Branch Divergence Analysis

CS380C Compilers – Final Project

Submitted to:

Professor Calvin Lin
Department of Computer Science
The University of Texas at Austin

TABLE OF CONTENTS

1. ABSTRACT

The Single Instruction Multiple Data (SIMD) execution model is becoming paramount in achieving a high throughput in microprocessors these days. Not only do GPGPUs base their design on the SIMD execution model, but also traditional latency oriented x86 processors now incorporate SIMD instruction sets. The challenges in employing the SIMD execution model are to find enough parallelism and to remove divergence. The former is usually done by programmers. For example, the CUDA programming model specifically enforces the programmers to write the parallel programs by exposing micro-architectural parameters such as the cache sizes, the SIMD width, etc. However, removing divergence is much more difficult for programmers as often the branch directions are not known at the compile time. Therefore, many researchers are attempting to mitigate the negative effects of branch divergence by innovative ideas [1]. In this report, we analyze the impacts that branch divergence has on the SIMD execution model, and later, introduce one of the most recent proposals, a branch fusion technique to tackle these divergent branches.

2. INTRODUCTION

A highly parallel GPGPU SIMD execution is increasing the gap between a throughput oriented programming model and a latency oriented programming model. However, the SIMD model is only effective when there are enough concurrently running threads that can hide long memory access latencies. Not enough parallelism or excessive branch divergence within a program reduce the number of concurrently running threads, and eventually, fails to hide long memory latencies. These causes translate to performance bottlenecks and remain to be interesting research questions to improve the SIMD execution performance. The main focus of our report is to analyze the effect of branch divergence.

A SIMD execution model consists of many processing elements (PEs) where a number of PEs executes the same instruction on different data sets. In most basic arithmetic operations, the efficiency of those PEs is very high assuming that the programmer wrote a highly parallel code. However, branches are executed differently on this platform. Since the direction of branches is not the same across different data sets, threads are divided into two different sets: taken and not taken sets. This reduces the efficiency of PEs as only a subset of a warp (a set of threads executing the same instruction) can be executed at the same time. Usually, this process is done through a mask where threads with the same branch direction are masked off, so those with the opposite branch directions get executed. In the next cycle, those previously masked off get executed. This lowers the parallelism, reduces the efficiency of the entire cores, and ultimately, hurts performance.

The size of basic blocks on branch divergence paths is critical to performance. If those basic blocks are large, then the duration of the execution on divergence paths becomes longer. In other words, there are many PEs sitting at idle as some threads are masked off due to branch divergence. The technique presented in this report is called branch fusion. The goal of this technique is to identify a segment of code that two divergent paths have in common and fuse them together. What we achieve at the end is smaller basic blocks on divergent paths and larger basic blocks on non-divergent paths, so we can increase the efficiency of PEs and hence its performance. By effectively decreasing the size of divergent paths, we believe that reasonable performance can be gained through this technique.

3. BRANCH DIVERGENCE AND FUSION

The branch divergence analysis aims to identify a set of divergent variables in a given CUDA program. A variable V is said to be divergent if there are two threads T1 and T2 with shared memory S1 and S2 such that S1(V) != S2(V) at the same moment of program execution, i.e. different threads see different values for the same variable. In the PTX execution model, the various threads are grouped into warps and at a branch statement, these threads in a warp take different paths (based on their active masks). Only when all the threads reconverge can the warp continue with the execution. This clearly shows that performance will significantly reduce if there is more branching in a program. Another main problem is that this divergence is input dependent and any kind of profiling method to identify these is going to suffer from a large slowdown. A definitive analysis to identify which branches are divergent and which branches never diverge can be implemented as a pass on the Ocelot compiler framework.

The following rules identify when a variable is divergent – v is the threadID (always different for different threads), v is defined atomically, v is data dependent on a divergent variable and v is sync dependent on a divergent variable.

> *- A variable v is data dependent on a variable u if there is some assignment instruction that defines v and uses u.*
>
> *- A variable v is sync dependent on a predicate variable p if v may reach a synchronization point with a different value for different threads, depending on how threads branch on p.*

The transitive closure of the set of dependent variables can be determined by program slicing given the propagation of data dependencies. But since we need to consider sync dependencies as well, simple program slicing doesn't work out, and we propose another definition for sync dependence, which they use for further stages of analysis.

- The influence region of a predicate is the set of basic blocks that may (or may not) be reached depending on the value of the predicate, up to the synchronization barrier (which is placed at the immediate post-dominator of the branch).

- For a branch instruction with lp as its synchronization point, a variable v is sync dependent if and only if, v is defined inside the influence region of the branch and v reaches lp.

The set of divergent variables can be found out by traversing its data dependence graph. Sync dependencies are handled by converting them into data dependencies. This is done by using the Gated-SSA form (based on the algorithm presented in the paper). The branch divergence analysis algorithm is done in two steps – changing the intermediate representation to the Gated-SSA form, and then traversing the data dependence graph to identify divergent branches. This completes the first part of the project.

This is a new method of divergence optimization proposed by the authors once we have identified the set of divergent branches. It aims at merging common code extracted from different divergent program paths. The simplest way of implementing this would be to identify the longest common subsequence of instructions between the two paths of a branch, and then merging them. This is solved using the Smith-Walterman sequence alignment algorithm – which involves building the profitability matrix to store the gains of each possible pairing of instructions and then traverse the matrix backwards to discover the overall best possible instruction alignment. A point to note here is that not all branches cause divergence, and the above method should be applied only for divergent branches. Also, we are allowed to merge instructions with different opcodes, as long as there exists an identity between these operands (eg: the comparison operator: $p = a > b$ is equivalent to $p = b < a$).

Our addition to the above project involves the peephole optimization, which is used specifically for non-divergent branches. We plan to incorporate a small look-ahead window for instructions. When it is known that all the threads in a warp go in the same path at a particular non-divergent branch instruction, we can replace the normal *br* instruction with the *br.uni* instruction provided in the PTX language to gain additional performance benefits.

4. METHODOLOGY

This section describes in detail the framework required to perform Divergence Analysis on CUDA programs. We used the Ocelot Compiler Framework [2] [3] for performing the divergence analysis pass and GPGPU-Sim Framework [4] [5] for the simulation and analysis. In addition, we have also utilized the NVIDIA Visual Compute Profiler to actually run benchmarks on real hardware. A brief overview of Ocelot is present in Section 4.1, GPGPU-Sim in Section 4.2 and Visual Compute Profiler in Section 4.3.

4.1 Ocelot Compiler Framework

GPU-Ocelot is a dynamic compilation and binary translation infrastructure for CUDA developed at Georgia Institute of Technology. It includes a well-developed compiler framework for the PTX Instruction Set Architecture and also implements the CUDA Runtime API to execute the PTX kernels on several types of backend execution targets.

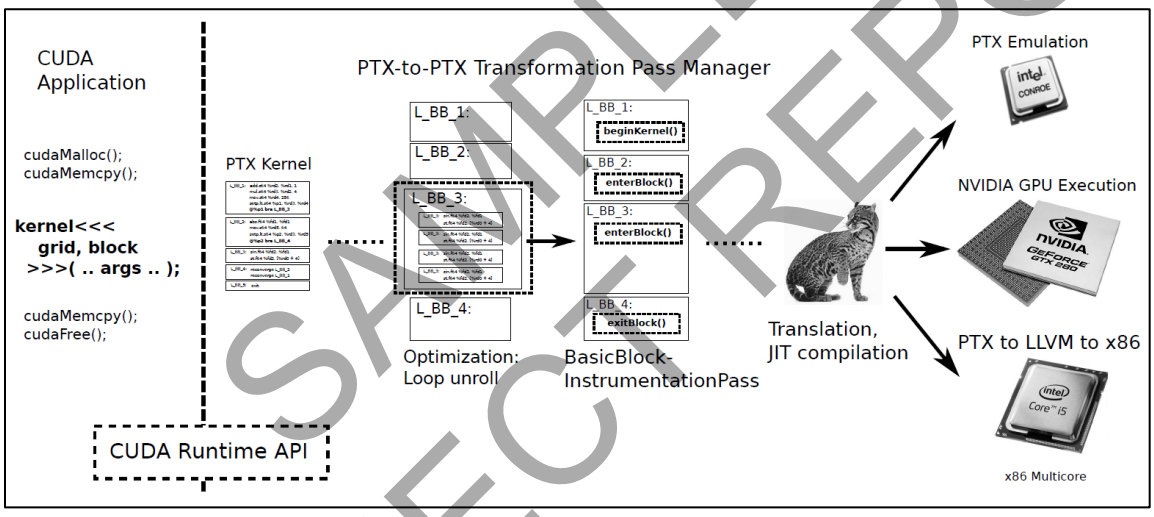


Figure 1: Ocelot Infrastructure

Figure 1 shows an overview of Ocelot's infrastructure. The CUDA application passes through the CUDA Runtime API which extracts the PTX kernel from the native CUDA code. This PTX kernel is then optimized by performing various passes after which it is translated to suit the specific backend target. Though we haven't used Ocelot's CUDA Runtime API to actually simulate execution the CUDA application in this project (we preferred GPGPU-Sim for simulation as it is more detailed), we heavily used the Compiler infrastructure provided to write the Divergence analysis pass. Ocelot is written in a way that is reminiscent of the LLVM Framework in the sense that one can easily add custom passes to the PTX to perform very detailed and specific optimizations. In essence, we chose Ocelot for the project as it helped in providing a layer of compilation support for GPU applications, which primarily is what we needed.

