A Translation Framework for Executing the Sequential Binary Code on CPU/GPU Based Architectures

Erzhou Zhu, Haibing Guan, Guoxing Dong, Yindong Yang, Hongbo Yang
Department of Computer Science and Engineering & Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiaotong University, Shanghai, China
Email: {ezzhu, hbguan, idalang, yasaka, yanghongbo819}@sjtu.edu.cn

***Abstract*—The method of using DBT (dynamic binary translation) to execute the source ISAs binary code on target platforms has been perplexed by low overhead for many years. GPU as a many-core processor has tremendous computational power. Employing GPU as a coprocessor to parallel execute the hot spot of binary code hold a great promise of substantially reduce the overhead of DBT. This paper presents a novel translation framework for constructing the virtual execution environment aiming at accelerating the process of DBT on CPU/GPU based architectures. With parallelizable parts (hot spots) of binary code and their related information, the framework converts the sequential code into PTX form and executes them on GPUs. Under the framework, we need not to rewrite the source code, and the binary compatibility issues between different GPUs are also resolved properly. Experimental results on several programs from CUDA SDK Code Samples and Parboil Benchmark Suite show that the framework can significantly improve the performance, usually have 10X speedup on average compared to X86 native platforms. Especially, when the scale of input become larger, the performance becomes even better.**

***Index Terms*—GPU, Virtual Execution Environment, Parallelization, CUDA, Dynamic Binary Translation, PTX**

I. INTRODUCTION

Dynamic binary translation (DBT) is a commonly used technology in virtualization area. It converts binary code of one ISA (source binary code) to the binary code of another ISA (target binary code) during the runtime. And the target code will be executed right after the conversion. Performance of DBT system, including the overhead of translating procedure and the execution of target code, has perplexed the researchers for many years. Many optimizations have been used to improve this point, such as translation block chaining, form large translation blocks (superblocks), reordering translated instructions to improve pipeline performance, borrow optimization techniques from conventional compilers. In fact, most present DBT systems can reduce translating costs into an acceptable range. Hence, main attention is laid on bringing down the executive time of target code.

Corresponding author: Haibing Guan (hbguan @sjtu.edu.cn)

Purely software optimization technique is not ideal for improving translation speed, thus software combines hardware optimization technique is a good choice. Although using the accelerating hardware outside of the personal computer could gain good speedup, but modern people emphasizing portability, they couldn't tolerate taking the accelerating hardware all the time. So how to fully use the inside PC component to accelerate the translation process is an inevitable trend.

The programmability and large-scale computing ability of GPU (Graphic Process Unit) has been proved most useful in computation-intensive applications, and there has been considerable interest in general purpose computation on GPUs (GPGPU) [1][2][3]. In many cases, performing general purpose computation in graphic hardware can provide a significant advantage over implementing on traditional CPUs. By this reason, employing GPU as a coprocessor to parallel execute the hot spot of binary code hold a great promise of substantially reduce the overhead of DBT.

There some issues to be resolved on the march of efficiently using of GPU to accelerate the process of DBT. First and the most important issue is to detect the hot spots of the sequential binary code. This issue is a part work of binary analysis [4][5], and has achieved much progress by many frameworks, such as Pin [6], DynamoRIO [7], and Valgrind [8]. One common example of hot spot is the heavily executed loops in the sequential binary code. Through binary analysis, information such as loop body, loop indices, loop bound and the dependence relationships can be obtained. Secondly, with the hot spot and its related information, we must convert the sequential hot spot into parallel fashion. With the recent decades, the technique of auto-parallelization has been significant developed. The strategy of auto-parallelization based on polyhedral model [9][10] has been successfully adapted by many projects for its easily transformable and be able to map the executable codes to multi-core architectures. Thirdly, since CPU/GPU based architectures are heterogeneous platforms, the problem of coordinating the CPU and GPU and enabling them to cooperate in harmony need to be resolved. Fourthly, developers are requested to rewrite the source code. Since traditional compute-intensive

doi:10.4304/jsw.6.12.2331-2340

applications are usually developed by common programming languages, such C and C++, and are compiled into binary codes that use different ISAs from the ones of GPUs. It is obliged to rewrite the hot spot of source binary code, and convert it to the forms of code that are appropriate executed in GPU. The last issue is binary incompatibility [11][12]. It is well known that, GPU hardware are evolving rapidly, this situation poses the problem that the codes that developed and tuned for one generation are not compatible with the next generation. Furthermore, different kinds of GPUs are also incompatible. In contrast with source code incompatible, a much tougher problem is binary incompatibility, which means that the application statically compiled with certain compiler may not work on the platform without the special GPU, let alone another kind of or another generation of GPU.

In order to deal with these issues, we construct and implement a virtual execution environment (GXBit we called) aiming at efficiently executing the sequential binary code in CPU/GPU based architectures. By employing the strategy of dynamic binary translation as well as dynamic-static combined binary analysis, GXBit enables the CPU and GPU to cooperate naturally and therefore applications can be efficiently executed on them. The introduction of GXBit will be described in the follow section.

In this paper, we specifically introduce the core component on constructing the GXBit—the translation framework [29]. The framework first transforms the nested loops within X86 binaries into PTX [13] code, then ports the generated PTX code to CUDA [14]. PTX defines a virtual machine and ISA for general purpose parallel thread execution and could be translated at install time to the target hardware instruction set. Speedup can be achieved via launching GPU to execute PTX code instead of running the corresponding sequential code on CPU.

