Exploring the Use of Novel Spatial Accelerators in Scientific Applications

Rizwan A. Ashraf
rizwan.ashraf@pnnl.gov
Pacific Northwest National Laboratory
Richland, Washington, USA

Roberto Gioiosa
roberto.gioiosa@pnnl.gov
Pacific Northwest National Laboratory
Richland, Washington, USA

ABSTRACT

Driven by the need to find alternative accelerators which can viably replace graphics processing units (GPUs) in next-generation Supercomputing systems, this paper proposes a methodology to enable agile application/hardware co-design. The application-first methodology provides the ability to come up with design of accelerators while working with real-world workloads, available accelerators, and system software. The iterative design process targets a set of kernels in a workload for performance estimates that can prune the design space for later phases of detailed architectural evaluations.

To this effect, in this paper, a novel data-parallel device model is introduced that simulates the latency of performance-sensitive operations in an accelerator including data transfers and kernel computation using multi-core CPUs. The use of off-the-shelf simulators, such as pre-RTL simulator Aladdin or multiple tools available for exploring the design of deep neural network accelerators (e.g., Timeloop) is demonstrated for evaluation of various accelerator designs using applications with realistic inputs. Examples of multiple device configurations that are instantiable in a system are explored to evaluate the performance benefit of deploying novel accelerators. The proposed device is integrated with a programming model and system software to potentially explore the impacts of high-level programming languages/compilers and low-level effects such as task scheduling on multiple accelerators. We analyze our methodology for a set of applications that represent high-performance computing (HPC) and graph analytics. The applications include a computational chemistry kernel realized using tensor contractions, triangle counting, GraphSAGE and Breadth-first Search. These applications include kernels such as dense matrix-dense matrix multiplication, sparse matrix-spare matrix multiplication, and sparse matrix-dense vector multiplication. Our results indicate potential performance benefits and insights for system design by including accelerators that realize these kernels along-side general purpose accelerators.

CCS CONCEPTS

• **Hardware → Emerging architectures**; • **Computer systems organization → Special purpose systems**.

KEYWORDS

Accelerated Computing, Simulation, Hardware/Software Co-design

ICPE '22, April 9–13, 2022, Bejing, China.

ACM ISBN 978-1-4503-9143-6/22/04…$15.00
https://doi.org/10.1145/3489525.3511690

ACM Reference Format:
Rizwan A. Ashraf and Roberto Gioiosa. 2022. Exploring the Use of Novel Spatial Accelerators in Scientific Applications. In *Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3489525.3511690

1 INTRODUCTION

The need of processing and analyzing extremely large amount of data with performance and power/energy constraints has motivated the development of many highly-specialized and energy-efficient computing systems. This trend is evident in embedded systems, such as mobile phones or smart sensors, where systems contain a myriad of small, specialized application-specific integrated circuit (ASIC) processors. Large-scale high-performance computing (HPC) systems have also embraced heterogeneous devices to speed up computation while maintaining a strict power budget. Looking forward, one can imagine that more application-specific accelerators will be incorporated into System on Chip (SoC) designs, which will contain general-purpose computing elements, such as central processing unit (CPU) and graphics processing unit (GPU) cores, application-specific accelerators (e.g., generalized matrix-multiplication (GEMM)), and semi-programmable coarse-grain reconfigurable devices. Designing such increasingly complex devices for future systems is becoming an incredibly difficult task.

To further increase this complexity, it has become evident that novel hardware concepts alone do not necessarily provide the required performance and energy efficiency increase and that, to take full advantage of emerging architectures, novel system software and algorithms also need to be co-developed in a tight, agile, and iterative co-design cycle [12]. Since both applications/algorithms and processing architectures are becoming increasingly complex, there is a need to develop agile, composable, and flexible methodologies for hardware/software co-design.

State-of-the-art design space exploration (DSE) tools mostly focus on the design of stand-alone accelerators [13]. While several tools exist that allow integration of custom processing elements in a SoC [10], they mostly provide an interface with the rest of the SoC but leave the actual design and DSE to external tools and do not generally tackle the co-development of system software and algorithms. Traditional DSE is still too slow to allow an agile hardware/software co-design cycle, which is not only a problem for those domains under strict time-to-market expectation but also for domains, such as scientific computing using artificial intelligence (AI), in which the hardware/software co-design space is too large and complex to be explored with traditional methodologies. A promising direction is represented by novel tools that enable fast, pre-RTL simulations

of hardware concepts, such as Aladdin [18], Structural Simulator Toolkit (SST) [11], and SST/Macro. However, these tools still focus mostly on the hardware side and do not provide an easy way to incorporate novel algorithms and system software.

In this work, we identify three primary challenges: (1) development of methodologies and tools that allow execution of full applications with realistic input set on novel hardware concepts; (2) development of composable methodologies that allow new hardware designs to be integrated with existing hardware components and other simulated accelerators; (3) development of agile methodologies and tools that provide fairly accurate solutions within a short term (pre-RTL) and greatly reduce the design space to be explored in a later phase. To solve these challenges, we proposed novel, agile, accurate, and composble methodologies and tools for true hardware/software co-design. Our methodology allows (1) new hardware concepts to be evaluated with a full, state-of-the-art software stack, (2) fast and accurate pre-RTL evaluation, (3) inter-operability with existing and trusted tools for hardware development, (4) full-system integration with existing hardware, such as GPU and deep learning (DL) accelerators.

Our hardware/software co-design methodology is based on an asynchronous task-based programming model and runtime for extremely heterogeneous systems, namely Minos Computing Library (MCL) [6], integrated with a hardware abstraction layer to incorporate novel hardware concepts modeled with third-party off-the-shelf simulators and emulators. The iterative design methodology is demonstrated in Figure 1. The advantages of our methodology is twofold. On one side, MCL runtime software allows the execution of full applications with representative input sets on modern heterogeneous systems. On the other, the hardware abstraction layer provides an easy and clean interface to include output parameters from most modern simulators and emulators without requiring any modification to those tools. The hardware abstraction layer provides a system-level view of the simulated devices, which can be queried with existing command line tools. We call our simulated device as *Proteus* (named with reference to Greek mythology). With the proposed Proteus device, the final result is a set of tools that enable true hardware/software co-design of applications/algorithms and architectural concepts. The new hardware designs can be integrated in existing heterogeneous systems while the MCL runtime will seamlessly execute or emulate tasks on either real or simulated devices. Also note that both MCL runtime and Proteus device can be used separately and/or in combination with other tools, which might be important for existing development cycles. To the best of our knowledge, our methodology is the first that allows studying new hardware designs next to existing heterogeneous devices, while executing full modern task-based applications, and transparently integrating third-party hardware development tools.

To demonstrate the effectiveness of our methodology, we first provide a set of sensitivity studies in which we integrate several external hardware development tools, namely Aladdin [18], Timeloop [13] and Accelergy [22], Sparseloop [23], MAESTRO [9], and Scale-Sim [15] using representative kernels from HPC, graph and AI domains. Next, we show how the selected hardware tools can be seamlessly integrated with existing heterogeneous devices in a multi-GPU system. Finally, we demonstrate how the MCL runtime scheduler can effectively employ all available resources to achieve

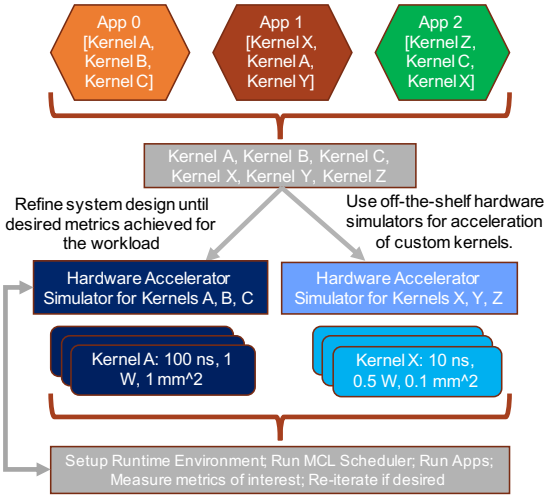

Figure 1: Design space exploration using Proteus: An iterative process to explore the design of domain-specific accelerators using hardware design tools for application performance engineers. The proposed methodology enables agile design space exploration for a workload of interest.

higher performance. Our results demonstrate orders of magnitude performance benefit by employing custom accelerators, especially for an application that employs dense computation kernels.