4.2 GPGPU-Sim

GPGPU-Sim is a cycle-accurate GPU simulator developed at the University of British Columbia that is used to simulate CUDA programs. In the CUDA programming model, the GPU is treated as a co-processor onto which an application running on a CPU can launch a massively parallel compute kernel. The kernel is comprised of a grid of scalar threads. Within a grid, threads are grouped into blocks, also called Cooperative thread arrays (CTAs). Within a single CTA, threads have access to a common fast memory called the shared memory and can, if desired, perform barrier synchronizations.

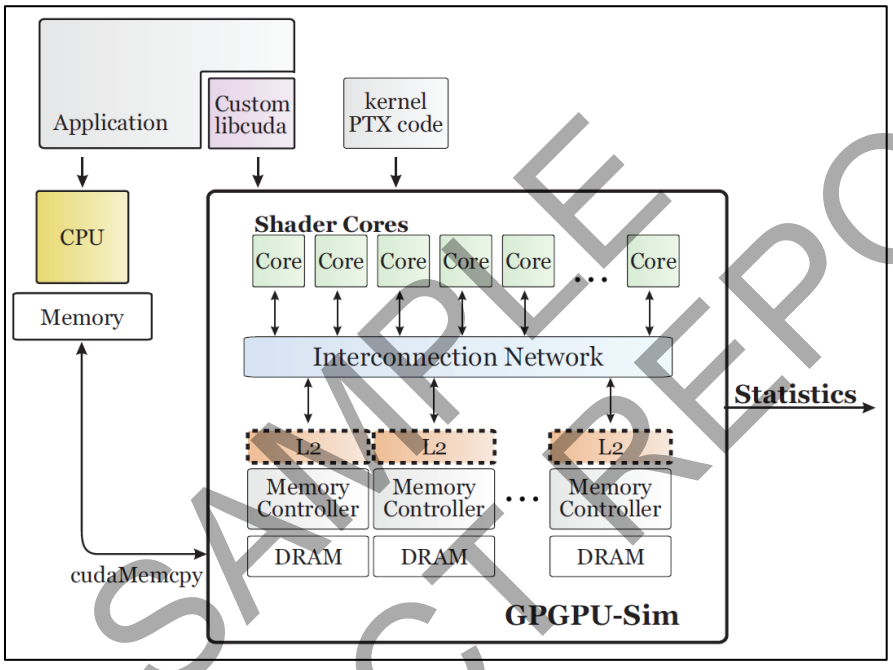


Figure 2: GPGPU-Sim Infrastructure

Figure 2 shows the baseline GPU Architecture that is modeled by GPGPU-Sim. The GPU consists of a collection of small data-parallel compute cores (shader cores), connected by an interconnection network to multiple memory modules (memory controller). Threads are distributed to shader cores at the granularity of entire CTAs, while per-CTA resources, such as registers, shared memory space, and thread slots, are not freed until all threads within a CTA have completed execution. If resources permit, multiple CTAs can be assigned to a single shader core, thus sharing a common pipeline for their execution. An important component here is the customized CUDA runtime which simulates the behavior of the actual GPU while enabling the logging of various useful metrics at each cycle of execution. Each call to a CUDA function in the program is replaced by an equivalent GPGPU-Sim CUDA call which has various instrumentation techniques to collect the required data. We use GPGPU-Sim in our project to calculate the effectiveness of our branch divergence analysis and to compare the performance of the optimized kernels with the original kernels.

4.3 NVIDIA Visual Compute Profiler

NVIDIA Visual Compute Profiler is a tool that can gather various statistics from the actual NVIDIA GPU card. Our profiler performs the gathering of statistics on NVIDIA's newest Fermi architecture. The reason why this tool is more accurate than many other simulators is that it monitors the actual hardware. When the GPU code is compiled and sent to the GPU card, the code is compiled again by NVIDIA's JIT and the programmer does not have any idea of what transformation is happening at this stage. For example, the register allocation on Fermi is done dynamically by this JIT and the programmer has no control over how many registers will be allocated per thread. At the same time, the programmer has no idea of how many registers are actually allocated as this is dynamically decided by the JIT. However, using this tool, we can get many different metrics such as the number of spill registers or the cache hit rate. Figure 3 shows the interface of the profiler along with various metrics that we can obtain from this tool.

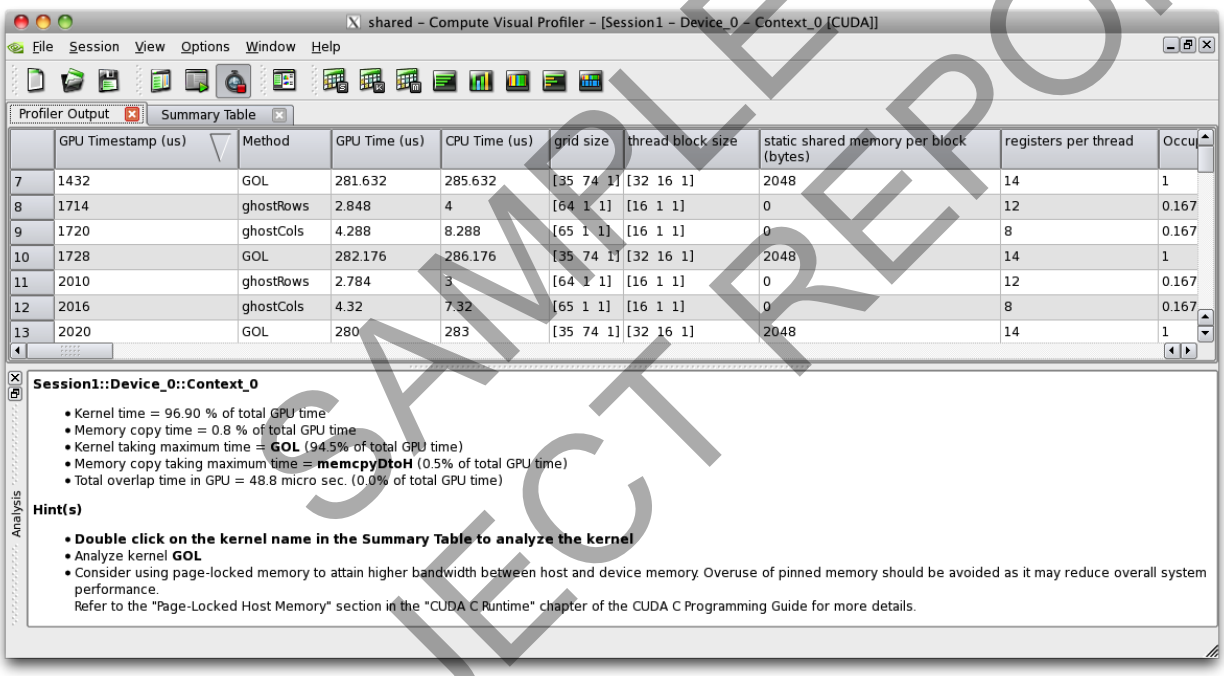


Figure 3: NVIDIA Visual Compute Profiler

Our particular interest was to look at the core computation time. When we put a timer in the CUDA code, the timer includes the total computation time plus the total data transfer time between the CPU and GPU. We are interested in how much time is saved in the computation part. In other words, we are interested in how much time is saved by fusing branches in the CUDA code. The profiler can distinguish how much the data transfer time takes, so that we can get the actual computation time. In our analysis section, we are using this computation time to analyze the performance improvement from the branch fusion.

5. IMPLEMENTATION

To begin implementing the Divergence Analysis pass, we first needed to thoroughly understand the different classes already implemented in Ocelot, which would be useful for the analysis. Ocelot documentation is almost non-existent so we had to spend quite some time scouting through the existing code to figure out the correct usage of each class and its methods. After a comprehensive study of the existing components, we identified the following classes as the most important for our project – DataflowGraph, Instruction, PostdominatorTree and BranchInfo along with their various subclasses. The general structure of writing a pass is very similar to LLVM which was convenient as we had worked on LLVM before. The implementation can be broadly split into two main sections -- divergence analysis and branch fusion. These are covered in Sections 5.1 and 5.2 respectively. Section 5.3 covers the details about adding the newly implemented passes on Ocelot and obtaining the CFG of each of the PTX kernels.

5.1 Divergence Analysis

The paper provided a clear set of rules as to when we could classify a branch as divergent or not. Following these guidelines strictly, we proceeded to analyze the data flow that could potentially add divergent sources and further compute the divergence propagation for each divergent source. The next step was to analyze the control flow to detect any new divergent variables caused due to originally identified divergent variables. To perform this, we had to obtain information from all possible divergent branches in the kernel. For each branch, we have to look at the postdominator block for any new divergent variables, and if found, the divergence spread had to be recomputed. This step was repeated till no further change was observed. An array of BranchInfo objects was used to store the set of possible diverging branches. We iterated through all the branch instructions in each basic block (*br* instructions can only be present at the end of each basic block, so we directly jumped to Block->Instructions.end() to achieve minimum overhead of running this pass) and stored the predicate information in the corresponding BranchInfo object. At the end of this pass, we had a list of divergent branches in the kernel. The complete code for the analysis part came up to around 400 lines and is present in Appendix A.

