Vol.2012-HPC-133 No.1
2012/3/26

A Three-Step Performance Automatic Tuning Strategy using Statistical Model for OpenCL Implementation of Krylov Subspace Methods

Cong LI[†1] and Reiji SUDA[†1,†2]

In this work, we propose a three-step performance automatic tuning strategy that will help the developers to write applications with self-adaptive performance. We are using OpenCL and Krylov Subspace Methods as our programming language and test problems respectively. By applying machine learning techniques, we build our statistical performance models of a specific runtime environment through data collected from experiments executed automatically. These models are used for searching computational performance related optimal tuning parameters. Finally we further optimize choices of these parameters using the iterative feature of Krylov Subspace Method. The choices of tuning parameters and statistical modeling strategies are crucial to the performance of our tuning strategy. In the paper, we evaluated the statistical models that we build for autotuning. The results show that the accuracy of SVM classification model can be as high as 100% and 94.32% for training dataset and test dataset of SpMV and as high as 100% and 96.21% for training dataset and test dataset of SAXPY.

1. Introduction

Solving linear systems is a job that locates at the central part of scientific computations. For instance, fluid simulations and circuit simulations usually involve solution of linear systems. Moreover, the matrices of linear systems usually appear with sparse structures. For example, the linear systems obtained from the discretization of continuous fields with Finite Difference Method or Finite Element Method are sparse matrices.

There has been many data structures and algorithms that are specially cut for solving sparse linear systems. But the efficiency of these data structures and algorithms are heavily depending on the sparsity of matrices arising from sparse linear systems.

Krylov Subspace Methods belong to the category of iterative methods that are popularly applied to solve sparse linear systems. This category of numerical methods employ many calculations manipulating vectors and matrices, for instances CG *(Conjugate Gradient)* method, GMRES *(Generalized Minimum Residual)* method and so on. As a result, the data structures for storing these vectors and matrices on memory will significantly affect the computing performance of Krylov Subspace Methods.

Besides the influence of the layout of matrices and vectors on memory, the characteristics of computing environment, for instances hardware environment and software environment and so forth, will also significantly affect the computing performance measured by *Flops* `or` *Wall-time*. Moreover, after the concept of GPGPU*(General Purpose GPU)* being introduced into the field of HPC*(High Performance Computing)*, the computing environments have become more and more complex and difficult to harness. OpenCL*(Open Computing Language)* is an open, royalty-free standard for cross-platform programming of modern processors, for instance CPU, GPGPU and even CELL processors. But one problem of OpenCL is that it only provides source code platform-portability not performance-portability, which means OpenCL source code can be compiled and run on any type of OpenCL supported platforms and processors, but the performance can be optimized if we could tune the original source code for the specific type of platforms and processors.

Tuning source code is an exhausting job even to a skilled programmer. Especially when the complexity of computer hardware and software is increasing as today's, programmers need to consider more tuning targets than before, such as performance, power consumption and so on.

Software Automatic Tuning or Autotuning is an state-of-art technique which is one of the most promising solutions to the performance portability problems mentioned above. Reiji Suda, et al.[1)] explained and discussed the concepts, topics and issues of software automatic tuning. They argue that the task of the *tuning mechanism* is to control the adaptabilities of the target software so to attain the optimal performance under the given conditions.

†1 Department of Computer Science, Graduate School of Information Science and Technology at University of Tokyo
†2 CREST, JST

In this work, we propose a software automatic tuning strategy for Krylov Subspace Methods. Our strategy firstly chooses one storage format from a list of matrix storage formats for a specific hardware and software environment using statistical models obtained by applying machine learning techniques, and then employs exhaustive search during runtime to select appropriate values of tuning parameters to further improve computing performance.

The rest of the paper is organized as follows: Section 2 introduces the background information of OpenCL programming model and machine learning, and then we analyse the structure Krylov subspace method in terms of matrix computation involved. Finally we discuss the related works in this section. Section 3 details our three-step tuning strategy. Section 4 shows the results and discussions. Section 5 summarizes the discussion and describes the future directions.

