J Supercomput (2013) 66:431–487
DOI 10.1007/s11227-013-0916-9

High-performance optimizations on tiled many-core embedded systems: a matrix multiplication case study

**Arslan Munir · Farinaz Koushanfar ·
Ann Gordon-Ross · Sanjay Ranka**

Published online: 5 April 2013

Abstract Technological advancements in the silicon industry, as predicted by Moore's law, have resulted in an increasing number of processor cores on a single chip, giving rise to multicore, and subsequently many-core architectures. This work focuses on identifying key architecture and software optimizations to attain high performance from *tiled many-core architectures* (TMAs)—an architectural innovation in the multicore technology. Although embedded systems design is traditionally power-centric, there has been a recent shift toward high-performance embedded computing due to the proliferation of compute-intensive embedded applications. The TMAs are suitable for these embedded applications due to low-power design features in many of these TMAs. We discuss the performance optimizations on a single tile (processor core) as well as parallel performance optimizations, such as application decomposition, cache locality, tile locality, memory balancing, and horizontal communication for TMAs. We elaborate compiler-based optimizations that are applicable to

A. Munir (✉) · F. Koushanfar
Department of Electrical and Computer Engineering, Rice University, Houston, TX, USA
e-mail: arslan@rice.edu

F. Koushanfar
e-mail: farinaz@rice.edu

A. Gordon-Ross
Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA
e-mail: ann@ece.ufl.edu

A. Gordon-Ross
NSF Center for High-Performance Reconfigurable Computing (CHREC), University of Florida, Gainesville, FL, USA

S. Ranka
Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA
e-mail: ranka@cise.ufl.edu

TMAs, such as function inlining, loop unrolling, and feedback-based optimizations. We present a case study with optimized dense matrix multiplication algorithms for Tilera's TILEPro64 to experimentally demonstrate the performance and performance per watt optimizations on TMAs. Our results quantify the effectiveness of algorithmic choices, cache blocking, compiler optimizations, and horizontal communication in attaining high performance and performance per watt on TMAs.

Keywords High-performance · Performance per watt · Embedded systems · Optimization · Many-core · Tiled many-core architecture

1 Introduction

The scaling of complementary metal-oxide-semiconductor (CMOS) transistors into the nanometer regime unveils the possibility of integrating millions of transistors on a single chip. A major challenge for the computer industry is efficient utilization of this ever-increasing number of on-chip transistors. Increasing clock frequency and single-core architectural innovations, such as deep pipelines, out-of-order execution, and prefetching, to exploit instruction-level parallelism (ILP) for enhancing single-thread performance yield diminishing returns as these innovations/techniques hit the *power wall* and the *ILP wall* [1]. Consequently, major segments of the computer industry conclude that future performance improvements must largely come from increasing the number of on-chip processor cores.

The transformation in the computer industry from single-core to multicore, and subsequently many-core necessitates efficient exploitation of thread-level parallelism (TLP) for attaining high performance. The terms *many-core* and *massively multicore* are sometimes used to refer to multicore architectures with an especially high number of cores (tens or hundreds) [2, 3]. Many-core technologies aim to exploit concurrency, high computational density, workload distribution, or a combination of these methods to attain high performance. The term *high-performance* refers to attaining superior performance quantified in terms of Mega operations per second (MOPS) or Mega floating point operations per second (MFLOPS) from an architecture. A *tiled many-core architecture* (TMA) is an emerging trend in many-core architecture in which processor cores (known as *tiles* in a TMA) are arranged in a regular, grid-like fashion, and a network on-chip (NoC) connects these tiles with each other and with I/O (input/output) devices. The increasing number of tiles in TMAs shifts the design focus from computation-oriented to communication-oriented, which makes a scalable NoC design imperative for TMAs.

TMAs are suitable for supercomputing and cloud computing [4] as well as embedded computing applications. Embedded system design is traditionally power-centric, but there has been a recent shift toward high-performance embedded computing due to the proliferation of compute-intensive embedded applications (e.g., networking, security, image processing). TMAs with massive computing resources on-chip and energy-saving features can be suitable for these high-performance embedded applications. Many TMAs (e.g., Tilera's TMAs [5]) offer low-power design features that allow idle tiles to be either clock gated (to save dynamic power) or power gated (to save static as well dynamic power).

Although TMAs offer tremendous computational power, extracting this computational power is non-trivial due to two main challenges: (1) on-chip data communication becomes more expensive as the number of on-chip processing cores/tiles increases; and (2) insufficient off-chip memory bandwidth limits the sustainable computational power. Overcoming these challenges requires many-core programmers to possess thorough knowledge of the underlying architecture. Without knowledge of the underlying TMAs' architectural features, programmers may experience performance degradation when migrating single-core software designs to TMAs [6]. John Hennessy, President of Stanford University, quotes on the challenge involved in attaining high-performance from parallel computers [7]:

"*... when we start talking about parallelism and ease of use of truly parallel computers, we're talking about a problem that's as hard as any that computer science has faced. ... I would be panicked if I were in industry.*"

The programming challenge in emerging TMAs is to exploit massive intrachip parallelism to obtain sustainable high performance. Many of these TMAs only support proprietary languages to exploit this intra-chip parallelism, which increases the programming effort and the time required to parallelize applications, which includes the learning curve of proprietary languages. Support for widely accepted high-level programming models, such as OpenMP (Open multiprocessing), would be beneficial for fast prototyping of applications on these TMAs. However, programming ease with high-level programming models comes at the expense of limited scalability for systems with a large number of processing elements. For example, language constructs in OpenMP can account for up to 12 % of the total execution time and developers are often advised to reduce the number of parallel regions to limit the impact of these overheads [8]. Nevertheless, quantifying the impact of existing and novel performance optimizations on TMAs would be beneficial for parallel programmers aiming to attain sustainable high performance from TMAs.

Attaining high performance from TMAs is typically an *iterative process*. Even a good design requires running the application, measuring the application's performance, identifying bottlenecks, and determining opportunities for improvement, modifying the application to achieve higher performance, and then remeasuring the application's performance, and so on. Obtaining high performance from TMAs requires determining how and where the execution time is being spent. Many times the bottleneck for attaining high performance is external memory latency or I/O throughput. Code optimizations and design decisions reflected in algorithmic-level changes can improve the performance by reducing external memory accesses. In cases where algorithmic-level changes and code optimizations fail to reduce the number of external memory accesses for an application, the attainable performance will be bounded by external memory bandwidth.

Previous works [9–12] discuss multicore architectures and performance of parallel applications, however, there has been limited discussion on high-performance optimization techniques that are applicable to TMAs. In this work, we focus on identifying key architecture and software optimizations to attain high performance from TMAs using Tilera's TILEPro64 and a dense matrix multiplication (MM) case study. Tilera's TMAs, to the best of our knowledge, are the first commercial TMA offering. Although dense MM algorithms have been studied extensively, optimizations on

TMAs have not yet been explored in detail. We discuss performance optimizations on a single tile as well as platform considerations for parallel performance optimizations, such as cache locality, tile locality, translation look-aside buffer (TLB) locality, and memory balancing. Our main contributions are as follows:

- A discussion of the architectural features of contemporary TMAs.
- Identification and discussion of key architecture and software optimization techniques that are applicable to TMAs.
- Elaboration and experimental demonstration of various compiler optimization techniques for TMAs, including inlining, loop unrolling, software pipelining, and feedback-based optimizations.
- Experimental demonstration of performance and performance per watt advantages of algorithmic optimizations that exploit cache blocking, parallelization, and horizontal communication on Tilera's TILEPro64 with dense MM as a case study.
- Quantification of the peak attainable performance from Tilera's TILEPro64.

Although dense MM algorithms have been studied extensively, optimizations on TMAs have not yet been explored in detail. Our work contributions advance the state of the art since TMAs are a potential architectural choice for future many-core embedded systems. Considering the TILEPro64's suitability to networking, security, video processing, and wireless network domains [2, 13, 14], this study provides insights for signal processing, security, and networking experts that aim to leverage the TILEPro64 for applications acceleration in their respective domains.

We point out that many of the optimizations discussed in this paper are also applicable to traditional CPUs, however, our work investigates these optimizations on TMAs and quantifies the impact of these optimizations on TMAs. The authors believe that it is imperative to investigate traditional optimization techniques on emerging many-core architectures to make programmers aware of the impact of existing optimization techniques on these many-core architectures. Programming experience with TMAs reveals that the compiler for these emerging TMAs (e.g., TILEPro64) supports sophisticated feedback-based optimizations that are not commonly available for traditional multicore architectures. Results highlight the effectiveness of algorithmic choices, cache blocking, compiler optimizations, and horizontal communication in attaining high performance from TMAs.

The remainder of this paper is organized as follows. A review of related work is given in Sect. 2. Section 3 gives an overview of architectural features of contemporary TMAs. Section 4 defines parallel computing metrics for TMAs and outlines the dense MM algorithms considered in our case study. Performance optimizations for TMAs including platform optimizations and compiler-based optimizations are discussed in Sect. 5. Section 6 presents the performance optimization results for our MM case study on Tilera's TMAs, with a focus on the TILEPro64. Section 7 summarizes conclusions and insights obtained from this study.

2 Related work

Previous work investigated performance analysis and optimization on multicore architectures. This section summarizes previous work related to performance analysis on multicore architectures, parallelized MM algorithms, cache blocking, and TMAs.

2.1 Performance analysis and optimization

In the area of performance analysis on parallel architectures, Brown et al. [15] compared the performance and programmability of the Born calculation (a model used to study the interactions between a protein and surrounding water molecules) using both OpenMP and Message Passing Interface (MPI). The authors observed that the OpenMP version's programmability and performance outperformed the MPI version, however, the scalability of the MPI version was superior to the OpenMP version. Sun et al. [16] investigated performance metrics, such as speedup, efficiency, and scalability, for shared memory systems. The authors identified the causes of superlinear speedups, such as cache size, parallel processing overhead reduction, and randomized algorithms for shared memory systems.

Some previous work explored performance optimizations. David Cortesi [17] studied performance tuning of programs running on the Silicon Graphics' SN0 systems, including the Origin2000, Onyx2, and Origin200 multiprocessor systems. The author described the architectural features and memory management of SN0 systems that impacted performance. The author further discussed cache optimizations including array padding to avoid cache thrashing, loop fusion, and cache blocking. Although the work captured performance tuning for Silicon Graphics' multiprocessors, our work applies the performance optimization techniques to emerging TMAs.

2.2 Parallelized MM algorithms

Krishnan et al. [18] described a new parallel algorithm for dense MM that had an efficiency equal to Cannon's algorithm for clusters and shared memory systems. The experimental results on clusters (IBM SP, Linux-Myrinet) and shared memory systems (SGI Altix, Cray X1) demonstrated the high performance of the proposed MM algorithm. The paper, however, did not compare the performance of the proposed algorithm with Cannon's algorithm. Our work implements both a blocked MM algorithm and Cannon's algorithm to provide an insight into the attainable performance of the two algorithms on TMAs.

Lee et al. [19] generalized Cannon's algorithm for the case when the input matrices were block-cyclic distributed (blocks separated by a fixed stride in the column and row directions were assigned to the same processor) across a two-dimensional (2D) processor array with an arbitrary number of processors and toroidal mesh interconnections. Performance analysis revealed that the generalized Cannon's algorithm generated fewer page faults than the previously proposed algorithm SUMMA (Scalable Universal Matrix Multiplication Algorithm) [20] that utilized broadcast communication primitives for the MM algorithm. Experimental results on an Intel Paragon showed that the generalized Cannon's algorithm performed better than SUMMA when the blocks were larger than 65×65, however, the generalized Cannon's algorithm exhibited worse performance than SUMMA for smaller block sizes. Results indicated that SUMMA maintained the same performance for all block sizes.

Much research focuses on evaluating parallel performance optimizations using MM as a case study. Li et al. [21] optimized MM for NVIDIA's Tesla C1060 graphics processing unit (GPU) and were able to attain 60 % of the GPU's theoretical peak performance (calculated from datasheets). To understand the impact of parallel algorithms on performance, A. More [22] implemented several versions of MM including simple, blocked, transposed, and BLAS (Basic Linear Algebra Subroutine) on an Intel Core Duo T2400 running at 1.83 GHz. Results revealed that carefully optimized MM implementations outperformed straightforward unoptimized implementations by orders of magnitude.

N. Higham [23] described FORTRAN-based level 3 BLAS (BLAS3) algorithms that were asymptotically faster than conventional versions. The author focused on Strassen's method for fast MM, which is practically useful for matrix dimensions greater than 100. Goto et al. [24] described the basic principles of a high-performance MM implementation that were used in the GotoBLAS library. The authors observed that the GotoBLAS MM attained near peak performance for various symmetric multiprocessor (SMP) architectures, such as Intel's Pentium 4, Intel's Itanium 2, and AMD's Opteron. We point out that BLAS algorithms are not available for existing TMAs, such as Tilera's TILE64/TILEPro64, nor do the existing TMAs' compilers support FORTRAN code fragments. Therefore, optimizations on TMAs can only be attained by parallelized algorithms tailored for TMAs, platform considerations, and compiler-based optimizations, which is the focus of this work.

2.3 Cache blocking

Optimizations using cache blocking have been studied in literature. Nishtala et al. [25] studied performance models of cache blocking for sparse matrix-vector multiply. The authors analyzed and verified the performance models on three processor architectures (Itenium 2, Pentium 3, and Power 4) and observed that while the performance models predicted performance and appropriate block sizes for some processors, none of the performance models were able to accurately predict performance and block sizes for all of the processor architectures. Our work takes an experimental approach to determine the best block size for cache blocking on TMAs.

Lam et al. [26] analyzed the performance of blocked code on machines with caches considering MM as a case study. By combining theory and experimentation, the work showed that blocking is effective generally for enhancing performance by reducing the memory access latency for caches, however, the magnitude of the performance benefit is highly sensitive to the problem size. The work focused only on blocking for single-core processor (DECstation 3100), however, our work analyzes blocking on multiple cores along with various compiler optimizations to attain high performance.

2.4 Tiled many-core architectures

TMAs have been studied in previous work. Wu et al. [11] described a tiled multicore stream architecture (TiSA) that consisted of multiple stream cores and an on-chip network to support stream transfers between tiles. In the stream programming model, which originated from the vector parallel model, an application is composed of a collection of data streams passing through a series of computation kernels running on stream cores. Stream cores were the basic computation units in TiSA, where the stream cores implemented stream processor architecture [27]. Each stream core had its own instruction controller, register file, and fully pipelined arithmetic and logic units (ALUs) that could perform one multiply-add operation per cycle. The authors implemented several benchmarks, such as MM, 3D-FFT, and StreamMD, on TiSA and were able to attain 358.9 GFLOPS for the MM benchmark. The paper, however, did not discuss high-performance optimizations for TiSA. Enric Musoll [10] studied performance, power, and thermal behavior of TMAs executing flow-based packet workloads and proposed a load-balancing policy of assigning packets to tiles that minimized the communication latency. The emphasis of the work was, however, on load-balancing and communication and not on performance optimization.

Vangal et al. [9] described the NoC architecture and Mattson et al. [12] described the instruction set, the programming environment, and their programming experience for Intel's TeraFLOPS research chip. The authors [9, 12] mapped several kernels, such as stencil, dense MM, spreadsheet, and 2D fast Fourier transform (FFT), to the TeraFLOPS chip. The authors were able to attain 1.0 teraFLOPS (TFLOPS) (73.3 % of theoretical peak attainable performance) for stencil, 0.51 TFLOPS (37.5 % of theoretical peak attainable performance) for dense MM, 0.45 TFLOPS (33.2 % of theoretical peak attainable performance) for spreadsheet, and 0.02 TFLOPS (2.73 % of theoretical peak attainable performance) for 2D FFT. Results indicated that the experimentally attainable performance on the research chip was far less than the theoretical peak attainable performance. These previous works, however, provided little discussion for attaining high performance from the chip.

Zhu et al. [28] investigated the performance of OpenMP language constructs on IBM's Cyclops-64 (C64) many-core architecture based on microbenchmarks. The authors observed that the overhead of OpenMP on the C64 was less than conventional SMPs. Our work differs from Zhu et al.'s work in that we investigate high performance on the TILEPro64 many-core architecture using Tilera's ilib application programming interface (API), which is designed to attain high performance on Tilera's architectures since the TILEPro64 does not support OpenMP.

Cuvillo et al. [8] mapped the OpenMP parallel programming model to IBM's C64 architecture. To realize this mapping, the authors exploited optimizations, such as the memory aware run time library that placed frequently used data structures in scratchpad memory and a barrier for collective synchronization that used C64 hardware support. The work, however, did not discuss techniques to obtain high performance from the C64. Garcia et al. [29] optimized MM for the C64 focusing on three optimizations: (1) balancing work distribution across threads; (2) minimal memory transfer and efficient register usage; and (3) architecture-specific optimizations, such as using special assembly functions for load and store operations. Their optimized MM implementation attained 55 % of the C64's theoretical peak performance. Our work differs from [29] in that we discuss additional optimizations, such as algorithmic optimizations, cache blocking, horizontal communication, and compiler-based optimizations.

Yuan et al. [1] investigated key architectural mechanisms and software optimizations to attain high performance for a dense MM on the Godson-T many-core prototype processor. The authors focused on optimizing on-chip communication and memory accesses. Results on a cycle-accurate simulator revealed that the optimized MM could attain 97 % (124.3 GFLOPS) of the Godson-T's theoretical peak performance due in part to the use of a BLAS-based MM sequential kernel. Although a BLAS-based kernel, in most cases, attains the best attainable performance, the absence of

BLAS-based routines for existing TMAs required us to use other optimizations in our work. The development of BLAS routines for Tilera's TMAs can be an interesting avenue for future research. Furthermore, since the BLAS-based routines are applicable only to some linear algebra applications, study of other high-performance techniques is important for parallel programmers to attain high performance for algebraic as well as nonalgebraic applications, which is the focus of our work.

Safari et al. [30] implemented a class of dense stereo vision algorithms on the TILEPro64 and were able to attain a performance of 30.45 frames per second for video graphics array (VGA) (640×480) images. The work demonstrated that emerging TMAs can achieve good performance in low level image processing computations. Our work is complementary to that work and focuses on high-performance optimization techniques for TMAs.

