cycles*, we infer the instruction issue order between the pipeline P0 and P1 is **out-of-order**.

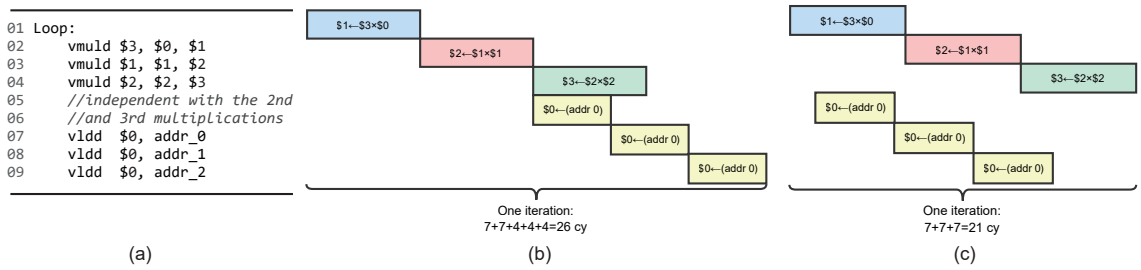



Fig. 3.2 The instruction issue order in between the in-order dual-issue pipeline of CPEs. (a) the micro-benchmark written in assembly language; (b) if the issue order is in-order, the latency of a single iteration of (a) would be *26 cycles*; (c) if the issue order is out-of-order, the latency would be *21 cycles*.

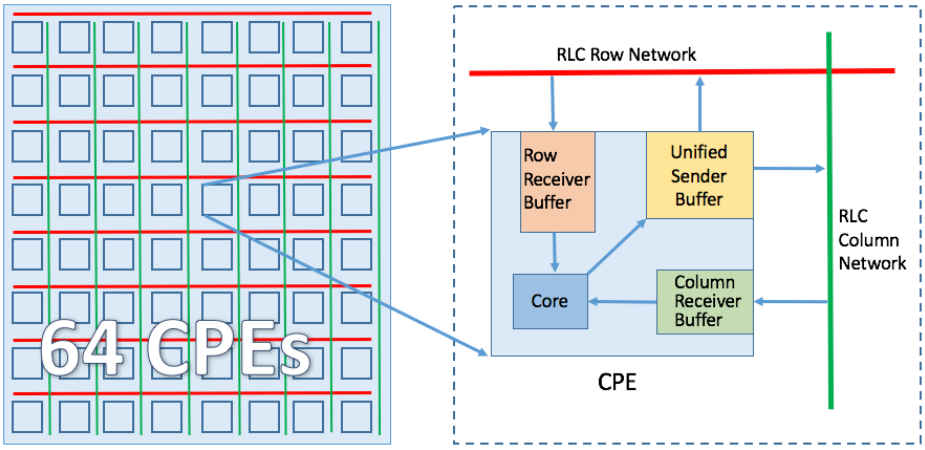



Fig. 3.3 RLC uses the three local buffers of each CPE and the 2D on-chip network to transfer register data among 64 CPEs of a single core-group.

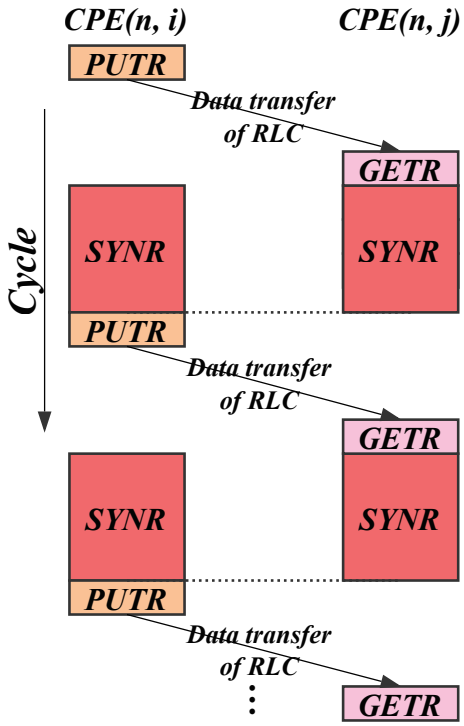
## 3.2 Register Communications

RLC can only support communication in the same row/column, in either the Peer-to-Peer (P2P) mode or the broadcast mode. In the former mode, a CPE can send its register data to any one of the 7 CPEs in the same row/column; In the latter mode, a CPE can broadcast its register data to the remaining 7 CPEs in the same row/column.

RLC takes three steps to transfer register data among CPEs. First, a source CPE puts the register data (with the SEND instruction `putr` for the row-based communication and `putc` for the column-based communication) into the unified sender buffer (Fig.3.3). The unified sender buffer connects to both the row and column networks and can hold up to 6 vector elements. If the unified sender buffer is full, the SEND instructions will be blocked. Second, the data transfer through the 2D on-chip network from the source CPE to a destination CPE. Third, the destination CPE moves the register data into the vectors register (with the RECEIVE instruction `getr` from the row receiver buffer and the `getc` from the column receiver buffer). Each receiver buffer connects to the network in the same direction and can only hold up to 4 vector elements. Therefore, the latency of a RLC transfer consists of three parts: the latency of a SEND instruction, the latency of register data transmission through the 2D on-chip network, and the latency of a RECEIVE instruction.

### 3.2.1 P2P RLC Latencies

**Publicly available data**: The latency of the P2P RLC is *10 cycles* [65].




(a)

```
01      //sender CPE(n, i):
02      LOOP:
03      //send data to CPE j
04      putr $data, j
05      //sync in row with CPE j
06      synr $mask
07
08      //receiver CPE(n, j):
09      LOOP:
10      //receive data from CPE i
11      getr $data
12      //sync in row with CPE i
13      sync $mask
```

(b)

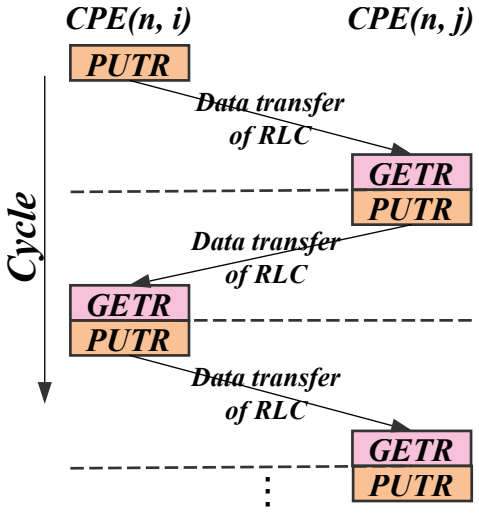
Fig. 3.4 Measuring the latency of a single P2P RLC. (a) an illustration of the CPE(n,0) sending its register data to the CPE(n,j); (b) a micro-benchmark to measure the latency.

**Benchmark approaches**: we measured three types of latency in the P2P mode: the latency of a single P2P RLC, the latency of the round-trip P2P RLC, and the latencies of the SEND/RECEIVE instructions.

First, to measure the latency of a single P2P RLC, we used the synchronizations instructions (`sync`/`synr`) to prevent the asynchronous SEND instructions from being overlapped in between the consecutive RLCs (Fig. 3.4a). We developed a micro-benchmark (Fig. 3.4b) and calculated the latency of a single P2P RLC by subtracting the latency of synchronization instructions (*14 cycles*) from the measured latency.

Second, to measure the latency of a round-trip P2P RLC, we conducted the *ping-pong* test (Fig. 3.5) on a pair of the sender and receiver CPEs. No forced syntonizations are required in this *ping-pong* test because the sender CPE will be blocked until it receives the data back.

Third, to examine the latency of the RECEIVE instructions (`getc`/`getr`), we created write after write (WAW) data dependencies in between the consecutive RECEIVE instructions by writing results to the same registers. In addition, we used two pairs of synchronizations in each iteration (Fig. 3.6a) to isolate the latencies of SEND and RECEIVE instructions. The first pair was used after the consecutive SEND instructions in the sender CPE and before the




(a)

```
01      //CPE(n, i):
02      LOOP:
03      //send data to CPE j
04      putr $data1, j
05      //receive data from CPE j
06      getr $data2
07
08      //CPE(n ,j):
09      LOOP:
10      //receive data from CPE i
11      getr $data1
12      //send data to CPE i
13      putr $data2, i
```

(b)

Fig. 3.5 Measuring the latency of a round-trip P2P RLC (a) an illustration of the *ping-pong* test for the CPE(n,i) and CPE(n,j); (b) a micro-benchmark to measure the latency.

consecutive RECEIVE instructions in the receiver CPE. The second pair was used at the end of each iteration.

Notably, we are not able to measure the latency of the SEND instructions (`putr`/`putc`) directly due to the asynchronous SEND instructions, whose latency can be overlapped by register data transfer through the on-chip network. We can only estimate the combined latency of a SEND instruction and a RLC message transfer by subtracting the latency of a RECEIVE instruction from the latency of a single P2P RLC.

**Results**: Tab. 3.1 shows the latencies of P2P RLC among the 64 CPEs of a single core-group. We observed that, surprisingly, **the latencies of the P2P RLC vary from *10 cycles* to *11 cycles* depending on the different locations of receiver CPEs**. In the row-based

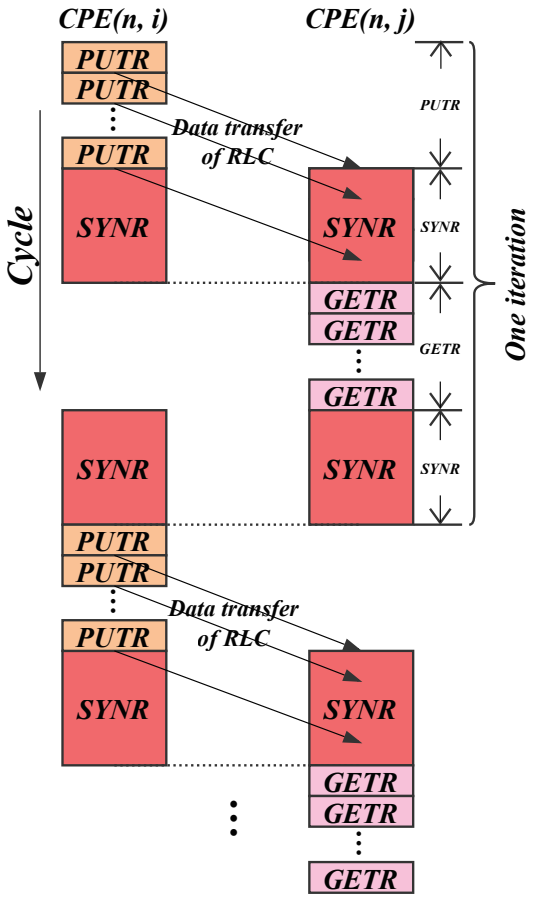
Table 3.1 The results of P2P RLC latencies

(a) RLC latencies in rows (cy)

| Sender No. | Receiver No. 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | / | 10 | 10 | 10 | 11 | 11 | 11 | 11 |
| 1 | 10 | / | 10 | 10 | 11 | 11 | 11 | 11 |
| 2 | 10 | 10 | / | 10 | 11 | 11 | 11 | 11 |
| 3 | 10 | 10 | 10 | / | 11 | 11 | 11 | 11 |
| 4 | 10 | 10 | 10 | 10 | / | 11 | 11 | 11 |
| 5 | 10 | 10 | 10 | 10 | 11 | / | 11 | 11 |
| 6 | 10 | 10 | 10 | 10 | 11 | 11 | / | 11 |
| 7 | 10 | 10 | 10 | 10 | 11 | 11 | 11 | / |

(b) RLC latencies in columns (cy)

| Sender No. | Receiver No. 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | / | 10 | 11 | 11 | 10 | 10 | 11 | 11 |
| 1 | 10 | / | 11 | 11 | 10 | 10 | 11 | 11 |
| 2 | 10 | 10 | / | 11 | 10 | 10 | 11 | 11 |
| 3 | 10 | 10 | 11 | / | 10 | 10 | 11 | 11 |
| 4 | 10 | 10 | 11 | 11 | / | 10 | 11 | 11 |
| 5 | 10 | 10 | 11 | 11 | 10 | / | 11 | 11 |
| 6 | 10 | 10 | 11 | 11 | 10 | 10 | / | 11 |
| 7 | 10 | 10 | 11 | 11 | 10 | 10 | 11 | / |




(a)

```
01      //sender CPE(n, i):
02      LOOP:
03      //send data to CPE j
04      putr $data, j
05      ...
06      putr $data, j
07      //sync in row with CPE j
08      synr $mask
09      //sync again
10      synr $mask
11
12      //receiver CPE(n, j):
13      LOOP:
14      //sync in row with CPE i
15      synr $mask
16      //receive data from CPE i
17      getr $data
18      ...
19      getr $data
20      //sync again
21      synr $mask
```

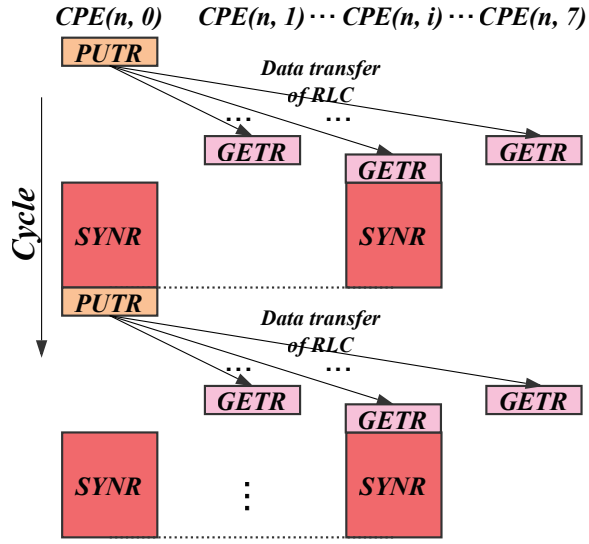
(b)

Fig. 3.6 Measuring the latency of RLC RECEIVE instructions. (a) an illustration of sending multiple RLCs from CPE(n,i) to CPE(n,j); (b) a micro-benchmark to measure the latency.

communications, a P2P RLC only requires *10 cycles* when receiver CPEs are in the row 0-3, but require one more cycle when receiver CPEs are in the row 4-7. The similar situation happens in the column-based communications. We repeated this test randomly on more than 500 SW26010 processors. The results kept same. The reason causing the non-equal RLC latencies remains unclear.

Second, the results of the *ping-pong* test showed the latency of a round-trip RLC was exactly twice as that of a single P2P RLC.

Third, by gradually increasing the number of the SEND and RECEIVE instruction pairs, we found that the latency of each iteration increased by *2 cycles* for each additional instruction pair, implying that the RECEIVE instruction might cost *1 cycle* (the other *1 cycle* might be cost by the asynchronous SEND instruction), and the SEND and RECEIVE instructions were fully pipelined. In addition, by subtracting the latency of the RECEIVE instruction (*1 cycle*)




(a)

```
01      //CPE(n, i):
02      LOOP:
03      //send data to CPE j
04      putr $data1, j
05      //receive data from CPE j
06      getr $data2
07
08      //CPE(n, j):
09      LOOP:
10      //receive data from CPE i
11      getr $data1
12      //send data to CPE i
13      putr $data2, i
```

(b)

Fig. 3.7 Measuring the latency of the broadcast RLC. (a) an illustration of the CPE(n,0) broadcasting its register data to the remaining 7 CPEs in the same row *n*; (b) a micro-benchmark to measure the latency.

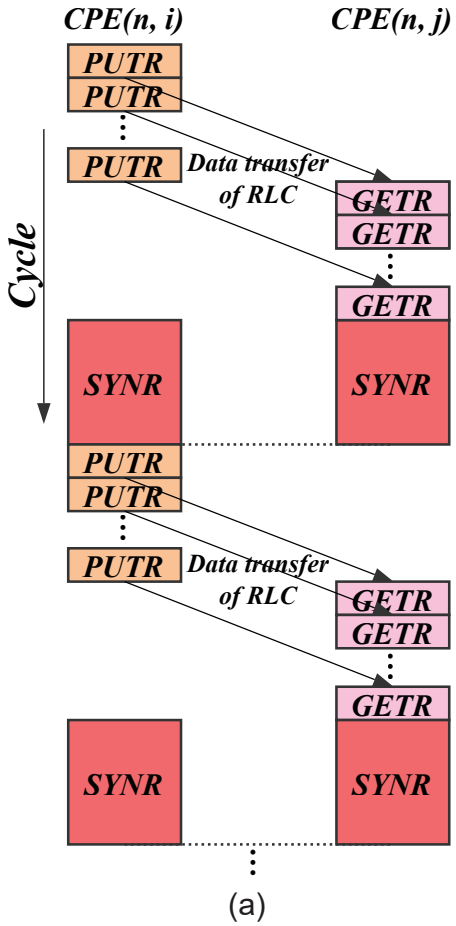
from the total latency, we can estimate that the latency of the SEND instruction combined with the data transfer through the on-chip network was *9-10 cycles*.

### 3.2.2 Broadcast RLC Latencies

**Publicly available data**: the latency of the broadcast mode is *14 cycles*, which is *4 cycles* longer than the P2P mode [65].

**The benchmark approach**: we developed a micro-benchmark (Fig. 3.7b) to measure the latency of the broadcast mode. As there is no grantee that the 7 receiver CPEs can receive a broadcasting message in the same cycle, we measured the latencies for the 7 CPEs respectively. Fig. 3.7a shows a scenario of the CPE(n,0) broadcasts in the row *n*. We also conducted tests for other scenarios, such as the column-based broadcast and broadcasting from different sender CPEs.

**Results**: the results show that **the broadcast mode has the same latency as the P2P mode**. We speculated that the broadcast mode might be implemented by default and the P2P mode might be implemented as a special case of the broadcast mode; the implementations could be similar to the mask operations in vector processing.




```
01      //sender CPE(n, i):
02      LOOP:
03      //successively send n data
04      putr $data_1, j
05      ...
06      putr $data_n, j
07      //sync in row with CPE j
08      synr $mask
09
10      //receiver CPE(n, j):
11      LOOP:
12      //receive n data
13      getr $data_1
14      ...
15      getr $data_n
16      //sync in row with CPE i
17      synr $mask
```

(b)

Fig. 3.8 Measuring the latency of consecutive RLCs. (a) an illustration of sending consecutive RLCs from CPE(n,i) to CPE(n,j); (b) a micro-benchmark to measure the latency.

### 3.2.3 RLC Bandwidths

**Publicly available data**: no public information is available about RLC bandwidths.

**The benchmark approach**: we measured the RLC bandwidths through two steps. First, we developed a micro-benchmark (Fig. 3.8b) to successively issue multiple pairs of SEND and RECEIVE instructions, then measured the latency from the first SEND instruction to the last RECEIVE instruction (Fig. 3.8a). Second, based on the measured latencies, we estimated the aggregated bandwidths of RLC.

**Results** Tab. 3.2 shows the measured latencies and gaps (minimal time interval between consecutive RLC transmissions) of a single iteration of the consecutive RLCs. We observed that when the numbers of the consecutive RLCs (denoted as $n$) increases, two kinds of gaps occur: one extra cycle (denoted as $g_1$) occurs in every RLC transmission and another extra cycle (denoted as $g_2$) occurs in every three RLC transmissions. Accordingly, we formulated

Table 3.2 The latencies and gaps of a single iteration of consecutive RLCs.

| | No. of consecutive RLCs | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Latency (cy) | 10 | 12 | 14 | 17 | 19 | 21 | 24 | 26 | 28 | 31 |
| Gap (cy) | 1 | 1 | 2 | 1 | 1 | 2 | 1 | 1 | 2 | 1 |

the latency of each iteration (denoted as $L$) as: $L = t_o + (1+g_1) \times (n-1) + \lfloor (n-1)/3 \rfloor \times g_2$, where $t_0$ is the latency of a single P2P RLC (*10 cycles*). We conducted further experiments when $n >= 10$ and found the formula still validates.

As aforementioned, a complete RLC includes three phases: 1) putting data from the unified sender buffer to the 2D on-chip network, 2) transferring the data through the on-chip network, and 3) getting the data from the network to the row/column receiver buffer. We believed that the gaps $g_1$ and $g_2$ should occur in the second phase, because the SEND and RECEIVE instructions in the first and third phases are fully pipelined. Despite the exact reason remains unknown, we speculated that the gaps may be caused by the underlying implementation of the data transfer protocol, e.g., data to be transferred could be split into two flits and launched in turn, leading to stalls for subsequent transfers.

We then used the latencies and gaps results in Tab. 3.2 to estimate the aggregated RLC bandwidths. We observed that a pair of CPEs (a sender CPE and a receiver CPE) transfer 32 byte (i.e. 256 bit) data in average $1 + g_1 + \frac{g_2}{3} = 2.33$ *cycles* for every P2P RLC. Therefore, we estimated the aggregated P2P RLC bandwidth to be: 32/2.33 bytes per cycle $\times$ 1.45 GHz $\times$ 32 CPE pairs per core-group $\times$ 4 core-group per processor= 2,549 GB/s. Similarly, for every broadcast RLC, a sender CPE projects 7 concurrent data paths and each data path transfers 32 bytes in average *2.33 cycles*. We thus estimated the aggregated broadcast RLC bandwidth to be: 7 concurrent data paths $\times$ 32/2.33 bytes per cycle $\times$ 1.45 GHz $\times$ 8 columns/rows per core-group $\times$ 4 core-groups per processor= 4,461 GB/s.

### 3.2.4 Routing Modes

In hardware, the RLC only supports the row/column-based communication modes. The routing mode is required if data is to be shared among CPEs in the different rows/columns. Despite not being supported in hardware, the routing mode can be implemented with the P2P mode. According to different routing behaviors, the routing modes can be classified into two kinds, the static and the dynamic routing. In the static routing, the destination CPE is known before the source CPE sends a RLC message. The ID of the destination CPE is contained

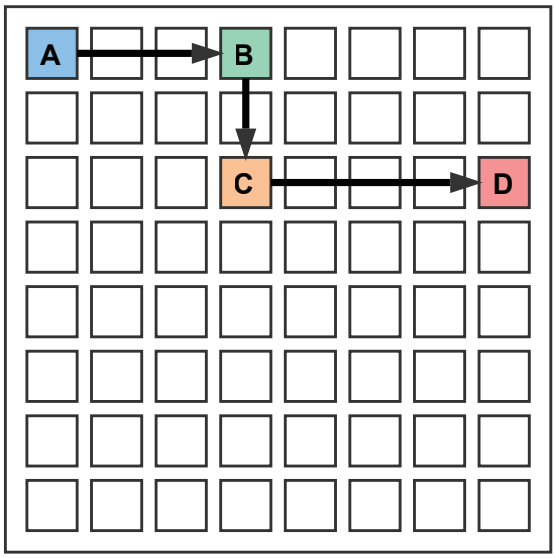
in the RLC message, and the message will be send to its destination through one or several intermediate CPEs. In the dynamic routing, in contract, the destination CPE is unknown in advance. Only the condition data is contained in the RLC message, and the message is sent to a router CPE. According to the results from checking the condition, the router CPE dispatches the RLC message to different destination CPEs.

