Platform-Aware Dynamic Configuration Support for Efficient Text Processing on Heterogeneous System

Mi Sun Park[†], Omesh Tickoo[‡], Vijaykrishnan Narayanan[†], Mary Jane Irwin[†], Ravi Iyer[‡]

[†]The Pennsylvania State University
University Park, PA 16802
{mup183, vijay, mji}@cse.psu.edu

[‡]Intel Corporation
Hillsboro, OR 97124
{omesh.tickoo, ravishankar.iyer}@intel.com

***Abstract*—Significant efforts have been made in accelerating computer vision and machine learning algorithms by utilizing parallel processors such as multi-core CPUs and GPUs. Although the suitability of GPU is well-known for computer graphics and image processing applications which require massively parallel floating-point computations, recent research movement towards general purpose computing on-GPU (GPGPU) makes it possible to take advantage of parallel processors to accelerate text processing applications as well. However, how to fully leverage different types of parallel processor architectures to obtain optimal performance (especially with text) without making specific efforts to each platform still remains a great challenge.**

We applied performance and accuracy enhancements to Naive Bayes algorithm to develop a practically sound implementation of text classification. A platform-aware dynamic configuration support automation flow is also proposed to support the seamless execution of our work across platforms. Experiments on various (integrated graphics, dedicated multiple GPUs) platforms demonstrate that our proposed approach improves both accuracy and performance of text classification.

I. Introduction

With the advent of the Internet and social media, the generation of data has been growing exponentially. It is reported that we create 2.5 quintillion (10^{18}) bytes of data every day and 90% of the world's data has been generated in the last two years alone [1] [2]. The era of big data has driven great interest in data mining and analysis to effectively extract information for user services. Due to the size and amount of data, it becomes harder to perform real-time text processing with general purpose CPUs. Further, although the evolution of GPU architecture and parallel programming models such as OpenCL [3] and CUDA [4] made it easier to accelerate computationally intensive applications using GPU, real-time text processing is still a great challenge. Especially the irregular nature of non-fixed length strings and less involvement of parallel floating-point computations have prevented text processing from taking full advantage of massively parallel processors. Therefore, research and development on an accelerator for real-time text classification will provide valuable contributions to text processing domain by sharing practical design and optimization techniques.

It should be noted that although CUDA (Compute Unified Device Architecture) [4] has more mature features and tools than cross-platform OpenCL (Open Computing Language) [3], it is specific to NVIDIA GPUs. OpenCL is a framework for parallel programming of heterogeneous systems that consist of multi-cores, GPUs and other processors. Currently there are plenty of OpenCL support hardware in the market. Major manufacturing companies such as Intel, AMD, NVIDIA and ARM provide their own OpenCL SDK to support for multi-core CPUs, desktop GPU and embedded GPU. In general, there exist two types of GPU: a power-efficient integrated (aka embedded) and a high-performance dedicated (aka discrete) GPU. Dedicated GPU has its own independent memory that leads to increased performance, but it is power-hungry and easily heat up. On the other hand, the integrated GPU doesn't have its own memory and shares the small portion of system memory with CPU, and consumes less power. In spite of the limited computing capability, the low-power and low-heat characteristics make integrated GPU ideal especially for embedded system including mobile devices.

There are over 100 existing GPU-accelerated applications [5] and over 15 GPU vendors including mobile GPU area. Although various applications of GPU have grown significantly, there is a portability issue. Additionally, the performance gains in these applications are hard to generalize because it heavily depends on a specific platform and vendor. To overcome the limitation, we propose an OpenCL-based platform-aware dynamic configurable automation flow that helps seamless execution across platforms and provides optimal performance. By leveraging the proposed automation flow, we were able to make a fair performance comparison between the two different (integrated and dedicated GPU) architectures. This provides valuable insights into the characteristics of integrated GPU that has restricted features and runs at lower frequency, compared to dedicated GPU.

In this paper, we focus on the analysis and optimization of Naive Bayes algorithm and present a highly parallel Naive Bayes multiclass classifier for real-time text classification. A wide range of text classification applications include spam filtering, medical diagnosis, automatic categorization of newspaper articles and language identification. Recent study on opinion mining and sentiment analysis of SNS (social networking sites) such as Twitter and Facebook can also benefit from this classifier.

This paper provides the following key contributions:

- A feedback loop enabled approach is presented to efficiently process Naive Bayes text classification in practice.
- A OpenCL-based dynamic configurable automation flow, which includes a data partitioning technique for concurrent computations on multiple devices, is proposed for cross-platform and high performance.

We have incorporated these efforts in the development of a Naive Bayes classifier for efficient text processing with several optimization techniques from both algorithm and practical implementation perspectives. The experiment result of our GPU-enabled text classifier demonstrates 7.3X speedup with accuracy improvement over CPU platform.