2. Preliminaries

2.1 Introduction to OpenCL

OpenCL is an open standard maintained by the Khronos group[2]. It is designed for general purpose computing on GPGPU, CPU and accelerators such as CELL processors and DSPs. There has existed OpenCL implementation for both Nvidia GPGPU products and AMD GPGPU products, and also both OpenCL programming SDK for AMD CPU series and Intel CPU series have been issued. The latest version of OpenCL standard is the OpenCL version 1.2, but we only employ OpenCL version 1.1 implementations in this work.

The platform model of OpenCL specification[3] contains a host connected to device(s). Execution is performed on an N-dimensional grid of work-items on devices by invoking data-parallel kernels. Work-items are kernel instances and organized into work-groups. Each work-item executes the same code but the specific execution pathway through the code and the data operated upon can vary per work-item. Each work-group can be uniquely identified by its work group ID, and each work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The NDRange is an N-dimensional index space of work-items, where N is one, two or three. Based on the specific characteristic of devices, the layout of NDRange can affect the performance, which means the NDRange layout can be one of tuning parameters for optimizing performance.

2.2 Machine Learning and Statistical Modeling

In Ethem Alpaydin's book[4], he defines machine learning as:

. . . . . Machine learning is programming computers to optimize a performance criterion using example data or past experience. We have a model defined up to some parameters, and learning is the execution of a computer program to optimize the parameters of the model using the training data or past experience. The model may be predictive to make predictions in the future, or descriptive to gain knowledge from data, or both. . . . . .

Most scientists using linear solvers are not primarily trained as numerical analysts and may lack the expertise to select suitable solvers[5]. And it is not only linear solver details have impact on the performance but also the data structures for storing the data on memory have. So it is convenient that the program have the ability of deciding how to tune itself automatically without or partially without the involvement of human. This is the reason that we choose machine learning as our tool to implement our software automatic tuning strategy. Specifically speaking, we use machine learning technique to build our statistical models on the basis of automatic experimenting, and then use these models to decide the value of tuning parameters. By tuning parameters, we mean the changes of their values can have impact on the computing performance, and the program tunes itself by changing the values of these parameters.

In the machine learning field, supervised learning is one of the most important types of learning algorithms. Both regression and classification are supervised learning problems where there is an input, X and output Y, and the task is to learn the mapping from the input to the output (p.9 in 4)).

2.3 Krylov Subspace Methods

Yousef Saad's book[6] explains the details of Krylov Subspace Methods. He mentions that a general projection method for solving the linear system

$$Ax = b \tag{1}$$

extracts an approximate solution x_m from an affine subspace $x_0+\mathcal{K}_m$ of dimension m by imposing the Petov-Galerkin condition

$$b - Ax_m \perp \mathcal{L}_m \tag{2}$$

where $\mathcal{L}_m$ is another subspace of dimension m. Here, x_0 represents an arbitrary

initial guess to the solution. The Krylov subspace method is a method for which the subspace $\mathcal{K}_m$ is the Krylov subspace

$$\mathcal{K}_m(A, r_0) = span\{r_0, Ar_0, A^2r_0, \ldots, A^{m-1}r_0\} \quad (3)$$

where $r_0 = b - Ax_0$. The different versions of Krylov subspace methods arise from different choices of the subspace $\mathcal{L}_m$ and from the ways in which the system is *preconditioned*. Most importantly, Krylov subspace methods are iterative methods, which means they execute the exactly same routines until the convergence criterion can be satisfied.

CG algorithm (algorithm 6.18[6]) is one of the most important algorithms of Krylov subspace methods. It contains one SpMV*(Sparse Matrix-Vector Multiply)* calculation, two vector inner product calculations, and three calculations of SAXPY (DAXPY) for each of the CG loops. This implies that we can improve the performance of CG algorithm by improving the performance of each of this three types of calculations respectively. Here is the detailed description of CG algorithm

$Compute \quad r_0 := b - Ax_0, p_0 := r_0$
$For \quad j = 0, 1, \ldots, until \quad convergence \quad Do:$
  $\alpha_j := (r_j, r_j)/(Ap_j, p_j)$
  $x_{j+1} := x_j + \alpha_j p_j$
  $r_{j+1} := r_j - \alpha_j Ap_j$
  $\beta_j := (r_{j+1}, r_{j+1})/(r_j, r_j)$
  $p_{j+1} := r_{j+1} + \beta_j p_j$
$EndDo$

