Neural Computing and Applications
https://doi.org/10.1007/s00521-018-3354-z

ORIGINAL ARTICLE

CPU versus GPU: which can perform matrix computation faster—performance comparison for basic linear algebra subprograms

Feng Li[1] · Yunming Ye[1] · Zhaoyang Tian[2] · Xiaofeng Zhang[1]

Received: 8 November 2017 / Accepted: 8 January 2018

Abstract
Matrix computing is the core component of machine learning and artificial intelligence. Fast matrix computations can facilitate many large-scale computational projects greatly. Basic linear algebra subprograms (BLAS) are proposed, which classify different matrices and provide a standardized interface. Currently, the most commonly used heterogeneous computing platforms are central processing unit (CPU) and graphics processing unit (GPU). At present, BLAS has been implemented on both CPU and GPU. However, due to the different characteristics of algorithms and hardware, a particular matrix method should be designed for a particular processor. It is important to choose the right processor for a particular matrix computation. This paper first briefly reviews the BLAS, and then introduces architecture and optimization methods of CPU and GPU. The effect of different subroutines in BLAS is studied through experiments. Finally, we discuss the reasons and the processor selection scheme of matrix computations.

Keywords Matrix computation · Basic linear algebra subprograms · CPU · GPU

1 Introduction

With the rapid development of large-scale web applications, there arises a huge demand for scientific computing [1–3]. Heterogeneous computing is already the third revolution of calculation after single-core computing and multi-core computing. It is also an effective way for massive calculation. Two major computing platforms are developed to cope with the emerging demands. The first one is the general-purpose central processing unit (CPU) [4] and the second one is the graphics processing unit (GPU) [5, 6]. CPU can run many types of applications and recently proposed multiple core versions to process data in parallel, while GPU is designed for graphics processing with many small processing elements. The massive processing capability of GPU allures some programmers to start exploring general-purpose computing with GPU, which becomes an emerging research field, called general-purpose computation on GPU (GPGPU).

Fundamentally, CPUs and GPUs are built based on different philosophies. CPUs are designed for a wide variety of applications and provide quick response to a single task. Architectural advances such as branch prediction, out-of-order execution, and super-scalar (in addition to frequency scaling) are responsible to improve performance. However, these advances came at the price of a high complexity and power consumption. As a result, main stream CPUs can pack only a small number of processing cores on the same die to stay within the power and thermal envelopes.

GPUs are built specifically for rendering graphics applications which have a large degree of data parallelism (e.g., each pixel is processed independently). Graphics applications are also latency tolerant (the processing of

✉ Feng Li
lifeng@stu.hit.edu.cn

✉ Xiaofeng Zhang
zhangxiaofeng@hit.edu.cn

Yunming Ye
yeyunming@hit.edu.cn

Zhaoyang Tian
zytian3-c@my.cityu.edu.hk

[1] Shenzhen Key Laboratory of Internet Information Collaboration, Harbin Institute of Technology Shenzhen Graduate School, Shenzhen, China

[2] Department of Electronic Engineering, City University of Hong Kong, Kowlloon Tong, Hong Kong

Published online: 25 January 2018

each pixel can be delayed if frames are processed at interactive rates). As a result, GPUs can make best trade between single-thread performance and parallel processing. For instance, GPUs can switch from processing one pixel to another one when long latency events, such as memory accesses, are encountered and can switch back to the former pixel. This approach works well when there exist a data-level parallelism.

Linear algebra, in particular the solution of linear systems of equations and eigenvalue problems, is fundamental to most calculations in scientific computing and is often the most computationally intensive part of such calculations. Thus this paper focuses on this area and evaluates algorithms and softwares for matrix computation.

To implement linear algebraic operations, certain low-level operations, such as dot product or matrix–vector product, are usually chosen to separate subprograms. This could be observed in many released source codes developed for related applications. This approach encourages structured programming and improves the code quality of the software by specifying basic building blocks and identifying these operations with unique mnemonic names. Since a significant amount of execution time in complicated linear algebraic programs may be spent on only a few low-level operations, we can reduce the overall execution time of the program by reducing the execution time spent on these operations.

The programming of these low-level operations involves algorithmic and implementation subtleties which are generally overlooked. If there is general agreement on standard names and parameter lists for some of these basic operations, the portability and efficiency could be achieved.

In 1979, Lawson et al. [7] described the advantages of adopting a set of basic routines for problems in numerical linear algebra. Basic linear algebra subprograms (BLAS) [7] is a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot product, linear combination, and matrix multiplication, which are widely adopted by many machine learning applications.

The contributions of this paper are summarized as: (1) we introduce BLAS and compare the different architectural features of CPU and GPU (2) we evaluate the different subprograms in BLAS through extensive experiments (3) we propose a processor selection scheme which can guide the optimization of the large computational task.

The rest of this paper is organized as follows: Sect. 2 introduces the related works. Section 3 introduces the details of BLAS. Section 4 describes the two main computing platforms, CPUs and GPUs, and provides a platform-specific optimization guide. Section 5 shows the performance of different subprograms on the two platforms. Section 6 discusses the reasons and proposes the platform selection scheme of matrix computations. Section 7 concludes our findings.

2 Related work

BLAS is a specification that prescribes a set of low-level routines for performing common linear algebra operations. Although the BLAS specification is general, BLAS implementations are often optimized to work on a particular machine to achieve substantial performance improvement. BLAS implementations take advantage of special floating-point hardware such as vector registers or single-instruction multiple data (SIMD) instructions.

Most libraries that offer linear algebra routines conform to the BLAS interface, allowing library users to develop programs that are agnostic of the BLAS library being used. Examples of BLAS libraries include: AMD Core Math Library (ACML) [8], ATLAS, Intel Math Kernel Library (MKL) [9], OpenBLAS and CUBLAS [10]. ACML is no longer supported by its producer. ATLAS is a portable library that automatically optimizes itself for an arbitrary architecture. MKL is a freeware and proprietary vendor library optimized for X86 and X86-64 with a performance emphasis on Intel processors. OpenBLAS is an open-source library that is hand-optimized for many of the popular architectures. CUBLAS is an implementation of BLAS on top of the NVIDIA CUDA runtime and allows users to access the computational resources of NVIDIA GPU.

Many numerical software applications use BLAS-compatible libraries to perform linear algebra computations, including Armadillo, LAPACK [11], LINPACK, GNU Octave, Mathematica, MATLAB [12], NumPy [13], and R. Nearly, all the matrix methods [14, 15] in machine learning will use BLAS. GPU-accelerated BLAS, which is a powerful tool for designing and deploying GPU-accelerated deep learning, is adopted by TensorFlow [16], Torch, Caffe [17], Theano and Deeplearning4j. On the top of these platforms, many deep learning approaches have been for task of text [18], image [19], video [20] and recommender system [21].

In this paper, we compare the computing efficiency of different subroutines on different platforms. Our goal is to provide a guide to researchers who are engaged in scientific computing and developing machine learning systems.

3 Basic linear algebra subprograms

3.1 BLAS routines

BLAS provides a standard interface to several common vector operations, and its naming convention consists of a

four- or five-letter mnemonic name preceded by one of the letters **S**, **D**, **C**, **Z** indicating precision type single, double, single complex and double complex. The single and double precision correspond to the basic data types in C or Fortran. The single complex precision is a structure that contains single-precision real and imaginary parts, while the double complex precision is a structure that contains double-precision real and imaginary parts.

