Accelerating Linpack Performance with Mixed Precision Algorithm on CPU+GPGPU Heterogeneous Cluster

WANG Lei[1, 2, 3] *ZHANG Yunquan*[1, 2] *ZHANG Xianyi*[1] *LIU Fangfang*[1]
1(Lab of Parallel Computing, Institute of Software, Chinese Academy of Sciences, Beijing 100190)
2(State Key Lab of Computing Science, Chinese Academy of Sciences, Beijing 100190)
3(Graduate University of Chinese Academy of Sciences, Beijing 100190)
{lei.beststones, yunquan.zhang, traits.zhang, liuff8265}@gmail.com

Abstract

In this paper, the mixed precision algorithm to solve the linear system of equations and the implementation of HPL package are introduced. We use this mixed precision algorithm to improve HPL package on CPU+GPGPU heterogeneous clusters, which is named for GHPL, and give the implementation mechanisms in detail. The experimental results are measured on the platforms of multi-core CPUs and CPU+GPGPU heterogeneous clusters. From the experimental results, we can find out that our GHPL program has good scalability on all the experimental environments and can sustain more than 1.7Teraflops both on the cluster with 16 nodes containing 32 NVIDIA Tesla C1060 GPUs and on the cluster with 8 nodes containing 32 NVIDIA GeForce GTX 295 GPUs, while the average speedup of it with respect to HPL is 3.06 and 2.40 respectively.

1. Introduction

The Linpack benchmark is very famous in the HPC society, since it is adopted as a performance metric for ranking supercomputers both in the TOP500[1] list of the world's fastest computers and in the HPC TOP100[2] list of the Chinese mainland fastest computers. In this paper we study and improve the HPL[3] package, High Performance Linpack, which is a reference implementation of the Linpack benchmark written by the researchers of the Innovative Computing Laboratory at the University of Tennessee. HPL is a software package that solves dense linear system of equations using direct LU method in double precision arithmetic on distributed-memory computers. It's the most widely used implementation of the Linpack benchmark. A testing and timing program is provided by HPL to quantify the accuracy of the obtained solution as well as the time it took to finish it. Details of the HPL implementation are available in [4].

Nowadays, the performance of 32-bit operations is usually much higher than the performance of 64-bit operations on General Purpose Graphics Processing Units (GPGPU). The Mixed precision algorithm uses a combination of 32-bit and 64-bit floating point arithmetic to enhance the performance of many dense and sparse linear algebra algorithms significantly while still delivering the 64-bit accuracy of the resulting solution. Thus, in this paper, we try to use the mixed precision algorithm to improve the HPL package on CPU+GPGPU heterogeneous clusters, where both CPUs and GPGPUs are used in synergy.

2. Related work

Mixed precision algorithms stem from the observation that, in many cases, a single precision solution of a problem can be refined to the point where double precision accuracy is achieved. The mixed precision approach was analyzed by Wilkinson[5] and Moler[6]. And it can be applied easily to various problems in linear algebra. The Algorithm 1 is the mixed precision algorithm to solve the linear system of equations presented in [7,8]. From this algorithm we can find out that the most computationally expensive operation, the factorization of the coefficient A, is performed using single precision arithmetic to take advantage of its higher performance. The only parts of operations that must be executed in double precision mode are the residual calculation and the iterative refinement of the solution. In this way that all operations with computational complexity of $O(n^3)$ is handled in single precision, while operations performed in double precision are of at most $O(n^2)$ complexity. In [8], Baboulin et al. also demonstrate by experiments that the maximum number of iterations will be no more than 5 when the condition number of the coefficient matrix A is smaller than 10^6. The limitation of this algorithm is that the condition number of such a coefficient matrix should not exceed the reciprocal of the accuracy of the single precision; otherwise the double precision algorithm should be used.