2.4 Related Works

There has been some works on applying machine learning techniques to linear system solver related problems. Shuting Xu and Jun Zhang[7] have been using clustering analysis and SVM*(Support Vector Machine)* classification techniques to predict whether the a matrix can be solved by a preconditioner (in a preconditioned iterative solver). In their method, the features of sparse matrices are calculated firstly for training, both unsupervised learning and supervised learning are applied for learning. They show that the overall accuracy of the prediction is above 90% for the ILU0 preconditioner and above 87% for the ILUK preconfitioners. In America Holloway and Tzu-Yi Chen' work[8], they also evaluate the effectiveness of machine learning as a tool for predicting whether a particular combination of preconditioner and iterative method will correctly solve a given sparse linear system, but the tool that they use is neural network. In anther paper[9], Shuting Xu and Jun Zhang apply SVR *(Support Vector Regression)* to predict the condition number the a matrix, their results are not as precise as the general direct computation methods.

Jee W. Choi and et al.[10] proposed a performance model-driven framework for autotuning of SpMV on GPU. Their model is analytical and requires offline measurement and run-time estimation, and they show that their model can identify the implementation that achieve within 15% of those found through exhaustive search. Besides tuning on the basis of existing code, Dominik Grewe and Anton Lokhmotov[11] propose a system-independent representation of sparse matrix formats that allows a compiler to generate efficient, system-specific code for sparse matrix operations. Katsuto Sato and et al.[12] proposed a history-based approach that uses profile data for performance prediction. Their method is for general algorithms rather than only for linear system solvers or SpMV kernels, and they are using general linear least square method for building a linear statistical model.

In Takao Sakurai and et al.'s work[13], they proposed an auto-tuning method for selecting the best SpMV algorithms out of seven implemented algorithms and showed that the maximum speedup established 12.7% for symmetric sparse matrix and 2.5% for unsymmetric ones. They use exhaustive search for algorithm selection. Satoshi Ohshima and et al.[14] argue that segmented scan method implementation of CPU is not suitable for GPGPU because of the utilization of shared memory and the branches with block. So they proposed an improved version of this method which is called branchless segmented scan method. In Yuji Kubota and Daisuke Takahashi's work[15], they proposed an auto-tuning strategy to automatically select matrix storage formats for SpMV by exhaustive search, and evaluated their strategy on CG solver.

3. Tuning Strategy

In this section, we detail our proposal of three-step tuning method. As we have explained in 2.3, CG method mainly contains three types of routines, which are Spmv, inner product and SAXPY (DAXPY). The time consumption of CG

method depends on the time consumption of them within each loop and the amounts of loops, and the performance of CG method only depends on time consumption of these three types of routines within each loop after the input problem and convergence criterion of CG method is given. In order to tune the CG method performance, we focus on tuning SpMV, inner product, and SAXPY (DAXPY) kernel.

In our proposal, we tune SpMV for a specific run-time environment by choosing an appropriate matrix storage format and and OpenCL work-group size using our statistical model. The storage formats that we are using are CSR, DIA, COO, ELLPACK, HYB[16]. And then we profile the first several loops of CG to decide a specific OpenCL SpMV kernel of the previously chosen storage format and to further tune the SpMV kernel by adjusting the work-group size for OpenCL kernel execution.

We have defined two types of kernels for vector inner product calculation. To tune the inner product kernels, we simply decide the inner product kernel type and work-group size by profiling the first several loops of CG.

When it comes to SAXPY (DAXPY), we define four types of kernels on the basis of OpenCL build-in vector data types. The kernel which is used in run-time is decided using statistical model, and so is the work group size. Like tuning SpMV, we further adjust work group size by profiling the first several loops of CG routine.

We call our method *Three-Step Performance Automatic Tuning Strategy* because the implementation of our strategy is done through three steps. The first step employ an automatic experimenting system to exhaustively profile our SpMV, inner product and SAXPY (DAXPY) kernels with different configurations to get our training dataset, and then in the second step we build our statistical models by learning from training dataset. In the last step, we further tune SpMV, inner product and SAXPY (DAXPY) using profiling data that is obtained from the first several loops of CG method execution.

3.1 Matrix Feature Extraction

In our training dataset for building performance model of SpMV kernels, we need extract information of input sparse matrix features. **Table 1** shows the symbols of each extracted features.