### The Static Routing

**Publicly available data**: no public information is available about the static routing, which is not supported in hardware.

**Benchmark approaches**: We developed micro-benchmarks to measure the latency of the multiple-hop static routing. Fig. 3.9a illustrates an example of a 3-hops static routing. The ID of the destination CPE(D) was contained in a 256 bit RLC message. As the source CPE(A) and the destination CPE(D) are in the different rows and columns, while a single P2P RLC can only communicate in the same row/column, we developed a micro-benchmark (Fig. 3.9b ) to transfer the RLC message from the source CPE(A) to the destination CPE(D) through the two intermediate CPE(B) and CPE(C).

**Results**: the results show that the latency of 3-hops static routing is triple as that of a single P2P RLC. We conducted further experiments on static routing with various amounts




(a)

```
01    //Core A:
02    //send data to Core B in row
03    putr $data, 3
04
05    //Core B:
06    //get data from Core A in row
07    getr $data
08    //send data to Core C in column
09    putc $data, 2
10
11    //Core C:
12    //get data from Core B in column
13    getc $data
14    //send data to Core D in row
15    putr $data, 7
16
17    //Core D:
18    //get data from Core C in row
19    getr $data
```

(b)

Fig. 3.9 The static routing mode of RLC. (a) an illustration of a 3-hop static routing; (b) a micro-benchmark to measure the latency of the 3-hop static routing.

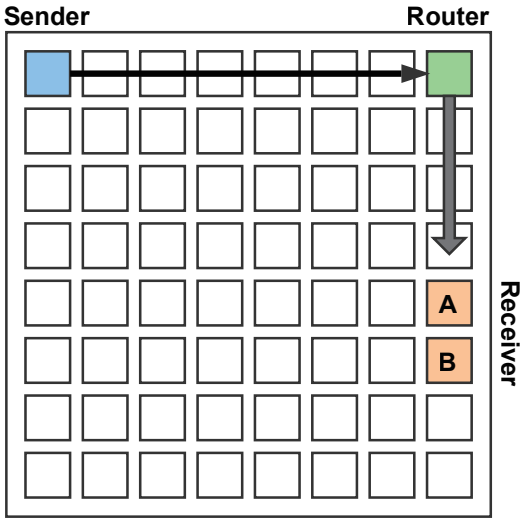
of hops. The results show that the latency of the $n$-hop static routing is $n$ times of that of the P2P mode.

### The Dynamic Routing

**Publicly available data**: no public information is available about the dynamic routing, which is not supported in hardware.

**Benchmark approaches**: Fig. 3.9a illustrates an example of a dynamic routing with two different destination CPEs. No destination CPE ID but only the condition data is contained in a 256 bit RLC message. The source CPE sends the RLC message to the router CPE in the same row. Then, the router CPE checks the condition data and, based on the checked results, dispatches the RLC message to the destination CPE(A) or CPE(B). We developed a micro-benchmark (Fig. 3.9b) to implement the dynamic routing mode and adjusted the hit rate of condition data to measure the latencies in different conditions.

**Results**: Fig. 3.11 shows the relative latency rate with the baseline of a 100% hit rate to the default branch A. We observed that the relative latency rate increases when the miss rate of the default branch A increases. In the case of the miss rate of 1.0 (i.e., all branches jump to B), the relative latency rate increases to 1.65X. This suggests that routing to the first (i.e. default) option is 1.65X faster than to the second option. We conducted further experiments




(a)

```
01 //sender:
02     //keep sending data to router
03     putr data, router
04
05 //router:
06     //get data from sender
07     getr data;
08     //choose the receiver based on data
09     if (data satisfy cond A){
10         //send to receiver A
11         putc data, A
12     } else if (cond B){
13         //send to receiver B
14         putc data, B
15     }
16
17 //receiver:
18     //keep receiving data
19     getc data
```

(b)

Fig. 3.10 The dynamic routing mode of RLC. (a) an illustration of a dynamic routing. (b) a micro-benchmark to measure the latency of the dynamic routing.

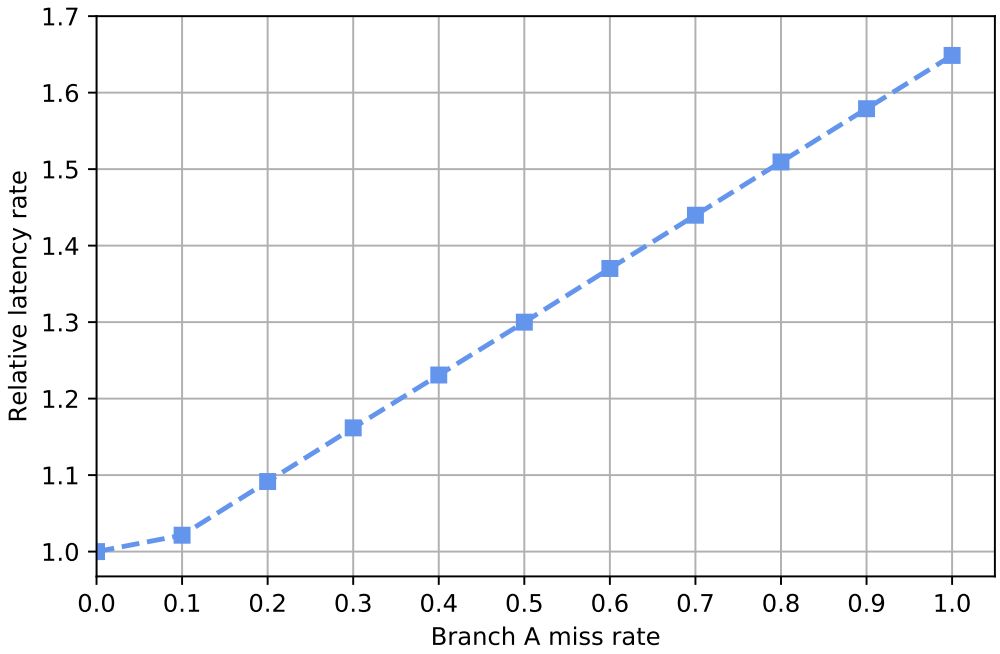



Fig. 3.11 The relative latency rate of the dynamic routing mode. The baseline is the latency when 100% hit to default branch A.

for three destinations and four destinations. All experiment results show the similar trend that routing to the first option is much faster than to the other options.

## 3.3 Memories

The SW26010 processor has two types of memories: an off-chip 8 GB main memory shared between a MPE and 64 CPEs of a core-group, and an on-chip 64 KB SPM of each CPE.

### 3.3.1 On-chip SPM

**Publicly available data**: each CPE adopts the 64 KB SPM as its local data memory (LDM). The latency of the LDMs is *4 cycles* and the bandwidth of the LDMs is 47.90 GB/s [65].

**Benchmark approaches**:we evaluated the latency and the two types of the bandwidth of LDMs. To measure the access latency of the LDMs, we adopted the *pointer-chasing* approach used in the BenchIT benchmark [66]. We initialized an array $A$ (of size $S$) residing in the LDMs with the stride addresses: $A[k] = (long)\&A[(k+stride)\%S]$, then repeatedly accessed $A$ with a pointer $p$ (initialized with $p = A$) by executing $p = (long*)(*p)$. To measure the peak bandwidth, we executed a sequence of vector load/store instructions. To

examine the sustainable bandwidth, we implemented the STREAM benchmark [67] with the athread library (a lightweight thread library similar to Pthread) [65] and SIMD compiler intrinsics. We also unrolled kernel of the STREAM benchmark for 12 times to fully utilize the in-order CPE pipeline while at the same time avoided the register spilling (i.e. moving a variable from a register to the LDM).

**Results**: we found that the latency to access the LDMs is the same as the publicly available data (*4 cycles*), while the peak bandwidth of LDMs is 46.4 GB/s per CPE, slightly lower than the publicly available data. The STREAM benchmark results in Copy, Scale, Add, and Triad are 43.74GB/s, 29.36GB/s, 31.71GB/s, and 30.9GB/s, respectively. The results show that only the STREAM Copy operation can achieve more than 70% of the peak bandwidth.

### 3.3.2 Off-chip Memory

Each core-group of the SW26010 processor adopts a 8GB DDR3 off-chip memory as its main memory. Two approaches can be used to access the main memory, DMA and global load/store instructions (`gld` and `gst`). The former can transfer a chunk of data while the latter can only access a single vector.

DMA has three modes, the *PE*, the *BCAST*, and the *RANK*. In the first mode, each CPE manages it own data transfer. In the second mode, one CPE sends a broadcast request to the MPE, then the requested data will be transferred from the main memory to the 64 CPEs of a single core-group. In the third mode, data blocks can be gathered/scattered in the row-major order from/to the 8 CPEs in the same row, as illustrated in Fig. 3.12.

#### DMA

**Publicly available data**: Only the read memory bandwidths of the *PE* mode for the data block size up to 16 KB (only $1/4$ of the SPM size) are listed in the public document [65].

**Benchmark approaches**: we implemented the STREAM benchmark to measure the bandwidths of the three DMA modes. We aligned the data in the main memory to 128 bytes (the size of MPE data cache line) and used the DMA compiler intrinsics to transfer the data. We measured the bandwidth of the DMA transfers with various amounts of CPEs and various sized data blocks.

**Results**: First, Fig. 3.13 shows the STREAM benchmark results of the *PE* mode, which are much more complete and preciser than the publicly available data. Second, the *BCAST* mode has a STREAM Copy bandwidth of 446.1 GB/s, 16X higher than that of the *PE* mode (27.9 GB/s). This was due to avoided the 63 redundant transfers. Third, as the data transfer

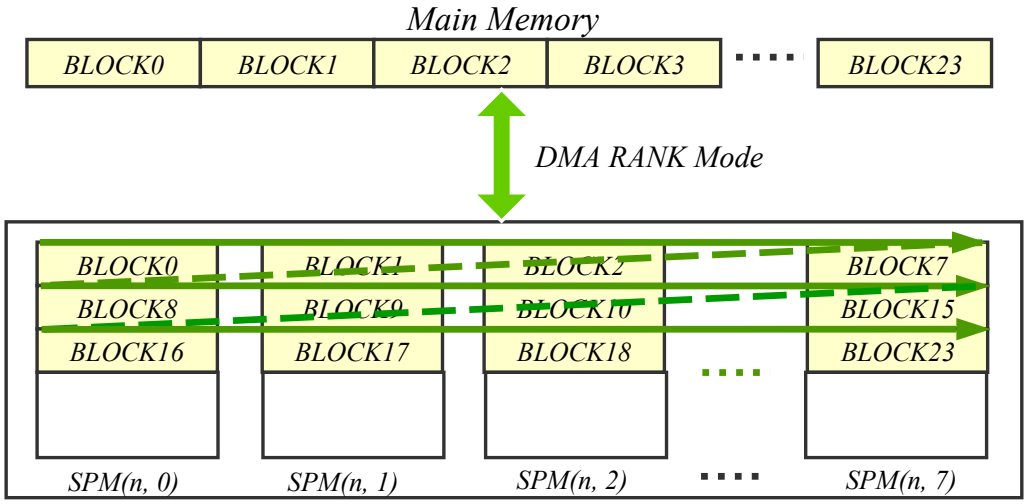



Fig. 3.12 The RANK mode of DMA can gather/scatter data blocks in the row-major order from/to the 8 CPEs in the same row.

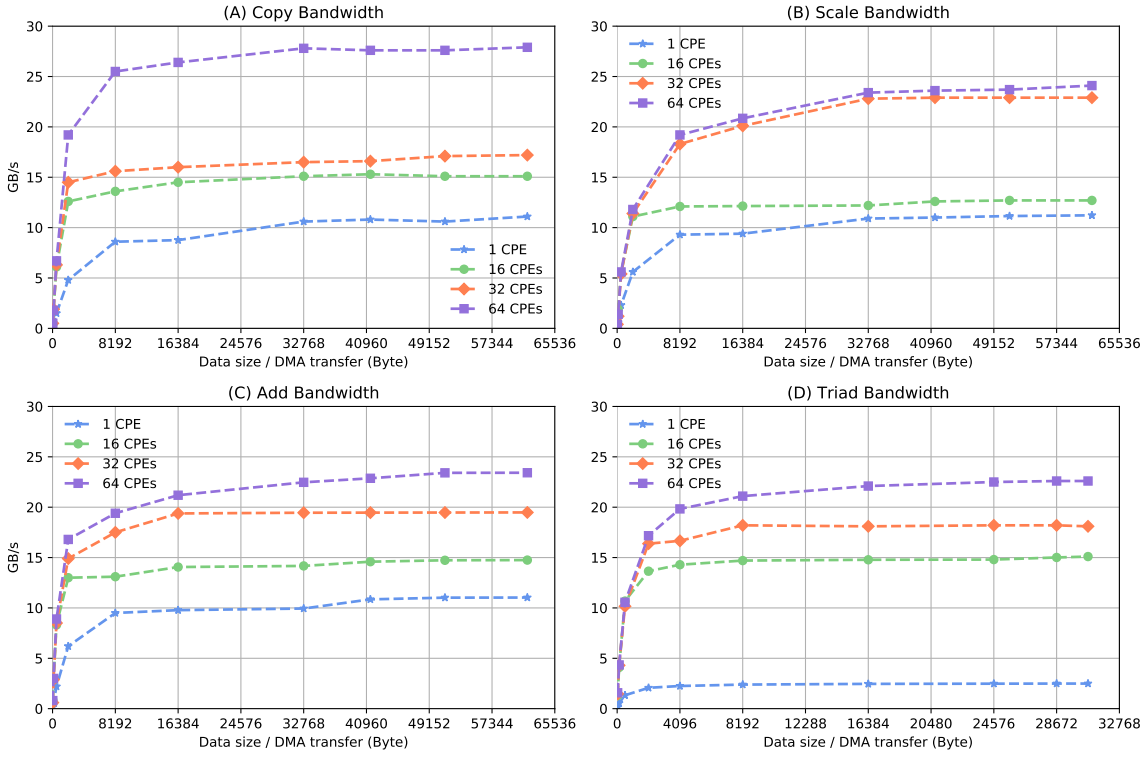



Fig. 3.13 The STREAM bandwidth results for the DMA PE mode. (a) the maximum Copy memory bandwidth is 27.9 GB/s; (b) the maximum Scale memory bandwidth is 24.1 GB/s; (c) the maximum Add memory bandwidth is 23.4 GB/s; (d) the maximum Triad memory bandwidth is 22.6 GB/s.

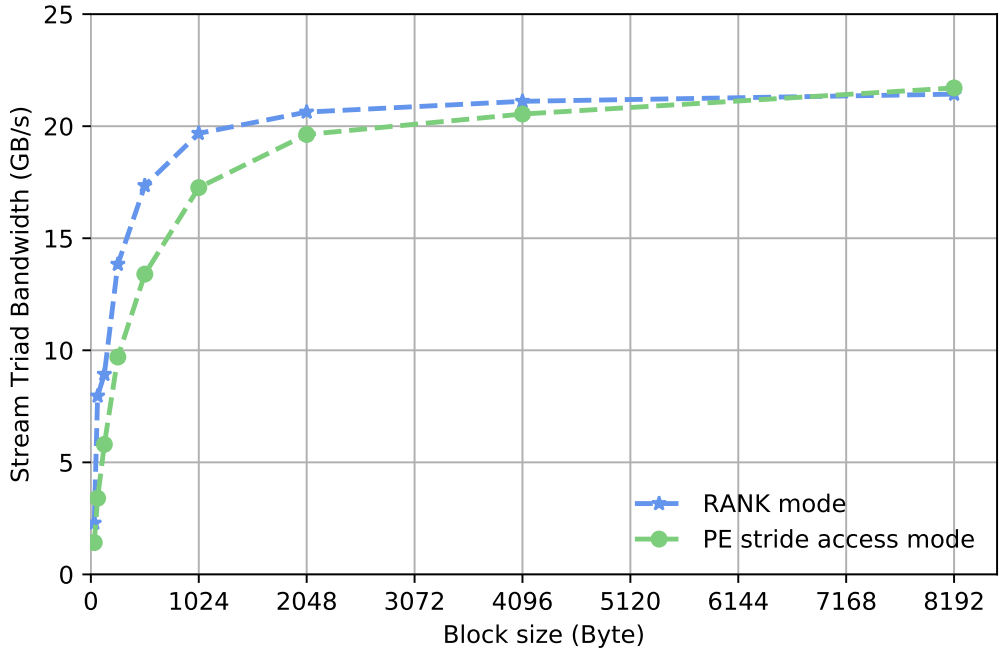



Fig. 3.14 Bandwidth comparison of the DMA *Rank* mode and the DMA *PE* mode.

pattern of the *RANK* mode can also be implemented with the unit-stride accesses *PE* mode, we compared the Stream Triad bandwidth of the two modes. As shown in Fig. 3.14, we found that the *RANK* mode outperformed the *PE* mode substantially when the data block size was smaller than 4 KB, while the two modes achieved the same bandwidth when data block size was larger than 6 KB.

### Global Load/Store Instructions

**Publicly available data**: the latencies of the global load/store instructions (`gld`/`gst`) are *177 cycles* and *278 cycles* respectively. The bandwidth of using the `gld` and `gst` instructions to access the main memory is unknown.

**Benchmark approaches**: first, as we cannot create data dependencies to measure the latency of the `gst` instruction, we only measured the latency of the `gld` instruction. For this measurement, we developed a *pointer-chasing* benchmark, which is similar to that used in the LDM evaluation but this time the data resided in the main memory. Second, to measure the bandwidth when using the two instructions, we implemented the STREAM benchmark with the `gld` and `gst` instructions.

**Results**: we observed that the latency of the `gld` instruction is dependent on the amount of CPEs that simultaneously access the main memory simultaneously (denoted as $n$). The

latency was *194-200 cycles* when $n <= 2$, while the latency increased to $84 \times n$ *cycles* when $3 < n <= 64$. We inferred that the maximum number of the `gld` instruction can be executed concurrently is 2; if the numbers become larger than 3, all `gld` instructions had to be executed sequentially.

In addition, the STREAM benchmark results show extremely low bandwidths when using the two instructions. The results in Copy, Scale, Add, and Triad are only 3.88 GB/s, 1.61 GB/s, 1.45 GB/s, and 1.48 GB/s per core-group, respectively.

## 3.4 Implications for Performance Optimizations

### 3.4.1 Applying the Roofline Model to the SW26010

To identify the key programming challenge of the SW26010 processor, we applied the roofline model [24] on the processor (Fig. 3.15). As can seen, due to the high computing flops and the low memory bandwidth, the SW26010 processor has an extremely high arithmetic-intensity of 32.84 Flops/bytes — almost 4X higher than that of the Intel KNL. The high arithmetic-intensity renders the most of the scientific kernel memory-bounded. However, if we could explore the data locality among the 64 CPEs by efficiently using the RLC, the arithmetic-intensity can be dropped significantly to only 1.2 Flops/bytes. Therefore, it is appropriate to argue that adopting the RLC to reuse on-chip data is a critical step to achieve the ultimate performance on the SW26010.

### 3.4.2 Case study 1: Instruction Scheduling

Understanding the instructions issue orders of the in-order dual-issue CPE pipeline is the key to scheduling instructions for better ILP. We studied the case of optimizing the *Scale* code of the STREAM benchmark for the on-chip SPM (in Sec. 3.3.1). We analyzed the assembly code generated by the `sw5cc` compiler (Lst. 3.1), and then manually scheduled instructions (Lst. 3.2) to achieve better memory bandwidth through two steps. First, we found the 9 `vldd` instructions (line 2-9 of Lst. 3.1) only issue to the pipeline P1 while leave the pipeline P0 idle. We moved forward the `vmuld` instructions without data dependences to co-issue with the `vldd` instructions in the same cycles. Second, to avoid the pipeline stalls caused by the WAW data dependency between line 17 and line 20 of Lst. 3.1, we moved backward the `vstd` instruction to hide the *7-cycle* latency of the `vmuld` instruction. The optimized *Scale* code achieved 41.21 GB/s, which is 1.40X more memory bandwidths compared with the original version (29.36 GB/s).

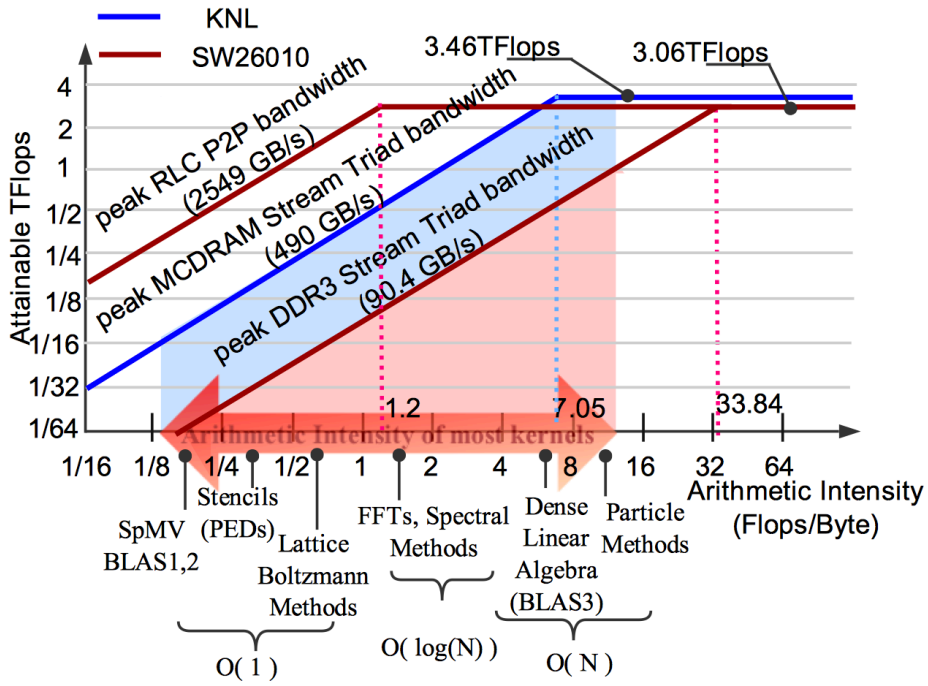



Fig. 3.15 An illustration of applying the roofline model on the SW26010 processor. As can seen, using the RLC to explore on-chip data locality can significantly reduce the processor's arithmetic-density from 33.84 Flops/bytes to only 1.2 Flops/bytes.

