Domain-Specific Virtual Processors as a Portable Programming and Execution Model for Parallel Computational Workloads on Modern Heterogeneous High-Performance Computing Architectures

Dmitry I. Lyakh[1]

Scientific Computing, National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge TN, 37831

Abstract

We advocate domain-specific virtual processors (DSVP) as a portability layer for expressing and executing domain-specific scientific computational workloads on modern heterogeneous HPC architectures, with applications in quantum chemistry. In particular, we introduce a system-wide recursive (hierarchical) hardware encapsulation mechanism into the DSVP architecture and specify a concrete microarchitectural design of an abstract DSVP from which specialized DSVP implementations can be derived for specific scientific domains. Subsequently, we demonstrate, an example of a domain-specific virtual processor specialized to numerical tensor algebra workloads, which is implemented in the ExaTENSOR library developed by the author with a primary focus on the quantum many-body computational workloads on large-scale GPU-accelerated HPC platforms.

Keywords: Scientific computing, high-performance computing, virtual machine, heterogeneous node architecture, tensor algebra.

1 Corresponding author's emails: liakhdi@ornl.gov, quant4me@gmail.com

Introduction

Quantum chemistry has always been rich on parallel computations. In particular, the quantum many-body formalism used in quantum chemistry [1] heavily relies on large-scale numerical linear and multi-linear (tensor) algebra, thus making quantum-chemistry applications highly attractive for massively parallel HPC platforms. Numerous parallel quantum-chemistry codes have been developed and widely adopted by computational chemistry community over time [2-8]. In particular, many of these codes were specifically designed to run on the distributed HPC systems based on multicore processors, following the stable hardware trend of 2000's. Since many quantum-chemical workloads are computationally dominated by a few widely used numerical motifs such as the dense matrix-matrix multiplication and the linear or eigenvalue solver, the transition to distributed multicore systems was relatively straightforward, accompanied by a rather strong support from the math library developers from both industry and academia (few examples include generic Intel MKL, Cray LibSci, IBM ESSL, UTK Plasma/Magma [9], as well as more specialized ELPA [10] libraries). The basic parallel programming tools, like MPI and OpenMP, were relatively easy to use while providing sufficient functionality for implementing parallel algorithms on distributed multicore HPC systems. Perhaps one of a few subtle issues that had to be explicitly accounted for was the non-uniform memory access (NUMA) in OpenMP programming that actually led many codes to restrict themselves to MPI only, instead of using the hybrid MPI+OpenMP programming model (for performance reasons).

The situation began to worsen when an attempt to scale out existing multicore HPC architectures failed to fit in an acceptable power envelope, thus mandating new hardware solutions with less power consumption per executed Flop and transferred Byte. Graphical cards with their superior compute density, lower power consumption and constantly improving general purpose programmability quickly filled in this niche, with an apogee in the fastest supercomputer in the world being based on the NVIDIA GPU cards. The Titan supercomputer [11] was deployed at the Oak Ridge Leadership Computing Facility (OLCF) as early as 2012, only few years after NVIDIA introduced a usable general-purpose GPU (GPGPU) programming model CUDA (Compute Unified Device Architecture). As the price to pay, relative homogeneity of the preceding multicore nodes was lost, leaving programmers with a more complex heterogeneous node architecture with separate memory spaces and different kinds of compute devices. This drastic change in HPC hardware, though inevitable, posed a grand challenge for many scientific codes. The issue of software portability, and especially performance portability, became acute.

For the following discussion, we define *software portability* as a continuous measure, specifically the inverse amount of programming/porting effort (inverse person-months) necessary for a successful launch of an existing code base on a newly emerged HPC architecture. Similarly, *software performance portability* is the inverse amount of programming/porting effort (inverse person-months) necessary for a successful launch of an existing code base on a newly emerged HPC architecture with an efficiency comparable or exceeding the efficiency achieved on the preceding HPC architectures. Importantly, our portability definitions are not limited to the application itself as they generally extend on the entire software stack necessary for a successful (and performant) execution of a given code on new hardware. However, in practice it is useful to exclude lower levels of the software stack from this consideration, for example, the operating system and hardware drivers are normally assumed provided with the new hardware. Ideally, compilers should also be excluded from consideration in this respect. Unfortunately, in practice, many application porting efforts experience numerous issues with compiler bugs and inefficiencies on new hardware, spending significant effort on compiler debugging and search for workarounds. This is becoming more pronounced as the languages evolve in complexity, thus rendering compiler stability and efficiency as one of the key aspects of software portability on new architectures. Different programming languages suffer to a different extent, depending on how fast new language features are adopted by a broad community of developers such that these new features can be well covered by extensive tests. Most notably, the stability of the compiler support of modern Fortran (2003+) has been suffering lately as the adoption of new features of this classical scientific programming language is relatively slow, partially due to compiler bugs, further worsened by a pronounced decline in popularity (C++ and Python have taken a large share of scientific programming nowadays).

Yet, regardless of the programming language, many existing scientific codes, especially legacy codes which have been around for decades, turned out to be rather difficult to port to new HPC architectures. Part of the problem is that many codes were not designed in a sufficiently generic and modular fashion to be able to run efficiently on vastly different hardware. Essentially, the computer hardware specifics has always been an indistinguishable part of the (scientific) application implementation, thus introducing a highly dynamic dependency into the application design. In many implementations, the hardware was not well encapsulated via a sufficiently portable interface, thus leading to a necessity of extensive code modifications every time a new specific kind of computing hardware is introduced, additionally plagued by the existence of multiple memory spaces within a heterogeneous node. The

basic (rather lower-level) parallel abstraction of a concurrent *thread* which is present in major compiled programming languages like C/C++ and Fortran (either the native C++11 std::thread or OpenMP/pthreads thread) is no longer suitable for a clean composition of complex algorithms that are executed on heterogeneous nodes with diverse computing units and multiple memory spaces. In particular, conventional threads do not play well across hardware boundaries and separate memory spaces. They are also rather inconvenient for expressing dependencies between work items as well as controlling the granularity of the work items. The introduction of the OpenMP 4.0+ and OpenACC directive-based programming tools for accelerators did help many codes to begin exploiting accelerated node architectures, sometimes quite successfully, but these tools do not generally solve the problem of flexible decomposition and distribution of complex computational work among diverse compute units in a uniform, portable, and composable fashion. In practice, these directive-based models mostly facilitate expression of the data-based parallelism on accelerators. A more general solution should exploit a tasks-based algorithm expression with their subsequent execution on any subset of computing resources available on the node, where both the task and resource subset granularity should be adjustable in order to achieve the best performance. Again, we stress that in any programming model the requirement of portability necessitates the ability to properly encapsulate the specifics of the computing hardware and memory, exposing a uniform, yet flexible interface to the programmer, which can easily be retargeted to new hardware kinds underneath. For example, in this regard the *executors* proposal to the C++ standard shows some promise at the language level [12]. And, of course, a number of library-implemented (mostly C++) task-based parallel runtimes and frameworks have already been available for experimentation for some time [13-20], providing different levels of hardware abstraction and means for a task-based parallelism expression.

On a more general level, the key to portability (and perhaps performance portability) is the use of a proper hierarchy of higher-level abstractions for expressing (scientific) computational workloads. It is also mandatory to separate the algorithm expression from its execution. Essentially, a portable parallel code design should consist of the following three major layers:

1. *Expression*: High-level computational problem expression in terms of domain-specific abstractions (a concrete domain-specific language may be helpful, but not necessary).
2. *Decomposition*: Hierarchical (static or dynamic) decomposition of the computational problem expressed in terms of domain-specific abstractions into elementary data and work items with controlled granularity.

3. *Execution*: Load-balanced, communication optimal scheduling and execution of the elementary work items on all available (hierarchical and possibly heterogeneous) computing resources encapsulated via portable interfaces.

In this design scheme, only the last (execution) layer is explicitly affected by a hardware change since new (optimized) computational kernels may be necessary for each new hardware kind and a new (heterogeneous) data storage scheme may be required to deal with hierarchical memory. In contrast, the second (decomposition) layer is affected only implicitly, namely the granularity of the hierarchical data and work decomposition may need to be adjusted for the new hardware. The first (expression) layer should in theory be fully agnostic with respect to the underlying hardware. To further improve portability in the execution layer (including performance portability), the code should maximize the use of (performance-)portable software libraries implementing numerical primitives the domain algorithms are composed of. For example, quantum many-body workloads in quantum chemistry heavily rely on matrix-matrix multiplications, tensor contractions, linear and eigenvalue solvers. Having these numerical motifs implemented in optimized math libraries would significantly reduce the portability pressure on the application developers. Finally, the entire execution layer, or even both the decomposition and execution layers, can in principle be hidden in a black-box task-based parallel runtime, thus moving most of the parallel programming complexity from the application developers to the parallel runtime and math library developers. Although this looks like an ideal solution for application developers, it is not always achievable in real life, at least for now. With a great hope that such ideal black-box solutions will become widely available in future, below we elaborate an alternative white-box parallel runtime design approach based on the concept of domain-specific virtual processors (DSVP). In its most generic essence, a DSVP can be viewed as a software processor designed to perform specific parallel computations within a certain (scientific) domain, for example quantum chemistry, on vastly different hardware in a portable fashion. We should immediately note that it is not our goal to reinvent and/or describe the general concept of a parallel virtual machine (PVM), which can be found elsewhere [21], but rather we focus on how this concept and its derivatives can be useful in handling the complexity of scientific programming on heterogeneous distributed HPC systems in specific domain areas such as quantum chemistry. In particular, we describe a concrete DSVP implementation, namely the tensor algebra virtual processor (TAVP) designed and implemented by the author in the form of the ExaTENSOR library [22]. Although the TAVP microarchitecture has its own unique design introducing a number of novel elements such as the fully hierarchical hardware encapsulation, it can also be viewed as a generalization and evolution of earlier efforts, specifically the so-called Super Instruction Architecture framework [23-25] used in the ACES-III [4,5] and ACES-IV

[6,7] software suites for expressing and executing quantum many-body algorithms operating on large dense arrays of numbers. Thus, in this paper we aim at delivering a general DSVP design formalization as a standalone programming and execution model for complex parallel scientific workloads that will cover our own work as well as relevant earlier efforts by others.

Abstract Domain-Specific Virtual Processor

As mentioned above, the main idea behind the concept of a domain-specific virtual processor (DSVP) is to introduce a portable virtual processor architecture, implemented in software instead of hardware, which is specifically tailored to a given class of parallel domain workloads and which is capable of directly executing the underlying set of domain-specific primitives (primitive numerical operations the domain algorithms are composed of) on any kind of hardware in a portable fashion. This set of domain-specific primitives actually defines the (domain-specific) instruction set of a given DSVP, which together with some other relevant design attributes comprises the DSVP architecture. Each concrete DSVP architecture is derived from the abstract DSVP architecture template described below. Importantly, we should note that the notion of "domain-specific" has a hierarchical meaning/structure in our discussion. In other words, we assume that at each level a given set of *more* domain-specific abstractions is based on a set of *less* domain-specific abstractions, that is, we generally assume multiple levels of abstraction ordered with respect to their domain specificity. For example, in the domain of *ab initio* quantum chemistry the top level may be concerned with the abstraction of a quantum many-body ansatz (wave-function form). The next level may include the following abstractions: Many-body operator, many-body operator contraction. At a lower level, we then may have: Tensor, tensor contraction, tensor addition, etc. At this point, our "domain" has moved from a higher-level "quantum many-body theory" to a lower-level "numerical tensor algebra". In general, a DSVP can be introduced at any of these levels, although it makes more sense to introduce a DSVP at some lower-level boundary which still possesses certain domain specificity, thus reducing the complexity of the domain-specific instruction set. Moreover, in general, a DSVP can also have a hierarchical structure such that a higher-level DSVP is composed of DSVP's operating at a lower level of abstraction. At each level, the (domain-specific) algorithm expression is accomplished in terms of the (domain-specific) abstractions used at that level.