`num_row` : the number of rows of the matrix.

`nnz` : the number of nonzero elements of the matrix.

`nzAveR` : the average value of number of nonzero elements over num_row.

`nzstdDev` : standard deviation of nzAveR over the number of nonzero elements of each row.

`nzmax` : the number of nonzeros elements in the row that contains the maximum number of nonzero elements.

`nzmin` : the number of nonzeros elements in the row that contains the minimum number of nonzero elements.

`diagDisAveAveRow` : if we let diagDisAveRow be the average value of distance from nonzero elements to its diagonal element of the row, then diagDisAveAveRow is the average value of diagDisAveRow over rows.

`diagDisAveAveRowStdDev` : the standard deviation of diagDisAveAveRow over each row's diagDisAveRow.

`avLowBand` : the average value of each row's maximum distance of the nonzero element in lower part of the matrix over rows.⋆1.

`avUpBand` : the average value of each row's maximum distance of the nonzero element in upper part of the matrix over rows.⋆2.

`avLowBandStdDev` : the standard deviation of avLowBand over each row's maximum distance of the nonzero element in lower part of the matrix.

`avUpBandStdDev` : the standard deviation of avUpBand over each row's maximum distance of the nonzero element in upper part of the matrix.

3.2 OpenCL Kernel Description

We implement ten types of kernels for SpMV. Since we have five types of matrix storage formats, there are two types of kernels for each of storage formats. One kernel of the the the two uses one OpenCL kernel thread for each row of the matrix, and the other one uses one work-group for the calculation of one row. The key difference is that the later type of the OpenCL kernel uses local memory to buffer the intermediate results, and then the kernel employs a reduction procedure

⋆1 if an element of the matrix is noted as a_{ij} , then all the elements in lower part of the matrix satisfies $i > j$

⋆2 if an element of the matrix is noted as a_{ij} , then all the elements in upper part of the matrix satisfies $i < j$

Table 1 Extracted sparse matrix features

num_row	nnz	nzAveR	nzstdDev	nzmax	nzmin	diagDisAveAveRow	diagDisAveAveRowStdDev	avLowBand	avUpBand	avLowBandStdDev	avUpBandStdDev

to obtain the values of each dimension of the output vector's as the final results.

For inner product kernels, we also implement two types of OpenCL kernels. The first type uses each kernel thread for each dimension of the vector, and offloads the reduction part totally on host. The other type of kernel also uses one kernel thread for each dimension of the vector, but a part of the reduction job is done on device using local memory.

We define five types of kernels for SAXPY (DAXPY). The only difference between these kernels is that they employ different lengths of OpenCL build-in vector type for calculation. The vector length that we are using are 1, 2, 4, 8 and 16.

In this work, the implementation details of each type of kernels are not important. The most important thing is that they are different in respect of the presentation of OpenCL kernel performance under different input problems and OpenCL work-group sizes, and our target is to let our program decide the which kernel to use automatically on the basis of criterion.

3.3 Statistical Modelling

In this work, we use two methods for modelling. They are SVM*(Support Vector Machine)* for classification and SVR*(SVM Regression)*.

SVM is one of the machine methods that is especially important to data mining and pattern recognition problems. Kristin P. Bennett and Colin Campbell[17] has discussed its benefits and shortcomings. Chin-Wei Hsu and et al.[18] have given an recipe for rapidly obtaining acceptable results using SVM in terms of practical utilization. SVR is an extension of SVM to regression problems. In Chih-Chung Chang and Chih-Jen Lin' work[19], the SVM for classification and SVR that we are using are called *C-Support Vector Classification* and ϵ-*Support Vector Regression* (ϵ-SVR). The detailed information of SVM for classification and SVR can be found in Alex J. Smola and Bernhard Schlkopf's article[20] and Christopher M. Bishop's book[21].

Kernel functions (chapter 6 in 21)) make the SVM nonlinear. In this work, we employ two types of kernel functions, which are Gaussian RBF (*Radial Basis Function*) kernel function and polynomial kernel function for modeling. Gaussian RBF kernel function is in the form of

$$exp(-\gamma\|x - x'\|^2) \tag{4}$$

where γ is an parameter that need to be assigned by users. Polynomial kernel function is in the form of

$$(\langle x, z\rangle + v)^d \tag{5}$$

where degree d and offset v are parameters that need to be chosen by users.

