Parallel Branch Prediction on GPU Platform

Liqiang He and Guangyong Zhang

College of Computer Science, Inner Mongolia University
Hohhot, Inner Mongolia 010021 P.R. China
liqiang@imu.edu.cn, zhang03_11@163.com

Abstract. Branch Prediction is a common function in nowadays microprocessor. Branch predictor is duplicated into multiple copies in each core of a multicore and many-core processor and makes prediction for multiple concurrent running programs respectively. To evaluate the parallel branch prediction in many-core processor, existed schemes generally use a parallel simulator running in CPU which does not have a real passive parallel running environment to support a many-core simulation and thus has bad simulating performance. In this paper, we firstly try to use a real many-core platform, GPU, to do a parallel branch prediction for future general purpose many-core processor. We verify the new GPU based parallel branch predictor against the traditional CPU based branch predictor. Experiment result shows that GPU based parallel simulation scheme is a promising way to faster simulating speed for future many-core processor research.

1 Introduction

Branch prediction is a common used function in nowadays superscalar or multicore microprocessor. It uses the branch history (either local or global history or both) to predict whether a next branch instruction is taken or not taken. The accuracy of a branch predictor affects the control flow of a running program with more or less instructions executed along the wrong paths and then affects the final performance of the program. Lots of researches have done related to branch prediction [1, 2] in the past decades.

Branch prediction research generally needs a simulator. Existed schemes include cycle-by-cycle based simulator which runs a program in its simulating environment and uses the real executing flow to investigate the functionality of a branch predictor, and trace based simulator which is much simple and faster than the former but loses some run-time accuracy.

In multicore and many-core processor, branch predictor is duplicated into multiple copies in each core of the processor. Each predictor records its own branch history from the running program in the host core and makes the particular prediction respectively. There is a big design space that can be explored for branch predictors in a many-core system. For example, Branch predictors in different cores can (a) cooperate between each other to increase the prediction accuracies for multi-threaded program, or (b) dynamically combine into together

W. Zhang et al. (Eds.): HPCA 2009, LNCS 5938, pp. 153–160, 2010.

to build a more powerful ones, or (c) switch off part of them to save power if their behaviors are same. Investigating or exploring the design space of the parallel branch prediction for a many-core processor needs a parallel branch predictor simulator. A general technique to build a parallel simulator in academic literature is to parallelize the traditional sequential simulator using *array* structure or *Pthread* programming. This technique may be suitable when doing research for multicore processor with less than sixteen cores but absolutely not useful or impossible for a multicore with more than thirty-two cores or a many-core cases.

In this paper, we try to use a real many-core platform, Graphic Processing Unit (GPU), to help faster simulating speed for massive parallel branch predictor research for future many-core general purpose processor. It is well known that GPU is original designed to target very regular massive parallel computing such as matrix operation, FFT, and lineal algebra. But the processor simulating, including branch prediction, cache accessing, pipeline processing, has a very irregular program behavior which GPU does not favor initially. To our best knowledge, this is the first work that tries to (a) map an irregular program to a regular organized GPU structure and (b) use the existed massive parallel GPU platform to do many-core microprocessor architecture research, especially parallel branch prediction in this paper.

We rewrite most of the code of the branch predictor component in a widely used superscalar processor simulator, SimpleScalar [3], and let them run in a real many-core platform, NVIDIA GeForce9600GT GPU processor [4]. We verify our result (including the control flow and branch predicting outputs of the simulated program) from GPU running against the one from the original CPU based running. Experiment results show that (a) the GPU based code can perform exactly the same functionality as the compared CPU based code which verifies the correctness of our code and proofs the ability of GPU to do irregular operations, and (b) the GPU code can potentially faster the simulating speed with its many-core structure when comparing with the serialized CPU code.

The rest of this paper is organized as follows. Section 2 presents GPU architecture. Section 3 simply introduces the rationale of the branch predictor in this paper and the organization of the parallel branch predictor in future many-core microprocessor. Section 4 describes the design and implementation of our GPU based parallel branch prediction simulator. Section 5 gives the experimental methodology and results. Section 6 discusses the related works, and Section 7 concludes this paper.

2 GPU Architecture

2.1 Hardware Model

The structures of CPU and GPU are shown in Figure 1. The more transistors on GPU are devoted to data processing rather than data caching and flow control. The GeForce 9600 GT architecture has 8 multiprocessors per chip and 8 processors (ALUs) per multiprocessor.

