Auto Tuning Method for Deciding Block Size Parameters in Dynamically Load-balanced BLAS

Yuta SAWA * and Reiji SUDA

Graduate School of Information Science and Technology, University of Tokyo
{y_sawa,reiji}@is.s.u-tokyo.ac.jp

Abstract. High-performance routines of BLAS (Basic Linear Algebra Subprograms) are constantly required in the field of numerical calculations. We have implemented DL-BLAS (Dynamically Load-balanced BLAS) to enhance the performance of BLAS when other tasks use CPU resources of multi-core CPU architectures. DL-BLAS tiles matrices into submatrices to make subtasks and dynamically assigns tasks to CPU cores. We found that the dimensions of submatrices used in DL-BLAS affect the performance. To attain high-performance we have to solve an optimization problem where variables are the dimensions of the submatrices. The search space of the optimization problem is so vast that exhaustive search is unrealistic. We propose an auto tuning search algorithm which consists of Diagonal Search and Reductive Search. Our auto tuning algorithm provides semi-optimal parameters in realistic computing time. Using our algorithm, we got parameters which gave us the best performance in most of cases. As a result, DL-BLAS reached higher performance than ATLAS and GotoBLAS in many performance evaluation tests.

Key words: DL-BLAS, Diagonal Search, Reductive Search

1 Introduction

High-performance matrix-matrix multiplication routines are constantly required in the field of numerical calculations. BLAS[1–5] provide *de-facto* standard interfaces of basic operations of vectors and matrices, and the interfaces of matrix-matrix multiplications in the BLAS are called Level 3 BLAS.

Because Level 3 BLAS routines consume the majority of computing time in many applications, there are many researches about high-performance Level 3 BLAS using parallelization techniques. The past studies assumed that the amount of machine resources was known before the calculation. Traditionally, parallelized BLAS routines have required using the machine resources exclusively to reach their highest performance.

These days, many multi-core personal computers appear, and we have many occasions to run some other applications concurrently with BLAS. Under such

* Now with Central Research Labolatory, Hitachi, Ltd. yuta.sawa.eh@hitachi.com

circumstances, the amount of machine resources available to BLAS changes dynamically, and it is difficult to know the exact amount of available resources before the calculations.

We implemented DL-BLAS (Dynamically Load-balanced BLAS) to get higher performance when there are some applications running concurrently with BLAS [6, 7]. In DL-BLAS, there are parameters which mean dimensions of submatrices, and the performance of DL-BLAS is affected by these parameters. So, a method to get optimal parameters for DL-BLAS is needed. Here, it is a problem that the search space is too vast to experiment with all the possible values of the parameters. Also, the relations between the performance and the parameters are not clear.

We propose an auto tuning algorithm that consists of Diagonal Search and Reductive Search to search parameters which provide high-performance DL-BLAS.

Diagonal Search is a heuristic algorithm which reduce the search space down to a one-dimensional space while the original search space of our problem is a two-dimensional space. Reductive Search is an algorithm which collects information about good submatrix sizes for various matrix sizes. We applied these algorithms to DL-BLAS parameters and experimental data, from which sub-optimal submatrix sizes for arbitrary matrix sizes can be calculated, was collected in less than half an hour, and the chosen parameters achieved the best performance in most of cases.

The rest of this paper is organized as follows. We introduce BLAS routines in the next section. In that section, we also explain the algorithm of DL-BLAS and parameters of them. In section 3, we describe the algorithms of Diagonal Search and Reductive Search. Section 4 presents the results of performance evaluation tests of the search algorithms and DL-BLAS.

2 Background

In this section, we describe BLAS and DL-BLAS (Dynamically Load-balanced BLAS). DL-BLAS is our previous work.

2.1 BLAS

BLAS are routines developed to separate basic linear algebra calculation routines from the other parts of the calculations. The original purpose of the separation was to improve reusability and readability of linear algebra calculation programs and to decrease the frequency of appearance of bugs in the codes [2]. During the last three decades, BLAS had been appreciated as efficient interfaces in addition to the original purpose because BLAS interfaces have much generality and availability. Many famous linear algebra libraries such as LAPACK (Linear Algebra PACKage) [8] and ARPACK (ARnordi PACKage) [9] call BLAS routines internally.

From 1970's, where the first BLAS appeared, so many studies about high-performance BLAS implementations have been there. For instance, both GotoBLAS [10–12] and ATLAS [13, 14] are well known high-performance BLAS routines.

GotoBLAS achieves more than 90% of theoretical peak performance on many CPU architectures, and is known as the fastest BLAS implementation. GotoBLAS uses assembly level tuning.

ATLAS is a tuning tool to provide high-performance BLAS. There are many candidates of BLAS implementations in ATLAS package and ATLAS automatically select the fastest one from the candidates by calculating with many sample matrices.

We propose parallel BLAS routines with dynamic load balancing features using those packages as building-blocks.