In general, computations in a given computational domain are expressed in terms of domain-specific (DS) operations operating on domain-specific (DS) data (at this point, "domain" is an abstract domain). The corresponding association relation between the *DS operation* and *DS data* is shown in the top-right

part of the class diagram depicted in Figure 1. The directional association here means that the DS operations *use* DS data, but the DS data is generally unaware of the DS operations. We will make an extensive use of an important additional assumption formally depicted by a circular composition arrow around the DS data, namely the DS data admits a *recursive decomposition*. That is, in general a DS data object is a collection of the constituent DS data objects of the same class but of smaller size (granularity). The decomposition terminates when a DS data object consists of itself. An immediate consequence of the recursive structure of the DS data is the induced recursive structure of the DS operations, formally depicted by a similar circular composition arrow in the diagram. The presence of the recursive decomposition requirement results in the presence of a deferred public *decompose*() method in both the DS data and DS operation classes.

In order to make DS operations processable (by some formal processor) one needs to encapsulate them into DS instructions (a DS *instruction* is a processable encapsulation of a DS operation). DS instructions *contain* DS operands encapsulating DS data (DS *operand* is a processable encapsulation of DS data). Essentially, both the DS operand and DS instruction classes contain some additional attributes necessary for their respective DS data and DS operation members to become processable by a DSVP. In particular, a DS instruction includes the following attributes: (a) Instruction id; (b) Instruction opcode; (c) Instruction status; (d) Instruction error code; (e) Instruction control field (an optional field containing instruction specification attributes). A DSVP *processes* DS instructions in order to perform the underlying DS operations on the corresponding DS data via invoking the public *process*() method which puts a DS instruction in the instruction processing pipeline. The instruction processing pipeline is determined by the DSVP microarchitecture. The DSVP microarchitecture is defined by a set of interoperating functional DS units the DSVP is composed of. Each DS unit in our DSVP design has a very specific well-defined functional role in processing DS instructions during specific phases of their lifetime. DS units normally progress independently, but they are generally aware of the existence of other DS units as well as the DSVP they belong to. DS units can move DS instructions between each other via DS ports contained in each DS unit. The source of an incoming DS instruction can be identified by the *id* of the DS port the instruction arrived to. In a distributed computing setting, a pair of DSVP can also pass DS instructions between each other via the *send*() and *receive*() methods. There are two special DS units responsible for the transfers of DS instructions between pairs of DSVP. Their interfaces are given by the DS instruction encoder (or simply DS *encoder*) and the DS instruction decoder (or simply DS *decoder*). The DS encoder converts a DS instruction into a portable DS *bytecode* which can be transferred between two DSVP's. The DS decoder performs the opposite

operation, that is, it decodes the received DS bytecode back into a DS instruction object processable by the DSVP. Finally, each DSVP will contain a DS *microcode*, namely a set of code bindings for performing actual DS operations (numerical primitives) on physical hardware for each DS instruction kind. The DS code bindings can either be provided in a form of an optimized numerical DS *driver library* for all relevant compute devices or they can be just-in-time generated upon the need (a combination of the two approaches is also possible). Note that for performance portability purposes the numerical DS driver library can be instantiated from a single template via device-specific auto-tuning and/or code generation (see for example [26]).

So far we have defined a number of structural design aspects of an abstract domain-specific virtual processor, which are rather generic. Now we will extend these further as well as define important behavioral aspects of the DSVP microarchitecture, specifically how it encapsulates the actual HPC platform architecture in a portable fashion and how it subsequently maps data and computations onto the physical hardware. As mentioned above, the requirement of a recursive data and work decomposition will play an ultimate role here. On the other hand, the full DSVP specification requires a proper encapsulation of the HPC platform on which the computations are to be executed. In our design, we require the HPC platform to admit a recursive (hierarchical) decomposition, up to the level of individual devices or even individual cores. To formalize the corresponding portable hierarchical hardware encapsulation scheme, one essentially needs to introduce two portable abstractions: (1) hierarchical compute resources; (2) hierarchical memory resources. In this way, we will cover all available general-purpose HPC platforms, including heterogeneous ones.

The hierarchical *compute resources* provided by an HPC platform are encapsulated via the *Virtual Compute Unit* (VCU) class. The hierarchical *memory resources* provided by an HPC platform are encapsulated via the *Virtual Memory Unit* (VMU) class. As schematically shown in the class diagram in Figure 2, both VCU and VMU objects are defined via a recursive aggregation, starting from the basic VCU and VMU instances. For example, the basic VCU instances can be CPU cores (in both multi- and many-core architectures), GPU streaming multiprocessors or whole GPU devices, FPGA devices, etc. The basic VMU instances can be NUMA DDR memory banks or whole DDR memory, high-bandwidth memory attached to specific devices, specialized memory resources used by GPU, like shared-memory, constant memory, etc. But regardless of the physical nature, the VCU and VMU instances expose only basic attributes to the client (in addition to their enumerated kind), for example the computational throughput (Flop/s) for VCU or the memory volume (Bytes) for VMU.

Subsequently, the basic VCU and VMU instances can further be aggregated into larger composite VCU and VMU instances, up to the full machine level. For example, CPU cores can be aggregated into a CPU socket, CPU sockets can be aggregated into a CPU processor, CPU processor and a GPU device can be aggregated into a heterogeneous node processor, heterogeneous node processors can be aggregated into a smaller node cluster, smaller node clusters can be aggregated into a larger node cluster, and, finally, larger node clusters can be aggregated into the full HPC machine. In general, each VCU instance can be associated with one or more VMU instances and each VMU instance can be associated with one or more VCU instances such that the associated VCU-VMU pairs represent possible compute+memory aggregates in which compute can access the associated memory, either directly or indirectly via a suitable API. These accessibility association links define the cross-topology map between VCU and VMU instances. Any general purpose HPC platform can be recursively decomposed into a coupled hierarchy of the VCU and VMU units. Such hierarchies are formed by a proper enumeration of the basic VCU and VMU units, followed by a recursive aggregation of those, up to the full machine (the root of the aggregation tree). Once the complete aggregation has been established, one can also perform an opposite *decomposition* operation on each VCU or VMU. Figures 3 and 4 illustrate the compute/memory aggregation for two OLCF HPC systems, Titan [11] and Summit [27]. Note that despite the availability of the coherent unified memory on Summit, we still distinguish the high-bandwidth memory close to GPU from the DDR memory close to CPU as the two have rather different bandwidths due to their affinity. In general, it is worth quantifying the topology map links that couple the VCU and VMU aggregation trees in terms of the bandwidth and latency. The VCU-VMU topology map can then be used for optimizing data placement and work scheduling decisions by the DSVP.

Having constructed the coupled VCU-VMU aggregation trees for a given HPC platform, one needs to provide a proper mapping of the individual nodes or the whole terminal subtrees of these trees to the DSVP instances associated with those hardware resources, such that each DSVP is associated with one or more VCU and/or VMU units, and each VCU or VMU unit is directly associated with only one DSVP (note that by owning a composite VCU or VMU unit a DSVP does not automatically own its constituent units). As formally depicted in Figure 2, by assigning DSVP's to the established VCU-VMU resource aggregates (starting from the aggregation tree root) one introduces a hierarchical DSVP composition in which each parent DSVP is decomposed into multiple separate child DSVP's associated with the constituent compute/memory resources. Figure 5 illustrates an example of a complete

encapsulation of a computer cluster consisting of 16 heterogeneous Titan CPU-GPU nodes [11] by a hierarchical DSVP:

(1) 16 cores + DDR3 memory → CPU block;
(2) 14 streaming multiprocessors + GDDR5 memory → GPU block;
(3) CPU block + GPU block → Node = **L2-DSVP**;
(4) 4 Nodes → Cluster = **L1-DSVP**;
(5) 4 Clusters → Full machine = **L0-DSVP**.

Note that the lowest-level DSVP is introduced only at the node level, not at the individual device level as this would likely incur a significant performance penalty since each DSVP always introduces a certain (constant) virtualization overhead. Thus, the lowest compute granularity level at which a DSVP should be introduced must be sufficiently coarse to amortize the (constant) virtualization overhead.

Having finalized the principal specification of the structural DSVP design, let us define the behavioral aspects of the DSVP microarchitecture in regard to a performance-portable data and work mapping. To recap, so far all physical compute and memory hardware resources of an arbitrary HPC platform have been encapsulated by the basic VCU and VMU instances, thus providing a portable hardware abstraction scheme. Subsequently, these basic VCU and VMU instances have been recursively aggregated into larger, composite (generally heterogeneous) VCU and VMU instances, up to the full machine level. The full-machine VCU+VMU resource pair is assigned to the L0-DSVP (level-0 DSVP). Then, by recursively decomposing the VCU+VMU resource pair assigned to a parent DSVP into the constituent VCU+VMU resource pairs with their subsequent distribution among the child DSVP's, a DSVP hierarchy is established, thus completing the structural DSVP specification.

As described at the beginning of this section, given a computational domain problem the DSVP is supposed to solve, the corresponding algorithm is expressed in terms of the DS operations operating on DS data. Both the DS data construction/destruction and the numerical DS operations the algorithm is composed of are fed as DS instructions into the L0-DSVP which represents the entire HPC platform as a serial (virtual) computer designed to directly process these DS instructions. Underneath, starting from the L0-DSVP, each parent DSVP decomposes the incoming DS data and DS operations, which are encapsulated in the DS instructions, and subsequently distributes their constituent (smaller) pieces among the child DSVP's associated with the constituent VCU and VMU resources. The leaves of the DSVP hierarchy tree (DSVP's that consist of themselves) perform the actual data processing expressed by the DS instructions, whereas composite DSVP's (internal tree nodes) only perform data/operation

decomposition and necessary meta-data updates. Possible dependencies between DS instructions are respected via following the corresponding DS data dependencies. Specifically, a DS instruction that consumes a specific DS data object cannot be issued until the DS instruction that produces that same DS data object has completed. Each DS data object is equipped with the so-called Read/Write (R/W) counter showing the current access status on the DS data object with the following possible values: 0 means no current reads or writes, -1 means an outstanding write/update operation, >0 is the number of outstanding read operations. Note that retrieving the R/W counter for a given DS data object may require a communication with another DSVP. Although a DSVP is supposed to perform data-driven DS instruction dependency checking transparently to the client, there are possible variations how this can be done exactly. Specifically, the closest to the root tree level at which a dependency check becomes enabled (for all subsequent levels) defines the granularity of the DS instruction dependency control. For example, enabling the instruction dependency check at the root DSVP level (L0-DSVP) will result in a serial synchronized execution of inter-dependent DS instructions sent to the root DSVP. By moving the instruction dependency control to one of the subsequent DSVP tree levels, the DS instructions at the preceding DSVP tree levels are freed from any dependency checks. They will progress uninterrupted, preserving the original instruction order, until they hit the first DSVP tree level where the DS instruction dependency check is active. By varying the depth of the instruction dependency control logic, one trades off finer dependency control granularity for an increased logic complexity. Importantly, the DS instruction dependency checks at the DSVP tree levels with an active dependency control only need to be performed on the child DS instructions produced by a given DSVP from a parent DS instruction. Clearly, since the parent DS instruction has already been issued, it cannot have dependencies, thus restricting the scope of the dependency check to among its newly spawned child subinstructions. However, this local dependency check may require retrieving non-local meta-data from another DSVP in general.

