Cross-Layer Compute and Memory Abstractions for Enhanced Programmability, Portability, and Performance

Nandita Vijaykumar
Carnegie Mellon University

1. Introduction

Over the last decade, there has been an increasing tread towards specialization at different levels of the computing stack. This specialization has driven large improvements in system performance and includes the development of domain specific languages, compilers, and frameworks (e.g., Halide, Pochoir, TensorFlow, CNTK); and hardware specialization in the form of accelerators (e.g. GPUs, FPGAs), specialized memories (e.g., 3D XPoint, HBM) and ASICs (e.g., TPUs). This trend, however, has led to a growing *disconnect* between the different levels of the computing stack: the programming model, OS/runtime/compiler, and the architecture. Today, these levels still interact with the interfaces and abstractions that were designed based on the traditional generic CPU-centric execution models and architectures, which are rapidly becoming superseded.

1.1. Problem: A Growing Disconnect in the Computing Stack

This *disconnect* has resulted in the critical goals of obtaining highly optimized **performance and execution efficiency**, **programming ease**, and **performance portability** becoming not only challenging to achieve, but also *conflicting* and *at odds* with with one another, necessitating a *tradeoff*. For example, code that is highly optimized for a specific system may require significant programming effort and development time, while sacrificing portability of performance. Similarly, highly portable code typically does not leverage characteristics of the compute and memory resources specific to any architecture, leaving performance on the table. Furthermore, as a result of this disconnect, even for software that is highly optimized for any system, the optimizations are typically *static*. This makes the application unable to dynamically adapt to specific program inputs and runtime conditions, e.g., co-running applications in virtualized environments. In large-scale data centers and cloud infrastructures designed to run a varying set of specialized software frameworks and applications, while employing a variety of compute architectures, memories, and accelerators, the implications of this disconnect is especially critical.

1.2. Research Goal and Directions

A key reason behind the disconnect is that the interfaces between the levels of the computing stack today are *fixed* and are designed to primarily only convey program *functionality*, e.g., via the ISA or virtual memory. No cohesive *flexible* abstractions exist to convey the application *semantics* or *optimizations* which would allow efficient exploitation of the compute and memory resources in modern architectures at different levels of the computing stack. As a result, each lower level of the stack is forced to *infer* or *predict* program characteristics to make optimizations, as opposed to cohesively retaining the application characteristics throughout the stack for synergistic optimizations at appropriate levels. Similarly, the *fixed* and *static* interfaces, do not allow code optimizations to easily adapt to different architectures and runtime conditions.

In my research, I aim to bridge this disconnect by designing richer interfaces and cross-layer abstractions to allow achieving the three-fold goal of highly-optimized performance, reduced burden on the programmer to produce optimized code, *and* portability of performance, *at the same time*. We aim to achieve this in two ways: First, by designing holistic cross-layer abstractions to enable better *expressibility* of program semantics and optimizations to the lower levels of the computing stack; thus enabling higher performance and efficiency by allowing the system to adapt to the application characteristics. Second, by enhancing the interfaces to be more *flexible* to provide the lower levels of the computing stack the ability to *dynamically* adapt the program to systems with different compute and memory resources and react to runtime conditions such as co-running applications and unpredictable input data.

With this goal in mind, my research falls into two categories, described briefly in the next two sections: *(i)* We propose a cohesive set of cross-layer memory abstractions that allow expressing application semantics across the computing stack, and the ability to transform code dynamically depending on runtime requirements to alter its memory access characteristics (Section 2). *(ii)* We proposed new compute abstractions for throughput-oriented systems such as modern GPUs to enable better utilization of compute and memory resources, while reducing programmer effort and making optimizations more portable (Section 3).

2. New Cross-Layer Semantic Memory Abstractions (Ongoing and Future Research)

Prior work primarily focus on optimizations at specific levels of the stack to improve program efficiency in its utilization of memory. However, we aim to leverage the benefits of *cross-layer synergistic* optimizations that enable combining the benefits of optimizations at each level of the stack. To this end, our first step is to design more powerful cross-layer memory abstractions. We target two important goals in designing such abstractions:

2.1. Expressibility: Holistic Expression of Application Semantics, Properties, and Optimizations

First, our goal is to leverage knowledge of the application semantics in its use of memory as well as any optimizations in the application itself to make more efficient use of the memory hierarchy. To this end, we need the cross-layer abstraction to concisely *express* application semantics, characteristics, and optimizations as known by the programmer, compiler, software profiler, etc. to the lower layers of the stack. This would enable more intelligent data placement, caching policies, prefetching,

memory scheduling, data layouts, etc. in complex memory hierarchies with memories with varying access latencies (e.g., NUCA, NUMA), bandwidth, capacities, and heterogeneous substrates (e.g., NVM, hybrid memory). Specifically we target expressing:

(i) **Data Locality and Independence:** Applications are often written (or transformed by the compiler) to exhibit spatial and temporal locality to more efficiently use the faster on-chip caches using techniques such as tiling, recursion, layout transformations, etc.; or partitioned in order to exploit parallelism e.g. in graphs, meshes, etc. However, these techniques are often *implicit* within the application and cannot be leveraged by the lower levels of the computing stack. Our goal here is to design an abstraction to *explicitly* express the locality and independence characteristics of the data to aid the OS/runtime/compiler/architecture to better manage the memory hierarchy.