BLAS promotes modularity by identifying frequent operations of linear algebra and specifying a standard interface for these operations. Efficiency is achieved by optimizing BLAS without modifying the higher-level referencing codes. Obviously, it is important to identify and define a set of basic operations that are rich enough to enable an expression of high-level algorithms and are simple enough to admit a high level of optimization on a variety of computers. Such optimizations can often be achieved through modern compilation techniques, or hand coding in assembly language alternatively. Use of these optimized operations can yield dramatic reduces in computation time.

BLAS functionality is categorized into three sets of routines called **levels**, which correspond to both the chronological order of definition and publication, as well as the degree of the polynomial in the complexities of algorithms; Level 1 BLAS operations typically take linear time, $O(n)$, Level 2 operations are quadratic time, $O(n^2)$, and Level 3 need operations cubic time, $O(n^3)$. Modern BLAS implementations typically provide all three levels.

3.1.1 Level 1 BLAS

The original set of BLAS performs low-level operations such as dot product and the adding of a multiple vector to another one. We refer to these vector–vector operations as Level 1 BLAS.

The following types of basic vector operations are performed by the Level 1 BLAS:

AXPY $y \leftarrow \alpha x + y$

DOT $dot \leftarrow x^T y$

SCAL $x \leftarrow \alpha x$

NRM2 $nrm2 \leftarrow \|x\|_2$

SWAP $x \leftrightarrow y$

COPY $y \leftarrow x$

AMAX $\text{amax} \leftarrow \|x\|_\infty$

ASUM $\text{asum} \leftarrow \|\text{re}(x)\|_1 + \|\text{im}(x)\|_1$

Typically, these operations involve $O(n)$ floating-point operations and $O(n)$ data items moved (loading or stored), where n is the length of the vectors.

3.1.2 Level 2 BLAS

With the advent of high-performance computers which employ vector processing, it is then recognized that additional BLAS should be proposed, since the Level 1 BLAS do not have sufficient granularity to admit optimizations such as reuse of data in registers and reduction in memory access. One needs to optimize at least the level of matrix–vector operations to acquire the potential efficiency; and the Level 1 BLAS inhibits this optimization, because they hide the matrix–vector operations from the compiler.

Thus, an additional set of BLAS, called the Level 2 BLAS [22], was designed for a small set of matrix–vector operations that occur frequently in the implementation of many common algorithms on linear algebra. The Level 2 BLAS involve $O(mn)$ scalar operations, where m and n are the dimensions of the matrix involved. The following two types of basic operations are performed by the Level 2 BLAS:

(1) **GEMV**: Matrix–vector products

$$y \leftarrow \alpha Ax + \beta y$$
$$y \leftarrow \alpha A^T x + \beta y$$
$$y \leftarrow \alpha A^H x + \beta y$$

where α and β are scalars, x and y are vectors, and A is a matrix.

(2) **TRSV**: Solution of triangular equations

$$x \leftarrow T^{-1} x$$
$$x \leftarrow T^{-T} x$$
$$x \leftarrow T^{-H} x$$

where T is a nonsingular upper or lower triangular matrix.

3.1.3 Level 3 BLAS

The Level 3 BLAS [23] is proposed for the matrix–matrix operations required for these purposes. The routines are derived in an obvious manner from some of the Level 2 BLAS, by replacing the vectors x and y with matrices B and C.

For arithmetic, the operations for the Level 3 BLAS have the following forms:

(1) **GEMM**: Matrix–matrix products

$$C \leftarrow \alpha AB + \beta C$$
$$C \leftarrow \alpha \mathrm{op}(A)B + \beta C$$
$$C \leftarrow \alpha A\mathrm{op}(B) + \beta C$$
$$C \leftarrow \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C$$

(2) **TRSM**: Solving triangular systems of equations with multiple right-hand sides:

$$B \leftarrow \alpha T^{-1}B$$
$$B \leftarrow \alpha T^{-T}B$$
$$B \leftarrow \alpha BT^{-1}$$
$$B \leftarrow \alpha BT^{-T}$$

where op means simple transposition in real arithmetic and conjugate transposition in complex arithmetic, α and β are scalars, A, B and C are rectangular matrices, and T is an upper or lower triangular matrix.

3.2 BLAS and machine learning

Linear algebra methods are basic ingredients in many data mining and pattern recognition techniques. These methods use matrices to model the problem. Subroutines in BLAS are used when solving and applying models.

For example, when performing latent semantic indexing or ancient handwriting recognition, singular value decomposition (SVD) is used. The SVD process uses some BLAS subroutines, such as GEMV, GEMM and so on. In the least square learning, we need to decompose matrix generated by training data in a product of an orthogonal matrix and a triangular matrix, which is called QR decomposition. The least square solution can be obtained by the result of QR decomposition via TRSV or TRSM. In deep neural network models, GEMM is used in both forward and backward propagation. GEMM is used in many places in Caffe, and the code example is shown in Figs. 1 and 2.

The matrices mentioned in these methods are real matrices, that is, they use only float or double precision. In the latest research, complex matrices or vectors are also used in machine learning problems. Researchers in DeepMind explore the use of complex values to form associative memory neural networks [25]. This system is used to enhance LSTM memory. The conclusion of the paper is that the network with complex values can obtain more memory capacity.

Of course, in practical problems, due to the huge amount of data, it is necessary to carry out distributed computing. Subtasks assigned to each machine will still use these subroutines.

4 CPU and GPU computing

4.1 Multi-core CPU

CPUs are designed as the central control units that perform basic compute and control operations of computer systems. Because the CPU is responsible for processing and responding, its single-thread performance is highly emphasized. Inside one single CPU core, various parallelism-oriented techniques, such as multiple functional units, pipelining, super-scalar, out-of-order execution, branch prediction, simultaneous multi-threading and deep cache hierarchies, have been designed.

Because of the so-called power wall, clock frequency of a single CPU core cannot be further increased. As a result, multiple CPU cores of lower clock frequency have to be combined onto a single chip for further performance improvements.

The latest multi-threaded multi-core intel-architecture offers several cores on the same die. The processor cores feature an out-of-order super-scalar microarchitecture, with newly added 2-way hyper-threading. In addition to scalar units, it also has 4-wide SIMD units that support a wide range of SIMD instructions. Each core has a separate L1 cache for both instructions and data, and a unified L2 date cache. All cores share an L3 data cache. The processor also features an on-die memory controller that connects to DDR memory. The architecture of CPU is shown in Fig. 3.

Because of the low-cost of commercial grade CPUs, they are widely used as main compute units of large-scale supercomputers. However, for the same reason, CPUs cannot achieve a performance balance between general purpose such as human–computer interaction and specific purpose applications such as scientific computing. As a result, more special purpose devices are required for massively parallel scientific computing.

4.2 Many-core GPU

GPUs, also known as graphics cards having a long history, are originally designed by NVIDIA in 1999 for photorealistic 3D rendering in computer games. Because basic numerical linear algebra operations play crucial roles in real time 3D computer graphics, GPUs are designed for this set of operations. Because GPUs offer higher peak performance and bandwidth, numerical linear algebra applications can deliver much higher performance than merely using multi-core CPUs. In year 2007, the birth of NVIDIA's CUDA programming model made GPU programming much easier than before.