2.2 DL-BLAS

Dynamic load balancing is seldom used for parallel BLAS because dense matrix calculation times are easy to predict. In 1991, however, Dackland et al. [15] suggested dynamic load-balanced systems. They proposed to create twice as many tasks as CPU cores for the load-balancing. Our implementation of DL-BLAS is a more effective approach than Dackland et al.'s suggestion.

GEMM (GEneral Matrix-Matrix multiplication) routines are the most frequently used routines in Level 3 BLAS. In this paper, we use only double-precision real routine, known as DGEMM, to establish our routines. GEMM can be written as the form $C = \alpha AB + \beta C$, where $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$, $C \in \mathbb{R}^{m \times n}$ and $\alpha, \beta \in \mathbb{R}$.

In DL-BLAS, matrices A, B and C are split to submatrices as follows:

$$\begin{aligned} A &= \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1\kappa} \\ A_{21} & A_{22} & \cdots & A_{2\kappa} \\ \vdots & \vdots & \ddots & \vdots \\ A_{\mu 1} & A_{\mu 2} & \cdots & A_{\mu\kappa} \end{pmatrix}, \\ B &= \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1\nu} \\ B_{21} & B_{22} & \cdots & B_{2\nu} \\ \vdots & \vdots & \ddots & \vdots \\ B_{\kappa 1} & B_{\kappa 2} & \cdots & B_{\kappa\nu} \end{pmatrix}, \\ C &= \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1\nu} \\ C_{21} & C_{22} & \cdots & C_{2\nu} \\ \vdots & \vdots & \ddots & \vdots \\ C_{\mu 1} & C_{\mu 2} & \cdots & C_{\mu\nu} \end{pmatrix}, \end{aligned} \qquad (1)$$

where

$$A_{ij} \in \mathbb{R}^{m_b \times k_b} \qquad (1 \le i < \mu,\ 1 \le j \le \kappa),$$

$$\begin{aligned} B_{ij} &\in \mathbb{R}^{k_b \times n_b} & (1 \le i < \kappa,\, 1 \le j \le \nu), \\ C_{ij} &\in \mathbb{R}^{m_b \times n_b} & (1 \le i < \mu,\, 1 \le j \le \nu), \end{aligned} \tag{2}$$

and

$$\nu = \lceil n/n_b \rceil, \quad \mu = \lceil m/m_b \rceil, \quad \kappa = \lceil k/k_b \rceil. \tag{3}$$

Note that the dimensions of rightmost and bottommost submatrices, such as $A_{i\kappa}$ or $A_{\nu j}$, may be smaller than the dimension shown in (2). The user of DL-BLAS is required to choose n_b, m_b and k_b, and ν, μ, κ are determined from them.

GEMM calculation $C = \alpha AB + \beta C$ is converted as:

$$C_{ij} = \beta C_{ij} + \alpha \sum_{l=1}^{\kappa} A_{il} B_{lj} \qquad (1 \le i \le \mu,\, 1 \le j \le \nu). \tag{4}$$

We treat the calculation for each pair (i, j) shown in (4) as a task, as shown in Algorithm 2.1. Note that GEMM routine is called in Algorithm 2.1, named unit-task routine.

Algorithm 2.1 Unit-Task routine of BLAS Calculation

```
1: $C_{ij} = \beta C_{ij} + \alpha A_{i1} B_{1j}$ (using GEMM routine)
2: for $l$ in 2 to $\kappa$ do
3:     $C_{ij} = C_{il} + \alpha A_{il} B_{lj}$ (using GEMM routine)
4: end for
```

We used ATLAS as GEMM in the experiments, because ATLAS can be compiled on many CPU architecture machines, and it provides good performances.

DL-BLAS uses unit-task routines to parallelize the GEMM calculationss by assigning tasks to CPU cores dynamically. The whole algorithm of the GEMM of DL-BLAS is shown in Algorithm 2.2.

GEMM-based BLAS [16, 17] uses a similar approach. In GEMM-based BLAS, GEMM routines are called from other Level 3 BLAS routines to exploit high performance of GEMM routines in other Level 3 BLAS routines. In GEMM-based BLAS, other Level 3 BLAS routines reaches the performance close to that of GEMM routines. But dynamic load balancing is not employed in GEMM-based BLAS.

