Adaptive Optimization for Petascale Heterogeneous CPU/GPU Computing

Canqun Yang, Feng Wang, Yunfei Du, Juan Chen, Jie Liu, Huizhan Yi and Kai Lu
School of Computer Science,
National University of Defense Technology,
Changsha, 410073, People's Republic of China
Email: {canqun, fengwang, duyunfei, juanchen, liujie, huizhanyi, kailu}@nudt.edu.cn

***Abstract*—In this paper, we describe our experiment developing an implementation of the Linpack benchmark for TianHe-1, a petascale CPU/GPU supercomputer system, the largest GPU-accelerated system ever attempted before. An adaptive optimization framework is presented to balance the workload distribution across the GPUs and CPUs with the negligible runtime overhead, resulting in the better performance than the static or the training partitioning methods. The CPU-GPU communication overhead is effectively hidden by a software pipelining technique, which is particularly useful for large memory-bound applications. Combined with other traditional optimizations, the Linpack we optimized using the adaptive optimization framework achieved 196.7 GFLOPS on a single compute element of TianHe-1. This result is 70.1% of the peak compute capability and 3.3 times faster than the result 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 released in November 2009.**

***Keywords*-GPU; heterogeneous; petascale; adaptive;**

I. 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 performance of 1.759 PFLOPS [1]. While Cray XT5 has thus claimed the fastest supercomputer crown in the Top500 list published in November 2009, another approach to achieving the petaflop-scale performance is heterogeneous computing, whereby the capability of each node is extended with the addition of various types of computational accelerators, such as Cell, GPUs, and FPGAs [2]. 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 [3], supports ECC throughout the memory hierarchy to enhance data integrity and reliability, especially for high-performance computing. Meanwhile, programming models, such as AMD's Brook+ [4], Nvidia's CUDA [5] and Khronos group's OpenCL [6], facilitate the development of general-purpose applications on modern GPUs. A C-like GPU programming language called BSGP (Bulk-Synchronous GPU Programming) [7] can significantly lower the code complexity with competitive performance compared with well-optimized CUDA programs in the graph field. As a result, a broad range of computationally demanding, complex applications have been successfully mapped to GPUs [8][9].

A trend is developing in high-performance computing in which general-purpose processors are coupled to GPUs used as accelerators [10]. Such systems are known as heterogeneous (or hybrid) CPU/GPU systems. TianHe-1, China's first Petaflop supercomputer [11], is one such system consisting of 5,120 Intel Xeon 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 for GPU-accelerated hybrid systems, making TianHe-1 the 5th fastest supercomputer in the Top500 list in November, 2009 [1]. Our optimization methods will provide valuable insights into the more general areas of developing high performance applications for large-scale CPU+GPU heterogeneous cluster systems.

While GPU-accelerated hybrid systems have superior advantages on power efficiency and performance/price ratio, how to use GPUs to deliver superior performance is still difficult, especially for a petascale system. Unbalanced workloads across the CPU cores and GPUs and the low-bandwidth communication between CPUs and GPUs are two main obstacles to performance. This paper describes how we address 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) [12], which is

978-0-7695-4220-1/10 $26.00 © 2010 IEEE
DOI 10.1109/CLUSTER.2010.12

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 [13]. Static methods lead to unbalanced workloads and cumulatively longer execution times while improving static methods by training/profiling consumes too much power to be practical in the petascale setting. 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.

The final adaptive optimization framework we developed for boosting the performance of Linpack on TianHe-1 used new techniques made possible by the hybrid nature of the system. Our framework thus consists of applying (1) a new adaptive work partitioning technique to dynamically improve the load balance for heterogeneity in TianHe-1, between the CPU cores and the GPU in a single CPU-GPU compute element, and (2) a new software pipelining technique for GPU computing to overlap kernel execution and data transfers. Our optimization framework is said to be *adaptive* since all our optimizations are performed when the workloads are assigned to the CPU cores and GPUs automatically at run time.

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 on 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.

In summary, the major contributions of this paper are as follows:

- We present an adaptive partitioning technique to distribute the computations in a program among the CPU cores and GPUs in a hybrid CPU/GPU system to achieve balanced workloads with negligible runtime overhead.
- We present a software pipelining technique for GPU computing to hide effectively the communication overhead between the CPU and GPU memories by overlapping it with kernel execution.
- We implement a version of Linpack developed by combining well-known optimizations with the two above-mentioned new techniques for TianHe-1. The resulting implementation delivers a Linpack performance of 0.563 PFLOPS, making TianHe-1 the 5th fastest supercomputer on the Top500 list released in November, 2009. In addition, the achieved efficiency by one CPU-GPU compute element represents 70.1% of its peak performance.

The rest of this paper is organized as follows. Section II discusses the related work. Section III provides an overview of the TianHe-1 system. Section IV propose an adaptive workload partitioning technique. In Section V, the software pipelining method is presented. Section VI discusses and analyzes our results. Section VII concludes.