The NVIDIA GPU is composed of an array of multiprocessors, called streaming multiprocessor (SM). Each

Fig. 1 Code in Caffe of CPU version

```
template <typename Dtype>
void ConvolutionLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {
  const Dtype* weight = this->blobs_[0]->cpu_data();
  for (int i = 0; i < bottom.size(); ++i) {
    const Dtype* bottom_data = bottom[i]->cpu_data();
    Dtype* top_data = top[i]->mutable_cpu_data();
    for (int n = 0; n < this->num_; ++n) {
      this->forward_cpu_gemm(bottom_data + n * this->bottom_dim_, weight,
          top_data + n * this->top_dim_);
      if (this->bias_term_) {
        const Dtype* bias = this->blobs_[1]->cpu_data();
        this->forward_cpu_bias(top_data + n * this->top_dim_, bias);
      }
    }
  }
}
```

Fig. 2 Code in Caffe of GPU version

```
template <typename Dtype>
void ConvolutionLayer<Dtype>::Forward_gpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {
  const Dtype* weight = this->blobs_[0]->gpu_data();
  for (int i = 0; i < bottom.size(); ++i) {
    const Dtype* bottom_data = bottom[i]->gpu_data();
    Dtype* top_data = top[i]->mutable_gpu_data();
    for (int n = 0; n < this->num_; ++n) {
      this->forward_gpu_gemm(bottom_data + n * this->bottom_dim_, weight,
          top_data + n * this->top_dim_);
      if (this->bias_term_) {
        const Dtype* bias = this->blobs_[1]->gpu_data();
        this->forward_gpu_bias(top_data + n * this->top_dim_, bias);
      }
    }
  }
}
```

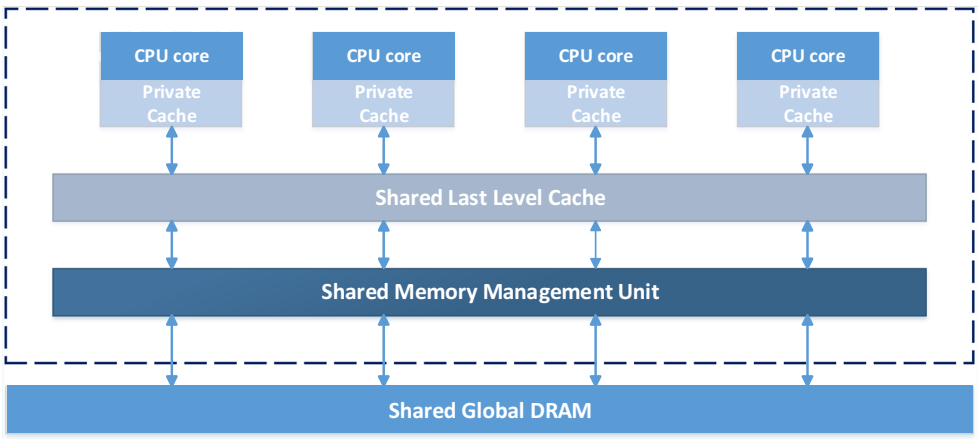


Fig. 3 CPU architecture

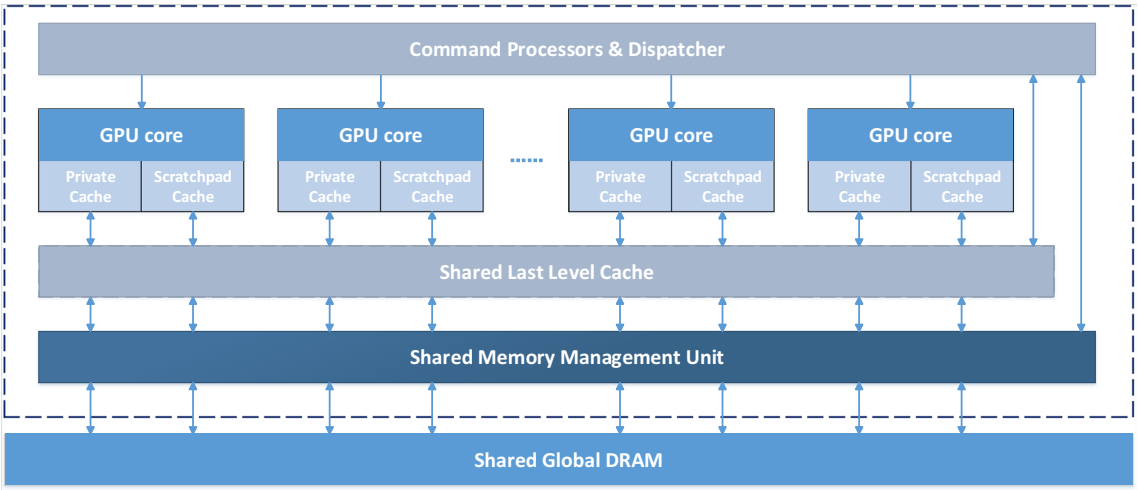


Fig. 4 GPU architecture

SM has 8 scalar processing units running in lockstep. The hardware SIMD structure is exposed to programmers through thread wraps. To hide memory latency, GPU provides hardware multi-threading support that allows hundreds of thread contexts to be active simultaneously. To alleviate memory bandwidth, the card includes various on-chip memories, such as local shared buffer. The GPU also has special functional units like the texture sampling unit, and math units for fast transcendental operations. The architecture of GPU is shown in Fig. 4.

Because GPU cores can execute massively parallel lightweight threads on SIMD units for higher aggregate throughput, applications with good data-level parallelism can be significantly accelerated by GPUs. As such, some modern supercomputers have already used GPUs as their accelerators. However, utilizing the power of GPUs requires rewriting programs in CUDA, OpenCL or other GPU-oriented programming models. This brings nontrivial software engineering overhead for applications with a lot of legacy code.

Furthermore, because GPUs have their own memory and GPU-controlled device memory. The data transfer may offset gained performance improvements from GPUs. Also, programming data transfer makes software more complicated and less easy to maintain.

4.3 Platform optimization

Traditionally, CPU programmers have heavily relied on increasing clock frequencies to improve performance and have not optimized their applications to thread-level parallelism (TLP) and data-level parallelism (DLP). However, CPUs are evolving to incorporate more cores with wider SIMD units, and it is critical for applications to be parallelized to exploit TLP and DLP. With multi-threading, memory access optimizations, and SIMD optimizations, the performance of both CPU and GPU will be increased.

In addition, CPUs heavily rely on caches to hide memory latency. Blocking is one technique which reduces the last-level cache (LLC) missed on CPUs. Program will obtain the best performance through underlying cache hierarchy.

For GPUs, the global inter-thread synchronization is very time-consuming, because it involves a kernel termination and new kernel call overhead from the host. It is necessary to minimize global synchronization. Another important optimization for GPUs was the use of the local shared buffer, which can reduce bandwidth consumption.