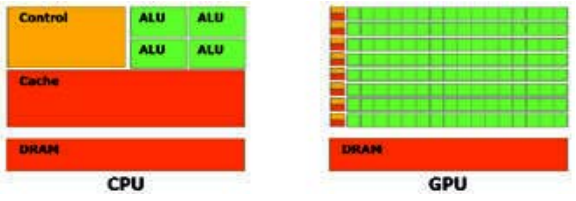

Fig. 1. The GPU Devotes More Transistors to Data Processing than CPU[5]

2.2 Memory Model

The memory model of NVIDIA GeForce 9600 GT is shown in Figure 2 (left). Each multiprocessor has on-chip memory of the following four types.

- One set of local 32-bit registers per processor. The total number of registers per multiprocessor is 8192.
- A shared memory that is shared by all the processors of a multiprocessor. The size of this shared memory per multiprocessor is 16 KB and it is organized into 16 banks.
- A read-only constant cache that is shared by all scalar processor cores and speeds up reads from the constant memory space, which is a read-only region of device memory. The size of the constant cache is 8 KB per multiprocessor.
- A read-only texture cache that is shared by all the processors in a multiprocessor, which speeds up reads from the texture memory space. The texture cache size is 8 KB per multiprocessor.

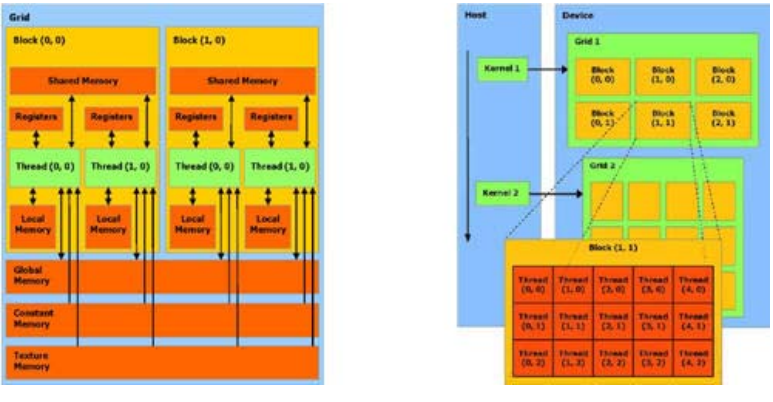

Fig. 2. CUDA Memory (left) and Programming (right) Model[5]

The local and global memory spaces are read-write regions of device memory and are not cached. A single floating point value read from (or written to) global memory can take 400 to 600 clock cycles. CPU and GPU transfer data through the global memory.

2.3 Programming Model

CUDA's programming model (Fig.2.right)assumes that the threads execute on a physically separate device that operates as a coprocessor to the host running the C program. It consists of a minimal set of extensions to the C language and a runtime library. GPU is a typical SIMD parallel model. The CPU implements parallel processing of multi-threads by calling ***kernel*** function which runs on GPU. A group of threads with multiple same instructions and different data streams form a ***block***, different blocks can execute different instruction streams, and many *blocks* form a ***grid***. *Thread*, *block* and *grid* form a three-dimensional-thread-space. For convenience, ***threadIdx*** is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one, two, or three-dimensional thread block.

3 Rationale of Branch Predictor

3.1 2Bits Branch Predictor in Single Core

2Bits branch prediction is a simple and well known prediction scheme. Although the prediction accuracy is much less than many up-to-date complicated branch predictors like OGEHL [1] and L_TAGE [2], but it is sufficient to be an example to show how to realize a parallel branch predictor in GPU. In a 2Bits branch predictor, there is a table to record the local histories of different branches, and each entry of the table uses 2 bits to trace the recent branch history of one branch instruction. A set-associate cache, BTB, provides the branch target if the branch is taken. It must store the entire PC in order to accurately match the branch instructions. Also, there is a Return Address Stack (RAS) to be used for sub-routine call and return.

3.2 Parallel Branch Prediction in Many-Core Processor

In multicore and many-core processor, each core has its own branch predictor to be used by the program running in it. Cores are connected through crossbar or grid. Communication between cores can be done through the on-chip link. All the components in a branch predictor, predicting table, BTB and RAS, must be duplicated in each core. The operations in these separate predictors are parallel and independent on each other.

In order to simulate the parallel branch prediction in multicore and many-core processor, people can change the scalar data structure in original simulator with ***array*** and ***for*** loop structure, or using ***Pthread*** parallel programming technique. But these methods are only useful when the number of cores is less than thirty-two in our experiment. When the cores are more then thirty-two, the simulating speed is dropped dramatically, and sometime it is unacceptable for the researcher. In this paper, we try to use an existed many-core platform, GPU, to construct our massive parallel branch predictor. It is well known that

GPU is designed for massive regular operation. But processor simulating, including branch predicting, has very irregular program behaviors. How to map such irregular applications into the GPU platform is interest for us. To our best knowledge known, this is the first work that tries to map such irregular program, especially branch predictor, into GPU platform.