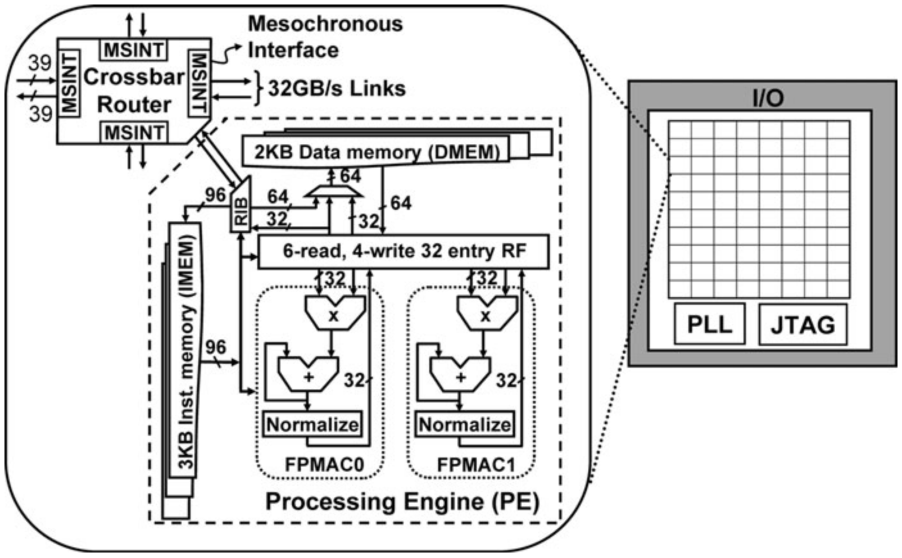
In our prior work, we cross-evaluated two multi-core architectural paradigms: symmetric multiprocessors (SMPs) and TMAs [31]. We based our evaluations on a parallelized information fusion application, Gaussian elimination, and an embarrassingly parallel benchmark. We compared and analyzed the performance of an Intel-based SMP and Tilera's TILEPro64 TMA. Results revealed that Tilera's TMAs were more suitable for applications with more TLP and little communication between the parallelized tasks (e.g., information fusion) whereas SMPs were more suitable for applications with floating point computations and a large amount of communication between processor cores due to better exploitation of shared memory in SMPs than TMAs. Insights obtained for SMPs, however, were limited to eight processor cores and the scalability of SMPs beyond eight processor cores was not investigated because of inexistence of an SMP platform with more than eight processor cores at the time of experimentation. Our current work differs from our previous work in that our current work does not cross-evaluate architectures, but rather focuses on attaining high-performance from TMAs.

3 Tiled many-core architecture (TMA) overview

A TMA is an emerging trend in many-core architecture that aims at exploiting massive on-chip resources furnished by recent advances in CMOS technology. TMAs combine each processor core with a switch/router to create a modular element called a *tile*, which can be replicated to create a many-core architecture with any number of tiles. The tiles are connected to an on-chip network and the switch (router) in each tile interconnects with the neighboring tiles. Each tile may consist of one or more computing cores and a router, which includes logic responsible for routing and forwarding the packets based on the routing policy. Examples of TMAs include the Raw processor, Intel's TeraFLOPS research chip, IBM's C64, and Tilera's TILE64, TILE*Pro*64, and TILE-Gx processor family [14, 32, 33]. This section discusses three TMAs: Intel's TeraFLOPS research chip, IBM's C64, Tilera's TILEPro64, and Tilera's TILE64.

3.1 Intel's TeraFLOPS research chip

Intel's TeraFLOPS research chip contains 80 tiles arranged as an 8×10 2D mesh network that is designed to operate at a maximum frequency of 5.7 GHz. The 80-



FPMAC: Floating-Point Multiply-Accumulator I/O: Input/Output
MSINT: Mesochronous Interface PLL: Phase-Locked Loop
JTAG: Joint Test Action Group RIB: Router Interface Block

Fig. 1 Intel's TeraFLOPS research chip (adapted from [9])

tile NoC architecture is implemented in 65 nm process technology and integrates a total of 100 million transistors on a 275 mm^2 die. The tiled design approach permits designers to use smaller cores that can easily be replicated across the chip. A single-core chip of this size ($\approx$100 million transistors) would require twice as many designers and roughly twice the design time [34]. The TeraFLOPS research chip uses a 2D mesh because of the large number of tiles and requirements of high bisection bandwidth and low average latency between tiles. The 2D on-chip mesh network provides a bisection bandwidth in excess of 320 GB/s. Intel's TeraFLOPS leverages low-power mesochronous clock distribution that facilitates scalability, high integration, and single chip realization of the TeraFLOPS processor.

Figure 1 shows one tile of Intel's TeraFLOPS research chip. Each tile consists of a processing engine (PE) connected to a 5-port router. The router in each tile connects to its four neighbors and the local PE via point-to-point links that can deliver data at 20 GB/s. These links support mesochronous interfaces (MSINT) that can provide phase-tolerant communication across tiles and lightweight global clock distribution at the expense of synchronization latency [35]. The router forwards packets between the tiles and can support 16 GB/s over each port. The PE contains two independent nine-stage pipelined single-precision floating-point multiply-accumulator (FPMAC) units, 3 KB single cycle instruction memory, and 2 KB data memory. The PE operates on a 96-bit Very Long Instruction Word (VLIW) that can encode up to eight operations per cycle. The 3 KB instruction memory can hold 256 96-bit VLIW instructions

Table 1 Theoretical peak performance and power consumption of Intel's TeraFLOPS research chip [12, 34]

Frequency	Voltage	Performance	Power
3.16 GHz	0.95 V	1.01 TFLOPS	62 W
4.27 GHz	1.07 V	1.37 TFLOPS	97 W
5.1 GHz	1.2 V	1.63 TFLOPS	175 W
5.7 GHz	1.35 V	1.81 TFLOPS	265 W

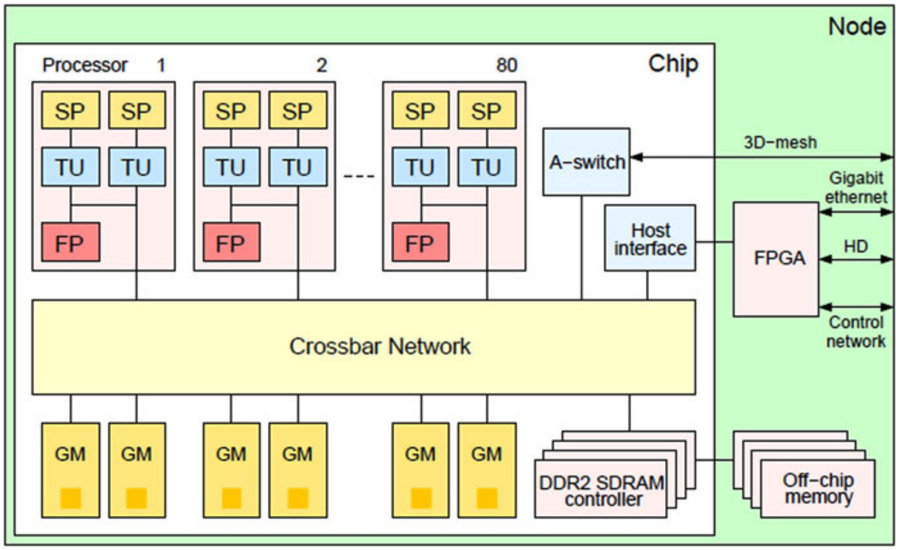
and the 2 KB data memory can hold 512 single-precision floating point numbers. The PE contains a 10-port (6 read and 4 write) register file that enables the PE to allow scheduling to both FPMACs, simultaneous loads, and stores from the data memory, packet send/receive from the mesh network, program control, and dynamic sleep instructions. The packet encapsulation between the router and the PE is handled by a router interface block (RIB). The fully symmetric architecture of Intel's TeraFLOPS research chip permits any tile's PE to send (receive) instruction and data packets to (from) any other tile.

The programming model for Intel's TeraFLOPS research chip is based on *message passing*. Each tile runs its own program and the tiles exchange messages to share data and to coordinate execution between the tiles. The message passing model is anonymous one-sided wherein any tile can write into the instruction or data memory of any other tile including itself. The TeraFLOPS chip can handle both single program multiple data (SPMD) as well as multiple program multiple data (MPMD) applications.

Since Intel's TeraFLOPS is a research chip, the TeraFLOPS chip has a very modest software environment with no compiler or operating system [12]. The programs for the TeraFLOPS chip are assembly coded and hand optimized. The program instructions are laid out in the instruction memory and the program is then launched simultaneously on all the tiles and progresses through the set of instructions in the program. The chip supports self-modifying code by sending new instructions as messages. The chip only supports single loop level with a single fixed stride (offset) across the memory. Nested loops require unrolling of the inner loops by hand. Hence, nested loops should be minimized or eliminated where possible. This modest software environment precludes the chip from full-scale application programming but is suitable for application kernels research.

The Intel's TeraFLOPS research chip implements fine-grained power management techniques to deliver power-efficient performance. The chip implements fine-grained clock gating and sleep transistor circuits to reduce active and standby leakage power, which can be controlled at chip, tile-slice/block, and individual tile levels depending on the workload. Approximately 90 % of the floating point units and 74 % of each PE is sleep-enabled [35]. The instruction set provides `WAKE`/`NAP` instructions that expose power management to the programmer, enabling the programmer to turn on/off the floating point units depending on the application's requirements. The chip allows any tile to issue sleep packets to any other tile or wake up any other tile depending on the processing task demands.

Table 1 summarizes Intel's TeraFLOPS research chip's available voltage and frequency settings for adjusting the performance and power. Operation at higher voltage and frequency can result in performance improvements but at a significant associated



DDR2 SDRAM: Double Data Rate 2 Synchronous Dynamic Random-Access Memory
FPGA: Field-Programmable Gate Array
SP: Scratchpad Memory
GM: Global Memory
TU: Thread Unit
FP: Floating Point Unit

Fig. 2 IBM Cyclops-64 chip (adapted from [8])

power consumption cost. For example, increasing (voltage, frequency) from (0.95 V, 3.16 GHz) to (1.35 V, 5.7 GHz) increases the performance by 79 % with an associated power consumption increase of 327 %.

3.2 IBM's Cyclops-64 (C64)

IBM's C64 is a petaFLOPS supercomputer that is intended to serve as a dedicate compute engine for high-performance applications, such as molecular dynamics, to study protein folding or image processing for real-time medical procedures [8]. A C64 chip is attached to a host system over several Gigabit Ethernet links. The host system facilitates application software developers and end users of the chip by providing a familiar computing environment.

Figure 2 depicts IBM's C64 chip, which consists of 80 processors, each with two thread units (TUs), one 64-bit floating point (FP) unit, and two static random-access memory (SRAM) banks of 32 KB each. Each TU is a 64-bit, single-issue, in-order, reduced instruction set computing (RISC) processor that operates at a clock frequency of 500 MHz. The FP unit can issue one double precision FP multiply and add instruction per cycle for a total peak theoretical performance of 80 GFLOPS per chip when running at 500 MHz. A portion of each SRAM bank can be configured as scratchpad (SP) memory. The remaining portions of SRAM together form a global memory (GM) that is uniformly accessibly from all TUs. The C64 chip provides 32 KB instruction caches (not shown in Fig. 2) where each instruction cache is shared by five processors. The C64 chip has no data caches. The C64 on-chip resources (e.g., TUs, on-chip memory banks) are connected to a 96-port crossbar network, which provides a bandwidth of 4 GB/s per port and a total bandwidth of 384 GB/s in each direction. The C64 chip provides an interface to the off-chip double data rate 2 (DDR2) synchronous dynamic random-access memory (SDRAM) and bidirectional interchip routing ports.

A C64 chip has an explicit three-level memory hierarchy: SP memory, on-chip memory (SRAM), and off-chip memory (DRAM). The memory hierarchy is software-managed such that the programmer can control the data movement between different levels of memory hierarchy. Having a software-managed memory hierarchy without caches saves the die area that would be required for hardware cache controllers and over-sized caches. The software-managed memory hierarchy provides the potential to improve not only performance but also energy efficiency at the cost of relatively complex programming as compared to architectures with hardware-controlled caches.

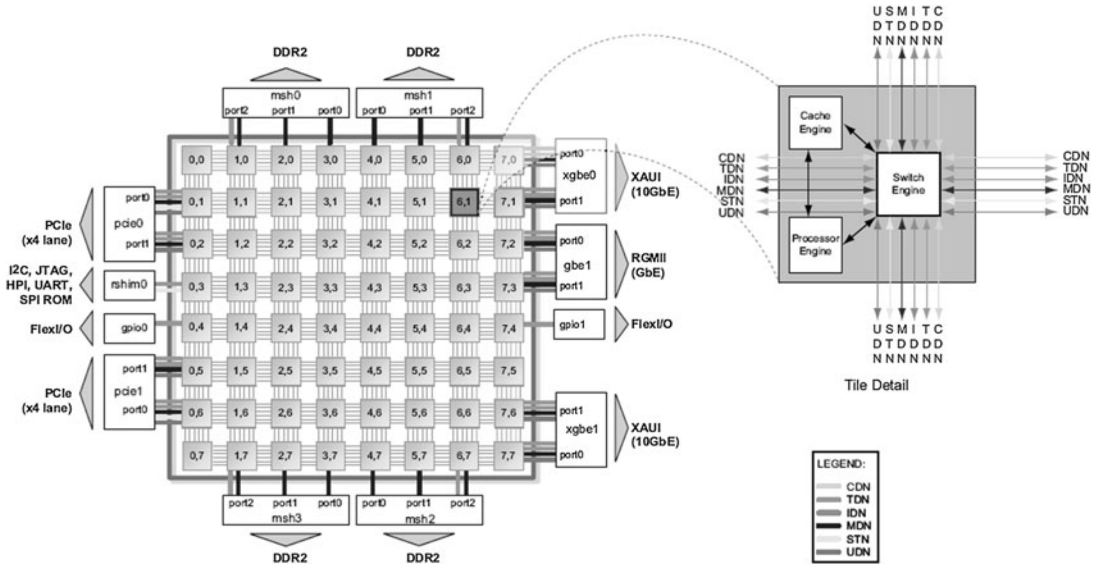
The C64 instruction set architecture provides efficient support for thread-level execution, hardware barriers, and atomic in-memory operations. The C64 architecture provides no resource virtualization mechanisms [8] (i.e., execution is non-preemptive and there is no hardware virtual memory manager). Only one single application can be run on the C64 chip at a time and the C64 microkernel will not interrupt the application execution unless an exception occurs. Lack of a virtual memory manager allows the three-level memory hierarchy of the C64 chip to be visible to the programmer.

The C64 architecture is a general purpose many-core architecture and energy-efficiency was not a key design consideration [29], therefore, there are no special features for energy savings. For example, the architecture does not allow processors to be turned off when unused nor can the clock rate to a set of processors or the whole chip be reduced.

3.3 Tilera's TILEPro64

Figure 3 depicts Tilera's TILEPro64 architecture. The TILEPro64 processor features an 8×8 grid of 64 tiles (cores) implemented in 90 nm process technology. The tiles are connected via multiple 2D mesh networks, designated as the iMesh interconnect by Tilera. The TILEPro64 integrates external memory and I/O interfaces on-chip that are connected to the tiles via the iMesh interconnect. Each tile shares an external off-chip global memory (also referred to as external memory henceforth), which consists of DRAM supported by four DDR2 memory controllers. Since Tilera's TMAs do not fall under the SMP architectural paradigm, a tile's access latency to external memory can be variable depending on the distance between the tile and external memory. Each tile can independently run a complete operating system or multiple tiles can be grouped together to run a multiprocessing operating system, such as SMP Linux [36]. Each tile contains a processor engine, a cache engine, and a switch engine [37].

The *processor engine* consists of a three-way VLIW pipelined processor with three instructions per bundle. Each tile has a program counter and is capable of issuing up to three instructions per cycle. Compile-time scheduling of VLIW operations results in lower power consumption as compared to dynamically-scheduled superscalar



DDR2: Double Data Rate 2 Synchronous Dynamic Random-Access Memory (SDRAM)

PCIe: Peripheral Component Interconnect Express

XAUI: X (Ten) Attachment Unit Interface

JTAG: Joint Test Action Group

CDN: Coherence Dynamic Network

TDN: Tile Dynamic Network

IDN: I/O Dynamic Network

MDN: Memory Dynamic Network

STN: Static Network

UDN: User Dynamic Network

Fig. 3 Tilera's TILEPro64 processor (adapted from [37])

processors. The TILEPro64 includes special instructions to support commonly-used operations in digital signal processing (DSP), encryption, network packet processing, and video processing, such as sum of absolute differences, hashing, and checksums.

The processor engine's VLIW architecture defines a 64-bit instruction bundle, which can specify either two or three instructions. The two-instruction (X1, X0) bundle encoding is known as *X-mode* and the three-instruction bundle encoding (Y2, Y1, Y0) is known as *Y-mode* where X0 $\in$ {arithmetic, compare, logical instructions, bit/byte instructions, multiply instructions} and X1 $\in$ {arithmetic, compare, logical instructions, control transfer instructions, memory management instructions}; Y0 $=$ X0, Y1 $\in$ {arithmetic, compare, logical instructions}, and Y2 $\in$ {memory instructions}. The individual instructions are typical RISC instructions.

The three-way VLIW processor engine in each tile contain three computing pipelines: P0, P1, and P2. P0 executes all arithmetic and logical operations, bit and byte manipulation, and all multiply and fused multiply (multiply-add) instructions. P1 can execute all arithmetic and logical operations, control flow instructions, and special-purpose register reads and writes. P2 executes all memory operations including loads, stores, and test and set instructions. Both the X- and Y-modes can issue instructions in any of three pipelines (P0, P1, and P2). The Y-mode instruction issue uses all of the three pipelines simultaneously whereas one of the pipelines remains in idle mode during X-mode instruction issue.

The *cache engine* contains the translation look-aside buffers (TLBs), caches, and a direct memory access (DMA) engine. Each tile has a 16 KB level one cache (8 KB instruction cache and 8 KB data cache) and a 64 KB level two cache, resulting in a total of 5.5 MB of on-chip cache with Tilera's *dynamic distributed cache (DDC)* technology. DDC provides a hardware-managed, cache-coherent approach to shared memory. DDC enables a tile to holistically view all of the tiles' on-chip caches as one large shared, dynamic distributed cache. DDC increases on-chip cache access and reduces the off-chip memory bottleneck. Each tile also contains a 2D DMA engine that can be configured by the application programmer to move data between the tiles and between the level two cache and the main memory. Tilera's ilib API provides various functions that enable application programmers to perform background DMA operations. The DMA engine operates autonomously from the processor engine and issues DMA load and store operations during cycles in which the cache pipeline is not being used by the processor engine.

The cache subsystem is nonblocking and supports multiple concurrent outstanding memory operations. The cache subsystem supports *hit under miss* and *miss under miss*, and permits the loads and stores to different addresses to be reordered to achieve high bandwidth and to overlap miss latencies, while ensuring that true memory dependencies are enforced. The processor engine does not stall on load or store cache misses and execution of subsequent instructions continue until the data requested by the cache miss is actually needed by another instruction.