The inherently hierarchical DSVP architecture is crucial for efficient hierarchical data placement and task scheduling (each DS operation encapsulated in a DS instruction forms a task). Essentially, as opposed to a flat machine view, a hierarchical machine representation simplifies the data placement and task scheduling decisions by removing the size of the HPC system from consideration. In other words, not only the hardware specifics but also the machine scale is hidden in the hierarchical DSVP architecture. The machine scale can be gradually expanded, level-by-level, via a breadth-first traversal of the DSVP tree. Essentially, each DSVP at all DSVP tree levels follows more-or-less same logic:

1. Decompose an incoming (parent) DS instruction into constituent subinstructions (child DS instructions of smaller granularity);
2. Schedule ready-to-be-executed subinstructions on all available resources:
   a) A composite DSVP distributes subinstructions and delegates their execution to its child DSVP's;
   b) A terminal DSVP distributes and actually executes subinstructions on its VCU/VMU units;
3. Test for completion of the scheduled subinstructions, and when they all have completed, retire the parent DS instruction back to where it came from.

Note that in this scheme the scheduling decisions are inherently local, that is, at each moment each DSVP distributes $O(1)$ DS instructions among $O(1)$ child DSVP's, thus not depending on the problem or machine size. Figure 6 depicts the corresponding class diagram. In each DSVP, new DS data objects are mapped by the Data Mapper onto either the VMU instances owned by the DSVP or its child DSVP's (if any). In the former case, the VMU instance is used for storing a DS data object. The Data Mapper class exposes a public method *map*() that accepts a list of VMU instances, a list of DS data objects, and a mapping policy that governs the distribution of the DS data objects among the available VMU instances. Similarly, newly spawned DS operations are scheduled by the Scheduler onto either the VCU instances owned by the DSVP or its child DSVP's (if any). In the former case, the VCU instance is used for executing the DS operation. The Scheduler class exposes a public method *schedule*() that accepts a list of VCU instances, a list of DS operations, and a scheduling policy that governs the distribution of the DS operations among the available VCU instances. In this scheme, the key to efficiency, and thus performance portability, is to match the granularity of the DS data and DS operations to the granularity of the DSVP VMU and VCU units at the corresponding levels of the recursive decomposition. Since the DS operation decomposition is induced by the DS data decomposition, the DS data decomposition needs to be adjusted for a given VCU/VMU aggregation tree (established before the DSVP starts). Both the Data Mapper and the Scheduler should be enhanced with a dynamic load balancer unit to balance the utilization of the memory and compute resources via, for example, data/work stealing, thus altering the instantaneous mapping/scheduling decisions based on the current policy and utilization of the VMU and VCU units. Additionally, the Scheduler should include a dynamic optimization logic based on the locality of DS operands and affinity of DS instructions to the VCU units which are close to the VMU units already holding some of the instruction operands.

The principal abstract DSVP architecture has been described. In the following section, we will demonstrate a concrete DSVP implementation, namely the Tensor Algebra Virtual Processor (TAVP) as implemented by the ExaTENSOR library [22] (here, *domain* = numerical tensor algebra). But before proceeding to the next section, let us highlight some pros and cons of the presented DSVP-based parallel computing model as opposed to some black-box, task-based parallel runtime. As we mentioned before, the key conceptual difference here is that a DSVP is aware of the domain data, operations and algorithms. The price to pay is the explicit dependence of the concrete DSVP architecture/microarchitecture on a specific domain it is designed for. So, how can this compete with a universal generic task-based parallel runtime system designed to execute arbitrary parallel workloads?[2] This question does not really have a clear answer in our point of view. It is very similar to the question of whether domain-specific programming languages generally provide any substantial benefits over advanced general-purpose languages, like C++. The only reasonable answer here is “It depends”. First of all, any complex large-scale computational problem will likely require a complex software solution spanning multiple layers in the software stack. In a sense, the overall implementation complexity is spread across the necessary software stack while its net value as a whole tends to be preserved. For example, the availability of a flexible and efficient parallel runtime that can be leveraged by a particular application would free the application developers from heavy parallelization efforts, dumping all that work on the parallel runtime developers instead. Or, an overly optimistic reliance on compiler functionality would move the implementation complexity from the application developers to the compiler development teams. In an ideal world, moving most of the implementation complexity from the application itself to the lower levels of the software stack would enormously increase the scientific programming productivity. However, in real life, in both cases above, the number of application developers working in different domains far outruns the number of developers working on compilers and parallel runtimes. In practice, the implementation of new features and/or bug fixes in compilers and parallel runtimes may sometimes take years and often easily extend for months. In light of this, the DSVP-based software architecture model may provide certain advantages as it puts more control and leverage into the hands of domain programmers, making them responsible for designing and implementing a parallel runtime system best suited for their computational domain problems. But since it does not have to support all possible computational workloads across domains, its implementation complexity should also be reduced. Here we should mention again that examples of such domain-specific parallel runtimes have existed before [28], [23-25], however their architectural design either

2 Some of the existing task-based distributed parallel runtimes, for example Legion/Realm [18], can formally be viewed as general-purpose software processors (with a black-box microarchitecture) used to execute domain workloads. This is contrasted with the white-box DSVP microarchitecture design specifically tailored to given domain workloads.

did not use the concept of DSVP or it introduced it in an ad hoc fashion without derivation from the abstract (base) DSVP architecture supplied with a clear specification.

In our model, a DSVP consists of a number of interoperating functional DS units, each with a very specific functional role. Consequently, similar to the physical processor design, a possibility of reusing some of the DS units in multiple DSVP implementations would significantly reduce the effort (and cost) of developing new DSVP, thus potentially attracting a larger community of developers who would consider using DSVP for solving their computational domain problems (we imply all possible domains, including scientific domains and numerical math domains). In addition, a clean composition of a DSVP in terms of the DS units with very specific functional roles allows leveraging existing thin software building blocks related to a given unit function instead of working with monolithic generic parallel runtimes. Furthermore, and this is becoming more and more important, the DS unit based DSVP microarchitecture is ideal for hardware co-design, where certain software implemented DS units may eventually find hardware assisted implementations with better performance, thus tailoring the hardware architecture to a given computational domain (domain-specific hardware co-design), without any need to significantly change the software. Thus, the DSVP model can be an effective mediator of such a domain-specific co-design in which hardware sees the given domain through the prism of a DSVP that has already converted all domain requirements into the necessary functional units. Yet, another advantage of the DSVP-based software architecture is a better virtualization of generic HPC resources, transforming a generic HPC platform into a custom virtual machine tailored for parallel processing of specific domain workloads. Finally, last but not least, debugging of domain algorithms and workloads on DSVP is naturally done in terms of high-level domain abstractions constituting the domain-specific instruction set architecture of the DSVP.

Tensor Algebra Virtual Processor

Numerical tensor algebra constitutes the computational basis of ab initio quantum many-body theory, normally consuming the dominant part of computing resources. Consequently, it is important to provide portable, exascale-ready numerical tensor algebra libraries that can be leveraged by quantum many-body codes in quantum chemistry, physics, and materials science [29-32]. ExaTENSOR has a goal to be one of such numerical tensor algebra libraries that is deliberatily built on top of the DSVP architecture. Specifically, it uses the *tensor algebra virtual processor* (TAVP) architecture as a white-box parallel runtime described below. The ExaTENSOR library is currently undergoing an extensive integration testing and debugging before its initial public release. Consequently, we do not have

performance results yet. However, in this regard, the purpose of this paper is to only provide a concrete example of a portable numerical library design based on the DSVP architecture.

First of all, as illustrated in Figure 7, the DS data is specialized as the Tensor class and the DS operation is specialized as the Tensor Algebra (TA) operation class. We define a *tensor* recursively, as a generally sparse collection of tensors, with the terminal tensors defined as dense multi-dimensional arrays of values of some (numerical) type (see Ref. [32] for more details). In this way, we automatically satisfy the recursive DS data composition requirement of the DSVP programming model. The TAVP instruction set encapsulates numerical tensor algebra primitives (primitive operations) as well as some special instructions (control instructions and auxiliary instructions), thus having three classes of processable instructions: Control, auxiliary, and tensor instructions, distinguished by their one-byte opcode. The *control* instructions include STOP (stops every DSVP), RESUME (does nothing), and DUMP_CACHE (dumps the meta-data cache on each DSVP). The *auxiliary* instructions include instructions necessary for defining/registering auxiliary DS objects, like linear spaces, etc., that are used for defining tensors. The control and auxiliary instructions, though necessary, are not important for the following discussion. The *tensor* instructions encapsulate numerical tensor algebra primitives including the following basic tensor algebra operations:

- CREATE/DESTROY: Creates/destroys a Tensor;
- LOAD/SAVE: Loads/saves a Tensor from/to persistent storage;
- INIT/TRANSFORM: Initializes/transforms a Tensor, either intrinsic or user-defined;
- COPY: Makes a copy of a Tensor;
- SLICE/INSERT: Extracts/inserts a slice from/to a Tensor;
- NORM1/NORM2: Computes the 1-norm/2-norm of a Tensor;
- MIN/MAX: Computes the min/max element of a Tensor;
- SCALE: Multiplies a Tensor by a scalar;
- ADD: Adds a Tensor to another Tensor;
- TRACE: Contracts one or more pairs of modes in a Tensor, producing a new Tensor;
- CONTRACT: Contracts one or more pairs of modes between two Tensors, producing a new Tensor (in each pair of modes, the two modes must not belong to the same Tensor argument).

These are the basic intrinsic tensor operations encapsulated by the TAVP instruction set (see Ref. [32] for additional details). Additionally, the TAVP instruction set provides special tensor instructions encapsulating generic unary, binary, and ternary tensor operations that can be dynamically instantiated

during run-time by registering external user-defined functions (tensor algebra microcode) before the TAVP starts. The possibility of such dynamic TA microcode binding/extension is crucial for composability, otherwise the TAVP would be too restrictive for general numerical tensor algebra algorithms which tend to use custom tensor operations quite often.

The TAVP subclass extends the abstract DSVP class. In compliance with the hierarchical DSVP design, the TAVP instances are organized in a tree which defines the TAVP hierarchy: The internal tree nodes are associated with multi-node aggregates of the target HPC platform, whereas the terminal tree nodes (leaves) are associated with individual nodes of the target HPC platform. As a consequence, the TAVP architecture is actually implemented by two TAVP subclasses:

A) TAVP-mng: TAVP-Manager specialization: Implements an internal node of the TAVP hierarchy;
B) TAVP-wrk: TAVP-Worker specialization: Implements a terminal node of the TAVP hierarchy.

A TAVP-mng virtual processor accepts parent tensor instructions, decomposes them into subinstructions, issues ready-to-be-executed subinstructions to its child TAVP's, accepts back the completed subinstructions from its child TAVP's, and finally retires each parent tensor instruction once all its subinstructions have completed. The parent tensor instruction is considered completed successfully if, and only if, all its subinstructions have completed successfully, otherwise the parent tensor instruction is considered completed with error. A TAVP-wrk virtual processor accepts tensor instructions, allocates local resources provided by VMU for their execution, fetches the remote tensor operands (if any), issues tensor instructions to all available VCU if there are no dependencies, tests the completion of issued tensor instructions, uploads the remote output tensor operands to their persistent storage location (if needed), frees the local resources used by the tensor instructions, and finally retires the completed tensor instructions, sending them back to the corresponding TAVP-mng instance.

