INTERNATIONAL JOURNAL OF
NUMERICAL ANALYSIS AND MODELING
Volume 9, Number 2, Pages 193–207

GPU ACCELERATED PARALLEL BRANCH PREDICTION FOR MULTI/MANY-CORE PROCESSOR SIMULATION

LIQIANG HE GUANGYONG ZHANG AND JINGDONG JIANG

Abstract. Branch Prediction is a common function in nowadays microprocessors. Branch predictor is duplicated in each core of a multi/many-core processor and makes prediction for multiple concurrent running programs respectively. To evaluate the parallel branch prediction in a multi/many-core processor, existing schemes generally use a parallel simulator running on a CPU that does not have a real massive parallel running environment to support the simulation and thus has a bad simulating performance. In this paper, we use a real many-core platform, GPU, to perform a parallel simulation of branch prediction for the future general purpose multi/many-core processor design. We verify the correctness of the GPU based parallel branch predictor against the traditional CPU based branch predictor. Experiment result shows that the GPU based parallel simulation scheme obtains a two to ten times of speedup over the CPU platform when the issue rate ranging from one to four instructions per cycle, and it shows that the GPU based scheme is a promising way to improve the simulation speed for future multi/many-core processor research.

Key words. Branch Prediction, Parallel Simulator, GPU, and Multi/Many-core Processor.

1. Introduction

Branch prediction is a commonly 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. 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 been done related to branch prediction [1-3] in the past decades.

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

In multicore and many-core processor, branch predictor is duplicated 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 multi/many-core system. For example, Branch predictors in different cores can (a) cooperate with each other to increase the prediction accuracies for multi-threaded program, or (b) be dynamically combined into more powerful predictors, or (c) switch off parts of them to save power if their behavior is the same. Investigating or exploring the design space of the parallel branch prediction for a multi/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

Received by the editors December 9, 2009 and, in revised form, January 10, 2011.
2000 *Mathematics Subject Classification.* 68Uxx.
This work is supported by Inner Mongolia Natural Science Foundation Project No. 20080404MS0901, and the Ph.D Research Startup Foundation of Inner Mongolia University No. 208041.

using array structure[4-5] or Pthread programming scheme[6]. In a general simulating environment without a big memory support, this technique may be suitable for research on multicore processor with less than sixteen cores but it is absolutely not useful or possible for multicore with more than thirty-two cores or for many-core cases. Some other researches [7-9] rely on FPGA to do parallel simulation for multi/many-core processors. Although the simulation speed is fast, the ability and the scalability are limited by the hardware itself.

In this paper, we extend our previous work [10] and use a real many-core platform, GPU (Graphic Processing Unit), to help improve the simulation speed for massive parallel branch predicting research for future multi/many-core processor. It is well known that GPU is originally designed to target regular massive parallel computing such as matrix operations, FFT, and lineal algebra. But the processor simulation, including branch prediction, cache accessing, pipeline processing, has very irregular program behavior which GPU does not favor initially. In this work, we try to (a) map an irregular simulating program to a regular organized GPU structure and (b) use the existing massive parallel GPU platform to help the multi/many-core processor architecture research, especially parallel branch prediction. Although only the branch prediction is considered, it is a case study and start point of research on multi/many-core simulation using GPU platform for the future microarchitecture research.

We rewrite most of the code of the branch predictor component in a widely used superscalar processor simulator, SimpleScalar [11], and let it run in an NVIDIA GTX275 GPU processor [12]. We verify our result (including the control flow and branch predicting outputs of the simulated program) from GPU runs 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 shows the ability of GPU to do irregular operations, and (b) the GPU code can potentially faster the simulation speed, up to ten times, for the branch prediction simulating with its many-core structure when compared with the serialized CPU code.

Comparing with our previous work, we make the following new contributions in this paper:

- Consider the specific GPU features, and optimize our implementation of the GPU based parallel branch prediction simulation.
- Verify the correctness of previous work [10], and show the speedup results on new hardware platform.
- Through experiment on three typical types of workloads, a trend of the simulating speedup on GPU platform is obtained, and the maximum speedup values at 8K simulated cores or threads are observed.
- Make sensitivity analysis of the instruction issue rates in the simulated cores and the number of instructions being simulated.

The rest of this paper is organized as follows. Section 2 presents the GPU architecture and programming model. Section 3 introduces the rationale of the branch predictor used 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 Section 6 presents and analyzes the results. Then, Section 7 discusses the related work, and finally Section 8 concludes this paper.

2. GPU as Parallel Computer

In this section, we introduce the architectural aspects of the NVIDIA GPU device used in this work and the corresponding programming model.

2.1. NVIDIA GPU architecture. The new generation of GPUs is applied to the general-purpose GPU computing [13]. Different from the philosophy of traditional CPU organization, more transistors on GPU are devoted to data processing rather than data caching and flow controlling. The fundamental building block of the NVIDIA GPU is the streaming multiprocessors (SMs), and each of them consisting of multiple streaming processors, and only one instruction fetch/decode unit. GPU is a typical SIMD (Single Instruction Multiple Data) parallel model, and all the processing cores must simultaneously execute the same instruction stream. Each SM has a shared register pool and shared memory space which is organized into banks, and the bank conflicts are avoided. The local and global memory spaces are read-write regions of the device memory and are not cached. A single floating point value read from (or written to) global memory can take about 400 to 600 clock cycles. Data processed in GPU is transferred from CPU, and the result is transferred back. All the transmission is done between the host memory and the GPU's global memory. A read-only constant cache is shared by all the scalar processor cores, and has very short access latency. Another read-only texture cache is shared by all the processors in a multiprocessor, which speeds up read operations from the texture memory space.