4.4 BLAS implementation

4.4.1 Matrix storage

The dense matrix A is assumed to be stored in row-major or column-major format in memory. For example, $m \times n$ dense matrix A can be stored as shown:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Its elements are arranged linearly in memory. Row-major format:

$$[a_{11} \quad a_{12} \quad \cdots \quad a_{1n} \quad a_{21} \quad \cdots \quad a_{m1} \quad \cdots \quad a_{mn}]$$

Column-major format:

$$[a_{11} \quad a_{21} \quad \cdots \quad a_{m1} \quad a_{12} \quad \cdots \quad a_{1n} \quad \cdots \quad a_{mn}]$$

4.4.2 Parallel analysis

Different subroutines are analyzed as follows:

DOT: Parallelism of DOT is of two levels: (1) multiplication operation is done in parallel (2) sum operation is done in two steps. For example, when calculating the inner product of vector $x = (x_0, x_1, x_2, \ldots, x_{n-1})^T$ and vector $y = (y_0, y_1, y_2, \ldots, y_{n-1})^T$. The product $x_i y_i, i = 0, 1, 2, \ldots, n-1$ can be obtained in parallel.

GEMV: Unlike DOT, GEMV's parallelism can be achieved by the independence of the elements in vector y when calculating $y \leftarrow \alpha Ax + \beta y$. $y_i \leftarrow \alpha A_i x_i + \beta y_i$ can be obtained by an independent thread, where A_i is the ith row of A [24].

GEMM: Similar to GEMV, GEMM's parallelism can be achieved by the independence of the elements of matrix C when calculating $C \leftarrow \alpha AB + \beta C$. $C_{ij} \leftarrow \alpha A_i B_j + \beta c_{ij}$ can be obtained by an independent thread, where A_i is the ith row of A and B_j is the jth column of B [26, 27].

TRSV: The calculation of the latter element depends on the result of the previous element, which is different from the previous three cases. When T is lower triangular case, the result x_i depends on $x_0, x_1, \ldots, x_{i-1}$. TRSV can take advantage of parallel hardware so $T_{i0}x_0 + T_{i1}x_1 + \cdots + T_{i,i-1}x_{i-1}$ can be calculated as the same way as DOT [28].

TRSM: Unlike TRSV, each element of the same row of the solution matrix is independent of each other. $B_{i0}, B_{i1}, \ldots, B_{ik}$ can be obtained in parallel after $B_{i-1,0}, B_{i-1,1}, \ldots, B_{i-1,k}$ is obtained, where we can achieve the parallelism of TRSM [23].

In the realization of same specific process, there are many parts which can be optimized, for example, in the realization of GEMV, GEMM, TRSV, and TRSM, the matrix can be calculated by using the cache, which can then improve the efficiency of the program.

4.4.3 Implementation optimization

The Level 1 BLAS allows efficient implementation on SIMD units. There is a limitation to the possible use of register allocations to improve the ratio of floating-point operations to data movement. The simplest and most central examples are the computation of a matrix–vector product. This can be coded as a sequence of n SAXPY operations ($y \leftarrow x + y$), but this obscures the fact that the result vector y could have been held in the register.

On SIMD units, one of the aims of such implementations is to keep the vector lengths as long as possible, and most algorithms compute one vector (row or column) at a time. In addition, on SIMD units, performance is increased by reusing the results of the registers which do not store the vector back into memory.

Unfortunately, this approach does not well suit for computers with a hierarchy of memory and parallel processor. Level 2 BLAS does not have a ratio of floating point to data movement that was high enough to make efficient reuse of data that resided in cache or local memory. It is often preferable to partition the matrix or matrices into blocks and to perform the computation by matrix–matrix operations on the blocks. By organizing the computation in this fashion, we provide for full reuse of data while the block is held in the cache or local memory. This approach avoids excessive movement of data to and from memory. In fact, it is often possible to obtain $O(n^3)$ floating-point operations while creating only $O(n^2)$ data movement. Parallelism can be exploited in two ways: (1) operations on distinct block may be performed in parallel; and (2) within the operations on each block, scalar or vector operations may be performed in parallel.

5 Comparison

We evaluate the performance of different routines in BLAS on the CPU and GPU and analyze the results in this section.

5.1 Methodologies

We evaluate the performance of different routines in BLAS on different platforms listed in Tables 1 and 2.

Xeon E5-2620v3 uses Haswell core, and supports two CPUs simultaneously, while Core i7 7700 and Core i5 7500 uses Kaby Lake core. Tesla K40c is Kepler architecture, while GTX1080Ti and GTX1070 is the Pascal architecture. At the same time, K40c is a professional computing card and can optimize certain double-precision calculation. The GTX1080Ti and GTX1070 is the game graphics and can optimize single-precision calculation. Both platforms run on the Ubuntu 16.04 operating system while with the latest Intel MKL 2017 and Nvidia CUDA 8.0 installed.

Since we are interested in comparing the CPU and the GPU architectures at the chip level to see if any specific architecture features are responsible for the performance difference, we do not include the data transfer time for GPU measurements.

We choose **DOT**, **GEMV**, **GEMM**, **TRSV**, and **TRSM** for comparison.

5.2 Performance evaluation

We evaluate Level-1, Level-2 and Level-3 BLAS on different platforms. The matrix is generated in a random way.

Table 1 Hardware configurations of CPUs

Name	Cores	Threads	Frequency (MHz)	L3 Cache (MB)
Intel Xeon E5-2620 v2 × 2	2 × 6	2 × 12	2100	2 × 15
Intel Core i7 7700	4	8	3600	8
Intel Core i5 7700	4	4	3400	6

Table 2 Hardware configurations of NVIDIA GPUs

Name	Compute capability	MP	Cores/MP	Frequency (MHz)	Global memory (GB)
Tesla K40c	3.5	15	192	745	12
GTX1080Ti	6.1	28	128	1582	11
GTX1070	6.1	15	128	1747	8