II. Related Works

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 [14] presented general principles for optimizing memory access, which the global memory access is reordered in a CUDA architecture to combine requests to the same or to contiguous memory locations. Gregorio [15] 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 can not be ignored. In Merge [16] 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 [17]. Qilin [13], 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 make Qilin not applicable for very large systems due to the inevitable performance fluctuating and the wasting of time and energy introduced by the training.

Using Cell accelerators [18], two years ago IBM built the first heterogenous petascale supercomputer called Roadrunner [19]. 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 data set for most applications. However for the GPU-accelerated systems the large data can only be stored on the hosts. We need transfer the referred part to the GPU and transfer the result back to the host after computing. So the communications between the host and the accelerators are more severe for the GPU-accelerated systems. That is also the reason why the efficiency of SPE can 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 [20]. Therefore, the methods used to leverage the accelerators are also different. TSUBAME [21] is another heterogenous system that contains commodity CPUs nodes and accelerated nodes

by using ClearSpeed [22] and NVIDIA Tesla S1070 [23]. 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 [24].

III. TianHe-1

Figure 1. The TianHe-1 system.

Fig. 1 shows the photo of the TianHe-1 supercomputer, which is the first Petaflop computing system in China. TianHe-1 is developed by National University of Defense Technology (NUDT). TianHe-1 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 2560 compute nodes were used in the TianHe-1 system while executing the Linpack benchmark. The ratio of performance and power consumption is 379.24 MFLOPS/W. In 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. 2. Compute nodes in the TianHe-1 system installed into 80 cabinets with 32 nodes per cabinet. Each node has two quad-core Intel Xeon processors, with $32GB$ shared memory, and an ATI Radeon HD4870 $\times$ 2 GPU card plugged on the PCI-E 2.0 slot. This GPU card consists of two independent RV770 chips, each with $1GB$ local memory. The two GPU chips can be used together or alone. One CPU processor and one GPU chip in the same node constitutes one basic heterogenous compute unit, which we call *compute element*. The total number of CPU processors in the compute nodes is 5120, 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 contributed by the CPUs is 214.96TFLOPS. The 5120 RV770 GPU chips contribute 942.08TFLOPS, occupying 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 $1.2us$ latency.

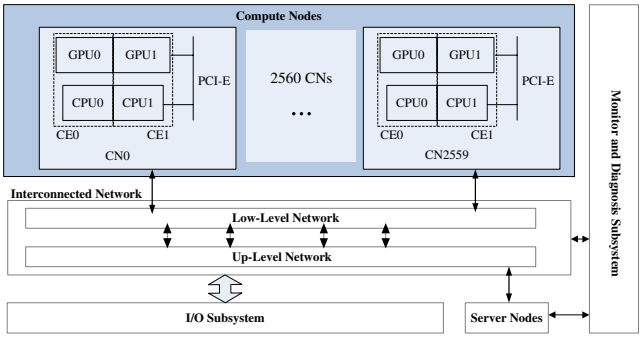

Figure 2. TianHe-1 architecture overview.

IV. Two-level adaptive task mapping

A. Motivation

To make full use of the computation capacity of CPUs and GPUs, we must map the workload to them. The performance of GPUs changes under different loads, while CPU performance is relatively stable for the load running on it. The Qilin programming system implements adaptive mapping, a fully automatic approach that decides the mapping from computations to processing elements on a CPU/GPU machine and that can tolerate changes in input problem sizes and hardware/software configurations. However, a training step must be executed to collect data for a database. This may not be satisfactory with large-scale CPU/GPU systems because training run wastes time and energy. Apart from that, Qilin does not consider the different computation capacity among CPU cores. Even when a CPU core is dedicated to control and to communicate with the GPU, the other CPU cores in a CPU are different, especially for the CPUs of such $E5450$ architecture. In this architecture, four CPU cores is 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 core that shares the same L2 cache. Even when the effect on the core is very small, the imbalance will result in a large decrease of performance because the end time is the last who finishes. For example, the peak performance of one compute element is 280.5 GFLOPS. If one CPU core's performance decreases 1 GFLOPS from 10 GFLOPS to 9 GFLOPS, the compute element performance loss will be enlarged to 28 GFLOPS in proportion. To avoid the load imbalance between the CPU and the GPU, as well as between different cores of large-scale heterogeneous systems, we propose a two-level dynamic adaptive task-mapping method with negligible runtime overhead compared with the training and running method.