2.2. CUDA programming model. NVIDIA GPU devices use CUDA (Compute Unified Device Architecture) as their programming model [14] in which the CUDA threads execute on a physically separate device that operates as a coprocessor to the CPU host. CUDA consists of a minimal set of extensions to the C language and a runtime library.

The CPU host implements parallel processing of multi-threads by calling ***kernel*** functions which run 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, to form a one, two, or three-dimensional thread block. The index of a thread is through its specific thread ID.

A warp which is a group of 32 threads from the same thread block is the main scheduling unit in CUDA. In fact, warp is a part of CUDA, but warp can be helpful in understanding and optimizing the performance of CUDA applications on devices. The number of active warps in a SM is an important factor in tolerating global memory access latency.

3. Rationale of Branch Predictor

In this section, first, we introduce the rationale of a simple branch predictor, 2Bits predictor [15], which acts as an example of showing how to map CPU code to a GPU program in next section. Then, we present the organization and working mechanism of the parallel branch predictor in a future multi/many-core processor.

3.1. 2Bits branch predictor in single core. 2Bits branch prediction is a simple and well known prediction scheme. Although its prediction accuracy is much lower than many up-to-date complicated branch predictors like OGEHL [1] and L_TAGE [2], it is sufficient to be an example to show how to realize a parallel branch predictor

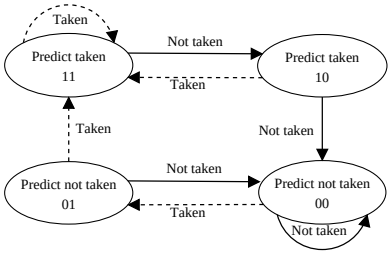

FIGURE 1. The states in a 2-bit prediction scheme [15]

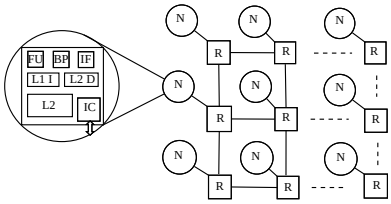

FIGURE 2. The structure of a 2D-mesh connected multi/many-core processor, where R is the router and N is the node which can be a core or a memory bank, etc

on GPU. In 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 particular branch instruction. The instruction fetch unit in a CPU uses the PC of a branch instruction to index the table, and obtains the predicting result (taken or not taken) according to the 2Bits value. The diagram of state translation of 2Bits prediction is shown in Figure 1. Due to the limitation of the table size, different branch instructions may be mapped to the same table entry which causes interfere between each other, named as *alias conflict*. In addition to the predicting table, there is another set-associate table, BTB (Branch Target Buffer), to provide branch target if a branch is predicted as taken. BTB must store the entire PC in order to accurately match the branch instructions. Also, there is a RAS (Return Address Stack) to be used for sub-routine *CALL* and *RETURN* instructions and it generally has 8 or 16 entries in modern microprocessor.

3.2. Parallel branch prediction in multi/many-Core processor. A multi/many-core processor has roughly several tens or more of cores on the chip. Each core can be a complex, multiple instructions issue rate, out-of-order execution processing unit, or a simple, single-issue, in-order one. Cores are connected through an on-chip network such as 2D-MESH, butterfly, or FAT tree [16]. Communication between cores is done through the interconnected network links. A many-core processor is organized either homogeneously or heterogeneously decided by whether or not the composed cores are the same. A typical structure of a 2D-mesh connected homogeneous many-core processor is shown in Figure 2.

As shown in Figure 2, every core in a multi/many-core processor has a branch predictor which is used by the program running in it. All the components, including the predicting table, BTB and RAS, are duplicated in each core. The operations in these separate predictors are parallel and independent of each other.

Some potential architectural optimization can be done for these predictors, for example, (a) multiple ones cooperate together to make a more accurate prediction

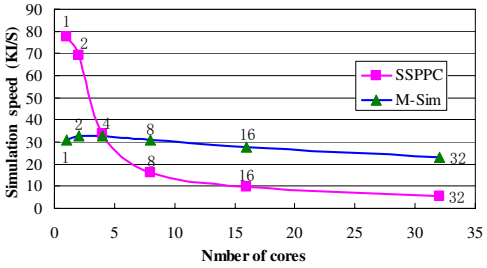

FIGURE 3. The trend of the simulation speed with the number of simulated cores increasing

for a multi-threaded program workload like the cooperate cache in [17], or (b) multiple predictors can be combined into one big and more powerful predictor for a particular running thread, or (c) switch off some predictors to save power if the program behaviors are the same.

