Optimizing Linpack Benchmark on GPU-Accelerated Petascale Supercomputer

Article *in* Journal of Computer Science and Technology · September 2011
DOI: 10.1007/s11390-011-0184-1 · Source: DBLP

Juan Chen
National University of Defense Technology
47 PUBLICATIONS **181** CITATIONS

Wang F, Yang CQ, Du YF *et al.* Optimizing linpack benchmark on GPU-accelerated petascale supercomputer. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 26(5): 854–865 Sept. 2011. DOI 10.1007/s11390-011-0184-1

Optimizing Linpack Benchmark on GPU-Accelerated Petascale Supercomputer

Feng Wang (王　锋) *Member, CCF, ACM*, Can-Qun Yang (杨灿群), Yun-Fei Du (杜云飞)
Juan Chen (陈　娟), Hui-Zhan Yi (易会战), and Wei-Xia Xu (徐炜遐)

School of Computer Science, National University of Defense Technology, Changsha 410073, China

E-mail: {fengwang, canqun, duyunfei, juanchen, huizhanyi, xuwx}@nudt.edu.cn

Received November 24, 2010; revised June 15, 2011.

Abstract In this paper we present the programming of the Linpack benchmark on TianHe-1 system, the first petascale supercomputer system of China, and the largest GPU-accelerated heterogeneous system ever attempted before. A hybrid programming model consisting of MPI, OpenMP and streaming computing is described to explore the task parallel, thread parallel and data parallel of the Linpack. We explain how we optimized the load distribution across the CPUs and GPUs using the two-level adaptive method and describe the implementation in details. To overcome the low-bandwidth between the CPU and GPU communication, we present a software pipelining technique to hide the communication overhead. Combined with other traditional optimizations, the Linpack we developed achieved 196.7 GFLOPS on a single compute element of TianHe-1. This result is 70.1% of the peak compute capability, 3.3 times faster than the result by using the vendor's library. On the full configuration of TianHe-1 our optimizations resulted in a Linpack performance of 0.563 PFLOPS, which made TianHe-1 the 5th fastest supercomputer on the Top500 list in November, 2009.

Keywords petascale, Linpack, GPU, heterogeneous, supercomputer

1 Introduction

The high-performance computing market is entering the petaFLOP era. Cray XT5 "Jaguar", which consists of 224 162 processor cores, has a peak performance of 2.331 PFLOPS and a Linpack[1] performance of 1.759 PFLOPS[2]. While Cray XT5 thus claimed the fastest supercomputer crown on Top500 list published in November, 2009, another approach to achieving the petaFLOP-scale performance is heterogeneous computing, whereby the capability of each node was extended with the addition of various types of computational accelerators, such as Cell, GPUs, and FPGAs[3]. In such heterogeneous (or hybrid) systems, CPUs offer generality over a wide range of applications while specialized accelerators provide better power efficiency and performance-per-dollar for specific computation patterns. For example, IBM Roadrunner is the first heterogeneous supercomputer with a peak performance of 1.375 PFLOPS and a Linpack performance of 1.042 PFLOPS. It is a hybrid design with 6 480 dual-core AMD Opterons and 12 960 IBM PowerXCell 8i accelerators.

GPUs are nowadays used not only as graphics accelerators but also highly parallel programmable processors due to the rapid increase in their capability and programmability. The computing power of GPUs has increased dramatically. For example, AMD HD5870's double-precision peak performance has reached 544 GFLOPS. Nvidia's new-generation CUDA architecture, Fermi[4], supports ECC (error correction code) throughout the memory hierarchy to enhance data integrity and reliability, especially for high-performance computing. Meanwhile, programming models, such as AMD's Brook+[5], Nvidia's CUDA[6] and Khronos Group's OpenCL[7], facilitate the development of general-purpose applications on modern GPUs. As a result, a broad range of computational demands for complex applications have been successfully mapped to GPUs[8-11].

A trend is developing in high-performance computing in which general-purpose processors are coupled to GPUs used as accelerators[12]. Such systems are known as heterogeneous (or hybrid) CPU/GPU systems. TianHe-1, China's first petaFLOP supercomputer[13], is such a system consisting of 5 120 Intel Xeon

Regular Paper
Supported by the National High Technology Research and Development 863 Program of China under Grant No. 2009AA01A128, the Major Science and Technology Project of China under Grant No. 2009ZX01036-001-003-001, the National Natural Science Foundation of China under Grant Nos. 61003087, 60903044, 60903059, 60970033, and 60673150.

processors and 5 120 AMD GPUs, delivering a peak performance of 1.206 petaFLOPs and a Linpack performance of 0.563 petaFLOPs. This paper describes our experience on developing an implementation of the Linpack benchmark to achieve this level of performance on TianHe-1, a scale never attempted before in GPU-accelerated hybrid systems, making TianHe-1 the 5th fastest supercomputer on the Top500 list in November, 2009[2]. Our optimization methods will provide valuable insights into the more general areas of developing high performance applications for large-scale CPU+GPU heterogeneous systems.

While GPU-accelerated hybrid systems have superior advantages of power efficiency and performance/price ratio, the problem of how to use GPUs to deliver superior performance is still difficult to solve, especially for a petascale system. Unbalanced workloads across the CPU cores and GPUs and the low-bandwidth communication between CPUs and GPUs are the two main obstacles to performance. This paper describes how we addressed these two obstacles when crafting a version of Linpack for the petascale TianHe-1 system. We have developed our Linpack implementation based on the standard open-source, High-Performance Linpack (HPL)[14], which is designed for homogeneous clusters. In our implementation, workloads are distributed adaptively to the CPU cores and GPUs with negligible runtime overhead, resulting in better load balancing than static partitioning methods[15]. In addition to adaptive load balancing, the CPU-GPU communication is effectively hidden by using a software pipelining technique, which is particularly useful for large memory-bound applications.

While our work focuses on boosting the Linpack performance on TianHe-1, it is expected to provide valuable insights into developing high-performance applications on hybrid CPU/GPU systems. In particular, our work indicates that adaptive load balancing can be useful for applications with repeating computations on large-scale CPU/GPU systems. In addition, careful attention must be given to choreographing the data movement between the CPU and GPU memories so that kernel execution and data transfers can be overlapped. This is particularly significant for GPU-accelerated systems since the GPU memories are typically small.

The rest of this paper is organized as follows. Section 2 provides an overview of the TianHe-1 system. Section 3 introduces the Linpack benchmark and our hybrid programming model. Section 4 proposes our implementation and optimization methods on TianHe-1 system. In Section 5 the evaluation results are analyzed. Related work and conclusion are presented in Section 6 and Section 7, respectively.

2 TianHe-1

The TianHe-1 supercomputer, the first petaFLOP computing system in China, developed by the National University of Defense Technology (NUDT) is a heterogenous cluster system combining Intel Xeon host processors with ATI Radeon HD4870 $\times$ 2 (RV770 architecture) GPU accelerators. The peak performance of TianHe-1 reaches 1.206 PFLOPS, and the measured Linpack performance is 563.1 TFLOPS. A total of 2 560 compute nodes were used in the TianHe-1 system while executing the Linpack benchmark. The ratio of performance to power consumption is 379.24 MFLOPS/W. On the Top500 list published in November, 2009, the TianHe-1 ranked No.5 and in the Green500 list, the TianHe-1 ranked No.8.