2.3 Motivation of the research

We show the relations between the dimensions of submatrices ($n_b = m_b$ and k_b) and the performance of DL-BLAS on Intel Core 2 Extreme QX9650 architecture. We calculated matrix multiplication of size $m = n = k = 1000$ with DL-BLAS. When we used the parameters $n_b = m_b = 159$ and $k_b = 165$, the performance of

Algorithm 2.2 GEMM of DL-BLAS

```
1: now = 0
2: tasknum = $\mu\nu$
3: do parallel
4:     lock()
5:     $t$ = now
6:     now = now + 1
7:     unlock()
8:     if $t \geq$ tasknum
9:         break
10:    end if
11:    $i$ = now/$\mu$
12:    $j$ = now $- i * \mu$
13:    $C_{ij} = \alpha C_{ij} + \sum_{l=1}^{\kappa} \beta A_{il} B_{lj}$ (use Unit-Task routine shown in Algorithm 2.1)
14: end do while true
```

DL-BLAS was about 21.5 GFLOPS which was minimum performance. On the other hand, using the parameters $n_b = m_b = 224$ and $k_b = 168$, the performance of DL-BLAS was about 37.0 GFLOPS which was maximum performance.

This difference of the performances is large. Also, the parameters which provide peak performance may not be the same if the CPU architecture and sizes of argument matrices are different.

The sizes of argument matrices n, m and k are given in runtime. We want to get good parameters n_b, m_b and k_b for given n, m and k without much cost.

3 Method

In this section, first we analyze the relations between performance and the numbers of tasks and processors.

After that, we show auto tuning algorithms of Diagonal Search and Reductive Search. These algorithms are executed once at the installation and collect information for the runtime parameter selection from experiments. Last, we show an algorithm of runtime parameter selection, which determines the parameters of DL-BLAS referring to the information collected by Diagonal Search and Reductive Research.

3.1 Parallel Efficiency

Even if we use dynamic load-balancing in DL-BLAS, we will not always have complete load balance. The degree of load imbalances depends on the numbers of cores available to DL-BLAS and tasks created by DL-BLAS. Specifically, if there are a smaller number of tasks, the load imbalance tends to be worse. From this point of view, a larger number of tasks seem better, but it is not the case. A larger number of tasks in DL-BLAS imply smaller submatrices, which would degrade the performance of each task. There is a trade-off between load balance and performance of submatrix calculations.

We propose to solve this trade-off in the following way. A lower bound of the number of tasks is chosen to keep the load imbalance in a certain degree, and choose the sizes of the submatrices to maximize the performance of submatrix calculations and to generate subtasks no less than the lower bound.

We have formulated the degree of load imbalances as the followings. We let the number of processors on a machine be $p \geq 1$, and the number of tasks be $t > 0$, where p and t are integers.

At first, we define $h(t, p)$ as the ratio of calculation time with a single processor to that with p processors for t tasks. Assume that it takes a unit time for any processor to process a task. Then a single processor takes t units of time to process t tasks. When t tasks are distributed among p processors, at least one processor must process $\lceil t/p \rceil$ tasks. Thus we have the following equation:

$$h(t, p) = t/\lceil t/p \rceil. \tag{5}$$

As the next step, we define $h'(t, p)$ as follows:

$$h'(t, p) = \frac{t/p}{\lceil t/p \rceil}. \tag{6}$$

Because $h'(t, p)$ is nearly equals to $h(t, p)/p$, $h'(t, p)$ is considered to express the ratio of performance between ideally parallelized case and actual case approximately. We call the value $h'(t, p)$ as parallel efficiency.

Using the equation (6) we prove the following theorem:

Theorem 1. *For every $1 \leq i \leq p$ and $s \geq t$, $h'(s, i) \geq 1 - \frac{p-1}{t+p-1}$.*

To prove this theorem, we use the following lemma:

Lemma 1. *For every p, $ph'(t, p)$ is maximum at $t \equiv 0$ (mod p) and local minimum at $t \equiv 1$ (mod p).*

Proof. Clearly, $ph'(t, p)$ takes its maximum value p at $t \equiv 0$ (mod p). In other cases, $\lceil t/p \rceil = \lceil (t + 1)/p \rceil$. So, following statement is true:

$$ph'(t + 1, p) = \frac{t + 1}{\lceil (t + 1)/p \rceil} = \frac{t + 1}{\lceil t/p \rceil} > \frac{t}{\lceil t/p \rceil} = ph'(t, p). \tag{7}$$

□

Using Lemma 1, we can prove the Theorem 1 as follows.

Proof. We have following inequality expression:

$$h'(t, p) = \frac{t/p}{\lceil t/p \rceil} \geq \frac{t/p}{\frac{(t-1)}{p} + 1} = \frac{t}{t + p - 1} = 1 - \frac{p - 1}{t + p - 1}. \tag{8}$$

Letting $r(t,p)$ be $\frac{p-1}{t+p-1}$, the following statements are true:

$$\begin{aligned} r(t,p) - r(s,p) &= \frac{p-1}{t+p+1} - \frac{p-1}{s+p+1} \\ &= \frac{(p-1)(s-t)}{(t+p+1)(s+p+1)} \geq 0, \end{aligned} \tag{9}$$

and

$$\begin{aligned} r(s,p) - r(s,i) &= \frac{(p-1)(s+i-1) - (i-1)(s+p-1)}{(s+p-1)(s+i-1)} \\ &= \frac{s(p-i)}{(s+p-1)(s+i-1)} \geq 0. \end{aligned} \tag{10}$$

Considering the equations (9) and (10), $r(t,p)$ is larger than $r(s,i)$. Thus $1 - r(t,p) \leq 1 - r(s,i)$, and the Theorem have been proven.

