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

```
__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

in a lot of scientific programs like solving linear systems of equations, clustering etc.

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

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.