An overview of the TianHe-1 system is shown in Fig.1. Compute nodes in the TianHe-1 system were

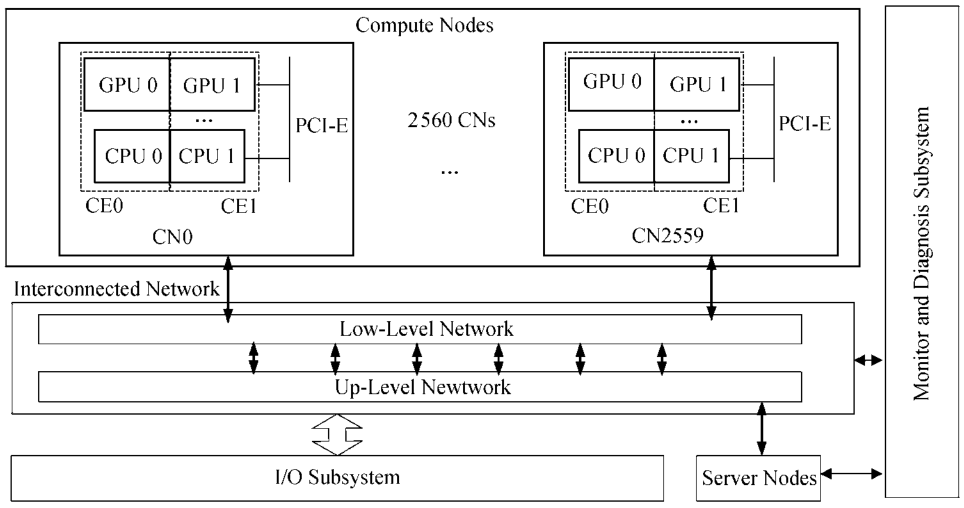

Fig.1. TianHe-1 architecture overview.

installed into 80 cabinets with 32 nodes per cabinet. Each node has two quad-core Intel Xeon processors, with 32 GB shared memory, and an ATI Radeon HD4870 $\times$ 2 GPU card plugged in the PCIe 2.0 slot. This GPU card consists of two independent RV770 chips, each with 1 GB local memory. The two GPU chips can be used together or alone. One CPU processor and one GPU chip in the same node constitute one basic heterogenous compute unit, which we call *compute element*. The total number of CPU processors in the compute nodes is 5 120, with 20 480 cores, including 4 096 Intel quad-core Xeon E5540 processors and 1 024 Intel quad-core Xeon E5450 processors. The peak performance by the CPUs is 214.96 TFLOPS. The 5 120 RV770 GPU chips contribute 942.08 TFLOPS, which accounts for 81.42% of the entire peak performance of the compute nodes.

The compute nodes are connected with two-level QDR Infiniband switches, which integrate four data lanes in each direction with 40 Gbps aggregate bandwidth and 1.2 μs latency.

3 Programming the Linpack Benchmark

3.1 Linpack Benchmark

Linpack[16] is a widely recognized industry benchmark for system-level performance of high-performance computing systems. It solves a dense $N \times N$ system of linear equations of the form $\boldsymbol{A}x = b$. The solution is obtained by performing LU factorization of the coefficient matrix $\boldsymbol{A}$ with partial pivoting, and then solves the resulting triangular system of equations. The workload of the Linpack benchmark is $(2/3)N^3 + 2N^2 + O(N)$.

The LU decomposition takes almost all the computation time of the benchmark. The computation time of the LU factorization is dominated by the matrix update and the upper (U) matrix factor. The matrix update is a form of the matrix-matrix multiply (DGEMM) which is an $O(N^3)$ operation. The latter uses a triangular solver with multiple right-hand-sides (DTRSM) kernel which is an $O(N^2)$ operation. Moreover, the two steps perform blocked data parallel patterns and agree with the executing model of the GPU. So we focus on the two steps and leverage the GPUs to accelerate them.

3.2 Hybrid Programming Model

Our implementation of the Linpack benchmark is based on the high performance linpack (HPL)[14], which is the reference implementation widely used to provide data for the TOP500. For the heterogeneous GPU cluster architecture of TianHe-1, we designed a hybrid programming model, which consisted of MPI, OpenMP and streaming computing. To make use of the computing capacity of the whole system, we fully explored the task parallel, thread parallel and data parallel of the Linpack, as shown in Fig.2.

The MPI is used by HPL originally for the homogeneous distributed-memory computers. In our implementation, we map one MPI process on each compute element, which is the hardware model of the basic computing unit of TianHe-1. These compute elements connect each other with the two-level QDR Infiniband switches. One CPU serves as the host and one GPU as the accelerator in a compute element. The host controls the communication (swap and broadcast of the pivot row) with others, and the compute-intensive tasks (DGEMM and DTRSM) are performed by the host and the accelerator cooperatively.

To parallelize the compute-intensive tasks, we use the OpenMP programming model on the host and configure four threads spawned at runtime when the progress enters the parallel region. Each thread is bound with one core of the CPU. One of the four threads controls the GPU to finish the major part of the computing workload, the other three threads deal

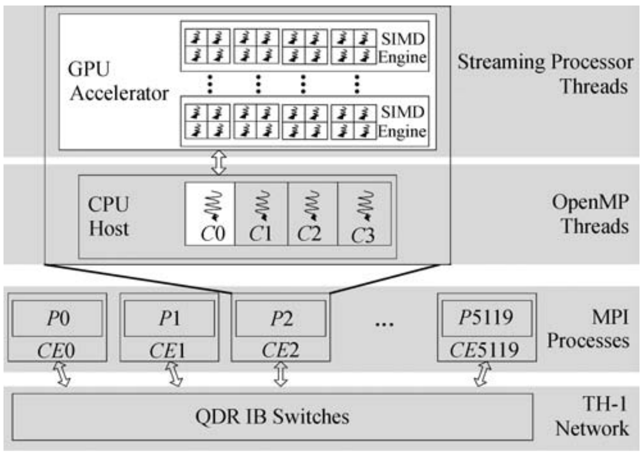

Fig.2. Hybrid programming model of Linpack implementation for the TianHe-1 system.

with the remaining fraction and at the end the four OpenMP threads are synchronized.

For the GPU accelerators, we used the streaming computing to develop the data parallel of the work assigned to them. We chose the CAL (compute abstraction layer) to program the kernels running on the GPU. The CAL is evolved from the CTM (close to the metal) SDK and it can make users access the low-level native instruction set and memory of the massively parallel streaming processors in the GPU architecture. The CAL API exposes the stream processors as a single instruction, multiple data (SIMD) array of computational processors. The GPU device has a scheduler that distributes the workload to the SIMD engines according to the domain of the execution specified as invoked. What is more, on the host side we used the streaming load/store operations to release the pressure of memory bandwidth and to avoid effecting other threads, when the dedicated thread transfers the data between the CPU and GPU.