II. Related Work

Since CUDA was released from NVIDIA in 2007 [4], a variety of GPU-based accelerators have been developed in across many research areas including visual categorization for efficiently managing large collections of images [6], automatic test pattern generation for high quality transition faults [7], and term frequency-inverse document frequency rank search engine for text mining [8], and executed on NVIDIA discrete GPU(s). Further, some researchers made efforts in designing OpenCL-accelerated applications of pattern classification, genetic programming tree evaluation and vehicle detection [9] [10] [11] to overcome the limitation of vendor-specific CUDA, while other researchers compared the performance of OpenCL and CUDA models [12] [13]. According to both the performance comparison studies between the models [12] [13], it is informed that OpenCL can be a good alternative to CUDA with portability on multiple architectures and insignificant performance loss.

Particularly for mobile embedded systems, OpenGL for Embedded Systems (OpenGL ES) programming model [14] and RenderScript

978-3-9815370-4-8/DATE15/©2015 EDAA

APIs [15] have been generally used to leverage the compute power of the embedded GPU. OpenGL ES is originally designed for rendering 2D/3D computer graphics, not for general-purpose computing, and RenderScript is designed only for Android platform. Recent research [16] shows the expansion of OpenCL domain to mobile heterogeneous system. It introduces a first OpenCL-based accelerator for image object removal algorithm on mobile GPU. This implies that our work can be easily deployed to the domain of mobile GPU platform as well.

Many efforts have been made to improve text classification algorithms and conduct comprehensive evaluation of the algorithms [17] [18] [19]. Although there is no one single algorithm in text classification that is best-known, Naive Bayes is definitely one of them and has proven to be efficient. Therefore, we revisit the algorithm, optimize it from a practical perspective and maximize parallel execution for fast computation. This paper presents a multiple GPUs-enabled highly parallel Naive Bayes text classifier by leveraging an OpenCL-based platform-aware dynamic configurable automation flow.

III. Naive Bayes Classifier

Naive Bayes (aka Naive Bayesian) classifier is a probabilistic classifier based on Bayes's theorem [20] with strong (naive) feature independence assumption. It is one of the most commonly used text classification algorithms. The goal of text classification (aka text categorization) is to determine which class a given document belongs to by finding the Maximum A Posterior (MAP) class. Naive Bayes multiclass classifier can be represented as:

$$C_{MAP} = \arg\max_{c \in C} P(c|d) \tag{1}$$

where C is a set of classes $(c_1, c_2, ..., c_m)$, d is a document represented as n features $(x_1, x_2, ..., x_n)$, and $P(c|d)$ is the posterior probability of a class given a document.

By applying Bayes's theorem, $P(c|d) = \frac{P(d|c)}{P(d)}$, and naive independence assumption, $P(x_1, x_2, ..., x_n|c) = \prod_{x \in X} P(x|c)$, we can now have the following equation:

$$C_{MAP} = \arg\max_{c \in C} \prod_{x \in X} P(x|c)P(c) \tag{2}$$

where $P(x|c)$ is the likelihood probability of each feature given a class and $P(c)$ is the prior probability of a class. After computing posterior probability of each class for the given document, Naive Bayes selects the most likely class which has the maximum posterior probability.

A. Enhanced Naive Bayes

We apply performance and accuracy enhancements to the original algorithm for more practical and reliable realization.

Performance: We applied *Laplace smoothing* and *log-domain conversion* methods to improve performance. Laplace smoothing (aka add-one smoothing) is an algorithm that simply adds one additional value to each word count to avoid a zero probability. It is simple but very effective, since a zero probability of any words eventually drives a zero posterior probability of a class that makes difficult for differentiating probabilities of classes and finding a MAP class.

Log-domain conversion is to perform addition instead of multiplication operation to prevent underflow when multiplying many small probabilities, as in the case of calculating likelihoods of rare words in a large document which have many words. We also realized that Naive Bayes uses $\arg\max$ for comparing the posterior probability of each class to find the most likely class. Thus, whichever technique we apply to calculate posterior probabilities doesn't significantly matter as long as we apply the same technique for computing posterior probabilities of all classes and compare them against each other. With this knowledge, we apply log over log_{10} since the smaller base of log can provide a wider dynamic range, which helps comparing with single-precision representation.

Accuracy: The current calculation for prior probability is oblivious of document size significantly. Larger documents have more words that provide additional information beneficial for text classification. The original formula for the prior computation for text classification is $P(c) = \frac{N_c}{N}$, where N_c is the number of documents in each class and N is the total number of documents in training. However, in practice, the size of documents varies significantly, therefore, we modified the formula by using words counts instead of document counts to avoid unbalanced document size as shown in Equation 3.

