Characterizing performance loss due to mapping General Purpose programs onto GPU Architectures.

Archit Gupta, Sohum Datta
University of California, Berkeley

1 Introduction

Graphics Processing Units (GPUs) are increasingly used for scientific and other compute-intensive tasks. Owing to the availability of large number of parallel resources (shader pipelines), they can achieve orders of magnitude higher performance on certain data-parallel applications than a conventional CPU.

However, since the shader cores were not designed for general purpose computing, there is performance loss due to a significant amount of control flow. Additionally, unlike vector processors, most GPUs do not have tightly coupled scalar core(s). Hence, in order to map general or scientic problems on a GPU architecture, the scalar part of the program can be processed in either of the two ways: (1) the scalar part is executed on the CPU and the parallel part on the GPU; or (2) the scalar (serial) code is executed on the GPU as part of the execution kernel thereby under-utilizing the parallel hardware avaliable.

While the former method involves significant data movement (CPU and GPU have seperate memory sub-systems), the latter approach causes the inclusion of significant amount of control flow in existing kernels.

2 Motivation

Figure 1 shows a code snippet of the Breadth-First Search (BFS) algorithm from the PARBOIL Benchmark Suite. One can see that the actual algorithm involves only the *visit_node* function to be called for each node in the graph. However, in order to map the code to a GPU, additional control flow has been added.

If branches like these lead to signifincat performance degradation for data-parallel programs, we can develop hardware solutions for improving execution of these branches on a GPU which can, in turn, significantly improve the overall performance.

```
__global__ void
BFS_kernel( ... )
{
  if(threadIdx.x < NUM_BIN){
    q.reset(threadIdx.x, blockDim);
  }
  __syncthreads();
  tid = blockIdx.x*THREADS_PER_BLOCK
                ...  + threadIdx.x;
  if( tid < no_of_nodes)
  {
    visit_node(q1[tid], threadIdx.x
        & MOD_OP, q, overflow,
        g_color, g_cost, gray_shade);
  }
  __syncthreads();
  if(threadIdx.x == 0){
    ...
    shift = atomicAdd(tail,tot_sum);
  }
  __syncthreads();
}
```

Figure 1: *Code snippet from an implementation of Breadth first search algorithm in CUDA*

3 Problem Statement

This project studies the performance loss due to execution of scientific and general purpose data-parallel applications on GPUs, and attempts to propose architectural changes to the GPU to mitigate such losses if they are significant.

Performance loss is defined as the number of cycles added due to control flow in the parallel code as a result of mapping a non-graphical application on a Graphical Processing Unit. The project contains two distinct objectives:

- **Characterizing performance losses**: To measure the performance loss in various representative algorithms of general-purpose data-parallel workloads by simulating its execution on a modern GPU architecture.
- **Propose architectural changes**: If collective losses turn out to be significant, then various architectural solutions can be explored which can potentially reduce them. The proposals will also be evaluated on timing characteristics or power consumption.

4 Related Work

Some of the major breakthroughs in identifying and resolving control divergence in GPUs has involved dynamic warps and reconvergence stacks[1]. In order ot handle register files and lane aware warp scheduling, we have also seen multi-level scheduling schemes[2]. However, to the best of our knowledge, none of the approaches have tried to analyze or characterize the control flow based on the mapping of algorithms onto a GPU.

5 Our Approach

The program for an algorithm assumes an implementation for the device running the program (CPU, GPU, SMT etc.), which in turn determines the control flow present. However, some control flow is inherent to the algorithm and cannot be eliminated by any practical hardware implementation. The branches associated with them will be called **intrinsic** branches.

Some branches are the result of mapping an algorithm onto a specific hardware . These branches will be called **extrinsic** branches. For instance, in a graphical processing routine increasing brightness of a picture, loops must be included in CPU code which cycles through all the pixels. The branches due to such loops are extrinsic, present only because of mapping onto CPU. GPU code of general purpose programs may also have extrinsic branches, as described in Section 2

Our approachs is organized as **milestones** towards achieveing the objective of the project:

- **Classifying Branches for each benchmark algorithm**. The CPU code of the algorithm is first used to identify intrinsic branches. It is relatively simple to identify intrinsic branches in CPU code as loops can be identified easily and are the most common extrinsic branches present.
  It is assumed that the intrinsic branches map similarly to the GPU code. That is, the intrinsic control portion of the code maps to similar control structures to both CPU and GPU. Using this assumption, the extrinsic branches of the algorithms are also identified by enumerating all other branches in the GPU code.

- **Measuring Performance loss due to extrinsic branches in GPU code for each algorithm**. The GPU code is simulated in a simulator and the additional cycles due to the identified extrinsic branches is counted for each algorithm. These cycle losses are then compared with the total execution time. This will tell us if the losses due to extrinsic branches are significant and can be an important metric to judge suitability of current GPU architectures to run general programs.

- **Proposing Hardware changes in GPU architectures to minimize the losses due to extrinsic branches**. If the losses found in the previous milestone are significant (for instance, more than 5%) then any light hardware change that can partially mitigate such losses will be beneficial. We will explore different changes to the shader pipeline, taking hints from CPU and vector pipelines and the knowledge gained in class, to minimize the losses. Since this depends on the outcome of the second milestone, we cannot say much about our approach in this milestone apriori.

6 Infrastructure Used

We chose the PARBOIL Benchmark Suite [3], which has a selection of algorithms widely used in throughput oriented programs in different scientific and commercial applications. The suite comes with a base code as well as optimized CUDA code for GPUs, which is ideal for our purpose.

For emulating GPUs we chose GPGPU-Sim[4]. This simulator now includes an integrated energy model GPUWattch, which has been validated by measuring power characteristics of two contemporary GPUs. We plan to use GPUWattch to predict power consumption due to our proposed architectural changes.

For simulating the execution of a modern CPU we chose SESC[5] (SuperESCalar Simulator), a widely used MIPS based microprocessor simulator developed by i-acoma group at UIUC.

We believe the above-stated infrastructure is sufficient for a basic study of the problem at hand. However, there is ample scope for expansion. We could evaluate programs supplied by the Rodinia Benchmark Suite[6] to diversify the representation of programs. Another possible direction might be to evaluate performance loss due to extrinsic branches in vector processors, and compare them to those in GPUs. Since both have similar philosophy in utilizing data-level parallelism, it will be interesting to see the differences in respective performance losses and the reasons behind them.

References

[1] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "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.

[2] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 308–317, ACM, 2011.

[3] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[4] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "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.

[5] P. M. Ortego and P. Sack, "Sesc: Superescalar simulator," in *17 th Euro micro conference on real time systems (ECRTS201905)*, pp. 1–4, 2004.

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