The *switch engine* consists of six independent networks: one static network (STN) and five dynamic networks. The five dynamic networks are: the I/O dynamic network (IDN), memory dynamic network (MDN), coherence dynamic network (CDN), tile dynamic network (TDN), and user dynamic network (UDN). The STN and the five dynamic networks constitute Tilera's iMesh interconnect and due to the mesh layout, each of the six networks intersects in every tile. The STN transfers scalar data between tiles with very low latency. The dynamic networks facilitate the switch engine by routing packet-based data between tiles, tile caches, external memory, and I/O controllers. The dynamic networks leverage dimension-order routing—the x-direction first, and then the y-direction—until the packets reach the destination tile's switch engine. To reduce latency, packet routing is pipelined so that portions of the packet can be sent over a network link even before the remainder of the packet arrives at a switch. The IDN is used for data transfers between tiles and I/O devices and between I/O devices and memory. The MDN transfers data resulting from loads, stores, prefetches, cache misses, or DMAs between tiles and between tiles and external memory. The MDN has a direct hardware connection to the cache engine. The CDN carries cache coherence invalidation messages. The TDN supports data transfers between tiles and has a direct hardware connection to the cache engine.

Of the five dynamic networks, only the UDN is visible to the user and, therefore, we elaborate on the UDN's functionalities. User-level APIs, such as ilib, are built on the UDN, which abstract the details of the underlying packetization, routing, transport, buffering, flow control, and deadlock avoidance in the network. The UDN supports low-latency communication using a packet-switched mesh network and can be used by applications to send messages between tiles. The UDN enables fine-grained stream programming and supports efficient streaming through a register-level first input first output (FIFO) interface and hardware supported steering of distinct streams into separate FIFOs. The UDN-based stream data transfers have high bandwidth and have an overhead of only a few cycles.

The tile architecture provides a cache-coherent view of data memory to applications (i.e., a read by a thread or process at address `A` will return the value of the most recent write to address `A`). The hardware does not maintain coherence for the instruction memory but provides hardware cache coherence for I/O accesses.

Tilera's TILEPro64 supports 32-bit virtual memory addressing and can swap pages between the DRAM and the hard disk [4, 5]. The memory is byte addressable and can be accessed in units of 1, 2, and 4 bytes. Portions of the physical address space can be declared private or shared at the page granularity. Memory allocation requests by a tile (e.g., by using `malloc()`) allocate private memory to the tiles by default. When referencing private memory, a given virtual address on different tiles will reference different memory locations. Hence, the use of private memory enables efficient SIMD processing (i.e., the same code can be run on different tiles with each tile working on different data). The private memory is automatically cached in the on-chip caches when referenced by a tile. Shared memory allows multiple tiles to share instructions and data conveniently. A shared memory address on different tiles refers to the same global memory location.

Tiles in Tilera's TMAs can allocate shared memory using Tilera's multicore components (TMC) `cmem` API. Tilera's processor architecture provides flexibility of structuring shared memory as either distributed coherent cached or uncached shared memory. The uncached shared memory is allocated in uncached global shared memory, which can be accessed by an application using DMA. DMA operations copy uncached memory into the tiles' local caches. Using DMAs to create local private copies of data from uncached memory is suitable for coarse-grained data sharing where a process/tile wants to operate exclusively for some duration on a chunk of shared data before relinquishing control of that data to other processes/tiles. Tilera's processor architecture provides sequential consistency within a single thread of execution and relaxed memory consistency for the memory shared among multiple tiles. The architecture provides a memory fence instruction to force ordering when needed, as depicted in our parallel MM code scripts in Appendixes A.3 and A.4.

Tilera's run time software stack enables executing applications on the tile processor [38]. The run time software stack includes an application layer, a supervisor layer, and a hypervisor layer. The *application layer* runs standard libraries, such as the C run time library and Tilera's TMC library. The *supervisor layer*, which is composed of the Linux kernel, provides system calls for user-space applications and libraries. The supervisor layer also enables multiprocess applications and multi-threaded processes to exploit multiple tiles for enhanced performance. The *hypervisor layer* abstracts the hardware details of Tilera's processors from the supervisor and manages communication between the tiles and between the tiles and the I/O controllers. Each tile runs a separate instance of the hypervisor.

Tilera's TMAs support energy saving features. The tile processor implements clock gating for power-efficient operation. The architecture includes a software-usable `NAP` instruction that can put the tile in a low-power `IDLE` mode until a user-selectable external event, such an interrupt or packet arrival.

3.4 Tilera's TILE64

The architecture for Tilera's TILE64 is similar to the architecture for Tilera's TILEPro64 (Sect. 3.3) with a few differences. In this section, we highlight only the differences between the TILE64 and TILEPro64. Regarding interconnection network, the TILE64 has one static network and four dynamic networks (the CDN is not present in the TILE64) as compared to the TILEPro64's five dynamic networks. The addition of the CDN in the TILEPro64 increases the available network bandwidth. The TILEPro64 provides memory bandwidth that exceeds 205 Gbps as compared to 200 Gbps for the TILE64. The bisection bandwidth provided by the TILEPro64 is 2660 Gbps as compared to 2217 Gbps for the TILE64. The TILEPro64 also implements additional instructions that are not present in the TILE64, which accelerate DSP applications that require saturating arithmetic and unaligned memory accesses [37, 39].

4 Parallel computing metrics and matrix multiplication (MM) case study

Parallel computing metrics quantify the performance and performance per watt of parallel architectures, such as TMAs, and enable architectural comparisons. The most appropriate metrics for a parallel architecture depends on the targeted application domain. For example, run time performance is an appropriate metric for comparing high-performance systems whereas performance per watt is a more appropriate metric for embedded systems that have a limited power budget. In this section, we characterize the parallel computing metrics for TMAs and briefly outline the dense MM algorithms that we consider for our case study.

4.1 Parallel computing metrics for TMAs

We discuss the following parallel computing metrics: run time, speedup, efficiency, computational density, power, and performance per watt.

Run time The *serial run time* T_s of a program is the time elapsed between the beginning and end of the program on a sequential computer. The *parallel run time* T_p is the time elapsed from the beginning of a program to the moment the last processor finishes execution.

Speedup Speedup measures the performance gain achieved by parallelizing a given application/algorithm over the best sequential implementation of that application/algorithm. Speedup S is defined as T_s/T_p, which is the ratio of the serial run time T_s of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm T_p to solve the same problem on p processors. The speedup is ideal when the speedup is proportional to the number of processors used to solve the problem in parallel (i.e., $S = p$).

Efficiency Efficiency measures the fraction of the time that a processor is usefully employed. Efficiency E is defined as S/p, which is the ratio of the speedup S to the number of processors p. An efficiency of one corresponds to the ideal speedup and implies good scalability.

Computational density/peak theoretical performance The computational density (CD) metric measures the peak theoretical performance of a parallel architecture. The CD for a TMA with p tiles can be given as

$$\mathrm{CD} = p \times f \times \sum_i \frac{N_i}{\mathrm{CPI}_i} \tag{1}$$

where f denotes the operating frequency, N_i denotes the number of instructions of type i requiring integer or floating point computations that can be issued simultaneously, and CPI_i denotes the average number of cycles per instruction of type i. CD can be useful in estimating the peak theoretical performance of an architecture. For example, the CD for Tilera's TILEPro64 with each tile containing a 32-bit, 3-way issue VLIW processor and operating at 866 MHz can be computed as: $\mathrm{CD}^{\mathrm{TILEPro64}} = 64 \times 866 \times 3$ MOPS $= 166,272$ MOPS $= 166.272$ GOPS. We point out that this is merely peak theoretical performance that cannot be attained for most real-world applications.

Power Power consumption of a tile in a TMA is composed of a dynamic component and a static component. The dynamic power consumption depends on the supply voltage, clock frequency, capacitance, and the signal activity. The static power consumption mainly depends on the supply voltage, temperature, and capacitance [10]. The power consumption P of tile (both static and dynamic) can be divided into three components: the core's power P_{core}, the router's power P_{router}, and the interconnect power P_{net}:

$$P = P_{\mathrm{core}} + P_{\mathrm{router}} + P_{\mathrm{net}} \tag{2}$$

The core's power is the power consumption in the tile's processor core. The router power is the power consumption in the tile's router, and the interconnect power is the power dissipated in the interconnection network due to traversal of messages/packets caused by the requests originating from that tile.

We propose a system-level power model to estimate the power consumption of TMAs that can be used in estimating the performance per watt. Our power model considers both the active and idle modes' power consumptions. Given a TMA with a total of N tiles, the power consumption of the TMA with p active tiles can be given as

$$P^p = p \cdot \frac{P_{\text{max}}^{\text{active}}}{N} + (N - p) \cdot \frac{P_{\text{max}}^{\text{idle}}}{N} \tag{3}$$

where $P_{\text{max}}^{\text{active}}$ and $P_{\text{max}}^{\text{idle}}$ denote the maximum active and idle modes' power consumptions, respectively. $P_{\text{max}}^{\text{active}}/N$ and $P_{\text{max}}^{\text{idle}}/N$ give the active and idle modes' power, respectively, per tile and associated switching and interconnection network circuitry (and can be determined by Eq. (2)). Our power model incorporates the power saving features of state-of-art TMAs (e.g., Tilera's TILEPro64), which provide instructions to switch the processor cores and associated circuitry (switches, clock, interconnection network) not used in a computation to a low-power idle state. For example, a software-usable `NAP` instruction can be executed on a tile in Tilera's TMAs to put the tile into a low-power IDLE mode [37, 39]. Investigation of a comprehensive power model for TMAs is the focus of our future work.

Performance per Watt The performance per watt metric takes into account the power consumption of a device while quantifying performance. For quantifying the performance per watt, we leverage our power model given by Eq. (3).

4.2 Matrix multiplication (MM) case study

To demonstrate high-performance optimizations for TMAs, our work optimizes dense MM algorithms for Tilera's TMAs since MM is composed of operational patterns that are amenable for providing high performance. MM provides a wide range of insights since MM is a key building block in many scientific and engineering applications. The sequential run time for the conventional MM algorithm multiplying two $n \times n$ matrices is $\mathcal{O}(n^3)$ and requires $2n^3$ operations [40]. To demonstrate high-performance optimizations, we implement the following variants of the MM algorithm: (1) nonblocked sequential algorithm; (2) blocked sequential algorithm; (3) parallelized blocked algorithm; and (4) blocked Cannon's algorithm. The code for our MM algorithms' implementations on Tilera's TILEPro64 is presented in the Appendix and interested readers can find further discussions on the MM algorithms in [40], which are not discussed in this paper for brevity.

The original matrices for all of our MM algorithms reside in the shared memory. Our optimized blocked MM algorithm divides the matrices into submatrix blocks such that the submatrices fit in the caches for faster data access. Cannon's algorithm is a memory-efficient MM technique for parallel computers with toroidal mesh interconnections. The original Cannon's algorithm assumes that the input matrices are block-distributed among the processors doing the computation for the MM algorithm [19]. Our case study implementation of Cannon's algorithm also divides the original matrices into submatrix blocks that fit in the caches, however, all of the participating tiles load the submatrix blocks from shared memory by appropriately calculating the indices.

5 Performance optimization on a many-core architecture

As with any other parallel architecture, the application performance while executing on a TMA depends on both the performance per tile and the scaling of the performance across multiple tiles. Hence, optimizations on a TMA require optimizing for a single tile as well as optimizing for multiple tiles, which results from the application decomposition into several parts for parallel execution. In order to fully exploit multiple-tile optimizations, parallel programming for a TMA requires programmers to have a more detailed understanding of the underlying architectural features as compared to the serial programming for a single-core architecture. This section presents performance optimizations on a single tile as well as platform considerations for parallel performance optimizations, such as chip locality, tile locality, cache locality, and memory balancing. This section also discusses compiler-based optimizations, such as scalar optimizations, function inlining, alias analysis, loop unrolling, loop nest optimizations, software pipelining, and feedback-based optimizations.

5.1 Performance optimization on a single tile

Performance optimizations on a single TMA tile requires effective use of the central processing unit (CPU) and memory subsystem (including the caches and TLBs) as well as an optimized working set [41]. We point out that the effective use of aspects for a single tile is also important for parallel performance optimizations, thus to avoid redundancy with parallel performance optimizations (Sect. 5.2), this subsection only discusses the CPU, memory subsystem, and optimized working set.

5.1.1 CPU

Efficiently using the CPU for performance optimizations requires selecting efficient algorithms, compiler optimizations, and special-purpose instructions available on a TMA, such as sum of absolute differences, hashing, and checksums, depending on the application. The sequential code should be written to take advantage of ILP as much as possible. For fast access, frequently used variables should be directed as register-stored in the code when possible using the `register` keyword (in the C programming language).

5.1.2 Memory subsystem

Efficiently using the memory subsystem, which comprises the cache subsystem and external memory, for performance optimizations aims at reducing the required external memory bandwidth. Since each level of the memory hierarchy is at least an order of magnitude faster than the next lower level (e.g., level one cache is faster than the level two cache), a fundamental guideline of good on-chip memory management (e.g., caches/SP memory) that is equally applicable to TMAs is to fetch data from external memory, which has much larger latency than on-chip memory, only once, and use as many fetched words as possible before fetching the next data. The external memory bandwidth demand can be reduced in three ways: (1) maximizing the use of local caches (and SP memory if available) by keeping the code and data working sets small; (2) exploiting temporal locality by reusing recently accessed data; and (3) exploiting spatial locality by accessing nearby locations.

5.1.3 Working set

An application's working set refers to the code and/or data currently (at any given instance) used by the application. For the best performance, the working set's code size should be kept small enough to fit in the instruction caches. Compiler performance optimizations, such as function inlining, and loop unrolling, must take the instruction cache sizes into consideration otherwise aggressive compiler optimizations may degrade performance due to increased code size. Similarly, the working set's data must be small enough to fit in the data caches, otherwise the program's performance will degrade due to excessive accesses to the external memory.

Blocking is a technique that enables the data's working set to fit in the on-chip caches. Blocking, which follows a divide-and-conquer paradigm, divides a problem into smaller subproblems such that the smaller subproblems fit in the on-chip caches. To enhance performance, blocking can be done in a nested fashion taking into account both the level one and level two cache sizes. However, blocking adds complexity to the code. Cache blocking is advantageous when the benefits from the added locality due to cache blocking outweighs the cost of the additional overhead to access the data structure due to blocking. Furthermore, proper blocking granularity (block sizes) is not easy to determine theoretically (only an upper bound can be calculated in most cases) and the best blocking granularity for an algorithm is determined by experiments, as shown in Sect. 6.

5.2 Parallel performance optimizations

TMAs provide high computational density on-chip using multiple tiles, and a good understanding of parallel computing issues is necessary to fully leverage this computational density. Attaining high performance from a parallel architecture including TMAs requires efficient/balanced application decomposition as well as efficient use of hardware resources by exploiting architectural considerations such as chip locality, tile locality, cache locality, and memory balancing.

5.2.1 Application decomposition

A TMA's parallel performance depends on an application's decomposition into several parts that can run in parallel. There are two types of application decomposition: data decomposition and functional decomposition. In *data decomposition*, the data that the application processes is divided into parts that can be processed in parallel.

Functional decomposition separates subfunctions of an application and distributes the subfunctions across multiple tiles. Ideally, an application should be decomposed such that each of the participating tiles share an equal amount of the processing load. The effectiveness of this decomposition, along with task mapping (how tasks/parts of an application are assigned to available tiles) and the communication mechanism, determines the *scalability* and attainable performance.

5.2.2 Chip locality and parallel programming paradigms

Chip locality refers to keeping communication, synchronization, and storage on-chip where possible. Different application programming paradigms exploit chip locality differently and, therefore, can affect the attained performance from a TMA. In *distributed memory programming*, the applications' decomposed components have private local memory and communicate explicitly with other components using message passing. Managing messages and explicitly distributing data structures in distributed memory programming can add considerable complexity to the software development. *Shared memory programming* uses shared memory objects to communicate with an application's decomposed components. Since sharing of data is not explicit as compared to the message passing, the accesses to these shared memory objects are indistinguishable from accesses to the local objects in the program source code and can only be determined by resolving the allocation point of memory and by use of specialized tools [41].

There are advantages of supporting distributed memory programming in TMAs. Although message passing requires explicitly managing data structures and adds complexity to the software development, the complexity is balanced by greater ease in avoiding race conditions when all data sharing is through explicit messages rather than through a shared address space [12]. Shared address space presents challenges to the programmers trying to attain high performance from a TMA. While a programmer can block data to fit into a cache or use prefetching to prepare a cache ahead of computation, programmers have little or no control over data eviction from the cache. Hence, software writing with predictable performance can be challenging given that the state of the cache is difficult to control. Furthermore, it is difficult to assure in a shared address space that no unintended sharing is taking place whereas a message passing architecture naturally supports isolation between modules since sharing can only occur through explicit exchange of messages. A TMA relying on the message passing paradigm can alleviate these issues at the cost of complex software development.

In practice, parallel applications aiming to attain high performance from a TMA may benefit from a hybrid of distributed and shared memory programming (supported by Tilera's TMAs) such that bulk data transfers are done explicitly using distributed memory programming whereas more dynamic accesses use shared memory programming. Explicit communication as in message passing, enabled by ilib API in Tilera's TILE64/TILEPro64 that leverages UDN, exploit chip locality better as compared to using shared memory for communication via read and writes, which may spill data out of the caches into the external memory.

5.2.3 Tile locality

Tile locality refers to accessing data locally or from nearby tiles' on-chip memory. In Tilera's TMAs, every fragment of code and data has a *home tile*, where the code/data can be cached. For the best performance, code and data accesses from the home tile should hit in the local cache. Code and data accesses from other (remote) tiles traverse the on-chip interconnect, which is faster than going to the external memory but slower than accessing a local cache. Tile locality is affected by cache coherence and thread migration.

Code and data accesses from caches in multi-core/many-core architectures must ensure cache coherence, which can have an impact on performance. *Coherence* ensures that there is only one value for each memory location at any point in time. Some TMAs, such as the TILE64, maintain coherence using *single-caching*, which restricts each memory location to be cacheable on only one tile at a time, requiring other tiles to access that location through the home tile where it is cached. For TMAs that maintain coherence via single-caching, tile locality can be exploited by caching data in the tiles that will most often access that data (these tiles become the home tile for that data). *Shared-default* programming styles, such as multi-threading where global variables are shared across all threads, make tile locality difficult to achieve in single-caching architectures. Fortunately, some TMAs with single-caching (such as the TILE64) offer *neighborhood caching*, which exploits tile locality. Neighborhood caching enables data and code to be cached locally and spread across neighboring tiles to benefit from the large aggregate cache size of the neighboring tiles and minimizes off-chip memory references. Therefore, programmers should take advantage of tile locality by enabling neighborhood caching when using TMAs that support this functionality.