After applying all these optimizations, our final version of Naive Bayes algorithm for text classification is represented in Equation 3.

$$C_{MAP} = \arg\max_{c \in C} \{logP(c) + \sum_{x \in X} logP(x|c)\} \tag{3}$$

where $P(c) = \frac{count(c)}{\sum_{c \in C} count(c)}$ and $P(x|c) = \frac{count(x,c)+1}{count(c)+|V|}$ with that $|V|$ is the number of unique features in training.

We have implemented a Naive Bayes text classifier based on the above equation. The performance and accuracy enhancements we applied to the original algorithm help to achieve a better performance over existing algorithms (as will be shown in Section V).

IV. GPU-Accelerated Parallel Text Classification

In this section, we describe details of design and optimization techniques for efficient text processing on heterogeneous system.

A. Profiling

Figure 1 shows a work flow diagram of Naive Bayes multiclass classifier for text classification with two distinct stages of off-line training and on-line testing. It should be noted that this work flow diagram is similar to most of the contemporary Naive Bayes classifiers that are written sequentially. Based on the analysis and profiling of existing implementations of Naive Bayes, it is found that the step of *computing likelihoods of features* in the work flow takes the most execution time (over 99%) when running on CPU, as shown in Table I.

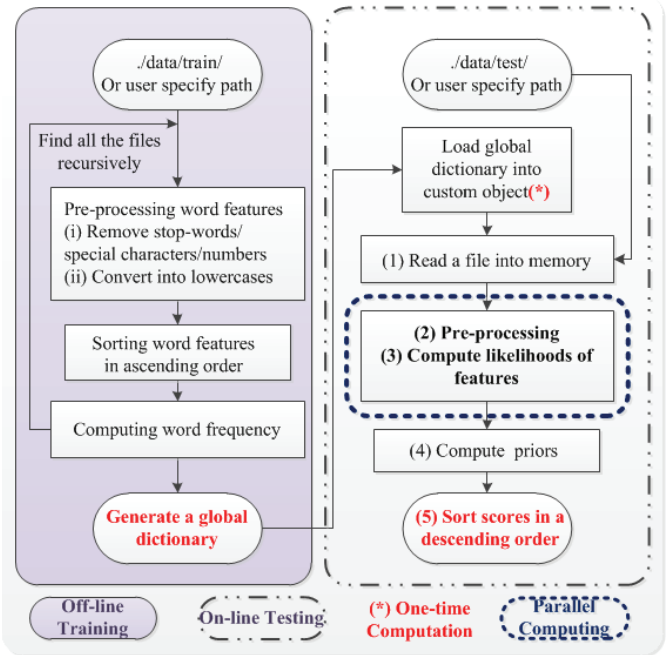

Fig. 1. Work Flow Diagram of Naive Bayes Classifier

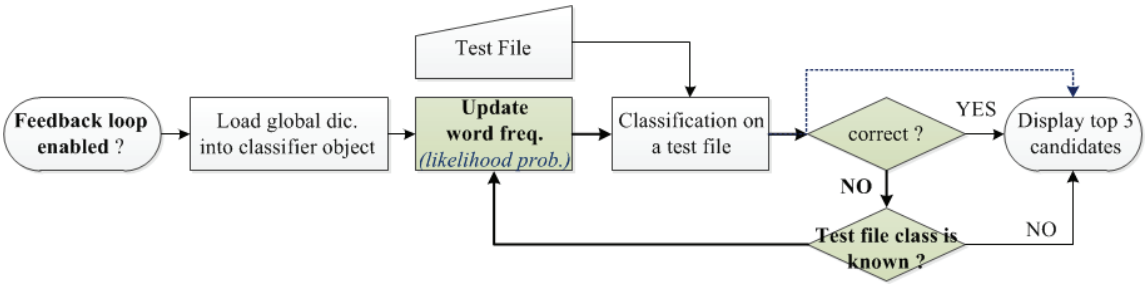

Fig. 2. Data Flow of our suggested Practical Approach

TABLE I
PROFILING OF NAIVE BAYES CLASSIFICATION ON CPU (W/ 13KB FILE)

Functions	Time (%)
(1) Read a test file into memory	0.03
(2) Preprocessing	**0.67**
(3) Compute likelihoods of features	**99.13**
(4) Compute priors	0.00
(5) Sort scores	0.17

Therefore, we restructured the sequential code to parallelize this step along with the previous step of *preprocessing* for maximum parallel execution. These two steps together take 99.8% of the execution time based on the profiling. The preprocessing step performs natural language processing-oriented operations such as removing stop-words, converting into lower cases, and omitting special characters/numbers from strings etc. Stop-words are words which contain no significant meanings and contributions in classification. In our case, we used a list of 517 English stop-words, and some examples of stop-words are a, but, or, the, and what. This preprocessing takes additional time to process, but it helps to improve accuracy of text classification. The computation time of the computing likelihoods step heavily depends on the number of words in a test file, the number of classes and the size of training data, since we need to compute likelihood of each word for each class and accumulate them, then we add a prior of each class in the later step. When we deal with bigger data, we can possibly obtain more performance gains by maximizing parallel execution on this computationally intensive step.

