Journal of Theoretical and Applied Information Technology
15th December 2012. Vol. 46 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

GXBIT: COMBINING POLYHEDRAL MODEL WITH DYNAMIC BINARY TRANSLATION

[1]ZHANG KANG, [2]ZHOU FANFU AND [3]LIANG ALEI
[1]China Telecommunication, Shanghai, China
[2]Department of Computer Science and Engineering,
Shanghai Jiao Tong University, Shanghai 200240, China
[3]School of Software, Shanghai Jiao Tong University, Shanghai 200240, China
E-mail: [1]zhangkang@shtel.com , [2]zhoufanfu@sjtu.edu.cn , [3]liangalei@sjtu.edu.cn

ABSTRACT

During the last decade, polyhedral model has been widely used as a mathematical model for auto-parallelization, and in recent years, with the development of multi-core architecture, polyhedral model has been employed to transform sequential code to parallel code that can run simultaneously on different cores. At the same time, the rapid development of GPU makes CPU/GPU architecture become increasingly popular because of GPU's powerful parallel processing capabilities. However, we have no other methods of using GPU except for CUDA and Stream SDK, which are all based on explicit programming. Apart from this, there are two constraints for explicit programming: binary incompatibility among different GPUs as well as the cost of rewriting source code. Considering these constraints, we use polyhedral model and a dynamic binary translator to build a virtual execution environment: GXBit. GXBit is composed of analysis and execution stage, and the former one is the main focus of this paper. Analysis stage uses binary instrumentation and binary analysis to probe potential parallel parts (usually nested loop) of a binary executable and then polyhedral model is employed to detect whether there is data dependence or not among all iterations of a nested loop. Execution stage is discussed briefly in this paper. Although there are performance loss in binary translation, GXBit has an 8x speedup on average to X86 compute-intensive version through the result of running two applications taken from the CUDA SDK Sample and one test case from the UIUC Parboil Benchmark Suite.

Keywords: *Auto-Parallelization, Polyhedral Model, GXBit, Virtual Execution Environment*

1. INTRODUCTION

Modern computers not only have multiple cores, they are also equipped with one or more GPUs to implement the efficient data level parallelism support for some computation-sensitive application domains such as image processing, linear algebra and encryption. To exploit the GPU's power of computation, vendors provide C-like explicit programming environment to program on a certain GPU such as NVIDIA's CUDA [13] and AMD's Stream SDK, whichis the only way we can use the GPU recently.

However, there are two constraints for explicit programming: the cost of rewriting the source code as well as binary incompatibility [2, 3] (i.e.an application compiled under a specific architecture cannot run on a new one). So we need an effective method to resolve these problems.

Currently, there is already a method of using dynamic binary translator (DBT) to overcome the binary incompatibility among different architectures. However, DBT gets serious performance loss when we get rid of this problem..

Now that there are no better methods of modifying or optimizing DBT itself to improve its performance, we can put our focus on the applications executed on DBT. Many compute-intensive applications spend most of their time in nested loops and if we can accelerate these nested loops, the performance of DBT will be improved. During the last decade, the polyhedral model has been widely used as a mathematical model for auto-parallelization [4, 5, 7, 10]. If we can get nested loops in a binary executable and use polyhedral

Journal of Theoretical and Applied Information Technology
15th December 2012. Vol. 46 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

model to analyze the data dependence among all iterations of nested loops, we can make some transformations about these nested loops those don't have data dependence and put them to execute on GPU. So, in order to overcome the two problems above and improve the performance of DBT, we use polyhedral model and a dynamic binary translator of our own lab to build a virtual execution environment: GXBit.

The rest of this paper is organized as follows. In section 2, the architecture of GXBit is discussed. Implementation of constructing CFG and building polyhedral mode are presented in Section 3. Section 4 briefly discusses the mechanism of GXBit to execute those parallel parts on GPU and shows the performance evaluation. Related work and conclusion s are discussed in Section 5.

2. GXBIT

As the main aim of GXBit is to improve the performance of DBT, it is necessary for us to talk about Crossbit, which was designed as a resourceable and retargetable binary translation infrastructure.

As GXBit is the improved edition of Crossbit, most parts of them are same and there are three main differences between Crossbit and GXBit. First of all, the execution mode of GXBit is 2-phase, one for binary analysis and the other for execution. Secondly, to those parallel parts, GXBit has to start the special translator to translate them into target instructions that can be recognized by the corresponding GPU. Finally, the execution engine of GXBit is also different from Crossbit's because GXBit needs to execute those parallel parts on GPU. The architecture of GXBit is shown in Fig. 1.