5.2 Branch Fusion

Once the divergence analysis portion was complete, the next part was the optimization technique which involved fusing the longest common subsequence of instructions in each path of the branch. This was written as a separate pass which would call the Divergence Analysis pass. The branch fusion pass was written as a runOnKernel pass which would be invoked every time a new kernel is launched on the GPU. This pass checks each branch to see if it's divergent or not. If divergent, we then proceeded to iterate through the instructions on each path looking for common instructions. Every time a matching instruction was found, we incremented its profitability index which then served as a metric to decide if the branches should be fused or not. If the profitability is low, then it was better not to fuse the branches as the additional cost in re-

arranging the basic blocks and inserting new edges to the control flow graph would far outweigh the benefits of branch fusion. We then had to implement a function called Finalize() which would iterate through the original phi functions and modify them to suit the changed arrangement. At this point, we also use one of Ocelot's inbuilt functions called UpdateRegisters to update the values in registers and replace any of them if necessary. This pass came up to around 300 lines of code and is present in Appendix B.

5.3 Registering Passes on Ocelot

Once the passes were written, the final step involved registering this pass into Ocelot's PassManager framework. A small snippet of code was inserted into PassManager.cpp and Ocelot was recompiled so that these newly implemented passes were registered. To execute this pass, we used PTXOptimizer, Ocelot's standalone tool used specifically for optimizing the PTX code (and not worrying about Ocelot's CUDARuntime). PTXOptimizer has command line parameters using which we can specify the PTX input file and the set of optimization passes that are to be performed. The command we used for running our benchmarks were:

```
$ PTXOptimizer –i input.ptx –p block-unification –c true
```

This command means that the pass called *'block unification'* is performed on the input file *'input.ptx'* and it generates the optimized file *'_optimized_input.ptx'* and the –c option is used to specify if we want the control flow graph also to be dumped or not.

The CFG is dumped in a .dot format which stores the list of basic blocks along with the instructions in each basic block. It also contains the edge information of the control flow graph. The PyGraphViz package is used to visualize the .dot files. The command for converting the above .dot file to the corresponding .pdf file is:

```
$ dot –Tpdf filename.dot –o filename.pdf
```

A sample .dot file and its corresponding CFG are shown in Figure 4 below.

```
digraph G{
0 [label="entry_0"] [color=BLACK]
1 [label="no_exit_1"] [color=RED]
2 [label="UnifiedReturnBlock_2"]
[color=GREEN]
3 [label="endif_1"] [color=RED]
4 [label="then_3"] [color=BLUE]
0 -> 1 [label="1"]
0 -> 2 [label="0"]
1 -> 4 [label="1"]
1 -> 3 [label="0"]
3 -> 1 [label="1"]
3 -> 2 [label="0"]
}
```

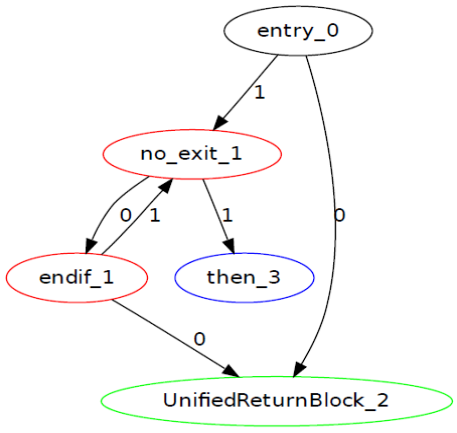


Figure 4: Using PyGraphViz to generate the CFG

6. ANALYSIS

In order to evaluate our implementation, we ran four different benchmarks on the GPGPU-Sim simulator. These four benchmarks have different degrees of divergence within the code. Some benchmarks do not show any room for the branch fusion whereas others offer many opportunities for the fusion process as they have many common instructions on both paths of the branch. The detailed control flow graphs of those four benchmarks are rather large, so they are attached in Appendix C. We have presented both the original and the optimized graphs and it is clear to see which basic blocks are fused.

We ran four benchmarks in order to evaluate our branch fusion implementation. Our evaluation metric is the number of instructions per second normalized to the original code. For example, if the performance for a particular benchmark stays the same, then our graph will show 1 when normalized to the original algorithm. Figure 5 shows the result of our evaluations across the different benchmarks. The *nqueen* benchmark solves the problem of placing N queens on an NxN chess board. In this benchmark, a single thread performs most computations. In the *bfs* benchmark, a breadth-first search on a graph is performed where each thread is assigned a graph

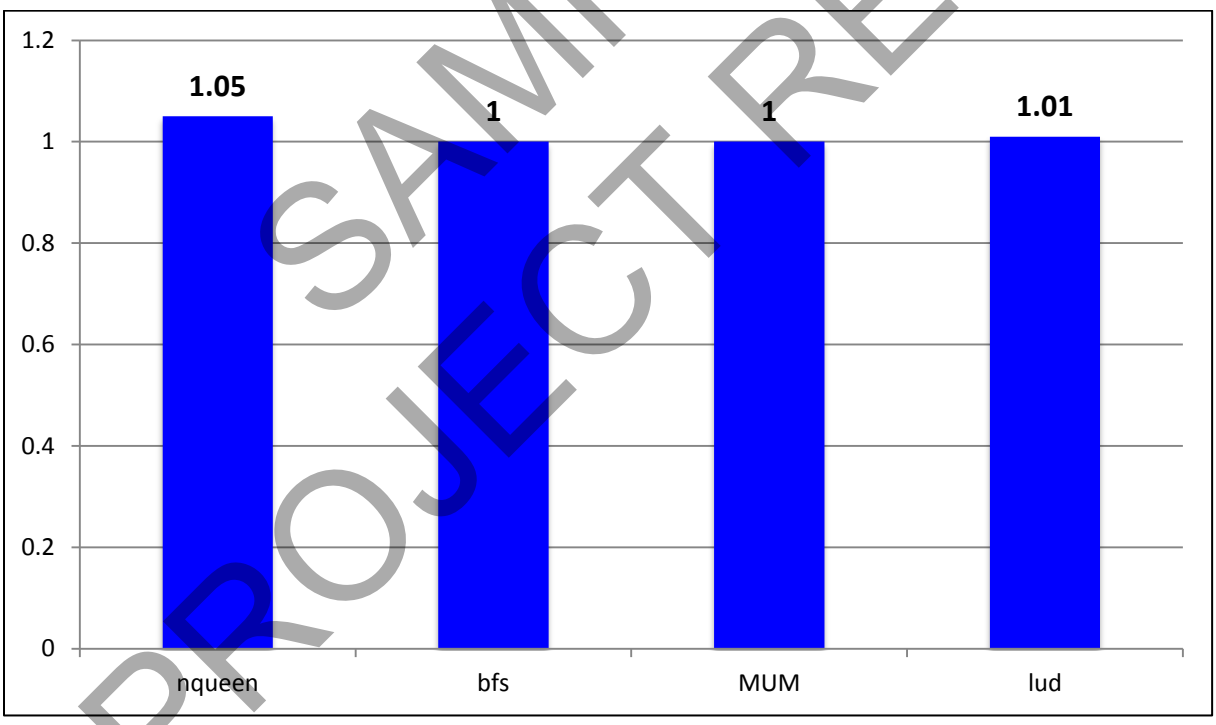


Figure 5: Performance Improvement in Percentage for Various Benchmarks

path. The *MUM* benchmark is a pairwise local sequence alignment algorithm for DNA. This benchmark is not compute intensive and spends most of the execution time on memory stalls. Lastly, the *lud* benchmark calculates the solutions of linear equations by decomposing the matrix

into upper and lower triangular matrices. It is again a compute bound application which assigns different calculations to different threads thereby negating the time spent on memory stalls.

The *bfs* benchmark does not have any branches which have a common set of instructions, so there is absolutely no opportunity for the fusion, so there is no performance improvement. The *MUM* benchmark has some branches where instructions on different branch paths can be merged. However, when our code analyzes these branches with our profitability metric, the result turns out that the overheads of fusing the branches will be greater than the savings. Therefore, our implementation left the original code as it is although we could fuse branches. The *lud* benchmark consists of long non-branch segments, so the proportion of the branch paths is small in the entire program. Our profitability metric still found that it is beneficial to fuse branches, so those branch paths were fused. Yet, as these paths a small portion of the entire program, we did not see huge improvement. Lastly, the *nqueen* benchmark contains a considerable amount of branch instructions, with a reasonably good profitability metric, so our implementation went ahead and fused them. That is why this benchmark achieved the largest performance gain. The detailed annotated CFGs are present in Appendix C for reference.

As seen from our evaluations, profitability metric is very useful as it can estimate where we actually get savings or not. For that reason, the figure has shown that none of our benchmarks have performance degradation since our implementation does not apply the technique if the profitability is low. We believe that our technique can be applied for any general benchmark without hurting performance as the technique will only be applied to those benchmarks that have high profitability.

7. CHALLENGES

There were a lot of challenges that we faced while implementing this project – The first of which was the lack of documentation for the Ocelot Compiler Framework. This made it particularly hard to identify the classes that we needed for implementing the Divergence Analysis pass. Only after a detailed study of the source code and looking through each of the class definitions, we could find the required details. It might have been possible to improve the efficiency of the code had we known the various required methods before-hand. However, the pass was still implemented correctly as described in the paper.

The next hurdle that we faced was the branch fusion technique. Initially, the plan was to have just a single pass to do both Divergence Analysis and Branch Fusion. However, we learnt that whenever we had to change the control flow graph of the PTX kernel for fusion, the new divergence propagation of the graph had to be recomputed for every change. This made us realize that the best way to proceed was to split them into two different passes, one for the

analysis and one for the optimization. The optimization pass which fused the branches could independently call the Divergence Analysis pass whenever any change to the CFG was made.

