Pytheus: An Easily Programmable Heterogeneous Computing System

Stefan B. O'Neil
North Carolina State University
soneil@ncsu.edu

Abstract

Solid-state drives have entered the mainstream of commercial storage devices. These drives come equipped with their own processors and memory to provide the necessary support for the flash translation layer. Numerous solutions exist for enabling the host machine to leverage these in-storage computational resources, making possible considerable performance gains. However, these existing approaches rely heavily on device- code running on the SSD, making them inflexible for adaptation to new applications, and inaccessible to application developers writing in modern, high-level programming languages. This paper proposes to redress this weakness by moving the task of code generation for the SSD to the host compiler. We target Pyston, a just-in-time Python compiler, and use built-in Python functionality to share relevant data structures and machine code directly with the SSD. [rest of abstract will deal with results]

1. Introduction

In keeping with the Von Neumann model of computer architecture, computer systems have traditionally separated computation, performed by a dedicated processor, from data storage, typically accomplished by a hard disk drive (HDD), or more recently, a solid state drive (SSD). Over the past few decades, processor performance has improved dramatically. Storage access latency has not enjoyed similar improvement.

The result of this mismatch is that storage access has become the most significant limiting factor for computing performance. This is particularly true for data intensive applications, which have become ever more prevalent with the advent of big data.

In the last few years, SSDs have entered the commercial mainstream of storage devices. SSDs offer significant performance improvement in access latency over HDDs. However, as flash-based devices, SSDs also differ from HDDs in a number of use-relevant ways. Unlike HDDs, SSDs have different granularity for reads and writes, cannot write to dirty pages before clearing them, and have lifetime limits on number of writes per block before a given block becomes unusable. These differences necessitate different access patterns from those appropriate for HDDs. To work seamlessly with the HDD-optimized storage interfaces common among host machines, SSDs must therefore perform additional logic known as the flash translation layer (FTL) to process traditional read-write requests from the host. To accomplish this, SSDs come equipped with their own internal processors and DRAM.

The additional computational resources in peripheral devices such as SSDs create an opportunity to bypass the bottleneck of data transfer by moving away from the classical von Neumann separation of computation and data storage. By moving some of the computational tasks in a program to near-data processors such as those found in SSDs, a more distributed model of computing can reduce the traffic between the central processing unit and peripheral devices.

The project undertaken in this paper is to adapt a proven programming model for in-storage processing called Morpheus for compatibility and ease of use with a high level programming language. We target the Python programming language, primarily for its simplicity, ease of use, and high popularity among application developers.

Our programming model consists of the application runtime environment, which shares relevant data structures and compiled SSD-compatible machine code with a kernel-space memory buffer, which the host then shares with the SSD. After the SSD executes this machine code and stores the results back into its own memory buffer, these are then copied directly back into the host-side memory buffer via Direct Memory Access.

The remainder of this paper is broken down into the following sections:

2. Heterogeneous Computing Models

The proliferation of auxiliary processing units in modern computing systems has spurred a huge amount of research interest into the potential performance improvements such systems provide. These auxiliary processing units include a wide array of different technologies. The most common of these are graphical processing units (GPUs), field programmable gate arrays (FPGAs), and in-storage processors. Within each of these types of technologies, there is a great deal of variability in the specific properties a given device may have. Different GPUs, for example, employ a wide variety of different memory hierarchies. Furthermore, there are many other types of auxiliary processing units that a computing system may employ, apart from the common three listed above. Prominent examples include the tensor processing unit developed by Google, vision processing units, and physics processing units.

One intuitive way to exploit the performance advantages of heterogeneous computing systems is to apply these diverse computational resources to application code. Many auxiliary processing units are designed specifically to handle types of computations that are difficult for traditional CPUs to perform efficiently, such as graphics rendering or the training of neural networks. FPGAs can be configured to perform operations common to a particular application much faster than a traditional CPU is capable of.

In addition to the speed advantage of specialized processors, a heterogeneous computing model can also improve application performance by moving the most data-intensive computations of an application closer to the data storage. In this way, auxiliary processing resources can act as a filter for data passing from the storage device to the main processor, reducing traffic across the main bus.

