CUDA On Hadoop

Describe CUDA On Hadoop here.

Hadoop + CUDA

Here, I will share some experiences about CUDA performance study on Hadoop MapReduce clusters.

Methodology

From the parallel programming point of view, CUDA can hlep us to parallelize program in the second level if we regard the MapReduce framework as the first level parallelization Figure 1. In our study, we provide Hadoop+CUDA solution for programming languages: Java and C/C++. The scheduling of GPU threads among grids and blocks is not concerned in our study.

For Java programmers

If your MapReduce program is written in Java, you may need JNI to make use of CUDA. However, JCuda provides an easy solution for us. We introduce CUDA to our Map stage. The CUDA code is called by map() method within Map class. It is easy to extend to Reduce stage if necessary. There are two ways to compile your CUDA code.

One is to write CUDA code as a String variable in your Java code. JCuda will automatically compile it for you. The compiled binary file is located in tasktrackers working directory that you can configure in mapred-site.xml file.

The other is little bit tricky. you can manually compile the CUDA code into binary files in advance and move them to tasktrackers working directory. And then every tasktracker can access those compiled binary files.

For C/C++ programmers

We employ CUDA SDK programs in our experiments. For CUDA SDK programs, we first digested the code and partitioned the program into portions for data generation, bootstrapping, and CUDA kernels, with the former two components transformed respectively into a standalone data generator and a virtual method callable from the map method in our MapRed utility class. The CUDA kernel is kept as-is since we want to perform the same computation on the GPU only in a distributed fashion. The data generator is augmented with the feature for taking command-line arguments such that we can specify input sizes and output location for different experiment runs. We reuse the code for boot-strapping a kernel execution into part of the mapper workload, thus providing a seamless integration of CUDA and Hadoop. The architecture of the ported CUDA SDK programs onto Hadoop is shown in Figure 2. For reusability, we have used object-oriented design by abstracting the mapper and reducer functions into a base class, i.e., MapRed. For different computing, we can override the following virtual methods defined by MapRed:

Figure 2

```
void processHadoopData(string& input);
void cudaCompute(std::map<string,string>& output);
```

The processHadoopData method provides a hook for the CUDA program to initialize its internal data structures by parsing the input passed from the HDFS. Thereafter, MapRed invokes the cudaCompute method, in which the CUDA kernel is launched. The results of the computation are stored in the map object and sent over to HDFS for reduction.

With all the parts ready, we developed a set of scripts for launching the experiment. Specifically, we perform the following steps in the scripts for all programs:

- Set up environment variables;
- Generate input data;
- Remove old data and upload new data to HDFS;
- Upload program binary onto HDFS (if changed);
- Remove the output directory from the previous run;
- Submit the program as a MapReduce Job and start timer;
- Report the runtime measurements and timestamps.

Observing that the launching logic of all programs are very similar (only the arguments to the input generators and program names differ), we have developed a common driver script which is parameterized by each individual launch script and performs all the processing accordingly. Thus, this modularization enables us to write launch scripts for new programs easily with a few lines of code only.