Java with Auto-Parallelization on Graphics Coprocessing Architecture

Guodong Han, Chenggang Zhang, King Tin Lam, and Cho-Li Wang
Department of Computer Science
The University of Hong Kong
Hong Kong, China
{gdhan, cgzhang, ktlam, clwang}@cs.hku.hk

***Abstract*—GPU-based many-core accelerators have gained a footing in supercomputing. Their widespread adoption yet hinges on better parallelization and load scheduling techniques to utilize the hybrid system of CPU and GPU cores easily and efficiently. This paper introduces a new user-friendly compiler framework and runtime system, dubbed Japonica, to help Java applications harness the full power of a heterogeneous system. Japonica unveils an all-round system design unifying the programming style and language for transparent use of both CPU and GPU resources, automatically parallelizing all kinds of loops and scheduling workloads efficiently across the CPU-GPU border. By means of simple user annotations, sequential Java source code will be analyzed, translated and compiled into a dual executable consisting of CUDA kernels and multiple Java threads running on GPU and CPU cores respectively. Annotated loops will be automatically split into loop chunks (or tasks) being scheduled to execute on all available GPU/CPU cores. Implementing a GPU-tailored thread-level speculation (TLS) model, Japonica supports speculative execution of loops with moderate dependency densities and privatization of loops having only false dependencies on the GPU side. Our scheduler also supports task stealing and task sharing algorithms that allow swift load redistribution across GPU and CPU. Experimental results show that Japonica, on average, can run 10x, 2.5x and 2.14x faster than the best serial (1-thread CPU), GPU-alone and CPU-alone versions respectively.**

I. Introduction

General-purpose graphics processing units (GPGPUs or GPUs for short) with hundreds to thousands of cores are leading to a more cost-effective and energy-efficient alternative to traditional CPUs for high-performance computing. Programming on a heterogeneous NUMA architecture, composed of multicore CPUs and GPUs, is inherently difficult. To exploit GPU resources generally requires application code porting using APIs like CUDA [1], adding extra burden to programmers.

In this work, we introduce a new compiler and runtime platform, called *Japonica (Java with Auto-Parallelization ON graphIcs Coprocessing Architecture)*, to make heterogeneous many-core programming more productive. Japonica enables a Java program to scale transparently on a GPU-based heterogeneous system. With transparent runtime support, application developers can utilize both CPU and GPU resources seamlessly while sticking to an idiomatic Java programming model (except adding a few annotations around selected loops). We adopt Java as the target language for unifying CPU and GPU programming in view of its popularity. Bridging the way for Java to transparent GPU computing is an important milestone welcoming a wide spectrum of legacy applications aboard GPU. Japonica can help get them run on the heterogeneous many-core machines effortlessly. Our methodology is to dynamically detect and translate the data-parallel parts (loop bodies) of the Java code to CUDA kernels for GPU execution. With computation-intensive components offloaded to GPU cores, the slowness of Java compared to C/C++ is no longer a concern.

Syntactically, Japonica can be said as a Java port of OpenACC [2]—latest programming standard using compiler directives for GPU computing. Performance-wise, Japonica approaches the optimal by the trinity of profiling, optimistic GPU execution and task scheduling. Based on our previous work GPU-TLS [3], a GPU-aware TLS execution engine, we add profiling support to the system for tracking the data dependency density [4] across iterations, which provides useful insights into the potential for optimistic parallelization and algorithmic work scheduling. With the knowledge about specific data dependencies, the scheduler can decide on the execution model for each task: sequential, speculatively parallel (TLS) or truly parallel.

In summary, the contributions of our work are two-fold:

- To our best knowledge, this work is the first Java port of OpenACC. We made the first functional Java-based code transformation framework for automatically generating multiple Java threads to run on CPU, CUDA kernels to run on GPU, and the code to profile the GPU execution.
- With GPU-tailored TLS and privatization support, our runtime can saturate the GPU with loops of moderate levels of true (or false) dependency for the best execution efficiency. We also design two profile-guided task scheduling schemes, namely *task sharing* and *task stealing*, to distribute loop chunks onto both CPU and GPU in a way to best balance their usage.