In retrospect, the implementation of both these passes weren't as hard as the problems we faced during the experimentation section. We identified a varied set of CUDA benchmarks with different properties. However, a lot of benchmarks failed to run on GPGPU-Sim as a few newly implemented CUDA calls used in the benchmarks were not yet supported in the simulator. (The set of benchmarks that failed include DotProd, MergeSort, Heartwall, NeuralNetworks, RayTrace etc). The benchmarks that we finally narrowed down to showed very little improvement but we are certain that other benchmarks (some of which have ideal CFGs for branch fusion) would certainly benefit a lot from this optimization pass.

To prove the above point, we have shown here the dump of running the optimization pass on Mergesort. The pass by itself makes a lot of changes to the original CFG which are certain to improve the performance. However, we don't have performance benefit numbers because we could not run this on the simulator.

```
$ PTXOptimizer –i mergeSort.ptx –p block-
unification –c true
>>>
Unifying:   ld.param.s32 %r180,
[__cudaparm__Z13lud_perimeterPfii_offset]
Unifying:   add.s32 %r181, %r180, 8
Unifying:   cvt.s64.s32 %r182, %r0
Unifying:   cvt.s32.u16 %r13, %tid.x
Unifying:   cvt.s64.s32 %r14, %r13
Unifying:   mul.lo.s32 %r184, %r181, %r776
Unifying:   mul.wide.s32 %r185, %r0, 4
Unifying:   add.s32 %r186, %r184, %r180
Unifying:   add.u64 %r187, %r185, %r775
Unifying:   add.s32 %r188, %r186, %r0
Unifying:   cvt.s64.s32 %r189, %r188
Unifying:   mul.wide.s32 %r190, %r188, 4
Unifying:   ld.param.s32 %r16,
[__cudaparm__Z13lud_perimeterPfii_offset]
Unifying:   mul.lo.s32 %r17, %r776, %r16
Unifying:   mul.wide.s32 %r18, %r13, 4
Unifying:   add.s32 %r19, %r17, %r16
Unifying:   add.u64 %r20, %r18, %r775
Unifying:   add.s32 %r21, %r19, %r13
Unifying:   cvt.s64.s32 %r22, %r21
Unifying:   mul.wide.s32 %r23, %r21, 4
Unifying:   ld.global.f32 %r229, [%r228 + -64]
Unifying:   st.shared.f32 [%r187 + 832], %r229
Unifying:   add.s32 %r230, %r224, %r776
Unifying:   add.s32 %r231, %r0, %r230
Unifying:   cvt.s64.s32 %r232, %r231
Unifying:   mul.wide.s32 %r233, %r231, 4
Unifying:   add.u64 %r234, %r777, %r233
Unifying:   ld.global.f32 %r235, [%r234 + -64]
Unifying:   st.shared.f32 [%r187 + 896], %r235
Unifying:   add.u32 %r236, %r180, %r4
Unifying:   add.u64 %r237, %r185, %r179
Unifying:   add.u32 %r238, %r236, 16
Unifying:   mul.lo.u32 %r239, %r776, %r238
Unifying:   add.u32 %r240, %r180, %r239
Unifying:   add.s32 %r241, %r240, %r0
Unifying:   cvt.s64.s32 %r242, %r241
Unifying:   mul.wide.s32 %r243, %r241, 4
Unifying:   add.u64 %r244, %r777, %r243
Unifying:   ld.global.f32 %r245, [%r244 + -64]
Unifying:   st.shared.f32 [%r237 + -64], %r245
Unifying:   add.s32 %r246, %r776, %r240
Unifying:   add.s32 %r247, %r246, %r0
Unifying:   cvt.s64.s32 %r248, %r247
Unifying:   @%p2 ld.global.f32 %r38, [%r794 +
0]
<<< End
```

To get past this issue of relying entirely on the simulator, the final benchmark LUD was actually run on a NVIDIA Fermi based Tesla GPU. NVIDIA's Visual Compute Profiler helped us gather data for this particular benchmark. Though it was not as detailed as the information we got from GPGPU-Sim, it was enough for us to conduct experiments on different matrix sizes and compute the improvement.

8. FUTURE SCOPE AND CONCLUSION

8.1 Future Scope

There is a lot of scope for improvement in this project that we have identified. The actual implementation can be simplified with the use of suitable methods which would reduce the overhead of running this pass on the PTX kernel. However, due to the time constraint and due to the lack of documentation, we went ahead with just the correct implementation and not worrying about the performance issues.

Another possible place for improvement is the merging of this pass with the Ocelot's Instrumentation Framework called Lynx. Lynx is a separate branch of Ocelot with the focus on dynamic instrumentation. Proceeding to the area of experimentation and benchmarking, we were restricted from running many benchmarks due to the lack of support from the simulator. We can extend our work to purely testing on physical hardware using the NVIDIA Visual Compute Profiler. Though the stats obtained would not be as detailed, it gives accurate and reliable numbers.

The goal of this project was to show the performance improvement of this particular optimization technique and we have successfully shown that it is possible (albeit very small improvements for the benchmarks that we tested).

8.2 Conclusion

In this project, we successfully implemented the divergence analysis pass and also the branch fusion optimization technique as presented in the PACT 2011 paper. Since branch divergence is one of the major bottlenecks in improving highly parallel GPU applications, it is important to optimize our code taking care of this problem accordingly. By working on this project, we also got exposure to various useful tools that will aid us in future research and also gave us an insight about new optimization techniques for GPGPUs.

9. ACKNOWLEDGMENTS

We would like to extend our warm thanks to the authors of Ocelot – Gregory Diamos and Andrew Kerr who have been kind enough to answer all our questions about Ocelot and clearing doubts that we had along the way. Special thanks to Bruno Coutinho, the author of the Divergence Analysis paper, for giving us the benchmarks to test our implementation. Last but not the least, our advisor Prof. Vijay Janapa Reddi, for introducing us to the world of GPGPUs and for his constant support throughout the semester.

10. REFERENCES

[1] "Divergence Analysis and Optimizations" – Bruno Coutinho, Diogo Sampaio, Fernando Magno Quinto Pereira and Wagner Meira Jr., Universidade Federal de Minas Gerais, Brazil

[2] "The Design and Implementation of Ocelot's Dynamic Binary Translator from PTX to multi-core x86" – Gregory Diamos, Georgia Institute of Technology, USA

[3] "Ocelot: A Dynamic Compiler for Bulk-Synchronous Applications in Heterogeneous Systems" – Gregory Diamos, Georgia Institute of Technology, USA

[4] "Analyzing CUDA Workloads Using a Detailed GPU Simulator" – Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong and Tor M. Aamodt, University of British Columbia, Canada

[5] "Exploring GPGPU Workloads: Characterization Methodology, Analysis and Microarchitecture Evaluation Implications" – Nilanjan Goswami, Ramkumar Shankar, Madhura Joshi and Tao Li, University of Florida, USA

