Characterizing performance loss from mapping general purpose applications onto GPU architectures

Archit Gupta
Department of Electrical Engineering
and Computer Sciences
University of California, Berkeley
Email: architgupta@berkeley.edu

Sohum Datta
Department of Electrical Engineering
and Computer Sciences
University of California, Berkeley
Email: sohumdatta@berkeley.edu

***Abstract*—Graphics processing units today provide an enormous amount of computing power for data-parallel scientific applications. However, since these were originally designed to run graphics applications, their fundamental architectural features restrict us from using them for general purpose computing. In order to tap the potential of these units, several hardware/software schemes have been proposed in the past. In this paper, we examine the different strategies that are adopted to map a general purpose application onto a GPU. Moreover, we quantify the performance loss that arises from this mapping of general purpose applications onto a GPU in the context of branch divergence. We also try to look at qualitative differences in segements of these applications which correspond to the underlying algorithm and the adopted mapping respectively in the context of branch behavior.**

I. Introduction

Graphics processing units (GPUs) are increasingly being used for various compute-intensive tasks like modelling mechanical systems, matrix manipulations, image processing, neural networks etc. Owing to the availability of a large number of parallel resources, on certain data-parallel applications, they can achieve a performance which is orders of magnitude higher than a conventional CPU.

However, this gain in performance is one side of a trade off. Since GPUs were originally designed to perform graphics manipulations, which are inherently data parallel, the performance of similar applications on a GPU results in significantly better performance as compared to a CPU. The flip side of the trade off is the loss of generality. General purpose CPUs are equipped with accurate branch prediction, Out of Order (OoO) cores, multi-level caches etc. in order to extract the maximum possible amount of instruction level parallelism (ILP) and to mitigate the performance hit that might arise from complicated control flow or poor spatial locality of data in the programs. On a GPU operating on a similar power/energy budget, these resources are reallocated to provide several, but extremely simple (almost bare) pipelines. This is done, assuming that the programs that run on this machine will exhibit a staggering amount of data parallelism.

	Intel Xeon E5-1660	NVIDIA Quadro M4000
Max power	130 W	120 W
Total Cores	6	1664
Clock Frequency	3.3 GHz	773 MHz
Memory Bandwidth	51.2 GB/s	192 GB/s
Price	$1089	$931

TABLE I: Comparing the state-of-art CPU and GPU available with similar power budgets and price range

Table I shows us some of the architectural features of a CPU and a GPU having a very similar power budget and lying in the same price range. The availability of such a vast number of computing resources has pushed a lot of general purpose programs onto a GPU. However, since these general purpose programs are not perfectly aligned with the GPU's model of execution, they are not able to fully utilize the computational resources that are available on the GPU.

The focus of the research community in computer architecture has been directed towards making changes in the GPU architecture which let it acommodate a wider set of general purpose applications without affecting its internal structure significantly. In this paper, we discuss the ways in which a general purpose application is mapped onto a GPU. Moreover, this mapping leads to some overheads (in terms of dynamic instruction count, control hazards etc.) which degrade the overall performance of the GPU. We try to measure this performance loss that arises from mapping general purpose applications onto a GPU.

II. Motivation

In order to demonstrate the problems that arise in mapping a general-purpose application onto a GPU, let us consider an example problem. Figure 1 shows a sample program implementing the dot product of two vectors $\vec{a}$ and $\vec{b}$. The product of the individual components of the two vectors is a parallelizable segment in the program, whereas the reduction sum is inherently serial. Vector dot products are fairly common in a lot of scientific programs like solving linear systems of equations, clustering etc.

To map this problem onto a GPU (or any machine which is capable of extracting data level parallelism), we need to

```
float reduction_sum;
float vector_dot(float* a,
                 float* b, int N)
{
   float  dot = 0;
   for (unsigned i=0; i < N; i++)
   {
      dot = a[idx] * b[idx];
      reduction_sum += dot;
   }
}
```

Fig. 1: CPU code for the dot product of two vectors $\vec{a}$ and $\vec{b}$, which are stored in the float arrays **a** and **b** respectively

```
__device__ float reduction_sum;

__global__
void vector_dot(float* a, float* b,
                        float* c, int N)
{
   int idx = (blockIdx.x + blockDim.x)
                       + threadIdx.x;
   if (idx < N)
   {
      c[idx] = a[idx] * b[idx];
   }
   __syncthreads();
   if (idx == 0)
   {
      for (unsigned i=0; i < N; i++)
      {
         reduction_sum += c[idx];
      }
   }
}
```