The rest of this paper is organized as follows. Section II presents the overall system design of Japonica. The process of code translation and static analysis are explained in Section III. In Section IV, we suggest how to parallelize DOALL loops and loops of modest dependency density using GPU-TLS. Section V presents the two task scheduling schemes. We discuss related work in Section VI and conclude in Section VII.

II. System Design of Japonica

Figure 1 depicts the overall structure of our system. The compile-time components include the *code translator*,

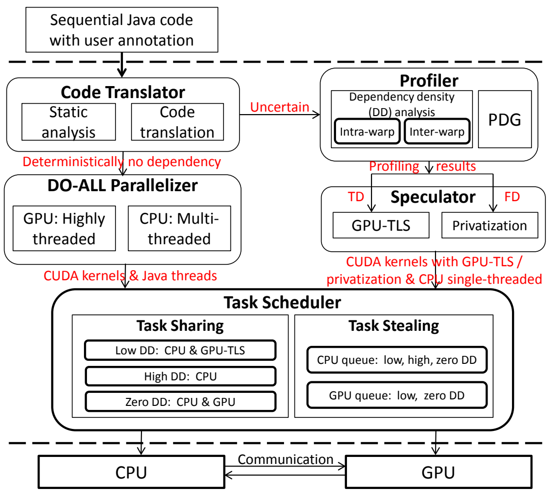


Figure 1: System Overview of Japonica

profiler, *DOALL parallelizer* and *speculator* while the *scheduler* is a run-time component balancing the workload on the heterogeneous platform. The compile-time components take annotated loops as input, and parallelize them via both static analysis and dynamic profiling. The run-time scheduler distributes the parallelized loops to all CPU and GPU cores according to the data dependency profile.

- **Code Translator**: it statically analyzes and counts data dependencies in the target loops, marking those loops carrying deterministic dependencies (detected by static analysis) and dynamic dependencies (determinable only at runtime) which need further analysis by the profiler. Necessary API functions for profiling are also inserted at this stage.
- **Profiler**: it gathers the dynamic information by executing the loops marked by the code translator on GPU in parallel, and performs intra-warp and inter-warp memory access dependency analyses at run-time. After dependency profiling, a quantitative model is used to compute the dependency density. Data-flow and control-flow dependencies between loops are also represented by the *program dependency graph (PDG)* with which the scheduler can flawlessly exploit task-level parallelism.
- **DOALL Parallelizer**: loops with zero loop-carried dependency can be parallelized by this component. It creates parallel code versions—CUDA kernels on GPU and Java threads on CPU—for all the DOALL loops deterministically found by static analysis. Data communication functions are injected and optimized to remove cyclic communication.
- **Speculative Execution Engine** (abbrev. **Speculator**): after profiling, the speculator supports speculative execution of loops with moderate *true dependency (TD)* and privatization of loops carrying only *false dependency (FD)* on the GPU by inserting function calls to our lightweight GPU-TLS library [3] which detects and recovers mis-speculation.
- **Task Scheduler:** it provides two dynamic schemes: *task sharing* and *task stealing*. The task sharing scheme divides the workload of the same task X across CPU and GPU according to their computational capabilities and X's dependency density. The task stealing scheme resembles a classical master-slave model: CPU and GPU have their respective task queues to await tasks from the scheduler; the one emptying its queue sooner can steal tasks from the other side. The scheduler distributes tasks according to the inter-task dependency derived from the PDG.

III. Code Translation

The code translator (see Figure 1) is a key component of our system. It includes static analysis of data dependency for user-annotated loops, and code generation for profiling on GPU. We implement our code translator based on JavaR [5], a prototype Java restructuring compiler that can be used to "externalize" the implicit parallelism in a Java program by means of multithreading.