APPENDIX A – Divergence Analysis Pass

```
#include <ocelot/ir/interface/PostdominatorTree.h>
#include <assert.h>

namespace analysis {

void DivergenceAnalysis::_analyzeDataFlow()
{
DataflowGraph &Graph = *_kernel->dfg();
DataflowGraph::const_iterator block = Graph.begin();
DataflowGraph::const_iterator endBlock = Graph.end();

/* Finding out divergent sources */
for (; block != endBlock; ++block)
{
	DataflowGraph::PhiInstructionVector::const_iterator phiInstruction = block->phis().begin();
	DataflowGraph::PhiInstructionVector::const_iterator endPhiInstruction = block->phis().end();
	for (; phiInstruction != endPhiInstruction; phiInstruction++)
	{
		for (DataflowGraph::RegisterVector::const_iterator si = phiInstruction->s.begin(); si !=
phiInstruction->s.end(); ++si)
		{
			_divergGraph.insertEdge(si->id, phiInstruction->d.id);
			si->type;
		}
	}

	DataflowGraph::InstructionVector::const_iterator ii = block->instructions().begin();
	DataflowGraph::InstructionVector::const_iterator iiEnd = block->instructions().end();
	for (; ii != iiEnd; ++ii)
	{
		ir::PTXInstruction *ptxInstruction = NULL;
		bool atomicvar = false;
		set<ir::PTXOperand::SpecialRegister> divSources;
		if (typeid(ir::PTXInstruction) == typeid(*(ii->i)))
		{
			ptxInstruction = static_cast<ir::PTXInstruction*> (ii->i);
			if(ptxInstruction->opcode == ir::PTXInstruction::Opcode::Atom)
			{
				atomicvar = true;
			}
			if (ptxInstruction->a.addressMode == ir::PTXOperand::AddressMode::Special)
			{
				divSources.insert(ptxInstruction->a.special);
			}
			if (ptxInstruction->b.addressMode == ir::PTXOperand::AddressMode::Special)
			{
				divSources.insert(ptxInstruction->b.special);
			}
			if (ptxInstruction->c.addressMode == ir::PTXOperand::AddressMode::Special)
			{
				divSources.insert(ptxInstruction->c.special);
			}
		}

		divSources.erase(ir::PTXOperand::SpecialRegister_invalid);
		DataflowGraph::RegisterPointerVector::const_iterator destReg = ii->d.begin();
		DataflowGraph::RegisterPointerVector::const_iterator destEndReg = ii->d.end();

		for (; destReg != destEndReg; destReg++)
		{
			if (divSources.size() != 0)
			{
				set<ir::PTXOperand::SpecialRegister>::const_iterator dirtSource =
divSources.begin();
				set<ir::PTXOperand::SpecialRegister>::const_iterator endDirtSource =
divSources.end();
				for (; dirtSource != endDirtSource; dirtSource++)
				{
					_divergGraph.insertEdge(*dirtSource, *destReg->pointer);
				}
			}

			DataflowGraph::RegisterPointerVector::const_iterator sourceReg = ii->s.begin();
			DataflowGraph::RegisterPointerVector::const_iterator sourceEndReg = ii->s.end();

			for (; sourceReg != sourceEndReg; sourceReg++)
```

```
                        {
                                _divergGraph.insertEdge(*sourceReg->pointer, *destReg->pointer);
                        }

                        if(atomicvar)
                        {
                                _divergGraph.setAsDiv(*destReg->pointer);
                        }
                }
        }
} //end of for loop
_divergGraph.computeDivergence();
}

/* Analyze the control flow and check which new divergent variables have been introduced */
void DivergenceAnalysis::_analyzeControlFlow()
{

std::set<BranchInfo> branches;

{
/* Maintain a list of branches that can be divergent (not a bra.uni and has a predicate) */
DataflowGraph::const_iterator block = _kernel->dfg()->begin();
DataflowGraph::const_iterator endBlock = _kernel->dfg()->end();

ir::PostdominatorTree dtree(_kernel->cfg());
for (; block != endBlock; ++block)
{
        ir::PTXInstruction *ptxInstruction = NULL;
        if (block->instructions().size() > 0)
        {
                /* Branch instructions can only be the last instruction of a basic block */
                DataflowGraph::Instruction lastInstruction = *(--block->instructions().end());
                if (typeid(ir::PTXInstruction) == typeid(*(lastInstruction.i)))
                {
                        ptxInstruction = static_cast<ir::PTXInstruction*> (lastInstruction.i);
                        if ((ptxInstruction->opcode == ir::PTXInstruction::Opcode::Bra) && (ptxInstruction->uni
== false) && (lastInstruction.s.size() != 0))
                        {
                                ir::ControlFlowGraph::iterator CFGBlock = _kernel->cfg()->begin();
                                ir::ControlFlowGraph::iterator CFGEndBlock = _kernel->cfg()->end();
                                for (; CFGBlock != CFGEndBlock; CFGBlock++)
                                {
                                        if (CFGBlock->label == block->label())
                                        {
                                                break;
                                        }
                                }
                                assert(CFGBlock != CFGEndBlock);
                                assert(lastInstruction.s.size() == 1);
                                unsigned int id = dtree.getPostDominator(CFGBlock)->id;
                                DataflowGraph::const_iterator postDomBlock = _kernel->dfg()->begin();
                                DataflowGraph::const_iterator endPostDomBlock = _kernel->dfg()->end();
                                for (; postDomBlock != endPostDomBlock; ++postDomBlock)
                                {
                                        if (postDomBlock->id() == id)
                                        {
                                                break;
                                        }
                                }
                                if (postDomBlock != endPostDomBlock)
                                {
                                        BranchInfo newBranch(&(*block), &(*postDomBlock),
*lastInstruction.s.begin()->pointer, lastInstruction,*_kernel->dfg(), _divergGraph);
                                        branches.insert(newBranch);
                                }
                        }
                }
        }
} //end of for loop
}

_branches = branches;
std::set<BranchInfo> divergent;

/* Populate the divergent branches set */
std::set<BranchInfo>::iterator branch = branches.begin();
std::set<BranchInfo>::iterator endBranch = branches.end();
```

```
while (branch != endBranch)
{
	if (isDivBlock(branch->block()))
	{
		divergent.insert(*branch);
		_divergentBranches.insert(*branch);
		branches.erase(branch);
		branch = branches.begin();
		endBranch = branches.end();
	}
	else
	{
		_notDivergentBranches.insert(*branch);
		branch++;
	}
}

/* Test for divergence on the post-dominator block of every divergent branch instruction */
while (divergent.size() > 0)
{
	BranchInfo branchInfo = *divergent.begin();
	branchInfo.populate();
	DataflowGraph::PhiInstructionVector phis = branchInfo.postDominator()->phis();
	DataflowGraph::PhiInstructionVector::const_iterator phi = phis.begin();
	DataflowGraph::PhiInstructionVector::const_iterator endphi = phis.end();

	bool newlyCreatedDiv = false;
	for (; phi != endphi; phi++)
	{
		DataflowGraph::RegisterVector::const_iterator source = phi->s.begin();
		DataflowGraph::RegisterVector::const_iterator endSource = phi->s.end();
		for (; source != endSource; source++)
		{
			if (branchInfo.isTainted(source->id))
			{
				_addPredicate(*phi, branchInfo.predicate());
				newlyCreatedDiv = true;
			}
		}
	}
	divergent.erase(branchInfo);

	/* When newly divergent variables are found */
	if (newlyCreatedDiv)
	{
		_divergGraph.computeDivergence();
		branch = branches.begin();
		while (branch != endBranch)
		{
			if (isDivBlock(branch->block()))
			{
				divergent.insert(*branch);
				_divergentBranches.insert(*branch);
				branches.erase(branch);
				branch = branches.begin();
				endBranch = branches.end();
			}
			else
			{
				_notDivergentBranches.insert(*branch);
				branch++;
			}
		}
	}
}
}

/* Adds a predicate as a predecessor of a variable */
void DivergenceAnalysis::_addPredicate(const DataflowGraph::PhiInstruction &phi, const
graph_utils::DivergenceGraph::node_type &predicate)
{
_divergGraph.insertEdge(predicate, phi.d.id);
}

DivergenceAnalysis::DivergenceAnalysis() //Constructor
{
_doCFGanalysis = true;
_kernel = NULL;
}
```

```
/* Analyze the control and data flows searching for divergent variables and blocks */
void DivergenceAnalysis::runOnKernel(ir::Kernel &k)
{
if (typeid(ir::PTXKernel) == typeid(k))
{
        _kernel = (ir::PTXKernel*) &k;
}
if (_kernel == NULL)
{
        return;
}
if(!_kernel->dfg()->ssa())
        _kernel->dfg()->toSsa();

run();
}


void DivergenceAnalysis::run()
{
if (_kernel == NULL)
{
        return;
}

_divergGraph.clear();
_analysisTime.tv_sec = 0;
_analysisTime.tv_usec = 0;
_branches.clear();
_divergentBranches.clear();
_notDivergentBranches.clear();

graph_utils::DivergenceGraph::node_set predicates;
_analyzeDataFlow();

if(_doCFGanalysis)
{
        _analyzeControlFlow();
}
}

bool DivergenceAnalysis::isDivBlock(DataflowGraph::iterator &block) const
{
if (block->instructions().size() == 0)
{
        return false;
}
return isDivBranch(--block->instructions().end());
}

bool DivergenceAnalysis::isPossibleDivBlock(DataflowGraph::const_iterator &block) const
{
if (block->instructions().size() == 0)
{
        return false;
}
const DataflowGraph::InstructionVector::const_iterator &instruction = --block->instructions().end();
if(typeid(ir::PTXInstruction) == typeid(*(instruction->i)))
{
        const ir::PTXInstruction &ptxI = *(static_cast<ir::PTXInstruction *> (instruction->i));
        return ((ptxI.opcode == ir::PTXInstruction::Bra) && !ptxI.uni);
}
return false;
}

bool DivergenceAnalysis::isPossibleDivBlock(const DataflowGraph::Block *block) const
{
if ((block == NULL) || (block->instructions().size() == 0))
{
        return false;
}
const DataflowGraph::InstructionVector::const_iterator &instruction = --block->instructions().end();
if(typeid(ir::PTXInstruction) == typeid(*(instruction->i)))
{
        const ir::PTXInstruction &ptxI = *(static_cast<ir::PTXInstruction *> (instruction->i));
        return ((ptxI.opcode == ir::PTXInstruction::Bra) && !ptxI.uni);
}
return false;
```

```
}

bool DivergenceAnalysis::isDivBranch(const DataflowGraph::InstructionVector::const_iterator &instruction) const
{
	return (isDivInstruction(*instruction) && isPossibleDivBranch(instruction));
}

bool DivergenceAnalysis::isPossibleDivBranch(const DataflowGraph::InstructionVector::const_iterator
&instruction) const
{
if(typeid(ir::PTXInstruction) == typeid(*(instruction->i)))
{
	const ir::PTXInstruction &ptxI = *(static_cast<ir::PTXInstruction *> (instruction->i));
	return ((ptxI.opcode == ir::PTXInstruction::Bra) && (!ptxI.uni));
}
return false;
}

bool DivergenceAnalysis::isDivInstruction(const DataflowGraph::Instruction &instruction) const
{
bool isDirty = false;
DataflowGraph::RegisterPointerVector::const_iterator reg = instruction.d.begin();
DataflowGraph::RegisterPointerVector::const_iterator endReg = instruction.d.end();
for (; (!isDirty) && (reg != endReg); reg++)
{
	isDirty |= _divergGraph.isDivNode(*reg->pointer);
}
if (isDirty)
{
	return true;
}

reg = instruction.s.begin();
endReg = instruction.s.end();

for (; (!isDirty) && (reg != endReg); reg++)
{
	isDirty |= _divergGraph.isDivNode(*reg->pointer);
}
return isDirty;
}

DataflowGraph::const_iterator DivergenceAnalysis::beginBlock() const
{
assert(_kernel != NULL);
return _kernel->dfg()->begin();
}

DataflowGraph::const_iterator DivergenceAnalysis::endBlock() const
{
assert(_kernel != NULL);
return _kernel->dfg()->end();
}

}

namespace std
{
	bool operator<(const analysis::BranchInfo x, const analysis::BranchInfo y)
	{
		return x.block()->id() < y.block()->id();
	}

	bool operator<=(const analysis::BranchInfo x, const analysis::BranchInfo y)
	{
		return x.block()->id() <= y.block()->id();
	}

	bool operator>(const analysis::BranchInfo x, const analysis::BranchInfo y)
	{
		return x.block()->id() > y.block()->id();
	}

	bool operator>=(const analysis::BranchInfo x, const analysis::BranchInfo y)
	{
		return x.block()->id() >= y.block()->id();
	}
}
```