B. Methodology

The main idea of our method is that we measure the performance of the GPUs and CPUs in GFLOPS at the

runtime and use it to guild the split of the next workload. To facilitate our discussion, let us introduce the following notations:

W: The workload of a program

$GSplit$: The fraction of the workload mapped to the GPU

$CSplit$: The fraction of the workload mapped to a CPU core

T_G: The actual time to execute the given program on the GPU

W_G: The workload of a GPU

P_G: The actual performance of a GPU

P'_G: The peak performance of a GPU

T_C: The actual time to execute the given program on the CPU

W_C: The workload of a CPU

P_C: The actual performance of a CPU

P'_C: The peak performance of a CPU

n: The number of cores in a CPU

The first level of our mapping technique is to map computations to the CPUs and the GPUs, which has two steps.

- In the first step, $Gsplit$ is obtained from the database_g indexed by the problem size for the program. The workload on the GPU and CPU are $W_G = W * GSplit$ and $W_C = W * (1 - GSplit)$, respectively.
- In the second step, T_G is obtained when the program is completed. Computing $P_G = W_G/T_G$ and $P_C = W_C/T_C$ allows a new mapping of $GSplit = \frac{P_G}{P_G+P_C}$, and then writing the new mapping to database_g indexed by the problem size. The new mapping is the next initial mapping for a program, whose problem size is in the same range as the problem size of that program.

The database_g has J items. Each item is a $GSplit$ value for the problem size within a range, which is $[(i-1) * \frac{W}{J} + 1, i * \frac{W}{J}]$ for item i. The initial value of each item is same, which is computed by $\frac{P'_G}{P'_G+P'_C}$.

The second level is to map computations to CPU cores in a CPU, which also has two steps.

- In the first step, $CSplit$ is obtained from the indexed database_c by the core number i; The workload of core i is $W_C * CSplit_i$.
- In the second step, T_{Ci} is computed when the program is completed, and $P_{Ci} = W_C * CSplit_i/T_{Ci}$. The new mapping of $CSplit_i = \frac{P_{Ci}}{P_{C1}+P_{C2}+\cdots+P_{Cn}}$ is conducted, and the new mapping is written to database_c indexed by the core number i. The new mapping is used as the next initial mapping for a program in CPU core i.

The database_c has n items for n CPU cores in a CPU. Each item is a $CSplit$ value for a CPU core and the initial value of each item is $1/n$.

C. Implementation in DGEMM

The DGEMM computes the product of matrix A_{M*K} and matrix B_{K*N}, multiplies the result by scalar α, and adds the sum to the product of matrix C_{M*N} and scalar β. This function belongs to the Level 3 Basic Linear Algebra Subprograms (BLAS3) library and dominates the computation time of HPL (High Performance Linpack). In the Linpack, the K is fixed and the other side of the matrix is smaller and smaller during the loop of the LU factorization. The effect on the DEGMM performance related with the matrices' shape can be combined into the workloads, i.e., the performance can be indexed only by the workload.

In the following discussion, we focus on a compute element which consists a quad-core Intel CPU and an RV770 GPU chip. To avoid disturbance from the computation of the CPU cores, a CPU core is dedicated to transferring data between the CPU and the GPU, and other three cores are involved in the matrix-matrix multiply operation. When we partition the workload across the CPU and GPU, the original matrix A is viewed as the union of two sub-matrices $A = A_1 \cup A_2$ with $M = M_1 + M_2$ (as shown in Fig. 3). The operation $C_1 = A_1 \times B$ is computed on the GPU, and the workload is $W_G = M * K * N * GSplit$. The operation $C_2 = A_2 \times B$ is computed on the CPU, and the workload is $W_C = M * K * N * (1 - GSplit)$. Mapping computations to CPU and GPU have two steps:

- In the first step, the mapping from database_g indexed by the float-point operation counts of the matrix-matrix multiply operation is obtained. The number of rows of the sub-matrices of A on GPU and CPU are $M_1 = M * GSplit$ and $M_2 = M * (1 - GSplit)$, respectively.
- In the second step, T_G is computed after the program is completed. Then, $P_G = M * K * N * GSplit/T_G$ is computed. The maximum value of the running time of each core is T_C, and $P_C = M * K * N * (1 - GSplit)/T_C$ is obtained. The new mapping of $GSplit = \frac{M*K*N*GSplit/T_G}{M*K*N*GSplit/T_G+M*K*N*(1-GSplit)/T_C}$ is conducted. Writing of the new mapping to database_g indexed by the float-point operation counts of the matrix-matrix multiply operation. The new mapping is the next initial mapping for a matrix-matrix multiply operation, whose float-point operation counts is in the same range as that matrix-matrix multiply operation.

Mapping computations on a CPU to three CPU cores has two steps:

- In the first step, the mapping of the three CPU cores is conducted, and the number of rows in matrix A each core get are $M_2 * CSplit_1$, $M_2 * CSplit_2$, and $M_2 * CSplit_3$, respectively.
- In the second step, T_{Ci} is obtained when the program is completed, and then $P_{Ci} = M_2 * K * N * CSplit_i/T_{Ci}$ for the core i is computed. The core i computes a new mapping $CSplit_i = \frac{P_{Ci}}{P_{C1}+P_{C2}+P_{C3}}$, and then writes the new mapping to database_c indexed by i as the next initial mapping for mapping computations to the core

i.