By profiling the OpenCL kernels and counting the amount of floats computations (single precision and double precision) involved in the OpenCL kernels, we can only get the information of performance. But we also need a criteria to translate the performance from real number into binary for classification. Here is the definition of our criterion

$$valThreshold = valMean + factor * (valTop - valMean) \tag{6}$$

where $valMean$ and $valTop$ are the average value and highest value over all the performance data in the training dataset, and $factor$ is a parameter which is in the range of $[0, 1]$ and need to be chosen by users. If a performance value is smaller than $valThreshold$, we label the corresponding training target as -1, otherwise we label the corresponding training target as 1.

4. Results and Discussions

To obtain the training dataset, we choose all the forty-five SPD*(Symmetric Positive Definite)* sparse matrices with the number of rows that are from 1000 to 5000 from the university of Florida sparse matrix collection. To collect the performance data, we use the procedure in **Fig. 1**. We employ this procedure to collect the performance data from the calculations of the forty-five chosen matrices one by one. We assign 0.1 to the *factor* parameter described in 3.3. Besides matrix features, we also collect data of storage formats, OpenCL kernels and run-time work-group size. **Table 2** represents the all the features we choose to train SpMV performance model. For the storage formats part of Table 2, if one

storage format is chosen, we assign 1 to the corresponding column, otherwise we assign 0 to the column. "kernels" column of Table 2 contains a sub-column space, and each column in the sub-column space represents a type a OpenCL kernel that we implemented. Like the assignment principle for storage formats columns, we assign 1 to one column in sub-columns if the corresponding OpenCL kernel type is chosen for the experiment. "wgs" column represents the work-group size that we use for NDRange layout at run-time. "matrix features" column also contains a sub-column space, please refer to Table 1 for details. **Table 3** represents the features that we choose to train the statistical performance models for SAXPY (DAXPY). "kernels" and "wgs" have the same meaning as they are in Table 2

1: Read the matrix by CSR format
2: Analyze the features of the matrix
3: Execute each of the two kernels of SpMv with the matrix and collect performance data
4: Translate the storage format form CSR to DIA, COO, ELL and HYB one by one and re-do step 3 after each of the translations
5: Execute inner product kernels with the vector length equating to the number of rows of the matrix and collect the performance data
6: Execute the SAXPY (DAXPY) kernels and collect the performance data

Fig. 1 Auto-experimenting design

Table 2 Features collected for SpMV related training

CSR	DIA	COO	ELL-PACK	HYB	kernels	wgs	matrix features

Table 3 Features collected for SAXPY(DAXPY) and Inner product related training

vector length	kernels	wgs

We use an open-source machine learning library that is called Shark (version 2.3.4)[22] for training and testing our models, and we run our test under Ubuntu 11.04. The hardware architectures that we have tested are AMD HD7970 GPGPU, AMD Phenom II X6 1090T CPU and Intel i7-3960X CPU.

We are using two types of criterion to evaluate the performance of our models. To SVM classification problems, we use the *ClassificationError* class of Shark library to evaluate the performance of our models. The class contains a method to compute the fraction of wrongly classified example. We will use the difference between 1 and the value of this error to represent the accuracy of our models. To SVM regression problems, *MeanSquaredError* class of Shark library contains methods to evaluate the MSE *(Mean Squared Error)* given by our models.

We use 5-fold-cross-validation[21] to search for an appropriate parameter C for our SVM model, and then we use the same value through all of our experiments. In this work, the chosen value of the parameter C is 50. When using RBF kernel function, the chosen value of the parameter γ is 0.5. We assign 4 and 1.0 to the *degree* and $offset$ parameters of polynomial kernel function for SpMV, and assign 6 and 1.0 to the *degree* and $offset$ parameters of polynomial kernel function for SAXPY(DAXPY). We use 60% of our dataset as the training dataset and use the left 40% as test dataset.

