

# High Performance Matrix Multiplication on Many Cores

Nan Yuan<sup>1,2</sup>, Yongbin Zhou<sup>1,2</sup>, Guangming Tan<sup>1</sup>, Junchao Zhang<sup>1</sup>, and Dongrui Fan<sup>1</sup>

<sup>1</sup> Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, P. R. China

<sup>2</sup> Graduate University of Chinese Academy of Sciences, Beijing 100039, P. R. China  
{yuannan, ybzhou, jczhang, fandr}@ict.ac.cn, tgm@ncic.ac.cn

**Abstract.** *Moore's Law* suggests that the number of processing cores on a single chip increases exponentially. The future performance increases will be mainly extracted from thread-level parallelism exploited by multi/many-core processors (MCP). Therefore, it is necessary to find out how to build the MCP hardware and how to program the parallelism on such MCP. In this work, we intend to identify the key architecture mechanisms and software optimizations to guarantee high performance for multithreaded programs. To illustrate this, we customize a dense matrix multiplication algorithm on Godson-T MCP as a case study to demonstrate the efficient synergy and interaction between hardware and software. Experiments conducted on the cycle-accurate simulator show that the optimized matrix multiplication could obtain 97.1% (124.3GFLOPS) of the peak performance of Godson-T.

## 1 Introduction

Previous techniques of increasing single thread performance through increased clock frequency and smarter architecture are now hitting the so-called *Power Wall* and *ILP Wall* [2]. Computer industry has widely consensus that future performance increases must largely come from increasing the number of processing cores on a die. This has led to swift changes in computer architectures in recent years: multi-core processors become commodity, and many-core processors [3][6][8][11] have entered the lexicon. Moore's Law suggests that the number of on-chip processing cores doubles every generation. It is anticipated that future microprocessors will accommodate tens, hundreds or even thousands of processing cores, implicating that exploiting *thread level parallelism* (TLP) will become increasingly important.

To present what performance gains might be possible for multithreaded programs on MCPs, we choose to evaluate matrix multiplication on our proposed Godson-T MCP prototype. The matrix multiplication (MM) is a key building block of scientific and engineering applications, while at the same time representing a regular operation which is easy to reason about. Although MM seems very simple, previous work on implementing and optimizing high-performance

parallel MM on MCPs seems not very promising as expected. For instance, the performance of MM on Cyclops-64 processor is 13.9GFLOPS, which is 43.4% of the peak performance[6]. A systolic-like MM algorithm on TRIPS obtains 5.10 FLOPS/Cycle, which is 31.9% of the peak performance[5]. MM on Intel 80-core Terascale processor is observed to run only 37% of the peak performance[8].

The inefficiency of parallel MM kernels on these MCPs motivates us to rethink the challenges we confront when we design and program MCPs. Although many-core processors (MCPs) provide tremendous computational capability, extracting computational power on such processors effectively is not trivial as the feature sizes shrink. To our knowledge, the challenges at least include: (1) as the population of processing cores increase, on-chip data communication (especially the global communication) on the expanding interconnect will become more and more expensive. (2) Insufficient off-chip memory bandwidth makes the computational capability and memory accessing capability potentially unbalanced. The number of transistors per die is increasing at a faster rate than chip signal pins. This disparity will potentially limit the scalability of MCPs.

To address these problems, our method is not to reduce the latencies, but to tolerate them. We demonstrate a solution on *how to build a many-core processor with effective architectural supports*, and *how to develop a parallel algorithm benefits from the hardware*. Our solution presents an efficient synergy and interaction of software and hardware, resulting in nearly peak performance of matrix multiplication on Godson-T.

The remainder of the paper is organized as follows. Section 2 gives a brief introduction of our proposed Godson-T MCP, which we used for our evaluation. Section 3 illustrates the problem and our experimental methodology. Section 4 presents the key optimizations of matrix multiplication kernel, highlighting the underlying architectural supports. Section 5 concludes the paper.

## 2 Overview of Godson-T Many-Core Processor

Godson-T is a processor prototype of many-core architecture designed with 65nm CMOS technology. The early version of Godson-T has been introduced and evaluated in[10][13]. In this work, we make several improvements to support multi-threaded program more efficiently. The overview of Godson-T is shown in Fig. 1, and more architecture details will be demonstrated in Sect. 4 when they are used.