□

For example, when $s \geq 16$ and $i \leq 4$, $h'(s,i) \geq 16/19 \simeq 0.842$. It means that when we have less than or equals to 4 processors and we have more than or equals to 16 tasks, the parallel efficiency is greater than 84%.

So in the following subsections, we will discuss how to find the submatrix size parameters that gives DL-BLAS high performance under the restriction that 16 or more tasks are generated.

3.2 Diagonal Search

In the sections 3.2 and 3.3, we describe our algorithms to collect performance information at the installation of DL-BLAS. Exhaustive search requires too much time. Thus we propose a set of efficient approximate search algorithms, Diagonal Search and Reductive Search.

In the experiments in this paper, we have an assumption that there are 4 physical CPU cores in each CPU, and we create more than 16 tasks for each performance evaluation test. So, the parallel efficiency is greater than 84% in each performance evaluation test.

As first, we will show a heuristics algorithm to get the efficient parameters (n_b, k_b) for a given set of matrix sizes (m, n, k). This algorithm is called as Diagonal Search in this paper.

Diagonal Search takes 5 arguments, n, m, k, $s_{\min}$ and $s_{\max}$. The new variables $s_{\min}$ and $s_{\max}$ mean the range of search space of n_b and k_b. In other words, n_b and k_b are searched from the following range:

$$s_{\min} \leq n_b \leq s_{\max}, \quad s_{\min} \leq k_b \leq s_{\max}. \tag{11}$$

The algorithm of Diagonal Search is shown in Algorithm 3.1. In this algorithm, a function "benchmark-with" is called. This function calculates BLAS problem as a benchmark, and returns FLOPS value of the calculation. DL-BLAS

Algorithm 3.1 Diagonal Search

```
1: $(n, m, k, s_{\min}, s_{\max})$ = input
2: $v_{\max} = 0$, $i_{\max} = 0$
3: for $i$ in $s_{\min}$ to $s_{\max}$ do
4:     $v$ = benchmark-with$(n, m, k, n_b = i, k_b = i)$
5:     if $v > v_{\max}$ then
6:         $v = v_{\max}$, $i = i_{\max}$
7:     end if
8: end for
9: $n'_b = i_{\max}$, $k'_b = i_{\max}$
10: for $j$ in $s_{\min}$ to $s_{\max}$ do
11:     $v$ = benchmark-with$(n, m, k, n_b = i_{\max}, k_b = j)$
12:     if $v > v_{\max}$ then
13:         $v = v_{\max}$, $n'_b = i_{\max}$, $k'_b = j$
14:     end if
15:     $v$ = benchmark-with$(n, m, k, n_b = j, k_b = i_{\max})$
16:     if $v > v_{\max}$ then
17:         $v = v_{\max}$, $n'_b = j$, $k'_b = i_{\max}$
18:     end if
19: end for
20: return $(n'_b, k'_b)$
```

calculates BLAS problem in the function "benchmark-with". The first to third arguments of the function "benchmark-with" are matrix sizes of benchmark problem, and fourth and fifth arguments are block sizes used in DL-BLAS.

This algorithm need only to calculate $3(s_{\max} - s_{\min})$ benchmarks, though the exhaustive calculation needs $(s_{\max} - s_{\min})^2$ calculations. When the value $(s_{\max} - s_{\min})$ is larger than 100, the number of function call of "benchmark-with" in exhaustive search is more than 30 times than that in Diagonal Search.

3.3 Reductive Search

The range $(s_{\min}, s_{\max})$ must be decided to call Diagonal Search. Also, we have to select a set of matrix sizes (m, n, k) for Diagonal Search, because we cannot execute Diagonal Search for all possible combinations of m, n and k.

Reductive Search is given as Algorithm 3.2. We use $(s_{\min}, s_{\max}) = (150, 250)$ and $\gamma = 4$.

Reductive Search begins its experiments with calling Diagonal Search with $(n, m, k) = (1000, 1000, 1000)$ and $(s_{\min}, s_{\max}) = (150, 250)$. This choice comes from our observation that the performance of ATLAS is similar for DGEMM with $(n, m, k) = (1000, 1000, 1000)$ and larger matrices. The value $\gamma = 4$ (or $s_{\max} = 250$) is chosen so that there are at least 16 tasks. The optimum parameters (n_b, k_b) found by Diagonal Search are stored in lists l_n and l_k.

In the following steps, Reductive Search choose $(s_{\min}, s_{\max})$ as $(0.5n_b, 0.9n_b)$, and $n = m = k = \gamma s_{\max}$, which is reduced into 0.9 times the previous size or less.

