Contention-Aware Scheduling of Parallel Code for Heterogeneous Systems

Chris Gregg Jeff Brantley Kim Hazelwood
Department of Computer Science, University of Virginia

Abstract

A typical consumer desktop computer has a multi-core CPU with at least two and possibly up to eight processing elements over two processors, and a multi-core GPU with up to 512 processing elements. Both the CPU and the GPU are capable of running parallel code, yet it is not obvious when to utilize one processor or the other because of workload considerations and, as importantly, contention on each device. This paper demonstrates a method for dynamically deciding whether to run a given parallel workload on the CPU or the GPU depending on the state of the system when the code is launched. To achieve this, we tested a selection of parallel OpenCL code on a multi-core CPU and a multi-core GPU, as part of a larger program that runs on the CPU. When the parallel code is launched, the runtime makes a dynamic decision about which processor to run the code on, given system state and historical data. We demonstrate a method for using meta-data available to the runtime and historical data from code profiling to make the dynamic decision, and we outline the runtime information necessary for making effective dynamic decisions, suggest hardware, operating system, and driver support.

1. Introduction

Because consumer computer systems are already heterogeneous, and because each of the heterogeneous processors has many cores that are able to run parallel code, there is a need for a method to determine how to optimally schedule parallel workloads for such systems. In general, if a programmer wants to run parallel code, he or she must decide what type of processor to target, write and compile the code for that architecture, and then assume that the processor will (1) be available to run the code when it becomes ready to launch, and (2) be the best processor to run that code on at that time. Because this decision is made statically at development and compile time, it may not be the best choice for every given state of the system. For instance, if the code is compiled to run on the GPU, and the GPU is busy with a graphical task at the moment the code is launched, there will be contention for the GPU resource. It may be the case, however, that the multi-core CPU could also have run the code and the parallel job could be completed in less time than it would have if it had to wait for the GPU. The task of handling a dynamic decision about which processor should be utilized could be

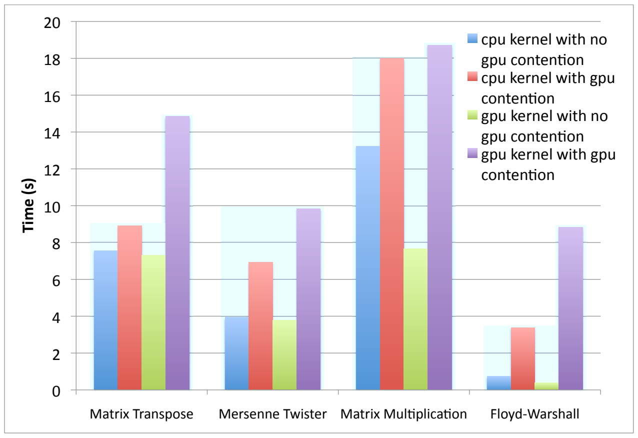

Figure 1. Parallel benchmarks demonstrating significant slowdown for GPU kernels when the GPU is under contention.

handled by a dynamic runtime (as we demonstrate in this paper), or it could be handled by the operating system, in a similar way that multicore scheduling for a single processor is done today. In either case, contention on a processor should be a large consideration in making that decision.

Figure 1 shows a set of benchmarks for OpenCL kernels that are run on both a CPU and a GPU in a heterogeneous system under different conditions. When the GPU is not under contention, it runs the kernels faster than on the CPU, but significantly slower when the GPU is processing other work as well. It is therefore our assertion that it is imperative to know whether devices are busy or not when making a dynamic decision about scheduling in a heterogeneous system.

In this research, we investigated what meta-information is available to a given runtime that is relevant to determining the most efficient use of heterogeneous resources, and we demonstrate a method for using that information along with historical data gathered from profiling to make dynamic scheduling decisions. We created a baseline framework for this sort of decision-making, and the framework includes a proof-of-concept demonstration of the idea, along with an initial set of rules for gathering and utilizing the runtime metadata. We show five programs containing parallel OpenCL kernels that can run on a CPU and a GPU. Four of the five kernels ran faster on the GPU under baseline conditions, and one had a crossover point when, given an increasing set of input data, it would switch from running faster on the CPU to the GPU. Specifically, we targeted contention on each processor, e.g., is the CPU busy with many threads, and would it be better to run the process on the GPU?, and strict speed per processor, e.g., is the code so favorable to the GPU that it is worth waiting if there is contention for the resource?