APPENDIX B – Branch Fusion Pass

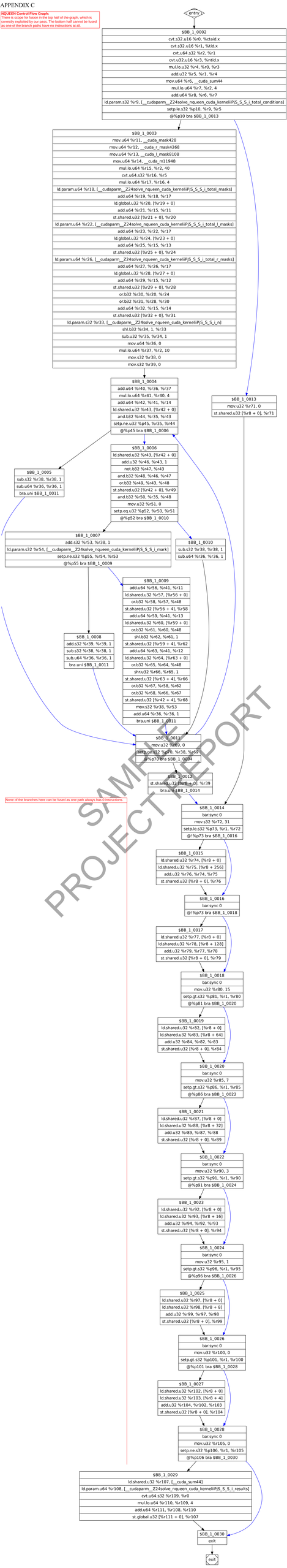
```
#include <ocelot/ir/interface/PTXKernel.h>
#include <ocelot/ir/interface/ControlFlowGraph.h>
#include <ocelot/ir/interface/PostdominatorTree.h>

using namespace analysis;

BlockUnificationPass::BlockUnificationPass():KernelPass(StaticSingleAssignment, "BlockUnification")
{
}

void BlockUnificationPass::initialize( const ir::Module& m )
{
}

void BlockUnificationPass::runOnKernel( ir::Kernel& k )
{
InstructionConverter instConv;
DataflowGraph::iterator unificationBranch;
DataflowGraph::iterator unificationTarget1;
DataflowGraph::iterator unificationTarget2;
BlockMatcher::MatrixPath bestPath;
float maxProfit = 0.0;

DivergenceAnalysis divAnalysis;
divAnalysis.runOnKernel(k);
do
{
        maxProfit = 0.0;
        DataflowGraph::iterator block = k.dfg()->begin();
        for (; block != k.dfg()->end(); ++block)
        {
                ir::ControlFlowGraph::const_iterator irBlock = block->block();
                DataflowGraph::const_iterator constBlock = block;
                if (irBlock->endsWithConditionalBranch() && divAnalysis.isDivBlock(constBlock))
                {
                        DataflowGraph::iterator fallthroughBlock = block->fallthrough();
                        DataflowGraph::iterator branchBlock = fallthroughBlock;
                        DataflowGraph::BlockPointerSet branchTargets = block->targets();
                        DataflowGraph::BlockPointerSet::const_iterator it = branchTargets.begin();
                        for (; it != branchTargets.end(); ++it)
                        {
                                if (*it != fallthroughBlock)
                                {
                                        branchBlock = *it;
                                        break;
                                }
                        }
                        ir::PostdominatorTree* pdomTree = k.pdom_tree();
                        ir::ControlFlowGraph::const_iterator postDomBlk = pdomTree->getPostDominator(block-
>block());
                        bool haveBranch2FallthroughPath = thereIsPathFromB1toB2(branchBlock->block(),
fallthroughBlock->block(),postDomBlk, new std::set<ir::ControlFlowGraph::BasicBlock*>);
                        bool haveFallthrough2BranchPath = thereIsPathFromB1toB2(fallthroughBlock->block(),
branchBlock->block(),postDomBlk, new std::set<ir::ControlFlowGraph::BasicBlock*>);
                        if (!haveBranch2FallthroughPath && !haveFallthrough2BranchPath)
                        {
                                // Calculate branch targets' unification gain
                                BlockMatcher::MatrixPath path;
                                float gain = BlockMatcher::calculateUnificationGain(k.dfg(), *fallthroughBlock,
*branchBlock, path,instConv, deviceCapability);
                                if (gain > maxProfit)
                                {
                                        maxProfit = gain;
                                        unificationBranch = block;
                                        unificationTarget1 = fallthroughBlock;
                                        unificationTarget2 = branchBlock;
                                        bestPath = path;
                                }
                        }
                }
        }

        if (maxProfit > 10.0)
        {
                // Unify the basic block pair with profit > some threshold
                cout << "Unifying blocks";
```