The contributions of this paper can be categorized as follows:

a) A virtual execution environment (GXBit). GXBit can automatically execute the sequential binary code on CPU/GPU architectures.
b) A translation framework for GXBit. The framework can transform the X86 ISA to PTX ISA, and can also deal with the transfer issue between CPU memory and GPU memory.
c) An intermediate representation (GVInst). GVInst brings the gap between sequential code and parallel code in the process of translation.

II. Overview Of GXBit

Before implementing the translation framework, it is needed to introduce the virtual execution environment (GXBit) that the translation framework works for. Actually, GXBit is a DBT system derives from the multi-sources and multi-targets DBT—Crossbit [28]. However, there are at least three main differences between GXBit and Crossbit. First of all, the execution mode of GXBit is two-phase. The second difference is the parallel parts.

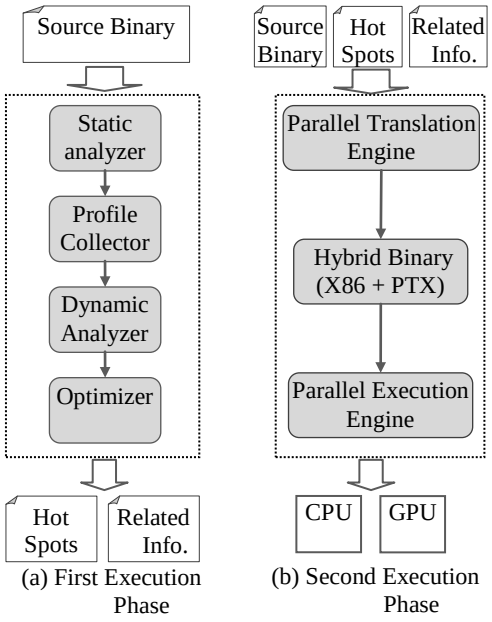

Figure 1. The Workflow of GXBit.

GXBit first extracts the hot spots from source binary code, and then converts these spots to the form that can be recognized and be parallel executed by GPU. Finally, the execution engine of GXBit is also different from Crossbit's, because GXBit needs to execute the paralleled parts on GPU.

Figure 1 is the workflow of GXBit. As the figure describes, the purpose of the first phase is to extract hot spots and their related information from source binary code. Specifically: a) The static analyzer scans the .text section of the input binary to find out all nested loops before the partial execution. These loops are recorded as candidates of hot spots. b) GXBIT starts a partial run for source binary. During the runtime, profile collector inserts additional GVInst after each memory access operation in the VBlocks (the basic translation unit in GXBit) of every candidate. When the translated code is being executed on target, all the accessed memory address can be monitored. c) Once the outmost-level of the candidate nested loop has been executed, the dynamic analyzer uses the monitored information to build a polyhedral model and determines whether the current nested loop has memory dependences between iterations. If there is no dependence, this candidate is regarded as a hot spot and can be parallelized on GPU. d) The optimizer performs certain optimizations to the VBlocks of the hot spots according to specific GPU and dumps all the profiled information and the optimized VBlocks to files.

In the second phase, GXBit utilizes the information collected from the first phase to accelerate the execution of source binary code by porting the hot spots to GPU. This process can be described as: a) GXBIT loads the source binary, profiled information and optimized VBlocks from files, and gets the entry and exit addresses of the hot spots. b) GXBIT starts a whole execution procedure to run the source binary. When the execution flow gets into a hot spot, the parallel translation engine is

triggered to transform the optimized VBlocks to PTX code. Thus forms a hybrid binary, which contains both x86 binary and PTX intermediate code. c) The parallel execution engine will let the hybrid binary run on a CPU/GPU environment, and handle the memory coherence on both architecture.

III. Intermediate Representations

Since raw binary code cannot directly run on GPU, so it is urgently need to design an intermediate representation (IR) that can mask the gap between CPU binary code and GPU's code. It is well known that the code to be executed by GPUs is the kernel function, and the kernel function can be written by CUDA instruction set architecture called PTX. As an assembly form, PTX code can be easily transformed from X86 instructions. On the other hand, the CUDA driver API provides functions to support both loading and executing the PTX code on GPU. So we adopt PTX as the target language of our translation framework. In order to facilitate transforming the hot spots of the binary code to PTX code, we introduce an IR layer (GVInst) to our translation framework.

GVInst is a RISC-like instruction set that provides type safety, flexibility, low-level operations and the capability of representing the critical parts of source binary executable to be translated to the heterogeneous architectures as well as all needed information.

A. Register Architecture

GVInst defines a general-purpose register architecture for virtual machine, it consists of 8 32-bit virtual registers (s0~s7) standing for the 8 general purpose registers in X86 architecture, 8 double precision floating virtual register (f0~f7) corresponding to the floating stack of X86 and infinite 32-bit virtual registers (v0~vn).

B . Addressing Model

GVInst defines RISC style load (LD)/store (ST) instructions to access memory, and the only addressing model it supported is displacement. As we known, an X86 instruction often involves more than one memory access operations. After decoding to GVInst, both explicit and implicit memory operations that represented as LD/ST instructions can be detected.