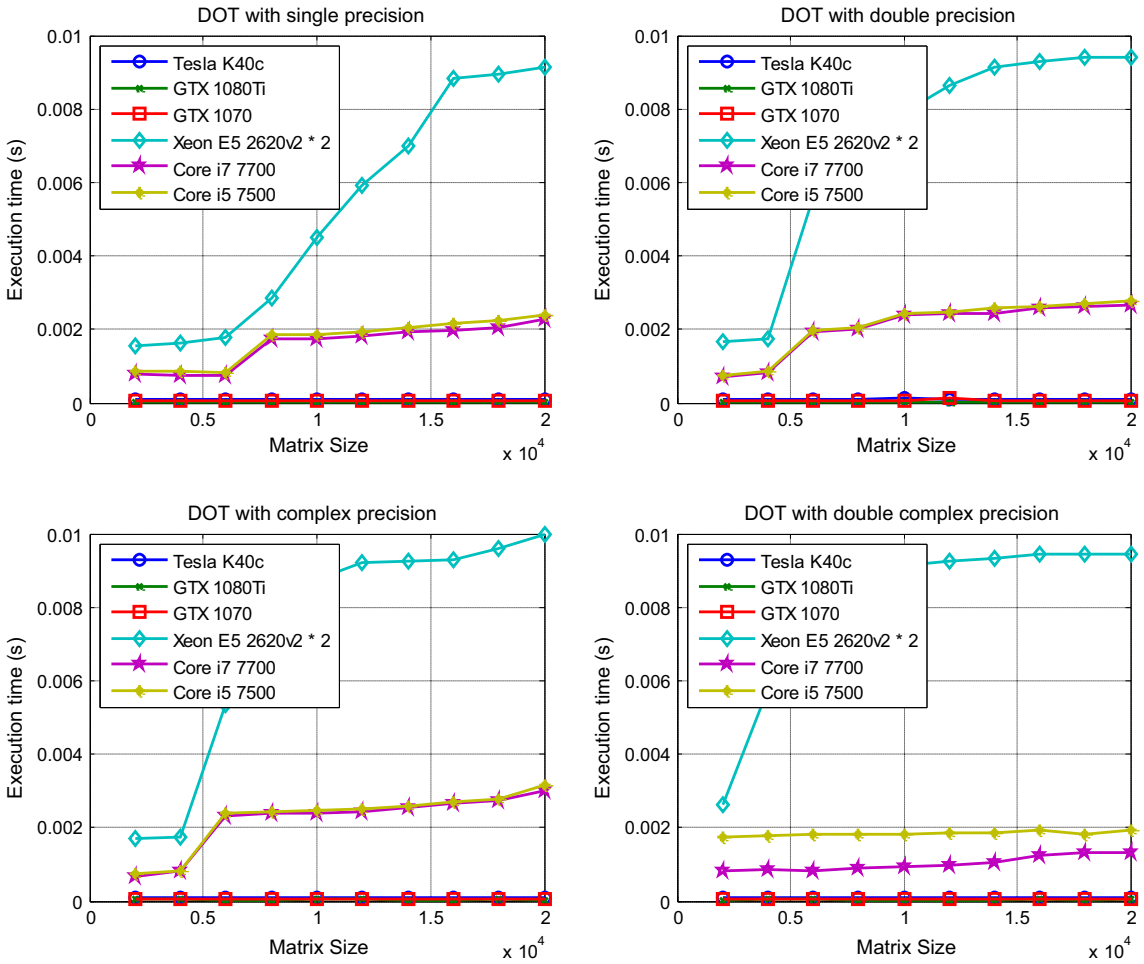


Fig. 5 Time consumption of DOT

We test three different randomly generated matrices and take their average. For CPU-calculated measurements, the computing duration is often disturbed by other calculation tasks, so we test five times and choose the smallest value as the time measurement.

Experimental programs are introduced:

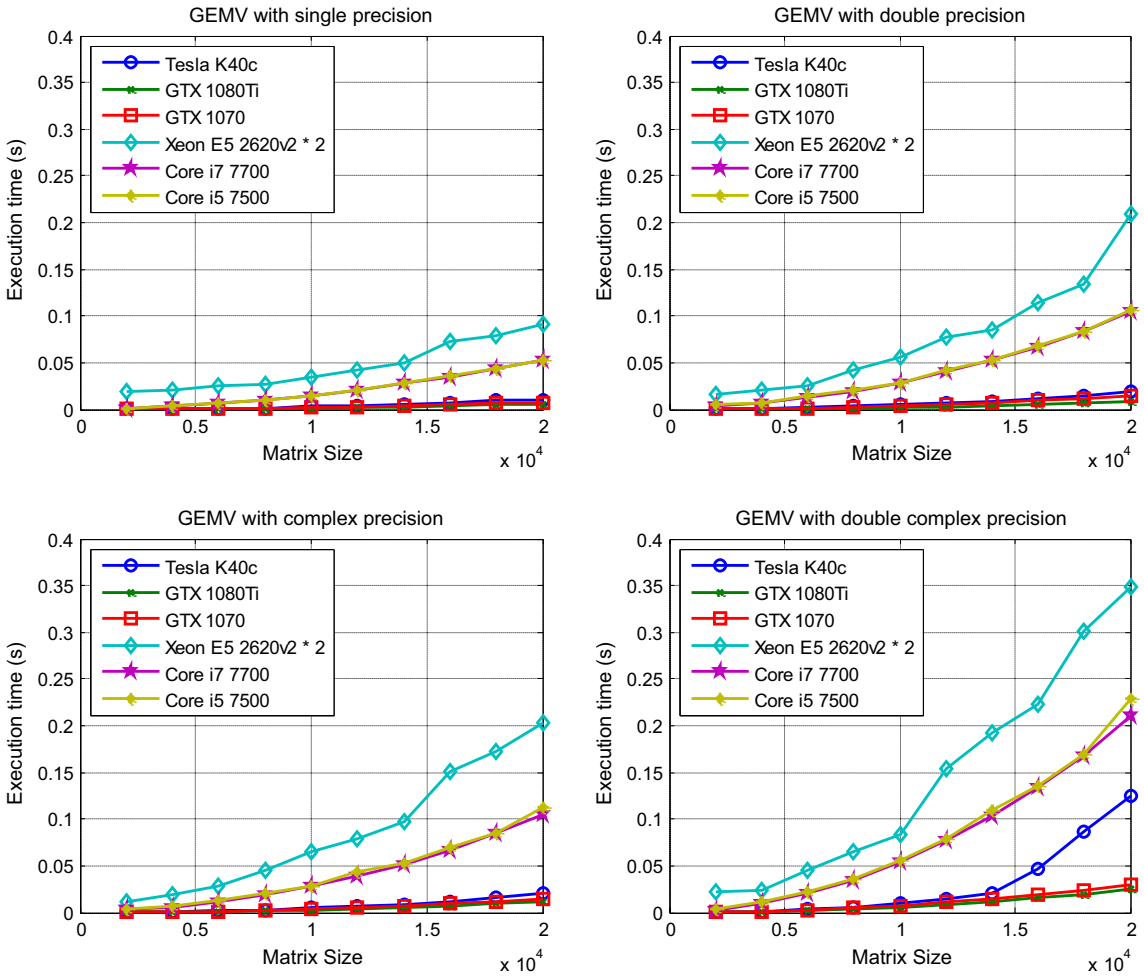


Fig. 6 Time consumption of GEMV

DOT: We calculate $x^T y$ for different precision and different sizes of vectors x and y, respectively, on the GPU and CPU.

GEMV: We calculate Ax for matrix A with the size $m \times n$ and the vector x of length n for $m = n$ for different precision on the GPU and CPU by varying m.

GEMM: We calculate AB on GPU and CPU, respectively, where A is $m \times n$ matrix and B is $n \times k$ matrix and we vary m, n and k simultaneously.

TRSV: We calculate $A^{-1}x$ on GPU and CPU, respectively, and A is an upper triangular matrix with the size $n \times n$ and the size of vector x is n.

TRSM: We calculate $A^{-1}B$ on GPU and CPU when k is modifying, and A is $n \times n$ upper triangular matrix, B is $n \times k$ matrix.

For a complete comparison, we perform experiments for four precision.

5.3 Performance comparison

First of all, we will evaluate the results of DOT, and the experimental results are plotted in Fig. 5.