Fig. 2: GPU kernel code for the dot product of two vectors $\vec{a}$ and $\vec{b}$, which are stored in float arrays **a** and **b** respectively. **c** is used for local computation and helps in parallelizing the computation

parallelize the parallel segment of the program and execute the serial part as is. GPUs today act as hardware accelerators and not standalone processors themselves. A CPU is responsible for setting up the data and instructions in the GPU memory and signalling the GPU to initialize computation. The serial segment can be executed in two ways 1) On the CPU - This involves a significant amount of overhead as some of the data has to be transferred between the CPU and the GPU (each of them has an independent memory subsystem). 2) On the GPU, as a part of the execution *kernel*.

A GPU *kernel* is a stream of instructions which is executed

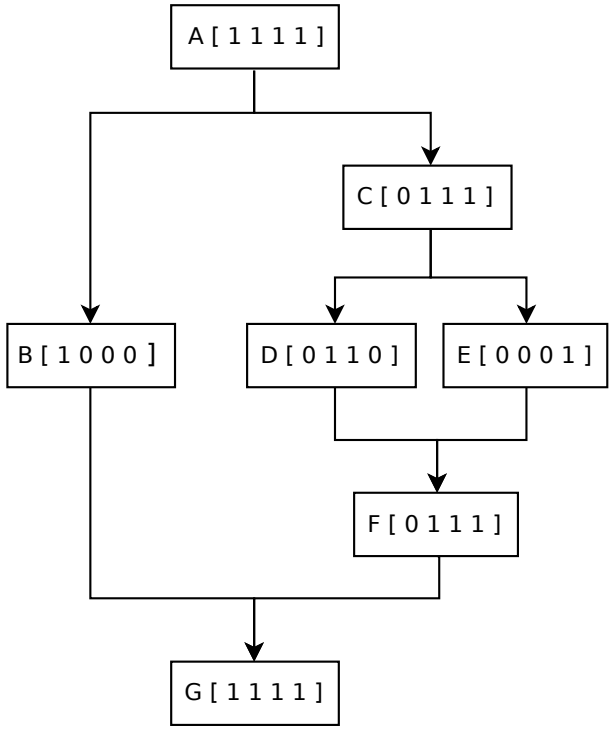


Fig. 3: A sample control flow graph to understand branch divergence. Each block in the graph depicts the name of a basic block followed by a 4 bit vector show the status of different threads while executing that basic block. The i^{th} bit represents the state of the i^{th} thread.

in one computational unit. Since the kernel is designed to run on all the compute units in parallel, running the serial segement inside the kernel involves disabling all-but-one compute units when the serial segment is being executed. Figure 2 shows an implementation of the same algorithm as Figure 1 (vector dot product) on a GPU. In this implementation, the parallel segment, which is taking the product of individual components of the two vectors is done in parallel by different threads (indexed by the variable 'idx'). After all the threads have completed this operation, the thread with index '0' computes the reduction sum. The reduction sum is calculated on the GPU to avoid the copy of the array **c** from the GPU to the CPU.

However, this operation causes the remaining compute units on the GPU to stall as they are still executing the '*NOT TAKEN*' part of the if statement. In this paper, we analyze the performance loss because of the presence of these inefficiencies in the *mapping of a general-purpose application onto a GPU.*

III. Overview of Graphic Processing Units

Branch Divergence

IV. Problem Description

Any implentation of an algorithm on CPU, GPU etc. requires a control flow graph and some branch statements. We

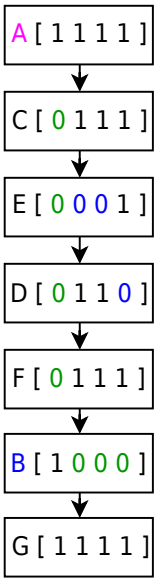


Fig. 4: The execution of the 4 threads as shown previously progresses as shown here on a GPU

classify each of these branches into two categories.

A. Intrinsic branches

An *intrinsic* branch refers to an algorithmic branch. These could be data-dependent decisions in a program, or branch instructions specified by the program's nature, like the branches resulting from the for loop in the GPU example presented in section II. The latter branch instructs the machine to iterate through all the entries in a vector in a sequence and compute a running sum.