A. Static Analysis

The goal of static analysis is to resolve as many memory accesses as possible and to reduce the overhead of dynamic profiling. Irresolvable memory accesses or accesses that may conflict with one another will be further profiled by inserting necessary functions. Based on different access patterns, each variable in the annotated for loop can be classified as one of the following: *live-in*, *live-out* and *temp*. The *temp* variable is declared inside the for-loops and cannot be accessed from outside. *live-in* and *live-out* variables are both declared outside the loop; they differ only in whether the variable will be updated in the loop. We implement the variable classification based on the Abstract Syntax Tree (AST) generated by JavaR. Along the AST tree traversal, variables declared under the for-loop node are classified as *temp* variable; variables appeared at the left side of an assignment statement are *live-in* variables; all other variables which only appeared at the right side of an assignment statement are *live-out* variables. The static analysis follows these rules: (1) we compress the memory accesses into a linear constraint in terms of loop iteration ID. If the memory access cannot be compressed and exists on the *live-out* variables list, it will be marked for profiling; (2) we check all pairs of *live-out* variables to examine the possible write-after-write conflicts; (3) we check all pairs comprising a *live-out* variable and a *live-in* variable to evaluate the possible read-write conflicts; (4) all the possible data conflicts will be further examined in the profiling phase.

B. Code Generation

Loops with uncertain dependency (marked in static analysis) are transformed into CUDA kernels to be profiled on GPU. To ease static analysis and to make the programming framework user-friendly, we retain JavaR's annotation approach to identifying implicit parallelism in the program, and modify it to conform to the OpenACC standard [2]. The user first has to identify the target for-loops which need parallelization, and marks them with

Table I: Main clauses in user annotations

Clauses	Description
parallel	Start parallel execution on the heterogeneous platform
private(list)	A copy of each variable in list is allocated for each execution element
copyin(list)	Allocate the data in list on the GPU and copies the data from the host to the GPU when entering the loop
copyout(list)	Allocates the data in list on the GPU and copies the data from the GPU to the host when exiting the loop
create(list)	Allocate the data in list on the GPU, but do not copy data between the host and device
threads(n)	Define the number of threads that must be used for parallel execution
scheme(s)	Choose the task scheduling scheme: *sharing* or *stealing*. Default scheme is *sharing*

annotations in Table I. The declaration of annotation on each for-loop follows the format:

$$/ * \mathit{acc\ parallel}\, [\mathit{clause}\, [],\ \mathit{clause}\, []...] * /$$

Users could define the specific array size by adding relative clause. The parameter for the memory allocation and data management clauses has the format: *arr*[*low*:*high*], which means the elements in array *arr* starting from *low* and ending at *high* will be handled. For defined arrays in the annotation, the translator would insert communication API calls into the host function. For instance, *copyin*(*arr*[1 : 1024]) indicates array elements from *arr*[1] to *arr*[1024] will be allocated and copied to GPU. If users do not define the specific arrays which need transformation in the annotated for-loop, our code translator could automatically generate necessary data movement APIs for the *live-in* and *live-out* varaibles generated by static analysis.

The loop index will also be remapped to the corresponding CUDA thread ID; loop bodies are transformed into kernel bodies as is the case of JavaR transformation. After annotated for-loops are completely translated to CUDA kernels and necessary data communication calls are inserted, the original loops will be replaced by calls to invoke the generated kernels through JNI.

IV. DOALL Loop Parallelization and GPU-TLS

Loops determined to be dependency-free during the static analysis phase or profiling phase could be safely parallelized without any software protection by our system. Similar to code translator, the DOALL parallelizer will generate transformed versions to be used in the task scheduling phase: multithreaded Java for CPU and CUDA kernels for GPU. Necessary communication calls and invoking functions are inserted.