The performance of Core i7 7700 is better than that of Xeon E5 2620v2, but the performance of the three CPUs is weaker than that of GPU. In addition, as the vector size

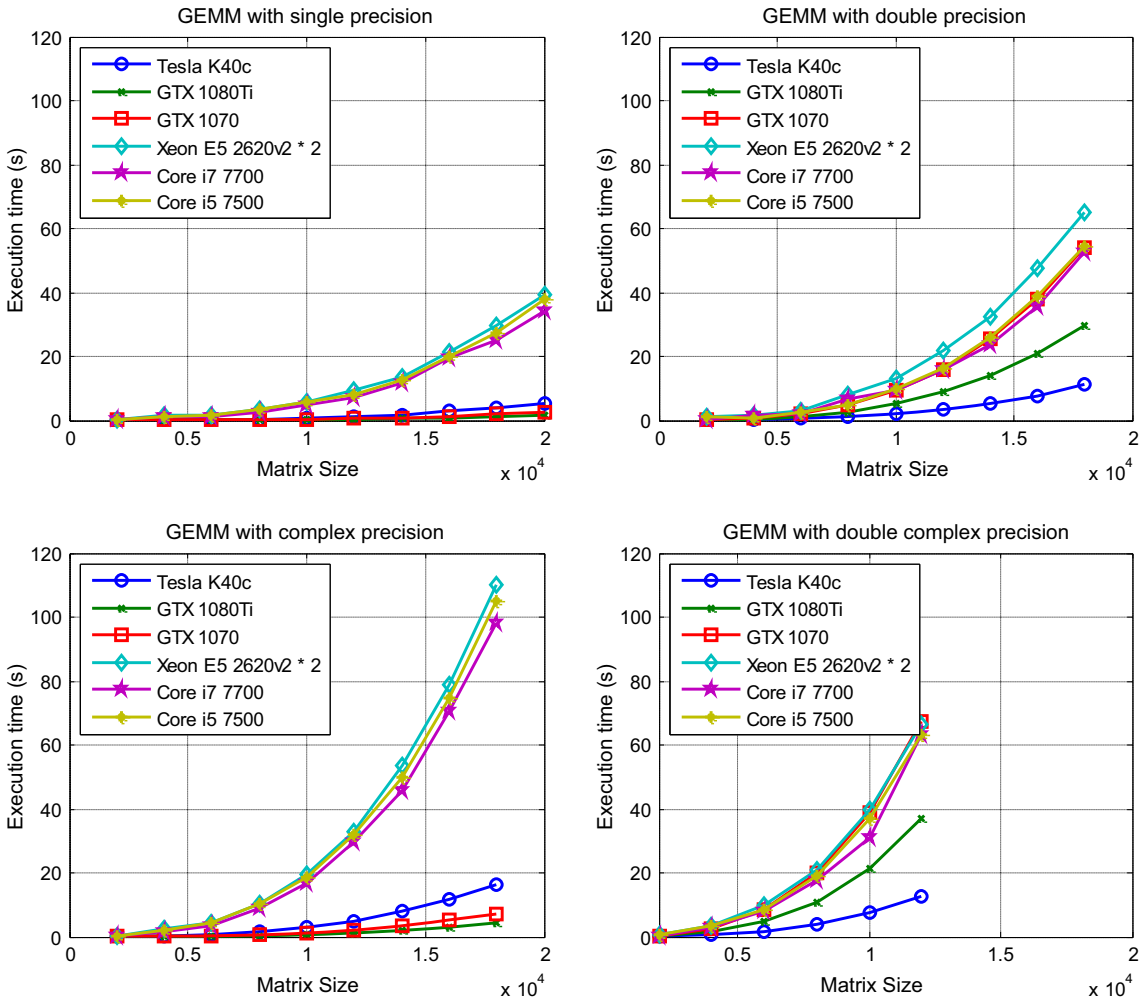


Fig. 7 Time consumption of GEMM

n increases, the computation time required by each processor does not change significantly.

The performance of GEMV is plotted in Fig. 6.

Since the amount of data and computations of GEMV is greater than that of DOT operation, the computation time required by each processor increases as the matrix increases and is not as stable as DOT operation. In this case, the advantages of GPU high throughput and high concurrency are gradually demonstrated, and the increase in time consumption is not as obvious as that of the CPU. The performance of GEMM is plotted in Fig. 7.

As the size of the matrix increases, the computational time required significantly increases. Meanwhile, the execution time of 1080Ti for the double-precision calculation is very slow, but is faster than that of CPU. In this case, GPU achieve a better performance. The performance of TRSV is plotted in Fig. 8. It is obvious that GPU is better than CPU. Due to the dependency of the results, TRSV concurrency is not as high as GEMV.

Then look at TRSM, and before the comparison of several functions is not the same way, we are fixed for the triangular matrix 10,000 and 20,000, the size of the solution matrix changes to see the calculation of time changes. The experiment result is plotted in Fig. 9. As TRSM implementation and TRSV different, GPU, only when the solution matrix is relatively large, GPU has the advantage, in the solution matrix is relatively small, GPU is not as efficient as CPU.

6 Discussion

In this section, we discuss the experimental results and propose how to choose processors for different subprograms.

DOT: On CPU, the computation time increases as the vector dimension increases. On GPU, the computation time is almost constant. Since the start-up cost of program takes

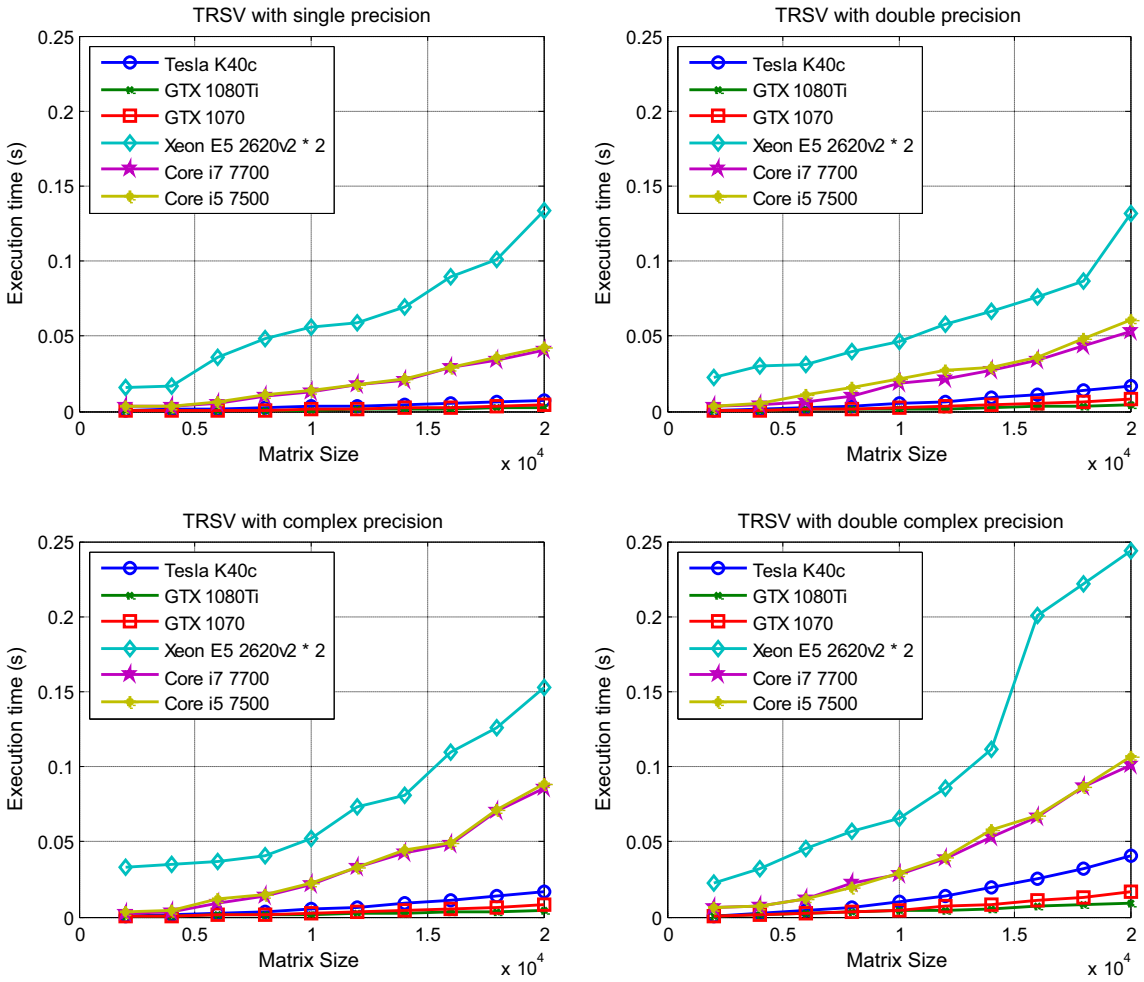


Fig. 8 Time consumption of TRSV

longer time than the computation time on GPU, the slight differences in computation time is negligible in this case.