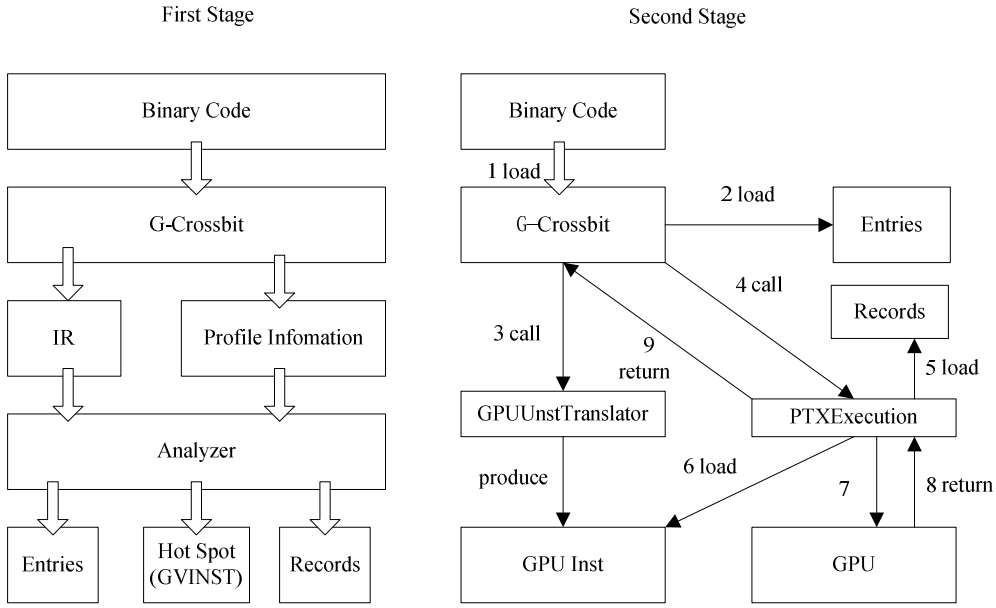

Figure 1 Architecture Of Gxbit. The Arabic Numbers Of The Second Phase Demonstrate The Execution flow When Gxbit Encounters A Parallel Part.

3. IMPLEMENTATION

In GXBit, we have to discover potential parallel parts, usually nested loops, and some information that will be needed for GXBit's execution of these parallel parts on GPU, so we need to use static binary analysis and dynamic analysis. At the first stage of GXBit, static binary analysis is used to probe the nested loops in the code section (.text) of source binary. Then the dynamic binary analysis starts to collect input and output information about these parallel parts at the runtime and after that polyhedral space is built. Finally, polyhedral model is used to determine whether there is data dependence between iterations for these potential parallel parts.

We have noted that the execution mode of GXBit is two phase and GXBit will execute these parts on GPU at the second stage, so after these potential parallel parts are probed and input and output information are collected, we have to store all of them in the form of intermediate representation.

In the process of building polyhedral space and analyzing data dependence between iterations of nested loops, GXBit must have the control flow

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

information of each parallel part, so it is necessary for GXBit to construct control flow graphic (CFG) for current execution path of binary executable. CFG is composed of hundreds of CFG nodes and each node contains the following information:

VBlock - a basic block located in current executing path.

entry point - address of the VBlock's first instruction.

path number - number of branch.

next block[2] - record the address of branch's first intruction.

The process of constructing CFG starts when the first stage of GXBit begins. When a VBlock located in current executing path is produced, GXBit creates new instance of CFG node with setting entry point of the new node to the address of current VBlock's first instruction and initializing other variables noted before.

To an n-nested loop, the process of building polyhedral space starts from the outermost loop and a new polyhedron is produced when the executing flow enters the inner loop and finally n polyhedrons with different dimensions are generated.

There is a classic book named Computer Architecture A Quantitative Approach [16], in which the author talk about three data hazards (RAW, WAW, WAR) in detail. RAW (read after write) happens when j tries to read a source before i writes it. WAW (write after write) happens when j tries to write an operand before it is written by i. WAR (write after read) happens when j tries to write a destination before it is read by i [16].

Through the analysis above, we get our solution to analyze the data dependence for iterations of nested loop. In GXBit, we record all reading and writing operations for memory by storing entry points of VBlocks which contain these operations and writing operations into two maps (container in STL) respectively when the static binary analysis began. When the work of building polyhedral model is completed, data dependence analysis begins. In the building polyhedral space, we talk about that each iterative analysis puts all instructions in this iteration as a point into polyhedral space, so when adding a new point into the polyhedral space, we compare reading and writing instructions contained in this point with instructions of other points in current polyhedral space to see whether data dependence exists or not.