Diagonal Search is called with those arguments, and the optimum parameters (n_b, k_b) are added to the list.

The iteration of Reductive Search terminates when the dynamic load balancing of "benchmark-with" function takes longer time than a serial execution by a single thread. Here, we assume there is no other tasks running concurrently with Reductive Search.

Algorithm 3.2 Reductive Search

```
1: $(s_{\min}, s_{\max})$ = input
2: $l_n$ = new list()
3: $l_k$ = new list()
4: repeat
5:    $m = \gamma s_{\max}, n = \gamma s_{\max}, k = \gamma s_{\max}$
6:    $(n_b, k_b)$ = Diagonal_Search$(m, n, k, s_{\min}, s_{\max})$
7:    $addFirst(l_n, n_b)$
8:    $addFirst(l_k, k_b)$
9:    $(s_{\min}, s_{\max}) = (0.5n_b, 0.9n_b)$
10: until benchmark-with$(m, n, k, n_b, k_b)$ > single-thread-benchmark-with$(m, n, k)$
11: return $(l_n, l_k)$
```

3.4 Parameter Selection

Algorithm 3.3 shows Parameter Selection, which determines the parameters (n_b, k_b) from the matrix size (n, m, k) whenever DL-BLAS is called. We have gotten list of parameters. When we calculate BLAS problems, we choose only one set of parameters (n_b, k_b) in the routines as fast as possible. The algorithm to choose the parameters is shown in Algorithm 3.3.

We used the value LIMIT = 16 to have more than 16 tasks, which let the parallel efficiency be greater than 84%.

Algorithm 3.3 Parameter Selection

```
1: for $i = 0$ to sizeof$(l_n) - 2$ do
2:    $n_b = l_n(i)$
3:    $k_b = l_k(i)$
4:    TaskNum = $\lceil n/n_b \rceil \lceil m/n_b \rceil$
5:    if TaskNum $\geq$ LIMIT then
6:       return $(n_b, k_b)$
7:    end if
8: end for
9: return $(l_n(\text{sizeof}(l_n) - 1), l_m(\text{sizeof}(l_n) - 1))$
```

There are less than 10 floating point operations in each loop calculation, and the length of the list was less than 20 in our experiments. When we calculate

BLAS problem $n = 10, m = 10, k = 10$, we need about 2000 of floating point operations. So, this calculation is thought not to be large overhead.

4 Experiments

4.1 Enviroinments

The list of the CPU architectures used in the following experiments is shown in Table 1.

CPU vendor	CPU	CPU Clock	OS
Intel	Core 2 Extreme QX9650	3 GHz	Fedora 8
AMD	Phenom 9600	2.3 GHz	Fedora 8
Intel	Core i7 365 (Hyper-Threading on)	3.2 GHz	Ubuntu
Intel	Core i7 365 (Hyper-Threading off)	3.2 GHz	Ubuntu
AMD	PhenomII X4 940	3 GHz	Fedora 8

Table 1. Used Machine Environments

All the CPU architectures shown in Table 1 have implementation of Goto-BLAS. ATLAS also can be compiled on all the CPU architectures above. The installation time of ATLAS was about 2-3 hours in all the CPU architectures shown above.

Intel Core i7 can be used with Hyper-Threading option, which provide simultaneous multi-threading mode. With this option, Intel Core i7 processor can handle 8 logical processors while only 4 processors are physically present.

The theoritical peak performance of each architecture shown above can be calculated by the following formula:

$$\text{theoretical-peak-performance} = 16\text{CPU-Clock}. \qquad (12)$$

4.2 Performance of Diagonal Search

In this section, we show the results of the Diagonal Search. In the following experiments, we calculated DGEMM problem with the matrix sizes $n = 1000$, $m = 1000$ and $k = 1000$ and all the parameters in the range $150 \leq n_b, k_b \leq 250$. We compare the results of the full search and Diagonal Search.

In all the experiments shown above, the results of exhaustive search and that of Diagonal Search are the same.

When we search parameters exhaustively, a few hours were needed, while Diagonal Search need less than 10 minutes in each computer. Also, the combinational algorithm of Diagonal Search and Reductive Search need less than 30 minutes in each computer. ATLAS requires more than an hour when we install ATLAS to the computers. So, we claim that the cost of Diagonal Search and Reductive Search can be allowed.

CPU Architecture	Exhaustive Search Performance	Diagonal Search
Core2 Extreme	37.0 GFLOPS	37.0 GFLOPS
Core i7 Hyper-Thread	34.4 GFLOPS	34.4 GFLOPS
Core i7	36.1 GFLOPS	36.1 GFLOPS
Phenom	17.1 GFLOPS	17.1 GFLOPS
Phenom II	34.8 GFLOPS	34.8 GFLOPS

Table 2. Comparison of Diagonal Search and Exhaustive Search

4.3 Analysis and Discussion of Diagonal Search

In this section, we analyze the reason why Diagonal Search found the same parameters as the exhaustive search. In an exhaustive search, we calculated a problem $m = n = k = 1000$ with the parameters $150 \leq n_b, k_b \leq 250$. The relations between the performance and the parameters are different in each CPU. We show results in two CPU architectures as examples. The two CPU architectures are Intel Core 2 Extreme and AMD Phenom. Figures 1 and 3 are results of the exhaustive search.