C. Instruction Format

The basic form of a GVInst instruction is like this:

Optr [.type] Opnd1, Opnd2, [Opnd3], [Opnd4]

Optr tells the function of this instruction. Opnd2, [Opnd3], [Opnd4] are the source operation number, and Opnd1is the destination operation number. In GVInst, most of instructions must have a type field to tell the operand's data type corresponding to type-size specifier on PTX.

As a whole, the instructions in GVInst can be divided into six categories: state mapping, memory accessing, data moving, computing, control transferring and comparing instructions. Table 1 gives the concise description of these instructions.

TABLE I.
THE CATEGORIES OF GVINST

Category	Example and its semantics
State Map (GET, PUT)	GET.type v, s ;Maps a source register (s) to a virtual register (v).
Memory Access (LD, ST)	LD.type v1, (v2.imm) ;Loads the memory data (v2.imm) into virtual register (v1).
Data move (MOV, LI)	LI.type v1, imm ;Stores an immediate data (imm) to a virtual register (v1).
Computation (ADD,SUB,MUL, NOT,AND, …)	ADD.type v1, v2, v3 ;Adds two values in virtual registers(v2,v3).
Control Transfer (JMP,BRANCH, CALL)	JMP V1, imm ;Directly jumps to a memory location (v1, imm).
Comparison (CMP)	CMP.type v1,v2,v3,cc ;Compares the two values(v2, v3) according to the tag (cc), and stores the result (to V1).

The GVInst code is organized in basic block. The term of basic block here means dynamic basic block, which is slightly different from the classic definition in compiler: a basic block is determined by the actual flow of a program as it is executed. It always begins at the instruction executed immediately after a branch of jump, follows the sequential instruction stream, and ends with the next branch of jump. Since loop test will turn to basic blocks containing no statements, extract statements from loop body will be done easily.

IV. The Implementation Of The Translation Framework

As presented in the previous section, in the overall architecture of GXBit, the translation framework standing at the point of receiving the marked hot spots and their related information (written in GVInst), automatically translating these hot spots to the form of code written in PTX and launching the underlying GPU to execute the PTX code. As figure 2 shows, the framework consists of two parts: the translation module, which automatically

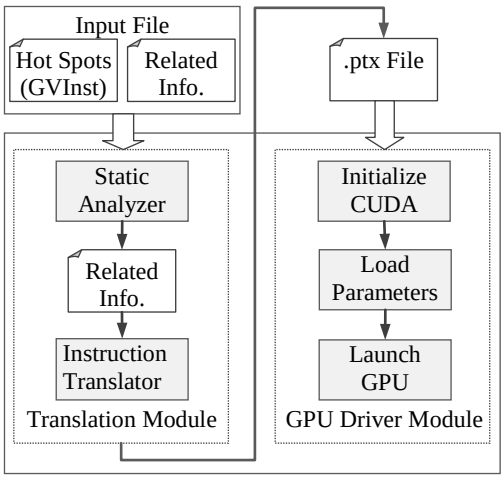

Figure 2. The Translation Framework.

generates PTX file from GVInst; and the GPU driver module, which ports the generated PTX file to GPU and returns the results. The following of this section will describe these components in detail.

A. Translation Module

As a common example of hot spot, in this section, we choose heavily executed loop as the example to demonstrate the procedure of the translation framework. Under this situation, the translation module is responsible for identifying the parts of nested loops, transforming them into PTX code and storing them into a .ptx file.

A) Static Analyzer

The static analyzer first identifies and marks various variables that appear in the loop bodies, then, based on the marked variables, extracts the parallable parts of loop body that really be executed in GPU.

a) Identify and Mark Out the Hot Spots Related Variables

It is has to point out that in GVInst only LD and ST can access memory. We add a filter which is sensitive to LD and ST to monitor the memory access. Once a memory access is detected, a variable checking process of the filter will be triggered. Our framework employs a simple mechanism to name the variables: 1) loop indices are marked as i, j, k...; 2) loop bounds are marked using "bound" prefix, this type of name has an extra field with a numeric character starting with zero. The value is incremented each time when a different loop bound is allocated; 3) other variables are marked using "var" prefix and also have a numeric field as 2).

Figure 3 shows an example for illustration. As soon as the filter detects the LD instruction in line 4 of figure 3(a), the checking process will be trigged. Here s5 stands for EBP in X86 instruction architecture set. So v29(ox0) is a memory address based on EBP with an offset of oxffffffff8. Then we look up the offset value of oxffffffff8 in the variable table, that offered by the input information file shown in the figure 2, to determine whether it is a variable. If it is, the codes will be converted to line 5 in figure 3(b), and the variable is marked as "var1".

```
1.  GET.s32     v6, s5
2.  LI.s32      v29, oxffffffff8
3.  ADD.s32     v29, v6, v29
4.  LD.s32      v1, v29 (ox0)
```

(a) Before marking variables.

```
5.  MOV.s32     var1, v1
```

(b) After marking variables.

Figure 3. Marking variables.

b) Extracting the Statements

The most executed parts of code in loop body should be extracted and be parallel executed on GPU, we call these parts of code as statements. Since for-loop occupy most situations of loops, so in the following we take the for-loop as the major situation. As described in section 3, GVInst employs basic blocks as the basic units to organize its instructions. This feature facilitates the process of extracting the statements from loop body. Under the situation of for-loops, loop tests always appear in the first several basic blocks. This information tells us that these blocks do not contain statements. Therefore, after marking the variables, we regard the basic blocks that only contain loop indices and loop bounds as the loop test part. Following this, we find loop test parts by sequentially examining the basic blocks from the beginning until meet the block with statements. On the other hand, the instructions behind the last ST instruction should be neglected. Considering there is no data dependence in loops, so instructions after the final ST instruction will not influence the results. Take all the situations in to account, the statements can be easily be extracted from the loop body.