Listing 3.1 The original *Scale* version

```
1  vldd   $1,160($6)
2  vldd   $7,0($6)
3  vldd   $3,96($6)
4  vldd   $2,224($6)
5  vldd   $4,64($6)
6  vldd   $26,128($6)
7  vldd   $14,192($6)
8  vldd   $9,256($6)
9  vldd   $11,32($6)
10 vmuld  $1,$27,$24
11 vldd   $10,288($6)
12 vmuld  $7,$27,$22
13 vldd   $12,320($6)
14 ...
15 vmuld  $12,$27,$16
16 vstd   $25,288($0)
17 vstd   $16,320($0)
```

Listing 3.2 The optimized *Scale* version

```
1  vldd   $1,160($6)
2  vldd   $7,0($6)
3  vldd   $3,96($6)
4  vldd   $2,224($6)
5  vldd   $4,64($6)
6  vmuld  $1,$27,$24
7  vldd   $26,128($6)
8  vmuld  $7,$27,$22
9  ...
10 vldd   $12,320($6)
11 vmuld  $9,$27,$20
12 vstd   $24,0($0)
13 vmuld  $26,$27,$19
14 vstd   $22,0($0)
15 vmuld  $14,$27,$17
16 ...
17 vstd   $16,320($0)
```

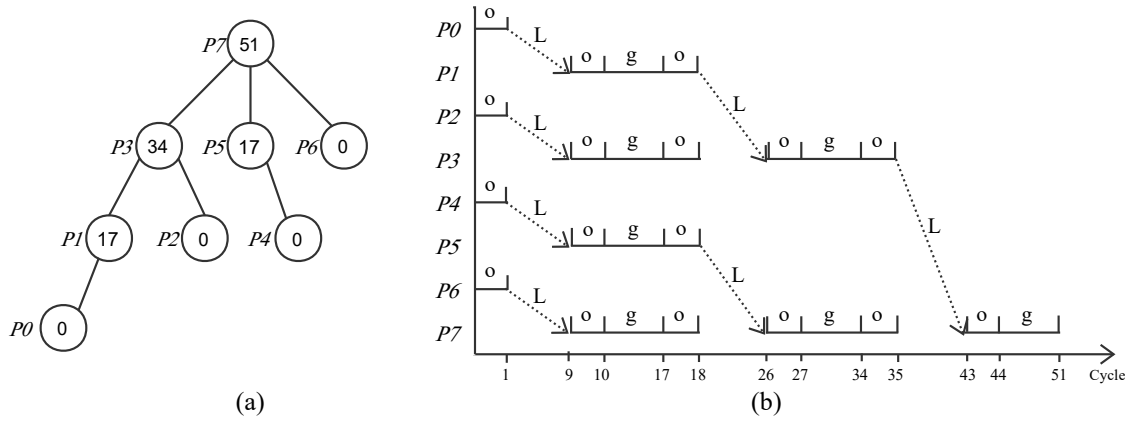



Fig. 3.16 Designing a row/column-based reduction operation with the logP model. (a) the optimal reduction tree for $l$=8, $g$=7, $o$=1, and $p$=8. (b) the activity of each CPE over time. The total latency is *51 cycles*.

### 3.4.3 Case study 2: Designing the Reduction Mode of RLC

The precise latencies of the instructions measured by our *swCandl*e benchmark can contribute to the design of efficient algorithms on the SW26010. We studied the case of designing the reduction mode of RLC. As the reduction mode is not supported in hardware, we implemented it with the P2P mode RLC. A straightforward implementation is sending 7 concurrent P2P messages to the first CPE in the row/column, and then adding the 8 partial sums one by one. As such, the latency of the implementation consists of 7 concurrent RLCs (each requiring *10 cycles*) and 7 sequential additions (each requiring *7 cycles*), which makes a total latency of $10+7*7=59$ *cycles*. This suggests that the performance bottleneck of this straightforward implementation is the 7 sequential additions (*49/59 cycles*).

To decrease the cost of the additions, we used the measured latencies to design an optimal reduction operation with the LogP model [68]. The main parameters required to build the LogP model are the upper bound latency $l$, the overhead $o$, the gap $g$, and the number of processor $p$. As mentioned in Sec. 3.2, the latency to transfer register data through the on-chip network $l$ is *8 cycles* (in fact, it is *8-9 cycles* depend on different receiver CPEs; we simplified it as *8 cycles* due to the design outcome remaining same), the overhead of putting/receiving register data to/from the network $o$ is *1 cycle*, the gap $g$ is *7 cycles* for one addition, and the number of CPEs $p$ is 8 for a row/column. As shown in Fig 3.16, we built the optimal reduction tree for the reduction on a row/column. The total latency of the reduction is $3*(10+7)=51$ *cycles*. The reduction on the $8\times 8$ CPEs of a single core-group requires twice as that on a row/column, resulting in a total latency of $2\times 51=102$ *cycles*.

| Architectural Features | | | Publicly Available Data | Our Benchmark Results |
|---|---|---|---|---|
| CPE pipeline | Instruction | latency (cycles) | N/A | Quantified |
| | | Issue order b/w pipelines | N/A | Out-of-order |
| RLC | P2P | Latency (cycles) | 10 | 10-11 |
| | | Bandwidth (GB/s) | N/A | 637.25 per core-group |
| | BCAST | Latency (cycles) | 14 | 10-11, the same as P2P |
| | | Bandwidth (GB/s) | N/A | 1115.19 per core-group |
| | *n*-hop Static routing | Latency (cycles) | N/A | *n* times as the P2P |
| | Dynamic Routing | Latency (cycles) | N/A | 1st option is the fastest |
| Memory | SPM | Latency (cycles) | 4 | 4 |
| | | Bandwidth (GB/s) | 47.9 | Copy:43.7 Scale: 41.2 Add: 38.6 Triad: 38.5 |
| | DMA | Bandwidth (GB/s) | 30.94 | PE mode: 27.9 BCAST mode: 446.1 RANK mode: 23 |
| | Global load | Latency (cycles) | 177 | 194~200 n<=2 84*n, n>=3 (n: the number of CPEs involved) |
| | | Bandwidth (GB/s) | N/A | Copy:3.88 Scale: 1.61 Add: 1.45 Triad: 1.48 |

Fig. 3.17 The summary of the key findings revealed by the *swCandle* benchmark suite.

## 3.5 Summary

The inadequate public information of the SW26010 processor's micro-architecture prevents global researchers from improving performances on the TaihuLight supercomputer. To address this issue, we developed the *swCandle* micro-benchmark suite to explore three major architectural features which significantly impact the performance: the CPE pipelines, register communications, and the memory access approaches. The benchmark revealed some unanticipated findings beyond the publicly available data. For instance, the instruction issue order in between the in-order dual-issue pipeline is out-of-order; the broadcast mode of register communications has the same latency as the peer-to-peer mode. In Tab. 3.17, we summarized the key findings of our study compared with the publicly available data. Moreover, we applied the roofline model, with the key parameters that we obtained from measuring the processors using the benchmark suite, to identify the key programming challenge of the SW26010 processor. Finally, based on the benchmark results, we proposed a systematic guideline for performance optimizations on the SW26010 and instantiated the guideline with two cases.

These findings can greatly benefit to the performance optimizations and modeling on the home-grown processor. In addition, in this study we developed a methodology that infers the processors micro-architecture designs from the benchmark results. This methodology can also be applied on other processors lacking of public information, such as the Matrix-2000 accelerator equipped in the Tianhe-2A supercomputer [69].

# Chapter 4

# Optimizing Compute-bound Kernels

In the previous chapter, we evaluated the three key architectural features of the SW26010 processor. We found the processor is high memory-bounded with the arithmetic-intensity of 33.X, almost 4X higher than Intel KNL. We applied the roofline model to the processor and found that the processor made the most scientific kernel memory-bounded (See Figure 3.15), even the normally compute-bound kernel DGEMM. This implies the significant programming challenge on the processor.

In this chapter, we evaluated two typical compute-bound kernels on the SW26010 processor: N-body and DGEMM. In **Section 4.1**, we optimize the N-body kernel. In **Section 4.2**, we develop a RLC-friendly algorithm for DGEMM. Based on the two optimization results, we summarize our key findings in **Section 4.3**.

## 4.1 Direct N-Body Simulation

### 4.1.1 Basic Algorithm

A N-body simulation numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body [48]. The N-body simulation can be classified into two types: the direct and the fast approach. The former is a brute-force technique that calculates the all-pairs interaction of $N$-bodies. The latter uses hierarchical domain decomposition of the bodies, such as the Barnes-Hut method [70], the fast multipole method (FMM) [71], and the particle-mesh methods [72]. In this section, we focus on the direct approach of the N-body simulation due to its simplicity. The optimizations we applied on the direct N-body simulation can be also used to the fast N-body simulation.

**Algorithm 4.1** The direct N-body algorithm

1: **for** all bodies $i$ **do**
2: **for** all bodies $j \neq i$ **do**
3: $(\triangle x, \triangle y, \triangle x) \leftarrow (x_i - x_j, y_i - y_j, z_i - z_j)$
4: $\gamma \leftarrow ((\triangle x)^2 + (\triangle y)^2 + (\triangle z)^2 + \varepsilon$
5: $s \leftarrow 1/\sqrt{\gamma}$
6: $s^3 \leftarrow s \cdot s \cdot s$
7: $(a_x, a_y, a_z) \leftarrow (a_x + s^3 \cdot \triangle x, a_y + s^3 \cdot \triangle y, a_z + s^3 \cdot \triangle z)$
8: **end for**
9: **end for**

Alg.4.1 presents the basic algorithm of the direct N-body kernel, where $x, y, z$ are the body coordinates, and $a_x, a_y, a_z$ are the accelerations in each direction. $s$ is the distance between body $i$ and body $j$, $\varepsilon$ is the smoothing factor, and the body mass is set to 1.

As presented, each single pairwise interaction (line 3-7 of Alg.4.1) requires **19** floating-point operations: specifically, 3 subtractions (line 3), 3 multiplies and 3 adds (line 4), 1 square root and 1 divide (line 5), 2 multiplies (line 6), and 3 multiplies and 3 adds (line 7). Notably, although on the SW26010 both the square root and division instructions require more cycles than the normal arithmetic operations, such as FMA, we still assigned a cost of 1 flop each. Meanwhile, with appropriate data reuse, each single pairwise interaction only needs 3 loads for the coordinates of body $j$ (line 3). As a result, the arithmetic intensity of N-body is about $19n^2/(3n \times 8) \approx 0.8n$ Flops/bytes, higher than that of the SW26010 if $n > 50$.

### 4.1.2 Initial Solution

As outlined in Alg.4.2, our initial attempt followed the owner-computes parallelization strategy. First, due to the independent computation of the pairwise interactions, we dis-

**Algorithm 4.2** The initial implementation of N-body

1: **for** block $\alpha$ in bodies $N$ **do**
2: DMA broadcast coordinates of body $i$ to each CPE
3: DMA broadcast accelerations of body $i$ to each CPE
4: **for** 64 CPEs **do**
5: DMA broadcast coordinates of body $j$ to each CPE
6: computing Alg.4.1 on block $\alpha$ with vectorization
7: **end for**
8: DMA updated coordinates of body $i$ to the main memory
9: **end for**

tributed the calculation across 64 CPEs by transferring the contents, such as coordinates and accelerations, with DMA.

Second, to compute a single pairwise interaction, 9 DP are required, including 6 DP for the coordinates of body *i* and *j*, respectively, and another 3 DP for the accelerations of body *i*. Furthermore, some spaces in the 64 KB LDM of CPEs are reserved by the operating system. Therefore, we found the maximum size of N-body that can be blocked in the CPE LDM is 768, which occur about $768(bodies) \times 9(DP) \times 8(bytes) \approx 55$ KB.

Finally, we vectorized the pairwise interactions with SIMD intrinsics and stored the coordinates and accelerations in a structure-of-arrays format.

### 4.1.3 Model-guided Optimizations

#### Semi-empirical Performance Model

We initially thought that achieving the optimum performance for N-body on the SW26010 would be straightforward due to its high arithmetic intensity. However, our initial implementation only achieved less than 10% efficiency — 66.4 GFlops in performance. Therefore, to identify the performance bottleneck and guide the optimizations, we developed a semi-empirical model [73] by combining the analytic method (counting the instruction latencies) and the empirical method (fitting the measured performance). The latencies of instructions are listed in Tab.3.1 and the latency of two successive instructions can be counted as in Equ.4.1.

$$\sum_{i}^{i+1} \mathscr{T} = \begin{cases} \mathscr{T}_{i,pl} + \mathscr{T}_{i+1,ab}, & \text{if } i, i+1 \text{ are independent} \\ \mathscr{T}_{i,ab} + \mathscr{T}_{i+1,ab}, & \text{if } i, i+1 \text{ are dependent} \end{cases} \tag{4.1}$$

As each line in the initial implementation has dependency on each other, we counted the total latency $\mathscr{T}_{total,init}$ as 132 cycles by simply adding the latency of each line. For instance, the latency of line 5 for the *rsqrt* operation $\mathscr{T}_{rsqrt,init}$ can be counted as $32 + 34 = 66$ cycles due to the dependency between the two math-intensive instructions, `vsqrtd` and `vdivd`.

This model can help us identify the potential performance bottlenecks of the N-body implementations. For instance, as Tab.4.1 shows, the *rsqrt* operations hamper the performance of the initial implementation as $\mathscr{T}_{rsqrt,init}$ is half of the total latency $\mathscr{T}_{total,init}$. In addition, according to Equ. 4.2, the model can predicate the optimized performance $P_{opt}$ by using the measured performance of the initial implementation $P_{init}$, 66.4 GFlops.

$$P_{opt} = \frac{\mathscr{T}_{total,init}}{\mathscr{T}_{total,opt}} \times P_{init} \tag{4.2}$$

### Loop Unrolling

Due to the in-order execution of the CPE cores, to minimize the pipeline stalls caused by read after write (RAW) data dependency, we manually unrolled the inner *j* loop. Due to the limited numbers of registers, 32, the maximum unrolling times is three.

According to our model, the latency of *rsqrt* with unrolling three times $\mathscr{T}_{rsqrt,unroll}$ is $(28+28+66)/3 = 40.6$ cycles and the total latency of the unrolling implementation $\mathscr{T}_{total,unroll}$ is 74.3 cycles. As presented in Tab. 4.1, our model thus predicates although the overall performance will be nearly doubled with the unrolling optimization, the bottleneck caused by *rsqrt* becomes even worse.

### Strength Reduction

Our model identified the two costly `vdivd` and `vsqrtd` instructions as the major performance bottleneck. Additionally, the Sunway math library has not supported the math-intensive *rsqrt* operation yet. Therefore, we used the software routine [74], as illustrated in Fig.4.1, to reduce the operation strength. In line 10, one iteration of approximation for the Newton's method can gain sufficient accuracy with the relative error up to 0.1%. To be consistent with the flops-rate of the initial implementation, we still count the software *rsqrt* as **2 Flops**.

Generally, the software approach costs longer latency than the corresponding hardware instruction on modern many-core processors, such as the NVIDIA GPU and Intel Xeon Phi. However, due to the inefficient hardware support, the strength reduction optimization with the software routine can reduce the *rsqrt* latency on the SW26010. According to our model, the latency of the software routine with unrolling three times $\mathscr{T}_{sqrt,soft}$ is $50/3 = 16.6$ cycles. In particular, the `vldd` and `vmuld` in line 5 take $(1+1+4)+(1+1+7) = 15$ cycles; the scalar shift right instruction `srlow` and `vsubd` in line 8 need $(1+1+1)+(1+1+7) = 12$ cycles; 2 multiplications and 1 fused-multiply-subtract (FMS) in line 10 require $1+1+3\times 7 =$

Table 4.1 The semi-empirical performance model for N-body

| **Implementations** | $\mathscr{T}_{rsqrt}$ | $\mathscr{T}_{total}$ | **Performance** |
|---|---|---|---|
| Initial (*init*) | 66 | 132 | 66.4 |
| Unrolling 3X (*unroll*) | 40.6 | 74.3 | 118 |
| if adding an EMU on *unroll* | 3 | 36.7 | 238 |
| Strength reduction (*soft*) | 16.3 | 50 | 175.3 |
| if adding more registers on *soft* | 9 | 27.9 | 314 |

* All the latencies $\mathscr{T}$ are in *cycles* and all the performances *P* are in *DP GFlops*. $P_{init}$ is measured while the remaining *P* are predicated by the model.

23 cycles. As presented in Tab. 4.1, our model predicates that the strength reduction optimization can significantly reduce the *rsqrt* latency by about 2.5X and thus boost the overall performance by about 1.5X.

#### Instruction Scheduling

The manual instruction scheduling requires programming in assembly language. However, coding in assembly from the scratch is challenging, even for the simple scientific kernels like N-body. To avoid the complicated calling conversations, we use the following three steps to code in assembly on the SW26010: First, we generate the initial assembly code with the compiler along with the "-O2" option. Second, we modify the kernel part of the assembly code. Third, we use the assembler and linker to execute the modified version.

As aforementioned, all floating-point arithmetic instructions can only be executed in *pipeline 0* and the data motion instructions are relegated to *pipeline 1*. Therefore, to balance the two single-issue pipelines, we need to manually mix these two kinds of instructions. List.4.1 lists the compiler-generated instructions for line 3-4 of Alg.4.1. Notably, the real assembly code is unrolled three times. For simplicity, we only list the code without unrolling. As shown in List.4.2, we manually forwarded the two subtractions (for $dx$ and $dy$) and the multiplication (for $dx^2$) to pair with the three loads (for $x, y, z$).

### 4.1.4 Results and Analysis

#### Theoretical Upper Bound

A common question in performance optimizations is when we should stop as the optimized performance is reaching the theoretical upper bound (i.e. the "speed of light"). We adopted

```
double software_rsqrt (double number) {
  long i;
  double x2, y;
  const double threehalfs=1.5D;
  x2 = number*0.5D;
  y  = number;
  i  = *(long *) &y;
  i  = 0x5fe6eb50c7b537a9-(i>>1);
  y  = *(double *) &i;
  y  = y*(threehalf-(x2*y*y));
  return y;}
```

Fig. 4.1 The software *rsqrt* routine

Listing 4.1 By the compiler

```
1  vldd   $1,ADDR_1 //x
2  vldd   $2,ADDR_2 //y
3  vldd   $3,ADDR_3 //z
4  vsubd  $1,$11,$1 //dx
5  vsubd  $2,$12,$2 //dy
6  vsubd  $3,$13,$3 //dz
7  vmuld  $1,$1,$21 //dx^2
8  vmad   $2,$2,$21,$21
9  vmad   $3,$3,$21,$21
10 vaddd  $21,$4,$21
```

Listing 4.2 Manual mixing

```
1  vldd   $1,ADDR_1 //x
2  vsubd  $1,$11,$1 //dx
3  vldd   $2,ADDR_2 //y
4  vsubd  $2,$12,$2 //dy
5  vldd   $3,ADDR_3 //z
6  vmuld  $1,$1,$21 //dx^2
7  vsubd  $3,$13,$3 //dz
8  vmad   $2,$2,$21,$21
9  vmad   $3,$3,$21,$21
10 vaddd  $21,$4,$21
```

the semi-empirical performance model (Sec. 4.1.3) to estimate the upper bound performance of the N-body kernel. We found the absolute latency of the instructions except the rsqrt operations of the single iteration of the N-body is 58 cycles. The absolute latency of hardware rsqrt is 66 cycles. By using the software routine, the absolute latency of the rsqrt can be reduced to 50 cycles. For each iteration of the N-body, we counted 19 DP Flops. Therefore, we estimated the initial performance for the N-body kernel on a single core-group to be $19/(58+50) \times 1.45 \times 4 \times 64 = 65.3$ GFlops. Due to the limited amount of vector registers available on each CPE, we can only unroll the loop for three times. The optimized peak performance of the N-body kernel then is estimated to be 195.9 GFlops.

### Experimental Results

As the maximum block size for a single CPE is 768, to achieve the best performance on 64 CPEs, we need $768 \times 64 = 49152$ bodies. As shown in Fig.4.2, the architecture-independent optimizations, such as loop unrolling and blocking (within the initial implementation), only gain a small fraction of the peak performance. On the other hand, targeting the lack of efficient hardware support for *rsqrt* and the two single-issue pipelines, the two architecture-specific optimizations, strength reduction and instruction mix, further boost the performance by nearly 2X. Our results shows we achieved 92% efficiency of the upper bound performance with 180 GFlops. Our analysis on the upper bound performance shows the performance of N-body on the SW26010 is bounded by the compute resource and no need to optimize its on-chip data communication.

In addition, as presented in Tab. 4.1, our model predicted the performances of the unrolling and strength reduction implementations well. More importantly, our model also predicted that if an efficient EMU could enable the *rsqrt* instruction to have 7-cycle absolute latency and 1-cycle pipelined latency (like FMA), or sufficient registers were available to

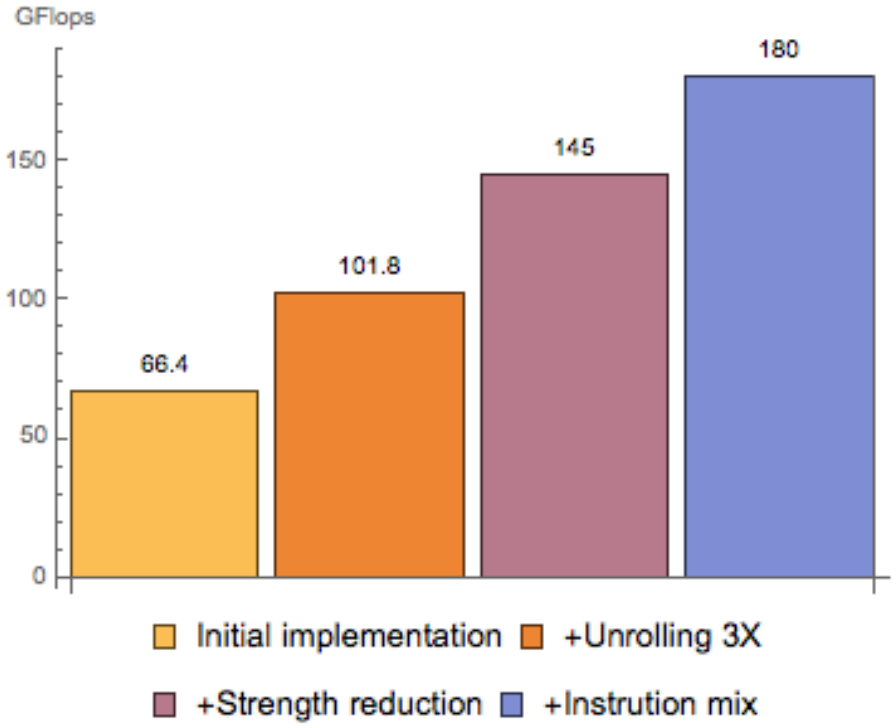



Fig. 4.2 Performance impact of the optimizations for N-body

unroll the strength reduction implementation 7 times, their performances would be even higher than that of our manually optimized implementation, 180 GFlops. The predication indicates that adding the two hardware features can simplify the programming complexity for math-intensive applications on the SW26010.

