Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Evaluation of the Configurable Architecture REPLICA with Emulated Shared Memory

by

Erik Alnervik

LIU-IDA/LITH-EX-A--14/010--SE

2014-02-06

Linköpings universitet

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Final Thesis

Evaluation of the Configurable Architecture REPLICA with Emulated Shared Memory

by

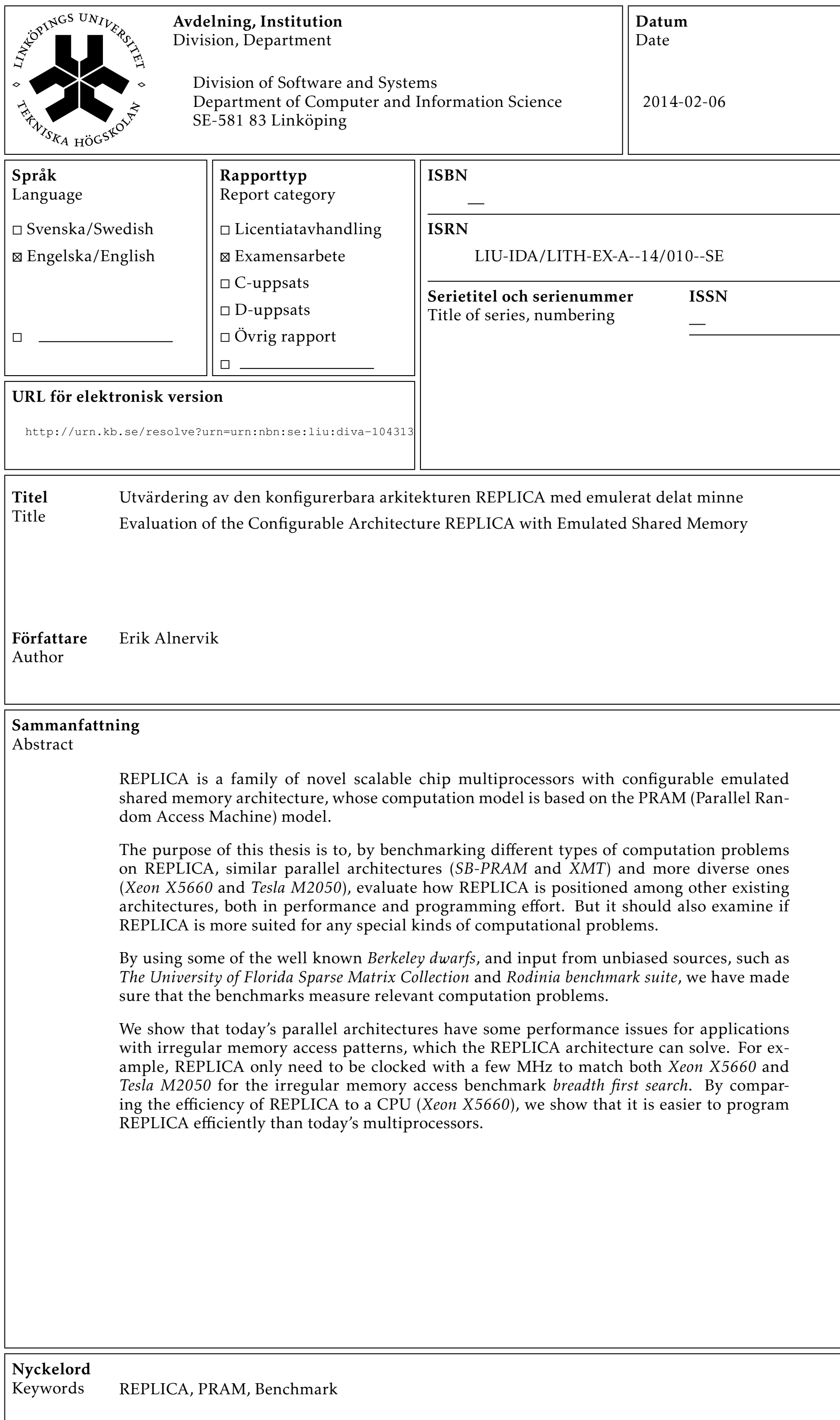
Erik Alnervik

LIU-IDA/LITH-EX-A--14/010--SE

2014-02-06

Supervisor: Erik Hansson

Examiner: Christoph Kessler



Avdelning, Institution
Division, Department

Division of Software and Systems
Department of Computer and Information Science
SE-581 83 Linköping

Datum
Date

2014-02-06

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ ________

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ ________

ISBN
—

ISRN
LIU-IDA/LITH-EX-A--14/010--SE

Serietitel och serienummer
Title of series, numbering

ISSN
—

URL för elektronisk version

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-104313

Titel
Title

Utvärdering av den konfigurerbara arkitekturen REPLICA med emulerat delat minne
Evaluation of the Configurable Architecture REPLICA with Emulated Shared Memory

Författare
Author

Erik Alnervik

Sammanfattning
Abstract

REPLICA is a family of novel scalable chip multiprocessors with configurable emulated shared memory architecture, whose computation model is based on the PRAM (Parallel Random Access Machine) model.

The purpose of this thesis is to, by benchmarking different types of computation problems on REPLICA, similar parallel architectures (*SB-PRAM* and *XMT*) and more diverse ones (*Xeon X5660* and *Tesla M2050*), evaluate how REPLICA is positioned among other existing architectures, both in performance and programming effort. But it should also examine if REPLICA is more suited for any special kinds of computational problems.

By using some of the well known *Berkeley dwarfs*, and input from unbiased sources, such as *The University of Florida Sparse Matrix Collection* and *Rodinia benchmark suite*, we have made sure that the benchmarks measure relevant computation problems.

We show that today's parallel architectures have some performance issues for applications with irregular memory access patterns, which the REPLICA architecture can solve. For example, REPLICA only need to be clocked with a few MHz to match both *Xeon X5660* and *Tesla M2050* for the irregular memory access benchmark *breadth first search*. By comparing the efficiency of REPLICA to a CPU (*Xeon X5660*), we show that it is easier to program REPLICA efficiently than today's multiprocessors.

Nyckelord
Keywords

REPLICA, PRAM, Benchmark

Sammanfattning

REPLICA är en grupp av konfigurerbara multiprocessorer som med hjälp utav ett emulerat delat minne realiserar PRAM modellen.

Syftet med denna avhandling är att genom benchmarking av olika beräkningsproblem på REPLICA, liknande (*SB-PRAM* och *XMT*) och mindre lika (*Xeon X5660* och *Tesla M2050*) parallella arkitekturer, utvärdera hur REPLICA står sig mot andra befintliga arkitekturer. Både prestandamässigt och hur enkel arkitekturen är att programmera effektiv, men även försöka ta reda på om REPLICA är speciellt lämpad för några särskilda typer av beräkningsproblem.

Genom att använda välkända *Berkeley dwarfs* applikationer och opartisk indata från bland annat *The University of Florida Sparse Matrix Collection* och *Rodinia benchmark suite*, säkerställer vi att det är relevanta beräkningsproblem som utförs och mäts.

Vi visar att dagens parallella arkitekturer har problem med prestandan för applikationer med oregelbundna minnesaccessmönster, vilken REPLICA arkitekturen kan vara en lösning på. Till exempel, så behöver REPLICA endast vara klockad med några få MHz för att matcha både *Xeon X5660* och *Tesla M2050* för algoritmen *breadth first search*, vilken lider av just oregelbunden minnesåtkomst. Genom att jämföra effektiviteten för REPLICA gentemot en CPU (*Xeon X5660*), visar vi att det är lättare att programmera REPLICA effektivt än dagens multiprocessorer.

Abstract

REPLICA is a family of novel scalable chip multiprocessors with configurable emulated shared memory architecture, whose computation model is based on the PRAM (Parallel Random Access Machine) model.

The purpose of this thesis is to, by benchmarking different types of computation problems on REPLICA, similar parallel architectures (*SB-PRAM* and *XMT*) and more diverse ones (*Xeon X5660* and *Tesla M2050*), evaluate how REPLICA is positioned among other existing architectures, both in performance and programming effort. But it should also examine if REPLICA is more suited for any special kinds of computational problems.

By using some of the well known *Berkeley dwarfs*, and input from unbiased sources, such as *The University of Florida Sparse Matrix Collection* and *Rodinia benchmark suite*, we have made sure that the benchmarks measure relevant computation problems.

We show that today's parallel architectures have some performance issues for applications with irregular memory access patterns, which the REPLICA architecture can solve. For example, REPLICA only need to be clocked with a few MHz to match both *Xeon X5660* and *Tesla M2050* for the irregular memory access benchmark *breadth first search*. By comparing the efficiency of REPLICA to a CPU (*Xeon X5660*), we show that it is easier to program REPLICA efficiently than today's multiprocessors.

Acknowledgments

I would like to thank Erik Hansson for his work as supervisor and his comments on this thesis, Martti Forsell for his technical support of the REPLICA architecture, and Christoph Kessler for his work as examiner and his comments on this thesis.

Linköping, February 2014
Erik Alnervik

Contents

Notation **xiii**

1 Introduction **1**
1.1 Purpose . . . 2
1.2 Thesis Outline . . . 2
1.3 The PRAM Model . . . 2
1.4 Performance Metrics . . . 4
1.4.1 Amdahl's Law . . . 5
1.5 The Dwarfs from Berkeley . . . 6
1.6 Related Work . . . 6
1.7 Some Available Parallel Benchmarks . . . 7
1.8 Publications . . . 8

2 The Architectures **9**
2.1 REPLICA . . . 9
2.1.1 Emulated Shared Memory . . . 10
2.1.2 Memory Modules . . . 10
2.1.3 The MBTAC Processor . . . 11
2.1.4 The Baseline Language . . . 14
2.1.5 The REPLICA Language . . . 16
2.1.6 IPSMSimX86 . . . 16
2.1.7 Limitations . . . 17
2.1.8 Previous REPLICA Works . . . 17
2.2 Xeon X5660 . . . 18
2.2.1 Xeon Machine Setup . . . 19
2.2.2 OpenMP . . . 19
2.3 XMT . . . 23
2.3.1 XMTC . . . 25
2.3.2 XMTSim . . . 27
2.4 Tesla M2050 . . . 27
2.4.1 CUDA C . . . 28
2.4.2 Tesla's Host Setup . . . 30
2.5 SB-PRAM . . . 30

2.5.1 Fork 31
2.5.2 pramsim 34

3 The Benchmark Suite 35
3.1 Measuring 35
3.2 Prefix Sum (PS) 37
3.2.1 PS for REPLICA 38
3.2.2 PS for Xeon 39
3.2.3 PS for XMT 40
3.2.4 PS for Tesla 40
3.2.5 PS for SB-PRAM 41
3.3 Dense Matrix-Matrix Multiplication (DeMM) 41
3.3.1 DeMM for REPLICA 42
3.3.2 DeMM for Xeon 44
3.3.3 DeMM for XMT 47
3.3.4 DeMM for Tesla 48
3.3.5 DeMM for SB-PRAM 50
3.4 Sparse Matrix-Vector Multiplication (SpMV) 51
3.4.1 SpMV for REPLICA 53
3.4.2 SpMV for Xeon 54
3.4.3 SpMV for XMT 55
3.4.4 SpMV for Tesla 55
3.4.5 SpMV for SB-PRAM 56
3.5 Breadth First Search (BFS) 57
3.5.1 BFS for REPLICA 59
3.5.2 BFS for Xeon 60
3.5.3 BFS for XMT 62
3.5.4 BFS for Tesla 62
3.5.5 BFS for SB-PRAM 63
3.6 Quicksort (QS) 63
3.6.1 QS for REPLICA 65
3.6.2 QS for Xeon 66
3.6.3 QS for XMT 67
3.6.4 QS for Tesla 68
3.6.5 QS for SB-PRAM 69
3.7 Summarizing the Benchmarks 69

4 Evaluation and Results 71
4.1 Efficiency for REPLICA and Xeon 71
4.2 Instruction Level Parallelism Speedup on REPLICA 74
4.3 Frequency Evaluation 75
4.3.1 Needed Frequency for PS 76
4.3.2 Needed Frequency for DeMM 77
4.3.3 Needed Frequency for SpMV 78
4.3.4 Needed Frequency for BFS 80
4.3.5 Needed Frequency for QS 81

4.4 Clock Cycles Evaluation . . . 82
4.4.1 Clock Cycles Evaluation for PS . . . 83
4.4.2 Clock Cycles Evaluation for DeMM . . . 83
4.4.3 Clock Cycles Evaluation for SpMV . . . 84
4.4.4 Clock Cycles Evaluation for BFS . . . 85
4.4.5 Clock Cycles Evaluation for QS . . . 86

5 Conclusions and Future Work **87**
5.1 Conclusions . . . 87
5.2 Future Work . . . 89

A Code **93**

B Results **123**

Bibliography **131**

Notation

Notations

Notation	Meaning
E_p	Efficiency for p processors
P	Number of processors
S_p	Speedup for p processors
T_{ID}	Thread ID
T_p	Execution time for p processors
$T_{processor}$	Number of threads per processor
T_{total}	Total number of threads

Abbreviations

Abbreviation	Meaning
ASIC	Application Specific Integrated Circuit
API	Application Program Interface
BFS	Breadth First Search
BLAS	Basic Linear Algebra Subprograms
cc	Clock Cycles
CMP	Chip Multiprocessor
CPU	Central Processing Unit
CRS	Compressed Row Storage
CSR	Compressed Sparse Row

CONTINUED ON NEXT PAGE

Abbreviations (continued from previous page)

Abbreviation	Meaning
CUDA	Compute Unified Device Architecture
DeMM	Dense Matrix-Matrix multiplication
ECLIPSE	Embedded Chip-Level Integrated Parallel Supercomputer
EMS	Emulated Shared Memory
FPGA	Field-Programmable Gate Array
GPGPU	General Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
LOC	Lines Of Code
MBTAC	Multibunched/threaded Architecture with Chaining
MCP	Many-core Processor
MIMD	Multiple Instruction Multiple Data
MTCU	Master Thread Control Unit
OOP	Object-Oriented Programming
PRAM	Parallel Random Access Machine
PS	Prefix Sum
QS	Quicksort
RAM	Random Access Machine
REPLICA	Removing Performance and programmability Limitations of Chip Multiprocessor Architectures
RTL	Runtime Library
SB-PRAM	Saarbrücken - Parallel Random Access Machine
SM	Streaming Multiprocessor
SMT	Simultaneous Multi-Threading
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread

continued on next page

Abbreviations (continued from previous page)

Abbreviation	Meaning
STL	Standard Template Library
SpMV	Sparse Matrix-Vector multiplication
TCU	Thread Control Unit
TLP	Thread-Level Parallelism
TBB	Threading Building Blocks
VILP	Virtual Instruction Level Parallelism
VLIW	Very Long Instruction Word
VTT	Valtion Teknillinen Tutkimuskeskus, in English: State Technical Research Center
XMT	Explicit Multi-Threading

1 Introduction

Previously, old software applications could gain performance by processor manufacturers continuously increasing clock frequency. Instead of using valuable time optimizing applications, developers could just wait for the next CPU (Central Processing Unit) release. This phenomenon is also referred to as *the free lunch* [71]. A few years ago the processor manufacturers found it much harder to keep up an increasing clock frequency, primarily because of the so-called *power wall*[1], and they were forced into today's multi-core era [68]. Since the clock frequency stopped increasing, most old applications also stopped gaining performance by the new *chip multiprocessors* (CMP), because they were not programmed to scale with the number of processors on the chips. Herb Sutter wrote in 2005 that concurrency (parallel programming) is the biggest revolution in software development since *Object-Oriented Programming* (OOP) [71]. He also stated that the free lunch is over. Applications now have to be programmed to scale with the number of processors on the chip to gain performance from future CMPs and *many-core processors* (MCP). This has been proved to be hard to accomplish for many reasons. Not all problems are possible to solve in parallel, and those that are might need more or less communication between processors, which will lower the performance. Programming in parallel also involves a lot more synchronization pitfalls, which do not exist in traditional sequential programming.

The performance difference between naively written C/C++ code and best-optimized code is called the *Ninja gap* [68]. This gap seems to grow with the number of processors, and without actions it can become a great performance bottleneck.

Due to these problems, there is clearly a need for research on alternative programming models which can simplify software development, and maximize the

[1] Power wall is the upper limit of power density in a circuit, due to keep the cooling costs low.

performance on chips with increasing number of processors, to lower the software development costs.

1.1 Purpose

The purpose of this thesis is to, by benchmarking evaluate the REPLICA architecture, a family of novel scalable CMPs with configurable emulated shared memory architecture, whose computation model is based on the PRAM model. By benchmarking different types of computation problems on REPLICA, similar parallel architectures and more diverse ones, we try to show how REPLICA is positioned among other existing architectures, both in performance and programming effort. Also, we try to find out if REPLICA is more suited for any special kind of computational problems.

1.2 Thesis Outline

This chapter gives the reader an introduction into the PRAM model, performance metrics and the used Berkeley dwarfs.

Chapter 2 gives an overview of the differently architectures and their programming languages that is used for the benchmark suite. The benchmark kernels are outlined in Chapter 3. The same chapter also describes how measurements are performed.

Evaluation and results are presented in Chapter 4. Conclusions and some proposals of future work are outlined in Chapter 5.

1.3 The PRAM Model

The *parallel random access machine* (PRAM) model can be seen as an extension of the *random access machine* (RAM) model which is used when describing sequential algorithms' time complexity [49]. While the RAM only has one processor, the PRAM consist of an arbitrary number of processors. This is naturally not a realistic assumption, but can be convenient when reasoning about parallel algorithms in general. Every processor in the PRAM shares the very same clock and memory. Sometimes the PRAM is described with one *instruction memory* for each processor, but in Figure 1.1 the instructions are stored in a *shared memory* because of simplicity. As in a RAM, one instruction takes exact one *time step* to execute in one of the processors of the PRAM [49]. It is important to distinguish between *time step* and *clock cycles*. Each time step consists of at least one clock cycle, further, all time steps do not need to be of the same time length. This means that the execution time for an instruction can vary in time, but instructions executed in parallel will always be in sync since all instructions always take exactly one time step. This applies even if there are different instructions executed in parallel. Since the processors execute instructions in sync, the programmer always know the state of

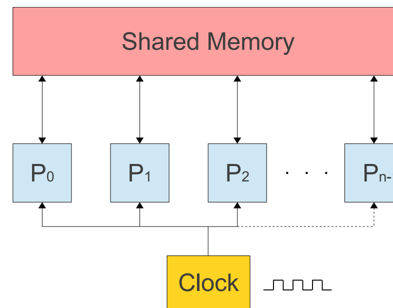


Figure 1.1: *An abstract parallel random access machine [49].*

all processors in the PRAM, unlike in other CMPs where parallel execution is not synchronous [37].

In this thesis the PRAM is a *multiple instructions multiple data* (MIMD) machine, which means that the processors do not need to execute the same instruction in the same time step, nor on the same data. This opens up the possibility that several processors within the same time step may try to access the same cell of the shared memory. It should however be clarified that read *and* write instructions do not access the memory simultaneously, even if they are executed within the same time step. Each time step is divided into three phases: a *read phase*, a *compute phase* and a *write phase* [49]. This implies that the read instructions always access the memory before write instructions do. If, and to what degree, concurrent memory accesses should be allowed, categorises the PRAM into these types [49]:

- **EREW:** *exclusive read, exclusive write* - every memory cell can only be read or written by one processor within the same time step.
- **CREW:** *concurrent read, exclusive write* - every memory cell can be read by multiple processors, but only one can write to it within the same time step.
- **CRCW:** *concurrent read, concurrent write* - multiple processors can read or write to the same memory cell concurrently. A convention is used to determine what will happen when more than one processor write to the same memory cell. There exist many suggested conventions, but this thesis will only bring up the ones mentioned in [49]:
  - **Weak:** multiple write accesses to the same cell are only allowed when writing a special value, for instance the value 0.
  - **Common:** multiple writing to the same cell is allowed when every writing processors tries to write the same value.
  - **Arbitrary:** if multiple processors write to the same memory cell, only one of processors will successfully write its value into the memory, and all the other processors' values will be lost.
  - **Priority:** the processors are assigned unique priorities, and when mul-

tiple processors write to the same cell, only the processor with highest priority will successfully write its value, and all the other processors' values will be lost.

– **Combining:** when multiple processors write to the same cell, the values are combined into a single value by some arithmetic function, such as addition.

As said before, the PRAM model has an arbitrary number of processors, and a such machine is not possible to manufacture. However any PRAM with a fixed number q of processors can be simulated by a PRAM with p processors in $\mathcal{O}(\lceil q/p \rceil)$ time steps [49].

1.4 Performance Metrics

In parallel computing *speedup* is a metric for the performance gained by executing an algorithm in parallel compared to serial [41]. The *absolute speedup* is defined as:

$$S_p = \frac{T_s}{T_p} \tag{1.1}$$

Where T_s is time for the best known sequential algorithm executing on a single processor, and T_p the time for a parallel algorithm executing with P processors [41]. Optimal speedup is obtained when $S_p = P$, which is called *linear speedup* [41]. Linear speedup is the theoretically maximum speedup that can be achieved, but in reality, there exist anomalies when the speedup exceeds the linear speedup, know as *superlinear speedup* [41].

This can, for example, occur if the available cache grows as the number of processors increases, resulting in that the single processor might need to do more expensive memory access [41]. Another example could be a string search algorithm which simply steps through the text until it finds the string it searches for. When the algorithm is executed in parallel, a second thread could search backwards. If the string that is searched for is placed at the end of the text, the second will find it and exit directly, which will lead to superlinear speedup, since it will execute faster than $T_s/2$.

Sometimes it is more convenient to use the parallel algorithm executed with a single processor, instead of the best known sequential algorithm when defining speedup. This is called *relative speedup* [49].

$$S_p = \frac{T_1}{T_p} \tag{1.2}$$

Where T_1 is the time for executing the parallel algorithm with a single processor. The relative speedup definition will mainly be used in this thesis.

Efficiency is a metric that describes how well an algorithm utilizes the processors [41]. An algorithm's efficiency is typically between zero and one, and defined

as:

$$E_p = \frac{S_p}{p} \tag{1.3}$$

Optimal efficiency is obtained when $E_p = 1$, which occurs when the speedup is linear [41].

1.4.1 Amdahl's Law

For an algorithm that executes in serial and has an execution time of T_s, the fraction of T_s that can be executed in parallel is defined as α, and the fraction that has to be executed in serial is defined as β. The time for executing this algorithm in parallel with P processors can be defined as:

$$\beta + \alpha = 1 \Rightarrow \alpha = 1 - \beta \tag{1.4}$$

$$T_p = \beta T_s + \frac{\alpha T_s}{P} = \beta T_s + \frac{(1-\beta)T_s}{P} \tag{1.5}$$

Then the speedup can be described as [42, 46]:

$$S_p = \frac{T_s}{T_p} = \frac{T_s}{\beta T_s + \frac{(1-\beta)T_s}{P}} = \frac{1}{\beta + \frac{(1-\beta)}{P}} \tag{1.6}$$

If the number of processors now goes to infinity we get an upper bound for the speedup that can be extracted by parallelism:

$$\lim_{P\to\infty} \frac{1}{\beta + \frac{(1-\beta)}{P}} = \frac{1}{\beta} \tag{1.7}$$

The upper bound of the speedup for different parallel fractions of an algorithm is shown in Figure 1.2.

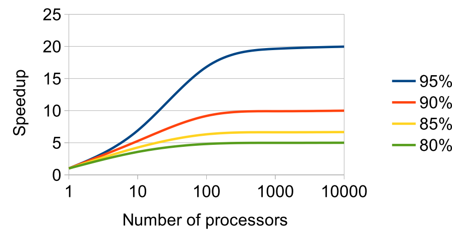


Figure 1.2: *The amount of speedup that can be extracted according to Amdahl's Law, based on the fraction of parallel work.*

It is important to point out that Amdahl's Law is a very pessimistic upper bound for speedup. The serial fraction of many algorithms is not constant, but depends on the problem size [46]. In many algorithms is the serial part heavily reduced when the problem size increases.

1.5 The Dwarfs from Berkeley

The technical report [18] from *University of California, Berkeley* suggest a number of application classes, called *dwarfs*, which should be used when evaluating parallel programming models and architectures.

A dwarf is an application class that captures a type of computation and communication pattern, which exist, and are likely to exist in many future applications [18].

Our benchmark suite will use the following dwarfs from the report:

- **Dense Linear Algebra:** This dwarf consist of dense vector and matrix operations, such as in BLAS [18]. These applications have often strided memory accesses, due to that matrices are represented as two-dimensional arrays [18]. The performance is typically limited by the computation capacity of the executing architecture [18].

- **Sparse Linear Algebra:** Due to a large number of zero values, the data sets are stored in some compressed format in order to reduce storage space and memory bandwidth required due to only accessing the nonzero values [18]. These applications have often irregular memory accesses, due to indirect addressing [18]. More about this in Section 3.4. The performance is limited by both memory bandwidth and computation capacity of the executing architecture [18].

- **Graph Traversal:** Applications that traverse graphs by visiting nodes and follow their edges [18]. These applications have irregular memory accesses, and do typically involve little computation [18]. The performance is limited by memory latency [18].

1.6 Related Work

The performance for regular and irregular work loads on the PRAM like architecture XMT [72] have been compared against the GTX280 graphics card from NVIDIA [27]. The comparison showed that XMT had an average speedup of 6.05 over GTX280 for applications with irregular work loads, and 2.07 times slowdown for the regular ones [27]. The XMT architecture has also been shown to outperform an Opteron processor from AMD [76]. The comparison was done by benchmarking a number of well known application algorithms.

The SB-PRAM architecture, a realization of the CRCW PRAM model, has been evaluated [66]. Its speedup for four application of the SPLASH and SPLASH-2 benchmark suite was compared against a cache based DASH [55] machine and the MIT Alewife machine [16, 66]. Overhead of the interconnection network was evaluated by comparing the execution time of the physical SB-PRAM against a simulated perfect shared memory [66]. The results showed a maximum overhead

of 1.37 %, and better speedup than the reference architectures for at least two of the SPLASH applications [66].

Past claims that GPUs have extensive speedups (between 10 and 1000 times) over CPUs for many important throughput computing kernels have been investigated by benchmarking a number of carefully selected kernels on an *Intel Core i7 960* CPU and a *Nvidia GTX280* GPU [54]. Optimization techniques and hardware features that can explain performance differences are discussed and analyzed for both architectures [54]. The investigation disputes the previous claims that GPUs have extensive speedups over CPUs, and claims that a GPU's speedup over a CPU is closer to 2.5 on average, according to their benchmark suite [54].

1.7 Some Available Parallel Benchmarks

NAS Parallel Benchmarks

The *NAS Parallel Benchmarks* (NPB) are developed and maintained by *NASA Advanced Supercomputer Division*, previously known as *Numerical Aerodynamic Simulation* (NAS) *Program* [5, 19]. Its purpose is to study and evaluate the performance of parallel supercomputers. The first version of NPB was released in a technical document in 1991, which only describes the problems that had to be solved algorithmically. Unoptimized sequential sample codes in Fortran were supplied, but were only to be considered as guidelines for implementations [19]. This gives the benchmarkers freedom to choose an implementation technique that best suits their architecture. The first version (NPB 1.0) consists of five smaller kernels, and three simulated applications, which derive from computational fluid dynamics applications.

When NPB 2.0 was released it was thought as a supplement, rather than replacement to the NPB 1.0. Unlike the first version, NPB 2.0 was specified with parallel source code using Fortran (and C later on) and MPI [4, 20].

Since then more benchmarks have been added, and also programming languages and models, such as OpenMP, High Performance Fortran and Java. The current version of NPB is 3.3 [5].

The High-Performance Linpack (HPL) Benchmark

The HPL benchmark is a portable software package for distributed-memory computers that solves a (double precision) dense linear system ($Ax = b$) which is randomly generated [67]. It requires a message passing interface (MPI) for communication, and an implementation of either *basic linear algebra subprograms* (BLAS) or *vector signal image processing library* (VSIPL) [67]. The HPL benchmark is used by the famous TOP500 site which ranks the fastest high performance computer systems in the world [57].

SPEC

The *Standard Performance Evaluation Corporation* (SPEC) is non-profit corporation that develops and sells standardized benchmarks, which measure the performance of different computer systems [7]. It was founded in 1988 by workstation vendors due to the need of a standardized performance tests [7]. SPEC also publishes vendors-submitted results on their site [7].

Rodinia

Rodinia is a heterogeneous benchmark suite which targets general purpose computing on multi-core CPUs and GPUs [70]. The choice of kernels have been greatly inspired by the Berkeley dwarfs [70]. All benchmarks are written with support for OpenMP, CUDA and OpenCL as parallel model [11].

1.8 Publications

Results of this thesis have also been partly published in [43] and [44].

2

The Architectures

This chapter gives an overview of the different architectures and their programming language that is used in this benchmark suite.

2.1 REPLICA

REPLICA (*Removing Performance and Programmability Limitations of Chip Multiprocessor Architectures* [32]) is a successor of the TOTAL ECLIPSE architecture [37], which is developed at VTT Oulu, Finland. It is a hybrid realization between the *arbitrary multioperation CRCW PRAM* and *Non-Uniform Memory Access* (NUMA) model [58]. With *multioperation* one has hardware support for operations that takes operands sent from more than one hardware thread, which are combined into a single result, like in the *combined PRAM*. From now on, when the text mentions *thread*, it is referring to *hardware thread*. Software threads are not considered unless it explicitly says so.

REPLICA has multiple MBTAC (*multibunched/threaded architecture with Chaining*) processors [37]. The number of MBTAC processors (P) in a REPLICA architecture are configurable, and so are the number of threads ($T_{processor}$), functional units, registers (for each thread), and memory units within each processor [37]. This thesis will consider configurations with 4, 16 and 64 MBTAC processors. These processors have three different configurations named T5, T7 and T11. The configuration name of a MBTAC processor refers to the number of functional units within the processor, where T5 has least units of the three, and T11 the most. For the different REPLICA configurations used in this thesis see Table 2.1. In the REPLICA project a FPGA prototype of the REPLICA architecture is currently under development, including an I/O and storage system [32]. Since no prototype is

REPLICA configuration name	Processor configuration name	P	$T_{processor}$
REPLICA-T5-4-512	T5	4	512
REPLICA-T5-16-512	T5	16	512
REPLICA-T5-64-512	T5	64	512
REPLICA-T7-4-512	T7	4	512
REPLICA-T7-16-512	T7	16	512
REPLICA-T7-64-512	T7	64	512
REPLICA-T11-4-512	T11	4	512
REPLICA-T11-16-512	T11	16	512
REPLICA-T11-64-512	T11	64	512

Table 2.1: *Considered REPLICA configurations. Number of processors (P), Threads per processors ($T_{processor}$). The REPLICA configurations are named according to following pattern:*
REPLICA-<PROCESSOR>-<P>-<$T_{processor}$>

available today, the benchmark will instead run on the cycle-accurate simulator IPSMSimX86, see Section 2.1.6.

2.1.1 Emulated Shared Memory

REPLICA is an *emulated shared memory* (ESM) machine [36, 32]. The shared memory of a PRAM is emulated with a cacheless distributed shared memory using a synchronous high-bandwidth communication network [37]. Memory modules are organized on-chip as a double acyclic two-dimensional multi mesh network [37], see Figure 2.1b. The data is routed through switches, and to avoid congestion each shared memory address is pseudo randomly placed among the memory modules by a polynomial hash function [37].

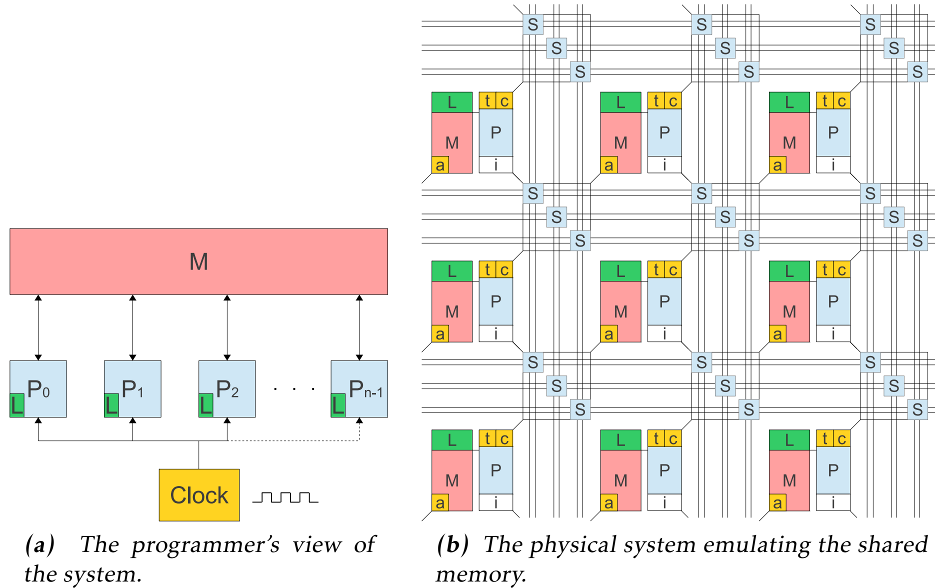
Instead of trying to remove high memory latency with caches, REPLICA hides the latency using multi-threaded processors. In short, while a thread is waiting for data from the shared memory, the processor executes other threads.

The cacheless solution makes memory accessing very time-consuming compared to cache-based systems, but since no caches are used, there is no need for a coherency protocol, which reduces communication over the network.

Figure 2.1 illustrates the communication network that emulates the much simpler PRAM model as it is viewed by the programmer.

2.1.2 Memory Modules

REPLICA has three different on-chip memory modules for each processor: *Shared data memory module*, *local data memory module* and *instruction memory modules* [37]. The shared data memory modules from all processors build together up the ESM, which stores all data that is shared between processors. The local data memory contains data that is private for a processor's threads [37]. An instruction memory stores the program code for each processor [37]. Instruction and local data



(a) The programmer's view of the system.

(b) The physical system emulating the shared memory.

Figure 2.1: *A comparison between the programmers view and the physical communication network that emulates the PRAM model (P=processor, M=shared memory, L=local memory, i=instruction memory, a=active memory unit, t=scratchpad, c=step cache, s=switch) [37].*

memory are accessed in one clock cycle [37]. To support multioperations and multiprefix operations the shared memory module has an *active memory unit* attached to its memory, which consists of a simple ALU and fetcher [37].

