Characterization and Transformation of Unstructured Control Flow in GPU Applications

Haicheng Wu, Gregory Diamos, Si Li, and Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA, USA
{hwu36, gregory.diamos, sli, sudha}@gatech.edu

ABSTRACT

Hardware and compiler techniques for mapping data-parallel programs with divergent control flow to SIMD architectures have recently enabled the emergence of new GPGPU programming models such as CUDA and OpenCL. Although this technology is widely used, commodity GPUs use different schemes to implement it, and the performance limitations of these different schemes under real workloads are not well understood.

This study identifies important classes of program control flows, and characterize their presence in real workloads. It is shown that most existing techniques handle structured control flow efficiently, some are incapable of executing unstructured control flow directly, and none handles unstructured control flow efficiently. A suggestion to reduce the impact of this problem is provided.

An unstructured-to-structured control flow transformation is implemented and its performance impact on a large class of GPU applications is assessed. The results quantify the importance of improving support for programs with unstructured control flow on GPUs. The transformation can also be used as a JIT compiler pass to execute programs with unstructured control flow on the GPU devices that do not support unstructured control flow. This is an important capability for execution portability of applications using GPU accelerators.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Control structures*; D.3.4 [**Programming Languages**]: Processors—*Run-time environments*

General Terms

Experimentation, Performance

Keywords

GPU, Unstructured Control Flow, Branch Divergence

1. INTRODUCTION

The transition to many core computing has been accompanied by the emergence of heterogeneous architectures driven in large part by the major improvements in joules/operation and further influenced by the evolution to throughput-oriented computing. This has coincided with the growth of data parallel computation that has become a pervasive and powerful model of computation whose importance has been amplified by the rate at which raw data are being generated today in all sectors of the economy and growing in the foreseeable future. The emergence of low cost programmable GPU computing substrates from NVIDIA, Intel, and AMD have made data parallel architectures commodity from embedded systems through large scale clusters such as the Tsubame [18] and Keeneland systems [**?**] hosting thousands of NVIDIA Fermi chips. Major research foci now include the development of programming models, algorithms, applications, performance analysis tools, productivity tools, and system software stacks.

Emerging data-parallel languages that implement *single instruction stream multiple thread* (SIMT) models [23] such as CUDA and OpenCL retain many of the control flow abstractions found in modern high level languages and simplify the task of programming these architectures. However, when the SIMT threads do not follow the same control path, performance suffers through poor hardware utilization and dynamic code expansion. This problem of *branch divergence* is critical to high performance and has attracted hardware and software support. The impact of branch divergence can be quite different depending upon whether the program's control flow is structured (control blocks have single entry and single exit such as if-then-else) or unstructured (control blocks have multiple entries or exits such as those using *goto* statements). In fact some GPUs will only support (and hence their compilers will only generate) structured control flow. Therefore it becomes important to understand the impact of unstructured control flow in GPU applications and performance effects of mitigating the negative impact. This understanding is critical to developing new techniques to execute program with unstructured control flows more efficiently in GPUs, which may lead to the support of some advanced features in GPGPU programming such as try/catch that were once inefficient on SIMD processors. Moreover, some highly unstructured applications such as ray tracing can also gain benefits from these kinds of techniques when running on GPUs.

A second reason for understanding the impact of unstructured control flow is the use of dynamic translation in supporting multiple GPU back-ends [7]. Having an on-die accelerator (such as GEN6 in Intel's Sandy Bridge) and an external high performance accelerator (such as AMD or NVIDIA GPU) will be a common configuration. The ability to execute a GPU kernel on either target to maximize performance is a desirable (and feasible) capability. Transformations between unstructured and structured control flow implementations are necessary when one of the GPUs does not natively support unstructured control flow, e.g., AMD Radeon [8]. Such transformations are necessary for true execution portability via dynamic translation.

In this paper we seek to analyze the occurrence and impact of unstructured control flow in GPU kernels. This paper makes the following contributions:

- Assesses the occurrence of unstructured control flow in several GPU benchmark suites.

CACHES'11 June 4th, 2011, Tucson, Arizona.

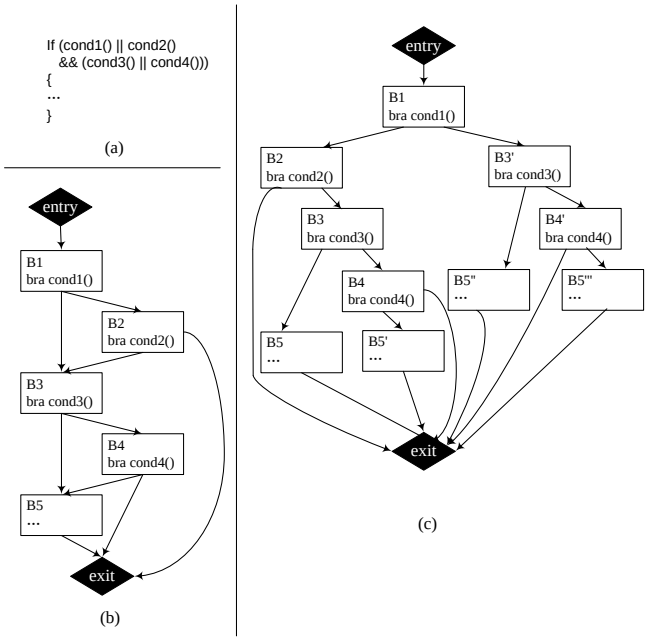

Figure 1: Example showing a compound condition that creates unstructured control flow: (a) code segment, (b) CFG having unstructured control flow, and (c) CFG after the Forward Copy transformation

