Accelerating Linpack with CUDA on heterogenous clusters

Massimiliano Fatica
NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara CA 95050
mfatica@nvidia.com

ABSTRACT

This paper describes the use of CUDA to accelerate the Linpack benchmark on heterogenous clusters, where both CPUs and GPUs are used in synergy with minor or no modifications to the original source code. A host library intercepts the calls to DGEMM and DTRSM and executes them simultaneously on both GPUs and CPU cores. An 8U cluster is able to sustain more than a Teraflop using a CUDA accelerated version of HPL.

1. INTRODUCTION

The Linpack benchmark is very popular in the HPC space, because it is used as a performance measure for ranking supercomputers in the TOP500 list of the world's fastest computers [1]. The Top 500 list was created in 1993 and it is updated twice a year at the International Supercomputing Conference in Europe and at the Supercomputing Conference in the US. In this study we used HPL [2], High Performance Linpack, which is a reference implementation of the Linpack benchmark written by the Innovative Computing Laboratory at the University of Tennessee. HPL is a software package that solves a (random) dense linear system in double precision arithmetic on distributed-memory computers. It is the most widely used implementation of the Linpack Benchmark and it is freely available from Netlib (http://www.netlib.org/benchmark/hpl). The HPL package provides a testing and timing program to quantify the accuracy of the obtained solution as well as the time it took to compute it.

We performed benchmarks on two different systems, a workstation with a single GPU and an 8-node cluster with multiple GPUs, with the following specifications:

1. SUN Ultra 24 workstation with an Intel Core2 Extreme Q6850 (3.0GHz) CPU and 8GB of memory plus a Tesla C1060 card.

2. Cluster with 8 nodes, each node connected to half of a Tesla S1070 system, containing 4 GPUs, so that each node is connected to 2 GPUs. Each node has 2 Intel Xeon E5462 (2.8GHz with 1600Mhz FSB) and 16GB of memory. The nodes are connected with SDR (Single Data Rate) Infiniband.

GPGPU '09 Washington DC, USA

Peak performance for the CPU is computed as the product of the number of cores, the number of operations per clock and the clock frequency. The CPUs in both systems have 4 cores and are able to issue 4 double precision operations per clock, so the peak performance is $16 * \text{clock frequency}$.

The first system has a peak double precision (DP) CPU performance of 48 GFlops, the second system has a peak DP CPU performance of 89.6 GFlops per node (total peak CPU performance for the cluster 716.8 GFlops).

2. GPU ARCHITECTURE AND CUDA

The GPU architecture has now evolved into a highly parallel multi-threaded processor with very high floating point performance and memory bandwidth.

The last generation of NVIDIA GPUs also added IEEE-754 double-precision support. NVIDIA's Tesla, a product line for high performance computing, has GPUs with 240 single precision and 30 double precision cores and 4 GB of memory. The double precision units can perform a fused multiply add per clock, so the peak double precision performance is $60 * \text{clock frequency}$. The PCI-e card (C1060) has a clock frequency of 1.296GHz and a 1U system with 4 cards (S1070) has a clock frequency of 1.44GHz.

The GPU is especially well-suited to address problems that can be expressed as data-parallel computations, i.e. the same program is executed on many data elements in parallel with high arithmetic intensity (the ratio of arithmetic operations to memory operations). CUDA [3] is a parallel programming model and software environment designed to expose the parallel capabilities of GPUs. CUDA extends C by allowing the programmer to define C functions, called kernels, that when called are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.

The software environment also provides a performance profiler, a debugger and commonly used libraries for HPC:

1. CUBLAS library: a BLAS implementation

2. CUFFT library: an FFT implementation.

The implementation described in this paper has been performed using the CUBLAS library and the CUDA runtime, no specialized kernels have been written.

3. FAST PCI-E TRANSFER

The sustainable bandwidth from the host to device (and vice versa) plays a key role in the acceleration of single DGEMM or DTRSM calls. CUDA offers a fast transfer mode that is enabled when page-locked memory (sometimes called pinned memory) is used. A PCI-e transfer from pageable memory incurs in an extra copy on the CPU before it transfers via DMA to the GPU. By using a special memory region, this extra copy can be eliminated. This special allocation is done using the function *cudaMallocHost* instead of a regular *malloc*, as shown in Table 1.

```
// Regular malloc/free
 double *A;
 A = malloc(N*N*sizeof(double));
 free(A);


// Page-locked version
 double *A;
 cudaMallocHost(A,N*N*sizeof(double));
 cudaFreeHost(A);
```

Table 1: Comparison between regular malloc/free calls and page-locked versions.

