Accelerating a Cloud-Based Software GNSS Receiver

Kamran Karimi, University of Calgary, Calgary, Canada
Aleks G. Pamir, Rx Networks Inc., Vancouver, Canada
M. Haris Afzal, Rx Networks Inc., Vancouver, Canada

ABSTRACT

This paper discusses ways to reduce the execution time of a software Global Navigation Satellite System (GNSS) receiver that is meant for offline operation in a cloud environment. Client devices register satellite signals they receive, and send them to the cloud, to be processed by this software. The goal of this project is for each client request to be processed as fast as possible, but also to increase total system throughput by making sure as many requests as possible are processed within a unit of time. The characteristics of the application provided both opportunities and challenges for increasing performance. This paper describes the speedups we obtained by enabling the software to exploit multi-core CPUs and GPGPUs. It mentions which techniques worked and which did not. To increase throughput, it describes how to control the resources allocated for each invocation of the software to process a client request, such that multiple copies of the application can run at the same time. It uses the notion of the effective running time to measure the system’s throughput when running multiple instances at the same time, and show how to determine when the system’s computing resources have been saturated.

Keywords: Cloud, GPU, High Performance Computing, Multi-Core, Software GNSS Receiver

INTRODUCTION

Implementing a software Global Navigation Satellite System (GNSS) receiver completely in software has received attention due to the flexibility it provides the designers and developers (Charkhandeh, Petovello, Watson, & Lachapelle, 2006). Adding features, configuration changes, defect fixing, and re-deployment are usually done easier with a software receiver than a hardware one. The downside is that a software receiver usually processes signals slower than a hardware receiver because it tends to emulate dedicated hardware, which may not be the most efficient way of designing the GNSS in software. This makes performance an important aspect of the design and implementation of a software receiver.

Many operations in a GNSS receiver, including but not limited to signal acquisition and tracking, are inherently independent of each other and are run in parallel when a standard
receiver is implemented in hardware (Petovello, O’Driscoll, Lachapelle, Borio, & Murtaza, 2008). A software receiver can exploit this same parallel execution possibility and benefit from multi-core CPUs and GPGPUs. For this reason this paper concentrates on parallelizing the execution using CPUs and GPUs. These two processor classes have very different characteristics, which greatly affect the approaches and the results.

Another possible requirement for a software receiver is the ability to process data in real-time. This requirement is evidently related to the performance aspect, as real-time operation implies processing data at least as fast as they are received. (Bartunkova & Eissfeller, 2012; Haak, Büsing, & Hecker, 2012) are examples of efforts to utilize modern parallel processing hardware to implement real time GNSS software. In this paper we focus on a software receiver which is meant for offline operation, in a cloud environment. However, we need to process many requests which must still be processed in an acceptable amount of time, as defined by user agreements. For this reason, achieving high performance is one of the main requirements of this project. Offline processing provides opportunities that we have exploited for performance increase, as explained later.

The target application is intended to be run in a cloud environment, where data, recorded on many clients, are received and processed. The results are then returned to the client to either directly provide the position estimates, or assist with satellite acquisition. While real-time processing is not a strict requirement, reducing response time and increasing total throughput are very important. Not only each client must wait as little as possible to receive a response (low response time), but the system as a whole must make sure that as many requests as possible are processed per unit of time (high total throughput). In order to achieve these goals, the application must be able to fully utilize the available hardware.

Since many instances of the application may be running at the same time, care should be taken to make sure all computational resources are used effectively and without conflicts. For example, starting many instances of the application, with each of them running on all cores on a CPU with low first or second level cache may cause cache conflicts, where each thread would invalidate cache data from other threads. Such conflicts would reduce the total performance of the system.

Using a public cloud environment to run the application adds to the complexity, because some aspects of the run time environment are beyond our control. Cloud server instances running this application may be with or without GPUs. Servers are started as client requests increase. Each server may run many instances of the application to process the requests. The application is passed a number of arguments that determine which resources are to be employed by it to process a request. These arguments allow us to dynamically control resource utilization.

When discussing GPUs, we focus on NVidia products because they were available to us for development, testing, and deployment. We chose the CUDA programming toolset because it appears to provide better performance than other GPU programming toolsets (Karimi, Dickson, & Hamze, 2010). We employed CUDA 5.0, the latest version available at the time (CUDA Toolkit, 2013). The paper’s descriptions and results may or may not apply to other GPU products or programming languages.

CPU AND GPU BACKGROUND

Methods of increasing software performance on a CPU include algorithmic optimizations, tuning the code manually or automatically by the compiler to minimize wasted cycles, using vectorization to speed up mathematical processing on each core, and using multiple cores to run portions of the code in parallel. Parallelization has a big potential for performance improvement because the focus of CPU manufacturers has switched to increasing the number of cores instead of adding complexity and speed to each core. Effective utilization of the available CPU cores depends to a large degree on the design

Copyright © 2014, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
A Graphical Workflow Modeler for Docking Process in Drug Discovery
www.igi-global.com/chapter/graphical-workflow-modeler-docking-process/64545?camid=4v1a

Web Services Gateway: Taking Advantage of the Cloud
www.igi-global.com/article/web-services-gateway/149918?camid=4v1a