Our results show that a dynamic decision can be made with information available at runtime, such as device parameters, e.g., number of cores and processor speed, historical data (from profiling and in-process logging), problem size, and device status, e.g., busy or free, and we demonstrate that it is worthwhile to make such a decision on heterogeneous systems.

2. Background and Related Work

There are three main factors that determine whether a kernel will run faster on a GPU or CPU. The first is simply the result of the baseline profiling information: if the device for which the baseline runs fastest is not busy, the kernel will most likely run faster on that device. In the case of kernels that are twice as fast running on the GPU as on the CPU, the baseline profiling data dominates the scheduling decision. The second main factor is also simple: if one of the devices is currently running another kernel (or doing other processing, such as handling graphics routines), then the kernel of interest will most likely run faster on the other device. If specific information is available about what the processor is currently busy doing (i.e., running a short or a long kernel, handling graphics processing, etc.) then this information is particularly useful. For this paper, we reasoned that if a device was "busy," the decision should be made to run on the other device. The third factor that could affect running times for each device is the data input size for each kernel. If the algorithm depends on the data fitting completely into the device memory, some cases will only fit on the device with the greater amount of memory. In cases where this happens, there is a tradeoff for the programmer: he or she could spend time re-writing the application and kernel to take into account the memory constraints of the GPU, or he or she could accept that the code would only run on the CPU for larger input sizes.

Although desktop heterogeneous computing is relatively new as of the last few years, heterogeneous computing has been present in distributed computing systems for decades, and there has been significant research in how to schedule jobs across grid computers to satisfy load balancing and task throughput concerns. There has been much research in the static assignment of tasks to processors. Oh and Ha use a static method dependent on profiling on individual processors, and make a scheduling decision based on this information[5]. Sih and Lee make compile-time scheduling decisions that balances loads across heterogenous processors[8]. Siegel et al. [7] published work related to code profiling for heterogeneous systems, and they outlined a method for automatic construction of code that can be run on any machine in a heterogeneous computing system. Maheswaran et al. [4] considered mapping heuristics to use for dynamic scheduling of jobs across many machines, considering both task affinity for certain processors, aging of tasks waiting to run, and machine ready times. Our research is similar yet is specific to desktop heterogeneous computing on a single machine. Others have investigated scheduling on desktop heterogeneous platforms, and YuHai et al. [9] have developed an algorithm similar to what we propose, but with a greater focus on deadline scheduling for a set of heterogeneous tasks. Our work emphasizes what to do with tasks as they become ready to launch, instead of a set of tasks that all need to run in any order. Jiménez et al. [2] have also published work that schedules code on a CPU or GPU based on run-time history, and Becci and Crowley [1] also published work that uses historical thread behavior to schedule recurring work. Our research utilizes run-time history as part of our solution, but our solution also includes a more robust set of meta-data to complement the scheduling algorithm.

The idea of using meta-data to inform scheduling decisions has been investigated before. Shelepov and Fedorova [6] developed a signature-based framework that embeds meta-data about heterogeneous jobs into program binaries and presents this data to a scheduler when a job needs to be run. Our work is also similar to this, but it further incorporates real-time system state into the calculation as well.

Some of the newest research on dynamic scheduling on heterogeneous systems has been done by Luk et al.[3]. In their work, they use system configuration, problem size, and historical data to break up a parallel application into pieces that run efficiently at the same time on both the CPU and the GPU. They do not, however, consider contention of either device, whereas our proposed method does.