To perform simulation for a parallel structure like a multi/many-core processor, existing schemes always rely on the traditional single core simulator such as SimpleScalar, sim-alpha[18], etc, and change the scalar data structure in the simulator with ***array*** and ***for*** loop structures, for example CMP-Sim and M-Sim[4-5], or use the ***Pthread*** parallel programming technique, SSPPC for instance[6]. Unfortunately, these methods are only useful and acceptable when the simulated cores are not too many in terms of simulation speed, such as less than thirty-two cores in our experiment. With the number of cores increasing, the simulation speed of these methods drops continuously and at some points it becomes unacceptable for the computer architecture researcher. Figure 3 shows the trend of the simulation speed with the number of cores increasing in M-Sim and SSPPC using the machine shown in Section 5. From Figure 3, when the number of cores is greater than 32, the simulation speed of M-Sim and SSPPC is lower than 20 and 5 kilo instructions per second. With this speed, the time to simulate a typical run for a 32-core processor, 100 million instructions for one configuration running for instance, will need 2 and 8 hours respectively.

To improve the simulation speed, in this paper we adopt an existing hardware many-core platform, GPU, to help the parallel multi/many-core simulation, specifically on massive parallel branch predictor simulation. It is well known that GPU is designed for massive regular operations, such as matrix multiplication, FFT, lineal algebra, etc. But processor simulating, including branch predicting, has very irregular program behavior. How to map such irregular applications onto the GPU platform is of interest to us. In this work, we try to map irregular programs such as branch predictor onto GPU platform. In the next section, we will present the details of our implementing techniques.

4. Parallel Branch Prediction on GPU Platform

4.1. Base infrastructure. We use a single core simulator, SimpleScalar, as our baseline implementation. It models a five-stage pipeline (*Fetch, Decode, Issue, Execute,* and *Write Back*), out-of-order superscalar microprocessor. The branch prediction is done at Decode stage. Five different predicting schemes, 2Bits, Two Level, Static Taken or Not Taken, and Combined scheme, are implemented in the simulator. In this paper, we select 2Bits predicting scheme as a case study to realize the parallel branch prediction on the GPU platform, and it is easy to apply our method for other predicting schemes.

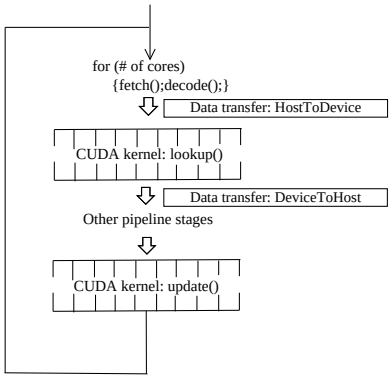

FIGURE 4. Overall structure of the parallelized simulator using CUDA

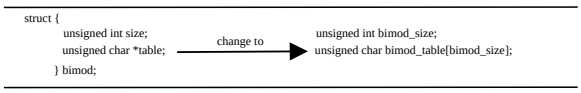

FIGURE 5. Changing structure to array definition

To make a branch prediction, SimpleScalar uses two functions, ***lookup()*** and ***update()***. The *lookup* function is called when the fetched instruction is a conditional branch, and the predicting result is returned using the instruction PC to access the predicting table. The *update* function is to update the predicting table using the actual branch result when the branch is resolved at *Execute* stage.

In addition to the predicting table, the BTB table needs to be accessed and given the branch target if a branch is predicted as taken. For sub-routine *CALL* and *RETURN* instructions the RAS will be used to save the return address.

4.2. Parallelization scheme on GPU. As discussed in Section 2, we use NVIDIA GPU and CUDA programming model as our parallelization platform. Before doing the parallelization, we firstly modify the single core simulator into multi/many-core version using the same method in [4-5]. Then, we port the sequential branch predicting code running in CPU onto the parallel GPU platform, and further improve the simulation speed through exploiting the specific GPU features.

The overall structure of the parallelized simulator is shown in Figure 4.

To realize the parallel branch prediction with CUDA, we make five changes. The details are presented as follows.

- Redefine the *Static* variable

In the original simulator code, there are many ***Static*** variables that are defined as local variables in functions and can store values even after the functions are returned. These static variables are good at transferring values between different function calls, but make the code confusing and hard to be parallelized. In our implementation, we change all these variables to global variables.

- Replace the *Structure* variable with Array

CUDA programming model suggests using *Array* instead of *Structure* variables in order to benefit from the short access latency from the coalesced global memory access. So, to get the best parallel speedup, we redefine all the ***Structure*** variables as ***Array*** variables. For example, in Figure 5, the *bimod* is changed to array variable.

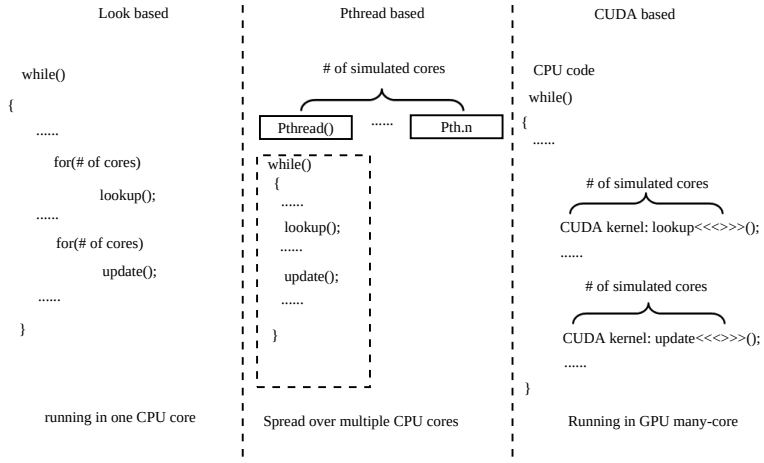

FIGURE 6. Comparison of three simulating schemes

Further more, in order to transfer values between CPU and GPU all the variables used as the interface of CUDA kernel need to be defined as two entities with the same size, one is for CPU, and another is for GPU. For example, in Figure 5 the predicting table, *bimod_table*, needs to be defined as *bimod_table_cpu* and *bimod_table_gpu* respectively.