Some of Tilera's TMAs, such as the TILEPro64, implement a DDC technology (Sect. 3.3) that maintains coherence across multiple copies of cached data. DDC technology also leverages the *home tile* concept, and directs all write updates and invalidates from other caches containing stale data after a write to the home tile. The TILEPro64 hardware distributes home tiles for memory regions at the cache line granularity. Each cache line's address is hashed where the hashing result determines which of the tiles to use as the home tile for that cache line. This hashing approach, referred to as *hash-for-home*, reduces potential hot spots when a particular region of memory is in heavy use and the home tile for that region of memory receives a disproportionate number of requests. On the TILEPro64, read-only data sections are cached using hash-for-home so that the cache lines are scattered across all of the hashing tiles. Additionally, the TILEPro64 caches read-only data *noninclusively*, which does not evict cache lines from the level two caches when a cache line is evicted from the level three cache on the home tile.

Tile locality is also affected by thread migration, which is an operating system's process management policy (handled by the supervisor layer in Tilera's processors (Sect. 3.3)) for moving threads/processes from one tile to another depending on the available resources. Thread migration is essentially a dynamic load-sharing policy that aims to ensure that there are no idle tiles when there are tasks waiting for execution on other tiles. Thread migration, which is enabled by default, can reduce performance in some cases due to context-switching overheads. To maintain the mapping between the home tile of frequently-accessed data/code and the accessing threads/processes, threads/processes can be locked to tiles using *affinity*. This locking prevents threads/processes from unnecessarily sharing tiles and mitigates context-switching overheads, which can help in attaining high performance.

5.2.4 Cache locality

A TMA generally consists of a large number of tiles, where each tile has a private processor and cache subsystem. Each tile is computationally less powerful and has small and low associativity caches, however, the aggregate cache size across all of the tiles is comparable to the cache size in high-end SMPs. Applications benefit from temporal and spatial locality, which in turn benefit from cache *associativity*. High associativity is important for exploiting cache locality because lower associativity caches suffer from conflicts (i.e., *cache thrashing* where multiple memory locations with temporal locality map to the same cache line, continually evicting each other), resulting in excess memory traffic.

Considering the significance of associativity for high performance, contemporary TMAs, which provide on-chip caches, offer a mix of cache associativities—from direct-mapped to fully-associative for different caches in the cache subsystem. For example, the TILEPro64 has direct-mapped level one instruction caches, two-way associative level one data caches, four-way associative unified level two caches, and fully-associative unified TLBs. To exploit temporal locality, memory accesses should be spread out so that the memory locations map to different cache lines over short spans of time. Instructions have natural spatial locality because when one instruction is executed, typically the next instruction is likely to execute as well, except for branches, which account for approximately 20–30 % of the instructions [41]. Instructions also exhibit temporal locality, such as program loops or functions that are called repeatedly.

5.2.5 Translation look-aside buffer (TLB) locality

Most modern multicore architectures and some TMAs (e.g., Tilera's TMAs) use virtual memory so that the size of usable memory is not constrained by the size of the physical memory. Physical memory is typically partitioned into fixed-sized pages. *Page tables* map virtual addresses to physical addresses and keep track of whether a page resides in memory or the page's disk location. Page tables reside in memory, which adds additional memory access latency to perform virtual-to-physical address translations. To mitigate this additional latency, a small TLB stores information for the most recently used pages, acting as a cache for the page table. A TLB makes a virtual-to-physical address translation faster whenever the virtual address hits in the TLB. On a TLB miss, the page table is accessed and the translation is added to the TLB. Some recent architectures (e.g., Intel's Nehalem, Cortex-A15 MPCore [42]) use two-level TLBs to increase performance, under the same premise as two-level caches in processors.

In Tilera's TMAs, a TLB maps virtual address to physical addresses of the home tile at a virtual memory page size granularity. On most architectures, the page size is

64 KB by default, implying that the total amount of memory mapped by each TLB is small. Instruction and data TLBs are maintained separately (e.g., the TILEPro64 has an 8-entry instruction TLB and a 16-entry data TLB). TLB misses are handled effectively by hypervisor in approximately 100 cycles when the hypervisor's internal data structure contains the TLB entry. If the hypervisor's internal data structure does not contain the TLB entry, the hypervisor must access the page table entry, which can require 1000 cycles. The hypervisor stores TLB entries as a direct-mapped cache that holds approximately 1000 TLB entries. Tilera's TLBs support variable-sized pages with a minimum page size of 4 KB. To reduce TLB misses for applications referencing a wide span of code or data, programmers can use the *huge pages* or *large pages* option in Tilera's Linux kernel, where each TLB entry maps 16 MB of contiguous virtual memory as opposed to the default 64 KB page size. For example, `hugepages=0,4,4,0` reserves 4 huge pages on memory controllers 1 and 2, and none on memory controllers 0 and 3 [38]. Although huge pages enable referencing a wide span of code or data, huge pages consume more physical memory and limit flexibility in mapping different pages with different cache properties.

An important difference between a cache miss and a TLB miss is that a cache miss does not necessarily stall the CPU [24]. A TLB miss causes the CPU to stall until the TLB has been updated with the new address. A small number of cache misses can be tolerated by algorithmic prefetching techniques as long as the data is read fast enough from the memory such that the data arrives at the CPU by the time the data is needed for computation. Prefetching can mask a cache miss, but not a TLB miss. Since TLB misses are more expensive than cache misses, minimizing TLB misses can enhance performance. Hence, programmers should use available options to adjust the mapping size of TLB entries (such as *huge pages* option in Tilera's TMAs) depending on the application size and the function call graph.

5.2.6 Memory balancing

The overall memory bandwidth is maximized if external memory accesses are evenly balanced across the available memory controllers. High-order bits in the physical address select the memory controller and the hypervisor maps virtual addresses to physical addresses to maximize the memory balance. For the TILEPro64, the programmer can view the chip as divided into four quadrants of sixteen tiles where each tile maps memory to a memory controller adjacent to a quadrant. To achieve memory balancing, the TILEPro64 supports *memory striping*, which automatically spreads accesses across the available four memory controllers. The memory striping can be controlled by hypervisor's command: `options stripe_memory`. The `stripe_memory` option requests the hypervisor to configure the architecture to stripe memory accesses across all available memory controllers. Even with striped memory, the client supervisor and user programs see one unified physical address space rather than four separate address spaces. Striping is performed at a granularity of 8 KB (i.e., if physical address A is on controller N, physical address A + 8192 will be on controller N + 1, and so forth). By default, the hypervisor stripes memory if all memory controllers have an equal amount of memory. Hence, to achieve memory balancing and consequently high performance, parallel programmers for TMAs should enable memory striping if the parallelized application is to be run on a large number of tiles (greater than 16 for Tilera's TILE64 and TILEPro64).

5.2.7 On-chip mesh interconnect

A parallel application's performance and scalability can be improved by reducing the load on the on-chip interconnect since the interconnect is shared by all of the tiles. If an algorithm's data flow is not carefully organized, excessive communication over the interconnect can become a performance bottleneck.

5.3 Compiler-based optimizations

Compiler-based optimizations remove artificial constraints imposed by a programmer and a programming language to improve efficiency and expose inherent parallelism. The primary benefits of compiler-based optimizations include faster execution time and typically smaller object code size. Compiler-based optimizations can speed up development time by reducing the time required for performance optimizations and letting the compiler optimizer improve low-level code quality. The compiler optimization options determine the optimization level applied to the program to be optimized. We point out that the compiler optimization options are compiler vendor specific. For example, Sun's Studio compiler provides five compiler optimization levels, -O1 to -O5, where each increasing level adds more optimization strategies for the compiler, with -O5 being the highest level [43]. The compiler optimization levels for Tilera's TMAs vary from -O0 to -O3. The -O0 option applies no optimizations and the -O3 option provides the highest level of aggressive compiler optimizations. The -Os optimization option optimizes the program for both size and performance. The -Os option applies -O2 optimizations except those optimizations that increase the program size. In this subsection, we discuss compiler-based optimizations for TMAs, focusing on the TILEPro64 compiler tile-cc/c++.

5.3.1 Scalar optimizations

Scalar optimizations improve the execution time of a program using local and/or global optimizations. *Local optimizations* are applied to instructions within a basic block and do not leverage a holistic view of the function/program, such as considering data flow analysis to make optimization decisions. Local optimizations include constant propagation, copy propagation, common subexpression elimination, constant folding, operation folding, redundant load/store elimination, constant combining, strength reduction, and code reordering [44]. *Global optimizations* are applied among operations within the same function. Global optimizations include loop induction variable recognition and elimination, loop global variable migration, loop invariant code removal, and dead code removal [44]. Scalar optimizations are enabled at optimization level -O2 or above.

5.3.2 Function inlining

An application's performance can benefit from function inlining because function inlining exposes more context to the scalar and loop-nest optimizers and eliminates the function call overheads, such as register saving and restoring, the call and return instructions, etc. Function inlining can be user-directed and/or automatic. User-directed function inlining is performed at all optimization levels and can be specified by using the keyword inline or #pragma directives inline and noinline. We point out that the inline keyword is advisory, not mandatory (i.e., the keyword serves as a hint to the compiler to inline the function, but the function might or might not actually be inlined). The #pragma inline and the #pragma noinline directives instruct the compiler to inline or not inline, respectively, a function or set of functions. These #pragma directives can have next-line, entire routine, or global scope. Automatic inlining is enabled by default at the -O3 optimization level, but restricts large increases in code size. Automatic inlining is also enabled by default at nonzero optimization levels below -O3, but function inlining is only allowed for functions that are not estimated to increase the code size. Too much inlining may degrade performance due to the code working set problem discussed in Sect. 5.1.3.

5.3.3 Alias analysis

Memory aliases are caused when multiple pointers resolve to the same physical memory location. Compilers are conservative in optimization of memory references involving pointers because aliases can be difficult, or impossible, to detect at compile time. The compiler must conservatively assume that locations referenced by different pointers point to the same location, which limits the effectiveness of certain optimizations, such as instruction scheduling. Programmers can assist the compiler with alias analysis by marking a pointer with the restrict keyword, which guarantees that the memory region referenced by that pointer is the only way to access that memory region over the lifetime of that pointer.

5.3.4 Loop unrolling

Loop unrolling exposes more ILP, eliminates branches, amortizes loop overhead, and enables cross-loop-iteration optimizations, such as read/write elimination across the unrolled iterations. For example, loop unrolling can amortize loop overhead by replacing four loop counter increments $i+=1$ with one loop counter addition $i+=4$ if the loop unrolling factor is four. Loop unrolling can be performed if the loop to be unrolled satisfies the following conditions: the loop does not have internal cycles; the loop does not have an indirect branch that may jump back into the loop; the loop contains a single basic block; and the loop is a counter-based loop. Loop unrolling can be guided by placing a #pragma unroll n immediately before the loop where n is a constant specifying the loop unrolling factor. The pragma can be applied to both inner and noninner loops. An n value of 0 or 1 directs the compiler not to unroll the loop. We point out that the unroll pragma is only processed if the optimization level is high enough (i.e., -O2 or -Os for inner loops, and -O3 for outer loops).

5.3.5 Loop nest optimizations

A numerical program's execution time is mostly spent in loops. Loop nest optimizations perform loop optimizations that can greatly increase performance by better exploiting caches and ILP. Loop nest optimizations include loop interchange, cache blocking, outer loop unrolling, loop fusion, and loop fission. The order of loops in a nest can affect the number of cache misses, the number of instructions in the inner loop, and the compiler's ability to schedule an inner loop. Cache blocking and outer loop unrolling are closely related optimizations used to improve cache and register reuse. Loop fusion fuses multiple loop nests to improve cache behavior, to reduce the number of memory references, and to enable other optimizations, such as loop interchange and cache blocking. Loop fission, which is the opposite of loop fusion, distributes loops into multiple pieces. Loop fission is particularly useful in reducing register pressure in large inner loops. Loop fusion and fission can be enabled using `#pragma fuse` and `#pragma fission`, respectively. Loop nest optimizations are enabled by default with -O3 optimization flag.

5.3.6 Code generation phase optimizations

The compiler's code generator processes an input program in an intermediate representation format to produce an assembly file. A program is partitioned into basic blocks, such that a new basic block is started at each branch target and large blocks arbitrarily end after a certain number of operations/instructions. Code generation optimizations are not done at optimization level -O0. The code generator performs standard local optimizations on each basic block, such as copy propagation and dead code elimination, at optimization level -O1. The code generator performs global register allocation and various innermost loop optimizations at optimization levels -O2, -Os, and -O3.

5.3.7 Software pipelining

Software pipelining is a type of out-of-order execution where reordering is done by the compiler instead of the processor. Software pipelining schedules innermost counting loops (i.e., loops with no other terminating conditions or conditional statements) such that the hardware pipeline remains full. Software pipelining can be enabled and disabled using `#pragma swp` and `#pragma noswp`, respectively.

5.3.8 Feedback-based optimizations

Feedback-based optimizations are an advanced compiler optimization technique that replaces a compiler's estimations with actual data collected at run time using a specially instrumented executable. Collected data includes how many times a given loop executes and how often an `if` predicate is true. Feedback-based optimizations require compiling the program at least twice: the first compilation creates an instrumented executable and second compilation uses the information collected by the first executable to generate the optimized executable. Feedback-based optimizations help

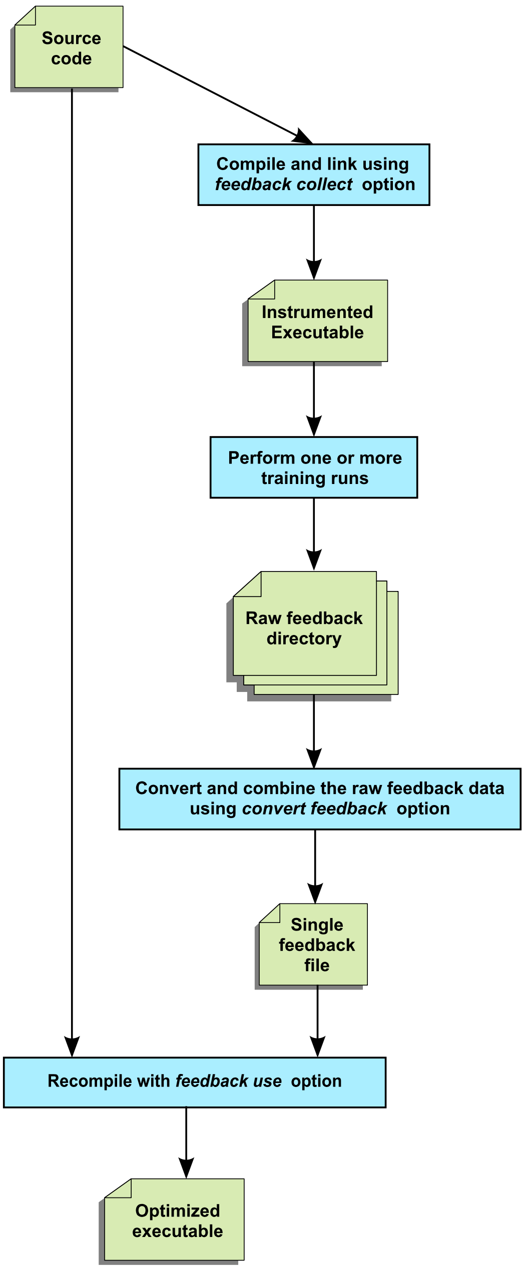

Source code
Compile and link using *feedback collect* option
Instrumented Executable
Perform one or more training runs
Raw feedback directory
Convert and combine the raw feedback data using *convert feedback* option
Single feedback file
Recompile with *feedback use* option
Optimized executable

Fig. 4 Feedback-based optimization workflow

compilers with loop unrolling and other loop optimizations, in addition to determining frequently executed blocks of code, which can be used to guide code scheduling, layout, and register allocation decisions.

Figure 4 depicts the feedback mechanism's workflow. The main steps are: (1) The source code/program is compiled and linked with the `feedback collect` option that creates an instrumented code that collects feedback data; (2) one or more training runs are performed on the instrumented executable with test inputs, also known as *training set*, to collect information about the program's typical behavior; (3) the compiler gathers information from the training run(s) in a raw feedback directory; (4) the compiler converts the contents of the raw feedback directory into a single feedback file, using the `convert feedback` option, suitable for use in the feedback optimization process; and (5) recompile and relink the source code with the `feedback use` option, which will optimize the executable using the feedback information generated previously. We point out that the same compiler optimization flag should be used for both the instrumented binary and the final optimized executable.

Feedback-based optimizations improve performance by leveraging both compiler and linker feedback. The compiler feedback improves code generation by recording various facts such as branch probabilities and executed loop counts. Based on the compiler feedback, the compiler rearranges code to straighten out likely code paths (e.g., remove branches) for more effective cache use. Linker feedback helps the compiler in selecting addresses for functions that reduce cache conflicts. During the creation of the instrumented executable, each instrumented function calls a run time library that tracks the order in which the functions execute. For example, if a program executes function f_1, then function f_2, and then function f_1 again. The feedback mechanism identifies if f_1 and f_2 alias in the cache, in which case f_2 will evict f_1, causing cache misses when f_1 is executed again. The feedback mechanism selects nonaliasing addresses for f_1 and f_2 so that both of these functions can coexist in the cache at the same time given a sufficient cache size.

A modern compiler uses feedback-based optimizations as a hint for optimizations and not as a guarantee of what future program input will look like since the data is gathered using a training/synthetic input. Therefore, a compiler does not make assumptions based on feedback-based optimizations that may produce incorrect code for inputs different than the training input.

6 Results

Attaining high performance and high performance per watt on TMAs is challenging mainly due to intricacies in parallel programming and parallel algorithms, complex hardware architecture, and scalability issues as the number of tiles increases. Some of the main factors influencing performance on TMAs include algorithmic choices, features of the underlying architecture, such as cache sizes, tile locality, and compiler-based optimizations. This section illustrates performance optimizations for TMAs, first on a single tile and then on multiple tiles, focusing on the TILEPro64 and dense MM as a case study. Each tile can independently run a complete operating system or multiple tiles can be grouped together to run a multiprocessing operating system, such as SMP Linux [36]. The TILEPro64 runs `Linux kernel 2.6.26.7-MDE-2.1.2.112814` version #1 `SMP`. We use Tilera's multicore development environment `ilib` API for parallelizing our MM algorithms and Tilera's `tile-cc` compiler for compiling and optimizing our programs. `tile-cc` is the cross-platform C compiler, which compiles ANSI (American National Standards Institute) standard

C to an optimized machine code for Tilera's processors. The compiler also supports software emulation of operations on data types such as floating-point and 64-bit integers.

Due to the limited programmability of the TILEPro64 as well keeping the number of experiments manageable, we were not able to evaluate all the optimizations discussed in Sect. 5. However, programmers can still benefit from the discussions in Sect. 5 for high-performance optimizations on contemporary and future TMAs that support these optimizations. Since some hardware architecture optimizations, such as memory balancing are automatically enabled by the TILEPro64's hypervisor, we do not elaborate on these hardware optimizations in our results. To obtain the performance and performance per watt results in this section, we implemented serial and parallel versions of MM with and without cache blocking (Sect. 4.2), and with and without leveraging compiler optimizations on the TILEPro64. Performance per watt calculations leverage our power model in Eq. (3). The power consumption values for the TMAs can be obtained from the devices' respective datasheets. For example, the TILEPro64 has a maximum active and idle mode power consumption of 28 W and 5 W, respectively [45, 46].