Dongarra et al. [9] implement the mixed-precision high performance LINPACK benchmark on the CELL processor and they also compare the performance of

978-0-7695-4108-2/10 $26.00 © 2010 IEEE
DOI 10.1109/CIT.2010.212

mixed-precision Linpack with double precision one. A new parallel processing environment for matrix multiplications by using both CPUs and GPUs and a load balancing method for minimizing the execution time are proposed in [10]. Fatica at NVIDIA Corporation [11] accelerates HPL package by implementing a library to intercept the calls to DGEMM and DTRSM on heterogeneous clusters with NVIDIA GPUs.

```
1: $LU \leftarrow PA$            $(\varepsilon_s)$
2:solve $Ly = Pb$         $(\varepsilon_s)$
3:solve $Ux_0 = y$        $(\varepsilon_s)$
  do k = 1, 2, ……..
4: $r_k \leftarrow b - Ax_{k-1}$       $(\varepsilon_d)$
5:solve $Ly = Pr_k$       $(\varepsilon_s)$
6:solve $Uz_k = y$        $(\varepsilon_s)$
7: $x_k \leftarrow x_{k-1} + z_k$       $(\varepsilon_d)$
  check convergence
  done
```

Algorithm 1. Mixed precision, iterative Refinement for Direct Solvers[5]

3. Improvement method

3.1. Preliminary implementation

In our preliminary implementation we simply convert the double precision (64-bit) operations into single precision (32-bit) operations and substitute the calls to double precision subroutines of BLAS to single precision one. The data structure and functions called to check the convergence are maintaining no change. Because the iterative refinement process uses double precision floating point operations, some data structures and functions should be provided with double precision. In our program, we define the data structure *HPL_T_dpmat* to store the double precision coefficient matrix A. The double precision operations functions such as *HPL_pddmatgen* are used to generate the double precision coefficient matrix randomly, *HPL_ddgemv*, *HPL_ddgemm*, *HPL_ddtrsm* etc. are used to call the subroutines in BLAS, and the *HPL_recv_D*, *HPL_send_D*, *HPL_broadcast_D*, *HPL_all_reduce_D*, *HPL_sum_D*, *HPL_max_D* etc. are used to communicate between processes.

3.2. *L* triangular matrix pivoting and permutation matrix *P* storage problem

HPL solves the dense linear system of equations of order *n*:

$$Ax = b; A \in R^{n \times n}; x, b \in R^n$$

by first performing LU factorization with partial pivoting on the *n* by (*n+1*) coefficient matrix:

$$P[A, b] = [[L, U], y]$$

Since the row pivoting (represented by the permutation matrix *P*) and the lower triangular factor *L* are applied to *b* as the factorization progresses, the solution *x* is obtained in one step by solving the upper triangular system $Ux = y$ .The lower triangular matrix *L* is left un-pivoted and the array of pivots is not returned. However, these information will be used at the 5th step of the Algorithm 1. Thus, we improve the HPL package in the following two aspects:

1) Creating an array ipiv[] to store the row pivoting information in every process.
2) Modifying the function of *HPL_pdgesv0* and *HPL_pdgesvk2* to store the row pivoting information and apply it to the previous columns of *L* after factoring a panel.

3.3. $Ly = Pb$ implementation

In HPL package it only uses one function to implement the 1st and 2nd step of Algorithm 1. But the 5th step of Algorithm 1 also needs the function $Ly = Pb$ to be computed alone. Thus we first apply the row pivoting matrix generated in 3.2 to the vector *b* and then implement a function called *HPL_pdtrsv_L* to solve $Ly = Pb$.

As Figure 1 shows, the operations in this function progress step by step on the block size of NB as factoring the panel does. To facilitate description, we assume that we are now dealing with the block L_{kk} on the diagonal. First we call the function *HPL_dtrsv* to solve $L_{kk}y = b'$ and get the solution vector *x*. Then we send the solution *x* to the process that owns the blocks at the below of the block L_{kk} so that the processes that receive *x* will call function *HPL_dgemv* to update their partial *b* elements. After finishing that, the updated partial *b* elements will be broadcasted along the row process grid, then the next block on diagonal can be operated in the same way until the block L_{nn} .

To improve the performance, we use the look-ahead technology in this paper to implement this function. After computing the block L_{kk} , we first send the solution *x* just to the process that owns the block next to L_{kk} (the block with sloping grain in Figure 1) rather than send it to all the process in the current column.

After that, this process will update its partial b elements and broadcast along the row process grid. After that the block $L_{(k+1)\ (k+1)}$ on the diagonal can be computed while the other processes owing blocks below L_{kk} are updating the partial b elements. In this way, these two parts can run in parallel, the performance can be improved a lot.