In summary, this paper makes the following broad contributions:

- A novel methodology for true, agile, and composable hardware/software co-design.
- A set of tools to support such co-design methodology and that allows execution of existing complex applications.
- Several co-design studies of novel hardware accelerators for HPC, graph analytics, and AI domains.
- A full performance and power evaluation of both the novel accelerators and the full integrated system.

The rest of this paper is structured in the following way. Section 2 summarizes existing knowledge and related work. Section 3 describes our novel methodology. Section 4 introduces the hardware abstraction layer. Section 5 shows our experimental evaluation. Finally, section 6 concludes this work.

2 RELATED WORK

Simulation is widely utilized to predict the performance of computer architectures before hardware is produced. It provides a platform to validate ideas before the expensive path of producing hardware is pursued. Emulation is also utilized for the same purpose. However, it limits exposure to observable system states and does not expose the internal state of the system in simulation. Therefore, simulations can be time consuming based on the level of accuracy and detail required. Both can correctly predict the performance of an application on an emerging architecture or system.

Cycle-level simulators such as Gem5 [3] are widely used in the industry to design next-generation processor architectures. Micro-architectural simulators model different functional units, register

Table 1: Summarized descriptions of hardware design exploration tools used in this work.

Simulator	Approach	Kernels	Design Space	Metrics
Aladdin [18]	pre-RTL simulation of dynamic traces	arbitrary functions expressed in high-level program	loop unrolling, pipelining, data partitioning	number of cycles, area and power for 45 nm
Timeloop [13]	exhaustive search (mapper) to find an efficient mapping given a problem onto specified accelerator architecture	dense computation expressed through bounded loops	architecture hierarchy and tiling	number of cycles, area and energy for 16nm, 40 nm and 65 nm [22]
Sparseloop [23]	exhaustive search to find an efficient mapping given a problem onto specified accelerator architecture	sparse computation expressed through bounded loops	architecture hierarchy, tiling, sparse input format and sparse optimization (skipping and gating)	number of cycles, area and energy for 16 nm, 40 nm and 65 nm [22]
MAESTRO [9]	analytical tool for modeling and evaluating the performance of different mappings of a problem onto parametered architecture	limited set of operators included within the tool	tiling factors and parameters of fixed architecture	number of cycles, area and power, etc.
Scale-Sim [15]	cycle level memory trace and utilization for evaluating the performance of different mappings of a CNN onto a systolic array	limited set of convolution-based operators to support CNNs	Parameters of fixed systolic architecture, dataflow mapping strategy (weight, input, output)	number of cycles, bandwidth requirements, utilization, etc.

files, load/store queues, pre-fetching logic, and the timing information for various micro-operations. To assess the impact of an micro-architectural level change such as increased instruction window, or a reliability mechanism, each and every instruction of a program is executed, whereby providing the opportunity to analyze the state of the micro-architecture (architectural state) at each cycle. This level of detail can be prohibitive and does not make it feasible to run large workloads.

Cycle-approximate simulators [19, 24] have been proposed to limit the complexity associated with cycle-accurate simulators while being faster and nearly as accurate using techniques such as statistical simulation, sampling, and generation of synthetic benchmarks. Simulation time can be reduced significantly at the expense of reduced accuracy or not getting the complete picture of micro-architectural state. For example, Zsim [17] reports up to a 24% difference in Instructions per cycle for a multi-core processor.

However, cycle-approximate simulators do not have the ability to execute operating system code. Whereas, full system emulators such as AMD SimNow [2] have the support to execute the complete software stack. Virtualization is used to emulate target system on a host processor. Only a functional representation of the system that ensures correctness of an executing application is maintained and timing or internal state of the system can not be provided. Therefore, different simulators are coupled together to gather cycle-accurate timing information for some fancy executed instruction and this information is utilized in the later phases of the emulation. Manifold [21] (QEMU/KVM and a cycle-accurate simulator) is an example of a coupled framework.

Our proposed work is along the lines of coupling capabilities of multiple frameworks to help with the design of next-generation computing nodes in systems that are expected to do well in the era of convergence of HPC, graph analytics and data analytics. Driven by these needs, we investigate the benefit of employing spatial accelerators whose hardware is not available. The characteristics of accelerators are defined by running stand-alone simulations. Frameworks such as Timeloop [13] and MAESTRO [9] are inspired by the wide interest in deep neural network acceleration. We utilize these tools in our work to accelerate kernels such as dense general matrix multiplication. The simulation frameworks utilized in our work are discussed in Section 3.3 and summarized in Table 1. We employ the outputs produced by these tools in our framework as a hybrid simulation capability by emulating these devices on existing CPUs along with other hardware accelerators in the system all while running state-of-the-art system software and operating system. This provides the opportunity to investigate system-level design issues and assess the benefit to real-world workloads with almost same speed as if the accelerator hardware is available.

3 METHODOLOGY

To solve the challenges listed earlier, we propose a novel agile, flexible, and composable hardware/software co-design methodology. We assume a SoC in which multiple computing elements co-exist. Some of these computing elements are general purpose, programmable devices (e.g., GPU cores) while others are highly-specialized, power-efficient accelerators designed for specific tasks. These may include, among others, accelerators for tensor algebra computations, AI accelerators (such as NVIDIA Deep-Learning Accelerator (NVDLA)), and/or graph analytics. Figure 2 shows our reference SoC design,

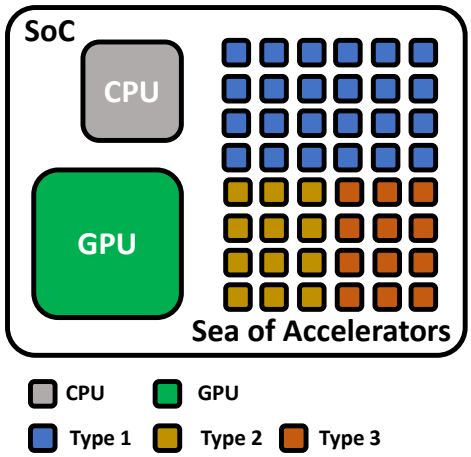

Figure 2: Our reference SoC design. An SoC is envisioned in which general purpose and programmable devices, such as CPU and GPU cores are placed next to highly-specialized computing elements designed, for example, to accelerate tensor algebra computation, AI, and/or graph analytics.

where the "sea of accelerators" represents the highly-specialized computing elements that could execute different functions.

We also assume that some of the computing elements are known or understood, while other represent new hardware designs that need to be integrated with the rest of the system. The objective of our methodology is to provide an agile and flexible simulation environment in which the part of the application that runs on the known computing elements is executed at native speed (with little insights on the execution unless specific instrumentation is employed) while the part of the application that executes on the novel hardware concepts is executed on hardware simulators that provide detailed information about the execution timing and power consumption. We refer to the two phases of this hybrid simulation environment as "Fast Path" and "Detailed Path."