B) Translate GVInst to PTX

Prior to introduce the detail of the translation procedure, it is needed to give an overview of the CUDA programming model. It abstracts the thread hierarchy as a grid of cooperative thread arrays (CTAs) which implement CUDA thread blocks. A CTA is an array of threads that execute the same code concurrently in a block. The code to be executed by GPUs is the kernel function. Threads in a block can efficiently communicate with each other through the on-chip shared memory. There is a limitation for the number of threads in a CTA. CTAs that execute the kernel can be batched together into a grid of CTAs. Each thread has a unique thread ID to specify its position is a CTA. In a typical 2D/3D execution domain, the threads in a block have increasing thread IDs along the X direction, and the same thread IDs along the Y and Z directions. Like the thread ID in a CTA, there are CTA IDs in a grid of CTAs and temporal grid IDs in grids as well.

The instruction translation module in figure 2 is responsible for transforming the statements (written in GVInst) into PTX code. Since the source to be translated is binary code, complex optimizations due to their high overhead are unsuitable to our framework. As a compromise, we introduce a simple strategy that translates each GVInst instruction to one or more PTX instructions in the execution sequence.

The following rules are employed to achieve the goal of translation:

a) Mapping Loop Indices to Thread IDs

Taking two nested loops for example, assume that 'i' and 'j' are the loop indices. In the CUDA programming model, 'i' can be viewed as the absolute thread ID along the Y direction, which is equal to (blockIdx.y*blockDim.y + threadIdx.y) in the CUDA. Correspondingly, 'j' is the absolute thread id along the X

direction, which equal to (blockIdx.x * blockDim.x + threadIdx.x). These values could be presented using through predefined, read-only special registers %tid, %ntid, %ctaid, %nctaid, and %gridid [13]. If there is a three nested loops, we can extend this rule into 3D thread hierarchy.

b) Mapping Mechanism of Virtual Registers and Variables

We create a one to one mapping table for recording the register allocation relationship between GVInsts and PTX. According to PTX ISA manual, registers in PTX may be typed (signed integer, unsigned integer, floating point, predicate) or untyped. Register size is also restricted; aside from predicate registers which are 1-bit, scalar register have a width of 8-, 16-, 32- , or 64-bits. Fortunately, our GVInst have defined a field to indicate the operands' type-size specifier. We allocate each variable marked in GVInsts with PTX registers as well. When translating GVInsts to PTX, we will replace the virtual registers and variable with corresponding PTX registers in the mapping table.

c) Transfer Data Values to GPU Memory

Except for loop indices, other variables will transfer their values by PTX kernel function's parameters using CUDA's global memory. We load each parameter value to its corresponding register recorded in the mapping table. In CUDA's memory hierarchy, the global memory is accessible by all threads in a context. It is the mechanism by which different CTAs and different grids can communicate. Comparing with global memory, shared memory is a per-CTA region of memory for threads in a CTA to share data. Although shared memory is much faster, using shared memory should modify the construction of the code which is too complex to handle in binary-level. If all threads could be loaded in one block (less than 512), we will prefer to choose shared memory. To take advantage of shared memory, we should modify the PTX code to ensure that each thread is responsible for load one element.

d) Wrap the Translated Code as a Kernel Function in a .ptx File

The translated code should be completed as a function. In other words, we should complement function heading for the translated code, including function name and parameter list. Furthermore, we need to review the mapping table to statics the registers used in PTX code, and initialize them at the beginning of the function.

B. GPU Driver Module

The workflow of the translation framework turns to the active driver module right after the .ptx file is generated. The GPU driver module is responsible for managing the execution of ported hot spots on GPU. Since the CUDA driver API providers a better interface for handling the assembly-like PTX code than GPU runtime API, our framework adapts the CUDA driver API to implement

```
for (i=0; I < height, i++)
    for (j =0; j < width; j++)
    {
                        …
                        }
```

Figure 4. An example of for-loop.

the GPU driver module. The following steps are used to launch GPU:

A) Initialize CUDA

The purpose of the initialization is to provide an executable environment for running PTX code. For the sake of this purpose, cuInit() and cuCtxCreate() must be called at the first time to initialize the GPU and generate CUDA context respectively. Then cuModuleLoad() and cuModuleGetFunction() are used to load .ptx file and return a function handle. At last, calls cuMenAlloc() to initialize the memory of GPU, and utilize cuMemcpyHtoD() to copy the data from CPU memory to GPU memory. The size of GPU memory to be allocated can be calculated from the input file of loop information. As an example, the memory size to be allocated shown in figure 4 is: width*height*sizeof(float).

B) Load parameters

This step loads the values of formal parameters that appear in PTX functions. The cuParam*() function family is used for loading parameters. In the process of parameter loading, we should consider the following issues: a) the order of loaded values should be the same as the order of the PTX functions' formal parameters. b) The offset of each parameter should be adjusted to meet its alignment. c) Since some variables' value and base address are directly stored in general purpose registers of X86 architecture after optimization, so the 8 general purpose registers should also be loaded as parameters.