As shown in Fig. 1(a), Godson-T has 64 homogeneous, dual-issue and in-order processing cores running at 1GHz. The 8-pipeline processing core supports 32-bit MIPS ISA (64-bit ISA will be supported in latter version) with synchronization instruction extensions. A floating-point multiply-accumulate operation can be issued to a fully-pipelined function unit in each cycle, so the peak floating-point performance of Godson-T is 128GFLOPS. Each processing core has a 16KB 2-way set-associative private instruction cache and a 32KB local memory. The local memory functions as a 32KB 4-way set-associative private data cache in default. It can also be configured as an explicitly-controlled and globally-addressed

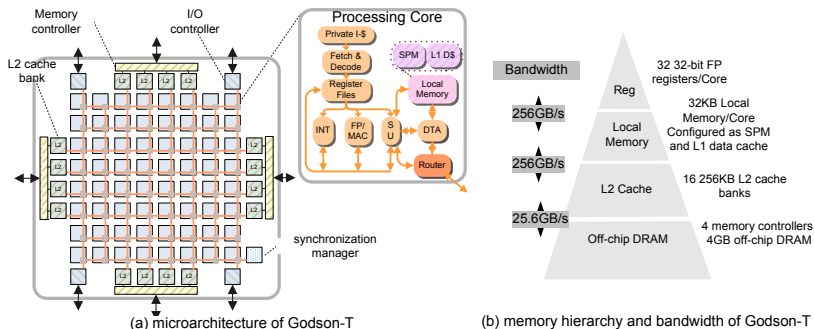


Fig. 1. An overview of Godson-T many-core processor.

Scratched-Pad Memory (SPM), or a hybrid of cache and SPM. A DMA-like co-processor Data Transfer Agent (DTA) is built in each core for fast data communication. When the processing core is doing calculations, DTA can be programmed to manage various data communication patterns in background. In addition, there are 16 address-interleaved L2 cache banks (256KB each) distributed along the perimeter of the chip. The L2 cache is shared by all processing cores and can serve up to 64 outstanding cache accessing requests in total. Four L2 cache banks on the same side of the chip share a memory controller. The memory hierarchy of Godson-T and the data bandwidth between the neighboring levels of the memory hierarchy are shown in Fig. 1(b). A dedicated synchronization manager (SM) is a centralized unit to collect synchronization requests and handles them. It provides architectural support for fast mutual exclusion, barrier and single/wait synchronization. An  $8 \times 8$  packet-switching 2-D mesh network connects all on-chip units. The 128-bit-width mesh network employs deterministic X-Y routing policy and provides total 2TB/s on-chip bandwidth among 64 processing cores.

### 3 Problem and Experimental Method

This work is an in-depth case study of optimizing SGEMM (Single-precision General Matrix Multiplication) on 32-bit Godson-T. SGEMM is the BLAS 3 subroutines used to multiply two single-precision general dense matrices. It includes many variations (e.g. scaling), but we only consider a simple case:  $C = A \times B + C$ , where  $A$ ,  $B$  and  $C$  are square matrices. Modern algorithms for SGEMM are block-based. To parallelize SGEMM, we decompose each of the three matrices into  $t^2$  square blocks and assign each processing core the computation of a number of such blocks. The computation of a  $C_{m,n}$  block requires  $t$  block multiplications and additions according to the following equation:

$$C_{m,n} += \sum_{k=0}^{t-1} A_{m,k} \times B_{k,n} \tag{1}$$

To exploit the spatial locality, it is best to assign the calculation of a  $C_{m,n}$  to a processing core to minimize the number of blocks move around. The detailed algorithm is illustrated in section 4.

Experiments are conducted on Godson-T Architecture Simulator (GAS). GAS is an event-driven and cycle-accurate simulator for fast and precise simulation of Godson-T MCP. The major configurations are listed in Table 1.

