Marcin PIETROŃ, Michał KARWATOWSKI, Kazimierz WIATR
AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY, 30 Mickiewicza Ave., 30-059 Krakow, Poland
ACC CYFRONET AGH, 11 Nawojki St., 30-950 Krakow, Poland

The Java profiler based on byte code analysis and instrumentation for many-core hardware accelerators

Abstract

One of the most challenging issues in the case of many and multi-core architectures is how to exploit their potential computing power in legacy systems without a deep knowledge of their architecture. The analysis of static dependence and dynamic data dependences of a program run, can help to identify independent paths that could have been computed by individual parallel threads. The statistics of reusing the data and its size is also crucial in adapting the application in GPU many-core hardware architecture because of specific memory hierarchies. The proposed profiling system accomplishes static data analysis and computes dynamic dependencies for Java programs as well as recommends parts of source code with the highest potential for parallelization in GPU. Such an analysis can also provide starting point for automatic parallelization.

Keywords: virtual machine, CUDA, GPU, profiling, parallel computing.

1. Introduction

The main challenge of many and multi-core architectures is how to leverage their computing power for systems that have not been built with parallelism in mind. In recent years there have been considerable efforts in automatic parallelization, mostly relying on static program analysis. Some of these methods are used in modern compilers for automatic vectorization. Recently some other static analysis systems for source code analysis, like Barvinok counting based on Erhart polynomials and polyhedral model, have been developed. In our system, Banerjee, Range Test, Omega Test and data analysis algorithms are implemented as a static analysis. It should be noted that these algorithms neither extract parallelism in all potential situations nor measure accurate distribution of the data used in the analyzed source code. Therefore the dynamic dependence analysis is performed by appropriate code instrumentation and profiling process. In the presented system, the instrumentation is performed by just in time analysis of java bytecode based on a BCEL tool [1]. The appropriate code injection and bytecode analysis described in the paper allow building a data flow graph, verifying dependences and checking data accessing and reusing.

Recent years have seen considerable efforts in automatic parallelization, mostly relying on static program analysis to identify sections amenable for parallel execution. Therefore a lot of algorithms, libraries and system for the source code analysis have been built [2], [3], [4].

Our system concentrates on the analysis of a sequential source code for the GPU hardware architecture. GPGPU is constructed as a set of parallel multiprocessors with multiple cores each. The cores share an Instruction Unit with other cores in a multiprocessor. Multiprocessors have dedicated memory chips which are much faster than the global memory, shared for all multiprocessors. These memories are: a read-only constant/texture memory and a shared memory. The GPGPU cards are constructed as massive parallel devices, enabling thousands of parallel threads, which are grouped in blocks with the shared memory, to run. A dedicated software architecture-CUDA makes it possible to program GPGPU using high-level languages such as C and C++ [5]. CUDA requires an NVIDIA GPGPU like Fermi, GeForce 8XXX/Tesla/Quadro etc. This technology provides three key mechanisms to parallelize programs: thread group hierarchy, shared memories, and barrier synchronization. These mechanisms provide fine-grained parallelism nested within the coarse-grained task parallelism. Creating the optimized code is not trivial and thorough the knowledge of GPGPUs architecture is necessary to do it effectively. The main aspects to consider are the usage of the memories, efficient division of the code into parallel threads and thread communications. As it was mentioned earlier, constant/texture, shared memories and local memories are specially optimized regarding the access time. Therefore programmers should optimally use them to speedup the access to the data on which an algorithm operates. Another important thing is to optimize synchronization and communication of the threads. The synchronization of the threads between blocks is much slower than in a block. If it is necessary, it should be solved by the sequential running of multiple kernels.

Our research work concentrates on extracting the necessary information from the sequential source code to check its ability for adaption to the GPGPU architecture. It presents built-in rules which help generating some patterns of the kernel code based on the structure of a profiled code and the results of profiling. The main advantage of the tool, in comparison with others [8-11], is that it can profile the code online without throwing the trace and its analysis. The presented profiler can schedule the instance of instructions and the dependence between them for a polyhedral model which can be used for mapping the sequential code to GPU hardware.

2. The system architecture

Our tool consists of dynamic and static analysis. The static analysis is mainly based on well-known algorithms like GCD and Banarjee tests, Range Test and Omega tests, Barvinok library and some other static techniques for memory and data flow analysis based on integer domain. This paper focuses only on the dynamic analysis implemented in our system.