After deciding which part of the algorithm should be parallelized, we also tried to reduce the number of host-device data transfer by recomputing intermediate functions directly on GPU without returning to the host. Due to the nature of heterogeneous system (host + device), minimizing host-device data transfer is an important factor for optimal performance, especially for discrete GPUs which communicate from/to host via PCIe. Since host-device data transfer has much lower bandwidth than global memory access, one large transfer is much better than many small ones. The blue dashed box on Figure 1 represents the most important and computationally intensive computations in the Naive Bayes algorithm, and now mapped on GPUs with one invocation of GPU kernel execution.

B. Practical Approach

We suggest a feedback loop enabled approach to efficiently compute Naive Bayes classification in practice. Figure 2 shows data flow for the proposed approach.

1) Simplified Approach: Based on the analysis of Naive Bayes algorithm and its exiting implementations, we observed that all of them compute likelihood of each word and a prior probability of each class on-the-fly during the test stage, regardless of utilizing pre-counted words' frequencies from the training stage. Since all the data needed for computing likelihood and prior probabilities can be obtained from the training stage, we decided to push computation into off-line training stage as much as possible and perform minimal process during on-line testing for high throughput.

This is the reason why we came up with a global dictionary. Figure 3 shows the sample of a global dictionary structure with pre-computed

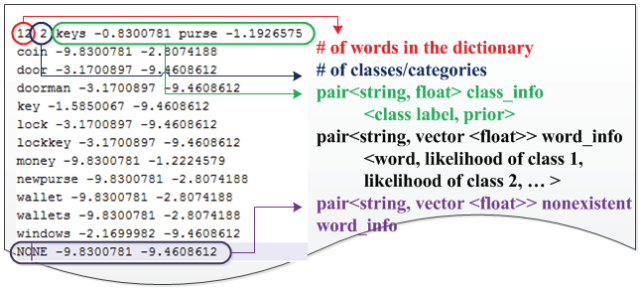

Fig. 3. Global Dictionary Structure with probabilities

probability. The idea is to store all necessary information such as class information (# of classes, a pair of class label and a prior), likelihood information (a pair of word string and likelihood of each class) and default likelihood for non-existent words etc. in the global dictionary and retrieve associated data by doing string matching against the dictionary when classifying a test documentation. The negative numbers in Figure 3 are because of computing log with fraction numbers. This simplified approach helps reducing classification computation time. Loading the global dictionary into classifier object is a one-time operation during initializing the classifier, therefore, it doesn't affect classification time.

2) Feedback Loop enabled Approach: Our simplified approach has one downside: it is unable to dynamically update the dictionary for fine tuning the classifier. In order to update the classifier, the simplified approach must go through the training stage again to generate a new global dictionary, which is usually time consuming. To overcome this issue, we added on-line feedback loop to the approach as shown in Figure 2. This feedback loop enabled approach is somewhat similar to most of the contemporary approaches by computing likelihood and prior probabilities on-the-fly, based on pre-counted words frequencies from training. The global dictionary used in this enhanced approach has words frequencies instead of probabilities. The main purpose of the feedback loop is to avoid redundant iterations of training stage and allows fine tuning of the classifier. In practice, training stage usually takes a large amount of time and how to train the classifier has a significant impact on the classification accuracy. Also, users are often encountered to a situation needed to add new classes or files to update the exiting classes of the classifier. Our feedback loop enabled approach can provide enough flexibility to satisfy these requirements.

Our flexible classifier supports both the approaches dynamically, therefore, determining an approach type should be purely based on user preference. Also after it classifies, it displays top 3 candidates of classes in a descending order of scores.

a) Log Approximation Fixed-point: Owing to the modern GPU architectures for fast floating-point computation with floating point units (FPU), we don't need to consider fixed-point computation seriously. However, many small embedded system especially with low-cost and low-power microprocessors don't have FPU. The internal GPU-like parallel processor, we are in early-stage design and plan to

use it as our final platform, doesn't intend to have FPU due to ultra low-power purpose. Therefore, we need to manipulate floating-point representations accordingly.

With the analysis of our classifier, we found that we only perform floating-point addition in the simplified approach, while we compute floating-point addition and division followed by log computation in the feedback loop enabled approach. In general, if we use floating-point operations in our program and compile for those FPU-less processors, it will use internal emulation libraries that are extremely slow. Although fixed-point computation is not a major focus of this research, it is good to consider what functions in our implementation should be modified or optimized in case it deploys on these FPU-less processors. We applied a computationally fast approximate logarithm algorithm with fixed-point by leveraging [21] method to our Naive Bayes implementation and analyzed effects on the execution time.