**Table 1.** Major architectural parameters for experiments

Processing Core	64 cores running at 1GHz, dual-issue, load-to-use latency=3 cycles, and FMAC latency=4 cycles.
L1 I-Cache	16KB, 2-way set associative, 32B/cacheline, 1 cycle for hit.
Local Memory	Configured to 32KB SPM, 16 64-bit-width SRAM sub-banks with 2 memory ports each (1 for read, 1 for write), 1 cycle for load and store.
L2 Cache	16 banks, total 4MB, 8-way set-associative, 64B/cacheline, 4 cycles for contentionless and hit request.
Memory Controller & Off-chip DRAM	4 memory controllers, running at 1GHz. 64-bit FSB. Each memory controller controls 1GB DDR2-800 DRAM. DRAM clock is 400MHz, $t_{CAS} = 5$ , $t_{RCD} = 5$ , $t_{RP} = 5$ , $t_{RAS}=15$ , $t_{RC} = 24$ measured in memory clock.
Mesh Network	2 cycles contentionless latency per hop.
Synchronization	Each synchronization request to SM and its acknowledgement from SM spend 6~66 cycles. Lock or Barrier request consumes another uncertain cycles until the synchronization dependence is resolved, e.g. barrier request should wait for barrier requests from all involved processing cores are collected in SM.

## 4 Customizing the Algorithm for Godson-T

In this section, we present the experience of optimizing a SGEMM algorithm on Godson-T. Subsection 4.1 focuses on the high-performance sequential blocked SGEMM kernel, which is groundwork of high-performance parallel SGEMM. Subsections 4.2 and 4.3 introduce our parallel algorithm of SGEMM, focusing on key architectural supports and software optimizations to address inefficient on-chip data communication and memory accessing. Subsection 4.4 discusses some related topics of our algorithm.

### 4.1 High-Performance Sequential SGEMM Kernel

In this subsection, we consider a sequential blocked SGEMM kernel, assuming that *all elements of matrix blocks are on the local memory when they are required*. To validate the premise, we configured the local memory of each processing core

as a manually-controlled SPM, and load the required data onto SPM before using it. After that, loading and storing each element of the data on SPM consume deterministic 1 cycle. Since each core has 32KB SPM, we selected the block dimension size  $40 \times 40$  for  $A$ ,  $B$  and  $C$  blocks in (1). The three blocked matrices occupy 18.75KB storage in total. The rest space of SPM is left for data communication between processing cores. We will explain it in the next subsection.

The sequential kernel for  $40 \times 40$  SGEMM on one processing core is initially programmed in common 3 nested loops. Compiled by gcc version 4.3.2 with -O3 optimization, the basic kernel performs only **0.16GFLOPS** even though all data resides in local SPM. A set of manual optimizations can be applied to the sequential kernel. With register tiling, we choose the tile size  $2 \times 4$ ,  $4 \times 2$ , and  $2 \times 2$  for  $A$ ,  $B$  and  $C$  sub-blocks (notated as  $a$ ,  $b$  and  $c$ ), respectively. The innermost loop calculates the sub-block multiplication  $c += a \times b$ . Sub-blocks  $a$ ,  $b$  and  $c$  consume 20 floating-point registers that can fit into 32 floating-point registers in MIPS ISA. Sub-block  $a$  is reused across the innermost loop. Leveraging on dual-issue pipeline of processing core, software pipeline and double buffering are applied to the innermost loop to eliminate pipeline stalls caused by register dependence. When the processing core calculates a sub-block multiplication, the core also stores the old result of  $c$  calculated in the last iteration, and load new  $b$  and  $c$  into spare 12 registers for the next sub-block multiplication. Since  $b$  and  $c$  use two sets of registers alternately, it is natural to unroll the innermost loop two times. Hence, the innermost loop calculates 2 sub-block multiplications. With carefully instruction scheduling, all memory access and other miscellaneous instructions (e.g. branch) can be issued and executed along with multiply-add instructions in the innermost loop. After these optimizations, the kernel performance significantly improves to **1.90GFLOPS**, which is **95%** of the peak performance. Since all effective calculations come from the innermost loop, it is better to enlarge the innermost loop times. We fuse the nested loops by encoding the strides of accessed sub-block addresses into 1-dimension arrays. Corresponding address stride of sub-blocks will be loaded out and added to current addresses to index next sub-blocks. After fusing two nested innermost loops into one, the innermost loop will iterate 100 times instead of 10 times. As a result, the kernel obtains nearly peak performance **1.99GFLOPS**. It could be found that this performance is about **12.4** $\times$  faster than the compiler-optimized one, which shows a lot of room of compilation technology.