## 4.2 DGEMM

### 4.2.1 Basic Algorithm

The DGEMM kernel computes $C = \alpha AB + \beta C$, where $A$, $B$, and $C$ are $\mathtt{M} \times \mathtt{K}$, $\mathtt{K} \times \mathtt{N}$, and $\mathtt{M} \times \mathtt{N}$ matrices, respectively, while $\alpha$ and $\beta$ are scalers. Without loss of generality, we assume $\alpha = \beta = 1$ and that all the three matrices are stored in row-major order, thus the the computation of $C = AB + C$ can be achieved by:

$$c_{i,j} = \sum_{p=0}^{k-1} a_{i,p} b_{p,j} + c_{i,j} \tag{4.3}$$

```
DMA_in bk_A and bk_B to each CPE;
for(i=0;i<n;i++)
   for(j=0;j<n;j++)
      for(k=0;k<n;k+=4)
          simd_load(va,local_A);
          simd_load(vb,local_B);
          vc += va*vb;
      simd_store(vc,tmp);
      bk_C[i][j]+= tmp[0]+tmp[1]+tmp[2]+tmp[3];
DMA_out bk_C back to the main memory;
```

Fig. 4.3 The pseudocode of the initial implementation for DGEMM

### 4.2.2 Initial Solution

Fig.4.3 outlines the pseudocode of our initial attempt, which breaks the three matrixes $A$, $B$, and $C$ into $48 \times 48$ blocks to fit into the 64 KB of CPE LDM, as $3 \times 48 \times 48 \times 8 = 55926 < 64000$. We offloaded the loop nest from the MPE to 64 CPEs with the Athread library, transferred the sub-matrix blocks to the CPE LDMs with the DMA intrinsics, and vectorized the inner-most $k$ loop with the SIMD intrinsics.

To compute a $n \times n$ block $C$, each CPE requires for performance $2n^3$ floating-point operations, loading a $n \times n$ block $A$ and a $n \times n$ block $B$ from the main memory, and storing the block $C$ to the main memory. As a result, the arithmetic intensity of the initial implementation ($n = 48$) is $2n^3/(3n^2 \times 8) = n/12 = 4$ Flops/bytes. According to the roofline model [24] , the maximum attainable performance of the initial implementation on the SW26010 will be only $4 \times 22.6 = 90.4$ GFlops, about 12% efficiency.

### 4.2.3 RLC-friendly Algorithm

According to the profiling results, the initial implementation spends more than half time on DMA and thus cannot achieve sufficient arithmetic intensity to fully utilize the many-core processor. To reduce the off-chip memory traffic, we need to reuse the data that already reside in the 64 CPEs. Therefore, we designed the 4-layer RLC-friendly algorithm to increase the arithmetic intensity of DGEMM, as depicted in Fig.4.4,

The layer 1 partitions the three matrixes $A$, $B$, and $C$ into blocks sized of $8bm \times 8bk$, $8bk \times 8bn$, and $8bm \times 8bn$, respectively. On the border between layer 1 and layer 2, double-buffered DMA is used to transfer all the three blocked matrices to 64 CPEs, and store the blocks $C$ back in the main memory.

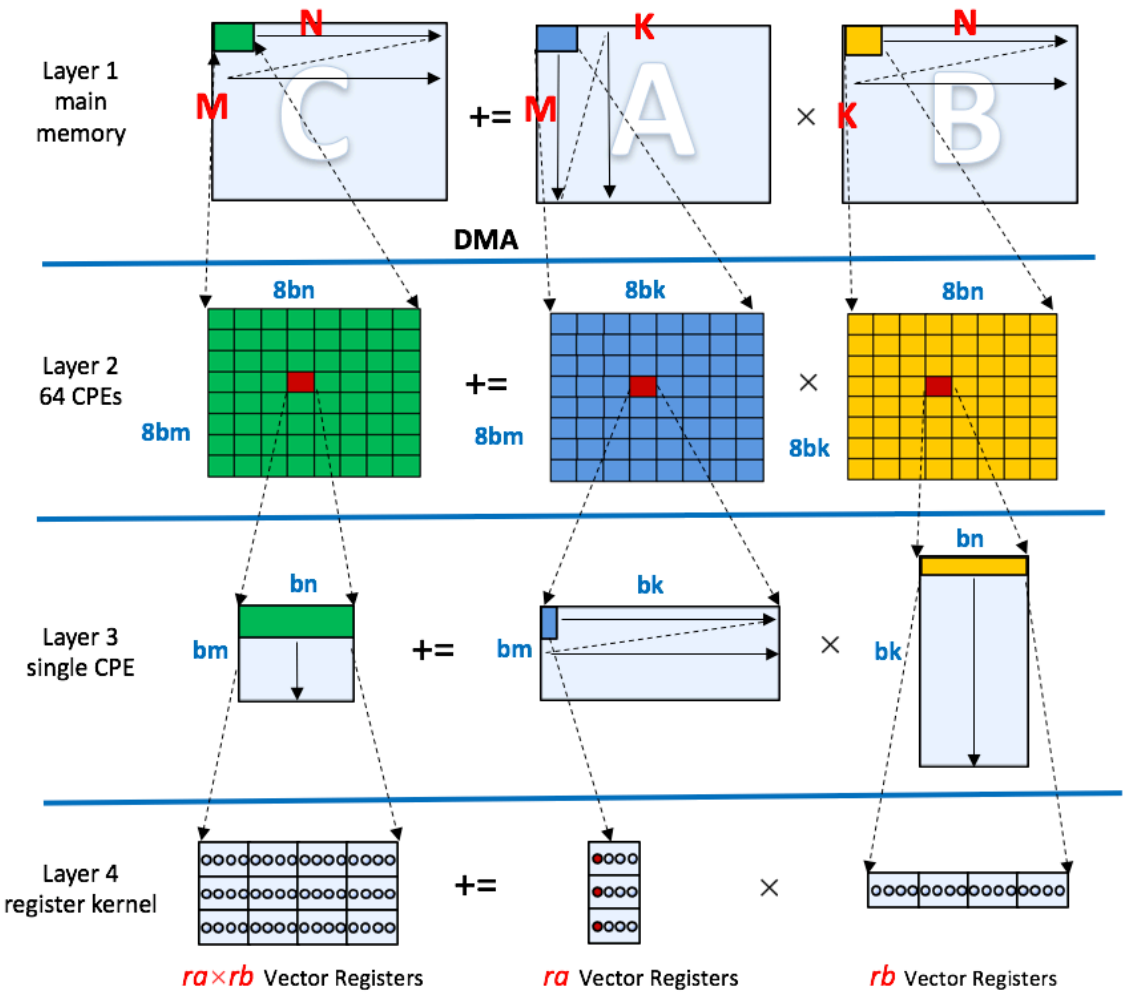



Fig. 4.4 Visual presentation of the RLC-friendly algorithm for DGEMM

Layer 2 is for the 64 CPEs and layer 3 is for a single CPE. Each CPE holds $bm \times bk$, $bk \times bn$, and $bm \times bn$ sized blocks of $A$, $B$, and $C$, respectively. To compute a block $C$, we use RLC to enable each CPE to obtain the required $A$ and $B$ from the registers of other CPEs.

Layer 4 (known as the *register kernel* in Goto [75]) dominates the entire computation time. In each iteration, the register kernel multiplies a $ra \times 1$ column of $A$ with a $1 \times rb$ row of $B$ to obtain a $ra \times rb$ matrix of $C$ in the registers of a CPE. The data of both $A$ and $B$ are received from other CPEs' registers via RLC. Additionally, register blocking is used for register double-buffering to hide the RLC latency.

In what follows, we describe the five key aspects of our RLC-friendly algorithm in a roughly bottom-up order.

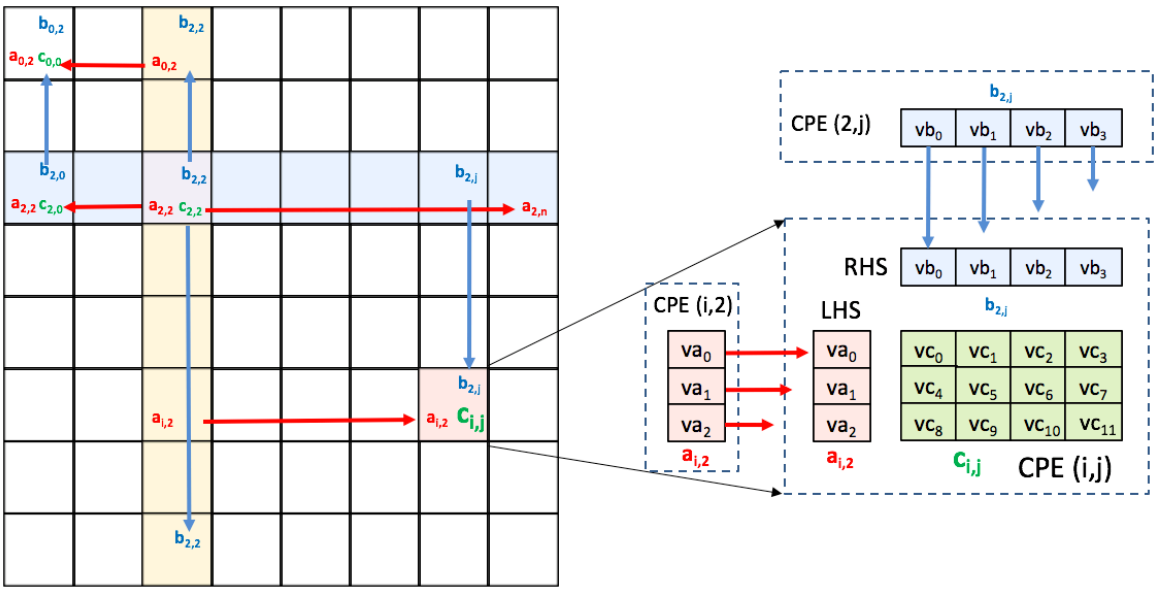



Fig. 4.5 Using RLC to compute the register kernels of the 64 CPEs in total 8 rounds

### Register Blocking

At layer 4, we look for the appropriate register blocking size. Each CPE has 32 vector registers but with one register being hardwired to 0; as a result only 31 vector registers are available for blocking. In addition, only $A$ and $B$ need to be blocked, as $C$ can continue adding the multiplication results from $A$ and $B$. Assuming $A$ and $B$ need $ra$ and $rb$ vector registers, respectively, $C$ thus needs $ra \times rb$ vector registers.

Therefore, the choice of $ra$ and $rb$ should meet $2(ra+rb)+ra \times rb \leqslant 31$. If we assume $ra \leqslant rb$, the maximum blocking factor is $ra = 3$ and $rb = 4$. We thus use 3 256bit vector registers to store the 3 64bit DP of $A$ (with SIMD extending by duplicating the 3 DP three times, respectively), 4 vector registers to store the 16 DP of $B$, and 12 vector registers to store the 48 DP of $C$.

### RLC for Register Kernel Computation

Using RLC to obtain the required $A$ and $B$ to compute $C$ for the register kernel is the central step of our RLC-friendly algorithm. As illustrated in Fig.4.5, the $c_{i,j}$ computation on any CPE $(i,j)$ among the 64 CPEs requires 8 $a_{i,n}$ resided in the CPEs of the same row $i$ and 8 $b_{n,j}$ resided in the CPEs of the same column $j$, thus $c_{i,j} = \sum_{n=1}^{8} a_{i,n} \times b_{n,j}$.

To obtain the 8 $a_{i,n}$ and 8 $b_{n,j}$, we use RLC to exchange register data among the 64 CPEs in total 8 rounds. In particular, for the $n$th round ($n$ is from 0 to 7), first, the 8 CPEs in the

Listing 4.3 The register kernel of the 48 CPEs only receiving data

```
getr va0_n
getr va1_n
getr va2_n
getc vb0_n
getc vb1_n
getc vb2_n
getc vb3_n
vmad va0,vb0,vc0,vc0
vmad va0,vb1,vc1,vc1
vmad va0,vb2,vc2,vc2
vmad va0,vb3,vc3,vc3
vmad va1,vb0,vc4,vc4
vmad va1,vb1,vc5,vc5
vmad va1,vb2,vc6,vc6
vmad va1,vb3,vc7,vc7
vmad va2,vb0,vc8,vc8
vmad va2,vb1,vc9,vc9
vmad va2,vb2,vc10,vc10
vmad va2,vb3,vc11,vc11
```

Listing 4.4 The register kernel of the CPE only sending data

```
ldder va0_n ADDR_0
ldder va1_n ADDR_1
ldder va2_n ADDR_2
vldc vb0_n ADDR_3
vldc vb1_n ADDR_4
vldc vb2_n ADDR_5
vldc vb3_n ADDR_6
vmad va0,vb0,vc0,vc0
vmad va0,vb1,vc1,vc1
vmad va0,vb2,vc2,vc2
vmad va0,vb3,vc3,vc3
vmad va1,vb0,vc4,vc4
vmad va1,vb1,vc5,vc5
vmad va1,vb2,vc6,vc6
vmad va1,vb3,vc7,vc7
vmad va2,vb0,vc8,vc8
vmad va2,vb1,vc9,vc9
vmad va2,vb2,vc10,vc10
vmad va2,vb3,vc11,vc11
```

row $n$ load 8 $b_{n,j}$, respectively, from their LDMs to registers, and similarly, the 8 CPEs in the column $n$ load 8 $a_{i,n}$. Second, via the 2D on-chip RLC network, the 8 CPEs in the row $n$ broadcast their $b_{n,j}$ to the 56 CPEs in the rest of 7 rows; the 8 CPEs in the column $n$ broadcast their $a_{i,n}$ to the 56 CPEs in the remaining of 7 columns. Third, the CPE ($i$,$j$) receives $a_{i,n}$ as the Left-Hand Side (LHS) from the CPE ($i$,$n$) in the same row, and $b_{n,j}$ as the Right-Hand Side (RHS) from the CPE ($n$,$j$) in the same column. Finally, the CPE ($i$,$j$) performs 12 FMA operations as $vc_0 += va_0 \times vb_0$ to compute $c_{i,j}$ in its *local* registers.

Due to the anonymous producer-consumer protocol that RLC uses, the data packages transferred via RLC do not contain the sender CPE's ID tag. Therefore, to ensure the correct sending and receiving order, we had to orchestrate the message sequences among the 64 CPEs manually, which significantly increases the programming complexity.

### Register Double-buffering

As RLC is asynchronous, we can use register double-buffering to hide the RLC latency with the register kernel computation. There are three types of register kernels in the 64 CPEs: 1) the 48 CPEs only receiving data; 2) the CPE that only sends data; 3) the remaining 15 CPEs

sending in row/column and receiving in column/row. As the third type can be considered as a mix of the first two types, for simplicity we only discuss the register kernels of the first two.

List. 4.3 outlines the register kernel of the 48 CPEs only receiving data, 7 independent RLC instructions (`getr` and `getc`) followed by 12 independent FMA operations. `va0` is the first vector of $a_{i,n}$ required for the current iteration while `va0_n` is the forthcoming `va0` for the next iteration.

Additionally, List. 4.4 outlines the register kernel of the CPE only sending data. The `ldder` instruction is a combination of loading a DP, extending to a 256bit vector by duplicating the 64bit DP three times, and `putr`. The `vldc` instruction is a combination of `vldd` and `putc`. These two instructions, however, only require *4* cycles to enable data available in registers for the *local* register kernel computation.

To overlap the RLC latency with the register kernel computation, we prefetched the data required for the next iteration, such as `va0_n`, The 7 independent RLC transfers in List. 4.3 cost $6+15=21$ cycles, and the 7 independent combination instructions in List. 4.4 need $6+4=10$ cycles to make registers data ready for the local register kernel computation. On the other hand, 12 independent FMA operations require $11+7=18$ cycles. As a result, the large fraction of the RLC latency in the 48 CPEs can be hidden as $21-18=3$, and the RLC latency in the CPE only sending data is perfectly hidden as $10<18$.

### DMA Double-buffering

Similarly to register double-buffering, we used DMA double-buffering to hide the DMA latency on the border between layer 1 and layer 2. As outlined in Fig.4.6, in the inner-most *j* loop, we prefetched the blocks *B* and blocks *C* required for the next iteration in order to overlap with the computation for the current iteration. Additionally, in the outer *i* loop, we double-buffered blocks *A*. To implement double-buffering, we split the matrices' block size into two halves, and, hence, doubled the blocks numbers.

```
for(k=0;k<K;k+=8bk)
    for(i=0;i<M;i+=8bm)
    DMA_in A(i+1,j);
        for(j=0;j<N;j+=8bn)
            DMA_in B(i,j+1);
            DMA_in C(i,j+1);
            C(i,j)+=A(i,k)*B(k,j);
            DMA_out C(i,j);
```

Fig. 4.6 The pseudocode of DMA double-buffering

**LDM Blocking**

At layer 1, we need to choose the optimum *bm*, *bn*, and *bk* for LDM blocking. First, due to DMA double-buffering, twice the number of the blocked sub-matrices *A*, *B*, and *C* need to be fitted into the 64 KB LDM, as shown in Equ.4.4.

$$2(bm \times bk + bk \times bn + bm \times bn) \times 8 < 64000 \tag{4.4}$$

Second, the inner-most *j* loop of Fig. 4.6 shows, to perform $2(8bm \times 8bn \times 8bk)$ flops on 64 CPEs, we need to load one $8bk \times 8bn$ block *B* and move the $8bm \times 8bn$ block *C* twice (both in and out). The cost of loading block *A* can be ignored as it is in the outer *i* loop. According to the roofline model [24], in order to achieve the near-peak performance, the arithmetic intensity of DGEMM should be as close as possible to that of the SW26010, as shown in Equ. 4.5.

$$\frac{Flops}{Bytes} = \frac{2(8bm \times 8bn \times 8bk)}{8(2 \times 8bm \times 8bn + 8bk \times 8bn)} \rightarrow 33.4 \tag{4.5}$$

Third, as shown at layer 3, *bm* and *bn* are the integer multiples of $ra = 3$ and $4 \times rb = 4 \times 4 = 16$, respectively. In addition, *bk* should be as large as possible to amortize the cost of data movement of the $bm \times bn$ blocks *C* between the main memory and CPE LDM.

As a result, to meet these three constraints, we chose the optimum $bm = 15$, $bn = 16$, and $bk = 121$, resulting in the three blocked sub-matrices loaded into LDM being *A*: $15 \times 121$, *B*: $121 \times 16$, and *C*: $15 \times 16$.

### 4.2.4 Results

In addition to *bm*, *bn*, and *bk*, we need `M`, `N`, and `K` to evaluate the DGEMM performance. As illustrated at layer 1, `M`, `N`, and `K` are the integer multiplies of *bm*, *bn*, and *bk*, respectively. Furthermore, the three matrices *A*, *B*, and *C* should be fitted into the 8GB main memory. We thus have $8 \times (M \times N + M \times K + K \times N) < 8000000000$. Due to the precisely software-controlled SPM, any `M`, `N`, and `K` meeting these two constraints achieved almost the same performance on the SW26010.

Table 4.2 Performances and efficiencies of DGEMM on the SW26010

| **Implementations** | **Performance** *(DP GFlops)* | **Efficiency** *%* |
|---|---|---|
| Initial | 70 | 9.1 |
| RLC-friendly | 679 | 88.7 |

As shown in Tab.4.2, our RLC-friendly algorithm achieved a 9.7X speedup of the initial implementation and 88.7% efficiency in a single core group of the SW26010 (i.e. 679 GFlops in performance). We also implemented the RLC-friendly algorithm without register double-buffering. However, the intermediate implementation only achieved 73.4% efficiency, indicating the significant performance impact of register double-buffering.

## 4.3 Summary

This section illustrates our attempt to tackle the programming challenge on the SW26010 processor with two typical scientific kernels. The first kernel, N-body, which is more compute-bound and math-intensive than DGEMM, revealed the slowness of transcendental operations on the SW26010. We thus carried out extensive computation-oriented optimizations, including strength reduction and instruction mix. In addition, our performance model indicates that adding hardware features, such as an efficient EMU or more registers, to the SW26010 can simplify the programing complexity for math-intensive applications.

We developed an RLC-friendly algorithm for the second kernel DGEMM. This manual solution achieved 88.7% efficiency in a single core group of SW26010 — i.e. 679 GFlops in performance. However, this is an extremely challenging approach, even for experienced programmers who have full knowledge of the SW26010. It is expected that a domain-specific language (DSL)-based compiler can be developed in the future, such as Physis [76] for stencils on the GPU, so as to automatically generate the correct RLC codes and ease the programming challenge.

# Chapter 5

# Optimizing the Memory-bound PCG

Despite the comprehensive optimizations, the single memory-bound kernel, such as SpMV, still cannot perform well on the SW26010 due to the limited memory bandwidth of the processor. Instead, the overall performance might be effectively improved by overlapping multiple memory-bound kernels within one application. This chapter aims to develop a sequence of technical solutions through the case of optimizing the Preconditioned Conjugate Gradient (PCG) [77]. Each iteration of PCG contains four types of key kernel: 1) two inner products (each requiring one all_reduce operation), 2) one preconditioner, 3) one SpMV, and 4) several AXPYs. PCG is a popular iterative method for solving sparse symmetric positive definite (PSD) systems of linear equations; it is notably used in the HPCG benchmark [25].

We begin this chapter with **Section 5.1**, where we analysis the bottleneck of the initial implementation of PCG and propose the RLC-friendly RNPCG algorithm for a single core-group of the SW26010. Next, in **Section 5.2**, we scale the RNPCG algorithm across multiple nodes of TaihuLight. Afterward, in **Section 5.3**, we implement RNPCG in OpenFOAM, a widely-used computational fluid dynamic (CFD) software. Fourth, we evaluate the performance and scalabilities of the optimized OpenFOAM on TaihuLight in **Section 5.4**. Finally, we summarize this chapter in **Section 5.5**.