Our methodology is based on three main components: (1) a modern task-based runtime based on MCL programming model and runtime (Section 3.1), (2) an interface to seamlessly plug in and out new hardware designs based on the Portable Computing Language (POCL) library (Section 4), and (3) a set of hardware development tools that provide detailed timing and power information, including Aladdin [18], and Accelergy [22] (Section 3.3). We start by factoring the application in computational tasks, which could be sequential or data parallel. Tasks that execute on known devices (i.e., on the fast path) are performed at native speed on the actual device. Tasks that execute on the novel accelerators (i.e., on the detailed path), instead, are modeled or simulated. Thus, for example, GEMM and spMV operations could be scheduled on novel accelerators while all other tasks could be scheduled on GPU devices. Users can easily adapt their algorithms to take advantage of the new hardware designs, e.g., by tiling large GEMMs or tensor contractions in a way that fits the new GEMM accelerator.

Our approach has several advantages: (1) The overall simulation is much faster than using cycle-accurate or cycle-approximate simulators. This, in turn, makes it possible to execute full applications or application workflows with realistic input sets, which is generally prohibitive on cycle-accurate simulators, during the DSE phase. Moreover, the increased speed enables an agile development cycle and potentially allows to explore a larger co-design space. (2) The proposed methodology is flexible and enables users to quickly plug devices in and out without modifying the application. For example, the system can be easily configured with varying number of GEMM accelerators to support convolution operation in AI workloads or with larger CONV accelerators. (3) Our methodology is composable. Different development tools can be integrated and used concurrently, including hardware simulators, performance models, or field-programmable gate array (FPGA) emulation. This allows hardware developers to continue using their trusted tools and still interact with the rest of the system when executing a full application. Overall, our novel methodology enables *true* hardware/software co-design and provides a valuable tool to design next-generation computing systems through the possibility of exploring a larger space in which both hardware concepts and algorithms can change simultaneously, along with the ability to execute large input sets.

3.1 Minos Computing Library

The Minos Computing Library (MCL) [6] is an asynchronous task-based programming model and runtime for extremely heterogeneous systems. MCL enables true portability across a wide range of heterogeneous devices, including CPU and GPU devices, FPGA and AI accelerators. The current MCL release supports all major GPU vendors, Xilinx and Altera FPGAs, and NVDLA. MCL consists of two main components: a system-level scheduler and one or more applications. MCL applications are factored into computational tasks, units of work that are generally executed on data-parallel devices. The tasks submitted by applications are executed asynchronously, which allows overlapping data transfer and task execution. Each application submits its request to a system-level scheduler, which selects the best available device to execute the task based on a variety of scheduling algorithms. Several schedulers are made available in MCL, including throughput-oriented, power-aware, locality-aware, and hybrid (locality-aware combined with throughput-oriented).

MCL has the ability to manage multiple classes of heterogeneous devices at the same time. Users can specify the device class on which they wish to execute a task but not the specific device. This design feature allows MCL to execute programs on systems that are very different from the one on which the application was originally developed. For example, applications developed on a laptop (e.g., single Intel GPU) can be executed on a NVIDIA DGX system (e.g., eight NVIDIA GPUs) without any modification. MCL will automatically manage the multiple GPUs, exploit data locality, perform load balancing, and manage asynchronous task execution.

In this work, we extend MCL to support a"simulated device." This include a new device class (`MCL_TASK_PROTEUS`) and a new kernel format. The former is a natural extension of the standard flags (`MCL_TASK_CPU`, `MCL_TASK_GPU`, `MCL_TASK_NVDLA`, `MCL_TASK_FPGA`) and allows users to indicate that certain tasks should be executed on a simulated device. Note that these flags can be combined, hence users can indicate more than one device class or all of them

(`MCL_TASK_ALL`), giving the MCL scheduler the flexibility to choose the best device based on contingent situation. The latter is required to support fixed-function accelerators. Since fixed-function accelerators are not programmable, we introduced the new kernel format `MCL_KERNEL_VOID` to indicate that the kernel is implemented in the accelerator and hence does not require the user to provide any kernel code. When MCL encounters one of such kernels, it will not attempt to compile it for a programmable device, as it is the case for OpenCL kernel code or SPIR-V. Instead, as for binary formats, MCL associates the kernel with only a specific device. Note that it is still necessary to specify which kernel a task executes even for fixed-function accelerators. In fact, similar to other formats, an accelerator may implement different kernels (e.g., single-precision GEMM, double-precision GEMM, SpMV, etc.), thus users need to indicate which functional unit they intend to use for a task.

3.2 Portable Computing Language

The Portable Computing Language (POCL) [8] is a multi-architecture, open-source OpenCL implementation based on the Low Level Virtual Machine (LLVM) framework and the clang C interface. POCL supports OpenCL 1.2 standard and many features from the 2.0 standard, including support for SPIR-V and shared virtual buffers. Currently, POCL supports various architectures, including various CPU models (with a "basic" and "pthread" OpenCL implementation), NVIDIA and AMD GPUs, through CUDA and HSA, respectively. Additionally, POCL implements "fixed-function" accelerators through `CL_DEVICE_TYPE_CUSTOM`, which provides an interface for developers to attach new hardware designs to POCL. The fixed-function accelerators appear on the system as any other OpenCL device, thus tools that generally work with OpenCL libraries (such as `clinfo`) can also see these accelerators. We leverage this interface to seamlessly include simulated devices among the list of OpenCL devices so that MCL can automatically detect the simulated devices. This goes beyond MCL and one can also write an OpenCL program to access the Proteus devices.

3.3 Simulation Frameworks

Aladdin. Aladdin [18], is a pre-register-transfer level (RTL) simulation framework that gives the ability to quickly explore the design of data parallel accelerators. This design philosophy aligns strongly with our work. Aladdin takes a high-level program as input in place of the kernel that needs to be accelerated and extracts a dynamic data dependence graph (DDDG) describing the accelerator. With this initial unconstrained graph, optimizations and hardware-related constraints are applied to create a model of accelerators' activity, essentially mapping an application dataflow graph onto preset set of hardware resources. Some architectural optimizations that can be applied during the mapping phase are loop unrolling (parallelization factor), loop pipelining and the number of memory ports available for the datapath to connect to the nearest memory hierarchy. Power and performance is estimated using dynamic traces obtained from a driver program. Cycle-level resource activity is tracked and is provided as input to the power model. Area estimation is done using pre-populated lookup tables corresponding to each resource in the DDDG. Validation of the Aladdin framework has been done against RTL simulators. Use of Aladdin eases exploration of various flavors of hardware accelerators beyond what is possible with other simulators used in this work. This makes Aladdin a valuable tool in our work.

Timeloop/Sparseloop. Timeloop [13] is a framework to enable design space exploration of arbitrary deep neural network accelerator designs. The workload is represented via nested loops with fixed bounds and linear indexing, and loop bodies that can be reordered. The architecture must be specified including number and type of processing elements, number and type of memory levels and the data types and widths, whereas the interconnection topology is automatically inferred by the tool. Mapspace exploration is the main facet of the Timeloop framework and involves coming up with a configuration of the accelerator that provides the best energy and performance efficiency. An efficient mapping is one that schedules operations and gets the best utilization of the available resources and moves data to/from the accelerator while achieving data re-use for a given input. Since there are many different possibilities, a heuristic-based search of possible mappings is done to meet user-defined criteria. To assist with this search, a model is used to quantify the quality of each mapping in terms of performance and energy utilization. The model uses Accelergy [22] for energy estimations.

Sparseloop [23] is an extension of Timeloop framework that models design space exploration of sparse tensor accelerators. To support sparse computations, the Timeloop framework is extended with the ability to specify the overheads related to metadata storage (associated with compressed storage formats), the probability distribution of the sparse inputs, and optimizations used in the hardware to support sparse computation (such as skipping and gating). Given this information, Sparseloop starts with the assumption that computation is dense, subsequently it starts to filter out ineffectual computations (zero results) and other metrics such as reduced number of memory accesses due to sparsity, etc. In a manner similar to Timeloop, Sparseloop can automatically search the potential mapping space given a workload and an architecture. The performance, area and energy numbers are reported after the analysis.