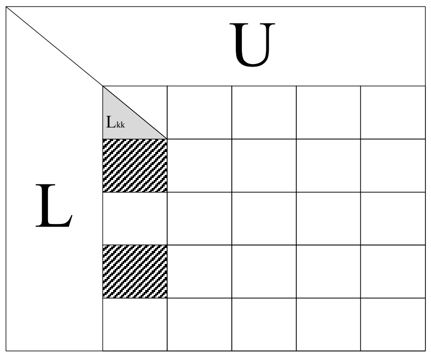

Figure 1. The progress diagram of HPL_pdtrsv_L

3.4. Iterative refinement and convergence checking

The iterative refinement process is implemented in accordance with the 4th to 7th step of Algorithm 1 that uses our improved functions. The convergence checking process is to check whether $\|AX-b\|_{\infty}/(eps*(\|X\|_{\infty}*\|A\|_{\infty}+\|b\|_{\infty})*N)<16.0$ (eps=1.110223e^{-16}) is satisfied just as HPL does. The operations to check convergence are using 64-bit floating point arithmetic.

3.5. Implementation on CPU+GPU heterogenous cluster platform

The SGEMM and STRSM subroutines are running simultaneously on both GPUs and CPU cores through linking with the library of GotoBLAS and CUBLAS for their intensive computational work.

The idea used in this work is as follows:

The SGEMM operation:

$$C=\alpha AB+\beta C$$

can be expressed as(as shown in figure 2):

$$C=\alpha(AB_1+AB_2)+\beta(C_1+C_2)$$

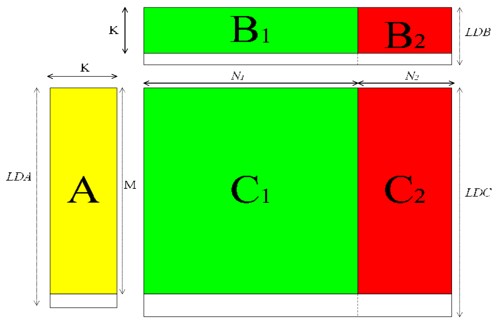

Figure 2.The green portion(left part) is performed on the GPU, while the red portion(right part) is performed on the CPU[11]

The STRSM operation:

$$op(A)*X=alpha*B$$

can be expressed as (as shown in figure 3):

$$op(A)*X=alpha*(B_1+B_2)$$

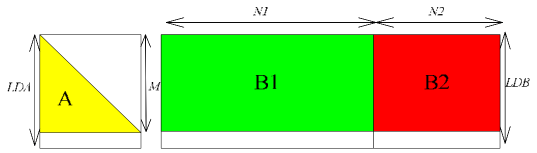

Figure 3.The green portion(left part) is performed on the GPU, while the red portion(right part) is performed on the CPU

The optimal split of the matrix size should keep the time consumed by the GPUs and CPUs equal and this is analyzed in [10,11].

In order to reduce the data transferring time cost between the host to device and device to host, we use the pinned memory mechanism provided by CUDA[12].

4. Experimental results

Our experimental results consist of two parts. In the first part we conducted our experiments on the system described in Table 1. In this part we compared the performance of our improved CPU-only HPL (not link with CUBLAS) with the typical HPL on multi-core platform. To be convenient, our improved HPL is named for MHPL.

Architecture	Clock [GHz]	Memory [GB]	BLAS	HPL	MPI	Compiler
AMD Opteron 870	2.0	16	Netlib*	HPL-2.0	Mpich-1.2.7	gcc3.4.3

Table 1. Hardware and software configuration of the multi-core system for performance experiments

From Figure 4 we can find out that the speedup is increased along with the increase of the matrix size. When the size of matrix is very small the speedup can below 1 since the time consumed by iterative refinement process is nearly the same as the time to

* http://www.netlib.org/blas/

factor the matrix. The performance is decreased due to the extra operations. Figure 5 also shows that MHPL has good scalability when the number of processes increase.