3. Approach

For this paper, we chose five different parallel applications to test, as shown in Table 1. The applications were implemented in C++ and the parallel kernels in OpenCL, enabling the kernels to run unmodified on either the CPU or GPU, requiring only a recompilation. We measured baseline kernel running times on an Intel Core 2 Duo dual-core processor running at 2.66GHz and an AMD Radeon HD 4350 GPU with 40 stream processors. The computer operating system was Linux (Ubuntu 9.04). To collect the baseline times, we ran a series of benchmarks for both the CPU kernels and the GPU kernels with the system in a relatively idle state. It is important to note that the run times were for the kernels only, and do not include the setup time for each kernel. OpenCL kernels are setup and compiled at runtime, and the setup times vary significantly for the CPU kernels and the GPU kernels. In all cases, the GPU kernels took longer to compile and set up than the CPU kernels; for our applications, the time to setup and compile a CPU kernel ranged from $0.1s$ to

$0.3s$, and the time to setup and compile a GPU kernel ranged from $0.3s$ to $2.3s$. For this paper, we decided to compile the kernels for both the CPU and GPU at the start of each program and time the kernels only. Our rationale was two-fold: first, because the setup for a kernel is distinct from the kernel launch, overall performance for critical kernel sections is independent from the setup time. Second, the performance for short running kernels (the majority of our test cases) that run faster on the GPU get masked by the slower setup and compile time, which is a fixed value for each kernel, but faster for CPU kernels. The methods for dynamic scheduling that we lay out in this paper could be modified to include setup and compile times, if desired.

We investigated how the data input size changed the running times on each device, and whether or not different input sizes should be considered in the decision making process. For each baseline experiment, we chose a data set that ranged from small to large according to the amount of memory used on each device. In certain cases, the input size matters greatly because the amount of memory on the CPU is much larger than that of the GPU, and certain test inputs failed with an `out of memory` error when run on the GPU but not on the CPU. In cases such as this, the runtime would schedule the kernel on the CPU so that it would actually complete.

Our dynamic scheduling algorithm depends strongly on whether or not there is contention on the CPU and GPU. We ran into some difficulty determining how "busy" each device was, as there are limited ways to find out this information, and it changes regularly. Unfortunately, we were unable to directly query the load on the CPU or GPU, nor were we able to determine even that there was a kernel running at all on the GPU. When a kernel is queued on a device, the device blocks other kernels from running until the first is complete. We recommend that future versions of the OpenCL SDK include the ability to query a device to determine whether a kernel is running, but at this point it is not possible. For our algorithm, we included a `checkIfGpuIsBusy()` function and seeded it with either "busy" or "not busy" depending on how we ran each individual experiment.

4. Analysis and Results

The initial baseline profiling of the selected applications yields varying patterns of runtime comparisons between the CPU and GPU, shown in Figures 3–7. For some applications, such as matrix multiplication, matrix transpose, and "embarrassingly parallel," there was a near constant factor between the CPU and GPU running times, as shown in the constant differences on the corresponding logarithmic plots. For the Binary Search, there appears to be a less clear difference in performance for smaller input sizes, but the CPU clearly dominates in performance by a 2x or 3x factor. Yet, in the Mersenne Twister application neither processor seems to dominate in performance over various input sizes. These results indicate that the role of input size in determining the preferred processor is application-dependent, and knowledge of the relationship to input size is most useful when a crossover point exists between the input size range that favors the CPU and that which favors the GPU. These results also demonstrate the relevance of a runtime decision for the subset of applications that exhibit similar performance between the CPU and GPU across a full range of input sizes.

Application	**Faster Processor (Baseline)**
Binary Search	Depends
Embarrassingly Parallel	GPU
Matrix Multiply	GPU
Matrix Transpose	GPU
Mersenne Twister	GPU

Table 1. Parallel Applications used in this paper. OpenCL was used for the parallel sections of code, which can run on both the CPU and GPU, without modification. *Faster Processor (Baseline)* refers to the processor for which the baseline data runs fastest. The *Embarrassingly Parallel* application used a very simple kernel that simply adds 1 to the values of an input vector.

A pseudocode implementation of the runtime processor selection is given in Figure 2. The code reads the historical running time data for the kernel with the associated data size from a file, and determines whether or not the CPU or GPU is busy. If the historical running time is less than the overhead of measuring CPU load, then the CPU is automatically selected. Next, the relative expected running time between the CPU and GPU is considered, and if the GPU will be faster and is not under contention, it is selected; otherwise, the CPU is selected. After each kernel finishes, the runtime for that kernel is updated in the historical data file and averaged with previous results.