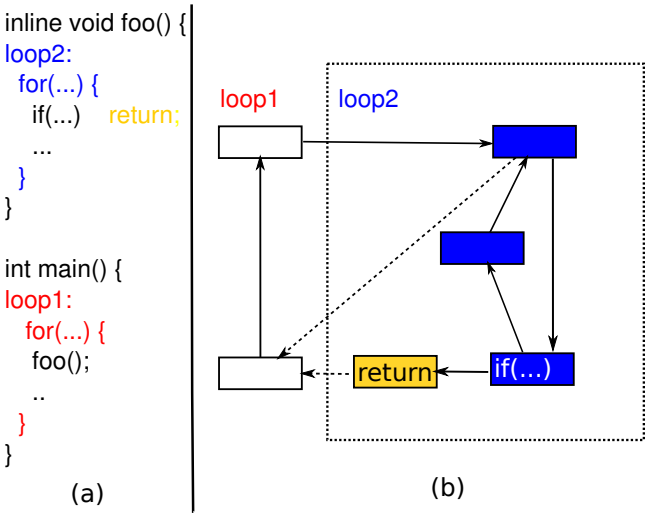

Figure 2: Example showing function inlining that creates unstructured control flow: (a) code segment and (b) CFG having unstructured control flow

- Establishes that unstructured control flow necessarily causes dynamic and static code expansion for state of the art hardware and compiler schemes. It shows that this code expansion can degrade performance in cases that do occur in real applications.
- Implements an compiler intermediate representation (IR) level transformation which can transform unstructured control flow to a structured control flow implementation. This transformation is useful for researching the performance of arbitrary control flows on GPUs, and also important for execution portability via dynamic translation.

The rest of the paper is organized as follows: Section 2 introduces unstructured control flow and its specific manifestations in GPU codes. Section 3 describes transformations for converting unstructured control flow to structured control flow. The experimental evaluation section, Section 4, assesses the impact of the transformations on several benchmark suites. The paper concludes with some general observations and directions for future work.

2. GPU CONTROL FLOW SUPPORT

Compilers can translate high level imperative languages such as C/C++ or Java into an intermediate representation (IR) that resembles a low level instruction set. Typical examples of IR are LLVM [16] , PTX [20] (CUDA GPU), or AMD IL [1] (AMD GPU). In the IR level, the Control Flow Graph (CFG) represents the execution path of the program. Every node of the graph is a group of sequentially executed instructions, and the edges are the jumps which are usually caused by conditional/unconditional branches.

Zhang and Hollander [25] classify control flow patterns into two categories, structured and unstructured. In general, commonly used control flow patterns (if-then-else, for-loop, do-while-loop, etc.) are **structured**. These patterns correspond to **hammock** graphs in the CFG which are defined as subgraphs having a single entry node and a single exit node [10]. On the contrary, **unstructured** control flow may have multiple entries or exits. Based on the classification, Zhang et al. introduced a generic approach to transform graphs with unstructured control flow to graphs possessing structured control flow. This transformation will be explained in section 3. The remainder of this section introduces the common sources of unstructured control flows and how this is supported in SIMT architectures.

2.1 Sources of Unstructured Control Flow

One of the most common sources is the *goto* statement used in C/C++ which allows control flow to jump to arbitrary nodes in the CFG. Similarly, longjumps and exceptions are two other sources of unstructured control flow.

However, even if the programming language forbids the use of *goto* statements (such as OpenCL), the compiler may also produce unstructured control flow in the IR level due to unintended side-effects of the language semantics. For example, in the code segment of Figure 1(a), the compiler does not need to evaluate all four conditions (which is known as a short-circuit optimization) and the CFG of the generated IR looks like Figure 1(b). This CFG has unstructured control flow because subgraph {B1, B2} and {B3, B4} both have two exits.

Moreover, CFG optimizations performed by compilers would also cause unstructured control flow [6]. Considering Figure 2(a), if function *foo()* is inlined into the *main()* function, the early return statement in loop2 would create the second exit from the loop, which is shown in Figure 2(b).

Since the above sources are very common in modern programming languages, normal programs usually have both the structured and unstructured control flows. If the system cannot execute unstructured parts efficiently, it would hurt the overall performance.