C. Automation Flow and Data Partitioning

As shown in Table II, there are generally recommended 12 basic steps for heterogeneous OpenCL programming. Basically host CPU builds up data on a device and then enqueues a kernel to execute on the device using the data. All the 12 steps are done in the host side, except the actual kernel execution after the host CPU deploys the kernel and tells the device to execute it in step (10). The steps from (1) to (7) are one-time OpenCL setup procedure that only execute once during initialization. The rest of the steps from (8) to (12) is device execution procedure for each test.

TABLE II
BASIC STEPS FOR OPENCL PROGRAMMING [3]

1	Obtain OpenCL platform	One-time setup
2	Obtain devices id	
3	Create context for device	
4	Create command-queue for target device	
5	Create program from source code	
6	Build the program	
7	Create kernel(s) from program functions	
8	Allocate device memory	GPU execution
9	Associate arguments to kernel with kernel object	
10	Deploy kernel for device execution	
11	Move output data to host memory	
12	Release context/program/kernels/memory	

By leveraging these steps, we propose a platform-aware dynamic configurable automation flow based on the properties of a given device/hardware and a target kernel. Particularly, we added two important steps to support true concurrent computations on multiple devices and provide optimal performance. The two steps are (a) checking available device count and (b) querying maximum kernel work group size for dynamic local work size assignment. Our proposed OpenCL-based dynamic configurable automation flow, shown in Figure 4, can be used for any application to support high performance, cross-platform and true concurrent computations on multiple devices. We initially applied two different methods to support multiple GPUs. The first is to have one context across all devices and one command-queue per device, while the second is to have one context and one command-queue per device. According to [22] of the performance comparison between the two methods, the latter with multiple host threads (one thread per device) is an ideal way to support true concurrent processing on multiple GPUs. Therefore, we first count the number of devices on a given hardware and automatically create the same number of contexts (one per device) accordingly.

After OpenCL setup is done, we have to consider two important execution configuration parameters for optimal performance: local work size (aka local work group size or # of work-items in a work-group) and global work size (aka total work-items). Choosing the work size is important for maximizing performance, although there is no rule of thumb. Local work size can be determined by querying maximum kernel work group size permitted by OpenCL support devices where work-items execute on. Since the maximum kernel

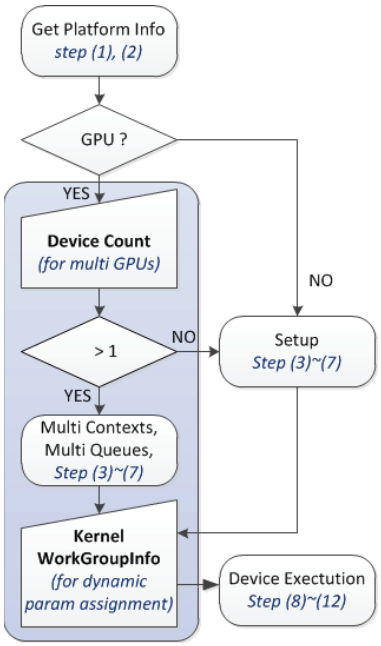

Fig. 4. Our proposed Dynamic Configurable Automation Flow

work group size is determined based on the resource requirements of the kernel [3], it provides a proper configuration for the specific kernel and the device. There is no constraint for global work size as long as it is a multiple of the local work size, therefore, we simply round up.

1) Data partitioning for multiple devices: We applied two ways to partition input test data for multiple devices. The first coarse-grain partitioning mechanism is to assign each test file to each device, since we use multiple host threads for each thread to have its own context and a test file to classify. This is simple but it has one disadvantage that is the computation time is determined by the slowest worker due to different file sizes. The second fine-grain method is to find proper memory index after copying a file to for each device. Unlike image pixels, words/strings have no fixed length. Therefore, we coarsely divide the memory by the number of target devices and increase memory index/address till it finds a space or carriage return or line feed for the exact index. This method is suitable for a large test file, otherwise, it would be better using one device from resource utilization perspective.

D. Optimizations

In general, there are three optimization strategies to improve overall performance in a heterogeneous system [4] [23]: (a) maximizing parallel execution, (b) optimizing memory access and (c) optimizing execution configuration. Maximizing parallel execution can be achieved by exposing data parallelism in target algorithms and overlapping memory transfer with computation. Memory optimization can be obtained by minimizing host-device data transfer, coalescing global memory access and maximizing the utilization of local memory that is much faster than global memory. Lastly, optimizing execution configuration is also important by finding the right parameters such as local and global work size for different applications on different GPU architectures to increase occupancy by hide latencies and keep the hardware busy.