Table 2 summarizes the notations used for the performance and performance per watt optimization results for the MM algorithm. Figures 5 and 6 summarizes the impact of various high-performance optimizations on the execution time and performance per watt, respectively, for MM on the TILEPro64. We present the performance per watt results in MOPS per watt (MOPS/W). Results reveal that execution time can be reduced by 152× and performance per watt can be increased by 76× when using compiler optimizations and parallelization of the optimized MM algorithm (that leverages cache blocking) on 16 tiles as compared to a naive MM algorithm (without cache blocking as shown in Appendix A.1) on a single tile without any compiler optimizations (higher performance per watt gains can be achieved by parallelization on a greater number of tiles). The detailed results and discussion of these optimizations are presented in the following subsections.

6.1 Data allocation, data decomposition, data layout, and communication

Data allocation, data decomposition, data layout, and communication depend on the application and the architecture, and play an important role in attainable performance. This section discusses these aspects as well as the communication networks leveraged for communicating the data between the tiles and between the tiles and external memory for the MM algorithms implemented in this work.

For MM algorithms on a single tile, we allocate the data in external memory using the `malloc()` function. The data allocated using `malloc()` is only accessible to the tile allocating the data. For our parallel algorithms, we leverage Tilera's TMC `cmem` API for data allocation. We allocate the data in external memory and make the data shared for our parallel algorithms using the `tmc_ cmem_malloc()` function, which maps the allocated shared memory at the same address on all the participating tiles/processes. Since our parallel MM algorithms operate on the same original matrices (A, B, and C), the data must be shared in the external memory. We point out that for parallel applications that can operate on their own private data, data can be

Table 2 Performance and performance per watt optimization notations used in our MM case study

Notation	Description
b	Level two cache block size for the blocked MM algorithm
b'	Level one cache subblock size for the blocked MM algorithm
$T_{\text{NB}}^{(p)}$	Execution time of the nonblocked MM algorithm on p tiles with no compiler optimization flag
$T_{\text{NB-O3}}^{(p)}$	Execution time of the nonblocked MM algorithm on p tiles with compiler optimization flag -O3
$T_{\text{B}}^{(p)}$	Execution time of the blocked MM algorithm on p tiles with no compiler optimization flag
$T_{\text{B-O3}}^{(p)}$	Execution time of the blocked MM algorithm on p tiles with compiler optimization flag -O3
$T_{\text{B-FBO}}^{(p)}$	Execution time of the blocked MM algorithm on p tiles with feedback-based optimizations
$\text{Perf}_{\text{NB}}^{(p)}$	Performance of the nonblocked MM algorithm on p tiles with no compiler optimization flag
$\text{Perf}_{\text{NB-O3}}^{(p)}$	Performance of the nonblocked MM algorithm on p tiles with compiler optimization flag -O3
$\text{Perf}_{\text{B}}^{(p)}$	Performance of the blocked MM algorithm on p tiles with no compiler optimization flag
$\text{Perf}_{\text{B-O3}}^{(p)}$	Performance of the blocked MM algorithm on p tiles with compiler optimization flag -O3
$\text{Perf}_{\text{B-FBO}}^{(p)}$	Performance of the blocked MM algorithm on p tiles with feedback-based optimizations
$\text{Perf/W}_{\text{NB}}^{(p)}$	Performance per watt of the nonblocked MM algorithm on p tiles with no compiler optimization flag
$\text{Perf/W}_{\text{NB-O3}}^{(p)}$	Performance per watt of the nonblocked MM algorithm on p tiles with compiler optimization flag -O3
$\text{Perf/W}_{\text{B}}^{(p)}$	Performance per watt of the blocked MM algorithm on p tiles with no compiler optimization flag
$\text{Perf/W}_{\text{B-O3}}^{(p)}$	Performance per watt of the blocked MM algorithm on p tiles with compiler optimization flag -O3
$\text{Perf/W}_{\text{B-FBO}}^{(p)}$	Performance per watt of the blocked MM algorithm on p tiles with feedback-based optimizations

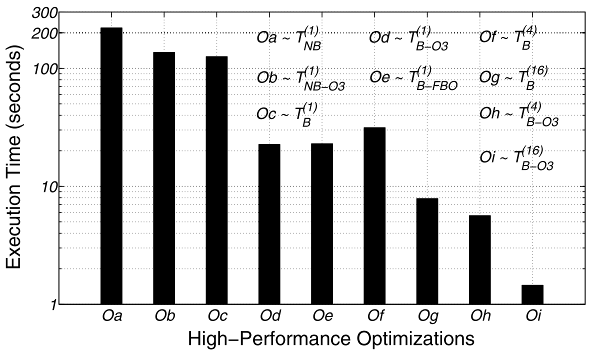


Fig. 5 The impact of high-performance optimizations on the execution time of MM on the TILEPro64 ($\sim$ denotes corresponds to)

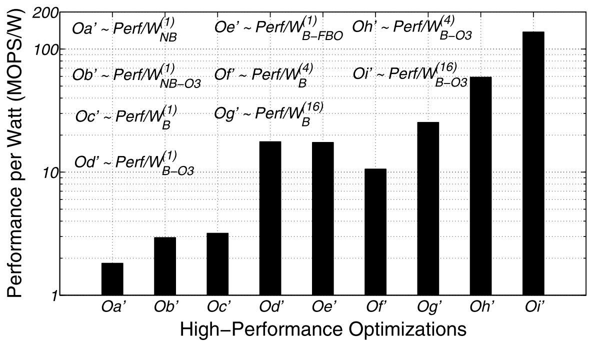


Fig. 6 The impact of high-performance optimizations on the performance per watt of MM on the TILEPro64 (∼ denotes corresponds to)

declared private for each tile/process in the external memory. Our experiments with various benchmarks on the TILEPro64 indicate that parallel applications with private data can attain better performance on the TMA as compared to the applications using shared data [31].

For our blocked MM algorithms, data is decomposed into small blocks of the original matrices that reside in the shared memory. Each data access from the shared memory by a tile fits in the local cache of the operating tile. Communication is overlapped by computation in our blocked MM algorithms as the tiles start operating on the requested data as soon as the first requested data byte is available from the memory. All subsequent computations by a tile on a given block access the data from the tile's local cache without accessing external memory until the computation requires processing data not already present in the cache.

For our MM algorithms, the tiles access data from the external memory using MDN (Sect. 3.3), which transfers data resulting from loads, stores, prefetches, and cache misses. Cannon's MM algorithm that uses horizontal communication (communication between tiles) leverages MDN as well as TDN that supports data transfers between tiles. The CDN, which carries cache coherence invalidation messages, is leveraged by both our blocked parallel MM algorithm and Cannon's MM algorithm. ilib functions used in our algorithms (see Appendixes A.3 and A.4 for code snippets) leverage UDN, which abstract the details of the underlying packetization, routing, transport, buffering, flow control, and deadlock avoidance in the network.

6.2 Performance optimizations on a single tile

Single-tile performance optimizations are important to attaining high performance on a TMA. Blocking algorithms that take into account cache sizes can significantly improve performance, however, selection of the appropriate block size is important for attaining high performance. Furthermore, compiler optimization flags, such as -Os, -O2, or -O3, enable most of the compiler-based optimizations to attain high performance. Feedback-based optimizations can further enhance performance by eliminating the compiler's guesswork for certain optimizations, such as loop unrolling. This subsection highlights these performance optimizations on a single tile of the TILEPro64.

Table 3 Performance and performance per watt of a blocked (B) and a nonblocked (NB) MM algorithm on a single tile of the TILEPro64 for different matrix sizes n

n	$T^{(1)}_{\text{NB-O3}}$ (s)	$T^{(1)}_{\text{B-O3}}$ (s)	$T^{(1)}_{\text{NB-O3}}/T^{(1)}_{\text{B-O3}}$	$\text{Perf}^{(1)}_{\text{NB-O3}}$ (MOPS)	$\text{Perf/W}^{(1)}_{\text{NB-O3}}$ (MOPS/W)	$\text{Perf}^{(1)}_{\text{B-O3}}$ (MOPS)	$\text{Perf/W}^{(1)}_{\text{B-O3}}$ (MOPS/W)
1024	134.59	22.65	5.9	15.96	2.98	94.81	17.69
2048	2171.58	215.33	10	7.91	1.47	79.78	14.88

6.2.1 Algorithmic optimizations and compiler optimizations

To evaluate the impact of algorithmic choices on performance and performance per watt, we implement a blocked integer MM algorithm considering the TILEPro64's cache sizes. We use an integer MM because the TILEPro64 does not have FP units and software emulation of FP may not provide meaningful insights into the performance gains achieved by various performance optimizations.

The blocked MM algorithm divides the original MM into optimized smaller sub-MMs that leverage the cache hierarchy using nested blocking for level one and level two data cache sizes of 8 KB and 64 KB, respectively, for the TILE64/TILEPro64. All three sub-MMs (corresponding to matrices A, B, and C) must fit in the cache for blocking to be useful. The required memory size for the blocks should also incorporate the size of data types, which depends on the processor architecture and compiler (e.g., 4 bytes for integer data types for a 32-bit processor and 32-bit compiler). For the level two cache, we calculate the block size b as $4 \cdot 3 \cdot b^2 \leq 64$ KB, which gives $b \leq 74$. For a power of two block size, $b \leq 64$. There is no need for our algorithms to use block sizes that are powers of two, however, we use block sizes that are powers of two to keep the number of experiments manageable and still show scaling. We use the constant 3 in the block size calculation corresponding to the level two cache because all of the three blocks from matrices A, B, and C ($C = A \cdot B$) must fit in the cache simultaneously and we use the constant 4 because the integer data type requires 4 bytes (32 bits) on the TILEPro64. Similar calculations for the block size corresponding to the level one cache b' gives $b' \leq 26$, or $b' \leq 16$, when considering the block size as a power of two.

To illustrate the impact of the algorithmic choices and cache blocking on attainable performance and performance per watt, we compare the performance and performance per watt of our blocked MM algorithm with a nonblocked MM algorithm. Table 3 depicts the performance and performance per watt of a blocked (B) and a nonblocked (NB) MM algorithm on a single tile using the compiler optimization level -O3 for two matrix sizes, $n = 1024$ and $n = 2048$, respectively, where n denotes the matrix dimensions of the square matrices A, B, and C. For the blocked MM algorithm, we select $b = 64$ and $b' = 4$, which satisfies the constraints on the calculated block sizes. The results indicate that the blocked MM algorithm provides $5.9\times$ and $10\times$ performance and performance per watt improvements over the non-blocked MM algorithm for $n = 1024$ and $n = 2048$, respectively. These results highlight the significance of carefully optimizing algorithms that are tailored for the underlying architectural specifications, such as the cache sizes, for attaining high performance and performance per watt from TMAs. The results also reveal that although compiler optimizations help both the blocked and nonblocked algorithms, compiler optimizations alone are not sufficient to achieve the maximum attainable performance (i.e., a nonblocked algorithm with aggressive compiler optimizations cannot attain the performance of a blocked optimized algorithm). We verified the results with other compiler optimization levels as well and present results show the best attainable performance from the compiler optimization levels.

Results in Table 3 highlight the effect of data sizes on the attainable performance per watt. For example, the attainable performance per watt for $n = 1024$ is $2\times$ greater than that for $n = 2048$ for the nonblocked MM algorithm. Similarly, the attainable performance per watt for $n = 1024$ is $1.2\times$ greater than that for $n = 2048$ for the blocked MM algorithm. This higher attainable performance per watt for smaller data sizes is due to the fact that smaller data sizes fit better in the cache and require lesser main memory requests to transfer data from main memory to the caches as compared to larger data sizes. We observe that even with large data sizes, the blocked MM algorithm helps in alleviating the performance per watt penalty for additional memory accesses. For example, the blocked MM algorithm reduces the performance per watt penalty due to additional memory accesses for $n = 2048$ over $n = 1024$ by 67 % as compared to the nonblocked MM algorithm.

6.2.2 *Evaluating block sizes and compiler optimizations*

Table 4 shows the performance and performance per watt of the MM algorithm running on a single tile with and without compiler optimizations for different matrix sizes n, block sizes b, and subblock sizes b'. The results reveal that as with traditional CPUs, block sizes are important in optimizing performance and performance per watt for a given TMA. Experiments indicate that block size calculations only give a constraint/upper bound (i.e., the maximum block size, for a given cache size) and the optimal block size for an algorithm can only be determined using a hit and try method (i.e., by selecting block sizes less than the maximum size obtained by the block size calculations, running the algorithm with each block size, measuring the performance, and then selecting the best block size based on the performance results). The results also reveal that compiler optimization level -O3 yields a significant improvement in execution time, performance, and performance per watt as compared to not using any compiler optimizations. For example, -O3 provides performance and performance per watt improvements of $5.6\times$ and $4.8\times$ for $n = 1024$ and $n = 2048$, respectively, when $b = 64$ and $b' = 4$. We also evaluated our blocked MM algorithm with other compiler optimization levels, such as -O2 and -Os, however, since -O3 yields the best performance, we present only results with -O3 for brevity.

The results in Table 4 depict that $b = 128$ and $b' = 8$ give the best results for $n = 1024$ and these sizes could be used for $n \leq 1024$, however, $b = 128$ and $b' = 8$ impose a $2\times$ lower performance than $b = 64$ and $b' = 4$ for $n = 2048$. Hence, we apply further optimizations based on block and subblock sizes of $b = 64$ and $b' = 4$, respectively.

Table 4 Performance of the blocked MM algorithm on a single tile of the TILEPro64 for different matrix sizes n, block sizes b, and subblock sizes b'

n	b	b'	$T_B^{(1)}$ (s)	$T_{B\text{-}O3}^{(1)}$ (s)	$\text{Perf}_B^{(1)}$ (MOPS)	$\text{Perf/W}_B^{(1)}$ (MOPS/W)	$\text{Perf}_{B\text{-}O3}^{(1)}$ (MOPS)	$\text{Perf/W}_{B\text{-}O3}^{(1)}$ (MOPS/W)
1024	32	4	126.08	22.97	17.03	3.18	93.49	17.44
	64	4	125.71	22.65	17.08	3.19	94.81	17.69
	128	4	125.56	22.53	17.1	3.19	95.32	17.78
1024	32	8	107.77	19.63	19.9	3.71	109.4	20.4
	64	8	107.52	19.4	19.97	3.72	110.7	20.65
	128	8	107.42	19.3	19.99	3.73	111.27	20.76
1024	32	16	117.73	29	18.24	3.4	74.05	13.82
	64	16	117.6	28.86	18.26	3.4	74.41	13.88
	128	16	117.55	28.82	18.27	3.4	74.51	13.9
2048	32	4	1042.06	218.21	16.49	3.08	78.73	14.69
	64	4	1033.87	215.33	16.62	3.1	79.78	14.88
	128	4	1040.77	216.22	16.5	3.08	79.46	14.82
2048	32	8	1164.97	414.43	14.75	2.75	41.45	7.73
	64	8	1162.07	412.22	14.78	2.76	41.68	7.78
	128	8	1162.04	419.41	14.78	2.76	40.96	7.64
2048	32	16	1362.94	434.48	12.6	2.35	39.54	7.38
	64	16	1360.57	433.25	12.63	2.36	39.65	7.4
	128	16	1360.07	418.55	12.63	2.36	41.05	7.66

6.2.3 Compiler directives-based optimizations

Programmers can specify appropriate compiler directives to achieve performance improvements. In this subsection, we discuss the performance improvements attained by some of the compiler directives discussed in Sect. 5.3. We point out that not all of the available compiler directives are appropriate for a given program/application to attain high performance. A programmer needs to speculate which compiler directives can be beneficial and then apply these compiler directives selectively depending on the program's constructs. Hit and try can also be beneficial with some compiler directives, such as loop unrolling to determine the best loop unrolling value. Using appropriate compiler directives for attaining high performance requires programming experience as inappropriate use of compiler directives can deteriorate performance.

Table 5 depicts the impact of various compiler directives with the -O2 optimization level on the performance of the nonblocked MM algorithm on a single tile of the TILEPro64 for $n = 1024$. We point out that many compiler directives are enabled only at specific compiler optimization levels. Experiments reveal that the compiler directives presented in Table 5 have a negligible impact on performance with compiler optimization levels -O0 and -O1. Results indicate that compiler optimization directives with the -O2 flag can increase the performance by $1.8\times$ or by 77 % as compared

Table 5 Compiler directives-based optimizations with compiler optimization level -O2 for the nonblocked MM algorithm

Compiler directives	$T^{(1)}_{\text{NB-O2}}(s)$
Without any compiler directives	136.32
#pragma inline here where call to MM kernel is made	121.43
#pragma inline here + #pragma blocking size (4,8) for outer MM loop	111.81
#pragma inline here + #pragma blocking size (4,8) for outer MM loop + #pragma blocking size (2,8) for inner MM loop	87.65
#pragma inline here + #pragma blocking size (4,8) for outer MM loop + #pragma blocking size (2,8) for inner MM loop + #pragma blocking size (2,8) for innermost MM loop	87.65
#pragma inline here + #pragma blocking size (4,8) for outer MM loop + #pragma blocking size (2,8) for inner MM loop + #pragma blocking size (2,8) for innermost MM loop + #pragma unroll 4	76.99
#pragma inline here + #pragma blocking size (4,8) for outer MM loop + #pragma blocking size (2,8) for inner MM loop + #pragma blocking size (2,8) for innermost MM loop + #pragma unroll 4 + #pragma swp	77.0

to not using any compiler directives. We also use hit and try to determine suitable parameters for compiler directives. For example, our experiments with #pragma unroll 2 does not give any performance improvement whereas #pragma unroll 4 increases the performance by 14 % for the nonblocked MM algorithm. We also observe that not all the compiler directives result in performance improvement, (e.g., #pragma swp does not enhance performance for the nonblocked MM algorithm).

In Table 5, #pragma blocking size (n1, n2) directs the compiler that if the specified loop is involved in blocking of the primary or level one (secondary or level two) cache will have the blocking size of n1 (n2). The specification of the blocking size for the secondary cache is optional, however, we observe better performance results are attained if the blocking size for both primary and secondary caches are specified. We point out that this compiler blocking directive is loop-based and different from cache blocking sizes we calculated earlier for MM algorithms. If a blocking size of 0 is specified, then the loop is not split and the entire loop is inside the block. #pragma unroll (n) directs the compiler to add (n − 1) copies of the inner loop body to the inner loop.