4 Optimization Methods and Implementations

4.1 Overview

We developed the Level 3 Basic Linear Algebra Subprograms (BLAS3)[17] library for TianHe-1 and optimized them for this GPU accelerated heterogeneous architecture. For the Linpack benchmark we focused on the DGEMM and DTRSM which are the main time-consuming functions. To accelerate the DGEMM and the DTRSM, all the computing capacities including those of the host and the accelerator were used. The load balance is the key issue to achieve the high performance because there is synchronization at the end of these functions. We developed a two-level adaptive task mapping method to get the near-optimal splitting between the CPUs and the GPUs for all the computing workloads. At the same time, the massive data parallel inner GPU increases the demand for the memory bandwidth and the data communication between the host and the accelerator. We developed some novel methods to solve or release these problems and used some other well-known optimizations for the Linpack implementation. Based on these optimization methods, with one single Compute Element we achieved 196.7 GFLOPS of the Linpack benchmark, which is 70.1% of the peak compute capability.

4.2 Two-Level Adaptive Task Mapping

The DGEMM computes the product of matrix $\boldsymbol{A}$ and matrix $\boldsymbol{B}$, multiplies the result by scalar α, and adds the sum to the product of matrix $\boldsymbol{C}$ and scalar β. The DTRSM solves the matrix equation, and one matrix operand is a unit or non-unit, upper or lower, triangular matrix. The DTRSM is performed "in place", meaning that the original matrix is lost and successively replaced by the result. All the scalars and matrix elements mentioned above are all in double precision.

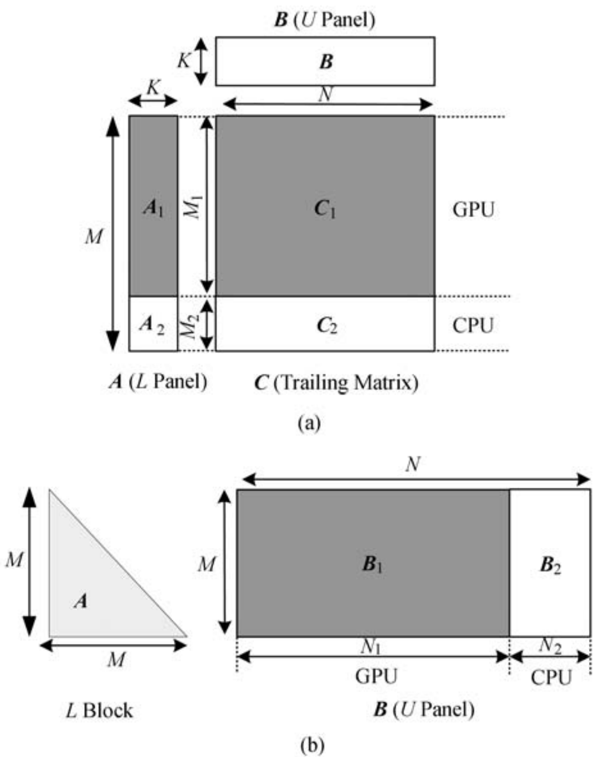

Fig.3. Workload split. (a) DGEMM split between the GPU and the CPU. (b) DTRSM split between the GPU and the CPU.

As shown in Fig.3, the workload of DGEMM and DTRSM can be split between the CPU and the GPU. The DGEMM in the Linpack is $\boldsymbol{C} = \boldsymbol{C} - \boldsymbol{A} \times \boldsymbol{B}$. The matrix $\boldsymbol{A}$ can be split to $\begin{pmatrix}\boldsymbol{A}_1\\ \boldsymbol{A}_2\end{pmatrix}$. So $\boldsymbol{C}$ can be calculated by combining the $\boldsymbol{C}_1 = \boldsymbol{C}_1 - \boldsymbol{A}_1 \times \boldsymbol{B}$, which is performed on the GPU, and the $\boldsymbol{C}_2 = \boldsymbol{C}_2 - \boldsymbol{A}_2 \times \boldsymbol{B}$, which is performed on the CPU. The DTRSM solves the matrix equation $\boldsymbol{A} \times \boldsymbol{X} = \boldsymbol{B}$ in the Linpack. The matrix $\boldsymbol{B}$ can be split to $(\boldsymbol{B}_1\ \boldsymbol{B}_2)$. The equation is split to two separate equations: $\boldsymbol{A} \times \boldsymbol{X}_1 = \boldsymbol{B}_1$ and $\boldsymbol{A} \times \boldsymbol{X}_2 = \boldsymbol{B}_2$, which can be performed on the GPU and CPU in parallel.

To find the best division of the computation across the CPU and GPU, we evaluated the executing time of the matrix multiply on one compute element (equipped with E5450 CPU) of TianHe-1 with different input sizes and different split ratios. All the input matrices are $N \times N$ square matrices. We define the Optimal Split Ratio (OSR) as the optimal work ratio mapped to the GPU. From the experiment, we got the

OSRs with different sizes of input matrices as shown in Fig.4. For the small sizes of the matrices ($N \leqslant 320$), the low computation-to-communication ratio renders the GPU less effective. As a result, the optimal split is to schedule all work on the CPU. For larger size ($832 \leqslant N \leqslant 4\,096$), the best choice is mapping all the work to the GPU, while for the other sizes, the range of the OSR is from 0.4 to 0.9. In the Linpack benchmark, the input matrices are not square, and the input sizes are smaller and smaller during the loop of the LU factorization. So, it is impossible to map the workload statically for every input size.

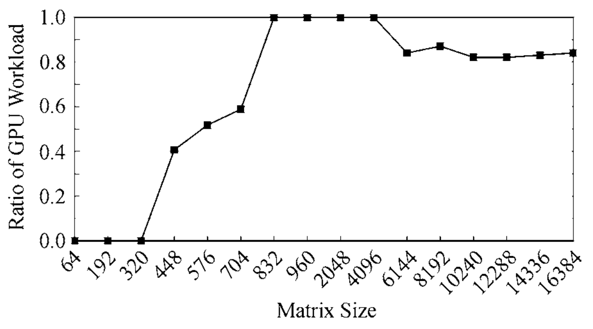

Fig.4. Optimal split ratio with different sizes.

In some cases, the CPU cores are not the same, especially for the CPUs like Xeon E5450 architecture. We dedicated one CPU core to control and communicate with the GPU. The other three CPU cores deal with the computing work. In Xeon E5450 architecture, four CPU cores are divided into two pairs and each pair shares the same L2 cache. So, the communication with the GPU not only affects the dedicated CPU core, but also affects the cores that share the same L2 cache. Even the effect on the core is very small, the imbalance will result in a large decrease in performance because the end time is the last step to finish. So, the split of the workload must adapt to this imbalance. We propose a two-level dynamic adaptive task-mapping method with negligible runtime overhead.

The main idea of this method is that we measure the performance of GPUs and CPUs in GFLOPS and use it to guide the split of the next workload. The optimal split by the GPU can be obtained got by this formula:

$$Gsplit = \frac{P_{GPU}}{P_{GPU} + P_{CPU}}. \tag{1}$$

P_{GPU} can be calculated by the formula:

$$P_{GPU} = 2 \times M_1 \times K \times N \times 10^{-9} / T_{GPU}, \tag{2}$$

$$P_{CPU} = 2 \times M_2 \times K \times N \times 10^{-9} / T_{CPU}. \tag{3}$$

M_1, M_2, K and N are illustrated in Fig.3. T_{GPU} and T_{CPU} are the executing time of DGEMM running on the GPU and CPU, respectively. The *Gsplit* is stored in a database called *database_g* which is indexed by the scale workload W.

$$W = 2 \times M \times K \times N \times 10^{-9}. \tag{4}$$

In the Linpack, the DGEMM with the same workload and the same shape is invoked many times. After the current DGEMM is finished, *database_g* will be updated according to the real performances of the GPU and CPU. The successive invoking will look up the database and use the *Gsplit* calculated by the previous formula.