- Task level parallelization

Similar as the Pthread based scheme, we exploit task level parallelization for the parallel branch prediction simulation. We rewrite the code of the two main functions, *lookup* and *update*, in GPU kernel fashion, and change the name to *bpred_lookup* and *bpred_update*, as shown below.

bpred_lookup <<< *NUM*, ... >>> (*parameter*, ...)

and

bpred_update <<< *NUM*, ... >>> (*parameter*, ...)

Where *NUM* is the number of predictors being simulated simultaneously. By varying the value of *NUM*, different numbers of branch predictors in a multi/many-core processor can be simulated.

As shown in Figure 6, the loop based scheme only uses one physical core to simulate the functionalities of different branch predictors, so even with a real multicore processor it can not exploit the existing core level parallelism and improve the simulation speed. Whereas for the Pthread based scheme, it uses a relatively heavy-weight pthread to simulate a single core. So with the number of simulated cores increasing, the pthreads compete the limited physical cores and memory resources more and more severely, and the overhead of context switching among pthreads will make the simulation speed dropping dramatically. In our contrast experiment using SSPPC simulator, when the number of simulated cores is greater than sixteen the simulation can not proceed due to the competition. Comparing with them, CUDA based scheme uses very lightweight thread (several cycles of switching overhead) to make a SIMD simulation and obtains a good performance speedup.

- Coalesced global memory access

In [14], CUDA suggests decreasing the overall access latency by using coalesced global memory access among the threads in a warp. To take advantage of this feature, we reorganize the data placement method for many structure variables

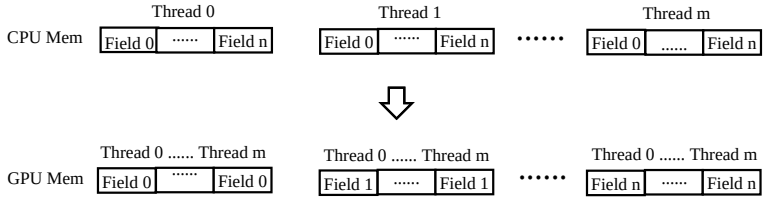

FIGURE 7. Data placement transformation for the coalesced global memory access

such that when different CUDA threads access the same field in the SIMD mode they can perform the coalesced global memory access. Figure 7 shows an example of such transformation.

- Using constant memory in GPU

In GPU, a value stored in the constant memory can be accessed by one thread and used by all the concurrent running threads in the same warp in order to greatly reduce the total access time. To benefit from this feature, we store all the unchanged variables, the *md_op2flag* array for instance, in the constant memory.

With the above techniques, we realize a parallel branch prediction for multi/many-core processor simulation with CUDA.

4.3. Issue about simulating performance. With the techniques presented in the last subsection, a basic parallel branch predictor for multi/many-core architecture research is constructed. Although the functionality is correct, the simulating performance may become an issue due to the intrinsic high complexity of multi/many-core processor organization.

First, because the simulated programs running in different cores have various behaviors, the fetched instructions from the programs at a given cycle may not be all branches. For this case, two options can be selected. One, the CUDA kernel only makes parallel predictions for the branch instructions and the other simulated instructions do not go through the kernel code. This method makes the kernel has variable numbers of threads. It must match the instructions with the corresponding simulated programs and cores. This asynchronous process steps make GPU difficult to work efficiently in its SIMD working mode. Another option, on the other hand, sends all the instructions to the kernel, and lets the kernel to decide whether making the prediction or not. The pseudo code of this method is shown in Figure 8. It makes the kernel easy to be understood and has regular behavior. We use this method in our implementation, and get a performance speedup.

Second, to speedup the design space exploration our implementation permits simultaneously running multiple configurations in the same CUDA kernel. For example, the size of predicting table, the size of BTB, and even the type of predicting scheme, can be different in the CUDA threads, and the threads can run concurrently in the kernel and send the predicting results. It is the same case as simulating a heterogeneous multi/many-core processor in which the cores have variable configurations.