In the current CUDA version (2.1) the amount of pinned memory is limited to 4 GB (this restriction will be removed in subsequent versions). *The changes from regular malloc calls to cudaMallocHost were the only changes made to the original HPL source code*, the rest of the acceleration is obtained with a host library that intercepts the relevant DGEMM and DTRSM calls (a change in the HPL Makefile is required for the proper linking).

One of the CUDA SDK examples (bandwidthTest) can measure the sustainable bandwidth. Table 2 shows how the bandwidth increases from around 2 GB/s for pageable memory to more than 5 GB/s for pinned memory. We also note an asymmetry in the H2D (host to device) and D2H (device to host) bandwidth. On certain systems, the disparity can be greater than the number shown in the table, so we will use two different parameters in our performance model later. The sustained bandwidth is related to the CPU, motherboard chipset, BIOS setting and the OS. The only way to get reliable transfer speed is to measure it.

	Sun Ultra 24	
	Pageable memory	Pinned memory
H2D	2132 MB/s	5212 MB/s
D2H	1882 MB/s	5471 MB/s

	Supermicro 6015TW	
	Pageable memory	Pinned memory
H2D	2524 MB/s	5651 MB/s
D2H	2084 MB/s	5301 MB/s

Table 2: PCI-e bandwidth for a 32 MByte payload on x16 PCI-e gen2 slots. H2D indicates host to device transfers, D2H indicates device to host transfers.

4. LINPACK BENCHMARK

The Linpack benchmark is an implementation of the LU decomposition to solve a dense $N \times N$ system of linear equations:

$$Ax = b$$

with $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$. The solution is obtained by Gaussian elimination with partial pivoting, resulting in a floating point workload of $2/3N^3 + 2N^2$. The benchmark is run for different matrix sizes N in an effort to find the size N_{max} for which the maximum performance R_{max} is obtained. The benchmark also reports the problem size $N_{1/2}$ where half of the performance $R_{max/2}$ is achieved.

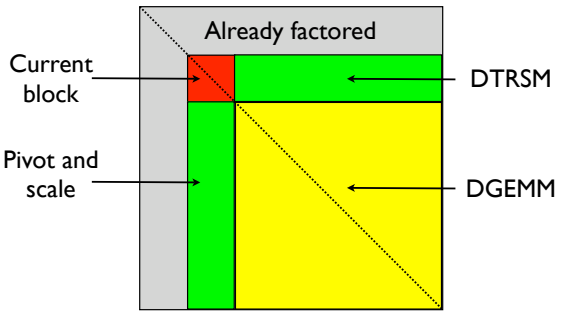

Figure 1: LU factorization: the grey area represents the portion of the matrix already factored. The red area is the current block being factorized. Once this factorization is ready, it is applied to the sub-matrix on the right. The final step is to update the trailing sub-matrix in yellow.

There are several variants of the blocked LU decomposition: left-looking, right-looking, Crout (see [4] for a technical description). Most of the computational work for these variants is contained in three routines: the matrix-matrix multiply (DGEMM), the triangular solve with multiple right-hand sides (DTRSM) and the unblocked LU factorization for operation within a block column. The right-looking variant of the LU factorization is shown in Figure 1. Details of the HPL implementation are available in [5].

Most of the Linpack runtime is spent in matrix-matrix multiplies for the update of trailing matrices (the yellow area in Figure 1). The bigger the problem size N is, the more time is spent in this routine, so optimization of DGEMM is critical to achieve a high Linpack score.

5. DGEMM PERFORMANCE

Matrix-matrix multiply is a basic building block in many algorithms and there are a set of functions in BLAS (Basic Linear Algebra Subroutines) to perform this operation: when the data is in double precision format, the function call is DGEMM.