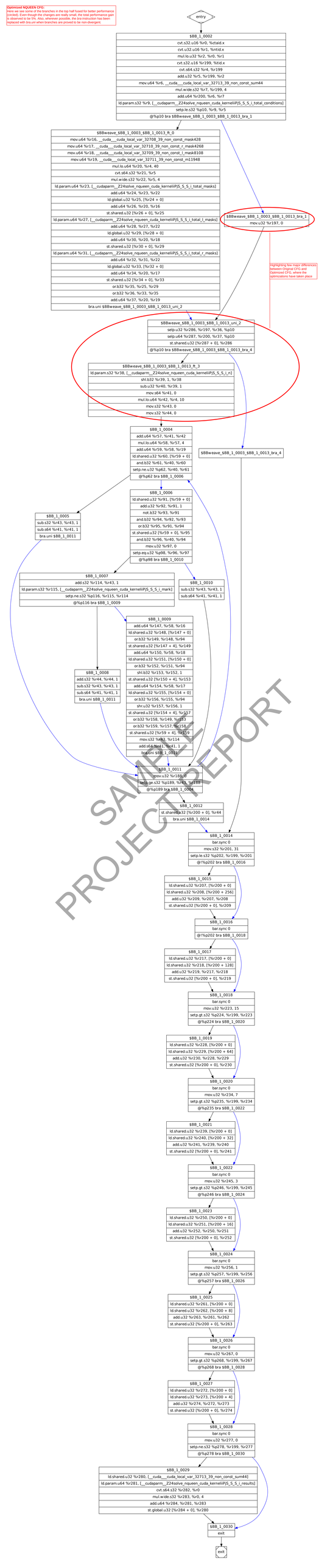
```
                fuseBlocks(unificationBranch, unificationTarget1, unificationTarget2, bestPath, k.dfg());
        }

        divAnalysis.run();
} while (false);
}

bool BlockUnificationPass::thereIsPathFromB1toB2(ir::ControlFlowGraph::const_iterator t,
                ir::ControlFlowGraph::const_iterator e,
                ir::ControlFlowGraph::const_iterator c,
                std::set<ir::ControlFlowGraph::BasicBlock*>* visited) const {
        const ir::ControlFlowGraph::BlockPointerVector& sucessors = t->successors;
        ir::ControlFlowGraph::BlockPointerVector::const_iterator succ = sucessors.begin();
        for (; succ != sucessors.end(); succ++) {
                if (visited->find(&(**succ)) == visited->end()) {
                        if (*succ == e) {
                                return true;
                        } else if (*succ != c) {
                                visited->insert(&(**succ));
                                if (thereIsPathFromB1toB2(*succ, e, c, visited)) {
                                        return true;
                                }
                        }
                }
        }
        return false;
}

void BlockUnificationPass::finalize()
{

}

void BlockUnificationPass::replaceRegisters(DataflowGraph* dfg, BlockExtractor& extractor)
{
const VariableAliases& varMap = extractor.getNewRegisterNames();
for (analysis::DataflowGraph::iterator block = dfg->begin(); block!= dfg->end(); ++block)
{
        // Iterate list of phi-functions, replacing the uses.
        for (DataflowGraph::PhiInstructionVector::iterator phi = block->phis().begin(); phi != block-
>phis().end(); ++phi)
        {
                for (unsigned u = 0; u < phi->s.size(); ++u)
                {
                        DataflowGraph::Register& phiRegId = phi->s[u];
                        VariableAliases::const_iterator regAliasIt = varMap.find(phiRegId.id);
                        if (regAliasIt != varMap.end())
                        {
                                phiRegId.id = regAliasIt->second;
                        }
                }
        }
}
}

// fusing the blocks together
void BlockUnificationPass::fuseBlocks(DataflowGraph::iterator branchBlock, DataflowGraph::iterator target1,
DataflowGraph::iterator target2, BlockMatcher::MatrixPath& extractionPath, DataflowGraph* dfg)
{
DataflowGraph::iterator oldFallthroughBlock = branchBlock;
DataflowGraph::iterator oldBranchBlock = branchBlock;
ir::ControlFlowGraph::const_iterator irBlock = branchBlock->block();
ir::Instruction* branchInst = irBlock->getTerminator();
ir::PTXInstruction* branchInstPtx = static_cast<ir::PTXInstruction*>(branchInst);
ir::PTXOperand* branchPredicate = &(branchInstPtx->pg);

std::string labelPrefix = "$BBfuse_" + target1->block()->label + "_" + target2->block()->label;
int blockNum = 0;

BlockExtractor extractor(dfg, target1, target2, extractionPath, *branchPredicate);
while (extractor.hasNext())
{
        if (extractor.nextStep() == BlockMatcher::Match || extractor.nextStep() == BlockMatcher::Substitution)
        {
                std::stringstream blockLabel;
                blockLabel << labelPrefix << "_uni_" << blockNum++;

                DataflowGraph::iterator newUnifiedBlock = dfg->insert(oldFallthroughBlock, target1,
blockLabel.str());
```

```
                extractor.extractUnifiedBlock(newUnifiedBlock);

                dfg->addEdge(newUnifiedBlock, target2, ir::ControlFlowGraph::Edge::Branch);
                if (oldFallthroughBlock == oldBranchBlock)
                {
                        dfg->removeEdge(oldBranchBlock, target2);
                }
                else
                {
                        dfg->removeEdge(oldFallthroughBlock, newUnifiedBlock);
                        dfg->redirect(oldBranchBlock, target2, newUnifiedBlock);
                        dfg->addEdge(oldFallthroughBlock, newUnifiedBlock, ir::ControlFlowGraph::Edge::Branch);

                        ir::PTXInstruction gotoPtx(ir::PTXInstruction::Bra);
                        ir::PTXOperand gotoLabelOperand(blockLabel.str(), ir::PTXOperand::Label,
ir::PTXOperand::s32);
                        gotoPtx.setDestination(gotoLabelOperand);
                        gotoPtx.uni = true;
                        ir::Instruction& gotoInst = gotoPtx;
                        dfg->insert(oldFallthroughBlock, gotoInst);
                }

                oldFallthroughBlock = newUnifiedBlock;
                oldBranchBlock = newUnifiedBlock;
        }
        else
        {
                std::stringstream fallthroughLabel;
                fallthroughLabel << labelPrefix << "_ft_" << blockNum++;
                DataflowGraph::iterator newFallthoughBlock = dfg->insert(oldFallthroughBlock, target1,
fallthroughLabel.str());
                std::stringstream branchLabel;
                branchLabel << labelPrefix << "_bra_" << blockNum++;
                DataflowGraph::iterator newBranchBlock = dfg->insert(oldBranchBlock, target2, branchLabel.str());
                extractor.extractDivergentBlocks(newFallthoughBlock, newBranchBlock);

                if (oldFallthroughBlock == branchBlock)
                {
                        const ir::PTXOperand braLabelOperand(branchLabel.str(), ir::PTXOperand::Label,
ir::PTXOperand::s32);
                        branchInstPtx->setDestination(braLabelOperand);
                }
                else
                {
                        ir::PTXInstruction braPtx(ir::PTXInstruction::Bra);
                        ir::PTXOperand braLabelOperand(branchLabel.str(), ir::PTXOperand::Label,
ir::PTXOperand::s32);
                        braPtx.setDestination(braLabelOperand);
                        braPtx.setPredicate(*branchPredicate);
                        ir::Instruction& bra = braPtx;
                        dfg->insert(oldFallthroughBlock, bra);
                }

                oldFallthroughBlock = newFallthoughBlock;
                oldBranchBlock = newBranchBlock;
        }
} // end of while loop

if (branchBlock->targets().size() == 0)
{
        unsigned int branchInstPos = branchBlock->instructions().size() - 1;
        dfg->erase(branchBlock, branchInstPos);
}
if (oldFallthroughBlock == oldBranchBlock)
{
        if (!(target1->block()->has_fallthrough_edge()))
        {
                dfg->setEdgeType(oldFallthroughBlock, target1, ir::ControlFlowGraph::BasicBlock::Edge::Branch);
                dfg->setEdgeType(oldFallthroughBlock, target2,
ir::ControlFlowGraph::BasicBlock::Edge::FallThrough);
        }
}
dfg->copyOutgoingBranchEdges(target1, oldFallthroughBlock);
dfg->erase(target1);
dfg->copyOutgoingBranchEdges(target2, oldBranchBlock);
dfg->erase(target2);
replaceRegisters(dfg, extractor);
}
```

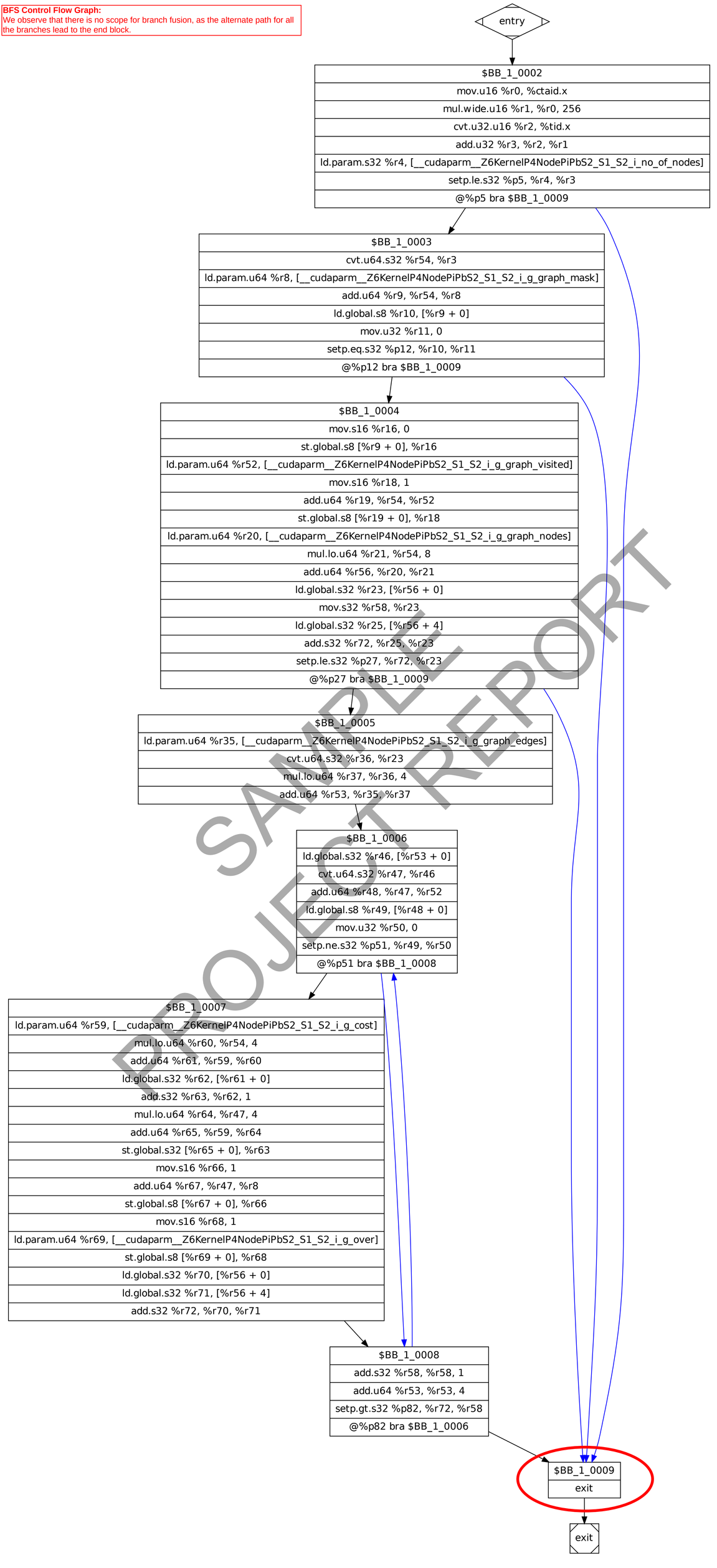
APPENDIX C

NQUEEN Control Flow Graph:
There is scope for fusion in the top half of the graph, which is correctly exploited by our pass. The bottom half cannot be fused as one of the branch paths have no instructions at all.