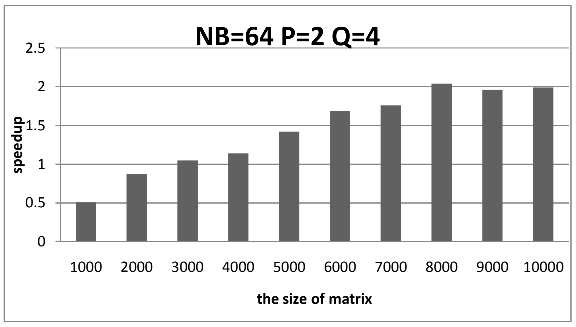

Figure 4 The speedup of MHPL over HPL when increasing the matrix size on CPU cluster

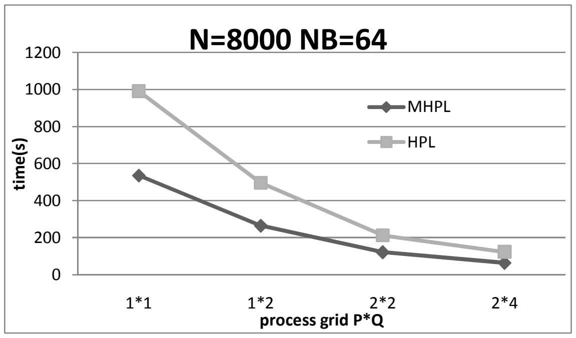

Figure 5. The performance comparison of MHPL and HPL with different number processes

Figure 6 and Figure 7 reflect the performance impact of the iterative refinement process. From them we can find out that along with the increase of the matrix size, the time consumed to refine the solution decreased and can be ignored to some extent.

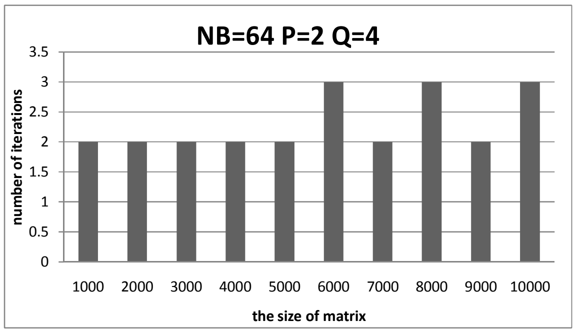

Figure 6. The number of iterations with the increasing of matrix size of MHPL

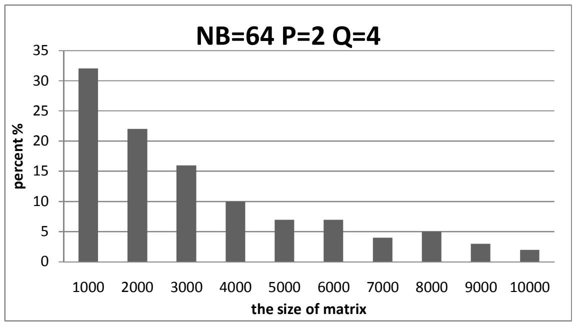

Figure 7. The ratio of the time consumed by iterative refinement process to the total execution time of MHPL

In the second part, the experimental results were measured on the systems described as follows:

1. One cluster with 16 nodes, each node equipped with 2 Tesla C1060 GPUs and 2 Intel Xeon E5410(2.33GHz), with 8GB of memory. These nodes are connected with DDR Infiniband.
2. One cluster with 8 nodes, each node equipped with 4 GeForce GTX 295 GPUs and 2 Intel Xeon E5430(2.66GHz) with 16GB of memory. These nodes are connected with DDR Infiniband.

The experiments are conducted on Linux RHEL4 64-bit OS using GotoBLAS2-1.08 as the host BLAS library. The compiler and MPI version are gcc-4.1.2 and openmpi-1.2.8 respectively, while the CUDA toolkit version is CUDA2.2 and the GPU driver version is NVIDIA x86_64 Kernel Module 190.18. In this paper we use GHPL to refer to this GPU accelerated HPL package. Each GPU is working in collaboration with 2 CPU cores.

In order to get the highest performance, we need to find out the optimal division of computation between CPUs and GPUs. Thus, we run the program with different portions of matrix on GPUs to perform SGEMM and STRSM. And we define the split ratio of matrix on GPU as Rsgemm for SGEMM and Rstrsm for STRSM. As Figure 8 shows that different split ratio, especially the Rsgemm, of matrix on GPU influence the performance greately. And we also find out that when Rsgemm and Rstrsm both equal to 0.92, we can get the optimal performance on our platform.