2.2 Impact of Branch Divergence in Modern GPUs

Modern programmable GPUs implement massively data parallel execution models. In this paper we analyze GPU kernels from CUDA applications compiled to NVIDIA's parallel thread execution (PTX) virtual instruction set architecture. PTX defines an execution model where an entire application is composed of a series of multi-threaded kernels. Kernels are composed of parallel work-units called *Cooperative Thread Arrays* (CTAs), each of which can be executed in any order subject to an implicit barrier between kernel launches. Threads within a CTA are grouped together into log-

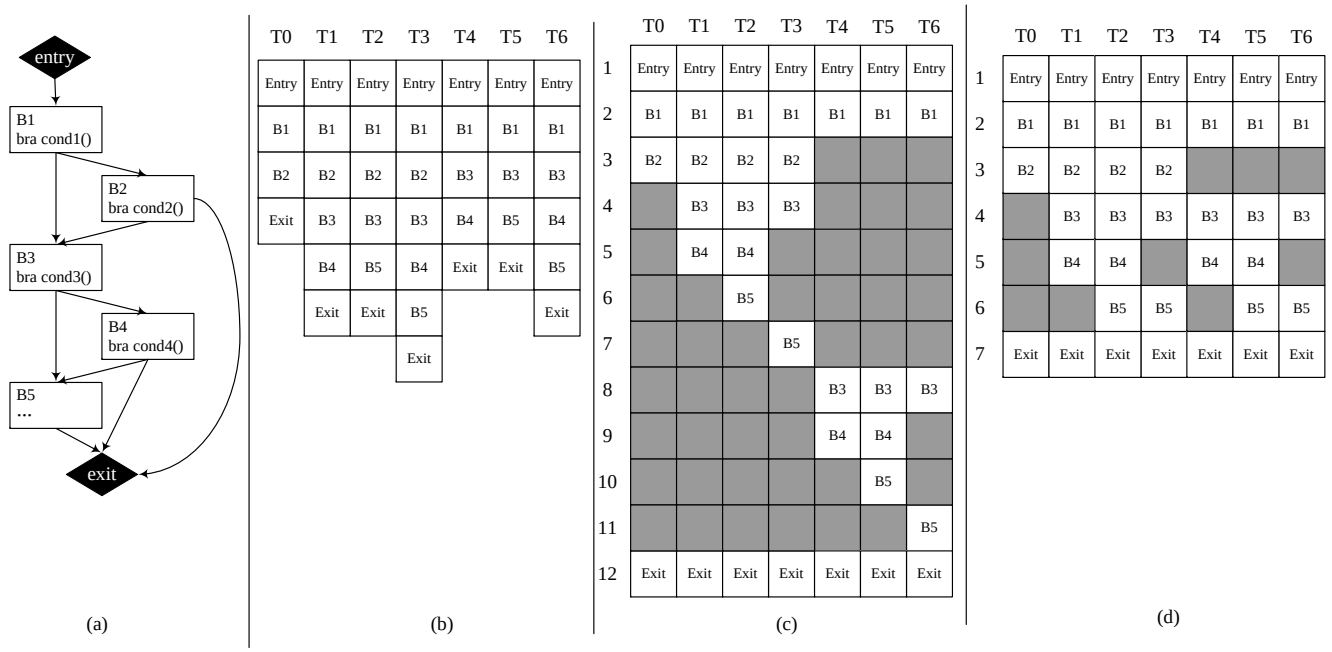

Figure 3: Example of mapping unstructured control flow into a SIMD unit: (a) unstructured CFG, (b) execution path, (c) re-converge at immediate post-dominator, and (d) re-converge at the earliest point

ical units known as *warps* that are mapped to *single instruction stream multiple data* (SIMD) units using a combination of hardware support for predication, a thread context stack, and compiler support for identifying re-converge points at control-independent code.

Since threads within the same *warp* have to execute the same instructions, branch control flow can potentially cause inefficiencies if the branch condition is not evaluated identically across all threads in a *warp*. In this case, some threads may take fall-through edge and the others may jump to the branch target, which is referred to as branch divergence. This can be handled by a process of serially enabling/disabling threads corresponding to the then/else branch. This effectively splits the *warp* into smaller subsets of threads which may then re-converge later in the execution.

The implementation details of re-convergence differ between GPUs. In AMD GPUs, the IR language (AMD IL) uses explicit instructions such as IF, ELSE, ENDIF, LOOP, ENDLOOP, etc., which means it only supports limited structured control flows [2]. The mapping of these control flows to the hardware is simple and fixed. It executes all the possible paths of the program (*then* part and *else* part for IF instructions) in a lock-step way, and threads re-converge at the END instructions such as ENDIF or ENDLOOP. If the compound condition code in Figure 1(a) is compiled for AMD GPUs, it has to generate CFG like Figure 1(c) which uses nested if-then-else to form a structured control flow. The Intel GEN5 graphics processors work in a similar manner [14].

However, mapping parallel programs with arbitrary control flows onto SIMD units is a difficult problem, because there is generally no guarantee that different parallel threads will ever be executing the same instructions. Thus, the re-convergence point may impact the overall performance. This will be discussed in the following subsection.

2.3 Unstructured Control Flow on GPUs

Although supporting structured control flow is sufficient for many graphics shading languages such as Microsoft DirectX and Khronos OpenGL, the migration to general purpose models such as OpenCL and CUDA that derive from C makes it advantageous to support unstructured control flow. Specifically, CUDA supports *goto* statements in the high level language. In addition its IR language, PTX, has many features in common with RISC ISAs, which support arbitrary branch instructions rather than explicit IF and LOOP instructions. Consequently as discussed in Section 2.1, compilation of CUDA programs can employ common CFG optimizations that are already widely used in other C/C++ program compilation frameworks and programmers do not need to worry about introducing unstructured control flow into programs that are not allowed on some GPU platforms.