The second level is to map the computations of CPU to each CPU core. This is different from the GPU and the CPU split. The split fractions are stored in the database named *database_c* and are indexed by the core number i, instead of the workload. The reason is that the performance of the CPU core is relatively stable. Suppose that the total number of CPU cores executing the DGEMM is n, the split fraction of the i-th CPU core is:

$$CSplit_i = \frac{P_{CPU[i]}}{\sum_{j=1}^{n} P_{CPU[j]}}, \tag{5}$$

where

$$P_{CPU[i]} = 2 \times M_{CPU[i]} \times K \times N \times 10^{-9} / T_{CPU[i]}. \tag{6}$$

$M_{CPU[i]}$ is the fraction mapped to the CPU core i. They are related with M_2 in (3).

$$M_2 = \sum_{i=1}^{n} M_{CPU[i]}, \tag{7}$$

$$T_{CPU} = \max[T_{CPU[i]},\ 1 \leqslant i \leqslant n]. \tag{8}$$

There are two databases, *database_g* and *database_c*. When DGEMM is invoked in the Linpack, the whole workload is calculated according to (4). The split ratio across the CPU and GPU can be obtained from *database_g* indexed by the workload. The OSR of the CPU cores can be obtained from *database_c* indexed by the core number. After all the DGEMM parts are finished, the OSRs in *database_g* and *database_c* are tuned according to (1) and (5), and then are stored into the databases. The overhead of the whole procedure includes only 5 system calls to get time, 8 divisions, 3 database stores and several floating-point multiply and add operations. For the DTRSM, it is the same with the DGEMM, except that the workload is calculated by the equation:

$$W_{DTRSM} = M^2 \times N \times 10^{-9}. \tag{9}$$

4.3 Software Pipelining

On the CPU/GPU heterogeneous platform, GPU communicates with CPU through PCIe (PCI Express) memory. Data are copied to PCIe memory first and then are transferred to the GPU local memory. With application of the PCIe 2.0, the bandwidth in the second process is very high, 4∼8 GBps. But the bandwidth between CPU host memory and PCIe memory is only on the order of hundreds MBps. The pinned memory, which is also called paged-locked memory, can be used to improve communication performance. Such a memory can be mapped to the PCIe memory, therefore the copying between the host memory and the PCIe memory can be eliminated. However, this memory resource is very limited (at most only 4 MB can be allocated each time for CAL), and allocating too much pinned memory will decrease the performance of the entire host system. To hide the data communication overhead, in this section we propose a neat software pipelining method that can overlap the kernel execution and the data transferring between the CPU and GPU. The large memory-bound applications will benefit from this optimization. Such applications need to break the computations and data sets into several tasks explicitly so as to fit the limited GPU memory. Each task has three phases: the data copying from the host to the GPU local memory, the kernel execution, and the results output. These tasks form one FIFO task queue. Our method pipelines the three phases among tasks to overlap data transfer with kernel execution.

We assume that there are N tasks in the current task queue. The task is referred to the computing entity, which is accelerated by the GPU and the tasks are independent each other. Every task has three phases: input, execution, and output. Codes or programs running on the GPU in a task are called kernels. All of the tasks in the queue are independent. The time of one task contains three parts: input time (*Tinput*), Kernel executing time (*Texecute*) and output time (*Toutput*). To simplify the presentation, we assume that all of the tasks have the same time in the three parts. Since there is no dependence among these tasks, we can overlap one task's execution and other tasks' input and output. In this pipelining method, all the data transfer can be hidden, except for the first task's input and the last task's output. As shown in Fig.5, all tasks are overlapped in the pipeline. This software pipelining has three stages: pipeline prologue, pipeline loop body, and pipeline epilogue. Pipeline prologue and epilogue are used for pipeline filling and draining. When the pipeline is full, one task can be completed every time of max (*Tinput, Toutput, Texecute*). If the *Texecute* is larger than *Tinput* and *Toutput*, the time consumed by all the tasks is $Tinput + Toutput + N \times Texecute$.

In GPUs, not only the capacity of local memory is not as large as other accelerators', but also the width and height of allocated memory are limited because of the texture's constrains. As an example, the maximum number of two-dimension addresses in the AMD RV770 chip is 8192×8192[5]. Matrixes that exceed this limit need splitting, so a large matrix-matrix multiply in Linpack is split into a series of tasks that form a task queue. For example, the matrix multiplication $\boldsymbol{A} \times \boldsymbol{B} = \boldsymbol{C}$ can be split to $\begin{pmatrix} \boldsymbol{A}_1 \\ \boldsymbol{A}_2 \end{pmatrix} \times (\boldsymbol{B}_1\ \boldsymbol{B}_2) = \boldsymbol{C}$, and four tasks are formed $T0 : \boldsymbol{C}_1 = \boldsymbol{A}_1 \times \boldsymbol{B}_1$, $T1 : \boldsymbol{C}_2 = \boldsymbol{A}_1 \times \boldsymbol{B}_2$, $T2 : \boldsymbol{C}_3 = \boldsymbol{A}_2 \times \boldsymbol{B}_1$, and $T3 : \boldsymbol{C}_4 = \boldsymbol{A}_2 \times \boldsymbol{B}_2$. To optimize the data-transfer time, we implemented the software pipelining among the tasks. Since the same matrix between tasks can be reused, the order of the four tasks is like $T0$, $T1$, $T3$, $T2$ by using the "bounce corner turn"[18] method. When $T1$ is executed, matrix $\boldsymbol{A}_1$ does not need to be transferred, neither do $\boldsymbol{B}_2$ for $T3$ and $\boldsymbol{A}_2$ for $T2$. In all, the entire matrix $\boldsymbol{A}$ and matrix $\boldsymbol{B}_1$ are skipped. In each task, the input phase is responsible for transferring the matrix from the CPU to the GPU. Then the execution phase finishes the matrix multiplication. At last, in the output phase the result of the multiplication is transferred to the CPU. Firstly,

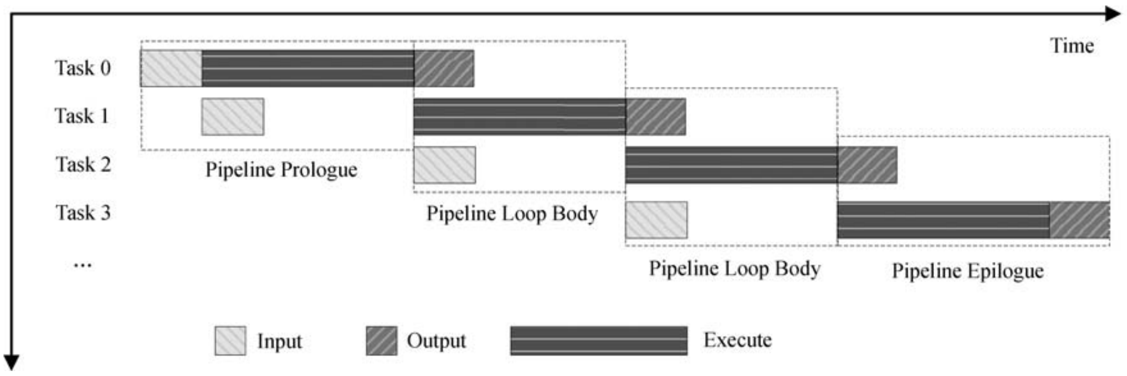

Fig.5. Software pipelining of task queue.