## 4.2 Optimizing On-Chip Communication

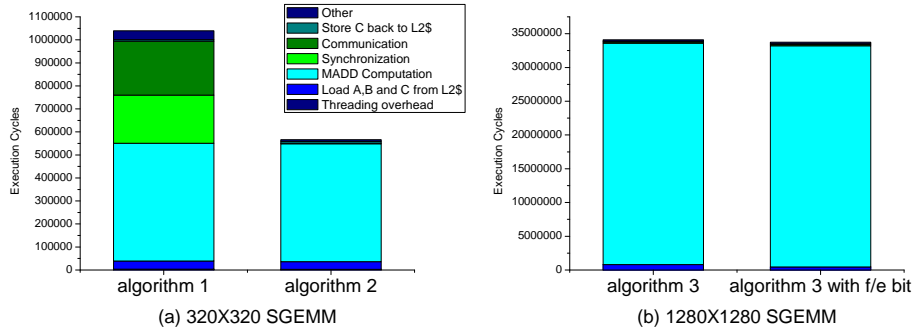
Conventional inter-thread communication on shared memory is implicitly guaranteed by cache coherence protocols. However, communication through cache hierarchy is inefficient by reason of the indirect communication path and limited bandwidth of cache hierarchy. Besides **vertical** data communication in memory hierarchy (i.e. between different levels of memory hierarchy), Godson-T also supports **horizontal** data communication (i.e. between different SPMs) for efficient inter-thread communication, leveraging on low on-chip communication latency and tremendous on-chip bandwidth. This feature is naturally achieved

by organizing all SPMs globally addressed and accessed, so that every core could directly access both local and remote SPMs with ordinary load/store instructions. Because of this, Cannon’s algorithm[4] can be easily implemented on the Godson-T’s  $8 \times 8$  mesh network. In Cannon’s algorithm, most of threads only communicate with their neighboring nodes with horizontal communication, so that the network contention and the L2 cache accesses are minimized. Initially, matrices  $A$ ,  $B$  and  $C$  are stored in normal row-major order arrays in off-chip memory. The algorithm for  $320 \times 320$  SGEMM (where  $t=8$ ) is described below.

**Algorithm 1 - Calculating  $320 \times 320$  SGEMM with 64 threads**

- ①  $thread_{i,j}$  ( $i$ th row,  $j$ th column of the mesh) retrieves  $40 \times 40$  matrix blocks  $A_{i,(i+j)\%8}$ ,  $B_{(i+j)\%8,j}$ ,  $C_{i,j}$  from L2 cache into local SPM;
- ② for (loop=1; loop $\leq$ 8; loop++)
- ③  $thread_{i,j}$  calculates  $C_{i,j} += A \times B$  on local SPM; // Sect. 4.1
- ④ if (loop!=8) then // data communication for the next iteration
- ⑤ barrier synchronization; // all threads complete computation
- ⑥  $thread_{i,j}$  gets  $A$  block from  $thread_{(i+1)\%8,j}$ ’s SPM, and  $B$  block from  $thread_{i,(j+1)\%8}$ ’s SPM to local SPM;
- ⑦  $thread_{i,j}$  stores  $C_{i,j}$  from SPM back to L2 caches;

The simulation shows that **algorithm 1** obtains **63.06GFLOPS** on Godson-T, which is **49.27%** of the peak performance. The first column of Fig. 2(a) shows the performance breakdown.



**Fig. 2.** Performance breakdown for different algorithms and input data sets. “Threading overhead” denotes the time for creating 64 threads. “Other” overhead includes the execution time of non-computational instructions, load imbalancing penalty, and etc.

It could be observed that communication and synchronization overhead cause 44.6% performance degradation. Note that SGEMM ought to be load-balanced theoretically. A large portion of synchronization overhead owes to imbalanced communication completion time: some threads are slower because of longer communication path. Therefore, inefficient communication is the major problem. This problem could be resolved on Godson-T by overlapping the cost of communication and computation. Data Transfer Agent (DTA) is built in each processing core to support asynchronous data communications. When the processing core is doing calculations, DTA can be programmed to manage data movements among