Fig. 3 shows the progress for the two matrix-matrix multiplications when using our technique to map computations to CPU cores and to the GPU. The float-point operation counts of the two matrix-matrix multiply computations are $M*K*N$ and $M'*K'*N'$, and $M*K*N \approx M'*K'*N'$.

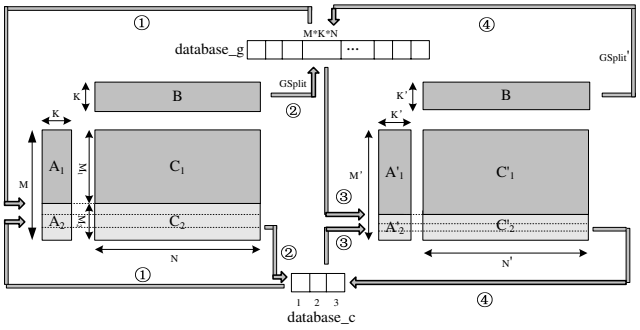

Figure 3. Our mapping method for the matrix-matrix multiplication.

① and ② are the procedures for $M * K * N$, and ③ and ④ are the procedures for $M' * K' * N'$. ① and ③ obtain the mappings of two databases, and ② and ④ write the new computed mappings to two databases. For the two problem sizes, the index item in the database_g is the same. The overhead for calculating the performances of the GPU and the CPU cores and the mapping includes only 5 system calls to get time, 8 divisions, 3 database stores and several floating-point add operations. Compared with the entire DGEMM running time, the overhead is negligible.

V. Software Pipelining

A. Motivation

On the CPU/GPU heterogeneous platform, GPU communicates with CPU through PCI-E memory. Data are copied to PCI-E memory first and then are transferred to GPU local memory. With application of the PCI-E 2.0, the bandwidth in the second process is very high to 4-8 GBps. But the bandwidth between CPU host memory and PCI-E memory is only on the order of hundreds MBps. Take the double-precision matrix-matrix multiplication with size $N = 10000$ as an example, the size of each matrix is 800 MB. Two matrixes need to be transferred to the GPU local memory and one matrix need to be output after computation on the GPU. On the assumption that the bandwidth between the host memory and the PCI-E buffer is 500 MBps and the bandwidth between the PCI-E buffer and the GPU local memory is 5 GBps, the time required for data transfer is $800*3/500+800*3/5000 = 5.28s$ without any optimization. The double-precision floating-point operation count is about $O(2*N^3) = 2000$ G. With the peak performance of an AMD RV770 GPU chip capable of 240 GFLOPS, the computing time is $2000/240 = 8.33s$. Therefore the communication is significant for matrix-matrix multiplication on the GPU. The pinned memory, which is also called paged-locked memory, can be used to improve communication performance. Such memory can be remapped to the PCI-E buffer, therefore the copying between the host memory and the PCI-E buffer can be eliminated. However, this memory resource is very limited (only 4 MB can be allocated at one 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 software pipelining method that can overlap kernel execution and data transfers between the CPU and GPU for large memory-bound applications. Such applications need to break down explicitly the computations and data sets into several tasks so as to fit within 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. Our method pipelines the three phases among tasks to overlap data transfer with kernel execution.

B. Methodology

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 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: the input time (*Tinput*), Kernel executing time (*Texecute*) and output time (*Toutput*). To simplify the presentation, we assume that the 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 of the data transfer can be hidden, except for the first task's input and the last task's output. As shown in Fig. 4, all tasks 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 tasks is $Tinput + Toutput + N \times Texecute$.

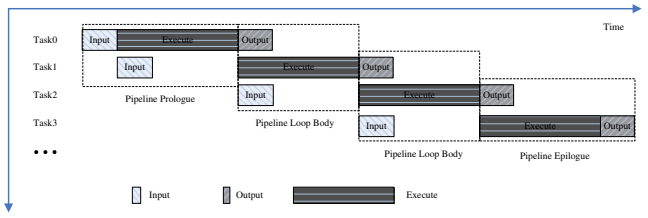

Figure 4. Software pipelining of task queue.

C. Implementation in DGEMM

In GPUs, not only the capacity of local memory is not as large as other accelerators's, 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 [4]. 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. As shown in Fig. 5, the matrix multiplication $A \times B = C$ is split to $\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \times (B_1 \ B_2) = C$ and four tasks are formed $T0 : C_1 = A_1 \times B_1$, $T1 : C_2 = A_1 \times B_2$, $T2 : C_3 = A_2 \times B_1$, and $T3 : C_4 = A_2 \times B_2$.