Table 2 and 3 show the experimental performance results using GHPL on the platform with different number of GPUs. For each process grid size between 1 and 32, we run GHPL with different size of NB parameter and different size of N parameter. Then we select the optimal results. Due to the limited host memory we can't run the program on larger matrix size to get higher performance. But we still get the score that break the Teraflop barrier on both heterogeneous clusters. From the tables we can also find out that the

NB size is bigger than the one a typical CPU Linpack running used. The reason for this is that it controls the parameter K for SGEMM. From figure 9 we can find out that GHPL has good scalability as the number of process increases. Figure 10 shows that along with the increase of the matrix size the performance is getting higher and higher. Thus, we can expect our GHPL's perspective performance on the cluster with larger host memory. Figure 11 and Figure 12 show the speedup of GHPL with respect to HPL for different number of processes on platform 1 and platform 2. Due to the limited host memory, the speedup is drop significantly as the number of processes increase.

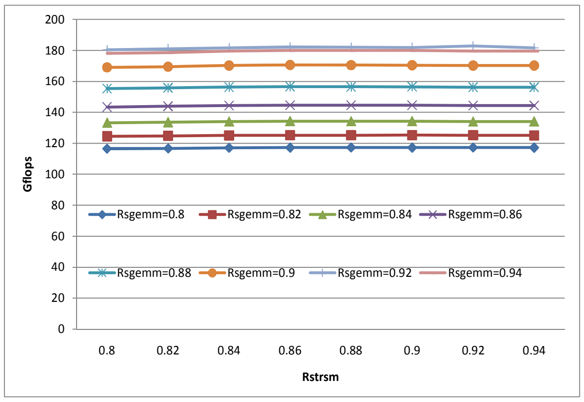

Figure 8. The performance results of using different portions of matrix on GPU to perform SGEMM and STRSM

T/V	N	NB	P*Q	Time(s)	Gflops
WR05R2R4	24000	1600	1	47.25	195.1
WR05R2R4	24000	1664	2	34.09	270.4
WR05R2R4	32000	960	4	51.74	422.2
WR05R2R4	48000	832	8	87.01	847.4
WR05R2R4	63360	960	16	118.52	1431
WR05R2R4	63360	896	32	98.96	1714

Table 2. Performance results using GHPL on platform with Tesla C1060

T/V	N	NB	P*Q	Time(s)	Gflops
WR05R2R4	32000	1920	1	88.736	246.2
WR05R2R4	32000	1664	2	67.28	324.7
WR05R2R4	32000	1600	4	44.39	492.2
WR05R2R4	48000	1152	8	83.79	880.0
WR05R2R4	62720	1152	16	109.73	1499
WR05R2R4	62720	1152	32	92.72	1774

Table 3. Performance results using GHPL on platform with GeForce GTX 295.