different memory parts in background. Compared to conventional DMA engine, DTA is a more advanced communication coprocessor for MCP. The first advantage is its versatility. DTA supports both horizontal and vertical data transfers. Moreover, DTA supports prefetch operation to command L2 caches to fetch bulk of data from off-chip memory into L2 caches. Besides accessing continuous data, DTA also supports 2-D stride data accessing. Hence, it is convenient to retrieve a blocked matrix from a large 2-D matrix array, while transposing it on-the-fly. The second advantage of DTA is its adaptation to network-on-chip. The delivered bandwidth of the network will degrade quickly if the injection bandwidth of the network is beyond some “saturation point” [9]. If DTAs send out their packets into network continuously and blindly, the on-chip network would probably be congested soon. DTA could assemble a group of requests or responses together into a single network packet for better utilization of bandwidth. Moreover, after sending a bunch of packets continuously, DTA sends a “probe” packet to the destination, and hangs until the “probe” packet is returned from the destination. Since the packet round-trip latency roughly reflects the contention status along the communication path, DTA can adaptively adjust the injection bandwidth to the network automatically with this simple flow-control mechanism.

---

**Algorithm 2 - Calculating  $320 \times 320$  SGEMM with 64 threads using DTA**

---

- ① *thread*<sub>*i,j*</sub> retrieves  $40 \times 40$  matrix blocks  $A_{i,(i+j)\%8}$ ,  $B_{(i+j)\%8,j}$ ,  $C_{i,j}$  from shared L2 cache into local SPM using vertical DTA operations;
  - ② for (loop=1; loop $\leq$ 8; loop++)
  - ③   barrier synchronization; // all data for communication is ready
  - ④   if (loop $\neq$ 8) then *thread*<sub>*i,j*</sub> gets  $A$  block from *thread* <sub>$(i+1)\%8,j$</sub> 's SPM, and  $B$  block from *thread* <sub>$i,(j+1)\%8$</sub> 's SPM to local SPM using horizontal DTA operations; // in parallel with the next step
  - ⑤   *thread*<sub>*i,j*</sub> calculates  $C_{i,j} += A \times B$  on local SPM (Sect. 4.1);
  - ⑥ *thread*<sub>*i,j*</sub> stores  $C_{i,j}$  from SPM back to L2 caches;
- 

The **algorithm 1** can be improved using DTA, described in **algorithm 2**. During the 1st step in **algorithm 2**, 64 threads retrieve data blocks from 16 L2 cache banks simultaneously, so that the network would be easily jammed. With network adaptation function of DTA, the utilized bandwidth of the network is **5.6** $\times$  larger than that of the naïve DTA without it. In the 4th step, we use non-blocked horizontal DTA operations to start the inter-thread communication, and overlap the overhead with local computation in the 5th step. The communication can be completely hidden in computation, so that the hard-wired global synchronization overhead in the 3rd step can be minimized to a few hundreds of cycles each time since the computation is load-balanced. The resultant performance dramatically improves to **115.70GFLOPS**, which is **90.39%** of the peak. The 2nd column of Fig. 2(a) shows the performance breakdown. Communication overhead is significantly reduced. Fetching data from off-chip memory becomes primary penalty and incurs 5.8% loss of the peak performance. It will be optimized in the next subsection.

### 4.3 Optimizing Memory Accessing

As stated previously, DTA is capable to command L2 caches to fetch data from the off-chip memory into L2 caches asynchronously. It helps improve performance by tolerating the off-chip memory latency. Larger SGEMM can be implemented based on  $320 \times 320$  SGEMM in **algorithm 2**. For instance, the algorithm for  $K \times K$  ( $K$  is a multiple of 320) SGEMM is shown in **algorithm 3**. Matrices  $A$ ,  $B$  and  $C$  are initially stored in three row-major order arrays in the off-chip memory. There are two modifications compared to **algorithm 2**: (1) while all threads are calculating a  $320 \times 320$  SGEMM during step 7, L2 caches are commanded by DTA prefetch operations in step 6 to fetch the data from off-chip memory into *least-recent-used* L2 cachelines for the next  $320 \times 320$  SGEMM. (2) Reuse  $C$  matrix block in  $320 \times 320$  SGEMM as much as possible, so it is unnecessary to load and store  $C$  blocks (step 5 and 8) every time.