The current state of the practice in determining re-convergence points for divergent SIMD threads is referred to as immediate post-dominator re-convergence [11] (the immediate post-dominator of a branch in a CFG, informally, is the node through which all paths from the branch pass through and which does not post-dominate any other post dominator). By using this method, the re-converge point is fixed for every divergent branch and can be calculated statically during compilation. For structured control flow, this method would re-converge at the end of loops or if-else-endif control blocks, which are as efficient as AMD GPUs. However, it may execute inefficiently for unstructured control flow. For example, in Figure 3, assume the *warp* size is 7 and these 7 threads take 7 different paths as shown in Figure 3(b), which is the worst case for this CFG. The immediate post-dominator of all branches is the exit node (see Figure 3(a)). Figure 3(c) shows how the SIMD unit executes these seven threads for re-converging at the immediate post-dominator. There are many empty slots in this figure and on average only 3.25 threads are enabled. It is also interesting to notice that the execution of CFG like Figure 1(c) is the same as Figure 3(c), which means AMD GPUs are also inefficient for this example.

Dynamic code expansion occurs when different paths originating from a divergent branch pass through common basic blocks before the re-convergence point. For example, in Figure 3(c), time slot 7 to 11 are running dynamically expanded code because B3, B4 and B5 have been already executed in time slot 4, 5 and 6.

The solution that reduces dynamic code expansion is to re-converge as early as possible. Figure 3(d) is an example where re-convergence happens much earlier than the immediate post-dominator. It saves execution time and has much better hardware resource occupancy. The inefficiency of re-converge at immediate post-dominator exac-

erbates the problem of branch divergence. To achieve performance improvements as shown in Figure 3(d), the compiler should be capable of identifying the potential early re-converge points and inserting necessary check instructions. It also needs the support from hardware to efficiently compare the program counter (PC) of each thread to check for re-convergence. To our best knowledge, there is still no technology that can achieve the efficiency shown in this example and thus there is still a lot of room for improvement in executing unstructured control flow in SIMD processors. If unstructured control flow can be handled more efficiently, some new language semantics, such as C++ try/catch style exceptions, can be added to the current programming model. Furthermore, if hardware can support unstructured control flow efficiently, compilers do not have to generate structured control flow like Figure 1(c). This can reduce the penalty caused by branch divergence.

2.4 Executing Arbitrary Control Flow on GPUs

Consequently, there are three ways to run programs with arbitrary control flows on different GPU platforms in an efficient (and hence portable) manner:

- The simplest method is to let compilers have the option to produce IR code only containing structured control flows. This IR code then can be compiled into different back-ends. This method may miss some optimization opportunities, but it is simplest to implement.
- Use a JIT compiler to dynamically transform the unstructured control flow to structured control flow online when necessary, i.e., the target GPU does not support unstructured control flow. The dynamic compilation may introduce some inevitable overhead.
- The most promising method is to develop a new technology (both compiler and hardware support) to replace current approaches to fully utilize the early re-convergence opportunity that is illustrated in Figure 3(d).

This paper presents a compiling technique to transform the unstructured control flow to structured control flow in Section 3.

3. CONTROL FLOW TRANSFORMATIONS

The principal result of Zhang's work is that the repeated application of three primary transformations can provably convert all possible unstructured programs into a structured format. However, their technique only applies to the programming language level instead of the IR level. To do the similar transformations in the IR level, some extra work is needed and the three original transformations have to be adapted.

The basic process has three steps:

1. Identify unstructured branches and some basic structured control flow patterns in the CFG, including if-then, if-then-else, self-loop, for-loop, and do-while loop,
2. Collapse the detected structured control flow pattern into a single node.
3. Use three sub transformations, which will be introduced here, to turn all detected unstructured control flow into structured control flow,

This process runs iteratively until there is only one node remaining.

The output of Step 1 is a control tree [19], which basically describes the components of all control flow patterns and their nested structures. Figure 4 shows a CFG and its control tree. It should be noticed that all unstructured control flows are detected in this step.

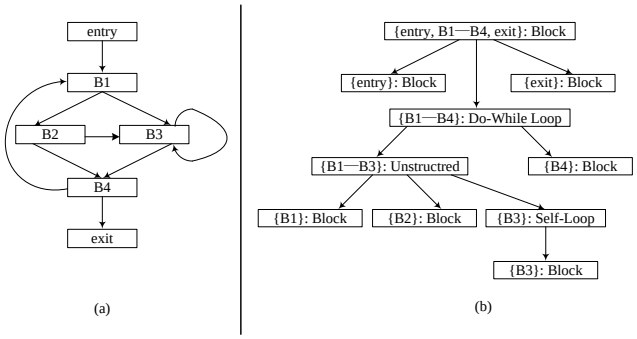

Figure 4: Example of (a) a simple CFG and (b) its Control Tree

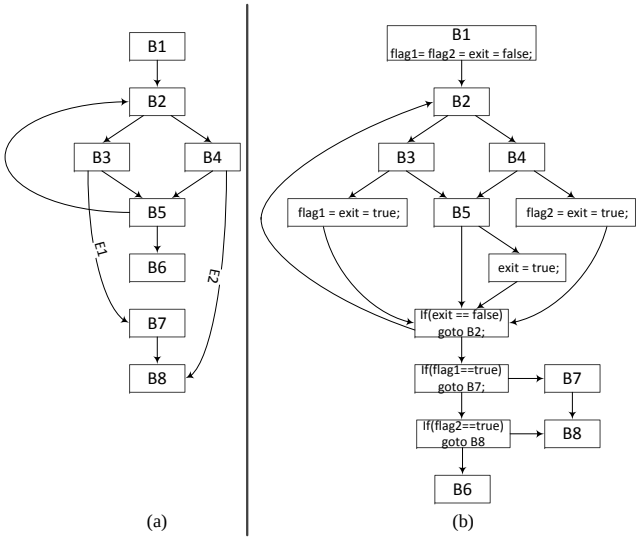

Figure 5: Example of a Cut transformation: (a) unstructured CFG (b) CFG after Cut transformation