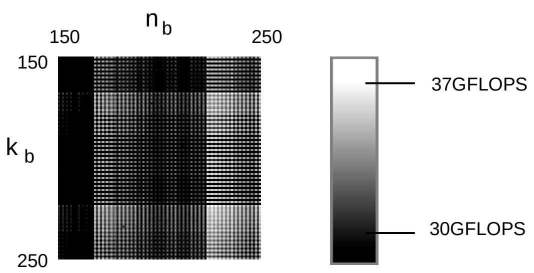

Fig. 1. Exhaustive Search Performance on Intel Core 2 Extreme

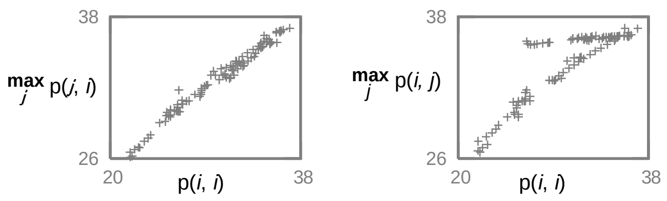

Fig. 2. Relations between $p(i,i)$, $\max_j p(i,j)$ and $\max_j p(j,i)$ on Intel Core 2 Extreme

We define $p(n_b, k_b)$ as $p(n_b, k_b) =$ benchmark-with$(1000, 1000, 1000, n_b, k_b)$, where "benchmark-with" is a function used in Algorithm 3.1. The first step of Diagonal Search calculates $p(i,i)$ for each $s_{\min} \leq i \leq s_{\max}$ in Line 3-8 of Algorithm 3.1. The next step calculates $p(i_{\max}, j)$ and $p(j, i_{\max})$ in Line 10-19.

So, if the value of $\max_j p(i,j)$ or $\max_j p(j,i)$ is large when $p(i,i)$ is large, Diagonal Search get high-performance. Figures 2 and 4 show the relations between $p(i,i)$, $\max_j p(i,j)$ and $\max_j p(j,i)$. Two graphs in Figure 2 and left graph

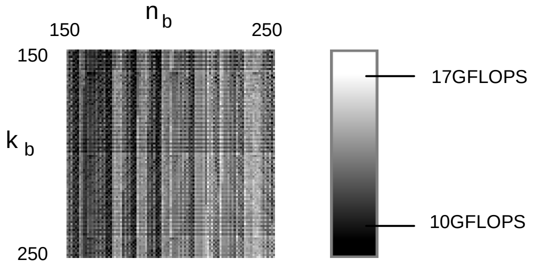

Fig. 3. Exhaustive Search Performance on AMD Phenom

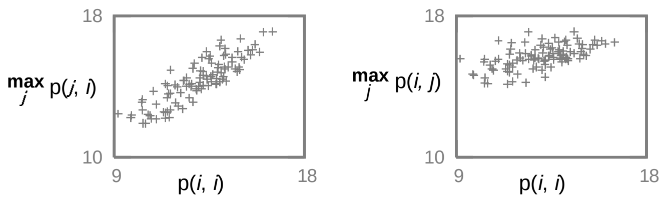

Fig. 4. Relations between $p(i,i)$, $\max_j p(i,j)$ and $\max_j p(j,i)$ on AMD Phenom

in Figure 4 show positive correlations. These positive correlations let Diagonal Search work well.

4.4 Performance Evaluation Tests

In this section, we show the results of the performance evaluation tests. In the tests, we calculate square matrix multiplication $1 <= m = n = k <= 2000$. We compare the performances of DL-BLAS, GotoBLAS and ATLAS. The parameters n_b and k_b in DL-BLAS are tuned by Diagonal Search and Reductive Search.

The tests are executed in two different circumstances, **no other task** and **busy loop running**. No other task means that there are not any tasks are running other than the test (benchmark) program, and BLAS routines can use machine resources exclusively. Busy loop running means that BLAS routines run sharing the machine resources with another process that executes an empty loop infinitely.

The results of the tests are shown in Figures 5-9.

We have gotten four remarkable observations from the graphs above.

At first, while the performance of GotoBLAS is fastest with no other tasks, that is low and unstable in busy loop running. So, if we run BLAS routines with other applications concurrently, using GotoBLAS is not a good choice.

Second, ATLAS is slower than DL-BLAS in with no other tasks on Intel Core 2 Extreme and AMD Phenom. On the other CPU architectures, the performance of ATLAS is almost the same to DL-BLAS.