MAESTRO. Modeling Accelerator Efficiency via Spatio-Temporal Resource Occupancy (MAESTRO) is another tool to explore the design space of deep neural network accelerators [9]. It is an analytical modeling tool to find efficient loop tilings and orderings, and spatio-temporal mapping of data onto compute units. Only built-in kernels or operations are supported by MAESTRO and there is no generic way to specify the problem as in Aladdin or to some extent in Timeloop (albeit, only loops). Also, the architecture is fixed. Different parameters of the architecture can be varied, e.g., number of processing elements, size of L1 and L2 caches, network-on-chip bandwidth, and off-chip bandwidth. MAESTRO operates in a similar manner to Timeloop and does different analyses including re-use of data elements spatially and temporally. A cost model is used to quantify the quality of each mapping. Activity counts are obtained during the analysis and used for energy estimations using look-up tables (based off of 28nm technology). The performance, area and power numbers are reported after the analysis.

Scale-Sim. Scale-Sim [15] is a tool for exploring the design space of convolutional neural network accelerator that has a systolic

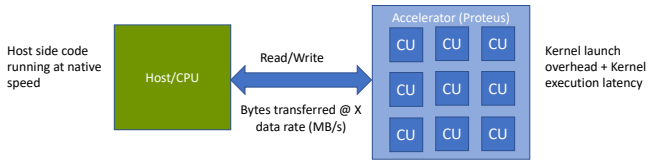

Figure 3: Proteus offload compute model: Proteus device models a data-parallel array of compute units with each being able to execute one work-group (according to OpenCL specification). Each compute unit is a collection of PEs as obtained through hardware design exploration tools. For example, each compute unit can perform 64x64 GEMM.

```
Platform #0: Portable Computing Language
 +-- Device #0: basic-Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz
 +-- Device #1: PROTEUS Dense Matrix Multiplication Kernel
 `-- Device #2: PROTEUS Dense Matrix Multiplication Kernel
Platform #1: NVIDIA CUDA
 +-- Device #0: NVIDIA Tesla V100-SXM2-16GB
 +-- Device #1: NVIDIA Tesla V100-SXM2-16GB
 +-- Device #2: NVIDIA Tesla V100-SXM2-16GB
 `-- Device #3: NVIDIA Tesla V100-SXM2-16GB
```

Figure 4: System level view with Proteus devices. POCL's OpenCL implementation lists CPU and 2 Proteus devices supporting GEMM kernels on a single platform. NVIDIA's OpenCL implementation lists the available NVIDIA GPU's on a separate platform. OCL-ICD is utilized to list devices from both the implementations.

architecture. It provides cycle accurate estimates of number of compute and stall cycles, resource utilization, on-chip SRAM cache and off-chip DRAM memory bandwidth requirements. Memory traces are generated at each cycle of the simulation. One can vary different parameters of the systolic architecture such as the height and width of the systolic array, the size of the on-chip caches and the dataflow (e.g., weight or output stationary). The number of cycles at the end of the simulation are used to estimate the performance of executing a kernel. In our work, we utilized the Scale-Sim simulator to do a GEMM operation on the systolic array. The tool does not report area and power information unlike other tools utilized in this work. Therefore, we used Aladdin to obtain area information for accelerators designed using Scale-Sim for some of our experiments.

Most tools are feasible for designing accelerators for dense computations, whereas only a few design tools provide the opportunity to explore sparse computations. In this work, for sparse computations, we used Aladdin and Sparseloop; whereas, Aladdin, Timeloop, MAESTRO, and Scale-Sim were used for dense compute kernels. A summary of all simulators used in this work is provided in Table 1.

4 PROTEUS EMULATOR

Proteus device models a data-parallel device with multiple compute units as shown in Figure 3. Each compute unit contains multiple processing elements (PEs) as obtained through the pre-hardware simulators. The Proteus device is emulated on a multi-core CPU and implemented using the POCL framework. The POCL framework provides all the basic functionality to realize a new OpenCL device. It handles all the boiler-plate code to realize the OpenCL standard. For example, the ability to check the legitimacy of arguments provided to an OpenCL function call and return appropriate code to the caller. Using this framework, a new device is implemented to realize the Proteus device. The Proteus device does not utilize the LLVM-based compilation framework inside POCL since it uses builtin kernels (i.e., using OpenCL's `clCreateProgramWithBuiltInKernels`).

At initialization phase of the Proteus device, each device is individually configurable to have different number of compute units, amount of device memory, the different kernels it supports and latency of various operations. This is done through the use of environment variables and configuration files. They are also used to instantiate multiple devices. For example, `POCL_DEVICES` can be used to instantiate multiple Proteus devices. Each device can implement different functions and is parameterizable through the use of environment variables, e.g., `POCL_PROTEUS0_PARAMETERS` corresponds to the configuration file that is used to configure the first Proteus device. Changing the number of devices does not require re-compilation of application code or the POCL framework and can occur seamlessly. The Proteus device can co-exist with other backends supported by POCL. Also, one does not have to restrict to POCL's OpenCL implementation for using other devices in the system and vendor libraries can be used, e.g., by using the installable client driver. In our experiments, we use POCL's OpenCL implementation alongside NVIDIA's OpenCL implementation. A system-level view of this configuration is presented in Figure 4. It includes two platforms with POCL's OpenCL implementation listing a CPU and two Proteus devices, and NVIDIA's OpenCL implementation listing the V100 GPUs.

The Proteus device uses off-line simulations to populate the various parameters of the device through configuration files. Off-line simulations were performed instead of dynamically loading hardware simulators to avoid the problem of the simulation taking longer than the time to execute a kernel. A long-running simulation can result in in-accurate performance estimates as the scheduler may not select the Proteus device for future invocation of kernels. A scheduler keeps track of available resources on a device and, if assigned tasks are still pending, then future work is not scheduled resulting in deteriorating performance. Therefore, this design choice of performing off-line simulations is adopted in this work.

Implementation. The data parallelism in the Proteus device is modeled through parallel threads. Based on the user configurable parameter of number of compute units, a pool of threads is formed at initialization time. These threads operating in parallel are used later on to perform work assigned to the device. Having multiple threads gives the ability to execute a kernel with multiple work-groups in a data-parallel manner. The work is pushed to the device when a kernel is enqueued for invocation from the application side. Queues manage the work sent to the device. In particular, two queues are used. One to enqueue all pending kernels that are ready to be executed on the compute units and the other is used to manage the decomposition of each kernel invocation into individual work-groups. A work-group is the minimum unit of work that will execute on a compute unit. The available compute units on the device get chunks of work-groups from the queue to perform computation on. Getting chunks of work-groups is equivalent to getting portions of a data array. The compute unit then simulates the occurrence of this

computation, i.e., no actual work is done. Essentially, the compute unit delays the execution of each work-group as specified during initialization of the device. Once the compute unit has finished the assigned work, it gets the next set of available work. If there is no work in the work-group or kernel queues, then the worker threads go into a waiting state. Upon completion of a kernel invocation, an OpenCL compatible signal is sent to mark the completion of a task.

The notion of simulating the work performed by a device may not be feasible for applications that require decisions to be made based on computed values. For example, deciding to take a certain execution path versus another path. There are alternative methods in which one may tackle this, e.g., by using pre-computed execution paths. But, it is to be expected that this may not work for every application, e.g., due to size of workload making pre-computation in-tractable. As with any other simulation-based framework, this is a limitation of proposed work.

Similar to kernel invocations, data transfer to/from the device is done through queues. Once the application initiates a read or write command, the Proteus device simulates the latency of these operations based on user-specified data-transfer rates during the initialization phase of the device. The data transfers can take place independently of the kernel invocations that take place asynchronously. The delays at the device side are based on the amount of data transferred and fixed overhead for each transfer.