## 5.1 PCG Optimizations

### 5.1.1 Basic Algorithm

As one of the well-known Krylov subspace methods, the PCG method [77] is used to solve symmetric positive definite (PSD) systems $Ax = b$. Alg. 5.1 shows the basic version of PCG. There, the $r_j$ vectors are computed residuals that in exact arithmetic are the Lanczos orthogonal basis vectors, scaled by constant factors. The $p_j$ vectors are basic vectors that

**Algorithm 5.1** Basic PCG to solve $Ax = b$

**Input:** A:$n \times n$ PSD matrix, $x_0$:initial guess, b:RHS, M:precond
**Output:** $x$: approximate solution
1: $r_0 \leftarrow b - Ax_0 \quad z_0 \leftarrow Mr_0 \quad \gamma_0 \leftarrow \langle z_0, r_0 \rangle$
2: **for** j= 1,2,.. until convergence **do**
3: **if** $j > 1$ **then** $\beta_j \leftarrow \gamma_{j-1}/\gamma_{j-2}$
4: **else** $\beta_j \leftarrow 0.0$
5: $d_j \leftarrow z_{j-1} + \beta_j d_{j-1}$
6: $w \leftarrow Ad_j$
7: $\delta_j \leftarrow \langle d_j, w \rangle$
8: $\alpha_j \leftarrow \gamma_{j-1}/\delta_j$
9: $x_j \leftarrow x_{j-1} + a_j d_j$
10: $r_j \leftarrow r_{j-1} - \alpha_j w$
11: $z_j \leftarrow Mr_j$
12: $\gamma_j \leftarrow \langle z_j, r_j \rangle$
13: test convergence (e.g., using $\beta_j$ )
14: **end for**

are *A-conjugate*: with respect to the inner product defined by $\langle p_i, p_j \rangle_A = p_j^T A p_i = 0$ when $(i \neq j)$.

Each PCG iteration (line 3-13) includes four kinds of kernels: one Spare Matrix-Vector Multiplication (SpMV) for coefficient A (line 6), one preconditioner application (line 11), two inner products (line 7 and 12), and three AXPY operations (line 5, 9, and 10).

Each kernel has a different communication pattern. The SpMV often requires only local communication. The preconditioner and two inner products involve global communication and requires participation of all processes. Vector operations AXPY can be calculated locally and do not require communication.

### 5.1.2 Initial Solution

Our initial attempt (Alg. 5.2) to parallelize PCG on a single core-group of the SW26010 (one MPE plus 64 CPEs) is based on the standard PCG. All of the matrices and vectors in the standard PCG are decomposed into 64 subdomains; namely, coefficient matrix A, preconditioner M, and other related vectors. Each subdomain is operated on a particular CPE.

The two inner products in the standard PCG (line 9 and line 16 of Alg. 5.2) require all_reduce operations (a combination of reduction and broadcast). To implement the all_reduce operations, we adopted a pair of DMA_reduce and DMA_broadcast through three steps. Taking the inner product in line 9 for example, first we used DMA to gather the $\delta_j$ on 64 CPEs (line 10). Next, we summed the 64 $\delta_j$ on the MPE before the result is used to

**Algorithm 5.2** Initial Parallel PCG on a Single Core-group

**Input:** A:$n \times n$ PSD matrix, $x_0$:initial guess, b:RHS, M:precond
**Output:** $x$: approximate solution

1: $r_0 \leftarrow b - Ax_0 \quad z_0 \leftarrow Mr_0 \quad \gamma_0 \leftarrow \langle z_0, r_0 \rangle$
2: **for** CPE $i = 0$ *to* 63 **do**
3:  **for** $j = 1, 2, ..$ until convergence **do**
4:  **if** $j > 1$ **then** $\beta_j \leftarrow \gamma_{j-1}/\gamma_{j-2}$ {Compute on MPE}
5:  **else** $\beta_j \leftarrow 0.0$
6:  *DMA_broadcast on* $\beta_j$
7:  $d_j \leftarrow z_{j-1} + \beta_j d_{j-1}$
8:  $w \leftarrow Ad_j$
9:  $\delta_j \leftarrow \langle d_j, w \rangle$
10:  *DMA_reduce on* $\delta_j$
11:  $\alpha_j \leftarrow \gamma_{j-1}/\delta_j$ {Compute on MPE}
12:  *DMA_broadcast on* $\alpha_j$
13:  $x_j \leftarrow x_{j-1} + a_j d_j$
14:  $r_j \leftarrow r_{j-1} - \alpha_j w$
15:  $z_j \leftarrow Mr_j$
16:  $\gamma_j \leftarrow \langle z_j, r_j \rangle$
17:  *DMA_reduce on* $\gamma_j$
18:  test convergence (e.g., using $\beta_j$)
19:  **end for**
20: **end for**

calculate $\alpha_j$ (line 11). Finally, we utilized DMA_broadcast to broadcast $\alpha_j$ on the 64 CPEs (line 12).

However, this initial attempt has three obvious weaknesses that prevent it from performing well on the SW26010 [78]. First, total 64 P2P DMA transmissions are required for the reduction operation, as the DMA reduction mode is not supported in hardware. Moreover, each of 64 DMA transmission is costly [44], due to the limited off-chip memory bandwidth of the SW26010. Finally, the DMA transmission cost cannot be hidden with computations due to unbreakable data dependencies.

### 5.1.3 RLC-friendly Algorithm

In order to find a solution to solve these three weaknesses, we evaluated various non-blocking PCG algorithms such as [56] [58] [59]. Among them, the non-blocking PCG (NBPCG) [59] requires the fewest vector operations to hide the all_reduce communications cost. Due to the limited numbers of the vector registers available on each CPE (only 32), we chose the NBPCG as the base for our PCG optimizations.

**Algorithm 5.3** RLC-friendly Non-blocking PCG on 64 CPEs

**Input:** A:$n \times n$ PSD matrix, $x_0$:initial guess, b:RHS, M:precond
**Output:** $x$: approximate solution

1: $r_0 \leftarrow b - Ax_0 \quad z_0 \leftarrow Mr_0$
2: $\gamma_0 \leftarrow \langle z_0, r_0 \rangle$
3: *RLC_allreduce on* $\gamma_0$
   $Z_0 \leftarrow Az_0$
4: **for** CPE $i = 0$ *to* 63 **do**
5:  **for** $j = 1, 2,..$ until convergence **do**
6:  **if** $j > 1$ **then** $\beta_j \leftarrow \gamma_{j-1}/\gamma_{j-2}$ {Compute on CPEs}
7:  **else** $\beta_j \leftarrow 0.0$
8:  $d_j \leftarrow z_{j-1} + \beta_j d_{j-1}$
9:  $s_j \leftarrow Z_{j-1} + \beta_j s_{j-1}$
10:  $\delta_j \leftarrow \langle d_j, s_j \rangle$
11:  $w \leftarrow Ad_j$
12:  $S_j \leftarrow Ms_j$ {Overlapping}
  *RLC_allreduce on* $\delta_j$
13:  $\alpha_j \leftarrow \gamma_{j-1}/\delta_j$ {Compute on CPEs}
14:  $x_j \leftarrow x_{j-1} + \alpha_j d_j$
15:  $r_j \leftarrow r_{j-1} - \alpha_j sj$
16:  $z_j \leftarrow z_{j-1} - \alpha_j S_j$
17:  $\gamma_j \leftarrow \langle z_j, r_j \rangle$
18:  $Z_j \leftarrow Az_j$ {Overlapping}
  *RLC_allreduce on* $\gamma_j$
19:  test convergence (e.g., using $\beta_j$)
20:  **end for**
21: **end for**

Our optimized RNPCG (Alg. 5.3) improves the initial implementation (Alg. 5.2) at two perspectives: 1) the all_reduce operations were fastened significantly by adopting the speedy on-chip communication RLC; 2) through instructions scheduling, the fastened all_reduce operations were overlapped with computations, which further improved the PCG performance. The sections below illustrate in detail how these two improvements were achieved.

### RLC_allreduce

The all_reduce operation includes two steps: reduction and broadcast. Reduction cannot be supported in hardware by RLC. Instead, RLC can only support, through hardware, the P2P and the broadcast modes. In order to address this limitation, we implemented the RLC_reduce with the P2P mode. As illustrated in Section 3.4.3, we adopt the LogP model

[68] to develop an efficient RLC_reduce implementation for a row/column in 51 *cycles* and for the $8 \times 8$ CPEs of a single core-group in 102 *cycles*.

On the other hand, supported by RLC through hardware, broadcast in the same row/column is a more straightforward step than reduction. Broadcast on a single core-group requires two rounds of row/column-based broadcast: first, a CPE broadcasts in the same row/column; then, the 8 CPEs in that row/column broadcast in the same column/row. As the latency of a single RLC_ broadcast is *10 cycles* [44], the RLC_broadcast for a single core-group takes *20 cycles*, which is about 5X faster than RLC_reduce.

### Instruction Scheduling

To overlap the two RLC_allreduce operations with the SpMV and the preconditioner kernels, parallelisms are required at both the thread-level and instruction-level. However, as the multi-threading is not supported on CPEs, the instruction-level parallelism becomes the only option. As aforementioned, each CPE has a dual-issue pipeline, P0 and P1. P0 supports computing operations of both the floating-point and integer, while P1 supports data motion (such as RLC instructions) and scalar integer operations. Because the `sw5cc` compiler cannot support RLC programing in C/C++, we had to program in assembly language and manually schedule instructions to hide the RLC_allreduce cost.

Without any loss of generality, we took a communication-computation overlapping conducted on CPE(4,0) as an example. The overlapping hides the all_reduce operation cost with the SpMV (see line 18 of Alg. 5.3). Fig. 5.1a lists the pseudo assembly instructions for the RLC_allreduce. In the reduction phase, the CPE(4,0) first receives the $\gamma_j$, then computes the sum locally, and finally sends the sum out (line 1-4 of Fig. 5.1a). In the broadcast step, it receives the final sum from CPE(0,0), and then broadcasts to the remaining 7 CPEs in row 4 (line 5-6 of Fig. 5.1a). Fig. 5.1b lists the pseudo assembly instructions for the SpMV kernel, which can be divided into three parts: the first two iterations (line 4-12 of Fig. 5.1b), the last two iterations (line 20-27 of Fig. 5.1b), and the remaining `n-4` iterations (line 13-19 of Fig. 5.1b).

As no data dependency exists between the RLC instruction `getr` (line 1 of Fig. 5.1a) and the integer data motion instruction `ldi` (line 8 of Fig. 5.1b), we can co-issue these two instructions to pipeline P1 and P0 simultaneously. Similarly, we co-issued the first four instructions of the RLC_allreduce with the four instructions in the first two iterations of the SpMV kernel, as well as the last two instructions of the RLC_allreduce with the two instructions in the last two iterations of the SpMV kernel. By using this instruction scheduling, we can hid the all_reduce communication with the SpMV computation.

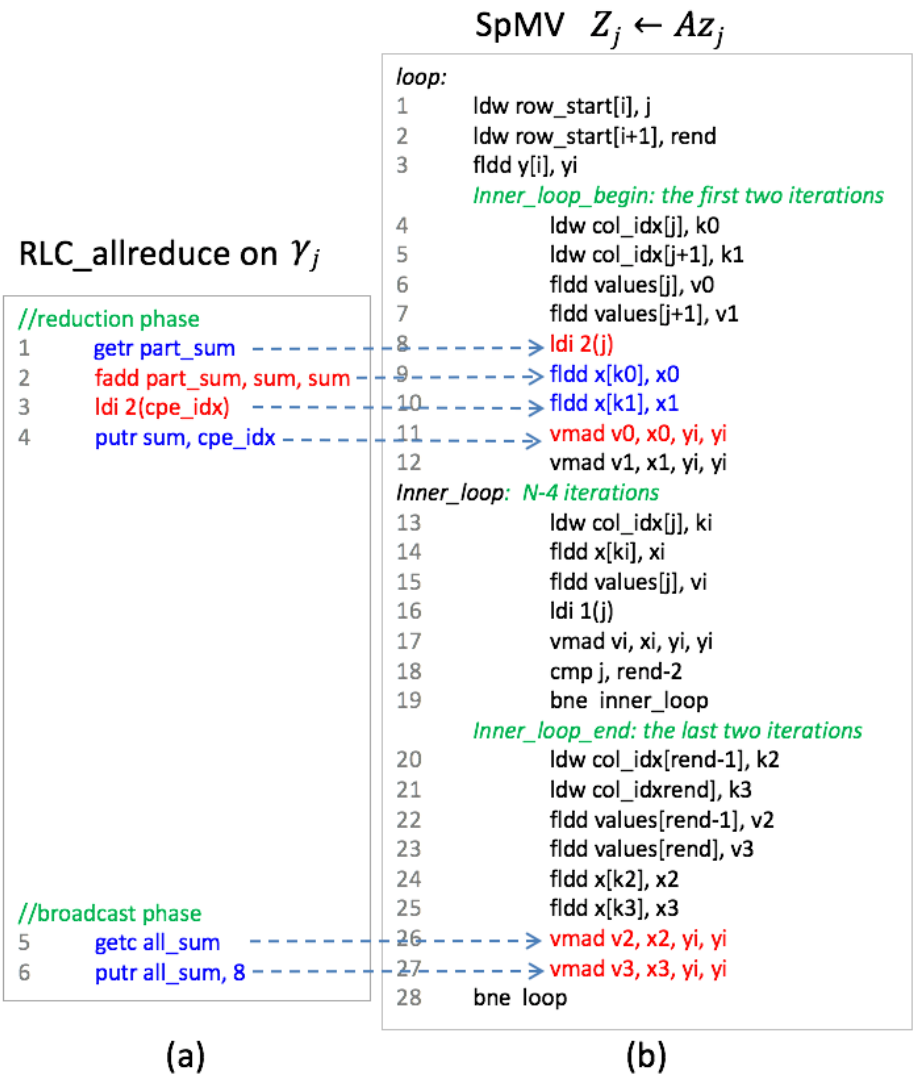



Fig. 5.1 Scheduling instructions to overlap the RLC_allreduce with the SpMV kernel. (a) the pseudo assembly code for the RLC_allreduce on the CPE(4,0); (b) the pseudo assembly code for the SpMV kernel.

### 5.1.4 Optimizations on the Key Kernels

Each PCG iteration contains four types of key kernel: 1) two inner products (each requiring one all_reduce operation), 2) one preconditioner, 3) one SpMV, and 4) several AXPYs. We have optimized the two all_reduce operations with RLC, and then overlapped them with the SpMV and preconditioner kernels. To improve the performance of the RNPCG further, we optimized the other three types of kernel, as illustrated below.

### Preconditioner

A preconditioner is a modification of an original linear system, and used to solve the linear system easier with an iterative method. In the preconditioned iterative method, the original linear equation $Ax = b$ is transformed into $M^{-1}Ax = M^{-1}b$ with the preconditioner M. Both the preconditioner M and the matrix $M^{-1}$A should not be computed explicitly. Instead, the matrix A and $M^{-1}$ will be iteratively solved. Notably, the preconditioning operation $M^{-1}$ should be inexpensive.

Simple preconditioners, such as Jacobi and Gauss-Seidel methods, have the low rate the convergence. For fast convergence, advanced preconditioners should be used, such as an *Incomplete LU* (ILU) factorization, where L is the lower and U is the upper triangular matrix. The ILU decomposition of SPD matrices is denoted by *Incomplete Chloesky* (IC) preconditioner, which is widely used in PCG solvers. As this work focuses on OpenFOAM, we chose the two fast preconditioners in OpenFOAM to work on: the Diagonal-based Incomplete Chloesky (DIC) and the Fast DIC [79]. These two fast preconditioners have the same algorithm but the latter moves the upper triangle calculation from the solution phase (needed in each iteration) to the construction phase (only needed once).

The straightforward implementation of the DIC preconditioner computes the construction phase on the MPE while the solution phase on 64 CPEs. Due to the data dependency in the forward and backward substitutions [57] of the solution phase, in order to exchange boundary data, the neighbor CPEs need to access the main memory by using the costly DMA.

To avoid the DMA communications in each PCG iteration, we proposed a localized version of the DIC preconditioner (LDIC). Alg. 5.4 shows the LDIC construction phase on the MPE. LDIC divides the DIC preconditioner into 64 subdomains on the MPE, then broadcasts them to the 64 CPEs with DMA. All data required for the forward and backward substitutions in the solution phase are kept in the LDM of each CPE. Thus each CPE can execute the LDIC solution phase independently. As there are no communications among CPEs in each iteration, we anticipate, despite it may more iterations to convergence, the LDIC preconditioner will boost the overall performance.

### SpMV

The second kernel SpMV requires irregular memory access and thus can hardly performance well on the SW26010 [44]. A full discussion of SpMV optimizations is beyond the scope of this paper; here, we only focus on one key SpMV optimization which adopts two different sparse data formats (Fig. 5.2): LDU and Sliced ELLPACK [80] .

**Algorithm 5.4** The Construction Phase of LDIC

**Input:** $a_{nn}$: matrix A, S: nonzero set $a_{ij} \neq 0$
**Output:** $d_{nn}$: LDIC preconditioner M

```
1: for i = 1 to n do
2:    set d_ii ← a_ii
3: end for
4: for CPE k = 0 to 63 do
5:    for i = n × k/64 to n × (k+1)/64 do
6:       set d_ii ← 1/d_ii
7:       for j = i+1 to n × (k+1)/64 do
8:          set d_jj ← d_jj − a_ii^2 * d_ii
9:       end for
10:   end for
11: end for
```

LDU is the default sparse data format in OpenFOAM. This format uses three arrays to store the non-zero elements in the lower triangle (L), the diagonal (D), and the upper triangle (U) of the coefficients matrix. In addition to these three arrays, LDU uses a further two arrays to hold the row and column index of each element. This five-array format is efficient in saving matrix results from finite volume discretization, but accessing the discrete column indices stored in the fifth array will cause costly DMA communications on the SW26010.

The other data format Sliced ELLPACK only uses three arrays. The first two arrays store the non-zero elements and their column indices; the third holds the indices of the first element in each strip, and put the total number of non-zero elements at the end. Additionally, the format is parameterized by the slice size $S$. Fig. 5.2 shows an example when $S = 2$. When $S = 1$, the format becomes the Compressed Sparse Row (CSR) format [57]. The advantage of this format is that it allows the value and column index to be stored in the column major order so that the access to them is contiguous. Therefore, in our optimizations, we converted the LDU to the Sliced ELLPACK and used $S = 4$ to enable vectorization.

### AXPY

*Alpha X Plus Y* is a computation $y := \alpha x + y$ involving two dense length $n$ vectors $x$ and $y$, and a scalar value $\alpha$. We optimized the 5 AXPY operations in the RNPCG (line 8-9, 14-16 of Alg. 5.3) with two methods, vectorization and loop fusion.

First, we vectorized these 5 AXPY operations with the 256-bit long vectors that can hold 4 double-precisions. We converted the default Array of Structure (AOS) format into the Structure of Array (SOA), then padded and aligned the data to be vectorized. We operated on the vector data with SIMD compiler intrinsics.

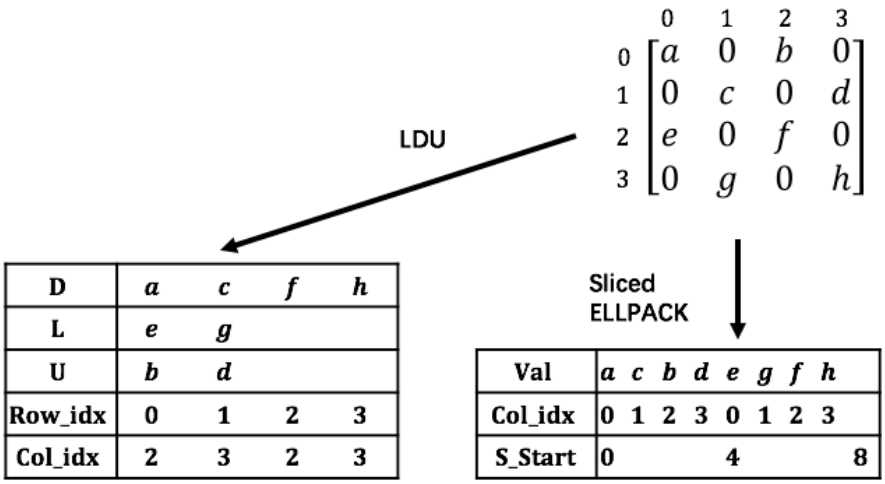



Fig. 5.2 An illustration of LDU and Sliced ELLPACK (strip size is 2) formats for a sparse matrix.

Second, we merged these operations into two pairs (line 8-9 and line 14-16) through loop fusion. We loaded each vector element once and performed multiple operations in the merged loops to reduce the cost of the loop initiation and avoid extra vector reads.

## 5.2 Scaling RNPCG on TaihuLight

### 5.2.1 Introduction to TaihuLight

Building with the SW26010 processor, the TaihuLight supercomputer has been the NO.1 in the TOP500 list since June 2016. It is based exclusively on processors designed and built in China. The complete system has a theoretical peak performance of 125.4 PFlop/s with 10,649,600 cores and 1.31 PB of primary memory. The supercomputer can be viewed as the following 6 levels, from the top-down perspective:

1. Node level: the whole system is consist of 40,960 compute nodes and each node includes one SW26010 processors.

2. Processor level: the processor provides 3.06 TFlops of peak performance both in single and double precision with an aggregated memory bandwidth of 130 GB/s. Each processor consists of 4 core groups.

3. Core group level: each core group includes one management processing element (MPE) and 64 computing processing elements (CPEs) in an 8-row times 8-column

manner. As indicated by the names, the MPE is used for management while the CPE is for computational purpose.

4. LDM level: each CPE has a 64 KB Scratch Pad Memory (SPM) as Local Data Memory (LDM). Direct Memory Access (DMA) is supported for data transportation across the LDM and the 8GB main memory. The STREAM triad bandwidth [81] of one core group is only about 22.6 GB/s [44], i.e. 0.35 GB/s per CPE. Thus the arithmetic intensity of the SW26010 is $756.2/22.6 = 33.4$ Flops/bytes, indicating that the many-core processor is highly memory-bound.

5. Register level: each CPE also has 32 registers, with one is hardwired to zero. unlike the CELL processor, CPE on the same core group can not access other CPE's LDM with DMA. Instead, a lightweight register-level communication called Register-level communication (RLC) is supported to share data among 64 CPEs in the same core group. The details of RLC is introduced in the below.

6. Pipeline level: each CPE has two execution pipelines (P0 and P1). P0 supports vectorized floating-point operations while P1 supports data motion operations.

### 5.2.2 The Scalable RNPCG

The barrier to the PCG scalability is the all_reduce operations. Our RNPCG overlaps the non-blocking all_reduce communications with computations at the core-group level. To scale the RNPCG on the multiple nodes of TaihuLight, the non-blocking all_reduce is required at the higher levels: the processor-level (each consisting of 4 core-groups) and node level (each consisting of 1 processor). We proposed a three-level non-blocking all_reduce to scale the RNPCG on TaihuLight. Taking the reduction phase of the all_reduce as an example, the phase requires the following three steps at each level.