2.1.3 The MBTAC Processor

The MBTAC (*Multibunched/threaded Architecture with Chaining*) processor is a dual-mode *Very Long Instruction Word* (VLIW) processor that allows chaining [37]. The processor uses multi-threading, which is implemented with a $T_{processor}$-stage pipeline to hide shared memory access latency [37], see Figure 2.2.

PRAM and NUMA mode

Each thread can either run in PRAM mode, or together with one or more threads in NUMA mode [37, 39, 40]. As default, threads run in PRAM mode, but can under execution time be configured to NUMA mode [37]. NUMA threads can join with one or more NUMA threads into a *thread bunch* [37]. Threads on separate processors can not join [37].

Programs with low amount of *thread-level parallelism* (TLP) do not benefit from multi-threading, simply because they do not have enough parallel work to make use of all thread-slots/stages in the pipeline. By joining unused threads into

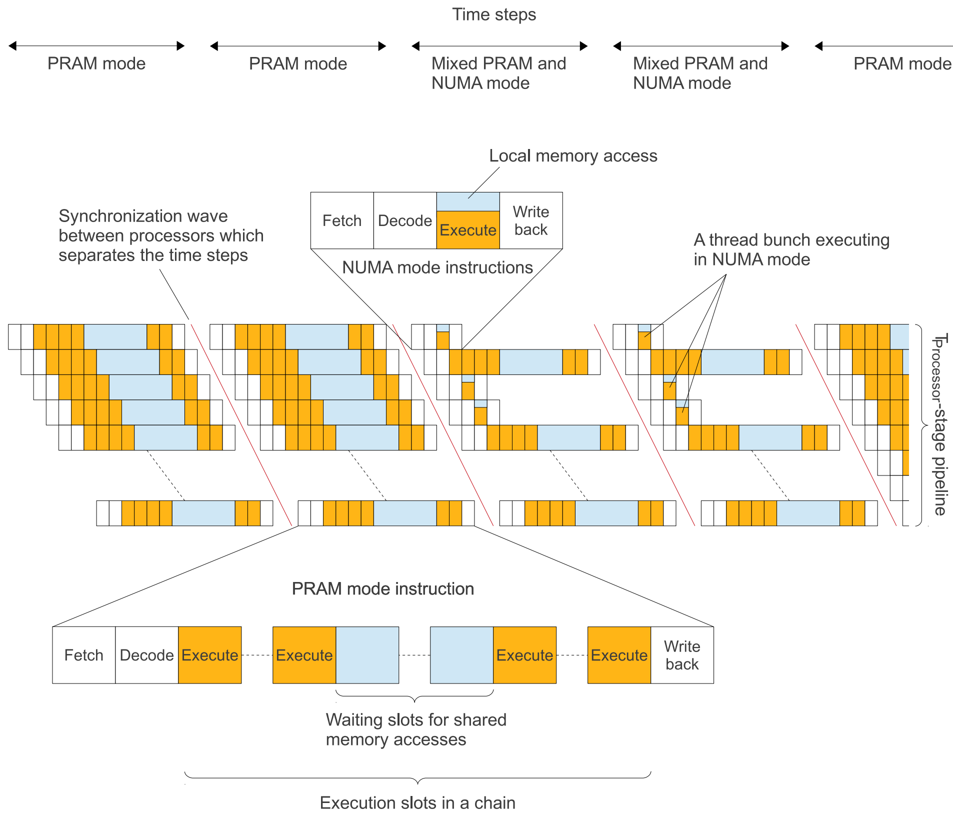


Figure 2.2: *The multi-threaded pipeline in a MBTAC processor [37].*

a thread bunch that executes the same code, one can make use of these empty thread-slots and execute sequential parts faster [37]. One time step in the PRAM model corresponds to that all threads have passed through the pipeline [37]. Resulting in that threads in PRAM mode execute one VLIW instruction per time step, while a thread bunch of n threads will execute n VLIW instructions. However, when a thread runs in NUMA mode, only local memory can be accessed efficiently, because of the high latency of the shared memory.

NUMA mode is out of scope for this evaluation.

Virtual Instruction Level Parallelism

One VLIW instruction consist of sub-instructions. Thanks to that the MBTAC processor has organized its functional units in a chain-like manner, these sub-instructions can have dependencies between each other [37]. This is called *virtual instruction level parallelism* (VILP) [37, 53]. It is however only possible to use VILP when the thread runs in PRAM mode. When a thread is running in NUMA mode, the functional units are organized as in a traditional VLIW processor [37].

Step Caches and Scratchpads

To speedup and reduce the number of memory accesses to the ESM, REPLICA uses *step caches* [37, 34]. This should not be confused with common caches, even if they work similarly. The main difference is that the step cache data is only valid within a single PRAM time step, and therefore coherency issues are avoided [37].

The step cache and scratchpad together form a filter that unburden the communication network.

When a thread wants to access data from the shared memory, it first searches through the step cache attached to its processor. A hit for a read access to means that another thread on the processor within the same PRAM time step already has received the requested data [37]. Now the read instruction can be completed by fetching the data from the step cache. Because REPLICA is an arbitrary CRCW PRAM, a write instruction can be ignored if a write access already has occurred within the same time step. If the search fails when trying to read, a request is sent to the ESM which is noted by the step cache [37].

Due to limited associativity of the step cache, it does not support multioperations or multiprefix operations, thus a *scratchpad* [35] is connected to the step cache which is used to store memory access data [37]. There exist two types of multioperations. The first type is a single instruction, `M`x, where the symbol "x" should be replaced with any of following operations: `ADD`, `SUB` (for subtraction), `AND`, `OR`, `MAX` or `MIN`, which does not use the step cache or scratchpad. Instead the operation is performed by the active memory unit at the shared memory module which holds the operand's address [37].

The second type consist of two instructions, `BM`x and `EM`x. With help from the scratchpad, `BM`x performs a reduction of the multioperation among the threads at the local processor without accessing the shared memory [37]. Instead, the step cache is used as a temporary target [37]. `EM`x finishes the multioperation by performing a reduction between the processors against the shared memory [37]. The two-step multioperations are to prefer over the single step ones when at least $\sqrt{T_{total}}$ threads, where T_{total} is the total number of threads, are performing the same multioperation, due to better performance [37].

The multiprefix operations work similar to multioperations, besides that a thread that executes the multiprefix operation also receives the value of the memory location that it had before the thread's operation was applied on it [37]. Both the single and two multiprefix instructions types are arbitrary ordered, meaning that there exists no specified order in which the threads are executing their operation on the memory location [37].

REPLICA does also have hardware support for ordered multiprefix operations, which is a sequence of three instructions: `BMP`x, `SMP`x and `OMP`x [38]. The threads then receive their values as if the operations were executed ordered based on the thread ID. The different multioperations and multiprefix operations are summarized in Table 2.2.

<table>
<tr><th rowspan="2">Operation</th><th colspan="3">Number of instructions:</th></tr>
<tr><th>1</th><th>2</th><th>3</th></tr>
<tr><td>Multioperations</td><td>Mx</td><td>BMx
EMx</td><td>-</td></tr>
<tr><td>Arbitrary ordered multiprefix operations</td><td>MPx</td><td>BMPx
EMPx</td><td>-</td></tr>
<tr><td>Ordered multiprefix operations</td><td>-</td><td>-</td><td>BMPx
SMPx
OMPx</td></tr>
</table>

Table 2.2: *Multioperation and multiprefix operation types. Multioperations and arbitrary ordered multiprefix operations can be executed with one or two instructions, depending on if the scratchpad should be used. Ordered multiprefix operations have to execute three instructions [37].*

2.1.4 The Baseline Language

REPLICA architectures can be programmed using a baseline language, which is a low level language with the parallel concepts of E[1] and Fork [32]. The baseline language is based on the ANSI C standard with assembler inlining and macros to support multioperations [58, 81]. A program written in REPLICA baseline is executed by all hardware threads from beginning to end according to the common *single program multiple data* (SPMD) style, and no software threads can be spawned.

The language has a simple convention to distinguish between private and shared variables. If a variable name ends with the character "_" it is a shared variable, else it is private [81]. This way it is possible to declare shared and private variables and still keep the syntax of C. In contrast to shared variables, built-in macros and variable names begin with "_". For built-in macros and variables see Tables 2.3 and 2.4.

Macro name	Description
`_start_timer`	Starts the simulator's timer.
`_stop_timer`	Stops the simulator's timer.
`_synchronize`	Synchronize the threads within a group.
`_exit`	Halts the simulator.
`_prefix()`	Macro for multiprefix operations.
`_aprefix()`	Macro for arbitrary multiprefix operations.
`_multi()`	Macro for multioperations.

Table 2.3: *Built-in macros for the baseline language [81].*

[1] The E language was developed for REPLICA's predecessor TOTAL ECLIPSE [33].

Variable name	Description
_thread_id	The thread's current ID number within its group.
_absolute_thread_id	The thread's absolute ID number.
_group_id	The thread's current group ID number. This is actually a pointer to the group's synchronization variable [81].
_number_of_threads	The number of threads within the thread's current group.
_absolute_number_of_threads	The total number of threads.
_private_space_start	Pointer to the start of the thread's private memory space.
_shared_heap	Pointer to the shared heap.
_shared_stack	Pointer to the shared stack.
_video_buffer_	An array allocating shared space for pixels of the screen.

Table 2.4: *Built-in variables for the baseline language [81].*

```
1  #include <replica.h>
2  #define N 10000
3
4  int a_[N];
5  int sum_ = 0;
6
7  int main()
8  {
9      int i;
10     for (i=_thread_id; i<N i+=_number_of_threads)
11     {
12         a_[i] = i;
13     }
14     _synchronize;
15
16     _start_timer;
17     for (i=_thread_id; i<N i+=_number_of_threads)
18     {
19         _multi(ADD, &sum_, a_[i]);
20     }
21     _synchronize;
22     _stop_timer;
23     _exit;
24     return 0;
25 }
```

Listing 2.1: *Baseline language example program.*

Listing 2.1 contains a simple baseline program which shows how these built-in macros and variables can be used.

The program first assigns values to the shared array a_ in parallel. Then the sum

of all values in the array is calculated and stored in the variable `sum_` using the multioperation macro `_multi()`.

Code written in the baseline language can be compiled using REPLICA's back-end compiler, which is based on LLVM [81]. The REPLICA benchmark suite is developed in the baseline language.

2.1.5 The REPLICA Language

To improve productivity a new easy-to-use high-level parallel programming language called *REPLICA language* is being developed [58]. Among many things, the new language will have a runtime library with support for handling threads, groups and tasks [58]. It will also provide standard parallel algorithms and generic data structures through the library [58].

Basic synchronous and asynchronous control constructs, such as `for`, `if`, `while`, `do` and `switch` will be built into the language [58]. Programmers will also be able to declare their own synchronous and asynchronous functions [58].

It will be possible to declare sequential blocks where threads join into a bunch which executes in NUMA mode in order to utilize the whole processor pipeline [58]. After the sequential block, the thread bunch can split back into separate threads, executing in PRAM mode.

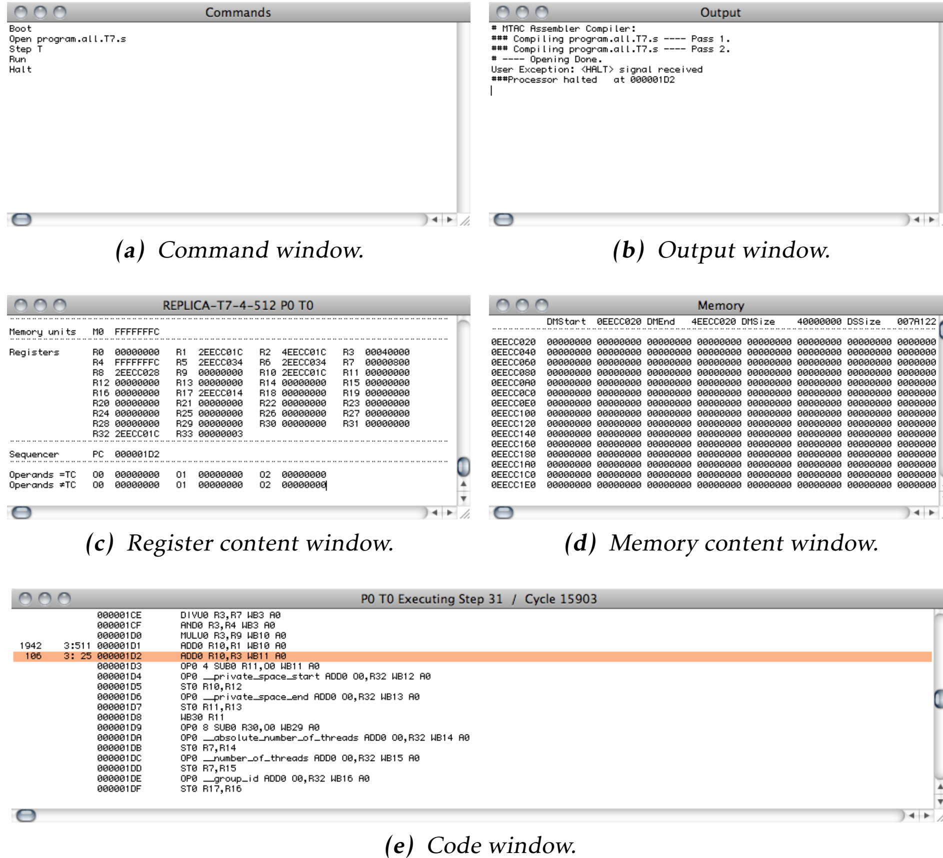
The REPLICA language will not be used in this evaluation

2.1.6 IPSMSimX86

IPSMSimX86 is a cycle-accurate simulator originally developed for REPLICA's predecessor ECLIPSE, but it has been updated to simulate different REPLICA configurations as well. The simulator allows the user to execute each instruction step by step, or run through a whole program. It is also possible to step multiple instructions and halt a running program. After simulating, IPSMSimX86 can generate a lot of statistics. Such as, execution time, frequency of different instructions, ratio of taken branches and total number of executed instructions.

In Figure 2.3 some of the simulator's windows are displayed. The *command window* shows the command history [81]. Messages and errors from the simulation are printed out in the *output window*. The window in Figure 2.3c displays the content of the registers. The *memory content window* displays the content of the whole memory [81]. The current executing instruction is marked in the *code window*, and the numbers in its far left tells how many threads are currently executing the instructions [81].

There also exist windows that display the value of global variables, statistics, and the screen. The screen is mapped to a specific address space (through the `_video_buffer_` array), which makes the pixels easy to modify.



(a) *Command window.*

(b) *Output window.*

(c) *Register content window.*

(d) *Memory content window.*

(e) *Code window.*

Figure 2.3: *Some of IPSMSimX86's windows.*

2.1.7 Limitations

The current REPLICA configurations do not support floating point operations, but these could be included as easily as for any other architecture [37]. Mass storage is currently not supported, however, the simulator has support for reading and writing from the host's (the computer running the simulator) file system using UNIX-like system calls.

2.1.8 Previous REPLICA Works

Andreas Lööw's master thesis [56] describes the need for a new simulator for the REPLICA architecture, issues with simulating PRAM architectures, and how it could be implemented. The resulting simulator was tested, evaluated, and compared against the current simulator. The main goal was to speedup simulation time rather than be user friendly.

Daniel Åkesson implemented the first version of the REPLICA compiler back-end using the LLVM [3] compiler framework, which will be a part of REPLICA's future tool-chain used to developing programs for REPLICA with a high-level and easy-to-use programming language [81]. The compiler takes code written in REPLICA's baseline language and generates assembler code for REPLICA. It also has the ability to do some optimizations and makes better use of REPLICA's instruction level parallelism by using a greedy scheduling algorithm [81].

There exists a source-to-source compiler, from the Fork language to REPLICA's baseline language, that is described, verified, and tested in the master thesis by Cheng Zhou [80]. Results also show that the execution overhead that is introduced by the Fork language compared to REPLICA's baseline language is little.

2.2 Xeon X5660

Xeon X5660 is a 64-bit server/workstation multi-core CPU from Intel [15]. It is a *symmetric multiprocessing* (SMP) microarchitecture with a shared 12 MB level 3 cache [15]. The chip has 6 hyper-threaded processors (cores) with a clock rate of 2.8 GHz [14]. *Hyper-threading* is Intel's trademark for its *simultaneous multi-threading* (SMT) technology [14]. A SMT processor has execution units that can execute instructions from more than one hardware thread within the same clock cycle. Xeon X5660 has 2 hardware threads per processor due to hyper-threading, and 12 threads in total [14]. When the processor is operating under its power and temperature limits due to low utilization, it can automatically speed up the clock rate over the normal frequency, up to 3.2 GHz [14, 15]. Intel calls this for *turbo boost*, and it can be used to increase performance for both single and multi-thread execution [15].

Each processor has its own level 1 and level 2 cache that can hold 32 kB and 256 kB, respectively [15].

Xeon also has support for SIMD instructions which can speed up vectorized computations [15].

There are significant differences between Xeon and REPLICA. When Xeon relies on caches to overlap the gap between its own and the main memory's clock rate, REPLICA tries to hide the latency with multi-threading. To some extent, Xeon also hides memory latency using hyper-threading, but not in the same degree as REPLICA. Since Xeon is cache-based, it needs a coherence mechanism to keep its cache coherence. This mechanism adds not only overhead, but also performance issues, such as *cache misses* and *false sharing* [41]. REPLICA, which does not use caches, does not have to deal with this issues. Xeon is an architecture with its main focus on ILP hardware, compared to REPLICA that concentrates on TLP using thousands of hardware threads. Neither is Xeon executing its threads synchronously.

2.2.1 Xeon Machine Setup

The Xeon machine did run Debian squeeze with Linux kernel 2.6.32-5-amd64 (x86_64), and GCC version 4.4.5 installed. All kernels were compiled with O2 optimizations.

2.2.2 OpenMP

OpenMP (open multi-processing) is a popular *application program interface* (API) standard, which provides a parallel programming model for shared memory architectures [65]. It uses the SPMD style, together with a fork-join model [65]. An OpenMP program starts executing in a sequential mode, from which it can spawn a desired number of software threads. The OpenMP specification versions are defined by the non-profit corporation *OpenMP architecture review board* (OpenMP ARB) which owns the OpenMP brand [6]. They do not implement the OpenMP API, but rather provide the specifications. The OpenMP ARB members consist of hardware and software vendors/organizations that produce products for OpenMP, or have great interests of the OpenMP standard [6]. The GCC version that we used supports the OpenMP 3.0 specification.

Sequential C, C++ and Fortran code can, with relative little effort, be parallelized using the OpenMP API. This is done by adding OpenMP *directives*, which specifies the OpenMP behavior. In C and C++ OpenMP directives are based on the `#pragma` compiler directives, and in Fortran comments are used instead [41, 65].

OpenMP directive in C/C++:

```
#pragma omp directive-name [clause list]
```

The intention with OpenMP directives is that is should be possible to write code that can be compiled for both serial and parallel architectures, though it is up to the programmer to make sure that both versions produce the same result [65].

Listing 2.2 shows an OpenMP program that sets the values of an array, calculates, and prints out the sum of the array. It can be compiled both with and without OpenMP support, and still produces the same result. The program starts executing sequentially until it reaches the first `parallel` directive, which will spawn a team of software threads. The region/scope/block after the `parallel` OpenMP directive will be executed in parallel by the team. The thread that executes the `parallel` directive becomes the *master thread* of the team, and will get *thread ID* 0 [65]. The number of threads that will be created, including the master thread, can be set by the `num_threads` clause in the directive's *clause list*, the environment variable `OMP_NUM_THREADS`, or at runtime using the runtime library routine `omp_set_num_threads(int` *num_threads*`)` [41]. However, the behavior of the program is implementation specific if the requested number of threads is higher than what the implementation supports [65]. The first directive in Listing 2.2 is directly followed by a `for` (or loop) directive, which specifies that the team inside the parallel region will cooperate by executing the for-loop's iterations in parallel. Iterations are distributed to threads in the team according

to default scheduling, or by a schedule specified through the directive's clause list. The default scheduling method is implementation specific and can not be changed [65]. At the end of the loop, there is an implicit barrier where all threads within the team wait until all iterations are executed [65]. This barrier is however optional, and can be removed by the `nowait` clause [65]. In this example the parallel region ends directly after the loop, so this barrier will have no effect. The parallel region also has an implicit barrier at its end, and only the master thread will continue executing thereafter [65].

```
1  #include <stdio.h>
2  #include <omp.h>
3  #define N 10000
4
5  int main()
6  {
7      int i;
8      int a[N];
9      int sum = 0;
10
11     #pragma omp parallel
12     { // Parallel region
13         #pragma omp for
14         for(i=0; i<N; ++i)
15         {
16             a[i] = i;
17         }
18     }
19
20     #pragma omp parallel for default(none)    \
21                             private(i)        \
22                             shared(a)         \
23                             reduction(+:sum)\
24                             schedule(static)
25     for(i=0; i<N; ++i)
26     {
27         sum += a[i];
28     }
29     printf("Sum: %d\n", sum);
30     return 0;
31 }
```

Listing 2.2: *OpenMP example program.*

The next directive in Listing 2.2 is the `parallel for` (or parallel loop), which is a shorthand for a `parallel` directive that only contains a `for` directive in its parallel region [65]. The first parallel region could also be specified in the same way by replacing the two pragmas with `#pragma omp parallel for`. It is also possible to place both `for` directives inside the first `parallel` directive's parallel region.

In the `parallel for` directive we have a few clauses. The `default(`*shared*|*none*`)` clause lets the user decide whether the data-sharing attribute of variables inside the parallel region should be implicitly set to *shared*, or if they have to be explicitly specified [65]. For a *shared* variable their exists only one instance of the vari-

able which all threads within the team share. If a variable is *private*, all threads in the team have their own instance of it. When the `default` clause is not used, or is set to *shared*, all variables are implicitly shared except for loop counter variables which are private [65].

The `private(`*variable-list*`)` clause sets the data-sharing attribute of variables in its list to *private*, and `shared(`*variable-list*`)` sets its variables to *shared* [65].

`reduction(`*operator*`:`*variable-list*`)` specifies that the variables in its list should be reduced using the given *operator*. Each thread in a team will get a private copy of the variables in the list. The private variables will be initialized to an appropriate value, and at the end of the parallel region be reduced back to a single variable using the specified operator. For legal operations and their initial values see Table 2.5.

Operator	Initialized value
+	0
*	1
−	0
&	~0
|	0
^	0
&&	1
||	0

Table 2.5: *Reduction operators and initialized values for the variable [65].*

The clause `schedule(`*kind[, chunk-size]*`)` specifies how the loop iterations are mapped to the threads [65]. There exist five different kinds of scheduling that can be set:

- `static`: Iterations are divided into chunks of size *chunk-size*, which are statically mapped to the threads in a round-robin fashion [65]. If no *chunk-size* is specified, the iterations are divided into at most as many chunks as there are threads inside the team [65].
- `dynamic`: Iterations are divided into chunks of *chunk-size*, and distributed to the threads as they become idling [65]. As default, *chunk-size* is set to 1.
- `guided`: Works similar to `dynamic`, except that the size of each chunk is proportional to the *number of unassigned iterations* divided by the *number of threads in the team* [65]. Here *chunk-size* specifies the minimum size of a chunk, which is set to 1 as default [65].
- `auto`: The scheduling technique is implementation specific [65]. The compiler is free to choose any mapping of iterations to threads.
- `runtime`: The scheduling technique is determined at runtime, rather than during compilation. This is done by the environment variable `OMP_SCHEDULE`,

or RTL routine `omp_set_schedule(omp_sched_t` *kind*, `int` *modifier*)-[65]. The second parameter, *modifier*, can be used to set the *chunk-size* [65].

In a parallel region it is possible to define sections that can be executed in parallel. This is done using the `sections` and `section` directives [65]. Inside the `sections` directive are a number of code blocks defined, which are assigned to threads in the team [65]. The code blocks are specified with the `section` directive. Each section is only executed once. Here is a short syntax example of how these directive can be used:

```
#pragma omp sections [clause list]
{
    #pragma omp section
    {/* Code block 1 */}
    #pragma omp section
    {/* Code block 2 */}
              ⋮
    #pragma omp section
    {/* Code block N */}
}
```

Synchronization and Memory Consistency in OpenMP

To synchronize a team of threads inside a parallel region, the `barrier` directive can be used. No thread within the team can continue to execute beyond the `barrier` directive before all the threads in the team have reached it [65]. Some directives have an implicit barrier at its beginning or end [65].

If the programmer wants to have some code executed in serial inside a parallel region, either the `single` or `master` directive can be used. The thread that first reaches the *single* directive will execute the serial region [41]. After the serial region there is an implicit `barrier`, but it can be removed by specifying the `nowait` clause [41]. The `master` directive works similarly but is always executed by the master thread, and has no implicit barrier [41].

OpenMP uses a relaxed memory consistency model where all OpenMP threads have their own *temporary view* of the shared memory [65]. This allows the compiler to store shared variables in registers instead of always loading them from the shared memory [65].

The `flush[`*variable-list*`]` directive enforces consistency between the thread's temporary view and the shared memory [65]. The programmer can specify which shared variables should be flushed in the optional variables list. If no variables are specified, all shared variables are flushed. The `flush` directive is also used implicitly by other directives, such as, `barrier`, `critical`, `parallel`, `parallel for` and `parallel sections` [41]. In combination with the `nowait` clause, `flush` is not implied [41].

Sometimes the programmer wants to make sure that a certain block of code only is executed by one thread at time. This is often done using mutexes. OpenMP provides the `critical` directive, which only lets one thread at a time execute its

content [65]. To avoid race conditions for a single shared variable, the `atomic` directive can be used. It ensures that the following variable assignment is updated atomically, and protects it from simultaneous writing [65].

For more details see the OpenMP specification [65]. Note that this is not a complete documentation of the OpenMP API and a lot of its content has not been mentioned.

2.3 XMT

The XMT (Explicit Multi-Threading) project started at the University of Maryland in 1997 [72]. Its goal is to build an easy-to-program parallel processor, by supporting a PRAM-like programming model [75].

The architecture runs in either serial or parallel mode [75], see Figure 2.4. Since the speedup that can be achieved by TLP is limited by the serial fraction of the program, the XMT project has proposed a relatively fat processor, called *master thread control unit* (MTCU), to execute the serial parts of programs [75]. The MTCU is very similar to a normal cache-based serial processor. The main difference is that the MTCU can go from serial mode to parallel mode by spawning an arbitrary number of threads, which is executed by lightweight *thread control units* (TCUs) [75]. During parallel mode the MTCU is inactivated [75].

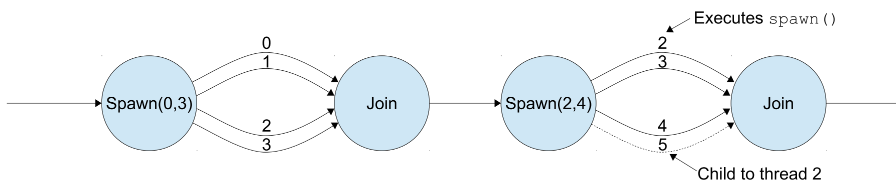


Figure 2.4: *The serial and parallel execution modes of XMT [75].*

The project is slightly more mature than the REPLICA project, and a 75 MHz prototype with 64 TCUs has already been synthesized on 3 Xilinx FPGAs [76]. Their next step is to build a 800 MHz ASIC prototype with 1024 TCUs [75], see the block diagram in Figure 2.5.

This thesis will look at the ASIC prototype, and the default configuration for the architecture will be used when simulating.

Each TCU executes independently in its own speed, and the instructions are not executed synchronously as in a REPLICA architecture [75]. This means that a TCU can execute a thread from its start to its end without ever needing to wait for any other thread. Since accesses memory through the interconnection network are very time consuming, each TCU has a private *prefetching buffer* which can prefetch values in advance [75]. The compiler is responsible for inserting prefetching instructions [75].

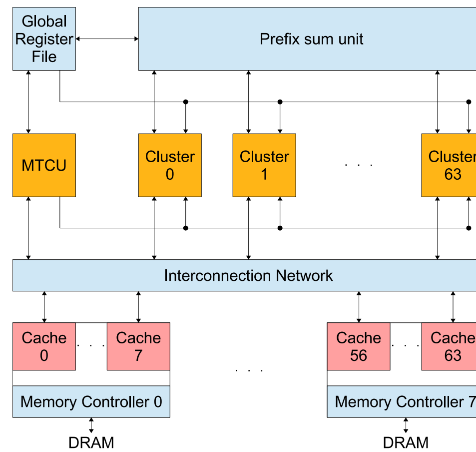


Figure 2.5: *Block digram of XMT [75].*

The 1024 TCUs are grouped into 64 clusters with 16 TCUs in each [75]. Larger functional units, such as multiplication and division, are shared among the TCUs within the same cluster, while smaller functional units and registers are private for each TCU [75]. Each cluster can therefore be seen as a SMT processor [75]. To speedup memory accesses and still avoid cache coherency problems, each cluster has a *read only buffer* which is used to store values that will not change during the current parallel execution [75]. The compiler is responsible for storing read safe data into the read only buffer [75].

The interconnection network that connects clusters with the memory cache modules is a *mesh of trees* network [75, 22]. Memory accesses from multiple clusters to a single cache module are queued and handled serially [75]. The memory address space is divided evenly among the memory modules, and is hashed in order to reduce congestions [75]. Each memory module also has support for the `psm()` (*prefix sum to memory*) operation which is an atomic fetch-and-add [75].

The clusters share a *prefix sum unit*, which enables the `ps()` (*prefix sum*) operation to perform a fast atomic fetch-and-add on any of the 8 global registers [75]. The prefix sum is first computed locally on each cluster before begin summed up in the prefix sum unit [75]. The `ps()` operation only allows TCUs to add 0 or 1 to the global register [75]. The programmer can use global registers by declaring variables as `psBaseReg` [75]. Since 2 global registers are used for managing the lower and upper ID boundary for spawned threads, the programmer is limited to only declare 6 `psBaseReg` variables [75]. The `psBaseReg` has to be declared as global, and can be accessed as a regular variable by the MTCU during serial mode execution [75]. During execution in parallel mode by the TCUs, the `psBaseReg` can only be accessed through the `ps()` operation [75]. Therefore is it only possible to set the `psBaseReg` variables in serial mode.

2.3.1 XMTC

XMT can be programmed using the SPMD-style language XMTC, which is an extension of C [21]. Code written in XMTC can then be compiled by XMT's GCC (v 4.0) based compiler [48].

The spawning statement `spawn()`, see Figure 2.4, carries the two parameters *low* and *high* which specify the lower and upper bound ID for the spawned threads [75]. As mentioned earlier, *low* and *high* are stored in the global register file [75]. When entering a parallel region each TCU will execute a thread at a time, starting with the thread which has the same ID as the TCU [75]. The thread ID within a parallel region can be accessed through the built-in variable `$` [21]. If the number of spawned threads exceeds the total number of TCUs, then the remaining threads will be executed as TCUs finish their threads and become idling [75]. A TCU is assigned a new thread by increasing the lower bound register using the `ps()` operation, which will receive a new thread ID for the TCU to execute [75]. This is repeated until all spawned threads are executed. The flowchart in Figure 2.6 illustrates the assignments of threads to TCUs. This hardware implementation provides an efficient dynamic scheduling of the spawned threads [75].

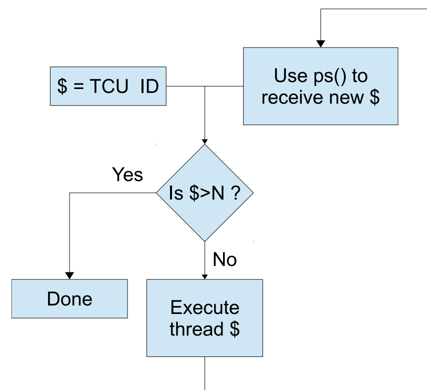


Figure 2.6: *The flowchart illustrates how* **N** *threads are assigned to a TCU [76].*

The spawning statement also makes the MTCU to broadcast the instructions within the parallel region to all clusters where they are stored [75]. Since the instructions are stored locally for each cluster, the number of instructions within a parallel region is limited [75, 21]. This can however by solved by letting the TCUs load instructions from the shared memory through the interconnection network, for larger parallel regions [75]. Nested `spawn()` statements are serialized using a loop and will not be executed in parallel [21].

The `sspawn()` statement makes it possible to spawn a single thread from within a parallel region, by simply incrementing the upper bound ID using the `ps()` operation [75]. The `sspawn()` statement takes one parameter which will receive the ID of the newly spawned thread, which can be accessed inside the initial-

ization block of the sspawn() statement [75, 21]. The new thread will start its execution at the beginning of the parallel region [75]. As soon as the upper bound ID is incremented, any idling TCU can start executing the new thread. The flowchart in Figure 2.6 does not describe this feature. If data have to be initialized for the new thread, some kind of synchronization mechanism between the parent and child thread has to be implemented by the programmer, so that the child thread waits for its parent to initialize the child's data [75]. This invalidates the restriction of independently executing TCUs to some degree [75].

Listing 2.3 displays an XMTC example program. The first spawn() statement spawns N threads which initialize the values in array all. The next spawn statement spawns N threads which copy all values greater than N/2 in array all to array smaller.

```
1  #include <xmtc.h>
2  #define N 10
3
4  psBaseReg smaller_count;
5
6  int main()
7  {
8      int all[N];
9      int smaller[N];
10     smaller_count = 0;
11
12     spawn(0, N-1)
13     {
14         all[$] = $;
15     }
16
17     spawn(0, N-1)
18     {
19         if(all[$] < N/2)
20         {
21             int index = 1;
22             ps(index, smaller_count);
23             smaller[index] = all[$];
24         }
25     }
26
27     return 0;
28 }
```

Listing 2.3: *XMTC example program.*

The XMTC compiler currently does not support any function calls within parallel regions, but will be supported in the future [21]. This restricts the programmer from using programming paradigms such as recursion.

Currently the TCU's can only store local variables and temporary values inside their registers [21]. This means that XMT can not deal with register spilling inside spawn blocks. Therefore special care has to be taken when declaring variables.

For more details of the XMTC language, see XMT's toolchain manual [21].