The dynamic analysis is performed by instrumenting the byte code. In scientific literature about binary instrumentation [6-8], there can be found several systems for profiling purposes. Most of them have been created for C language. The proposed profiling system is one of the first tools which enable adapting Java programs in GPU hardware accelerators (Fig. 1). The results of running the profiler can help transforming the chosen parts of the source code to the GPU kernel code. It concentrates mainly on checking the dependence and data reusing while executing the analyzed program. The system uses the BCEL library for parsing and instrumenting the source code. During parsing the byte code, the data flow graph is generated. The program loops are the parts of the source code where algorithms spend most of their execution time and are the sources of hidden parallelism. Therefore the data flow in the program loops should be monitored with great care. The dependences between primitive types in the loops are detected from the generated data flow graph. The instructions with array types are instrumented (Fig.2) with additional instructions and data structures which write information if an array element (with a specific subscript value) was read or written (left or right hand side value) and in which loop the iteration instruction is executed. This data can detect the dependence during execution of the program and the lengths of the dependence vectors. This methods can also compute a critical path of the analyzed program. The first data flow graph is generated during parsing the byte code then it can be expanded by profiling. Additionally, the histograms of read and write operations of the array elements are computed with the variances of the first, last elements and the widths of histograms of accessing array elements in all iterations of loops. The data distribution gained by the computed histograms and its variances allow defining the degree of data reusing between the iterations of loops and the type of data access (e.g. if it is coalesced, random

etc.). In the case of conditional instructions, static techniques are performed to detect if an instruction can have any influence on the data dependence inside the loop.

The main rules of the instrumentation process described above are as follows:
- instrumentation of array data read instructions (Fig.2),
- instrumentation of array data write instructions (Fig.2),
- instrumentation of array data read and write instructions for counting the number of accesses and standard deviation,
- instrumentation single variables read and write for counting the number of accesses.

After the static and dynamic analysis, the manual or automatic code generation can be performed. The source code in Java can be ported to the JCuda tool. The results gained from the static and dynamic analysis are the input for the rules that can be used for the manual or automatic generated JCuda source code. JCuda is a Java API for communicating with the GPU and invoking CUDA kernels. The main rules of adapting the analyzed source code to the graphic card hardware platform are described below:
- if the data is reused between the iterations (between the threads) this data should be transferred to the shared memory,
- data reused by only single iteration should be transferred to the local memory (registers),
- data which is reused, read only and without regular accesses should be allocated in the texture memory,
- common constant values used by the threads should be written to the constant memory,
- data with single access but without coalesced access should be transferred in a group in a coalesced manner to the shared memory and then read from this memory for further computing.

Pre-scheduling is made during the byte code parsing. During analyzing the byte code instructions on a stack machine, the system builds the data flow graph just in time. Each instruction receives its own number which describes when the instruction can be executed not to change the algorithm logic [12, 13]. The second schedule algorithm is run after the profiling process. The independent code should be mapped between the block threads (mapping the data to registers) and the whole blocks (mapping the data to shared memory) to minimize the communication between the blocks (if needed, the multi kernel invocation should be generated).

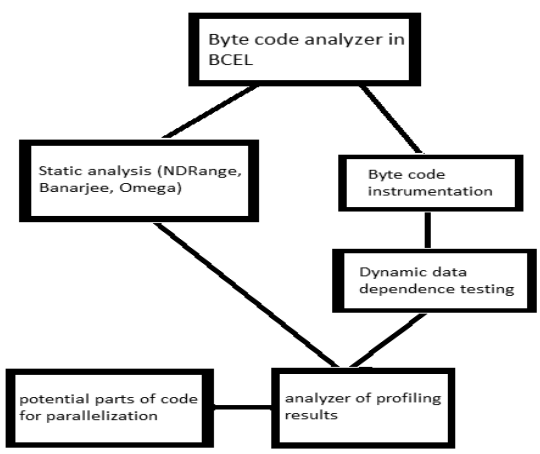

Fig. 1. Architecture of the Java profiler tool

The work allocated to the thread should be considered with the type of data accesses (minimizing the data conflicts) and the sum of the data needed for the block threads execution (limited size of the shared and local memories and minimized cost of the data transfer from the device memory). To adapt the analyzed source code, some basic loop transformations must be performed. The most important in the case of graphic cards is loop tiling which helps mapping the loops iteration between the blocks and assign the work and data to the threads. This process mainly works on integer domain. Therefore the tools for integer set operation can be used [3, 4].

It is worth noting that very often some memory access patterns during mapping should be optimized (avoiding bank conflicts in shared memories, coalesced access to global memory) [11]. Currently it is manually made in our application but the reports received from the profiler inform whether some conflicts or not optimal memory accesses exist in the analyzed source code.

```
for (int i = 1; i < 100; i++) {
        test_1[i] = 100;
        test_2[i] = test_1[i-1] + 10;
 }
```

Fig. 2. Example code for instrumentation

```
for (int i = 1; i < 100; i++) {
        test_1[i] = 100;
        test_1_mon[i] = i;
        test_2[i] = test_1[i-1] + 10;
        if (test_1_mon[i-1] < i) {
          dist_vectors[i-1] = i-test_1_mon[i];
        }
 }
```

```
27:  iconst_1
28:  istore          %6
30:  iload           %6
32:  bipush          100
34:  if_icmpge       #93
37:  aload_1
38:  iload           %6
40:  bipush          100
42:  iastore
43:  aload_3
44:  iload           %6
46:  iload           %6
48:  iastore
.
.
.
70:  if_icmpge       #87
73:  aload           %5
75:  iload           %6
77:  iconst_1
78:  isub
79:  iload           %6
81:  aload_3
82:  iload           %6
84:  iaload
85:  isub
86:  iastore
87:  iinc            %6      1
90:  goto            #30
```

Fig. 3. Instrumented code for RAW for the code from Fig. 2

3. Experiments and results

This section presents the results gained by running our tool on linear algebra operations. The matrix multiplication source code was taken as an input for profiling. After profiling, strong regular data reusing between the iteration (histograms) can be seen. The two external loops are parallelized and the reused data between the iterations is transferred (in a coalesced manner) to the shared memories. Tab.1 presents the results of matrix multiplication from the implementation based on profiling the data (without full elimination of bank conflicts).