To exploit dynamic parallelism in loops with moderate data dependency density on GPU, we proposed a software Thread Level Speculation (TLS) library named GPU-TLS [3]. GPU-TLS adopts an *incremental* solution dividing the target loop into several sub-loops and each sub-loop is coupled with a GPU kernel. The speculative execution of a GPU kernel has four phases: *Speculative Execution (SE)*, *Dependency Checking (DC)*, *Commit* and *Mis-speculation Recovery*. In the first phase, GPU executes the iterations in parallel as if there were no cross-iteration dependencies. During the execution, each thread buffers the possibly unsafe memory updates instead of updating the main memory. Some meta-data are taken around memory accesses to aid later mis-spceulation checks. The DC phase determines if the speculation is successful. With the memory access tracking metadata, we can tell whether the speculative execution has violated some inter-iteration dependencies. This phase aims to find if there are dependency violations and where they happen, if any (i.e. which threads have fallen in violation). For those threads not found to have violated dependencies, the commit phase copies their buffered memory updates to the global memory.

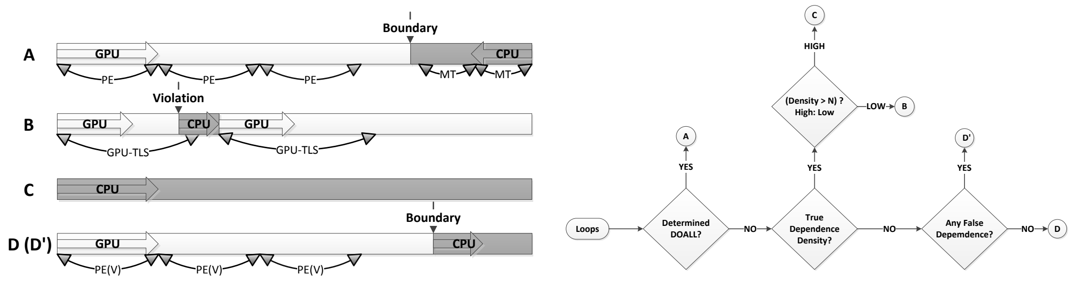
V. Profile-Guided Task Scheduling

In a heterogeneous system, CPU and GPU has disparate computational capabilities. Most static schemes distribute workload based on the raw peak performance of CPU and GPU, which are however suboptimal since the practical performance of CPU and GPU varies significantly across applications. Therefore, we aim at developing more intelligent and dynamic distribution schemes.

A. Task Sharing Scheme

We propose a novel preference-based solution by setting a global *boundary* for *cooperative* execution on CPU and GPU. Iterations before the boundary are preferential to be executed on GPU and movement of related data to the GPU will be done in advance and asynchronously with the kernel execution to avoid cyclic communication [6] and to hide some latency. Meanwhile, the iterations beyond the boundary are more suited to the CPU. The left part of the data set is divided into uniform chunks of moderate size, which are executed on GPU in an ascending order, while the right part is executed on CPU in a descending order. We use the following formula to represent the boundary value $\frac{C_g * F_g}{C_g * F_g + C_c * F_c}$, where $C_{c,g}$ is the core count and $F_{c,g}$ is the frequency value. Our empirical experiments prove that in most cases this value could guarantee sufficient data for GPU computation and no extra data transfer.

Figure 2 (a) shows the execution model for loops with different TD densities and figure 2 (b) shows the execution workflow. Loops determined as DOALL by static analysis are executed in mode A in Figure 2 (a). Mode A distributes data to GPU for Parallel Execution (PE) there while executing the rest on CPU by means of multithreading (MT). Other loops will pass through the profiler to compute the dependency density. The profiled loops with low density will be executed in mode B: execute on GPU in parallel with the help of our GPU-TLS. When violation is found, the scheduler forwards control to CPU and detects whether the following several warps of threads contain TD in the profiling results. If not, the scheduler launches another kernel from the violating warp to continue execution on GPU. Otherwise, these warps should be executed on CPU sequentially and detection is repeated after execution finishes. For loops of high TD density (implying that parallel execution cannot benefit performance-wise), they are dispatched to CPU for sequential execution (mode C). Profiled loops without TDs