4. EXECUTION AND EVALUATION

As the first phase has been discussed before, we put our focus on the second phase, the same to the first phase, GXBit firstly loads binary executable and starts the execution engine, which will load the entries analyzed at the first stage for all parallel parts.

This section will present experimental results. Table 1 shows the hardware configuration of our experimental environment. As we note before, we evaluate our virtual execution environment's performance by running two applications from CUD SDK Sample and one test case from Parboil Benchmark Suite. We compare the time of these applications running on GXB with running directly on X86 and present the speedup of GXBit.

Table 1. Hardware And Software Configuration Details

Hardware configuration		Software configuration	
CPU	4*Intel Xeon 5110 clocked at 1.60Ghz (1066Mhz FSB), 4M L2 cache	OS	Linux with kernel 2.6.18
RAM	8GB, DDR2-667	Complier	GCC3.4.3, NVCC2.3
GPU	NVIDIA GeForce GTX 260,896MB DRAM, 27 multiprocessors, clocked at 1243MHz	CUDA Version	2.3

The two applications that we choose from CUDA SDK Sample are Matrix Multiplication and ConvolutionFFT2D[14]. This application is multiplication of two matrices. To make a more perfect experiment, we set the size of matrices as 128*128, 512*512 and 1024*1024. To check the correctness of the first stage, we examine the entries and disassemble the corresponding binary executable (objdump -d app-name >output) to locate these potential parallel parts. To the size of 128*128, after executing the first stage, we get the entries at the following: Enter = 0x0804820e, exit = 0x8048292.

Table 2 Performance Of Matrix Multiplication

MatriX size	Native (ms)	GXBit (ms)
128 * 128	25	35
512 * 512	1970	930
1024 * 1024	47109	2530

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 3 Performance Of Convolution FFT2D

Input size	Native (ms)	GXBit (ms)
1000 * 1000	1615	1350
2000 * 2000	6520	930
4000 * 4000	25860	15320

After disassembling the executable, we locate the entry point and find that the instruction located in 0x804820e is the first instruction of inner loop and that is what we expect. As examining entries are similar, we don't show this process in the following experiments. Table 2 shows the performance of this application with different sizes and Figure 2 displays the speedup ratio. From Figure 2, we can see that the speedup of GXBit rises with the increase of matrix's size. The performance of the 128*128 is abnormal because the size of matrices is too small and the acceleration in computation-intensive part can't make up the performance loss of DBT.

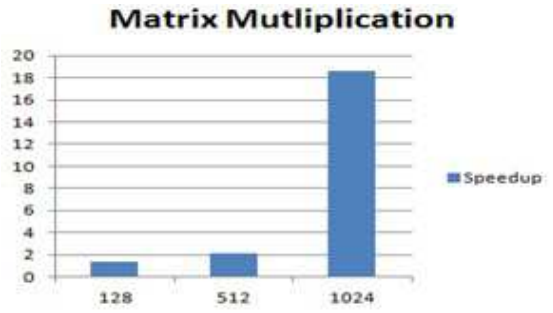

Figure 2 Speedup Of Matrix Mutliplication

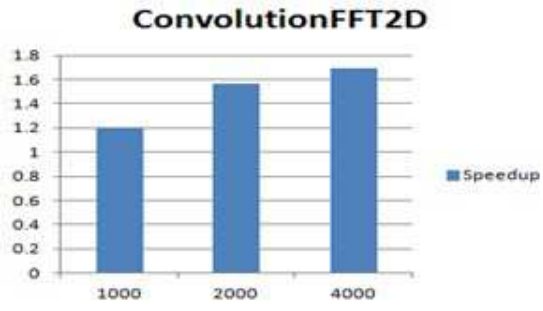

Figure 3 Speedup Of Convolution FFT2D

This application uses Fast Fourier Transformation (FFT) algorithm to implement a Fourier-based general 2D convolution, which is more efficient than the straightforward method. Similar to matrix multiplication, we set the input data array size as 1000*1000, 2000*2000, and 4000*4000. Table 3 shows the performance and Figure 3 shows the speedup ratio. In the Figure 3, we clearly see that the speedups do not vary between each other because of the overhead of random initializing the input data array and this process will be executed by binary translation procedure of GXBit.

5. RELATED WORK

In the process of transforming the sequential code to a parallel one, to find the potential parallel regions is the first step, and there are a lot of researches about that on both source code and binary level. [8, 11, 18, 9, 19, 20] do the work in source level. In binary level, Moseley [1] uses an instrumentation-based approach together detailed information about loops. For our work, we not only discover the nested loops of a binary executable, but also use polyhedral mode to analyze the data dependence and put these loops without data dependence to execute on GPU as well.