C) Launch GPU

At this stage, the first thing is to determine the scale of the computation, such as determining the number of threads per block and the number of blocks per grid. The total number of threads could be calculated by using loop indices and loop bounds. As an example, the total number of threads of the for-loop in figure 4 is width*height. The cuFunSetBlockShape() and cuLaunchGrid() are used to distribute the GPU computing resources and launch GPU

```
cuFuncSetBlockShape ( cuFunction, 16, 16, 1 );
cuLaunchGrid ( cuFunction, ( width +dimBlock.x
– 1 ) / dimBlock.x, \( height + dimBlock.y – 1 ) /
dimBlock.y );
```

Figure 5. Example of setting block and grid with CUDA driver API.

```
/************************************
*Note:
*In this example, we assume that:
* %r2 = blockIdx.y*blockDim.y + threadIdx.y
* %r3 = blockIdx.x*blockDim.x + threadIdx.x
* %r13 = height
* %r14 = width
************************************/
setp.ge.u32 %p0, %r2, %r13;
@%p0 exit;
setp.ge.u32 %p0, %r3, %r14;
@%p0 exit;
```

Figure 6. Instructions Added to avoid redundant threads.

to perform computation respectively. Figure 5 shows the example (appeared in figure 4) of using these API to set block size and grid size.

The total number of blocks should be the integer times of 16, this because the number of 16 can increase the efficiency of GPU memory accessing. Actually, in CUDA programming model, there are 16 threads in a half-wrap. Under this situation, only one data transfer operation is enough to accomplish the task of the 16 threads of a half-wrap to access memory. However, if the number is not the integer time of 16, the threads that have been executed will be exceed the number really needed. To overcome this shortcoming, we insert two exit instructions into the generated .ptx file, as shown in figure 6.

When finished running the kernel function on GPU, we call cuMemcpyDtoH() to return the results back to CPU memory.

V. PERFORMANCE EVALUATION

This section presents the performance evaluation of the translation framework. Table 2 shows the hardware and software configurations of our experimental environment.

We evaluate our translation framework by running 2 applications from CUDA SDK Sample [16] and 2 test case from Parboil Benchmark Suite [17]. As a whole, we examine the effectiveness of the translation framework from two aspects: a) comparing the kernels' performance under different environments; b) deploying the translation framework on a completely virtual execution environment---GXBit.

TABLE II.
HARDWARE AND SOFTWARE CONFIGURATION DETAILS

A) HARDWARE CONFIGURE

CPU	4 * Intel Xeon 5110 clocked at 1.60Ghz (1066Mhz FSB), 4M L2 cache
RAM	8GB, DDR2-667
GPU	NVIDIA GeForce GTX 260, 896MB DRAM, 27 multiprocessors, clocked at 1243MHz

B) SOFTWARE CONFIGURATION

OS	Linux with kernel 2.6.18
Compiler	GCC3.4.3 NVCC2.3
CUDA version	2.3

A. Evaluating the Performance of Kernels

This experiment examines the effectiveness of our translation framework by comparing kernels performance of different programs. We present four experimental results, the two programs that have been chosen from CUDA SDK Sample are Matrix Multiplication and ConvolutionFFT2D, and the other two programs that have been chosen from Parboil Benchmark Suite are MRI-FHD and MRI-Q.

The following tables show the results of running CUDA SDK Sample programs.

Table 3 shows the performance data of running the kernel function in Matrix Multiplication with different input size: 128*128, 512*512, and 1024*1024.

Table 4 shows the results of ConvolutionFFT2D's kernel. This application uses Faster Fourier Transformation (FFT) algorithm to implement a Fourier-based general 2D convolution, which is more efficient than the straightforward. Similar to Matrix Multiplication, we also set three different input data size: 1000*1000, 2000*2000, and 4000*4000.

From the experimental data demonstrated in the above tables, we can conclude that after transforming the hot spots to PTX code by the translation framework: a) The kernel functions achieve consistently much better performance on GPU than the code directly running on CPU (the "native" column in the tables). b) The kernel function that runs on GPU also exhibit better performance than the ones optimized with –O3 flag (the "Native-O3" column in the tables). c) The performance of the experiments exhibits better along with the increasing input scale of data size.

In the experiment, we also compare the performance of running two different versions of the generated PTX code on GPU: the one is generated by our translation framework, and the other one is generated by NVCC (the "NVCC" column in the tables). With a tinge of regret, our translation framework cannot achieve the same performance as NVCC did. The code form of the input

TABLE III.
PERFORMANCE COMPARISON OF MATRIX MULTIPLICATION KERNEL

Matrix Size	Native (ms)	Native –O3 (ms)	Translation Framework (ms)	NVCC (ms)
128*128	31	7	0.44	0.05
512*512	1960	450	21.45	1.61
1024*1024	40620	17400	171.02	12.46

TABLE IV.
PERFORMANCE COMPARISON OF CONVOLUTION FFT2D KERNEL

Image Size	Native (ms)	Native –O3(ms)	Translation Framework (ms)	NVCC (ms)
1000*1000	1570	290	35.43	1.99
2000*2000	6312	1180	141.12	7.95
4000*4000	25240	4690	563.64	33.24

TABLE V.
PERFORMANCE COMPARISON OF CONVOLUTION MRI-FHD KERNEL

Input Size	Native (ms)	Translation Framework (ms)	NVCC (ms)
Small (32_32_32_dataset)	13065	47.97	5.27
Large (64_64_64_dataset)	70340	255.17	23.09

TABLE VI.
PERFORMANCE COMPARISON OF CONVOLUTION MRI-Q KERNEL

Input Size	Native (ms)	Translation Framework (ms)	NVCC (ms)
32	13078	47.86	3.26
64	69912	254.73	11.96

may be the major reason behind this phenomenon. The input of NVCC is the source code, so the PTX code generated by NVCC can efficiently utilize the underlying GPU resources. However, the input of our translation is binary code. Take this reason in to consideration, we still satisfy with the results.