Afterwards, the three transformations are performed repeatedly with the help of the control tree to transform the unstructured control flow. These transformations are conceptually and functionally equivalent to the ones used in Zhang's work [25] (the detailed algorithm and correctness proof can be found in their original work) and can be explained through the application of three primitive transformations.

- Cut: The Cut transformation moves the outgoing edge of a loop to the outside of the loop. For example, the loop in Figure 5(a) has two unstructured outgoing edges, E1 and E2. What cut transformations do is i) use three flags to label the location of the loop exits; ii) combine all exit edges to a single exit edge; iii) use three conditional checks to find the correct code to execute after the loop. It should be noted that after the transformation, the CFG in this example is still unstructured and needs other transformations to make it structured.
- Backward Copy: Backward Copy moves the incoming edges of a loop to the outside of the loop. For instance, Figure 6(a) has an unstructured incoming edge E1 into the loop. To transform it, the backward copy uses the loop peeling technique to unravel the first iteration of the loop and point all incoming edges to the peeled part. In this example, the CFG after the transformation is also unstructured. This transformation is rarely needed (see the experiment part in Section 4) because usually neither programers nor compilers would create loops with multiple entries.

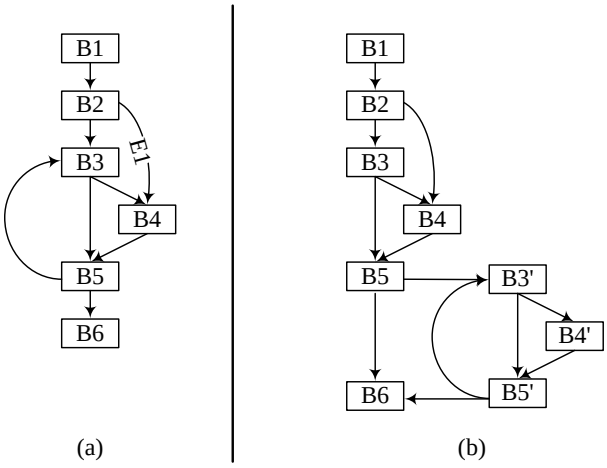

Figure 6: Example of a Backward Copy transformation

- Forward Copy: Forward Copy handles the unstructured control flow in the acyclic CFG. After Cut and Backward Copy transformations, there are no unstructured edges coming into or going out of loops. As a consequence, CFGs inside every loop can be handled individually and all structured loops can be collapsed into abstract single CFG nodes. Forward Copy eliminates all remaining unstructured branches by duplicating their target CFG nodes. For example, in Figure 1(b), B5 needs to be duplicated because edge B4->B5 is unstructured. If Forward Copy is performed multiple times, some subgraphs may be duplicated more than once and it may eventually lead to exponential code expansion. The final result shown in Figure 1(c) duplicates B5 three times and duplicates B4 and B3 once respectively. Actually, Figure 1(c) spans all possible paths between the entry node and the exit node.

  Comparing Figure 1(c) and Figure 3(c), it is interesting to see that the dynamically expanded code caused by re-convergence at the immediate post-dominator is exactly the same as the duplicated code in Forward Copy. This is not a coincidence. In fact Forward Copy can be used to measure the worst case of dynamic code expansion in immediate post-dominator reconvergence. The proof is not difficult, because Forward Copy and the worst case of re-converge at the immediate post-dominator are both spanning the CFG in a depth-first order. The detailed proof is omitted here due to the page limit.

All the above three transformations insert new instructions into the original program. The cut transformation, especially, has to use several new variables to store flag values, which introduces new register pressure. It needs to use more conditional branches as well when exiting the loop, which may cause more divergence in the GPU architecture (see Section 2.2).

This transformation is not only useful in characterizing unstructured control flows as discussed above. It can also be used in dynamic compilers which are used in heterogeneous systems, since the support for unstructured control flows in different GPU devices is different. This kind of transformation allows the program to run on several different back-ends without the interference of users, which is very useful for the large clusters comprised of different GPU backends.

4. EXPERIMENTAL EVALUATION

This section evaluates how often unstructured control flows are used in real GPU programs and how they may impact the performance over a large collection of CUDA benchmarks from CUDA SDK 3.2 [21], Parboil 2.0 [13], Rodinia 1.0 [5], Optix SDK 2.1 [22] and some third party applications. The CUDA SDK contains a large collection of simple GPU programs. Parboil benchmarks are compute intensive. Rodinia's collection is chosen to represent different types of GPU parallel programs, which are more complex than those in the CUDA SDK. Optix SDK includes several ray tracing applications. The three third party GPU applications used are renderer [24] (a 3D renderer), sphy-raena [3] (a SQL querry Engine), and mcrad [9] (a radiative transport equation solver).