Third, DL-BLAS is faster than ATLAS with busy loop running on the CPU architectures except Intel Core i7.

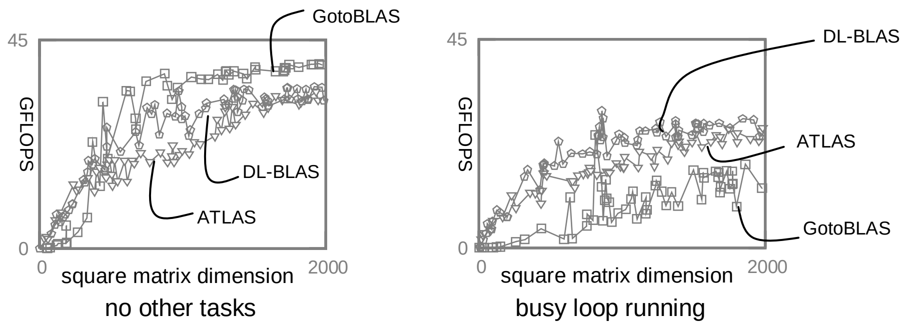

Fig. 5. Square matrix multiplication performance on Intel Core 2 Extreme

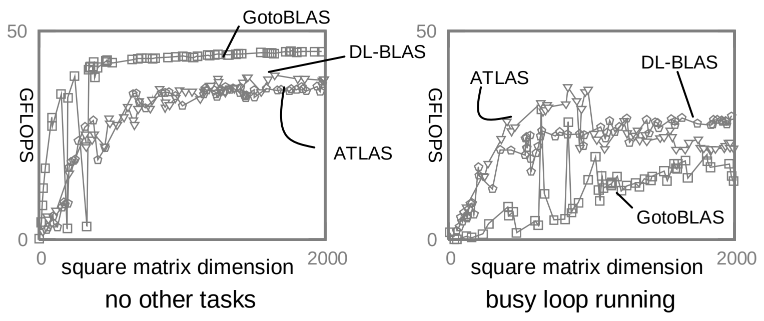

Fig. 6. Square matrix multiplication performance on Intel Core i7

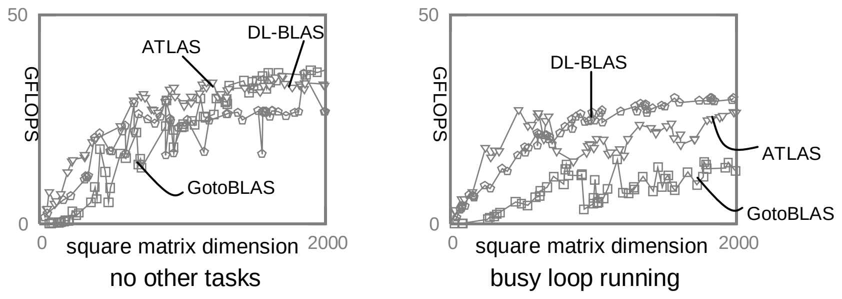

Fig. 7. Square matrix multiplication performance on Intel Core i7 with Hyper-Threading

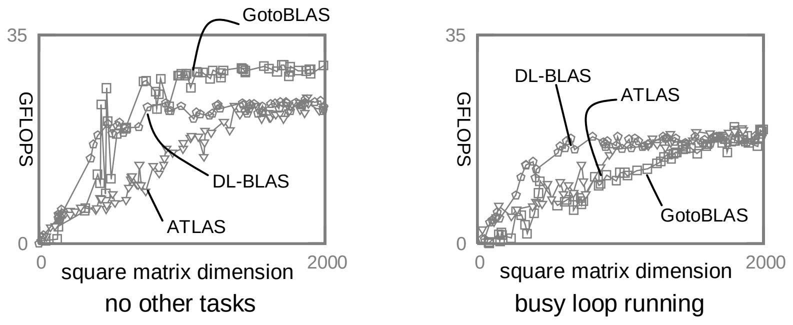

Fig. 8. Square matrix multiplication performance on AMD Phenom

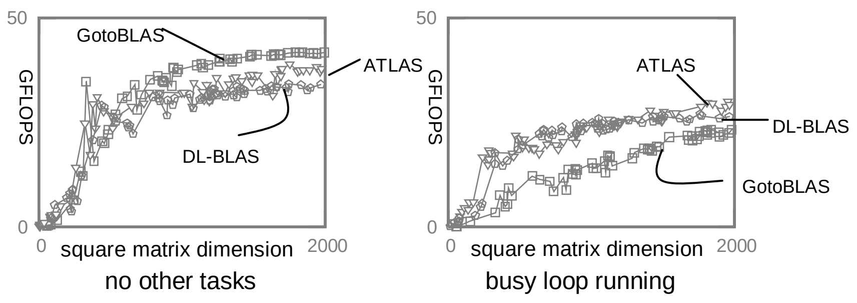

Fig. 9. Square matrix multiplication performance on AMD Phenom2

At last, ATLAS is faster than DL-BLAS with busy loop running on Intel Core i7 when the problem is $m = n = k < 1000$. ATLAS used only less than 4 CPU cores when $m = n = k < 1000$, and thus allowed execution of busy loop without degrading its own performance.