---

#### **Algorithm 3 - Calculating $K \times K$ SGEMM with 64 threads**

---

```

①  $M = K/320$ ;
② for ( $r=0$ ;  $r<M$ ;  $r++$ ) // row
③   for ( $c=0$ ;  $c<M$ ;  $c++$ ) // column
④     for ( $s=0$ ;  $s<M$ ;  $s++$ )
⑤        $thread_{i,j}$  retrieves  $40 \times 40$   $A_{r \times 8+i, s \times 8+(i+j)\%8}$ ,  $B_{s \times 8+(i+j)\%8, c \times 8+j}$ 
         blocks from L2 caches into local SPM using DTA;  $40 \times 40$  block
          $C_{r \times 8+i, c \times 8+j}$  is retrieved if  $s==0$ ;
⑥       DTAs command L2 caches to prefetch  $320 \times 320$  block A, B for the
         next iteration;  $320 \times 320$   $C$  block is prefetched if  $s==M-1$ ;
⑦       Compute  $320 \times 320$  SGEMM(algorithm 2); // in parallel with step 6
⑧       if ( $s==M-1$ ) then  $thread_{i,j}$  stores current  $C_{r \times 8+i, c \times 8+j}$  back to L2
         caches using DTA.
```

---

We simulate  $1280 \times 1280$  SGEMM in our experiment. The off-chip prefetching in L2 caches can be completely overlapped by computation cycles in step 7. Therefore, when a  $320 \times 320$  SGEMM calculation is completed, data for the next  $320 \times 320$  SGEMM calculation is supposed to have already loaded into L2 caches. The resultant performance of the algorithm is **123.01GFLOPS**, which is **96.10%** of the peak performance. The first column of Fig. 2(b) shows the performance breakdown.

In these algorithms, we use hard-wired barrier synchronization in our algorithm: a thread sleeps after issuing a barrier request to synchronization manager (SM). After collecting all barrier requests, SM sends out acknowledgements to wake up all involved processing cores. Besides this coarse-grained synchronization mechanism, Godson-T also provides fine-grained full/empty-bit synchronization mechanism on SPM as implemented in [1][14], enabling to further hide the overhead of communication into computation. With it, a processing core could continue its computation as soon as the required element of the processing instruction arrives. DTA coprocessor supports synchronized vertical and horizontal communication operating on full/empty bits as well.

The fine-grained synchronization scheme is used to tolerate memory latency in the 5th step of **algorithm 3**. Instead of using ordinary DTA operation, we

use synchronized DTA operations to retrieve  $A$ ,  $B$  and  $C$  blocks from L2 caches. When the data element arrives to SPM, the corresponding full/empty bit is set. Local synchronized load instructions in the sequential kernel guarantee that the processing core is stalled when the accessed memory location is “empty” and resumed as soon as the memory location is set to “full”. With this optimization, the memory latency of fetching data from L2 caches can be partly overlapped with kernel calculation. The resultant performance for  $1280 \times 1280$  SGEMM increases to **124.33GFLOPS**, which is **97.14%** of the peak performance. The 2nd column of Fig. 2(b) shows the performance breakdown.

#### 4.4 Discussion

**Two-Level Latency-Tolerance and Horizontal Communication.** It is observed the performance is primarily gained from two-level latency-tolerance mechanism and horizontal communication enabled by Godson-T architecture. With the two-level latency-tolerant framework, the on-chip network latency and the off-chip memory latency are expected to be hidden by useful computation. In SGEMM case we evaluated, both types of the latencies could be completely tolerated by computation, resulting in very efficient performance. The horizontal communication takes advantage of tremendous on-chip data bandwidth and on-chip data temporal locality. With the horizontal communication, data of a thread could be imported from on-chip SPMs besides shared low-level caches. It is complementary to latency-tolerance framework since it minimizes the cache hierarchy accesses that needs to be tolerated. Therefore, fetching data from neighboring SPMs is encouraged.

For a problem, we could construct an algorithm in two steps to handle memory latency and communication latency, respectively. At first, since the memory-processor gap remains a problematic issue, divide-and-conquer approach is chosen to exploit the best data temporal locality for each level of shared memory hierarchy (i.e. multi-level tiling). Vertical DTA operations are used to overlap the cost of on-chip shared cache accesses, off-chip memory accesses and the computation. Based on it, we parallelize the algorithm with multithreading. Inter-thread communication is realized by horizontal DTA operations among SPMs as much as possible. Therefore, a message-passing like algorithm is defined on the top of the memory hierarchy. The resultant algorithm effectively combines shared memory programming with one-sided message passing programming, two very distinct multithreaded programming paradigms in literature.