2.3.2 XMTSim

XMTSim is a cycle-accurate simulator for the XMT architecture [48]. The DRAM is however not simulated, but rather modeled as latency components [48]. The simulator can be used to simulate different XMT configurations [48]. As mentioned earlier, the default XMT configuration with 1024 TCUs will be used for our simulations. The simulated DRAM clock frequency is $\frac{1}{4}$ of the frequency for the XMT chip, and the latency of the DRAM is 20 DRAM clock cycles [47].

XMTSim does not simulate all features of the MTCU accurately, which will make it serial execution less efficient [47]. But since the serial fractions of the benchmark kernels are small, it will have not have any significant effects on the number of executed clock cycles [47].

2.4 Tesla M2050

Tesla M2050 is a GPU based on the Fermi architecture and manufactured by NVIDIA [13]. *Tesla* is in this case the class name for NVIDIA's server boards, and should not be confused with the architecture family name *Tesla* [77]. Ever since 2003 have shading languages, such as OpenGL and DirectX, been used for *general purpose computing on graphics processing units* (GPGPU) [12]. But since the APIs were designed for graphical computations, the programs needed to be translated into a graphical problem, which led to extensive programming effort and restrictions for the programmer [12].

In 2006 NVIDIA released the *compute unified device architecture* (CUDA) *Tesla* together with the *CUDA Toolkit*, in order to facilitate the general purpose GPU programming [12, 77]. The CUDA Toolkit made it possible to program the massively parallel GPU architecture using the C extended CUDA language [12]. NVIDIA shipped their second CUDA capable architecture *Fermi* with double precision support in 2010, and later the *Kepler* architecture [77].

As mentioned earlier, Tesla M2050 is a Fermi architecture with compute capability 2.0 [62, 13]. The compute capability is specified by the major and a minor revision number of a CUDA device [62]. It has 14 multi-threaded *streaming multiprocessors* (SM) with a clock rate of 1.15 GHz, see Figure 2.7, and 3 GB dedicated global device memory [13]. The memory is clocked at 1.55 GHz [13]. Each SM has a 64 kB on-chip memory which is used both as shared memory and level 1 cache [62]. The on-chip memory can be configured to either a 48 kB shared memory and a 16 kB level 1 cache, or to a 16 kB shared memory and a 48 kB level 1 cache [62]. The shared memory is manged explicit by the programmer [62]. Tesla also has a unified level 2 cache of 786 kB, which is shared by all SMs [12].

The SM is a *single instruction multiple thread* (SIMT) processor designed to execute hundreds of threads [62]. The instructions for each executing thread is pipelined in order to exploit ILP [62]. A SM executes a single instruction for a group of 32

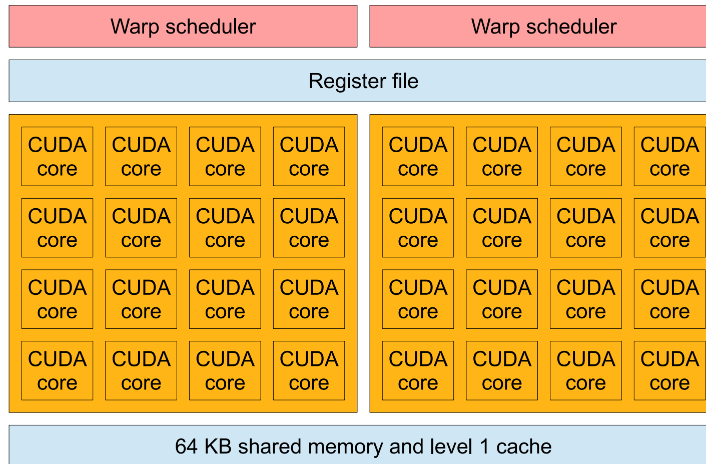


Figure 2.7: *The streaming multiprocessor [12].*

threads simultaneously [62]. The group is called a warp [62]. It works similarly to a SIMD architecture [62]. If the control flow for threads within the same warp diverges due to branches, the different control flows are serialized and executed one by one [62]. This means that all branches that any of the threads within a warp will take, have to be taken for all threads in the warp. Threads which should not execute the current instruction are disabled. Programs should therefore be written so that the control flow within a warp does not diverge, or it can have significant impact on performance.

Warps are executed independently and scheduled by *warp schedulers* [12]. Each SM has two *warp schedulers* which enable a SM to execute two warps concurrently [12]. The warp scheduler selects one warp each and its instruction is executed by 16 *CUDA cores* and four special functional units [12]. No dependency checks have to be issued since warps execute independently [12].

Each SM has 32 CUDA cores, which gives Tesla 448 CUDA cores in total [13].

2.4.1 CUDA C

CUDA C makes it possible to write *CUDA kernels* in C and execute them on a CUDA device [12]. A CUDA kernel is executed by a *grid* of *thread-blocks*, see Figure 2.8. The number of thread-blocks and threads within each thread-block is configurable, but the maximum number of threads in each thread-block for the Fermi architecture is 1024 threads [62]. To decide the optimal size of the thread-blocks for performance is not trivial, and is often determined by testing. The threads and thread-blocks can be indexed in one, two or three dimensions. Threads are locally indexed within each thread-block. The indices of a thread or thread-block are stored in the three-dimensional built-in vectors `threadIdx` and `blockIdx` [62]. The size of each dimension for the grid and thread-blocks can be accessed by the built-in vectors `gridDim` and `blockDim` [62].

Each thread-block is mapped and executed by a SM independently [62]. A SM can hold multiple thread-blocks concurrently, which is useful since a single thread-block normally does not contain enough threads to hide the memory and instruction latency [77].

The number of thread-blocks that can be executed in parallel depends on the number of available SMs on the chip [62]. The threads within the thread-blocks are divided into warps which are executed by the SM's CUDA cores [62]. Threads within the same thread-block can be synchronized by the `__syncthreads()` function, which works as a barrier for the threads in a thread-block [62]. To synchronize between thread-blocks the CUDA kernel has to be split into two CUDA kernels, where the synchronization is desired.

```
1  __global__ void cuda_kernel(int *matrix, int N)
2  {
3      int i = blockIdx.y*blockDim.y + threadIdx.y;
4      int j = blockIdx.x*blockDim.x + threadIdx.x;
5      if(i < N && j < N)
6      {
7          int index = i*N +j;
8          if (i == j)
9              matrix[index] = 1;
10         else
11             matrix[index] = 0;
12     }
13 }
14
15 int main()
16 {
17     int host_matrix[N*N];
18     int *device_matrix;
19
20     // Allocated memory on device
21     cudaMalloc(&device_matrix, N*N*sizeof(int));
22
23     // Setup grid and thread-block sizes
24     dim3 dimBlock(32, 32);
25     dim3 dimGrid(1+(N-1)/dimBlock.x, 1+(N-1)/dimBlock.y);
26
27     // Invoke cuda kernel
28     cuda_kernel<<<dimGrid, dimBlock>>>(device_matrix, N);
29     cudaDeviceSynchronize();
30
31     // Copy the result from device to host memory
32     cudaMemcpy(host_matrix, device_matrix, N*N*sizeof(int),
           cudaMemcpyDeviceToHost);
33
34     // Free allocated memory on device
35     cudaFree(device_matrix);
36
37     return 0;
38 }
```

Listing 2.4: *CUDA-kernel example.*

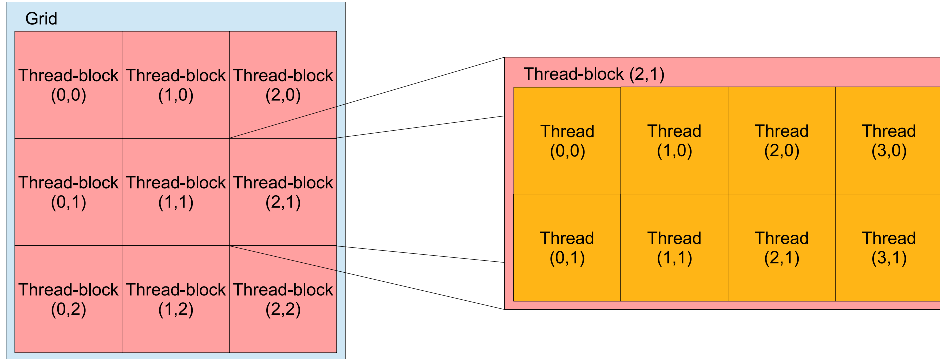


Figure 2.8: *A two-dimensional 3×3 grid with its 2×4 thread-blocks [62].*

The shared memory is visible among threads within the same thread-block, and the global memory can be accessed by all threads [62].

Listing 2.4 displays how a CUDA kernel is invoked by its host. The CUDA kernel creates a N×N *identity matrix* by mapping each index of the matrix to a thread, which sets their values. The keyword `__global__` defines a function as a CUDA kernel, and the grid of thread-blocks executing the CUDA kernel is specified by the `<<<...>>>` grid configuration syntax [62]. In Listing 2.4 a two-dimensional grid containing 32×32 thread-blocks is configured to compute the identity matrix.

CUDA kernels are non-blocking, and the host continues executing its program after the invocation. The `cudaDeviceSynchronize()` function can be used to make the host wait until the CUDA kernel has finished its execution [62].

For more details about CUDA C, see the programming guide [62].

2.4.2 Tesla's Host Setup

In our experiments, the Tesla's host machine had two Intel Xeon E5520 running Arch Linux, kernel 3.10.10-1ARCH (x86_64), together with CUDA toolkit 5.0 installed.

2.5 SB-PRAM

SB-PRAM was developed during a research project in the nineties at the University of Saarbrücken and is, as REPLICA, an emulated shared memory architecture, which realizes the CRCW PRAM model [49, 50]. SB-PRAM does however follow the slightly stronger *priority* CRCW PRAM model mentioned in Section 1.3. The thread with the highest ID will win a concurrent write conflict, and successfully write its value [49]. The last SB-PRAM prototype machine had 64 cacheless multi-threaded processors, with 32 hardware threads each, to hide

memory access latency [49]. The shared memory is emulated by a butterfly interconnection network which supports concurrent read, writes and multiprefix operations [49]. The butterfly construction of the network also makes the memory access time uniform for all processors [49]. The address space is hashed in order to avoid congestions in the interconnection network [49]. The SB-PRAM was realized on separate chips, FPGAs and boards [49].

The processors are clocked at 8 MHz, and the interconnection network at 32 MHz [49]. The clock rate for the processors could however be higher if the number of hardware threads, which hide the memory access latency, is increased. More about this in Section 3.1.

2.5.1 Fork

Fork [52] is a high-level PRAM language which is based on ANSI C, and it has been implemented for SB-PRAM [49, 50]. It uses the SPMD style, and as for REPLICA baseline there are no software threads used [49]. Fork has an elegant way of handling thread groups through control flow constructs, and supports for *synchronous* and *asynchronous* execution mode [49].

During execution in synchronous mode all threads within the same group are always executing the same instruction within the same PRAM step [49]. This is called *strictly synchronous* execution, and means that all threads within the same group have to follow the same control flow path, and that their program counters stay equal for each PRAM step [49]. If threads within the same synchronous executing group take different branches due to a conditional statement, the group automatically splits into new groups [49]. Within the new groups threads are executing synchronously, but not among separate groups. When all the new groups have finished their branch, the new groups are merged back to the parent group [49]. A section of code can be declared as synchronous e.g. by the `start` statement [49]:

```
start {/* Synchronous mode */}
```

In asynchronous mode threads within the same group are not necessarily executing the same instruction, or control flow [49]. If threads within the same group take different branches no new groups are created [49]. The asynchronous mode can therefore be used to reduce overhead of group managing [49]. The term *asynchronous* can be misleading since all threads still execute instructions (PRAM steps) synchronously, just like the PRAM mode of REPLICA [49]. A section of code can be declared as asynchronous e.g. by the `farm` statement [49]:

```
farm {/* Asynchronous mode */}
```

After each `farm` statement's code section follows an implicit barrier for the threads [49]. The `main()` function is asynchronous by default [49]. Threads within the same group that execute asynchronously can be synchronized explicitly by the `barrier` statement [49]. To avoid deadlocks, each thread within the same group has to execute the same number of `barriers` before leaving an asynchronous region [49].

A section of code that should be executed by a single thread can be declared by the `seq` statement [49]:

```
seq {/* Sequential code */}
```

The section is executed in asynchronous mode by any thread in a group [49]. The `seq` statement should only be used inside *synchronous* or *straight* sections [49].

Functions can be declared to be executed in synchronous or asynchronous mode using the qualifier `sync` or `async` [49]:

```
sync int foo(){return 0;}
async int bar(){return 1;}
```

Synchronize function should be called from synchronous regions, and asynchronous functions from asynchronous regions [49]. Asynchronous functions can be called from synchronous regions using the `start` or `join`[2] statement, and synchronous functions from asynchronous regions using the `farm` or `seq` statement [49]:

```
farm bar();
start foo();
```

There is also a qualifier named `straight` which makes it possible to declare functions which can be called from both synchronous and asynchronous regions [49]. Since both regions should be able to call such a function, it must not contain any statements that makes the control flow of the group to diverge [49]. Therefore are control flow statements, such as `if`, `for` and `switch`, only allowed inside a `straight` function if all threads within the same group will follow the same control flow path [49]. By declare a function as `straight`, the programmer does not need to duplicate the function in order to make it executable for both synchronous and asynchronous mode [49]. The `start` or `join` statement should be used when calling synchronous functions from a function declared as `straight`, and the `farm` or `seq` statement when calling asynchronous functions [49].

Variables can be declared as either shared or private using the qualifier `sh` or `pr` [49]:

```
sh int a = 1;
pr float b = 2.0;
```

Variables without qualifier are by default private [49]. Shared variables that are declared globally are shared by all threads, and shared variables that are declared locally in a code section or as parameters of a function are only shared within the groups that executes that section or function [49].

There exist some built-in variables, macros and functions in Fork. The built-in variables are displayed and described in Table 2.6. As mentioned earlier, SB-PRAM has support for multiprefix operations which can by accessed through the intrinsic functions `mpadd()`, `mpmax()`, `mpor()` or `mpand()` [49]. The functions correspond to multiprefix *add*, *max*, *or* and *and*.

[2] The `join` statement is out of scope for the thesis. For more details see [49].

Variable name	Description
__STARTED_PROCS__	Total number of threads.
__PROC_NR__	The thread's absolute ID number.
@	The thread's current group ID number.
$	The thread's absolute ID number, but can also be set to any desired value by the programmer.
$$	The thread's current ID number within its group.
#	The number of threads within the thread's current group.

Table 2.6: *Built-in variables in Fork [49].*

Besides the implicit group splits due to control flow statements inside a synchronous region, there is a statement which can split a group into any desired number of new groups [49]. The statement is named `fork()` and has given the name of the language [49]:

```
fork(e1; @=e2; $=e3) {/* ... */}
```

Expressions e_1,e_2 and e_3 define the number of new groups that should be created, the new group ID for the thread, and the value for the $ variable respectively [49]. The new groups are indexed $0, ..., e_1 - 1$, and if no thread evaluates e_2 to a group's ID, then the group will be empty [49]. Threads that evaluate e_2 to a value outside the range $0, ..., e_1 - 1$ will skip the code section of the `fork` statement [49]. The assignment of $ is optional and can be skipped. After all groups have executed the `fork` statement and its code section, the groups are merged back to the parent group [49].

The Fork program in Listing 2.5 executed by the simulator for a machine with 4 hardware threads will produce the following output:

```
#0000# Start:  $$=0, @=0, #=4
#0001# Start:  $$=1, @=0, #=4
#0002# Start:  $$=2, @=0, #=4
#0003# Start:  $$=3, @=0, #=4

#0000# Branch 1:  $$=0, @=0, #=2
#0001# Branch 1:  $$=1, @=0, #=2
#0002# Branch 2:  $$=0, @=0, #=1
#0003# Branch 3:  $$=0, @=1, #=1
```

The `pprintf()` function is a variant of `printf()`, which can be called by multiple number of threads simultaneously, without having the different outputs mixed up when printing to the screen [49]. Figure 2.9 shows a tree, which il-

lustrates the groups that will be created during the execution of the program.

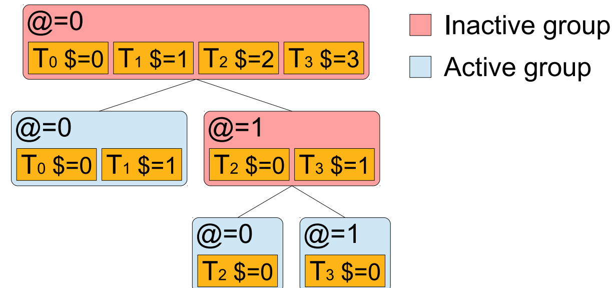


Figure 2.9: *A tree illustration of how the groups are created during execution of a Fork program [49].*

```
1  #include <fork.h>
2  #include <io.h>
3
4  void main(void)
5  {
6      pprintf("$$=%d, @=%d, #=%d\n", $$, @, #);
7      start {
8          seq printf("\n");
9          if($$ < 2)
10             farm pprintf("Branch 3: $$=%d, @=%d, #=%d\n", $$, @, #);
11         else
12         {
13             if($$ < 1)
14                 farm pprintf("Branch 1: $$=%d, @=%d, #=%d\n", $$, @, #);
15             else
16                 farm pprintf("Branch 2: $$=%d, @=%d, #=%d\n", $$, @, #);
17         }
18     }
19 }
```

Listing 2.5: *Example program in Fork.*

Programs written in Fork can be compiled with the *fcc* [50] compiler. fcc (version 2.0) has been used for compiling the SB-PRAM benchmarks.

For more details and features of Fork and its libraries, see [49].

2.5.2 pramsim

The *pramsim* (version 2.0) [50] simulator has been used to simulate the SB-PRAM benchmarks. When simulating, the number of processors and threads can be configured [49]. pramsim simulates at the instruction level of the processors, and does not simulate the interconnection network [49]. Previous work [66] has however shown that the wallclock execution times on the physical hardware and the simulator differ with less than 1.37 % in execution time for all the observed tests.

3
The Benchmark Suite

This chapter describes the benchmark suite, its benchmarks and their kernels, and how measuring is performed. The suite consists of five benchmarks, with at least one kernel for each benchmark. See Table 3.1 for an overview of the benchmark suite.

Benchmark name	**Dwarf**	**Memory access**
PS	-	Regular
DeMM	Dense Linear Algebra	Regular
SpMV	Sparse Linear Algebra	Irregular
QS	Graph Traversal (Sorting)	Irregular
BFS	Graph Traversal	Irregular

Table 3.1: *An overview of the benchmark suite: Prefix Sum (PS), Dense Matrix-Matrix multiplication (DeMM), Sparse Matrix-Vector multiplication (SpMV), Quicksort (QS) and Breadth First Search (BFS)*

3.1 Measuring

This section describes how the different architectures performance are measured.

Measuring on REPLICA

The amount of executed cycles on REPLICA was measured using the `_start_timer` and `_end_timer` macros around the benchmark's kernel. The macros expand to an inline assembler statement containing a trap instruction, which the simulator

recognizes and automatically measures the elapsed clock cycles between these two trap instructions.

Measuring on XMT

The manual for the XMT toolchain [21] does not describe any way to define timers for specific parts of the code. Therefore the total number of cycles is measured using the flag `-cycle` when starting xmtsim. To avoid measuring initializing of data, the input data is initialized at compile time, and output is received by the flag `-bindump` when simulating with xmtsim. However, there does exist a `readtimer64()` macro in previous work [31] by the XMT project. But due to lack of documentation, this macro was only used to check that the `-cycle` flag produced about the same result. The difference between these was small. Sometimes the `readtime64()` macro gave less performance, probably due to the `printf()` function that prints out the elapsed cycles.

Measuring on Xeon

The execution times for Xeon are measured by calculating the difference between a time stamp before the kernel is executed, and one after. Each kernel is executed and measured one thousand times from which an average is calculated. This reduces the overhead and nondeterministic effects of the operating system. An alternative way could have been to measure one thousand executions and from that calculate the average. This solution would add less overhead to the measured time since the time stamp functions only had to be called twice. The problem with this solution is that if the kernel destroys the input data it has to be reset before the next execution. For some kernels, only the size of the input data affects execution time, while the content of the data is irrelevant (e.g., matrix-matrix multiplication). But this is not true for all kernels. The execution time for a sorting algorithm normally depends on the content of the input and its size. However to ensure that a correct result is computed, and to measure all kernels in the same manner, the alternative solution is not used. The chip is warmed up by doing some dummy work before any measuring starts, to make sure it runs on its maximum clock frequency.

The time stamps are produced using the `int clock_gettime(clockid_t clk_id, struct timespec *tp)` function from the GNU C library (glibc) [9]. `CLOCK_MONOTONIC` has been used as clock. The clock resolution is according to `int clock_getres(clockid_t clk_id, struct timespec *res)` 1 ns. However, the average time between two time stamps called directly after each other one thousand times without warming up the chip is 58.186 ns. When the chip is warmed up this overhead decreases. See A.1 for the implementation of timer functions.

After the kernel has been launched, some data will remain in Xeon's cache. Because of this side effect of how the measuring is done, the remaining launches are likely to execute faster than the first one. This should be remembered when looking on the results.

Measuring on Tesla

For Tesla, the time spent in the kernel and the time used for copying data between the device and host (both ways) are measured separately. This makes it possible to compare the computation time, transmit time, and their ratio to the total time. When the input data consists of more than one array, the asynchronous function `cudaMemcpyAsync()` has been used. After the asynchronous function calls, a single sync makes sure that all transmissions are finished. If transmit time is much greater than the computation there is less use to optimize the kernel further since it will not affect the overall time significantly. The chip is warmed up before measuring by executing the kernel once, a so-called flying start. Benchmarks for Tesla uses the same timer functions as used for Xeon. Also both computation and transmit code blocks are measured one thousand times, from which an average is calculated in order to reduce the nondeterministic effects. The resolution of the `CLOCK_MONOTONIC` clock on the host which the Tesla chip is installed on, is 1 ns, and the average overhead for the clock is 43.584 ns.

As Xeon, Tesla might run faster after the first launch.

Measuring on SB-PRAM

`getct()` is a function in SB-PRAM's Fork library which returns a time stamp that can be used in a similar manner as done for Xeon and Tesla. However, there is a difference. The time stamp is not based on the system clock, but on how many clock cycles each thread has consumed. To get the total number of clock cycles between two time stamps, the elapsed cycles between them have to be multiplied with the number of threads per processor ($T_{processor}$). SB-PRAM has 32-threaded processors and the elapsed cycles should therefore be multiplied with 32. But if SB-PRAM should be clocked with a frequency comparable to REPLICA's, it would need more threads to hide the increased frequency gap between memory and processor. The final SB-PRAM machine had 64 processors with 32 threads each. But to make it comparable with REPLICA, we measure as if it had 4 processors with 512 threads. It should be a realistic assumption since the SB-PRAM's simulator does not simulate the network in any case. Now the simulated SB-PRAM has as many processors and threads as REPLICA-T5-4-512, REPLICA-T7-4-512 and REPLICA-T11-4-512 (see Table 2.1).

3.2 Prefix Sum (PS)

The prefix sum does not belong to any dwarf class, but is a fundamental building block in many PRAM algorithms [49, 77]. There exist two forms of prefix sums, *inclusive* and *exclusive* [77]. The inclusive version takes an array of size n, and computes for each entry the sum of all elements with lower or equal index to its

own [77].

$$\text{Input: } [a_0,\ a_1,\ a_2, \dots a_{n-1}]$$

$$\text{Inclusive output: } [a_0, \sum_{i=0}^{1} a_i, \sum_{i=0}^{2} a_i, \dots \sum_{i=0}^{n-1} a_i]$$

For exclusive prefix sum the output is shifted to the right, so that the last element is excluded, and the first index will always be set to zero.

$$\text{Exclusive output: } [0,\ a_0, \sum_{i=0}^{1} a_i, \dots \sum_{i=0}^{n-2} a_i]$$

A native sequential implementation of the inclusive prefix sum can be seen in Listing 3.1.

```
1 // Input array
2 int a[] = {...};
3
4 for(i=1; i<N; ++i)
5     a[i] += a[i-1];
```

Listing 3.1: *Native implementation of the PS kernel.*

It should be pointed out that the *prefix sums problem* is a general problem where the "summing"-operation not necessarily has to be the addition, but could be any binary associative operation [49, 77].

3.2.1 PS for REPLICA

Since prefix sum is very common in PRAM algorithms, REPLICA has hardware support for this type of operation, which works similarly to an atomic fetch-and-add operation. It is performed by calling three multiprefix assembler instructions in a series. REPLICA baseline has the built-in macro `_prefix(`p, o, m, c`)`, which handles the inline assembler. The difference compared to a normal fetch-and-add operation is that if the operation is executed by multiple threads simultaneously (within the same PRAM step), REPLICA will order the operations by their thread ID. It is therefore called *ordered multiprefix operation*.

The parameter `o` is the operation that will be applied on the target of address `m` with the operand `c`. `p` is a variable that will receive the value of address `m` after (but within the same PRAM step) all threads with lower ID than its own have performed their operations.

So if the (exclusive) prefix sum of array `a_[n]` should be computed in-place by n threads, the threads can be mapped based on their ID to a value in the array, which they shall compute the result of, simultaneously. This can be done by the following multiprefix operation:

```
_prefixsum(a_[_thread_id], ADD, &sum_, a_[_thread_id])
```

Where sum_ is a shared variable that is initialized to zero.

To support problems that are greater than the number of threads, all that has to be done is to insert a loop:

```
for(i=_thread_id; i<N; i+=_number_of_threads)
    _prefixsum(a_[i], ADD, &sum_, a_[i]);
```

Since the shared variable *sum_* is not set back to zero after each iteration, the next iteration will continue where the previous left off.

See Listing A.2 for the benchmark code. The built-in variable _number_of_threads is declared as global, and will therefore not be put into a register, which decreases the performance since it is used frequently [69]. Therefore it was copied into a local variable declared as temp, see Listing A.2. By doing this, we got a speedup between 1.2 and 1.4.

If the inclusive prefix sum is desired, it can be obtained by the sequence:

$$a_[1],\ a_[2],\ a_[3], \dots a_[n-1],\ sum_$$

_prefix(p, o, m, c) is a general macro and works not only for sums, but also for SUB (for subtraction), AND, OR, MAX and MIN. The _aprefix() macro works similarly to _prefix(), but does not ensure the ordering by ID as _prefix, and is therefore an *arbitrary multiprefix operation*. Because _aprefix() does not have to care about the execution order among the threads it only need perform two assembler instructions. So if the order is irrelevant this kernel could be executed even faster using _aprefix() instead.

3.2.2 PS for Xeon

An inclusive prefix sum kernel was implemented using OpenMP. It first divides the input into T_{total} chunks, and spawns a thread for each chunk (including the master thread). The threads are mapped to a chunk based on its thread ID. Then the threads compute the prefix sum of its chunk, and copy the last element into a shared array at index T_{ID}. The shared array's prefix sum is then computed by a single thread. Now each thread can finish its work by coping the value from index $T_{ID}-1$ in the shared array, and adding it to each element in its chunk. Note that the master thread's chunk already contains the correct result. Figure 3.1 illustrates the implementation using 4 threads. The mentioned method is based on the technical report: *Prefix sums and their applications* [24].

The prefix sums of the chunks and of the shared array are computed by a similar method as the native one in Listing 3.1. However, the native version makes an unnecessary load from a[i-1]. It is more efficient to store the current sum in a temporary variable. See Listing A.3 for the complete kernel.

The GNU C++ library has a parallel version of the *std::partial_sum()* function which was compared against the implemented version:

```
__gnu_parallel::partial_sum(a , a+N, a);
```

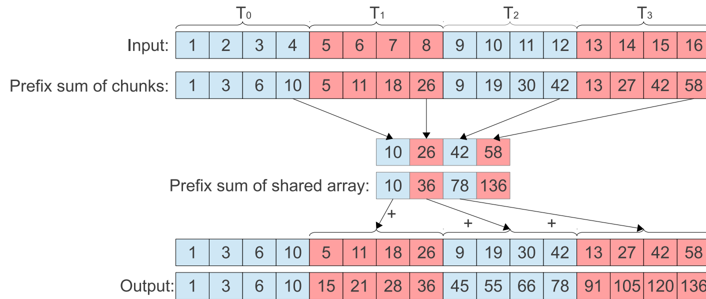


Figure 3.1: *Illustration of the prefix sum implementation [60].*

a is an array with N elements. The first argument of *partial_sum()* is the input array, and the third is the output array.

Both GNU's and the implemented version showed best performance when using 6 threads, which means that hyper-threading should not be used. The implemented version was almost twice as fast as GNU's, and therefore chosen to be Xeon's PS kernel.

3.2.3 PS for XMT

The `psm()` function can not be used to compute the prefix sum since XMT's threads do not execute synchronously. Instead we have used the function `prefix_sum_int()` from earlier work by the XMT project [31] as kernel, see Listing 3.2.

```
1 prefix_sum_int(a, a, N);
```

Listing 3.2: *The PS kernel for XMT.*

a is an array with N elements. The first argument is the input, and the second is the output.

3.2.4 PS for Tesla

Tesla's prefix sum benchmark uses the `inclusive_scan()` function from the *Thrust* library as kernel. *Thrust* comes with the CUDA toolkit and is a C++ template library, which has a similar interface as the C++ STL (*standard template library*) [64]. It is called to compute the prefix sum in-place, see Listing 3.3

```
1 thrust::inclusive_scan(a, a+N, a);
2 cudaError_t err = cudaThreadSynchronize();
```

Listing 3.3: *The PS kernel for Tesla.*

a is a *Thrust device pointer* for the input data of size N. A *Thrust device pointer* (`thrust::device_ptr<T>`) is a data structure template that stores a pointer to an object allocated at the device memory [64]. It helps functions to distinguish between device memory pointers and host memory pointers.

`inclusive_scan()` takes the beginning of input, end of input and beginning of output as its first, second and third argument.

3.2.5 PS for SB-PRAM

The prefix sum can easily be computed on the SB-PRAM in the same way as on REPLICA, using only a loop and the multiprefix operation `mpadd()`, see Listing 3.4.

```
1  sh int sum = 0;
2  pr int i;
3  for (i=$; i<N; i+=#)
4  {
5      a[i] = mpadd(&sum, a[i]);
6  }
```

Listing 3.4: *The PS kernel for SB-PRAM.*

`a` is a shared array of size `N` that stores the input data. `sum` is a shared integer which works as an accumulator for the prefix sum. The symbols `$` and `#` holds the thread ID and group size respectively. The above kernel is executed in a synchronous region, to ensure the iterations to be executed in order.

Since SB-PRAM uses priority among threads to solve writing conflict there exist no arbitrary ordered multiprefix operations.

3.3 Dense Matrix-Matrix Multiplication (DeMM)

Dense matrix-matrix multiplication (DeMM) is a fundamental *linear algebra* dwarf. The benchmark kernels assume that the result matrix has been set to *zero* for simplicity. So the correct description of the problem is $Y = AB + Y$, where Y , A and B are square matrices. Listing 3.5 shows the conventional implementation of the matrix-matrix multiplication, on which the implementations for the different architectures are based.

```
1  // Matrices
2  int A[N][N] = {...};
3  int B[N][N] = {...};
4  int Y[N][N] = {...};
5
6  // Y = AB + Y
7  for(i=0; i<N; ++i)
8      for(j=0; j<N; ++j)
9          for(k=0; k<N; ++k)
10             Y[i][j] += A[i][k] * B[k][j];
```

Listing 3.5: *The conventional implementation of the DeMM kernel [41].*

There exist however matrix-matrix multiplication algorithms with lower asymptotic complexity, such as *Strassen's algorithm* [41]. But for simplicity the conventional algorithm has been chosen. Table 3.2 shows the dimensions of input matrices for the DeMM kernels. It also shows the number of elements and integer

multiplication operations needed to perform the matrix-matrix multiplication.

Dimensions	Elements	Multiplication operations
128×128	16384	2097152
256×256	65536	16777216
384×384	147456	56623104
512×512	262144	134217728
1024×1024	1048576	1073741824

Table 3.2: *Input matrices for the DeMM kernel.*

3.3.1 DeMM for REPLICA

The first thing to do when converting the algorithm to REPLICA, is to divide the problem into at least as many subtasks as there are threads. It is not enough to parallelize the outermost loop since it only consists of up to 1024 iterations for our input. For the largest configuration of REPLICA we need 32 768 subtasks, which is more than the number of elements within the smallest input matrices. There are however enough multiplication operations for every thread.

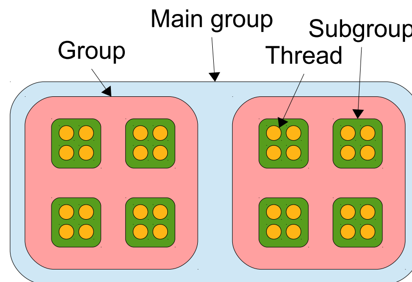


Figure 3.2: *An illustration of the group hierarchy.*

To map the multiplication operations to threads we expand REPLICA's single level group concept into 3 levels (one level for each loop) using subgroups. The group hierarchy is illustrated in Figure 3.2. Each loop can now be mapped to a group level:

```
for(i=group_id; i<N; i+=main_group_size)
    for(j=subgroup_id; j<N; j+=group_size)
        for(k=_thread_id; k<N; k+=subgroup_size)
            multi(ADD, &y_[i][j], a_[i][k]*b_[k][j]);
```

If we wish to compute a multiplication between two matrices of dimension $N \times N$, and create N groups containing N subgroups, with N threads, each thread would only need to compute a single integer multiplication:

```
i=group_id;
j=subgroup_id;
k=_thread_id;
multi(ADD, &y_[i][j], a_[i][k]*b_[k][j]);
```

Unfortunately, there are not N^3 threads available, so we have to decide how to divide the threads into groups as efficient as possible. Since multioperations becomes more efficient as more threads cooperates, we want as many threads as possible to cooperate. With cooperation we mean that they write to the same address. This is done by letting the groups that accumulate variable `k`, which is subgroups, be as large as the number of rows. To balance work among the threads, it is important that all subgroups are of equal size, and since all REPLICA configurations have a power of two number of threads, the subgroups have to contain a power of two number of threads as well. So if the number of rows not is a power of two, it is rounded down to the next power of two number.

When the size of the subgroups is decided, the subgroups are put together into groups. The optimal group size is equal to the number of rows, but all groups have to be of equal size, which means that they have to be a power of two. When the size of the groups and their subgroups are determined, we can compute how many groups there are in the main group.