Database applications, with their high volume of data access, are a prime candidate for acceleration by means of heterogeneous computing systems. Several projects have proposed using FPGAs to implement database operations (list of citations). In 2014, Woods et al. demonstrated that FPGAs could be used to implement not only basic tasks such as projection and selection-based filtering, but also more complex SQL operators such GROUP-BY and WHERE. Since then, various others have extended this approach with systems that move an even greater portion of database computations from the host processor down to FPGA pipelines. Ziener et al. demonstrate how far this approach can be taken in their paper, FPGA-Based Dynamically Reconfigurable SQL Query Processing, in which they describe and prototype a system in which a pipeline of FPGAs can be reconfigured to completely handle any query from a library of modules covering all major SQL operations.

One complicating factor for sharing information between different components of a heterogeneous system is that of mapping between addresses in separate address spaces. Unifying the address space of heterogeneous components with that of the host memory could allow for more seamless data transfer. The project FlashTier uses this technique as part of an architecture for using a flash-based storage device as a cache for a higher capacity storage drive. For the sake of facilitating this design strategy, Olson et al. propose an interface for coherence messages between the host and auxiliary hardware accelerators called Crossing Guard.

2.1 In-storage Processing

Of particular relevance for our research is the use of processors built into storage devices as components of a heterogeneous system. All modern SSDs contain contain a built-in processor and accompanying DRAM memory cache. These are necessary for SSDs to transparently map logical block addresses from the host to physical locations within the flash memory, and to perform flash management algorithms such as wear leveling and garbage collection. These tasks are collectively known as the Flash Translation Layer (FTL).

If made available to the host, SSD processor time and memory space not utilized for FTL can be applied to other tasks. The opportunity is especially great because of two advantages that these in-storage processors enjoy:

1. lower power consumption than typical host processors
2. lower latency and higher bandwidth accessing storage, without adding traffic to the main bus

One very popular area of application for in-storage computing is big data analytics. Big data applications are heavily bottlenecked by storage bandwidth. Because in-storage processors can access this data before it reaches the main bus however, using in-storage processing to perform initial filtering of the data has the potential to greatly reduce traffic between the host and the storage device. Choi and Kee demonstrate the potential benefits of this storage-level data filtering using model analysis in "Energy Efficient Scale-In Clusters with In-Storage Processing for Big-Data Analytics". Aside from performance improvements, they also demonstrate the potential for reducing energy consumption. Their findings indicate that energy consumption is dominated by the cost of transferring data between storage and host, meaning that lowering this traffic is likely to deliver significant energy cost reduction in addition to performance improvement. Do et al. also examine the potential of in-storage processing to improve performance of database applications in their paper, Query Processing on Smart SSDs: Opportunities and Challenges. To aid in this analysis, they implemented some basic database operations on an SSD, and modified an SQL database management system to offload those operations to the SSD. Kang et al. take a similar approach in Enabling Cost-effective Data Processing with Smart SSD, targeting the Hadoop MapReduce framework for optimization instead.

Jun et al. implement a system called BlueDBM which introduces a number of optimizations for a distributed, big-data-oriented system built on SSDs. These optimizations include both in-storage processing, implemented with an FPGA at the storage device level, and a global address space for the distributed system. They are able to demonstrate meaningful performance gains by use of their in-storage processing engine.

The Willow project focuses more exclusively on leveraging in-storage processing for application code. Willow consists of a system driver, a SSD with a custom interface, and a template for user programs called SSD-Apps, which handles many of the complexities of writing in-storage code for the user. They demonstrate the performance benefits of this approach with six of SSD-Apps of their own, which show marked performance improvements over an implementation not utilizing in-storage processing power.

The Biscuit project also exposes in-storage computational resources to the programmer, but does so in a more user-friendly manner. Specifically, Gu et al. develop a framework based on the flow-based programming model that supports C++ programming and another of other important features, such as multithreading and dynamic loading and unloading of tasks onto the SSD.