We also employed our framework to generate kernels in Parboil Benchmark Suite: MRI-FHD and MRI-Q. Table 5 and table 6 show the experimental results of running these programs. From these experiments we further prove that the performance our framework works better than that of the native platforms.

Finally, to better understand the achieved performance of our translation framework, we present (figure 7) the comparison of the performance between "native" and "translation framework" that run hot spots (or kernels) respectively. The data shown in the figure is derived from equation (1).

$$Speedup = \frac{Execution\ time\ of\ native\ platform}{Execution\ time\ of\ our\ framework} \quad (1)$$

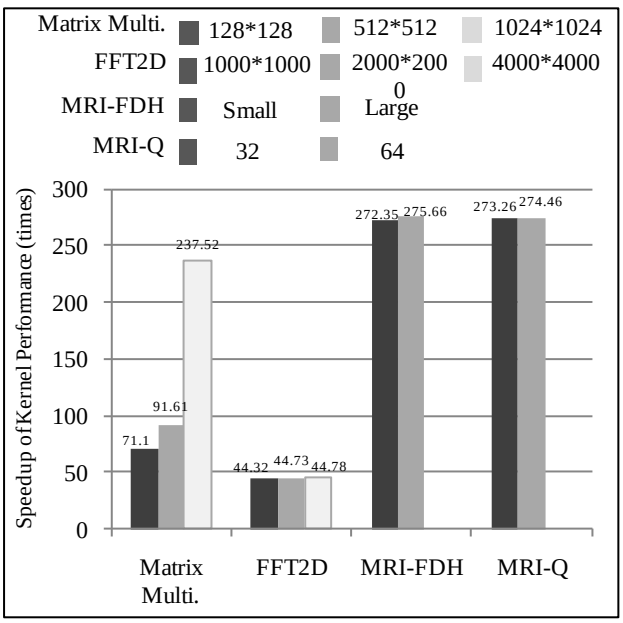

Figure 7. Performance of the kernels gained by the translation framework (Compared to Native Platform).

In equation (1), the execution time of native platform refers to the execution time of hot spots that directly run on CPU platform; the execution time of our framework refers to the execution time of hot spots (or kernel functions written by PTX code) that run on GPU. Since our translation framework can fully exploit the large-scale computing ability of GPU (to parallel execute the hot spots), we gain consistently better performance (up to hundred times of speed up) over the native platform.

B. The Performance of the Framework in Really Environment

In order to demonstrate the feasibility of our translation framework, we evaluate the performance of deploying the framework on a complete virtual execution environment---GXBit. As mention in section 2, GXBit is a virtual execution environment based on CPU/GPU architectures. It is designed for supporting existed binary executable written by sequential language to take advantage of GPU to accelerate the execution of hot spots automatically. Figure 8 shows the performance of the programs running on GXBit over the native ones. The y-axis represents the times of speedups of the programs beyond native. The times of speedups are derived from equation (2):

$$Speedup = \frac{Execution\ time\ of\ programs\ running\ on\ native\ platform}{Execution\ time\ of\ programs\ using\ GXBit} \quad (2)$$

In equation (2), the execution time of native platform refers to the execution time of applications that directly run on CPU platform; the execution time of programs using GXBit refers to the execution time of the applications that run on CPU/GPU based platform. By employing our translation framework, GXBit can fully exploit the large-scale computing ability of GPU. When an application runs on GXBit, the regular process is performed by CPU, and the execution will transfer to

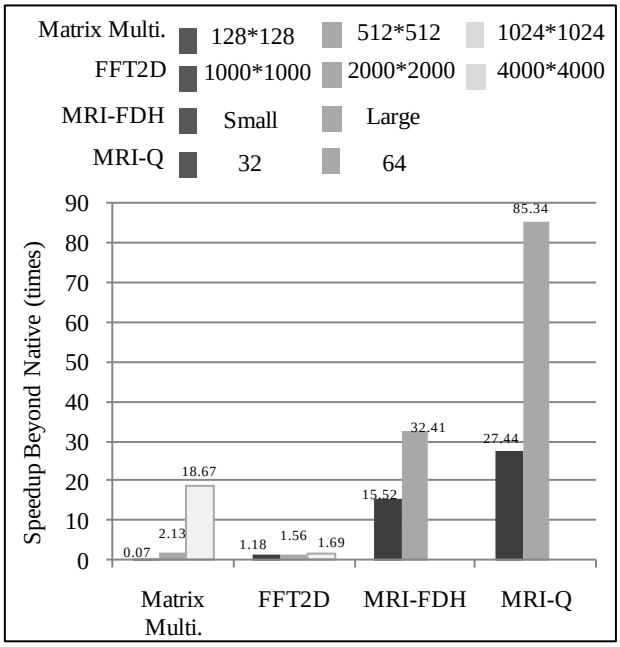

Figure 8. Performance of the programs running on GXBit compared with running on Native.