Following the general DSVP microarchitecture principles, each of the two TAVP subclasses is composed of interoperating TA units extending the abstract DS unit class. Figures 8 and 9 schematically illustrate the structural block diagram of the TAVP-mng and TAVP-wrk virtual processor microarchitecture, respectively. The TAVP-mng virtual processor is composed of eight TA units and one special unit called Tensor Cache. The TAVP-wrk virtual processor is composed of six TA units and one special unit called Tensor Cache. The Tensor Cache is an associative map that is used to store meta-data for each Tensor object currently in use by the TAVP. This is a shared resource accessible by all TA units. Whenever a TA unit needs to access or update information on a specific Tensor object, it sends the corresponding request to the Tensor Cache. We should note that in the current TAVP

implementation used in the ExaTENSOR library the Tensor Cache class is not derived from the DS unit class, but nothing actually prevents it from being implemented as an extension of the DS unit class. In the following discussion, let us describe the TA units implementing the TAVP-mng and TAVP-wrk virtual processors. We should also remind again that TAVP's are organized in a tree (TAVP hierarchy), thus hierarchically encapsulating the underlying HPC system. Figure 10 illustrates a simple example of such a tree, where the leaves are the TAVP-wrk virtual processors, each associated with a single compute node of the target HPC system. Then, according to the tree, a pair of nodes is aggregated into a 2-node cluster managed by a TAVP-mng virtual processor. Finally, two 2-node clusters are aggregated into a 4-node cluster managed by the root TAVP-mng virtual processor. In this example, the full HPC resource consists of four nodes while the TAVP hierarchy consists of four TAVP-wrk instances as well as three TAVP-mng instances, a total of seven TAVP instances. Thus, the number of TAVP virtual processors generally exceeds the number of computing nodes, requiring additional nodes to run the TAVP-mng virtual processors. In our simple synthetic example here, such an overhead is enormous as only four out of seven nodes will perform the actual tensor computations, resulting in a 75% overhead (alternatively, joining the four TAVP-wrk instances to a single TAVP-mng would reduce the overhead to 25%). However, in practice, the TAVP hierarchy should be formed with much larger branching factors in the TAVP tree, for example, a single TAVP-mng can easily manage 64 TAVP-wrk virtual processors (64 computing nodes), and the next level TAVP-mng virtual processor could manage 16 lower-level TAVP-mng virtual processors, resulting in 64x16=1024 TAVP-wrk virtual processors (1024 managed compute nodes in total) and 16+1=17 TAVP-mng virtual processors (assuming only two TAVP-mng levels in the TAVP tree). In this case, the overhead would drop to less than 2%, which is totally acceptable. Another way to reduce the overhead of running TAVP-mng instances is to map them to subsets of CPU cores on multicore systems.

The TAVP-mng virtual processor consists of the following TA units, as shown in Figure 11:

(1) u-Decoder (implements DS decoder interface): Decodes incoming TA bytecode from the parent TAVP-mng instance (or from the client if it is the root TAVP-mng) and passes it to Locator.
(2) Retirer (implements DS encoder interface): Accepts completed tensor instructions from Collector, retires them and then encodes them into TA bytecode, subsequently sending it back to the parent TAVP-mng instance (or client if it is the root TAVP-mng).
(3) Locator (implements DS encoder interface): Accepts new (parent) tensor instructions from u-Decoder as well as previously decomposed (child) tensor instructions from Decomposer and locates the meta-data for all tensor operands by sending remote requests to other TAVP-mng

virtual processors at the same level of the TAVP hierarchy. The fully located and dependency-free tensor instructions are passed to Decomposer, others are deferred.

(4) l-decoder (implements DS decoder interface): Decodes incoming TA bytecode from the meta-data location requests (from other TAVP-mng instances) and passes it to Locator.

(5) Decomposer: Accepts tensor instructions from Locator, decomposes them into subinstructions (child tensor instructions), and passes them to Dispatcher if they have been located or back to Locator otherwise.

(6) Dispatcher: Accepts tensor instructions from Decomposer, schedules and issues them for execution on constituent compute resources via delegating their execution to the child TAVP instances (either TAVP-mng or TAVP-wrk).

(7) Collector: Collects previously issued completed tensor instructions from the child TAVP instances (either TAVP-mng or TAVP-wrk), matches them to their respective parent tensor instructions, passing the latter to Retirer once all the subinstructions have completed.

(8) c-Decoder: Decodes incoming TA bytecode for Collector.

The TAVP-wrk virtual processor consists of the following TA units, as shown in Figure 12:

(1) Decoder (implements DS decoder interface): Decodes the incoming TA bytecode from the parent TAVP-mng instance and passes the decoded tensor instructions to Resourcer.

(2) Resourcer: Accepts tensor instructions from Decoder, checks for data dependencies between tensor instructions (locally), performs output operand substitution to maximize the number of concurrent tensor instructions in the fly, allocates local VMU resources for tensor instructions without dependencies, and passes them to Prefetcher. Also, accepts tensor instructions from Uploader, frees the local VMU resources used by those instructions and passes them to Retirer.

(3) Prefetcher: Accepts tensor instructions from Resourcer, initiates input prefetch for remote tensor operands, tests the completion of the previously issued prefetches, and, once completed, passes tensor instructions to Dispatcher. The data prefetch is implemented via MPI-3 one-sided RMA operations.

(4) Dispatcher: Accepts tensor instructions from Prefetcher, schedules and issues them for execution on all available VCU instances via a portable execution interface (via TA microcode bindings), tests the completion of the previously issued tensor instructions, and, once completed, passes them to Uploader.

(5) Uploader: Accepts executed tensor instructions from Dispatcher, initiates output upload for tensor operands (if needed), tests the completion of the previously issued uploads, and, once

completed, passes tensor instructions to Resourcer. The data upload is implemented via MPI-3 one-sided RMA operations.

(6) Retirer (implements DS encoder interface): Accepts tensor instructions from Resourcer, checks the completion of all locally spawned operations associated with the full completion of a retired tensor instruction, and, if fully completed, encodes the fully completed tensor instructions into TA bytecode and sends it back to the parent TAVP-mng instance. The full completion check is associated with the completion of dynamically spawned local Accumulate instructions that are automatically injected in the instruction pipeline by Resourcer when it performs output argument substitution for an increased concurrency.

As one can see, each TA unit in both the TAVP-mng and TAVP-wrk microarchitectures has a very specific role, progressing on its own. This makes the TAVP design highly modular and weekly coupled. In general, the TA units can be replaced by other compliant implementations, can be reused in another TAVP, as well as new TA units can be added to an existing TAVP (to increase its efficiency, for example). In our current implementation each TA unit is run by a dedicated CPU thread, thus introducing certain virtualization overhead. This overhead is rather negligible for “fat” heterogeneous HPC nodes, like the Summit nodes [27]. However, for “slimmer” HPC nodes, one may need to map non-computing TAVP threads to a smaller subset of CPU cores via oversubscription.

Finally, the tensor algebra microcode (TA microcode), that is, the actual implementation of numerical tensor algebra primitives for different devices is provided by the *tensor algebra driver library*, called TAL-SH [33]. TAL-SH exposes a uniform task-based interface for asynchronously performing basic tensor operations on chosen computing devices (currently TAL-SH supports multi- and many-core CPU and NVIDIA GPU architectures, but it is truly asynchronous only for the NVIDIA GPU so far). TAL-SH encapsulates heterogeneous memory management (by providing a universal memory allocator/deallocator), data transfers between separate memory spaces, and asynchronous execution of basic tensor operations on different computing devices available on the node. TAL-SH is also extensible in terms of incorporating new tensor algebra kernels for already supported as well as new device kinds. Currently, TAL-SH encapsulates a multi-core CPU or an NVIDIA GPU as a single VCU. Correspondingly, the full CPU DDR memory or the GPU GDDR/HBM memory is encapsulated as a single VMU. The VMU resources are preallocated during the TAL-SH initialization in order to avoid severe performance penalties associated with the dynamic memory allocation. The task scheduling logic used by TAL-SH is rather simple, exploiting only the information on the task granularity and

arithmetic intensity (to decide CPU vs GPU), current tensor operand location, and current load of each VCU.

Having described the TAVP architecture, the numerical tensor algebra workloads are executed by the TAVP according to the hierarchical execution model described in the previous section. Specifically, the client (application) sends symbolic tensor instructions to the ExaTENSOR interpreter which translates them into a TA bytecode, subsequently sending it to the root TAVP-mng. The root TAVP-mng receives the TA bytecode, decodes it into tensor instructions and then processes those as described above. Specifically, each TAVP-mng decomposes tensor instructions into subinstructions and schedules them for execution on the lower-level TAVP instances. Ultimately, the tensor instructions arrive at the terminal TAVP-wrk instances which perform the actual numerical processing of the tensor operations encapsulated by these tensor instructions. Figure 13 schematically illustrates the full picture. As mentioned in the previous section, for a given algorithm, the key to performance portability is to match the granularity of tensors and tensor operations to the granularity of the VMU/VCU units at each level of decomposition. Although necessary, this granularity matching alone is not sufficient for high performance of distributed tensor operations and proper communication-avoiding algorithms should be employed by the TAVP-mng logic.

Conclusions

We believe that the hierarchical DSVP-based parallel computing model can provide an attractive route to portability, including performance portability, for a wide range of scientific applications in different computational domains targeting Exascale computing. The DSVP-based software design puts more leverage in hands of domain expert programmers, it focuses on higher-level algorithm expression, debugging and profiling, and it provides better opportunities for hardware co-design. The DS unit based DSVP microarchitecture is highly modular, composable, extensible, and reusable. We have also demonstrated a concrete implementation of the hierarchical tensor algebra virtual processor (TAVP) in the ExaTENSOR library which will be used as a numerical backend in the DIRAC quantum-chemistry software suite [34]. The TAVP development effort took about 1.5-2.0 person-years, showing that similar efforts could be realistically undertaken in other scientific domains where the DSVP-based application design could provide a better ability to handle an ever increasing complexity of large-scale parallel scientific workloads. A conceptual scheme of such portable scientific software development based on the DSVP model is illustrated in Figure 14.

Acknowledgements

This research used resources of the Oak Ridge Leadership Computing Facility (OLCF), which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. We would also like to specifically acknowledge the DOE Office of Science funding allocated for the Center for Accelerated Application Readiness (CAAR) program at OLCF.

References

1. I. Shavitt, R. J. Bartlett. Many-body methods in chemistry and physics: MBPT and Coupled-Cluster Theory. *Cambridge Molecular Science Press*, 2009, ISBN: 9780521818322.
2. http://www.msg.ameslab.gov/gamess/index.html
3. http://www.nwchem-sw.org/index.php/Main_Page
4. http://www.qtp.ufl.edu/ACES/
5. V. Lotrich, N. Flocke, M. Ponton, A. Yau, A. Perera, E. Deumens, R. J. Bartlett. J. Chem. Phys. 128, 194104 (2008).
6. https://github.com/UFParLab
7. B. A. Sanders, J. N. Byrd, N. Jindal, V. F. Lotrich, D. I. Lyakh, A. Perera, R. J. Bartlett. Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS'17), 2017, Orlando FL, ISSN: 1530-2075.
8. http://www.mpqc.org/
9. E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, S. Tomov. J. Phys.: Conf. Series 180. Proceedings of the 5th Annual Conference of Scientific Discovery through Advanced Computing (SciDAC 2009), Jun 14-18, 2009, San Diego CA.
10. A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H.-J. Bungartz, H. Lederer. J. Phys: Cond. Matt. 26, 213201 (2014).
11. https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/
12. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0443r1.html
13. T. Heller, P. Diehl, Z. Byerly, J. Biddiscombe, H. Kaiser. Proceedings of OpenSuCo'17, Supercomputing 2017, Denver CO, Nov 2017.
14. A. Tran Tan, J. Falcou, D. Etiemble, H. Kaiser. Int. J. Parallel Prog. 44, 449-465 (2016).
15. T. G. Mattson, R. Cledat, V. Cave, V. Sarkar, Z. Budimlic, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, M. Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, N. Vrvilo. Proceedings of 2016 IEEE High Performance Extreme Computing Conference (HPEC), Waltham MA, Sep 13-15, 2016. DOI: 10.1109/HPEC.2016.7761580.