6.2.4 Feedback-based optimizations

Tables 6 and 7 depict performance and performance per watt, respectively, for the feedback-based optimizations applied to the blocked MM algorithm to investigate further performance and performance per watt enhancements on the TILEPro64. The results reveal that feedback-based optimizations do not improve performance and performance per watt for the TILEPro64 as compared to the when using compiler optimization level -O3. However, both the compiler optimization level -O3 and feedback-based optimization improve the performance and performance per watt by 5.5× and 4.8× for $n = 1024$ and $n = 2048$, respectively, for the blocked MM algorithm as compared to the blocked MM algorithm without any compiler optimizations.

Table 6 Performance of the blocked MM algorithm on a single tile of the TILEPro64 using feedback-based optimizations for different matrix sizes n

n	b	b'	$T_{\mathrm{B}}^{(1)}$ (s)	$T_{\mathrm{B\text{-}O3}}^{(1)}$ (s)	$T_{\mathrm{B\text{-}FBO}}^{(1)}$ (s)	$\mathrm{Perf}_{\mathrm{B}}^{(1)}$ (MOPS)	$\mathrm{Perf}_{\mathrm{B\text{-}O3}}^{(1)}$ (MOPS)	$\mathrm{Perf}_{\mathrm{B\text{-}FBO}}^{(1)}$ (MOPS)
1024	64	4	125.71	22.65	22.96	17.08	94.81	93.53
2048	64	4	1033.87	215.33	214.73	16.62	79.78	80.0

Table 7 Performance per watt of the blocked MM algorithm on a single tile of the TILEPro64 using feedback-based optimizations for different matrix sizes n

n	b	b'	$\mathrm{Perf/W}_{\mathrm{B}}^{(1)}$ (MOPS/W)	$\mathrm{Perf/W}_{\mathrm{B\text{-}O3}}^{(1)}$ (MOPS/W)	$\mathrm{Perf/W}_{\mathrm{B\text{-}FBO}}^{(1)}$ (MOPS/W)
1024	64	4	3.19	17.69	17.45
2048	64	4	3.1	14.88	14.92

Table 8 Performance of the blocked MM algorithm on a single tile of the TILE64 using feedback-based optimizations for different matrix sizes n

n	b	b'	$T_{\mathrm{B}}^{(1)}$ (s)	$T_{\mathrm{B\text{-}O3}}^{(1)}$ (s)	$T_{\mathrm{B\text{-}FBO}}^{(1)}$ (s)	$\mathrm{Perf}_{\mathrm{B}}^{(1)}$ (MOPS)	$\mathrm{Perf}_{\mathrm{B\text{-}O3}}^{(1)}$ (MOPS)	$\mathrm{Perf}_{\mathrm{B\text{-}FBO}}^{(1)}$ (MOPS)
1024	64	4	130.18	28.98	22.96	16.5	74.1	93.53
2048	64	4	1109.08	307.34	214.71	15.49	55.9	80.01

Table 9 Performance per watt of the blocked MM algorithm on a single tile of the TILE64 using feedback-based optimizations for different matrix sizes n

n	b	b'	$\mathrm{Perf/W}_{\mathrm{B}}^{(1)}$ (MOPS/W)	$\mathrm{Perf/W}_{\mathrm{B\text{-}O3}}^{(1)}$ (MOPS/W)	$\mathrm{Perf/W}_{\mathrm{B\text{-}FBO}}^{(1)}$ (MOPS/W)
1024	64	4	3.08	13.82	17.45
2048	64	4	2.89	10.43	14.93

To investigate whether feedback-based optimizations can be useful for TMAs other than the TILEPro64), we evaluated the TILE64. Tables 8 and 9 depict performance and performance per watt results, respectively, for the feedback-based optimizations applied to the blocked MM algorithm implemented on the TILE64 to provide comparison with the corresponding results for the TILEPro64 (Tables 6 and 7). By using only compiler optimization flags, the better performance of the TILEPro64 over the TILE64 could be attributed to slightly higher memory bandwidth of the TILEPro64 than the TILE64 (Sect. 3.4). The results for the TILE64 reveal that feedback-based optimizations can improve the performance as compared to compiler optimization level -O3, which indicates that there is room for improvement even after applying optimization level -O3. For example, feedback-based optimizations provide $1.3\times$ and $1.4\times$ performance and performance per watt improvements for $n = 1024$ and $n = 2048$, respectively, when $b = 64$ and $b' = 4$ for the TILE64. These results indicate that the TILEPro64 can exploit compiler optimizations better than the TILE64, which is why the TILEPro64 is able to attain performance close to feedback-based optimizations only with the compiler optimization flags. Since a programmer does not know in advance how much an architecture can exploit optimizations enabled by compiler optimization flags, the programmer can leverage feedback-based optimizations, when feasible, to provide further performance enhancements over the attainable performance using only compiler optimization flags.

6.3 Parallel performance optimizations

Parallel performance optimizations are important to leverage the high computing density of TMAs and require a parallel algorithm to be run on multiple tiles of a TMA. The parallel algorithm can either be a parallel version of a corresponding sequential algorithm or the parallel algorithm can be designed from scratch considering a parallel architecture. Block size selection for a parallel blocking algorithm is equally important as that for the serial blocking algorithm. The block sizes selected from the single-tile optimization of a serial blocking algorithm may be suitable for the parallel algorithm if the parallel blocking algorithm's design is such that the decomposed algorithm running on a single tile is similar in structure to the corresponding serial blocking algorithm. As with the optimizations on a single-tile, compiler optimization flags also help in attaining high performance from the parallel algorithm. Depending on the data distribution, parallel algorithms that leverage TMA features, such as horizontal communication, can be useful for attaining high performance from the parallel algorithm. Additionally, feedback-based optimizations are also beneficial in attaining parallel performance optimizations for TMAs. This subsection discusses these parallel performance optimizations focusing on the TILEPro64.

6.3.1 Algorithmic optimizations and compiler optimizations

We parallelize our blocked MM algorithm for the TILEPro64 to achieve performance enhancements via parallelization. Tables 10 and 11 depict the performance and performance per watt, respectively, of parallelized blocked MM algorithm running on four and sixteen tiles, $p = 4$ and $p = 16$, respectively, for different block sizes b and subblock sizes b' with and without compiler optimizations. The results reveal that compiler optimizations can greatly improve the performance and performance per watt of a parallelized blocked algorithm. For example, compiler optimization level -O3 yields 5.6× and 5.4× performance and performance per watt improvements when the blocked MM algorithm is executed on $p = 4$ and $p = 16$, respectively, when $n = 1024$, $b = 64$, and $b' = 4$. Similarly, compiler optimization level -O3 yields 5× and 4.8× performance and performance per watt improvements when the blocked MM algorithm is executed on $p = 4$ and $p = 16$, respectively, when $n = 2048$, $b = 64$, and $b' = 4$.

To quantify the parallel performance improvements, Table 12 also depicts the speedups attained by our parallelized blocked MM algorithm. We use the serial and parallel run times with compiler optimization level -O3 to calculate the speedups: $S^{(4)} = T_{\text{O3}}^{(1)}/T_{\text{O3}}^{(4)}$ and $S^{(16)} = T_{\text{O3}}^{(1)}/T_{\text{O3}}^{(16)}$ where $S^{(4)}$ and $S^{(16)}$ denote speedups for $p = 4$ and $p = 16$, respectively. The results reveal that the parallelized blocked MM algorithm attains ideal or close to ideal speedups for our selected block and subblock sizes ($b = 64$ and $b' = 4$). The results also reveal that a poorly-selected block

Table 10 Performance of the parallelized blocked MM algorithm for a different number of tiles p for the TILEPro64 for different matrix sizes n, block sizes b, and subblock sizes b' ($S^{(p)}$ denotes the speedup using p tiles)

n	b	b'	$T_{\mathrm{B}}^{(4)}$ (s)	$T_{\mathrm{B\text{-}O3}}^{(4)}$ (s)	$S^{(4)}$	$T_{\mathrm{B}}^{(16)}$ (s)	$T_{\mathrm{B\text{-}O3}}^{(16)}$ (s)	$S^{(16)}$	$\mathrm{Perf}_{\mathrm{B}}^{(4)}$ (MOPS)	$\mathrm{Perf}_{\mathrm{B\text{-}O3}}^{(4)}$ (MOPS)	$\mathrm{Perf}_{\mathrm{B}}^{(16)}$ (MOPS)	$\mathrm{Perf}_{\mathrm{B\text{-}O3}}^{(16)}$ (MOPS)
1024	32	4	31.53	5.73	3.4	7.9	1.49	13.0	68.1	374.78	271.83	1,441.26
	64	4	31.42	5.64	3.4	7.87	1.45	13.3	68.35	380.76	272.87	1,481.02
	128	4	31.39	5.61	3.4	15.73	2.84	6.8	68.41	382.8	136.52	756.16
1024	32	8	26.94	4.88	4.0	6.75	1.25	15.4	79.7	440.06	318.14	1,717.99
	64	8	26.86	4.82	4.0	6.73	1.22	15.8	79.95	445.54	319.09	1,760.23
	128	8	26.85	4.79	4.0	13.48	2.46	7.8	79.98	448.33	159.31	872.96
1024	32	16	29.45	7.44	2.6	7.43	2.07	9.3	72.92	288.64	289.03	1,037.4
	64	16	29.39	7.38	2.6	7.41	2.05	9.4	73.07	290.99	289.81	1,047.55
	128	16	29.37	7.36	2.6	15.01	3.92	4.9	73.12	291.78	143.07	547.83
2048	32	4	258.72	52.65	4.0	64.82	13.8	15.6	66.4	326.3	265.04	1,244.92
	64	4	257.59	51.84	4.0	64.56	13.49	16.0	66.69	331.4	266.11	1,273.53
	128	4	257.93	52.05	4.0	64.73	13.6	15.8	66.61	330.06	265.41	1,263.22
2048	32	8	287.14	107.33	2.0	72.7	30.57	7.0	59.83	160.06	236.31	561.98
	64	8	286.87	107.12	2.0	72.76	30.52	7.0	59.89	160.38	236.12	562.9
	128	8	286.88	107.13	2.0	72.8	30.53	7.0	59.88	160.36	235.99	562.72
2048	32	16	353.04	152.85	1.4	90.61	52.74	4.1	48.66	112.4	189.6	325.75
	64	16	352.47	150.57	1.4	90.4	51.57	4.2	48.74	114.1	190.04	333.14
	128	16	352.29	152.76	1.4	90.38	52.8	4.1	48.77	112.46	190.08	325.38

and subblock sizes can give results far from ideal. For example, using $b = 128$ and $b' = 8$ for $n = 1024$ gives $S^{(16)} = 7.8$, an efficiency of only 49 %. We observe that the blocked MM algorithm can attain 8.4× and 8.2× better performance and performance per watt as compared to the nonblocked MM algorithm for $p = 4$ and $p = 16$, respectively, when $n = 1024$, $b = 64$, and $b' = 4$. Similarly, the performance and performance per watt improvements for the blocked MM algorithm over the nonblocked MM algorithm is 10.8× and 11.5× for $p = 4$ and $p = 16$, respectively, when $n = 2048$, $b = 64$, and $b' = 4$.

Results for the parallelized MM algorithm also verify that the attainable performance per watt for smaller data sizes is better than the attainable performance per watt for larger data sizes because smaller data sizes require fewer memory accesses as compared to larger data sizes. For example, the attainable performance per watt is 15 % and 16 % higher for $p = 4$ and $p = 16$, respectively, for $n = 1024$ as compared to that for $n = 2048$. We point out that ideal speedups may not be attainable with different kernels, such as comparison-based sorting and some real-world applications, because of frequent memory accesses, data sharing, load imbalance, and inherent overheads in parallelization, such as synchronization and data communication between tiles.

Table 11 Performance per watt of the parallelized blocked MM algorithm for a different number of tiles p for the TILEPro64 for different matrix sizes n, block sizes b, and subblock sizes b'

n	b	b'	Perf/W$_{\text{B}}^{(4)}$ (MOPS/W)	Perf/W$_{\text{B-O3}}^{(4)}$ (MOPS/W)	Perf/W$_{\text{B}}^{(16)}$ (MOPS/W)	Perf/W$_{\text{B-O3}}^{(16)}$ (MOPS/W)
1024	32	4	10.57	58.2	25.29	134.07
	64	4	10.6	59.12	25.38	137.77
	128	4	10.62	59.44	12.7	70.34
1024	32	8	12.38	68.33	29.59	159.81
	64	8	12.41	69.18	29.68	163.74
	128	8	12.42	69.62	14.82	81.2
1024	32	16	11.32	44.82	26.89	96.5
	64	16	11.35	45.18	26.96	97.45
	128	16	11.35	45.3	13.31	50.96
2048	32	4	10.31	50.67	24.65	115.81
	64	4	10.36	51.46	24.75	118.47
	128	4	10.34	51.25	24.69	117.51
2048	32	8	9.29	24.85	21.98	52.28
	64	8	9.3	24.9	21.96	52.36
	128	8	9.3	24.9	21.95	52.35
2048	32	16	7.56	17.45	17.64	30.3
	64	16	7.57	17.72	17.68	30.99
	128	16	7.57	17.46	17.68	30.27

Table 12 Performance and performance per watt for a parallelized nonblocked (NB) and a parallelized blocked (B) MM algorithm for a different number of tiles p for the TILEPro64 for different matrix sizes n. S_{B} denotes the speedup for the blocked MM algorithm

n	p	$T_{\text{NB-O3}}^{(p)}$ (s)	$T_{\text{B-O3}}^{(p)}$ (s)	S_{B}	$T_{\text{NB-O3}}^{(p)}/T_{\text{B-O3}}^{(p)}$	Perf$_{\text{NB-O3}}^{(p)}$ (MOPS)	Perf/W$_{\text{NB-O3}}^{(p)}$ (MOPS/W)	Perf$_{\text{B-O3}}^{(p)}$ (MOPS)	Perf/W$_{\text{B-O3}}^{(p)}$ (MOPS/W)
1024	4	47.55	5.64	4.0	8.4	45.16	7.01	380.76	59.12
1024	16	11.84	1.45	15.6	8.2	181.38	16.87	1,481.02	137.77
2048	4	558.78	51.84	4.0	10.8	30.74	4.77	331.4	51.46
2048	16	154.64	13.49	16	11.5	111.1	10.33	1,273.53	118.47

Table 12 also illustrates the impact of algorithmic choices on parallel performance and performance per watt by providing a comparison between parallelized blocked (B) and parallelized nonblocked (NB) MM algorithms. The results reveal that the parallelized blocked MM algorithm attains ideal or near-ideal speedups when block sizes are selected appropriately (e.g., ideal or near-ideal speedups are attained for $b = 64$ and $b' = 4$ whereas the attained speedups are far from ideal for inappropriate block sizes). Results indicate that with appropriate block size selection, the parallelized blocked MM algorithm provides better performance and performance per watt as compared to the non-blocked MM algorithm. For example, the parallelized blocked MM algorithm attains 743 % and 717 % better performance per watt than the parallelized nonblocked MM algorithm for $p = 4$ and $p = 16$, respectively, when $n = 1024$ ($b = 64$ and $b' = 4$ for the blocked MM algorithm).

6.3.2 Horizontal communication

Conventional inter-thread communication on shared memory can be inefficient due to the memory subsystem's limited bandwidth. Although all contemporary architectures provide vertical data communication (communication between different levels of the memory hierarchy), many TMAs (e.g., Tilera's TMAs) also support horizontal data communication (communication between different caches at the same level). Horizontal data communication provides efficient interthread communication by leveraging low on-chip communication latency and high on-chip bandwidth. To illustrate horizontal data communication for attaining high performance and performance per watt, we implement Cannon's MM algorithm [40] on the TILEPro64.

In Cannon's algorithm, only neighboring tiles communicate with each other using horizontal communication. Horizontal communication in Cannon's algorithm minimizes level two cache and main memory accesses whereas communication with neighboring tiles minimizes the network contention [1]. We also use blocking with Cannon's algorithm so that the submatrix blocks fit in the caches. For Cannon's algorithm, we also experiment with separate temporary submatrices (D, E, F) to store the blocked submatrices to overlap communication and computation, however, we are able to attain similar performance without using temporary matrices because the memory footprint/size increases by using separate temporary matrices. Hence, we present results for Cannon's algorithm that does not use separate temporary submatrices.

Tables 13 and 14 depict the performance and performance per watt, respectively, of parallelized blocked Cannon's algorithm for MM running on $p = 4$ and $p = 16$ tiles for different block sizes b and b', with compiler optimization level -O3 and without compiler optimizations. The results reveal that Cannon's algorithm attains close to ideal speedups for appropriate block and subblock sizes. We also observe superlinear speedup for $n = 2048$, $p = 4$, $b = 64$, and $b' = 4$. Superlinear speedups are achieved when a working set's data completely fits in the tiles' combined caches but was unable to fit in an individual tile's cache. The larger combined cache size, as well as the horizontal communication exploited by Cannon's algorithm, help in attaining superlinear speedup for appropriate block and subblock sizes. Results show that compiler optimizations can significantly enhance performance and performance per watt in addition to the horizontal communication leveraged in Cannon's algorithm. For example, compiler optimization level -O3 yields $5\times$ and $4.8\times$ improvements when Cannon's algorithm is executed on $p = 4$ and $p = 16$, respectively, when $n = 1024$, $b = 64$, and $b' = 4$. Similarly, compiler optimization level -O3 yields $4.5\times$ and $4.2\times$ improvements when the blocked Cannon's MM algorithm is executed on $p = 4$ and $p = 16$, respectively, when $n = 2048$, $b = 64$, and $b' = 4$.