(a) Execution models (b) Workflow diagram

Figure 2: Our Task Sharing Scheme Design

will be checked for FDs and employ mode D (D'). Since FD does not limit concurrency, we use Parallel Execution (with Variable privatization), PE(V), to handle loops with FD. The privatized variables are only updated after all the iterations finish execution and data are copied back to the host memory. However, since we check TDs on GPU (lock-step SIMD), there may exist possible TDs when executing on CPU in parallel. So mode D processes the data on CPU sequentially and the boundary statically divides the CPU and GPU data sets.

Algorithm 1: Task Stealing Scheme Distribution

```
1  $GPUQ \leftarrow \emptyset, CPUQ \leftarrow \emptyset$;
2  while $jobPool \neq \emptyset$ do
3      $taskSet \leftarrow$ getTasks($jobPool$);
4      while $taskSet \neq \emptyset$ do
5          $task \leftarrow$ getTask($taskSet$);
6          push $task$ to worker queue;
7      if $GPUQ \neq \emptyset \wedge (task \leftarrow$ steal($CPUQ$)$) \neq 0$ then
8          push $task$ to $GPUQ$;
9      if $CPUQ \neq \emptyset \wedge (task \leftarrow$ steal($GPUQ$)$) \neq 0$ then
10         push $task$ to $CPUQ$;
11     wait until all tasks in $taskSet$ are done;
```

B. Task Stealing Scheme

We propose a workflow-aware task stealing scheme (See Algorithm 3). Inter-loop dependencies are analyzed to compose the PDG. Here a task or job refers to a loop and the PDG gives us a workflow. The tasks created by the parallelizing compiler form a job pool. Initially, the task scheduler gets a batch of data-independent tasks from the pool by topological sort (line 3). Then, the scheduler distributes the tasks to the worker queue, CPUQ or GPUQ, (in lines 5-6) according to the following rules: it is obligatory for loops with high TD density and loops without TD after profiling to be assigned to CPU and GPU respectively; loops with mediate TD density are suited to CPU while loops determined as DOALL at compile time are suited to GPU. After distribution, the scheduler will check if the GPU and CPU queues have tasks. If not, it steals a *preferential* task and push it onto the empty queue from another queue (line 7-10). The scheduler waits until all these topological sorts of tasks are done (implemented by adding synchronization at necessary points). During execution, when one worker finishes all its assigned tasks, it will steal a preferential task from the other queue if some tasks are waiting for execution there.

C. Selection Criteria of the Scheduling Schemes

Every time only one scheme can be used for each application. It is essential to choose a proper scheme to make efficient use of the computing resources on the heterogeneous platform. By the nature of the two schemes, task sharing is preferable for applications with heavy computations centralized in only one or few loops while task stealing is more suitable for those with computations evenly distributed across several data-independent loops.

VI. Performance Evaluation

A. Methodology

Our experiments were conducted on a platform equipped with one Intel Xeon X5650 CPU (running at 2.66 GHz) and one Nvidia M2050 GPU (Fermi). Table II shows the specific benchmarking applications. We compare the performance of the two scheduling schemes against the best available CPU or GPU implementations. For CPU multithreaded implementations, we set the number of threads as 16, which could achieve the best performance among different thread count configurations on average. We also reserve another two threads: one for managing GPU and the other for CPU multithreaded management. We take all the wall-clock time into consideration, which includes the time taken to transfer data between CPU and GPU when GPU kernels are launched. In the following experiments, all the applications are divided into two groups based on which scheduling scheme they adopt. Applications with various characteristics are evaluated with different strategies.