The support for built-in kernels by a Proteus device requires one to define the inputs and outputs of the kernel. These are described using the configuration file. At runtime, when the application creates a kernel to run on a specific device, our implementation checks whether the programmer provided the correct kernel arguments. One can query the Proteus devices on an existing system to know which built-in kernels are supported. Additionally, it is possible to query the device to get information about the work-group size, i.e., the smallest unit of work performed by each compute unit. This provides a flexible and OpenCL compliant programming interface.

Design Flow. A typical flow to explore the design space of accelerators using the proposed setup is shown in Figure 1. The initial step is to identify the set of applications that need to be targeted for performance optimization. A common set of performance critical kernels is identified across all the applications. Then, hardware simulators are used to explore various architectural design options of the accelerators for desired kernels. For example, if one decides to design an accelerator with the GEMM accelerators; the next step is to devise the smallest amount of work to be performed by each compute unit on the accelerator. This determination is not rigid and can be refined iteratively in line with the goals of the proposed framework. Based on this design choice that may be dictated by some characteristic of the application, the hardware design space of accelerators is explored. The hardware design space itself has tradeoffs and one has to balance between power and performance efficiency goals. This can be dictated by end usage of the accelerators, e.g., HPC platforms require high performance, whereas, mobile computing platforms require high energy efficiency.

The available simulators are used to obtain key metrics such as performance and power numbers for kernels of interest and various configuration files are populated for Proteus devices based on the requirements of the experiments. At this step, one needs to decide the number of compute units to include on a single device among other parameters. This may again be dictated by application characteristics. For example, an application may realize tiled GEMMs with each tile being of dimensions 512x512. Then to perform these GEMMs efficiently on an accelerator where each compute unit is able to execute 64x64 GEMMs, a total of 64 compute units may be included on a single device. This gives the ability to perform a GEMM of dimensions 512x512 within similar time period as that required to perform one unit of work on a single compute unit.

Next, the configuration files are utilized to setup various systems with varying number of devices. The MCL's runtime environment is then setup and the task scheduler is initialized. Applications are now executed and metrics of interest are measured. The performance estimates are then used to refine the accelerator designs and system setups as desired. Here, one has multiple paths forward if performance and power goals have not been met. It may include trying varying number of devices, or devices with varying number of compute units. This can all be done at the MCL level. To further refine the design of individual compute units, one can go back and try different optimizations at the architectural exploration phase. This may include trying different unit of work performed by each compute unit and/or using more area (number of PEs) to decrease the latency. The next section highlights the design flow in more detail with some applications.

5 EXPERIMENTS AND RESULTS

We design our experiments to evaluate the performance benefit of having specialized accelerators for a set of kernels that have broad applicability across multiple domains. We demonstrate a typical design flow that would be followed to explore the use of novel accelerators as highlighted in Figure 1 and discussed in Section 4. The experiments start with design space exploration for these kernels and getting power/performance metrics by using off-the-shelf simulators (Section 3.3). After this phase, various configurations of Proteus devices are instantiated and the end-to-end execution runtime (performance) of various applications in hybrid setup with simulated Proteus devices and GPUs devices is quantified.

5.1 Experimental Setup

We use NVIDIA DGX-V system with 8 Tesla V100 GPUs and Intel Xeon E5-2698 CPUs as our baseline in the experiments. The CPUs are used for the Proteus devices. In all of our experiments, speedup as compared to a pure GPU-based system is presented. Moreover, to ensure a fair application-level comparison, both Proteus and GPU devices use the same application code (i.e., OpenCL). It is possible to obtain better performance from the GPUs using CUDA instead of OpenCL, however, at the expense of reduced portability. The Proteus device has been added to POCL version 1.6. MCL version 0.5 is used as a baseline to which minor changes are made to detect and schedule tasks onto Proteus devices. We use OpenCL Installable Client Driver (OCL-ICD) [20] in our experiments to list devices simultaneously from the POCL-based OpenCL and NVIDIA's OpenCL implementations. The OCL-ICD was slightly modified to list the Proteus devices first so that the MCL scheduler is able to utilize these devices first for tasks that can be run on both the Proteus devices and the GPU devices.

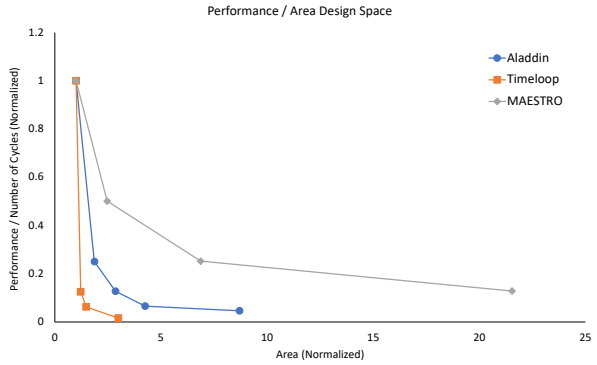

Figure 5: Performance/Area tradeoff for dGEMM kernel using multiple simulators. Each design point is obtained by searching wide range of mappings possible on a fixed architecture. Optimal design consumes least area and provides highest performance (y and x-axes: lower is better).

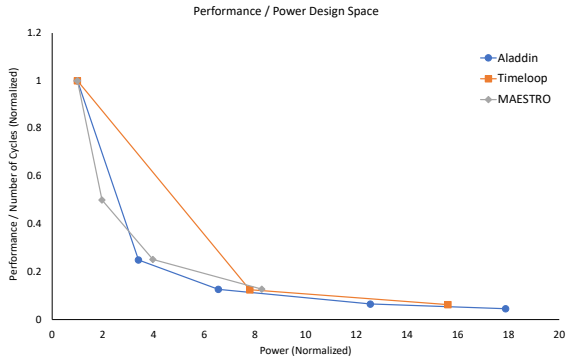

Figure 6: Performance/Power tradeoff for dGEMM kernel using multiple simulators. The design points have one-to-one correspondence with design points in Figure 5.

5.2 Accelerator Design Space Exploration

In our work, we target applications with both dense and sparse computation kernels to evaluate the efficacy of using specialized accelerators. The kernels as enumerated below are widely used in various application domains, including HPC, AI, and graph. We also highlight the simulators that were used to explore the accelerator design for each kernel below.

- **dGEMM**: dense general matrix-matrix multiplication, this involves the multiplication of two dense matrices. We use the single-precision floating point representation for data elements. dGEMM is a widely utilized kernel in HPC and AI applications. With particular interest due to the deep learning community for design of novel deep learning accelerators, there is a wide variety of tools that allow early design exploration. In this work, we use Timeloop [13], MAESTRO [9], Scale-Sim [15] and generic pre-RTL simulator Aladdin [18] for the design of dGEMM compute units.
- **spMV**: sparse matrix-vector multiplication, this involves the multiplication of a sparse matrix with a dense vector. We use compressed storage format to represent the sparse matrix elements. spMV is utilized in HPC (e.g., linear solvers) and graph applications. We explore the accelerator design for this kernel using Aladdin and Sparseloop [23].
- **spMspM**: sparse matrix-sparse matrix multiplication, this involves the multiplication of two sparse matrices. Again, compressed storage format is used to represent the sparse matrix elements. The accelerator for this kernel is also designed using Aladdin and Sparseloop, both of which utilize the Gustavson's Algorithm for implementation [25].

Figures 5 and 6 show design space exploration of dGEMM using simulators Aladdin, Timeloop and MAESTRO. In the figures, different data points represent different design points that were explored. The designs obtained through each simulator are normalized by the design with the slowest execution time. Each design point is an optimal design obtained through the simulator. For example, Timeloop uses a heuristic-based search to come up with an optimal design out of all possible mappings. As parallelism is increased through the addition of more PEs and/or more memory capacity at various levels, the designs consume more power/area and becomes more performant. At some point, the designs start to saturate and there is no noticeable performance gain obtained by increasing parallelization factors available in the simulators. Also, the power and area consumption may become prohibitive for some application domains, especially, as one looks to include more or more compute units on a single accelerator.