**Performance Model.** To generalize our algorithm with different machine or program configurations, we establish a performance model to study our parallel algorithm quantitatively. Let  $P^2$  denote the number of processing cores and assume processing cores are distributed on a square 2-D mesh network. Let  $U$  denote the size per SPM and  $L$  denote the size of L2 caches. Let  $O$  denote the DTA horizontal bandwidth,  $V$  denote the average DTA vertical bandwidth between a processing core and a L2 cache,  $B$  denote the bandwidth between last

level caches and off-chip memory. Let  $N \times N$  denotes the dimension of square matrix blocks computed on SPM, thus  $PN \times PN$  denotes the dimension of square matrix blocks calculated in whole processor. At last, let  $S$  denote the width of data elements measured in byte ( $S = 4$  for single-precision floating-point data and  $S = 8$  for double-precision floating-point data).

The sequential blocked MM kernel could calculate a multiply-and-accumulate operation per cycle perfectly when all required data have already loaded onto SPM. Here, we only count the cost of memory accessing, communication and computation, and omit other insignificant overhead like synchronization and creating threads. The parallel efficiency is said efficient if both horizontal communication and off-chip memory prefetching overhead could be hidden into computation cycles. For a  $K \times K$  ( $K$  is a multiple of  $N$ ) GEMM problem, a thread in the algorithm fetches and stores  $N \times N$   $C$  blocks  $K^2/P^2N^2$  times, loads  $N \times N$   $A$  and  $B$  blocks  $K^3/P^3N^3$  times, and calculates  $K^3/P^2$  multiply-accumulates, so the efficient efficiency is shown in (2). From it, it can be observed that it is critical to preserve delivered on-chip bandwidth  $V$  for better efficiency. For the configuration  $P = 8$ ,  $S = 4$ ,  $N = 40$ ,  $V \approx 1.5\text{B/cycle}$ ,  $K = 1280$ , the theoretical efficient efficiency is 97.96%. As  $K$ ,  $P$  or  $N$  gets larger, the efficient efficiency can be better.

$$Efficiency = \frac{\frac{K^3}{P^2}}{\frac{2SN^2}{V} \times \frac{K^2}{P^2N^2} + \frac{2SN^2}{V} \times \frac{K^3}{P^3N^3} + \frac{K^3}{P^2}} = \frac{1}{\frac{2S}{N}(\frac{1}{K} + \frac{1}{PN}) + 1} \quad (2)$$

Since we apply double buffering and computation-communication overlapping at both SPM and L2 cache level, there are at least 4 constrained inequalities to ensure the efficient efficiency of the algorithm.

At SPM level, each SPM should be large enough to accommodate 5 matrix blocks for double buffering, so:

$$S \times 5N^2 \leq U \quad (3)$$

Inter-thread communication of  $A$  and  $B$  blocks should overlap with the computation of the sequential kernel in Sect. 4.1:

$$N^3 \geq S \times 2N^2/O \quad (4)$$

L2 caches also need to be large enough to accommodate  $C$  matrix blocks for current iteration and  $A$ ,  $B$  and  $C$  matrix blocks for next iteration:

$$S \times 4P^2N^2 \leq L \quad (5)$$

Prefetching  $A$ ,  $B$  and  $C$  matrix blocks from off-chip memory to L2 caches should be overlapped with computation, so:

$$N^3 \times P \geq 3S \times N^2P^2/B \quad (6)$$

For the current configuration on Godson-T  $P = 8$ ,  $S = 4$ ,  $N = 40$ ,  $U = 32KB$ ,  $L = 4MB$ ,  $O = 16B/cycle$  and  $B = 25.6B/cycle$ , all inequalities are satisfied.

**Extending for DGEMM.** Calculating DGEMM requires more bandwidth and more storage space. For  $S = 8$  and  $U = 32KB$ ,  $N$  should not be larger than 28 to satisfy (3). Let  $N = 28$ , (3)~(6) are still satisfied under current Godson-T configuration. It means that our algorithm is also efficient for DGEMM. The efficiency for DGEMM is 94.69% for  $K = 1260$  according to (2).