Table 13 Performance of a parallelized blocked Cannon's algorithm for MM for a different number of tiles p for the TILEPro64 for different matrix sizes n, block sizes b, and subblock sizes b'

n	b	b'	$T_B^{(4)}$ (s)	$T_{B\text{-}O3}^{(4)}$ (s)	$S^{(4)}$	$T_B^{(16)}$ (s)	$T_{B\text{-}O3}^{(16)}$ (s)	$S^{(16)}$	$\text{Perf}_B^{(4)}$ (MOPS)	$\text{Perf}_{B\text{-}O3}^{(4)}$ (MOPS)	$\text{Perf}_B^{(16)}$ (MOPS)	$\text{Perf}_{B\text{-}O3}^{(16)}$ (MOPS)
1024	32	4	28.82	5.79	3.3	7.25	1.52	12.7	74.51	370.9	296.2	1,412.8
	64	4	28.73	5.68	3.3	7.22	1.5	12.9	74.75	378.1	297.43	1,431.6
	128	4	28.7	5.65	3.3	7.24	1.49	13.0	74.82	380.1	296.61	1,441.3
1024	32	8	24.28	5	3.9	6.13	1.3	14.8	88.45	429.5	350.32	1,651.91
	64	8	24.18	4.89	3.9	6.1	1.28	15.1	88.81	439.16	352.05	1,677.72
	128	8	24.15	4.84	4.0	6.1	1.26	15.3	88.92	443.7	352.05	1,704.35
1024	32	16	26.89	7.48	2.6	7.14	2.28	8.5	79.86	287.1	300.77	941.88
	64	16	27.03	7.47	2.6	7.1	2.26	8.5	79.45	287.48	302.46	950.21
	128	16	26.9	7.5	2.6	7	2.23	8.6	79.83	286.33	306.78	963.0
2048	32	4	238.64	52.79	4.0	60.33	14.36	15.0	72.0	325.44	284.76	1,196.37
	64	4	236.42	52.1	4.1	60.1	14.11	15.3	72.67	329.75	285.85	1,217.57
	128	4	237.09	53	4.0	60.04	14.55	14.8	72.46	324.15	286.14	1,180.75
2048	32	8	264.62	106.98	2.0	70.23	30.83	7.0	64.92	160.59	244.62	557.24
	64	8	265.77	107.12	2.0	69.68	31.93	6.7	64.64	160.38	246.55	538.05
	128	8	266.05	107.84	2.0	69.87	32.27	6.7	64.57	159.31	245.88	532.38
2048	32	16	331.5	150.82	1.4	86.32	51.89	4.1	51.82	113.91	199.02	331.08
	64	16	331.04	150.82	1.4	86.06	52.15	4.1	51.9	113.91	199.63	329.43
	128	16	331.29	151.03	1.4	85.87	52.33	4.1	51.86	113.75	200.07	328.3

Comparison of Tables 10 and 13 reveals that our parallelized blocked MM algorithm always attains equal or better performance than the parallelized blocked Cannon's algorithm on the TILEPro64 for our selected block and subblock sizes $b = 64$ and $b' = 4$. One explanation for the blocked MM algorithm's better performance than Cannon's algorithm for optimal block and subblock sizes could be that Cannon's algorithm stresses the on-chip network both for communication with external memory and for inter-tile communication, whereas the blocked MM algorithm uses the on-chip network only for communication with external memory.

6.3.3 Feedback-based optimizations

After observing that our parallelized blocked MM algorithm attains performance that is comparable to the blocked Cannon's algorithm, we investigate the impact of feedback-based optimizations on the performance of the parallelized blocked MM algorithm. Tables 15, 16, 17 depict the execution time, performance, and performance per watt, respectively, of our parallelized blocked MM algorithm with compiler optimization level -O3 and with feedback-based optimizations. Results reveal that feedback-based optimizations provide only a negligible performance improvement over compiler optimization level -O3 for the TILEPro64, which corroborates

Table 14 Performance per watt of a parallelized blocked Cannon's algorithm for MM for a different number of tiles p for the TILEPro64 for different matrix sizes n, block sizes b, and subblock sizes b'

n	b	b'	$\text{Perf/W}_{\text{B}}^{(4)}$ (MOPS/W)	$\text{Perf/W}_{\text{B-O3}}^{(4)}$ (MOPS/W)	$\text{Perf/W}_{\text{B}}^{(16)}$ (MOPS/W)	$\text{Perf/W}_{\text{B-O3}}^{(16)}$ (MOPS/W)
1024	32	4	11.57	57.59	27.55	131.42
	64	4	11.61	58.71	27.67	133.17
	128	4	11.62	59.02	27.59	134.07
1024	32	8	13.73	66.69	32.59	153.67
	64	8	13.79	68.19	32.75	156.1
	128	8	13.81	68.9	32.75	158.54
1024	32	16	12.4	44.58	27.98	87.62
	64	16	12.34	44.65	28.14	88.39
	128	16	12.4	44.46	28.54	89.58
2048	32	4	11.18	50.53	26.49	111.29
	64	4	11.28	51.2	26.59	113.26
	128	4	11.25	50.33	26.62	109.84
2048	32	8	10.08	24.94	22.76	51.84
	64	8	10.04	24.9	22.93	50.05
	128	8	10.03	24.74	22.87	49.52
2048	32	16	8.05	17.69	18.51	30.8
	64	16	8.06	17.69	18.57	30.64
	128	16	8.05	17.66	18.61	30.54

Table 15 Execution time of the parallelized blocked MM algorithm for a different number of tiles p for the TILEPro64 for different matrix sizes n

n	b	b'	$T_{\text{B}}^{(4)}$ (s)	$T_{\text{B-O3}}^{(4)}$ (s)	$T_{\text{B-FBO}}^{(4)}$ (s)	$T_{\text{B}}^{(16)}$ (s)	$T_{\text{B-O3}}^{(16)}$ (s)	$T_{\text{B-FBO}}^{(16)}$ (s)
1024	64	4	31.42	5.64	5.62	7.87	1.45	1.44
2048	64	4	257.59	51.84	50.73	64.56	13.49	13.23

our findings for a single tile of the TILEPro64. These results indicate that compiler optimization level -O3 provides close to peak attainable performance for the TILEPro64 when proper load balancing and cache blocking is used.

6.3.4 Peak attained parallel performance

Leveraging our algorithmic and compiler optimizations, we are able to attain peak performance of 7.2 GOPS for the MM algorithm running on $p = 57$ tiles of the TILEPro64. We also quantify the peak attainable performance for FP benchmarks on the TILEPro64. For FP performance benchmarking, we use an embarrassingly

Table 16 Performance of the parallelized blocked MM algorithm for a different number of tiles p for the TILEPro64 for different matrix sizes n

n	b	b'	$\text{Perf}_{\text{B}}^{(4)}$ (MOPS)	$\text{Perf}_{\text{B-O3}}^{(4)}$ (MOPS)	$\text{Perf}_{\text{B}}^{(16)}$ (MOPS)	$\text{Perf}_{\text{B-O3}}^{(16)}$ (MOPS)	$\text{Perf}_{\text{B-FBO}}^{(4)}$ (MOPS)	$\text{Perf}_{\text{B-FBO}}^{(16)}$ (MOPS)
1024	64	4	68.35	380.76	272.87	1, 481.02	382.11	1, 491.31
2048	64	4	66.69	331.4	266.11	1, 273.53	338.65	1, 298.55

Table 17 Performance per watt of the parallelized blocked MM algorithm for a different number of tiles p for the TILEPro64 for different matrix sizes n

n	b	b'	$\text{Perf/W}_{\text{B}}^{(4)}$ (MOPS/W)	$\text{Perf/W}_{\text{B-O3}}^{(4)}$ (MOPS/W)	$\text{Perf/W}_{\text{B}}^{(16)}$ (MOPS/W)	$\text{Perf/W}_{\text{B-O3}}^{(16)}$ (MOPS/W)	$\text{Perf/W}_{\text{B-FBO}}^{(4)}$ (MOPS/W)	$\text{Perf/W}_{\text{FBO}}^{(16)}$ (MOPS/W)
1024	64	4	10.61	59.12	25.38	137.77	59.33	138.73
2048	64	4	10.36	51.46	24.75	118.47	52.58	120.8

parallel benchmark that generated normally distributed random variates based on Box-Muller's algorithm. We are able to attain 618.4 MFLOPS for the embarrassingly parallel benchmark running on 57 tiles of the TILEPro64. We observe that the TILEPro64 delivers higher performance for benchmarks with integer operations as compared to the benchmarks with FP operations. For example, we were able to attain 11.6× better performance for integer benchmarks as compared to FP benchmarks while running the benchmarks on 57 tiles. The better performance and performance per watt for integer operations on the TILEPro64 is because Tilera's TMAs do not contain dedicated FP units. We point out that benchmarks could not be run on more than 57 tiles of the TILEPro64 as remaining tiles are reserved by the TILEPro64 for other purposes [47].

7 Conclusions and insights

This work provides an overview of tiled many-core architectures (TMAs) and discusses contemporary TMA chips, including Intel's TeraFLOPS research chip, IBM's C64, and Tilera's TILEPro64. Research on TMAs indicates that the TeraFLOPS research chip is suitable for kernel studies whereas IBM's C64 and Tilera's TILEPro64 can run full-scale applications. Our work focuses on Tilera's TILEPro64 for demonstrating performance optimizations on TMAs. We highlight platform considerations for parallel performance optimizations, such as chip locality, cache locality, tile locality, translation look-aside buffer locality, and memory balancing. We elaborate on compiler-based optimizations for attaining high performance, such as function inlining, alias analysis, loop unrolling, loop nest optimizations, software pipelining, and feedback-based optimizations.

To demonstrate high-performance optimizations on TMAs, we optimize dense matrix multiplication (MM) on Tilera's TILEPro64. Results verify that an algorithm must consider the underlying architectural features, such as the cache sizes, in order to maximize the performance attained from TMAs. For example, our blocked MM algorithm, which exploits cache blocking, provides $6\times$ and $10\times$ performance and performance per watt improvements over a nonblocked MM algorithm for $n = 1024$ and $n = 2048$, respectively (n denotes the matrix size). Results reveal that the performance advantage of blocking increases as the data size increases. Experiments verify that the best blocking granularity for an algorithm is determined by experiments whereas calculations can only specify an upper bound for block sizes in most cases. We parallelize our blocked MM algorithm on multiple tiles to enhance performance by exploiting thread-level parallelism. We are able to attain linear speedups using parallelization and proper load balancing on Tilera's TMAs. Experiments reveal that appropriate use of the cache subsystem (e.g., by blocking) also significantly improves attainable performance and performance per watt gains from parallelism. For example, the blocked MM algorithm attains 743 % and 717 % better performance per watt than the non-blocked MM algorithm for $p = 4$ and $p = 16$, respectively, when $n = 1024$ (p denotes number of tiles).

Experiments indicate that blocking and parallelization alone are not sufficient to achieve maximum attainable performance from TMAs and compiler-based optimizations can provide tremendous enhancements in attainable performance and performance per watt. Results show that compiler optimization level -O3 yields $5.6\times$ and $5.4\times$ performance and performance per watt improvements when the blocked MM algorithm is executed on $p = 4$ and $p = 16$, respectively. Furthermore, as with traditional processors, appropriate use of compiler directives can enhance attainable performance from TMAs. For example, compiler optimization directives with optimization level -O2 can increase the performance by $1.8\times$ (77 %) as compared to not using any compiler directives for Tilera's TILEPro64. Experiments on the TILE64 and TILEPro64 suggest that advanced compiler optimization techniques, such as feedback-based optimizations, can improve performance on some TMAs, but cannot contribute much to the performance enhancements on other TMAs. For example, feedback-based optimizations provide $1.3\times$ and $1.4\times$ performance and performance per watt improvements for $n = 1024$ and $n = 2048$, respectively, for the TILE64 whereas negligible performance improvements are observed for the TILEPro64.

Results demonstrate that an algorithm exploiting horizontal communication, such as Cannon's algorithm, provides an effective means of attaining high performance on TMAs. However, our results reveal the our parallelized blocked MM algorithm, which is much simpler than Cannon's algorithm, is able to obtain comparable performance as that of Cannon's algorithm. Results suggest that optimized code that takes advantage of the memory hierarchy sizes via blocking can attain comparable performance to the algorithms that exploit horizontal communication on TMAs.

Leveraging our algorithmic and compiler optimizations, we are able to attain peak performance of 7.2 GOPS for the MM algorithm running on 57 tiles of the TILEPro64. We are able to attain 618.4 MFLOPS for an embarrassingly parallel floating point benchmark running on 57 tiles of the TILEPro64.

Our programming experience with TMAs suggest that TMAs can deliver scalable performance per watt for applications with sufficient thread-level parallelism (TLP). These application domains include, but are not limited to, networking, security, video processing, and wireless networks. In networking applications, packets of different flows have little or no dependencies among them, and thus enable exploitation of TLP to the fullest extent. An example of a security application exploiting available TLP effectively could be encryption of different data blocks in parallel on different tiles. Similarly, in image processing applications, different blocks of pixels can be operated on in parallel on different tiles. TMAs in base stations for wireless networks can handle processing for different users on different tiles exploiting inherent TLP in the application. For all these application domains, the working set can be chosen to fit in the TMA's tiles' caches based on appropriate cache blocking. As demonstrated in our experiments, different compiler optimization flags as well as compiler directives (`pragmas`) can enhance the attainable performance and performance per watt for these applications. Furthermore, feedback-based optimizations, where feasible, can further enhance the attainable performance and performance per watt for these applications.

Research and programming experience with TMAs suggest some hardware/software optimizations. Research suggests that a portion of on-chip transistors should be used for on-die memory to provide sustainable high performance. Programming experience advocates that application programmers input need to be considered in the design of many-core chips as small easier to incorporate changes in the instruction set can potentially have a large impact on the chip programmability. For example, incorporation of `jump` instruction in Intel's TeraFLOPS research chip could have allowed addition of nested loops [12]. Research on TMAs reveals the scalability advantages of message passing architectures that allow data sharing through explicit messages rather than through a shared address space, which enables easy avoidance of race conditions. However, since software development with message passing alone can be complex, a TMA supporting both shared memory and message passing programming paradigm can benefit large scale applications development. This hybrid messaging passing-shared memory programming paradigm would enable programmers to take advantage of the two programming paradigms' features depending on the application structure and decomposition.

Acknowledgements This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Space and Naval Warfare Systems Command (SPAWAR N66001-11-1-4103), the Office of Naval Research (ONR R16480), and the National Science Foundation (NSF) (CNS-0953447 and CNS-0905308). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSERC, the SPAWAR, the ONR, and the NSF. Furthermore, the views expressed are those of the author(s) and do not reflect the official policy or position of the Department of Defense or the US Government. We would like to acknowledge Dr. Alan D. George, Director of the NSF Center of High-Performance Reconfigurable Computing (CHREC) at the University of Florida, Gainesville, Florida, USA, for providing access to CHREC resources and Tilera's TILE64 and TILEPro64 for this work as well as discussions on high-performance computing with the leading author of this article.

Appendix: Matrix multiplication algorithms' code snippets for Tilera's TILEPro64

This appendix section provides code snippets of our matrix multiplication algorithms for Tilera's TILEPro64. The code snippets are presented selectively to provide an understanding of our algorithms and some portions of the code are skipped for conciseness.

A.1 Serial non-blocked matrix multiplication algorithm

A.1.1 SerialNonBlockedMM.h

```
#ifndef MATRIXMULTIPLICATION_H_
#define MATRIXMULTIPLICATION_H_

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

#define     m    1024   // specify the dimension of matrices
#define     n    1024   // specify the dimension of matrices


void MatrixMultiplication(int *A, int *B, int *C);

#endif /* MATRIXMULTIPLICATION_H_ */
```

A.1.2 SerialNonBlockedMM.c

```
#include "SerialNonBlockedMM.h"

int main(int argc, char **argv)
{
   int *A, *B, *C;
   // pointers pointing to input matrices A and B and output matrix C

   A = (int *) malloc (m * n * sizeof(int));
   B = (int *) malloc (m * n * sizeof(int));
   C = (int *) malloc (m * n * sizeof(int));

   // initialize matrices
             .
             .
             .
   MatrixMultiplication(A, B, C);

   return 0;

} /* main function */

// integer matrix multiplication kernel
void MatrixMultiplication(int *A, int *B, int *C)
{
   int i, j, k;
   for (i = 0; i < n; i++)
   {
      for (j = 0; j < n; j++)
      {
         for (k = 0; k < n; k++)
         {
            C[i*n+j] = C[i*n+j] + (A[i*n+k] * B[k*n+j]);
         }
      }
   }
} /* MatrixMultiplication function */
```

A.2 Serial blocked matrix multiplication algorithm

A.2.1 SerialBlockedMM.h

```
#ifndef MATRIXMULTIPLICATION_H_
#define MATRIXMULTIPLICATION_H_

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

#define     m    1024   // specify the dimension of matrices
#define     n    1024   // specify the dimension of matrices
#define     bs   64     // block size for L2
#define     ssb  4      // block size for L1

void MatrixMultiplicationBlockedMain(int *A, int *B, int *C, int N);

void MatrixMultiplicationSubBlockedL2(int *A, int *B, int *C, int N);

void MatrixMultiplicationSubBlockedL1(int *A, int *B, int *C);

#endif /* MATRIXMULTIPLICATION_H_ */
```

A.2.2 SerialBlockedMM.c

```
#include "SerialBlockedMM.h"

int main(int argc, char **argv)
{
   int *A, *B, *C;

   // The algorithm tiles n x n original matrix in NxN sub-matrices
   int N = n/bs;

   A = (int *) malloc (m * n * sizeof(int));
   B = (int *) malloc (m * n * sizeof(int));
   C = (int *) malloc (m * n * sizeof(int));

   // initialize matrices
              :
              :
   MatrixMultiplicationBlockedMain(A, B, C, N);

   return 0;

} /* main function */

void MatrixMultiplicationBlockedMain(int *A, int *B, int *C, int N)
{
   int i, j, k;
   int tempbsn, tempibsn, tempjbs, tempC;
   tempbsn = bs * n;
   for (i = 0; i < N; i++)
   {
      tempibsn = i * tempbsn;
      for (j = 0; j < N; j++)
      {
         tempjbs = j*bs;
         tempC = tempibsn + tempjbs;

         for (k = 0; k < N; k++)
         {
            MatrixMultiplicationSubBlockedL2(&A[tempibsn + k * bs],
                        &B[k * tempbsn + tempjbs], &C[tempC], N);
         }
      }
   }
} /* MatrixMultiplicationBlockedMain function */

void MatrixMultiplicationSubBlockedL2(int *A, int *B, int *C, int N)
{
   int i, j, k, M;
   int tempissbbsN, tempssbbsN, tempjssb, tempC;
   tempssbbsN = ssb * bs * N;
   M = bs/ssb;

   for (i = 0; i < M; i++)
   {
      tempissbbsN = i * tempssbbsN;

      for (j = 0; j < M; j++)
      {
         tempjssb = j * ssb;
         tempC = tempissbbsN + tempjssb;

         for (k = 0; k < M; k++)
         {
            MatrixMultiplicationSubBlockedL1(&A[tempissbbsN + k * ssb],
                        &B[k * tempssbbsN + tempjssb], &C[tempC]);
         }
      }
   }
} /* MatrixMultiplicationSubBlockedL2 function */

void MatrixMultiplicationSubBlockedL1(int *A, int *B, int *C)
{
   int i, j, k, temp;
   for (i = 0; i < ssb; i++)
   {
      for (j = 0; j < ssb; j++)
      {
         temp = 0;
         for (k = 0; k < ssb; k++)
         {
            temp += A[i*n+k] * B[k*n+j];
         }
         // storing the value of temp in C[i*n+j]
         C[i*n+j] += temp;
      }
   }
} /* MatrixMultiplicationSubBlockedL1 function */
```