Suite	Number of Benchmarks	Number of Transformed Benchmarks
CUDA SDK	56	4
Parboil	12	3
Rodinia	20	9
Optix	25	11

Table 1: Existence of unstructured control flows in different GPU benchmark suites

As for the compilation tools, NVCC 3.2 is used to compile CUDA programs to PTX code. Optix SDK benchmarks are running under Optix's own execution engine. A GPU compilation infrastructure, Ocelot 1.2.807 [7], is used for several other purposes: back-end code generation, PTX transformation, functional emulation, trace generation and performance analysis.

4.1 Static Characterization

The first set of experiments attempts to characterize the existence of certain types of control flow in existing CUDA workloads by using the unstructured to structured transformations introduced in Section 3. The transformation is implemented as a static optimization pass in Ocelot and it is applied to the PTX code of all benchmarks. The optimization can detect unstructured control flow and classify them by the type of transformations used(Cut, Backward Copy, or Forward Copy). The correctness of the transformation is verified by comparing the output results of the original program and the transformed program. Table 1 shows the number of applications having unstructured control flow in four examined GPU benchmark suites. Out of the 113 applications examined, 27 contain unstructured control flow, indicating that at the very least an unstructured to structured compiler transformation is required to support general CUDA applications on all GPUs. It is also the case that more complex applications are more likely to include unstructured control flow. Almost half of the applications in the Rodinia and Optix include unstructured control flow.

Table 2 shows the usage of different transformations. The first column is the benchmark name. The second column is the number of branch instructions every benchmark has. The third to the fifth columns show the number of times each transformation is used for every benchmark. The statistics show that Backward Copy appears to be nonexistent in current workloads which follows the common practice that programmers rarely write a loop with multiple entries. Cut transformations are necessary in programs that involve loops, but the shallow levels of nesting of GPU programs, especially those simple programs, makes this operation less common. Forward transformations are used most often. Further research shows that short-circuiting is the main trigger of these transformations. As explained in Section 2.2, short-circuiting does not run efficiently on the GPU platform.

The sixth and seventh column of Table 2 is the static code size of the benchmarks before and after the transformation. Static Code size is calculated by counting the PTX instructions of the benchmark. Usually the larger its code size is, the more complex control flow the program may have and the more transformations it needs.

Benchmark	Branch Instruction	Cut	Forward Copy	Backward Copy	old code size	new code size	Static Code Expansion (%)
CUDA SDK							
mergeSort	160	0	4	0	1914	1946	1.67
particles	32	0	1	0	772	790	2.33
Mandelbrot	340	6	6	0	3470	4072	17.35
eigenValues	431	0	2	0	4459	4519	1.35
PARBOIL							
bfs	65	1	0	0	684	689	0.73
mri-fhd	163	1	0	0	1979	1984	0.25
tpacf	37	0	1	0	476	499	4.83
RODINIA							
heartwall	144	0	2	0	1683	1701	1.07
hotspot	19	1	0	0	237	242	2.11
particlefilter_naive	29	3	5	0	155	203	30.97
particlfilter_float	132	2	4	0	1524	1566	2.76
mummergpu	92	2	26	0	1112	2117	90.38
srad_v1	34	0	1	0	572	595	4.02
Myocyte	4452	2	55	0	54993	62800	14.20
Cell	74	1	0	0	507	512	0.99
PathFinder	9	1	0	0	136	141	3.68
OPTIX							
glass	157	0	7	0	4385	4892	11.56
julia	1634	14	22	0	14097	18191	29.04
mcmc_sampler	101	0	3	0	4225	4702	11.29
whirligig	143	0	8	0	4533	5303	16.99
whitted	173	0	6	0	5389	5841	8.39
zoneplate	297	0	3	0	3397	3400	0.09
collision	101	0	4	0	2585	2595	0.39
progressivePhotonMap	127	0	4	0	3905	3960	1.41
path_trace	29	1	0	0	1870	1875	0.27
heightfield	46	1	0	0	1761	1771	0.57
swimmingShark	51	1	0	0	1990	2000	0.50
mcrad	415	11	10	0	4552	5238	15.07
sphyraena	1125	4	3	0	4393	4418	0.57
Renderer	7148	943	179	0	70176	111540	58.94

Table 2: Unstructured to structured transformation statistics

Benchmarks that have one Cut and zero Forward Copy, such as path_trace and heightfield, show that the number of instructions inserted by Cut is small. However, this is not the case for the Forward Copy. In the benchmark mummergpu, its code size was doubled by 26 Forward Copy transformations. The code size increment caused by Forward Copy depends on the size of the shared CFG nodes that need to be duplicated. Column eight is the relative static code expansion. Figure 7 shows the code expansion of those benchmarks using at least one Forward Copy with an average of 14.76%. For those benchmarks having a large number of transformations, such as mummergpu, julia, and Renderer, the static code expansion is significant.

Among all the benchmarks, Renderer has far more transformations than the rest. Other graphics benchmarks (particles, Mandlebrot, Optix SDK suite) also have more unstructured control flow on average. It is fair to say that graphics applications have great potential to improve performance if unstructured control flow could be handled more efficiently.