Ultimately, the potential speedup of a dynamic runtime decision is the ratio of execution time when a static (development- or compile-time) decision is made to the execution time when the decision is made dynamically. This speedup may be a function of any of the above-described factors, such as program input size and processor contention.

Figure 3 shows the results with the Matrix Multiply application when the GPU is under contention. Because the GPU baseline times are almost always more than twice the CPU baseline times, even with contention the GPU still produces better results. Figure 5 shows similar results for the Embarrassingly Parallel application, in which case the GPU runs the kernel significantly faster than the CPU. This was the one application which also produced relatively long running kernels, up to 256 seconds on the CPU and 14 seconds on the GPU. For the largest input sizes, the GPU is unable to run the kernel because of memory overflows, and the algorithm therefore chooses to run the kernel on the CPU, in which case it is better to run the kernel slowly instead of re-

```
int chooseDevice(int inputDimensions[],bool dataInCache,
                 float choiceOverheadTime) {
  int useGpu = 1;
  int useCpu = 2;
  int contentionFactor = 2;

  GpuBaselineTime = getHistoricalGpuBaseline(inputDimensions);
  CpuBaselineTime = getHistoricalCpuBaseline(inputDimensions);

  if (CpuBaselineTime < choiceOverheadTime) return useCPU;

  bool cpuBusy = checkIfCpuBusy();
  bool gpuBusy = checkIfGpuBusy();

  if ((gpuBusy && cpuBusy) || (!gpuBusy && !cpuBusy))
     if (GpuBaselineTime < CpuBaselineTime) return useGPU;
     else return useCPU;
  else if (gpuBusy && (GpuBaselineTime *
                       contentionFactor < CpuBaselineTime)
     return useGPU;
  else if (gpuBusy && (GpuBaselineTime *
                       contentionFactor >= CpuBaselineTime)
     return useCPU;
  else if (cpuBusy && (CpuBaselineTime *
                       contentionFactor < GpuBaselineTime)
     return useCPU;
  else return useGPU;
}

void updateHistory(deviceType, inputDimensions[],
                   newKernelTime){
  readHistoryFromFile(deviceType,inputDimensions[],
                      &historicalTime,&historyCounter);
  historicalTime = (historicalTime * historyCounter
                    + newKernelTime) / ++historyCounter;
  writeHistoryToFile(deviceType,inputDimensions[],
                     historicalTime,historyCounter);
}
```

Figure 2. The dynamic decision pseudocode.

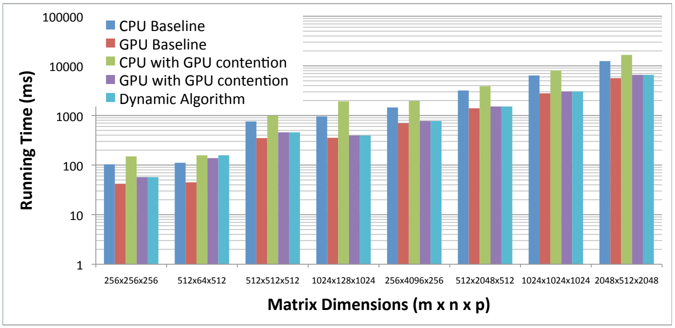

Figure 3. Matrix Multiply results when GPU is under contention. The dynamic algorithm chose correctly for all cases except for $512 \times 64 \times 512$, where the baseline GPU measurement was less than $2x$ the baseline CPU measurement.

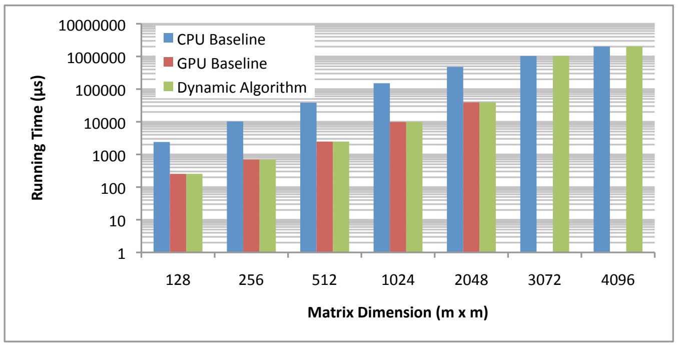

Figure 4. The Matrix Transpose baseline, with a simple dynamic choice without contention. Because of the algorithm used in the OpenCL code, when the matrix dimension reached 3072 and higher, the GPU reported an `Out of Memory` error. In those cases, the dynamic algorithm correctly chose to run the kernel on the CPU.