GPU when a parallel hot spot is detected. The execution will turn back to CPU for the following process right after the GPU have finished execution the hot spot.

The performance would be promoted when the number of the speedup is greater than one. As the figure shows, the performance is improved in most of cases running on GXBit, except for the Matrix Multiplication with 128*128 input data size. In theory, we can gain much performance by employing our framework to port the hot spots to GPU. However, as a virtual execution environment, the binary-level input nature of GXBit is one of reason that slows the process of execution. Additionally, in order to run the applications on CPU/GPU based architectures, GXBit needs extra operations to generate hybrid binary code and transfer data between CPU memory and GPU memory. Therefore, if the performance we gained from GPU cannot compensate for the consumed performance of the above two situations, the overall performance of GXBit would be degraded. This is the direct reason why the speed of running Matrix Multiplication with 128*128 input data size on GXBit is slower than the one running on the native platform.

Figure 8 also shows that the speedups of running ConvolutionFFT2D on GXBit are not ideal even close to the performance of native ones. This because the improvement gained from GPU is eliminated by the overhead of binary translation procedure of GXBit of random initializing the input data array. Actually, GXBit plays a role of binary translator when executing the program except for the parts of kernel functions.

VI. Related Works

The powerful computing ability and the explicit programming environment of GPU have attracted much attention on transforming programs written with other languages to CUDA to obtain more performance improvements. However, the programming environment provided by CUDA is different from the traditional ones. In order to employ the powerful GPU, developers are still needed to rewrite the source code to bring the gaps between the architectures of CPUs and GPUs. For the purpose of avoiding rewrite the source code, many works have emerged to support the CUDA backend. Baskaran [18][25] designed and implemented a transforming framework with an aim to automatically transform affine C programs into CUDA. Lee [19] also developed a compiling framework to complete-automatic transform OpenMP to CUDA. Par4all [20] is a new tool that can translate C and FORTRAN programs to CUDA to accelerate to speed of programs executing. For supporting multi-core architectures, Bondhugula [21] implemented a framework to automatically generate OpenMP parallel code from C programs. However, these works are based on source code and the program analysis techniques based on source code are so mature that they are easier to implement. Considering our translation framework is working on binary-level, so there are many differences from them.

It is also needed to pay much attention on resolving the asymmetric issues produced by the heterogeneous architectures. It is critical to fully utilize the underlying hardware resources on the march of achieving high performance. There are many researches on avoiding the problems caused by asymmetric memory system of the heterogeneous platforms. Gelado [22] gave an asymmetric distributed share memory model for heterogeneous parallel systems. Bratin [23] designed a programming model for heterogeneous X86 platforms. Nathan [24] and Scott [26] did the similar things to bring the gaps of architectures between the accelerators and the host CPUs. However, both of their works are based on their special designed hardware. These special hardware-based interfaces between the bus of host-ends and the accelerators can avoid the problems brought by the heterogeneous memory systems. Yang [27] and Baskaran [15] have derived several methods to resolve the issues of memory optimizing on CUDA. Most of the above works are not limited to CPU/GPU architecture, and the inputs of these platforms are also not limited to binary-level. However, the ideas and the methods behind them have given us much help on designing our translation framework.

VII. Conclusions

In this paper, we presented a novel translation framework for constructing the virtual execution environment with an aim to accelerate the process of DBT on CPU/GPU based architectures. With the input information of binary-level hot spots and their related information, the translation framework can automatically transform the sequential binary code to PTX code, and execute them on GPUs. By introducing the intermediate representation---GVInst, the issues of rewriting source code and the binary compatibility between different GPUs were properly resolved. In the process of translation, by using the mechanisms of identifying and marking variables, our framework efficiently extracted the statements from the loop bodies, then translated these statements into PTX form, and stored them into a .ptx file. In the stage of launching GPU, we employed CUDA driver API other than CUDA runtime API on the reason that the former offers a better level of controlling the assembly-like PTX code. This API also provides us many useful functions for resolving the memory management issue between CPU and GPU. Experiments for benchmark programs have shown that our translation framework achieved better performance than the native ones. Especially, the larger scale of the input data, the higher performance we gained.

In the future, we will consider the following problems: a) the translation mechanisms that affect on the performance should be optimized and be further studied. b) The issue of GPU memory utilization and computation workload distribution should be investigated. c) Extending the framework so as to support the AMD/ATI stream. Finally, we will improve and perfect the framework, and enabling it to support “real-life” situations other than only focusing on benchmarks.

ACKNOWLEDGMENT

This work was supported by The National Natural Science Foundation of China (Grant No. 60970108, 60970107), The Science and Technology Commission of Shanghai Municipality (Grant No. 09510701600, 10ZR1416400, 10DZ1500200, 10511500102), IBM SUR Funding and IBM Research-China JP Funding.

REFERENCES

[1] K.Fatahalian, J.Sugerman, and P.Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication", Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2004, pp.133-137.

[2] N.K.Govindaraju, S.Larsen, J.Gray, and D.Manocha, "A memory model for scientific algorithms on graphics processors", Proceedings of the 2006 ACM/IEEE conference on Supercomputing, 2006.