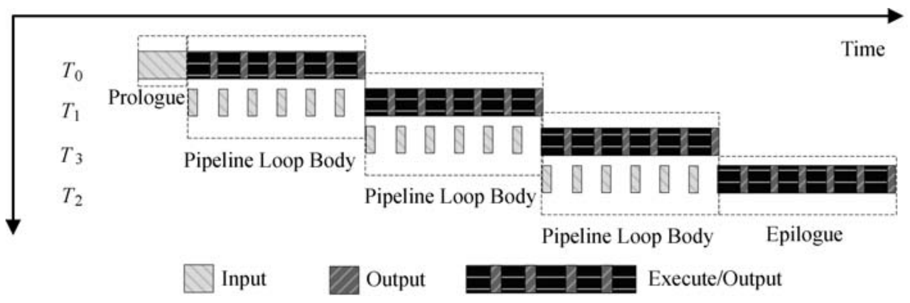

Fig.6. Pipelining among multiple tasks of matrix-matrix multiply.

the output phase can be optimized, because the matrix multiplication can be blocked to execute and output using the double-buffering. When the result of one block is output, the next block's multiply operations can be started. Therefore, the output stage is combined with execution stage to form a new stage which we call *Execution*/*Output*(*EO*) stage. Secondly, the input phase is also overlapped with the *EO* stage. To avoid the data transferring conflict between the input stage and the output stage, the input stage is split to small pieces which are transferred at the intervals of the output. Fig.6 shows the pipeline among the four tasks. The input stage of Task $T1$ in the pipeline is the prologue, and the epilogue in the pipeline is the *EO* stage of Task $T2$. In the pipeline loop-body stage, two tasks are running at the same time. We can observe that the input and output time can be overlapped, except for the input time for task $T0$ in the pipeline.

As we can see in Fig.6, two tasks in a task queue are active at most. We build two objects to control them, current task (CT) object and next task (NT) object. CT object always controls the first task in the queue, and NT object controls the second task if it exists. CT object has three states: *Idle*, *Input*, and *EO*. NT object has two states: *N-Idle* and *N-Input*. The *Idle* state is used to initialize CT object when a new task is controlled. The *Input* state corresponds with the prologue stage of the pipeline, while the *EO* state matches the loop body stage and the epilogue stage. The *N-Idle* state is used to initialize NT object as a new task is controlled. The *N-Input* state is used to do the task's matrix input and is overlapped with CT object's *EO* state.

Fig.7 shows the state transitions of *CT* and *NT* objects. The procedure names above the dash lines indicate the activities done in the corresponding state. The procedures trigger the events which drive the transitions of the states. There are three procedures, namely *Initialization*, *InputM*, and *Overlap*. The *Initialization* procedure is shared by the *Idle* state of the *CT* object and *NIdle* state of the *NT* object. The *Overlap* procedure is shared by the *EO* state of the *CT* object and *NInput* state of the *NT* object.

• *Initialization*. a) If the task queue has never been dealt with, the *CT* object is initialized by the first task and the *NT* object is initialized by the second task. The addresses and the shapes of the matrices are set to the objects. Then the "*CT Start*" event is triggered and causes the state of *CT* transition to *Input*. b) If the previous state of *CT* is *EO* and the previous state of *NT* is *N-Input*, the first task in the queue is removed. Then the object *NT* is copied to the object *CT*. After that, we initialize the *NT* object with the first task in the queue. Finally the event "*Task Copy from NT end*" is triggered and the state of *CT* transforms to *EO* state. The state of *NT* transforms to the *N-Input* state. c) If the previous state of *CT* is *EO* and there are no tasks in the queue, the algorithm is finished.

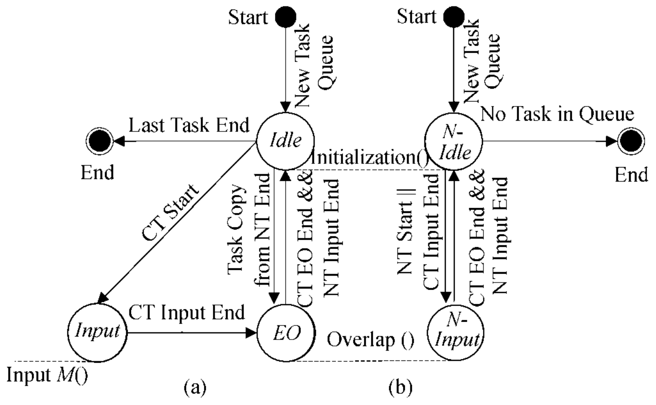

Fig.7. State diagrams of CT and NT. (a) Current task (CT) State diagram. (b) Next task (NT) state diagram.

• *InputM*. The matrices of the *CT* object are transferred to the local memory of the GPU. After that, the state of the *CT* object becomes *EO* and *NT* state becomes *NInput*.

• *Overlap*. Using double-buffering and blocked matrix multiplication optimizations, we overlap the computation and the output of the *CT* object. At the intervals of the computation and the output, we perform the input of the *NT* object.

4.4 Traditional Optimizations Combination

Apart from the two novel methods mentioned above, related with the hybrid nature of TianHe-1 system, we employed a combination method consisting some traditional and important optimizations for homogeneous systems. These optimizations are processes and threads affinity, streaming load/store[19], memory management strategy tuning, look-ahead technology, and large block size.

The total number of the MPI processes running the Linpack is 5 120. We mapped and bound one MPI process on one compute element of TianHe-1. Four OpenMP threads were created by one process. One of the four threads was dedicated to data communication with the GPU, while the other three threads were involved in computing. All the threads were bounded to the CPU cores using the thread affinity. This affinity can minimize thread migration and context-switching cost, and reduce the cache-coherency traffic among CPU cores. For the Intel Xeon E5450 processors, to release the pressure of memory bandwidth, we used SSE4.1 streaming load/store instructions to perform memory accesses, without any pollution of the processor caches. In the Linpack, for communicating with other nodes many temporary memory blocks are allocated and freed dynamically. With the increase in the problem size, these memory blocks are larger and larger, and the memory marshaling overhead and memory management become significant as well. To improve the performance of these memory operations, we need to change the strategy of the memory management. First, for all the block sizes we used the recycling memory allocator instead of the *mmap* allocator, avoiding expensive system calls. Second, we enlarged the minimum size of the top-most releasable memory chunk through the *mallopt* function call from the default 128 KB to 2 GB to decrease the frequency of trimming freed memory.