Table 2: Performance and area estimates for dGEMM accelerators obtained through simulators. Number of Proteus devs represents the iso-area equivalent of NVIDIA V100 GPU.

Simulator	Unit Latency (Cycles)	Number of Proteus devs	Total Compute Units
Aladdin	34274	17	1062
Timeloop	256	1	53
MAESTRO	4242	1	39
Timeloop	640	1	59

One can appropriately choose a design for their application by the work proposed in this paper. This may involve a back and forth process to come up with an optimal design as discussed earlier. For our work, we choose an arbitrary design point on pareto front of the performance/area curve in Figure 5, i.e., a design point with minimal possible area and highest performance.

To assess the benefit of using specialized dGEMM accelerators in an application, we setup a system with as many accelerators that can replace a single NVIDIA V100 GPU with same area. This is representative of a strategy that may be employed to design a computing system. Since designs produced by different simulators have varying degrees of area estimates, one tends to get a wide variation in number of compute units available. In our experiments, the lowest area estimates are produced by Aladdin, whereas the highest area estimates are produced by MAESTRO. The design points used for each simulator are shown in Table 2 and corresponds to the capability to perform a dGEMM of dimensions 64x64 on a single compute unit. We have bounded the number of compute units

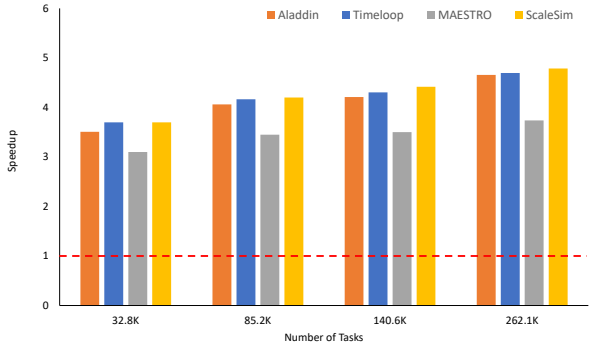

Figure 7: Iso-area end-to-end execution time comparison for a dGEMM kernel with different sized-inputs. The reported speedup is using specialized accelerators designed using Aladdin, Timeloop, MAESTRO and Scale-Sim as compared to a single NVIDIA V100 GPU. Higher speedup is better.

on a single Proteus device to 64. Results, later on, show diminishing returns by adding more and more compute units on a single device.

With the above setup, we quantify the performance benefit of invoking large-sized dGEMM on the accelerators. The input dimensions used are: 32768x32768, 45056x45056, 53248x53248, and 65536x65536. The computations are done using single-precision floating point values. On the application side, tiles of dimensions 1024x1024 are created and sent as tasks to be executed on the devices (discussed later). The MCL scheduler was set up to use the least number of devices possible to complete the computation, i.e., only handful of Aladdin Proteus devices are used. Figure 7 shows the speedup achieved by using the specialized accelerators as compared to executing dGEMM on a single GPU device. These runtime measurements were done at the host side and corresponds to running a complete application from start to finish, including the serial portion. Therefore, a performance improvement bound is expected based on Amdahl's Law. That said, Figure 7 shows comparable speedups obtained by using accelerators designed using various accelerators. One anomaly is the accelerators based on MAESTRO, which indicates that there is some optimal number of compute units that should be available on a device to get best performance. In next subsection, we assess the performance benefits of using the accelerators in different applications and highlight various tradeoffs that can be investigated while designing an accelerated system.

5.3 Application/Hardware Co-Design

We target four different applications from HPC and graph analytics domains for our co-design study. One application has a dense computation kernel, whereas the remaining have sparse computation kernels. A synopsis of each application is included below:

- **CCSD(T)-proxy**: NWChem [1] is a widely used computational chemistry HPC application that uses the couple cluster CCSD(T) method [14]. CCSD(T) is a tensor contraction operation. Since tensor contractions are computationally expensive, their acceleration is beneficial to many HPC applications. A common way that is also adopted in this work is to decompose the tensor contraction into transpose-transpose-GEMM-transpose (TTGT). TTGT performs transposes of the input tensors followed by a GEMM and a final transpose of the output. We mimic this computation. In our implementation, the transposes are launched on the GPUs, and once the transposes are complete. The GEMMs are launched onto the specialized accelerators. Once the GEMMs are complete, the final transposes are launched on the GPUs. The main time consuming kernel is the dGEMM in this application.
- **Triangle Counting**: This is a popular graph algorithm used for community detection, among other applications [4]. It involves counting the number of triangles in a graph. A triangle is a sub-graph that is composed of three mutually adjacent nodes in a graph. There are multiple algorithms to count the number of triangles in a graph. In this work, we use the approach proposed in [4] and highlighted as Algorithm 2 in [16]. This algorithm only uses the adjacency matrix of a graph. In our implementation, the adjacency matrix is stored in compressed format, i.e., compressed sparse row representation. The main time consuming computation in this algorithm is a spMspM, whereby the adjacency matrix is multiplied with itself. The acceleration of spMspM kernel is considered in our work. After the matrix multiplication, an element-wise multiplication is performed with the adjacency matrix. Upon completion of the element-wise multiplication, a reduction operation is performed to obtain the final result.
- **Breadth-First Search (BFS)**: This is another popular graph algorithm with wide variety of uses. In this work, we realize linear algebra based implementation of BFS [5]. To perform search on a graph, a vector with a unit value in the index from where search is desired can be multiplied with the sparse graph matrix. One such iteration performs one step of the search. To perform multiple searches from multiple vertices in parallel and to increase data re-use, this problem can be formulated as a spMspM by having the second matrix as a concatenation of multiple vectors.
- **GraphSAGE-proxy**: GraphSAGE algorithm [7] produces low-dimensional embeddings of nodes in a large graph. These embeddings can be used for machine learning tasks such as classification and clustering. The GraphSAGE algorithm is designed to work well with evolving Graphs (with unseen nodes) as compared to other algorithms. Given rich feature information at each node of the graph, embeddings at each node are generated by aggregating feature vectors of adjacent nodes and scaling using weight matrices. In our implementation, the graph is represented as an adjacency list to ease with the search of neighboring nodes. Feature vectors of neighboring nodes are aggregated on each node with a cut-off factor as defined in the algorithm. The weight matrices are hyper-parameters of the algorithm. The main computational kernel in this algorithm is a spMV operation for multiplication of the sparse weight matrix with a dense aggregated feature vector.

Given these applications, we identified the common set of kernels as part of our design flow. Once the metrics of interest for accelerator designs are measured using the simulators and final

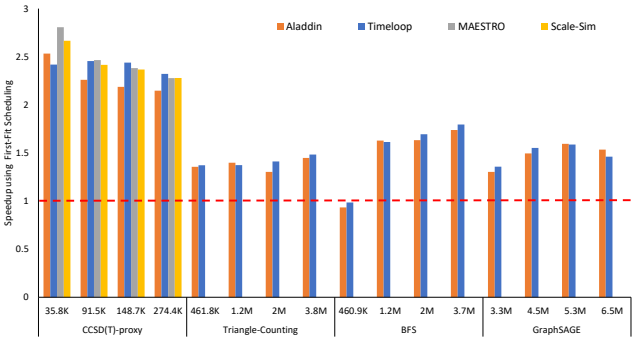

Figure 8: Speedups obtained for applications with varying input sizes as compared to a system with 8 NVIDIA V100 GPUs. The designed system is setup to have 50%-50% breakdown among the specialized accelerators and GPUs (i.e., 4 Proteus devices and 4 GPUs). The scheduler uses the Proteus devices first when scheduling tasks that can execute on both the Proteus and GPU devices.