To polyhedral model, there have been many works using it to optimize regular programs, especially for nested loops. Bondhugula[5] designed and implemented an automatic polyhedral source-to-source transformation framework to optimize regular programs (sequences of possibly imperfectly nested loops). Pouchet[7,17] used polyhedral model to achieve good performance on large loop nests, Baskaran [6] employed polyhedral model to develop a compiler framework for automatic parallelization and performance optimization of affine loop nests on GPGPUs. In these works, the usage of polyhedral model is based on source code, but in our work, it is based on binary level.

6. CONCLUSION & FUTURE WORK

In this paper, we describe how to use binary analysis and polyhedral model to detect potential parallelizable parts in X86 binaries, analyze whether there are data dependence in these parts or not and briefly discussed how to execute these parallelizable parts on GPU. Several data structures are designed to store information about these parallelizable parts, such as entry and exit point, data address needed to copy from main memory to GPU's memory and etc. Our experimental results validate the correctness of probing parallelizable parts in X86 binaries and demonstrate the performance improvements using GXBit.

GXBit focuses on simple benchmarks, not deal with more general and complicated applications. As future work, we plan to tackle more universal and practical applications.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

REFERENCES

[1] Tipp Moseley, Danial A. Connors, Dirk Grunwald, Ramesh Peri, Identifying potential parallelism via loop-centric profiling, In Proceedings of the 4th International Conference on Computing frontiers. (2007) 143-152.

[2] Nathan Clark, Why Should I Rewrite My Software When Dynamic Compilation Can Be Good Enough, Workshop on Software Tools for Multi-Core Systems (STMCS), 2008.

[3] Nathan Clark, Jason Blome, Michael Chu, Scott Mahlke, Stuart Biles, and Krisztian Flautner, An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors, International Symposium on Computer Architecture (2005) 272-283.

[4] Uday Bondhugula, J.Ramanujam, P.Sadayappan, PLuTo: A polyhedral automatic parallelizer and locality optimizer for multicores. http://pluto-compiler.sourceforge.net.

[5] Uday Bondhugula Albert Hartono, J. Ramanujam, P. Sadayappan, A Practical automatic polyhedral parallelizer and locality optimizer, In ACM SIGPLAN Programming Languages Design and Implementation. (2008).

[6] Muthu Manikandan Baskaran, J. Ramanujan, A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs (2008).

[7] Louis-Noel Pouchet, Cdric Bastoul, Albert Cohen, John Cavazos. Iterative optimization in the polyhedral model: part ii, multidimensional time (2008).

[8] L. Shih-Wei, D. Amer, et al. SUIF Explorer: An interactive and inter-procedural parallelizer, 34 (1999).

[9] W. Thies, V. Chandrasekhar, S. Amarasinghe, A practical approach to exploiting coarse-grained pipeline parallelism in C programs, MICRO (2007).

[10] Georgios, Tournavitis, ZhengWang, Bjorn Franke, Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping, In Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (2009) 177-187.

[11] S. Rul, H. Vandierendonck, K. De Bosschere, A dynamic analysis tool for finding coarse-grain parallelism. In HiPEAC Industrial Workshop (2008).

[12] http://www.crossbit.org.

[13] http://developer.nvidia.com/object/cuda.html.

[14] Victor Podlozhnyuk, FFT-based 2D convolution, NVIDIA CUDA Sample Documentation, 2007.

[15] http://impact.crhc.illinois.edu/parboil.php.

[16] John L. Hennessy, David A. Patterson, Computer Architecture A Quantitative Approach, 4th edition, 2006.

[17] Louis-Noel Pouchet, Uday Bondhugula, Cedric Bastoul, J. Ramaujsm, and P. Sadyppan, Combined Iterative and Model-driven Optimization in an Automatic Parallization, In Proceeding of the 2010 ACM/IEEE International Conference for High Performance Computing, Network, Storage and Analysis. (2010) 1-11.

[18] X Li, W Zhou, D Liu, Polyhedral model based application source codes analysis for ASIP design, System and Informatics (ICSAI). (2012) 962-965.

[19] Bruno Cuervo Parrino, Julien Naxboux, Eric Violard, and Nicolas Magaud, Dealing with arithmetic overflows in the polyhedral model, IMPACT (2012).

[20] A. Jimborean, P. Clauss, B. Pradelle, L. Mastrangelo and V. Loechner, Adatping the polyhedral model as a framework for efficient speculative parallization, PPoPP (2012).