**Scaling for More Cores.** Inequalities (5) and (6) constrain the number of processing cores. To exploit the best utilization of SPM storage, let  $S \times 5N^2 = U$  to maximize  $U$  in (3), then we get:

$$P^2 \leq 5L/4U, \text{ and } P^2 \leq B^2U/45S^3 \quad (7)$$

With current configuration of  $U$ ,  $L$  and  $B$ , we could know that the number of processing cores  $P^2$  is constrained to 160 by the size of L2 caches  $L$  for both SGEMM and DGEMM. However, it is anticipated that the size of L2 caches will be enlarged as well as the number of processing cores increases. Surprisingly, current off-chip bandwidth  $B$  shows a lot of room to support more cores. Our algorithm is not bandwidth bounded.

## 5 Conclusion

In this paper, we propose a new version of Godson-T processor of many-core architecture, and examined the adaptation of a high-performance matrix multiplication algorithm to Godson-T many-core processor. With architectural supports for latency-tolerance and horizontal communication on Godson-T, the overhead of on-chip communication and off-chip memory accessing are minimized. The cycle-accurate simulation shows that for  $1280 \times 1280$  SGEMM, our algorithm achieves 124.3GFLOPS, which is about 97.1% of the peak performance. In theoretical analysis, we also show that our algorithm is also efficient for DGEMM and more processing cores.

This work proves that many-core processor is very promising to exploit thread-level parallelism efficiently. The approach is the combination of efficient architectural supports and corresponding programming methodology. We believe that many-core processors have great potential to serve as application accelerators, competitive to the existing accelerators[7][12] that are chiefly optimized for data-level parallelism. Our future work includes generalizing and automating the proposed two-level latency-tolerant programming methodology on Godson-T.

**Acknowledgement** This work is partially supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2005CB321600, the National Natural Science Foundation of China under Grant No. 60736012 and 60803030, EC under grant MULTICUBE FP7-216693, the Beijing Natural Science Foundation under Grant No.4092044, and the National High-Tech Research and Development Plan of China under Grant No.2009AA01Z103.

## References

1. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the 4th international conference on Supercomputing*, 1990.
2. K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al. The landscape of parallel computing research: A view from berkeley. *Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December*, 18(2006-183):19, 2006.
3. D. Burger, S.W. Keckler, K.S. McKinley, M. Dahlin, L.K. John, C. Lin, C.R. Moore, J. Burrill, R.G. McDonald, W. Yoder, et al. Scaling to the End of Silicon with EDGE Architectures. *Computer*, 37(7):44–55, 2004.
4. L.E. Cannon. A cellular computer to implement the Kalman filter algorithm. 1969.
5. Jeffrey R. Diamond, Behnam Robatmili, Stephen W. Keckler, Robert van de Geijn, Kazushige Goto, and Doug Burger. High performance dense linear algebra on a spatially distributed processor. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 63–72, 2008.
6. Z. Hu, J. del Cuvallo, W. Zhu, and G.R. Gao. Optimization of dense matrix multiplication on IBM Cyclops-64: Challenges and experiences. *LECTURE NOTES IN COMPUTER SCIENCE*, 4128:134, 2006.
7. U.J. Kapasi, W.J. Dally, S. Rixner, J.D. Owens, and B. Khailany. The Imagine stream processor. In *Proceedings 2002 IEEE International Conference on Computer Design*, pages 282–288, 2002.
8. T.G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.
9. S.S. Mukherjee, F. Silla, P. Bannon, J. Emer, S. Lang, and D. Webb. A comparative study of arbitration algorithms for the Alpha 21364 pipelined router. In *Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems*, 2002.
10. G. Tan, D. Fan, J. Zhang, A. Russo, and G.R. Gao. Experience on optimizing irregular computation for memory hierarchy in manycore architecture. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 279–280, 2008.
11. M.B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrati, B. Greenwald, H. Hoffman, P. Johnson, J.W. Lee, W. Lee, et al. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE micro*, 272:02, 2002.
12. S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *Proceedings of the 3rd conference on Computing Frontiers*, pages 9–20, 2006.
13. X. Ye, V.H. Nguyen, D. Lavenier, and D. Fan. Efficient parallelization of a protein sequence comparison algorithm on manycore architecture. In *Proceedings of the 9th international conference on Parallel and Distributed Computing, Applications and Technologies*, pages 167–170, 2008.
14. W. Zhu, V.C. Sreedhar, Z. Hu, and G.R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *Proceedings of the 34th annual International Symposium on Computer Architecture*, pages 35–45, 2007.