After profiling, we have parallelized over 99% of the sequential program and executed the paralleled code on GPUs with only one invocation of kernel execution meaning a single data transfer from host CPU to the device GPUs. We also applied memory optimization techniques [23] such as using pinned memory and coalescing global memory access for optimal performance.

Pinned (aka page-locked) memory is the memory which prevents from being paged-out by operating system and provides higher bandwidth between host and device, since it allows the device to use DMA-transfer over PCIe. But it is also noted that over-allocating pinned memory might reduce overall system performance, because it reduces the amount of memory available to operating system and other programs. This default version of pinned memory is especially useful when loading or storing the data multiple times with discrete GPU. A different version, mapped pinned memory (aka zero-copy memory), allocates pinned host memory, map device memory to the pinned host memory and return the host pointer. In other words, kernels read the data directly from the host memory without explicitly copying the data from the host memory to the device memory. This mapped memory is better when kernels read and write the data exactly once with integrated GPU. Therefore, since each of these pinned memory type heavily depends on applications and target hardware, our implementation calls proper APIs dynamically after identifying a platform (integrated or discrete) type.

Coalesced memory access, which is highly recommended by [23], is for efficient global memory access on GPUs by combining multiple memory accesses into a single aligned memory access. It can be achieved when neighboring threads access neighboring locations in memory.

Optimizing execution configuration is somewhat heuristic, since it heavily depends on applications and GPU architectures. To avoid time-consuming process of experiments, we added a new *dynamic parameter assignment* step for finding proper configurations as our proposed automation flow shows in Figure 4. This step provides appropriate values of essential parameters dynamically based on the resource requirements of the kernel and the target device properties.

V. Experimental Results

We used two different (integrated and discrete) platforms and the platform specifications are provided in Table III. The discrete platform has a NVIDIA graphics card that contains two GPUs. Experiments on these different platforms can help us to verify our proposed dynamic configurable automation flow by dynamically configuring execution configuration parameters based on the hardware properties and kernels.

TABLE III
Platform Specification

	#1: Integrated Platform		#2: Discrete Platform	
	Intel CPU	Intel Graphics	Intel CPU	NVIDIA GPU
Model	i7-4770	HD 4600	Xeon E5405	GTX 590
# of Compute Units	8	20	4	16 x512 proc.
Freq. (GHz)	3.4	0.350	2	0.607
Memory (GB)	8	1.6	4	1.5
TDP (Watt)	84		80	365

A. Classification

Table IV shows the text classification accuracy of our Naive Bayes classifier. This experiment is conducted with 20 newsgroups data set [24] and cross-validation technique. The 20 newsgroups data set has 20 classes/categories and a newsgroup article is classified as belonging to one of the 20 newsgroups. The cross-validation is a model evaluation method which randomly divides documents into training and validation data and repeatedly test on the validation data for estimating accuracy using the model of the training data. It is a technique to predict how accurately a training model will perform in practice. We also applied this method to existing implementations of Naive Bayes and Linear SVMs classifier from CMU Bow toolkit [25] for classification accuracy comparison, since we applied performance and accuracy enhancements to the original Naive Bayes algorithm. For this experiment, we used up to 20 classes with 200 documents per each class and averaged out the results after 20 times of testing. Table IV shows that our classifier performs better than the other two existing implementations.

TABLE IV
Text Classification Accuracy with up to 20 classes

# classes	Our Naive Bayes	Naive Bayes [25]	SVMs [25]
20	87.50 %	86.15 %	77.05 %
18	89.89 %	88.37 %	78.13 %
16	89.81 %	88.58 %	79.99 %
14	90.79 %	89.14 %	78.40 %
12	91.42 %	91.51 %	78.71 %
10	92.70 %	91.89 %	79.06 %

B. Influence of Optimization Techniques

As the steps for OpenCL programming in Table II show, there are three distinct processes: one-time setup, GPU execution and kernel execution. The OpenCL setup from getting platform information to creating the kernel is one-time execution and the time on our integrated platform CPU takes about 154 ms. Since this is part of initialization time, it doesn't affect the throughput of our system. The second process from allocating device memory to writing device output back to host is considered as GPU execution, which operates for each test file. The process of deploying kernel for device execution is kernel execution that indicates pure computation time on GPU. For measurement, we used OpenCL profiling events for kernel execution and query performance counter API for GPU execution.

