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

Archit Gupta, Sohum Datta
University of California, Berkeley

Branch divergence in GPGPUs has attracted a lot of interest in the computer architecture research community. Various solutions have been suggested to mitigate divergence in SIMD architectures. However, a lot of these branches are artifacts of mapping of an algorithm to a GPU and are not intrinsic to the algorithm. In this project, we will analyze the branches that a GPU encounters and classify them according to their nature in the underlying algorithm. Most solutions rely on some assumptions regarding the nature of branches encountered by the GPU, which might not be entirely true if the extrinsic branches account for most of the performance degradation. If the performance loss due to extrinsicn branches is significant, we will explore different architectural solutions in GPUs to reduce the same.

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.

```
__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*

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.

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 in some order. However, in order to map the code to a GPU, additional code has been added. This adds a significant amount of control flow to the program.

More importantly, the behavior of branches in this part of the program is significantly different from the branches that are intrinsic to BFS. If branches like these lead to signifincat performance degradation for common data-parallel general purpose programs, we can develop hardware solutions specificalyy improving execution of these branches on a GPU which can, in turn, significantly improve the overall performance.

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 refers to 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. Another way to interpret it is that the performance loss equals the number of cycles taken to execute a parallel code of the application on the GPU less the number of cycles required to execute an equivalent graphical application of similar length on the GPU. Parallel code refers to a native assembly language code optimized to execute the general purpose application on a GPU.

As such, the project contains two distinct objectives:

- **Characterize 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. It is important that the workload represented have enough Data-level Parallelism (DLP) to get significant speedup (relative to CPU) when executing on parallel hardware, and be general enough for execution on other hardware implementations such as CPUs or Vector machines.

- **Propose architectural changes**: If the losses from the collective measurements turn out to be significant, then various architectural solutions will be explored which can (partially) reduce such losses. The proposal will be studied and the resulting reduction of losses will be measured along with any changes in general timing characteristics or power consumption in the machine.

Related Work

Our Approach

Any code for an algorithm assumes a hardware implementation 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. Intrinsic control flow may change in implementation according to the underlying hardware (¡TODO: VERFIY¿ eg. switch-case statements for a CPU may change to thread-id-cases in GPU implementations). However, they cannot be eliminated completely and can at best be optimized for the native hardware.

Some branches are the result of mapping an algorithm onto a hardware which is not optimized for its execution. These branches will be called **extrinsic** branches. For instance, in a graphical processing routine increasing brightness of a picture by a constant value, loops must be included in CPU code which cycles through all the pixels in the picture and increases its brightness. The branches due to such loops are extrinsic branches, present only because of mapping on a CPU. GPU code of general purpose programs may also have extrinsic branches, as ¡TODO: VERIFY¿ described in the Motivation section.

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

- **Classifying Branches for each benchmark algorithm**. The first step will be to enumerate and classify branches in each GPU program into intrinsic and extrinsic branches. To do so, we first use a CPU code of the algorithm to identify intrinsic and extrinsic branches. It is relatively simple to identify extrinsic branches in CPU code as loops can be identified easily and are the most common(¡TODO: only extrinsic?¿) extrinsic branches present. Hence, the intrinsic branches of the CPU code are also calculated.
  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 and

therefore have same number of intrinsic branches. This assumption is reasonable as CUDA (the language used in benchmarks for GPU code) is very similar to C++ and most compiler optimizations that can be done in the non-parallelized code is the same (¡TODO: is this reasoning correct?). Nevertheless, this assumption will be verified by examining the GPU and CPU code for the simpler algorithms (such as, ¡TODO: include examples here¿) in the benchmark. 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.
  Moreover, the loss thus encountered in the GPU code will be compared with equivalent loss in the CPU code. The inter CPU – GPU comparison will indicate whether the fractional cycle loss due to extrinsic branches (reduced by branch prediction) is better handled in CPU and may give us insights into improving GPU hardware. (¡TODO: Is this even useful, as the number of cycles in a CPU will be very much larger due to looping, and moreover these are entirely differnt hardware paradigms.)

- **Proposing Hardware changes in GPU architectures to minimize the losses due to extrinsic branches**. If the losses found in the previous milestone is 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.

Infrastructure Used

In this project, the primary infrastructure is the benchmark suite that represents the general body of algorithms commonly used in both data parallel architectures like GPUs and vector processors, as well as in modern Our-of-order processors. We chose the PARBOIL Benchmark Suite **??** developed by the IMPACT research group at UIUC. It 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 of comparing performance loss due to extrinsic branches for each representative algorithm when mapped to GPUs versus to CPUs.

We also need simulators for emulating the a modern Out-of-Order superscalar CPU and a General Purpose GPU. For emulating GPUs we chose GPGPU-Sim**??**, an open-source GPU simulator developed at UBC by Tom Aamodt. This simulator is very flexible in the types of GPU architectures it can simulate, and is quite accurate in predicting performance of benchmark suites. It now includes an integrated energy model GPUWattch, which has been validated by measuring power characteristics of two contemporary GPUs. The error in energy predictions is close to 10%, which is acceptable for the first-order approximations. 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 (SuperESCalar Simulator), a widely used MIPS based microprocessor simulator developed by i-acoma group at UIUC. It models a fully out-of-order complete with branch prediction, caches, and buses.

We believe the above-stated infrastructure is sufficient for a basic study of the problem at hand. However, there is ample scope for expansion. For example, we could evaluate programs supplied by the Rodinia Benchmark Suite**??** to diversify the representation of compute-intensive algorithms used in GPUs today. Using Rodinia, however, may require additional effort as it does not provide base implemenations optimized for CPUs. 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

[Figueredo and Wolf, 2009] Figueredo, A. J. and Wolf, P. S. A. (2009). Assortative pairing and life history strategy - a cross-cultural study. *Human Nature*, 20:317–330.