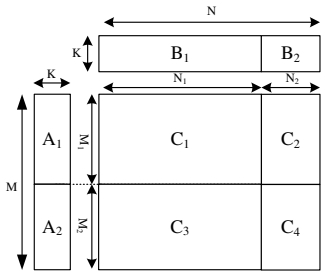

Figure 5. Splitting the matrix-matrix multiply.

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 in a task queue in Fig. 5 is organized as $T0$, $T1$, $T3$, $T2$ by using the "bounce corner turn" [19] method. When $T1$ is executed, matrix A_1 does not need to be transferred, so neither do B_2 for $T3$ and A_2 for $T2$. In all, the entire matrix A and matrix 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. In fact the output phase can be optimized because the matrix multiplication can be blocked to execute and output. 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 $Execution/Oputput(EO)$ stage. Fig. 6 shows the implementation of the blocking matrix multiplication for task $T0$. Matrix A_1 is blocked by rows with H being the height of the blocks. The number of blocks is $\lceil M_1/H \rceil$. Two $H * N_1$ matrixes CB_0 and CB_1 are allocated in the GPU local memory as output buffers. Every block in A_1 is multiplied by B_1 serially and the results are written alternately into CB_0 and CB_1. When one block matrix multiplication begins to be executed, the results of previous multiplication is transferred to the host memory. Thus, the output of the results is overlapped with the execution of the matrix multiplication. It not only omits the output time, but decreases the amount of memory resources stored for C_1 from $M_1 * N_1$ to $H * N_1 * 2$.

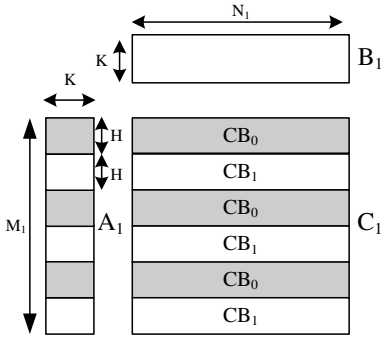

Figure 6. Blocking matrix-matrix multiply on GPU.

The input phase is also split to several blocks to avoid the conflict between the input stage and the output stage, since only one thread in our implementation is dedicated to transfer data with GPU. Fig. 7 shows the pipeline among the four tasks in Fig. 5. 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.

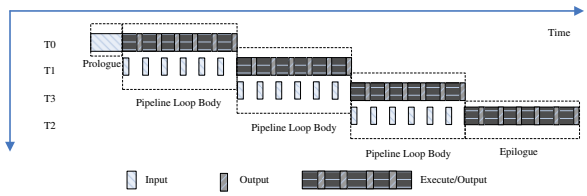

Figure 7. Pipelining among multiple tasks of matrix-matrix multiply.

As we can see from Fig. 7, 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.

Table I shows the pipeline in Fig. 7 shifted in time. At time 0, $T0$ is the first task in the task queue, and $T1$ is the second: $T0$ is controlled by CT object, and $T1$ is controlled

Table I
THE PIPELINE SHIFTED IN TIME.

Time Steps	Current Task			Next Task	
	Idle	Input	EO	N-Idle	N-Input
0	T0			T1	
1		T0		T1	
2			T0		T1
3	T1			T3	
4			T1		T3
5	T3			T2	
6			T3		T2
7	T2				
8			T2		

by NT object. At time 1, $T0$ is in the *INPUT* state, and the state of $T1$ is unchanged. At time 2, $T0$ is in the *EO* state, and the state of $T1$ is changed to the *N-INPUT* state, during which the input matrixes of $T1$ are transferred. At time 3, $T0$ completes the computation and is removed from the task queue. Therefore, $T1$ is the first task in the task queue controlled by CT object, and $T3$ is the second task controlled by NT object. At time 4, $T1$ is in the *EO* state and $T3$ is in the *N-INPUT* state. The subsequent tasks proceed in the same fashion until there are no tasks in the queue.

VI. EVALUATION

A. Methodology

We developed our Linpack implementation based on HPL. The Linpack benchmark, which solves a dense $N \times N$ system of linear equations of the form $Ax = b$, is used as a performance measure for ranking supercomputers in the Top500list. The solution is obtained by performing LU factorization of matrix A with partial pivoting, and then solves the resulting triangular system of equations. The workload of the Linpack benchmark is $(2/3)N^3 + O(N^2)$. Almost all of the computation time of the benchmark is occupied by the LU decomposition. The computation time of the LU factorization is dominated by the matrix update step, which is a form of the matrix-matrix multiply (DGEMM), an $O(N^3)$ operation.

We used the Intel Math Kernel Library (MKL) version 10.2.1.017 for CPU and ACML-GPU 1.0 (AMD Core Math Library for Graphic Processors) [25]. The version of HPL is 2.0 and MPI Library is mvapich2 $1.0.2p1$. The compiler used is Intel icc version 10.1. The flags using in the Intel Compiler are "-O3 -opt-malloc-options=1". The flag "opt-malloc-options" is used to improve the performance of malloc(), since there are a lot of malloc/free calls in our Linpack implementation.