A.3 Parallel blocked matrix multiplication algorithm

A.3.1 ParallelBlockedMM.h

```
#ifndef MATRIXMULTIPLICATION_H_
#define MATRIXMULTIPLICATION_H_
```

```
#include <stdio.h>
#include <stdlib.h>
#include <ilib.h>      // Tilera's parallel API ilib
#include <tmc/cmem.h>
#include <time.h>
#include <sys/time.h>

#define     m    1024   // specify the dimension of matrices
#define     n    1024   // specify the dimension of matrices
#define     bs   64     // block size for L2
#define     ssb  4      // block size for L1
#define     NUM_TILES         16
#define     MASTER_RANK       0

// The implementation of the following functions is similar to the
// serial blocked MM algorithm and are skipped in the ParallelBlockedMM.c
// file for brevity
void MatrixMultiplicationSubBlockedL2(int *A, int *B, int *C, int N);

void MatrixMultiplicationSubBlockedL1(int *A, int *B, int *C);

#endif /* MATRIXMULTIPLICATION_H_ */
```

A.3.2 ParallelBlockedMM.c

```
#include "ParallelBlockedMM.h"

int main(int argc, char **argv)
{
   int *A, *B, *C;

   ilibStatus status;

   int processRank = 0;
   int N = n/bs;      // number of sub-blocks of main matrix of size n x n
                      // is equal to N^2; each NxN is of size bs x bs

   ilib_init();       // initializes ilib library state

   // creating parallel processes
   if (ilib_proc_go_parallel(NUM_TILES, NULL) != ILIB_SUCCESS)
      ilib_die("Failed to go parallel.");

   // determining rank of processes
   processRank = ilib_group_rank(ILIB_GROUP_SIBLINGS);

   // Let process with rank 0 (root process) handle the initialization
   // of matrices
   if(processRank == 0)
   {
      // process rank 0 (root process) allocates shared memory for
      // matrices A, B, and C
      tmc_cmem_init(0);
      A = (int *) tmc_cmem_malloc (m * n * sizeof(int));
      B = (int *) tmc_cmem_malloc (m * n * sizeof(int));
      C = (int *) tmc_cmem_malloc (m * n * sizeof(int));

      // initialize matrices
              ⋮
      // Performing memory fence to guarantee that the initialized
      // array values are visible to other processes
      ilib_mem_fence();
   }

   // broadcasts the shared memory addresses of A, B, and C
   // from tile 0/process 0 to the other tiles
   if(ilib_msg_broadcast(ILIB_GROUP_SIBLINGS, MASTER_RANK,
                         &A, sizeof(A), &status) != ILIB_SUCCESS)
      ilib_die("Failed to broadcast address of A");

   if(ilib_msg_broadcast(ILIB_GROUP_SIBLINGS, MASTER_RANK,
                         &B, sizeof(B), &status) != ILIB_SUCCESS)
      ilib_die("Failed to broadcast address of B");

   if(ilib_msg_broadcast(ILIB_GROUP_SIBLINGS, MASTER_RANK,
                         &C, sizeof(C), &status) != ILIB_SUCCESS)
      ilib_die("Failed to broadcast address of C");

   // This barrier ensures that the broadcast of shared addresses
   // is complete before further processing
   ilib_msg_barrier(ILIB_GROUP_SIBLINGS);

   for (i = 0; i < N; i++)
   {
      for (j = 0; j < N; j++)
      {
         if(processRank == ((i + j) % NUM_TILES))
         {
            for (k = 0; k < N; k++)
            {
               MatrixMultiplicationSubBlockedL2(&A[i * bs * n + k * bs],
                      &B[k * bs * n + j * bs], &C[i * bs * n + j * bs], N);
            }
         }
      }
   }

   // This barrier makes sure that all the processes have finished the
   // assigned computations of matrix multiplication
   ilib_msg_barrier(ILIB_GROUP_SIBLINGS);

   // The process with rank 0 frees the allocated memory
   if(processRank == 0)
   {
      free(A);
      free(B);
      free(C);
   }

   ilib_finish();

   return 0;

} /* main function */
```

A.4 Parallel blocked cannon's algorithm for matrix multiplication

A.4.1 ParallelBlockedCannonMM.h

```
#ifndef MATRIXMULTIPLICATION_H_
#define MATRIXMULTIPLICATION_H_

#include <stdio.h>
#include <stdlib.h>
#include <ilib.h>      // Tilera's parallel API ilib
#include <tmc/cmem.h>
#include <time.h>
#include <sys/time.h>

#define     m    1024   // specify the dimension of matrices
#define     n    1024   // specify the dimension of matrices
#define     bs   64     // block size for L2
#define     ssb  4      // block size for L1
#define     NUM_TILES         16
#define     N                 4      // equal to sqrt(NUM_TILES)
#define     MASTER_RANK       0

void MatrixMultiplicationSubBlockedL2(int *A, int *B, int *C, int L2B);

void MatrixMultiplicationSubBlockedL1(int *A, int *B, int *C);

#endif /* MATRIXMULTIPLICATION_H_ */
```

A.4.2 ParallelBlockedCannonMM.c

```
#include "ParallelBlockedCannonMM.h"

int main(int argc, char **argv)
{
   int r, c, s;      // loop variables for Cannon Rounds
   int loop;

   int *A, *B, *C;

   ilibStatus status;

   int processRank = 0;

   int pi, pj;  // pi for i index of tile rank and pj for j index of tile rank

   // operates on this matrix size in one Cannon Round
   int CannonMatrix = bs * N;
   int L2B = n/bs;
   int M = n/CannonMatrix;

   int Ai, Aj, Bi, Bj, Ci, Cj;     // (i,j) coordinates of blocks
   int As, Bs, Cs;                 // start indices of blocks

   int tempnbs = n*bs;
   int pipluspj= 0;

   ilib_init();

   // creating parallel processes
```

```
if (ilib_proc_go_parallel(NUM_TILES, NULL) != ILIB_SUCCESS)
   ilib_die("Failed to go parallel.");

processRank = ilib_group_rank(ILIB_GROUP_SIBLINGS);

pi = processRank/N;
pj = processRank % N;
pipluspj = pi + pj;

// Let process with rank 0 (root process) handle the initialization
// of matrices
if(processRank == 0)
{
   // process rank 0 (root process) allocates shared memory for
   // matrices A, B, and C
   tmc_cmem_init(0);
   A = (int *) tmc_cmem_malloc (m * n * sizeof(int));
   B = (int *) tmc_cmem_malloc (m * n * sizeof(int));
   C = (int *) tmc_cmem_malloc (m * n * sizeof(int));

   // initialize matrices
           .
           .
           .
   // Performing memory fence to guarantee that the initialized
   // array values are visible to other processes
   ilib_mem_fence();
}

// broadcasts the shared memory addresses of A, B, and C
// from tile 0/process 0 to the other tiles
if(ilib_msg_broadcast(ILIB_GROUP_SIBLINGS, MASTER_RANK,
                      &A, sizeof(A), &status) != ILIB_SUCCESS)
   ilib_die("Failed to broadcast address of A");

if(ilib_msg_broadcast(ILIB_GROUP_SIBLINGS, MASTER_RANK,
                      &B, sizeof(B), &status) != ILIB_SUCCESS)
   ilib_die("Failed to broadcast address of B");

if(ilib_msg_broadcast(ILIB_GROUP_SIBLINGS, MASTER_RANK,
                      &C, sizeof(C), &status) != ILIB_SUCCESS)
   ilib_die("Failed to broadcast address of C");

// This barrier ensures that the broadcast of shared addresses is complete
// before further processing
ilib_msg_barrier(ILIB_GROUP_SIBLINGS);

int temprN = 0, tempcN = 0, tempsN = 0;

for (r = 0; r < M; r++)
{
   temprN = r*N;
   for (c = 0; c < M; c++)
   {
      tempcN = c*N;
      for (s = 0; s < M; s++)
      {
         tempsN = s*N;
         Ai = temprN + pi;
         Aj = tempsN + (pipluspj) % N;
         Bi = tempsN + (pipluspj) % N;
         Bj = tempcN + pj;
```

```
            if(s == 0)
            {
               Ci = temprN + pi;
               Cj = tempcN + pj;
            }

            As = (Ai * tempnbs) + (Aj * bs);
            Bs = (Bi * tempnbs) + (Bj * bs);
            Cs = (Ci * tempnbs) + (Cj * bs);

            for (loop = 1; loop <= N; loop++)
            {
               // performs matrix multiplication on the local block
               MatrixMultiplicationSubBlockedL2(&A[As], &B[Bs], &C[Cs], L2B);
               if (loop != N)
               {
                  Ai = temprN + pi;
                  Aj = tempsN + (pipluspj + loop) % N;
                  Bi = tempsN + (pipluspj + loop) % N;
                  Bj = tempcN + pj;
                  As = (Ai * tempnbs) + (Aj * bs);
                  Bs = (Bi * tempnbs) + (Bj * bs);
               }
            }
         }
      }
   }

   ilib_msg_barrier(ILIB_GROUP_SIBLINGS);

   // The process with rank 0 frees the allocated memory
   if(processRank == 0)
   {
      free(A);
      free(B);
      free(C);
   }

   ilib_finish();

   return 0;

} /* main function */

void MatrixMultiplicationSubBlockedL2(int *A, int *B, int *C, int L2B)
{
   int i, j, k, L1B;   // L1B for L1 cache block size
   int tempissbbsL2B, tempssbbsL2B, tempjssb, tempC;
   tempssbbsL2B = ssb * bs * L2B;
   L1B = bs/ssb;

   for (i = 0; i < L1B; i++)
   {
      tempissbbsL2B = i * tempssbbsL2B;

      for (j = 0; j < L1B; j++)
      {
         tempjssb = j * ssb;
         tempC = tempissbbsL2B + tempjssb;
         for (k = 0; k < L1B; k++)
         {
            MatrixMultiplicationSubBlockedL1(&A[tempissbbsL2B + k * ssb],
                              &B[k * tempssbbsL2B + tempjssb], &C[tempC]);
         }
      }
   }
} /* MatrixMultiplicationSubBlockedL2 function */

void MatrixMultiplicationSubBlockedL1(int *A, int *B, int *C)
{
   int i, j, k, temp, tempin;

   for (i = 0; i < ssb; i++)
   {
      tempin = i*n;
      for (j = 0; j < ssb; j++)
      {
         temp = 0;
         for (k = 0; k < ssb; k++)
         {
            temp += A[tempin+k] * B[k*n+j];
         }
         C[tempin+j] += temp;
      }
   }
} /* MatrixMultiplicationSubBlockedL1 function */
```

References

1. Yuan N, Zhou Y, Tan G, Zhang J, Fan D (2009) High performance matrix multiplication on many cores. In: Proc of the 15th international Euro-Par conference on parallel processing (Euro-Par'09), Delft, The Netherlands, August 2009
2. MAXIMUMPC (2007) Fast forward: multicore vs manycore. June. Available online: http://www.maximumpc.com/article/fast_forward_multicore_vs_manycore
3. Wikipedia (2013) Multi-core processor. February. Available online: http://en.wikipedia.org/wiki/Manycore
4. Tilera (2013) Tilera cloud computing. February. Available online: http://www.tilera.com/solutions/cloud_computing
5. Tilera (2013) Tilera TILEmpower platform. February. Available online: http://www.tilera.com/sites/default/files/productbriefs/TILEProEmpower_PB021_v4.pdf
6. Levy M, Conte T (2009) Embedded multicore processors and systems. IEEE MICRO 29(3):7–9
7. Asanovic K, Bodik R, Demmel J, Keaveny T, Keutzer K, Kubiatowicz J, Morgan N, Patterson K, Sen D, Wawrzynek J, Wessel D, Yelick K (2009) A view of the parallel computing landscape. Commun ACM 52(10):56–67
8. Cuvillo Jd, Zhu W, Gao GR (2006) Landing OpenMP on Cyclops-64: an efficient mapping of OpenMP to a many-core system-on-a-chip. In: Proc of ACM 3rd conference on computing frontiers (CF), Ischia, Italy, May 2006
9. Vangal SR, Howard J, Ruhl G, Dighe S, Wilson H, Tschanz J, Finan D, Singh A, Jacob T, Jain S, Erraguntla V, Roberts C, Hoskote Y, Borkar N, Borkar S (2008) An 80-tile sub-100-W TeraFLOPS processor in 65-nm CMOS. IEEE J Solid-State Circuits 43(1):29–41
10. Musoll E (2010) A cost-effective load-balancing policy for tile-based, massive multi-core packet processors. ACM Trans Embedded Comput Syst 9(3):24
11. Wu N, Yang Q, Wen M, He Y, Ren J, Guan M, Zhang C (2011) Tiled multi-core stream architecture. In: Transactions on high-performance embedded architectures and compilers IV (HiPEAC IV), vol 4, pp 274–293

12. Mattson TG, Wijngaart RVd, Frumkin M (2008) Programming the Intel 80-core network-on-a-chip terascale processor. In: Proc of IEEE/ACM conference on supercomputing (SC), Austin, Texas, November 2008
13. Crowell T (2011) Will 2011 mark the beginning of manycore? January. Available online: http://talbottcrowell.wordpress.com/2011/01/01/manycore/
14. Tilera (2012) Manycore without boundaries: TILEPro64 processor. May. Available online: http://www.tilera.com/products/processors/TILEPRO64
15. Brown R, Sharapov I (2008) Performance and programmability comparison between OpenMP and MPI implementations of a molecular modeling application. In: Lecture notes in computer science, vol 4315. Springer, Berlin, pp 349–360
16. Sun X, Zhu J (1995) Performance considerations of shared virtual memory machines. IEEE Trans Parallel Distrib Syst 6(11):1185–1194
17. Cortesi D (1998) Origin2000 and Onyx2 performance tuning and optimization guide. Available online: http://techpubs.sgi.com/library/dynaweb_docs/0640/SGI_Developer/books/OrOn2_PfTune/sgi_html/index.html
18. Krishnan M, Nieplocha J (2004) SRUMMA: a matrix multiplication algorithm suitable for clusters and scalable shared memory systems. In: Proc of the international parallel and distributed processing symposium (IPDPS), Santa Fe, New Mexico, April 2004
19. Lee H-J, Robertson JP, Fortes J (1997) Generalized Cannon's algorithm for parallel matrix multiplication. In: Proc of the ACM international conference on supercomputing (ICS), Vienna, Austria, July 1997, pp 44–51
20. van de Geijn RA, Watts J (1995) Summa: scalable universal matrix multiplication algorithm. University of Texas at Austin, Tech rep. Available online: http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail&id=oai%3Ancstrlh%3Autexas_cs%3AUTEXAS_CS%2F%2FCS-TR-95-13
21. Li J, Ranka S, Sahni S (2012) GPU matrix multiplication. In: Rajasekaran S (ed) Handbook on multicore computing. CRC Press, Boca Raton
22. More A (2008) A case study on high performance matrix multiplication. Available online: mm-matrixmultiplicationtool.googlecode.com/files/mm.pdf
23. Higham N (1990) Exploiting fast matrix multiplication within the level 3 BLAS. ACM Trans Math Softw 16(4):352–368
24. Goto K, Geijn R (2008) Anatomy of high-performance matrix multiplication. ACM Trans Math Softw 34(3):1–25
25. Nishtala R, Vuduc RW, Demmel JW, Yelick KA (2004) Performance modeling and analysis of cache blocking in sparse matrix vector multiply. Tech rep UCB/CSD-04-1335, EECS Department, University of California, Berkeley. Available online: http://www.eecs.berkeley.edu/Pubs/TechRpts/2004/5535.html
26. Lam MD, Rothberg EE, Wolf ME (1991) The cache performance and optimizations of blocked algorithms. In: Proc of the fourth ACM international conference on architectural support for programming languages and operating systems (ASPLOS), Santa Clara, California, April 1991, pp 63–74
27. Rixner S (2002) Stream processor architecture. Kluwer Academic, Norwell
28. Zhu W, Cuvillo Jd, Gao GR (2005) Performance characteristics of OpenMP language constructs on a many-core-on-a-chip architecture. In: Proc of the 2005 and 2006 international conference on OpenMP shared memory parallel programming (IWOMP'05/IWOMP'06), Eugene, Oregon, June 2005
29. Garcia E, Venetis I, Khan R, Gao G (2010) Optimized dense matrix multiplication on a many-core architecture. In: Proc of the ACM Euro-Par conference on parallel processing
30. Safari S, Fijany A, Diotalevi F, Hosseini F (2012) Highly parallel and fast implementation of stereo vision algorithms on MIMD many-core Tilera architecture. In: Proc of the IEEE aerospace conference, Boston, MA, August 2012, pp 1–11
31. Munir A, Gordon-Ross A, Ranka S (2012) Parallelized benchmark-driven performance evaluation of SMPs and tiled multi-core architectures for embedded systems. In: Proc of the IEEE international performance computing and communications conference (IPCCC), Austin, Texas, December 2012
32. Keckler S, Olukotun K, Hofstee H (2009) Multicore processors and systems. Springer, Berlin
33. Tilera (2012) Manycore without boundaries: TILE64 processor. April. Available online: http://www.tilera.com/products/processors/TILE64
34. Intel (2013) Intel's teraflops research chip. February. Available online: http://download.intel.com/pressroom/kits/Teraflops/Teraflops_Research_Chip_Overview.pdf
35. Hoskote Y, Vangal S, Singh A, Borkar N, Borkar S (2007) A 5-GHz mesh interconnect for a TeraFLOPS processor. IEEE MICRO 27(5):51–61

36. IBM (2012) Linux and Symmetric Multiprocessing, February. Available online: http://www.ibm.com/developerworks/library/l-linux-smp/
37. Tilera (2009) Tile processor architecture overview for the TILEPro series. In: Tilera official documentation. November
38. Tilera (2010) Multicore development environment system programmer's guide. In: Tilera official documentation. March
39. Tilera (2009) Tile processor architecture overview. In: Tilera official documentation. November
40. Kumar V, Grama A, Gupta A, Karypis G (1994) Introduction to parallel computing. Benjamin-Cummings, Redwood City
41. Tilera (2010) Multicore development environment optimization guide. In: Tilera official documentation. March
42. ARM (2012) Cortex-A15 MPCore: technical reference manual. April. Available online: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0438e/DDI0438E_cortex_a15_r3p0_trm.pdf
43. Oracle (2013) Sun studio 12: Fortran programming guide. February. Available online: http://docs.oracle.com/cd/E19205-01/819-5262/aeuic/index.html
44. Mahlke S, Warter N, Chen W, Chang P, Hwu W-m (1991) The effect of compiler optimizations on available parallelism in scalar programs. In: Proc of 20th annual IEEE international conference on parallel processing (ICPP), Austin, Texas, August 1991
45. Williams J, Massie C, George A, Richardson J, Gosrani K, Lam H (2010) Characterization of fixed and reconfigurable multi-core devices for application acceleration. ACM Trans on Reconfigurable Technology and Systems 3(4)
46. Tilera (2010) TILEmPower appliance user's guide. In: Tilera official documentation. January
47. Tilera (2009) Tilera multicore development environment: iLib API reference manual. In: Tilera official documentation. April