B. Extrinsic branches

Some of the branches in a particular implementation of a program arise because of the limitations of the device. The branch arising from running the reduction sum on a single kernel in section II is one such branch. The limitation that forces us to use this branch is the fact that all the compute units in a GPU are bound to execute the same kernel. These branches have been referred to as *extrinsic* branches.

Our definitions imply that extrinsic and intrinsic branches are present in both GPU and CPU implementations of a program. The parallel segments of programs are generally implemented as loops in a CPU implementation, leading to branch instructions. These branches will also be called extrinsic as they are not dictated by the algorithm but are a result of the CPU's limitation of being inherently serial.

Estimating Performance impact

In section III, we saw that branch instructions lead to a unique problem called branch divergence on a GPU. Since we have categorized branches into two categories, we will calculate the performance impact of each of the two categories based on branch divergence. Continuing on our sample control flow graph in Figure 3, let's say that the branch at A is an EXTRINSIC branch and the branch at C is an INTRINSIC branch. Because of its lock-step execution, at each cycle, all the threads execute the same basic block. Figure 4 shows the sequence of basic blocks and the active threads in each execution block in the GPU.

We can think of each bit in the execution flow to be a computational slot. Slot (or thread) 1 is deactivated by the branch at A and it remains deactivated when the GPU is executing basic blocks C, D, E, and F. Also, the branch at A deactivates slot 2, 3 and 4 when the GPU is executing basic block B. Similarly, we can see that the branch at B deactivates slot 4 when D is being executed and slots 2 and 3 when E is being executed.

For the sake of simplicity, let us assume that each basic block has the same length, equal to 1 (it is trivial to accommodate different basic block sizes. We are making simplifying assumptions for the sake of illustration). Let us compute the fraction of total available slots that were:

1) Active (f_A)
2) Inactive due to an extrinsic branch (f_{E_i})
3) Inactive due to an intrinsic branch (f_{I_i})

It is easy to calculate these numbers for our system and assumptions. We have a total of 28 computational slots, of which, 7 remain inactive because of the extrinsic branch at A and 3 remain inactive because of the intrinsic branch at C. So, we can say that

$$f_A = \frac{18}{28} = 0.64$$
$$f_{E_i} = \frac{7}{28} = 0.25$$
$$f_{I_i} = \frac{3}{28} = 0.11$$

We can see that, in this case, 25% slots are wasted because of the presence of an extrisinc branch. In the upcoming sections, we will discuss the framework for applying this metric to a set of real word problems and present our findings.

V. Our Approach

A. Tagging CUDA Benchmarks

In order to measure our metrics, we use selected benchmarks from the RODINIA benchmark suite [1]. This comprises of a large of set of data parallel applications written for a variety of parallel platforms including openMPI, openCL and CUDA. We use the CUDA implementations and simulated these benchmarks on GPGPU-SIM [2] in order to be able to easily instrument some of the architectural parameters required for our measurements.

Framework: GPGPU-Sim runs in two different modes: simulating Parallel Thread eXecution (PTX) and PTXPlus. PTX is a pseudo-assembly language used by Nvidia's CUDA Programming environment. Instead of being run directly on the machine, the driver recompiles the PTX into the native SASS implementation before execution. PTXPlus is a pseudo-language specifically used to model the functioning of SASS ISA of the Nvidia GT200 series GPUs. PTXPlus extends the

regular PTX by augmenting it with addressing modes, new instructions and data-types. The extension is a one-to-one map of most of the instructions of the regular SASS. However, the extension is only useful for a very limited range GPUs from Nvidia.

We decided to use the PTX assembly code for our purposes:

- The PTX compiler is a standard pseudo-assembly code for all CUDA kernels. The machine readable SASS changes with the GPU family and may not be backward compatible. A PTX code, however, is embedded in the executable; a newer family of processors may do a Just-in-Time(JIT) compilation to produce a machine-compatible code.
- Most of the hardware-specific optimizations such as rearrangements and predication of instructions are done on the recompiled SASS code. Hence, the PTX can be thought of as a faithful reproduction of the high-level code in a lower-level assembly construct.
- The SASS translation of the High-level Control structures are very difficult to predict a priori. Since identifying branches at the High-Level control structures is the only way, it is very difficult to trace these constructs to the machine code.
- The GPGPU-Sim does not simulate the execution of the SASS code directly. Moreover, its PTXPlus infrastructure is only compatible for a limited range of Nvidia GPU families. The PTXPlus also known to ignore certain SASS instructions, such as those that modify the SIMT Stack.