16. A. Danalis, H. Jagode, G. Bosilca, J. Dongarra. Proceedings of 2015 IEEE International Conference on Cluster Computing, Chicago IL, Sep 8-11, 2015, pp. 304-313.
17. B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, L. Kale. Proceedings of Supercomputing 2014, New Orleans LA, Nov 16-21, 2014, pp. 647-658.
18. M. Bauer, S. Treichler, E. Slaughter, A. Aiken. Proceedings of Supercomputing 2012 (SC'12), Salt Lake City Utah, Nov 10-16, 2012.
19. J. J. Wilke, J. C. Bennett, R. Clay. Proceedings of Runtime Systems for Extreme Scale Programming Models and Architectures, Supercomputing 2015, Austin TX, Nov 15-20, 2015.
20. http://hihat.modelado.org
21. V. S. Sunderam, G. A. Geist. Parallel Computing 25, 1699 (1999).
22. D. I. Lyakh. Parallel numerical tensor algebra library for heterogeneous HPC systems built on top of the tensor algebra virtual processor. Software to be publicly released in 2018.
23. V. F. Lotrich, J. M. Ponton, A. S. Perera, E. Deumens, R. J. Bartlett, B. A. Sanders. Mol. Phys. 108, 3323 (2010).
24. E. Deumens, V. F. Lotrich, A. S. Perera, R. J. Bartlett, N. Jindal, B. A. Sanders. Annu. Rep. Comput. Chem. 7, 179-191 (2011).
25. N. Jindal, V. F. Lotrich, E. Deumens, B. A. Sanders. Int. J. Parallel Prog. 44, 309-324 (2016).
26. F. G. van Zee, R. A. van de Geijn. ACM Trans. Math. Softw. 41, Article 14 (2015).
27. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/
28. D. Sunderland, B. Peterson, J. Schmidt, A. Humphrey, J. Thornock, M. Berzins. Proceedings of 2016 Second International Workshop on Extreme Scale Programming Models and Middleware, Supercomputing 2016, Salt Lake City Utah, No 13-18, 2016.
29. E. Solomonik, D. Matthews, J. Hammond, J. Demmel. Proceedings of 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, Boston MA, May 20-24, 2013.
30. E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, J. Demmel. J. Parallel Distr. Comput. 74, 3176 (2014).
31. J. A. Calvin, C. A. Lewis, E. F. Valeev. Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, Austin TX, Nov 15, 2015, Article 4.
32. D. I. Lyakh, W. Joubert. *Exascale Scientific Applications: Scalability and Performance Portability*, Edited by T. P. Straatsma, K. B. Antypas, T. J. Williams, CRC Press, Taylor and Francis Group, ISBN: 9781138197541.

33. https://github.com/DmitryLyakh/TAL_SH
34. http://diracprogram.org/doku.php

Figure captions

Figure 1: The class diagram illustrating relations between the key concepts of the DSVP model.

Figure 2: The class diagram illustrating a recursive aggregation of the virtual memory units (VMU) and virtual computing units (VCU), thus inducing a recursive decomposition of the DSVP instances.

Figure 3: The hardware aggregation tree imposed on two Titan nodes (see text for details).

Figure 4: The hardware aggregation tree imposed on two Summit nodes (see text for details).

Figure 5: A complete hierarchical encapsulation of 16 Titan nodes by the DSVP hierarchy: Individual Titan nodes are encapsulated (virtualized) by the L2-DSVP instances, a quartet of L2-DSVP instances is aggregated into an L1-DSVP instance, a quartet of L1-DSVP instances is aggregated into the L0-DSVP instance representing the full (16-node) machine.

Figure 6: The class diagram illustrating mutual relations between the VMU/VCU classes, DS Data and DS Operation classes, and the Data Mapper and Work Scheduler classes. Note the symmetry between data/memory and work/computations.

Figure 7: The class diagram illustrating a specialization of the DSVP model to the numerical tensor algebra domain, introducing the Tensor Algebra Virtual Processor (TAVP) model.

Figure 8: The structure of the TAVP-mng virtual processor composed of interoperating functional units and the tensor meta-data cache.

Figure 9: The structure of the TAVP-wrk virtual processor composed of interoperating functional units and the tensor meta-data cache.

Figure 10: The TAVP hierarchy tree that virtualizes a system of four compute nodes. Each compute node is virtualized by a TAVP-wrk instance. A pair of the TAVP-wrk instances is managed by a TAVP-mng instance. A pair of the lower-level TAVP-mng instances is managed by the root TAVP-mng instance.

Figure 11: The class diagram showing relations between different TA units the TAVP-mng virtual processor is composed of (a shared Tensor Cache is omitted). The bidirectional association lines designate a cooperation (instruction flow) between the linked TA units.

Figure 12: The class diagram showing relations between different TA units the TAVP-wrk virtual processor is composed of (a shared Tensor Cache is omitted). The bidirectional association lines designate a cooperation (instruction flow) between the linked TA units.

Figure 13: A schematic illustration of the full TAVP programming and execution model. On top, the tensor operations scheduled by the client undergo a recursive (hierarchical) decomposition into progressively smaller pieces, induced by the recursive (hierarchical) tensor decomposition. On bottom, the HPC system hardware is recursively aggregated into progressively larger (heterogeneous) compute and memory units. In the middle, the hierarchical tensor algebra virtual processor (TAVP) recursively maps the data and work onto the hierarchical memory and compute resources at each level of the decomposition, thus removing not only the hardware specificity but the HPC system scale as well.
Figure 14: A conceptual scheme of portable scientific software development based on the DSVP programming and execution model.

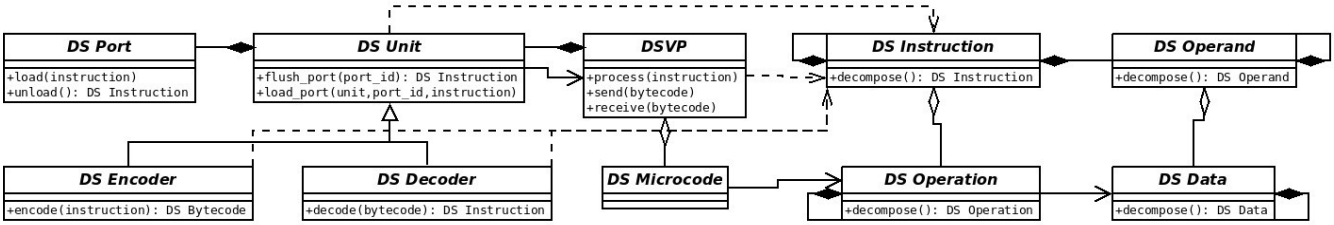
DS Port
+load(instruction)
+unload(): DS Instruction
DS Unit
+flush_port(port_id): DS Instruction
+load_port(unit,port_id,instruction)
DSVP
+process(instruction)
+send(bytecode)
+receive(bytecode)
DS Instruction
+decompose(): DS Instruction
DS Operand
+decompose(): DS Operand
DS Encoder
+encode(instruction): DS Bytecode
DS Decoder
+decode(bytecode): DS Instruction
DS Microcode
DS Operation
+decompose(): DS Operation
DS Data
+decompose(): DS Data

Figure 1

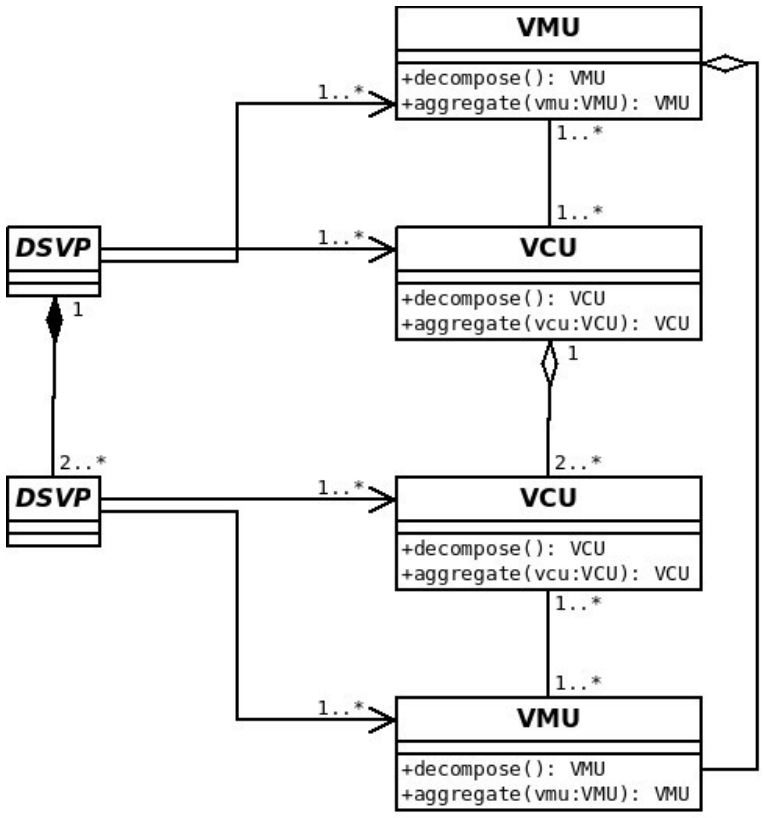
VMU
+decompose(): VMU
+aggregate(vmu:VMU): VMU
1..*
1..*
1..*
DSVP
1..*
VCU
+decompose(): VCU
+aggregate(vcu:VCU): VCU
1
1
2..*
2..*
DSVP
1..*
VCU
+decompose(): VCU
+aggregate(vcu:VCU): VCU
1..*
1..*
1..*
VMU
+decompose(): VMU
+aggregate(vmu:VMU): VMU