In the Linpack, the panel factorization lies on the critical path of the overall algorithm because of the cartesian property of the distribution scheme. To overlap the communication and the computation, we can pipeline the panel factorization and the trailing matrix update. When the k-th panel has been factored and then broadcast, the factorization of the panel $k+1$ begins as soon as its data have been updated. This optimization is called a look-ahead or the pipelined updating. HPL selects various depths of look-ahead and a depth of zero corresponding to having no look-ahead. Look-ahead optimization consumes some extra memory to keep all the panels of columns currently in the look-ahead pipe and increases the demand for the communication bandwidth on the network. By conducting empirical performance analysis and modeling, we decided that a look-ahead of depth 1 can achieve the best performance gain for TianHe-1 system.

The column number of the panel in the Linpack is called the block size (NB), which is used for the data distribution as well as for the computational granularity. In a typical CPU-Only Linpack running, the block size NB is 196 in our experiments, while in the GPU accelerated Linpack, large block sizes tend to be chosen to make full use of the GPU's compute capacity. But, if the block size is too large it will cause load imbalance between processes. Finally, we chose the block size of 1 216 in our Linpack evaluation which is significantly different from that of the CPU-Only system.

5 Evaluation

We used the Intel Math Kernel Library (MKL) version 10.2.1.017 for the CPUs and ACML-GPU (AMD Core Math Library for Graphic Processors) version 1.0[20] for the GPUs. The version of HPL is 2.0 and MPI Library is mvapich2 1.0.2p1. The compiler used is Intel icc version 10.1.

On a single compute element, we first evaluate and analyze the DGEMM performance linked with ACML-GPU optimized by our two methods. We also give the DGEMM performance of the CPU linked with Intel MKL. Then we evaluate the performance of our Linpack implementation on the same configurations. The GPU engine clock frequency is the standard 750 MHz for the single compute element test.

On the multi-compute elements, we focus on the evaluation of our Linpack implementation. The increase in the matrix size requires longer running time, which results in unstable status of GPUs because of higher temperature. We decreased the GPU core clock frequency from standard 750 to 575 MHz and the memory frequency from 900 to 625 MHz, with the highest temperature dropping from about 110 to 92°C.

5.1 Single Compute Element Results

We executed our implementation of DGEMM and Linpack on the TianHe-1 compute nodes to evaluate the impacts of different optimizations. All the optimizations are based on the vendor's library ACMLG 1.0. Finally we used all the optimizations to get the best results. To compare with the traditional CPU-only systems, we also give the results without using GPUs.

Fig.8 presents the performance of our DGEMM implementation on a single TianHe-1 compute element. The x axis of the graph is the size of the matrix, and the y axis indicates the achieved performance of the benchmark in GFLOPS. The performance of the adaptive method is the second run result, and the first run updates the databases.

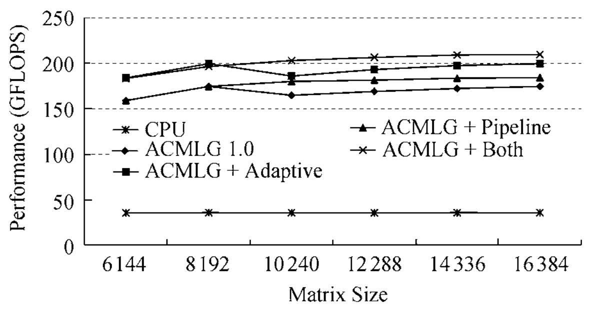

Fig.8. DGEMM performance by matrix size.

In Fig.8, we can see that for the original ACMLG library 1.0 there is a performance turning point when the matrix size is 8 192. Normally, for the CPU processors the performance increases smoothly to a steady standard as the matrix size rises because of the so-called volume-to-surface effect of the matrix multiplication[21]. However, the maximum number of 2D address of each dimension is 8 192 on the RV770 platform. A matrix larger than 8 192 is blocked using a size of 8 192 along each dimension. This may result in matrix multiplications for sizes smaller than 8 192. These small matrix multiplications affect the performance negatively because the overhead of data transfers between the host and the GPU is relatively higher.

Our pipeline method overcame this problem because we overlapped the communications and the computing between the two matrix multiplications. Our implementation is based on the ACMLG 1.0 library, and the matrices whose sizes are smaller than 8 192 are not blocked, so the performance of these matrix multiplication remains the same.

The performance benefit from the pipeline method for GPU computing reaches on average 7.61% when the matrix size N is more than 8 192. The benefit from the adaptive mapping technique reaches on average 14.64%. The two optimization methods can improve the performance by an average of 22.19% when N is more than 8 192. These evaluations indicate that the two optimization methods are independent.

In Fig.9, the performance of the Linpack implementations is evaluated on a single TianHe-1 compute element. Compared with Fig.8, the impact of our combination method of some traditional optimizations (TRCB) presented in the Subsection 4.4 is added. The x axis of the graph is the size of the matrix (the Linpack input N), and the y axis indicates the achieved performance of the benchmark in GFLOPS. Please note that the parameters of HPL were all carefully tuned. The databases used in the adaptive method were just the initial version. During the running of the Linpack, the databases were updated continuously according to the real performance of the CPUs and GPUs.

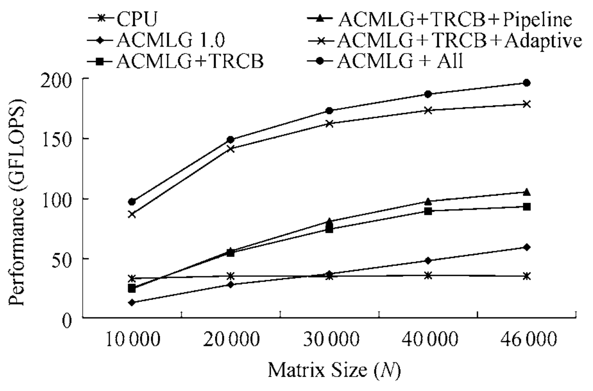

Fig.9. Linpack performance by matrix size.

On a single compute element, our implementation of Linpack achieved 196.7 GFLOPS for a matrix of size $N = 46\,000$. This performance result is 70.1% of the peak compute capability of a TianHe-1 compute element. These results also show that our implementation of Linpack utilizing the AMD RV770-based accelerators can outperform host-only implementation by a factor of 5.49 for large problem sizes. Compared with 59.2 GFLOPS (21.1% of the peak) linked to the ACMLG 1.0 library, our optimized Linpack runs 3.3 times faster.

The most important optimization was using the two-level adaptive dynamic load balance, which nearly doubled the performance at large problem sizes even after the Linpack was optimized by the TRCB method. The adaptive method does not only speed up the matrix multiply of large sizes using CPU's compute capacity, but also releases the problem that the GPU is very slow when dealing with small matrices operations which are the cartesian properties of the Linpack. The TRCB method improved the performance by up to 57.6% compared with ACMLG 1.0 and the pipelining method contributed 13.1%.

5.2 Multiple Compute Element Results

Fig.10 shows the performance scaling across multiple cabinets of the TianHe-1 system. The achieved performance of one single cabinet (64 compute elements) was 8.02 TFLOPS, and the achieved performance of 80 cabinets was 563.1 TFLOPS. This implies that the scaling efficiency is 87.76% from 1 to 80 cabinets.