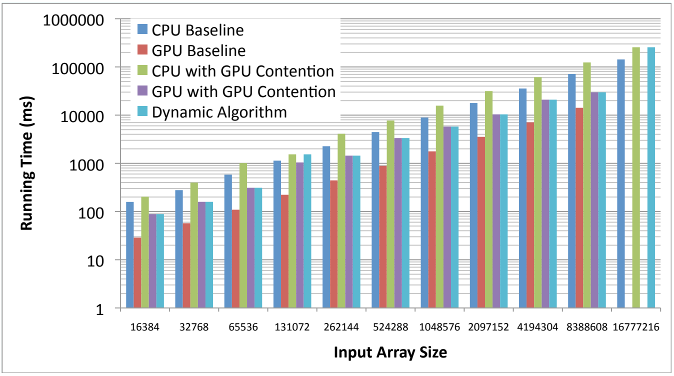

Figure 5. The Embarrassingly Parallel results. The Embarrassingly Parallel application simply takes an array of values and increments each value. The GPU runs consistently faster, even with GPU contention. The dynamic algorithm chose correctly in all cases where the GPU baseline was $2x$ greater than the CPU baseline. For the largest input array size, the dynamic algorithm recognized that the array would not fit in the GPU memory, and correctly chose to run the kernel on the CPU.

ceiving an out of memory error by trying to run on the GPU. Figure 4 demonstrates how the algorithm compensates for the out of memory error as well.

Figure 6 demonstrates what happens when historical data is taken into consideration. In the first run, with CPU contention, the algorithm chooses incorrectly for the three smallest input values. However, after the historical data is updated with the new running times, the algorithm can correctly chooses to run all the kernels on the best processor.

In the case of processor contention, it would be best to make a decision based on the amount of contention each device is currently experiencing. This is not an easy task, although the operating system could make a good approximation based on the number of processes it is managing. In order to demonstrate that even a small amount of contention information can lead to a suitable decision, we decided on a very simple metric: if the GPU is busy and the GPU baseline is greater than $2x$ faster than the CPU baseline for a given application, run the kernel on the GPU, otherwise run the kernel on the CPU. A similar calculation is made if the CPU is under contention. Figures 3 and 5 shows that this metric is appropriate in most cases, and Figure 6 shows that when this

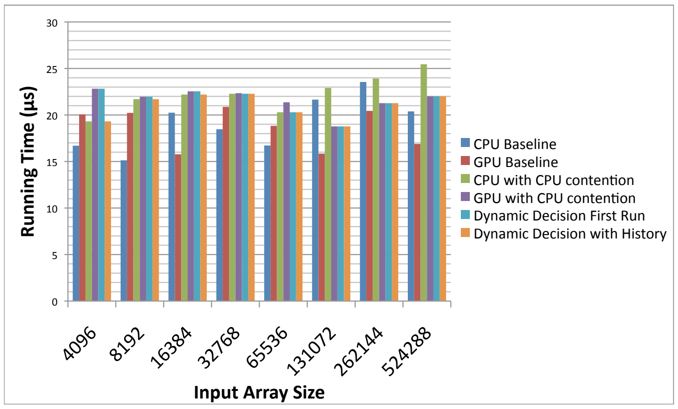

Figure 6. Binary Search results with CPU under contention. For small values of input array size, the GPU is hindered by data transfer onto the device. The dynamic algorithm did not choose correctly for the lowest three input array sizes on the first run, but on the second run the algorithm used the history to correctly choose the best device.