If A(M,K), B(K,N) and C(M,N) are the input matrices, a DGEMM call will compute $C = \alpha AB + \beta C$. The BLAS interface can handle leading dimensions of A, B and C (lda, ldb and ldc) different from M,K and N. It also provides support for transposed input and output, but we will focus on non-transposed matrices in the following analysis. For the example in figure 2, the BLAS call will be:

```
DGEMM('N','N',m,n,k,alpha,A,lda,B,ldb,beta,C,ldc)
```

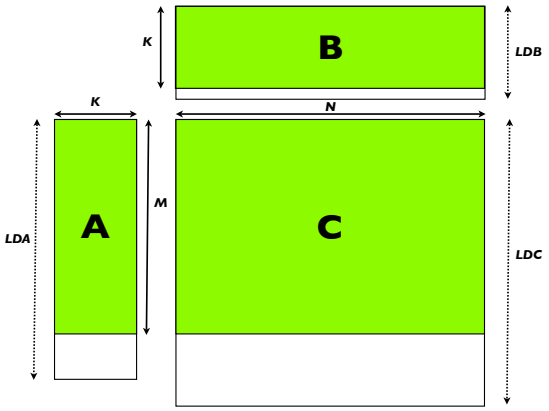

Figure 2: Matrix-matrix multiply: $C := \alpha * op(A) * op(B) + \beta * C$

DGEMM is one of the few workloads where CPUs sustain performance close to their peak. There are several high-performance implementations available: a popular one is the GotoBLAS library from University of Texas and vendor-provided libraries like Intel MKL and AMD ACML.

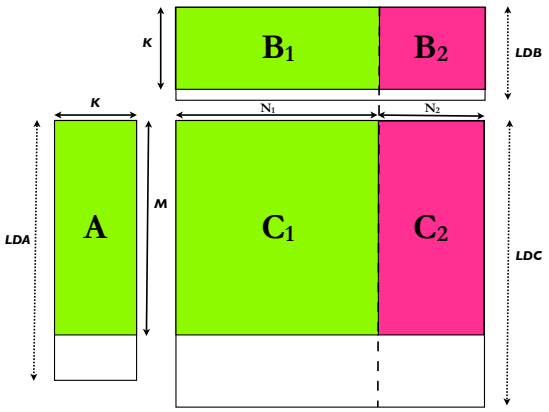

Figure 3: The green portion is performed on the GPU, while the red one is performed on the CPU)

The basic idea used in this work is very simple: to accelerate the matrix-matrix multiply operation, the original matrices C and B are viewed as the union of two sub-matrices $C = C_1 \cup C_2$ and $B = B_1 \cup B_2$ with $N = N_1 + N_2$ (as shown in figure 3).

The DGEMM operation:

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

can be expressed as :

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

The original call:

```
DGEMM('N', 'N', m,n,k,alpha,A,lda,B,ldb,beta,C,ldc)
```

is translated into two independent calls:

```
DGEMM('N','N',m,n1,k,alpha,A,lda,B1,ldb,beta,C1,ldc)
DGEMM('N','N',m,n2,k,alpha,A,lda,B2,ldb,beta,C2,ldc)
```

Since these operations are independent, we can perform the first on the GPU and the second on the CPU cores. Once the data needed from the GPU has been transferred we can overlap the execution of the two DGEMMs as shown in figure 4.

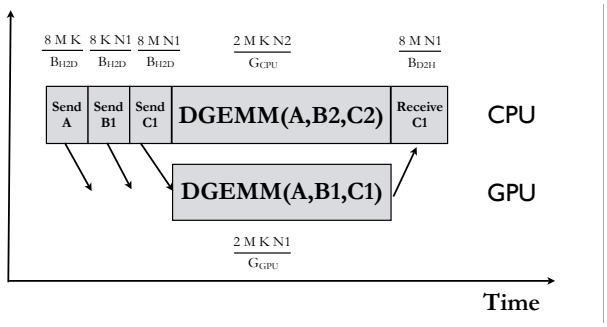

Figure 4: Data flow to split the DGEMM call between CPU cores and GPU.

Using the helper function of CUBLAS to move data between the CPU and GPU, it is very easy to implement this flow (figure 5).

In order to maximize performance, we will need to find the best division of computation between CPU and GPU. Since the GPU and CPU have two different memory spaces, the data necessary to perform the GPU DGEMM will need to be copied across the PCI-e bus.

