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*—The abstract goes here.**

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 certian 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.

```
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

The focus of the research community in computer architecture has been directed towards making changes in the GPU architecture which let it acomodate 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.

```
__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

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

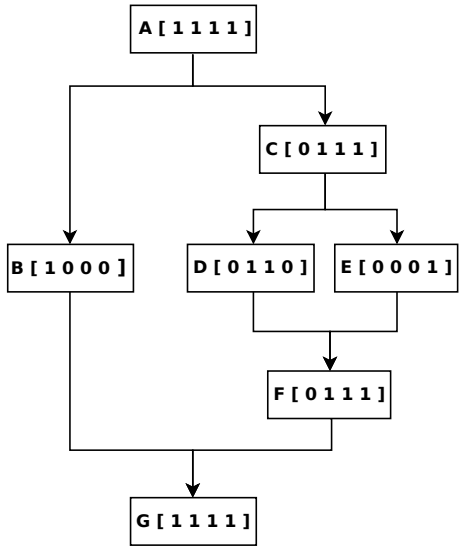

Fig. 3: A sample control flow graph to understand branch divergence. Each block in the graph depicts the name of a basic block followed by a 4 bit vector show the status of different threads while executing that basic block. The i^{th} bit represents the state of the i^{th} thread.

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 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.

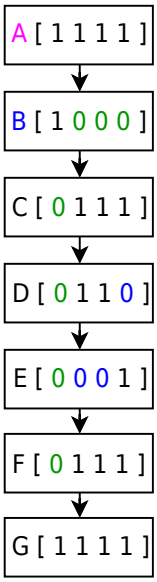

Fig. 4: The execution of the 4 threads as shown previously progresses as shown here on a GPU

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 beng 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 wil 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 copmutational 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, 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 accomodate 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

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

V. Related Work

VI. Our Approach

A. Tagging CUDA Benchmarks

Framework:

Assumptions about the compiler:

B. Modifications to GPGPU-SIM

VII. Results

A. Static/Dynamic branch counts

B. Performance impact

C. Branch characteristics

VIII. Conclusion

Acknowledgment

The authors would like to thank...

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.