5. Experiment Setup

We run our parallel branch prediction on an Intel Core 2 Quad Q9500 2.8GHz machine equipped with 2GB DDR3 memory and an NVIDIA GTX275 GPU. The bidirectional bandwidth of the on board PCI-E bus is 8GB/s. To build the code, we use the CUDA developing environment SDK2.3 [14]. We select -O3 as the

```
__global__ void bpred_lookup (inst,PC,·····)
{
  ......
   if (inst is a branch)
          rst = predicting (PC);
   else
          rst = -1;
  ......
}
```

FIGURE 8. Pseudo code of the bpred_lookup kernel

TABLE 1. Hardware details of GPUs and CPU in the experiments

Name	**Parameters**
CPU	Intel Core 2 Quad CPU Q9500 2.83GHz, DDR3 1033MHz 2GB
GTX275	GDDR3 896MB, 30 SM, 8 SPs/SM, 16384 registers/SM, Max. 1024 threads/SM, 16KB constant/shared memory per SM, 1.3 compute capability

optimization level for the CPU code and -O2 for the GPU kernel. The operating system is Fedora Core 10 Linux. Table 1 lists the hardware details for the CPU and GPU.

In our experiment, we simulate a homogeneous multi/many-core processor with the same 2Bits branch predictor configuration in each simple single-issue, five pipeline stages core. The 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 times of different branch prediction implementations, and do not present the meaningless prediction results. In the CPU based implementation, we use the loop method to perform the simulation due to its relatively fast simulation speed and only count the time of making branch prediction for the same number of simulated cores. In GPU part, we count the time of data transmission between CPU and GPU plus the time of making parallel branch prediction. The speedup of GPU vs. CPU is calculated using the two simulation times.

The twenty-six benchmark programs are selected from SPEC CPU 2000[19]. The multi-program workload running in the multi/many-core processor has a big combination possibility. So, to obtain the upper bound and lower bound of the speedup, we construct three types of workloads, and shown in Table 2. The *MIX3* type of workload has the maximum irregular behavior that is difficult for GPU to handler. Whereas the *MIX1* type has a good regular behavior (all the combined programs have the same executing stream) which favors the GPU running mode. And the *MIX2* is among the two other types. In each workload, all the programs are simulated at least 20 Million instructions.

6. Result

6.1. Speedup of GPU vs. CPU for MIX1 workload. Figure 9 shows the speedups of GPU implementation over CPU ones for two *MIX1* type of workloads,

TABLE 2. Workload description in our experiment

Type	Description (assume an N-th workload running in an N-core processor)
MIX1	The workload includes N same programs.
MIX2	The workload is organized as half of the running warps have regular behaviors and the other half ones do not.
MIX3	Any two adjacent threads are different, like "abc...zabc...z..."

ammp and *applu*. From the figure, with the number of cores or threads being simulated ranges from 32 to 8K the speedup goes from a minus value to a positive value. Here, a core need not be a physical processing unit, and it can be a logical core similar as the one in a Simultaneous Multithreading processor [20] or a processor supporting HyperThreading [21]. Also, as described in Section 4.3 multiple configurations can be simulated in a same time of CUDA kernel running, so threads in Figure 9 may represent a same group of cores with different configurations. This is the same for other results in this section. When the speedup has a minus value, it means the simulation speed in CPU is faster than in GPU, and otherwise the GPU based one is faster than the CPU ones.

For *ammp* and *applu*, when the number of cores or threads is greater than 1K, using GPU to make parallel branch prediction is faster than using CPU to do the same thing. The maximum speedups of GPU at the 8K cores or threads' cases are close to three. Not only for these two example workloads, through our experiments we found that all the workloads have similar performance trends when the number of cores or threads changes from 32 to 8K, and most of them begin getting positive simulation speedups on GPU platform when they are more than 1K.

The reason that CPU based implementation is faster than the GPU one when the simulated cores or threads are not too many, say less than 1K, is because the large data transmission overhead between the two platforms at the time of launching CUDA kernels in GPU implementation. In addition, for processor simulation application it is difficult to adopt the asynchronous transfer scheme provided in CUDA due to the operation dependency between the simulated pipeline stages. In other words, it can not start a data transmission between CPU and GPU to do the parallel branch prediction until the previous pipeline stages, *fetch* and *decode* for example, finish.

Table 3 lists the detailed running times counted in second and the speedups of GPU vs. CPU for twenty-six *MIX1* workloads. Here, we only show the time to make branch prediction, and ignore the parts used for simulating other pipeline stages. All the twenty-six workloads have worse simulating performances on GPU platform when the number of cores or threads is less than 1K. When the number is greater than 1K the GPU based parallel simulation obtains a better performance than the CPU based one. The maximum speedup is close to 3.8 times at 8K's case.

We also show the GPU occupancy values of the two kernels in Table 4. Each ***block*** includes 128 threads, and an ***SM*** can have up to 1024 threads, so the kernels in our implementation have the maximum number of active blocks and the GPU occupancy is 100%. In addition, each thread uses 16 and 13 registers in *bpred_lookup* and *bpred_update* respectively, and the overall usage is less than the total number of registers available in an SM, 16384. And the memory usage is a same case as the register usage.