B. Evaluation of the Task Sharing Scheme

Performance of DOALL applications: Figure 3 shows the speedup of four DOALL applications, including GEMM, VectorAdd, BFS and MVT, using the task sharing scheme over 16-thread CPU version. All the four applications have a compute-intensive loop which is annotated for parallelization by our system. During the static analysis phase, all the loops are marked as deterministic DOALL and executed in mode A. We compare the performance among the CPU 16-thread version, GPU-only version, a simple cooperative version (50% workload to CPU and the other 50% to GPU) and our task sharing scheme.

Table II: Summary of benchmarks used in our experiments

Benchmark	Origin	Description	Input Problem Size (n)	Serial Time(n=1)	Scheme
GEMM	PolyBench	Dense matrix multiplication	n*512*512 matrix	80597.8 ms	Sharing
VectorAdd	CUDA SDK	Vector addition	n*2048*2048 elements	3548.6 ms	Sharing
BFS	Rodinia K	Breadth First Search	n*65536 nodes	1423.7 ms	Sharing
MVT	PolyBench	Matrix transpose	n*2048*2048 matrix	379.7 ms	Sharing
Guass-Seidel	PolyBench	Iterative method	n*512 matrix	1139.37 ms	Sharing
CFD	Rodinia	Computational fluid dynamics	n*4096 edges	199.411 ms	Sharing
Sepia	Merge	Modify RGB value	n*2048*2048 image elements	334.8 ms	Sharing
BlackScholes	Intel RMS	European option pricing	n*5120 options	121.3 ms	Sharing
BICG	PolyBench	Bi-Conjugate gradient method	n*2048*2048 matrix	19.2 ms	Stealing
2MM	PolyBench	2 Matrix multiplications	n*256*256 matrix	26414.0 ms	Stealing
Crypt	Java Grande	IDEA encryption/decryption algorithm	n*1024*1024 text elements	2231.5 ms	Stealing

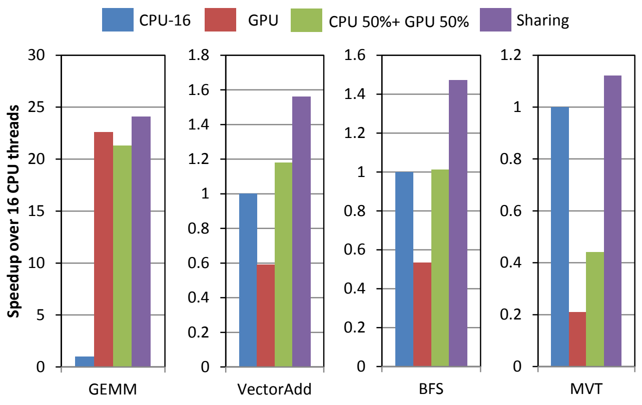


Figure 3: Speedup by task sharing for DOALL apps

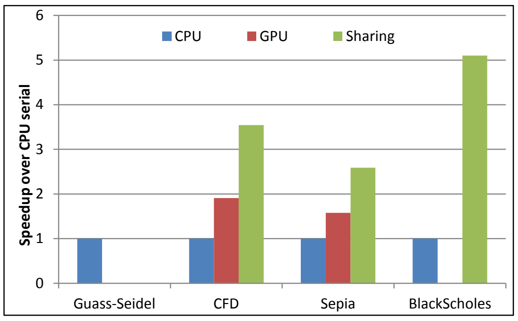


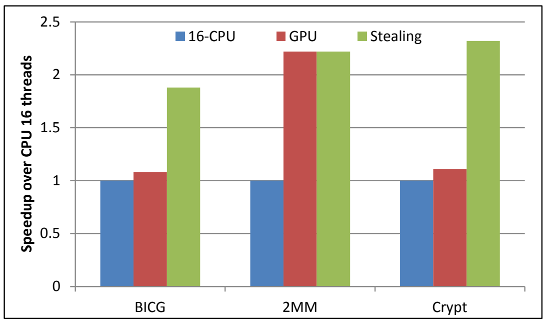
Figure 4: Speedup by task sharing for DOACROSS apps