TABLE V
Performance Comparison of Two Approaches on the Integrated Platform (w/ 60,188 dictionary words, 517 stop words)

Test File	Average Time (s)	Simplified Approach	Enhanced Approach
50KB	Kernel Execution	0.193	0.195
	GPU Execution	0.197	0.200
25KB	Kernel Execution	0.111	0.113
	GPU Execution	0.115	0.116
13KB	Kernel Execution	0.065	0.067
	GPU Execution	0.069	0.071

Table V shows the performance comparison of our simplified approach and feedback loop enabled enhanced approach on the integrated platform with different sizes of test files. The execution time for both the approaches are similar, it is probably because GPU performs floating-point calculation very fast using FPU. Compared to the simplified version, the feedback loop enabled version computes one additional floating-point division and logarithmic calculation. It is observed that there is not much execution gap between the GPU execution and kernel time. It is because we experimented on the integrated platform meaning the host CPU and graphics reside on the same die, therefore, the communication cost is relatively small.

TABLE VI
Floating- vs Fixed-point Computation in our feedback loop enabled approach on CPU

	Log10		Log	
	floating	fixed	floating	fixed
Computation Time (ms)	15.24	17.92	15.18	17.82

The effects of data types, floating and fixed-point, on the classification time of 1KB test file are provided in Table VI. It is measured on CPU with two different log bases. From the result, we can verify that computation with fixed-point is faster than with floating-point. It is also observed that using *log* takes slightly shorter time than using *log10*. This further agrees with our decision of using *log* over *log10*, since when analyzing the algorithm we found that using smaller base of log results bigger difference between the numbers after log computation that leads to help comparing the scores of each class.

Table VII shows the performance improvement using mapped pinned memory (aka zero-copy memory) by mapping a buffer into host memory, and loading data directly from the host without allocating and transferring the data in advance. From experiment with 25KB test file, we gained 17% speed improvement with higher memory bandwidth by using mapped pinned memory on the integrated platform. Intel Vtune Amplifier XE 2013 [26] is used for measuring GPU OpenCL kernel performance on our Intel integrated platform.

TABLE VIII
EXECUTION TIME (S) COMPARISON WITH VARIOUS LOCAL WORK SIZE (W/ 200KB TEST FILE)

	Work Size		Work Size		Work Size		Work Size		Work Size		Work Size		Work Size	
	Local	Global	Local	Global	Local	Global	Local	Global	Local	Global	Local	Global	Local	Global
	16	27632	32	27648	64	27648	128	27648	256	27648	512	27648	1024	27648
Integrated Graphics	0.709		0.709		0.709		0.709		0.708		**0.707**		X	
Discrete GPU	0.860		0.715		**0.615**		0.661		0.723		0.730		0.743	
Discrete Two GPUs	0.457		0.382		**0.326**		0.354		0.388		0.393		0.377	

TABLE VII
IMPROVEMENT USING MAPPED PINNED MEMORY ON THE INTEGRATED PLATFORM

	Computation Time (s)	GPU Memory Bandwidth (GB/sec)	
		Read	Write
Non-Mapped Pinned	0.115	1.212	0.130
Mapped Pinned	0.098	1.301	0.140

C. Comparison with Different Platforms

The performance comparison with various local work size for fine tuning on integrated and discrete platforms is provided in Table VIII. As we previously discussed, local and global work size are two important factors to determine optimal performance. Since global work size is a multiple of local work size, we changed local work size and experimented on various platforms to see the effects on execution time. One interesting result is observed that local work size doesn't severely affect on the integrated platform, while it has a significant impact on the discrete platform. From the experiments, it is found that the local size of 512 and 64 are optimal configuration to the integrated and discrete platforms respectively. The test results also show that our work on the discrete platform performs better than the integrated platform. It is because the discrete GPUs run at higher frequency than the integrated graphics, and consumes more power.

TABLE IX
PERFORMANCE COMPARISON OF TEXT CLASSIFICATION ON MULTI-CORE CPUS AND GPUS (W/ 200KB TEST FILE)

	Time (s)
Native Multithreaded Implementation on multi-core CPUs	2.734
Our OpenCL-based Implementation on a GPU	0.615
Our OpenCL-based Implementation on two GPUs	0.326

Table IX shows that our OpenCL-accelerated Naive Bayes classifier on two GPUs and a single GPU provides 7.3X and 3.4X speedups over native multithreaded implementation with 2 active threads on multi-core CPUs respectively. Because of the portability of our work and proper data partitioning for multiple devices, the performance can further scale up with more number of GPUs.

VI. CONCLUSION

We applied performance and accuracy enhancements to Naive Bayes algorithm for more practical and reliable implementation. A feedback loop enabled approach is presented to efficiently process text classification in practice. And, an OpenCL-based dynamic configurable automation flow is proposed. This flow can help applications seamlessly executing across different platforms by dynamically configuring parameters based on target hardware and kernels. Experiments on various platforms have been performed to verify the effectiveness of our text processing by executing the same code. The experiment result shows our highly parallel text classifier provides 7.3X speedup over CPU implementation with accuracy improvement.