The important factors in the analysis are going to be:

B_{H2D}: Bandwidth from host to device expressed in GB/s

G_{GPU}: Sustained performance of DGEMM on the GPU expressed in $GFlops$

G_{CPU}: Sustained performance of DGEMM on the CPU expressed in $GFlops$

B_{D2H}: Bandwidth from device to host expressed in GB/s

A DGEMM call on the host CPU performs $2KMN$ operations, so if the CPU cores can perform this operation at G_{CPU} the total time is:

$$T_{CPU}(M,K,N) = 2\frac{MKN}{G_{CPU}}$$

The total time to offload a DGEMM call to the GPU has an I/O component that accounts for both the data transfer from the CPU memory space to the GPU memory space and vice versa plus a computational part once the data is on the GPU. We can express this time as:

$$T_{GPU}(M,K,N) = 8\frac{(MK+KN+MN)}{B_{H2D}} + 2\frac{MKN}{G_{GPU}} + 8\frac{(MN)}{B_{D2H}}$$

the factor 8 is the size of a double in bytes.

The optimal split will be

$$T_{CPU}(M,K,N2) = T_{GPU}(M,K,N1) \quad \text{with} \quad N = N1+N2$$

```
// Copy A from CPU memory to GPU memory devA
   status = cublasSetMatrix (m, k    , sizeof(A[0]), A, lda, devA, m_gpu);
// Copy B1  from CPU memory to GPU memory devB
   status = cublasSetMatrix (k ,n_gpu, sizeof(B[0]), B, ldb, devB, k_gpu);
// Copy C1  from CPU memory to GPU memory devC
   status = cublasSetMatrix (m, n_gpu, sizeof(C[0]), C, ldc, devC, m_gpu);

// Perform  DGEMM(devA,devB,devC) on GPU
// Control immediately return to CPU
   cublasDgemm('n', 'n', m, n_gpu, k, alpha, devA, m,devB, k, beta, devC, m);

// Perform  DGEMM(A,B2,C2) on CPU
   dgemm_cpu('n','n',m,n_cpu,k, alpha, A, lda,B+ldb*n_gpu, ldb, beta,C+ldc*n_gpu, ldc);

// Copy devC  from GPU memory to CPU memory C1
   status = cublasGetMatrix (m, n, sizeof(C[0]), devC, m, C, *ldc);
```

Figure 5: Code to overlap CPU and GPU DGEMM computations following the data flow of Fig. 4

For an initial approximation of the optimal split fraction $\eta = N1/N$, we can omit the transfer time ($O(N^2)$) compared to the computation ($O(N^3)$). From a simple manipulation, the optimal split is

$$\eta = \frac{G_{GPU}}{G_{GPU} + G_{CPU}}$$

On the cluster, where the quad core Xeon has a DGEMM performance of 40 GFlops and the GPU a DGEMM performance of 82 GFlops, this formula predicts $\eta = 0.67$, very close to the optimal value of 0.68 found by experiments.

It turns out that on Intel systems using Front Side Bus (FSB), it is better not to overlap the transfer to the GPU with computations on the CPU (the memory system cannot supply data to both the PCIe and the CPU at maximum speed). The flow depicted in Figure 4 gives better results than trying to overlap data transfer too. On the new Intel systems with Quick Path Interconnect (QPI), this may not be the case and an overlap of data transfer and CPU computation may lead to increase performances.

It is clear from this model, that we want to maximize B_{H2D}, G_{GPU} and B_{D2H} in order to minimize the time to offload a DGEMM call to the CPU. As discussed in the previous section on fast PCI-e transfer, the way to maximize B_{H2D} and B_{D2H} is to use page-locked memory, so the missing piece is how to achieve the best possible G_{GPU}.

The DGEMM function call in CUBLAS maps to several different kernels depending on the size of the matrices. The best performance is achieved when M is multiple of 64 and K and N multiple of 16. Performance numbers for different choices of M, K and N are shown in table 5. When all the above conditions are satisfied, the GPU can achieve 82.4 Gflops, 95% of the peak double precision performance.