TABLE 3. Running times (in second) and the speedups of GPU vs. CPU for 26 MIX1 type of workloads, (C: CPU, G: GPU, S: Speedup)

# of C./Th. / Ben.	2048			4096			8192		
	C	G	S	C	G	S	C	G	S
ammp	85.1	54.6	1.56	168.2	63.7	2.64	336.5	90.1	3.73
applu	71.1	53.0	1.34	140.2	62.2	2.25	299.1	88.6	3.38
apsi	73.6	53.2	1.38	144.8	62.5	2.32	298.4	89.1	3.35
art	76.0	53.5	1.42	150.0	62.7	2.39	318.3	89.2	3.57
bzip	74.1	53.2	1.39	145.9	62.5	2.33	289.9	89.1	3.25
crafty	84.2	54.6	1.54	165.6	63.7	2.60	332.5	90.1	3.69
eon	80.3	54.0	1.49	157.8	63.1	2.50	325.0	89.7	3.62
equake	80.8	54.0	1.50	159.6	63.2	2.53	318.8	89.7	3.55
facerec	82.3	54.2	1.52	161.4	63.4	2.55	322.5	89.8	3.59
fma3d	84.2	54.5	1.54	165.9	63.7	2.60	331.5	90.0	3.68
galgel	77.5	53.7	1.44	152.5	62.9	2.42	305.4	89.5	3.41
gap	72.1	53.0	1.36	142.3	62.3	2.28	299.6	88.8	3.37
gcc	83.8	54.3	1.54	165.1	63.5	2.60	329.8	90.0	3.66
gzip	74.0	53.2	1.39	146.0	62.5	2.34	290.3	89.0	3.26
lucas	73.0	53.2	1.37	144.0	62.4	2.31	307.4	89.0	3.45
mcf	72.2	53.0	1.36	142.5	62.3	2.29	293.2	88.8	3.30
mesa	72.1	53.0	1.36	142.0	62.3	2.28	292.7	88.9	3.29
mgrid	76.4	53.5	1.43	149.8	62.7	2.39	298.8	89.2	3.35
parser	83.7	54.4	1.54	165.1	63.6	2.60	330.5	90.0	3.67
perlbmk	84.3	54.4	1.55	165.6	63.6	2.60	330.8	89.9	3.68
sixtrack	77.5	53.7	1.44	152.0	62.9	2.42	303.3	89.6	3.39
swim	72.0	53.1	1.36	142.1	62.3	2.28	293.1	88.8	3.30
twolf	85.7	54.5	1.57	168.7	63.6	2.65	336.9	90.1	3.74
vortex	76.0	53.5	1.42	149.7	62.7	2.39	298.0	89.1	3.34
vpr	86.1	54.6	1.58	169.1	63.8	2.65	337.7	90.2	3.74
wupwise	72.9	53.2	1.37	143.6	62.4	2.30	296.0	88.9	3.33

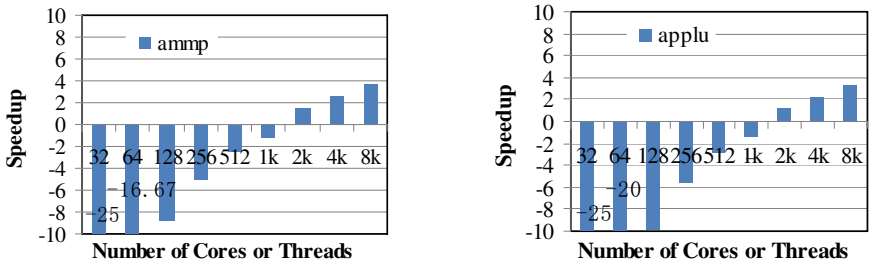

FIGURE 9. Speedups of GPU vs. CPU implementation with different number of cores or threads being simulated

6.2. Speedup of GPU vs. CPU for MIX2 and MIX3 workloads. We show the running times and speedups for *MIX3* workload in Table 5. Similar as for *MIX1* workloads, the GPU based parallel implementation has no performance benefit when the simulated cores or threads are less than 4K, and maximum speedup at 8K cores' case is more than two times. The reason of bad performance is because

TABLE 4. Characteristics of the kernels running on GTX75 GPU

Parameter	Lookup	Update	Note (Maximum Value)
Threads/Blk.	128	128	512
Registers/Thread	14	10	# of reg. / # of th. in one SM
Shared Mem./Blk.	36B	36B	Size of shared mem. / # of blks
Active Threads/SM	1024	1024	1024
Active Warps/SM	32	32	32
Active Blocks/SM	8	8	8
Occupancy of SM	100%	100%	Act. warps / Max. # of warps

TABLE 5. Running time (s) and speedups for MIX3 workload

# of C./Th.	32	64	128	256	512	1k	2k	4k	8k
CPU(s)	1.8	2.4	3.5	6.5	14.2	34.7	79.3	166.6	338.5
GPU(s)	77.2	76.6	76.9	77.0	78.6	82.0	88.2	106.3	151.2
Speedup	0.02	0.03	0.05	0.08	0.18	0.42	0.90	1.57	2.24

TABLE 6. Speedup comparison for three types of workloads

# of C./Th.	32	64	128	256	512	1k	2k	4k	8k
MIX1	0.04	0.05	0.11	0.19	0.38	**0.75**	1.45	2.44	3.48
MIX2	0.03	0.04	0.05	0.10	0.22	**0.54**	1.20	2.20	2.97
MIX3	0.02	0.03	0.05	0.08	0.18	0.42	**0.90**	1.57	2.24

the workload has very irregular behaviors among the composed programs, and at each time of launching a CUDA kernel many simulated instructions are not branch thus waist a lot of running time and transmission bandwidth. This suggests us only transferring the branch instructions when simulating this type of workload. Due to the implementing complexity, we leave this as our further work.