The next step is to reorder the loops so that the loop with the most iterations is the innermost one. Since the i-loop is mapped to the group level with the smallest accumulation variable (`main_group_size`), it should be the innermost loop. The k-loop is mapped to the group level with the largest accumulator variable (`subgroup_size`), and should therefore be the outermost loop. By reordering the loops this way, the conditional checks are reduced. This can be proved by the mathematical expression, which describes the total number of conditional checks for the three loops:

$$N_o + N_o N_m + N_o N_m N_i \tag{3.1}$$

Where N_o, N_m and N_i are the numbers of iterations for the outermost, middle and innermost loop respectively. The innermost loop does only affect one of the three terms, and has therefore less effect on the total number of conditional checks, compared to the outermost and middle loop.

The innermost loop's basic block[1] consists only of 3 VLIWs, which limits the compiler to do ILP and VILP optimizations. It is possible to create a larger basic block by unrolling the loops. If the k- and i-loop are unrolled by four, the innermost basic block grows to 25 VLIWs which computes $4 \cdot 4 = 16$ integer multiplications. We can then calculate the number of needed instructions per integer multiplication and compare the different versions. The original version needs $\frac{3}{1} = 3$ VLIWs per integer multiplication, compared to $\frac{25}{16} \approx 1.56$ VLIWs for the unrolled version. This is a speedup of 1.92 for the unrolled version, and since the unrolled version does not need to iterate as many times as the original version, it also reduces

[1] A *basic block* is a block of code which does not contain any branches except for the last instruction, nor any entry points except for the first instruction.

some overhead of the loops. But some extra checks that take care of the last iterations have to be added, if the number of rows is not a multiple of four. Also, the comparison is not entirely fair because the number of clock cycles can vary for different VLIWs.

By testing, it was observed that the kernel showed better performance when storing the matrix in an one-dimensional array instead of in a two-dimensional one. When using an one-dimensional array, the row offsets have to be computed explicit. By placing the j-loop as the innermost, the row offset (`i*N` and `k*N`) can be reused, and since one of the operands only depends on `i` and `k`, it can be reused as well. This is shown in the benchmark source code, see Listing A.4.

If the number of rows is not a multiple of the group size, then the groups and threads with lower ID will have more work than these with higher ID. To avoid this we let the threads shift their group and thread ID after each loop, see Listing A.4.

Some issues were observed when implementing this kernel. The compilation aborts during register allocation due to an assertion failure. This could be solved by marking the variable causing the abortion as `volatile`. Unfortunately, this variable is used in the innermost loop, which most likely affects the performance negatively.

Since REPLICA has hardware support for computing the integer binary logarithm, we wanted to use it for calculating the group sizes. Here is an example:

```
unsigned max = _number_of_threads<N ? _number_of_threads:N;
unsigned exp;
asm("LOGD0 %1   WB%0 A0": "=r"(exp): "r"(max) :);
unsigned sub_group_size = 1<<exp;
```

For unknown reasons this technique produced the wrong results. It was solved by implementing a much slower software solution, using a loop:

```
unsigned max = _number_of_threads<N ? _number_of_threads:N;
unsigned sub_group_size;
for(subgroup_size=1; subgroup_size<=max; subgroup_size<<=1)
    ;
sub_group_size >>= 1;
```

The software solution is of course much slower than using the assembler instruction, especially for large matrices. This however only affects the initial part of the kernel, and is just a small fraction of the total amount of work.

3.3.2 DeMM for Xeon

The main issue when implementing a matrix-matrix multiplication for Xeon is cache locality. For the conventional implementation in the innermost loop, see Listing 3.5, matrix `B` is accessed column-wise. The cache will store the elements in `B` as they are requested, and since the size of an integer is not enough to fill an entire cache line, the cache line will be filled with extra elements with higher

address next to the requested element. The extra elements are marked with hashing in Figure 3.3a. If the level 1 cache is large enough, the extra cached elements can be used in the next round for the innermost loop. This is unfortunately not the case for larger matrices. The extra elements will then likely be overwritten by other elements before the next round, which leads to cache misses and reduced performance [30].

A possible solution is to transpose `B` before computation, so that it can be accessed row-wise, but to transpose `B` adds extra work. A better solution is to use *loop tiling* in order to increase cache locality [30]. Loop tiling splits the matrices into submatrices, so that a submatrix row fits into a single cache line [30]. Listing 3.6 shows the conventional matrix-matrix multiplication from Listing 3.5 with loop tiling using `DIM`×`DIM` submatrices. It computes the contribution of each submatrix one at a time, so that the extra cached elements are accessed before being overwritten. Figure 3.3b shows how the kk-loop in Listing 3.6 iterates over the submatrices in `A` and `B`, in order to compute the result submatrix in *Y*. It can be compared with Figure 3.3a which shows how the k-loop in Listing 3.5 iterates over `A` and `B` while computing a single element in `Y`.

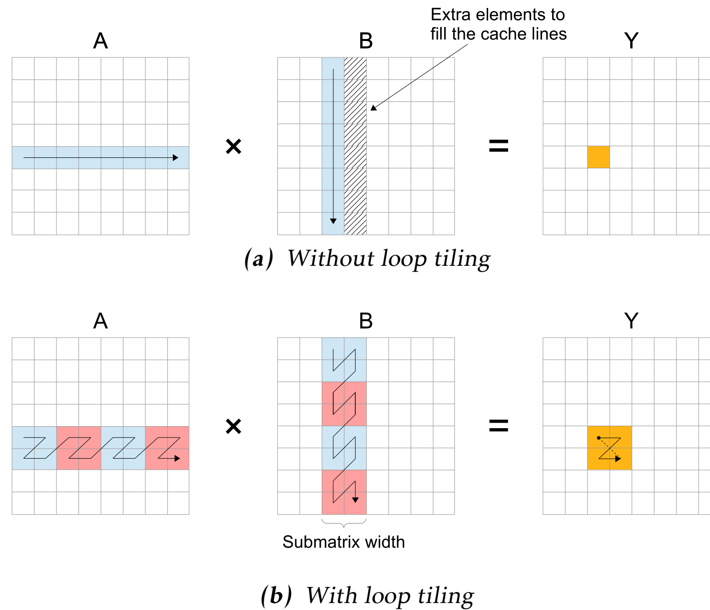
```
1 #define DIM (chache-line-size/size-of-int)
2 for(ii=0; ii<N; ii+=DIM)
3     for(jj=0; jj<N; jj+=DIM)
4         for(kk=0; kk<N; kk+=DIM)
5             for(i=ii; i<ii+DIM; ++i)
6                 for(j=jj; j<jj+DIM; ++j)
7                     for(k=kk; k<kk+DIM; ++k)
8                         Y[i][j] += A[i][k] * B[k][j];
```

Listing 3.6: *The conventional matrix-matrix multiplication using loop tiling.*

The cache line size of Xeon's level 1 cache is 64 bytes, and can therefore hold 16 integers. This means that 16×16 submatrices should be used. Unlike REPLICA, Xeon's DeMM kernel does assume that the number of rows for the matrices is a multiple of 16.

The matrix-matrix multiplication in Listing 3.6 can be executed in parallel by parallelizing the outermost loop using OpenMP, see Listing 3.7. The threads should be `static`-scheduled, since each loop iteration consists of an equal amount of work. The kernel will be measured using 6 threads, because tests showed that the Xeon then performed best. This means that hyper-threading should not be exploited. However, a parallelized conventional version did execute slightly faster using 12 threads (hyper-threading), compared to using 6 threads.

Further optimizations can be found in the Xeon final DeMM kernel in Listing 3.7. The second innermost loop (the j-loop) has been unrolled once, and row offset calculations are reused as in REPLICA's DeMM kernel. The innermost loop uses two temporary variables (`sum0` and `sum1`) which store the accumulating results. These results are stored in matrix `y` after each round of the innermost loop (the k-loop).



(a) Without loop tiling

(b) With loop tiling

Figure 3.3: *A comparison of how an entry in the result matrix is computed with, and without loop tiling. For simplicity the cache line size in (a) and (b) is 8 bytes, and therefore 2×2 submatrices are used for tiling.*

```
#define DIM 16
#pragma omp parallel default(none) shared(a, b, y)
{
    unsigned i, j, k, ii, kk, jj;
    #pragma omp for schedule(static)
    for(ii=0; ii < N; ii+=DIM)
    {
        for(jj=0; jj < N; jj+=DIM)
        {
            for(kk=0; kk<N; kk+=DIM)
            {
                for(i=ii; i < ii+DIM; ++i)
                {
                    unsigned iN = i*N;
                    for(j=jj; j < jj+DIM; j+=2)
                    {
                        int sum0 = y[iN+j];
                        int sum1 = y[iN+j+1];
                        for(k=kk; k<kk+DIM; ++k)
                        {
                            sum0 += a[iN+k]*b[k*N+j];
                            sum1 += a[iN+k]*b[k*N+j+1];
                        }
                        y[iN+j] = sum0;
                        y[iN+j+1] = sum1;
                    }
                }
            }
        }
    }
}
```

Listing 3.7: *The DeMM kernel for Xeon.*

Xeon's SIMD-instructions only support floating points, and therefore these have not been used. But since matrix-matrix multiplication normally is computed using floating point, an alternative kernel called DeMM-BLAS will also be compared against REPLICA DeMM. DeMM-BLAS uses the OpenBLAS' `cblas_sgemm()` function to compute single precision floating point matrix-matrix multiplication. OpenBLAS is a highly tuned and hand-optimized third-party BLAS library, based on *GotoBLAS2* [78]. It was compiled to target *NEHALEM* which is the name of our Xeon's micro-architecture [78]. OpenBLAS detects Xeon's hyper-threading, and avoids to schedule threads on the same processor, which implies that only 6 threads will be used [78]. The DeMM-BLAS kernel is displayed in Listing 3.8, where `CblasNoTrans` tells that matrix `a` and `b` are not transposed. `CblasRowMajor` tells that the matrices are stored in row major.

```
1 // y = a*b + y
2 cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
3             N, N, N, 1.0, a, N, b, N, 0.0, y, N);
```

Listing 3.8: *The DeMM-BLAS kernel for Xeon.*

3.3.3 DeMM for XMT

The conventional matrix-matrix multiplication in Listing 3.5 can easily be parallelized by replacing the outermost loop (the i-loop) with a `spawn(0, N)` statement, and setting variable `i` to `$`. But since XMT has more threads than some input matrices have rows, it will not make use of all its threads. A better solution can be obtained using *loop collapsing*. Loop collapsing takes a set of nested loops and merges them into a single loop [49]. In our case, we want to take the two outermost loops in Listing 3.5, and merge them into a single loop, which can replaced with a `spawn(0, N*N)` statement:

```
for(ij=0; i<N*N; ++ij)
```

The original loop variables then have to be reconstructed in order to be used:

```
i = ij / N;
j = ij % N;
```

Even if XMT is designed to have low overhead for fine grained parallelism, spawning `N*N` threads, which is one thread for each element in the result matrix, will have a noticeable overhead. Also, because each iteration of the innermost loop contains the same amount of work, there is no need for the dynamic hardware scheduling that XMT provides. Therefore, we only want to spawn one thread per TCU, and map them to equally large chunks of the merged outermost loop. This technique is used in earlier work by the XMT project [59].

The complete DeMM kernel for XMT is displayed in Listing 3.9, which also has some further optimizations. The innermost loop has been unrolled and uses the temporary variable `sum` to hold the accumulated result, which is stored inside matrix `y` after each round. The row offsets are also reused by the variables `rN` and `kN`.

```
#define NB_THREADS 1024
const int NN = N*N;
spawn(0, NB_THREADS)
{
    int start_chunk = NN*$/NB_THREADS;
    int end_chunk= NN*($+1)/NB_THREADS;
    end_chunk = end_chunk<NN ? end_chunk : NN;
    int ij;
    for (ij = start_chunk; ij<end_chunk; ++ij)
    {
        int sum = y[ij];
        int i = ij/N;
        int j = ij%N;
        int rN = i*N;
        int kN = 0;
        int k;
        for(k=0; k<N-1; k+=2, kN+=N+N)
        {
            sum += a[rN+k]*b[kN+j] + a[rN+k+1]*b[kN+N+j];
        }
        y[ij] = k<N ? sum+a[rN+k]*b[kN+j] : sum;
    }
}
```

Listing 3.9: *The DeMM kernel for XMT.*

When implementing the DeMM kernel for XMT some problems due to register spilling was observed. For example, when trying to increase the unrolling degree of the innermost loop to four, the compiler aborted due to register spilling. This is because XMT currently cannot deal with register spills inside a `spawn` block [21].

3.3.4 DeMM for Tesla

The DeMM kernel for Tesla is based on the matrix-matrix multiplication example in the *CUDA C Programming Guide* [62]. The host set up a $(N/32)\times(N/32)$ grid of 32×32 thread-blocks, see Listing 3.10. The matrices are divided into 32×32 submatrix. Each thread-block is mapped to a submatrices in `y` for whose result it is responsible to compute. A thread-block is only responsible for one submatrix result in `y`. The result for a submatrix is obtained by letting a thread-block perform the needed submatrix-submatrix multiplications. The result from each submatrix-submatrix multiplication is accumulated and stored in the thread-block's result submatrix.

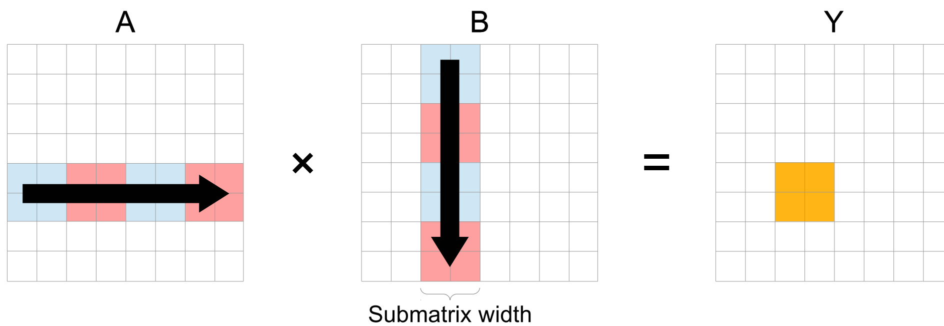
```
dim3 dimBlock(32, 32);
dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);
DeMM<<<dimGrid, dimBlock>>>(a, b, y, N);
cudaError_t err = cudaThreadSynchronize();
```

Listing 3.10: *The DeMM kernel for Tesla.*

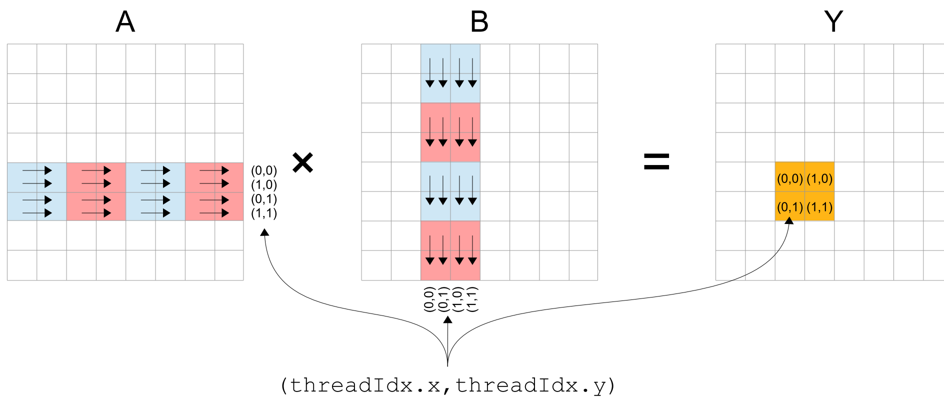
Since the thread-blocks and submatrices have the same dimensions, each thread can be mapped by its local index to compute an element in the thread-block's result submatrix.

To summarize: a thread-block computes the result for a single submatrix in `y`, and a thread computes the result for a single element in `y`. Figure 3.4a shows how the thread-blocks iterates over their submatrices in order to calculate the results for their submatrix in matrix `y`. How the threads iterate within a submatrix is displayed in Figure 3.4b.

Before a thread-block performs a submatrix-submatrix multiplication, it stores the submatrices that are being multiplied inside its shared memory in order to speed up memory accesses.



(a) How a 2×2 thread-block iterates over its submatrices. Indices for the thread-block are `(blockIdx.x,blockIdx.y)=(1,2)`.



(b) How the threads of a 2×2 thread-block iterate within their submatrices.

Figure 3.4: *Figure (a) shows how a thread-block iterates over its submatrices, and (b) how the same thread-block's threads iterate within the submatrices. Before each submatrix-submatrix multiplication is performed, the submatrices of A and B are stored inside the shared memory.*

For simplicity, this kernel assumes that the matrix rows are multiples of 32, due to the submatrix sizes. The thread-block size is limited to 32×32 since $32 \cdot 32 = 1024$, which is the maximum number of threads a thread-block for Tesla M2050

can consist of. We observed that smaller thread-blocks have less performance, and therefore the thread-block size 32×32 was chosen.

Tesla has also a floating point kernel for matrix-matrix multiplication called DeMM-BLAS. The BLAS kernel uses the `cublasSgemm()` function from the CUBLAS library [61]. The CUBLAS library is an optimized implementation of BLAS, provided by the vendor NVIDIA [61]. The kernel is displayed in Listing 3.11, where `CUBLAS_OP_N` tells that matrices `a` and `b` are not transposed. The matrices `a`, `b` and `y` are located at the device memory.

```
1  const float alpha = 1.0;
2  const float beta  = 1.0;
3  cublasHandle_t handle;
4  cublasStatus_t ret = cublasCreate(&handle);
5  // Kernel begins
6  // y = a*b + y
7  ret = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
8                    N, N, N, &alpha, b, N, a, N, &beta, y, N);
9  cudaError_t err = cudaThreadSynchronize();
10 // Kernel ends
```

Listing 3.11: *The DeMM-BLAS kernel for Tesla.*

3.3.5 DeMM for SB-PRAM

The matrix-matrix multiplication kernel for SB-PRAM is very similar to XMT's kernel, see Listing 3.12. It uses loop collapsing as. The main difference is that the innermost loop has been unrolled by a factor of eight. All loops in the kernel are executed asynchronously due to the `farm` statement. A similar implementation is outlined in [49].

```
1  unsigned ij;
2  unsigned NN = N*N;
3  farm for(ij=$; ij<NN; ij+=__STARTED_PROCS__)
4  {
5      const unsigned i=ij/N;
6      const unsigned iN=i*N;
7      const unsigned i=ij%N;
8      int sum=y[ij];
9      unsigned k, kN_j;
10     for(k=0,kN_j=i; k<N-7; k+=8,kN_j+=8*N)
11     {
12         sum += a[iN+k]*b[kN_j]      + a[iN+k+1]*b[kN_j+N]
13           +  a[iN+k+2]*b[kN_j+2*N] + a[iN+k+3]*b[kN_j+3*N]
14           +  a[iN+k+4]*b[kN_j+4*N] + a[iN+k+5]*b[kN_j+5*N]
15           +  a[iN+k+6]*b[kN_j+6*N] + a[iN+k+7]*b[kN_j+7*N];
16     }
17     for(; k<N; ++k,kN_j+=N)
18     {
19         sum += a[iN+k]*b[kN_j];
20     }
21     y[ij] = sum;
22 }
```

Listing 3.12: *The DeMM kernel for SB-PRAM.*

3.4 Sparse Matrix-Vector Multiplication (SpMV)

Sparse matrix-vector multiplication (SpMV) is a *sparse linear algebra* dwarf and is a fundamental part of many popular iterative methods, such as the *conjugate gradient method*. There exists a great number of formats to compress sparse matrices. Some are specialized for matrices with certain patterns, and some for more general matrices. This benchmark use symmetric matrices from *The University of Florida Sparse Matrix Collection* [1], and is mainly using the *compressed sparse row format* to store the matrices inside the kernel. The *coordinate format* is also used to see if this format is better suited for the REPLICA architecture since it is much easier to balanced the computational work among the threads, as will be shown in this chapter.

The used sparse matrices that have been used are displayed in Table 3.3.

Matrix name	Rows	Values	Values per row
Internet	124651	207214	1.7
Lugn2	109460	492564	4.5
ASIC_680ks	682712	2329176	3.4
t2em	921632	4590832	5.0

Table 3.3: *The matrices that have been used for the SpMV kernel [1].*

Coordinate Format

The *coordinate (COO) format*, which also can be called *IJV* or *triplet format*, is probably the simplest of all sparse matrix storage formats. Three arrays store the *value*, *row index* and *column index* of each nonzero value [25]. The required storage when using the COO format depends only on the number of nonzero values. The matrix's shape or value pattern does not affect the required space. This makes it a very general format for sparse matrices. Each scalar operation in a matrix-vector product needs two indirect addressings which makes it a quite inefficient format. It is often used when storing matrices on files, or when converting between two different formats. Sometimes the values are sorted by row or column index. The *matrix market* exchange format for sparse matrices is one example where it is used. Figure 3.5 shows a symmetric matrix stored in the COO format.

Compressed Sparse Row

Compressed sparse row (CSR), also called *compressed row storage* (CRS) [23], is a well-known and commonly used sparse matrix format [25]. It makes no assumption about the sparsity structure of the stored matrix, and like the COO format it stores only nonzero values [23]. A drawback is that every scalar operation in a matrix-vector product needs indirect addressing which makes it a not very time-efficient format [23]. The CSR format is similar to the COO format. The difference is that the array storing the row indices is replaced with a compressed array that points out the start index for each row in the other two arrays. Also the

$$Matrix = \begin{bmatrix} 1 & 0 & 8 & 0 & 0 \\ 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 9 & 0 & 0 \\ 0 & 0 & 3 & 2 & 0 \\ 0 & 0 & 0 & 0 & 5 \end{bmatrix} \longrightarrow \begin{cases} nonzero\ value = [1,8,7,9,3,2,5] \\ row\ index \quad\ \ = [0,0,1,2,3,3,4] \\ column\ index = [0,2,1,2,2,3,4] \end{cases}$$

Figure 3.5: *An example of how a* 5×5 *matrix with 7 nonzero values is stored with the COO format using zero-based indexing [23].*

nonzero values have to be stored row-wise. The new array's length is equal to the matrix's number of rows plus one. Since the start index for a row implicitly is the end of the previous one it is only necessary to store the end of the last row. That is why the "plus one" is needed. The end of the last row will however always be equal to the number of nonzero values, when using zero-based indexing.

In contrast to the COO format, the shape of the matrix affecting the required storage. Matrices with fewer rows require less space than matrices with the same number of nonzero values that have more rows.

A symmetric $n \times n$ matrix with nnz nonzero values is stored using $2nnz + n + 1$ elements instead of n^2 [23].

Figure 3.6 shows how the same matrix as in Figure 3.5 is stored using the CSR format.

$$Matrix = \begin{bmatrix} 1 & 0 & 8 & 0 & 0 \\ 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 9 & 0 & 0 \\ 0 & 0 & 3 & 2 & 0 \\ 0 & 0 & 0 & 0 & 5 \end{bmatrix} \longrightarrow \begin{cases} nonzero\ value = [1,8,7,9,3,2,5] \\ column\ index = [0,2,1,2,2,3,4] \\ row\ start \quad\ \ = [0,2,3,4,6,7] \end{cases}$$

Figure 3.6: *An example of how a* 5×5 *matrix with 7 nonzero values is stored with the CSR format using zero-based indexing [23].*

Listing 3.13 describes a native implementation of a sparse matrix-vector multiplication. The outer loop iterates over the number of rows in the result vector `y`. In the inner loop all nonzero values from one row are multiplied with the column in vector `x` that correspond to that row, and the result is stored in the temporary variable `sum`. Then the temporary variable is stored in the result vector `y` .

The CSR format is almost equivalent to the *Compressed sparse column* (CSC) which is also called *Compressed Column Storage* (CCS), except that the values are stored

column-wise, column indices are replaced with row indices, and the beginning of each column is stored instead of the beginning of each row [23].

Other Sparse Matrix Formats

Some sparse matrix formats take advantage of the sparsity structure, or try to speed up computation by storing it more computation-efficiently. The *Diagonal format* takes advantage of matrices that are, or almost are diagonal [25]. The *ELL-PACK format* on the other hand exploits matrices where each row has nearly the same number of nonzero values [25]. A common way to improve the performance of the CSR format is to use the *block compressed sparse row format*, which stores blocks of values instead of nonzero values [25]. This decreases the amount of indirect addressing, but might store unnecessary zero values. To gain performance it is important to use a good block size, which depends on the nonzero pattern and the hardware [25].

```
1  // Matrix stored in the CSR format
2  int value[NNZ] = {...};
3  unsigned col_index[NNZ] = {...};
4  unsigned row_start[ROWS+1] = {...};
5
6  // Vectors
7  int x[ROWS] = {...};
8  int y[ROWS] = {...};
9
10 // y = Ax
11 for(row=0; row<ROWS; ++row)
12 {
13     int sum = 0;
14     for(i=row_start[row]; i<row_start[row+1]; ++i)
15         sum += value[i] * x[col_index[i]];
16     y[row] = sum;
17 }
```

Listing 3.13: *Native implementation of the SpMV kernel.*

3.4.1 SpMV for REPLICA

The native implementation in Listing 3.13 can be parallelized by replacing the outermost loop with:

```
for(row=_thread_id; row<ROWS; row+=_number_of_threads)
```

But since the threads then are statically mapped to the loop's iterations, it will result in unbalanced workload if each row does not have the same amount of nonzero values. A better solution is to use a dynamic mapping at runtime. This can be implemented by having a variable that corresponds to the next iteration that has not yet been mapped to a thread. When a thread has finished an iteration it accumulates this variable using the *_aprefix* macro, and receives a new row to compute. For REPLICA's SpMV benchmark see Listing A.5.

Since REPLICA has efficient multioperations it can be interesting to see if the COO format gives better performance compared to the CSR format. Therefore

there is a sparse matrix-vector multiplication benchmark called SpMV-COO, which uses the COO format instead of CSR, implemented for REPLICA. The biggest advantage of using the COO format is that the nonzero elements can easily be mapped evenly over the threads. Its disadvantage is that multiple threads will need to be able to write their results into the same row of the `y` vector simultaneously. Most architectures have some atomic operations that makes this possible, but these operations are normally very slow, and that is why we only implemented SpMV-COO for REPLICA. The SpMV-COO benchmark is displayed in Listing A.6. The SpMV-COO kernel assumes that the result vectors are initialized to zero.

3.4.2 SpMV for Xeon

The native implementation (see Listing 3.13) can easily be parallelized by adding the OpenMP pragma before the outer loop. The scheduling should not be static since the number of nonzero values at each row can vary a lot. A better choice is to use the `guided` clause.

The irregular memory accesses caused by the sparse matrices format is the main issue for cache-based architectures such as Xeon.

The *parallel optimized sparse kernel interface* (pOSKI) is an autotuned library for sparse matrix kernels, developed by the *Berkeley benchmarking and Optimization group* at the University of California, Berkeley [26]. pOSKI is based on the former *optimized sparse kernel interface* (OSKI) library [26]. pOSKI does currently only support double-precision [26], which would be quite unfair to compare with a kernel using integers values. Still it would be interesting to compare results from this highly tuned library with REPLICA's results. Unfortunately we were unsuccessful in compiling pOSKI on the target machine.

The *sparse library* (version 1.5.2) by Yzelman at Katholieke Universiteit Leuven, provides basic operations for a wide range of sparse matrix formats [79]. The `McCRS::zax()` function has been used as a kernel for Xeon. `McCRS::zax()` uses OpenMP to parallelize each vector product. The main difference to the parallelized native implementation is that the sparse library uses pointer-arithmetic instead of array-style indexing. The pointers are declared with the `restrict` [2] keyword, which tells the compiler that no other pointer within its scope will point to the same address. This is true even if the pointers are used with an offset. By knowing this, the compiler might be able to make further optimizations. But GCC generated identical code even when the restrict keyword was removed. The benchmark was compiled using the `-fstrict-aliasing` flag. The nonzero values were of integer type.

Tests showed that the `McCRS::zax()` was significantly slower than the parallelized native implementation. It turned out to be the choice of `dynamic` scheduling instead of `guided`, that had a huge impact. Therefore `McCRS::zax()`'s scheduling has been replaced with `guided`. The data type `long unsigned integer` was originally used for indexing, but was replaced with `unsigned integer` to make it more fair, since REPLICA does not support the former type.

3.4.3 SpMV for XMT

There exist sparse matrix-vector multiplication implementations in earlier work by the XMT project [74]. One of these uses nested `spawn` statements, which XMT currently is not supporting. Another "embarrassingly parallel" implementation where each row of the matrix is processed in parallel is also shortly described, which seems to work like the SpMV kernels for Xeon and REPLICA. For our implementation the outermost loop has been replaced with a `spawn(0, N-1)` statement, see Listing 3.14. Here the dynamic thread scheduler of XMT becomes very handy for matrices with unevenly distributed nonzero values between rows.

```
1  spawn(0, ROWS-1)
2  {
3      int row = $;
4      int start = row_start[row];
5      int end = row_start[row+1];
6      int sum = 0;
7      int i;
8      for(i=start; i<end; ++i)
9      {
10         sum += val[i] * x[col_index[i]];
11     }
12     y[row] = sum;
13 }
```

Listing 3.14: *The DeMM kernel for XMT.*

3.4.4 SpMV for Tesla

For the Tesla benchmark the `spmv_csr_vector_kernel()` CUDA template kernel from the CUSP library [29] has been used, see Listing 3.15. CUSP is a CUDA template library for sparse linear algebra and graph computation [29]. If the constant `UseCache` in Listing 3.15 is set to `true`, it is possible to map vector `x` to Tesla's texture memory, and use it as a cache. This part does not support integer values, which is needed to compile the CUSP library for integer values, even if `UseCache` is set to `false`. Therefore a few modifications of the CUSP library's source code were done, in order to compile it for integer values support. Tests showed however that Tesla performed better without the cache, and it was therefore not used. Since the modifications were not used they should not affect the results, and are therefore not outlined here.

The constants in Listing 3.15 are based on the `__spmv_csr_vector()` template function in the CUSP library, in order to make sure that the kernel is set up correctly.

```
1  const bool UseCache = false;
2  const unsigned THREADS_PER_BLOCK = 128;
3  const unsigned VECTORS_PER_BLOCK = THREADS_PER_BLOCK/THREADS_PER_VECTOR;
4  const unsigned MAX_BLOCKS = 14*1024;
5  const unsigned NUM_BLOCKS = std::min(MAX_BLOCKS,
6    static_cast<unsigned>((ROWS+(VECTORS_PER_BLOCK-1))/VECTORS_PER_BLOCK));
7
8  // Kernel begins
9  cusp::detail::device::spmv_csr_vector_kernel
10   <unsigned,int,VECTORS_PER_BLOCK,THREADS_PER_VECTOR,UseCache>
11   <<<NUM_BLOCKS,THREADS_PER_BLOCK>>>(ROWS,row_start,col_index,val,x,y);
12
13 cudaError_t err = cudaThreadSynchronize();
14 // Kernel ends
```

Listing 3.15: *The SpMV kernel for Tesla.*

The value of the constant THREADS_PER_VECTOR depends on the average number of nonzero values per row. Since THREADS_PER_VECTOR has to be known at compile time, the code in Listing 3.15 was embedded into a template function called SpMV():

```
template <unsigned THREADS_PER_VECTOR>
void SpMV(unsigned* row_start, unsigned* col_index, int* val, int* x, int* y);
```

The value for THREADS_PER_VECTOR can then be set by calling this template function:

```
const unsigned NNZ_PER_ROW = NNZ/ROWS;
if      (NNZ_PER_ROW <=  2) SpMV<2>(row_start, col_index, val, x, y);
else if (NNZ_PER_ROW <=  4) SpMV<4>(row_start, col_index, val, x, y);
else if (NNZ_PER_ROW <=  8) SpMV<8>(row_start, col_index, val, x, y);
else if (NNZ_PER_ROW <= 16) SpMV<16>(row_start, col_index, val, x, y);
else                        SpMV<32>(row_start, col_index, val, x, y);
```

This method and the used values for THREADS_PER_VECTOR comes from the spmv_csr_vector() template function in the CUSP library.

In the CUDA template kernel spmv_csr_vector_kernel() each row in the sparse matrix is computed by THREADS_PER_VECTOR threads in parallel [29]. Threads that cooperate by processing the same row rely on the implicit synchronization among threads with the same warp [29]. Therefore the maximum value for THREADS_PER_VECTOR is limited to Tesla's warp size, which is 32 threads.

3.4.5 SpMV for SB-PRAM

The sparse matrix-vector multiplication for SB-PRAM is almost identical to REPLICA's kernel, even if Fork makes the code slightly simpler to write compared to REPLICA baseline. The main difference between them is that Fork allows SB-PRAM to execute its kernel asynchronously. This is achieved by the farm statement.

```
1  sh unsigned counter = 0;
2  unsigned row;
3  farm for(row=$; row<ROWS; row=mpadd(&counter,1))
4  {
5      int sum = 0;
6      unsigned start = row_start[row];
7      unsigned end = row_start[row+1];
8      unsigned i;
9      for(i=start; i<end; ++i)
10     {
11         sum += val[i] * x[col_index[i]];
12     }
13     y[row] = sum;
14 }
```

Listing 3.16: *The DeMM kernel for SB-PRAM.*

3.5 Breadth First Search (BFS)

The graph algorithm *breadth first search* (BFS) is a widely used technique for traversing graphs which can be applied on many problems.

In these benchmark implementations of the BFS the minimum number of steps it takes to go from a root to any other of the nodes is computed. The number of steps for each node is stored as a result.

The graphs (see Table 3.4) to be traversed comes from the benchmark suite Rodinia 2.4 [11]. The BFS kernels also use the same storage format that Rodinia uses. The format consist of two arrays. The first array contains the nodes, and a second array contain the edges that connect the nodes to each other.

A node is represented as a `struct` which holds the *number of edges* that belong to the node, and an offset to its edges in the second array:

```
typedef struct Node_t
{
    int offset;
    int num_edges;
} Node;
```

The edges are directed and belong to the node from which they come. An edge is simply represented by the index of the node it points to.