[3] General-Purpose Computation Using Graphics Hardware. http://www.gpgpu.org/.

[4] Cristina Cifuentes and K. John Gough, "Decompilation of Binary Programs", SOFTWARE: PRACTICE AND EXPERIENCE, VOL. 25(7), 1995, pp.811-829.

[5] Tipp Moseley, Daniel A. Connors, Dirk Grunwald, Ramesh Peri, "Identifying potential parallelism via loop-centric profiling", Proceedings of the 2007 International Conference on Computing Frontiers, 2007, pp.143-152.

[6] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, Vijay Janapa Reddi and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation", Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005, pp.190-200.

[7] Nicholas Nethercote and Julian Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation", Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, 2007, pp.89-100.

[8] Derek Bruening, Timothy Garnett, Saman Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization", International Symposium on Code Generation and Optimization, 2003, pp. 265-275.

[9] C. Ancourt and F.Irigoin, "Scanning polyhedral with do loops", Symposium on Principles and Practice of Parallel Programming, 1991, pp.39-50.

[10] U. Bondhugula, A. Hartono, J. Ramanujan, and P. Sadayappan, "Apractical automatic polyhedral parallelizer and locality optimizer", Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, 2008, pp.101-113.

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

[12] Nathan Clark, Jason Bolme, Micheal 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, pp. 272-283.

[13] "PTX: Parallel Thread Execution ISA", Version 2.1, 2010.

[14] "NVIDIA CUDA Programming Guide", Version 3.1, 2010.

[15] Muthu Manikandan Baskaran, J. Ramanujan, Sriram Krishnamoorthy, "Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories", Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, 2008, pp.1-10.

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

[17] "Parboil benchmarkSuite", http://impact.crhc.illinois.edu/parboil.php.

[18] Muthu Manikandan Baskaran, J. Ramanujan, "A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs", Proceedings of the 22nd annual international conference on Supercomputing, 2008, pp.225-234.

[19] Seyong Lee, Seung-Jai Min, Rudolf Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization", Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, 2009, pp.101-110.

[20] Par4All, http://www.par4all.org/

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

[22] Isaac Gelado, Javier Cabezas, John Stone, Sanjay Patel, Nacho Navarro and Wen-mei Hwu, "An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems", Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems, 2010, pp.347-358.

[23] Bratin Saha, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Mohan Rajagopalan, Jesse Fang, Peinan Zhang, Ronny Ronen, Avi Mendelson, "Programming Model for a Heterogeneoous x86 Platform", Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, 2009, pp.431-440.

[24] Nathan Clark, Amir Hormati, and Scott Mahlke, "VEAL: Virtualized Execution Accelerator for Loops", 35th International Symposium on Computer Architecture (ISCA), 2008, pp.389-400.

[25] Muthu Manikandan Baskaran, J. Ramanujan and P. Sadayappan, "Automatic C-to-CUDA Code Generation for Affine Programs", 19th International Conference on Compiler Construction, 2010, pp.244-263.

[26] Hyunchul Park, Yongjun Park, and Scott Mahlke, "Polymorphic Pipeline Array: A Flexible Multicore Accelerator with Virtuzlied Execution for Mobile Multimedia Applications", Procedings 42nd International Symposium on Micro architecture, 2009, pp.370-380.

[27] Yi Yang, Ping Xiang, "A GPGPU Compiler for Memory Optimization an Parallelism Management", Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, 2010, pp.86-97.

[28] Yindong Yang, Haibing Guan, Erzhou Zhu, Hongbo Yang, Bo Liu, "CrossBit: A Multi-Sources and Multi-Targets DBT", The First International Conference on Cloud Computing, GRIDs, and Virtualization, 2010, pp.41-47.

[29] Guoxing Dong, Kai Chen, Erzhou Zhu, Yichao Zhang, Zhengwei Qi and Haibing Guan, "A Translation Framework for Virtual Execution Environment on CPU/GPU Architecture", the Third International Symposium on Parallel Architectures, Algorithms and Programming, 2010, pp.130-137.

Erzhou Zhu is currently a Ph.D student at Shanghai Jiao Tong University, China. He received the M.S. degree and B.S. degree in Computer Science and Technology in Anhui University, Anhui, China, in 2004 and 2008 respectively. His research interests include virtual machine, binary translation and computer architecture.

Haibing Guan received his Ph.D. degree in computer science from the TongJi University (China), in 1999. He is currently a professor with the Faculty of Computer Science, Shanghai Jiao Tong University (Shanghai, China). His current research interests include, but are not limited to, computer architecture, compiling, virtualization and hardware/software co-design.

Guoxing Dong is currently a Master Degree Candidate student at Shanghai Jiao Tong University, China. He received the B.S. degree at Shanghai University in 2008, China. His main research interests are in binary translation and CPU-GPU Co-Processing.

Yindong Yang is currently a Ph.D student at Shanghai Jiao Tong University, China. He received the M.S. degree at School of Computer, Electronics and Information from Guangxi University in 2007, China. In 2004 he received his BS degree at School of Information and Technology from Jiangnan University, China. His main research interests are in virtual machines, computer architecture and compiling.

Hongbo Yang is currently a Ph.D. student at Shanghai Jiao Tong University, China. He received the M.S. degree in 1995 and received his B.S. degree in 1998 at Institute of Airforce Meteorologyity, China. His main research interests are in virtual machines, computer architecture and compiling.