The GEMM (dense matrix multiplication) is a famous CPU-bound application. Since the performance of GPU exceeds the 16-thread CPU version too much, the sharing scheme does not contribute to a noticeable speedup over the GPU-only version. Furthermore, the GPU has already finished computing on its assigned data set, delimited by the boundary, before the CPU finishes its work, and the system will transfer more data to GPU, bringing about extra overhead. The sharing scheme achieves much better performance in VectorAdd, BFS and MVT. For Vectoradd, it achieves a 1.56 times better speedup than the 16-thread CPU version, 2.64 times better speedup than the GPU-only version and 1.32 times better speedup than the simple cooperative version. For BFS, our sharing scheme respectively attains 1.12, 5.33 and 2.55 times better speedups than the other three counterparts. In the MVT evaluation, it achieves 1.47, 2.75 and 1.46 times better performance.

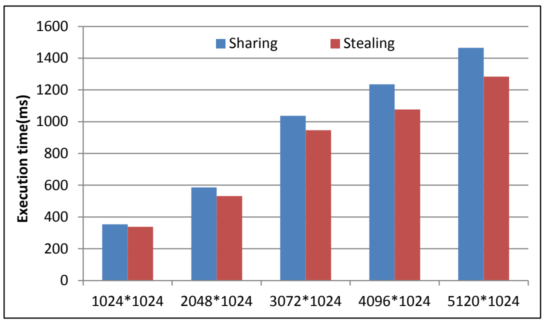
Performance of DOACROSS applications: The evaluation of loops with uncertain data dependency density is shown in Figure 4. Guass-Seidel is a famous iterative method containing high data dependency, so our system distributes all the workloads to CPU (mode C). CFD and Sepia are two applications containing non-deterministic dependencies. However, after profiling, it turns out there is no true dependency; but false dependency exists. Therefore both applications are dispatched in mode D. Since part of the workload is offloaded to CPU, the sharing scheme achieves a better performance: 3.55 times better speedup than the serial CPU-only version and 1.86 times better speedup than the GPU-only version in CFD; 2.59 and 1.64 times better speedup than the CPU-only and GPU-only versions respectively. Another application to be evaluated is Blackscholes, which is used for European option pricing, and the profiler detects little true dependency in it (the data dependence value measured in our experiment is about 0.012), therefore, our system uses GPU-TLS (mode B) to accelerate the process, and speedup over sequential execution is noticeable—5.1 times better.

C. *Evaluation of the Task Stealing Scheme*

We show the performance of task stealing in Figure 5. The BICG method contains two independent and deterministic DOALL loops with similar workload. We rewrite the BICG method and divide each loop into four subloops evenly. According to the distribution rules, at the beginning, all the eight loops are assigned to the GPU queue. Then our scheduler finds that the CPU queue is empty and moves one loop to it from the GPU queue. After CPU finishes the loop in its queue, it will steal another loop from the GPU queue. Eventually, the CPU finishes 62.5% workload of all subloops and achieves 1.88 and 1.82 times better performance than the CPU 16-thread and GPU-only versions. Another program we evaluated is the 2MM application. There are two deterministic DOALL loops; however, the second loop depends on the output of the first. Therefore, our task stealing scheme divided the two loops into two task batches and processed the batches sequentially. As the two loops are DOALL, they are assigned to GPU for execution. Here the GPU contributes all the computations. Crypt contains two DOALL loops: one for encryption and another for decryption.



(a) Speedup



(b) Comparison with task sharing scheme

Figure 5: Performance of applications scheduled by the task stealing scheme

The decryption process depends on the encryption output. Like BICG, we divide each loop into eight subloops and eventually get 16 dependent loops. Following our scheduler's guidance, all the loops are efficiently executed on GPU and CPU. The speedup is shown in Figure 5(a), task stealing achieves 2.32 and 2.09 times better speedup over CPU-only and GPU-only versions. Crypt is also used to compare the performance of task stealing with task sharing to evaluate its effectiveness in applications with several loops containing close workload. The results in Figure 5(b) show that task stealing is more efficient than task sharing for Crypt.