The practical difficulties as well as the inefficiencies of simulating the SASS were the main reasons for adopting PTX. However, we are aware that PTX is not an accurate representation of the machine code. Its features are considerably high level and may be a serious problem in capturing low-level branch behaviour: it uses an arbitrarily large logical register set, and many individual instructions may be compounded into a single PTX operation. Hence, it is reasonable to expect significant difference in the behaviour of branches in a machine code.

Assumptions about the compiler: The procedure to tag the branches according to their nature in the program involves forming a trace from High-Level control structures to branch instructions. This conversion generally occurs through several passes of optimizations by a compiler and finally the linker. Hence, the foundation of our framework was to pass identification marks that are preserved across the compilation process.

We have used assembly-language labels as identification tokens. There are two kinds of labels for the two kinds of branches, and a branch inherits the nature of the label that immediately precedes it in program order. Once a label is paired to a branch, it is invalidated and is never used to identify another branch. We have modified the simulator so that a branch will not be accepted if it is not preceded my a valid label. In order to enforce the intended mapping of branches, we have developed the following assumptions to guarantee mapping correctness:

```
__asm__("EXTRN:");
// to exit loop if <init> fails
// <condition> initially
for (<init>; <condition>; <update>)
{
    ....
    <iteration blocks>
    ....
    __asm__("EXTRN:");
    // succedded by <update> and
    // the <condition>-test-branch.
}
```

Fig. 5: The template for a For-loop in C, a widely used HLC structure

- The PTX code generated from the High-level Control structures (henceforth, HLC structures) preserve the order of the operations that appear in the distinct blocks within these HLC structures. As a corollary, any assembly-language label will also occur in the same order among its neighbouring instructions within a structure. During the extensive trials of this assumption, we discovered that the PTX code was not always a plain representation as we had expected. Several optimizations were done which were difficult to predict. Loop unrolling was the especially unpredictable as the error in the map depends on the extent of unrolling which is highly context-specific. We turned-off all compiler optimizations, and didn't observe any violations to our assumptions.
- Any HLC structure has a definite character of being extrinsic or intrinsic. The branches that they produce inherit their nature. In some cases, we may not be able to clearly define the label of a high-level structure but can do so for each of its constituent branches. Since any HLC structure can be expressed as a control-flow graph or a decision tree with nodes representing binary decisions, it can always be expressed as a sequence of intrinsic or extrinsic branches by the nature of its nodes.
- A given HLC structure has a definite pattern in which its constituent branch appears. Such a pattern will certainly be present in the assembly of the HLC structure, and will be a subset of any other branches that may be produced by the HLC structure. For instance, the two branches for the loop that appear immediately before and at the very end of the iteration body in 5 are compulsory. There may be additional branches too (such as, if the ¡*condition*¿ contains boolean expressions compounded by logical operations). This is because any HLC structure has a certain minimum representation as a decision tree, and its node there form the compulsory branches. A substitution of its nodes by any decision tree for a boolean expression does not remove these branches.
- Assuming that we have correctly identified the compul-

sory patterns of all major HLC structures, if the number of static branches equal the number of branch labels, then our mapping is correct. This assertion can be easily proved: our branch labels create a ordered partition of our program. If we have correctly identified compulsory patterns of all HLC structures in the code, then each partition will have at least one branch and the first such branch in the program order will be preceded by a correctly-matched branch label. Therefore, as the premise says that labels equal branches, there is exactly one branch (correctly matched) in each partition. Hence, the mapping is correct.

B. Modifications to GPGPU-SIM

Branch target buffers: In order to accommodate some of the metrics that we wish to measure, we had to make a few changes to the existing GPGPU-SIM architecture. Besides measuring the performance impact of branch instructions (as mentioned in section IV), we are also interested in knowing if extrinsic and intrinsic branches have characteristically different behavior. In order to measure this, we implemented branch target buffers (BTB) in GPGPU-SIM. Each shader core has its own BTB. Any branch instruction that is executed at least once reserves an entry in its shader's BTB. Since the branch instructions (static count) are few in number and we are doing a software simulation, it is fairly easy to maintain a BTB which can grow dynamically in size (unlike a practical hardware implementation, which would have a fixed size). Figure **??** shows the BTB resulting from the execution of the "backprop" algorithm from RODINIA benchmark suite. The different columns of the BTB are described below