Optimized NQUEEN CFG:
Here we see some of the branches in the top half fused for better performance (circled). Even though the changes are really small, the total performance gain is observed to be 5%. Also, wherever possible, the bra instruction has been replaced with bra.uni when branches are proved to be non-divergent.
entry
$BB_1_0002
cvt.s32.u16 %r0, %ctaid.x
cvt.u32.u16 %r1, %ntid.x
mul.lo.u32 %r2, %r0, %r1
cvt.s32.u16 %r199, %tid.x
cvt.s64.s32 %r4, %r199
add.u32 %r5, %r199, %r2
mov.u64 %r6, __cuda___cuda_local_var_32713_39_non_const_sum44
mul.wide.s32 %r7, %r199, 4
add.u64 %r200, %r6, %r7
ld.param.s32 %r9, [__cudaparm__Z24solve_nqueen_cuda_kerneliiPjS_S_S_i_total_conditions]
setp.le.s32 %p10, %r9, %r5
@!%p10 bra $BBweave_$BB_1_0003_$BB_1_0013_bra_1
$BBweave_$BB_1_0003_$BB_1_0013_ft_0
mov.u64 %r16, __cuda___cuda_local_var_32708_39_non_const_mask428
mov.u64 %r17, __cuda___cuda_local_var_32710_39_non_const_r_mask4268
mov.u64 %r18, __cuda___cuda_local_var_32709_39_non_const_t_mask8108
mov.u64 %r19, __cuda___cuda_local_var_32711_39_non_const_m11948
mul.lo.u64 %r20, %r4, 40
cvt.s64.s32 %r21, %r5
mul.wide.s32 %r22, %r5, 4
ld.param.u64 %r23, [__cudaparm__Z24solve_nqueen_cuda_kerneliiPjS_S_S_i_total_masks]
add.u64 %r24, %r23, %r22
ld.global.u32 %r25, [%r24 + 0]
add.u64 %r26, %r20, %r16
st.shared.u32 [%r26 + 0], %r25
ld.param.u64 %r27, [__cudaparm__Z24solve_nqueen_cuda_kerneliiPjS_S_S_i_total_masks]
add.u64 %r28, %r27, %r22
ld.global.u32 %r29, [%r28 + 0]
add.u64 %r30, %r20, %r18
st.shared.u32 [%r30 + 0], %r29
ld.param.u64 %r31, [__cudaparm__Z24solve_nqueen_cuda_kerneliiPjS_S_S_i_total_r_masks]
add.u64 %r32, %r31, %r22
ld.global.u32 %r33, [%r32 + 0]
add.u64 %r34, %r20, %r17
st.shared.u32 [%r34 + 0], %r33
or.b32 %r35, %r25, %r29
or.b32 %r36, %r33, %r35
add.u64 %r37, %r20, %r19
bra.uni $BBweave_$BB_1_0003_$BB_1_0013_uni_2
$BBweave_$BB_1_0003_$BB_1_0013_bra_1
mov.u32 %r197, 0
Highlighting few major differences between Original CFG and Optimized CFG, where the optimizations have taken place
$BBweave_$BB_1_0003_$BB_1_0013_uni_2
selp.u32 %r286, %r197, %r36, %p10
selp.u64 %r287, %r200, %r37, %p10
st.shared.u32 [%r287 + 0], %r286
@!%p10 bra $BBweave_$BB_1_0003_$BB_1_0013_bra_4
$BBweave_$BB_1_0003_$BB_1_0013_ft_3
ld.param.s32 %r38, [__cudaparm__Z24solve_nqueen_cuda_kerneliiPjS_S_S_i_n]
shl.b32 %r39, 1, %r38
sub.u32 %r40, %r39, 1
mov.s64 %r41, 0
mul.lo.s64 %r42, %r4, 10
mov.s32 %r43, 0
mov.s32 %r44, 0
$BB_1_0004
add.s64 %r57, %r41, %r42
mul.lo.s64 %r58, %r57, 4
add.s64 %r59, %r58, %r19
ld.shared.u32 %r60, [%r59 + 0]
and.b32 %r61, %r40, %r60
setp.ne.u32 %p62, %r40, %r61
@!%p62 bra $BB_1_0006
$BBweave_$BB_1_0003_$BB_1_0013_bra_4
$BB_1_0006
ld.shared.u32 %r91, [%r59 + 0]
add.u32 %r92, %r91, 1
not.b32 %r93, %r91
and.b32 %r94, %r92, %r93
or.b32 %r95, %r91, %r94
st.shared.u32 [%r59 + 0], %r95
and.b32 %r96, %r40, %r94
mov.u32 %r97, 0
setp.eq.u32 %p98, %r96, %r97
@!%p98 bra $BB_1_0010
$BB_1_0005
sub.s32 %r43, %r43, 1
sub.s64 %r41, %r41, 1
bra.uni $BB_1_0011
$BB_1_0007
add.s32 %r114, %r43, 1
ld.param.s32 %r115, [__cudaparm__Z24solve_nqueen_cuda_kerneliiPjS_S_S_i_mark]
setp.ne.s32 %p116, %r115, %r114
@!%p116 bra $BB_1_0009
$BB_1_0010
sub.s32 %r43, %r43, 1
sub.s64 %r41, %r41, 1
$BB_1_0008
add.s32 %r44, %r44, 1
sub.s32 %r43, %r43, 1
sub.s64 %r41, %r41, 1
bra.uni $BB_1_0011
$BB_1_0009
add.u64 %r147, %r58, %r16
ld.shared.u32 %r148, [%r147 + 0]
or.b32 %r149, %r148, %r94
st.shared.u32 [%r147 + 4], %r149
add.u64 %r150, %r58, %r18
ld.shared.u32 %r151, [%r150 + 0]
or.b32 %r152, %r151, %r94
shl.b32 %r153, %r152, 1
st.shared.u32 [%r150 + 4], %r153
add.u64 %r154, %r58, %r17
ld.shared.u32 %r155, [%r154 + 0]
or.b32 %r156, %r155, %r94
shr.u32 %r157, %r156, 1
st.shared.u32 [%r154 + 4], %r157
or.b32 %r158, %r149, %r153
or.b32 %r159, %r157, %r158
st.shared.u32 [%r59 + 4], %r159
mov.s32 %r43, %r114
add.s64 %r41, %r41, 1
bra.uni $BB_1_0011
$BB_1_0011
mov.u32 %r188, 0
setp.ge.s32 %p189, %r43, %r188
@!%p189 bra $BB_1_0004
$BB_1_0012
st.shared.b32 [%r200 + 0], %r44
bra.uni $BB_1_0014
$BB_1_0014
bar.sync 0
mov.s32 %r201, 31
setp.le.s32 %p202, %r199, %r201
@!%p202 bra $BB_1_0016
$BB_1_0015
ld.shared.u32 %r207, [%r200 + 0]
ld.shared.u32 %r208, [%r200 + 256]
add.u32 %r209, %r207, %r208
st.shared.u32 [%r200 + 0], %r209
$BB_1_0016
bar.sync 0
@!%p202 bra $BB_1_0018
$BB_1_0017
ld.shared.u32 %r217, [%r200 + 0]
ld.shared.u32 %r218, [%r200 + 128]
add.u32 %r219, %r217, %r218
st.shared.u32 [%r200 + 0], %r219
$BB_1_0018
bar.sync 0
mov.u32 %r223, 15
setp.gt.s32 %p224, %r199, %r223
@%p224 bra $BB_1_0020
$BB_1_0019
ld.shared.u32 %r228, [%r200 + 0]
ld.shared.u32 %r229, [%r200 + 64]
add.u32 %r230, %r228, %r229
st.shared.u32 [%r200 + 0], %r230
$BB_1_0020
bar.sync 0
mov.u32 %r234, 7
setp.gt.s32 %p235, %r199, %r234
@%p235 bra $BB_1_0022
$BB_1_0021
ld.shared.u32 %r239, [%r200 + 0]
ld.shared.u32 %r240, [%r200 + 32]
add.u32 %r241, %r239, %r240
st.shared.u32 [%r200 + 0], %r241
$BB_1_0022
bar.sync 0
mov.u32 %r245, 3
setp.gt.s32 %p246, %r199, %r245
@%p246 bra $BB_1_0024
$BB_1_0023
ld.shared.u32 %r250, [%r200 + 0]
ld.shared.u32 %r251, [%r200 + 16]
add.u32 %r252, %r250, %r251
st.shared.u32 [%r200 + 0], %r252
$BB_1_0024
bar.sync 0
mov.u32 %r256, 1
setp.gt.s32 %p257, %r199, %r256
@%p257 bra $BB_1_0026
$BB_1_0025
ld.shared.u32 %r261, [%r200 + 0]
ld.shared.u32 %r262, [%r200 + 8]
add.u32 %r263, %r261, %r262
st.shared.u32 [%r200 + 0], %r263
$BB_1_0026
bar.sync 0
mov.u32 %r267, 0
setp.gt.s32 %p268, %r199, %r267
@%p268 bra $BB_1_0028
$BB_1_0027
ld.shared.u32 %r272, [%r200 + 0]
ld.shared.u32 %r273, [%r200 + 4]
add.u32 %r274, %r272, %r273
st.shared.u32 [%r200 + 0], %r274
$BB_1_0028
bar.sync 0
mov.u32 %r277, 0
setp.ne.s32 %p278, %r199, %r277
@%p278 bra $BB_1_0030
$BB_1_0029
ld.shared.u32 %r280, [__cuda___cuda_local_var_32713_39_non_const_sum44]
ld.param.u64 %r281, [__cudaparm__Z24solve_nqueen_cuda_kerneliiPjS_S_S_i_results]
cvt.s64.s32 %r282, %r0
mul.wide.s32 %r283, %r0, 4
add.u64 %r284, %r281, %r283
st.global.u32 [%r284 + 0], %r280
$BB_1_0030
exit
exit
SAMPLE
PROJECT REPORT

BFS Control Flow Graph:
We observe that there is no scope for branch fusion, as the alternate path for all the branches lead to the end block.



MUM Control Flow Graph:
Another interesting case that we studied - This benchmark had some scope for branch fusion methods (in the circled region below) but due to the low value of profitability index calculated, it was not chosen for the optimization. It would take more effort to fuse and regenerate the new control flow graph than running it as it is.