4 Parallel Branch Prediction in Many-Core Processor

We use SimpleScalar as our baseline implementation. It models a five-stage, out-of-order superscalar microprocessor. In this paper, we select 2Bits prediction scheme as an example to realize in GPU platform, but it is easy to port our method for other prediction schemes.

To do branch prediction, SimpleScalar uses two functions, ***lookup()*** and ***update()***. The *lookup* function is to do predicting using the PC of branch instruction, and the *update* is to do updating predicting table using the actual branch result when the previous prediction is wrong.

To port the CPU codes to GPU, we need to do two steps. First, there are many ***Static*** variables in original code that can not be handled by GPU program. So we need redefine all these variables to global variables such that they can be ported into GPU code. Second, we rewrite the two functions, *lookup* and *update*, to GPU kernel fashion. To do this, (a) all the definitions of ***Structure*** need to change to ***Array***, and the corresponding variables need to be redefined, (b) for the variables used as the interface of *kernel* we need define two same entities, one is for CPU, and the other is for GPU, and the values of GPU variables need to be transferred from the corresponding CPU ones, and (c) the prediction result needs to be copied back to CPU in order to be used by other part of the codes. The method to call the two GPU kernels in CPU part is as follows:

$$bpred\_lookup <<< 1, NUM >>> ( )$$

and

$$bpred\_update <<< 1, NUM >>> ( )$$

Where *NUM* is the number of predictors we want to parallel simulate. Through varying the value of *NUM*, we can model different number of parallel branch predictors in a multicore or many-core processor.

5 Experiment Setup and Result

Our experiment is conducted in Fedora Core 8 Linux system. The GPU chip is NVIDIA GeForce9600 GT, and we use CUDA 2.1 programming environment. Four benchmark programs, *mgrid*, *applu*, *apsi*, and *crafty*, are selected from SPEC CPU 2000 [7]. In our 2Bits branch predictor, branch history table has 2K entries, BTB is 4 way-associated and has 512 sets, and RAS has 8 entries. We validate our GPU based implementation against the CPU based one, and

prove the correctness of the GPU one in logic semantics. Thus, in the following content, we only show and compare the running time of different implementations, and do not present the prediction results due to the meaningless of this work.

In our experiment, due to the hardware (global memory and register file) limitation we can only fork up to four parallel predictors in GPU. Although it is not desirable for our motivation of this work, it is acceptable for us at present time because this is our first step to target our final objective. The main task for us in this stage is to verify our thinking that GPU can handle irregular applications as the regular ones, and try to investigate the potential speedup of using GPU vs. CPU. Improving the parallelism will be our next work in the future. For the experiment, we construct two-threads, three-threads, and four-threads workloads using a same SPEC program, and compare the running time in GPU platform with the one in CPU platform. To be fairness, we run the same experiment four times and use the average number as the final result.

Table 1 shows the program running times in CPU and GPU respectively. It is clear to see that GPU based running time is much longer than the CPU based one, and it is not like the intuitively desired result. Through carefully analysis we know that it is because of the low parallelism (the maximum is four) of our experiments. When the kernels are called from CPU, lots of data need to be transferred to GPU, and the calling and returning also need time to finish, thus the communication between CPU and GPU takes most of the running time for our GPU based implementation which causes the performance degradation. If we can successfully increase the parallelism, then the GPU code will surpass the CPU one.

In another view, Figure 3 (left) shows the running times of four programs with different number of threads. When the number increases from one to four, the time does not change a lot, especially at 2, 3, and 4 threads cases. Compare with the data shown in Figure 3 (right), the running times in CPU increase

Table 1. Program running times in CPU and GPU

Benchma	No. of Thread	1	2	3	4
mgrid	**CPU**	393.30	439.11	477.94	526.87
	GPU	8416.0	8445.28	8457.31	8428.12
	Ratio	21.39	19.23	17.69	15.99
applu	**CPU**	332.08	347.91	365.42	376.76
	GPU	2757.15	3243.70	3243.09	3237.75
	Ratio	8.30	9.32	8.87	8.59
apsi	**CPU**	379.29	421.45	453.84	492.92
	GPU	7072.14	8973.61	8942.55	8984.73
	Ratio	18.64	21.29	19.70	18.22
crafty	**CPU**	435.07	553.10	648.38	742.95
	GPU	18040.9	21757.1	21773.5	21748.8
	Ratio	41.46	39.33	33.58	29.27