Graph name	Number of nodes	Number of nodes
graph4096	4096	24576
graph65536	65536	393216
graph1MW_6	1000000	5999970

Table 3.4: *The graphs that have been traversed by the BFS kernel [11].*

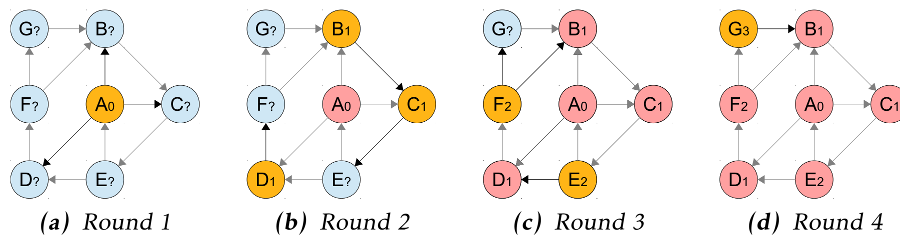
```
1   typedef struct Node_t
2   {
3       int offset;
4       int num_edges;
5   } Node;
6
7   Node node[NODES] = {...};
8   int edge[EDGES] = {...};
9   int cost[NODES] = {...};
10  int visited[NODES] = {...};
11  int queueA[NODES];
12  int queueB[NODES];
13  int queue_end;
14  int next_queue_end;
15
16  int* queue = queueA;
17  int* next_queue = queueB;
18
19  // Put root in queue
20  queue[0] = root_id;
21  queue_end = 1;
22  cost_[root_id] = 0;
23  visited[root_id] = 1;
24
25  while(queue_end)
26  {
27      next_queue_end = 0;
28      // Visit all nodes in the queue
29      for(n=0; n<queue_end; ++n)
30      {
31          // Look up the visiting node in queue
32          int node_id = queue[n];
33          // Iterate over its neighbors
34          for(v=node[node_id].offset; v<node[node_id].offset+
                node[node_id].num_edges; ++v)
35          {
36              int neighbor_id = edge_[v];
37              // Check if neighbor is visited
38              if (!visited[neighbor_id])
39              {
40                  // if not put it in next queue and set cost
41                  next_queue[next_queue_end] = neighbor_id;
42                  cost[neighbor_id] = cost[node_id]+1;
43              }
44          }
45          // Update current queue
46          swap(queue, next_queue);
47      }
48  }
```

Listing 3.17: *A native implementation of the BFS kernel.*

Listing 3.17 shows a native implementation of BFS, which is based on *Algorithm 1* in [17]. The root node is placed in a shared queue data structure `queue`. Then all nodes in `queue` are visited. For the first round `queue` only contains the root node. When a node is visited, all its not yet visited neighbors are added to `next_queue` and have their cost set. After all neighbors have been added the two queues are

swapped, and the algorithm starts over to visit the nodes in `queue`. The algorithm stops when `queue` after a round is empty, that is, if no unvisited neighbors were found after a round. The cost for a node is equal to the number of rounds needed before being visited.

Figure 3.7 shows how a small graph is traversed by the algorithm. In Figure 3.7a the root node *A* is placed in the `queue` of nodes to be visited. The edges to node *A*'s neighbors are marked with black. The next figure (3.7b) shows the next round, followed by round 3 and 4 in Figures 3.7c and 3.7d.



(a) Round 1 *(b)* Round 2 *(c)* Round 3 *(d)* Round 4

Figure 3.7: *An illustration of how REPLICA's BFS kernel traverses a small graph with node A as root. The nodes are labeled **XY**, where **X** is the node name and **Y** is its cost. **Yellow** marks nodes placed in the queue, **red** already visited nodes and **blue** not yet visited nodes.*

3.5.1 BFS for REPLICA

REPLICA's BFS benchmark, see Listing A.7, is based on the native implementation in Listing 3.17. The middle loop (n-loop) in Listing 3.17 which iterates over the queue can easily be parallelized for REPLICA. It will however need a synchronization among all threads after each round since the amount of work per thread will vary. For the first round there will not be enough nodes in the queue to utilize all threads, therefore should also the innermost loop should be parallelized. This has been done in Listing A.7 by using groups of threads. Before each round the threads are divided into as many groups as there are nodes in the queue. If the number of nodes is greater than the number of threads, then each thread will become its own group. The middle loop (n-loop) will then be processed in parallel by the groups. This means that a thread will only be mapped to one iteration in the middle loop, if the number of groups is equal to or less than the number of nodes in the queue. Otherwise the number of groups is equal to the number of threads, which is why the variable `_number_of_threads` is used as increment value for loop variable `n`.

The innermost loop is processed in parallel by one group. Since nodes can be reached from multiple edges, more than one thread might try to set the same unvisited node to visited during the same time step, which can lead to a node being visited more than once. Therefore the `_aprefix()` macro is used to read and set the address that tells if a node is visited.

The same macro is used in order to avoid race conditions when multiple threads write nodes to next_queue.

In the native implementation the cost of a node's neighbor is set by:

```
cost[neighbor_id] = cost[node_id]+1;
```

But since the cost of a node is equal to the round number (variable k in Listing A.7) it is found that it is cheaper to just set it to its round number:

```
cost_[neighbor_id] = k;
```

The built-in variables _group_id and _thread_id are not used to manage groups, since we want to synchronize all threads in all groups using the _synchronize macro at the end of each round. Instead new group and thread ID variables have been defined: group_id and thread_id.

3.5.2 BFS for Xeon

For Xeon the OpenMP version of BFS in Rodinia has been used [11]. The native implementation is suffering from an irregular memory access pattern, which probably is why Rodinia's BFS kernel uses another technique, see Algorithm 3.1. Instead of putting nodes into a queue, nodes that are to be visited are marked. For each round are marked nodes are visited, the cost of their neighbors is set and these are marked to be visited in the next round. Note that there are two marking variables for each node. The first one tells if a node is to be visited in this round, and the second one tells if it should be visited in the next round.

Before the next round, each node found unvisited has to update its marking variables from *to be visited next round* to *visit this round*. This is the analogous operation to swapping queue and next_queue in the native implementation.

To visit the marked nodes of single round, all nodes have to be checked if they are marked, and to update the marking variables, all nodes have to be checked once again. This results in two loops over all nodes for every round. At the first look this seems very inefficient, but since the nodes are accessed consecutively, the cache can be used more efficiently compared to the native implementation. Only the first loop that iterates over the marked loops is parallelized, probably due to that the second loop's body, which updates the marking variables, consists of very little work, and parallelizing it adds too much overhead. Tests showed that it was executed faster in sequential than in parallel.

A parallelized version of the native implementation in Listing 3.17 was also implemented (see Listing A.8) which was compared to the Rodinia BFS kernel. In order to avoid race conditions the __sync_fetch_and_add() function was used in the same manner as the _aprefix() macro was used for REPLICA's BFS kernel.

By comparing Rodinia's kernel to the parallelized native implementation we can make sure that Rodinia's kernel does perform well.

Tests showed that both Rodinia's kernel and the parallelized native implementation executed best with 6 threads, which means that hyper-threading should not be used. One could expect that at least the parallelized native implementation would perform better with hyper-threading, due to high latency for irregular memory accesses, but this was clearly not the case. The Rodinia kernel was roughly twice as fast as the parallelized native implementation and was therefore chosen to be compared against REPLICA.

We have only compared the two versions using the input graphs in Table 3.4. If Rodinia's kernel will keep being faster than the parallel native implementation for larger graphs or for graphs with few edges per node is not tested.

```
mark_c[root] ← true
stop ← false
while stop = false do
    stop ← true

    for all n ∈ node[ ] in parallel do
        if mark_c[n] = true then
            mark_c[n] ← false
            for all edge v held by n do
                neighbor ← edge[v]
                if visited[neighbor] = false then
                    cost[neighbor] ← cost[n] + 1
                    mark_n[neighbor] ← true
                end if
            end for
        end if
    end for

    for all n ∈ node[ ] do
        if mark_n[n] then
            visited[n] ← true
            mark_n[n] ← false
            mark_c[n] ← true
            stop ← false
        end if
    end for

end while
```

Algorithm 3.1: *Pseudocode of Rodinia's BFS kernel for Xeon.*

3.5.3 BFS for XMT

The XMT implementation of BFS (see Listing A.9) is also based on the native implementation in Listing 3.17, and is therefore very similar to REPLICA's kernel. An almost identical algorithm for XMT is presented in [74].

The BFS implementation for XMT in Listing A.9 ha nested `spawn()` statements, and as mentioned before nested parallelism is currently not supported by XMT. The XMT compiler supports however nested `spawn` statements by transforming them into sequential loops. This results in that only the middle loop is actually executed in parallel.

It might be possible to parallelize the innermost loop by spawning as many threads as there are TCUs and dividing them into groups, just like in REPLICA's implementation. But since the threads are not executing in a synchronous manner, it is not possible to use the `psm()` function to compute the size of a group without using some kind of synchronization mechanism after it. Managing groups is therefore likely to add a lot of overhead for each round.

Also note that the kernel has to spawn once for each round in order to keep the threads in the same round and memory coherence. Otherwise it would be tempting to just spawn all threads once in order to reduce overhead.

3.5.4 BFS for Tesla

For Tesla the CUDA version of BFS in Rodinia has been used [11]. Rodinia's CUDA version of BFS is originally from [45]. It uses the marking technique described for the Xeon BFS kernel, instead of using a queue. As it is not possible to synchronize threads over different thread-blocks, the rounds are synchronized by running a kernel for both the loops which iterate over the nodes. Algorithm 3.2 shows how the CPU invokes the CUDA kernels in Algorithms 3.3 and 3.4. *Kernel1()* visits all the marked nodes in parallel, and *kernel2()* updates the stop, visited and marking variables

The CUDA kernels are invoked with thread-blocks of 512 threads in each, which makes the number of thread-blocks to be $\frac{\#NODES}{512}$.

$mark_c[root] \leftarrow true$
$stop \leftarrow false$
while $stop = false$ **do**
    $stop \leftarrow true$
    Invoke CUDA $kernel1(mark_c,\ mark_n,\ node,\ edge,\ cost)$
    Invoke CUDA $kernel2(mark_c,\ mark_n,\ stop)$
end while

Algorithm 3.2: *Pseudocode of Rodinia's BFS kernel for Tesla. See Algorithm 3.3 for kernel1, and Algorithm 3.4 for kernel2.*

$n \leftarrow thread_id$
if $mark_c[n] = true$ **then**
  $mark_c[n] \leftarrow false$
  for all edge v held by n **do**
    $neighbor \leftarrow edge[v]$
    if $visited[neighbor] = false$ **then**
      $cost[neighbor] \leftarrow cost[n] + 1$
      $mark_n[neighbor] \leftarrow true$
    end if
  end for
end if

Algorithm 3.3: *Pseudocode of Rodinia's BFS CUDA kernel1 for Tesla.*

$n \leftarrow thread_id$
if $mark_n[n]$ **then**
  $visited[n] \leftarrow true$
  $mark_n[n] \leftarrow false$
  $mark_c[n] \leftarrow true$
  $stop \leftarrow false$
end if

Algorithm 3.4: *Pseudocode of Rodinia's BFS CUDA kernel2 for Tesla.*

3.5.5 BFS for SB-PRAM

The BFS kernel for SB-PRAM, see Listing A.10, is almost identical to REPLICA's. The main difference is that the group managing is a bit simpler done in Fork compared to REPLICA baseline. Also, the two innermost loops are executed in asynchronous mode. Group sizes can change during execution in asynchronous sections as threads leave the asynchronous sections. Therefore the initial group size is stored in a temporary variable.

3.6 Quicksort (QS)

Quicksort is one of the most popular sorting algorithms, and is based on the well known *divide and conquer* paradigm. Listing 3.18 show a native implementation of quicksort. The `quicksort()` function will recursively call itself until its array size is equal to, or smaller than one integer. If its array is larger than one, then the first element is chosen as `pivot`. The values inside the array are then reordered so that values lower than the `pivot` are placed before the `pivot`, and values higher than the `pivot` are placed after the pivot. We call this partitioning. The `pivot`'s position then becomes the point where the array is split into two new

partitions, which is used as input for the recursive calls. The `pivot` value is already sorted and is therefore not part of the new partitions. Since it is the `pivot` that defines where the array is split, it is very important to choose a good pivot.

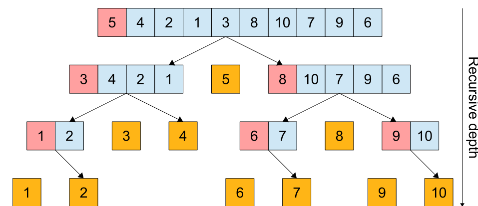


Figure 3.8: *Illustration of the recursive calls for the native implementation of quicksort.* ***Red*** *denotes pivots,* ***yellow*** *denotes sorted values and* ***blue*** *denotes unsorted values.*

Figure 3.8 illustrates the recursive calls for the native implementation of quicksort in Listing 3.18.

```
1  void quicksort(int* first, int* end)
2  {
3      if(first < end-1)
4      {
5          int pivot = first[0];
6          int* s = first;
7          for(int* i=first+1; i!=end; ++i)
8          {
9              if(*i <= pivot)
10             {
11                 s++;
12                 swap(s, i);
13             }
14         }
15         swap(first, s);
16         quicksort(first, s);
17         quicksort(s+1, end);
18     }
19 }
20
21 int main()
22 {
23     int a[N] = {...};
24     quicksort(a, a+N);
25     return 0;
26 }
```

Listing 3.18: *The native implementation of quicksort [41].*

Three pseudo-random input data sets of different sizes have been generated by using the `rand()` function from the *GNU C Library* [9]. Each data set consists of ten integer arrays. The array sizes for the first, second and third set are 10 000, 100 000 and 1 000 000. The same data sets have been sorted by all architectures.

3.6.1 QS for REPLICA

Listing A.12 shows the QS kernel for REPLICA. It consist of three non-recursive phases, which is displayed in Listing A.12, A.13 and A.14. In the first phase the partitioning is performed by the threads within the same group using fine grain parallelism. Each partitioning is done using an input and an output array. The threads use the `_aprefix()` macro in order to place their elements in the output array. The arrays are after each partitioning swapped with each other so that the output becomes the input and vice versa. When partitioning is done, the group splits into two groups which are mapped to each new partition. The size of each group is based on the fraction of elements in its partition, which means that the group with the largest fraction of elements will have the largest fraction of threads[2]. The native implementation in Listing 3.18 can only handle one partition at a time, and puts the second one on an implicit stack by the recursive call. Since we create a group for each partition there is no need to put any of the partitions on a stack. Values equal to the pivot are stored between the two new partitions in the original array since they do not need to be sorted any more. If one of the partitions contains less than two elements, it does not need to be sorted further, and the group does not have to be divided in order to continue sorting the larger one.

Each group needs a shared address space for its shared accumulators `last_less_p` and `first_greater_p`, to be able to use the `_aprefix()` macro for partitioning. The address space between `_shared_heap` and `_shared_stack`, see Figure 3.9, is used for this purpose. A group also needs an address space for the `_group_id`, which is used by the built-in `_synchronize` macro to synchronize threads within the same group. The `_group_id` will be set to `_shared_stack`, see Figure 3.9. Each time a group splits, its shared address space is divided equally between the two new groups. This is illustrated in Figure 3.9. A better solution could be to divide the shared address space based on the fraction of threads. Because the address space is reduced for each group split, it might eventually be too small to split further.

The second phase starts, see Listing A.13, when the shared space for a group is too small to split. Since no shared space is available, the partitioning is processed sequentially. It is however still possible to split the group and map the two new groups to the new partitions. Sequential partitioning can easily be performed in-place, so the temporary array is no longer needed. So if the input is placed in the temporary array when entering the second phase, it will copied over to the original array while doing the partitioning, so that the copying time is not wasted.

When the group only consists of one thread it switches to the third phase. Since the group no longer can be split, the boundaries of the second array after a partitioning are put on an explicit stack.

[2] For even better load balancing one might also have chosen a work-proportional subdivision of the threads, where expected work is $\mathcal{O}(n \cdot log(n))$ for n elements.

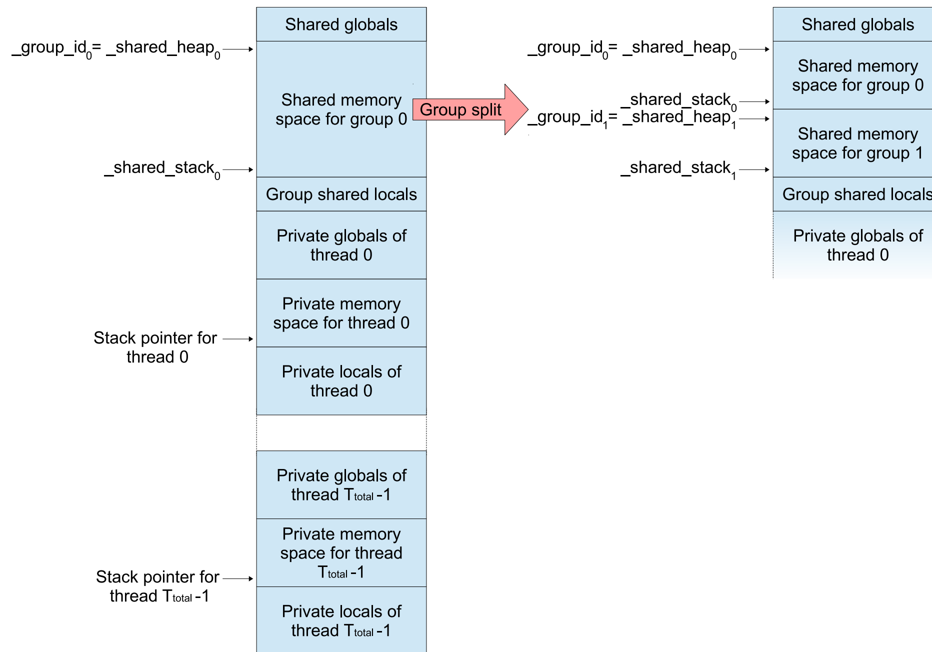


Figure 3.9: *An illustration of how the shared memory is divided during a group split in the QS kernel [81].*

Listings A.13 and A.14 have been included into Listing A.12 with the preprocessor directive `#include`, instead of using function calls, after observing that nested function calls sometimes fail to pass the correct parameter values, due to a bug.

The choice of the pivot is very crucial for the performance of a quicksort implementation, and therefore the pivot is set to the median of three elements in the partition which shall be sorted.

For simplicity, group and shared data are saved and restored in the main function since *par_quicksort()* can return from multiple places.

3.6.2 QS for Xeon

For the Xeon QS kernel the fastest of `quicksort()`, `balanced_quicksort()` and `parallel_sort()` have been chosen. `quicksort()` and `balanced_quicksort()` come from the parallel mode of the *GNU C++ library* [10]. `parallel_sort()` is a quicksort implementation from Intel's multi-threaded C++ template library *Threading Building Blocks* (TBB) [8] . For all data sets, all three implementations performed best when using 12 threads, but `parallel_sort()` was roughly twice as fast as both `quicksort()` and `balanced_quicksort()`.

```
1 tbb::parallel_sort(a, a+N);
```

Listing 3.19: *The QS kernel for Xeon.*

3.6.3 QS for XMT

In [59] a quicksort implementation for XMT is described, and the code can be found in the file *quick2.c* among the second generation of XMT empirical work [73]. This implementation was used as the QS kernel for XMT. But because the second generation of XMT had a slightly different syntax and features it had to be ported. The changes are outlined in Listing A.11.

The implementation consists of two phases. The first phase begins with partitioning the array into two sub-partitions in parallel by spawning a thread for each element. Each thread is then placing its value at either the begin or end of a temporary output array, depending on if its value is less or greater than the pivot. Writing conflicts are solved by using the prefix sum operation `ps()`. The begin and end locations of the new partitions are stored in an auxiliary array. Then the temporary array becomes the input and the original array becomes the output. This is repeated as long as there is less partitions than a certain threshold. The partitions are processed after each other, but the partitioning itself is done in parallel. However, partitions with less elements than three are sorted sequentially.

In the second phase a thread is mapped to each partition, and these are sorted using quicksort. But instead of recursively calling itself, which is not possible since XMT currently does not support function calls inside parallel regions, the thread spawns a new thread for sorting the partition with the higher values than the pivot. If there is only one higher value than the pivot, there is no need to spawn a new thread since the value is already sorted. The lower values are sorted by just reseting its input to the new partition, and therefore there are no function calls needed. This continues until the thread's partition consists of less than three values. The partition with less than three values is easily by a single comparison.

The second phase has an advantage over REPLICA since the work load is dynamically scheduled by its programming model. Similar behavior could be implemented for REPLICA using a task pool.

Since the first phase swaps the data between the original and a temporary array, its is not certain that the data is in the original array when phase two takes over. This means that the sorted array might be placed in the temporary when the kernel finishes. Therefore the implementation was changed so that, if enough partitions are created to leave the first phase but the array is in the temporary array, then the first phase will run an extra round in order to place the data back in the original array. The pseudocode in Algorithm 3.5 describes the two phases in the QS kernel for XMT.

The threshold for when to leave the first phase was chosen by testing for which value of 0, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024 the code had the best performance.

For arrays of 10 000 values the best threshold was 64, and for 100 000 and 1 000 000 values 1024 was the best threshold.

It should also be mentioned that there exists a second implementation of quicksort among the second generation of XMT empirical work [73] in the file *quick_hybrid.c*. The difference between the two implementations is that *quick_hybrid.c* processes all partitions in parallel. It did however not perform as well as *quick2.c*, and was therefore not chosen for this kernel.

```
input ← a[ ]
output ← temp[ ]
// Phase 1
while #partitions < THRESHOLD || input ≠ a do
    for all p partition ∈ input[ ] do
        spawn a thread for each value ∈ p do
            place value → output[ ]
        end spawn
    end for
    swap(input, output)
end while
// Phase 2
spawn a thread for each p partition ∈ input[ ] do
    while #values ∈ p > 2 do
        partition p → p_low and p_hi
        if #values ∈ p_hi > 2 then
            sspawn a thread to sort p_hi
        end if
        p ← p_low
    end while
    if p[0] > p[1] then
        swap(p[0], p[1])
    end if
end spawn
```

Algorithm 3.5: *Pseudocode of the QS kernel for XMT.*

3.6.4 QS for Tesla

NVIDIA provides CUDA implementations of quicksort, but these implementations need to run at a Kepler architecture, and are not compatible with Fermi [63]. Instead *GPU Quicksort* [28] has been used as the QS kernel for Tesla. *GPU Quicksort* consists of two phases, where the goal of the first one is to divide the input array into a large number of sub-arrays which can be sorted independently [28]. Since the number of sub-arrays is low in the first phase, multiple thread-blocks might be partitioning one the same sub-array [28]. This means that thread-blocks need to be able to synchronize between each other, which is done by splitting the first phase into different CUDA kernels [28].

When the number of sub-arrays is larger than a specified $threshold_{\#arrays}$, the algorithm switches to the second phase [28]. Now there exist enough sub-arrays so that each thread-block can be assigned to sort its own sub-array, and there is no need for synchronizations between thread-blocks any more [28]. The second phase runs completely on Tesla with a single CUDA kernel call. If the number of values inside a sub-array is less than a given $threshold_{\#values}$, the quicksort switches to use a bitonic sort algorithm instead.

The number of threads per thread-block and the different thresholds were chosen by testing all combinations of:

- **Threads per thread-block** = $\{32, 64, 128, 256\}$
- $\mathbf{Threshold_{\#arrays}}$ = $\{32, 64, 128, 256, 512, 1024, 2048\}$
- $\mathbf{Threshold_{\#arrays}}$ = $\{64, 128, 256, 512, 1024, 2048, 4096\}$

Tests showed that *GPU Quicksort* performed best with 128 threads per thread-block for all three array sizes. The best $threshold_{\#arrays}$ was 128 for array sizes 10 000 and 100 000. 2048 sub-arrays was best for array size 1 000 000. The best $threshold_{\#values}$ was 2048 for array sizes 10 000 and 100 000; 1024 was best for array size 1 000 000.

The fairness of the comparison can be questioned since GPU Quicksort switches to a bitonic sort algorithm for quite large sub-arrays, which probably will benefit Tesla.

3.6.5 QS for SB-PRAM

The Fork compiler [50] comes with a set of example programs where two are quicksort implementation for arbitrarily large arrays. The idea was to use one of those in the comparison, but unfortunately SB-PRAM ran out of memory under simulating for both of them. Therefore the SB-PRAM quicksort benchmark has been left out.

3.7 Summarizing the Benchmarks

In Table 3.5 the different kernels' are origin summarized. The kernels might differ from the source to fit the problem, or due to optimizations. An $\times$ denotes that it has been implemented by the author.

The *lines of code* (LOC) for each kernel (not the whole program) are presented in table 3.6. For some of the kernels the LOC are not known, due to that the libraries were too complex to easily follow. Empty lines, comments and variable declarations without assignment are not counted. Curly brackets are counted as if they are written on separate lines. Code lines that take care of last iterations due to unrolled loops have not been counted, since not all architectures have checks for this. For Tesla's benchmarks, allocation, transfer and kernel calls in the

host's code are counted and summed together with the code lines in the CUDA kernel.

SB-PRAM has least LOC, and CUDA the most. It is however hard to draw any conclusions about programming effort out of Table 3.6, more than that CUDA seems to need more LOC than the rest of the architectures.

Benchmark	REPLICA	Xeon	XMT	Tesla	SB-PRAM
PS	×	×	Previous XMT work [31]	Thrust [64]	×
DeMM	×	×	Previous XMT work [59]	CUDA C programming guide [62]	×
DeMM-BLAS	-	OpenBLAS [78]	-	CUBLAS [64]	-
SpMV	×	SL [79]	×	CUSP [29]	×
SpMV-COO	×	-	-	-	-
BFS	×	Rodinia [11]	Previous XMT work [59]	Rodinia [11]	×
QS	×	TBB [8]	Previous XMT work [59]	GPU Quicksort Library [28]	-

Table 3.5: *Summarising the benchmark kernels' origins.*

Benchmark name	REPLICA	Xeon	XMT	Tesla	SB-PRAM
PS	4	33	47	?	4
DeMM	34	30	21	36	16
DeMM-BLAS	-	?	-	?	-
SpMV	14	13	12	79	12
SpMV-COO	4	-	-	-	-
BFS	32	36	33	66	32
QS	305	?	174	483	-

Table 3.6: *Lines of code for the different benchmark kernels.*

4

Evaluation and Results

In this chapter some abbreviations will be used. For instance, REPLICA-4, REPLICA-16 and REPLICA-64 refer to REPLICA's architecture with 4, 16 and 64 MBTAC processors, respectively. Further, Xeon-1, Xeon-4, Xeon-6 refer to the Xeon chip, using 1, 4 and 6 processors, respectively.

4.1 Efficiency for REPLICA and Xeon

Since there exists no sequential configuration for REPLICA, the efficiency for the implemented kernels has been obtained by using REPLICA-4 as reference. Therefore the number of processors should be divided by 4 when calculating the efficiency:

$$S_p = \frac{T_4}{T_p} \tag{4.1}$$

$$E_p = \frac{S_p}{P/4} \tag{4.2}$$

Efficiency for Xeon's kernels is obtained by using a single processor as reference, and calculated by:

$$S_p = \frac{T_1}{T_p} \tag{4.3}$$

$$E_p = \frac{S_p}{P} \tag{4.4}$$

The charts in Figure 4.1 illustrate the efficiency of REPLICA-16 and REPLICA-64 relative to REPLICA-4, and of Xeon-4 and Xeon-6 relative to Xeon-1.

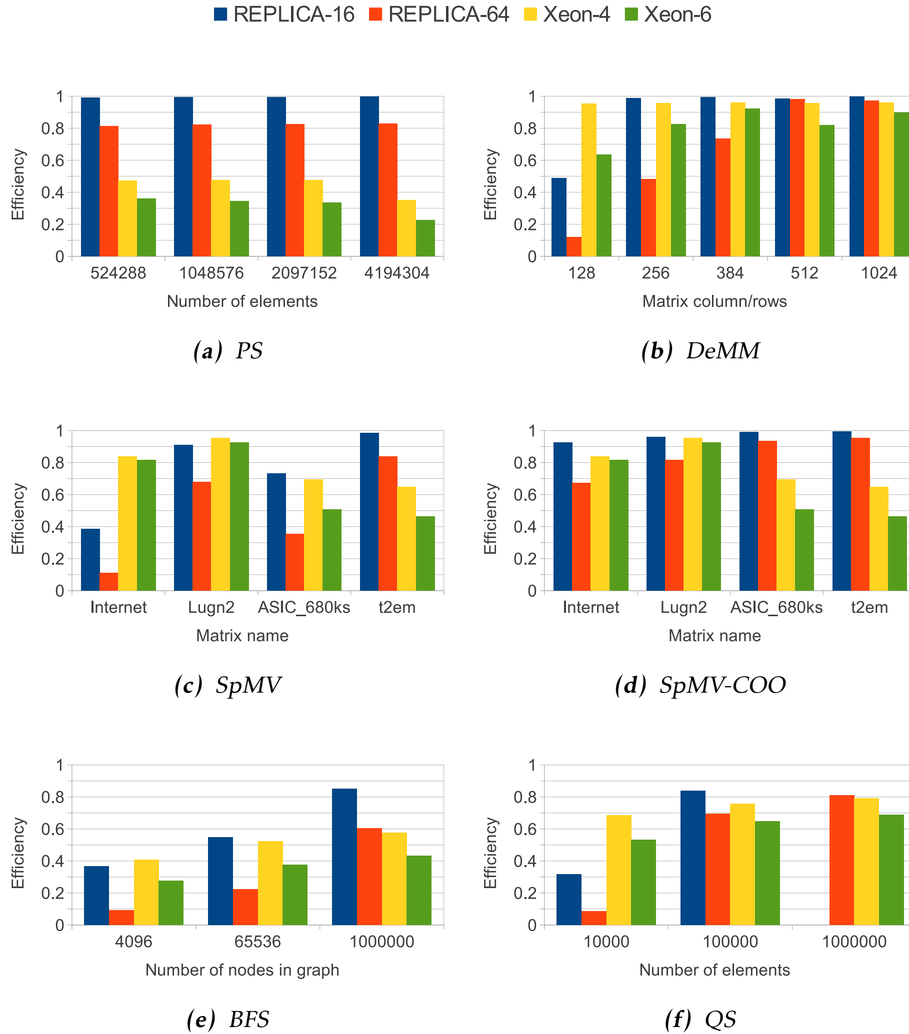


Figure 4.1: *Efficiency of REPLICA-16 and REPLICA-64 relative to REPLICA-4, and of Xeon-4 and Xeon-6 processors relative to Xeon-1. Kernels BFS, PS, DeMM, SpMV and SpMV-COO have been measured with REPLICA's T11 configuration, and QS with T5.*

Due to simulation issues, the QS kernel did not work correctly on the REPLICA's T11 configuration for all inputs. The efficiency for the QS kernel displayed in Figure 4.1f is therefore measured using the T5 configuration. All other kernels in Figure 4.1 use the T11 configuration. The efficiency measurements for 100 000

and 1 000 000 elements in the QS kernel are only based on a single input, instead of an average of 10 inputs. Also the efficiency of REPLICA-16 for 1 000 000 elements is missing due to the mentioned issue.

Figure 4.1a shows that even if REPLICA-64 has more than 10 times more processors to utilize, it still has more than the double of Xeon-6 efficiency for the PS kernel. Also, remember that REPLICA-64 has 32 768 threads (512 threads per processor) that should be utilized. The hardware support for multiprefix operations gives REPLICA a huge advantage over Xeon when computing the prefix sum. The lower efficiency for REPLICA-64 can probably be explained by that the interconnection network is being stressed by the fetch-and-increment done by all 64 processors for each iteration. The four inputs for the PS kernel require 2.1 MB, 4.2 MB, 8.4 MB and 16.8 MB space. Since Xeon's level 3 cache only can store 12 MB, cache misses are expected for the 16.8 MB input. The latency due to cache misses seems to have greater impact when the kernel is executed in parallel. This can be seen in Figure 4.1a where the biggest input yields lower efficiency compared to the smaller inputs on Xeon. In other words, Xeon uses its cache more efficiently when executing in sequential.

DeMM executing on REPLICA-64 (see Figure 4.1b) shows very low efficiency for 128×128 matrices as input. Since a matrix-matrix multiplication of 128×128 matrices contains enough work to utilise all threads of REPLICA-64, the conclusion is that the sequential overhead for dividing the threads into groups are high, relative to the amount of parallel work. Improvements of the part that divides the threads could be done. For example, by getting the assembler instruction `LOGD` to work properly, as mentioned in Section 3.3.1.
For the greater matrices, REPLICA presents better efficiency than Xeon.

The two smallest inputs for the BFS kernel (see Figure 4.1e) have too little parallel work to make use of all REPLICA-64's processors, but the trend shows an increased efficiency while the number of nodes grows. This is due to low parallel work at the kernels' start. When the number of nodes grows, the effect due to low parallel work at start reduces, since its ratio of the total amount of work decreases. Even though REPLICA-16 has more threads than the smallest graph has nodes, it still has better efficiency than Xeon-4. For the greatest input even REPLICA-64 has better efficiency than Xeon-4.

REPLICA shows very low efficiency for the QS kernel (see Figure 4.1f) on small inputs. This is not surprising since REPLICA-64 has 3 times more threads than there are elements to sort. For a sufficient number of elements, REPLICA's efficiency is even better than Xeon's.

According to Figures 4.1c and 4.1d the COO format is to be preferred over CSR for REPLICA's SpMV kernel, when nonzero values are distributed very unevenly among the rows. To determine this the actual performance for each configuration has to be taken into account as well. It can not be done with the efficiency metric alone. Also, there might exist a better implementation of the SpMV kernel using the CSR format. A SpMV kernel implementation for Xeon using the COO format

was not implemented since it would most likely perform very poorly due to high latency of both adding an extra indirect addressing (see the coordinate format in Section 3.4), and the use of atomic operations. In order to compare the efficiency of REPLICA's SpMV-COO kernel with something, Xeon's SpMV kernel (using the CSR format) has been included in the chart. When looking at all the inputs, REPLICA's SpMV-COO kernel shows, in general, higher efficiency than Xeon.

Overall, the irregular memory access algorithms in our benchmark suite, compared to regular ones, are harder to implement efficiently, due to irregular work loads. Algorithms with regular work loads on the other hand, are very easy to implement efficient on REPLICA, as long as the input is sufficiently large. Thanks to REPLICA's multioperations, the SpMV kernel can be implemented with regular work load, which makes it more efficient, by using the storage format COO instead of CSR. It is also possible that REPLICA's NUMA mode could increase the efficiency for kernels with low thread utilization.