Figure 2

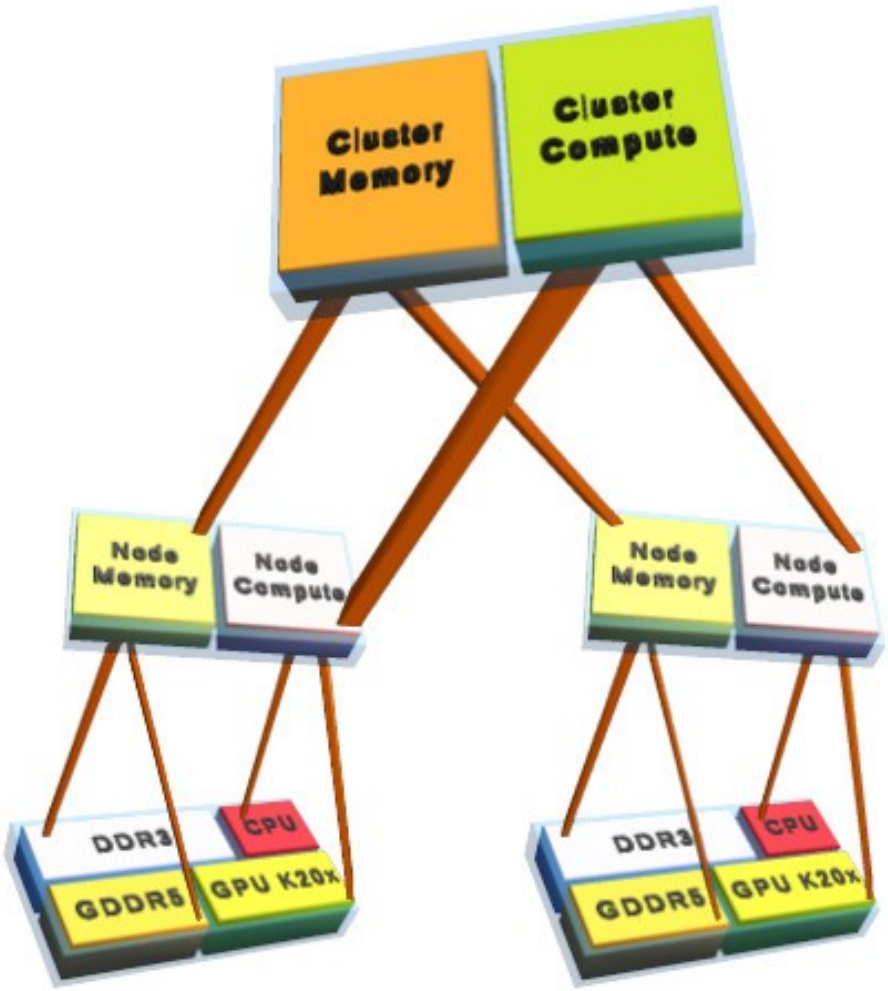
Cluster Memory
Cluster Compute
Node Memory
Node Compute
Node Memory
Node Compute
DDR3
CPU
GDDR5
GPU K20x
DDR3
CPU
GDDR5
GPU K20x

Figure 3

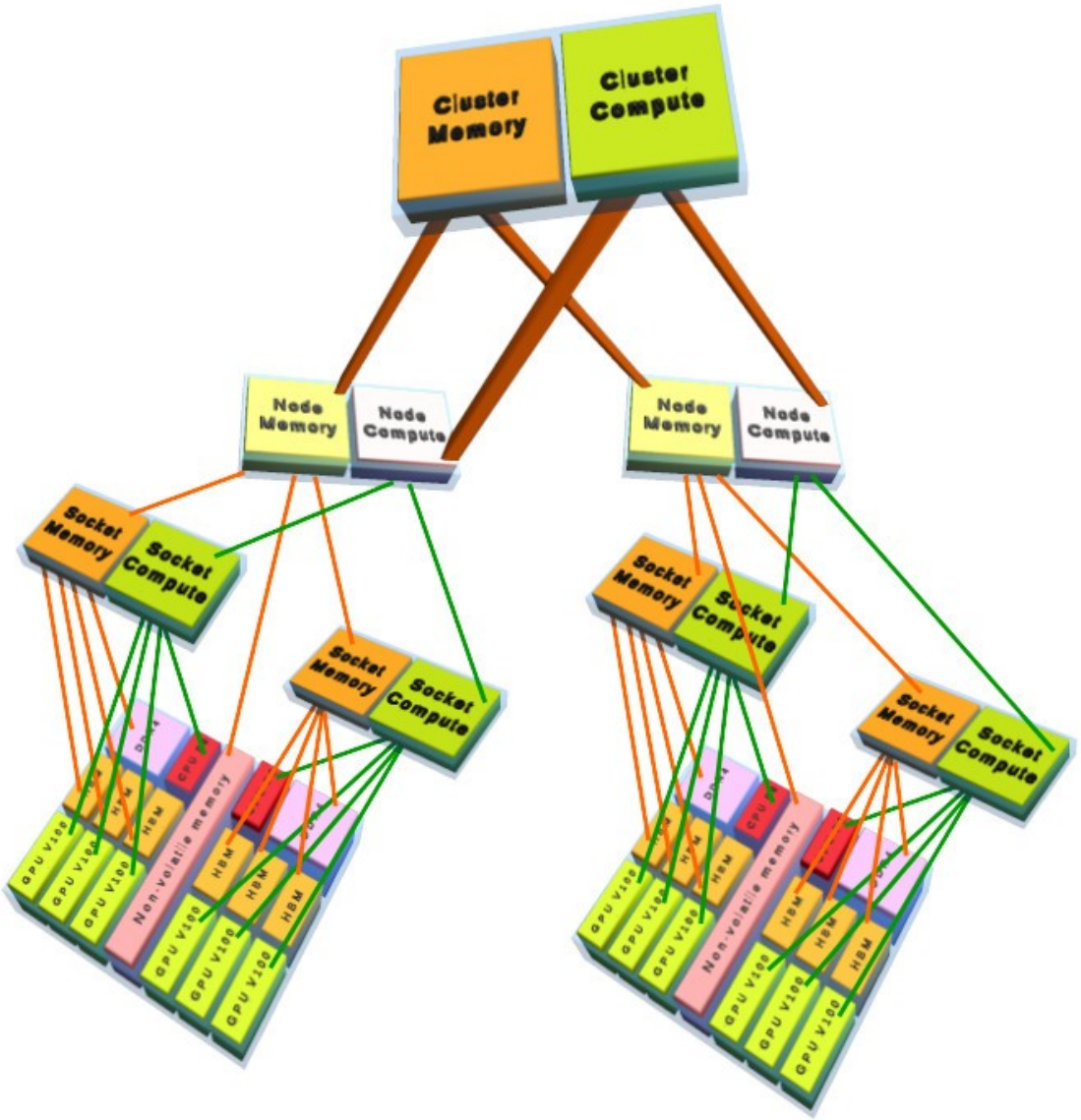
Cluster Memory
Cluster Compute
Node Memory
Node Compute
Node Memory
Node Compute
Socket Memory
Socket Compute
Socket Memory
Socket Compute
Socket Memory
Socket Compute
Socket Memory
Socket Compute
DRAM
CPU
NVM
HBM
GPU V100
Non-volatile memory

Figure 4

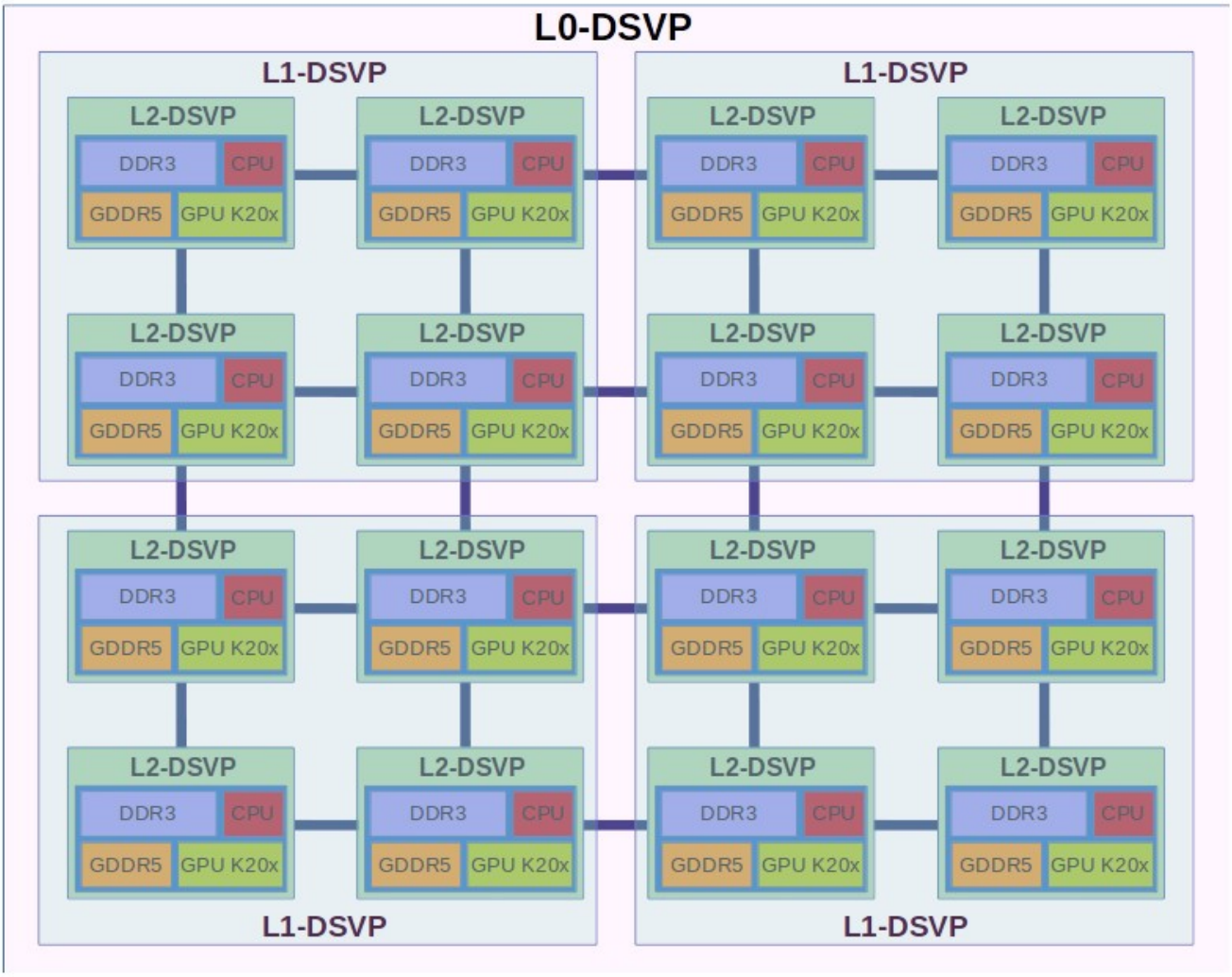
L0-DSVP
L1-DSVP
L1-DSVP
L2-DSVP
DDR3
CPU
GDDR5
GPU K20x
L1-DSVP
L1-DSVP

Figure 5

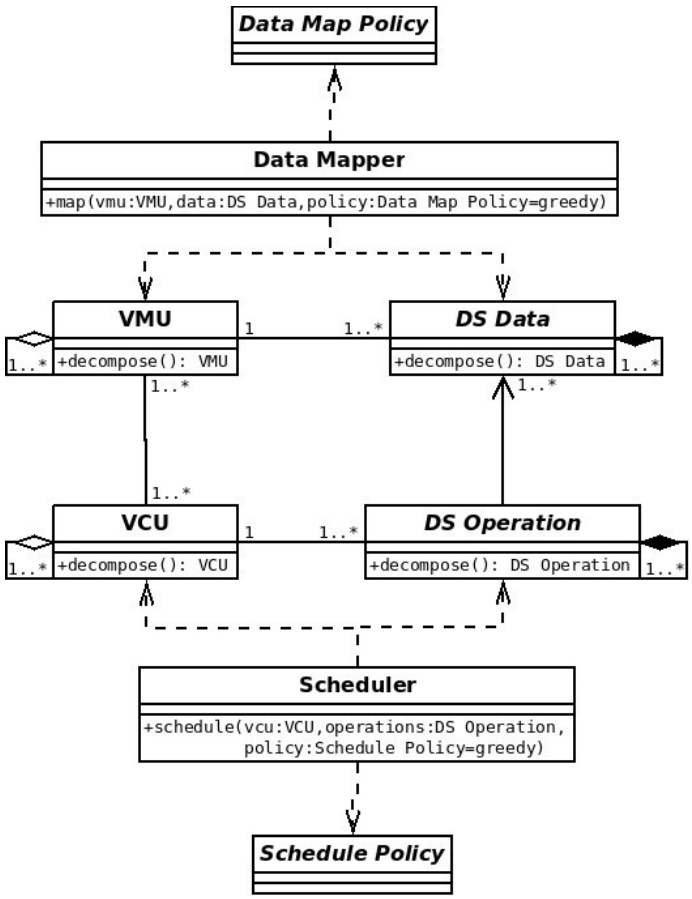
Data Map Policy
Data Mapper
+map(vmu:VMU,data:DS Data,policy:Data Map Policy=greedy)
VMU
+decompose(): VMU
DS Data
+decompose(): DS Data
VCU
+decompose(): VCU
DS Operation
+decompose(): DS Operation
Scheduler
+schedule(vcu:VCU,operations:DS Operation,
policy:Schedule Policy=greedy)
Schedule Policy
1
1..*