M	K	N	M%64	K%16	N%16	$Gflops$
448	400	12320	Y	Y	Y	82.4
12320	400	1600	N	Y	Y	72.2
12320	300	448	N	N	Y	55.9
12320	300	300	N	N	N	55.9

Table 3: DGEMM performance on the Tesla S1070 GPU (1.44 GHz) with data resident in GPU memory

One of the benefits of the split between the GPU and CPU cores is that we can always send optimally sized DGEMMs to the GPU. We can also apply the same idea multiple times (and in multiple directions) to multiply matrices that will not fit in the GPU memory or to use multiple GPUs. The library used for the Linpack runs does exactly this: it can split B and C or A and C, it handles arbitrary sizes and it always sends the optimal workload to the GPU.

Figure 6 shows the DGEMM performance for CPU only, GPU only (including transfer time) and for the combined GPU/CPU approach. The presented approach is very effective in combining the performance of the two devices, and can be applied to DTRSM too.

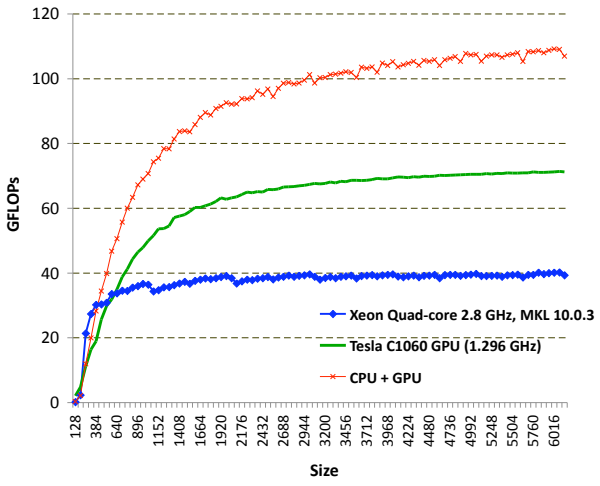

Figure 6: DGEMM performance on the CPU (4 cores), on the GPU (including I/O time) and on the CPU+GPU using the current approach

6. RESULTS

The Linpack runs have been performed on Linux RHEL4 64-bit OS using the Intel MKL version 10.0.3 as the host BLAS library. We use as many MPI processes as there are GPUs. A shared library intercepts the DGEMM and DTRSM calls and decides whether to send them back to the CPU (if too small) or to execute them on both the GPU

and CPU cores. Each GPU is working in collaboration with a certain number of cores (4 on the two systems used for benchmarks) and relies on the host library for the utilization of all the cores (usually done with OpenMP or pthreads). It is important to notice that the overall performance numbers are dependent on the host CPU, the chipset, and the host BLAS library. We use the parameter NB in the HPL.dat input file to control the parameter K for the DGEMM. While a typical CPU Linpack run will use NB equal to 128 or 196, with the CUDA accelerated Linpack we tend to use bigger block sizes ($NB = 960, 1152$).

Table 4 shows the Linpack scores on the Sun workstation. In all the tables, N indicates the size of the matrix, NB the block size, P and Q the number of processes in a 2D grid (for parallel runs), T/V encodes details of the run such as look-ahead, type of recursive factorization and recursive stopping criterium (see [2] for a detailed explanation).

The peak performance of the CPU is 48 GFlops, a CPU-only Linpack sustains performance in the $70\% - 80\%$ range for very large problem sizes. The CUDA accelerated version with pinned memory (limited today to 4GB per MPI process) delivers 83.2 Gflops, 66% of the combined 125 GFlops system peak performance (the Tesla C1060 has a peak performance of 77.7 Gflops). The accelerated solution also delivers good performance for small problem sizes and is not very sensitive to the size of NB.

T/V	N	NB	P	Q	Time(s)	$Gflops$
WR00L2L2	23040	960	1	1	97.91	83.28
WR00L2L2	7432	960	1	1	5.47	50.01
WR00L2L2	7432	1152	1	1	5.47	53.56

Table 4: Linpack performance on a Sun Ultra 24 workstation with 1 Intel(R) Core2 Extreme Q6850 (3.0Ghz) and a Tesla C1060 using HPL with pinned memory for different N and NBs.

Table 5 shows the same results using pageable memory. The use of pinned vs. regular pageable memory increases the performance from 70 to 84 Gflops.