(ii) **Generic Properties:** The data used in different applications have different properties in terms of which parts are Read-Only or Read/Write, access patterns (e.g. regular/random), compressibility, approximability, etc. Expressing these properties would enable the lower levels of the stack to make more intelligent decisions on data placement, mappings, prefetching, etc., to make better use of the different available memories which have varying characteristics.

(iii) **Data Structure Semantics:** Applications typically have several key data structures that determine the way that data is accessed (e.g., DSLs are often contain a single key data structure). Today, these data structures are expressed simply using its layout in memory—and valuable information regarding the semantics of the data structure itself, its fields, and its constrained access characteristics, which could be used in making more efficient data placement, layout, and movement choices, are lost.

2.2. Flexibility and Control: Enabling Static and Dynamic Program Adaptation

Second, in addition to expressing the above, our goal is to enable *flexibility* in its implementation. This would enable different levels of the stack to *alter* or *tune* the way the application uses the memory hierarchy, statically (via recompilation) or dynamically (on-the-fly); thereby allowing the system to adapt to changes in the execution environment (e.g., in the presence of co-runners in a virtualized environment) and changes in the input data. This would also make the performance more portable across a wide range of systems with varying memory hierarchies and features.

3. Richer Compute Abstractions and Interfaces in GPUs (Completed Research)

In accelerators such as modern day GPUs, the available parallelism as well as the memory resources need to be explicitly managed by the programmer, i.e., there exists no powerful abstraction between the architecture and the programming model to easily manage these resources. Hence, the management of these resources is tied to the programming model itself. This leads to underutilization of resources (and hence, significant loss in performance) when the application is not perfectly tuned for a given GPU. Even when an application is perfectly tuned for one GPU, our experiments show that there can still be a significant degradation in performance when running the same program on a different GPU. Furthermore, it's unclear how to write such architecture-specific programs in virtualized environments, where the same resources are being shared by multiple programs. Even in dedicated environments, the existing interface means that the management of resources is completely *static*, where there could be significant *dynamic* underutilization of resources at runtime, leaving significant performance on the table.

As a part of my research, I proposed two different ways [1, 2, 3] to change this interface between the programming model and architecture in GPUs to enhance resource efficiency, ease of programming, and portability, as discussed below.

3.1. Hardware-Software Cooperative Resource Virtualization: Efficient Management of Parallelism and Locality

To achieve the three-fold goal of enhanced programmability, portability, and resource efficiency, we designed a new hardware-software cooperative framework, ***Zorua***[1], to enhance the interface between the programming model and architecture for the management of several critical compute and memory resources. These resources (threads, registers, scratchpad, etc.) determine the parallelism and locality exploited by the application and are, hence, critical determinants of performance. Zorua decouples the resource management as specified by the programming model and the actual utilization in the system hardware by effectively *virtualizing* each of the resources. As as result of this virtualization, the performance of any program can be far less sensitive to the program implementation making high performance easier to get and far more portable. It also enables the system to adapt gracefully to multiple applications running on the same GPU, irrespective of how the application is written.

3.2. Leveraging Idle Memory and Compute Resources with Helper Thread Execution

Even with highly optimized code, imbalances between the compute/memory resources requirements of an application and the available resources can lead to significant idling of compute units and available memory bandwidth. To *leverage* this undesired wastage, we proposed extensions to the programming model and architecture to support the execution of *helper threads* (or *assist warps* in GPUs). Assist warps allow light-weight execution of secondary code alongside the primary application to enable utilization of idling compute and memory resources in the GPU. The assist warps can be used to perform optimizations to accelerate the program (such as data compression or prefetching), perform background tasks, system-level tasks, etc. To enable definition, use, and efficient execution of assist warps, we proposed and designed a new framework referred to as ***CABA***[2, 3].

References

[1] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu. Zorua: A Holistic Approach to Resource Virtualization in GPUs. In *MICRO*, 2016.

[2] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarangnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu. A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps. In *ISCA*, 2015.

[3] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, S. Ghose, R. Ausavarangnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu. A Framework for Accelerating Bottlenecks in GPU Execution with Assist Warps. In *Advances in GPU Research and Practice, Elsevier*, 2016.