Figure 6

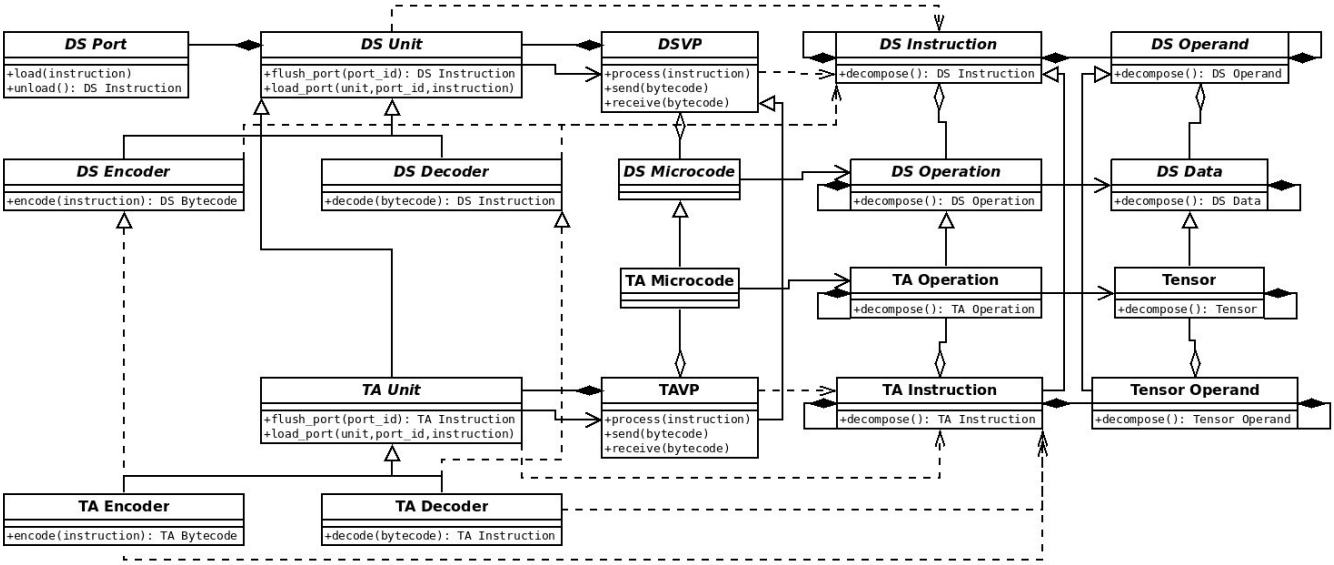
DS Port
+load(instruction)
+unload(): DS Instruction
DS Unit
+flush_port(port_id): DS Instruction
+load_port(unit,port_id,instruction)
DSVP
+process(instruction)
+send(bytecode)
+receive(bytecode)
DS Instruction
+decompose(): DS Instruction
DS Operand
+decompose(): DS Operand
DS Encoder
+encode(instruction): DS Bytecode
DS Decoder
+decode(bytecode): DS Instruction
DS Microcode
DS Operation
+decompose(): DS Operation
DS Data
+decompose(): DS Data
TA Microcode
TA Operation
+decompose(): TA Operation
Tensor
+decompose(): Tensor
TA Unit
+flush_port(port_id): TA Instruction
+load_port(unit,port_id,instruction)
TAVP
+process(instruction)
+send(bytecode)
+receive(bytecode)
TA Instruction
+decompose(): TA Instruction
Tensor Operand
+decompose(): Tensor Operand
TA Encoder
+encode(instruction): TA Bytecode
TA Decoder
+decode(bytecode): TA Instruction

Figure 7

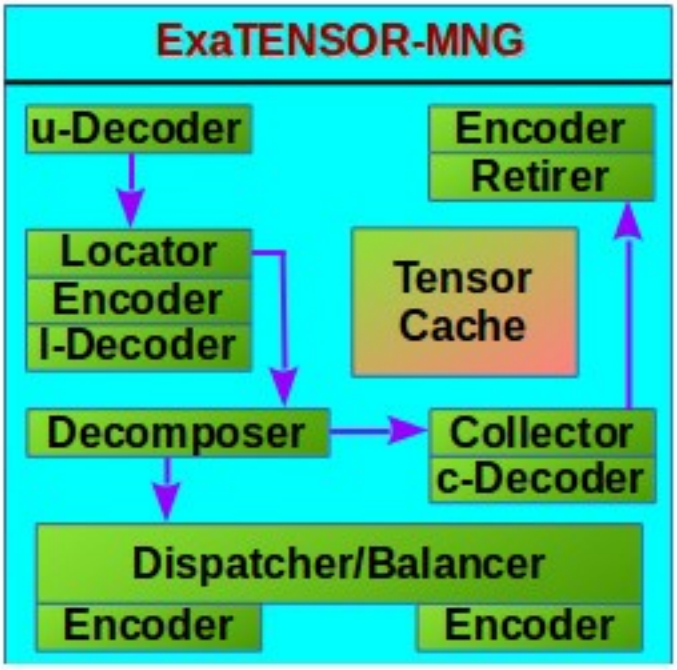
ExaTENSOR-MNG
u-Decoder
Encoder
Retirer
Locator
Encoder
I-Decoder
Tensor
Cache
Decomposer
Collector
c-Decoder
Dispatcher/Balancer
Encoder
Encoder

Figure 8

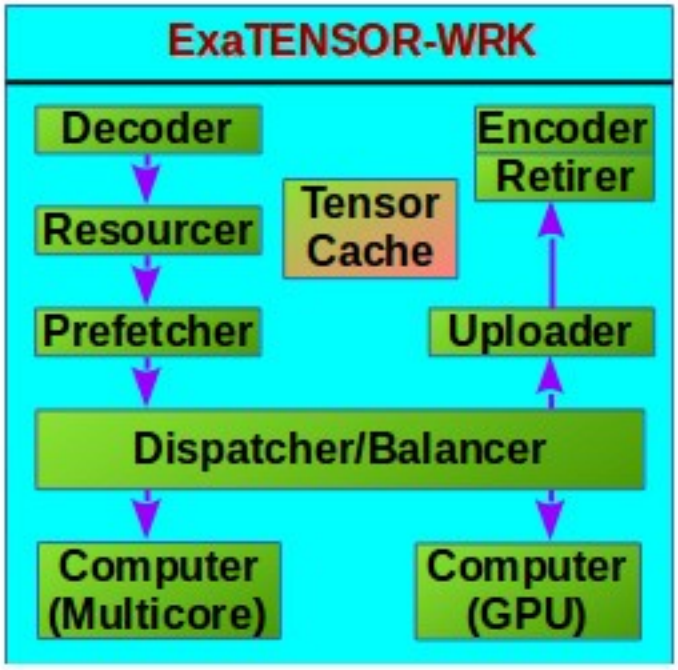
ExaTENSOR-WRK
Decoder
Resourcer
Prefetcher
Tensor Cache
Encoder
Retirer
Uploader
Dispatcher/Balancer
Computer (Multicore)
Computer (GPU)

Figure 9

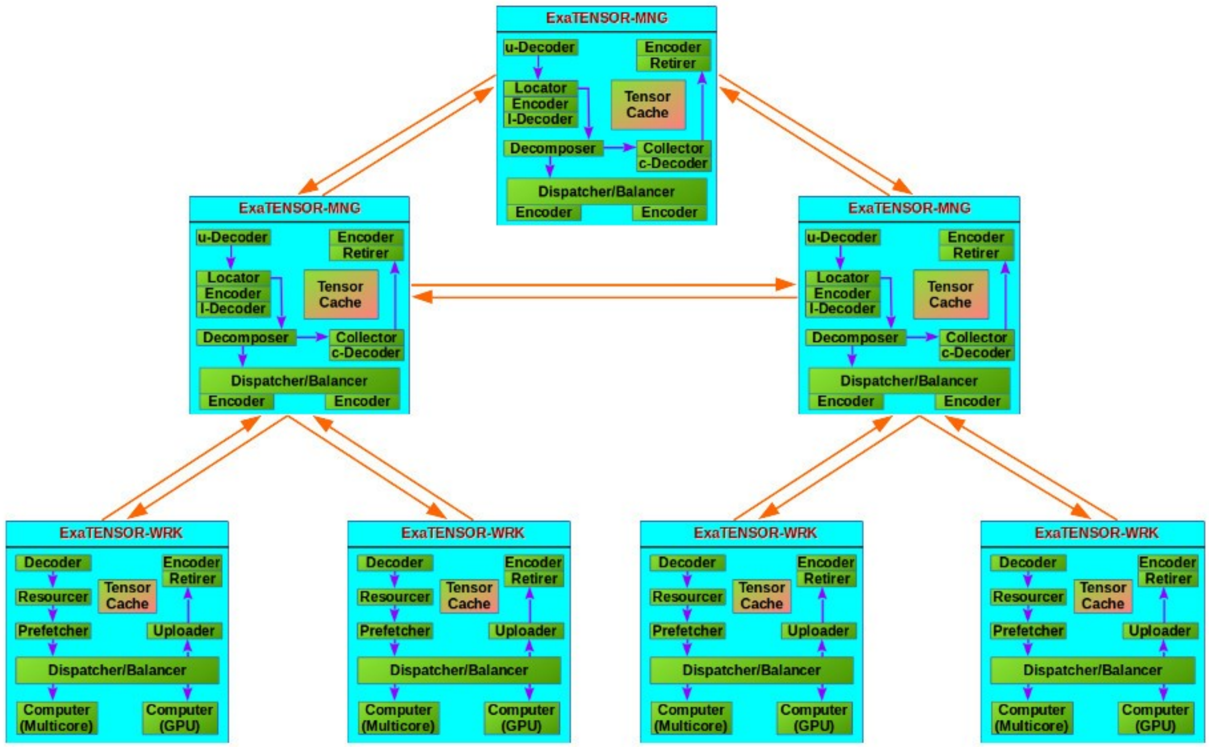
ExaTENSOR-MNG
u-Decoder
Encoder
Retirer
Locator
Encoder
l-Decoder
Tensor
Cache
Decomposer
Collector
c-Decoder
Dispatcher/Balancer
Encoder
Encoder
ExaTENSOR-WRK
Decoder
Resourcer
Prefetcher
Uploader
Computer
(Multicore)
Computer
(GPU)