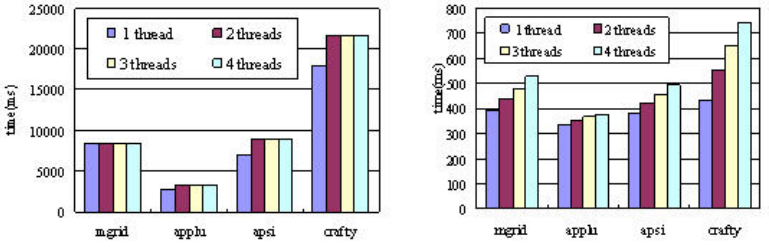

Fig. 3. Running times in GPU (left) or CPU (right) with different number of threads

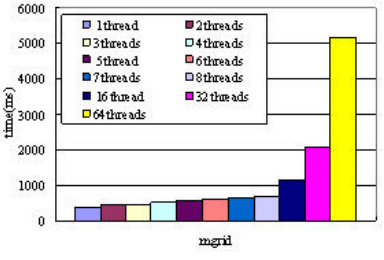

Fig. 4. Running times of ***mgrid*** in CPU with different number of threads

continuously with the number of threads increasing. For crafty, the times increase dramatically for four threads running vs. the single thread running. All the data shows that GPU has a good extendibility than CPU in terms of the parallel program running.

To further show the CPU limitation, we run mgrid in CPU platform with up to sixty-four threads. Figure 4 shows that when the number of parallel running program is greater than sixteen, the running time increases dramatically in CPU platform. Thus if we can improve the parallelism of our GPU based code, a big speedup can be obtained. To do this, we must take the hardware limitation into account when designing the software. As presented in section 2, the global memory, especially the register file, are very limited resources, so we must split out big kernel into small ones such that we can fork more parallel threads and let them run concurrently in GPU and get speedup over the CPU implementation.

6 Related Works

Branch prediction has got long time attention in industry and academic research literature. Most of the researches focus on how to improve the predicting accuracy [1, 2], and less work has been done on how to simulate massive parallel branch predicting for future large scale multicore and many-core processor.

GPU was announced initially for graphic process, but recently due to the highly parallel structure and powerful computing capability GPU has been

widely used in massive scientific computing applications [8, 9], such as GPGPU [10]. Most of applications in GPGPU are regular computation, and very few works have been done for irregular application, microprocessor simulating for instance.

This work, to our best knowledge known, is the first work that targets the GPU on very irregular application, multicore and many-core processor simulating. Although we do not obtain performance speedup over CPU based code, we verify our idea of port CPU code to GPU platform, and know the potential speedup of GPU vs. CPU ones. The further improvement of our code will leave for our future work.

7 Conclusion

This paper, firstly, investigates how to map an irregular application, parallel branch predicting for multicore and many-core microprocessor in GPU platform using NVIDIA CUDA programming environment. It verifies the correctness of the GPU implementation and gets the view of potential speedup of GPU implementation over the CPU one. It discusses the details of the GPU implementation, and analyzes the existed issues in current code and presents the future work of this paper.

Acknowledgements

This work is supported by Inner Mongolia Natural Science Foundation Project No. 208091, and the Ph.D Research Startup Foundation of Inner Mongolia University No. 208041.

References

1. Seznec, A.: Analysis of the OGEHL predictor. In: Proceedings of the 32th International Symposium on Computer Architecture (IEEE-ACM), Madison (June 2005)
2. Seznec, A.: A 256 Kbits L-TAGE predictor, CBP-2 (December 2006)
3. Burger, D., Austin, T.M.: The SimpleScalar Tool Set, Version 2.0. ACM SIGARCH Computer Architecture News 25(3), 13–25 (1997)
4. NVIDIA GeForce 9600 GT, `http://www.nvidia.com/object/product_geforce_9600gt_us.html`
5. NVIDIA CUDA: Programming Guide. Version 2.2. (4/2/2009)
6. Lee, J.K.L., Smith, A.J.: Branch prediction strategies and branch target buffer design. Computer 17(1) (January 1984)
7. Henning, J.: SPEC CPU2000: Measuring CPU Performance in the New Millennium. IEEE Computer, Los Alamitos (2000)
8. Kerr, A., Campbell, D., Richards, M.: QR Decomposition on GPUs. In: Proceeding of 2nd Workshop on GPGPU 2009, Washington, D.C., USA, March 8 (2009)
9. Gulati, K., Croix, J.F., Khatri, S.P., Shastry, R.: Fast Circuit Simulation on Graphics Processing Units (IEEE) (2009)
10. GPGPU, `http://www.nvidia.cn/object/cuda_home_cn.html`