We bound one process on one compute element, then this process created four CPU threads. One thread was dedicated to the data communication with the GPU, while the other three threads are involved in computing. Thread affinity was used to bind threads to CPU cores. This affinity can minimize thread migration and context-switching cost and reduces the cache-coherency traffic among 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 a typical CPU-Only Linpack running, the block size NB is 196 from 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 too large block size will cause load imbalance between processes. By analyzing the experiment results, we empirically chose the block size of 1216 in our Linpack evaluation to make a compromise.

On a single compute element, we first evaluated 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 evaluated 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 focused on the evaluation of our Linpack implementation. With the increasing of the problem size, the Linpack has longer running time, which results in unstable status of GPUs because of the 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 110 to 92 degree Celsius. When we evaluated our linpack implementation on TianHe-1 with full $5,120$ compute elements, $5,120$ processes were forked, with one process on each compute element. The process grid is 64×80. The size of the matrix N is $2,240,000$. The final result of the Linpack output is 563.1 TFLOPS.

B. Single compute element results

We executed our implementation of DGEMM and Linpack on the TianHe-1 compute nodes. We evaluated the following five items:

- CPU: The measured performance of the standard DGEMM implementation linked with the Intel MKL.
- ACMLG: The measured performance of the DGEMM implementation linked with the ACML-GPU.
- ACMLG+adaptive: The measured performance of the DGEMM implementation linked with the ACML-GPU optimized with adaptive dynamic load balance.
- ACMLG+pipe: The measured performance of the DGEMM implementation linked with the ACML-GPU optimized with the GPU computing pipeline.
- ACMLG+both: The measured performance of the DGEMM implementation linked with the ACML-GPU optimized with the two methods.

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

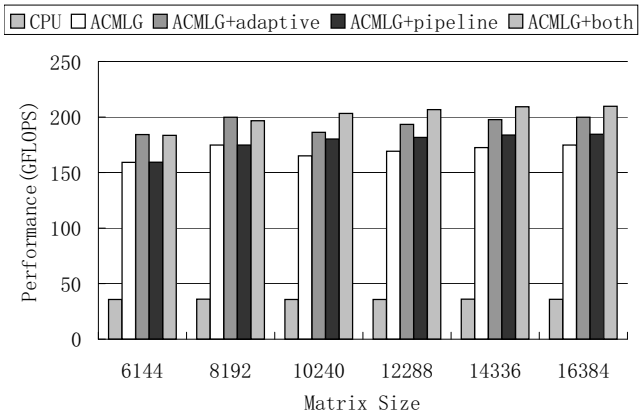

Figure 8. DGEMM performance by matrix size.

indicates the achieved performance of the benchmark in GFLOPS. The performance from the adaptive method is the second run result and the first run updates the databases. The performance benefit of the adaptive mapping technique is on average 14.64%. The pipeline method has no performance benefit when the matrix size N is less than or equal to 8192, since only one task is in the queue in the pipeline under this condition. The performance benefit of the pipeline method for GPU computing is on average 7.61% when the matrix size N is more than 8192. The two optimization methods can improve the performance by an average of 22.19% when N is more than 8192, since the two optimization methods are independent.

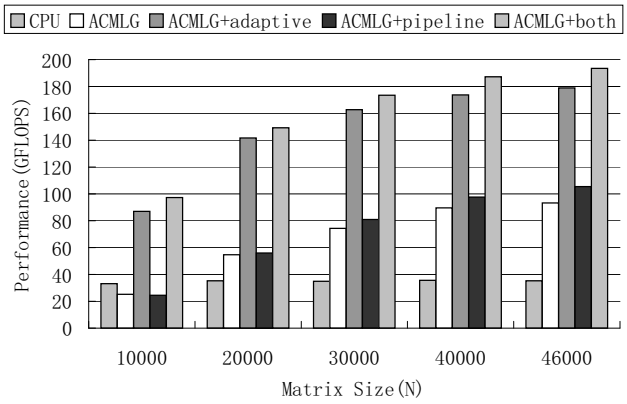

Figure 9. Linpack performance by matrix size.

Fig. 9 presents the performance of the Linpack implementations, each of which is linked to one of the above five DGEMM implementations on a single TianHe-1 compute element. 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. The databases used in the adaptive method is just the initial version. During the running of the Linpack, the databases are updated continuously according to the real performance of the CPUs and GPUs. On a single compute element, for a matrix of size $N = 46000$ our implementation of Linpack achieves 196.7 GFLOPS which is 70.1% of the peak compute capability. Compared with 59.2 GFLOPS (21.1% of the peak) linked to the ACMLG library, our optimized Linpack runs 3.3 times faster. 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.