The conclusion of this comparison between REPLICA and Xeon's efficiency is that, for sufficient large data sets REPLICA is much easier to program efficiently.

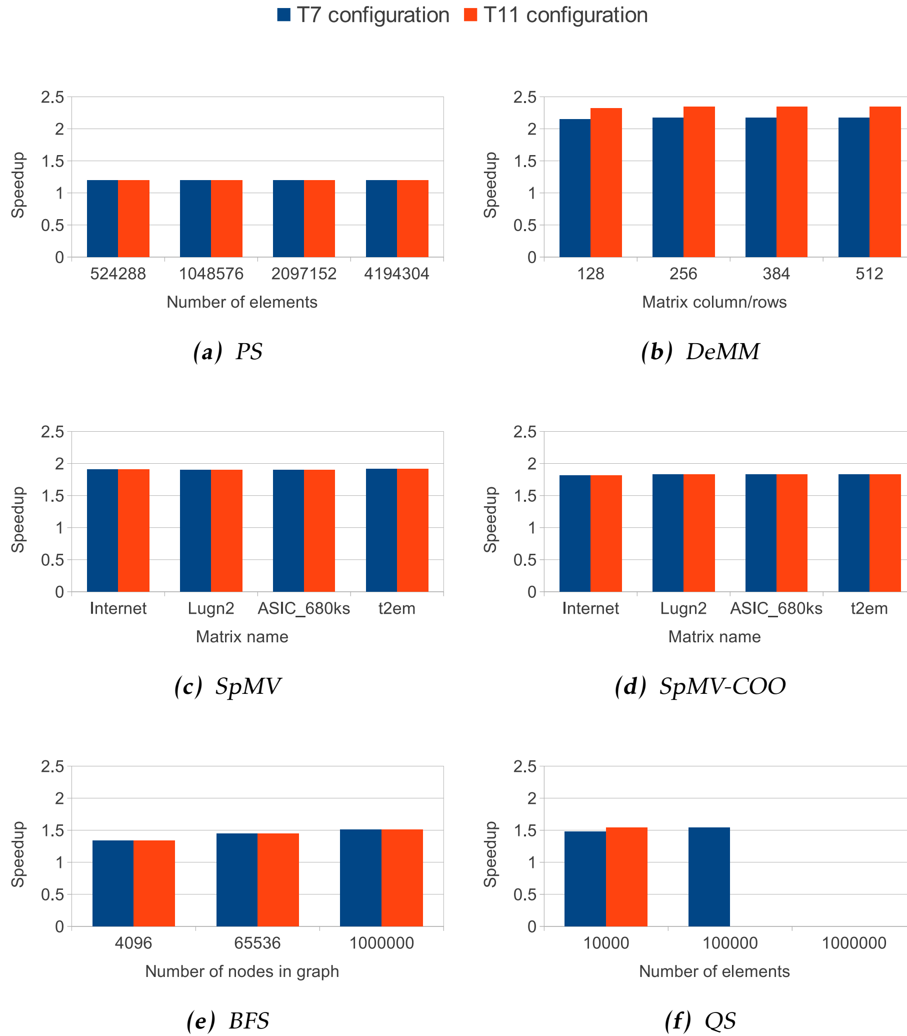
4.2 Instruction Level Parallelism Speedup on REPLICA

The charts in Figure 4.2 show the speedup for T7 and T11 relative to T5, using 4 processors. The gained speedup for T7 and T11 comes from exploiting more ILP and VILP than T5.

That the PS kernel has the least speedups could be expected, since it only consists of one loop including a multiprefix instruction.

Programs with a lot of control flow statements, for example conditional jumps, tends to chop the program into small basic blocks. The compilers that is used will only do ILP and VILP optimizations within these basic blocks, which results in less ILP for programs with a lot of control flow statements. Unrolling the innermost loop for the DeMM kernel increased the ILP of the most frequently executed basic block. This is probably the main reason why it had the best ILP speedup among the kernels. The observant reader can see that the graph traversal dwarfs (QS and BFS) have less speedup compared to the linear algebra dwarfs. The dense linear algebra dwarf, DeMM, have higher speedup compared to the sparse algebra dwarfs, SpMV and SpMV-COO. This can probably be explained by that graph traversal codes tend to use a lot of control flow statements compared to code for linear algebra. There are of course too few kernels representing the different dwarfs to prove that this is true for all kernels within these dwarfs.

The T7 configuration seems to be a good choice over the fat T11, since the compiler fails to find enough ILP to utilize its functional units well (at least for these kernels). The silicon space could be used for other features instead, such as more on-chip memory.



(a) PS *(b) DeMM*

(c) SpMV *(d) SpMV-COO*

(e) BFS *(f) QS*

Figure 4.2: *ILP speedup for T7 and T11, relative to T5.*

4.3 Frequency Evaluation

This section will evaluating REPLICA by calculating what clock rate REPLICA needs to run at to be able to execute the benchmarks as fast as Xeon and Tesla. The *needed frequency* is calculated by dividing the execution time for Xeon or Tesla with the number of clock cycles that REPLICA executes for the same benchmark. The needed frequency is based on REPLICA's T11 configuration, except for the QS benchmark where T5 has been used instead.

Estimating the maximum clock rate for REPLICA architectures is out of scope for

this thesis, but REPLICA's interconnection network has been estimated to run at the clock rate of at least 2 GHz with an old 65 nm technology [43].

4.3.1 Needed Frequency for PS

Figure 4.3 shows REPLICA-16 and REPLICA-64's needed frequency for executing the PS kernel as fast as Xeon does. REPLICA-16 needs to operate on roughly a third of Xeon's clock rate to match Xeon execution time for the four smallest inputs. As mentioned in Section 4.1, Xeon struggles with cache misses for the largest input which is clearly visible in the chart. Due to Xeon's cache misses, REPLICA-16 needs a clock rate less than 600 MHz, and only 176 MHz for REPLICA-64.

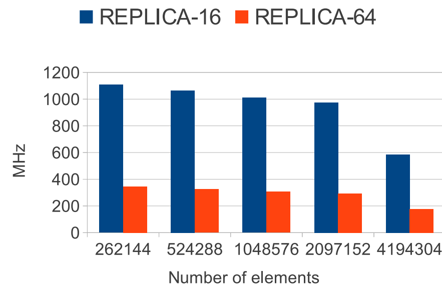
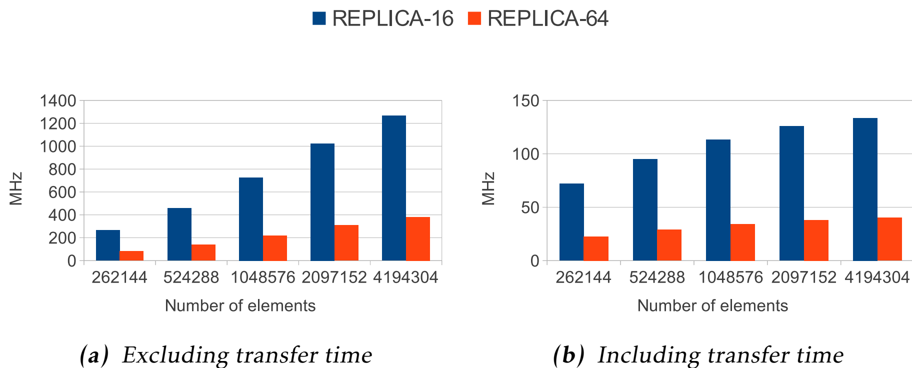


Figure 4.3: *Needed frequency in MHz against Xeon for PS.*

According to Figure 4.4a, Tesla performs better while the problems size increases. When transfer time is included (see Figure 4.4b) this effect seems to flatten out for larger problems. Also a very low frequency is needed for both REPLICA-16 and REPLICA-64 to match Tesla. REPLICA-64 shows overall good performance compared to both Xeon and Tesla. The needed frequency for REPLICA-4 is displayed in Tables B.18, B.19 and B.20,



(a) *Excluding transfer time*

(b) *Including transfer time*

Figure 4.4: *Needed frequency in MHz against Tesla for PS.*

4.3.2 Needed Frequency for DeMM

The very varied results of DeMM made it hard to show them in charts, so the results are presented in tables. Table 4.1 shows the needed frequency in MHz for REPLICA with 4, 16 and 64 processors running the DeMM kernel. When using a 1024×1024 matrix as input, REPLICA-16 and REPLICA-64 only need to operate on 442.5 MHz and 113.4 MHz respectively to match Xeon's execution time. REPLICA seems to have better performance (relative to Xeon) when the number of rows grows. This is probably due to that REPLICA's multioperation gets more effective as more threads cooperate.

When REPLICA's DeMM kernel is compared to Xeon's highly optimized and tuned DeMM-BLAS kernel, the needed frequency is by roughly a factor of 10 larger, see Table 4.2. Although the comparison is a bit unfair since OpenBLAS is much more optimized than REPLICA's kernel, REPLICA-64 needs lower frequency than Xeon's clock rate (see Section 2.2), except for 128×128 matrices.

Rows	REPLICA-4 (MHz)	REPLICA-16 (MHz)	REPLICA-64 (MHz)
128	1 607.3	825.3	835.9
256	2 077.7	526.2	270.0
384	3 095.3	777.9	263.2
512	1 984.0	503.9	126.5
1024	1 764.6	442.5	113.4

Table 4.1: *Needed frequency in MHz against Xeon for DeMM.*

Rows	REPLICA-4 (MHz)	REPLICA-16 (MHz)	REPLICA-64 (MHz)
128	16 819.6	8 637.1	8 747.8
256	21 328.8	5 401.4	2 772.1
384	34 178.8	8 590.1	2 906.7
512	24 630.3	6 255.0	1 570.3
1024	25 377.7	6 364.4	1 630.6

Table 4.2: *Needed frequency in MHz against Xeon for DeMM-BLAS kernel.*

Tesla's transfer time (see Table B.13) is 99 % of the total execution time for both DeMM and DeMM-BLAS kernels. Since the kernel spend almost all the executing time on transferring data between host and device, it does not matter if the DeMM or DeMM-BLAS kernel is executed, because there is no point to optimize code whose execution time is less than 1 % of the total time. The long transfer time leads to a serialisation of the kernel, which leads to very poor results for Tesla when transfer time is included. With transfer time included REPLICA's needed frequency is just few MHz, see Table 4.3

When the transfer time is excluded REPLICA's needed frequency is very high, for both DeMM (see Table 4.4) and DeMM-BLAS (see Table 4.5).

Rows	REPLICA-4 (MHz)	REPLICA-16 (MHz)	REPLICA-64 (MHz)
128	2.7	1.4	1.4
256	24.3	6.2	3.2
384	38.2	9.6	3.3
512	31.9	8.1	2.0
1024	41.6	10.4	2.7

Table 4.3: *Needed frequency in MHz against Tesla for DeMM-BLAS kernel including transfer time.*

Rows	REPLICA-4 (MHz)	REPLICA-16 (MHz)	REPLICA-64 (MHz)
128	16 905.3	8 681.1	8 792.4
256	32 504.1	8 231.4	4 224.6
384	46 566.6	11 703.5	3 960.2
512	36 580.9	9 289.9	2 332.2
1024	36 483.0	9 149.5	2 344.2

Table 4.4: *Needed frequency in MHz against Tesla for DeMM excluding transfer time.*

Rows	REPLICA-4 (MHz)	REPLICA-16 (MHz)	REPLICA-64 (MHz)
128	20 276.5	10 412.3	10 545.7
256	44 062.8	11 158.6	5 726.9
384	121 499.6	30 536.3	10 332.9
512	86 050.5	21 852.9	5 486.0
1024	131 939.9	33 089.0	8 477.6

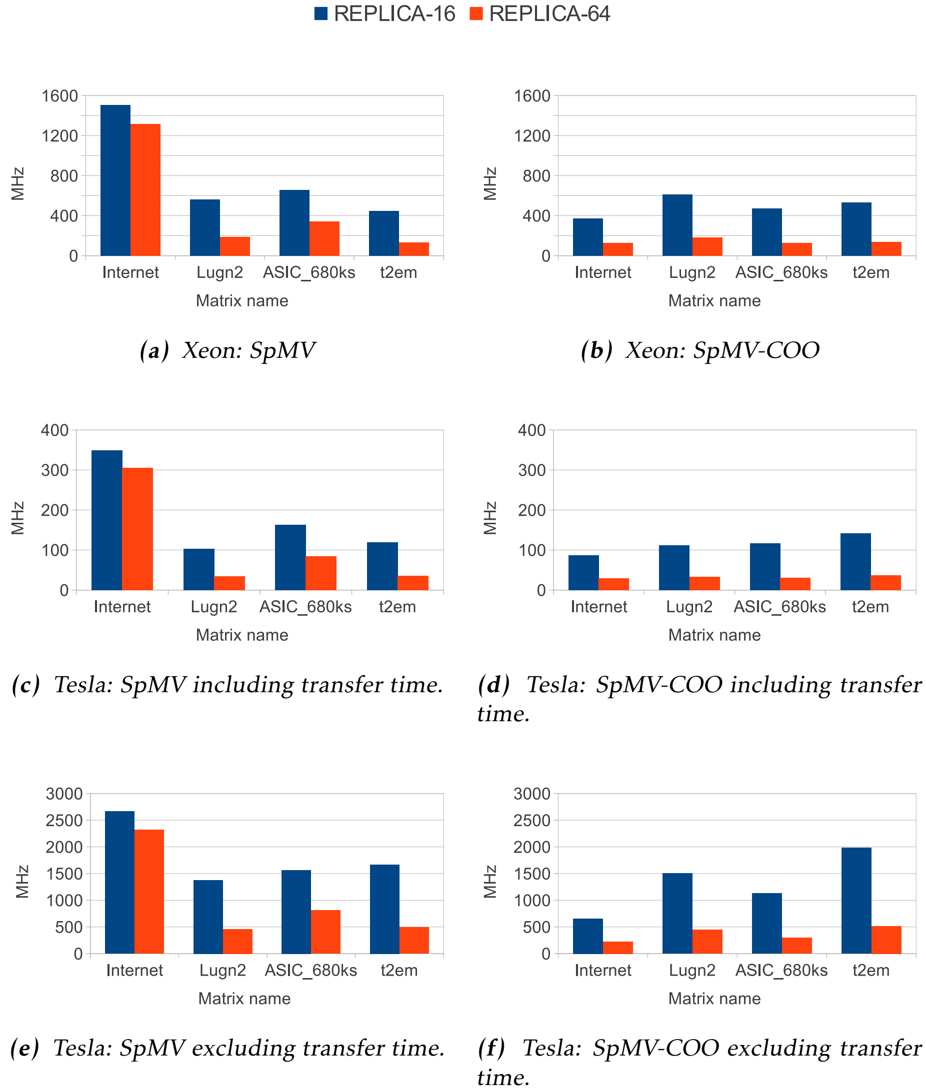
Table 4.5: *Needed frequency in MHz against Tesla for DeMM-BLAS kernel excluding transfer time.*

The DeMM and DeMM-BLAS benchmarks show that dense matrix-matrix multiplication suits both Xeon and Tesla. This is because the memory access pattern is known at compile time, and data can easily be stored in Xeon's cache and Tesla's shared memory before its needed. But when Tesla's transfer time is included it perform very poorly. The DeMM kernel is too small to be beneficial for Tesla.

4.3.3 Needed Frequency for SpMV

Figure 4.5 shows again that REPLICA's performance is highly reduced for the *Internet* matrix when using the CSR format. This is because the nonzero values are very unevenly distributed among the matrix's rows. The number of nonzero values for a row is between 0 and 138 values. 41 % of the 124651 rows are empty, but will still have to be visited by a thread. This results in unbalanced work load, which has a greater impact on REPLICA than on Xeon and Tesla, since the idling threads still keep their slots in the MBTAC processor's pipeline. Idling threads which block working threads can hopefully be solved in the future by merging

thread slots using NUMA-mode.



(a) Xeon: SpMV

(b) Xeon: SpMV-COO

(c) Tesla: SpMV including transfer time.

(d) Tesla: SpMV-COO including transfer time.

(e) Tesla: SpMV excluding transfer time.

(f) Tesla: SpMV-COO excluding transfer time.

Figure 4.5: *Needed frequency in MHz against Xeon and Tesla, for SpMV and SpMV-COO kernels.*

However, the alternative kernel SpMV-COO solves the issue with unevenly distributed nonzero values, thanks to REPLICA's multioperations. Since the SpMV-COO kernel only exists for REPLICA, it has been compared against Xeon and Tesla's SpMV kernels.

REPLICA needs much lower clock rate than Xeon, except for the *Internet* ma-

trix, for the SpMV kernel. When the COO format is used, the needed frequency for REPLICA-64 is reduced even more for all matrices, except for the *t2em* matrix. For REPLICA-16 the needed frequency is only reduced for the *Internet* and *ASIC_680ks* matrix. The highest needed frequency for REPLICA-16 and REPLICA-64 for SpMV-COO are 611 MHz and 108 MHz, respectively.

REPLICA's needed frequency against Tesla, with transfer time included, looks much like the needed frequency against Xeon. But the needed frequency is much lower when comparing against Tesla than against Xeon. REPLICA-16 needs less than 200 MHz for both the SpMV and the SpMV-COO kernel, if the *Internet* matrix is excluded.

If Tesla's transfer time is excluded, the maximum needed frequency for REPLICA-64 executing SpMV-COO is 512 MHz.

The SpMV and SpMV-COO kernels show that REPLICA performs well against both Xeon and Tesla, if the work is balanced.

4.3.4 Needed Frequency for BFS

While looking at the REPLICA's needed frequency for BFS against both Xeon (Figure 4.6) and Tesla (Figure 4.7), it is clear that an important algorithm such as BFS suits today's architectures badly, compared to REPLICA. The needed frequency for REPLICA-4, REPLICA-16 and REPLICA-64 against Xeon are 257 MHz, 76 MHz and 27 MHz, for the largest graph.

REPLICA needs a bit higher frequency to match Tesla, but the needed frequency is still low, both when including and excluding the transfer time. If the transfer time is excluded, the needed frequency for REPLICA-4, REPLICA-16 and REPLICA-64 are 618 MHz 182 MHz and 64 MHz, respectively, for traversing the largest graph.

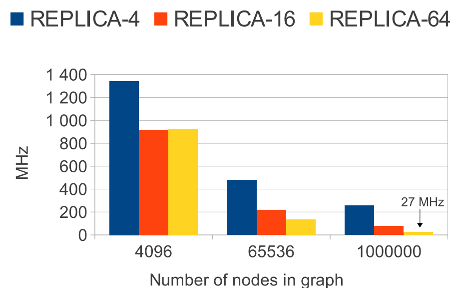
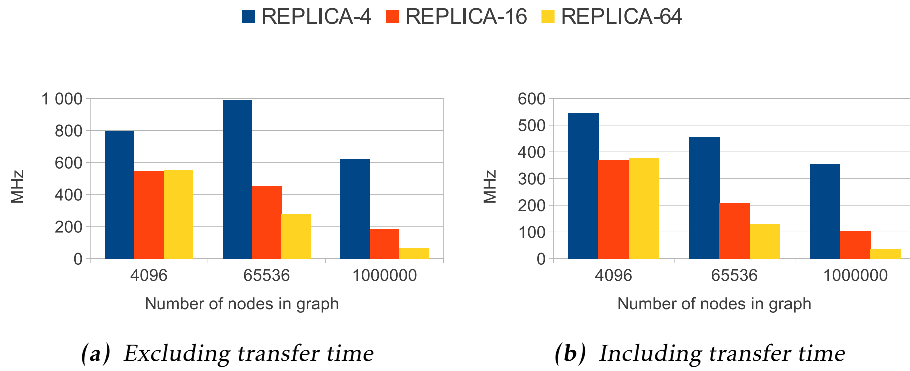


Figure 4.6: *Needed frequency in MHz against Xeon for BFS.*

The lack of parallel work for the smallest graph leads to idling thread-slots in the MBTAC processor's pipeline, which results in decreased performance for REPLICA. But when the graph grows, more node neighbors are queued for each iteration, which means more parallel work and less idling thread-slots.



(a) Excluding transfer time *(b) Including transfer time*

Figure 4.7: *Needed frequency in MHz against Tesla for BFS.*

The results of the BFS kernels show that the REPLICA architecture fills a gap in today's spectrum of parallel computing architectures.

4.3.5 Needed Frequency for QS

Due to problems with simulator or compiler the T7 and T11 configuration did not run the QS benchmark correctly. Therefore the lightweight configuration T5 has been used instead.

REPLICA-4's needed frequency against Xeon for executing the QS kernel is very high, see Figure 4.8. It needs to run at roughly 8.7 GHz to match Xeon for the smallest input. For the middle size input REPLICA-16 and REPLICA-64's needed frequency are 2.4 GHz and 1.5 GHz respectively, which both are lower than the frequency that Xeon operates on. As mentioned earlier REPLICA-16 is missing results for the largest input, but REPLICA-64 has a needed frequency of 720 MHz which is roughly one quarter of Xeon's clock rate. The needed frequency also decreases for REPLICA-16 and REPLICA-64 as the input size grows.

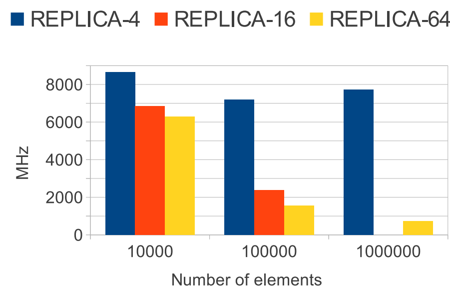
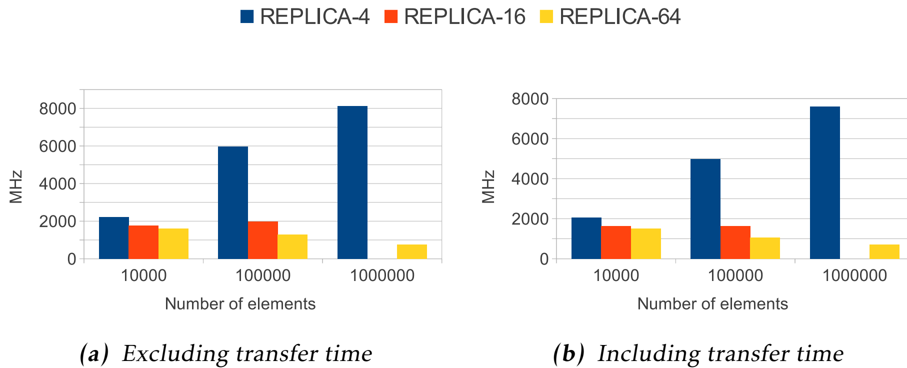


Figure 4.8: *Needed frequency in MHz against Xeon for QS.*

The needed frequency against Tesla for small input sizes is not as high as com-

pared to Xeon, see Figure 4.9. This might be due to communication overhead between host and device, since the kernel is executing on both host and device. The transfer time at the start and end is however not the reason, since the needed frequencies for when the transfer time is included and when not are about the same frequency. The needed frequency for REPLICA-16 and REPLICA-64 against Tesla, including transfer time, are 1.6 GHz and 1.1 GHz respectively for the middle input size. For the largest input size REPLICA-64 needs a clock rate of 708 MHz to match Tesla, including transfer time.

Only for REPLICA-64 the needed frequency decreases as the input size grows.



(a) Excluding transfer time *(b) Including transfer time*

Figure 4.9: *Needed frequency in MHz against Tesla for QS.*

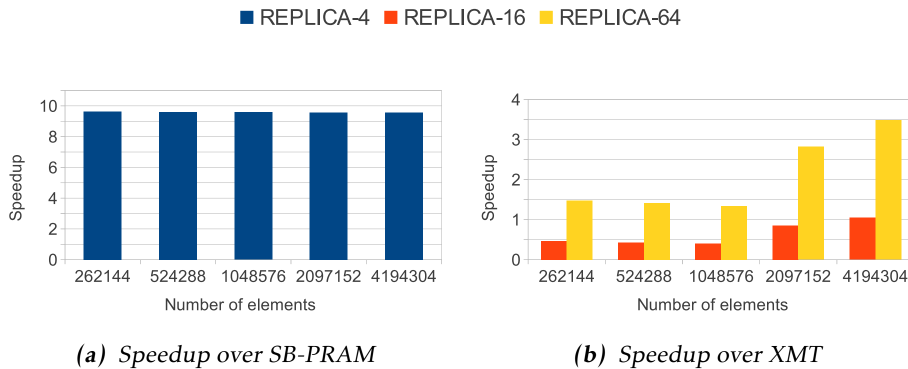
Tests have shown that it might be possible to reduce REPLICA's execution time for sorting the largest array with around 40 % by getting the T11 configuration to work, and not treating values equal to the pivot separately during partitioning. The pivots shall instead be treated as either larger or smaller than the pivot. But it can lead to reduced performance if the input contains a lot of repeated values.

4.4 Clock Cycles Evaluation

This section will compare REPLICA's performance against SB-PRAM and XMT by comparing executed clock cycles within each benchmark kernel. REPLICA-4 will be compared against SB-PRAM since they have the same number of processors and threads. A TCU cluster can be seen as a processor with 16 threads, and since XMT has 64 clusters it is natural to compare it with REPLICA-64. But it can also be of interest to see how the smaller configuration REPLICA-16 stands against XMT. Therefore REPLICA-16 is also compared against XMT. Again the T11 configuration is used for REPLICA except for the QS kernel where T5 will be used.

4.4.1 Clock Cycles Evaluation for PS

REPLICA-4 has a speedup of 9.6 over SB-PRAM for all input sizes, see Figure 4.10a. That the speedup stays almost constant for all input sizes is natural since both kernels iterate over a multiprefix operation the same number of times. The multiprefix operation seems to execute roughly 9 times faster on REPLICA compared to SB-PRAM, when all threads on the chip are cooperating. The high speedup can probably be due to that REPLICA is a VLIW architecture and has a more powerful instruction set. REPLICA's scratchpad and step cache also make the multioperations and multiprefix operations more effective when executed by a large number of threads.



(a) Speedup over SB-PRAM

(b) Speedup over XMT

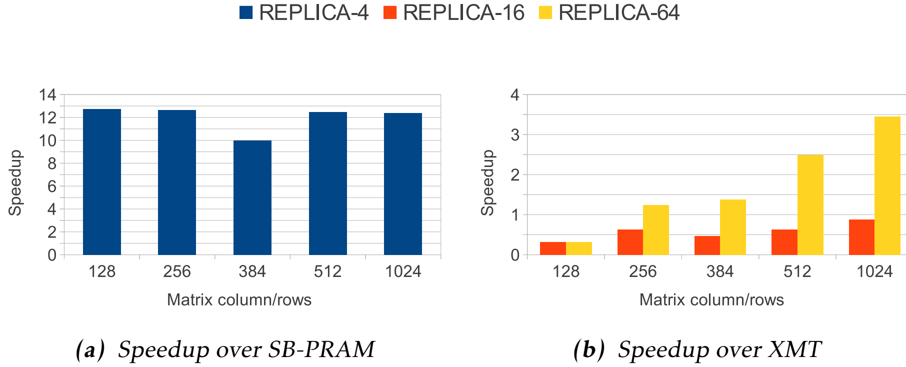
***Figure 4.10:** REPLICA's speedup over SB-PRAM and XMT for PS.*

Also REPLICA-64 shows speedup over XMT for all inputs in Figure 4.10b. For the three smallest inputs REPLICA-64's speedup over XMT is around 1.4, which is quite modest. But for the two largest inputs sizes the speedups are much higher. REPLICA-64 has a speedup of 3.5 over XMT for the largest input. REPLICA-16 executes slower than XMT for all input sizes, except for the largest one which has a small speedup of 1.05 over XMT. XMT's performance against both REPLICA-16 and REPLICA-64 seems to decrease as the number of elements grows.

4.4.2 Clock Cycles Evaluation for DeMM

REPLICA-4 has a speedup over SB-PRAM of around 12.5 for all input matrices, except for the 384×384 matrix where the speedup is 10.0, which is shown in Figure 4.11a. The lower speedup for the 384×384 matrix is due to that $\frac{384}{4}$ (= 96) is not a power of two. The reason why 384 is divided by 4 is that the loops are unrolled 4 times. Because the number of threads on REPLICA-4 is a power of two (2048), also the group sizes have to be a power of two, if they shall be of equal sizes. But since 96 is not, the group sizes are set to the nearest power of two rounded downwards from 96, which is 64. This results in a less efficient use of the multioperation than if the group sizes would have been $\frac{\#ROWS}{4}$. SB-PRAM does not use groups in its kernel and is not affected by the this. The high speedup over SB-PRAM is probably due to REPLICA's multioperation and a more

optimized compiler.



(a) Speedup over SB-PRAM

(b) Speedup over XMT

Figure 4.11: *REPLICA's speedup over SB-PRAM and XMT for DeMM.*

According to the chart in Figure 4.11b XMT executes with less clock cycles than REPLICA-16 for all matrices, although XMT's performance over REPLICA-16 seems to shrink for larger matrices. REPLICA-64 presents speedup over XMT for all matrices except the smallest one. The increasing speedup for REPLICA-64 over XMT can be due to an increased advantage of the multioperations when larger groups are cooperating. If this is the case, REPLICA-64's speedup over XMT will probably grow until the matrices are beyond 2048×2048, since the group then will fully utilize the scratchpad and step cache. Another possible explanation is that REPLICA's kernel has more initial overhead when threads are divided into groups. The initial overheads will be more and more insignificant as the matrices grows.

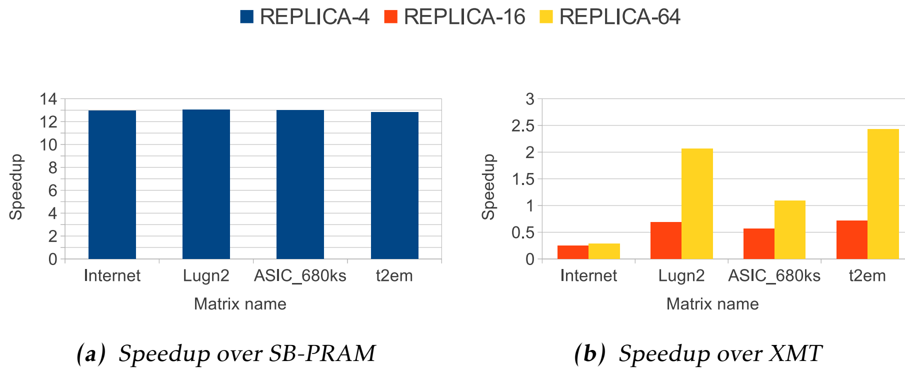
REPLICA-64 has speedup of 3.5 over XMT for the largest matrices.

4.4.3 Clock Cycles Evaluation for SpMV

For the SpMV kernel, REPLICA again shows a stable speedup over SB-PRAM, see chart in Figure 4.12a. The speedup is close to 13 for all matrices. The reason for the high speedup is again probably due to better optimizing by the compiler, and maybe more efficient multiprefix operations.

Since the SpMV-COO kernel only exists for REPLICA, it has been compared against SB-PRAM and XMT's SpMV kernels. The chart in Figure 4.13a shows REPLICA-4's speedup over SB-PRAM when REPLICA runs SpMV-COO instead of SpMV. It shows an increased speedup for REPLICA over SB-PRAM for the *Internet* matrix, which has unevenly distributed nonzero values. A small increased speedup for the *ASIC_680ks* matrix can also be observed, but the speedup for both *Lugn2* and *t2em* decreases to roughly 11.

The speedup in Figure 4.12b shows that XMT has better performance than REPLICA-16 for all matrices. But XMT also performs better than REPLICA-64 for unevenly distributed nonzero values in the matrix, such as the *Internet* matrix. For

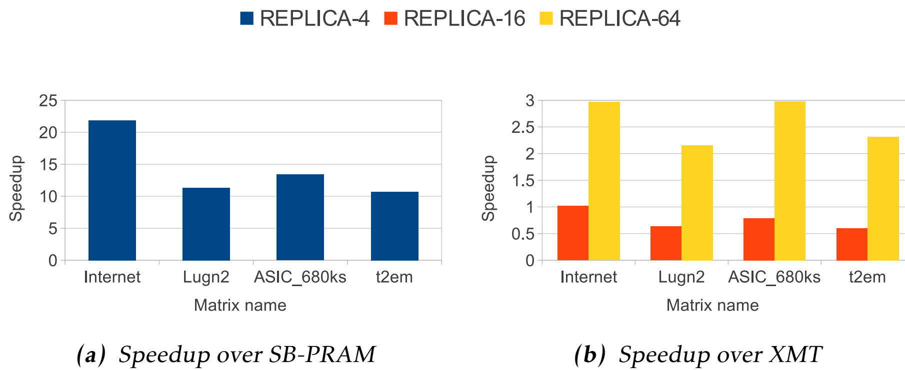


(a) Speedup over SB-PRAM *(b) Speedup over XMT*

Figure 4.12: *REPLICA's speedup over SB-PRAM and XMT for SpMV.*

the other matrices REPLICA-64 has a speedup of 2.1, 1.1 and 2.4 over XMT.

When comparing the SpMV-COO to XMT's SpMV in Figure 4.13b, REPLICA-64 presents speedups between 2.2 and 3.0 over XMT. REPLICA-16, on the other hand, only has a very small speedup over XMT for the *Internet* matrix.



(a) Speedup over SB-PRAM *(b) Speedup over XMT*

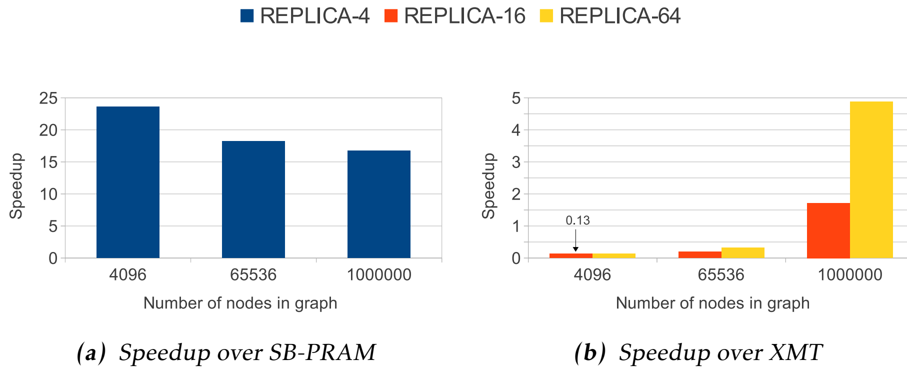
Figure 4.13: *REPLICA's speedup over SB-PRAM and XMT for SpMV-COO.*

4.4.4 Clock Cycles Evaluation for BFS

The chart in Figure 4.14a shows again a very high speedup for REPLICA over SB-PRAM. As for the other kernels this might be due to that REPLICA's compiler makes better optimizations, and that its multiprefix operations are more efficient than SB-PRAM's implementation. It is also possible that the Fork language introduces more overhead than the REPLICA baseline language, since Fork is a high-level language compared to REPLICA baseline.