- PC – Program counter for the branch instruction
- TYPE – This field can take two values. 'ext', implying that the branch is an extrinsic branch and 'int', implying that the branch is an intrinsic branch
- TARGET – The program counter to which the branch instruction jumps
- INSTANCES – Every thread that was active when the branch instruction was encountered by the warp add 1 to the instances field
- TAKEN – Of the threads that were active when the branch instruction was encountered, 'TAKEN' represents the fraction of threads that took the branch
- OCCUPANCY – Occupancy gives the average number of threads that were active immediately after the branch instruction was executed.[1]
- DYN COUNT – It represents the total number of times some warp executed this particular branch instruction

Thread status tables: Thread status tables are used to evaluate the performance metric described in section IV. For every warp instruction being currently executed, thread status tables keep track of status of each thread in the warp. A warp can be marked as active if it is active in the current cycle. Otherwise, it is marked "INACTIVE EXTRINSIC" or "INACTIVE INTRINSIC", depending on the last branch instruction that deactivated the thread. Every time a warp diverges, the current status of all the threads is stored in a stack (similar to the SIMT stack used for PDOM convergence in [3]) and the new status is updated depending on the nature of the current branch. This can be best illustrated by an example. Let us consider the control flow graph shown in Figure 3. We will discuss the update of thread status tables assuming that the branch at A (say an extrinsic branch) has been executed. Moreover, C is an intrinsic branch. The execution proceeds as shown in Figure 4.

PC	Reconvergence PC	Thread status
C	G	E A A A
B	G	A E E E

TABLE II: Thread status tables implementing a stack structure to keep track of thread status values as threads diverge and converge subsequently. The stack grows downwards (B represents the current top of stack and will be executed next)

The thread status represents the status of threads when the respective basic block will be executed (A – active, E – deactivated by an extrinsic branch, I – deactivated by an intrinsic branch). Basic block B will executed first. Divergence at C leads to new entries being pushed into the stack, which gets updated to look like:

PC	Reconvergence PC	Thread status
B	G	A E E E
D	F	E A A I
E	F	E I I A

TABLE III: Thread status tables describing the threads immediately after branch at C is executed.

One can readily see that the thread status table maintains the status of threads in a warp across the various divergence/convergence events. Every instruction reads it thread status from the stack when it has been executed and updates the counts of respective status values. These values are then aggregated over all warps and reported to measure the performance impact

VI. Results

A. Static/Dynamic branch counts

Figure 7 shows the distribution of intrinsic and extrinsic branch instructions in terms of their static and dynamic counts. Static counts are measured by analyzing the compiled PTX code, whereas, the dynamic count is obtained from the branch target buffer shown in Figure 6.