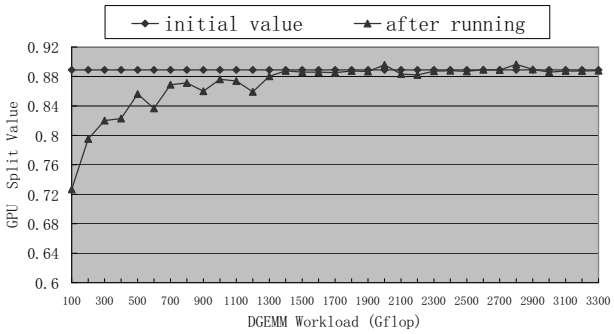

Figure 10. GPU split ratio changes with the workload.

Fig. 10 shows the workload fraction occupied by the GPU is related with the input problem size. The initial value is set to 0.889 according to the peak performance of the CPU and GPU. After the running of the Linpack, the adaptive framework adjusted the split values of the GPU stored in the *database_g* according to the real performance. We can see that the values are significant different with the initial values when the workloads are less than 1300 Gflop. With the increasing of the workloads, the fluctuation of the split value is not that dramatically.

C. Multiple compute element results

To evaluate the scalability of our implementation, we ran our Linpack implementation across all 64 compute elements in one cabinet. To compare our adaptive mapping against Qilin, we ran our Linpack implementation linked with the ACML-GPU optimized by using our adaptive mapping. When using Qilin, we trained the database for each compute element and every input problem size of the Linpack. The results of this experiment are shown in Fig. 11. With the increasing of the number of processes, our mapping method provides a larger performance than does Qilin system. When the number of processes is 64, our method is 15.56% faster. The reason for the improvement is that our method can adapt to the variability in a system when the number of processes increases. The mapping obtained by using a training run will incur the load imbalance for the actual running of the Linpack. Additionally, to train the databases used in

Qilin system, it took us about two hours on one cabinet configuration. The power consumption of one cabinet we measured is about 18.5 kw, without concerning the air-conditioning and UPS equipments. So the energy used for the training is about $18.5 * 2 = 37$ kwh. On the full 80-cabinet configuration, even the training time keeps the same, the energy used will be $37 * 80 = 2,960$ kwh.

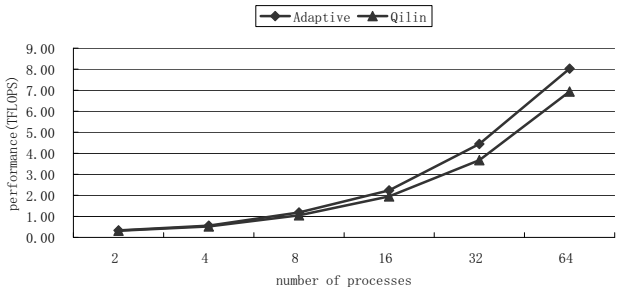

Figure 11. Performance by number of processes within a single cabinet.

Fig. 12 shows performance scaling across multiple cabinets of the TianHe-1 system. Here we see the achieved performance of a single cabinet was 8.02 TFLOPS, and the achieved performance of 80 cabinets was 563.1 TFLOPS with the problem size increasing from $280,000$ to $2,400,000$. This implies that the scaling efficiency is 87.76% from 1 to 80 cabinets.

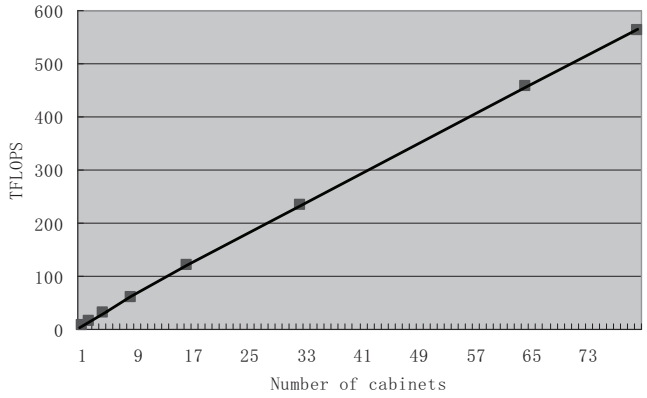

Figure 12. Performance scaling by cabinets.

Fig. 13 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. During the continuing 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.

VII. Conclusions

Unbalanced workloads across CPUs and GPUs and the data transfer between CPUs and GPUs are two main obstacles to performance in GPU computing. We have described our experience on addressing these two obstacles when developing an implementation of Linpack for a petascale hybrid system called TianHe-1, which combines traditional x86-64 host processors with AMD HD4870 $\times$ 2 GPUs. In our implementation, an adaptive partitioning technique is used to distribute workload adaptively to the CPU cores and GPUs with negligible runtime overhead, resulting in better load balancing than static partitioning methods. Our software pipelining technique can hide the CPU-GPU communication overhead effectively, which is particularly useful for large memory-bound applications.