The work that most directly inspired the current project is described in a paper titled, Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing. Dr. Tseng et al present a model of computation called Morpheus, in which the host machine compiles and transmits code to the SSD, which then processes data and transmits the results back to a corresponding program running on the host machine, generated by the same compiler. By targeting the data-intensive task of deserialization specifically, of ASCII stored values into C-compatible integer arrays their Morpheus-SSD implementation reduced context switching by an average of 98% in the host machine, energy consumption by upwards of 42%, and execution time by varying but significant margins on a wide range of applications.

The primary drawback of the Morpheus project is its reliance on the programmer to manually write code intended for execution on the SSD. Code to be executed on the SSD must be written in C, and is exceptionally challenging to debug. These factors make Morpheus less flexible and programmer-friendly than is practical for adoption by mainstream application developers.

2.2 Host-SSD Interface and Communication

The FTL is a layer of relatively complex logic and computation that exists, in essence, to allow SSDs to work with host requests optimized for hard disk drives. While this may have eased their introduction into a world dominated by HDDs, it seems reasonable to suppose that an interface designed with SSDs in mind would be likely to yield better results. This idea has motivated a number of researchers to develop and evaluate communication interfaces specifically designed to work with SSDs. In addition to having the potential to improve SSD performance, such an interface would also ease the computational burden imposed by the FTL, making in-storage resources even more available for other purposes.

In Application-Managed Flash, Lee et al. implement a system called Application Managed Flash (AMF) which, as the name implies, moves the burden of flash management up to the host processor. They achieve this by means of a modified log-structured file

system, a new I/O interface better suited to SSDs (for example, by disallowing overwrites), and a revised, greatly reduced FTL, which can run on either a device driver or directly on the SSD. Another project, called LightNVM, also defines a new I/O interface, and introduces a new Linux subsystem for managing open channel SSDs.

Key-value stores offer an alternative system for managing flash storage directly from the host. Both DIDACache and KAML manage impressive experimental results by implementing this key-value approach. The main distinguishing feature between these two approaches lies in KAMLs multi-log system, which provides stronger support for parallel access to different channels within the SSD. KAML also supports fine-grained locking and atomic updates of multiple key-value pairs at a time. DIDACache, on the other hand, uses a unified global namespace, and takes advantage of this with a host-SSD integrated garbage collector which works across the entire SSD.

3. Just-in-Time Compilation

Just-in-time (JIT) compilation is a method of code execution in which portions of program code are compiled or recompiled during runtime. JIT compilation is highly compatible with dynamic languages such as Ruby and Python, but also offers strong performance advantages over pure interpretation. Because of the popularity of high level dynamic programming languages, JIT compilers for such languages provide a useful environment for making cutting edge advances in systems technologies more accessible to general application developers.

3.1 JIT Code Generation for Heterogeneous Systems

Leveraging the performance advantages of heterogeneous systems requires writing code for auxiliary processing units. Coding for these heterogeneous computing resources often requires detailed hardware knowledge, and the use of specific low-level programming languages such as OpenCL. This is a significant barrier for application developers trying to leverage the advantages of heterogeneous computing systems in their own programs.

Using JIT compilers to generate the appropriate code for the target heterogeneous processor instead can remove this burden from the programmer. The compatibility of JIT compilers with dynamic programming languages means that this approach can bring direct, native control of heterogeneous computing resources to such languages.

In Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation, Fumero et al. modify a compiler and interpreter for the programming language R to automatically detect parallel operations, and JIT compile these operations into OpenCL code ready to run on a GPU.

Trainiti et al. build a runtime system called the Orchestrator which manages workloads across a heterogeneous architecture called SAVEHSA which relies on JIT compilation to re-compile relevant code snippets for the hardware accelerator Orchestrator chooses to assign it to.

Rubinsteyn et al. approach this problem for Python by implementing a library called Parakeet. This library is designed to speed up the widely-used Python scientific computing library Numpy. It does this by JIT compiling user-marked functions, and dividing their execution across the host and a GPU accelerator. Specifically, it looks for operations on Numpy arrays, which are highly parallelizable.

4. Detailed System Overview

5. Generation of Machine Code from Pyston

6. Results

7. Related Work

8. Conclusion

References

[1] P. Q. Smith, and X. Y. Jones. ...reference text...