The configurations of the full 80 cabinets are shown in Table 1. HPL provides many tuning parameters, such as broadcast topology and block size, which have to be carefully tuned in order to achieve the best performance. For TianHe-1 system, N, NB, P, Q, and

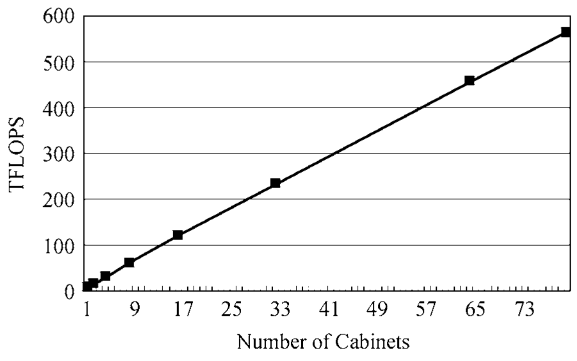

Fig.10. Performance scaling by cabinets.

Table 1. HPL Parameters Used in Evaluation with Full Configuration of TianHe-1.

Matrix Size (N)	2 240 000
Block Size (NB)	1 216
Process Mapping	Row-major
Process Grid ($P \times Q$)	64 × 80
Panel Factorization	Left-looking
NBMIN, NDIV	2, 2
Panel Broadcast	2ringM
Look-Ahead Depth	1
Swap	Mix (threshold = 1 536)
Matrix Form	L1: trans, U: no-trans
Equilibration	No
Alignment	8 double precision words

look-ahead depth significantly impact the HPL performance whereas the others play relative minor roles. With the coefficient matrix size $N = 2\,240\,000$ on $P \times Q = 64 \times 80 = 5\,120$ processes, the final result was 563.1 TFLOPS which was ranked No.5 on the Top500 list in November, 2009.

Fig.11 shows the progress of executing Linpack on the TianHe-1 system. Even at 97.17% of the progress, the performance is still 604.74 TFLOPS. At 2.83% of the progress, the performance drops dramatically about 41.6 TFLOPS to 563.1 TFLOPS. We believe that it is because the GPU is less effective when the matrix size is relatively small and this can be a potential optimization to improve the performance further. For the whole system, the performance and the stability of the network also play an important role during the running of the Linpack benchmark.

6 Related Work

The speedup of applications brought by GPUs has been realized. Many program model and optimization methods have been developed that provide performance superior to commodity CPUs in some fields. Ryoo[22] presented general principles for optimizing memory access, by which the global memory access is reordered in a CUDA architecture to combine requests to the same or to contiguous memory locations. Gregorio[23] uses four GPUs to accelerate the matrix-matrix product and the Cholesky factorization operation in the FLAME programming system.

But in the multi-core era of CPUs, the computation capacity still increases, following the Moore's law, which cannot be ignored. In Merge[24] framework, the computation-to-processor mappings are determined statically. Fatica statically maps the major computation DGEMM and DTRSM in Linpack to both GPU and CPU cores[25]. Qilin[15], a heterogeneous programming system, proposes adaptive mapping, an automatic technique to map computations to processing elements on a CPU+GPU machine. To find the near-optimal mapping, a program in Qilin first needs to conduct a training step and does not tune the mapping when running, which makes Qilin not applicable for very large

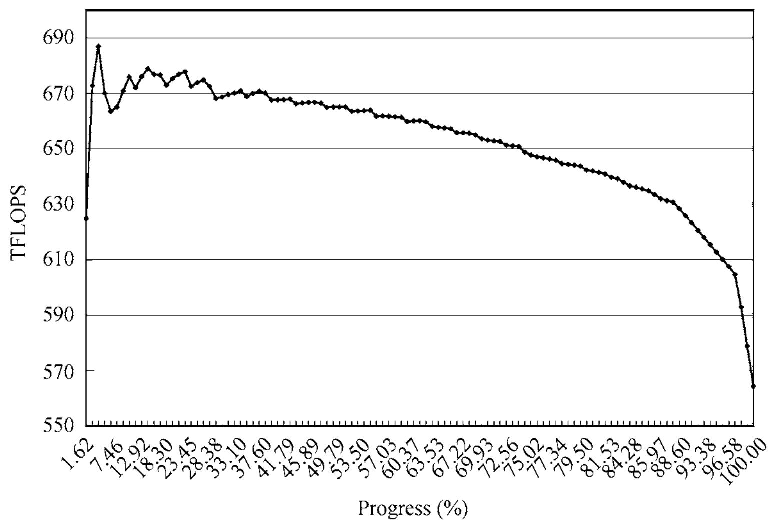

Fig.11. Performance of Linpack running on TianHe-1.

systems due to the inevitable performance fluctuating and the wasting of time and energy introduced by the training.

Using Cell accelerators[26], in 2008 IBM built the first heterogenous petascale supercomputer called Roadrunner[18]. This system was very different than a GPU-accelerated system. Compared with only 1 ∼ 2 GB memory on GPUs, each Cell blade contains 8 GB memory, which is large enough to store the entire dataset of most applications. The frequency of communications between the host and the accelerators decreased greatly while the data are being computed. So, the efficiency of SPE can therefore be up to 99.87% for the optimized matrix-matrix multiply kernel. On the other side, the memory controller of Cell can support only 25.6 GB/s, which is much lower than GPUs. For example the bandwidth of the AMD RV770 GPU chip can reach up to 115 GB/s[27]. Therefore, the methods used to leverage the accelerators are also different. TSUBAME[28] is another heterogenous system that contains commodity CPUs nodes and accelerated nodes by using ClearSpeed[29] and NVIDIA Tesla S1070[30]. On TSUBAME, an effective method is used to port applications developed for homogeneous assumptions to heterogeneous systems and achieves inter-node load balancing when testing the Linpack benchmark[31]. In June, 2010, Dawning built another petascale GPU-accelerated supercomputer system called Nebulae, which was ranked No. 2 on Top500 list with a Linpack performance of 1.271 PFLOPS. So far, the details of the optimization for Nebulae have not been unveiled. In August, 2010, the TianHe-1 was updated to TianHe-1A system which is equipped with 14 336 Xeon X5670 processors, 2 048 NUDT FeiTeng-1000 processors and 7 168 Nvidia Tesla M2050 GPUs as the accelerators. TianHe-1A supercomputer achieved 2.566 PFLOPS and ranked No. 1 on Top500 list of November 2010.

7 Conclusions

With the emerging of the massive GPU-accelerated supercomputer systems, the compute efficiency they can achieve is still a question, especially for the petascale level peak performance. We have described the implementation and optimizations of the Linpack benchmark for TianHe-1 supercomputer system, which combines traditional x86-64 host processors with AMD HD4870 × 2 GPUs. We have proposed the design of the hybrid programming model to take advantage of the heterogeneous hardware architecture. The optimizations we employed include the two-level adaptive workload distributing at runtime, software pipelining, and the combination method of traditional optimizations. On a single TianHe-1 compute element, this implementation of Linpack achieves 196.7 GFLOPS, which is 3.3 times faster than the vendor's library, and 70.1% of the peak compute capability. On the multiple compute elements configuration, our implementation has a good scalability, achieving 8.02 TFLOPS on one cabinet, and 563.1 TFLOPS on the full 80-cabinet TianHe-1 configuration.