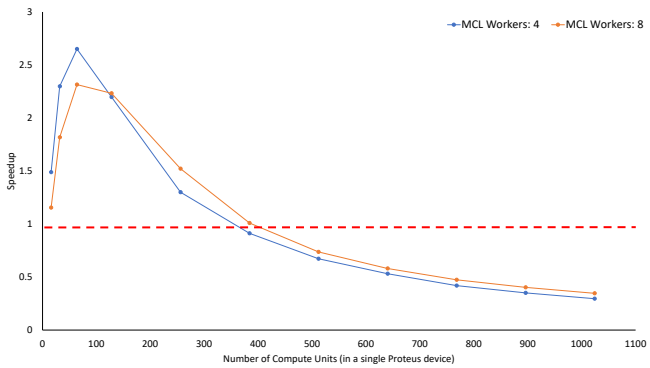

Figure 9: Impact of varying number of compute units on a Proteus device. The speedup plot is for the CCSD(T)-proxy application as compared to a GPU-based system. The plot shows that there is an optimal number of compute units that can be included on a Proteus device.

selections are made, we continue to set up the system. We use three different setups for our applications corresponding to the three computational kernels. A common setup is used for Triangle Counting and BFS applications since both need acceleration of the spMspM kernel. In each setup, 50%-50% breakdown of specialized accelerators and general purpose GPUs is done. Therefore, we utilize 4 Proteus devices and 4 GPU devices in each setup. Our baseline is a 8-GPU NVIDIA DGX system. In our experiments, the MCL scheduler is setup to utilize the Proteus devices first and the GPUs only if resources on the Proteus devices have been exhausted. This is a power-efficient setup as opposed to using all the available resources in an equitable manner. At the same time, some tasks can only be run on the GPUs, e.g., transposes in CCSD(T) proxy and element-wise multiplication in Triangle Counting. This system setup represents a typical usage scenario whereby specialized accelerators work in conjunction with general-purpose accelerators.

Figure 8 shows the speedup achieved for applications as compared to the system with 8 GPU devices. The CCSD(T)-proxy application uses dGEMM accelerators designed using Aladdin, Timeloop, MAESTRO and Scale-Sim. Triangle counting and BFS use spMspM accelerators designed using Aladdin and Sparseloop. Lastly, GraphSAGE uses a spMV accelerator designed using Aladdin and Sparseloop. To re-iterate, an accelerator designed using a specific simulator means that the latency of executing one work-group in a kernel is assessed through simulations. The total number of compute units to include on a single device is another design parameter as discussed later. Four different input sizes are used in each case. In case of CCSD(T)-proxy application, dense input of up to dimensions 65536x65536 is evaluated; in case of graph applications, graphs of sizes up to 262K are evaluated (with sparsity level of about 95%). The total number of tasks sent to the devices is shown on the x-axis corresponding to each application. The total number of computational tasks generated by the applications range between 23.7K and 6.5M. This is dependent on the tiling factor used at the application side. For example, with input dimensions of 32768x32768 and a tiling factor of 1024x1024 in CCSD(T) proxy application, the total number of transpose tasks generated by the application are: 32768/1024 * 32768/1024 * 3; whereas, the total number of GEMM tasks are: 32768/1024 * 32768/1024 * 32768/1024. Some important factors to determine the tiling factor at the application side are: the number of available resources on a single device (to avoid over-subscription), finding the right balance between data transfer overheads and computation time, and the number of tasks to be managed by the MCL scheduler based on available resources on the host side (more tasks is equal to more work for MCL runtime).

In our experiments, we find that the number of compute units available on a single Proteus device can impact overall application performance. This is one of the highlights of our work, since early rapid exploration is made possible. Figure 9 shows that there is an optimal number for the CCSD(T)-proxy application. Too few compute units can hurt performance as well as having a large number of compute units. Achievable performance starts to saturate after too many compute units are included, something that may also not be possible due to the thermal design limit. Part of this is due to emulation of the Proteus device taking place on a CPU with limited capability to scale the number of parallel threads. The work queues on the device side also play a role in the limited scalability. In a real device, these queues are implemented in hardware but are size limited. Based on this experiment, we choose to limit the number of compute units on each Proteus device to 64 for dGEMM kernel. Similarly, the number of compute units for each kernel are tweaked to get optimal performance from the Proteus device for the other applications/kernels.

Figure 9 also shows the effect of varying the number of MCL workers that are parallel threads created by the runtime to manage execution of tasks for things such as task offloading to assigned resource, status checking and returning error code to the scheduler upon task completion [6]. The number of workers to use is a design-time parameter and can affect performance since too many worker threads can have high overhead but not have enough work

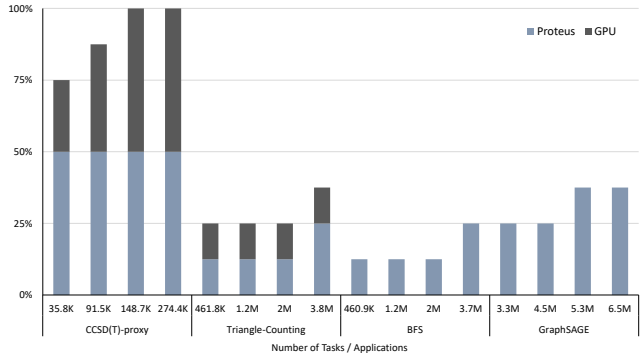

Figure 10: System Utilization broken down across Timeloop-based Proteus devices and GPU devices. The highest utilization for Proteus devices is 50% across the system, meaning all four Proteus devices are in use. Similarly, a 100% utilization means that all eight devices including the four GPUs in the system are being utilized.

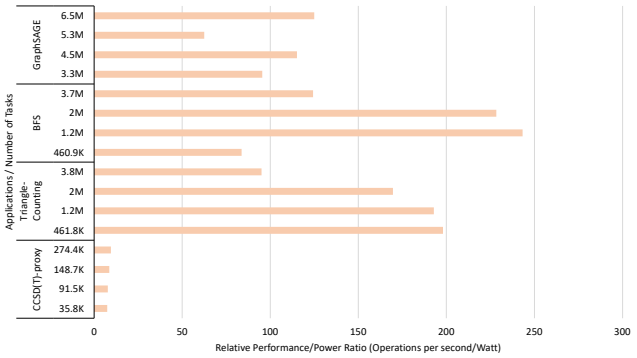

Figure 11: Performance per Watt (OPS/W) due to Proteus devices (Aladdin) as compared to the GPU-based system.

Table 3: Average data transfer amount between host and devices for investigated applications.

Application	Bytes/Transfer
CCSD(T)-proxy	4 MB
Triangle Counting	19.36 KB
BFS	19.2 KB
GraphSAGE	32.4 KB

to do and vice versa. We choose to use 4 MCL workers in our experiments based on the average number of used resources across all applications and the number of available tasks to manage.

The utilization of devices in the system for each application is given in Figure 10. These results only include the system based on Timeloop-based Proteus devices. The utilization profile across systems based on other Proteus devices is similar. Only up to half of available Proteus devices are utilized in Triangle Counting and BFS applications, whereas only a single GPU device is utilized to do work other than spMspM in Triangle Counting application. Similarly, GraphSAGE utilizes between 2 to 3 Proteus devices based on the input size. Looking at other results in Figure 10, one can observe that the system is fully utilized only in the case of CCSD(T)-proxy application, whereas the system is partially utilized in all other cases. In majority of the cases, all of the available Proteus devices are not utilized, meaning system is over-provisioned and there is room for improvement. Once again, this points to the importance of quickly trying different system configurations before pursuing the path of a fixed architecture.