GEMV and **GEMM**: On CPU and GPU, the computation time increases as the matrix dimension increases. The experiment on GPU is faster than that of CPU.

TRSV: Although there are some serial data dependencies, but on the GPU due to the sum operation can be carried out in parallel, the effect of all the GPU is still better than the CPU.

TRSM: This is the most complex subprogram. When k decreases, GPU takes longer time than CPU (even lower than TRSV). This is because the parallelism of the TRSM on GPU is based on the parallelism of the solution vector, but not on the sum. Therefore, the computation time is longer. When the dimension of the solution vector increases, the advantages of the GPU become significant again.

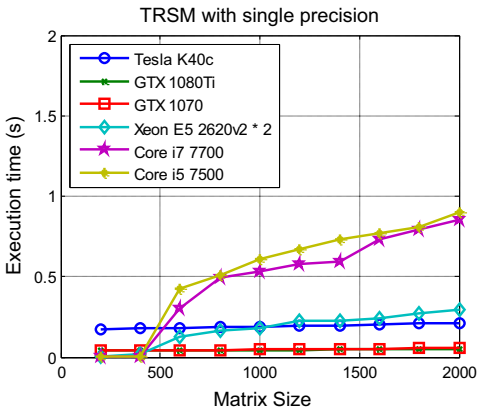
The final processor selection scheme is shown in Table 3.

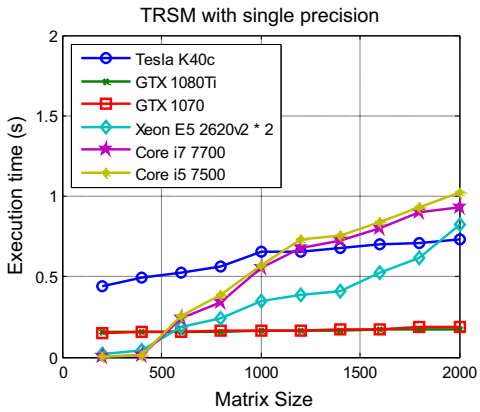
7 Conclusion

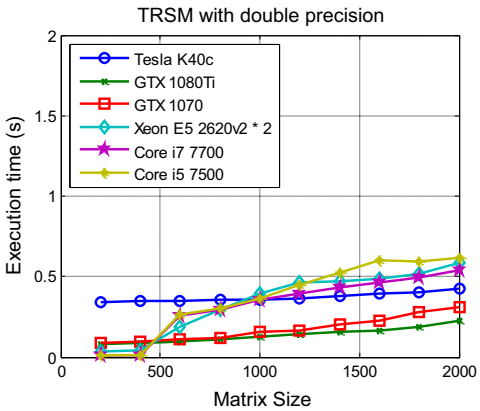
Effective selection of the platforms can speed up the calculation, which is important for the machine learning and scientific computing.

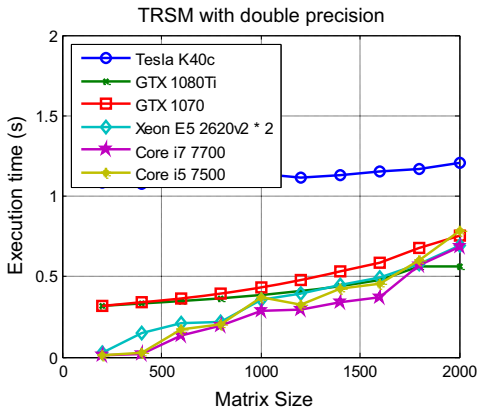
CPU has a high frequency characteristics, GPU has a high parallel characteristics. Therefore, the calculation of higher response requirements should be calculated by the CPU, the calculation of higher parallel requirements should be calculated by the GPU.

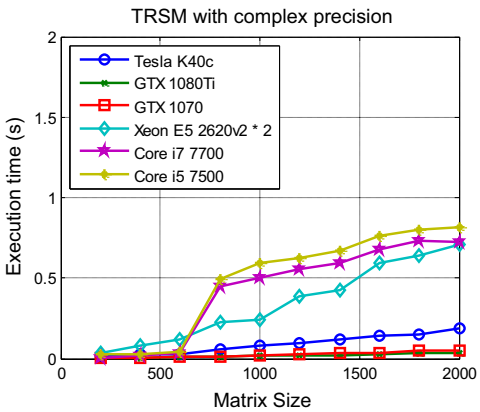
Matrix computation is the core component of many machine learning applications. For ease of use, BLAS is summarized and categorized into three levels. **GEMV** and **GEMM** are used in artificial neural networks, and **TRSV** and **TRSM** are used in least squares learning. Now some existing BLAS library is developed for different platforms. We compare the performance of MKL for multi-core CPU and CUBLAS for GPU.

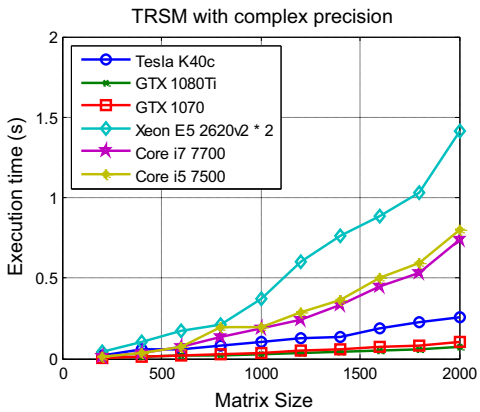

TRSM with single precision
Tesla K40c
GTX 1080Ti
GTX 1070
Xeon E5 2620v2 * 2
Core i7 7700
Core i5 7500
Execution time (s)
Matrix Size

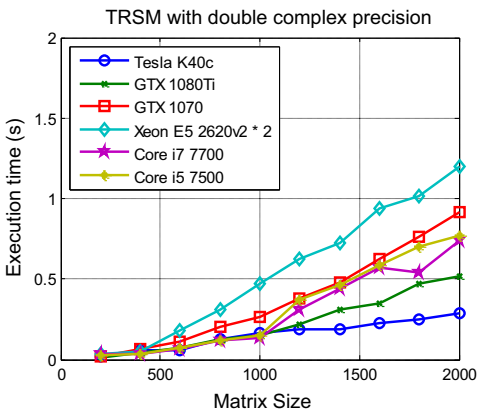

TRSM with single precision
Tesla K40c
GTX 1080Ti
GTX 1070
Xeon E5 2620v2 * 2
Core i7 7700
Core i5 7500
Execution time (s)
Matrix Size