In Table 6, we compare the speedups for three types of workloads. From the table, it is clear to see that with a more regular behavior in the workload the GPU based implementation gets a higher performance speedup. And for multi-core processor simulation if the GPU based implementation can successfully reduce the transmission overhead between CPU and GPU, then the GPU one can get performance speedup, otherwise the researchers should use CPU platform or seek other schemes, FPGA for example, to help shorter the simulation time. For many-core simulation, researcher should also carefully organize the data and reduce the transmission time in order to obtain a better performance speedup.

6.3. Sensitivity to the instruction issue rate in the simulated cores. In Table 7, we compare the speedups of GPU implementation with different instruction issue rates in the composed cores for three types of workload. These results show the potential improvement on simulation time for the processor equipped with multiple instructions issue rate, out-of-order complex cores. From the table, when the issue rate changes from one instruction to four instructions per cycle the GPU based implementation gets a better performance speedups, especially for *MIX2* and *MIX3* workloads when the simulated cores or threads are more than 512. The maximum speedups are close to 9 and 7.2 times respectively. The reason is because that at four instructions issue rate the possibility to have a branch instruction in a cycle is much higher than that at the one instruction issue rate due to a basic block

TABLE 7. Comparison of speedups with different instruction issue rates (1 means one inst/cycle, and 4 means four inst/cycle)

# of C./Th.		32	64	128	256	512	1k	2k	4k	8k
MXI1	**1**	0.04	0.05	0.11	0.19	0.38	**0.75**	1.45	2.44	3.48
	4	0.16	0.23	0.38	**0.66**	1.19	3.92	6.14	8.34	10.34
MXI2	**1**	0.03	0.04	0.05	0.10	0.22	**0.54**	1.20	2.20	2.97
	4	0.08	0.12	0.18	0.33	**0.72**	1.78	3.98	6.48	8.84
MIX3	**1**	0.02	0.03	0.05	0.08	0.18	0.42	**0.90**	1.57	2.24
	4	0.08	0.12	0.16	0.28	**0.67**	1.61	3.42	5.71	7.19

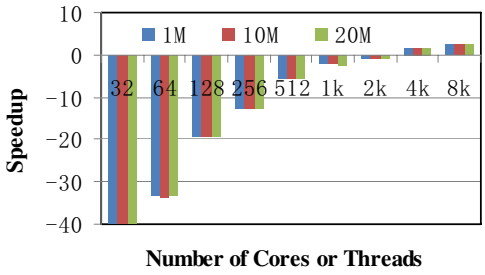

FIGURE 10. Speedups of GPU based parallel branch prediction with different number of simulated instructions for each program

in a program typically including four to seven instructions, and thus the CUDA kernels do not suffer from the useless data transmission between CPU and GPU as in the single issue rate case and spend most of the time on effective works. For *MIX1* workload, the GPU gains better performance when the number is greater than 256, and maximum speedup is 10 times due to the less time spending on data transmission.

6.4. Sensitivity to the number of simulated instructions. Figure 10 shows the speedup sensitivities of our parallel branch prediction to the number of instructions being simulated for each program in a workload. The workload is *MIX3* type. From the figure, when the number of simulated instructions ranges from one million to twenty millions, the speedups do not have obvious changes for all the cases when the number of simulated cores or threads varies from 32 to 8K. This helps us quickly finding the speedup results in our experiment without waiting long time to simulate too many instructions, and also verify the correctness of our experimental results using the methodology in Section 5.

6.5. Discussion. From the above experimental results, we can see that with the number of simulated cores or threads increasing the speedup of GPU implementation increases continuously. This implies the effectiveness of improving the CUDA threads parallelism could help improving the simulation performance. And, the linear increasing curve of the performance speedup also shows the good scalability of our GPU implementation. In addition, to further faster the simulation speed, more attention to the GPU optimization techniques and hardware features should be paid.

7. Related Work

As described in Section 1, researchers seek software [4-6] and hardware [7-9] schemes to help future multi/many-core processor simulation. Unfortunately, the software based schemes suffer from the long simulating time in CPU platform when the simulated cores are more than thirty-two, and the hardware schemes, although have fast simulation speed, but have little scalability due to the limited hardware resources on chip.

In industry and academic research literature, branch prediction has got long time attention. Most of the researches focus on how to improve the predicting accuracy [1-3], and less work has been done on how to simulate massive parallel branch predicting for future large scale multi/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 [22-23], such as in GPGPU [13]. Most of applications in GPGPU are regular computation, and very few works have been done for irregular applications, microprocessor simulation for instance.

[24] develops a trace based cache simulator on GPU platform, and gained 2.44x of performance improvement over CPU platform. Because it exploits the set-partitioning as the source of CUDA thread parallelism, it can not be used for general multi/many-core processor simulating except for shared last-level cache simulating in multicore. This work is the first work that targets the GPU on future multi/many-core processor simulation, especially the parallel branch predictor simulation.

8. Conclusion

Performance simulation for multi/many-core processor is an important process to help making design decision of the microarchitecture and organization. To do this, researchers reply on software or hardware schemes. The software schemes lose the effectiveness when they use the host CPU to simulate relative more cores, whereas the hardware schemes are faster but have little scalability.

