Program Analysis Tools for Application Specific Architectures

Maya B. Gokhale, Matthew J. Sottile
Los Alamos National Labs
Los Alamos, NM 87545
Email: {maya,matt}@lanl.gov

I. Introduction

Application specific acceleration using Field Programmable Gate Arrays (FPGAs) and Floating Point Arrays (FPAs) offers the opportunity to significantly augment the computing power of microproprocessors. In these architectures, acceleration boards contain FPGAs or FPAs along with dedicated memory banks of SRAM or DRAM. The board communicates with a host microprocessor through a high speed I/O bus or hypertransport.

Application-specific accelerators have been shown to deliver speedups of 10 100 $\times$ on some kernel functions such as matrix multply or FFT. However, finding suitable computational kernels in large legacy applications can be challenging. Such kernels must have sufficient computational density to offset the cost of communicating input data and results between acceleration engine and host, yet must still be of an appropriate size to fit on the FPGA/FPA resources available. The data must fit on the accelerator board, especially if it is reused during the computation, or else it must be partitioned and sent to the board in chunks. The kernels must be coded in a different language (or language dialect) from the rest of the application. Code must be inserted into the overall application to load the kernels, synchronize processing, and communicate data and results.

In this work, we discuss tools we have recently developed to help understand partitioning choices between microprocessor and acceleration engine in large, legacy applications. Our tools OpMix and MemFoot extend the widely used program performance analysis tools Tau [1] and Valgrind [2].

II. Operation Mix of Compute Kernels

Identifying regions of a kernel for acceleration requires understanding of the instruction stream. Due to limitations of modern accelerator hardware, it is not possible to choose an arbitrary stream of instructions to be implemented on an accelerator while still achieving some speedup over a traditional microprocessor. For example, the ability to execute general double precision floating point arithmetic is limited on many implementations. Similarly, we may find that a region of code where a large amount of time is spent performs a very irregular instruction stream, possibly with a high branch count, making accelerator implementation difficult. The OpMix tool will help automate the process of identifying regions of code where both a large amount of time is spent and the instruction mix is appropriate for acceleration.

The tool is based on a simple process of identifying candidate regions of code based on profile data, extraction of those regions of code, and analysis of the instruction stream that they require. Profiling is performed using the TAU performance analysis toolkit [1]. Profiles can be taken based on time or hardware counters. We can use the hardware counter data, in addition to simple time measurements, to characterize the behavior of regions of code with respect to their memory locality (through cache miss statistics) and instruction stream regularity (branching behavior). Code in C, C++, or Fortran may be automatically instrumented for profiling at both the function and loop level. Custom definition of regions for profiling can also be added manually to organize profiled regions in a manner that better reflects semantic relationships within the source code.

We are able to use profile data to identify the source regions where most of the time (or other measured quantity) was spent. Given these regions, we would like to then decide how appropriate they are for acceleration. To perform this analysis, we use the GNU libbfd [3] library and code based on the GNU binutils package to map source code locations to instructions in the compiled binary. This allows us to determine the instruction mix for the profiled regions of interest in a static sense. This information, combined with dynamically measured data within profiles of both time and executed instruction counts (such as branch or floating point operations), allows us to form a description of the program both in terms of what regions of code consume the most time, and which of these regions execute instruction streams most appropriate for an accelerator.

This instruction mix information can be combined with the analysis of the memory footprint to yield a breakdown of the code into regions that are conducive to acceleration from both a memory and computational requirement perspective.

III. Memory Footprint Analysis

Once a computational kernel has been identified and its operations have been evaluated, it is necessary to understand the data requirements of the kernel. Data used by the kernel must be transferred to the acceleration unit as the kernel executes, and results returned back to the microprocessor main memory. The data may be supplied in full, in discrete buffers,

or may be streamed to the kernel. In order to decide which of these options would be most appropriate for a specific kernel, a first step is to understand the amount of data being processed within the kernel. The kernel's memory footprint can be used to guide selection of the communication options. For example, if the amount of data processed by the kernel is larger than the amount of on-board memory, then buffering or streaming is necessary.

A kernel's memory footprint is often difficult (or impossible) to ascertain from static analysis of the source code, either manually or algorithmically. Often, arrays are dynamically allocated, and their extents depend on parameters supplied at run time in data files, and/or complex control logic throughout the application. Running different data sets may result in very different memory footprints. For this reason, we have built a tool that reports the number of load/store operations on a subroutine basis.

The MemFoot tool is built on the Valgrind framework. Valgrind has a number of tools that help detect memory management and threading bugs. There is a memory error detector, a cache profiler, a heap profiler. While Valgrind can report on load and store operations, it reports on total number of loads or stores across the entire program, broken down into a per-function basis. For our application of kernel code profiling, it is desirable to profile loads and stores for specific small regions of code. It is also significantly faster to insert instrumentation only on the small kernel being studied. Further, valgrind gives only the total number of load and store operations. We would like to know the number of *unique* addresses accessed, and the number of times each was accessed. This information helps to determine how much data must be transferred to and from the accelerator board. It can also help identify specific variables that should be kept on-chip in registers.

As an example of its usage, the MemFoot tool was applied to a Monte Carlo Radiative Heat Transfer Simulation [4]. A subroutine "taskcode" contains the main computation. While there were 143.5M load or store operations in taskcode, there were only 3887 unique 32-bit words. Further, a small subset of those words were accessed thousands of times. Knowing these characteristics, we can determine that for a Xilinx FPGA implementation, the data can be store in Block RAM. On the Clearspeed, the data can be spread across the per-PE 6KB local memory.

IV. Summary

Application specific heterogenous architectures are expected to become increasingly important in boosting the performance of general purpose microprocessors. To effectively exploit such architectures on legacy applications, tools are needed that help identify dense computation kernels and to understand the instruction mix and memory requirements of the kernels. In the workshop we will describe a tool set being developed to help reveal these computational characteristics of compute intensive codes and its performance on a set of scientific applications.

References

[1] S. Shende and A. D. Malony, "The TAU Parallel Performance System," *International Journal of High Performance Computing Applications*, submitted.

[2] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *Electronic Notes in Theoretical Computer Science*, 2003.

[3] "libbfd: the binary file descriptor library," *Documentation for the GNU binary utilities*, April 1992.

[4] P. J. Burns and D. V. Pryor, "Vector and parallel monte carlo radiative heat transfer," *Numerical Heat Transfer, Part B: Fundamentals*, 1989.