4.2 Dynamic Characterization

Since a better technology to run unstructured control flows is unknown, we cannot get an accurate performance impact of reconvergence at immediate post-dominator. However, we can use the functional emulator provided by Ocelot to count the number of

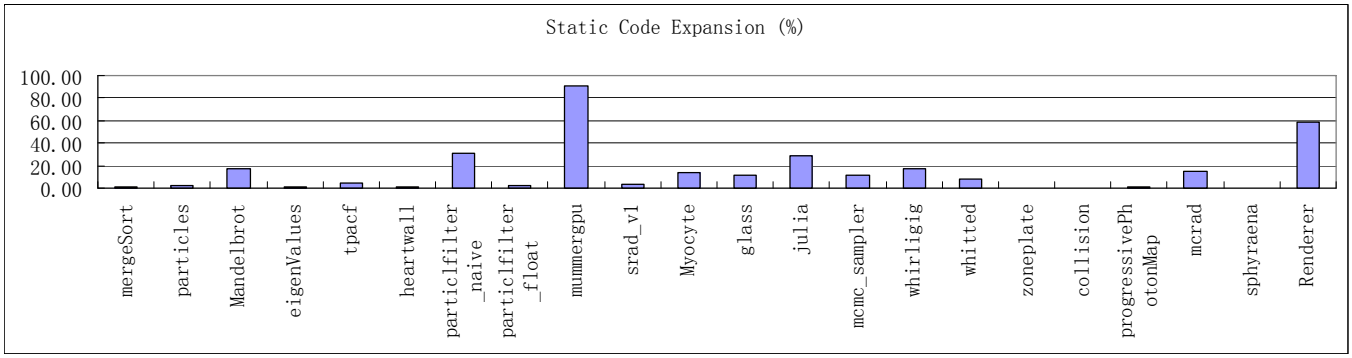

Figure 7: Static code expansion of benchmarks using Forward Copy

Benchmark	Dynamic Code Expansion (number of instructions)	Original Dynamic Instruction Count
mergeSort	0	192036155
particles	8	277126005
Mandelbrot	86690	40756133
eigenValues	7100	628718500
tpacf	2082509458	11724288389
heartwall	749028	121606107
mummergpu	11947451	53616778
Myocyte	205924	7893897
sphyraena	0	133386
Renderer	1153435	279729298

Table 3: Upper limit of dynamic code expansion

instructions executed due to the lack of earlier re-convergence such as instructions executed during time slots 4–11 in Figure 3(c).

In this experiment, we first determined which basic blocks (CFG nodes) form the unstructured control flow from our static transformation. These basic blocks might be dynamically expanded at runtime (like B3, B4, and B5 in Figure 3(a)). Then, we used the emulator to count the number of times these basic blocks will run in a divergent *warp*. For example, in Figure 3(c), B3 and B4 each runs twice in the divergent warp and B5 runs four times. Assuming each basic block has 1 instruction, we can say at most 8 instructions which are executed in time slots 4–11 may miss the early re-convergence (actual number of dynamic expanded instructions is 5, executed in time slots 7–11). Although this method overestimates the performance degradation, the overestimated part is limited to the initial execution of these instructions as they should not be counted. Take Figure 3(c) as an example, the overestimated part is time slots 4–6 during which B3, B4 and B5 are executed for the first time.

Table 3 shows the upper limit of dynamic code expansion for benchmarks using Forward Copy. Some benchmarks are not included due to the issues of the emulator. The results vary greatly. Some benchmarks, such as particles and mergeSort, have very low value because the unstructured part is executed infrequently or *warps* do not diverge when executing them. However, other benchmarks, such as mummergpu and tpacf, have a significant value meaning the application repeatedly executes these unstructured control flows. In this case, not having earlier re-convergence will impact the performance significantly. It is also interesting to notice that benchmark tpacf has low static code expansion but high dynamic code expansion, which means the unstructured part is executed very frequently.

4.3 Case Study

In this experiment, we modified the Ocelot emulator and rewrote the mummergpu benchmark to force it to re-converge as early as possible. The benchmark rewriting includes adjusting the PTX code layout, so re-convergence points appear at the earliest point like B3 appears after B1 and B2 in Figure 3(d). We measured the dynamic instruction count (instructions dynamically executed by all threads) of two mechanisms: re-converging at immediate post-dominator and re-converging at the earliest point. The result shows that re-converging at earliest place can reduce **14.2%** (from 53616778 to 46008916) dynamic instructions, which further demonstrates that it is a promising research area.

5. RELATED WORK

SIMD architectures have been designed with basic support for control flow since their large-scale deployment in the 1960s. Since that time, new designs have been incrementally augmented with compiler-assisted hardware support for non-nested structured control flow and eventually all hammock graphs [25]. These designs have culminated in support for all forms of unstructured control flow without any static code expansion.

ILLIAC IV [4], which is in general considered to be the first large-scale SIMD supercomputer, was designed around the concept of a control processor that maintained a series of predicate bits, one for each SIMD lane. Its instruction set could be used to set the predicate bits to implement simple common structured control flows such as if-then-else blocks and loops.

The primary limitation of a single predicate register is its inability to handle programs with nested control flow. In 1982 the CHAP [17] graphics processor introduced the concept of a stack of predicate registers to address this problem. CHAP includes explicit instructions for if, else, endif, do, while statements in the high level language. This is currently the most popular method of supporting control flow on SIMD processors and is also used by the AMD Evergreen and Intel GEN5 graphics processors.