T/V	N	NB	P	Q	Time(s)	$Gflops$
WR00L2L2	23040	960	1	1	117.06	69.66
WR00L2L2	23040	1152	1	1	117.06	70.64
WR00L2L2	7432	960	1	1	6.09	44.94
WR00L2L2	7432	1152	1	1	5.97	45.86

Table 5: Linpack performance on a Sun Ultra 24 workstation with 1 Intel(R) Core2 Extreme Q6850 (3.0Ghz) and a Tesla C1060 using HPL with pageable memory

Tables 6 and 7 show the results of Linpack runs using 16 GPUs.

A small 8-U cluster can easily break one Teraflop. Using pinned memory and a small problem size (for a cluster run) fitting in 4GB per process, the cluster delivers 1.089Tflops. For comparison, the first system to break the Teraflop barrier on Top 500 was ASCI Red in June 1997. The ASCI Red system at Sandia National Laboratory achieved performance of 1.068 Tflop/s with 7264 Pentium P6 processors.

T/V	N	NB	P	Q	Time(s)	$Gflops$
WR10L2R4	92164	960	4	4	479	1089

Table 6: Linpack performance on cluster with 8 nodes (each with 2 Intel Xeon 2.8Ghz) and 4 Tesla S1070, for a total of 16 CPUs and 16 GPUs with pinned memory

T/V	N	NB	P	Q	Time(s)	$Gflops$
WR12R2C1	118336	960	4	4	1037	1065

Table 7: Linpack performance on cluster with 8 nodes (each with 2 Intel Xeon 2.8Ghz) and 4 Tesla S1070, for a total of 16 CPUs and 16 GPUs with pageable memory

When running with Infiniband, we have observed a possible problem with pinned memory depending on the size of broadcasted panel that we are currently investigating. A workaround is to disable the RDMA features of Infiniband. All the cluster runs were performed with OpenMPI 1.2.5. With this MPI stack is very easy to disable RDMA via a flag in mpirun (–mca btl_openib_flags 1).

6.1 Additional results

Using a pre-release version of CUDA, we can now use more than 4GB of page-locked memory per MPI process.

On the workstation, the biggest problem that can be solved with the available memory is $N = 32320$ and the Linpack score is now 90 Gflops, 72% of peak performance.

Sun Ultra 24						
T/V	N	NB	P	Q	Time(s)	$Gflops$
WR00R2L2	32320	1152	1	1	250.01	90
Cluster						
T/V	N	NB	P	Q	Time(s)	$Gflops$
WR11R2L2	118114	960	4	4	874	1258

Table 8: Linpack performance using HPL with pinned memory (pre-release CUDA version)

On the cluster, we can now use all the memory on the nodes and solve a problem with $N = 118114$. As shown in Table 8, the Linpack score is now 1.258 Teraflops.

In order to put this result in perspective, looking at the historic performance data collected by Jack Dongarra [6], this small cluster is ahead of a 304 cores HP Integrity rx2600 Itanium2 1.3 GHz cluster with Myrinet (Linpack score of 1253 Gflops) and a 400 cores IBM eServer 2.2 GHz Opteron cluster with Infiniband (Linpack score of 1246 Gflops).

7. CONCLUSIONS

With CUDA and Tesla, it was very easy to boost the Linpack performance of both a workstation and cluster. As the library that intercepts the DGEMM and DTRSM improves (with better heuristics for splitting), as new chipsets that can sustain better I/O transfer appear, and as current limitations in the amount of pinned memory are relaxed, CUDA accelerated Linpack results will keep getting faster. The use

of GPUs allows one to reduce the number of host nodes required to reach a target performance level and can significantly reduce the cost of high performance interconnects.

8. REFERENCES

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

[2] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - a portable implementation of the high performance Linpack benchmark for distributed memory computers, version 2.0. `http://www.netlib.org/benchmark/hpl/`

[3] NVIDIA CUDA Compute Unified Device Architecture Programming Guide

[4] J. J. Dongarra, L. S. Duff, D. C. Sorensen, and H. A. V. Vorst. *Numerical Linear Algebra for High Performance Computers*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

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

[6] J. J. Dongarra. Performance of various computers using standard linear equations software. Technical report, 2008. `http://www.netlib.org/benchmark/performance.pdf`