- At the core-group level, we used RLC_reduce (Fig. 3.16) to gather the partial sums from 64 CPEs, then used DMA to transfer the result from the CPE(0,0) to the processor memory.

- At the processor level, as the data in the main memory can be shared among the 4 core-groups without extra memory copy, we added the partial sum of the 4 core-groups without any communications.

- At the node level, we adopted the non-blocking MPI communication available in MPI-3 [59]. We started `MPI_Iallreduce()`, then processed the computation-intensive kernel, such as the SpMV or preconditioner.

The broadcast phase, on the other hand, takes the reversed three steps. As a result, an all_reduce operation on TaihuLight has three partial cost at the three levels: 1) the cost of the RLC_allreduce and the two P2P DMA transmissions (one for reduction and the other for broadcast); 2) the cost of the three vector additions; and 3) the cost of the MPI all_reduce. Only the third partial cost at the node level will increase when the core-group count increases.

## 5.3 Implementing RNPCG in OpenFOAM

This section describes how we implemented the RNPCG in OpenFOAM on TaihuLight. We first introduce the background of OpenFOAM, then present the three key techniques used to implement the RNPCG as a linear solver of OpenFOAM.

One of the major challenges that China's home-grown TaihuLight supercomputer [82] faces is to build a rich software ecosystem, which comprises a multitude of tools, software and applications that is able to leverage the affordability of the hardware and to attract users from a wide range of domains. So far although several real-world applications including the two Gordon Bell Prize Winner [83] and three Gordon Bell Prize finalists [32] [64] [84] have been developed on TaihuLight, a variety of domain-specific software is still missing, which means the ecosystem of TaihuLight failed to attract potential users. Computational Fluid Dynamics (CFD) is one of the major domain areas that supercomputers support. The most widely-used domain-specific software in CFD is Open Source Field Operation and Manipulation (OpenFOAM) [85]. Yet, porting OpenFOAM onto TaihuLight is a challenging task, due to the highly memory-bound nature of both the software and the processor of the supercomputer.

### 5.3.1 Introduction to OpenFOAM

Open Source Field Operation and Manipulation (OpenFOAM) [85] is an open source object-oriented library for numerical simulations in continuum mechanics. It features a broad range of solvers, such as Laplace and Poisson equations, incompressible flow, multiphase flow, and user-defined models. Instead of being of a ready-to-use software, it is more likely as a framework for CFD users to build their own code.

OpenFOAM is designed for Intel CPUs, but porting work has been done on NVIDIA GPUs and Intel KNL. For example, the linear solvers have been well optimized on GPUs, including QR factorization[86], Cholesky factorization [87], sparse direct solvers [88], conjugate gradient and multi-grid solvers [89]. In addition, a special version of OpenFOAM is optimized for the high bandwidth memory on Intel KNL [90].

**Algorithm 5.5** The Construction Phase of FDIC

**Input:** $a_{nn}$: matrix A, S: nonzero set $a_{ij} \neq 0$
**Output:** $d_{nn}$: LDIC preconditioner M

```
1: for i = 1 to n do
2:     set d_ii ← a_ii
3: end for
4: for i = 1 to n do
5:     set d_ii ← 1/d_ii
6:     for j = i+1 to n do
7:         set d_jj ← d_jj − a_ii^2 * d_ii
8:     end for
9: end for
```

### Preconditioners

OpenFOAM provides 4 types of preconditioner and the PCG solver can use the first three:

- diagonalPreconditioner: a diagonal preconditioner.
- DICPreconditioner and DILUPreconditioner [79]: a diagonal Incomplete Cholesky preconditioner for symmetric and asymmetric matrices respectively.
- FDICPreconditioner: the faster version of the DIC Preconditioners. The fast version moves the upper triangle calculation from the solution phase (needed in each iteration) to the construction phase (only needed once). The construction phase of FDIC is shown in Alg. 5.5.
- GAMGPreconditioner: a geometric agglomerated algebraic multigrid preconditioner.

### Linear Solvers

OpenFOAM offers 7 linear solvers to solve linear systems:

- diagonalSolver: a diagonal solver for both symmetric and asymmetric problems.
- GAMG: a geometric agglomerated algebraic multi-grid solver.
- ICC: an incomplete Cholesky preconditioned conjugate gradient solver.
- PBiCG: a preconditioned bi-conjugate gradient solver for asymmetric matrices.
- PCG: a preconditioned conjugate gradient solver for symmetric matrices. This is the focus of this chapter.
- smoothSolver: an iterative solver using smoother for symmetric and asymmetric matrices based on preconditioners.

#### Standard Solvers

The linear solvers in OpenFOAM are invoked by the standard solvers. OpenFOAM does not offer a generic standard solver to solve all cases.As a result, to solve a class of problems, users need to choose a specific solver. These standard solvers can be divided into several categories, such as incompressible flow, heat transfer, and combustion. Moreover, the solver's name indicate the algorithm it used, e.g., the simpleFoam solver uses the SIMPLE algorithm, and pimpleFoam uses the PIMPLE algorithm. The complete list of OpenFOAM standard solvers is available online [91].

### 5.3.2 Compiling the Mixed C/C++ Code

Tab. 5.1 compares the features of two compilers available on TaihuLight. The `swg++` compiler is a ported version of GNC C++ compiler for software compatibility. The `sw5cc` compiler is a customized compiler for high performance but with limited C++ support. As OpenFOAM requires some advanced object-oriented features such as template classes and operator overloading those cannot be supported by `sw5cc`, in this work we use `swg++` to compile the OpenFOAM framework on the MPE, and use `sw5cc` to compile the PCG kernels converted in C on the 64 CPEs.

Compiling and linking programs with the `sw5cc` compiler need the following three steps:

1. compile the source code with `sw5cc` into the relocatable object file on MPE;
2. compile the source code the `sw5cc` compiler into the relocatable object file on the CPE cluster. This step is optional if the code is only running on the MPE.

Table 5.1 Comparison of the two compilers available on TaihuLight

| **Features** | **swg++ (vxx23232)** | **sw5cc (v5.421-495)** |
|---|---|---|
| Performance | slow | fast |
| Compatibility | good | bad |
| Static library loading | yes | yes |
| Dynamic library loading | **no** | **no** |
| MPE support | yes | yes |
| CPE support | no | yes |
| C/Fortran support | full | full |
| C++ support | good | **limited** |

3. use the `sw5ld` linker to the link the relocatable object files statically both the MPE and CPE clusters, along with the additional necessary libraries files. Please note the latest version of `sw5ld` has only supported static libraries, not dynamic libraries yet.

OpenFOAM is written in C++, however, as aforementioned, C++ is not supported by the `sw5cc` compiler. We thus rewrote the four key RNPCG kernels into C. As as result, the modified OpenFOAM mixed with C++/C source code. To address this compiling issue, we respectively compiled the OpenFOAM framework with the `swg++` to run on a MPE, and compiled the RNPCG kernels with the `sw5cc` to run on 64 CPEs.

### 5.3.3 Linking Static Libraries

In order to add user-customized features, OpenFOAM must heavily reply on dynamic libraries. However, the dynamic libraries are supported by neither the `swg++` nor `sw5cc`. A straightforward solution to this issue is linking up all of 1743 re-locatable object files of OpenFOAM as static libraries. However, this naive solution would greatly increase the file size of the executable binary toward 200MB. In order to keep the binary size small, we only manually linked the missing libraries as reported in error messages. However, identifying the missing libraries was time-consuming.

### 5.3.4 Scaling Standard Solvers

To scale OpenFOAM standard solvers at the node level, we need to adopt non-blocking MPI communications. However, the parallel communication library of OpenFOAM, *PStream*, only supports the blocking MPI communication. Therefore, we replaced the default blocking MPI communications with the non-blocking communications, such as `MPI_Iallreduce()` and `MPI_Waitall()`.

## 5.4 Performance Evaluation

### 5.4.1 Experimental Setup

#### Hardware

We evaluated the RNPCG performance on two test nodes (Tab. 5.2), a Sunway node and an Intel node; we tested the PCG scalabilities on TaihuLight.

Table 5.2 Specifications of the two test nodes. We chose the Intel E5-2695v3 CPU to compare with the SW26010, as the two processors were released in the same year.

| **Features** | **Sunway node** | **Intel node** |
|---|---|---|
| Processor | SW26010 | Intel CPU E5-2695v3 |
| Release year | 2015 | 2015 |
| Frequency | 1.45GHz (MPE&CPE) | 2.3GHz |
| Vector length | 256-bit | 256-bit |
| Cores | $4 \times (64 + 1) = 260$ | 14 |
| DP Flops | 700G per core-group | 18.4G per core |
| SPM/Cache | 64KB per CPE core | 192KB L1 |
| Memory | 32GB DDR3 | 96GB DDR4 |

### Software

On the Sunway node, we used the `swg++` v4.5.3 to compile OpenFOAM v3.0.1. This OpenFOAM version is not the most recent version but is, to date, the latest version that can be successfully compiled on TaihuLight. Moreover, we compiled the PCG key kernels written in C with the `sw5cc` v5.421. The optimization level of the `sw5cc` compiler was set at "-O2" instead of "-O3", in order to generate human-readable assembly code for further manual instruction scheduling. On the Intel node, we compiled with the same version of OpenFOAM (v3.0.1) with the Intel C/C++ compiler 2017_update1.

### Test Cases

We evaluated RNPCG performance with two standard OpenFOAM benchmarks [92]. The first benchmark is *Cavity*. We solved it with the standard OpenFOAM solver `icoFOAM`, and augmented the mesh density 400X and 6400X for the single core-group and strong scaling test, respectively. The second is *PitzDaily*. We solved it with `simpleFOAM`, and augmented the mesh density 25X for both the single core-group and strong scaling test.

For pre-processing, we used the OpenFOAM mesh generation utility `blockMesh` and the OpenFOAM domain decomposition tool `decomposePar`. Due to the limited memory capacity of each core-group (8GB), we conducted the pre-processing tasks on the Intel node, then transferred the data to TaihuLight for computations. All of the computations were performed in double-precisions.

Table 5.3 Breakdown latencies of the RLC_allreduce and the DMA_allreduce on the single core-group.

| **Apparoches** | **Reduction** *(cy)* | **Broadcast** *(cy)* | **All_reduce** *(cy)* |
|---|---|---|---|
| RLC | 102 | 20 | 122 |
| DMA | 6670 | 565 | 7235 |

### 5.4.2 Evaluation Methods and Results

#### RLC_allreduce vs. DMA_allreduce

We developed a benchmark in assembly language to identify the performance gap between RLC_allreduce and DMA_allreduce. The all_reduce operation includes two steps: reduction and broadcast. For the reduction step, as aforementioned, we adopted the reduction operation designed with the LogP model (Fig. 3.16) for RLC_reduce, and the 64 P2P DMA transmissions for DMA_reduce. For the broadcast step, we used the 2-round row/column-based RLC broadcast for RLC_broadcast, and the default DMA broadcast for DMA_broadcast. The data transferred by each all_reduce operation is a 256-bit long vector.

Tab. 5.3 shows the latency breakdown results for the two all_reduce operations. 1) The on-chip communication RLC was 59X faster than the off-chip communication DMA for the all_reduce operation. 2) For both RLC and DMA, the reduction steps were far slower (5X-11X) than the broadcast steps. This was because the broadcast was supported in hardware, while the reduction was not.

#### Single Core-group Results

We evaluated the performance impact of the RNPCG optimizations in the three test cases. As shown in Fig. 5.3, the "Intel" version is the standard PCG with the FDIC preconditioner on a single core of the Intel test node; the "Baseline" version is our initial implementation (in Alg. 5.2) with the FDIC on the single core-group of the SW26010; the "RNPCG" version is our non-blocking PCG with the optimization described in Sec. **??** on the single core-group.

Fig. 5.3 shows the elapsed time of the four key kernels of the three versions. 1) In all three test cases, the RNPCG can achieve a 6.3X-8.9X speed up compared with the baseline version. The cost of the all_reduce was reduced by RLC, then hidden with the preconditioner and SpMV kernels. For example, in *Cavity*, the two all_reduce operations optimized with RLC took only *0.32s* in total, and thus the cost can be fully hidden by the preconditioner (*1.42s*) and SpMV (*1.41s*) kernels. 2) We observed that our optimizations on the three key kernels improved their performance significantly. Specifically, the localized preconditioner can speedup the performance by up to 8.4X compared with the FDIC. The details will be

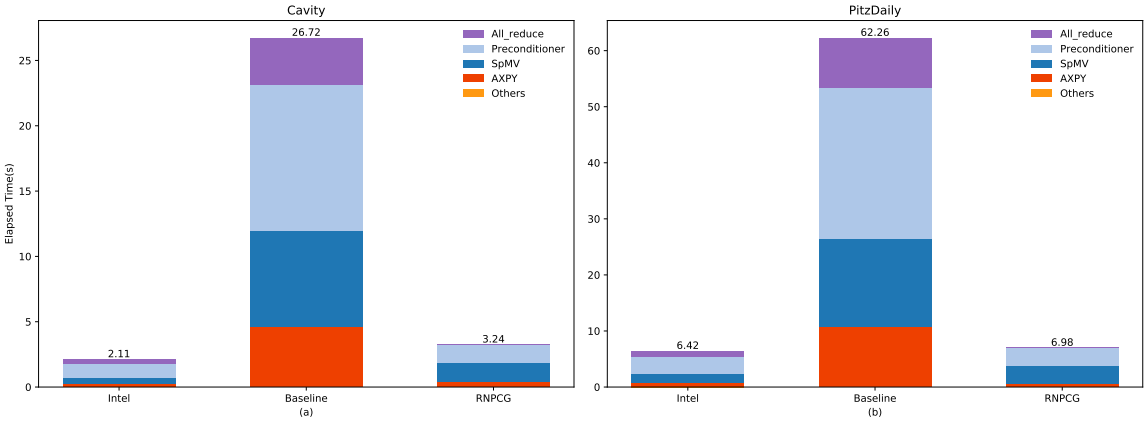



Fig. 5.3 The breakdown of the performance impact of RNPCG optimizations on a single core-group.

discussed later. The Sliced ELLPACK format can boost the SpMV performance by about 5X compared with the default LDU format. The vectorization and loop fusion can also improve the AXPY performance with 6X-16X. 3) We also found that the RNPCG on the single core-group of the SW26010 can only achieve 65%-93% of the performance of the standard PCG on the single core of the Intel CPU E5-2695v3. This was largely caused by the slow SpMV performance due to the limited off-chip memory bandwidth of the SW26010.

### Preconditioners

To compare our LDIC preconditioner with the two preconditioners used in OpenFOAM, *Jacobi* and FDIC, we evaluated their convergence rate and performance by using the same RNPCG on a single core-group. In Tab. 5.4, the test case names are followed by the problem size $n$ and the dimensions contained in the parentheses. The dimensions for *Cavity* are $x, y, z$, and for *PitzDaily* are $x_1$, $x_2$, $x_3$, $y_1$, $y_2$, $y_3$, $z_1$, $z_2$. "Iterations" is the iteration number to converge. "Setup" is the time to construct a preconditioner, while "Solution" is the time to solve a linear system.

The results in Tab. 5.4 show that, in both two cases, the LDIC was the fastest among the three preconditioners. Despite it requires 1.3X more iterations, the LDIC preconditioner performed about 3.4X-3.8X faster than the FDIC in total elapsed time (the sum of the setup time and solution time). This was because, as a localized preconditioner, LDIC can avoid costly DMA communications in each PCG iteration. This could also explain that, despite there being 3X more iterations, the simple preconditioner *Jacobi* could outperform the advanced preconditioner FDIC.

Table 5.4 Evaluating the three preconditioners with two test cases on a single core-group. Our LDIC preconditioner is the fastest in the solution time for solving a linear system.

| **Test cases** | **Precon.** | **Iterations** | **Setup** *(s)* | **Solution** *(s)* |
|---|---|---|---|---|
| **Cavity** | *Jacobi* | 2318 | 0.05 | 6.01 |
| 160,000 | FDIC | 725 | 0.067 | 11.17 |
| (400,400,1) | LDIC | 940 | 0.08 | 3.24 |
| **PitzDaily** | *Jacobi* | 2889 | 0.192 | 12.18 |
| 305,625 (90,900, | FDIC | 927 | 0.234 | 27.44 |
| 125,90,45,35,50,65) | LDIC | 1178 | 0.26 | 6.98 |

### Strong Scaling Result

To understand the performance impact of the non-blocking optimization for PCG, the baseline PCG we adopted for both the strong and weak scaling tests is the non-blocking implementation (Alg. 5.2) with the three key kernel optimizations (described in Sec. 5.1.4). Additionally, we ran each scalability test 10 times and reported arithmetic mean of the performance.

To evaluate the strong scalability of RNPCG, we fixed the total problem size of the two cases at (1600,1600,1) for *Cavity* and (90,900,125,90,45,35,50,65) for *PitzDailiy*, and then gradually increased the core-group count from 65 (single core-group) to 66,560 (1024 core-groups).

Figure 5.4 shows the elapsed time of the two PCGs in the two cases. The result shows 1) within one node (4 core-groups), the RNPCG is 1.6X-2X faster than the baseline PCG. This was because RLC_allreduce was far faster than DMA_allreduce (see Tab. 5.3); 2) from the 8 core-groups onward, RNPCG proved to be about 1.2X-1.3X faster than the baseline. This was because the cost of the MPI_allreduce communications increased while the computations per core-group decreased. 3) RNPCG cannot be scaled up any further when the core-groups grow to 512 (33,280 cores) and beyond, as the computation cost becomes too low to hide the increasing MPI all_reduce cost.

### Weak Scaling Result

To evaluate the weak scalability of RNPCG, we fixed the mesh size of the two cases on each core-group at (400,400,1) for *Cavity* and (90,900,125,90,45,35,50,65) for *PitzDaily*, and gradually increased the core-group count from 65 (single core-group) to 16,640 (256 core-groups).

Figure 5.5 shows the single-iteration runtime of the two PCG in the two cases. Within one node (4 core-groups), RNPCG was 1.6X-2X faster than the baseline PCG due to the fast RLC_allreduce. The RNPCG demonstrated a more consistent performance than the baseline

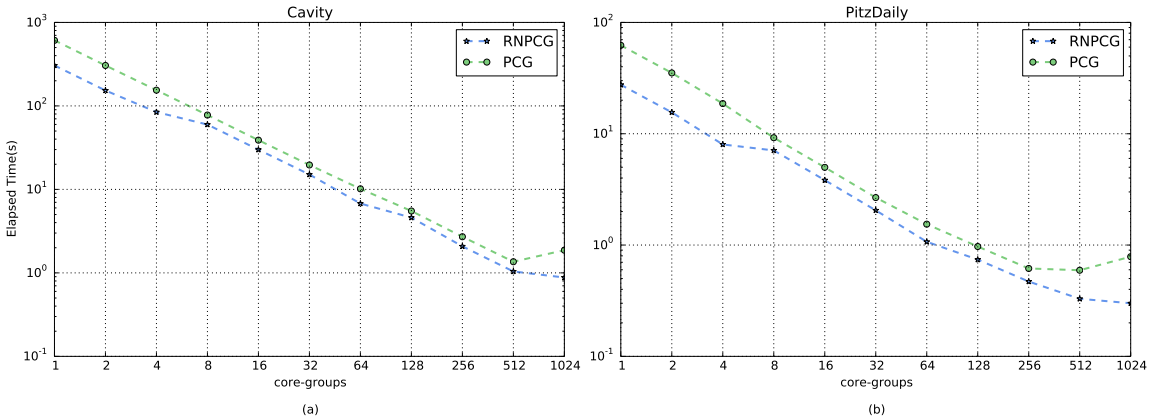



Fig. 5.4 Strong Scaling of Scalable RNPCG on TaihuLight

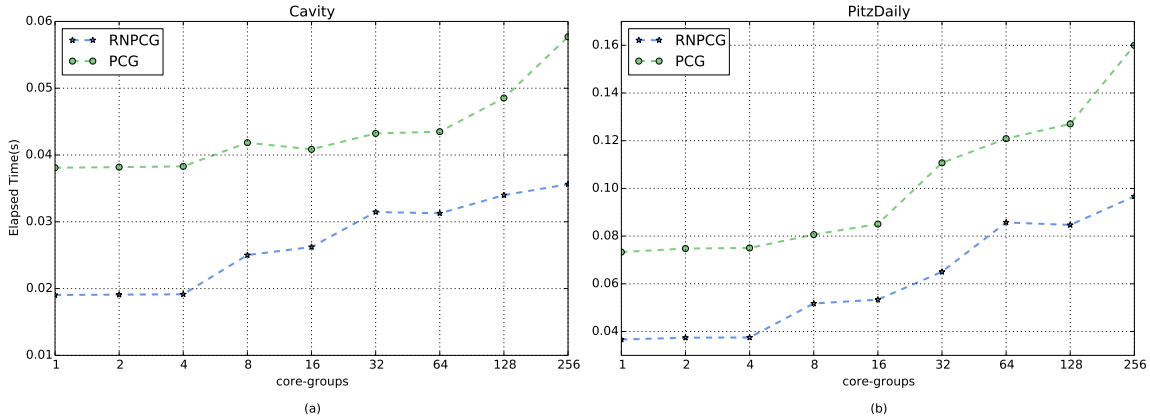



Fig. 5.5 Weak Scaling of Scalable RNPCG on TaihuLight

PCG, as the computations on each core-group, used to hide the all_reduce communication cost, remained the same while the core-group count increased.

## 5.5 Summary

The experimental results showed our RLC-friendly non-blocking optimization on PCG could achieve at most 8.9X speedup on the single-core group of the SW26010, and, moreover, scale up to 66,560 cores of TaihuLight. The PCG optimization requires three key building blocks: 1) a non-blocking algorithm to break data dependencies among the four PCG kernels; 2) minimized communication cost to be hidden with computations; and 3) instruction scheduling to overlap the communication-intensive kernels with the compute-intensive kernels. To fulfill the three requirements, we chose NBPCG [59] as the non-blocking algorithm, minimized the

all_reduce communication cost with RLC, and manually scheduled instructions to overlap the RLC_allreduce kernel with the SpMV and preconditioner kernels.

The prior work such as [44] indicated that due to the limited memory bandwidth of the SW26010, despite comprehensive optimizations, the single memory-bound kernel – such as SpMV – still cannot perform well on the processor. However, our study suggests the overall performance of an algorithm can be effectively improved by overlapping multiple memory-bound kernels within the algorithm, and, thus, provides a promising optimization approach for those multi-kernel memory-bound algorithms to achieve better performance on the SW26010.