Figure 10

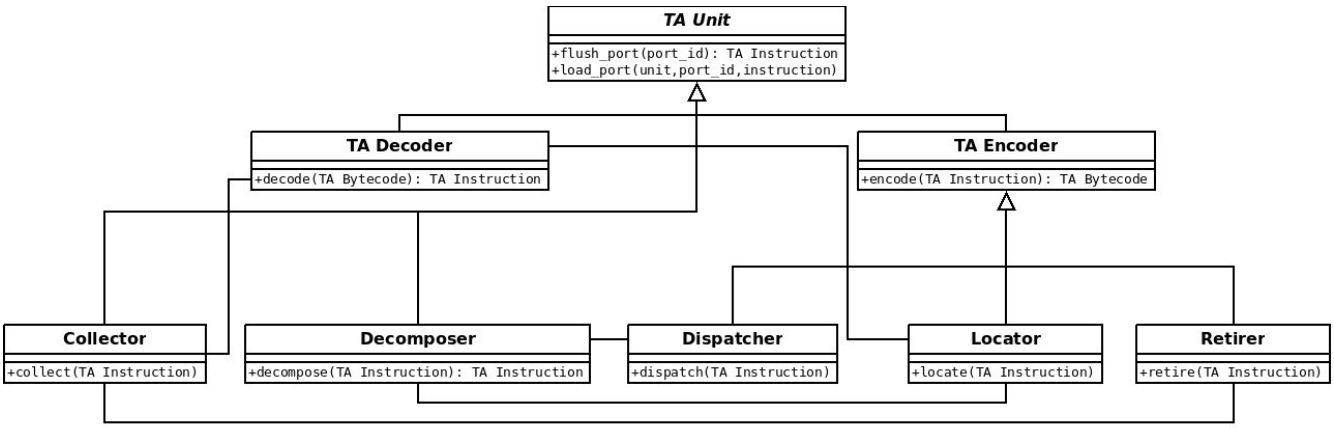
TA Unit
+flush_port(port_id): TA Instruction
+load_port(unit,port_id,instruction)
TA Decoder
+decode(TA Bytecode): TA Instruction
TA Encoder
+encode(TA Instruction): TA Bytecode
Collector
+collect(TA Instruction)
Decomposer
+decompose(TA Instruction): TA Instruction
Dispatcher
+dispatch(TA Instruction)
Locator
+locate(TA Instruction)
Retirer
+retire(TA Instruction)

Figure 11

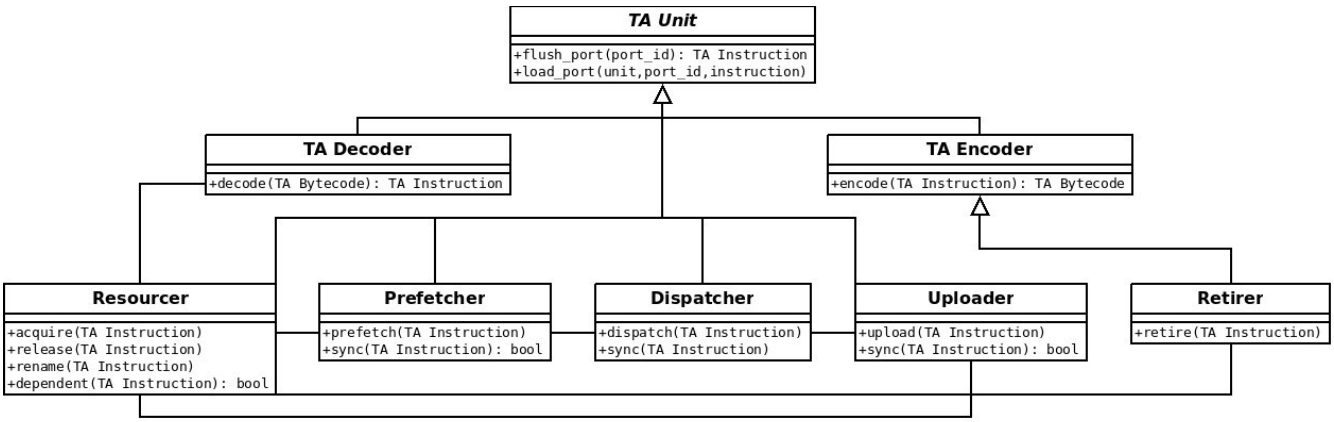
TA Unit
+flush_port(port_id): TA Instruction
+load_port(unit,port_id,instruction)
TA Decoder
+decode(TA Bytecode): TA Instruction
TA Encoder
+encode(TA Instruction): TA Bytecode
Resourcer
+acquire(TA Instruction)
+release(TA Instruction)
+rename(TA Instruction)
+dependent(TA Instruction): bool
Prefetcher
+prefetch(TA Instruction)
+sync(TA Instruction): bool
Dispatcher
+dispatch(TA Instruction)
+sync(TA Instruction)
Uploader
+upload(TA Instruction)
+sync(TA Instruction): bool
Retirer
+retire(TA Instruction)

Figure 12

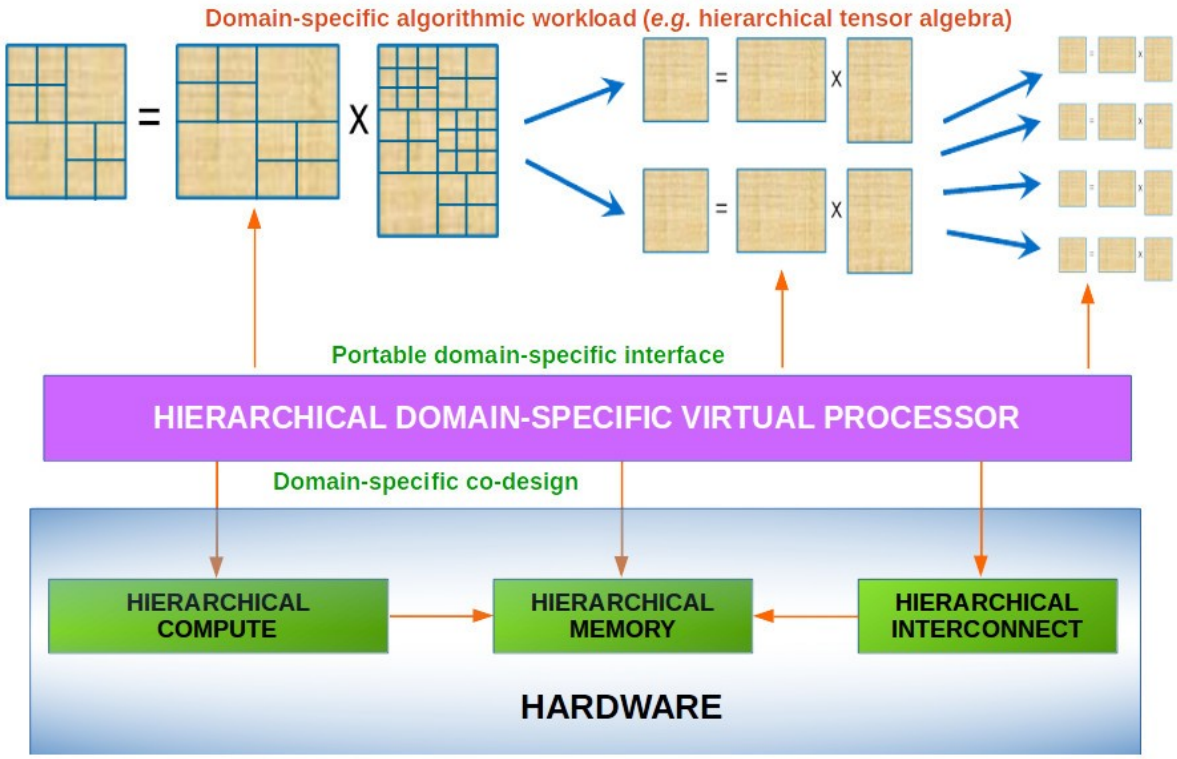
Domain-specific algorithmic workload (e.g. hierarchical tensor algebra)
Portable domain-specific interface
HIERARCHICAL DOMAIN-SPECIFIC VIRTUAL PROCESSOR
Domain-specific co-design
HIERARCHICAL COMPUTE
HIERARCHICAL MEMORY
HIERARCHICAL INTERCONNECT
HARDWARE

Figure 13

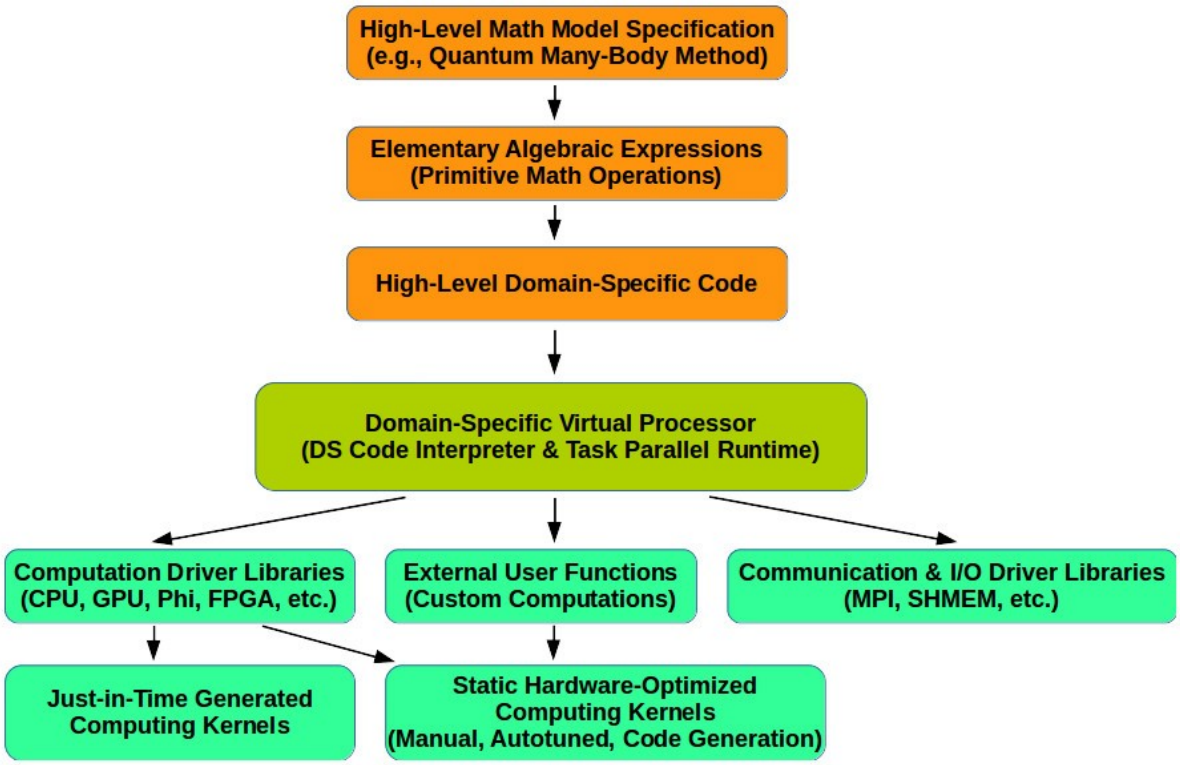
High-Level Math Model Specification
(e.g., Quantum Many-Body Method)
Elementary Algebraic Expressions
(Primitive Math Operations)
High-Level Domain-Specific Code
Domain-Specific Virtual Processor
(DS Code Interpreter & Task Parallel Runtime)
Computation Driver Libraries
(CPU, GPU, Phi, FPGA, etc.)
External User Functions
(Custom Computations)
Communication & I/O Driver Libraries
(MPI, SHMEM, etc.)
Just-in-Time Generated
Computing Kernels
Static Hardware-Optimized
Computing Kernels
(Manual, Autotuned, Code Generation)

Figure 14