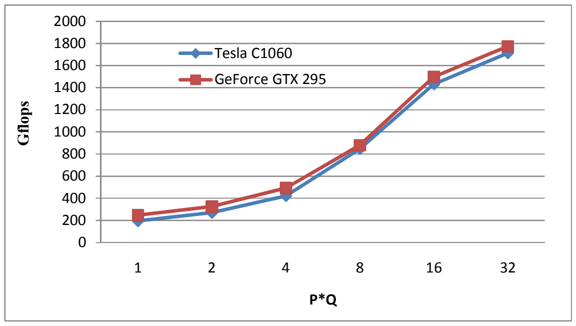

Figure 9. GHPL performance results with different number of processors

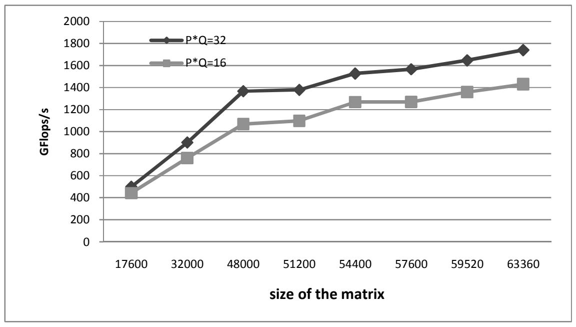

Figure 10. The performance results of GHPL with different matrix sizes

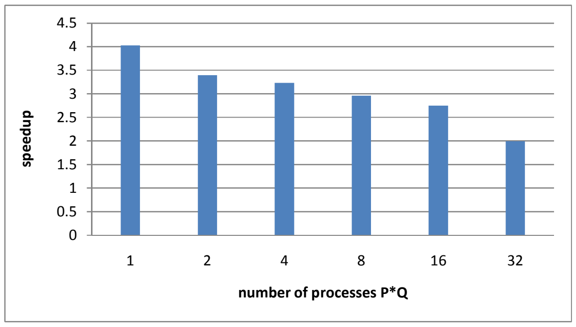

Figure 11. The speedup of GHPL with respect to HPL on platform 1

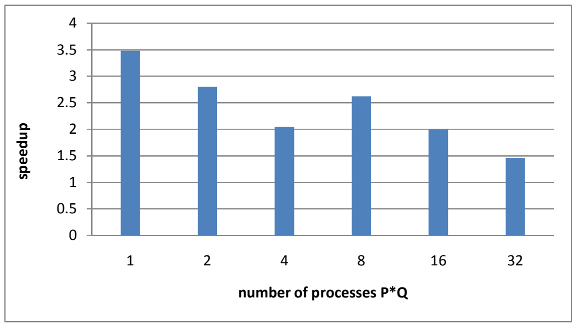

Figure 12. The speedup of GHPL with respect to HPL on platform 2

6. Conclusions and future work

In this paper, we use mixed precision algorithm to improve HPL package. Some experiments are conducted on both multi-core platform and CPU+GPGPU heterogeneous clusters. From our experimental results, we find out that the improved HPL which is named for GHPL has good scalability on all the experimental environments. On multi-core platform the average speedup of GHPL with reference to HPL is nearly 2, very close to the speed of the full single precision solver while delivering the same accuracy as the full double precision one. On the two CPU+GPGPU clusters, though the host memory is limited, the performance of GHPL can sustain more than 1.7Teraflops, while the average speedup of GHPL with respect to HPL is 3.06 and 2.40 respectively.

In future, we can test our GHPL program on the platform with large host memory so that we can obtain much higher performance. We can also transplant our program to the heterogeneous platform with ATI GPU by linking with ACML-GPU library.

7. Acknowledgement

We thank Prof. Wei Ge and Dr. Xianfeng He for their kindly support and providing us with the CPU+GPGPU experimental environments. Our work is partially supported by the NSFC project (No. 60533020), and national 863 project (No.2006AA01A125, No.2009AA01A134 and No.2009AA01A129).

8. References

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

[2] http://www.rdcps.ac.cn/English.htm

[3] http://www.netlib.org/benchmark/hpl/

[4] J. Dongarra, P. Luszczek, A. Petitet, " The Linpack Benchmark: Past, Present and Future", Concurrency and Computation: Practice and Experience, Vol. 15, No. 9, 2003.

[5] Moler, C. B.: Iterative Refinement in Floating Point. J. ACM(2) (1967) 316-321

.

[6] Wilkinson, J. H.: The Algebraic Eigenvalue Problem. Oxford, U.K.: Clarendon, 1965.

[7] J.Langou, J. Langou, P.Luszcek, J.Kurzak, A.Buttari and J.J.Dongarra. Exploiting the performance of 32bit floating point arithmetic in obtatining 64 bit accuracy. In Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, 2006, Tampa USA.

[8] M.Baboulin et al., Accerlating scientific computations with mixed precision algorithms, Computer Physics Communications(2008), doi:10.1016/j.cpc.2008.11.005

[9] Jakub Kurzak, Jack Dongarra. Implementation of the Mixed-Precision High Performance LINPACK Benchmark on the CELL Processor. University of Tennessee Computer Science, Tech. Rep. UT-CS-06-580, LAPACK Working Note 177), September 2006.

[10] Ohshima S, Kise K, Katagiri T, et al. Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment[C]//VECPAR'06 - 7th International Meeting on High Performance Computing for Computational Science. Rio de Janeiro, Brazil: Springer, 2006: 305-318.

[11] M.Fatica, "Accerlerating linpack with CUDA on heterogenous clusters," in Proc. Of 2nd Workshop on General Purpose Processing on Graphics Processing Units, 2009

[12] NVIDIA CUDA Compute Unified Device Architecture Programming Guide