To support unstructured control flow, a technique refer to as immediate post-dominator re-convergence is developed, which extends the concept of a predicate stack to support programs with arbitrary control flow. This is done by finding the immediate post-dominator for all potentially divergent branch and inserting an explicit re-converge instruction. During execution, predicate registers are pushed onto the stack on divergent branches and popped when re-converge points are hit. In order to resume execution after all paths have reached the post-dominator, the program counter of the warp executing the branch is adjusted to the instruction immediately after the re-converge point.

All of the previous techniques have been implemented in commercial SIMD processors. Dynamic warp formation is a technique originally described by Fung et al. [11] that increases the efficiency of immediate post-dominator re-convergence by migrating threads between warps if they are executing the same instruction. However, the power and hardware complexity to support detecting this and dynamically creating a new warp from a pool of threads at the same PC, which requires fully associative PC comparisons across

active warps every cycle and register file changes may outweigh the performance advantages. Like post-dominator re-convergence, this scheme supports all program control flow.

As to the area of GPU application characterization, Kerr et al. [15] and Goswami et al. [12] respectively characterized a large of of GPU benchmarks by using a wide range of metrics covering control flow, data flow, parallelism, and memory behaviors. Goswami also researched the similarities between different benchmarks. Their studies are valuable for future GPU compilation and microarchitecture design. Instead, our work focuses on the execution of unstructured control flow in GPUs and provides insight, characterizations, and suggestions for future GPU designs.

6. CONCLUSIONS

This work addresses the problem of running arbitary programs on any GPU device. The current state of practice is not satisfactory, since the support of unstructured control flow is very poor. Some GPU devices do not support unstructured control flow at all, while others do not support it efficiently because they will miss the earliest re-converge point. We propose an IR level control flow transformation that can turn an unstructured control flow into a structured one. This transformation is used to characterize the existence of unstructured control flow in a large number of benchmarks. The result verifies the importance of the problem. Further, the transformation is also usful in the dynamic compiler used in a heterogeneous system.

In the future, we will focus on automatically finding earliest reconvergence points in an unstructured control flow by using different compiler and hardware techniques to improve execution efficiency of arbitary programs on GPUs. At the same time, unstructured to structured transformation is also a useful technique. Its impact on the memory hierarchy deserves further research.

7. ACKNOWLEDGEMENTS

This research was supported by NSF under grants IIP-1032032, CCF-0905459, and OCI-0910735, by LogicBlox Corporation, and equipment grants from NVIDIA Corporation. We would also like to thank Andrew Kerr, Tri Pho and Naila Farooquie for helping us set up the experiments. Tips from the anonymous referees also greatly helped shape this paper.

8. REFERENCES

[1] AMD. *Compute Abstraction Layer (CAL) Technology: Intermediate Language (IL)*. AMD Corporation, 2.0 edition, 2009.

[2] AMD. *Evergreen Family Instruction Set Architecture Instructions and Microcode*, March 2010.

[3] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103. ACM, 2010.

[4] W. Bouknight, S. Denenberg, D. McIntyre, J. Randall, A. Sameh, and D. Slotnick. The Illiac IV system. *Proceedings of the IEEE*, 60(4):369 – 388, Apr. 1972.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.*, volume 9, pages 44–54, October 2009.

[6] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Technical report, 2001.

[7] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *Proceedings of PACT '10*, pages 353–364. ACM, 2010.

[8] R. Dominguez, D. Schaa, and D. Kaeli. Caracal: Dynamic translation of runtime environments for gpus. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pages 5–11. ACM, 2011.

[9] R. Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[10] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[11] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[12] N. Goswami, R. Shankar, M. Joshi, and T. Li. Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications. In *2010 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2010.

[13] Impact Research Group. Parboil benchmark suite, 2009. `http://impact.crhc.illinois.edu/parboil.php`.

[14] Intel. *Intel G35 Express Chipset Graphics Controller Programmers Reference Manual*, January 2009.

[15] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of ptx kernels. 2009.

[16] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[17] A. Levinthal and T. Porter. Chap - a SIMD graphics processor. *SIGGRAPH Comput. Graph.*, 18(3):77–82, 1984.

[18] S. Matsuoka. The road to TSUBAME and beyond. *High Performance Computing on Vector Systems 2007*, pages 265–267, 2008.

[19] S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[20] NVIDIA. *NVIDIA Compute PTX: Parallel Thread Execution*. NVIDIA Corporation, Santa Clara, CA, 2.1 edition, October 2009.

[21] NVIDIA, 2010. `http://developer.nvidia.com/cuda-toolkit-32-downloads`.

[22] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics*, 29:66:1–66:13, July 2010.

[23] S. Rixner, W. Dally, U. Kapasi, B. Khailany, A. López-Lagunas, P. Mattson, and J. Owens. A bandwidth-efficient architecture for media processing. In *Proc. of the 31st Annual International Symposium on Microarchitecture*, pages 3–13. IEEE Computer Society Press, 1998.

[24] T. Tsiodras. `http://users.softlab.ece.ntua.gr/~ttsiod/`.

[25] F. Zhang and E. H. D'Hollander. Using hammock graphs to structure programs. *IEEE Trans. Softw. Eng.*, pages 231–245, 2004.