XMT outperforms both REPLICA-16 and REPLICA-64 for small graphs. But for the largest graph it is the opposite. REPLICA-16 and REPLICA-64 have a speedup of 1.7 and 4.9 over XMT for the largest graph. Small graphs have much less par-

allel work than larger graphs. The result of little parallel work is idling threads. Since an idling TCU in XMT does not block active threads from accessing the shared resources, it can perform better than REPLICA when the amount of parallel work is low. REPLICA can probably solve this by grouping idling threads into one thread bunch, using NUMA mode.



(a) *Speedup over SB-PRAM* *(b)* *Speedup over XMT*

Figure 4.14: *REPLICA's speedup over SB-PRAM and XMT for BFS.*

4.4.5 Clock Cycles Evaluation for QS

Although the results have been obtained using the lightweight T5 configuration, REPLICA-64 shows speedup over XMT for the two largest inputs. The speedup is however not that high, roughly 1.3, but it will grow if the issues with T7 and T11 are solved.

As mentioned in Section 3.6.5 the SB-PRAM ran out of memory when simulating the QS benchmark, therefore, there are no results for it.

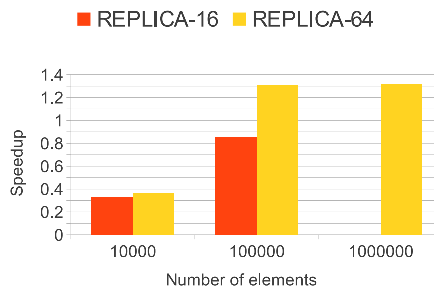


Figure 4.15: *REPLICA's speedup over XMT for QS.*

5 Conclusions and Future Work

5.1 Conclusions

In this thesis we have, by benchmarking different types of algorithms on REPLICA, Xeon, XMT, Tesla and SB-PRAM, tried to show how REPLICA is positioned among similar parallel architectures and more diverse ones. The selected reference consist of both commercially-off-the-shelf and in-house research oriented architectures. By using well-known Berkeley dwarfs and input from unbiased sources, such as *The University of Florida Sparse Matrix Collection* [1] and *Rodinia benchmark suite* [11], we have made sure that the benchmarks measure relevant computation problems. When possible, the benchmarks have used highly optimized state-of-the-art vendor-provided or third-party libraries, to make the reference implementations more objective.

We have shown that it is easier to program REPLICA efficiently than Xeon, considering that REPLICA has thousands of threads more than Xeon to utilize, see Section 4.1. It is very easy to assign extremely fine grained computation tasks efficiently on REPLICA, even if a low-level language such as REPLICA baseline was used. The *lines of code* (LOC) metric indicates that it is more time-consuming to develop programs for Tesla, and not very surprisingly our results show that in order to gain any performance out of Tesla as a computational accelerator, the kernel has to be relatively large, due to high data transfer times. When the transfer time was included, only the quicksort (QS) kernel showed good results for Tesla. But this kernel is actually executing on both the CPU and Tesla. Further, this kernel also switched from using the quicksort algorithm to a bitonic sort algorithm at relatively large thresholds.

Based on its speedup over the T5 MBTAC configuration for our benchmarks, T7 is

probably the most cost-efficient configuration among the three evaluated MBTAC configurations. The T7 configuration obtains between 1.2 and 2.2 speedup over the T5 configuration, by exploiting more ILP and VILP. The T11 configuration offers little more speedup than T7, but the extra silicon space can be used for more useful features, such as more on-chip memory.

Overall REPLICA performed well against the reference architectures. Best results were obtained for the breadth first search (BFS) kernel with the largest graph, where REPLICA only needed a few MHz to match Xeon and Tesla. REPLICA also had almost 5 times speedup over XMT for this kernel. In general REPLICA performed better for larger inputs. The kernel for which REPLICA needed the highest clock rate to match Xeon and Tesla was the BLAS dense matrix-matrix multiplication kernel (DeMM-BLAS). This was no surprise since both Xeon and Tesla used highly optimized BLAS libraries. Regular memory access algorithms are also easy to implement efficiently on cache-based architectures such as Xeon, and matrix multiplication is a regular dataparallel operation that GPUs such as Tesla are optimized for.

Thanks to REPLICA's multioperations, sparse matrix-vector multiplication for the COO format can be implemented efficiently with very little effort. Since REPLICA can compute sparse matrix-vector multiplication efficiently, there is no need for wasting valuable time on converting matrices to (for other architectures) more efficient formats. The benefits for the COO format over CSR had been even greater if we had included the time it took to convert the matrices to CSR.

Some issues regarding the different architectures have also been observed. Code with unbalanced or low amount of parallel work leads to idling threads. Idling threads can have a great performance impact, especially on REPLICA since its idling threads, if running in PRAM mode, still occupy their thread-slots in the MBTAC processor's pipeline, which results in bad utilization and performance. It can most likely be solved by grouping idling threads into one or several thread-bunches, which execute as a single thread in NUMA mode. This was not tested. Also, some codes produce small basic blocks which make it difficult for the compiler to optimize for ILP and VILP. In some cases, this can be solved by unrolling loops, so that the loop iterates over a larger basic block, which makes it easier for the compiler to optimize for ILP and VILP. XMT does, in contrast to REPLICA, have problems with register spills for large basic blocks within parallel regions. If the compiler fails to store local variables and temporary values in the TCU's register file, it will halt the compilation. This was observed when trying to unroll the innermost loop for dense matrix-matrix multiplication (DeMM). XMT also limits the programmer by not supporting nested parallel regions, or function calls inside parallel regions. The XMT project is planning to support this and solve the register spill issue in a future XMT version [21].

It is noticeable that REPLICA's tool-chain is in an early stage. A few bugs with nesting functions, register allocation and the simulator IPSMSimX86 have been observed. Also managing shared data among groups in REPLICA baseline is a bit tedious, but will be solved by the REPLICA language, which is under develop-

ment.

The XMT programming model does not allow execution of code beyond a parallel region before all software threads have been executed, due to an implicit barrier after each parallel region. This means that a single TCU which executes a software thread can block all other TCU's on the chip from execution.

Both Tesla and XMT can not support global synchronization between software threads, since all threads on XMT, and blocks on Tesla, must be able to execute independently and sequentially. A global synchronization behavior can however be implemented by splitting the parallel region, or CUDA kernel, into two parts where the global synchronization is desired.

If one compares the thread synchronization and memory consistency between the architectures, then REPLICA and SB-PRAM have the strongest programming models. They do however not spawn software threads, but a similar behavior can be implemented by using a task pool [51].

We conclude that today's parallel architectures have some performance issues for applications with irregular memory access patterns, which the REPLICA architecture can solve.

5.2 Future Work

The benchmark suite can be expanded further so that more dwarfs are represented within it, and existing kernels can always use more optimizations. REPLICA could use a BLAS implementation of the dense matrix-matrix multiplication with lower complexity than $\mathcal{O}(n^3)$, for a more fair comparison against the OpenBLAS and CUBLAS implementations, preferably using floating point. Also one could implement a shared task queue for REPLICAs quicksort, so that idling threads can unburden threads under heavy workload. It would also be interesting to evaluate if there is a sorting algorithm that is better suited for the REPLICA architecture, such as merge sort.

Some of the benchmarks kernels have low thread utilization for small input sizes. It should be tested if it is possible to increase the efficiency of these kernels by using NUMA mode.

In REPLICA baseline, all the built-in variables are globals. Since global variables are never placed in registers, this can lead to unnecessary loss of performance, if these variables are used frequently [69]. It is of course possible for the programmer to copy any built-in variable into a local variable, but this has at least two draw backs. One is that the programmer then has to ensure the consistency of the local copy and the built-in variable. Secondly, the built-in routines will not benefit from local variables, since they only know about the built-in ones. It should be considered to make it possible for the compiler to store the built-in variables in registers.

In future the benchmark suite could be ported to the REPLICA language, and compared against the existing REPLICA baseline implementation.

Appendix

A

Code

```
#include <unistd.h>
#include <time.h>

struct timespec startTime, endTime;
double total_time = 0.0;

inline int getRealTime(struct timespec *ts)
{
    const clockid_t id = CLOCK_MONOTONIC;

    if(id != (clockid_t)-1 && clock_gettime(id, ts) != -1)
        return 0;
    return 1;
}

double elapsedTime(struct timespec *start, struct timespec *stop)
{
    return ((double)(stop->tv_sec - start->tv_sec)) +
           ((double)(stop->tv_nsec - start->tv_nsec) / (double)1e9);
}
```

Listing A.1: *timer.h*

```
#include "replica.h"

int source_[N];
int sum_;

int main()
{
    unsigned i;
    unsigned temp =_number_of_threads;
    sum_ = 0;

    // Kernel starts
    _start_timer;

    for (i=_thread_id; i<N; i+=temp)
    {
        _prefix(source_[i], ADD, &sum_, source_[i]);
    }

    // Kernel ends
    _stop_timer;
    _synchronize;

    _exit;
    return 0;
}
```

Listing A.2: *The PS kernel for REPLICA.*

```
#define NUMBER_OF_THREADS 1024

inline int prefix_sum_sequential(int* front, int* end)
{
    int sum = *front;
    ++front;
    while (front != end)
    {
        *front = sum = sum + *front;
        ++front;
    }
    return sum;
}

int main()
{
    const unsigned N = // Problem size
    int* front = // Points to the array
    int* end = front + N;

    // Initial work...

    // Kernel begins
    int sum[NUMBER_OF_THREADS];
    int chunk_size = (N+NUMBER_OF_THREADS-1)/NUMBER_OF_THREADS;

    #pragma omp parallel default(none) shared(front, end, sum, chunk_size)
    {
        int t_id = omp_get_thread_num();
        int offset = t_id * chunk_size;

        // Calculate chunk sizes
        int* t_front = (offset<N ? front+offset : end);
        int* t_end = (offset+chunk_size<N ? t_front+chunk_size : end);

        // Calculate prefix sum on local chunk
        sum[t_id] = (t_front==t_end ? 0 : prefix_sum_sequential(t_front,
            t_end) );

#       if NUMBER_OF_THREADS != 1
        #pragma omp barrier
        #pragma omp single
        {
            // Calculate prefix sum on sums
            prefix_sum_sequential(sum, sum + NUMBER_OF_THREADS);
        }

        if (t_id != 0)
        {
            int value = sum[t_id - 1];
            while (t_front != t_end)
                *t_front++ += value;
        }
#       endif
    }
    // Kernel ends

```

```
57      return 0;
58  }
```

__Listing A.3:__ The PS kernel for Xeon.

```
1  #include "replica.h"
2
3  int a_[N*N];
4  int b_[N*N];
5  int y_[N*N];
6
7  int main()
8  {
9      unsigned i;
10     unsigned NN = N*N;
11
12     // Kernel starts
13     _synchronize;
14     _start_timer;
15
16     // Calculate the subgroup sizes
17     // The subgroup sizes should be:
18     //  - ilog2(size)==0, since equal large subgroups is desired
19     //  - size<=N/4, since its loop are unrolled by four
20     unsigned max = N>>2 < _number_of_threads ? N>>2 : _number_of_threads;
21     unsigned subgroup_size;
22     for(subgroup_size=1; subgroup_size<=max; subgroup_size<<=1)
23         ;
24     subgroup_size>>=1;
25
26     // Calculate #subgroups
27     unsigned number_of_subgroups = _number_of_threads / subgroup_size;
28
29     // Calculate the group sizes
30     // The subgroup sizes should be:
31     //  - ilog2(size)==0, since equal large groups is desired
32     //  - size<=N, since its loop are NOT unrolled
33     max = N < number_of_subgroups ? N : number_of_subgroups ;
34     unsigned group_size;
35     for(group_size=1; group_size<=max; group_size<<=1)
36         ;
37     group_size>>=1;
38
39     // Calculate #groups
40     unsigned number_of_groups = number_of_subgroups / group_size;
41     // Unrolled accumulator variable, marked as volatile due to register
           allocation bug
42     volatile unsigned volatile_4_number_of_groups= 4*number_of_groups;
43
44     // Divide the threads into groups
45     unsigned group_id = _thread_id / (group_size*subgroup_size);
46     unsigned subgroup_id = (_thread_id-(group_id*group_size*subgroup_size)
           ) / subgroup_size;
47     _thread_id = _thread_id % subgroup_size;
48
49     // Compute Y = AB+Y
50     unsigned r,c,k;
51     // Unrolled by 4
52     for(k=_thread_id<<2; k<N-3; k+=subgroup_size<<2)
53     {
54         // Reuse row offsets
55         unsigned kN=k*N;
```

```
56          unsigned k1N=kN+N;
57          unsigned k2N=k1N+N;
58          unsigned k3N=k2N+N;
59
60          // Each row is calculated by one group
61          for(r=subgroup_id; r<N; r+=group_size)
62          {
63              // Reuse row offsets
64              unsigned rN=r*N;
65
66              // Reuse operands
67              int a=a_[rN+k];
68              int a1=a_[rN+k+1];
69              int a2=a_[rN+k+2];
70              int a3=a_[rN+k+3];
71
72              // Unrolled by 4
73              for(c=group_id*4; c<N-3; c+=volatile_4_number_of_groups)
74              {
75                  _multi(ADD,&y_[rN+c],a*b_[kN+c]+a1*b_[k1N+c]+a2*b_[k2N+c]+
                        a3*b_[k3N+c]);
76                  _multi(ADD,&y_[rN+c+1],a*b_[kN+c+1]+a1*b_[k1N+c+1]+a2*b_[
                        k2N+c+1]+a3*b_[k3N+c+1]);
77                  _multi(ADD,&y_[rN+c+2],a*b_[kN+c+2]+a1*b_[k1N+c+2]+a2*b_[
                        k2N+c+2]+a3*b_[k3N+c+2]);
78                  _multi(ADD,&y_[rN+c+3],a*b_[kN+c+3]+a1*b_[k1N+c+3]+a2*b_[
                        k2N+c+3]+a3*b_[k3N+c+3]);
79              }
80
81              // Take care of the rest if: (N/4)%number_of_groups!= 0
82              for(; c<N; ++c)
83              {
84                  _multi(ADD,&y_[rN+c],a*b_[kN+c]+a1*b_[k1N+c]+a2*b_[k2N+c]+
                        a3*b_[k3N+c]);
85              }
86              // Balance work among groups
87              group_id=(group_id+1)%number_of_groups;
88          }
89          // Balance work among subgroups
90          subgroup_id=(subgroup_id+1)%group_size;
91      }
92      // Balance work among threads
93      //_thread_id=(_thread_id+1)%group_size;
94
95      // Take care of the rest if: (N/4)%subgroup_size != 0
96      for(; k<N; ++k)
97      {
98          // Reuse row offset
99          unsigned kN=k*N;
100
101         for(r=subgroup_id; r<N; r+=group_size)
102         {
103             // Reuse row offset
104             unsigned rN=r*N;
105
106             // Reuse row operand
107             int a=a_[rN+k];
108
```

```
109             // Unrolled by 4
110             for(c=4*group_id; c<N-3; c+=4*number_of_groups)
111             {
112                 _multi(ADD,&y_[rN+c],a*b_[kN+c]);
113                 _multi(ADD,&y_[rN+c+1],a*b_[kN+c+1]);
114                 _multi(ADD,&y_[rN+c+2],a*b_[kN+c+2]);
115                 _multi(ADD,&y_[rN+c+3],a*b_[kN+c+3]);
116             }
117             // Take care of the rest if: (N/4)%number_of_groups!= 0
118             for(; c<N; ++c)
119             {
120                 _multi(ADD, &y_[rN+c], a*b_[kN+c]);
121             }
122         // Balance work among groups
123             group_id=(group_id+1)%number_of_groups;
124         }
125         // Balance work among subgroups
126         subgroup_id=(subgroup_id+1)%group_size;
127     }
128     // Balance work among threads
129     //_thread_id=(_thread_id+1)%group_size;
130 
131     // Restore built-in variables
132     _thread_id = _absolute_thread_id;
133     _number_of_threads = _absolute_number_of_threads;
134 
135     // Kernel ends
136     _synchronize;
137     _stop_timer;
138 
139     _exit;
140     return 0;
141 }
```

Listing A.4: *The DeMM kernel for REPLICA.*

```
#include "replica.h"

// Sparse matrix
int value_[NNZ];
unsigned col_index_[NNZ];
unsigned row_start_[ROWS+1];

// Vectors
int x_[ROWS];
int y_[ROWS];

unsigned counter_;

int main()
{
    // Kernel starts
    _start_timer;

    counter_ = _number_of_threads;
    unsigned row = _thread_id;
    while(row<ROWS)
    {
        int sum = 0;
        unsigned start = row_start_[row];
        unsigned end = row_start_[row+1];
        unsigned i;
        for(i=start; i<end; ++i)
        {
            sum += value_[i] * x_[col_index_[i]];
        }
        y_[row] = sum;
        _aprefix(row, ADD, &counter_, 1);
    }

    // Kernel ends
    _synchronize;
    _stop_timer;

    _exit;
    return 0;
}
```

Listing A.5: *The SpMV kernel for REPLICA.*

```
#include "replica.h"

// Sparse matrix
int row_[NNZ];
int col_[NNZ];
int val_[NNZ];

// Vectors
int x_[ROWS];
int y_[ROWS];

int main()
{
    unsigned temp = _number_of_threads;

    // Kernel starts
    _start_timer;

    unsigned i;
    for(i=_thread_id; i<NNZ; i+=temp)
    {
        _multi(ADD, &y_[row_[i]], val_[i] * x_[col_[i]]);
    }

    // Kernel ends
    _synchronize;
    _stop_timer;

    _exit;
    return 0;
}
```

Listing A.6: *The SpMV-COO kernel for REPLICA.*

```
#include "replica.h"

typedef struct Node_t
{
    int offset;
    int num_edges;
} Node;

Node node_[NODES];
int edge_[EDGES];
int cost_[NODES];
int visited_[NODES];

int queueA_[NODES];
int queueB_[NODES];

int group_size_[NUMBER_OF_THREADS];

int queue_end_;
int next_queue_end_;

int main()
{
    _start_timer;
    // Kernel starts

    int* queue = queueA_;
    int* next_queue = queueB_;

    // Put root in queue
    queue[0] = 0;
    queue_end_ = 1;
    cost_[0] = 0;
    visited_[0] = 1;

    int k = 1;
    do
    {
        // Reset next queue
        next_queue_end_ = 0;

        // Setup groups
        int group_id = _thread_id % queue_end_;
        group_size_[group_id] = 0;
        int thread_id;
        _aprefix(thread_id,ADD,&group_size_[group_id],1);

        // Spawn over queue (active nodes)
        for(int n=group_id; n<queue_end_; n+=_number_of_threads)
        {
            // Look up the visiting node in queue
            int node_id = queue[n];

            // Spawn (iterate) over its neighbors
            for(int v=node_[node_id].offset+thread_id; v<node_[node_id].
                offset+node_[node_id].num_edges; v+=group_size_[group_id
                ])
```

```
56              {
57                  int neighbor_id = edge_[v];
58
59                  // Check if neighbor is visited
60                  int is_visited;
61                  _aprefix(is_visited,ADD,&visited_[neighbor_id],1);
62                  if (!is_visited)
63                  {
64                      // if not put it in next queue and set cost
65                      int index;
66                      _aprefix(index,ADD,&next_queue_end_,1);
67                      next_queue[index] = neighbor_id;
68                      cost_[neighbor_id] = k;
69                  }
70              }
71          }
72
73          _synchronize;
74
75          // Update current queue
76          int* tmp = queue;
77          queue = next_queue;
78          next_queue = tmp;
79
80          queue_end_ = next_queue_end_;
81
82          ++k;
83      }
84      while(queue_end_);
85
86      // Kernel ends
87      _synchronize;
88      _stop_timer;
89
90      _exit;
91      return 0;
92  }
```

Listing A.7: *The BFS kernel for REPLICA.*

```
psBaseReg next_queue_end; // define as global

int* queue = queueA;
int* next_queue = queueB;

// Put root in queue
queue[0] = 0;
int queue_end = 1;
cost[0] = 0;
visited[0] = 1;

int k=1;
do
{
    // Reset next queue
    next_queue_end = 0;

    // Spawn over queue
    spawn(0, queue_end-1)
    {
        // Look up the visiting node in queue
        int id = queue[$];

        // Spawn (iterate) over its neighbors
        spawn(node[id][0], node[id][0]+node[id][1]-1)
        {
            int neighbor_id = edges[$];

            // Check if neighbor is visited
            int is_visited = 1;
            psm(is_visited, visited[neighbor_id]);
            if (!is_visited)
            {
                // if not put it in next queue and set cost
                int index = 1;
                ps(index, next_queue_end);
                next_queue[index] = neighbor_id;
                cost[neighbor_id] = k;
            }
        }
    }

    // Update current queue
    int* tmp = queue;
    queue = next_queue;
    next_queue = tmp;
    queue_end = next_queue_end;

    ++k;
}
while(queue_end);
```

Listing A.8: *The parallelized native implementation of BFS for Xeon.*

```
psBaseReg next_queue_end; // define as global

int* queue = queueA;
int* next_queue = queueB;

// Put root in queue
queue[0] = 0;
int queue_end = 1;
cost[0] = 0;
visited[0] = 1;

int k=1;
do
{
    // Reset next queue
    next_queue_end = 0;

    // Spawn over queue
    spawn(0, queue_end-1)
    {
        // Look up the visiting node in queue
        int id = queue[$];

        // Spawn (iterate) over its neighbors
        spawn(node[id][0], node[id][0]+node[id][1]-1)
        {
            int neighbor_id = edges[$];

            // Check if neighbor is visited
            int is_visited = 1;
            psm(is_visited, visited[neighbor_id]);
            if (!is_visited)
            {
                // if not put it in next queue and set cost
                int index = 1;
                ps(index, next_queue_end);
                next_queue[index] = neighbor_id;
                cost[neighbor_id] = k;
            }
        }
    }

    // Update current queue
    int* tmp = queue;
    queue = next_queue;
    next_queue = tmp;
    queue_end = next_queue_end;

    ++k;
}
while(queue_end);
```

Listing A.9: *The BFS kernel for XMT.*

```
1  sh int queue_space[NUM_NODES];
2  sh int queue_size;
3  sh int next_queue_size;
4
5  pr int* queue = queue_space;
6  pr int* next_queue = &queue_space[1];
7
8  // kernel begins
9
10 // Put root in queue
11 queue[0] = 0;
12 queue_size = 1;
13 cost[0] = 0;
14 visited[0] = 1;
15 int k=1;
16
17 do
18 {
19     // Reset next queue
20     next_queue_size = 0;
21
22     // Setup groups
23     fork(queue_size; @=$%queue_size;)
24     {
25         // Spawn over queue (active nodes)
26         int n;
27         int group_size = #; // # can change value in asynchronous mode
28         farm for(n=@; n<queue_size; n+=__STARTED_PROCS__)
29         {
30             // Look up the visiting node in queue
31             int node_id = queue[n];
32
33             // Spawn (iterate) over its neighbors
34             int v;
35             for(v=nodes[node_id].offset+$$; v<nodes[node_id].offset+nodes[
                   node_id].num_edges; v+=group_size)
36             {
37                 int neighbor_id = edges[v];
38
39                 // Check if neighbor is visited
40                 if(!mpmax(&visited[neighbor_id], 1))
41                 {
42                     // if not put it in next queue and set cost
43                     int index = mpadd(&next_queue_size, 1);
44                     next_queue[index] = neighbor_id;
45                     cost[neighbor_id] = k;
46                 }
47             }
48         }
49     }
50
51     // Update current queue
52     queue = next_queue;
53     next_queue = &next_queue[next_queue_size];
54     queue_size = next_queue_size;
55     ++k;
56 }
```

```
57 while(queue_size);
58 // kernel ends
```

Listing A.10: *The BFS kernel for SB-PRAM.*

```
1  diff --git a/quick2.c.org b/quick2.c.new
2  index 4f66ceb..1c34adc 100644
3  --- a/quick2.c.org
4  +++ b/quick2.c.new
5  @@ -2,21 +2,30 @@
6  
7   /* quick.c
8    */
9  -#include <xmt.h>
10 -#include <stdio.h>
11 +#include <xmtc.h>
12 +#include "data.h"
13 
14 -#define PROBLEM_SIZE  131072
15  #define SORT_THRESHOLD 2
16  #define FORK_THRESHOLD 1
17 -#define PHASE_SWITCH_THRESHOLD 0
18 +
19 +#if PROBLEM_SIZE == 10000
20 +#    define PHASE_SWITCH_THRESHOLD 64
21 +
22 +#elif PROBLEM_SIZE >= 100000
23 +#    define PHASE_SWITCH_THRESHOLD 1024
24 +
25 +#endif
26 
27  #define TWO_ELEMENTS   -3
28  #define ONE_ELEMENT   -2
29  #define ZERO_ELEMENTS  -1
30 +psBaseReg global_count;
31 +psBaseReg low;
32 +psBaseReg hi;
33 
34  int do_quicksort(int *input, int start, int end, int *scratch)
35  {
36 -  int hi, low, pivot, pivotval,temp;
37 +  int pivot, pivotval,temp;
38    int i;
39 
40    if( end - start < 0) {
41 @@ -53,24 +62,25 @@ int do_quicksort(int *input, int start, int end, int *
       scratch)
42    low = 0;
43    hi = 1;
44 
45 -  spawn(end - start + 1, start);
46 +  spawn(start, end)
47    {
48 -    int TID;
49 +    int TID=$;
50      int index;
51      int curval = input[TID];
52 
53      if(TID != (start + pivot)){
54        if(curval <= pivotval){
55 -        index = ps(&low,1);
56 +        index = 1;
```

```
57  +    ps(index,low);
58          scratch[index + start] = curval;
59        }
60        else{
61  -       index = ps(&hi,1);
62  +       index = 1;
63  +    ps(index,hi);
64          scratch[end + 1 - index] = curval;
65        }
66      }
67        }
68  -  join();
69
70     scratch[start + low] = pivotval;
71
72  @@ -84,14 +94,11 @@ int do_quicksort(int *input, int start, int end, int *
        scratch)
73   }
74
75
76  -void main(int argc, char *argv[])
77  +int main()
78   {
79     int start[PROBLEM_SIZE*2], end[PROBLEM_SIZE*2];
80  -  int global_count = 0;
81  +  global_count = 0;
82
83  - int input[PROBLEM_SIZE] =
84  -#include "input.65536.i"
85  -;
86
87     int scratch[PROBLEM_SIZE];
88     int i;
89  @@ -105,22 +112,20 @@ void main(int argc, char *argv[])
90     end[0] = PROBLEM_SIZE - 1;
91     start[0] = 0;
92
93  -  spawn(PROBLEM_SIZE*2-1, 1);
94  +  spawn(1,PROBLEM_SIZE*2-1)
95     {
96  -    int TID;
97  +    int TID=$;
98       start[TID] = -1;
99       end[TID] = -1;
100    }
101 -  join();
102
103
104    input_p = &input[0];
105    output_p = &scratch[0];
106
107
108 -  begin();
109    /* phase 1 */
110 -  while (number_of_partitions < PHASE_SWITCH_THRESHOLD)
111 +  while (input_p != input || number_of_partitions <
        PHASE_SWITCH_THRESHOLD)
112      {
```

```
113        if(number_of_partitions == 0)
114         break;
115 @@ -150,15 +155,18 @@ void main(int argc, char *argv[])
116              {
117                if(n > 0)
118                  {
119 -                   start[++global_count] = start[i];
120 +             global_count += 1;
121 +                   start[global_count] = start[i];
122                     end[global_count] = start[i]+n-1;
123                  }
124                else
125                  {
126 -                   start[++global_count] = start[i];
127 +             global_count += 1;
128 +                   start[global_count] = start[i];
129                     end[global_count] = start[i];
130                   }
131 -              start[++global_count] = start[i]+n+1;
132 +             global_count += 1;
133 +              start[global_count] = start[i]+n+1;
134               end[global_count] = end[i];
135              }
136         }
137 @@ -192,19 +200,20 @@ void main(int argc, char *argv[])
138    global_count = last;
139
140    /* phase 2 */
141 - fspawn(number_of_partitions, first);
142 + spawn(first, last-1)
143    {
144 -   int TID;
145 +   int newTID;
146 +   int TID=$;
147      int pivot, pivotval;
148 -   int my_end, my_start;
149 +   int my_end=0;
150 +   int my_start=0;
151      int low, hi, temp;
152
153 -   print_fork_spin_start();
154 -   while ((my_end = end[TID]) < 0 ||
155 -          (my_start = start[TID]) < 0)
156 -     refresh();
157 -   ;
158 -   print_fork_spin_end();
159 +   do{
160 +       psm(my_end, end[TID]);
161 +       psm(my_start, start[TID]);
162 +   }
163 +   while (my_start < 0 || my_end < 0);
164
165      while(my_end - my_start + 1 > SORT_THRESHOLD)
166        {
167 @@ -241,10 +250,11 @@ void main(int argc, char *argv[])
168         /* continue with low-partition, fork for hi-partition */
169         if(my_end - low + 1 > FORK_THRESHOLD)
170           {
```

```
171 -           int next_index = ps(&global_count, 1);
172 +        int next_index = 1;
173 +        ps(next_index, global_count);
174             start[next_index] = low;
175             end[next_index] = my_end;
176 -           xfork();
177 +        sspawn(newTID){}
178           }
179 
180         //my_end = hi + 1;
181 @@ -259,8 +269,6 @@ void main(int argc, char *argv[])
182         }
183       }
184     }
185 -   join();
186 -   stop();
187 
188     /*
189       for(i=0; i<PROBLEM_SIZE; i++){
```

Listing A.11: *The diff of quick2.h for XMT.*

```
#define SWAP_P(a, b) {int *p=(a); (a)=(b); (b)=p;}
#define SWAP(a, b) {int p=(a); (a)=(b); (b)=p;}

struct stack_t{
    int lo;
    int hi;
};

int a_[SIZE];
int t_[SIZE];

struct stack_t stack[128];

void par_quicksort(int lo, int hi)
{
    int *input;
    int *output;
    int pivot;
    int j;
    int *last_less_p;    // Shared value to index less-than-pivot-elements
         in temp array
    int *first_greater_p;// Shared value to index greater-than-pivot-
        elements in temp array
    int last_less;
    int first_greater;

    // Return if #elements <= 1 than
    if (lo >= hi)
        return;

    input = a_;
    output = t_;

    // The first phase
par_quicksort_start:

    // Trivial sequential sort if #elements == 2
    if (lo == hi-1)
    {
        if (input[lo] > input[hi])
        {
            pivot = input[lo];
            a_[lo] = input[hi];
            a_[hi] = pivot;
        }
        else if(input != a_)
        {
            a_[lo]=input[lo];
            a_[hi]=input[hi];
        }
        return;
    }

    // If the group only consist of a single thread or if the shared space
         is to small, than switch to next phase.
    if(_number_of_threads == 1 || (unsigned)_shared_stack-(unsigned)
        _shared_heap < 16)
```

```
54      {
55          // Include the second phase
56  #       include "inline_par_quicksort.c"
57          return;
58      }
59
60      // Choose median of three elements as pivot
61      pivot = input[(hi+lo)>>1];
62      last_less = input[lo];
63      first_greater = input[hi];
64      if (pivot < last_less)
65          SWAP(pivot, last_less)
66      if (first_greater < pivot)
67      {
68          pivot = first_greater;
69          if (pivot < last_less)
70              pivot = last_less;
71      }
72
73      // Set shared addresses for prefix sum operation
74      last_less_p = (int*)(_shared_stack - 4);
75      first_greater_p = (int*)(_shared_stack - 8);
76
77      // Set start index in temp array for less-than-elements
78      *last_less_p = lo;
79      // Set start index in temp array for greater-than-elements
80      *first_greater_p = hi;
81
82      // Partitioning in parallel using prefix sum operation
83      for (j=_thread_id+lo; j<=hi; j+=_number_of_threads)
84      {
85          int index;
86          int value = input[j];
87          if (value < pivot)
88          {
89              _aprefix(index, ADD, last_less_p, 1);
90              output[index] = value;
91          }
92          else if (value > pivot)
93          {
94              _aprefix(index, ADD, first_greater_p, -1);
95              output[index] = value;
96          }
97          // else equal to pivot
98      }
99      _synchronize;
100
101     // The last less-than-number
102     last_less = (*last_less_p) - 1;
103     // The first greater-than-number
104     first_greater = (*first_greater_p) + 1;
105
106     // Write pivots to original array
107     // If there only is one pivot element we want to skip the
            synchronization.
108     if(first_greater - last_less - 1 > 1)
109     {
110         for (j=_thread_id+last_less+1; j<first_greater; j+=
```

```
                _number_of_threads)
111             a_[j] = pivot;
112         _synchronize;
113     }
114     else
115         a_[last_less+1] = pivot;
116 
117     // Partitioning is done
118 
119     // Both arrays are greater than 1 element (the usual case)
120     if (last_less-lo > 0 && hi-first_greater > 0)
121     {
122         // Divide the group based on the two partitions sizes
123         // Check at compile time if overflow might occur.
124         // If it might, than lower the precision for the group sizes
                computation
125 #       if (SIZE-1)*(512*NUMBER_OF_PROCESSORS-1) < 0xFFFFFFFF
126             int number_of_threads_for_less = (int)(((unsigned)(last_less-
                    lo+1)*(unsigned)_number_of_threads)/(unsigned)(last_less-
                    lo+1+hi-first_greater+1));
127 #       else
128             int number_of_threads_for_less = (int)((unsigned)
                    _number_of_threads*(((unsigned)(last_less-lo+1)*(unsigned
                    )4000)/(unsigned)(last_less-lo+1+hi-first_greater+1)))
                    /4000;
129 #       endif
130 
131         // Correction of the new group size
132         if (!number_of_threads_for_less)
133             number_of_threads_for_less = 1;
134         if (number_of_threads_for_less == _number_of_threads)
135             number_of_threads_for_less -= 1;
136 
137         if (_thread_id < number_of_threads_for_less)
138         {
139             // Set shared space for new group
140             _shared_stack = (_shared_heap + _shared_stack) >> 1;
141             _shared_stack &= 0xfffffffc;
142             _shared_stack -= 4;
143 
144             // Set group variables
145             _group_id = _shared_stack;
146             _number_of_threads = number_of_threads_for_less;
147             *((int*)_group_id) = number_of_threads_for_less;
148 
149             // Set sort data and start sorting
150             hi = last_less;
151             SWAP_P(input,output)
152             goto par_quicksort_start;
153         }
154         else
155         {
156             // Set shared space for new group
157             _shared_heap = (_shared_heap + _shared_stack) >> 1;
158             _shared_heap &= 0xfffffffc;
159 
160             // Set group variables
161             _group_id = _shared_stack;
```

```
162                 _number_of_threads -= number_of_threads_for_less;
163                 _thread_id -= number_of_threads_for_less;
164                 *((int*)_group_id) = _number_of_threads;
165 
166                 // Set sort data and start sorting
167                 lo = first_greater;
168                 SWAP_P(input,output)
169                 goto par_quicksort_start;
170             }
171         }
172         // Only left side is greater than 1 element
173         else if (last_less - lo > 0)
174         {
175             // Store the right side's element (if exist) in original array
176             if (hi == first_greater)
177                 a_[hi] = output[hi];
178 
179             // Set sort data and start sorting
180             hi = last_less;
181             SWAP_P(input,output)
182             goto par_quicksort_start;
183         }
184         // Only right side is greater than 1 element
185         else if ((hi - first_greater) > 0)
186         {
187             // Store the left side's element (if exist) in original array
188             if (lo == last_less)
189                 a_[lo] = output[lo];
190 
191             // Set sort data and start sorting
192             lo = first_greater;
193             SWAP_P(input,output)
194             goto par_quicksort_start;
195         }
196         // Both left and right side are less than 2 element
197         else
198         {
199             // Store the left and right side's element (if exist) in original
                    array
200             if (lo == last_less)
201                 a_[lo] = output[lo];
202             if (hi == first_greater)
203                 a_[hi] = output[hi];
204         }
205 }
206 
207 int main()
208 {
209     // Kernel starts
210     _start_timer;
211 
212     // Save group and shared data variables
213     int _old_thread_id = _thread_id;
214     int _old_number_of_threads = _number_of_threads;
215     int _old_group_id = _group_id;
216     int _old_shared_stack = _shared_stack;
217     int _old_shared_heap = _shared_heap;
218     _shared_stack -= 4;
```

```
219
220     par_quicksort(0, SIZE-1);
221
222     // Restore group and shared data variables
223     _thread_id = _old_thread_id;
224     _number_of_threads = _old_number_of_threads;
225     _group_id = _old_group_id;
226     _shared_stack = _old_shared_stack;
227     _shared_heap = _old_shared_heap;
228
229     // Kernel ends
230     _stop_timer;
231     _synchronize;
232
233     _exit;
234     return 0;
235 }
```