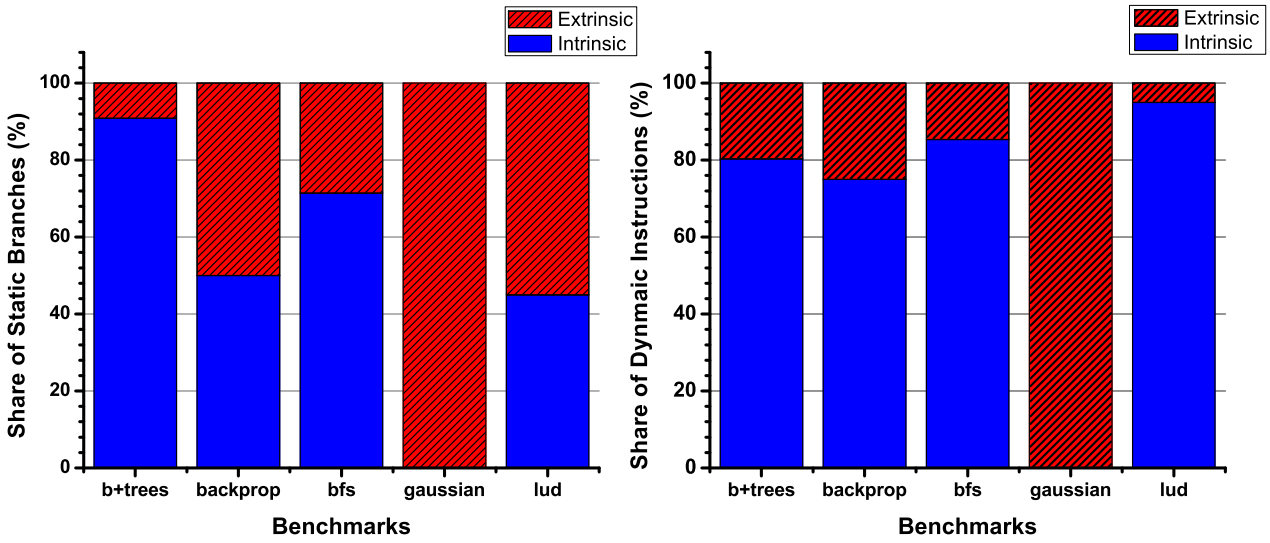
B. Performance impact

As described in Sections IV and V, for each PTX instruction, we measure the total number of slots that were 1. Active, 2. Inactive due to an intrinsic branch and 3. Inactive due to an

[1]The initial idea was to characterize the branch instructions based on this value as well. However, we have left that as a future prospect due to the lack of data

PC	TYPE	TARGET	INSTANCES	TAKEN	OCCUPANCY	DYN COUNT
b0	ext	e8	65536	0.9375	0.937500	2048
198	int	278	65536	0.0000	0.000000	2048
1e8	int	250	262144	0.7656	0.765625	8192
270	int	1b8	262144	0.7500	0.750000	8192
290	ext	2a0	65536	0.0625	0.062500	2048
508	ext	518	65536	0.0002	0.000244	2048

Fig. 6: Some of the fields in the branch target buffer implemented in GPGPU-SIM



(a) static count for branch instructions

(b) dynamic count for branch instructions

Fig. 7: Distribution of the number of branches encountered in different benchmarks

extrinsic branch. Finally, for any benchmark, we evaluate the aggregate of these metrics over the entire period of time when the benchmark was running. In Figure 8, we report the fraction of total available slots that were wasted due to intrinsic and extrinsic branches respectively.

C. Branch characteristics

In Figure 9, we report the weighted average of taken fractions (normalized by the instances for each branch, as reported in Figure 6) for each class of branches.

VII. Discussion

Static versus Dynamic Branches: The figures 7a and 7b demonstrate that the benchmarks we have tested tend to have more share of intrinsic branches in their Dynamic branch counts than their Static counterpart. The share of intrinsic branches rises the most for *lud*: from less than 50% to above 95%. *lud*'s LU Decomposition algorithm has a loop which executes all the intrinsic branches in successive iteration. Most of the extrinsic branches only execute once and are involved in initializations or intermediate data movement and setup. Hence, we see a dramatic improvement in the share of intrinsic branches for *lud*. *backprop* and *bfs* also have a similar behaviour, perhaps largely due to their iterative nature even in a data-parallel GPU code. *b+trees*, however, sees a reduction of share of intrinsic branches. This is perhaps due to the repeated data-bound checks that it must do while pointer-chasing down the tree. Finally, *gaussian* has no intrinsic branches, a feature unique to it as most branches in its code are data-bound checks while performing scalar product. With the exception of *gaussian* (The Gaussian Elimination algorithm), it seems that a task which is iterative and has loops in its main Kernel code may have a higher share of intinsic branches to its dynamic branch count. This may suggest that most intrinsic branches are in the critical, repetitive body of the code which is a reasonable expectation.

Share of Computational Slots Inactive: The results for this metric 8 does not demonstrate a clear distinction between the static and dynamic branches. Although there is insufficient

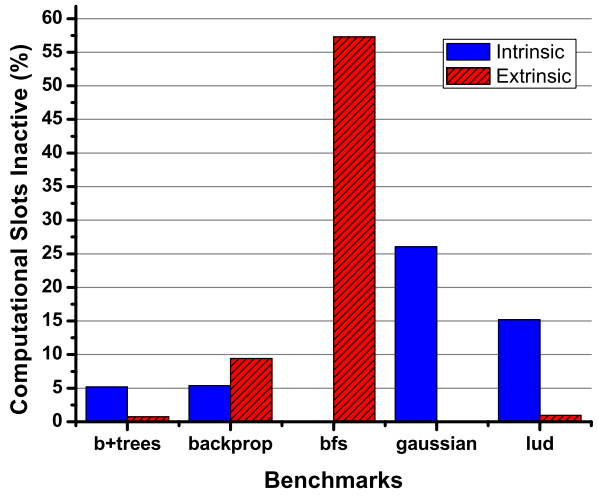


Fig. 8: Percentage of computational slots wasted by the two categories of branch instructions