ACKNOWLEDGEMENT

This work was supported in part by grants from Intel and NSF Expeditions in Computing Visual Cortex on Silicon 1317560.

REFERENCES

[1] "ViaWest Data Center." [Online]. Available: http://www.viawest.com/sites/default/files/asset/document/ViaWest_Big_Data_Infographic.pdf
[2] "IBM Analytics - IT Business Intelligence." [Online]. Available: http://www.ibm.com/smarterplanet/us/en/business_analytics/article/it_business_intelligence.html
[3] "OpenCL 1.2 Specification." [Online]. Available: http://www.khronos.org/opencl/
[4] "CUDA C Best Practices Guide." [Online]. Available: http://www.nvidia.com/
[5] "NVIDIA GPU Applications." [Online]. Available: http://www.nvidia.com/object/gpu-applications.html
[6] K. E. van de Sande, T. Gevers, and C. G. Snoek, "Empowering Visual Categorization With the GPU," *Trans. Multi.*, vol. 13, no. 1, pp. 60–70, Feb. 2011.
[7] K.-Y. Liao, S.-C. Hsu, and J.-M. Li, "GPU-based N-detect transition fault ATPG," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, 2013, pp. 1–8.
[8] Y. Zhang, F. Mueller, X. Cui, and T. Potok, "GPU-accelerated text mining," in *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*, 2009.
[9] D. Bharangar, A. Doeger, and Y. Mittal, "Implementation of Fast Artificial Neural Network for Pattern Classification on Heterogeneous System," *IJSER*, 2013.
[10] D. A. Augusto and H. J. C. Barbosa, "Accelerated Parallel Genetic Programming Tree Evaluation with OpenCL," *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 86–100, Jan. 2013.
[11] K.-M. Cheng, C.-Y. Lin, Y.-C. Chen, T.-F. Su, S.-H. Lai, and J.-K. Lee, "Design of vehicle detection methods with opencl programming on multi-core systems," in *Embedded Systems for Real-time Multimedia (ESTIMedia), 2013 IEEE 11th Symposium on*, Oct 2013, pp. 88–95.
[12] J. Fang, A. Varbanescu, and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," in *Parallel Processing (ICPP), 2011 International Conference on*, 2011, pp. 216–225.
[13] C.-L. Su, P.-Y. Chen, C.-C. Lan, L.-S. Huang, and K.-H. Wu, "Overview and comparison of OpenCL and CUDA technology for GPGPU," in *Circuits and Systems (APCCAS), 2012 IEEE Asia Pacific Conference on*, 2012, pp. 448–451.
[14] "The Khronos Group, The OpenGL ES 3.0 Specification." [Online]. Available: http://www.khronos.org/opengles/
[15] "Google Inc., RenderScript - Android Development Guide." [Online]. Available: http://developer.android.com/guide/topics/renderscript/compute.html
[16] G. Wang, Y. Xiong, J. Yun, and J. R. Cavallaro, "Accelerating Computer Vision Algorithms Using OpenCL on the Mobile GPU ? A Case Study," in *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, may 2013.
[17] G. Forman, "An Extensive Empirical Study of Feature Selection Metrics for Text Classification," *J. Mach. Learn. Res.*, vol. 3, pp. 1289–1305, Mar. 2003.
[18] "Multinomial Naive Bayes for Text Categorization Revisited," in *AI 2004: Advances in Artificial Intelligence*, ser. Lecture Notes in Computer Science, G. Webb and X. Yu, Eds., 2005, vol. 3339.
[19] S. Pitigala, C. Li, and S. Seo, "A comparative study of text classification approaches for personalized retrieval in PubMed," in *Bioinformatics and Biomedicine Workshops (BIBMW), 2011 IEEE International Conference on*, 2011, pp. 919–921.
[20] T. Bayes, "An essay towards solving a Problem in the Doctrine of Chances," *Philosophical Transactions of the Royal Society of London*, vol. 53, pp. 370–418, 1763.
[21] C. Turner, "A Fast Binary Logarithm Algorithm [DSP Tips Tricks]," *Signal Processing Magazine, IEEE*, vol. 27, no. 5, pp. 124–140, 2010.
[22] "Single vs. Multiple contexts with multiple GPUs." [Online]. Available: https://devtalk.nvidia.com/default/topic/473251/cuda-programming-and-performance/single-vs-multiple-contexts-with-multiple-gpus/
[23] "OpenCL Best Practices Guide." [Online]. Available: http://www.nvidia.com/
[24] "20 Newsgroups Data set." [Online]. Available: http://www.nvidia.com/
[25] A. K. McCallum, "Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering," http://www.cs.cmu.edu/ mccallum/bow.
[26] "Intel VTune Amplifier XE 2013." [Online]. Available: http://software.intel.com/en-us/intel-vtune-amplifier-xe