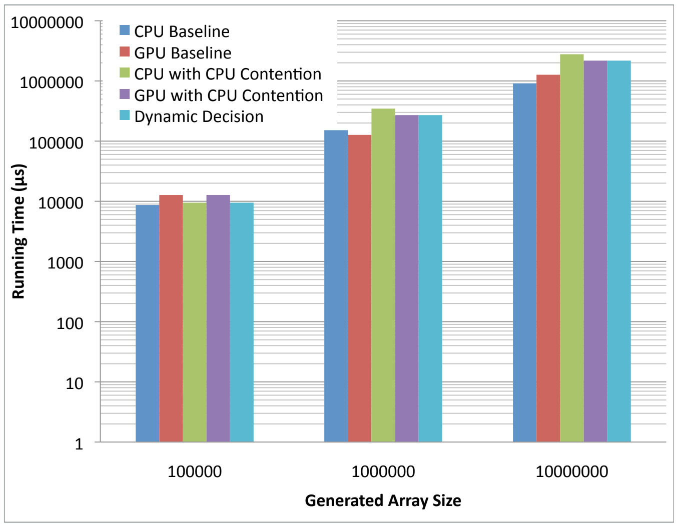

Figure 7. The Mersenne Twister results, with CPU contention. When the CPU is under contention, the dynamic algorithm makes the correct decision in each case.

metric is not appropriate, historical data can be used in subsequent kernel runs to adjust and make the appropriate decision. Figure 7 demonstrates that for the Mersenne Twister application with CPU contention, there are times when running on the CPU is still preferable to running on the GPU, and the algorithm chooses correctly in these cases.

In the case of the GPU, measuring contention is limited. The most knowledge a single application can obtain is whether the kernel queue is blocked waiting for the GPU, indicating the GPU is already running some other kernel. If an application runs with comparable performance on either processor, it could conceivably enqueue its kernel for the GPU, query whether the queue was blocked, and if so, run the kernel on the CPU. Unfortunately, this technique does not currently work, due to an apparent bug in OpenCL that enforces serialization of all kernels. That is, if a kernel is enqueued for the GPU and then the CPU, the CPU version will not execute until the GPU version runs to completion.

5. Conclusions and Future Work

Parallelized applications could experience a speedup, in an average sense, by dynamically selecting their target device for execution at runtime, rather than statically at compile-time. Input size, processor contention, and historical running time are variables of interest in making this runtime selection, and processor contention, while difficult to directly measure, can be extremely important to the dynamic decision.

An intelligent runtime decision, particularly with regard to contention, would require knowledge of other active kernels' expected execution times. Future work might focus on a more centralized, common kernel scheduler to achieve the best performance for the whole set of active kernels. In addition to performing the decision-making algorithm in the runtime environment, it would would also be beneficial if the operating system (with more contention data available) takes part in the scheduling decision. Additionally, the sample applications were tested with an input size up to the memory limit of the GPU. Yet applications with larger data sets could be refactored to use multiple kernel calls, incurring with associated overheads of transferring data between main memory and the GPU. Future work should explore the effects on performance of splitting a task over multiple kernel calls.

References

[1] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 29–40, Ischia, Italy, 2006. ACM.

[2] V. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 19–33. Springer, 2008.

[3] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 45–55, New York, NY, USA, 2009. ACM.

[4] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. *Heterogeneous Computing Workshop*, 0:30, 1999.

[5] H. Oh and S. Ha. A Static Scheduling Heuristic for Heterogeneous Processors. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, page 577. Springer-Verlag, 1996.

[6] D. Shelepov and A. Fedorova. Scheduling on heterogeneous multicore processors using architectural signatures. In *Proceedings of the Workshop on the Interaction between*

Operating Systems and Computer Architecture, 2008.

[7] H. J. Siegel, H. G. Dietz, and J. K. Antonio. Software support for heterogeneous computing. *ACM Computing Surveys*, 28(1):237–239, 1996.

[8] G. Sih and E. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE transactions on Parallel and Distributed systems*, 4(2):175–187, 1993.

[9] Y. YuHai, Y. Shengsheng, and B. XueLian. A new dynamic scheduling algorithm for real-time heterogeneous multiprocessor systems. In *Intelligent Information Technology Application, Workshop on*, pages 112–115, Dec. 2007.