Our contributions to the enrichment of the software ecosystem of TaihuLight is threefold. First, the RLC-friendly non-blocking optimization that we proposed for PCG can be broadly applied to the algorithms or applications containing multiple memory-bound kernels. Second, we have ported tens of OpenFOAM standard solvers (such as `icoFOAM` and `simpleFOAM`) with RNPCG, and these ported standard solvers will attract more CFD users to conduct simulations on TaihuLight, the current top supercomputer on the TOP500 list. Third, the experience that we have developed in compiling and linking OpenFOAM can be leveraged to port other complicated domain-specific software written in C++ (such as LAMMPS [**?** ]) onto TaihuLight.

Our next step is to fully port OpenFOAM on TaihuLight. OpenFOAM has 7 linear solvers and PCG is just one of them. We will port and optimize the rest of the linear solvers, such as GAMG and the PBiCG. As most of these linear solvers are multi-kernel memory-bound algorithms, we will apply the RLC-friendly non-blocking optimization, developed in this study, on them for better performance.

# Chapter 6

# Discussions

Based on the knowledge and experience gained from the performance evaluation and optimizations in the previous three chapters, in this chapter, we discuss in depth our thoughts on the SW26010 and the register communication. First, we propose a performance guideline in **Section 6.1** to guide the systematic performance optimizations on the SW26010. The guideline is concluded based on the benchmark results (Chapter 3) and optimization experience on the two compute-bound kernels (Chapter 4) and the memory-bound PCG algorithm (Chapter 5). Next in **Section 6.2**, we compare the SW26010 processor with the STI CELL processor, and discuss the reasons why the CELL failed and how the SW26010 fixed them. Finally, in **Section 6.3**, we discuss the potential architectural designs of the next generation Sunway processor, including the possible improvements to the register communication.

## 6.1 A Performance Guideline for the SW26010

Based on the programming challenges that we identified through the benchmark results and optimizations on the two compute-bound kernels and the memory-bound PCG algorithm, we propose a three-level programming guideline for the SW26010. Given the highly memory-bound processor, the central strategy of this guideline is to increase the arithmetic intensity (the *flops-to-bytes* ratio) of applications, that is, to decrease the number of the *bytes* at the first and second level, and increase the number of the *flops* at the third level.

### 6.1.1 CPE Level

The optimization objective at this level is to explore the instruction-level parallelism (ILP) with the in-order dual-issue pipelines. This should be achieved through the following four methods.

- Loops should be unrolled and instructions should be reordered for the in-order execution.
- Floating-point instructions and data movement instructions should be paired for the dual-issue pipeline.
- Due to the lack of the efficient hardware support, transcendental operations should be replaced with software routines. For example, the reciprocal square root (*rsqrt*), which is a combination of a division and a square root, requires *68 cycles* for the absolute latency and *58 cycles* for the pipelined latency. It can be replaced with a software routine [74], which has a pipelined latency of only *16.6 cycles* after being unrolled three times.
- Loops should be vectorized for the 256 bit vector units.

### 6.1.2 Inter-CPE Level

The programming challenge at this level is reusing data that has already reside in 64 CPEs with RLC. This challenge can be tackled by the following approaches.

- RLC-friendly algorithms should be designed to fit the scientific kernels into the default row/column-based communication pattern of RLC.
- RLC transmissions should be overlapped with computations to hide the RLC communication cost.
- Consecutive RLC SEND instructions should be co-issued with floating-point instructions to hide the RLC gaps.
- For algorithms requiring random data access, they should adopt the dynamic routing mode of RLC. For example, to shuffle the data among 64 CPEs in the Breadth-First Search (BFS) algorithm, H. Lin et al. [93] use the 32 CPEs in the column 0-3 as producers, the 16 CPEs in the column 4-5 as routers, and the 16 CPEs in the column 6-7 as consumers to dynamically route register data.
- The assembly language should be programed to avoid the unsolved performance issue with RLC intrinsics.

### 6.1.3 Core-group Level

The optimization objective at this level is to minimize the memory traffic overhead. This can be addressed by the following methods.

- Data should be blocked for the 64 KB CPE LDM to reduce memory traffic.
- The inefficient global load/store instructions should be eliminated. We found that some useless global load/store instructions can be generated by the *sw5cc* compiler [65] due to an unknown reason. Therefore, programers should check the assembly code and eliminate these inefficient memory access instructions so as to prevent them from slowing down the performance.
- Data transferred between the main memory and LDMs should use DMA, and the latency of DMA should be hidden by double-buffering.

## 6.2 The Fall of the CELL and the Rise of the SW26010

The notoriously difficult programming challenges and the radical architectural design of the SW26010 processor reminder us a similar processor, the STI CELL processor [14] [16]. In this section, we compare these two processors. First, we discuss the architectural design of the two processors. Both of them are heterogenous-many cores processor with some similar architectural features. Next, we analyzed the reasons caused the failure of the CELL processor and the success of the SW26010 processor in terms of both business and technical perspectives. Particularly, we pointed out that the key issue to cause the significant programming challenges on the CELL processor was the slow intra-chip communication due to the high latency of DMA. In contract, the SW26010 processor avoids this issue by using lightweight RLC to share on-chip data among cores.

### 6.2.1 Companions of the Two Processors

At the first impression, the two processors share many similarities in terms of their architectural designs. As shown in Table 6.1, both processors adopted the heterogeneous many-core architecture design with one management core and several compute cores. Each compute core has an in-order dual-issue pipeline and a small amount of software-managed SPM as its local data store. Moreover, they both adopt DMA for vertical data motions (to transfer data between the off-chip main memory and the on-chip SPM).

The two processors were both used to build groundbreaking supercomputers. The STI CELL processors were used to built the supercomputer Roadrunner [94], the first PetaFlops

Table 6.1 Comparisons of the CELL processor and the SW26010 processor

| **Features** | **CELL** | **SW26010** |
|---|---|---|
| Released Year | 2006 | 2015 |
| Supercomputer used | Roadrunner, first 1PF | TaihuLight, first 100PF |
| Management Core | 1PPE/dual-issue/in-order/ | 1 MPE/dual-issue/in-order |
| Compute Core | 8 SPE/7-stage/SP>DP | 64 CPE/7-stage/SP=DP |
| Off-chip Memory | –/DDR2 | 8GB/DDR3 |
| Vertical Data Motions | DMA | DMA |
| Off-chip Memory BW. | 25.6GB/s | 31GB/s |
| On-chip Memory | 256KB/SPM/local store | 64KB/SPM/LDM |
| On-chip Memory BW. | *N/A* | 46.4 GB/s per CPE |
| Horizontal Data Motions | *DMA* | *Register Communication* |

system in the world. Roadrunner achieved a sustainable performance of 1.38 PetaFlops/s in DP and 2.91 PetaFlops/s in SP with 6480 nodes. Each node includes three blades: one blade contains two dual-core AMD Opteron processors; the other two contain two PowerXCell 8i processors each. Thus Roadrunner contains an equal number of general-purpose microprocessors and special-purpose accelerators. On the other hand, the SW26010 processors were used to build TaihuLight, the first 100 PetaFlops (in peak performance) supercomputer in the world. The system overview of TaihuLight is presented in Secetion 5.2.1.

The two supercomputers, however, faced totally different situation in building a software ecosystem. Roadrunner failed to build a mature software ecosystem due to lack of real-world applications; TaihuLight has earned the world-wide reputations by scaling up a mount of real-world applications, including two Gordon Bell Prize winner [64] [83] and three Gordon Bell Prize finalist [84] [95] [96]. As shown in Table 6.2, the real-world applications developed on TaihuLight has covered a broad range of domains.

Table 6.2 Real-world applications developed on TaihuLight

| **Applications** | **Domains** | **Programming approach** |
|---|---|---|
| Fully-Implicit Solver | Weather Forecast | Native |
| Surface | Wave Ocean | Native |
| Phase Field Simulations | Material Science | Native |
| CAM | Weather Forecast | SWACC |
| CAM-SE | Weather Forecast | Native |
| NES | Earthquake | Native |
| GTC-P | Laser Plasma | SWACC |
| swDNN | Deep Learning | Native |

### 6.2.2 The Reasons Caused the CELL Failing

In 2009, IBM ended the production line of the CELL processor. Although there are few research papers systematically explained the reasons why the CELL processor failed, we found some informative discussions online, such as [97] [98] [99]. We distilled these discussions into the following three key points, which are from both the business and the technical perspectives.

- **Commercially unsuccessful in the video game market**. The CELL-based Sony PS3 failed to compete with Microsoft XBOX 360. This failure resulted in the high cost of the processor caused by the low manufacturing yield.
- **The strong competitor in the HPC market**. In the HPC market since 2007, the CELL processor had met with the increasing competition from NVIDIA GPUs. NVIDIA GPUs had larger production volumes and thus could provide lower prices. In 2008, just two years after Roadrunner was built, Tokyo Institute of Technology built the world's first GPU-based supercomputer TSUBAME 1.2 [100] with 170 NVIDIA Tesla S1070. Since then, increasing number of GPU-based supercomputers ranked in the TOP500 list [101].
- **Programming difficulties**. One key reason that lacking of real-world applications on Roadrunner is the CELL's programming difficulty. The programming difficulty can be classified into two aspects: 1) lacking of high-level programming approaches for common developers to achieve reasonable performance, e.g., 50-60% of the peak performance; 2) the architectural design flaw prevents the ninja programers [102] from achieving the ultimate performance; the flaw is that DMA was unfit for the fine-grained parallelism due to its long latency [45].

### 6.2.3 The Reasons caused the SW26010 Rising

Ten year later after the CELL was designed, the SW26010 designers dealt with the three key issues caused the failure of the CELL processor with the following four solutions.

- The SW26010 is not a commodity product thus has no price. The processor was designed by National High Performance Integrated Circuit Design Center of China as a research project.
- As a home-grown processor, the SW26010 processor gains strong support from the Chinese government. In 2015, the US government blocked Intel from selling its

Table 6.3 RLC vs. DMA for the all_reduce operation in terms of latencies on 64 CPEs of the SW26010 processor

| **Methods** | **Reduction** *(cy)* | **Broadcast** *(cy)* |
|---|---|---|
| RLC | 102 | 20 |
| DMA | 6670 | 565 |

products to upgrade the Tianhe-2 supercomputer, which ranked the TOP on the TOP500 list for 6 times. Thereafter, the Chinese government decided to build the 'flagship' supercomputers only with its own home-grown processors. TaihuLight was built with the SW26010 processor in 2015. In 2017, another flagship supercomputer Tianhe-2A [69] was upgraded by replacing the Intel KNC with the home-grown Matrix-2000 accelerators. The same situation happens in other countries as well, e.g., Japan plans to develop a home-grown ARM processor for its next generation flagship supercomputer 'Post-K'.

- To reduce the programming difficulty for the common programmers, a directive-based programming approach called Sunway OpenACC is provided. The customized OpenACC is based on the OpenACC 2.0 standard [103] and extended with some specific features for the SW26010 processor, such as a fine control over buffering of multi-dimensional array, and packing of distributed variables for data transfer [82] [32].

- The SW26010 processor adopted the lightweight register communications instead of the costly DMA for the intra-chip communication. Based on the lessons learnt from the CELL processor, DMA has been approved too slow for the fine-grained inter-core communication due to its high initial cost and transmission cost. Usually, the ILP at the pipeline level takes 1-cycle; the coarse-grained concurrency at the processor level takes 10,000-cycle. Therefore, we can infer the fine-grained parallelism at inter-core level require 10-100 cycles. RLC can fit well into this range as indicated by the results in Table 6.3. To conduct a broadcast and a reduction operation on 64 CPEs, it only requires 20 and 102 cycles, respectively.

## 6.3 Implications For the Next Generation SW Processor

China plans to build two Exascale supercomputers in 2021-2022. One of the Exascale machine might use the next generation of Sunway Processor. This section discusses the implications for the next generation Sunway processor based on our performance optimizations

experience gained on current Sunway processor. As the current Sunway processor SW26010 is the 4th generation, to be concise, we denote the next generation as **SW5**. It is noteworthy that our discussion on the SW5 designs is solely based on our own assumptions; we receive no information or indications from the SW5 designers.

### 6.3.1 Design Goal

The electricity cost limits the nodes number of Exascale systems. In China, there is no such strict power consumption cap of 20 Megawatt for an exascale system like in the US [104]. As TaihuLight takes 40,960 nodes for 15 Megawatt, we assume the Exascale supercomputer will have the similar node number in the less restrict power cap. As the Exascale system requires at lest 10X compute flops than TaihuLight, we infer that **the next generation requires at lest 10X faster in compute flops than the SW26010**.

### 6.3.2 Design Details

Due to the great success of the SW26010 processor used in TaihuLight, the SW5 processor is highly likely to adopt the similar architectural design. The 10X performance increase of the SW5 can be achieved by both adding more cores onto the processor and enlarging the vector length. As shown in Fig. 6.1, two design options can be adopted: the option 1 is to increase the CPE number on a core-group to $16 \times 16$ while keep the same core-group number on a single processor; the option 2 is to increase the CPE number on a core-group to $10 \times 10$ and also the core-group number on a single processor to 16. Both two options enlarge the vector length from 256 bit to 512 bit. We discuss the design details of the two options in three perspectives: computer cores, memory hierarchy, and the register communication.

#### Compute Cores

**Clock Rate:** Technology projections [6] indicate that clock-speeds will not change appreciably and remain near 1-2 GHz by 2020. As a result, we can only expect a slight increase in the clock rate of the SW5, compared with the 1.45GHz of the SW26010.

**Instruction Issue Order:** Due to the high power consumption and the design complexity of out-of-order execution [105], the SW5 may still adopt the in-order execution.

**SIMD:** SIMD is the most popular approach to increase peak performance at the chip level. The SIMD units number on x86 chips has doubled in recent years. For example, Intel CPUs had a SIMD width of 4 slots in Haswell and Broadwell processors, and increase to 8 slots in the Skylake and Kabylake processors. Similarly, the SW26010 processor uses 4 slots

| | | SW26010 | SW5 design option-1 | SW5 design option-2 |
|---|---|---|---|---|
| Release Year | | 2015 | 2021-2022 | |
| NO. of core-groups/processor | | 4 | 4 | 16 |
| NO. of CPEs/core group | | 8x8 | 16x16 | 10x10 |
| Compute cores | Clock Rate | 1.45GHz | 1.5-1.6GHz | |
| | Vectorization | 256 bit | 512 bit | |
| | DP vs.. SP | 1:1 | 1:2 | |
| | EMU | N/A | Maybe | |
| | Register number | 32 | 64-96 | |
| | Flops | 3.06TFlops | 24TFlops | 24TFlops |
| Memory | Off-chip Memory | 8GB DDR3 | DDR4 or DDR5 | |
| | SPM size | 64KB | 128-256KB | |
| Register com. | Comm. pattern | Row/column-based | Routing | |
| | Buffers | 6/4/4 | 20/10/10 | 12/8/8 |

Fig. 6.1 Two design options of the next generation Sunway processor

(i.e. 256 bit) and we expect the SW5 will increase the slots to 8 slots (i.e. 512 bit) to double the peak performance.

**FPU:** The performance of SP and DP are the same on the SW26010. However, as the deep learning applications and frameworks become one of the major workload on supercomputers, we expect that the performance ratio between SP and DP on the SW5 might be doubled.

**Extended Math Unit (EMU):** The transcendental operations require much longer latencies on the SW26010 than NIDIA GPUs and the Intel KNL. For example, due to the lack of the efficient hardware support, the `rsqrt` operation takes *68 cycles* for the absolute latency and *58 cycles* of the pipelined latency on the SW26010. In contract, with EMU, the same operation only requires *6 cycles* on NVIDIA GPUs and the Intel KNL. These transcendental operations become the major performance bottleneck for the applications such as N-body kernels (Section 4.1). Although adding an EMU on the SW5 is highly recommended from the programmers' perspective, due to its design complexity and large space required, the EMU is less likely to be implemented on the SW5. Instead, the more cost-effective approach is to add more registers for the loop unrolling optimization.

**Register Numbers:** Due to the in-order execution of the CPE pipeline, the loop unrolling is the most efficient optimization to explore the ILP. However, the limited register number of each CPE prevents the optimization from improving the performance because of the register spilling [106] (i.e. moving a variable from a register to local memories). Each SW26010

CPE has only 32 vector registers. From programmers' perspective, we suggest to double or even triple the registers number on the SW5.

### Memory Hierarchy

**Off-chip Memory**: The limited memory bandwidth of the SW26010 processor is the root cause of the programming challenges. Although in this thesis we emphasis the efficient intra-chip communication to minimize memory traffic, it is still desirable for the high memory bandwidth on the SW5. Two options can be adopted for the high bandwidth memory. The first option is the next generation of DDR DRAM. Although the DDR4 standard was finalized in 2012 and the SW26010 is released 2015, the processor still used the previous generation DDR3. This delay is partly because the memory controllers in processors need to be updated, and the update normally takes years. JEDEC, the organization in charge of defining new standards for computer memory, plans to finalize the next generation memory DDR5 standard in 2018. Considering the schedule to build the Exascale supercomputer in 2022, there are about 3-4 years to adopt the DDR5 DRAM; the situation is almost the same when building TaihuLight. Therefore, it is hard to predict whether the DDR5 will be adopted on the SW5. The second option could be HBM [107]. HBM was adopted in the NVIDIA GPU Titan V and the Intel KNL processor and Intel FPGA.

**On-chip Memory**: The on-chip cache has two disadvantages: consuming power in the range of 25% to 45% of the total chip power [108]; wasting energy and bandwidth to move unused data between cores. As a result, SW26010 adopted the software management of memory SPM as its local data memory (LDM) for CPEs. We expect that the SW5 will continue using SPM and may increase the SPM size from 64 KB to 128 KB or even larger.

### Intra-chip Communication

Two options (Fig. 6.1) can be adopted to add more cores on the SW5: one is $16 \times 16$ and the other is $10 \times 10$. Compared with the $8 \times 8$ cores on the SW26010, the more cores on the SW5 will bring more challenges in exploring on-chip data locality.

**Buffers:** On the SW26010, the unified sender buffer can only hold 6 vector elements and the two receiver buffers can only hold 4 vector elements for each direction, row or column. If these buffers are full, the RLC transmission will be blocked Due to the increasing number of cores, we suggest to increase the buffer size as well.

**Communication Patterns:** The RLC on the SW26010 can only support the row/column-based communication, which prevents the RLC from being applying to a broader range of applications, such as the applications requiring random memory access. We suggest the RLC

on the SW5 to support more flexible communication patterns in hardware, such as the routing and collective communications.

# Chapter 7

# Conclusion and Future Work

This chapter concludes the thesis by summarizing the work and the main contributions it described, and noting areas in which further work is required.

## 7.1 Thesis Statement Revisited

Due to the end of Moore's law in the recent decade, instead of improving the clock frequency, more cores are adding to a single chip, resulting in the emerging of a new breed of many-core processor architecture. As the memory performance has been lagging behind the processor performance for a couple of decades, the increasing flops-to-byte ratio of many-core processors makes most application memory-bound. One promising solution to this challenge is exploring on-chip data locality with an efficient inter-core communication. The inter-core communication has two basic schemas: the load/store and message-passing. The former such as the cache-coherence is easy-to-program but less scalable; the latter such as register communications is scalable but hard-to-program.

Despite of its strong scalability, the register communication brings the significant increasing programming challenge to many-core processors. This study aims to tackle the key programming challenges of the many-more processors with register communications. Taking the SW26010 processor as a research vehicle, we identified the programming challenges through a comprehensive evaluation of the processor, and developed a systematic optimization solution. Our work on the SW26010 includes three studies.

The purpose of the first study was to to illuminate the uncharted area of the SW26010 processor in order to provide important information for performance optimizations. First, we developed a micro-benchmark suite, mostly written in assemble language, to evaluate the key micro-architectural features of the SW26010 processor. The benchmark revealed some unanticipated findings beyond the publicly available data. For instance, the instruction

issue order in between the in-order dual-issue pipeline is out-of-order; the broadcast mode of register communications has the same latency as the peer-to-peer mode. Second, we applied the roofline model, with the key parameters that we obtained from measuring the processors using the benchmark suite, to identify the key programming challenge of the SW26010 processor. Third, based on the benchmark results, we proposed a systematic guideline for performance optimizations on the SW26010 and instantiated the guideline with two cases. The methodology we developed in this study, that infers a processor's micro-architecture designs from benchmark results, can also be applied on other processors lacking of public information.

Based on the findings reveled in the first step, we conducted the second one in which we optimized two compute-bound kernels. The first kernel is the direct N-body simulation. Due to the lack of efficient hardware support, the reciprocal square root (*rsqrt*) operations turned out to be the performance bottleneck of N-body on the SW26010. We applied the computation-oriented optimizations and achieved about 25% efficiency in a single core-group of the SW26010. The second kernel is double-precision general matrix-multiplication (DGEMM). We designed a novel algorithm for RLC and applied several on-chip communication-oriented optimizations. These endeavors improved the efficiency to up to 88.7% in a single core-group of the SW26010.

In contrast to the compute-bound kernels, due to the limited memory bandwidth of the SW26010, the single memory-bound kernel – such as Sparse matrix-vector multiplication (SpMV – cannot perform well on the processor, despite comprehensive optimizations. However, we anticipated, the overall performance of an algorithm can be effectively improved by overlapping multiple memory-bound kernels within the algorithm, and, thus, can provide a promising optimization approach for those multi-kernel memory-bound algorithms to achieve better performance on the SW26010. The aim of the third step in this study is to optimize the memory-bound algorithm Preconditioned Conjugate Gradient (PCG). First, in order to minimize the all_reduce communication cost of PCG, we developed a new algorithm RNPCG, a non-blocking PCG leveraging the on-chip register communication. Second, we optimized three key kernels of the PCG, including proposing a localized version of the Diagonal-based Incomplete Cholesky (LDIC) preconditioner. Third, to scale the RNPCG on TaihuLight, we designed the three-level non-blocking all_reduce operations. With these three steps, we implemented the RNPCG in the computational fluid dynamics software OpenFOAM. The experimental results on TaihuLight show that 1) compared with the default implementations of OpenFOAM, the RNPCG and the LDIC on a single-core group of SW26010 can achieve a maximum speedup of 8.9X and 3.1X, respectively; 2) the scalable RNPCG can outperform the standard PCG both in the strong and the weak scaling up to 66,560 cores.

## 7.2 Summary of Research Contributions

The thesis makes the following three key contributions:

- This thesis presented the first micro-benchmark suite to explore the micro-architectural features of the SW26010 processor. By using the micro-benchmark suite, we quantified the key micro-architectural features of the SW26010 and revealed some unanticipated findings beyond the publicly available data. For instance, the broadcast mode of register communications has the same latency as the peer-to-peer mode.
- This thesis presented the novel algorithms using register communication for the compute-bound kernel DGEMM. We designed RLC-friendly algorithms for DGEMM and implemented in the assembly with manual instruction scheduling to achieve about 90% efficiency on the single core-group of SW26010. In addition, we found that `rsqrt` is the bottleneck that hampers the performance of the N-body kernel, and we thus optimized the `rsqrt` with a software routine.
- This thesis presented the first PCG optimization on the SW26010 processor. We designed the RLC-friendly non-blocking PCG (RNPCG) for the SW26010. With the optimizations on the three key PCG kernels, RNPCG achieved at most 8.9X speedup than the default implementation. Furthermore, we scaled RNPCG on TaihuLight up to 66,560 cores.