This paper investigates how to map an irregular application, parallel branch predicting for multi/many-core microprocessor on GPU platform using the NVIDIA CUDA programming model. It verifies the correctness of the GPU implementation and obtains the simulation speedup over the CPU implementation for three different types of workloads.

The experimental results show that (a) when the simulated cores or threads are not too many the CPU implementation is faster than the GPU based one, (b) when the number is greater than 1K the GPU gains simulation speedup over CPU, (c) the simulation speedups of GPU increase with the simulated instruction issue rate, and (d) for the parallel branch prediction the simulation speedups are not sensitive to the number of simulated instructions.

References

[1] Seznec A (2005) Analysis of the OGEHL predictor, In Proceedings of the 32th international symposium on computer architecture: 394-405, Madison

[2] Seznec A (2007) A 256 Kbits L-TAGE branch predictor. Journal of Instruction-Level Parallelism Special Issue: The Second Championship Branch Prediction Competition

[3] Seznec A (2011) Storage free confidence estimation for the TAGE branch predictor. In Proceeding of 17th international symposium on high-performance computer architecture, San Antonio, Texas

[4] Sandeep B, Rama S (2006) CMP-SIM: an environment for simulating chip multiprocessor (CMP) architectures. University of Texas at Dallas

[5] Sharkey J (2005) M-Sim: A flexible multithreaded architectural simulation environment. Technical Report CS-TR-05-DP01, Department of Computer Science, State University of New York at Binghamton
[6] Donald J, Martonosi M (2006) An efficient, practical parallelization methodology for multi-core architecture simulation. IEEE Computer Architecture Letters 5(2): 14-14
[7] Chiou D, Sunwoo D, Kim J, Patil N, Reinhart WH, Johnson DE, Xu Z (2007) The FAST methodology for high-speed SoC/computer simulation, Proceedings of the 2007 IEEE/ACM international conference on computer-aided design: 295-302, NJ, USA
[8] Bhattacharjee A, Contreras G, Martonosi M (2008) Full-System chip multiprocessor power evaluations using FPGA-Based emulation, In Proceeding of the 13th international symposium on Low power electronics and design: 335-340, New York, USA
[9] Liu GX, Li GH, Gao P, Qu H, Liu ZH, Wang HX, Xue YB, Wang DS (2010) Cycle-Accurate 64+-Core FPGA-Based hybrid simulator. In Proceeding of WARP - 5th annual workshop on architectural research prototyping
[10] He LQ, Zhang GY (2009) Parallel branch prediction on GPU platform. High Performance Computing and Applications, Lecture Notes in Computer Science, Springer Berlin Heidelberg, LNCS 5938: 153-160
[11] Burger D, Austin AD (1997) The SimpleScalar tool set, Version 2.0. ACM SIGARCH Computer Architecture News 25(3): 13-25
[12] NVIDIA GeForce GTX75. NVIDIA Corporation Web. http://www.nvidia.com/object/product_geforce_gtx_275_us.html. Accessed 30 Dec. 2010
[13] GPGPU. GPGPU Organization Web. http://gpgpu.org/. Accessed 26 Dec. 2010
[14] NVIDIA CUDA? Programming Guide. Version 2.3. 7/2009
[15] Hennessy JL, Patterson DA (2007) Computer architecture: a quantitative approach. Fourth Edition
[16] Benini L, De MG (2002) Networks on chips: a new SoC paradigm. IEEE Computer Society 35(1): 70-78
[17] Susmit B, Diana F, Alan S, Ryan D, Timothy S, Frederic TC (2009) Multi-Execution: multi-core caching for data-similar executions. In Proceedings of the 36th international symposium on computer architecture, Austin, Texas
[18] Doug RD, Burger D, Keckler SW, Austin T (2001) Sim-alpha: a validated, execution-driven alpha 21264 simulator. Technical Report TR-00-04, Department of Computer Sciences, University of Texas at Austin
[19] Henning J (2000) SPEC CPU2000: measuring CPU performance in the new millennium. IEEE Computer 33(7): 28-35
[20] Tullsen DM, Eggers SJ, Levy HM (1995) Simultaneous Multithreading: maximizing on-chip parallelism. Proceedings of the 22th annual international symposium on computer architecture:392-403, Washington USA
[21] Intel Hyper-Threading Technology. Intel Corporation Web. http://www.intel.com/technology/platform-technology/hyper-threading/index.htm. Accessed 26 Dec. 2010
[22] Kerr A, Campbell D, Richards M (2009) QR decomposition on GPUs. In Proceeding of 2nd workshop on GPGPU '09, Washington, D.C., USA
[23] Gulati K, Croix JF, Khatri SP, Shastry R (2009) Fast circuit simulation on graphics processing units. In Proceeding of design automation conference:403-408
[24] Wang H, Gao XP, Long X, Wang ZQ (2009) GCSim: a GPU-based trace-driven simulator for multi-level cache. Advanced Parallel Processing Technologies, Lecture Notes in Computer Science, Springer Berlin Heidelberg, LNCS 5737: 177-190

College of Computer Science, Inner Mongolia University Huhhot, Inner Mongolia 010021 P. R. China

E-mail: liqiang@imu.edu.cn zhang03_11@163.com and csjiangjd@gmail.com
URL: http://arch.imu.edu.cn/hlq