Table 4 to **Table 7** show the results of our evaluations. "train.rbf" means evaluations of training dataset using RBF kernel function, and "test.rbf" means evaluations of test dataset using RBF kernel function. "train.poly" and "test.poly" means evaluations using polynomial kernel function. "XXX.s" and "XXX.d" means the training and test dataset are obtained from the OpenCL kernel executions under single precision and double precision on XXX hardware. "Accuracy" data are obtained using $1 - classification\ error$. Values in Table 4 and **Table 5** are measured by percentage.

From the evaluation results, we can see that the statistical models built from polynomial kernel function present a higher prediction accuracy and lower MSE on test dataset than training dataset. This behaviour is abnormal, because the statistical models are obtained by learning from the training dataset rather than the test dataset. Table 4 and Table 5 illustrate that statistical models that are built from RBF kernel function for SVM classification problems represent a higher prediction performance to SpMV than to SAXPY(DAXPY). **Table 6** and Table 7 illustrate that statistical models that are built from RBF kernel function for SVM regression problems represent similar prediction performance to both

SpMV and SAXPY(DAXPY).

Table 4 Accuracy of statistical models of SpMV

%	HD7970.s	HD7970.d	X1960.s	X1960.d	i7-3960X.s	i7-3960X.d
train.rbf	100	100	100	100	100	100
test.rbf	93.46	94.32	87.78	86.08	83.81	83.52
train.poly	77.08	75.09	81.63	82.76	83.05	83.14
test.poly	94.32	93.46	87.78	86.08	83.81	83.52

Table 5 Accuracy of statistical models of SAXPY(DAXPY)

%	HD7970.s	HD7970.d	X1960.s	X1960.d	i7-3960X.s	i7-3960X.d
train.rbf	100	100	100	100	100	100
test.rbf	1.7	1.32	96.21	93.18	3.22	2.84
train.poly	42.8	39.77	56.44	60.6	45.45	44.06
test.poly	99.81	100	95.08	91.18	98.3	98.67

Table 6 MSE of statistical models of SpMV

	HD7970.s	HD7970.d	X1960.s	X1960.d	i7-3960X.s	i7-3960X.d
train.rbf	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
test.rbf	0.38	0.28	0.07	0.07	1.7	1.6
train.poly	1.58	1.37	0.23	0.2	4.97	4.67
test.poly	0.24	0.21	0.09	0.09	2.21	2.07

Table 7 MSE of statistical models of SAXPY(DAXPY)

	HD7970.s	HD7970.d	X1960.s	X1960.d	i7-3960X.s	i7-3960X.d
train.rbf	0.0001	0.0001	0.00008	0.00008	0.00009	0.00009
test.rbf	0.28	0.27	0.003	0.0022	0.099	0.097
train.poly	0.98	0.92	0.01	0.01	0.46	0.46
test.poly	0.18	0.17	0.003	0.003	0.14	0.14

5. Conclusion and Future Works

We have proposed a three-step auto-tuning strategy for CG method. The main idea behind this strategy is to employ statistical performance model to off-load the burden of exhaustive searching for the appropriate matrix storage format on OpenCL run-time. To further calibrate tuning parameters, we also employ on-line tuning using the first several loops of CG method.

We described the statistical performance models built by using machine learning techniques and evaluated the performance of our models. We have shown that RBF kernel function can be quit efficient to classification problems of OpenCL SpMV kernels on all of our tested hardwares, but it only acts efficient on parts of our tested hardwares to classification problems of OpenCL SAXPY(DAXPY) kernels. When it comes to SVM regression problems, we have shown that RBF kernel function is universally more efficient than polynomial kernel function under our experiment configurations.

Our future target is to future improve the performance of our statistical performance models, and improve our tuning strategy to make it more practical and robust. Furthermore, we will integrate our statistical performance model into CG implementation to test the effectiveness of our three-step tuning strategy.