Discussion. The performance benefit obtained through Proteus devices in case of CCSD(T)-proxy application with dense compute kernel is significantly higher than the rest of the applications using sparse kernels (CCSD(T)-proxy highest speedup is 2.8 compared to about 1.8 obtained for BFS). We attribute this to the overhead of doing the data transfers over-weighing the benefit obtained due to kernel acceleration as assessed through our profiling experiments. Table 3 shows the average data transfer in each application. It can be noticed that the amount of data transfer is significantly higher in each data transfer request for the CCSD(T)-proxy application as compared to the remaining applications using sparse kernels. The number of data transfer requests also tends to be higher for applications using sparse kernels. This is amplified due to the high overhead to do the data transfers in case of Proteus devices modeled off the PCIe 3.0 bus as compared to the GPU devices with proprietary high-speed data bus (i.e., NVLink). Our work thus shows the importance of overall system design, not just that of a stand-alone accelerator.

We also compared the Performance/Watt benefit of using the accelerators with results shown in Figure 11. These results are obtained by calculating the number of tasks submitted at the application side and the time to perform the kernel computation. It is possible to do these measurements at the device side, e.g., by using hardware performance counters on the GPU. A similar mechanism can easily be developed for the Proteus device, but the limitation is the capability of OpenCL programming model to report these events. Currently, there is no such support in the OpenCL standard. This is part of our future work. Moreover, the power estimates obtained by the simulators are under-estimates to say the least. We have used the *nvidia-smi* utility to report GPU power that includes the power consumed by all the electronics on the GPU board. That said, the benefit of using the Proteus devices especially for applications with sparse kernels is evident (e.g., in majority of cases, more than or close to 100X performance/watt advantage is obtained as compared to the GPU devices). The accelerators designed for sparse kernels tend to consume less area and hence are less power-hungry when compared to their dense kernel counterparts.

Future Work. We have already discussed some of the directions this work may take in the future. We highlight these points below:

- *Enabling low-level design-space exploration using domain-specific languages*: there is a need to include Proteus device as a backend in one of the popular compiler frameworks to enable automated utilization of novel accelerators. The compiler can assist with tiling of code to most efficiently utilize underlying devices, among other things. This is part

of a broader effort to ease the adoption of accelerators for high-level programmers.

- *GPU as an emulation device*: we noticed throughput issues with CPU-based Proteus emulation framework. Although, there is a limit to the number of accelerators that may be included in a single chip, but we feel some of the throughput related issues can be resolved if the GPU is used as an emulation platform.
- *Inclusion of Network-on-Chip (NoC) effects*: a limitation of our work is that we do not include the impact of the interconnect needed for multiple compute units and other components on a single Proteus device. The NoC may lead to contention issues when too much traffic is generated in the system leading to degraded performance. We consider an ideal NoC in our work, i.e., no congestion or zero-latency to do the data transfers to the individual compute units once the data is on the device. Work is in progress to consider such delays in our device by using the SST architectural simulator [11].
- *Performance counters for profiling*: measuring various metrics of interest on the Proteus device is a relatively easy and low overhead task to gain an understanding of the various bottlenecks in the overall system. The main hindrance to realizing this is due to the communication issue with the host side using the OpenCL programming model. We plan to tackle this in future work through either a stand-alone interface or proposing extensions to the OpenCL API.

6 CONCLUSIONS

We proposed a coupled simulation methodology that gives the ability to explore the architectural design space of domain-specific hardware accelerators driven through the needs of applications in an agile and composable manner. We demonstrated the design and use of the Proteus device model as an emulation platform for assessing the performance advantage gained for multiple applications running realistic workloads. The utilization of a portable programming model enables hardware/software co-design for domains ranging from embedded to high-performance computing. In this work, our experiments demonstrated the successful integration of Proteus devices with GPU devices for kernel acceleration in HPC and graph applications along with the complete system software stack. Our results highlighted the importance of considering system-level effects such as overheads of data transfers and not just the stand-alone design of an individual accelerator. Overall, we demonstrated the capability of our methodology to do agile co-design of applications along with multiple accelerator designs.

ACKNOWLEDGMENTS

Support for this work was provided through U.S. Department of Energy's (DOE) Office of Advanced Scientific Computing Research as part of the Center for Artificial Intelligence-focused Architectures and Algorithms. This work is also supported by Pacific Northwest National Laboratory's (PNNL) Lab-directed Research and Development initiative Data Model Convergence. PNNL is operated by Battelle for the DOE under Contract DE-AC05-76RL01830.

REFERENCES

[1] E. Aprà and et al. 2020. NWChem: Past, present, and future. *The Journal of Chemical Physics* 152, 18 (2020), 184102. https://doi.org/10.1063/5.0004997
[2] B Barnes and J Slice. 2005. SimNow: A fast and functionally accurate AMD X86-64 system simulator. In *IEEE International Workload Characterization Symposium.*
[3] Nathan Binkert and et al. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7.
[4] Paul Burkhardt. 2017. Graphing trillions of triangles. *Information Visualization* 16, 3 (2017), 157–166.
[5] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. 2007. High-Performance Graph Algorithms from Parallel Sparse Matrices. In *Applied Parallel Computing. State of the Art in Scientific Computing*. Springer Berlin Heidelberg, 260–269.
[6] Roberto Gioiosa, Burcu O. Mutlu, Seyong Lee, Jeffrey S. Vetter, Giulio Picierro, and Marco Cesati. 2020. The Minos Computing Library: Efficient Parallel Programming for Extremely Heterogeneous Systems. In *Proceedings of the 13th Annual Workshop on General Purpose Processing Using Graphics Processing Unit* (San Diego, California) *(GPGPU '20)*. 1–10.
[7] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA). 1025–1035.
[8] Pekka Jääskeläinen, Carlos S. de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo H. Takala, and Heikki Berg. 2014. pocl: A Performance-Portable OpenCL Implementation. *International Journal of Parallel Programming* 43 (2014), 752–785.
[9] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. *IEEE Micro* 40, 3 (2020), 20–29. https://doi.org/10.1109/MM.2020.2985963
[10] Paolo Mantovani and et al. 2020. Agile SoC Development with Open ESP. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.
[11] Richard. Murphy, Arun F. Rodrigues, Peter Kogge, and Keith Douglas Underwood. 2004. The structural simulation toolkit : a tool for bridging the architectural/microarchitectural evaluation gap. (12 2004).
[12] none. 2018. Basic Research Needs for Microelectronics. (10 2018). https://doi.org/10.2172/1545772
[13] Angshuman Parashar and et al. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 304–315.
[14] Krishnan Raghavachari, Gary W. Trucks, John A. Pople, and Martin Head-Gordon. 1989. A fifth-order perturbation comparison of electron correlation theories. *Chemical Physics Letters* 157, 6 (1989), 479–483.
[15] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2018. SCALE-Sim: Systolic CNN Accelerator Simulator. *arXiv preprint arXiv:1811.02883* (2018).
[16] Siddharth Samsi and et al. 2020. GraphChallenge.org Triangle Counting Performance. *IEEE High Performance Extreme Computing Conference* (Sep 2020).
[17] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel). 475–486.
[18] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 97–108.
[19] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) *(ASPLOS X)*. 45–57.
[20] Brice Videau and Vincent Danjean. 2012–2021. ocl-icd. https://github.com/OCL-dev/ocl-icd.
[21] J. Wang, J. Beu, R. Bheda, T. Conte, Z. Dong, C. Kersey, M. Rasquinha, G. Riley, W. Song, H. Xiao, P. Xu, , and S. Yalamanchili. March 2014. Manifold: A Parallel Simulation Framework for Multicore Systems. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.
[22] Yannan N. Wu, Joel S. Emer, and Vivienne Sze. 2019. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*.
[23] Y. N. Wu, P.-A. Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. 2021. Sparseloop: An Analytical, Energy-Focused Design Space Exploration Methodology for Sparse Tensor Accelerators. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.
[24] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. 2003. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture* (San Diego, California) *(ISCA '03)*. 84–97.
[25] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. 687–701.