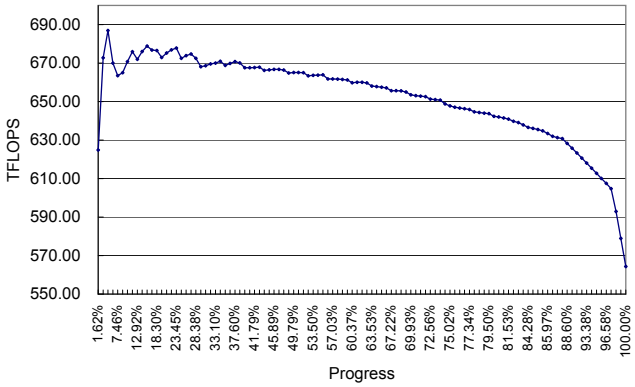

Figure 13. Performance of Linpack running on TianHe-1.

On a single TianHe-1 compute element, our adaptive optimization framework improves the performance 22.19% for the matrix-matrix multiply (DGEMM). Our implementation of Linpack achieves 196.7 GFLOPS, which is 70.1% of the peak compute capability and 3.3 times faster than the result using the vendor's library. On the multiple compute elements configuration, Linpack has a good scalability, achieving 8.02 TFLOPS on one cabinet, and 563.1 TFLOPS on the full 80-cabinet TianHe-1 configuration. The Linpack implementation makes TianHe-1 the 5th fastest supercomputer on the Top500 list released in November 2009.

Acknowledgment

We thank the anonymous reviewers for their helpful comments. This work is supported by the National High Technology Research and Development Program of China (863 Program) No.2009AA01A128; National Natural Science Foundation of China (NSFC) No.60903044.

References

[1] http://www.top500.org.

[2] J. Villarreal and W. Najjar, "Compiled hardware acceleration of molecular dynamics code," in *FPL '08: International Conference on Field Programmable Logic and Applications*. Heidelberg, Germany: IEEE Computer Society, 2008.

[3] NVIDIA, "Fermi compute architecture whitepaper," 2009.

[4] AMD, "Amd stream computing user guide v 1.4.0," Feb. 2009.

[5] NVIDIA, "Cuda programming guide," June 2007.

[6] A. Munshi, "Opencl parallel computing on the gpu and cpu," in *ACM SIGGRAPH 2008*, 2008.

[7] Q. Hou, K. Zhou, and B. Guo, "Bsgp: bulk-synchronous gpu programming," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–12, 2008.

[8] E. Roberts, J. E. Stone, L. Sepulveda, W. mei W. Hwu, and Z. Luthey-Schulten, "Long time-scale simulations of in vivo diffusion using gpu hardware," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–8.

[9] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus," in *ICS '09: Proceedings of the 23rd international conference on Supercomputing*. New York, NY, USA: ACM, 2009, pp. 256–265.

[10] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004.

[11] J. C. Sun, G. X. Yuan, L. B. Zhang, and Y. Q. Zhang, "2009 china top100 list of high performance computer," http://124.16.137.70/2009-China-HPC-TOP100-20091101-eng.htm, 11 2009.

[12] A. Petitet, R. C. Whaley, J. J. Dongarra, and A. Cleary, *HPL - A portable implementation of the high-performance linpack benchmark for distributed memory computers*, http://www.netlib.org/benchmark/hpl/, 2006.

[13] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: ACM, 2009, pp. 45–55.

[14] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 73–82.

[15] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn, "Solving dense linear systems on platforms with multiple hardware accelerators," in *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2009, pp. 121–130.

[16] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 2, pp. 287–296, 2008.

[17] M. Fatica, "Accelerating linpack with cuda on heterogenous clusters," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. New York, NY, USA: ACM, 2009, pp. 46–51.

[18] C. R. Johns and D. A. Brokenshire, "Introduction to the cell broadband engine architecture," *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 503–519, 2007.

[19] M. Kistler, J. Gunnels, D. Brokenshire, and B. Benton, "Petascale computing with accelerators," in *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2009, pp. 241–250.

[20] http://en.wikipedia.org/wiki/Radeon_R700.

[21] T. Hamano, T. Endo, and S. Matsuoka, "Power-aware dynamic task scheduling for heterogeneous accelerated clusters," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–8, 2009.

[22] ClearSpeed Technology Inc, http://www.clearspeed.com/.

[23] NVIDIA, http://www.nvidia.com/object/product_tesla_s1070_us.html.

[24] T. Endo and S. Matsuoka, "Massive supercomputing coping with heterogeneity of modern accelerators," in *IPDPS '08: Proceedings of the 2008 IEEE International Symposium on Parallel&Distributed Processing*, 2008, pp. 1–10.

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