VII. Related Work

Prospector [7] proposed a stride-based dependency profiling algorithm to advise how to parallelize the identified sections. The algorithm, called SD^3 [8], exhibits stride patterns and computes data dependencies directly in a compressed format. However, for irregular accessing, which lacks a regular stride, the overhead of computing stride patterns may dramatically increase. GRace [9] proposed a mechanism to detect races in GPU programs. It combines static analysis with a dynamic checker for logging and analyzing information at runtime. Though with modest overhead, GRace only focuses on data race detection and leaves the burden of parallelization to programmers (it simply reports where races may happen). Besides this, coarse-grained data races could not reflect the real data dependency which is a critical factor of parallelization. Qilin [10] proposed adaptive mapping of computations to processing elements on heterogeneous multiprocessors. Qilin predicts application execution times on CPU and GPU based on history analysis in order to define a fractional variable for assigning workloads. Similar to Qilin, Scogland, et al. [11] proposed a heterogeneous task scheduling scheme for OpenMP-based applications. Iterations are distributed according to the computational capacity ratio of CPU to GPU. However, once the ratio is defined, the workload distribution could not be changed during execution.

VIII. Conclusion

In this paper, we design an automatic Java loop parallelization and task scheduling solution for GPU-based heterogeneous many-core architectures. Thanks to the support of GPU-TLS, privatization and data dependency profiling, the Japonica system supports parallel execution of not only deterministic DOALL loops but also the loops containing modest level of non-deterministic data dependencies. We implemented two efficient task scheduling schemes to balance the overall workload across CPU and GPU in different situations. The concept of setting a boundary in the task sharing scheme could reduce the overhead caused by cyclic communication and task dispatching. The task stealing scheme can readily achieve balanced workload among the data-flow independent loops.

Acknowledgment

This research is supported by Hong Kong RGC grant HKU 7180/11E.

References

[1] NVIDIA, "CUDA parallel computing platform," 2012, http://www.nvidia.com/object/cuda_home_new.html.

[2] "The OpenACC application programming interface," 2011, http://www.openacc-standard.org/.

[3] C. Zhang, G. Han, and C.-L. Wang, "GPU-TLS: an efficient runtime for speculative loop parallelization on GPUs," ser. CCGrid-13, 2013 (in press).

[4] C. von Praun, R. Bordawekar, and C. Cascaval, "Modeling optimistic concurrency using quantitative dependence analysis," ser. PPoPP '08, 2008, pp. 185–196.

[5] A. J. C. Bik, J. E. Villacis, and D. B. Gannon, "JavaR: a prototype Java restructuring compiler," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1181–1191, 1997.

[6] T. B. Jablin *et al.*, "Automatic CPU-GPU communication management and optimization," *SIGPLAN Not.*, vol. 46, no. 6, pp. 142–151, 2011.

[7] M. Kim, H. Kim, and C.-K. Luk, "Prospector: a dynamic data-dependence profiler to help parallel programming," ser. HotPar '10, 2010.

[8] M. Kim, H. Kim, and C. Luk, "SD^3: a scalable approach to dynamic data-dependence profiling," ser. MICRO-43, 2010, pp. 535–546.

[9] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal, "GRace: a low-overhead mechanism for detecting data races in GPU programs," ser. PPoPP '11, 2011, pp. 135–146.

[10] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," ser. MICRO-42, 2009, pp. 45–55.

[11] T. Scogland, B. Rountree, W. Feng, and B. de Supinski, "Heterogeneous task scheduling for accelerated OpenMP," ser. IPDPS '12, 2012, pp. 144–155.