This work demonstrates that it is possible to achieve high performance for some applications on the CPU/GPU heterogeneous system, under condition that the parallel is explored at every possible level, such as process level, thread level, data level, etc. The workload distribution and the communication between the hosts and accelerators are the key factors to this kind of system. We believe that our optimization methods can be applied to other computationally intensive, workload divisible applications.

References

[1] Dongarra J J, van de Geijn R A, Walker D W. Scalability issues affecting the design of a dense linear algebra library. *J. Parallel Distrib. Comput.*, 1994, 22(3): 523-537.

[2] http://www.top500.org, Nov. 10, 2010.

[3] Villarreal J, Najjar W. Compiled hardware acceleration of molecular dynamics code. In *Proc. International Conference on Field Programmable Logic and Applications* (*FPL 2008*), Heidelberg, Germany, Sept. 8-10, 2008, pp.667-670.

[4] NVIDIA. Fermi compute architecture whitepaper, 2009.

[5] AMD. AMD stream computing user guide v 1.4.0, Feb. 2009.

[6] NVIDIA. CUDA programming guide, June 2007.

[7] Munshi A. Opencl parallel computing on the GPU and CPU. In *Proc. ACM SIGGRAPH 2008*, Los Angeles, USA, Aug. 11-15, 2008.

[8] Falcao G, Yamagiwa S, Silva V, Sousa L. Parallel LDPC decoding on GPUs using a stream-based computing approach. *Journal of Computer Science and Technology*, 2009, 24(5): 913-924.

[9] Roberts E, Stone J E, Sepulveda L, Mei W, Hwu W, Luthey-Schulten Z. Long time-scale simulations of in vivo diffusion using GPU hardware. In *Proc. the 2009 IEEE International Symposium on Parallel & Distributed Processing* (*IPDPS 2009*), Rome, Italy, May 23-29, 2009, pp.1-8.

[10] Meng J, Skadron K. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proc. the 23rd International Conference on Supercomputing* (*ICS 2009*), Yorktown Heights, USA, Jun. 8-12, 2009, pp.256-265.

[11] Di P, Wan Q, Zhang X, Wu H, Xue J. Toward harnessing DOACROSS parallelism for multi-GPGPUs. In *Proc. the 39th International Conference on Parallel Processing*, San Diego, USA, Sept. 13-16, 2010, pp.40-50.

[12] Fan Z, Qiu F, Kaufman A, Yoakum-Stover S. GPU cluster for high performance computing. In *Proc. the 2004 ACM/IEEE Conference on Supercomputing* (*SC 2004*), Pittsburgh, USA, Nov. 6-12, 2004, p.47.

[13] Sun J C, Yuan G X, Zhang L B, Zhang Y Q. 2009 China top100 list of high performance computer. http://124.16.137.70/2009-China-HPC-TOP100-20091101-eng.htm, Nov. 2009.

[14] Petitet A, Whaley R C, Dongarra J J, Cleary A. HPL — A portable implementation of the high-performance

linpack benchmark for distributed memory computers. http://www.netlib.org/benchmark/hpl/, 2006.

[15] Luk C K, Hong S, Kim H. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proc. the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (*Micro-42*), New York, USA, Dec. 12-16, 2009, pp.45-55.

[16] Dongarra J J, Luszczek P, Petitet A. The linpack benchmark: Past, present and future. *Concurrency and Computation: Practice and Experience*, 2003, 15(9): 803-820.

[17] Dongarra J J, Du Croz J, Hammarling S, Duff I S. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 1990, 16(1): 1-17.

[18] Kistler M, Gunnels J, Brokenshire D, Benton B. Petascale computing with accelerators. In *Proc. the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP 2009*), Raleigh, USA, Feb. 14-18, 2009, pp.241-250.

[19] Baliga H, Cooray N, Gamsaragan E, Smith P, Yoon K, Abel J, Valles A. Original 45nm Intels Core2 processor performance. *Intel Technology Journal*, 2008, 11: 157-168.

[20] AMD. AMD core math library for graphic processors release notes for version 1.0, 2009.

[21] Agarwal R, Balle S M, Gustavson F G, Joshi M, Palkar P. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 1995, 39(5): 575-582.

[22] Ryoo S, Rodrigues C I, Baghsorkhi S S, Stone S S, Kirk D B, Hwu W M W. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP 2008*), Salt Lake City, Feb. 20-23, 2008, pp.73-82.

[23] Quintana-Ortí G, Igual F D, Quintana-Ortí E S, van de Geijn R A. Solving dense linear systems on platforms with multiple hardware accelerators. In *Proc. the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP 2009*), Raleigh, USA, Feb. 14-18, 2009, pp.121-130.

[24] Linderman M D, Collins J D, Wang H, Meng T H. Merge: A programming model for heterogeneous multi-core systems. *SIGOPS Oper. Syst. Rev.*, 2008, 42(2): 287-296.

[25] Fatica M. Accelerating linpack with CUDA on heterogenous clusters. In *Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units* (*GPGPU-2*), Washington DC, USA, 2009, pp.46-51.

[26] Johns C R, Brokenshire D A. Introduction to the cell broadband engine architecture. *IBM J. Res. Dev.*, 2007, 51(5): 503-519.

[27] ATI Radeon rv770. http://en.wikipedia.org/wiki/Radeon_R700.

[28] Hamano T, Endo T, Matsuoka S. Power-aware dynamic task scheduling for heterogeneous accelerated clusters. In *Proc. Int. Parallel and Distributed Processing Symposium*, Rome, Italy, May 23-29, 2009, pp.1-8.

[29] Clearspeed Technology Inc. http://www.clearspeed.com/.

[30] NVIDIA. http://www.nvidia.com/object/product_tesla_s1070_us.html, Nov. 10, 2010.

[31] Endo T, Matsuoka S. Massive supercomputing coping with heterogeneity of modern accelerators. In *Proc. the 2008 IEEE International Symposium on Parallel & Distributed Processing* (*IPDPS 2008*), Miami, USA, Apr. 14-18, 2008, pp.1-10.

Feng Wang received his Bachelor's and Master's degrees both in computer science from the National University of Defense Technology, China, in 2000 and 2002 respectively. He is currently an assistant professor and pursuing his Ph.D. degree at the university. His research interests include compiler techniques for high performance, compiler optimization and verification for embedded systems, and parallel programming. He is a member of CCF and ACM.

Can-Qun Yang received the M.S. and Ph.D. degrees both in computer science from the National University of Defense Technology (NUDT), China, in 1995 and 2008, respectively. Currently he is a professor at the university. His research interests include programming languages and compiler implementation. He is the major designer dealing with the compiler system of the TianHe Supercomputer.

Yun-Fei Du received the B.S. degree from the Beijing Institute of Technology in 2001 and the Ph.D. degree from the National University of Defense Technology (NUDT), China, in 2008. He is currently an assistant professor at NUDT. His research interests focus on parallel and distributed systems, fault tolerance, and scientific computing.

Juan Chen received the Ph.D. degree from the National University of Defense Technology, China. Currently she is an assistant professor at the university and her interests include large-scale parallel computing, low-power compiler, and GPU computing.

Hui-Zhan Yi received the Ph.D. degree from the National University of Defense Technology, China. Currently he is an assistant professor at the university. His research interests include low-power compilation optimization, parallel programming languages.

Wei-Xia Xu is a professor at the National University of Defense Technology, China. His research interest focuses on the computer architecture.