## 7.3 Future Work

The evaluation and optimizations work on the SW26010 presented in this thesis provide fruitful seeds to cultivate a wider research agenda on the register communication.

#### A Library for Register Communications

A register communications library to automate message orchestration at C/C++ level. Due to the increasing arithmetic-intensities of many-core processors, sharing on-chip data among cores is essential to achieve ultimate performance on the many-core processors. One promising solution to share on-chip data is using the lightweight register communication. However, the benefits of the register communication can be undermined by its programming challenges. For instance, programming the register communication on the SW26010 is painful. Because the register messages contains no 'Sender's ID', the receiver core is unable to identify the sender core. Programmers are required to manually orchestrate the message sequence in order

to ensure the correct sending and receiving order. Additionally, this manual programming approach can be only implemented in assembly language so far due to the inefficient compiler support. Therefor, a C/C++ library is required by the register communication to reduce its programming difficulty.

### A Performance Model for RLC-friendly Algorithms

A performance model to guide the RLC-friendly algorithms designs. RLC-friendly algorithms should be designed for applications to achieve ultimate performance on the SW26010. In this thesis, we demonstrate how to design the RLC-friendly algorithms for the compute-bound DGEMM and the memory-bound PCG. However, due to the variance of applications, generalizing the design process of the RLC-friendly algorithm is challenging. Based on the benchmark results we measured in Chapter 3, we plan to build a comprehensive performance model to address this design challenges.

# References


[1] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6): 52–59, 1997.

[2] Chris A Mack. Fifty years of moore's law. *IEEE Transactions on semiconductor manufacturing*, 24(2):202–207, 2011.

[3] Shekhar Borkar. Design challenges of technology scaling. *IEEE micro*, 19(4):23–29, 1999.

[4] Nam Sung Kim, Todd Austin, David Baauw, Trevor Mudge, Krisztián Flautner, Jie S Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore's law meets static power. *computer*, 36(12):68–75, 2003.

[5] Jim Held, Jerry Bautista, and Sean Koehl. White paper from a few cores to many: A tera-scale computing research review. 2006.

[6] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The Landscape Of Parallel Computing Research: A View From Berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[7] Sriram R Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Tiju Jacob, Shailendra Jain, et al. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, 2008.

[8] Timothy G Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 38. IEEE Press, 2008.

[9] Satoshi Matsuoka, Hideharu Amano, Kengo Nakajima, Koji Inoue, Tomohiro Kudoh, Naoya Maruyama, Kenjiro Taura, Takeshi Iwashita, Takahiro Katagiri, Toshihiro Hanawa, et al. From flops to bytes: disruptive change in high-performance computing towards the post-moore era. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 274–281. ACM, 2016.

[10] Rakesh Kumar, Timothy G Mattson, Gilles Pokam, and Rob Van Der Wijngaart. The case for message passing on many-core chips. In *Multiprocessor System-on-Chip*, pages 115–123. Springer, 2011.

[11] Avadh Patel and Kanad Ghose. Energy-efficient mesi cache coherence with pro-active snoop filtering for multicore microprocessors. In *Low Power Electronics and Design (ISLPED), 2008 ACM/IEEE International Symposium on*, pages 247–252. IEEE, 2008.

[12] Taeweon Suh, Douglas M Blough, and H-HS Lee. Supporting cache coherence in heterogeneous multiprocessor systems. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 1150–1155. IEEE, 2004.

[13] Sabela Ramos and Torsten Hoefler. Modeling communication in cache-coherent smp systems: a case-study with xeon phi. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 97–108. ACM, 2013.

[14] Dac Pham, Shigehiro Asano, Mark Bolliger, Michael N Day, H Peter Hofstee, C Johns, J Kahle, Atsushi Kameyama, John Keaty, Yoshio Masubuchi, et al. The design and implementation of a first-generation CELL processor. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 184–592. IEEE, 2005.

[15] James A Kahle, Michael N Day, H Peter Hofstee, Charles R Johns, Theodore R Maeurer, and David Shippy. Introduction to the cell multiprocessor. *IBM journal of Research and Development*, 49(4.5):589–604, 2005.

[16] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF*

*'06: Proceedings of the 3rd conference on Computing frontiers*. Lawrence Berkeley National Laboratory, ACM, May 2006.

[17] Jack Dongarra. Report on the sunway taihulight system. *www. netlib. org. Retrieved June*, 20, 2016.

[18] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, et al. The Sunway Taihulight Supercomputer: System And Applications. *Science China Information Sciences*, 59 (7):072001, 2016.

[19] Stephen W Keckler, William J Dally, Daniel Maskit, Nicholas P Carter, Andrew Chang, and Whay S Lee. Exploiting fine-grain thread level parallelism on the mit multi-alu processor. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 306–317. IEEE Computer Society, 1998.

[20] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C. C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pages 88–598, Feb 2008. doi: 10.1109/ISSCC.2008.4523070.

[21] INC TILERA. Tile-gx processor family. *Tilera, Inc*, 2012.

[22] National Instruments VISA documentation. URL www.ni.com/visa.

[23] Anant Agarwal, Liewei Bao, John Brown, Bruce Edwards, Matt Mattina, Chyi-Chang Miao, Carl Ramey, and David Wentzlaff. Tile processor: Embedded multicore for networking and multimedia. In *Hot Chips*, volume 19, 2007.

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

[25] Jack Dongarra and Michael A Heroux. Toward a new metric for ranking high performance computing systems. *Sandia Report, SAND2013-4744*, 312:150, 2013.

[26] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78. ACM, 2002.

[27] Nvidia tesla v100 gpu architecture whitepaper. URL https://goo.gl/nFScwD.

[28] Intel. Intel xeon phi processor overview. URL https://goo.gl/ryiY3S.

[29] Sabela Ramos and Torsten Hoefler. Capability models for manycore memory systems: a case-study with xeon phi knl. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 297–306. IEEE, 2017.

[30] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109. IEEE, 2010.

[31] Don Anderson and Tom Shanley. *Pentium processor system architecture*. Addison-Wesley Professional, 1995.

[32] Haohuan Fu, Junfeng Liao, Wei Xue, and et al. Refactoring And Optimizing The Community Atmosphere Model (CAM) On The Sunway Taihulight Supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 83. IEEE Press, 2016.

[33] Min Tian, Weidong Gu, Jingshan Pan, and Meng Guo. Performance Analysis And Optimization Of Palabos On Petascale Sunway BlueLight MPP Supercomputer. In *International Conference on Parallel Computing in Fluid Dynamics*, pages 311–320. Springer, 2013.

[34] Li-Shiuan Peh and Natalie Enright Jerger. On-chip networks (synthesis lectures on computer architecture). *Morgan and Claypool, San Rafael*, 2009.

[35] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1):1, 2006.

[36] Jose Duato, Sudhakar Yalamanchili, and Lionel M Ni. *Interconnection networks: an engineering approach*. Morgan Kaufmann, 2003.

[37] Wang Zhang, Ligang Hou, Jinhui Wang, Shuqin Geng, and Wuchen Wu. Comparison research between xy and odd-even routing algorithm of a 2-dimension 3x3 mesh topology network-on-chip. In *Intelligent Systems, 2009. GCIS'09. WRI Global Congress on*, volume 3, pages 329–333. IEEE, 2009.

[38] Leslie G Valiant and Gordon J Brebner. Universal schemes for parallel communication. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 263–277. ACM, 1981.

[39] Tse-yun Feng. A survey of interconnection networks. *Computer*, 14(12):12–27, 1981.

[40] William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[41] Parviz Kermani and Leonard Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks (1976)*, 3(4):267–286, 1979.

[42] William J Dally and Charles L Seitz. The torus routing chip. *Distributed computing*, 1 (4):187–196, 1986.

[43] Zhiyi Yu, Ruijin Xiao, Kaidi You, Heng Quan, Peng Ou, Zheng Yu, Maofei He, Jiajie Zhang, Yan Ying, Haofan Yang, et al. A 16-core processor with shared-memory and message-passing communications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(4):1081–1094, 2014.

[44] Zhigeng D Xu, James Lin, and Satoshi Matsuoka. Benchmarking sw26010 many-core processor. In *Parallel and Distributed Processing Symposium Workshops, 2017 IEEE International [in press]*. IEEE, 2017.

[45] Yong Dou, Lin Deng, Jinhui Xu, and Yi Zheng. Dma performance analysis and multi-core memory optimization for swim benchmark on the cell processor. In *Parallel and Distributed Processing with Applications, 2008. ISPA'08. International Symposium on*, pages 170–179. IEEE, 2008.

[46] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In

*Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246. IEEE, 2010.

[47] Jianbin Fang, Henk Sips, Lilun Zhang, Chuanfu Xu, Yonggang Che, and Ana Lucia Varbanescu. Test-driving intel xeon phi. In *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*, pages 137–148. ACM, 2014.

[48] L Nyland, M Harris, and J Prins. Fast N-Body Simulation with CUDA. *GPU Gems3*, 2007.

[49] Rio Yokota and Mustafa Abduljabbar. *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, volume One, chapter N-body Methods. Morgan Kaufmann, 2014.

[50] N. Arora, A. Shringarpure, and R. W. Vuduc. Direct n-body kernels for multicore platforms. In *2009 International Conference on Parallel Processing*, pages 379–387, Sept 2009. doi: 10.1109/ICPP.2009.71.

[51] Alejandro Duran and Larry Meadows. *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, volume One, chapter A Many-core Implementaton of the Direct N-body Problem. Morgan Kaufmann, 2014.

[52] *Intel Xeon Phi Processor High Performance Programming (Knights Landing Edition)*, chapter 23. Morgan Kaufmann, 2016.

[53] A Karp. Speeding up N-body Calculations on Machines without Hardware Square Root. pages 1–20, May 2001.

[54] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. Fast implementation of dgemm on fermi gpu. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 35. ACM, 2011.

[55] Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, and et al. Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems Based on Intel® Xeon Phi™ Coprocessor. In *IPDPS '13: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 126–137. IEEE Computer Society, May 2013.

[56] Mark Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, University of California, Berkeley, April 2010.

[57] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[58] Pieter Ghysels and Wim Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Computing*, 40(7):224–238, 2014.

[59] Paul R Eller and William Gropp. Scalable non-blocking preconditioned conjugate gradient methods. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 18. IEEE Press, 2016.

[60] David Hysom and Alex Pothen. Efficient parallel computation of ilu (k) preconditioners. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, page 29. ACM, 1999.

[61] Edmond Chow and Aftab Patel. Fine-grained parallel incomplete lu factorization. *SIAM journal on Scientific Computing*, 37(2):C169–C193, 2015.

[62] Kengo Nakajima. *Parallel Iterative Linear Solvers with Preconditioning for Large Scale Problems*. PhD thesis, The University of Tokyo, 2002.

[63] Kai Wang, Sang-Bae Kim, Jun Zhang, Kengo Nakajima, and Hiroshi Okuda. Global and localized parallel preconditioning techniques for large scale solid earth simulations. *Future Generation Computer Systems*, 19(4):443–456, 2003.

[64] Chao Yang, Wei Xue, Haohuan Fu, and et al. 10M-Core Scalable Fully-Implicit Solver for Nonhydrostatic Atmospheric Dynamics. In *The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016.

[65] NSCCWX. Sunway taihulight compiler user guide, 2016. URL http://www.nsccwx.cn/.

[66] Guido Juckeland, Stefan Borner, Michael Kluge, Sebastian Kolling, Wolfgang E Nagel, Stefan Pflüger, Heike Roding, Stephan Seidl, Thomas William, and Robert Wloch. Benchit-performance measurements and comparison for scientific applications. In *PARCO*, pages 501–508, 2003.

[67] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. URL http://www.cs.virginia.edu/stream/. A continually updated technical report.

[68] David E Culler, Richard M Karp, David Patterson, Abhijit Sahay, Eunice E Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken. LogP: A Practical Model Of Parallel Computation. *Communications of the ACM*, 39(11):78–85, 1996.

[69] Jack Dongarra. Report on the Tianhe-2A System. Technical report, University of Tennessee, Knoxville, 2017.

[70] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

[71] Leslie Greengard. *The Rapid Evaluation of Potential Fields In Particle Systems*. MIT press, 1988.

[72] Roger W Hockney and James W Eastwood. *Computer Simulation Using Particles*. CRC Press, 1988.

[73] T Hoefler, W Gropp, W Kramer, and M Snir. Performance Modeling For Systematic Performance Tuning. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC11)*, pages 1–12, 2011.

[74] Chris Lomont. Fast Inverse Square Root. *Tech-315 nical Report*, page 32, 2003.

[75] Kazushige Goto and Robert A Geijn. Anatomy Of High-Performance Matrix Multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12, 2008.

[76] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-scale GPU-accelerated Supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063398.

[77] Jonathan Richard Shewchuk. An Introduction To The Conjugate Gradient Method Without The Agonizing Pain, 1994.

[78] Delong Meng, Minhua Wen, Jianwen Wei, and James Lin. Hybrid Implementation and Optimization of OpenFOAM on the SW26010 Many-core Processor. *HPC China*, 2016.

[79] Claude Pommerell. *Solution Of Large Unsymmetric Systems Of Linear Equations*. PhD thesis, 1992.

[80] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. *HiPEAC*, 5952: 111–125, 2010.

[81] John D McCalpin. Stream benchmark. *URL: http://www. cs. virginia. edu/stream/stream2*, 2002.

[82] Haohuan Fu, Junfeng Liao, Jinzhe Yang, and et al. The Sunway TaihuLight supercomputer: system and applications. *Science China Information Sciences*, 59(7): 072001–16, June 2016.

[83] Haohuan Fu, He Conghui, Nan Ding, Xiaohui Duan, Lin Gan, Yishuang Liang, Xinliang Wang, Jinzhe Yang, Yan Zheng, Weiguo Liu, et al. 15-Pflops Nonlinear Earthquake Simulation on Sunway TaihuLight: Enabling Depiction of Realistic 10 Hz Scenarios. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 1. ACM, 2017.

[84] Haohuan Fu, Conghui He, Bingwei Chen, et al. Redesigning Cam-Se For Peta-Scale Climate Modeling Performance And Ultra-High Resolution On Sunway Taihulight. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 10. ACM, 2017.

[85] Henry G Weller, G Tabor, Hrvoje Jasak, and C Fureby. A Tensorial Approach To Computational Continuum Mechanics Using Object-Oriented Techniques. *Computers in physics*, 12(6):620–631, 1998.

[86] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. QR Factorization On A Multicore Node Enhanced With Multiple Gpu Accelerators. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 932–943. IEEE, 2011.

[87] Hatem Ltaief, Stanimire Tomov, Rajib Nath, Peng Du, and Jack Dongarra. A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators LAPACK. *Working note 223*.

[88] Luc Buatois, Guillaume Caumon, and Bruno Levy. Concurrent Number Cruncher: A GPU Implementation of A General Sparse Linear Solver. *International Journal of Parallel, Emergent and Distributed Systems*, 24(3):205–223, 2009.

[89] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 917–924. ACM, 2003.

[90] Openfoam for intel knights landing. URL https://goo.gl/PZ5P4h.

[91] OpenFOAM Standard Solvers. URL https://goo.gl/af6MHd.

[92] OpenFOAM Tutorial. URL https://goo.gl/KFjU9q.

[93] Heng Lin, Xiongchao Tang, Bowen Yu, Youwei Zhuo, Wenguang Chen, Jidong Zhai, Wanwang Yin, and Weimin Zheng. Scalable Graph Traversal on Sunway TaihuLight with Ten Million Cores. In *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE, 2017.

[94] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the Petaflop Era: The Architecture and Performance of Roadrunner. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 1:1–1:11, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9. URL http://dl.acm.org/citation.cfm?id=1413370.1413372.

[95] Jian Zhang, Chunbo Zhou, Yangang Wang, and et al. Extreme-Scale Phase Field Simulations of Coarsening Dynamics on the Sunway TaihuLight Supercomputer. In *The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016.

[96] Fangli Qiao, Wei Zhao, Xunqiang Yin, and et al. A Highly Effective Global Surface Wave Numerical Simulation with Ultra-High Resolution. In *The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016.

[97] End of the line for IBM's Cell, 2009. URL https://goo.gl/NsJjAF.

[98] Mark cerny explains why the cell processor was not included in the ps4, 2013. URL https://goo.gl/3QvomC.

[99] Is the Cell processor an absolute failure for Sony?, 2010. URL https://goo.gl/WTuPv3.

[100] Satoshi Matsuoka, Takayuki Aoki, Toshio Endo, Akira Nukada, Toshihiro Kato, and Atushi Hasegawa. Gpu accelerated computing–from hype to mainstream, the rebirth of vector computing. In *Journal of Physics: Conference Series*, volume 180, page 012043. IOP Publishing, 2009.

[101] TOP500 Supercomputers List. URL https://www.top500.org/.

[102] Nadathur Satish, Changkyu Kim, Jatin Chhugani, and et al. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *ACM SIGARCH Computer Architecture News*, volume 40, pages 440–451. IEEE Computer Society, 2012.

[103] OpenACC Standard Committee et al. The openacc application programming interface, version 2.0. *Standard document, OpenACC-standard. org*, 2013.

[104] DOE E3 Report. URL https://goo.gl/bBkb1o.

[105] Michael K Gowan, Larry L Biro, and Daniel B Jackson. Power considerations in the design of the alpha 21264 microprocessor. In *Proceedings of the 35th annual Design Automation Conference*, pages 726–731. ACM, 1998.

[106] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. Modulo scheduling with integrated register spilling for clustered vliw architectures. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 160–169. IEEE Computer Society, 2001.

[107] JEDEC Standard. High bandwidth memory (hbm) dram. *JESD235*, 2013.

[108] Preeti Ranjan Panda, Nikil D Dutt, and Alexandru Nicolau. *Memory issues in embedded systems-on-chip: optimizations and exploration*. Springer Science & Business Media, 1999.

# Appendix A

# Peer-reviewed Publications

## A.1 International Conferences or Journal Articles

1. **James Lin**, Minhua Wen, Delong Meng, Akira Nukada, and Satoshi Matsuoka. Optimizations of Preconditioned Conjugate Gradient on TaihuLight for OpenFOAM, *2018 18th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, ACM/IEEE, 2018.

2. **James Lin**, Zhigeng Xu, Akira Nukada, Naoya Maruyama, and Satoshi Matsuoka. Optimizations of Two Compute-Bound Scientific Kernels on the SW26010 Many-Core Processor. *2017 46th International Conference on Parallel Processing (ICPP)*, pp. 432-441. IEEE, 2017.

3. **James Lin**, Zhigeng Xu, Linjin Cai, Akira Nukada, and Satoshi Matsuoka. Evaluating the SW26010 Many-core Processor with a Micro-benchmark Suite for Performance Optimizations, *Journal of Parallel Computing*. (*submitted*)

4. Shuo Li, **James Lin**, Accelerating Asian Option Pricing On Many-Core Architectures. *Concurrency and Computation: Practice and Experience*, 2016, 28(3): 848-865

5. Yueming Wei, Yichao Wang, Linjin Cai, William Tang, Bei Wang, Stephane Ethier, Simon See and **James Lin**. Performance and Portability Studies with OpenACC Accelerated Version of GTC-P. *2016 17th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, Guangzhou, China, 2016.

## A.2 International Workshops or Domestic Conferences

1. Minghua Wen, Min Chen, and **James Lin**, Optimizing a Particle-in-Cell Code on Intel Knights Landing, In *IXPUG Workshop Asia 2018*, ACM, Tokyo, Japan, 2018

2. Zhigeng Xu, **James Lin**, and Satoshi Matsuoka. Benchmarking SW26010 Many-Core Processor. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 743-752. IEEE, 2017.

3. Yichao Wang, **James Lin**, Linjin Cai, William Tang, Stephane Ethier, Bei Wang, Simon See and Satoshi Matsuoka, Porting and Optimizing GTC-P on TaihuLight Supercomputer with Sunway OpenACC. *HPC China 2016*, Xi'an, China, 2016 (**Best Paper**)

4. Haipeng Wu, Minhua Wen, Simon See and **James Lin**, Parallelization and Optimization of Laser-Plasma-Interaction Simulation Based on Kepler Cluster. *HPC China 2016*, Xi'an, China, 2016.

5. Yumeng Si, Jianwen Wei, Simon See and **James Lin**, Optimizing a Galaxy Group Finding Algorithm on SMP vs. Distributed Memory Cluster. *HPC China 2016*, Xi'an, China, 2016.

6. **James Lin**, Qiang Qin, Shuo Li, Minhua Wen and Satoshi Matsuoka, Evaluating Intel AVX2 Vgather Instructions with Stencils, *HPC China 2015*, Wuxi, China, 2015

7. **James Lin**, Akira Nukada, Satoshi Matsuoka, Modeling Gather and Scatter with Hardware Performance Counters for Xeon Phi, *ACM/IEEE CCGrid15 Doctoral Symposium*, Shenzhen, China, 2015

8. Suttinee Sawadsitang, **James Lin**, Simon See, Francois Bodin, Satoshi Matsuoka, Understanding Performance Portability of OpenACC for Supercomputers, *PLC15*, IPDPS workshop, Hyderabad, India, 2015

9. Jianwen Wei, Zhigeng Xu, Bingqiang Wang, Simon See and **James Lin**, Accelerating Gene Clustering on Heterogeneous Clusters. *HPC China 2015*, Wuxi, China, 2015.

10. He Hao, Yumeng Si, Jianwen Wei, Minhua Wen and **James Lin**, Optimize Irregular Memory Access in Astronomic Clustering Application. *HPC China 2015*, Wuxi, China, 2015.

11. **James Lin**, Shuo Li, Jiaming Zhao, Satoshi Matsuoka, Node-level Memory Access Optimization on Intel Knights Corner, *HPC China 2014*, Guangzhou, China, 2014

12. Guanghao Jin, **James Lin**, Toshio Endo, Efficient Utilization Of Memory Hierarchy To Enable The Computation On Bigger Domains For Stencil Computation In CPU-GPU Based Systems, *2014 International Conference on High Performance Computing and Applications (ICHPCA), Bhubaneswar*, India, 2014 (**Best Workshop Paper**)

13. Yichao Wang, Qiang Qin, Simon See and **James Lin**, Performance Portability Evaluation for OpenACC on Intel Knights Corner and NVIDIA Kepler. *HPC China 2013*, Guilin, China, 2013.