5 Conclusion

DL-BLAS is an implementation of high-performance BLAS routine for the environments where other tasks are running concurrently. DL-BLAS needs sophisticated method of selecting sizes of submatrices as parameters, which affect the performance.

In this paper, we proposed an auto tuning algorithm which consists of Diagonal Search and Reductive Search. The two algorithms create several candidates for the parameters of DL-BLAS, and DL-BLAS select one from the candidates. The calculation time of the algorithms is shorter than the installation time of ATLAS.

Using the parameters gotten by the auto tuning algorithm, we have evaluated the performance of DL-BLAS. As performance evaluation tests, we solved square matrix multiplication problems. On many CPUs the performance of DL-BLAS was at least in the same range as that of ATLAS when no other tasks are running concurrently. In the experiments with busy loop running, the performance of DL-BLAS was better than GotoBLAS.

In this research, our trial problems are square matrix multiplications ($n = m = k$) and we calculate DGEMM routine. As future works, we will evaluate DL-BLAS in different trial problems, for example, rectangle (not square) matrix multiplications, single precision calculations, complex calculations, symmetric matrix multiplication routines etc.

Acknowledgement

This research was partially supported by New IT Infrastructure for the information-explosion Era of MEXT Grant-in-Aid for Scientific Research on Priority Areas, and also by 4th IJARC Blue Sky program of Microsoft Corporation.

References

1. Basic Linear Algebra Subprograms (BLAS) Technical Forum Standard: BLAS (Basic Linear Algebra Subprograms). http://www.netlib.org/blas/ (January 26 2009)
2. Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: Basic Linear Algebra Subprograms for Fortran Usage. ACM Transactions on Mathmatical Software **5** (1979) 308–323
3. Dongarra, J.J., Croz, J.D., Hammarling, S., Hanson, R.J.: An extended set of FORTRAN Basic Linear Algebra Subprograms. ACM Transactions on Mathematical Software **14**(1) (March 1988) 1–17
4. Dongarra, J.J., Croz, J.D., Hammarling, S., Duff, I.: A set of level 3 Basic Linear Algebra Subprograms. ACM Transactions on Mathematical Software **16**(1) (March 1990) 1–17
5. Blackford, L.S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., Whaley, R.C.: An updated set of Basic Linear Algebra Subprograms (BLAS). ACM Transactions on Mathematical Software **28**(2) (June 2002) 135–151
6. Sawa, Y., Suda, R.: BLAS parallelization for binary distrubution for multi-core processors (In Japanese). JSIAM Annualy meeting 2008 425–426
7. Sawa, Y.: Adaptive Parallelization of BLAS for Multi-Tasking Environment. Master Thesis, Department of Computer Science, Graduate School of Information Science and Technology, University of Tokyo (February 2009)
8. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J.D., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide. Third edn. Society for Industrial and Applied Mathematics, Philadelphia, PA (1999)
9. Lehoucq, R., Maschhoff, K., Sorensen, D., Yang, C.: ARPACK - Arnoldi Package. http://www.caam.rice.edu/software/ARPACK/ (January 26 2009)
10. Goto, K.: GotoBLAS. http://www.tacc.utexas.edu/resources/software/ (January 26 2009)
11. Goto, K., van de Geijn, R.: High-performance implementation of the level-3 blas. ACM Trans. Math. Softw. **35**(1) (2008) 1–14
12. Goto, K., van de Geijn, R.: Anatomy of high-performance matrix multiplication. ACM Transactions on Mathematical Software **34**(3) (2008)
13. Whaley, R.C., Petitet, A.: Automatically Tuned Linear Algebra Software (ATLAS). http://math-atlas.sourceforge.net/ (January 26 2009)
14. Whaley, R.C., Petitet, A.: Minimizing development and maintenance costs in supporting persistently optimized BLAS. Software: Practice and Experience **35**(2) (February 2005) 101–121
15. Dackland, K., Elmroth, E., Kagstrom, B., Loan, C.V.: Design and evaluation of parallel block algorithms: Lu factorization on an ibm 3090 vf/600j, PPSC (1991) 3–10
16. Kagstrom, B., Ling, P., van Loan, C.: GEMM-Basd Level3 BLAS. ACM Transactions on Mathematical Software (TOMS) (1998) 288–302
17. Kagstrom, B., Ling, P., van Loan, C.: GEMM Based BLAS. http://www.netlib.org/blas/gemm_based/ (January 26 2009)