Tab. 1. Results of matrix multiplication from the implementation based on profiling the data

size of matrix	GPU time, ms	CPU time (MKL BLAS), ms
256×256	0.4	6
512×512	4.3	24
1024×1024	34	158
2048×2048	285	956
4096×4096	2817	990

4. Conclusions and future work

The proposed methodology of profiling the Java source code has following advantages: it is based on the byte code therefore it is portable between different versions of Java, it can be adapted to different languages based on a java virtual machine e.g. Scala and with little modification rewritten for Python or Ruby languages. The important advantage is that the data analysis and its dependence can be run online. Therefore profiling can be run quite fast without throwing the trace.

The next step of further research will be to create a module for a partially automatic JCuda generator from the Java source code based on the polyhedral model generated by the static and dynamic analysis [6, 7].

This research is supported by the European Regional Development Program no. POIG.02.03.00-12-137/13 PL-Grid Core.

5. References

[1] BCEL tool - https://commons.apache.org/proper/commons-bcel/
[2] Baskaran M., Bondhugula U., Krishnamoorthy S., Ramanujam J., Rountev A., and Sadayappan P.: A compiler framework for optimization of affine loop nests for gpgpus. In ICS'08: Proceedings of the 22nd annual international conference on Supercomputing, pp. 225-234, New York, NY, USA, 2008. ACM.
[3] Verdoolaege S., Isl: An integer set library for the polyhedral model. Mathematical Software – ICMS 2010. Lecture Notes in Computer Science Series, vol. 6327. Springer, pp. 299-302.
[4] Verdoolaege S., Grosser T.. Polyhedral extraction tool. IMPACT 2012, Paris, France.
[5] NVIDIA CUDA Programming Guide 7.0. Nvidia Corporation.
[6] Bridges M.J., Vachharajani N., Zhang Y., Jablin T., and August D.I.: Revisiting the sequential programming model for the multicore era. Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 69-84, January 2008.
[7] Harris T. and Singh S.: Feedback directed implicit parallelism. In ICFP'07: Proceedings of the 12th ACM SIGPLAN international conference on functional programming, pp. 251-264, New York, NY, USA, 2007.
[8] Rul S., Vandierendonck H., and Bosschere K.D.: Extracting coarse-grain parallelism in general-purpose programs. In Proceedings of the 2008 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 281-282, Salt Lake City, Feb.2008.
[9] Thies W., Chandrasekhar V., and Amarasinghe S.: A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In MICRO'07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 356-369, Washington, DC, USA, 2007. IEEE Computer Society.
[10] von Praun C., Ceze L., and Cascaval C.: Implicit parallelism with ordered transactions. Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, pp. 79-89, New York, NY, USA, 2007, ACM.
[11] Baskaran M., Ramanujam J., and Sadayappan P.: Automatic c-to-cuda code generation for affine programs. In Proceedings of the International Conference on Compiler Constructions, number 6011 in Lecture Notes in Computer Science, pp. 244-263. Springer-Verlag, March 2010.
[12] Feautrier P.: Dataflow analysis of array and scalar references. International Journal of Parallel Programming, vol. 20, pp. 23-53, 1991.
[13] Feautrier P.: Some efficient solutions to the affine scheduling problem. International Journal of Parallel Programming, vol. 21, pp. 313-347, 1992.

Received: 21.04.2015 Paper reviewed Accepted: 02.06.2015

Marcin PIETROŃ, PhD

He received MSc degree in electronic engineering and in computer science in 2003 and PhD in 2013 from the AGH University of Science and Technology, Kraków, Poland. He currently works in Academic Computing Centre CYFRONET AGH and University of Science and Technology. His research interests include parallel computing, automatic parallelization and data mining.

e-mail: pietron@agh.edu.pl

Michał KARWATOWSKI, MSc, eng.

He received the BSc Eng. and MSc degrees in electronic engineering in 2013 and 2014 respectively from the AGH University of Science and Technology, Kraków, Poland. Currently PhD student. His research interests include usage of hardware accelerators in complex computations, control and energy efficient systems of small and big scale, mainly using Field-Programmable Gate Arrays.

e-mail: mkarwat@agh.edu.pl

Prof. Kazimierz WIATR, DSc, eng.

He received the MSc and PhD. degrees in electrical engineering from the AGH University of Science and Technology, Kraków, Poland, in 1980 and 1987, respectively, and the DSc degree in electronics from the University of Technology of Łódź in 1999. Received the professor title in 2002. His research interests include design and performance of dedicated hardware structures and reconfigurable processors employing FPGAs for acceleration computing. He currently is a director of Academic Computing Centre CYFORNET AGH.

e-mail: wiatr@agh.edu.pl