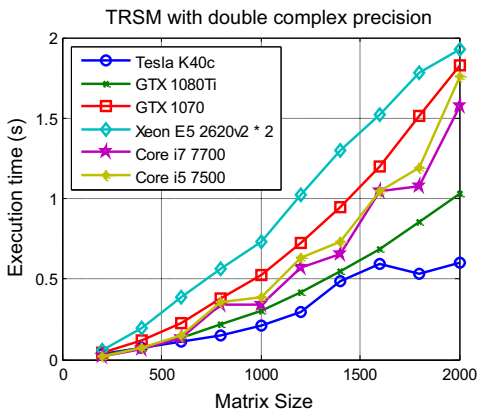

TRSM with double precision
Tesla K40c
GTX 1080Ti
GTX 1070
Xeon E5 2620v2 * 2
Core i7 7700
Core i5 7500
Execution time (s)
Matrix Size


TRSM with double precision
Tesla K40c
GTX 1080Ti
GTX 1070
Xeon E5 2620v2 * 2
Core i7 7700
Core i5 7500
Execution time (s)
Matrix Size


TRSM with complex precision
Tesla K40c
GTX 1080Ti
GTX 1070
Xeon E5 2620v2 * 2
Core i7 7700
Core i5 7500
Execution time (s)
Matrix Size


TRSM with complex precision
Tesla K40c
GTX 1080Ti
GTX 1070
Xeon E5 2620v2 * 2
Core i7 7700
Core i5 7500
Execution time (s)
Matrix Size


TRSM with double complex precision
Tesla K40c
GTX 1080Ti
GTX 1070
Xeon E5 2620v2 * 2
Core i7 7700
Core i5 7500
Execution time (s)
Matrix Size


TRSM with double complex precision
Tesla K40c
GTX 1080Ti
GTX 1070
Xeon E5 2620v2 * 2
Core i7 7700
Core i5 7500
Execution time (s)
Matrix Size

◀**Fig. 9** Time consumption of TRSM, we are fixed for the triangular matrix 10,000 and 20,000, the size of the solution matrix changes to see the calculation of time changes

Table 3 Processor selection scheme

Routines	Condition	Selection
DOT	–	GPU
GEMV	–	GPU
GEMM	–	GPU
TRSV	–	GPU
TRSM	k is small	CPU
TRSM	k is large	GPU

When we are designing machine learning systems or other numerical computing applications, **TRSM** should be calculated by the CPU when the matrix is smaller, and the rest should be calculated by the CPU.

Acknowledgements This research was supported in part by NSFC under Grant Nos. 61572158 and 61602132, Shenzhen Science and Technology Program under Grant Nos. JSGG20150512145714247, JCYJ20160330163900579 and JCYJ20170413105929681. And manuscript is approved by all authors for publication. I would like to declare on behalf of my co-authors that the work described was original research that has not been published previously and not under consideration for publication elsewhere, in whole or in part. All the authors listed have approved the manuscript that is enclosed.

Compliance with ethical standards

Conflict of interest No conflict of interest exits in the submission of this manuscript.

References

1. Oh KS, Jung K (2004) GPU implementation of neural networks. Pattern Recogn 37(6):1311–1314
2. Baptista D, Morgado-Dias F (2013) A survey of artificial neural network training tools. Neural Comput Appl 23(3–4):609–615
3. Baptista D, Abreu S, Freitas F et al (2013) A survey of software and hardware use in artificial neural networks. Neural Comput Appl 23(3–4):591–599
4. Lee VW, Kim C, Chhugani J et al (2010) Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. Int Symp Comput Archit 38(3):451–460
5. Owens JD, Luebke D, Govindaraju NK et al (2007) A survey of general-purpose computation on graphics hardware. Comput Gr Forum 26(1):80–113
6. Brodtkorb AR, Hagen TR, Saetra ML et al (2013) Graphics processing unit (GPU) programming strategies and trends in GPU computing. J Parallel Distrib Comput 73(1):4–13
7. Lawson CL, Hanson RJ, Kincaid DR et al (1979) Basic linear algebra subprograms for fortran usage. ACM Trans Math Softw 5(3):308–323
8. AMD, AMD Core Math Library (ACML). http://developer.amd.com/acml
9. Wang E, Zhang Q, Shen B et al (2014) Intel math kernel library. High-Performance Computing on the Intel Xeon Phi. Springer International Publishing, Berlin, pp 167–188
10. Barrachina S, Castillo M, Igual FD et al (2008) Evaluation and tuning of the level 3 CUBLAS for graphics processors. In: IEEE international symposium on parallel and distributed processing, pp 1–8
11. Anderson E, Bai Z, Bischof C et al (1999) LAPACK users' guide. Society for Industrial and Applied Mathematics, Philadelphia, PA
12. Moler C (2000) Matlab incorporates LAPACK. Increasing the speed and capabilities of matrix computation, MATLAB News and NotesCWinter
13. Walt S, Colbert SC, Varoquaux G (2011) The NumPy array: a structure for efficient numerical computation. Comput Sci Eng 13(2):22–30
14. Huang Z, Ye Y, Li X et al (2017) Joint weighted nonnegative matrix factorization for mining attributed graphs. Pacific-Asia conference on knowledge discovery and data mining. Springer, Cham, pp 368–380
15. Zhang H, Ho JKL, Wu QMJ et al (2013) Multidimensional latent semantic analysis using term spatial information. IEEE Trans Cybern 43(6):1625–1640
16. Abadi M, Agarwal A, Barham P et al (2016) Tensorflow: large-scale machine learning on heterogeneous distributed systems
17. Jia Y, Shelhamer E, Donahue J et al (2014) Caffe: convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM international conference on Multimedia, pp 675–678
18. Zhang H, Li J, Ji Y et al (2017) Understanding subtitles by character-level sequence-to-sequence learning. IEEE Trans Industr Inf 13(2):616–624
19. Uzair M, Shafait F, Ghanem B et al (2015) Representation learning with deep extreme learning machines for efficient image set classification. Neural Comput Appl, pp 1–13
20. Zhang H, Cao X, Ho JKL et al (2017) Object-level video advertising: an optimization framework. IEEE Trans Industr Inf 13(2):520–531
21. Guo H, Tang R, Ye Y et al (2017) DeepFM: a factorization-machine based neural network for CTR prediction. In: The twenty-sixth international joint conference on artificial intelligence (IJCAI), pp 1725–1731
22. Dongarra J, DuCroz J, Hammarling S et al (1988) An extended set of FORTRAN basic linear algebra subprograms. ACM Trans Math Softw 14(1):1–17
23. Dongarra J, DuCroz J, Hammarling S et al (1990) A set of level 3 basic linear algebra subprograms. ACM Trans Math Softw 16(1):1–17
24. Mukunoki D, Imamura T, Takahashi D (2015) Fast implementation of general matrix–vector multiplication (GEMV) on Kepler GPUs. In: 23rd Euromicro international conference on parallel, distributed and network-based processing (PDP), IEEE,, pp 642–650
25. Danihelka I, Wayne G, Uria B et al (2016) Associative long short-term memory. arXiv preprint arXiv:1602.03032
26. Nath R, Tomov S, Dongarra J (2010) An improved MAGMA GEMM for Fermi graphics processing units. Int J High Perform Comput Appl 24(4):511–515
27. Nakasato N (2011) A fast GEMM implementation on the Cypress GPU. ACM SIGMETRICS Perform Eval Rev 38(4):50–55
28. Romine CH, Ortega JM (1988) Parallel solution of triangular systems of equations. Parallel Comput 6(1):109–114