Listing A.12: *The QS kernel for REPLICA.*

```
1   {
2       int top=0;
3       int i;
4       // If the input data is not in the original array it has to copied
            back when partitioning
5       if(input !=  a_)
6       {
7           // Choose median of three elements as pivot
8           pivot = (hi+lo)>>1;
9           if (input[lo] < input[pivot])
10              SWAP(input[pivot], input[lo])
11          if (input[hi] < input[lo])
12          {
13              SWAP(input[lo],input[hi])
14              if (input[lo] < input[pivot])
15                  SWAP(input[pivot],input[lo])
16          }
17          pivot = input[lo];
18          last_less = lo-1;
19          first_greater = hi+1;
20
21          // Partitioning can not be done in parallel since the group has
                run out of shared space
22          for(j=lo+1; j<=hi; ++j)
23          {
24              int value = input[j];
25              if (value <= pivot)
26                  a_[++last_less] = value;
27              else if (value > pivot)
28                  a_[--first_greater] = value;
29          }
30          i = last_less+1;
31          a_[i] = pivot;
32
33          // Fork if group size is > 2, otherwise keep sorting sequential
34          if(_number_of_threads > 1)
35              goto fork_group;// Goto the second phase
36          else
37              goto seq_split;// Goto the third phase
38      }
39
40      // Second phase
41      // Sort until done or the group consist of one thread
42      while(lo < hi && _number_of_threads > 1)
43      {
44          // Trivial sequential sort if #elements <= 3
45          if (hi-lo <= 2)
46          {
47              if (a_[lo+1] < a_[lo])
48                  SWAP(a_[lo+1],a_[lo])
49              if (a_[hi] < a_[lo+1]) // OBS hi and lo+1 can be the same
                    element
50              {
51                  SWAP(a_[hi],a_[lo+1])
52                  if (a_[lo+1] < a_[lo])
53                      SWAP(a_[lo+1],a_[lo])
54              }
```

```
55                return;
56            }
57
58            // Choose median of three elements as pivot
59            pivot = a_[hi];
60            if (pivot < a_[lo])
61                SWAP(pivot, a_[lo])
62            if (a_[hi-1] < pivot)
63            {
64                SWAP(a_[hi-1],pivot)
65                if (pivot < a_[lo])
66                    SWAP(pivot,a_[lo])
67            }
68            a_[hi]=pivot;
69
70            i = lo;
71            j = hi-1;
72
73            // Sequential partitioning
74            while (i < j)
75            {
76                while (i < hi && a_[i] <= pivot)
77                    ++i;
78                while (j > lo && a_[j] >= pivot)
79                    --j;
80                if (i < j)
81                    SWAP(a_[i],a_[j])
82            }
83            if (a_[i] > pivot)
84                SWAP(a_[i],a_[hi])
85
86    fork_group:
87            // Both arrays are greater than 1 element (the usual case)
88            if(lo < i-1 && i+1 < hi)
89            {
90                // Divide the group based on the two partitions sizes
91                int number_of_threads_for_less = ((i-lo)*_number_of_threads)/(
                      hi-lo);
92
93                // Correction of the new group size
94                if (!number_of_threads_for_less)
95                    number_of_threads_for_less = 1;
96                if (number_of_threads_for_less == _number_of_threads)
97                    number_of_threads_for_less = _number_of_threads-1;
98
99                // The group split
100               if (_thread_id < number_of_threads_for_less)
101               {
102                   // Set new input data and group size
103                   hi=i-1;
104                   _number_of_threads = number_of_threads_for_less;
105               }
106               else
107               {
108                   // Set new input data, thread id and group size
109                   lo=i+1;
110                   _number_of_threads -= number_of_threads_for_less;
111                   _thread_id -= number_of_threads_for_less;
```

```
112             }
113         }
114         else
115         {
116             if (lo < i-1)
117             {
118                 hi=i-1;
119             }
120             else if (i+1 < hi)
121             {
122                 lo=i+1;
123             }
124             else
125                 return;
126         }
127     }
128 
129     // If not done, start sorting sequential (without forking)
130     if(lo < hi)
131     {
132         // Include the third phase
133 #       include "inline_seq_quicksort.c"
134     }
135     return;
136 }
```

Listing A.13: *inline_par_quicksort.c included in Listing A.12.*

```
1  // The third phase
2  {
3      // Push array into an explicit stack
4      stack[top].lo=lo;
5      stack[top++].hi=hi;
6
7      // Run until the stack is empty
8      while(0 < top)
9      {
10         // Pop array from stack
11         hi=stack[--top].hi;
12         lo=stack[top].lo;
13 seq_quicksort_start:
14
15         // Trivial sequential sort if #elements <= 3
16         if (hi-lo <= 2)
17         {
18             if (a_[lo+1] < a_[lo])
19                 SWAP(a_[lo+1],a_[lo])
20             if (a_[hi] < a_[lo+1]) // OBS hi and lo+1 can be the same
                   element
21             {
22                 SWAP(a_[hi],a_[lo+1])
23                 if (a_[lo+1] < a_[lo])
24                     SWAP(a_[lo+1],a_[lo])
25             }
26             continue;
27         }
28
29         // Choose median of three elements as pivot
30         pivot = a_[hi];
31         if (pivot < a_[lo])
32             SWAP(pivot, a_[lo])
33         if (a_[hi-1] < pivot)
34         {
35             SWAP(a_[hi-1],pivot)
36             if (pivot < a_[lo])
37                 SWAP(pivot,a_[lo])
38         }
39         a_[hi]=pivot;
40
41         i = lo;
42         j = hi-1;
43
44         // Sequential partitioning
45         while (i < j)
46         {
47             while (i < hi && a_[i] <= pivot)
48                 ++i;
49             while (j > lo && a_[j] >= pivot)
50                 --j;
51             if (i < j)
52                 SWAP(a_[i],a_[j])
53         }
54         if (a_[i] > pivot)
55             SWAP(a_[i],a_[hi])
56
```

```
57 seq_split:
58         // Put the smallest array on the stack, and continue with the
               largest
59         if(i-1-lo < hi-i+1)
60         {
61             if (lo < i-1)
62             {
63                 stack[top].lo=lo;
64                 stack[top++].hi=i-1;
65             }
66             if (i+1 < hi)
67             {
68                 lo=i+1;
69                 goto seq_quicksort_start;
70             }
71         }
72         else
73         {
74             if (i+1 < hi)
75             {
76                 stack[top].lo=i+1;
77                 stack[top++].hi=hi;
78             }
79             if (lo < i-1)
80             {
81                 hi=i-1;
82                 goto seq_quicksort_start;
83             }
84         }
85     }
86 }
```

Listing A.14: *inline_seq_quicksort.c included in Listing A.12.*

B

Results

	REPLICA configurations								
Elements	T5-4	T7-4	T11-4	T5-16	T7-16	T11-16	T5-64	T7-64	T11-64
262144	395521	329857	329857	99903	83514	83514	29765		26071
524288	790017	658689	658689	198783	166003	166001	58745		50607
1048576	1579009	1316353	1316353	396543	330964	330963	116305		100227
2097152	3156993	2631681	2631681	792063	660905	660903	231832		199318
4194304	6312961	5262337	5262337	1583103	1320787	1320783	464003		397376

Table B.1: *REPLICA's results in cc for PS.*

	REPLICA configurations								
Rows	T5-4	T7-4	T11-4	T5-16	T7-16	T11-16	T5-64	T7-64	T11-64
128	2055102	956310	886542	540667	490715	455251	168003		461087
256	16135923	7425183	6889611	4062479	1880174	1744742	1049816		895454
384	72399831	33276399	30885819	18148690	8361674	7762490	6122514		2626670
512	128583439	59088355	54850975	32368708	14990503	13929619	8113077		3496932
1024		471950046	438196698		118334343	109894467			28155599

Table B.2: *REPLICA's results in cc for DeMM.*

	REPLICA configurations								
Matrix	T5-4	T7-4	T11-4	T5-16	T7-16	T11-16	T5-64	T7-64	T11-64
Internet	1031365	540477	540473	684370	351501	351190	598757		306244
Lugn2	1236476	650564	650591	333969	178558	178630	107103		59799
ASIC_680ks	6880682	3616393	3616379	2374589	1233103	1233086	1243215		640386
t2em	11072300	5776370	5776446	2789857	1465600	1465600	721821		431405

Table B.3: *REPLICA's results in cc for SpMV.*

	REPLICA configurations								
Matrix	T5-4	T7-4	T11-4	T5-16	T7-16	T11-16	T5-64	T7-64	T11-64
Internet	582260	320628	320628	153336	86664	86660	47738		29753
Lugn2	1367844	749164	749164	351884	194952	194952	98437		57388
ASIC_680ks	6433635	3514663	3514663	1619479	888113	888111	419765		234845
t2em	12668978	6918246	6918246	3181329	1742324	1742325	815785		453301

Table B.4: *REPLICA's results in cc for SpMV-COO.*

	REPLICA configurations								
Graph	T5-4	T7-4	T11-4	T5-16	T7-16	T11-16	T5-64	T7-64	T11-64
graph4096	379138	254910	254910	223369	165189	165189	210966		160603
graph65536	1984849	1300562	1300562	903654	585065	585065	489006		335184
graph1MW_6	20999696	12809253	12809253	6024894	3762253	3762253	2026131		1347884

Table B.5: *REPLICA's results in cc for BFS.*

	REPLICA configurations								
Elements	T5-4	T7-4	T11-4	T5-16	T7-16	T11-16	T5-64	T7-64	T11-64
10000	1419152			1124248			1030453		
100000	10312303			3398471			2211043		
1000000	114164612						10627652		

Table B.6: *REPLICA's results in cc for QS.*

Elements	XMT (cc)	SB-PRAM (cc)
262144	38571	3176448
524288	71675	6322176
1048576	134729	12613632
2097152	561833	25196544
4194304	1383573	50362368

Table B.7: *XMT and SB-PRAM's results for PS.*

Rows	XMT (cc)	SB-PRAM (cc)
128	146992	11299840
256	1109574	87060480
384	3625069	307842048
512	8720495	683700224
1024	97328445	5419069440

Table B.8: *XMT and SB-PRAM's results for DeMM.*

Matrix	XMT (cc)	SB-PRAM (cc)
Internet	88221	7005184
Lugn2	123625	8482816
ASIC_680ks	699705	47050752
t2em	1049901	74182656

Table B.9: *XMT and SB-PRAM's results for SpMV.*

Graph	XMT (cc)	SB-PRAM (cc)
graph4096	18216	4654080
graph65536	85769	17054720
graph1MW_6	4653889	154180608

Table B.10: *XMT and SB-PRAM's results for BFS.*

Elements	XMT (cc)	SB-PRAM (cc)
10000	374496	
100000	2900783	
1000000	13958950	

Table B.11: *XMT and SB-PRAM's results for QS.*

Elements	Xeon (sec)	Tesla transfer time (sec)	Tesla kernel time (sec)
262144	7.54E-05	8.44E-04	3.15E-04
524288	1.56E-04	1.39E-03	3.63E-04
1048576	3.28E-04	2.46E-03	4.55E-04
2097152	6.78E-04	4.61E-03	6.45E-04
4194304	2.26E-03	8.85E-03	1.04E-03

Table B.12: *XMT and SB-PRAM's results for PS.*

Rows	Xeon (sec)	Tesla transfer time (sec)	Tesla kernel time (sec)
128	5.52E-04	3.31E-01	5.24E-05
256	3.32E-03	2.83E-01	2.12E-04
384	9.98E-03	8.08E-01	6.63E-04
512	2.76E-02	1.72E+00	1.50E-03
1024	2.48E-01	1.05E+01	1.20E-02

Table B.13: *Xeon and Tesla's results for DeMM.*

Rows	Xeon (sec)	Tesla transfer time (sec)	Tesla kernel time (sec)
128	5.27E-05	3.31E-01	4.37E-05
256	3.23E-04	2.83E-01	1.56E-04
384	9.04E-04	8.08E-01	2.54E-04
512	2.23E-03	1.72E+00	6.37E-04
1024	1.73E-02	1.05E+01	3.32E-03

Table B.14: *Xeon and Tesla's results for DeMM-BLAS.*

Matrix	Xeon (sec)	Tesla transfer time (sec)	Tesla kernel time (sec)
Internet	2.33E-04	8.74E-04	1.32E-04
Lugn2	3.19E-04	1.61E-03	1.30E-04
ASIC_680ks	1.88E-03	6.81E-03	7.87E-04
t2em	3.26E-03	1.14E-02	8.78E-04

Table B.15: *Xeon and Tesla's results for SpMV.*

Graph	Xeon (sec)	Tesla transfer time (sec)	Tesla kernel time (sec)
graph4096	1.47E-04	1.15E-04	2.47E-04
graph65536	1.96E-03	1.10E-03	9.46E-04
graph1MW_6	3.59E-02	1.13E-02	1.49E-02

Table B.16: *Xeon and Tesla's results for BFS.*

Elements	Xeon (sec)	Tesla transfer time (sec)	Tesla kernel time (sec)
10000	1.64E-04	4.67E-05	6.40E-04
100000	1.43E-03	3.44E-04	1.72E-03
1000000	1.48E-02	9.34E-04	1.41E-02

Table B.17: *Xeon and Tesla's results for QS.*

Elements	REPLICA-4 (MHz)	REPLICA-16 (MHz)	REPLICA-64 (MHz)
262144	4374.2	1107.5	345.7
524288	4228.7	1065.7	324.9
1048576	4018.5	1010.4	306.0
2097152	3882.1	974.9	294.0
4194304	2333.2	585.6	176.2

Table B.18: *Needed frequency in MHz against Xeon for PS kernel.*

Elements	REPLICA-4 (MHz)	REPLICA-16 (MHz)	REPLICA-64 (MHz)
262144	1046.2	264.9	82.7
524288	1814.8	457.4	139.4
1048576	2893.9	727.6	220.3
2097152	4079.9	1024.6	309.0
4194304	5044.9	1266.2	381.0

Table B.19: *Needed frequency in MHz against Tesla for PS kernel excluding transfer time.*

Elements	REPLICA-4 (MHz)	REPLICA-16 (MHz)	REPLICA-64
262144	284.5	72.0	22.5
524288	376.8	95.0	28.9
1048576	450.9	113.4	34.3
2097152	501.2	125.9	38.0
4194304	531.7	133.5	40.2

Table B.20: *Needed frequency in MHz against Tesla for PS kernel including transfer time.*

Bibliography

[1] The University of Florida sparse matrix collection. URL http://www.cise.ufl.edu/research/sparse/matrices/. Fetched 2013-05-15. Cited on pages 51 and 87.

[2] Using the GNU compiler collection (GCC). URL http://gcc.gnu.org/onlinedocs/gcc/. Fetched 2013-08-22. Cited on page 54.

[3] LLVM. URL http://llvm.org. Fetched 2013-05-12. Cited on page 18.

[4] NAS parallel benchmarks changes, . URL http://www.nas.nasa.gov/publications/npb_changes.html. Fetched 2013-04-03. Cited on page 7.

[5] NAS parallel benchmarks, . URL http://www.nas.nasa.gov/publications/npb.html. Fetched 2013-04-03. Cited on page 7.

[6] OpenMP. URL http://www.openmp.org/. Fetched 2013-10-30. Cited on page 19.

[7] The SPEC Organization. URL http://www.spec.org/spec/. Fetched 2013-10-30. Cited on page 8.

[8] Threading Building Blocks version 4.2 update 2. URL http://www.threadingbuildingblocks.org. Fetched 2013-08-14. Cited on pages 66 and 70.

[9] The GNU C Library, . URL http://www.gnu.org/software/libc/. Fetched 2013-06-13. Cited on pages 36 and 64.

[10] The GNU C++ library, . URL http://gcc.gnu.org/libstdc++/. Fetched 2013-08-14. Cited on page 66.

[11] Rodinia: Accelerating compute-intensive applications with accelerators. URL http://www.cs.virginia.edu/~skadron/wiki/rodinia/. Fetched 2013-10-30. Cited on pages 8, 57, 60, 62, 70, and 87.

[12] Whitepaper: NVIDIA's next generation CUDA compute architecture: Fermi, 2009. URL http://www.nvidia.com/content/PDF/fermi_white_

papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Cited on pages 27 and 28.

[13] TESLA M2050 / M2070 GPU computing module supercomputing at 1/10th the cost, 2010. URL http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_M2050_M2070_Apr10_LowRes.pdf. Cited on pages 27 and 28.

[14] Intel Xeon Processor 5600 Series Product Brief, 2010. URL http://www.intel.com/content/www/us/en/processors/xeon/xeon-5600-brief.html. Cited on page 18.

[15] Intel Xeon processor 5600 series datasheet, vol. 1, 2011. URL http://www.intel.com/content/www/us/en/processors/xeon/xeon-5600-vol-1-datasheet.html. Cited on page 18.

[16] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, 1995. Cited on page 6.

[17] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9. doi: 10.1109/SC.2010.46. URL http://dx.doi.org/10.1109/SC.2010.46. Cited on page 58.

[18] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html. Cited on page 6.

[19] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. 1994. URL http://www.nas.nasa.gov/assets/pdf/techreports/1994/rnr-94-007.pdf. Technical Report RNR-94-007. Cited on page 7.

[20] D.H. Bailey, T. Harris, W.C. Saphir, R.F. Van der Wijngaart, A.C. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. 1995. URL http://www.nas.nasa.gov/assets/pdf/techreports/1995/nas-95-020.pdf. Report NAS-95-020. Cited on page 7.

[21] A. O. Balkan, U. Vishkin, G. C. Caragea, and A. Tzannes. *XMT toolchain manual for XMTC language, XMTC compiler, XMT simulator and paraleap XMT FPGA computer, version 0,82.1*, 2010. URL http://www.umiacs.

umd.edu/users/vishkin/XMT/manual4xmtc1out-of2.pdf. Cited on pages 25, 26, 27, 36, 48, and 88.

[22] A.O. Balkan, G. Qu, and U. Vishkin. A mesh-of-trees interconnection network for single-chip parallel processing. In *Application-specific Systems, Architectures and Processors. International Conference on*, pages 73–80, 2006. doi: 10.1109/ASAP.2006.6. Cited on page 24.

[23] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the solution of linear systems: Building blocks for iterative methods, 2nd edition*. SIAM, Philadelphia, PA, 1994. Cited on pages 51, 52, and 53.

[24] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990. Cited on page 39.

[25] J.H. Byun, R. Lin, K. A. Yelick, and J. Demmel. Autotuning sparse matrix-vector multiplication for multicore. Technical Report UCB/EECS-2012-215, EECS Department, University of California, Berkeley, 2012. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-215.html. Cited on pages 51 and 53.

[26] J.H. Byun, R. Lin, K. A. Yelick, and J. Demmel. pOSKI: Parallel optimized sparse kernel interface library user's guide for version 1.0.0. Technical report, 2012. URL http://bebop.cs.berkeley.edu/poski. Cited on page 54.

[27] G.C. Caragea, F. Keceli, A. Tzannes, and U. Vishkin. General-purpose vs. GPU: Comparison of many-cores on irregular workloads. In *Proceedings of the Second Usenix Workshop on Hot Topics in Parallelism*. Usenix, 2010. Cited on page 6.

[28] D. Cederman and P. Tsigas. GPU-Quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:4:1.4–4:1.24, 2010. ISSN 1084-6654. doi: 10.1145/1498698.1564500. URL http://doi.acm.org/10.1145/1498698.1564500. Cited on pages 68, 69, and 70.

[29] S. Dalton and N. Bell. CUSP version 0.3.1. URL http://cusplibrary.github.io/. Fetched 2013-07-27. Cited on pages 55, 56, and 70.

[30] U. Drepper. What every programmer should know about memory, 2007. URL http://people.redhat.com/drepper/cpumemory.pdf. Cited on page 45.

[31] J. A. Edwards and U. Vishkin. Brief announcement: speedups for parallel graph triconnectivity. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 190–192, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1213-4. doi: 10.1145/2312005.2312042. URL http://doi.acm.org/10.1145/2312005.2312042. Cited on pages 36, 40, and 70.

[32] M. Forsell. Removing performance and programmability limitations of chip multiprocessor architectures. URL http://www.vtt.fi/sites/replica/index.jsp?lang=en. Fetched 2013-03-15. Cited on pages 9, 10, and 14.

[33] M. Forsell. E - A language for thread-level parallel programming on synchronous shared memory NOCs. *WSEAS Transactions on Computers*, 3(3): 807 – 812, 2004. ISSN 1109-2750. Cited on page 14.

[34] M. Forsell. Step caches - a novel approach to concurrent memory access on shared memory MP-SOCs. In *23rd NORCHIP Conference*, pages 74–77, 2005. doi: 10.1109/NORCHP.2005.1596992. Cited on page 13.

[35] M. Forsell. Realizing multioperations for step cached MP-SOCs. In *System-on-Chip, 2006. International Symposium on*, pages 1–6, 2006. doi: 10.1109/ISSOC.2006.321972. Cited on page 13.

[36] M. Forsell. Configurable emulated shared memory architecture for general purpose MP-SOCs and NOC regions. In *Networks-on-Chip, 2009. NoCS 2009. 3rd ACM/IEEE International Symposium on*, pages 163–172, 2009. doi: 10.1109/NOCS.2009.5071464. Cited on page 10.

[37] M. Forsell. TOTAL ECLIPSE - An efficient architectural realization of the parallel random access machine. In *Parallel and Distributed Computing*, pages 39–64. InTech, 2010. ISBN 978-953-307-057-5. doi: 10.5772/9446. Cited on pages 3, 9, 10, 11, 12, 13, 14, and 17.

[38] M. Forsell and J. Roivainen. Supporting ordered multiprefix operations in emulated shared memory CMPs. In H. R. Arabnia, editor, *Proc. International conference on parallel and distributed processing techniques and applications (PDPTA'11)*, pages 506–512, Las Vegas, USA, 2011. CSREA. ISBN 1-60132-195-3. Cited on page 13.

[39] M. Forsell, E. Hansson, C. Kessler, J.M. Mäkelä, and V. Leppänen. Hardware and software support for NUMA computing on configurable emulated shared memory architectures. In *IEEE 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, pages 640–648, 2013. doi: 10.1109/IPDPSW.2013.146. Cited on page 11.

[40] M. Forsell, E. Hansson, C. Kessler, J.M. Mäkelä, and V. Leppänen. NUMA computing with hardware and software co-support on configurable emulated shared memory architectures. *International Journal of Networking and Computing*, 4(1), 2014. ISSN 2185-2847. URL http://ijnc.org/index.php/ijnc/article/view/80. Cited on page 11.

[41] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. ISBN 0201648652. Cited on pages 4, 5, 18, 19, 22, 41, and 64.

[42] J. L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988. ISSN 0001-0782. doi: 10.1145/42411.42415. URL `http://doi.acm.org/10.1145/42411.42415`. Cited on page 5.

[43] E. Hansson, E. Alnervik, C. Kessler, and M. Forsell. A quantitative comparison of emulated shared memory architectures to current multicore CPUs and GPUs. In T. Nordström and Z. Ul-Abdin, editors, *Proceedings of MCC-2013*, pages 43–46. Halmstad Univ., 2013. Cited on pages 8 and 76.

[44] E. Hansson, E. Alnervik, C. Kessler, and M. Forsell. A quantitative comparison of PRAM based emulated shared memory architectures to current multicore CPUs and GPUs. In *Proc. of 11th Workshop on Parallel Systems and Algorithms (PASA'14)*, 2014. Cited on page 8.

[45] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU Using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-77219-7, 978-3-540-77219-4. URL `http://dl.acm.org/citation.cfm?id=1782174.1782200`. Cited on page 62.

[46] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008. ISSN 0018-9162. doi: 10.1109/MC.2008.209. Cited on page 5.

[47] F. Keceli and U. Vishkin. XMTSim: A simulator of the XMT many-core architecture. Technical Report UMIACS-TR;2011-02, University of Maryland, 2011. URL `http://hdl.handle.net/1903/13893`. Cited on page 27.

[48] F. Keceli, A. Tzannes, G.C. Caragea, R. Barua, and U. Vishkin. Toolchain for programming, simulating and studying the XMT many-core architecture. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1282–1291, 2011. doi: 10.1109/IPDPS.2011.270. Cited on pages 25 and 27.

[49] J. Keller, C. Keßler, and J. L. Träff. *Practical PRAM programming*. John Wiley & Sons, Inc., New York, NY, USA, 2000. ISBN 0471353515. Cited on pages 2, 3, 4, 30, 31, 32, 33, 34, 37, 38, 47, and 50.

[50] C. Kessler. The PRAM programming language Fork. URL `http://www.ida.liu.se/~chrke/fork95.html`. Fetched 2013-10-30. Cited on pages 30, 31, 34, and 69.

[51] C. Kessler and E. Hansson. Flexible scheduling and thread allocation for synchronous parallel tasks. In *Proc. of 10th Workshop on Parallel Systems and Algorithms (PASA'12)*, 2012. Cited on page 89.

[52] C. Keßler and H. Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *IJPP*, 25(1):17–50, 1997. ISSN 0885-7458. doi: 10.1007/BF02700045. URL `http://dx.doi.org/10.1007/BF02700045`. Cited on page 31.

[53] M. Kessler, E. Hansson, D. Åkesson, and C. Kessler. Exploiting instruction level parallelism for REPLICA - A configurable VLIW architecture with chained functional units. In *Proc. 18th Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'12)*, July 2012. Cited on page 12.

[54] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38(3): 451–460, 2010. ISSN 0163-5964. doi: 10.1145/1816038.1816021. URL http://doi.acm.org/10.1145/1816038.1816021. Cited on page 7.

[55] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The dash prototype: Logic overhead and performance. *Parallel and Distributed Systems, IEEE Transactions on*, 4(1):41–61, 1993. ISSN 1045-9219. doi: 10.1109/71.205652. Cited on page 6.

[56] A. Lööw. A functional-level simulator for the configurable (many-core) PRAM-like REPLICA architecture. Master's thesis, Linköping Univ., 2012. URL http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-79049. Cited on page 17.

[57] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. TOP500. URL http://www.top500.org/. Fetched 2013-10-22. Cited on page 7.

[58] J.M. Mäkelä, E. Hansson, D. Åkesson, M. Forsell, C. Kessler, and V. Leppänen. Design of the language REPLICA for hybrid PRAM-NUMA many-core architectures. In *IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 697–704, 2012. doi: 10.1109/ISPA.2012.103. Cited on pages 9, 14, and 16.

[59] D. Naishlos, J. Nuzman, C. W. Tseng, and U. Vishkin. Evaluating the XMT parallel programming model. *High-Level Parallel Programming Models and Supportive Environments*, pages 95–108, 2001. Cited on pages 47, 67, and 70.

[60] H. Nguyen. *GPU Gems 3*. Addison-Wesley Professional, first edition, 2007. ISBN 9780321545428. Cited on page 40.

[61] *CUBLAS library, Version 5.5*. NVIDIA, 2013. URL http://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf. Cited on page 50.

[62] *CUDA C Programming Guide, Version 5.5*. NVIDIA, 2013. URL http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Cited on pages 27, 28, 29, 30, 48, and 70.

[63] *CUDA Samples, Version 5.5*. NVIDIA, 2013. URL http://docs.nvidia.com/cuda/pdf/CUDA_Samples.pdf. Cited on page 68.

[64] *Thrust quick start guide, Version 5.5*. NVIDIA, 2013. URL http://docs.

nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf. Cited on pages 40 and 70.

[65] *OpenMP Application Program Interface, Version 3.0*. OpenMP Architecture Review Board, 2008. URL http://www.openmp.org/mp-documents/spec30.pdf. Cited on pages 19, 20, 21, 22, and 23.

[66] W. J. Paul, P. Bach, M. Bosch, J. Fischer, C. Lichtenau, and J. Röhrig. Real PRAM Programming. In B. Monien and R. Feldmann, editors, *Euro-Par*, volume 2400, pages 522–531. Springer, 2002. ISBN 3-540-44049-6. Cited on pages 6, 7, and 34.

[67] A. Petitet, R. C. Whaley, Dongarra J., and A. Cleary. HPL - A portable implementation of the high-performance Linpack benchmark for distributed-memory computers. URL http://www.netlib.org/benchmark/hpl/. Fetched 2013-10-22. Cited on page 7.

[68] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? *SIGARCH Comput. Archit. News*, 40(3):440–451, June 2012. ISSN 0163-5964. doi: 10.1145/2366231.2337210. URL http://doi.acm.org/10.1145/2366231.2337210. Cited on page 1.

[69] H. Schildt. *C: The Complete Reference*. McGraw-Hill, Inc., New York, NY, USA, 4 edition, 2000. ISBN 0072121246, 9780072121247. Cited on pages 39 and 89.

[70] C. Shuai, M. Boyer, Jiayuan M., D. Tarjan, J.W. Sheaffer, L. Sang-Ha, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, 2009. doi: 10.1109/IISWC.2009.5306797. Cited on page 8.

[71] H. Sutter. The free lunch is over. *Dr. Dobb's Journal*, 30(3), 2005. URL http://www.gotw.ca/publications/concurrency-ddj.htm. Cited on page 1.

[72] U. Vishkin. Explicit Multi-Threading (XMT): A PRAM-on-chip vision, . URL http://www.umiacs.umd.edu/~vishkin/XMT. Fetched 2013-06-15. Cited on pages 6 and 23.

[73] U. Vishkin. The second generation of XMT empirical work, . URL http://www.umiacs.umd.edu/users/vishkin/XMT/index5-06.html. Fetched 2013-07-22. Cited on pages 67 and 68.

[74] U. Vishkin, G. C. Caragea, and B. Lee. *Handbook of parallel computing: Models, algorithms and applications*, chapter Models for advancing PRAM and other algorithms into parallel programs for a PRAM-on-chip platform. Chapman and Hall/CRC Press, 2007. ISBN 9781584886235. Cited on pages 55 and 62.

[75] X. Wen. *Hardware design, prototyping and studies of the explicit multi-threading (XMT) paradigm*. PhD thesis, College Park, MD, USA, 2008. AAI3324912. Cited on pages 23, 24, 25, and 26.

[76] X. Wen and U. Vishkin. FPGA-based prototype of a PRAM-on-chip processor. In *Proceedings of the 5th conference on Computing frontiers*, CF '08, pages 55–66, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-077-7. doi: 10.1145/1366230.1366240. URL http://doi.acm.org/10.1145/1366230.1366240. Cited on pages 6, 23, and 25.

[77] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley, 2013. ISBN 9780321809469. Cited on pages 27, 29, 37, and 38.

[78] Z. Xianyi, W. Qian, and Z. Chothia. OpenBLAS version 0.2.8. URL http://www.openblas.net/. Fetched 2013-09-13. Cited on pages 47 and 70.

[79] A.N. Yzelman. URL http://people.cs.kuleuven.be/~albert-jan.yzelman/software.php. Fetched 2013-08-1. Cited on pages 54 and 70.

[80] C. Zhou. A source-to-source compiler for the PRAM language Fork to the REPLICA many-core architecture. Master's thesis, Linköping Univ., 2012. URL http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-80835. Cited on page 18.

[81] D. Åkesson. An LLVM back-end for REPLICA: Code generation for a multi-core VLIW processor with chaining. Master's thesis, Linköping Univ., 2012. URL http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-77405. Cited on pages 14, 15, 16, 18, and 66.

Linköpings universitet

Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/

© Erik Alnervik