Acknowledgments This work is partially supported by Grant-in-Aid for Scientific Research (B) "Adaptive Auto-tuning Technology Aiming Complex Multicore and Multiprocessor Environments" and JST CREST "ULP-HPC: Ultra Low-Power, High-Performance Computing via Modeling and Optimization of Next Generation HPC Technologies"

References

1) Naono, K., Teranishi, K., Cavazos, J. and Suda, R.: *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, chapter1, Springer (2010).
2) Khronos, G.: OpenCL - The open standard for parallel programming of heterogeneous systems, Khronos Group (online),
available from ⟨http://www.khronos.org/opencl/⟩ (accessed 2012-02-20).
3) Khronos OpenCL Working Group: *The OpenCL Specification version 1.1 Document Revision: 44* (2011).
4) Alpaydin, E.: *Introduction to Machine Learning*, The MIT Press, 2nd edition (2010).
5) Bhowmick, S., Eijkhout, V., Freund, Y., Fuentes, E. and Keyes, D.: Application of Alternating Decision Trees in Selecting Sparse Linear Solvers, *Software Automatic Tuning: From Concepts to the State-of-the-Art Results* (2010).
6) Saad, Y.: *Iterative methods for sparse linear systems*, SIAM, 2nd edition (2003).

7) Xu, S. and Zhang, J.: A new data mining approach to predicting matrix condition numbers, *Commun. Inf. Syst.*, Vol.4, No.4, pp.325–340 (2004).
8) Holloway, A. and Chen, T.-Y.: Neural Networks for Predicting the Behavior of Preconditioned Iterative Solvers, *ICCS '07*, pp.302–309 (2007).
9) Xu, S. and Zhang, J.: A data mining approach to matrix preconditioning problem, Technical Report 433-05, Department of Computer Science, University of Kentucky, Lexington KY (2005).
10) Choi, J.W., Singh, A. and Vuduc, R.W.: Model-driven autotuning of sparse matrix-vector multiply on GPUs, *PPoPP '10*, pp.115–126 (online), DOI:http://dx.doi.org/10.1145/1693453.1693471 (2010).
11) Grewe, D. and Lokhmotov, A.: Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation, *GPGPU-4*, (online), DOI:http://dx.doi.org/10.1145/1964179.1964196 (2011).
12) Sato, K., Komatsu, K., Takizawa, H. and Takizawa, H.: A History-Based Performance Prediction Model with Profile Data Classification for Automatic Task Allocation in Heterogeneous Computing Systems, *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, pp.135–142 (online), DOI:http://dx.doi.org/10.1109/ISPA.2011.36 (2011).
13) Sakurai, T., Naono, K., Katagiri, T., Nakajima, K., Kuroda, H. and Igai, M.: Sparse Matrix-Vector Multiplication Algorithm for Auto-Tuning Interface "OpenATLib", IPSJ SIG Technical Report Vol.2010-HPC-125 No.2 (2010).
14) Ohshima, S., Sakurai, T., Katagiri, T., Nakajima, K., Kuroda, H., Naono, K., Igai, M. and Itoh, S.: Optimized Implementation of Segmented Scan Method for CUDA, IPSJ SIG Technical Report Vol.2010-HPC-126 No.1 (2010).
15) Kubota, Y. and Takahashi, D.: Optimization of sparse Matrix-Vector Multiplication by Auto Selecting Storage Schemes on GPU, IPSJ SIG Technical Report Vol.2010-HPC-128 No.19 (2010).
16) Bell, N. and Garland, M.: Efficient Sparse Matrix-Vector Multiplication on CUDA, Nvidia technical report, NVIDIA Corporation (2008).
17) Bennett, K.P. and Campbell, C.: Support Vector Machines: Hype or Hallelujah?, *SIGKDD Explorations*, Vol.2, p.2000 (2003).
18) Hsu, C.-W., Chang, C.-C. and Lin, C.-J.: A Practical Guide to Support Vector Classification, Technical report, Department of Computer Science, National Taiwan University (2003).
19) Chang, C.-C. and Lin, C.-J.: LIBSVM: A library for support vector machines, *ACM Transactions on Intelligent Systems and Technology (TIST)*, Vol.2 (online), DOI:http://dx.doi.org/10.1145/1961189.1961199 (2011).
20) Smola, A.J. and Schlkopf, B.: A tutorial on support vector regression, *Statistics and Computing*, Vol.14 (online), DOI:http://dx.doi.org/10.1023/B:STCO.0000035301.49549.88 (2004).
21) Bishop, C.M.: *Pattern Recognition and Machine Learning (Information Science and Statistics)*, Springer (2006).
22) Igel, C., Glasmachers, T. and Heidrich-Meisner, V.: Shark, *Journal of Machine Learning Research*, Vol.9, pp.993–996 (2008).