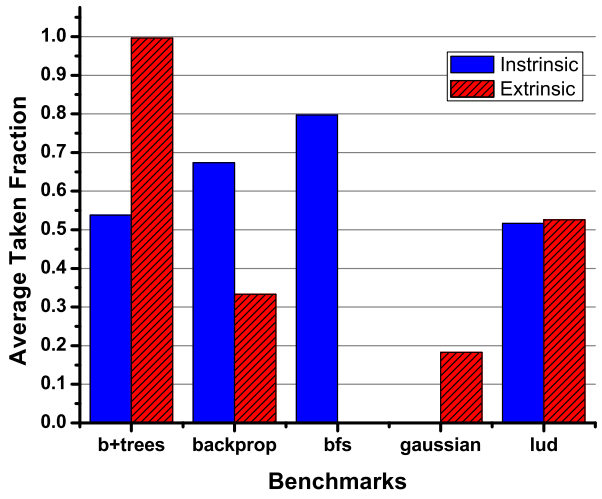


Fig. 9: Characterizing extrinsic and intrinsic branches based on the fraction of times they are taken when encountered

data to deny existence of any distinct behaviour between the two types, it seems almost certain that extrinsic branches seem to have a significant share in the Inactive slots. For instance, three of the five tested (*gaussian*, *lud* and *backprop*) have a larger share of inactivity due to their extrinsic branches. It is especially perplexing for *lud*, where although most dynamic branches are overwhelmingly intrinsic (7b) they contribute to little or nothing to Inactivity. Moreover, *bfs* wastes more than half of its slots available on inactive intrinsic branches. In any case, we think the data here is too small to discern a general trend, but maybe sufficient to proceed with further investigation on the share of inactivity due to extrinsic branches.

Taken Fraction: The results of taken fraction (Figure 9) are also in-conclusive. This is the metric which clearly demonstrates that more data will be needed, as no distinct extrinsic branch behaviour is visible as was expected. Although *b+trees* have always taken extrinsic branches, and *bfs* has none; *backprop*, *lud* and *gaussian* have extrinsic taken fractions quite close to 0.5. Hence, much more work can be done in this direction in the future.

VIII. Conclusion

Extrinsic branches seem to cause significant Inactivity. The results from Section VI , though not exhaustive, suggest that extrinsic branches may cause large inactivity in some benchmarks. However, it isn't clear that the inactivity is a result of the nature of such branches, or rather of inefficient programming practises.

While learning the algorithm in order to understand and label the HLC structures, we often came across extra branches in critical sections of the code (for example, in a very important function, in the main iterative loop, or the core of the kernel algorithm) present just to support an extra queue or array that made the program easier to write. We believe such unnecessary branches hurt performance significantly, and can be easily removed (albeit making the code less readable and maintainable).

Hence, a natural extension of this project could be to grade extrinsic branches by the number of times they are executed and their behaviour. These extrinsic branches may be further classified into those present in critical sections (often executing) and those used solely for setup or initialization. With such classification we may be able to focus on the most offending types of such branches and explore the different solutions , both architectural and software, that could mitigate them.

Another class of extrinsic branches that we encountered were those that are employed to run a large serial portion of a task between two heavily-parallel portion on the Kernel. Of the few cases that we saw, most of them were present to make the entire task run on the GPU. However, the serial portion often had no relation with the rest of the task, and could have exploited the ILP on a CPU much better than on a GPU. These branches, extrinsic for a very different reason, cannot be as clearly differentiated. Because a relatively small serial code is faster as a part of the Kernel code: the time lost in memory transfer to or from the host (CPU) and the device (GPU) to start executing the next Kernel, may negate any benefit of running the serial part on the CPU. Hence, we believe more exploration could be done to understand the meaning of *extrinsic* and *intrinsic* in the huge variety of general-purpose GPU code that we have today.

Acknowledgment

The authors would like to thank...

REFERENCES

[1] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, IEEE, 2009.

[2] A. Bakhoda *et al.*, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 163–174, IEEE, 2009.

[3] W. W. Fung *et al.*, "Dynamic warp formation and scheduling for efficient gpu control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 407–420, IEEE Computer Society, 2007.